T.C.
DOKUZ EYLÜL UNIVERSITY
ENGINEERING FACULTY
ELECTRICAL & ELECTRONICS ENGINEERING DEPARTMENT

MODEL-BASED TESTBENCH CODE GENERATOR FOR EMBEDDED SYSTEM WITH NODAL ANALYSIS

Final Project

by

Enver Kaan ÇABUK

Advisor

Asst.Prof.Dr. Özgür Tamer

June,2022
IZMIR

THESIS EVALUATION FORM

We certify that we have read this thesis and that in our opinion it is fully adequate, in scope and qualify as an undergraduate thesis, based on the result of the oral examination taken place on ___/___/________

Asst.Prof.Dr. Özgür Tamer
(ADVISOR)

Assoc. Prof. Dr. Yavuz ŞENOLL	Dr. BARIŞ BOZKURT
(COMMITTEE MEMBER)	(COMMITTEE MEMBER)

Prof. Dr. Mehmet KUNTALP
(CHAIRPERSON)

ACKNOWLEDGEMENTS

First and foremost, I’d like to express my gratitude to my advisor, ÖzgürTAMER, for accepting my project and assisting me in making the right decisions and conducting the best research by sharing his experiences and ideas about project management and flow with me.

Especially, I’d like to express my gratitude to Cankut Beşer, who has been both a friend and a mentor to me.

I’d like to express my gratitude to my DEU best friends for their unquestioning support and contributions throughout my university career.

Finally, I’d like to express my gratitude to my family for believing in me and always supporting me in achieving my goal.

ABSTRACT

Embedded systems have a massive global impact. Software integrated in automotive, transportation, medical equipment, communication, energy, and a variety of other systems is rapidly triggering innovation. Few test techniques, approaches, tools, and frameworks have been studied by practitioners and researchers in the last several decades to test an embedded system in a cost-effective way.

Thanks to the model-based test method, faults are found by comparing the behavior when the system is under test with the expected behavior when the system is under test.

This project generates code for another device that acts as a testbench to test embedded systems on the target device. The generated testbench software imitates the system to which the controller to be tested will be connected, applies all possible inputs to the controller , and collects the outputs and compares them with the expected outputs. This method can be used if there is a cause-and-effect relationship between behaviors.

The purpose of this project is to take advantage of embedded system tests that often repeat each other and have certain patterns. These repeated patterns are reused. Instead of a manual system, an automatic system was used for the reuse of the patters. This automation reduces the time and effort spent on testing.

Keywords: Automatic Code Generation, Automatic Test Vector Generation, Testbench for Microcontroller, Low-Cost Hardware-In-The-Loop Simulation, Model Based Testing

ÖZET

Gömülü sistemler büyük bir küresel etkiye sahiptir. Otomotiv, ulaşım, tıbbi ekipman, iletişim, enerji ve çeşitli diğer sistemlere entegre edilmiş yazılımlar, inovasyonu hızla tetikliyor. Gömülü sistemi uygun maliyetli bir şekilde test etmek için son birkaç on yılda uygulayıcılar ve araştırmacılar tarafından birçok test tekniği, yaklaşımı ve aracı geşliştirilmiştir.

Model tabanlı test yöntemi sayesinde, sistem test edildiğinde ortaya çıkan davranış ile sistem test edildiğinde beklenen davranış karşılaştırılarak hata bulunur.

Bu proje, hedef cihazdaki gömülü sistemleri test etmek için bir test tezgahı görevi gören başka bir cihaz için kod üreten bir programdır. Üretilen testbench yazılımı, hedef cihazın bağlanacağı sistemi taklit eder, olası tüm girişleri hedef cihaza uygular ve çıktıları toplar ve beklenen çıktılarla karşılaştırır. Davranışlar arasında neden-sonuç ilişkisi varsa bu yöntem kullanılabilir.

Bu projenin amacı, genellikle birbirini tekrar eden ve belirli kalıplara sahip gömülü sistem testlerinden yararlanmaktır. Bu tekrarlanan desenler yeniden kullanılır. Kalıpların yeniden kullanımı için manuel sistem yerine otomatik sistem kullanılmıştır. Bu otomasyon, test için harcanan zamanı ve eforu azaltır.

Anahtar Kelimeler: Otomatik Kod Üretimi, Otomatik Test Vektörü Üretimi, Mikrodenetleyici için Test tezgahı, Düşük Maliyetli Hardware-In-The-Loop Simulation, Model Tabanlı Test

Contents

List of Tables

List of Figures

1. INTRODUCTION

Consumer electronics have become an inextricable aspect of daily life. In today's world, we have witnessed that every product we used before is put into a controller to make it more useful and smarter, such as smart screwdrivers, dishwashers, etc. Designs become more complex under strict design constraints as designers try to add additional functionality to these products. The demand and market competition for electronic products create the need for designers to reduce design and testing time. There are some constraints that the controller in modern electronic products must meet. These are mainly performance, cost, size, and power consumption. Products that meet these constraints stand out in a competitive marketplace. It is neither cost-effective nor practical to design a complex system such as an Application Specific Integrated Circuit (ASIC) or Field-Programmable Gate Array (FPGA). Because ASICs do not enable design modifications at the end of the design cycle, ASICs have limited design flexibility. And FPGA boards are more expensive than other options.

Microcontrollers are general purpose computing systems. They gain purpose according to the loaded program. The microcontroller in a phone and dishwasher may be the same, but they have different functions. These functions depend on the loaded code. Therefore, microcontrollers are widely used in modern electronics. They are cheap, flexible, and practical. Specialized microcontrollers are available for low energy consumption.

Functional testing in FPGA and ASIC design is facilitated by simulation programs. For these hardware, test benches are produced in simulation and the results are followed. Even these test benches are automatically generated. However, studies to facilitate and automate microcontroller functional testing give limited results. Each microcontroller company has its own language and each produced device has different features. For these reasons, the test circle of embedded systems developed with a microcontroller is getting longer. The testing process can take between 60% and 70% of the overall design effort [16]. This rate may vary depending on the testing method used. For this reason, automatic function testing is a feature desired by designers.

1.1 Embedded Systems

An embedded system can be defined as a computer system . This system has a processor, memory, and input/output peripheral devices. Input and output signals can be analog or digital. This system performs one or more dedicated or specialized functions. It is a part of larger mechanical and electronic system. Embedded systems are generally controlled by one or more microcontrollers or digital signal processors [15].

A microcontroller unit(MCU) is developed using a central processing unit (CPU) , volatile and data nonvolatile and several types of peripherals. All of this components are embedded into a single semiconductor integrated circuit (IC) chip and are connected to each other by a data bus. They are usually used to perform specific tasks in a particular application.

- Central Processing Unit

  Central Processing Unit or CPU work as the brain of the Microcontroller. According to the sequence of instructions supplied by the programmer, the CPU manages data flow, performs arithmetic operations, and supply control signals to the rest of the microcontroller.

- Memory

  Non-volatile(Flash) memory contains a list of machine language instructions that contain what operations the CPU should perform in order. Instructions are retained even if the microcontroller is not powered. Volatile memory stores data with the operation of the microcontroller. These are the data of the operations performed. If the microcontroller is not powered, these data are deleted.

- Peripherals

  Peripherals are hardware interface modules that connect the microcontroller to external systems and there are many types. These are data converters, clock generation, timing, analog signal processing, input/output, serial communication.

The design of an embedded system requires the parallel development of both software and hardware [5]. As a result, embedded system hardware and software designers collaborate to develop embedded systems from the ground up.

Embedded Systems Design Cycle

1. Determining overall requirements

   In this stage, designers addresses the question like product needs to work as itended, size, weight, and cost limits and the specific hardware the product will use.

2. Designing the System Architecture

   In this stage, designer address the question like selection of power supplier to the system, connection to the ,Internet, and operating system needs to be embedded?

3. Selection of Operating System(OS)

   In this stage, designer select the real time or non-real time operating system.

4. Choosing the Controller and Peripherals

   In this stage,designers select a microcontroller that is estimated to more than meet the needs. Also required peripherals selected like displays, sensors, etc.

5. Choosing the Development Platform

   In this stage, designers select the hardware platform, the development tools

6. Code the Applications, Optimize, Debug, and Test

   In this stage, a programming language suitable for the selected microcontroller, designer tools and selected development tools is selected. Code is generated using this language. During or after generation , the code is optimized, debugged and tested. The coding of an embedded system can be done by using the following programming languages.

   Assembly language, C ,Object oriented languages like C++, Python, etc.

7. Verify the software on the host system

   Designers use a simulator to verify the software code will work within the system.

8. Verify the software on the target system

   Designers load the program to the target devices using programmer devices. And they will verify that it works on the target system.

9. Ongoing maintenance and updates

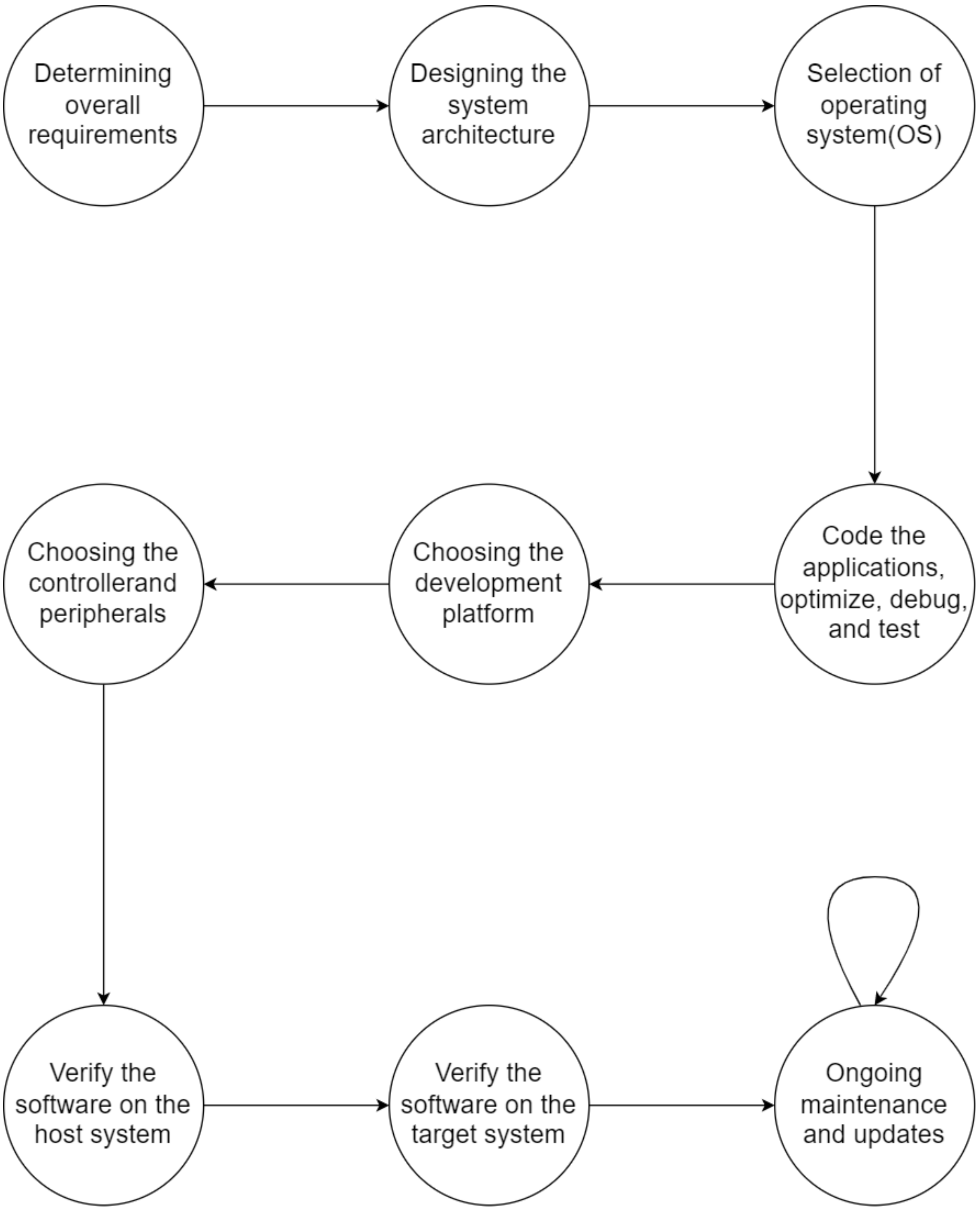

Determining overall requirements
Designing the system architecture
Selection of operating system(OS)
Code the applications, optimize, debug, and test
Choosing the development platform
Choosing the controllerand peripherals
Verify the software on the host system
Verify the software on the target system
Ongoing maintenance and updates

Figure 1.1: Embedded Systems Design Cycle.

Embedded Systems Testing Method

1. Black-box testing
2. White-box testing
    (a) Unit test
    (b) Subsystem test
    (c) System Integration Test
    (d) Validation test
3. Direct testing of systems
    (a) Model-based test
    (b) Test-Bench test
    (c) Load test
    (d) Fault Test
    (e) Spec-based test

1.2 Automatic Code Generation

A code generator is basically a program that generates a code for the existence of another program. Code generation is a highly beneficial technology that can have a stunning impact on software engineering projects.

The code generators are divided into two high-level categories [10]: active and passive. In the passive model, the code generator generates a set of codes, this code is editable by the user. But if the generator is run after the edit, the modification is erased. Active generators allow to run generator multiple times over same output and it also maintain responsibility for the code long term. When modifications to the code are required, team members may enter parameters into the generator and rerun the generator to update the code.

Code generation has two approaches. These are visitor-based and template-based. The visitor-based approach is the generation of code while iterating through the textual representation of the model, with the guidance of some visitor mechanism [7].

In the template-based approach, the metacodes in the target text are selected according to the desired rule, iterated, and the code is generated by derivation [7].

Type of Code Generation

1. Code munger

   The meaning of ‘Munging’ is the process of changing data to another format . The code munger examines and analyzes the input file before creating one or more output files. [4]

2. Inline-code expander

   An inline code expander reads the input code and scans the pre-marked lines [15]. It generates the output code by adding the production code to the pre-marked lines of the input code.

3. Mixed-code generator

   The mixed code generator is the combination of an in-line expander and a code munger. In the mixed code generator, output of the generator can also be re-used as an input. The pre-marked lines on the mixed code generator indicate the area. And if the generator is run again, only this related area will change. This area is generated by a specially formatted comments line. [4]

4. Partial-class generator

   The definition and template files are the partial-class generator input. Additions are made to the template according to the definition file, and output is produced. The output changes depending on the variables in the definition file. The functionality is given to the output code manually. [4]

5. Tier or Lier generator

   A tier or layer generator is similar to a partial class generator. The biggest difference is that it is generated in the code that provides the functionality. [4]

Techniques of Code Generation

1. Templates and Filtering Techniques

   The code is generated by applying templates to a subset of the textual source model , typically after filtering the source model. Textual templates are used to represent generalized source code fragments. Figure 1.2 shows the generation of code using templates and the filtering technique.

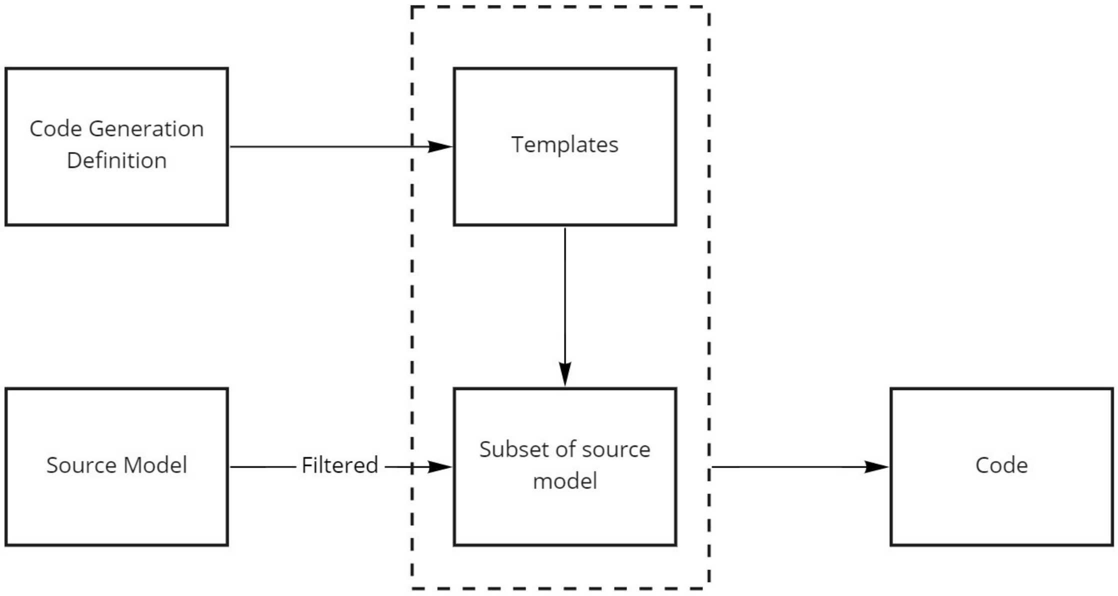


Figure 1.2: Templates and Filtering Code Generation Technique.

2. Templates and Meta-model Technique

   A metamodel is the model of a model [12]. Templates and Metamodel is a derivative of templates and filtering pattern. A meta-model is generated from the specification. Templates are defined from these metamodels. The output code is generated based on the metamodel. Templates are used to add the source code. Figure 1.3 shows the generation of code by templates and metamodel technique.

3. API-based generator technique

   API-based code generators typically generate code in a particular programming language. As a result, it gives the user with a unique framework that makes the code generation process more straightforward. Templates are widely used to specify the source code. The API depends on the metadata / syntax of the target language . Figure 1.4 shows the generation of code by API-based generators technique.

4. In-line generation technique

   During pre-complier, the in-line generator technique decides whether to include text fragments in the source code based on conditions. Figure 1.5 shows the generation of code by In-line generation technique.

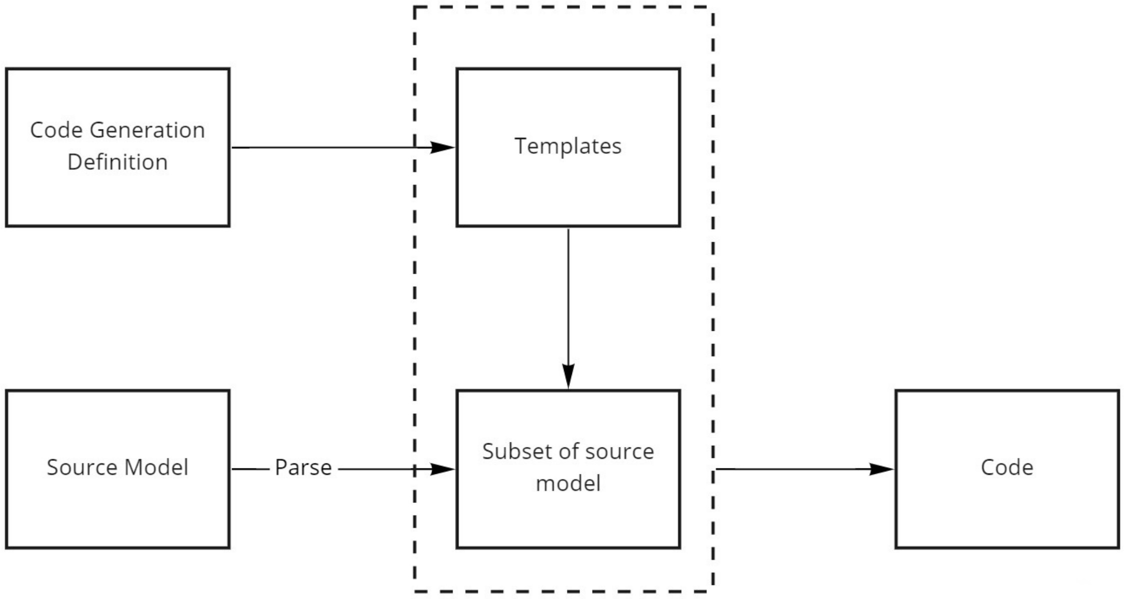


Figure 1.3: Templates and Meta-model Generation Technique.

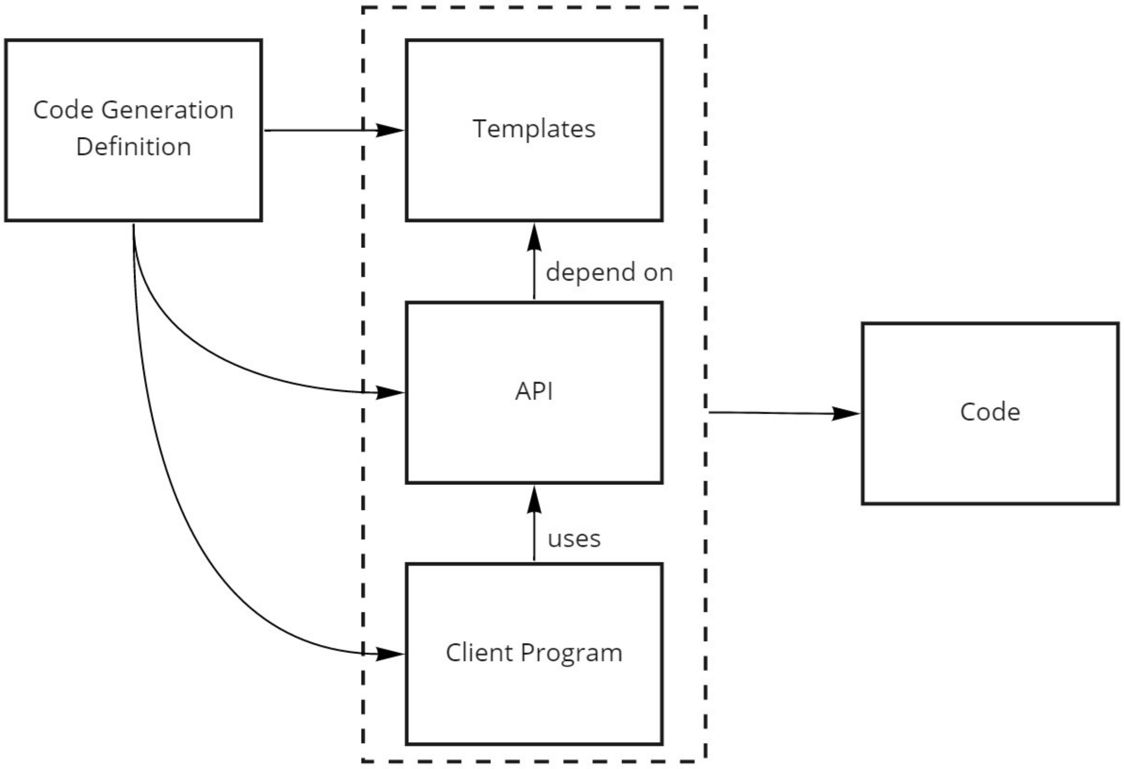


Figure 1.4: API-Based Generation Technique.

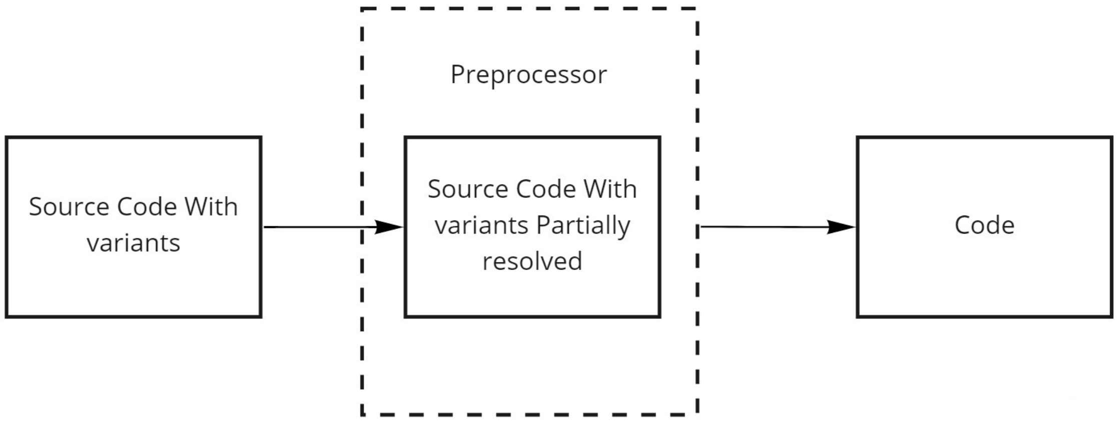

Preprocessor
Source Code With variants
Source Code With variants Partially resolved
Code

Figure 1.5: In-line Generation Technique.

Benefits of Code Generation

1. Quality

   Large amounts of handwritten code tend to have varying quality because developers discover new or convenient ways as they work. When you generate code from templates, you get a consistent code base right away, and when you alter the templates and run the generator, the bug fixes and coding enhancements are implemented uniformly throughout the code base.

2. Consistency

   Since the variable names in the code generated by the code generator are consistent with those in the code generator interface, it is easier to understand and modify the generated code.

3. Reducing Design Cycle

   In projects that already have code generators, there is a huge time difference between hand-coding and automatic generation. Although automatic production cannot provide all the desired features, these features can be added to the generated code thanks to its easy understanding. For large-volume and repetitive projects where the code generator is not ready, designing a code generator may also consume less time.

4. Design decisions that stand out

   High-level business principles are hidden by the intricacies of implementation code. Code generators use short, specific description files from the final code, containing an abstract of the design, to describe the design. Small exceptions are significantly more evident in a five-line definition file than in the 500 lines of implementation code that follow.

1.3 Testbench

A test bench, also known as a testing workbench, is an environment in which soundness, functionality and correctness are tested. It is a term derived from a time when electronics was developed, when an engineer in the laboratory tested the accuracy of the device under testing(DUT) by connecting the electronic circuitry to specific components, signal sources, and measuring instruments.

Testbenches are still used in software and hardware. Testbenches,which are used in hardware , have been transferred to the computer environment as a simulation because they have less effort, cost, and danger.

One of the most important simulation-based testbenches is used in circuits designed and synthesized with hardware description language(HDL). Because these circuits must be tested. Since the synthesis takes place in the computer environment, the testbench must also take place in the computer environment. A testbench should contain an instance of the model, input signals. Inputs and outputs as a result of the simulation are examined, checking whether the behavior of the hardware matches the expected behavior.

Input signals implemented in testbenches are called test vectors or patterns. In most cases, the input test vectors are created by hand. However, as the number of inputs increases in complex and large designs, it takes more time to generate elements of the test vector. The test engineer may bias some test vectors [14]. These test vectors can cause an important case. For these reasons, test vector generators are used. The main idea of the testbench for digital systems is shown in Figure 1.6.

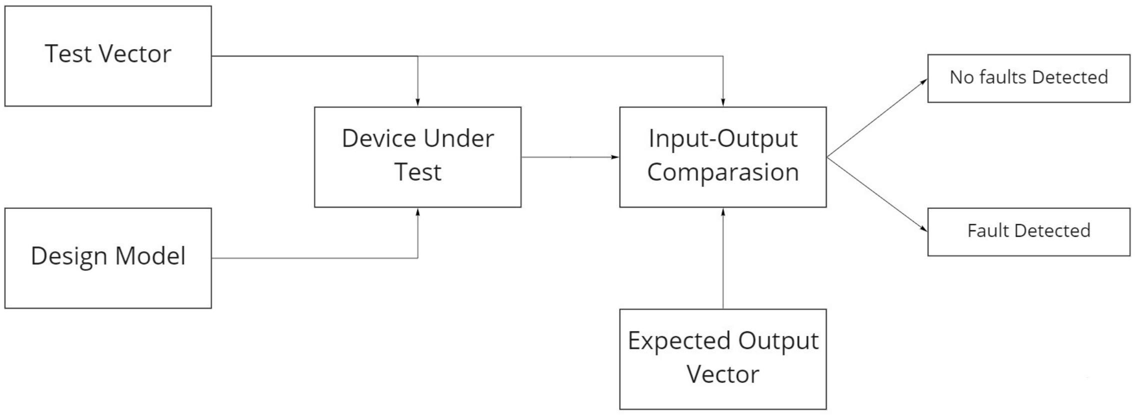


Figure 1.6: Testbench simulation flow.

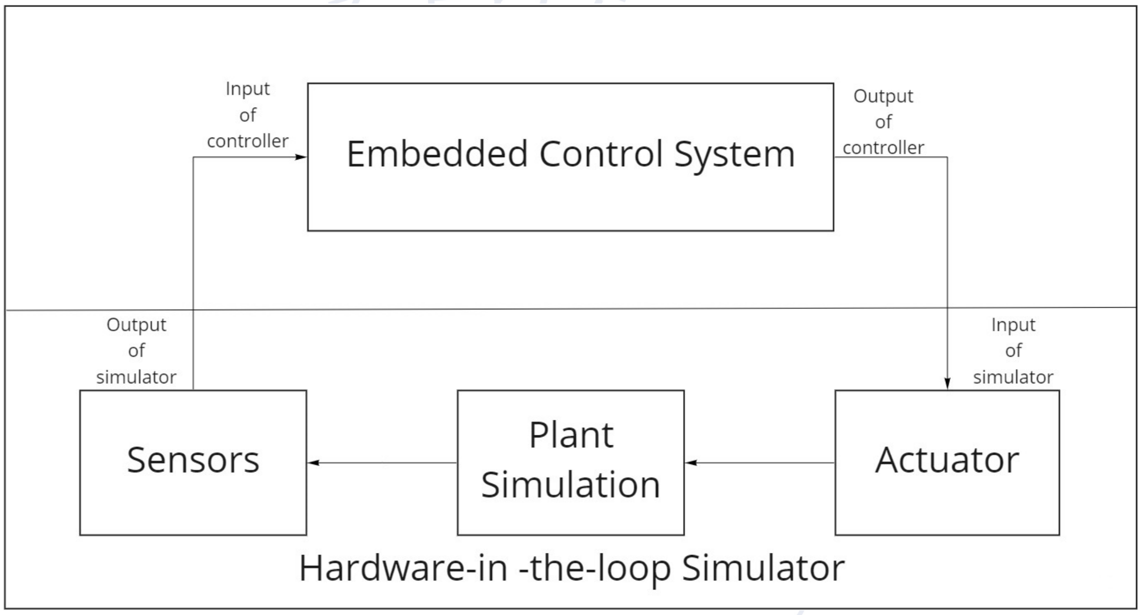


Figure 1.7: Hardware-in-the-loop simulation flow.

1.4 Hardware-in-the-loop simulation

The hardware-in-the-loop (HIL) simulation technique is utilized in the development and testing of complex real-time embedded systems [13]. HIL simulation is realized by connecting the controller to the imitation of the physical system(plant) to which it will actually be connected. This simulated system analyzes the responses of the controller by giving the signals that the actual system will give. In order to electrical emulate the physical system, the characteristics of the sensors, actuators and other peripheral units in the system must be simulated. Using these electrical emulations, an interface is created between the plant simulation and the embedded system under test. The plant simulation determines the values that the sensor will measure, which are then read by the embedded system. The changes in the actuator vary according to the control signals applied by the embedded system. Condition changes in the actuators may cause changes in the values in the plant simulation. Errors and deficiencies are determined by following this cause-effect relationship. The main idea of hardware-in-the-loop for embedded systems are demonstrated in Figure 1.7.

1.5 Scope of Project

Every product designed today is subjected to many tests during the development process or when it is completed. The purpose of testing during development is to make development faster and with less affordable . The purpose of the tests after the development is completed is to provide a faultless product to the customer and to ensure customer satisfaction and safety.

There are several ways to perform these tests in embedded systems. Software and hardware can be tested separately or both can be tested together when the software is on the hardware. A tool produced by Rapita systems [1]performs the software test automatically. The hardware-in-the-loop simulations device and software are the same. It is a test instrument close to autonomy with the help of its tools. However, it has not been widely used in hardware-in-the-loop simulations due to the expensiveness of the components that connect the controller to the computer environment. The test tool produced by Rapita systems, on the other hand, cannot provide sufficient integrity for embedded systems because it only tests the software.

The main purpose of the project is to reduce the testing and development process in embedded systems. While fulfilling this purpose, the tested embedded system is connected to another microcontroller that acts as a testbench. This microcontroller imitates the system and applies the possible test vectors that the system will give to the control unit of the embedded system to be tested. It determines faults by checking whether the feedback from the controller of the embedded system matches the expected feedback. The test vector is automatically generated using given user parameters. This project can also be defined as an automatic low-cost hardware in-the-loop simulator [6] [8]. The difference from the hardware-in-the-loop simulation is that the effects of the connected actuators and sensors' behavior on the physical system are not examined. In this project, the system is tested by examining changes in electrical state in the external peripherals. In the hardware-in-the-loop simulation, the values applied to the sensors are such as temperature and pressure, while the values applied to the sensors in this project are taken from the user in volts. In the hardware-in-the-loop simulation, the values measured from the actuators are measured in torque,rpm units, while the values measured in this project are measured as the duty cycle and frequency of the PWM signal.

2. METHODOLOGY

The program to be developed was created by using the information obtained previous work . In this section, the sub-sections of the program will be examined separately and the program will be produced using the method, technique, program language, algorithm and hardware that are determined to be suitable for these sections. The project can be divided into the following main headings;

- Embedded system testing with microcontroller acting as a testbench
- Test Vector Generation
- Automatic Code Generation
- User Interface

2.1 Embedded system testing with microcontroller acting as a testbench

This section is about how to test embedded systems (target devices) with a microcontroller (test device) that acts as a testbench. It includes configurations, application code, and case scenarios required for the test device to test correctly. The STM32F407-DISC development board was used as the test device and C was chosen as the programming language. The configurations will be made according to this card and the programming language. The most common applications in embedded systems include following function:GPIO, ADC, PWM, communication interface , interrupt, timer, memory , ALU. These functions are generally related to each other in accordance with the cause and effect relationship. A state change in one of these functions may cause a state change in other functions. By using this feature, functions in the embedded system, depending on the cause and effect relationship, can be tested.

In this project, peripheral functions are called nodes. Non-peripheral functions are called essential nodes. Connections between functions are called node connections. When testing one of the functions, we generate this function as a node. This node contains the properties of the function, its parameters and information about which state it will be triggered by. The example of the generated node is shown in Figure 2.1.

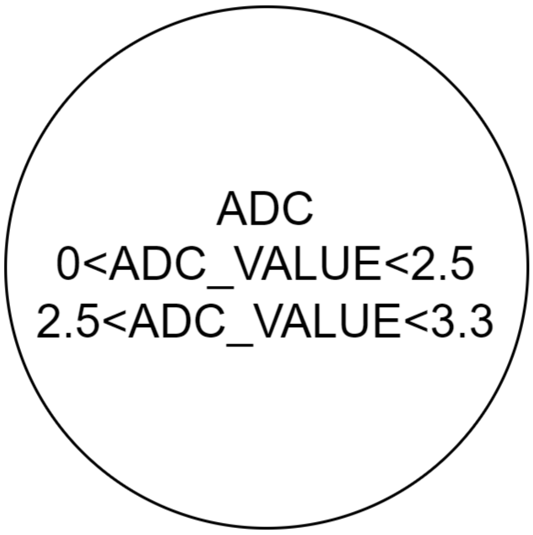


Figure 2.1: Generating and defining node.

After the cause node is generated, the result node is generated. This node contains the properties of the function, its parameters, and information about the state change on which the state change at the cause node creates. After the generation of the cause and effect node, these nodes are connected. These nodes and their connections are on the target device. They will be generated automatically according to the data provided by the user. An example of the connection node is shown in Figure 2.2.

After these generations, two nodes are generated for the tester devives. One of these nodes is the node that triggers the cause node, that is, applies the signal that causes the state change. The other node is a node that checks the status of the result node, that is, examines which signal it applies to the component to which it will be connected. These two nodes generated for the tester devices are also connected to each other. Because in order for the test to take place, these two nodes must have a cause-effect relationship with each other. With the generation of these nodes and connections, the testing of the desired functions can be performed. The example of the test node diagram is in Figure 2.3.

In some cases, the two nodes may not be directly connected to each other in a cause-and-effect relationship. That is, there may be an indirect cause-and-effect relationship. This is because it has uncontrollable nodes. These uncontrollable nodes are nonenvironmental nodes that we have named before. These functions cannot be controlled as there is no link connecting them to the outside. Therefore, these essential nodes must be connected to at least two controllable nodes. And there must be an indirect cause-and-effect relationship between these two controllable

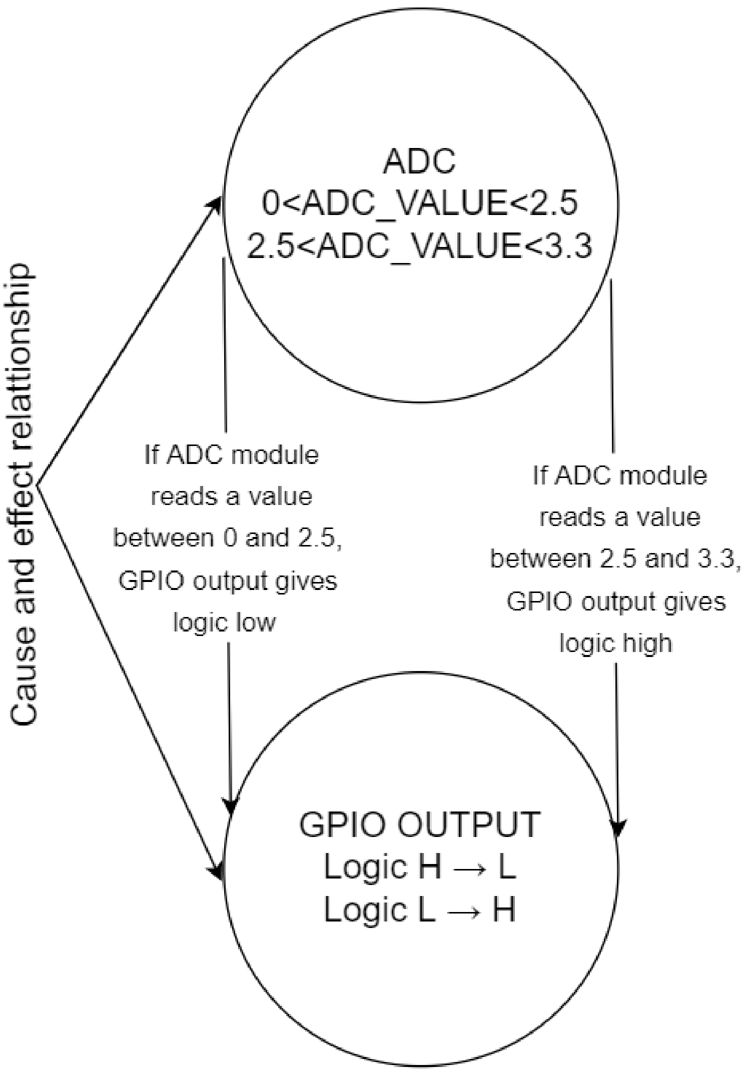
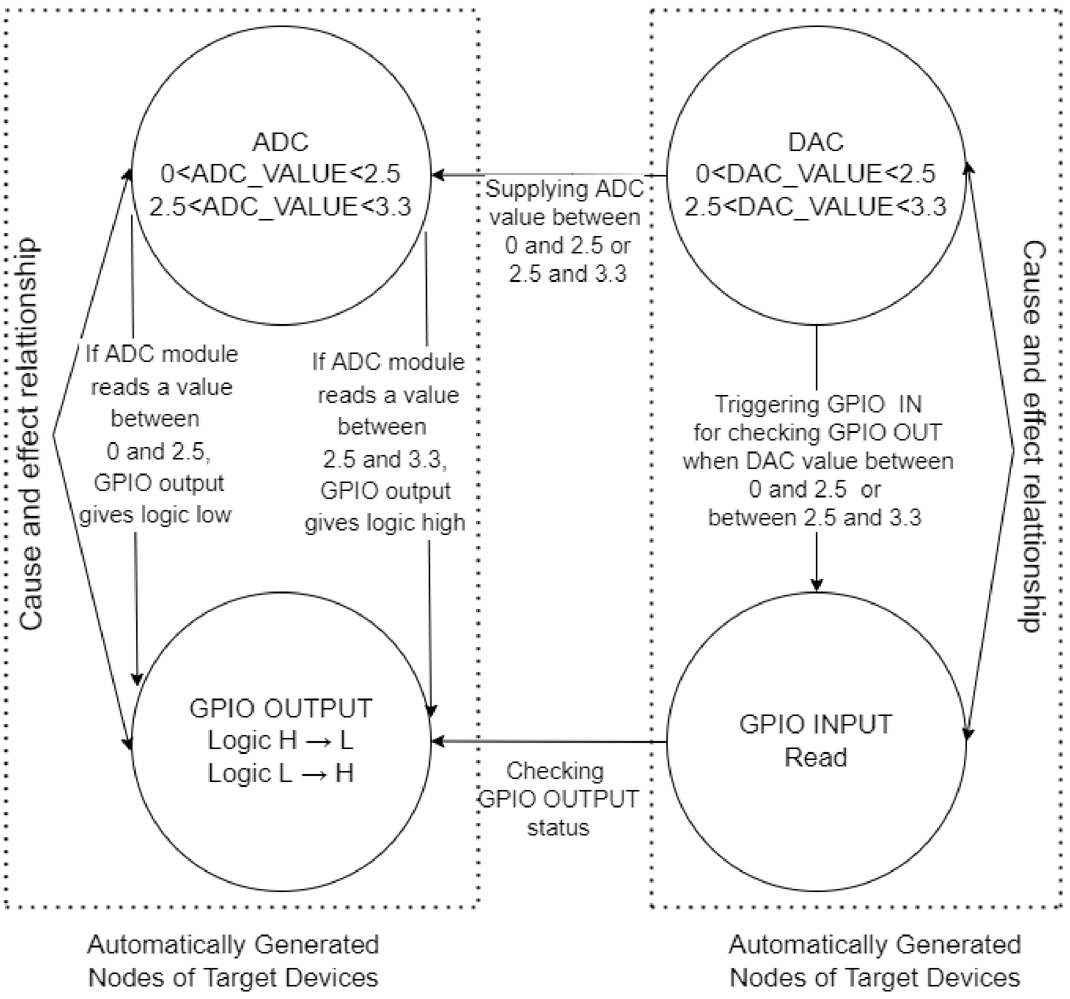

ADC
0<ADC_VALUE<2.5
2.5<ADC_VALUE<3.3
Cause and effect relationship
If ADC module reads a value between 0 and 2.5, GPIO output gives logic low
If ADC module reads a value between 2.5 and 3.3, GPIO output gives logic high
GPIO OUTPUT
Logic H → L
Logic L → H

Figure 2.2: Connecting nodes.



ADC
0<ADC_VALUE<2.5
2.5<ADC_VALUE<3.3
DAC
0<DAC_VALUE<2.5
2.5<DAC_VALUE<3.3
Supplying ADC value between 0 and 2.5 or 2.5 and 3.3
Cause and effect relattionship
If ADC module reads a value between 0 and 2.5, GPIO output gives logic low
If ADC module reads a value between 2.5 and 3.3, GPIO output gives logic high
Triggering GPIO IN for checking GPIO OUT when DAC value between 0 and 2.5 or between 2.5 and 3.3
Cause and effect relattionship
GPIO OUTPUT
Logic H → L
Logic L → H
GPIO INPUT
Read
Checking GPIO OUTPUT status
Automatically Generated Nodes of Target Devices
Automatically Generated Nodes of Target Devices

Figure 2.3: Test node diagram.

nodes. The example of the test node diagram with indirect connection is shown in Figure 2.4.

In some special cases, output nodes can be tested regardless of cause and effect relationship. In this case, the output node may be connected to one or more essential nodes. As an example of this situation, the LED that turns on when the system is running or the same string is displayed continuously. The example of the test node diagram without cause node is in Figure 2.5. According to this technique, we can list which nodes we will generate in the tester device to test the nodes on the target device as follows. This table fill according to most used function in embedded system.

Table 2.1: Nodes selection for the tester device based on the nodes of the target device.

Node of Target Device	Node of Tester Device
GPIO IN	GPIO OUT
GPIO OUT	GPIO IN
ADC	DAC
PWM OUT	PWM IN (Timer)
UART	UART

In some applications, the GPIO OUT node can be controlled with a PWM input node. For example toggling led. In addition, it should be said that the test method of interrupt and GPIO input nodes is the same.

2.2 Test vector generation

This section is about how to generate test vector using user provide data.

Test vectors are used in all test methods similar to Testbench. Test vectors are vectors that are applied to the inputs and contain the expected outputs as a result of the application. By applying the test vector to the testbenches, the behavior of the system during the test is examined. The test outputs and expected outputs are compared. Errors are found according to the result of the comparison.

Within this project, test vectors will be generated by analysis of user-defined parameters and graphical models. Graphical models will express node connections. Parameters will contain information about the properties of nodes and state changes. A model coverage-based approach will be used to increase the probability of test vectors finding all possible errors. Because of their similarities, some code coverage approaches will also be used.

To generate a test vector based on model coverage, at least one value from each user-provided interpolation interval that causes a state change must be added to the test vector [2]. Interpolation

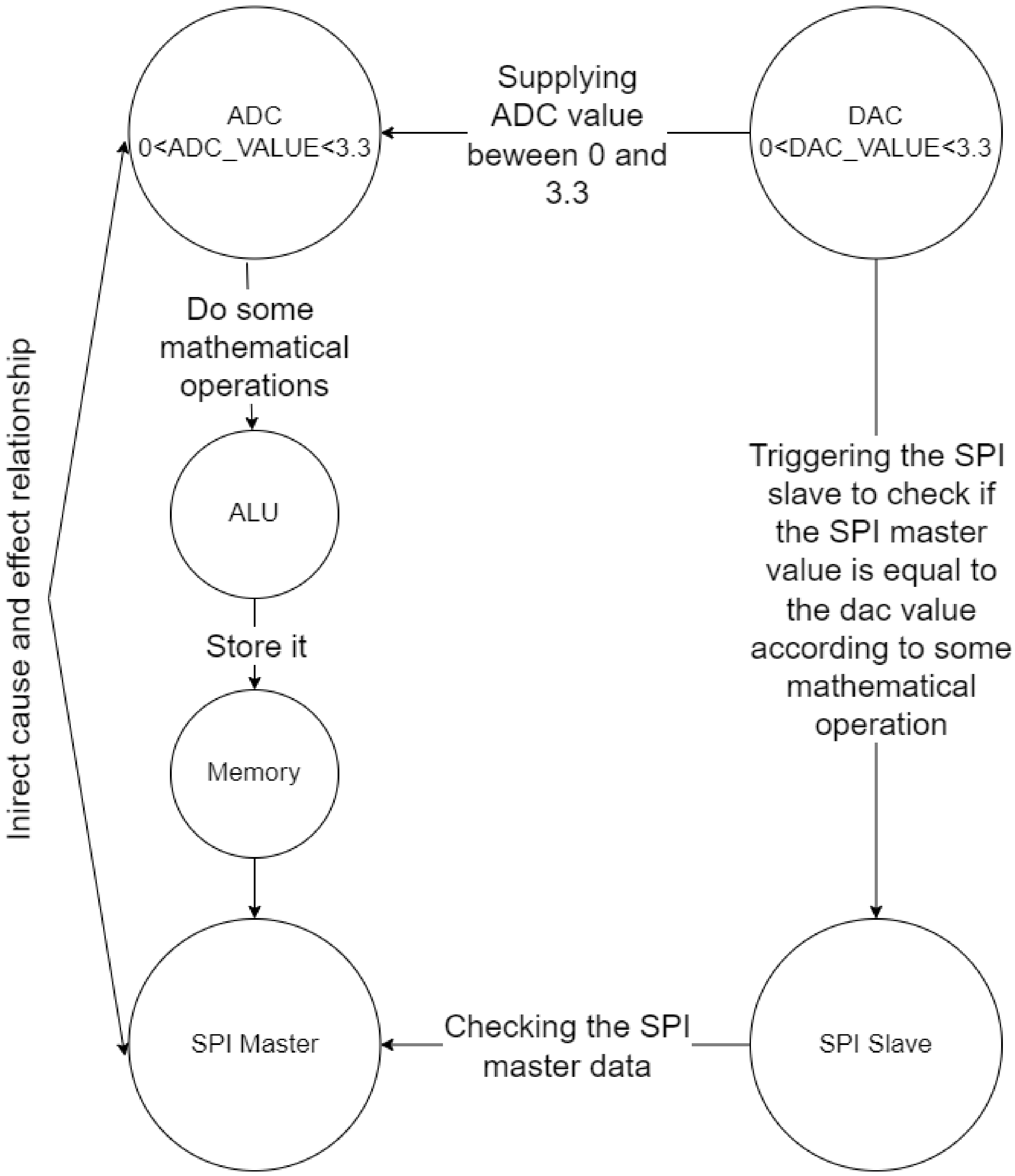

ADC
0<ADC_VALUE<3.3
Supplying ADC value beween 0 and 3.3
DAC
0<DAC_VALUE<3.3
Do some mathematical operations
ALU
Store it
Memory
Inirect cause and effect relationship
Triggering the SPI slave to check if the SPI master value is equal to the dac value according to some mathematical operation
SPI Master
Checking the SPI master data
SPI Slave

Figure 2.4: Test node diagram with indirect connection.

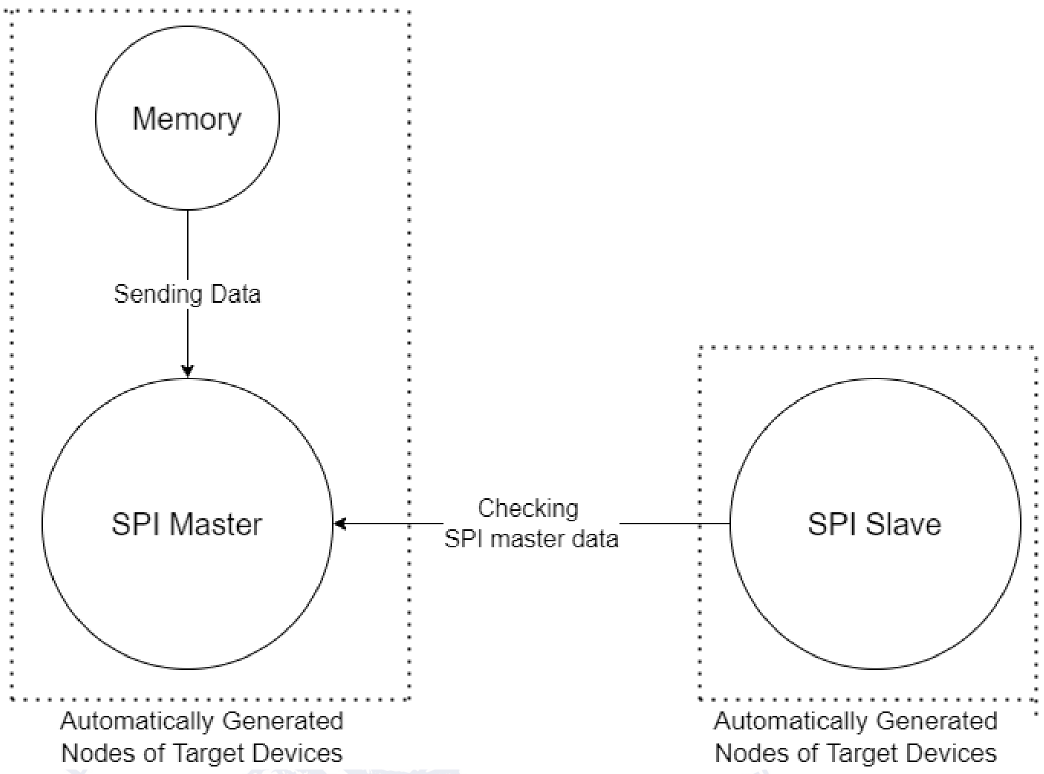

Memory
Sending Data
SPI Master
Checking SPI master data
SPI Slave
Automatically Generated Nodes of Target Devices
Automatically Generated Nodes of Target Devices

Figure 2.5: Test node diagram withoutcause node.

ranges are selection stataments in graphical models. These selection statement include equal, greater, less than, not equal, and combinations thereof.

In the Figure 2.7 there is an embedded system flowchart for fire alarm. ADC is connected to the sensor and GPIO_IN is connected to the button. GPIO_OUT1 is connected to the fire bell, UART is connected to the displays and GPIO_OUT2 is connected to the electrical and natural gas control system. If the value read by the sensor is above 1.5 or the button is pressed, the fire alarm bell will ring and the warning message will be printed on the screens. If the value read by the sensor is above 2.5, the electricity and natural gas of the building will be cut off.

In the following part, test vectors will be generating with model coverage . This generation will be done with an approach in code covarage [11].

According to flowchart, this system has two inputs and three outputs. Inputs are ADC and GPIO_IN. Outputs are UART, GPIO_OUT1 and GPIO_OUT2. For generating test vector of this embedded system, selection statements should be examined. It has three selection statements according to flow chart. And these two selection related to ADC. One of them is related to GPIO_IN.

In the first ADC related selection statement , if the ADC value is greater than 1.5, it will

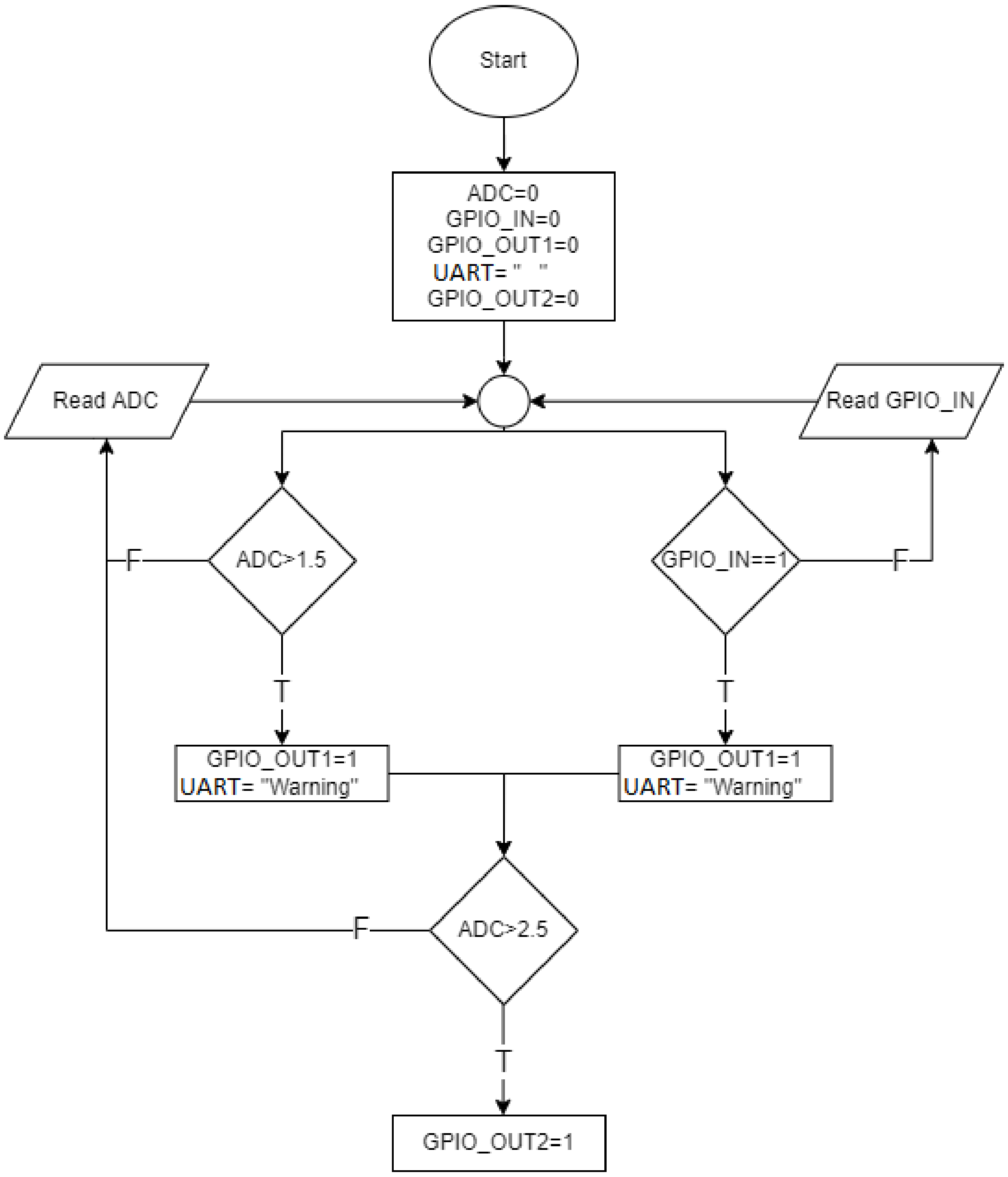


Figure 2.6: Flowchart of embedded system for fire alarm.

change GPIO_OUT1 state from logic 0 to logic 1 and it will changes the string of UART. And in the second ADC related selection statement , if the ADC value is greater than 2.5, it will change GPIO_OUT2 state from logic 0 to logic 1. Accordingly, the interpolation points of the ADC are 1.5 and 2.5. GPIO_IN related selection statement , if the GPIO_IN state is equal to logic high , it will change GPIO_OUT1 state and it will changes the string of UART. Accordingly, the interpolation points of the GPIO_IN are logic 0 and logic 1. After this determination, generated test vector are in the table below. By inserting the generated vectors into different combinations, other vectors can be derived. However, the vectors generated in this way already get 100% coverage except ADC. Because according to the test results, it was observed that the DAC could not give the desired voltages completely.

If the test is time-dependent, that is, a program that measures and compares the delays, if it is desired to produce a program, delay information is added to the test vectors.

Table 2.2: Generated test vector.

ADC	GPIO_IN	GPIO_OUT1	GPIO_OUT2	UART
$0 < ADC < 1.5$	0	0	0	” ”
$1.5 < ADC < 2.5$	0	1	0	”Warning”
$ADC > 2.5$	0	1	1	”Warning”
$0 < ADC < 1.5$	1	1	0	”Warning”

The test vectors applied to the common nodes mentioned in the first part can have the following states.

Output element of test vector

- For GPIO OUT

  GPIO output can have two state. These are logic high or logic low.

  GPIO_IN==1 or GPIO_IN==0

- For DAC

  In our previous work, it was mentioned that there can be test vectors as much as the resolution of the DAC. However, according to the test results, it was observed that the microcontroller board used would not allow this. As can be seen from the graph below, the expected voltage and the measured voltage are different. To solve this problem, the midpoint between the upper limit and the lower limit of the DAC will supply.

- For UART Transmit

  UART transmit data can be expressed in two ways in this program. These are string and hexadecimal. ”Temperature” is defined as 0x55, 0x9a, 0x37 via the interface. If XXX is written anywhere, the program detects that this element will be created randomly and requests the ranges of this element through the interface. For example, ”Temperature=XXX” , 0x55, 0x9a,XXX , 0x37.According to the test results, there was no loss in UART communication.

Inpuıt element of test vector

- For GPIO IN

  GPIO input can have two state. These are logic high or logic low.

  GPIO_OUT==1 or GPIO_OUT==0

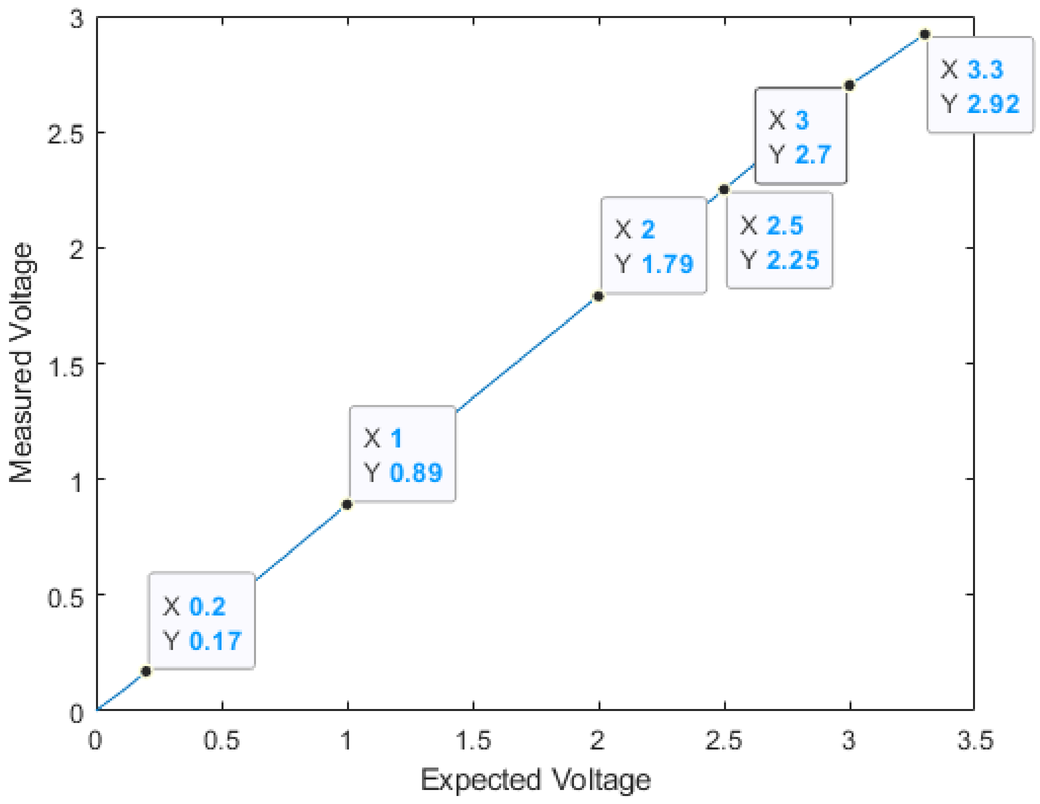


Figure 2.7: Test Result of DAC.

- For PWM IN

  The test vectors of PWM IN are the expected frequency and duty-cycle received through the interface. If they are connected with the DAC node, the DAC value and the duty-cycle can be directly proportional to each other. PWM IN node has been tested with STM32 and it has been determined that the calculated duty cycle is correct and the frequency is calculated with a small tolerance value. In the table below, the expected frequency vs. the measured frequency is illustrated. The maximum tolerance value is +6%.

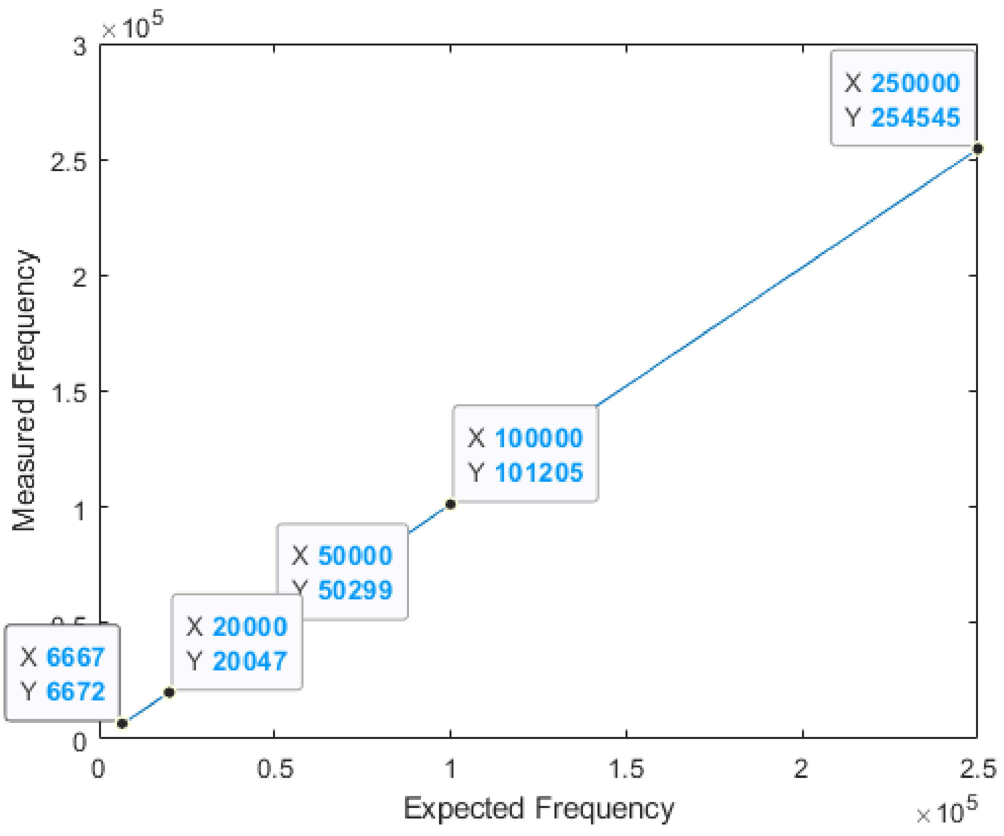


Figure 2.8: Frequency Test Result of PWM.

- For UART Receive

  UART received data also can be expressed in two ways in this program. These are same with UART Transmit. Definition of string, hexadecimal and random variable are same. The UART receive node can be connected with the DAC node. And the received values can contain the DAC value. In these cases, XXX is used and their range should be set to 0.

2.3 Automatic Code generation

This section is about how to generate code automatically.

In this project, Python language was chosen for the automatic code generator. It was chosen because it is an easy, practical and high-level language. C language was chosen as the microcontroller coding language for the following reasons.

- Processor independent
- Portability
- Performance
- Bit manipulation
- Memory management

The most important of these is memory management. Since the generated code as a result of automatic code generation is not optimized, memory restrictions are important in this project.

Embedded system software can be divided into 3 main topics. These are source, drivers and middlewares. In this project, source files are generated. The well-known drivers used will be added to the file. Source has fragments such as configuration, interrupt handler, startup, application code.

In configuration, the device is configured.

In the interrupt handler, the interrupt vectors are set.

Startup is the hardware setup before the microcontroller starts operating.

Application codes are code blocks prepared to fulfill a specific purpose.

These fragments can be written to a single file, or they can be written to different files and called each other.

Embedded system software consists of many code blocks. These blocks include header, typedef, define, macro, variable, function prototype, function, main function, main loop. With the combination of these code blocks, files containing driver, middleware, configuration, interrupt handler, initialization can be generated or added to the main code. A template like the one below can be generated for the main file from these blocks.

```
#include "microcontroller.h"
/*Header area*/
```

```
/* Private typedef */
/* Private define */
/* Private macro */
/* Private variables */
/* Function prototype area*/
/* Function without protype*/
/* Private user code */
int main(void)
{
/* Inside of the main function*/
/* Function calling area*/
while(1)
{
/*Inside of the infinite loop in main function*/
}
/* After the infinite loop*/
}
/* After the main funciton*/
/* Function area*/
/* Timer config  area*/
/* Handler area*/
```

In order to generate a code suitable for the test method described in section 1, the following must be done:

Nodes should be produced in accordance with the definitions given by the user. These nodes have a pre-prepared pattern and the parameters provided by the user and devices datasheet are assigned. These variables are the parameters related to the node. (node type, pin number etc.)

Some codes in this code are written to the configuration file, some to the header file, some to the main code file or other files.

After the cause and effect nodes are created with the same method, the codes of the connection between these two nodes, namely the application codes, are produced. Application codes are possible patterns that have already been created. Usually, these patterns include giving the input signals sequentially and comparing the output signals with the expected signals.

Variables in the test vector are assigned to the variables in the pattern.

```
Testvector=[["DAC1_1" ,"3.3","2.5","1","1"],["DAC1_2" ,
 "2.5","1.5","1","0"], ["DAC1_3" ,"1.5","0","0","0"]]

with open("main.c", "r") as in_file:
buf = in_file.readlines()
with open("main.c", "w") as out_file:
for line in buf:
if line == "/* Private variables */\n":
line = i=0 ; \n"
if line == "/*Inside of the infinite loop
in main function*/\n":
line = line +f" DAC_Value=(Testvector[i][1]+
Testvector[i][2])/2 ; \n"

line = line +f" i=i+1;  \n"
line = line +f" if (Testvector[0][1]>DAC_Value>
 Testvector[0][2]){\n"

line = line +f" if (GPIO_IN2==Testvector[0][5] && GPIO_IN1
==Testvector[0][4]){\n"

line = line +f" printf( Part Testvector[0][0] sucseed);}\n"
line = line +f" else {\n"
line = line +f" printf( Part X failed);}} \n"
line = line +f" else if v(Testvector[1][1]>DAC_Value>
 Testvector[1][2]) {\n"

line = line +f" if (GPIO_IN2==Testvector[1][5] && GPIO_IN1==
Testvector[1][4]){\n"

line = line +f" printf( Part Testvector[1][0] sucseed);}\n"
line = line +f" else {\n"
line = line +f" printf( Part X failed);}} \n"
line = line +f" else if v(Testvector[2][1]>DAC_Value>
 Testvector[2][2]) {\n"

line = line +f" if (GPIO_IN2==Testvector[2][5] && GPIO_IN1==
Testvector[2][4]){\n"

line = line +f" printf( Part X sucseed);}\n"
line = line +f" else {\n"
line = line +f" printf( PartTestvector[2][0] failed);}} \n"
out_file.write(line)
```

In the code above, an application code is integrated into the main code. As in the example above, we can add anything we want to the desired code block of the desired file. And we can add the user defined parameter and automatic generate node parameter(configuration). If ADC, UART, PWM nodes are to be found in the tester device, besides the default libraries, the hal libraries of these nodes should be added to the STM32F4xx_HAL_Driver folder.

2.4 User Interfaces

There are multiple styles of interaction [9]. They are as follows:

- Command line
- Menu selection
- Form fill-in
- Direct manipulation
- Anthropomorphic
- Graphical

The user interface in this project will use the form fill-in style. However, this interface is dynamic. It will change depending on user input.

Parameters such as test vectors, node parameters, node connections, number depending on the node type will be defined in the user interface. In other words, a model will be created. This model will be added to the code generator with the API.

The following questions will be answered in the form.

•How many GPIO IN pins does the system have?

•How many GPIO OUT pins does the system have?

•How many DAC pins does the system have?

•How many UART interfaces does the system have?

•How many ALU operation does the system have?

After this data is entered, the ”Generate Node” button is pressed.After clicking, 4 widgets appear. Parameters and test vectors of DAC, UART, PWM and ALU are entered into these widgets.

The following questions will be answered in the widgets.

•How many ranges exist for DAC?

•1. upper bound of range for ADC?

•1. lower bound of range for ADC?

•Data Transmission Mode of UART?

→Only Ttransmit

→Only Receive

→Transmit and Receive?

•UART baudrate?

•UARTtransfer continuity?

→Transfer Continous

→Transfer One Time

•Transmitted data with UART?

•Received data with UART?

•How many random variable exist at transmitted data?

•How many random variable exist at received data?

•What is the upper limit of 1. random variable in transmitted data?

•What is the lower limit of 1. random variable in transmitted data?

•PWM frequency?

•PWM dutycycle?

•ALU operation?

These parameters will be entered by clicking the push buttons on the widgets.At the end, ready-to-connect nodes are created with the ”Define Parameter” button.A list of connectable nodes appears under the ”Define parameter” button. In the widget below it, these connectable nodes are combined according to the cause-effect relationship between them.

ADC1-2.5,1.5→GPIO_OUT1

ADC1-2.5,1.5→UART_Receive

GPIO_IN1→GPIO_OUT1

GPIO_IN1→UART_Receive

ADC1-3,2.5→GPIO_OUT2

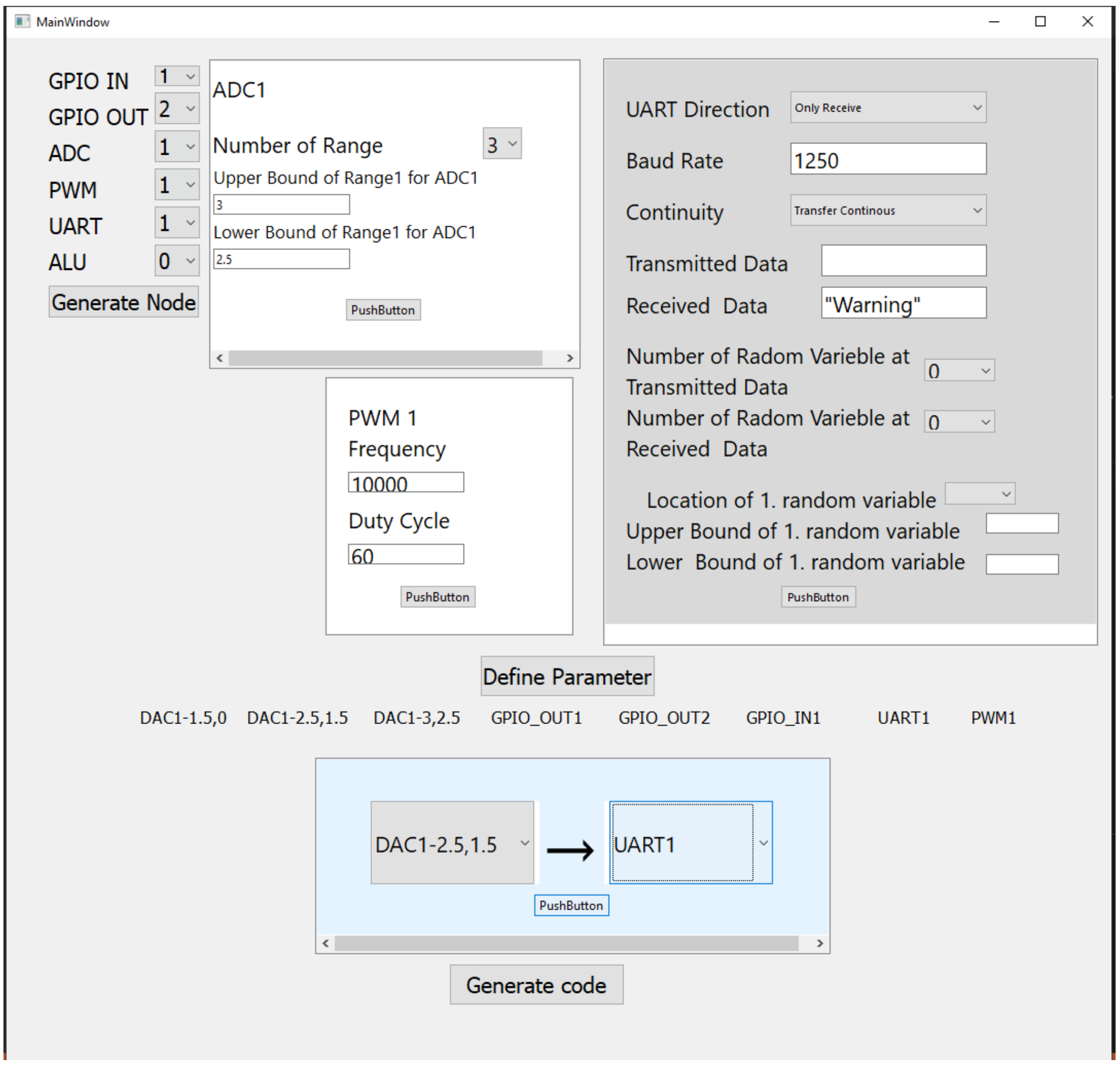
Finally, the code is generated by clicking ”Generate code”. The resulting code files are transferred to the desired Integrated Development Environment (IDE) and the code is compiled and loaded into the microcontroller. After the test is over, a feedback should be created to get the test success and errors. In order to receive this feedback, it was requested to use the ”Virtual Com Port” feature with STM32. It has been proven by terminal emulators that STM32 sends messages to the computer, and a code created with python that reads serial communication data with USB was run, but the data could not be read. Different computers, compliers, then this is assumed to originate from python or kernel. Serial communication is provided in the C # program, but the integration of these two languages has been abandoned assuming that it may cause other bugs. In this case, the feedback will be carried out by lighting the led. The executed code and the received code are attached below.

```
import serial
import time
SerialObj = serial.Serial("COM7")
SerialObj.baudrate = 9600
SerialObj.bytesize = 8
SerialObj.parity   ='N'
SerialObj.stopbits = 1

time.sleep(3)
SerialObj.open()
line=SerialObj.readline()
print(line)
SerialObj.close()
```

```
SerialException:
Cannot configure port, something went wrong.
Original message: OSError(22, 'Parameter Error.', None, 87)
```

The user interface design is in figure 2.9.



GPIO IN 1
GPIO OUT 2
ADC 1
PWM 1
UART 1
ALU 0

ADC1
Number of Range 3
Upper Bound of Range1 for ADC1
3
Lower Bound of Range1 for ADC1
2.5

PWM 1
Frequency
10000
Duty Cycle
60

UART Direction Only Receive
Baud Rate 1250
Continuity Transfer Continous
Transmitted Data
Received Data "Warning"
Number of Radom Varieble at Transmitted Data 0
Number of Radom Varieble at Received Data 0
Location of 1. random variable
Upper Bound of 1. random variable
Lower Bound of 1. random variable

DAC1-1.5,0 DAC1-2.5,1.5 DAC1-3,2.5 GPIO_OUT1 GPIO_OUT2 GPIO_IN1 UART1 PWM1

DAC1-2.5,1.5 → UART1

Figure 2.9: User İnterface.

3. TEST RESULT

In previous section , we defined peripheral function as a node and non-peripheral functions as essential nodes. These nodes were DAC, UART, PWM, GPIO IN and GPIO OUT.And these esential nodes are ALU and memory. In the progress of the project, the memory essential node was merged with other nodes. Certain configurations must be made to generate these nodes.Below are the optimized configurations for the selected development board.

Configuration of Nodes

1. GPIO IN

   (a) main.c

   ```
   __HAL_RCC_GPIOB_CLK_ENABLE();

     /*Configure GPIO pins :  PB13 */
     GPIO_InitStruct.Pin = GPIO_PIN_13;
     GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
     GPIO_InitStruct.Pull = GPIO_NOPULL;
     HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
   ```

   (b) stm32f4xx_hal_conf.h

   ```
   #define HAL_GPIO_MODULE_ENABLED

   #ifdef HAL_GPIO_MODULE_ENABLED
     #include "stm32f4xx_hal_gpio.h"
   #endif /* HAL_GPIO_MODULE_ENABLED */
   ```

The configuration in the stm32f4xx-hal-conf.h file covers all GPIOs. This configuration is not added again for another defined GPIO. here the stm32f4xx_hal_gpio.h library is called. The line of code added to the main.c file specifies the peripheral's fan in, GPIO mode and whether the specified pin has a pull up/down resistance. At the same time, it enables the clock of the relevant port. In the main.c configuration above, the mode is digital input, Port B Pin 13 is selected as fan in. No resistor is connected to the relevant pin.

2. GPIO OUT

   (a) main.c

```
__HAL_RCC_GPIOB_CLK_ENABLE();

HAL_GPIO_WritePin(GPIOB, GPIO_PIN_13, GPIO_PIN_RESET);

  /*Configure GPIO pin : PB13 */
  GPIO_InitStruct.Pin = GPIO_PIN_13;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
```

   (b) stm32f4xx_hal_conf.h

```
#define HAL_GPIO_MODULE_ENABLED

#ifdef HAL_GPIO_MODULE_ENABLED
  #include "stm32f4xx_hal_gpio.h"
#endif /* HAL_GPIO_MODULE_ENABLED */
```

   The configuration in the stm32f4xx_hal_conf.h file covers all GPIOs. This configuration is not added again for another defined GPIO. Here the stm32f4xx_hal_gpio.h library is called. The line of code added to the main.c file specifies the peripheral's fan out, GPIO mode, switching speed and whether the specified pin has a pull up/down resistance . At the same time, it enables the clock of the relevant port and resets the corresponding pin. It is enough to enable the clock once for a port. In the main.c configuration above, the mode is digital output, Port B Pin 13 is selected as fan out. No resistor is connected to the relevant pin.And switching speed selected as low.

3. DAC

   (a) main.c

```
DAC_HandleTypeDef hdac;

  DAC_ChannelConfTypeDef sConfig = {0};
  hdac.Instance = DAC;
  if (HAL_DAC_Init(&hdac) != HAL_OK)
  {
    Error_Handler();
  }

  sConfig.DAC_Trigger = DAC_TRIGGER_NONE;
  sConfig.DAC_OutputBuffer = DAC_OUTPUTBUFFER_ENABLE;
if (HAL_DAC_ConfigChannel
(&hdac, &sConfig, DAC_CHANNEL_1) != HAL_OK)
  {
    Error_Handler();
  }
```

   (b) stm32f4xx_hal_conf.h

```
#define HAL_DAC_MODULE_ENABLED

#ifdef HAL_DAC_MODULE_ENABLED
  #include "stm32f4xx_hal_dac.h"
#endif /* HAL_DAC_MODULE_ENABLED */
```

(c) stm32f4xx_hal_msp.c

```
void HAL_DAC_MspInit(DAC_HandleTypeDef* hdac)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  if(hdac->Instance==DAC)
  {
    __HAL_RCC_DAC_CLK_ENABLE();
    __HAL_RCC_GPIOA_CLK_ENABLE();
    GPIO_InitStruct.Pin = GPIO_PIN_4;
    GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
  }
}

void HAL_DAC_MspDeInit(DAC_HandleTypeDef* hdac)
{
  if(hdac->Instance==DAC)
  {
    __HAL_RCC_DAC_CLK_DISABLE();
    HAL_GPIO_DeInit(GPIOA, GPIO_PIN_4);
  }
}
```

The configuration in the stm32f4xx_hal_conf.h file covers all DACs. This configuration is not added again for another defined DAC. Here the stm32f4xx_hal_dac.h library is called. The line of code added to the main.c file define the DAC handle sructure, select the channel , specifies external trigger and enable output buffer for selected channel. In the stm32f4xx_hal_msp.c file has two function.These functions initialize and deinitialize the DAC. During initialization, it defines fanout, GPIO mode and whether the specified pin has pull-up/down resistor. And the corresponding clocks are enabled.During deinitialization, it changes peripheral registers state to their default reset values.

4. PWM IN

(a) main.c

```
TIM_HandleTypeDef htim2;

static void MX_TIM2_Init(void)
{
 TIM_ClockConfigTypeDef sClockSourceConfig = {0};
TIM_SlaveConfigTypeDef sSlaveConfig = {0};
TIM_IC_InitTypeDef sConfigIC = {0};
   TIM_MasterConfigTypeDef sMasterConfig = {0};
htim2.Instance = TIM2;
htim2.Init.Prescaler = 9;
 htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim2.Init.Period = 4967299245;
htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
   htim2.Init.AutoReloadPreload =
 TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
  {
      Error_Handler();
}
  sClockSourceConfig.ClockSource =
 TIM_CLOCKSOURCE_INTERNAL;
   if (HAL_TIM_ConfigClockSource
(&htim2, &sClockSourceConfig) != HAL_OK)
    {
    Error_Handler();
}
    if (HAL_TIM_IC_Init(&htim2) != HAL_OK)
  {
      Error_Handler();
}
   sSlaveConfig.SlaveMode = TIM_SLAVEMODE_RESET;
     sSlaveConfig.InputTrigger = TIM_TS_TI1FP1;
   sSlaveConfig.TriggerPolarity =
 TIM_INPUTCHANNELPOLARITY_RISING;
    sSlaveConfig.TriggerPrescaler = TIM_ICPSC_DIV1;
 sSlaveConfig.TriggerFilter = 0;
   if (HAL_TIM_SlaveConfigSynchro
(&htim2, &sSlaveConfig) != HAL_OK)
       {
    Error_Handler();
     }
     sConfigIC.ICPolarity =
 TIM_INPUTCHANNELPOLARITY_RISING;
       sConfigIC.ICSelection =
 TIM_ICSELECTION_DIRECTTI;
   sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
     sConfigIC.ICFilter = 0;
  if (HAL_TIM_IC_ConfigChannel
(&htim2, &sConfigIC, TIM_CHANNEL_1) != HAL_OK)
       {
    Error_Handler();
     }
      sConfigIC.ICPolarity =
 TIM_INPUTCHANNELPOLARITY_FALLING;
         sConfigIC.ICSelection =
 TIM_ICSELECTION_INDIRECTTI;
    if (HAL_TIM_IC_ConfigChanne
l(&htim2, &sConfigIC, TIM_CHANNEL_2) != HAL_OK)
       {
    Error_Handler();
     }
        sMasterConfig.MasterOutputTrigger =
TIM_TRGO_RESET;
           sMasterConfig.MasterSlaveMode =
TIM_MASTERSLAVEMODE_DISABLE;

      if (HAL_TIMEx_MasterConfigSynchronization
(&htim2, &sMasterConfig) != HAL_OK)
       {
    Error_Handler();
     }
     }
```

(b) stm32f4xx_hal_conf.h

```
#define HAL_TIM_MODULE_ENABLED

#ifdef HAL_TIM_MODULE_ENABLED
 #include "stm32f4xx_hal_tim.h"
#endif /* HAL_TIM_MODULE_ENABLED */
```

(c) stm32f4xx_hal_msp.c

```
void HAL_TIM_Base_MspInit
(TIM_HandleTypeDef* htim_base)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  if(htim_base->Instance==TIM2)
  {
    __HAL_RCC_TIM2_CLK_ENABLE();
    __HAL_RCC_GPIOA_CLK_ENABLE();
    GPIO_InitStruct.Pin = GPIO_PIN_15;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    GPIO_InitStruct.Alternate = GPIO_AF1_TIM2;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

```
        HAL_NVIC_SetPriority(TIM2_IRQn, 0, 0);
        HAL_NVIC_EnableIRQ(TIM2_IRQn);
          }
          }
    void HAL_TIM_Base_MspDeInit
    (TIM_HandleTypeDef* htim_base)

    {
         if(htim_base->Instance==TIM2)
        {
      __HAL_RCC_TIM2_CLK_DISABLE();
      /**TIM2 GPIO Configuration
          */
          HAL_GPIO_DeInit(GPIOA, GPIO_PIN_15);
        HAL_NVIC_DisableIRQ(TIM2_IRQn);
             }     }
```

(d) stm32f4xx_hal_it.h

```
    void TIM2_IRQHandler(void);
```

(e) stm32f4xx_hal_it.c

```
    extern TIM_HandleTypeDef htim2;

     void TIM2_IRQHandler(void)
      {
             HAL_TIM_IRQHandler(&htim2);
            }
```

The configuration in the stm32f4xx_hal_conf.h file covers all timer based function. This configuration is not added again for another defined timer based function. Here the stm32f4xx_hal_tim.h library is called. In the code added to the main.c file, the clock settings (period, prescaler, divider, etc.) and other configurations of the timer that controls the PWM are performed. In the stm32f4xx_hal_msp.c file has two function.These functions initialize and deinitialize the timer. During initialization, it defines fan-in, GPIO mode, peripheral connected to the selected pin and whether the specified pin has pull-up/down resistor. And the corresponding clocks are enabled.At the same time, it defines interrupt, determines its priority and enables it. During deinitialization, it changes peripheral registers state to their default reset values. And disable the interrupt.On the other hand, stm32f4xx_hal_it.c writes a function that calls interrupt handlers. In order to be able to call this function anywhere, it defines the prototype of this function in the stm32f4xx_hal_it.h file.

5. UART

   (a) main.c

```
 UART_HandleTypeDef huart4;

static void MX_UART4_UART_Init(void)
 {
   huart4.Instance = UART4;
   huart4.Init.BaudRate = 1250;
   huart4.Init.WordLength = UART_WORDLENGTH_8B;
   huart4.Init.StopBits = UART_STOPBITS_1;
   huart4.Init.Parity = UART_PARITY_NONE;
     huart4.Init.Mode = UART_MODE_TX_RX;
     huart4.Init.HwFlowCtl = UART_HWCONTROL_NONE;
     huart4.Init.OverSampling = UART_OVERSAMPLING_16;
     if (HAL_UART_Init(&huart4) != HAL_OK)
       {
     Error_Handler();
    }}
```

   (b) stm32f4xx_hal_conf.h

```
#define HAL_UART_MODULE_ENABLED

#ifdef HAL_UART_MODULE_ENABLED
 #include "stm32f4xx_hal_uart.h"
```

```
#endif /* HAL_UART_MODULE_ENABLED */
```

(c) stm32f4xx_hal_msp.c

```
void HAL_UART_MspInit(UART_HandleTypeDef* huart){
GPIO_InitTypeDef GPIO_InitStruct = {0};
if(huart->Instance==UART4)
{
__HAL_RCC_UART4_CLK_ENABLE();
__HAL_RCC_GPIOA_CLK_ENABLE();
__HAL_RCC_GPIOC_CLK_ENABLE();
GPIO_InitStruct.Pin = GPIO_PIN_1;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
GPIO_InitStruct.Alternate = GPIO_AF8_UART4;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
GPIO_InitStruct.Pin = GPIO_PIN_10;
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
GPIO_InitStruct.Alternate = GPIO_AF8_UART4;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}
}
 void HAL_UART_MspDeInit(UART_HandleTypeDef* huart){
  if(huart->Instance==UART4)
  {
  __HAL_RCC_UART4_CLK_DISABLE();
  HAL_GPIO_DeInit(GPIOA, GPIO_PIN_1);
  HAL_GPIO_DeInit(GPIOC, GPIO_PIN_10);
  }
  }
```

The configuration in the stm32f4xx_hal_conf.h file covers all UART. This configuration is not added again for another defined UART. Here the stm32f4xx_hal_uart.h library is called.

The line of code added to the main.c file define the UART handle sructure, select the uart , specifies worldlenght, parity, stopbits and set teh baudrate. In the stm32f4xx_hal_msp.c file has two function.These functions initialize and deinitialize the UART. During initialization, it defines fanout, GPIO mode, switching frequency, peripheral connected to the selected pin , and whether the specified pin has pull-up/down resistor. And the corresponding clocks are enabled.During deinitialization, it changes peripheral registers state to their default reset values.

6. Clock

   (a) main.c

```
void SystemClock_Config(void){
RCC_OscInitTypeDef RCC_OscInitStruct = {0};
RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
  __HAL_RCC_PWR_CLK_ENABLE();
__HAL_PWR_VOLTAGESCALING_CONFIG
(PWR_REGULATOR_VOLTAGE_SCALE1);
RCC_OscInitStruct.OscillatorType =
 RCC_OSCILLATORTYPE_HSE;
 RCC_OscInitStruct.HSEState = RCC_HSE_ON;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
 RCC_OscInitStruct.PLL.PLLM = 8;
 RCC_OscInitStruct.PLL.PLLN = 336;
 RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
RCC_OscInitStruct.PLL.PLLQ = 7;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK){
  Error_Handler();
  }
  RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|
RCC_CLOCKTYPE_SYSCLK|RCC_CLOCKTYPE_PCLK1
|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource =
 RCC_SYSCLKSOURCE_PLLCLK;
 RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
 RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;
 if (HAL_RCC_ClockConfig
(&RCC_ClkInitStruct, FLASH_LATENCY_5) != HAL_OK){
  Error_Handler();
  }
  }
```

(b) stm32f4xx_hal_conf.h

```
#define HAL_RCC_MODULE_ENABLED

#ifdef HAL_RCC_MODULE_ENABLED
  #include "stm32f4xx_hal_rcc.h"
#endif /* HAL_RCC_MODULE_ENABLED */
```

The configuration in the stm32f4xx_hal_conf.h file call stm32f4xx_hal_rcc.h library . The line of code added to the main.c file defines clock type, divider of clock and similar parameter to generate desired frequency for clcok.

Some parameters of the above configurations can be reproduced by changing and functions can be created for the configurations .

In order for any microcontroller to work, application codes are required apart from the configurations.In this project, application codes read the changing state of GPIO IN, changing the state of GPIO OUT, changing the voltage level given by DAC, data transfer by UART or PWM IN frequency and duty cycle measurement can be given as examples.At the same time, node connections are application codes. However, they will be examined separately in this section.

Application Code

1. GPIO IN

   (a) main.c

   ```
   HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1)
   ```

   The line of code added to the main.c file chekcs the logic state of related pin. It returns the logic state of pin as a binary number.

2. GPIO OUT

   (a) main.c

   ```
   HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12, GPIO_PIN_SET);
   ```

   This code added to the main.c file changes the logic state of the relevant pin. If it is GPIO_PIN_SET, the logic state returns to 1. If it is GPIO_PIN_RESET, the logic state returns to 0.

3. DAC

   (a) main.c

   ```
    HAL_Delay(1000);
    dac\_value=931;
    HAL_DAC_SetValue
   (&hdac, DAC_CHANNEL_1, DAC_ALIGN_12B_R, dac\_value);
    HAL_Delay(1);



    HAL_Delay(1000);
   dac\_value=3101;
    HAL_DAC_SetValue
   (&hdac, DAC_CHANNEL_1, DAC_ALIGN_12B_R, dac\_value);
    HAL_Delay(1);
   ```

This code added to main.c changes the voltage supplied by the DAC depending on the variable i. According to the STM32F407 datasheet the supplied voltage is 0V if dac_value is equal to zero, and the supplied voltage is 3.3V if dac_value is equal to 4095.

4. PWM IN

   (a) main.c

```
/* Private user code */
__IO uint32_t ICValue1=0;
__IO uint32_t Frequency1=0;
__IO uint32_t Duty1=0;
__IO uint32_t i=1;
int frequency1comp;
int duty1comp;
//Function without protype
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
if(htim->Channel==HAL_TIM_ACTIVE_CHANNEL_1)
{
 if(i==2)
{
ICValue1=HAL_TIM_ReadCapturedValue(htim,TIM_CHANNEL_1);
if(ICValue1!=0)
{
Duty1=(HAL_TIM_ReadCapturedValue
(htim,TIM_CHANNEL_2)*100)/ICValue1;
Frequency1=(HAL_RCC_GetHCLKFreq())/2/ICValue1;
}
}
}
}
```

•

```
.
.
//Inside of the infinite loop in main function
ICValue1=0;
  Duty1=0;
  Frequency1=0;
 HAL_TIM_IC_Start_IT(&htim2,TIM_CHANNEL_1);
 HAL_TIM_IC_Start_IT(&htim2,TIM_CHANNEL_2);
Delay_ms(200);
 HAL_TIM_IC_Stop_IT(&htim2,TIM_CHANNEL_1);
 HAL_TIM_IC_Stop_IT(&htim2,TIM_CHANNEL_2);
 i=2;
Delay_ms(200);
```

This code added to main.c measure the frequency and utycycle of PWM usign input capture mode of STM32 timer. Channel 1 triggered by rising edge of PWM and it counts the time until next rising edge and process repeat.The value counted by the timer is the period of the PWM. The formula for frequency is f = 1 / T.Frequency is calculated using this formula.Channel 2 triggered falling edgeof PWM and it counts the time between rising edge and falling edge.The duty cycle can be found by dividing this time calculated via Channel 2 by the period.

5. UART

(a) main.c

```
/* Private user code */
 uint8_t pData0[]={0x7e, 0x00, 0x55, 0x00, 0x22};
  uint8_t pData1[3]="0";
  uint8_t pData1comp[3]={0x88, 0x00, 0x11};
 char sData2[31];
  uint8_t pData2[31];



int x00=rand()\%12+44;
  pData0[1] = (uint8_t) x00 ;
  int x01=dac\_value;
  pDatal0[3] = (uint8_t) x01 ;
  HAL_UART_Transmit
(&huart4, pData0, sizeof(pData0), 1000);
  HAL_Delay(250);
  HAL_UART_Receive
(&huart4, pData1, sizeof(pData1), 1000);
  HAL_Delay(250);

int x10=rand()\%126+25;
   sprintf(sData2, "Temperature \%d",x11);
  for(int k=0; k==strlen(sData2);k++)
  {
  pData2[k]= (uint8_t)sData2[k];
  }
  HAL_UART_Transmit(&huart5, pData2, sizeof(pData2), 1000);
  HAL_Delay(250);
```

This application code allows data transfer in hexedcimal or string format with uart. It also allows adding random values in the desired range into the sent data. Random value sending feature is arranged in accordance with some sensor data transfer protocol.

It has been mentioned in the previous study that nodes with cause-effect relationships are required for this test device to work. This cause-effect relationship is achieved by connecting nodes to each other. These connections are made using the if selection structure. After the cause node is triggered, the behavior of the result node is examined and compared with the expected behavior. If these two behaviors match, there is no error.These connections may vary according to the structure of the nodes. GPIO IN, PWM In can only be used as a result node. DAC and GPIO Out are only used as cause nodes. UART can be used as both cause and effect nodes.

Node Connection

1. GPIO IN

(a) main.c

```
//Begin of GPIO_1
if(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1))
// or if(!HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1))
{
errorgpioin1=1;
}
else
{
errorgpioin1=0;
}
```

GPIO IN nodes can only be used as result nodes and the connection is created by enclosing the application node.. It is added to the end of the relevant reason node, so that it is not affected by the delays of other nodes. The location of the relevant reason node is found by the comment lines added to the application code of the reason node.

2. GPIO OUT

   (a) main.c

```
HAL_GPIO_WritePin(GPIOD, GPIO_PIN_12, GPIO_PIN_SET);
// End of GPIO_OUT1
```

GPIO OUT is used as a reason node. No code is required to connect this reason node to a result node. Just a comment line indicating the end of the reason node is sufficient. It is valid for other reason nodes as well.

3. DAC

   (a) main.c

```
 //Begin of Range1 midpoint for DAC1
 HAL_Delay(1000);
dac\_value=931;
```

```
 HAL_DAC_SetValue
(&hdac, DAC_CHANNEL_1, DAC_ALIGN_12B_R, dac\_value);
 HAL_Delay(1);
 //End of Range1 midpoint for DAC1

 //Begin of Range2 midpoint for DAC1
 HAL_Delay(1000);
dac\_value=3101;

 HAL_DAC_SetValue
(&hdac, DAC_CHANNEL_1, DAC_ALIGN_12B_R, dac\_value);
 HAL_Delay(1);
 //End of Range2 midpoint for DAC1
```

The DAC is only used as the reason node. Unlike GPIO_OUT, the comment line specifying where the result node should be inserted is positioned at the end of the DAC's ranges, not at the end of the DAC application code.

4. PWM IN

   (a) main.c

```
//Inside of the infinite loop in main function
  //Begin of the PWM1
ICValue1=0;
  Duty1=0;
  Frequency1=0;
 HAL_TIM_IC_Start_IT(&htim2,TIM_CHANNEL_1);
 HAL_TIM_IC_Start_IT(&htim2,TIM_CHANNEL_2);
Delay_ms(200);
 HAL_TIM_IC_Stop_IT(&htim2,TIM_CHANNEL_1);
 HAL_TIM_IC_Stop_IT(&htim2,TIM_CHANNEL_2);
 i=2;
Delay_ms(200);
if (duty1!= duty1comp)
```

```
{
errorpwm1duty=1;
}
else
{
errorpwm1duty=0;
}
if (frequency1>(frequency1comp*(1+ tolarence))
 || frequency1<(frequency1comp*(1- tolarence)))
{
errorpwm1freq=1;
)
else
{
errorpwm1freq=0;
}

//End of the PWM1
```

PWM is always used as a result node. It is appended to the end of the cause node or its ranges.

5. UART

   (a) main.c

```
  HAL_UART_Transmit
(&huart4, pData0, sizeof(pData0), 1000);
  HAL_Delay(250);
//End of Tranmision1

//Begin of Receive1
  HAL_UART_Receive
(&huart4, pData1, sizeof(pData1), 1000);
  HAL_Delay(250);
```

```
if (pData1!= pData1comp)
{
errorUart1\_Receive=1;
}
else
{
errorUart1\_Receive=0;
}
```

UART can be used as both cause and effect node. If it transmits, it is the reason node, if it receives, it is the result node. If it does both, it is both the cause and effect node. However, by separating these application codes, it would be a better method to generate code as if it were two nodes. Since transmit is a reason node, a comment line specifying the location is sufficient for the connection to add a result node at the end. Since receive is a result node, it should be added under a reason node.

Codes with if-else statements added to the result nodes hide the error occurrences. Thanks to these codes, the connections that cause the error will be defined.

In this part, the accuracy of the program has been tested. A case scenario including all possible parameters has been created to perform the test.The scenario prepared for this program is as follows. The scenario prepared for this program is as follows. It is a simple vehicle control system. Data will be received from the vehicle's speed sensor and the Uart and this data will be printed on the screen with the Uart. The sensor that reads the engine coolant temperature will be read by ADC and the temperature LED will light up at certain intervals. With the components connected to the doors, it will be determined whether the doors are closed and if they are not, the light will turn on. By pressing a button, the air conditioner will operate. This air conditioner motor will be driven by pwm.The sensor in the accelerator pedal allows the motor of the gas pump to be driven by pwm.

The code has been prepared for this scenario. To test this code, the code of the tester device is generated with the program. While generating this code, the noed connections are entered as follows.

UART4Transmit → UART5Receive

DAC1 → GPIO_IN 1

GPIO_OUT1 → GPIO_IN2

GPIO_OUT2 → PWM1

DAC2 → PWM2

After connecting the hardware cards and running the codes, it was observed that no errors were found in the error variables. And the leds indicating a successful test turned on. Apart from this test, many unit tests have been made. And no errors have been found. A situation that is not a problem in theory but causing a problem in practice has been detected. If there is more than one GPIO_IN/GPIO_OUT connection, it may be prone to errors while connecting the hardware.

4. PROGRESS UP TO DATE

In order to increase the efficiency and accuracy of the project, it started with the definition and limitation of the project. At this stage, not how the project will be done, but what to do in the project is considered. Similar projects, theses and articles have been researched.

According to the researches, it has been determined that this project has similarities with HIL simulators and HDL testbenches. Therefore, articles on embedded systems, automatic code generation, testbench(test vector), hardware-in-the-loop simulation have been emphasized for the project.

There are 3 titles required to complete the project. These are the generation of patterns that can fulfill its function for the tester device, the printing of these patterns on the pre-prepared template during automatic code generation, the production of the test vector with the parameters provided by the user and its integration into the pattern. These 3 titles are the most time-consuming in the work package.

The spring semester gantt chart is shown in Table 3.1 ,Table 3.2 and Table 3.3. In these tables, the work packages that have been successfully completed so far, the inputs and outputs of these work packages are explained.

Table 4.1: Tester Code Development

Tester Code Development
Responsible: Enver Kaan ÇABUK Time: 8 week
Input(s): •Configuration of Nodes •Application Code for Nodes •Connection of Nodes
Output(s): •Template •Patterns
Status: Completed

Table 4.2: User Interface and Test Vectors

User Interface and Test Vectors
Responsible: Enver Kaan ÇABUK Time: 4 week
Input(s): •Design of User Interface and Integration to Main Code •Design of Code Generator and Integration to Main code
Output(s): •User Interface •Test Vector Generator
Status: Completed

Table 4.3: Code Generator

Code Generator	
Responsible: Enver Kaan ÇABUK	Time: 12 week
Input(s): •Regeneration and Adjustment of Patterns depending on Parameters •Automatic Placement of Patterns in the Template	
Output(s): •Generator Program	
Status: Completed	

5. WORK PLAN

This chapter includes Gantt Chart of the project in Table 4.1 . The work packages are as follows:

•Literature Survey

•Design procedure

•Preparation of the research report

•Generating patterns and templates for the test device

•User Interfaces

•Generating source code from patterns, templates and use defined parameters

•Preparation of the final report

Table 5.1: Simplified Gantt Chart of Project

TASK TITLE	TASK COMPLET-	WEEK 1	WEEK 2	WEEK 3	WEEK 4	WEEK 5	WEEK 6	WEEK 7	WEEK 8	WEEK 9	WEEK 10	WEEK 11	WEEK 12	WEEK 13	WEEK 14	WEEK 15	WEEK 16	WEEK 1	WEEK 17	WEEK 18	WEEK 19	WEEK 20	WEEK 20	WEEK 21	WEEK 22	WEEK 23	WEEK 24	WEEK 25	WEEK 26	WEEK 27	WEEK 28	WEEK 29	WEEK 30	WEEK 31	WEEK 32	WEEK 33	WEEK 34	WEEK 35	WEEK 36
Literature Survey																																							
Embedded System	100%	🟥																																					
Automatic Code Generation	100%		🟥	🟥																																			
Testbench	100%				🟥																																		
Hardware-in-the-loop simulation	100%					🟥	🟥																																
Design procedure																																							
System testing with microcontroller	100%							🟥	🟥																														
Test Vector Generation	100%									🟥	🟥																												
Automatic Code Generation	100%											🟥	🟥																										
User Interfaces	100%													🟥																									
Microcontroller and Other Component	100%														🟥																								
Preparation of the research report																																							
Writing Fall Term Thesis	100%									🟥	🟥	🟥	🟥	🟥	🟥																								
Generating patterns and templates for the test device																																							
Generating Templates	0%																							🟥	🟥														
Generating patterns	0%																									🟥	🟥	🟥	🟥	🟥	🟥								
User Interfaces																																							
Designin user Interfaces	0%																															🟥							
Test vector generation																																							
Test vector generator	0%																																🟥	🟥	🟥				
Generating source code from patterns, templates and use defined parameters																																							
Generating template based code	0%																							🟥	🟥														
Generating pattern based code	0%																									🟥	🟥	🟥	🟥	🟥	🟥	🟥							
Adding test vector	0%																																🟥	🟥	🟥				
Preparation of the final report																																							
Writing Spring Term Thesis	100%																																	🟥	🟥	🟥	🟥	🟥	🟥

6. COST ANALYSIS

As mentioned in the selection of hardware and components, 1 products will be purchased. This is one STM32F407G-DISC1 development board. In order to avoid cost, the development board has been decided and no extra PCB has been produced.

Table 6.1: Cost Analysis of Project

Component	Number of pieces	Prices
STM32F407G-DISC1	1	₺304.82
Total		₺304.82

6.1 Environmental Impact of the Project

As mentioned before, embedded system software and testing are time consuming. Computer environment is generally used in both. Automating the test shortens the process and causes less use of the computer environment. This reduces the carbon footprint of embedded system projects. It reduces the damage caused by the projects to the environment. This effect can be considered as a minor environmental impact factor.

6.2 Economical Impact of the Project

Competent test engineers should be employed in embedded system testing. Because in case of insufficient tests, the designer company may cause grievances. They may pay heavy compensation for these grievances. The salaries and training of these competent people are costly. The use of test automation ensures that these high costs are avoided. Test automation will not perform insufficient testing and will also prevent potential compensation lawsuits.

At the same time, since this project can be used as a test driven development, it reduces the design time and spends less time and effort. According to researches, reducing working times under certain conditions increases efficiency [3]. These effect can be considered as a major economical impact factor.

6.3 Social Impact of the Project

The main purpose of this project is to make people's work easier. The result of this facilitation of work causes a decrease in working times. During this time, the employees can perform different activities, socialize, improve themself or develop something.

6.4 Health Impact of the Project

This project generally facilitates the work of test engineers. If the test engineers do not use this project, they may need to produce other software for testing. Although these software developments do not harm health in the short term, they cause some occupational diseases in the long run such as osteochondrosis, sciatica and arthritis. These occupational diseases are usually problems related to musculoskeletal system functions, eye diseases and nervous system disorders [17]. Thanks to this project, these diseases will be prevented as the test is performed automatically.

6.5 Political and Ethical Impact of the Project

This project has no impact on politics or ethics.

7. CONCLUSION

The literature review provided information on the methods and techniques that will be used to design the desired program, as well as the components that these methods and techniques will require.

According on the study performed in the fall semester, a draft of the project was developed. Determined methods, techniques and their components were examined separately. The most appropriate techniques and choices were made to address the requirements while developing the methods and choices. Since there is no article or project like this project, approaches in different fields have been synthesized. The testbench , test vectors approaches in HDL, approaches related to code generation and HIL simulators were synthesized and generated to this project. In fact, components such as code generation and test vector should be added so that we can generalize the project, which works like a low-cost HIL simulation, and adapt it to each target system. It has been tried to increase the accuracy of the test by using the code coverage-based method in test vector production. The approach in code generation was chosen as a template-based passive generator, which was created by combining some of the techniques we examined in the research.

In the spring semester, each pattern was tested and integrated into the template. The data was taken from the user and assigned to the variables in the patterns that make up the test vector with that data. Finally, the integrity test was applied.

8. REFERENCES

[1] Embedded software testing tools. Web pages: `https://www.rapitasystems.com/embedded-software-testing-tools`. Accessed: 2021-12-26.

[2] Jonny Andersson. Automatic test vector generation and coverage analysis in model-based software development, 2005.

[3] Timo Anttila. Reduced working hours : reshaping the duration, timing and tempo of work. 01 2005.

[4] Andrejs Bajovs, Oksana Nikiforova, and Janis Sejans. Code generation from uml model: State of the art and practical implications. *Applied Computer Systems*, 2013.

[5] Arnold Berger. *Embedded systems design: an introduction to processes, tools, and techniques*. CRC Press, 2001.

[6] David S Bowden and James A Mynderse. From industry to the classroom: A low-cost hardware-in-loop simulator for classic controls experiments. In *2019 ASEE Annual Conference & Exposition*, 2019.

[7] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003.

[8] Alceu Bernardes Castanheira De Farias, Reurison Silva Rodrigues, André Murilo, Renato Vilela Lopes, and Suzana Avila. Low-cost hardware-in-the-loop platform for embedded control strategies simulation. *IEEE Access*, 2019.

[9] Wilbert O Galitz. *The essential guide to user interface design:an introduction to GUI design principles and techniques*. John Wiley & Sons, 2007.

[10] Jack Herrington. *Code generation in action*. Manning Publications Co., 2003.

[11] Alair Dias Junior and Diogenes Junior Cecilio da Silva. Code-coverage based test vector generation for systemc designs. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI'07)*, pages 198–206. IEEE, 2007.

[12] Ling Liu and M Tamer Özsu. *Encyclopedia of database systems*, volume 6. Springer New York, NY, USA:, 2009.

[13] Ron Ogan. *Hardware-in-the-Loop Simulation*, pages 167–173. 05 2015.

[14] Khaled Saab, Naim Ben-Hamida, and Bozena Kaminska. Parametric fault simulation and test vector generation. In *Proceedings Design, Automation and Test in Europe Conference and Exhibition 2000 (Cat. No. PR00537)*, pages 650–656. IEEE, 2000.

[15] Julio Sanchez and Maria P Canton. *Embedded systems circuits and programming*. CRC Press, 2012.

[16] Deepika P Vishala I Nari. Validation, timing analysis and power analysis of multifunctional io cell. *Journal of Critical Reviews*, 2020.

[17] MS Yarova. Occupational diseases of it-workers. the causes and recommendations for preventing occupational diseases. *BHTY*, 2020.