Leila Diana Demeter

Graphical User Interface development using Embedded Wizard and Yocto Project for an IVD device

Metropolia University of Applied Sciences

Bachelor of Engineering

Information Technology

Bachelor's Thesis

18 April 2023

Abstract

Authors:	Leila Demeter
Title:	Graphical User Interface (GUI) development using Embedded Wizard and Yocto Project for an IVD device
Number of Pages:	41 pages
Date:	18 April 2023
Degree:	Bachelor of Engineering
Degree Programme:	Degree Programme in Information Technology
Professional Major:	Professional Major Smart IoT Systems
Supervisors:	Keijo Länsikunnas, Suman Dahal

The research in this thesis aims to clarify how Graphical User Interface (GUI) development differs in the In Vitro Diagnostics (IVD) field compared to fields lacking regulations and standards such as the ones guiding the processes in the life-cycle of IVD devices.

The overall life cycle of an IVD device generally consists of the following steps: Proposal, Feasibility, Development, Validation activities and Post-release Processes. Each of these phases produces documentation for the purpose of traceability and consistency, and all of these steps serve to further perfect the device until the end of its life cycle.

The most important regulations affecting the IVD field are Regulation (EU) 2017/746 (EU IVDR), ISO 13485, ISO 14971 and the IEC 62304.

The development of the GUI in this thesis project used Embedded Wizard Studio (EW) as a development tool. EW is a relatively new technology that allows the developer to create reactive GUI modules and build the infrastructure of communication between the device and the user while using the Device Class and DeviceDriver.c.

Finally, the C code generated by EW was compiled using Yocto Project creating an image that can be flashed to the target device with the UUU tool, which proved that EW is a suitable tool for GUI development and integration of an IVD device.

Keywords: IVD, IVDR, MDR, Embedded development, Embedded Wizard Studio, Yocto Project, Product development process, Product life-cycle

Contents

List of Abbreviations

1 Introduction 1

2 Product development process of IVD devices 3

2.1 Standards and Regulations 3
2.1.1 Regulation (EU) 2017/746 (EU IVDR) 4
2.2 Preliminary Steps in the life cycle 6
2.2.1 Proposal 6
2.2.2 Defining Requirements 7
2.2.3 Feasibility 7
2.3 Development Phase 9
2.4 Release 10
2.4.1 Validation 10
2.4.2 Regulatory Approvals 10
2.4.3 First launch 11
2.5 Post-release Processes 11
2.6 Deliverables 12

3 Embedded Wizard 14

3.1 Embedded Wizard Environment 15
3.1.1 Chora 15
3.1.2 Platform Packages 15
3.2 Creating the GUI 17
3.2.1 Initialising Project 17
3.2.2 Development Steps 21
3.2.3 Device Interface 28
3.2.4 Generating code for Target 31

4 Yocto Project 32

4.1 Yocto Environment 32
4.1.1 Poky 33
4.1.2 Metadata 33
4.1.3 Open-Embedded Project 35
4.1.4 Bitbake 35

4.1.5 Board support package 35
4.2 Setting up the build environment and building the Image 36
4.3 Flashing the image on Target 37
5 Project Outcome 38
6 Conclusion 49
References 40

List of Abbreviations

BSP: Board Support Package is a collection of data that defines the support a particular hardware device, set of devices, or hardware platform needs.

CAPA: Corrective and Preventive Action is a system to correct and analyse information and identify and investigate product and quality problems.

EU: European Union is a supranational political and economic union of 27 primarily European states

GUI: Graphical User interface is a digital interface for the user to interact with graphical components.

EW: Embedded Wizard or Embedded Wizard Studio is a tool created for GUI development.

IVD: In Vitro Diagnostics are tests used to detect diseases, conditions and infections.

IVDR: In Vitro Diagnostics Regulations is a set of regulations in the EU that governs the production and distribution of IVD devices in Europe

MDR: Medical Device Regulations is a set of regulations in the EU that governs the production and distribution of medical devices in Europe

QMS: Quality Management System is a formalised system that documents processes, procedures, and responsibilities to achieve quality policies and objectives.

R&D: Research and development is a set of innovative activities by companies or governments aimed at developing new services or products, or improving existing ones.

SRA: Stringent Regulatory Authorities are national-level drug regulation authorities approved by the World Health Organization to apply stringent standards for marketing drugs, vaccines and other medical tools.

TPP: Target Product Profile is a description of the desired characteristics of a product that is aimed at a particular disease or diseases.

V&V: Verification and Validation are two procedures intended to verify that the final product meets the requirements set and it fulfils its intended purpose

1 Introduction

Medical science has advanced exponentially over the last century with the invention of numerous solutions able to detect viral and bacterial infections, conditions, and other diseases. Such tests are usually conducted either in the natural environment of the microorganisms, cells, or other biological molecules, generally meaning the body of the patient (in vivo), or in a sterile environment, outside of the microorganisms' natural habitat (in vitro).

As explained by the World Health Organization (2023) in vitro tests can be performed in laboratories, health care facilities, ambulances, or at home and range from small, handheld tests to complex medical instruments. Collectively these tests are called in vitro diagnostics (IVD), but this name is also commonly used to refer to the devices that run the tests and present the results to the operator, typically a medical professional.

Diagnosis is a critical part of health care processes, therefore the development of an IVD device is a highly regulated and complex process, including mechanical, electrical, and software elements. Each field faces numerous challenges in complying with the regulations, both within the European Union (EU) and globally. The purpose of this thesis is to introduce a subsection of software development for IVD devices, the Graphical User Interface (GUI) development, through a project created for an undisclosed IVD company. The project is using Embedded Wizard Studio as a development tool and Yocto Project as the build tool.

Software development in the case of IVD devices needs to consider numerous regulations and standards, therefore the process of development is often rigid, following the same roadmap throughout the lifecycle of the device. The first chapter of this thesis describes the process of development in regards to IVD software, including the aforementioned standards and regulations that needed to be considered, detailing the steps throughout the entire development

including post-release actions, and listing the deliverables of each phase in the development process.

The majority of the work in the life cycle of an IVD device is performed during the development phase. The second chapter of this thesis is aimed at introducing the tool used for creating the GUI, Embedded Wizard Studio including actual developmental steps taken during the project this document is based on. Firstly, it introduces the Embedded Wizard environment, including the file system and Chora, the development language of Embedded Wizard. Secondly, in this section of the thesis, the development of the GUI and the tools used for this thesis project will be described in a step-by-step process.

Lastly, the GUI developed through Embedded Wizard Studio needs an external tool to be able to be flashed onto the chosen device, which in this thesis project is Yocto Project or simply Yocto. The third chapter will introduce the environment of Yocto and describe the steps needed to build an image for the host machine, and the process of flashing the image created.

2 The product development process of IVD devices

IVD devices are highly regulated inside the EU as well as globally, therefore software development for these devices has to be planned out in detail and each step taken must be well documented. In this chapter the Product Development Process of these devices is described in detail, introducing regulations that affect the process and the final product, and listing the necessary documentation that should be derived from each step of the process.

The Product Development Process refers to all actions taken during the complete lifecycle of the IVD device, as illustrated in Figure 1.

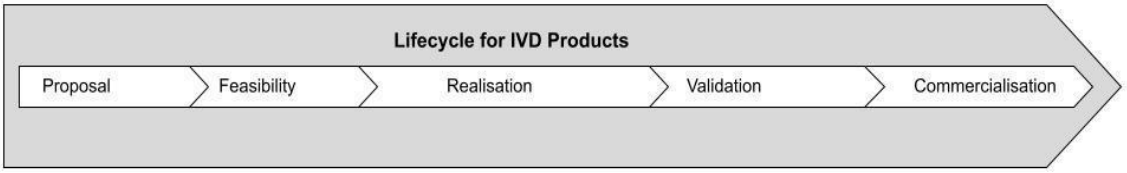


Figure 1. Steps within the Lifecycle of an IVD Product

2.1 Standards and Regulations

Due to the criticality of diagnostics in society, many regulations and standards have been defined to ensure quality, safety, and reliability. While regulations are legally enforceable explains Novak and Fekonja (2022), standards are rather guidelines that have been agreed upon at different scale institutions ranging from regional to international.

While many such standards and regulations exist in different fields of the market for specific IVD devices, some are widely used regardless of the classification of the product. ISO 13485, for example, is the regulation for Quality Management, ISO 14971 is the regulation for risk management and IEC 62304 is the Software life cycle process regulation, which describes the life cycle of the IVD product. (Novak and Fekonja, 2022)

Most significantly however, IVD devices are bound by the rules of Regulation (EU) 2017/746 (EU IVDR) as opposed to medical devices not falling under the IVD category that is regulated by Regulation (EU) 2017/745 (EU MDR). While the IVDR is meant to cover all IVD devices and the MDR does not include any IVD-related regulations, the two regulations are related and are designed to work together in harmony. This is especially important in the case of medical devices that have both IVD and non-IVD components. The main difference in these certifications is the process through which they are acquired. While for the IVDR certificate performance evaluation and performance, studies are required, for MDR clinical evaluation and the clinical evaluation report are needed. Either of these requirements is added to the CE technical file. The other significant difference between MDR and IVDR is how the post-market data is handled. For MDR an ongoing post-market follow-up is required, while for IVDR post-market surveillance and vigilance are needed. (Zvonar Brkic, 2021)

Since the thesis project is built for an IVD device, the product is covered under the IVDR, which is detailed in Chapter 2.1.1.

2.1.1 Regulation (EU) 2017/746 (EU IVDR)

Arguably one of the most important regulations of IVD devices introduces their classification based on the risk they pose to the patient and the public. In May of 2022, the new Regulation (EU) 2017/746 (EU IVDR) entered into application, replacing its predecessor, the EU In Vitro Diagnostics Directive (IVDD) 98/79/EC. IVDR defines the classification system applicable to IVDs.

Schroeder (2022) explains that the IVDR establishes four risk classes ranging from the lowest risk class (class A) to the highest (class D).

Class A has the lowest risk both to patients and public health, resulting in this class being excluded from the IVDR required notified body oversight. Examples of class A IVD tools include specimen receptacles, laboratory instruments and

buffer solutions. Since the device in the Thesis Project does not directly handle patient samples and is only reading the results of the inserted tests, it is categorised as risk class A.

Class B classified devices are a moderate risk to the patient, but a low risk to public health. This is the default classification for devices not covered by any other rules. Examples of IVDs from this class are self-testing tools, such as pregnancy, fertility and cholesterol tests, furthermore, the tests read by the device in this thesis project are also categorised as class B.

Class C covers devices that have a high patient risk with moderate public health risk. These are devices that are used for detecting infectious agents that on a faulty result have a high probability to cause severe disability or death, without a high risk to propagate the infectious agents.

Class D classified devices carry a high patient and public health risk by exposure to life-threatening transmissible agents, or by handling infectious diseases with high risk or propagation.

These classifications are outlined by 7 clearly defined rules, to guide manufacturers in identifying the risk class of an IVD device. These rules can be found in Annex VIII Chapter 2 of the Regulation (EU) 2017/746.
The rules are as follows:

1. Devices are classified as Class D if they are meant to detect or to be exposed to transmissible agents, especially if the agent can cause a life-threatening disease with a high, or suspected high risk of propagation.
2. With specific exceptions, devices that are intended to be used for blood grouping or tissue typing to assess immunological compatibility are classified as class C. Exemptions such as the ABO, Rhesus, Kell, Kidd and Duffy systems are classified as class D.

3. Class C type devices include (but are not limited to) devices that detect or are exposed to sexually transmitted diseases, infectious agents without the risk of propagation, and especially agents with a significant risk of severe disability or death in case of an erroneous result.
4. Devices aimed at self-testing are classified as class C except for a few tests that have a lower risk for the patient's health, for example, the above-mentioned pregnancy, fertility and cholesterol tests.
5. IVDs used for general laboratory use are class A classified.
6. All devices not covered in the rules above are classified as class B.
7. Control devices without a quantitative or qualitative assigned value are classified as class B.

2.2 Preliminary steps in the life cycle

The primary steps of the life cycle of an IVD product revolve around planning the device and the development process in detail, freezing ideas into documentation.

2.2.1 Proposal

Initially, a planning phase begins in which the terms and conditions of the project are discussed, including costs, a timeline with the development process broken down into achievable goals, and the planning of the development team. This phase is most often called the Proposal or the Planning stage, and it is a primarily business-driven phase that is meant to evaluate the marketability of the idea considering the expected costs. (Your Ideas Consulting, 2018)

The Research and Development side of this step consists of the creation of a proof of concept and the initial selection of technologies able to support the concept created by analysing the customer needs as well as estimated marketability. (Mugambi, Peter, Martins, and Giachetti, 2018)

2.2.2 Defining requirements

As part of the Proposal, documents for requirements are created detailing product, user, and quality requirements, furthermore, market research is conducted to further define use cases and requirements based on products of competitor companies and market gaps. Most of these documents are further developed in later stages with new insight.

In this step, the Software development plan is initiated detailing the overall process planned for the software development including fields of responsibility and breakdowns of all levels of the processes from managerial through technical to supporting process plans.

Software requirements are also defined in a software requirement specification document. In the case of this thesis project, software requirements range from non-functional requirements, such as development according to regulation IEC 62304:2006 to functional requirements, like boot-up animation, the possibility of adding a new user or deleting an existing one. The Software requirements documentation needs to include every requirement that the software needs to or should fulfil alongside the priority of each requirement ranging from ‘Nice to have’ to ‘Mandatory’. While cloud services are marked as ‘Nice to have’ in the documentation of this project, an about section where the software informs the user about certain device-related information is marked as a ‘Mandatory’ requirement.

2.2.3 Feasibility

As soon as the Proposal is approved, and the required documentation is created, the process enters the Feasibility phase. This step includes further enhancements on the deliverables from the Proposal phase as well as vast Research and Development efforts in defining requirements as specific as components in case of hardware, or software used for the development of the

source code and environment, pinpointing resources to approximately 90%. (Your Ideas Consulting, 2018)

For the thesis project, the selected tools requiring validation were Embedded Wizard Studio and Yocto. Most notably a GxP assessment needs to be carried out to determine how the intended use of tools affects the quality of the final product. For example for Embedded Wizard Studio and Yocto, the assessment concluded that the software is not GxP critical, since the end product goes through automated GUI testing and UI/UX testing, which ensures the quality of the software produced by these tools.

Furthermore, according to Mugambi, Peter, Martins, and Giachetti (2018), the already identified customer needs are finalised in this step, translating the user requirements into a product requirements document often called Target Product Profile (TPP), specifying the context in which the product will be used, commonly called use cases.

Within this step development plans are also finalised, including clinical, regulatory, manufacturing, and marketing plans.

The final goal of Feasibility is to present the best product solution considering alternative realisations throughout the process, creating a 90% stable plan as well as the product risk analysis is usually initiated at this phase of the process.

Usually, a demo device is also produced to prove the feasibility of all product requirements.

2.3 Development phase

Based on the finalised product requirements, development starts with the selected components and tools after finalising Feasibility. During this phase all components are finalised, often taking into consideration feedback from end users and stakeholders.

The developers often re-evaluate priorities, and the importance of features is reassessed to create the most optimal design, however, in the case of medical devices and IVD devices, the process of making changes is more complex than in the case of regular software. Every change created has to have a solid basis and it needs to go through approval from the project board or steering committee.

The end goal of the Development phase is to create the final product, freeze the design, and make it ready for external testing by the intended user. By the end of this phase, the product must be manufacturable.

The final product is verified at the end of this phase to conclude if all technical risks are mitigated, and that the device meets all the predetermined performance requirements (Sippola, 2020) as well as manufacturing and quality control documentation is created for all components, making manufacturing consistent and verification assured.

At the end of the development phase tests are conducted according to an already defined Software Test Plan defined in the Planning phase. Usually, Unit, Integration and SW System testings take place as well as Validation (user) testing, Exploratory testing and Regression Testing.

In the case of the thesis project Embedded Wizard offers a test framework for testing the GUI, as well as the active build during the development phase is continuously tested through Jenkins.

2.4 Release

Before the final product of the Development phase can be released, there are further validation activities that need to be conducted by the developer to ensure the device is compliant with all regulations and standards. Clinical validation studies need to be undertaken and prepared to be submitted for approval, and lastly, the already drawn marketing plan has to be executed, and communication with vendors is established. (Mugambi, Peter, Martins, and Giachetti, 2018)

2.4.1 Validation

All the documentation from the Proposal to the end of the Development phase has to be finalised and validated before the launch of the product including any deviations during the process. This part of the process is called validation and verification (V&V).

2.4.2 Regulatory approvals

Due to the critical nature of IVD devices and often the global scope of the market, a vast number of regulatory approvals need to be conducted before the product can be released, often depending on the country of distribution, or the risk category the product belongs to. Mugambi, Peter, Martins, and Giachetti (2018) explain that approvals are handled by stringent regulatory authorities (SRA-s), such as the Food and Drug Administration of the United States and the European Commission Directorate General of the EU. Global assurance is provided by the WHO Prequalification of In-Vitro Diagnostics Program (WHO DxPQ), which simplifies the evaluation of product quality and safety for lower and middle-income countries that cannot independently evaluate products and manufacturers. Some countries accept international approvals, while others require local acceptance procedures accompanied by performance evaluations carried out by country-level laboratories.

2.4.3 First launch

Following successful Validation and with all Regulatory approvals acquired, the manufacturing sites begin to assemble the product and ready them for sale. When products are ready to be sold, the Launch begins, and the product life cycle enters the Post-release phase.

2.5 Post-release processes

The post-release responsibilities of the developer consist of detecting and fixing bugs that went unnoticed during testing and fixing any new security issues that arise from advancing malicious actors and malware. This is done through either the developer's customer service or a third-party technical-support organisation.

The issues that were not resolved before release are often reported on in post-market surveillance organised by the manufacturer. This task is part of the Corrective and Preventive Action (CAPA) process, which often already starts during the Planning phase in the form of prevention of foreseeable issues. (Sippola, 2020)

The CAPA process consists of the following steps:

- Recognition of the nonconformity and systematic data collection
- Description of the nonconformity
- Risk assessment and prioritisation
- Effect analysis
- Correction (if required)
- Root cause analysis
- Action definition
- Conducting defined actions
- Conclusion of the CAPA process
- Effectiveness evaluation
- Reporting and Management review

As the list suggests, the process of recognizing and correcting a problem in the product is also a process that needs to be well thought out and documented.

In software, problems are usually solved by releasing a new software update, but apart from bug fixes the update often contains new functionalities added to the software that were not finalised before release. (Shaha, 2022) Software updates are a likely solution for implementing the ‘Important’ and ‘Nice to have’ requirements that the initial release does not include.
In the case of the thesis project, a likely later addition would be connectivity for example.

Furthermore, the manufacturer often offers training for the customers concerning the use of the product.

2.6 Deliverables

Throughout the development process every step, plan and intention must be traceable and well-documented to comply with the regulations and standards mentioned in chapter 2.1, especially IEC 62304:2006, which describes the Software life cycle processes mentioned in the Thesis so far.

During the Planning and feasibility phase, references are created to the Quality Management System (QMS) and Risk Management. A Risk management and a Verification plan have to be developed, and the classification based on the rules previously mentioned, that are defined in the EU IVDR has to be made. Furthermore, Software Development, Integration, Test and Configuration Management Plans are also created alongside SW Requirement Specification and documentation on the overall SW architecture and descriptions of the working principles.

In the Software Development phase, the Product requirements are documented, a Hazard or Risk Analysis is conducted as well as the source

code is documented containing version control in a Software Unit Implementation document.

During Software System testing the Test Protocol and a Testing and Verification Summary report are produced containing the Test Cases, Test Reports, a description of identified anomalies and all the Test Summary Reports.

Before release, any additional Software Release Documentation must be completed and all of the above documents must be finalised and approved.

Post-release actions as per the CAPA process have to be also well documented by conducting a risk assessment on any irregularities and resulting bug fixes.

During the life cycle of an IVD device, numerous documents are created based on the risk classification, the type of device and the approach of the developer with two main focuses being traceability and quality assurance.

3 Embedded Wizard

Technology rarely stays a secret in these times, and with competitors being able to often match the functionality of each device on the market, appealing to potential customers is often achieved by creating aesthetic, intuitive and straightforward features. A significant deciding factor for many is the Graphical User Interface (GUI).

Several tools can be used to develop GUI for embedded devices and the selection of the tool used for the project is not limited by regulations, since the end product is heavily tested and validated before release. The process however is well documented for traceability reasons.

For this project, there were two commercial alternatives and three open-source options considered besides Embedded Wizard Studio (EW), such as Qt, Storyboard, Chromium Embedded Framework and Electron. Since Embedded Wizard was the newest and most modern technology with support for numerous target devices, it was concluded to be the best option, guaranteeing end-of-life support for the final product. Therefore, after verifying that the software could be used to implement all the GUI requirements the decision was made to use Embedded Wizard Studio.

Embedded Wizard Studio is developed and distributed by TARA Systems GmbH, a Munich-based German company, and it specialises in the creation of GUI for embedded systems. Originally released in 2003 intended to replace the M2-Builder. The first stable release of Embedded Wizard was in May 2019, releasing version 11 of the tool.

The primary advantage of Embedded Wizard, explained by Banach and Schweyer (2023) is the ability to develop independently of the platform with the help of Chora, the programming language of EW. Chora supports object-oriented programming and resource-constrained devices.

Embedded Wizard also offers a 'What you see is what you get' (WYSIWYG) front end for graphical editing, support for animations with a 3D perception, and vector graphics.

3.1 Embedded Wizard Environment

Banach and Schweyer (2023) further detail that the Embedded Wizard Environment consists of Embedded Wizard studio, the aforementioned WYSIWYG Integrated Development Environment (IDE), and platform packages that serve the purpose of a hardware abstraction layer.
These platform packages are offered by TARA Systems for numerous embedded chipsets, colour formats, and operating systems. They each contain a code generator and converter for bitmaps and True Type fonts.

3.1.1 Chora

Embedded Wizard Studio is built upon the programming language, Chora. Chora is a platform-independent, object-oriented programming language, it was developed based upon widely known languages like C, C++, Java, and JavaScript, therefore it is largely similar in syntax to C++ and JavaScript minimising required training efforts.

Chora is used as source code for visual elements and as a medium for saving any EW project. This allows developers to implement the complete functionality of the GUI without any additional development tools, like external code editors. (Banach and Schweyer, 2023)

3.1.2 Platform Packages

The most notable feature of Embedded Wizard Studio is the ability to develop simultaneously for several target devices within the same project devices, which is achieved through Platform Packages.

Embedded Wizard supports over 60 different platform configurations through the individually optimised platform packages, that contain four major elements. These elements are the Code Generator, the Resource Converters, the Runtime Environment (RTE) and the Graphics Engine (GFX).
The Code Generator translates the platform-independent project into code comprehensive to the target system. This includes for example the possibility to add files to the Keil MDK-ARM environment or a GCC makefile.
This thesis project uses a specific processor as a target device and the generated C code is added to a makefile using autotools or automake.

Image assets and font files used in the GUI are converted to be suitable for the target system by the Resource Converters. This process can result in colour reduction dithering and compression of resources or for example, pre-rasterisation of font glyphs in case the target has no True Type font engine.

The Runtime Environment is a library with the function of establishing communication between the target system and the generated code. The RTE contains all the necessary common functions that the GUI needs to run.

Another library that is a part of each platform package is the Graphics Engine. All the graphical operations are performed through the GFX, for example, drawing lines, 3D transforming images or rendering vector graphics. The GFX is optimised to use graphics hardware acceleration, but if it is not available in the target system the library will perform the graphical operations through optimised SW routines.

3.2 Creating the GUI

The Project described in this document is a GUI designed and developed for an IVD device. Parts of the visual elements are hidden or modified in this thesis due to confidentiality reasons, however, the process of building the elements, and the resulting functionalities are included fully.

3.2.1 Initialising Project

Loading up Embedded Wizard Studio, the software prompts the user with a dialogue asking if it should open an example of a previous project, or if the user wants to create an entirely new project. The same dialogue contains links to resources that are very useful, and sometimes even vital for the development process. See Figure 2.

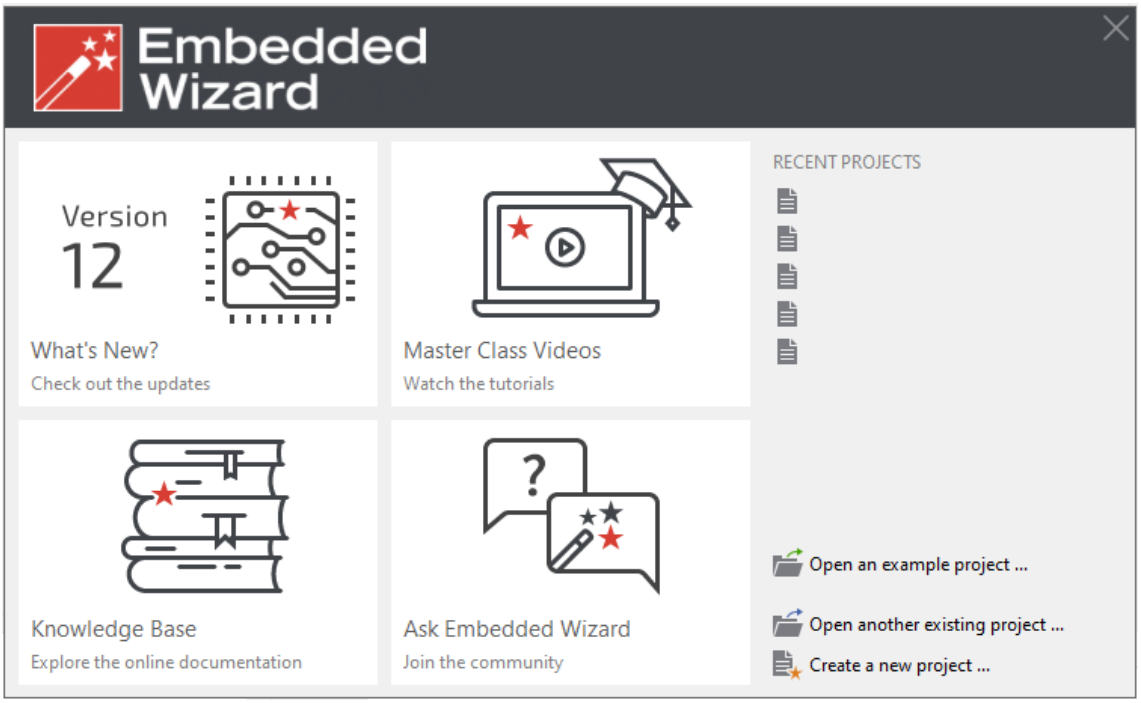


Figure 2. Embedded Wizard Studio view on starting the software

After starting a new project, the user needs to select from some settings, such as the platform package the user intends to use, the size of the screen and the name and location of the project save files. Other platforms can be added later in the project as well.

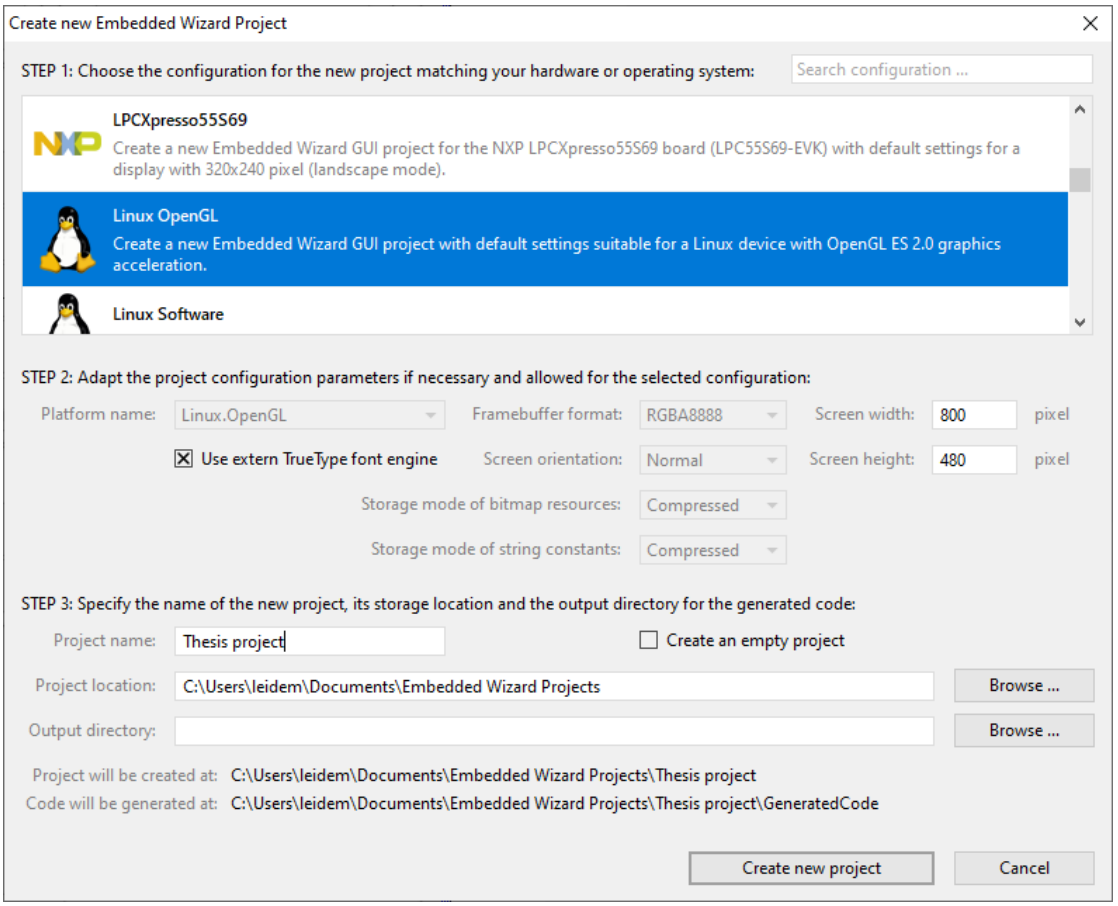


Figure 3. Creating New Project in Embedded Wizard Studio

For the target device in this project, Yocto Project is used to compile the generated code, therefore the Platform is Linux OpenGL, and the size of the screen is the default 800x400 as shown in Figure 3 above.

Creating the new project, Embedded Wizard generates the elements necessary for an Embedded Wizard project to function. These elements are represented by bricks on the screen. Bricks are a member of the Embedded Wizard framework representing elements such as Variables, Slot Methods, Methods and Configurations for visual elements, for example, Gauges. The size of the bricks can be adjusted, and they can be rearranged on the screen, however, their size and location do not affect the GUI. Upon selecting a brick, the details of any items appear on the right side of the screen under the list of elements module, and after double clicking on one, it either opens, if it is a Script type brick, like a Slot Method, or it opens the layout of the module in a new tab allowing the user to modify how certain attributes are processed.

As seen in Figure 4 under Profile Configuration the software generates a brick called profile, populated with attributes using the settings from the Create New Project view. In the case of an application that is simultaneously developed for multiple platforms, another profile can be added with different attributes, such as a Platform package.

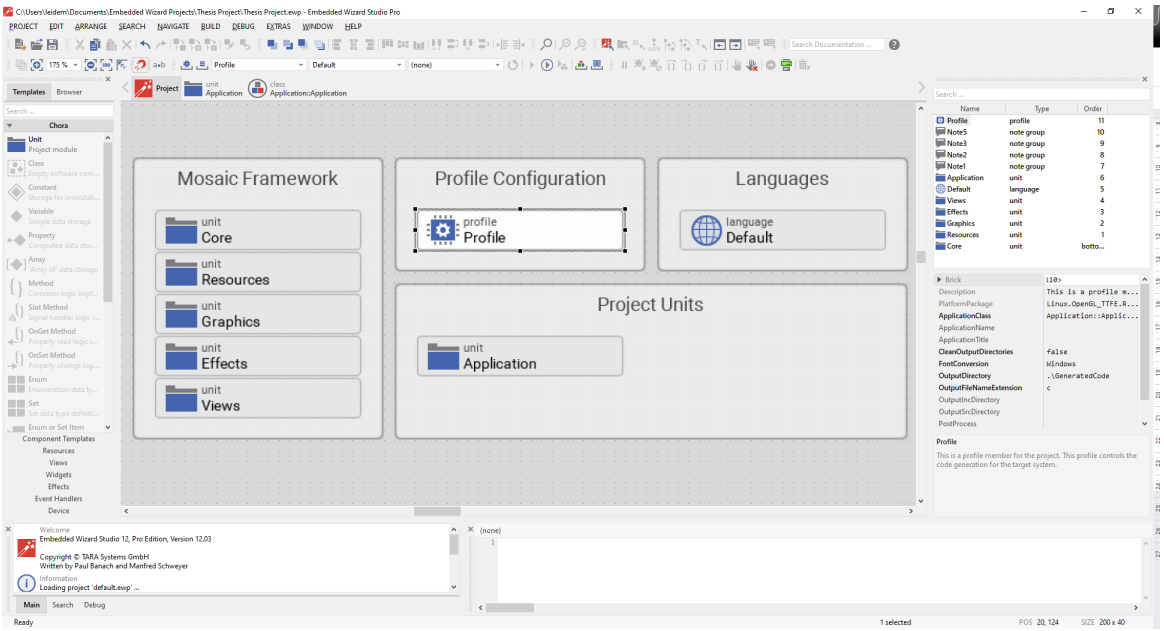


Figure 4. Parts of a new Embedded Wizard Studio Project

Alongside the profile brick, Embedded Wizard Studio also creates a default language brick for localisation as well as units containing the basic libraries provided by the software. These units are called Mosaic Framework, and they include the key components a typical GUI application uses, such as key and touch handlers, timers, effects and other graphical elements. These units can not be edited or modified, however, the working unit Application inherits the functionalities of the Framework and can extend or override them if necessary.

Lastly, it also generates the unit called Application, which is the main workspace. It is possible and recommended to use several units for elements within the project, for example in this Thesis Project besides the automatically generated units, there is a Bitmaps unit and a Strings unit containing all the bitmaps and string constants needed for this project.

Within the Application unit after initialisation, there is a class called Application which is the root component of the GUI belonging to the SuperClass Core::Root as well as a basic Font brick. All fonts used in the project must be created in this unit before they can be deployed in classes.

Within the root class Application, a simple demo is generated on project initialisation. Upon playing it in the prototyper, a simple function begins to play, in which the text “Welcome to Embedded Wizard” fades in and out in randomised locations and randomised colours on the screen size that was selected during the initialisation of the project.

The Prototyper is a very important part of the development process since it allows the developer to try out everything that is seen on the screen at runtime. It can be run from the two arrow icons in the second line of the taskbar. Left to these icons there are three scroll-down menus for changing the style, language and profile, and finally, on the left of the profile selection, the two build icons can be found.

3.2.2 Development Steps

Everything visual that happens on runtime needs to be present in the root class Application either as an invoked component or as a pop-up dialogue.

A new component is created from the Application unit view by using the left-hand side browser to search for Component or by opening up the Component Templates dropdown menu from under the search bar and dragging and dropping the component element.

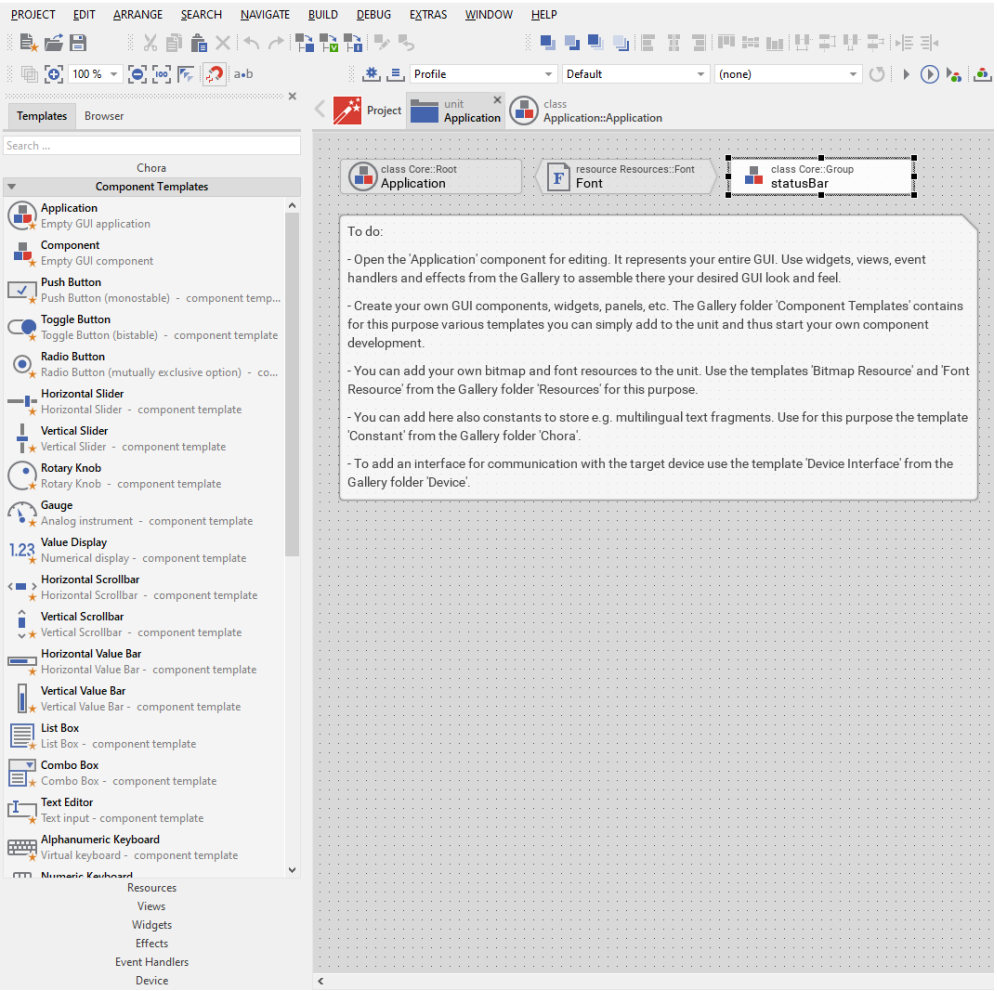


Figure 5. Adding New Component named ‘statusBar’

Figure 5 shows the component renamed ‘statusBar’ as well as the location from which it can be found. Upon opening the component the right dimensions need to be set using the Bounds property brick.

The width of the application is 800 pixels, while the height is 400, therefore the status bar would look the most appealing with the dimensions of 800x40. As a background, a filled rectangle is the most simple solution filled with the colour #414445FF. The colour of the rectangle can be changed as well as the dimensions with the rest of the attributes in the right-hand tab mentioned before in chapter 3.2.1, for changing the dimensions of any visual elements, dragging it with the mouse or by using keypad actions is also possible.

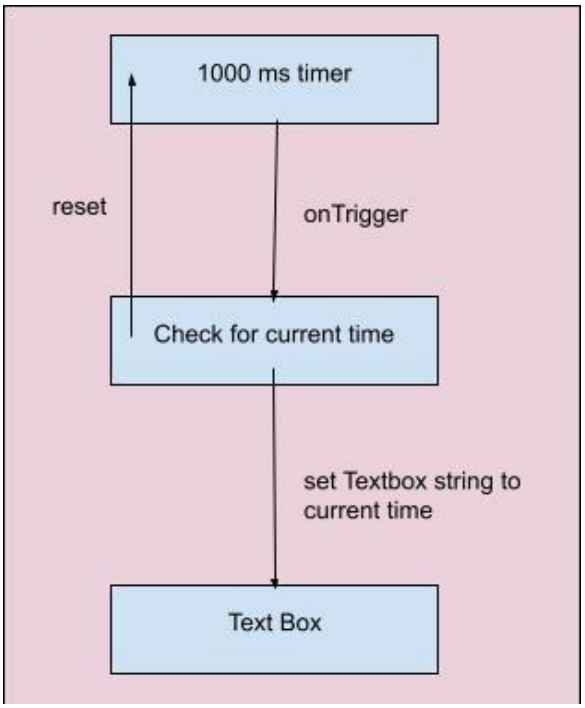
Since status bars usually display the time, the first element of the thesis project is a text field for the output of the time, and a simple slot method triggered by a timer. The timer in this example is set to trigger every 1000 milliseconds, which runs the code within the slot method.

slot Application::statusBar.clockHandler

```
1 var Core::Time currentTime = (new Core::Time).CurrentTime;
2 this.clock.String = currentTime.Format( "%H:%M" );
3 this.clockTimer.Enabled = true;
4
5
6
7
```

Figure 6. Clock updating program in Chora

As Figure 6 shows, updating the time in the prototyper happens through accessing the Time class from within the Core library, creating a local variable of the same class and setting it equal to the CurrentTime element of the class, then using Core::Time currentTime and formatting it to display time as hh: mm the text field is populated with the current time. Finally, the code restarts the timer, which as result checks the current time every second. See Figure 7.



15:47

Figure 7. The block diagram and output of the ‘clockHandler’ script

In practice, time is most often data that is passed to the GUI through the Device Interface originating from the board. The Device Interface and communication with the board are explained in Chapter 3.2.3.

To add the finished status bar to the application, from the Application editor view, on the left tab, next to the Templates ear, the component can be searched for by name in the Browser view. Aligning this element on top of the main view creates a local iteration of the status bar with the clock functionality.

Elements of the status bar component within the root Application can be accessed easily, whereas accessing root elements from the status bar works a bit differently.

In the case of a hamburger menu, an icon from a taskbar should handle opening and closing the component. As user interactions are handled by event handlers in Embedded Wizard, a device with a touch screen can use Touch

handlers. It is possible to assign a Slot Method from the root Application to touch handlers of local iterations of the taskbar and the menu. Since these touch handlers get initialised with the function assigned to them by the slot method, they can be assigned within the Init() method of root Application. The Init() method is called automatically when the component is created, unlike other methods that need to be invoked.

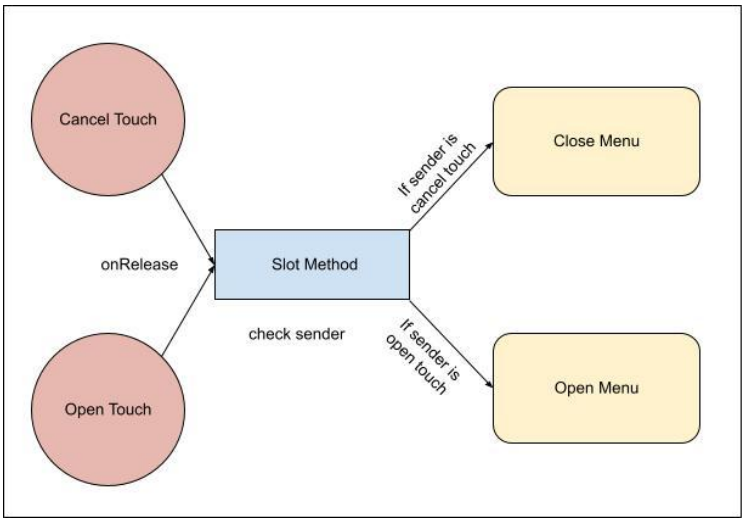


Figure 8. Block diagram of Slot Method choosing an action based on the sender

inherited method void Application::Application.Init(arg handle aArg)

```
1 //  Assign toggleMenu Slot to the hamburger menu and the X
2 //  touch handlers in the compobebts
3
4 this.taskBar.openMenuTouch.OnRelease = toggleMenuSlot;
5 this.menu.closeMenuTouch.OnRelease = toggleMenuSlot;
6
```

slot Application::Application.toggleMenuSlot

```
1  //  this script checks which touch handler called the slot
2  //  and opens or closes the menu accordingly
3
4  if ( sender == this.menu.closeMenuTouch )
5  {
6   this.menu.Visible = false;
7   this.menu.Enabled = false;
8  }
9  else if ( sender == this.taskBar.openMenuTouch )
10 {
11  this.menu.Visible = true;
12  this.menu.Enabled = true;
13 }
14
15
```

Figure 9. Init() and toggleMenu scripts

Figures 8 and 9 show the simple script assigning toggleMenu to the Touch Handlers in the taskbar and the menu, followed by the contents of toggleMenu. This example shows how user interactions are handled between components.

In case the interaction between the components is not directly initiated by user input, a slot type variable or property is used to create access to the root application from the components.

While slot methods within the same layer or below can be signalled from another script, if the slot called is in a layer above the method called, a placeholder needs to be created within the layer of the sender. For instance, if the user picks a menu option, that closes the menu and opens up another view within the root Application, the slot method for opening the new view needs to be created within the root, and in the menu component a slot type variable is created to serve as a host for the openView slot. The created slot method is then assigned to the slot variable within the menu component, and it can be called by another slot from within the menu component.

In case the action is initiated by a Property change, Property Observers can be set up in the root, or any layer above the layer where the Property is defined. For example, if the clock in the status bar needs to update a clock on the main view, within the status bar there needs to be a string type Property, which on change notifies all observers. The observer set up in root has a OnEvent slot, describing the actions that need to be performed when the Property in the status bar changes.

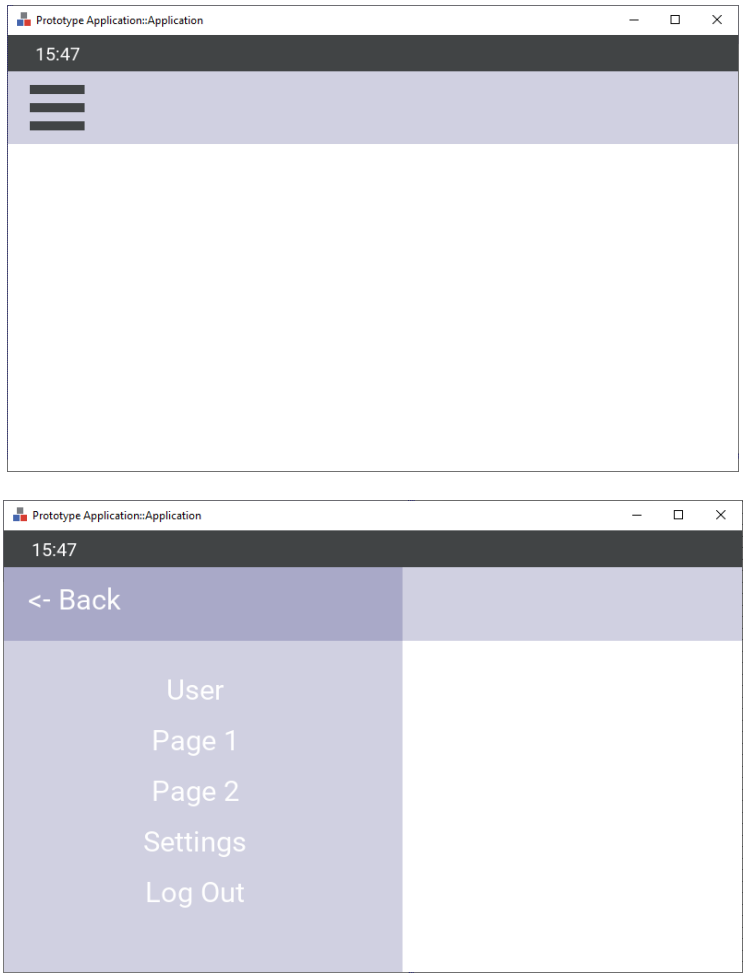


Figure 9. Main View and Menu Open View

An example of a more complex view in the thesis project is the keyboard component. Embedded Wizard has an alfa version keyboard on which the developer can freely base the keyboard used in the end product. This Alfa version contains the scripts for loading the keyboard layout from a string variable as well as the script necessary for understanding input through the keyboard. The thesis project in addition to the basic project contains localised keyboard layouts, pop-up bubbles for the character being held down as well as a dialogue window for special characters for each layout on specific keys being held down for a set period.

Figure 10. Keyboard and Special key Pop-up

As illustrated in Figure 10, character layout is defined in a string type constant in the Strings unit and localised so that each language loads its respective keyboard layout. The special keys are defined in string-type variables within the class, and upon recognising the character being pressed, a pop-up view appears with the special characters connected to the base character. This function is also case sensitive, therefore a capital A will return the capitalised special characters ‘Æ Ä Å À Á Â Ã Ā Ă Ą’.

Fonts are a member of the resources library and they can be called and defined in the Application unit. Some of the attributes that can be set for this member are the Font Name, the Height in pixels, the Height Mode, if the font is Italic or Bold, if the Kerning data is read from the True Type file or if the default kerning is used offered by Embedded Wizard Studio, and lastly the Ranges.
Some special characters are not included in the default range of font, therefore any font used with these characters needs to be modified and the range set to be able to print and read such characters. For Latin letters to contain all the special characters the range used in the project is 0x20-0x17F which is the hexadecimal notation of 32-383. For Chinese characters, the range is 0x4E00-0x9FF and for Korean it is 0xAC00-0xD7AF. The ranges are determined by UNICODE. Not all fonts support all ranges, therefore the font has to be carefully selected based on not just design but taking localisation into account.

Language is a global, built-in variable in Embedded Wizard that any initialised language member can be assigned to on runtime. Both bitmaps and constants

have language-specific value slots, Therefore for a localised app text elements in the GUI should always use string-type constants as a source. In addition to using constants, the components need to be set multilingual as well from the Application unit view.

3.2.3 Device Interface

Communication between the GUI and backend is done through a class named Device Class or Device Interface. Located in the Application unit, it consists of a class block and an autoobject block as well as an Inline Code block. The Inline Code is the tool to incorporate external dependencies to the project, like C header files, type declarations or functions. The autoobject is a globally defined instance of a class making it accessible throughout the entire Application.

Communication with the backend is also possible without the Device Interface class, however, it is good practice to create a dedicated access point for such communication to separate the View and Controller from the Model. This allows users to make independent changes easier and to create a GUI capable of controlling several hardware variants and product types.

The communication between GUI elements goes as follows: The View and Controller elements in the GUI communicate through the autoobject with the Device Class, which communicates with the Device Driver usually named DeviceDriver.c. The Device Driver is an additional layer between the device and the GUI automatically created at the start of a new project. During development, the Device Driver is constantly updated with new functionality.

In case the GUI expects input from the backend, UpdateProperty methods as well as System Events can be used to wait or probe for input. In the case of the Thesis Project, the communication is implemented through a Property and the methods, UpdateProperty(), OnSetProperty() and OnGetProperty(). For example, when the Device is notified that the test is inserted, the Device Driver

calls the UpdateProperty() function of the Device Class, which in turn changes the bool type Property to the opposite value.

Placed within the Application class there are Property observers set to observe the Application::Device.Property, the Property within the autoobject of the Device Class. On change, the OnSetProperty() method sends a notifyobservers command to all observers following the Property. The Observer on receiving the signal initiates the script assigned to it, which in the case of the Thesis Project is, whenever the device senses the test inserted, it closes the window after the test and signals the Device class to start the measurement.

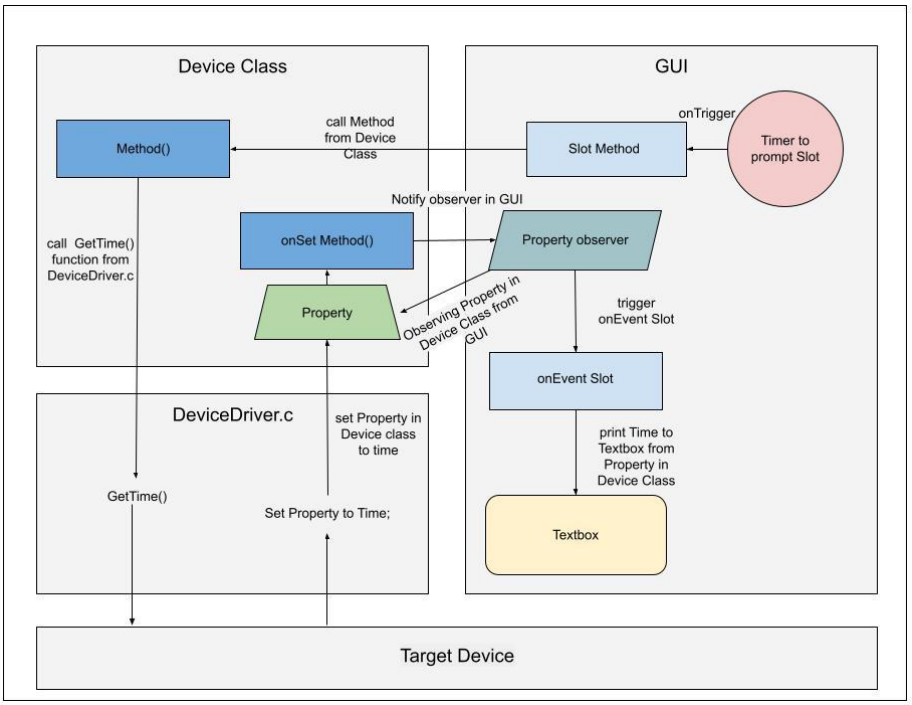


Figure 11. The infrastructure of communication between the GUI and the Device

Figure 11 shows the architecture that supports communication throughout the device. The GUI reacts to for example a timer with a Slot Method attached to it as an onTrigger action, The Slot has a script which communicates with a specific Method in the Device Class.

```
slot Application::Application.getTime
1 Application::Device.GetTime();
2
```

Figure 12. Calling GetTime() Method from the Device Class

Figure 12 shows how the onRelease script calls the Method() from the Device class.

```
method void Application::DeviceClass.GetTime()
1 trace "Timer prompted from method";
2
3 $if $profile==Profile
4 native
5 {
6   int promptTimer = DeviceDriver_GetTime();
7 }
8 $endif
```

```
int DeviceDriver_GetTimer(void)
{
    //Script prompts target device for current time
}
```

Figure 13. The script in the Device Interface calls the function from the Device Driver

Figure 13 demonstrates the Method calling the function defined in DeviceDriver.c. The script sends a command and prompts the target device to get the current Time.

Once the device gets the Time, it sends the result to DeviceDriver.c where a function sets the value of a Property designated for the Time equal to the value it received from the target device. Inside the GUI there is a Property observer defined, set to watch the Time property. On a new value, the Property notifies its observers through a Method. On receiving the notification, the Property observer executes a script to display the Time from the Property in Device Class. Figure 14 below shows the script that copies the value of the Property.

```
slot Application::Application.showTime
1 this.timeText.String = Application::Device.Time;
2
3
```

Figure 14. Script of the Property observer

3.2.4 Generating Code for Target

When the GUI is ready to be deployed on the hardware, Embedded Wizard Studio generates all the necessary files for the target device after selecting the profile the user wants to build and pressing one of the Build icons. See Figure 15.

Figure 15. Build this profile, Build in Batch and profile selection menu

All the files generated are saved in a folder named after the profile unless otherwise specified. This folder contains C files and headers for each unit and C header files for each class and bitmap in the project. Furthermore, .inc and .lst files is listing the units and connecting the modules to the C files.

A collection of the files above is added to Makefile.am, which is essentially the recipe created with autotools to generate a Makefile. The Makefile is then used by the Yocto build to create the image that can be flashed to the target device.

4 Yocto Project

The final step in this Thesis project is building the image for the device, which is done by using Yocto Project.

Yocto Project is an open-source Linux Foundation collaborative project created to assemble customised Linux distributions for embedded and IoT devices.

Yocto takes a set of data as input defining the result, for example, kernel configurations, hardware-specific data and packages and binaries for the filesystem, and produces the output, which is the Linux kernel, the Root filesystem, the Bootloader, the Device Tree and the Toolchain.

The Toolchain includes the compiler, the tools needed to create the code for the target device, and everything else in the Yocto build depends on it, while the Device tree contains information about the target device and the hardware structure the distribution is built for. The Bootloader is the program that initialises the board and loads the Kernel.

The Kernel is the heart of the system, it manages resources and interfacing with the hardware while the Root filesystem consists of libraries and programs that are run once the Kernel is initialised.

4.1 Yocto Environment

The prerequisites of building a Yocto distribution explained by the Linux Foundation (2023) are 50 Gigabytes of free disk space, a Linux machine running a supported distribution (for example a recent release of Fedora, Debian or Ubuntu), and the following packages:

- Git 1.8.3.1 or greater
- tar 1.28 or greater
- Python 3.6.0 or greater
- gcc 7.5 or greater

- GNU make 4.0 or greater

For this project an Ubuntu virtual machine is used for building the Yocto Image for the target device, equipped with the specified memory and packages.

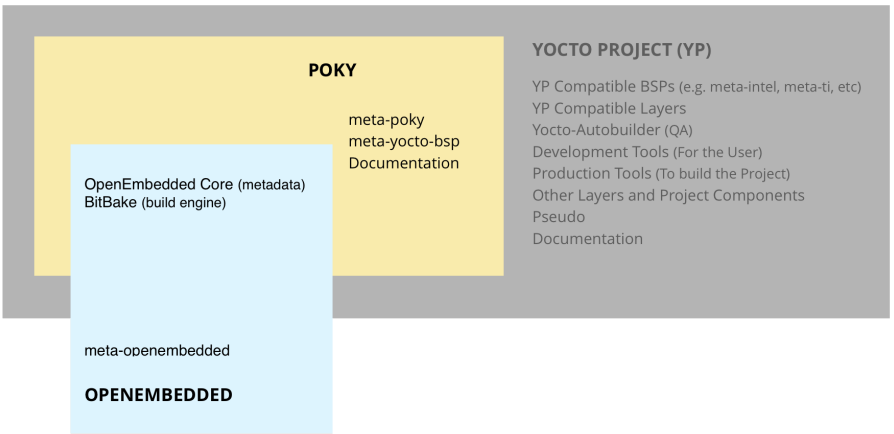


Figure 15. The Yocto Project Environment (source: Linux Foundation (2023))

4.1.1 Poky

Poky is the reference distribution of the Yocto Project. Yocto Project uses Poky to build images. It is a combined repository of Bitbake, Open Embedded core, Meta-yocto-bsp and the documentation.

4.1.2 Metadata

Metadata is a set of data that describes or provides information about other data. Metadata when in the context of Yocto refers to build instructions, commands and data used to indicate the build version, the source of build requirements and changes and additions to the software.

Metadata consists of the following:

- configuration files (.conf)
- recipes (.bb and .bbappend)
- classes (.bbclass)
- includes (.incl)

```
SUMMARY = "     GUI application"
SECTION = "
LICENSE = "CLOSED"

PR = "r1"

DEPENDS = "autoconf-archive-native                                                                    python3-native"
EXTRANATIVEPATH += "python3-native"

RDEPENDS_${PN} =

SRCREV = "${AUTOREV}"
BPV = "0.0"
PV = "${BPV}+git${SRCPV}"

SRC_URI =  github.org/GUI

S = "${WORKDIR}/git"

inherit autotools pkgconfig
```

Figure 16. Recipe for the GUI software pointing to the repository

Recipe files contain a set of instructions that are processed during build by the task scheduler, bitbake. They inform on the source of the code, configuration options, compilation options, patches to apply and the license information. Recipes can be created by tools such as devtool or recipetool. The recipe file for the Thesis project is shown in Figure 16 pointing to the repository where the C files produced by EW are stored.

A collection of related recipes is called a Layer. They are essentially the containers of these recipes. Poky comes with a few layers, including meta, meta-poky, meta-selftest, meta-skeleton and meta-yocto-bsp. Also included in the thesis project are layers for additional parts of the device, like the layers for the sensors, the camera and the motor. The purpose of layers is to categorise metadata by functionality and create a possibility for later, reuse and customisation.

Configuration files hold global definitions of variables, default or user-defined as well as hardware configuration information. These files direct the build system on what to build and include in the image to support the target platform. Types of configuration files are:

- machine configuration
- distribution configuration
- compiler tuning configuration
- general configuration
- user configuration (local.conf)

Classes are also elements in Yocto. The purpose of a class is to abstract commonly used functionality to share amongst multiple recipe files. Recipes inherit from classes, for example, the name of the classes.

4.1.3 Open-Embedded Project

Open Embedded or oe-core shares a collection of metadata with Yocto, however, while Yocto offers largely tools, metadata and board support messages for a core set of architectures, oe-core focuses on providing a comprehensive set of metadata. Shortly, Open Embedded is designed to be a foundation.

4.1.4 Bitbake

Bitbake is a task scheduler, similar in functionality to make. It parses a mixed code of Python and shell script. The parsed code generates and executes tasks ordered by the code's dependencies. It builds fetched packages into bootable images, furthermore, it tracks tasks and utilises the maximum processing power to reduce the build time and increase predictability.

The initial build of the Thesis project takes 5-10 hours, later significantly less, depending on the amount and quality of changes made. When the only change in the new build is the GUI, the process takes about 5 minutes.

4.1.5 Board support package

A Board support package is a collection of information detailing the support a particular target device needs. This includes information on hardware features present, kernel configuration and additional hardware drivers.

Yocto comes with a set of Board support packages called Meta-yocto-bsp included in Poky. Meta-yocto-bsp supports BeagleBone, EdgeRouter and some other standard 32 or 64-bit devices.

Since the project in this thesis is built for a different processor a separate board support package needs to be included in the build. Websites like layers.openembedded.org and yoctoproject.org grant access to many board support packages.

The board support package used in the project can be changed in *~projectfolder/build/conf/bblayers.conf*, while the target device can be named in *~projectfolder/build/conf/local.conf*.

4.2 Setting up build environment and building the image

When building a basic Yocto distribution through Poky, the developer first needs to access the skeleton that Poky offers by cloning it from the official yoctoproject.org git repository.

When the project layers are all on the build machine, the command *./fetch.py layer_data.json* can be run in the terminal in the Project folder to fetch additional metadata, and then with the *patch* command patches are applied.

The command *. poky/oe-init-build-env* is used to set up the build environment. Then finally, *bitbake image-name* starts the build of the image using bitbake.

The Embedded Wizard Studio files are built with gcc compiler. The binary is then copied to Yocto.

Bitbake first reads the architecture, policies, configuration details and patches defined by the developers, and then the build system fetches and downloads the source code from specified locations. The third step for Bitbake is to extract all sources into the temporary directory, which is followed by the application of patches. In the fifth step software is configured and compiled. Next, the temporary staging area gets software installed, the binary package feed is generated for creating the final root file image, and finally, the build system generates the file system image and the customised extensible SDK for application development in parallel.

4.3 Flashing the image on the target

After the build is finished, the image to be flashed onto the target device can be found in the *~projectfolder/build/tmp/deploy/images/machine* folder. The .wic type file is then flashed on the target machine using the Universal Update Utility (UUU) tool.

5 Project Outcome

The purpose of this project was to create a GUI that can communicate with the target device and send and receive data on runtime incorporating user input. Both communications from the GUI towards the backend and communication initiated by the hardware towards the GUI have been successfully established by the end of the project timeline.

The development process was greatly cooperative with other developers, designers and other professionals, and the project was highly dependent on the other members of the development team, which offered significant advantages, such as accessible expertise in a very broad set of fields.

Furthermore, due to the criticality of the IVD field every step of the development process had to be documented and many of the decisions, may they be design decisions or software development decisions, had to be based on careful assessment and often involved a validation process.

In the end, all goals that were set out for the GUI for this project period have been met, which established Embedded Wizard Studio as a great tool for a project this scale.

For the future of this project, it would be interesting to consider cloud connectivity as the next step. Both from the development perspective and the IVD/MD fields' unique responsibilities considered, such a step would pose further significant challenges. For example, privacy (ISO 27001) and security (SOC 2) standards would need to be met, as well as further EU and other local regulations need to be followed if the project is to head in that direction.

6 Conclusion

Software development for IVD devices is a demanding, strictly controlled process, where almost every new functionality results in an additional regulation or standard needing to be considered.

In the Graphical User Interface of such devices, there are often dozens, if not hundreds of possible layouts for different cases, which are all interconnected like a web, being triggered by a combination of attributes. This fragile net of information is then connected with the just as complex and possibly even more intricate backend, resulting in a beautiful, but often just as frustrating system created by several people responsible for snippets of the whole project.

While this process is undoubtedly tedious sometimes, it is still a great experience to be part of creating something that will benefit humanity and to have the opportunity to work with great professionals in their respective fields. The regulations that are often so binding help keep track of the priorities and often spare the project team from taking too big of a bite from the endless possibilities of Software and Hardware development. Nevertheless, embedded development offers a seemingly endless opportunity to learn new skills, technologies and tools.

References

Banach, Paul & Schweyer, Manfred. 2023. Embedded Wizard Documentation. doc.embedded-wizard.de. (accessed first: 05.06.2022)

IEC 62304:2006(en). Medical device software — Software life cycle processes (accessed first: 01.02.2023)

Linux Foundation. 2023. Yocto Project documentation. docs.yoctoproject.org (accessed first: 01.09.2022)

Mugambi,Melissa Latigo & Peter,Trevor & Martins, Samuel F & Giachetti, Cristina. 2018. How to implement new diagnostic products in low-resource settings: an end-to-end framework. ncbi.nlm.nih.gov. (accessed first: 10.02.2023)

Novak,Maruša & Fekonja, Ana. 2022. IVD medical device software development and regulations. biosistemika.com (accessed first: 27.01.2023)

Shaha, Anuradha. 2022. A Guide to SaMD (Software as a Medical Device). operonstrategist.com (accessed first: 20.02.2023)

Regulation (EU) 2017/746 of the European Parliament and of the Council of 5 April 2017 on in vitro diagnostic medical devices and repealing Directive 98/79/EC and Commission Decision 2010/227/EU. 2017. eur-lex.europa.eu (accessed first: 25.02.2023)

Schroeder, Wade. 2022. Explaining IVDR Classification for In Vitro Medical Devices. greenlight.guru (accessed first: 25.02.2023)

Sippola, Juha. 2020. Medical software development according to Medical Device Regulation (MDR) – part 4: Medical device software lifecycle processes. blog.pinja.com (accessed first: 01.03.2023)

World Health Organization. 2023. In Vitro Diagnostics. who.int (accessed first: 25.01.2023)

Your Ideas Consulting. 2018. Product Development Overview. your-ideas.net (accessed first: 25.01.2023)

Zvonar Brkic, Kristina. 2021. Comparisons of EU MDR and IVDR regulations. advisera.com (accessed first: 15.03.2023)