VECTOR

Hochschule Esslingen
University of Applied Sciences

Unit Tests with Google Test Framework

Writing Unit Tests according to the Test2020 internal guideline for an automotive Ethernet Component

Project thesis about the completion of the
practical internship semester
at Esslingen University of Applied Sciences,
Faculty Machines and systems

carried out in

Vector Informatik GmbH
Stuttgart

presented by
Philipp Saarmann
Carl Benz Straße 15
74321 Bietigheim-Bissingen
Matrikel-Nr.: 763530

Supervisor:
Prof. Dr.-Ing. Markus Kaupp(Esslingen University of Applied Sciences)
Tao Chen, GüntherPiehler(Vector Informatik GmbH)

Processing period:
1st of September 2023 until 29th of February 2024

Confirmed and approved through internship institution*:*

(Stamp/Signature)

Piehler,
Günther

Digital signiert von Piehler, Günther
DN: cn=Piehler, Günther,
email=Guenther.Piehler@vector.com
Datum: 2024.03.11 15:03:50 +01'00'

Declaration

I herewith declare on my word of honour,

1. that I have written my thesis on my own and did not use any unnamed sources or aid
2. that I have marked the places where I took quotes from literature and where I used thoughts of some other author´s work
3. that I have not presented the thesis in any other examination

I´m aware of the fact, that a false declaration would produce legal effects.

Bietigheim-Bissingen, 11.03.2024

Place, Date

D Saravanan

Signature

Table of contents

Introduction

1.1 Presentation of the Company

Vector is a proficient partner in the evolution of automotive electronics, a role it has held for over 30 years. With 31 locations worldwide and 400 employees, Vector supports manufacturers and suppliers in the automotive industry and related sectors by providing a professional platform of tools, software components, and services for the development of embedded systems. The company is also actively involved in education, research, and various societal initiatives. [1]

The primary focus of Vector products lies in the development of complex software projects, electronic systems, and their networking using serial bus systems such as CAN, LIN, FlexRay, and Ethernet. Some key products offered by Vector include: [1]

- CANalyzer: An analysis tool for bus systems such as CAN, FlexRay, Ethernet, LIN, etc. [1]
- CANoe: A development tool that supports simulation, diagnostics, and serves as a test tool for car ECUs. It is widely used by most car and truck OEMs and suppliers. [1]
- CANape: Software designed for calibrating ECUs, supporting various protocols (CCP, XCP, etc.) and bus systems (CAN, FlexRay, etc.). [1]
- PREEvision: Software used for modeling electrical/electronic systems.
- Development tools and complete software stacks (MICROSAR) for AUTOSAR control units, including a real-time operating system. This system is used by many OEMs and conforms to the AUTOSAR OS specification. Its modules enable the implementation of safety-related functions in vehicles according to ISO 26262 up to ASIL D. [1]

1.2 Presentation of the department

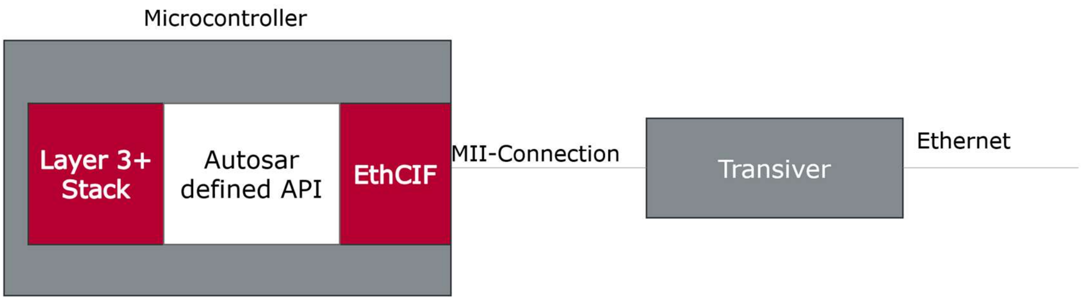
The department is developing the Ethernet interface component for the Microsar product line. The Microsar product line essentially consists of our AUTOSAR code and the generator for it, along with everything connected to it for the output. The Ethernet interface component is responsible for everything that is typically connected to the data link layer in the OSI Model. This component directly accesses the registers on the ECU. Features of this component include basic communication, which entails sending and receiving frames in both interrupt mode and polling mode. In interrupt mode, frame processing occurs directly within the interrupt. In polling mode, frame processing is carried out in cyclic tasks. [2]

Another feature is multi-controller support, allowing the Ethernet controller to operate on multiple controllers of the same type within the same ECU. Additionally, there's physical address filtering, enabling frames to be filtered with MAC addresses. [2]

The component also offers time synchronization, allowing frames to be timestamped. Furthermore, it provides package gap configuration, determining the minimal length of a gap between two Ethernet frames. [2]

Quality of service is another feature, enabling frames to be processed between queues and priorities. The component supports forwarding and queuing of time-sensitive streams, allowing the bandwidth of a sender queue to be limited. [2]

Lastly, there's a BSW (Basis Software) Virtual Device (BSVID) feature, enabling the basis software component to run in different partitions and be executed in parallel. [2]



2 Definitions

2.1 Definition of my task

My primary responsibility was to create unit tests for the Ethernet controller interface in accordance with the Vector Internal Test 2020 Working Guidelines, using the Google Test framework and Google Mock. [2]

As my work progressed, additional tasks arose, such as evaluating the performance of our two different frameworks and the new remote PCs. We also had to address a compiler issue in our test toolchain that required resolution. In addition, parameterized tests were introduced, which is a feature of the Google Test framework. [2]

A special task was also undertaken concerning the TX unit, which is by far the largest unit. Its size posed a significant challenge, resulting in the development of a slightly different test structure compared to the usual structure of our unit tests. [2]

2.2 Definition of Unit Testing

Unit testing is typically a type of software testing that tests individual units or components of a software application isolated from the rest of the application. The purpose of unit testing is to validate that each unit of the software performs as expected according to its design specifications. Unit tests are usually automated and focus on testing small, specific parts of the code base, such as functions, methods, or classes, in our deparment a public API. They help to ensure that each unit of code behaves correctly and produces the expected output for a given set of inputs. Unit tests play a crucial role in identifying bugs early in the development process, promoting code reliability, and facilitating software maintenance and refactoring. [3]

2.3 Definition different testing Methods

Black box testing is a software testing technique where the internal workings or implementation details of the software under test are unknown to the tester. The tester focuses only on the external behaviour of the software. He tests its functionality against specified requirements or inputs. Black box testing treats the software as a black box. Only inputs and outputs are considered, without any knowledge of how the software processes them internally. [3]

White-box testing, also known as clear-box or structural testing, is a software testing technique where the tester has access to the internal code, structure and implementation details of the software tested. In white-box testing, the internal logic, branches, paths and data structures of the software are tested. This is done by examining the source code, design documentation or architecture. To ensure that all code paths and conditions are thoroughly tested, white-box test cases are designed based on an understanding of the software's internal workings. [3]

Grey box testing is a software testing technique that combines elements of both black box and white box testing. Grey box testing involves the tester having partial knowledge of the inner workings of the software being tested. The tester designs test cases that validate the functionality and behaviour of the software, taking some account of its internal structure and logic, using a combination of black-box and white-box testing techniques. Grey box testing allows the tester to take advantage of both the external and internal perspectives in order to achieve comprehensive test coverage [3]

In theory, our team should only use grey box testing, according to our working guideline. However, this was not always possible. So in our day-to-day work we tried to use a mixture of gray box and white box testing. While our test specifications followed the principles of grey box testing, focusing on external behaviour without detailed internal knowledge, the implementation of the tests leaned more towards white box testing. This approach allowed us to strike a balance between verifying external functionality and ensuring comprehensive coverage of internal logic and code paths.

3 State of the art

3.1 Toolchain

The toolchain used in our department is called a 'test plan'. It consists of a set of scripts created by our CDK department. These scripts are versatile and provide utilities such as dynamic and static code analysis and documentation generation. [2]

Our working guide required us to access public APIs only, which meant that if we needed to test a particular internal function within a public API, we had to arrange the test to reach that particular part of the function via the public API call. To test, we primarily used the Microsoft Visual Studio Build Tools, on the 2019 release. Our build system is CMake, and we used Ninja as our build generator. [2]

To measure coverage, we used VectorCAST, a product from Vector, which also served as our testing framework. However, we are looking to move away from VectorCAST and migrate to the Google Test Framework, as our test framework in the background, VectorCAST will still be used to check code coverage. [2]

3.2 My task in context of the big picture

In my role, I focused on a specific aspect of dynamic code analysis. This involved generating a test report that describes the unit tests and code coverage of the functions within our units. We require 100% code coverage on a unit-by-unit basis to achieve an ASIL-D release. This is a requirement of our quality management. However, we recognise that this is not something that can be achieved by black box testing alone. We formost need to use the available information from the CDD, as well as the public API drief description of public API, this however need to be substuted with checking the C Code [2]

3.3 Work environment

Our Google Test unit testing group worked within a Scrum framework environment, using Jira for ticketing, and monitoring our work

3.4 Google Test

Google Test, commonly known as gtest, is a specialised C++ library designed primarily for unit testing. It operates under the BSD 3-clause licence, which ensures flexibility in its use and distribution. Google Test follows the xUnit architecture, a systematic approach to evaluating software components. [2]

One of the key features of Google Test is its platform compatibility, supporting Linux, Microsoft Windows and MacOS. It integrates seamlessly with different operating systems, including those using POSIX and Windows platforms, enabling developers to run unit tests on C and C++ codebases with minimal modification. [2]

Notably, Google Test has gained popularity beyond its development at Google. Many prominent projects, such as the Android open source project, the Chromium projects (including the Chrome browser and ChromeOS), the LLVM compiler, protocol buffers, the OpenCV computer vision library, the Robot operating system, and the Gromacs molecular dynamics simulation package, use Google Test for their unit testing needs. [2]

Overall, Google Test provides a robust framework for unit testing C++ projects, offering flexibility, platform compatibility and integration with various software development ecosystems. [2]

The most used features in our department are:

3.4.1 Assert_EQ

Assert_EQ is an assertion macro provided by Google Test for comparing two values for equality. It checks that the expected value is equal to the actual value. If the assertion fails, the test fails and produces an error message indicating the values that did not match. [4]

3.4.2 Expect_Call/On_Call

These macros are used for mocked functions or methods during unit testing. Expect_Call sets the expectation that a particular function or method will be called with specified arguments, while On_Call specifies the behaviour or return value of the function or method when it's called during the test. These macros can be called with different options that defne the behaviour of the mocked function in the Test Code: [4]

3.4.2.1 WillRepeatedly:

WillRepeatedly is a method used to specify the behaviour of a mocked method that is expected to be called multiple times.It defines the action to be taken each time the mocked method is called during the test.Typically, you would use WillRepeatedly in conjunction with EXPECT_CALL to specify repetitive behaviour. [4]

3.4.2.2 WillOnce:

WillOnce is used to define the behaviour of a mocked method for a single call.It specifies the action to be taken on the first call to the mocked method only.This is useful if you want a particular behaviour to occur only once during the test. [4]

3.4.2.3 DoAll:

DoAll is a method used to define a sequence of actions to be performed when a mock method is invoked.It allows you to specify multiple actions to be performed in sequence, such as returning a value, performing a side effect, etc.DoAll is often used when you need to perform multiple actions in response to a single method call. [4]

3.4.2.4 Times

The Times() method is used in conjunction with EXPECT_CALL to specify how many times a mock method should be expected to be called during the test. Times(n) is a method used to specify the expected number of times a fake method should be called during the test. n is an integer representing the exact number of times the method should be called. If the method is called more or less than Times(n), the test will fail. It can also be useed Times(0) to specify that a method should not be called at all during the test. [4]

3.4.2.5 WillByDefault:

WillByDefault is used to set a default action for a mock method when no explicit behaviour is specified.It defines what the mock method should do if no other expectations or behaviours are set.This is particularly useful for defining a default return value or action for mock methods that may not be called explicitly during testing.

3.4.3 Fixtures and Test_F

Fixtures in Google Test allow you to set up common test preconditions and cleanup steps for multiple test cases within a test suite. Test_F is a test fixture used to group related test cases. The Fixture class can contain SetUp and TearDown methods. These methods are called before and after each test-case. After a test-case has run, the values in the fixture are reset to the value before the test-case was run. [4]

3.4.4 Parametrized Test and TEST_P

Parametrized tests enable you to run the same test logic with different input values. TEST_P is a macro used to define parametrized test cases. It allows you to specify parameters and iterate through different sets of values for testing. [4]

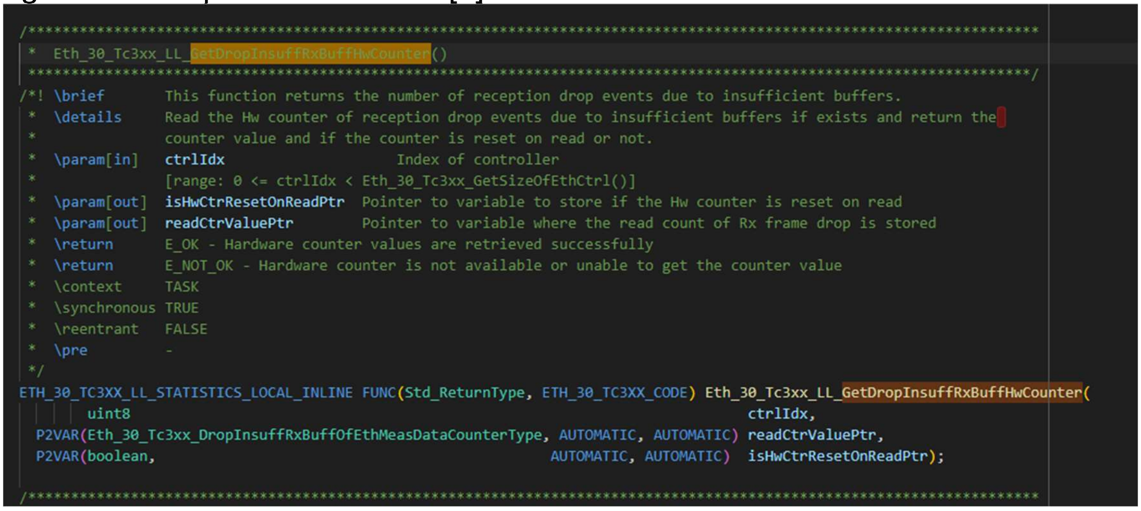
4 Description of implementation

4.1 Methodology Writing Unit tests with Google Test

4.1.1 Guidelines and Requirements for a Test

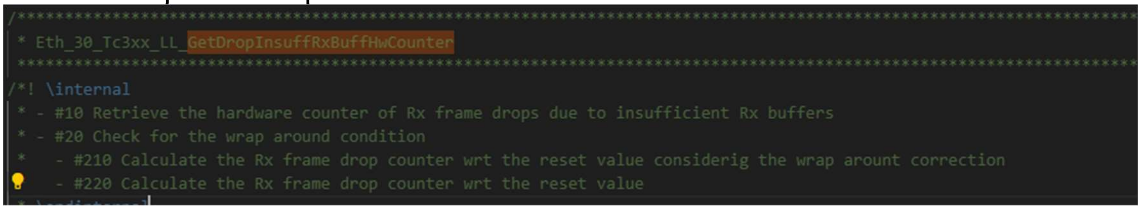
The Test 2020 working guideline states that the purpose of unit tests is to verify the implementation of a component's detailed design. As such, our test descriptions are based on the information provided in the public API documentation, both in brief and detailed forms. [6] Our aim was to closely follow the Triple A testing structure (Arrange, Act, and Assert)

1. Arranging the necessary preconditions and configuring the test environment to prepare for the test execution. Asserting the expected outcome.[5]
2. Performing the specific action or operations being tested, usually by invoking the function or method under examination. [5]
3. Check the result or state of the action taken during the 'Act' phase against the expected outcome. [5]



```
/*****************************************************************************************************
 *  Eth_30_Tc3xx_LL_GetDropInsuffRxBuffHwCounter()
 ****************************************************************************************************/
/*! \brief       This function returns the number of reception drop events due to insufficient buffers.
 *  \details     Read the Hw counter of reception drop events due to insufficient buffers if exists and return the
 *               counter value and if the counter is reset on read or not.
 *  \param[in]   ctrlIdx                    Index of controller
 *               [range: 0 <= ctrlIdx < Eth_30_Tc3xx_GetSizeOfEthCtrl()]
 *  \param[out]  isHwCtrResetOnReadPtr  Pointer to variable to store if the Hw counter is reset on read
 *  \param[out]  readCtrValuePtr        Pointer to variable where the read count of Rx frame drop is stored
 *  \return      E_OK - Hardware counter values are retrieved successfully
 *  \return      E_NOT_OK - Hardware counter is not available or unable to get the counter value
 *  \context     TASK
 *  \synchronous TRUE
 *  \reentrant   FALSE
 *  \pre         -
 */
ETH_30_TC3XX_LL_STATISTICS_LOCAL_INLINE FUNC(Std_ReturnType, ETH_30_TC3XX_CODE) Eth_30_Tc3xx_LL_GetDropInsuffRxBuffHwCounter(
        uint8                                                                   ctrlIdx,
  P2VAR(Eth_30_Tc3xx_DropInsuffRxBuffOfEthMeasDataCounterType, AUTOMATIC, AUTOMATIC) readCtrValuePtr,
  P2VAR(boolean,                                              AUTOMATIC, AUTOMATIC)  isHwCtrResetOnReadPtr);

/*****************************************************************************************************
```

Brief description of a public API in the source code



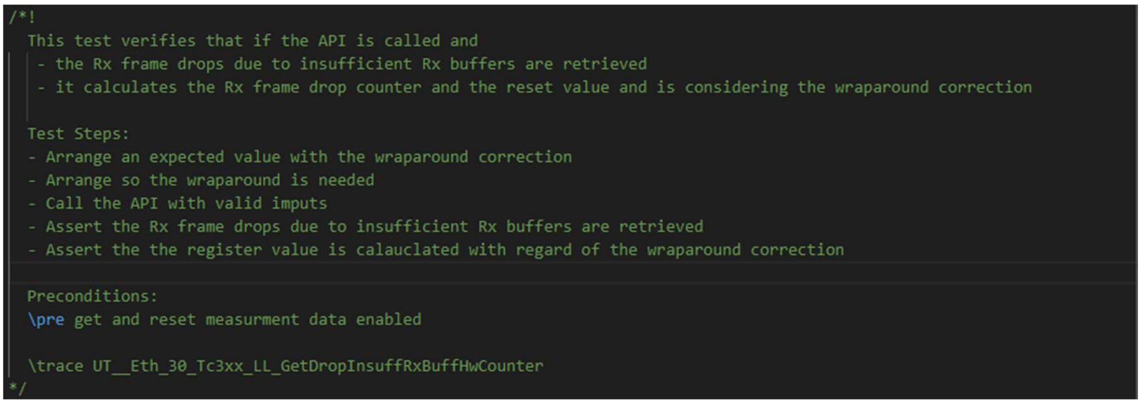
```
/*******************************************************************************************************
 * Eth_30_Tc3xx_LL_GetDropInsuffRxBuffHwCounter
 *******************************************************************************************************
/*! \internal
 * - #10 Retrieve the hardware counter of Rx frame drops due to insufficient Rx buffers
 * - #20 Check for the wrap around condition
 *   - #210 Calculate the Rx frame drop counter wrt the reset value considerig the wrap arount correction
     - #220 Calculate the Rx frame drop counter wrt the reset value
```

Detailed description of a public API in the source code

The first stage in the writing of a test is the drafting of a test specification. This doxygen comment should begin with a brief test description. After the test description, tests are outlined in test steps, but only when the complexity of the tests requires it. Our working guideline does not provide precise instructions on when test steps are necessary.In our department, we follow a rule regarding the number of test steps required. If a test can be described with only two test steps, we do not write additional steps. If three test steps are needed, it is optional to include them. However, if a test requires four or more test steps, it is mandatory to include them. It is important to note that the subsequent test code may not follow the order of the test steps, but the test steps themselves must adhere to the Triple A structure. [6]

Subsequently, the preconditions are described. These preconditions involve variant switches implemented as macros in the source code. [6] Currently, nine different configurations are being tested in which various features are enabled or disabled. These variants are generated from a program designed to explore all possible variant interactions across as few configurations as possible.

Finally, the trace is included, which is mandatory. The trace signals indicate which test is associated with which public API in the test plan, facilitating the organization and management of the testing process. [6]



```
/*!
  This test verifies that if the API is called and
   - the Rx frame drops due to insufficient Rx buffers are retrieved
   - it calculates the Rx frame drop counter and the reset value and is considering the wraparound correction

  Test Steps:
  - Arrange an expected value with the wraparound correction
  - Arrange so the wraparound is needed
  - Call the API with valid imputs
  - Assert the Rx frame drops due to insufficient Rx buffers are retrieved
  - Assert the the register value is calauclated with regard of the wraparound correction

  Preconditions:
  \pre get and reset measurment data enabled

  \trace UT__Eth_30_Tc3xx_LL_GetDropInsuffRxBuffHwCounter
*/
```

Example Test specification

4.1.2 Writing a Test Case

However, with the Google Test Framework, especially when dealing with mocks, it can be challenging to distinguish between the Arrange and Assert parts. This is mainly due to the Expect call macro. This asserts that a function in a mocked unit will be called with certain parameters. It also arranges the behaviour of that function using the WillRepeatedly and WillOnce macros, as well as the DoAll argument macros. This blurs the line between Arrange and Assert.

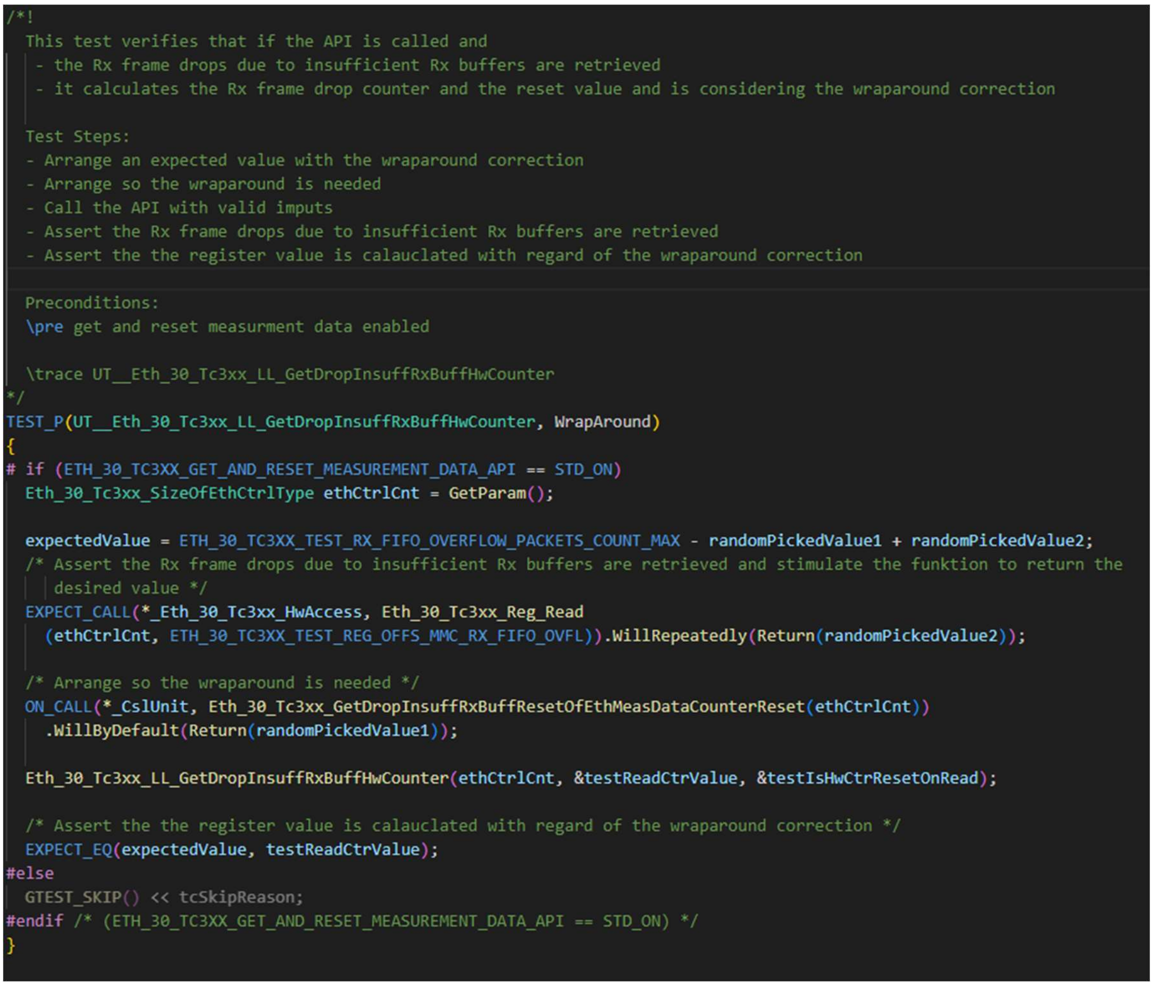
I also tried to incorporate the software engineering concepts of KISS(Keep It Simple, Stupid) and DRY(Don't Repeat Yourself). These principles emphasise simplicity in design and code structure. They also emphasise the avoidance of redundancy. [7]

By following the KISS principle, I aimed to keep the design and implementation of our tests simple and easy to understand. This involved breaking down complex tasks into smaller, more manageable components and avoiding unnecessary complications. [7]

Similarly, following the DRY principle helped ensure that I avoided duplicating code or logic across our test suite. Instead, I tried to encapsulate common functionality into reusable modules or functions, reducing the risk of errors and making our codebase more maintainable. [7]

My approach to writing test code is commendable and in line with software development best practice. Here are my key points based on my approach to writing code: [7]

1. Self-explanatory code: I strive to write code that clearly expresses its intent and functionality without the need for extensive comments or external documentation. Well named variables, functions to make the code self-explanatory. [7]
2. Readable code: I prioritise readability by following consistent coding conventions and formatting guidelines from our working guidelines. This helps ensure that the code is easy to navigate and understand, even for developers who are new to the project or unfamiliar with the codebase. [7]
3. Clear intent: Make sure that the purpose and functionality of each section of code is clear from the code itself. Avoid ambiguity and overly complex logic that may obscure the intent of the code. [7]
4. Comments for clarification: I try to use comments sparingly, they can be valuable for explaining the rationale behind certain design decisions or complex algorithms, they should focus on providing insight into why the code behaves in a certain way rather than repeating what the code does. [7]



```
/*!
  This test verifies that if the API is called and
   - the Rx frame drops due to insufficient Rx buffers are retrieved
   - it calculates the Rx frame drop counter and the reset value and is considering the wraparound correction

  Test Steps:
  - Arrange an expected value with the wraparound correction
  - Arrange so the wraparound is needed
  - Call the API with valid imputs
  - Assert the Rx frame drops due to insufficient Rx buffers are retrieved
  - Assert the the register value is calauclated with regard of the wraparound correction

  Preconditions:
  \pre get and reset measurment data enabled

  \trace UT__Eth_30_Tc3xx_LL_GetDropInsuffRxBuffHwCounter
*/
TEST_P(UT__Eth_30_Tc3xx_LL_GetDropInsuffRxBuffHwCounter, WrapAround)
{
# if (ETH_30_TC3XX_GET_AND_RESET_MEASUREMENT_DATA_API == STD_ON)
  Eth_30_Tc3xx_SizeOfEthCtrlType ethCtrlCnt = GetParam();

  expectedValue = ETH_30_TC3XX_TEST_RX_FIFO_OVERFLOW_PACKETS_COUNT_MAX - randomPickedValue1 + randomPickedValue2;
  /* Assert the Rx frame drops due to insufficient Rx buffers are retrieved and stimulate the funktion to return the
     desired value */
  EXPECT_CALL(*_Eth_30_Tc3xx_HwAccess, Eth_30_Tc3xx_Reg_Read
    (ethCtrlCnt, ETH_30_TC3XX_TEST_REG_OFFS_MMC_RX_FIFO_OVFL)).WillRepeatedly(Return(randomPickedValue2));

  /* Arrange so the wraparound is needed */
  ON_CALL(*_CslUnit, Eth_30_Tc3xx_GetDropInsuffRxBuffResetOfEthMeasDataCounterReset(ethCtrlCnt))
    .WillByDefault(Return(randomPickedValue1));

  Eth_30_Tc3xx_LL_GetDropInsuffRxBuffHwCounter(ethCtrlCnt, &testReadCtrValue, &testIsHwCtrResetOnRead);

  /* Assert the the register value is calauclated with regard of the wraparound correction */
  EXPECT_EQ(expectedValue, testReadCtrValue);
#else
  GTEST_SKIP() << tcSkipReason;
#endif /* (ETH_30_TC3XX_GET_AND_RESET_MEASUREMENT_DATA_API == STD_ON) */
}
```

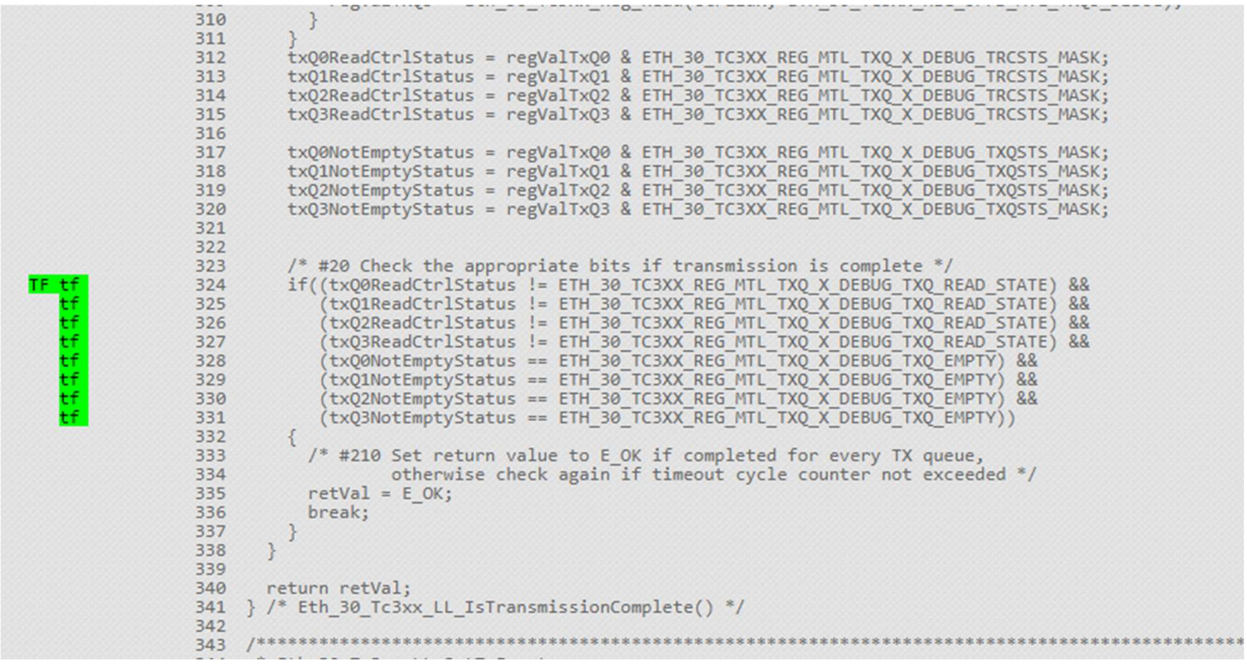
Example Test specification

4.2 Parameterized tests

4.2.1 The problem

As mentioned in the section above, I try to follow the DRY concept. However, I also consider how VectorCAST measures 100% code coverage. It's not just about traversing all possible paths. It's also about testing each decision point to determine the path taken. In the past, however, this approach resulted in us copying tests multiple times to cover every decision point within a function.

The Google testing framework provides an elegant solution to this problem through parameterised tests, which work by iterating through a parameter list containing an array of different parameters within a single test case. As a result, each decision can be tested on its own without duplicating code. As an added benefit, if a test fails, our setup in Visual Studio Code can identify the parameter that is not working and jump to the run with the specific parameter. This is particularly useful when testing multiple controllers. It makes debugging much more convenient. Unfortunately, at the time there was no guidance in our working guide on how to use this feature, in particular how to describe the parameters in the test specification, so we could not be sure of getting ASIL-D approval from our quality management for this component using this feature.

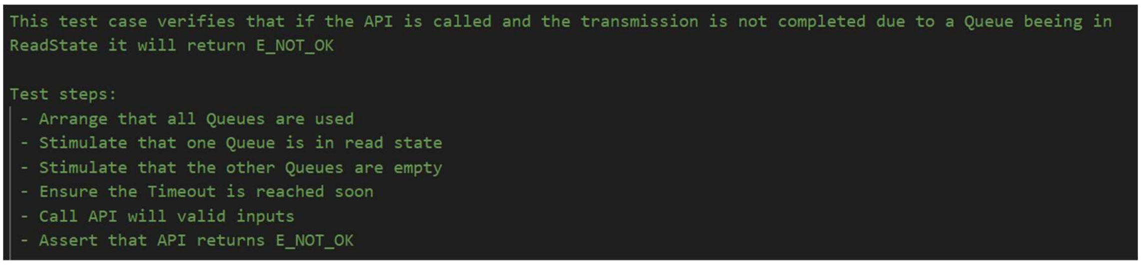


```
        }
      }
      txQ0ReadCtrlStatus = regValTxQ0 & ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TRCSTS_MASK;
      txQ1ReadCtrlStatus = regValTxQ1 & ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TRCSTS_MASK;
      txQ2ReadCtrlStatus = regValTxQ2 & ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TRCSTS_MASK;
      txQ3ReadCtrlStatus = regValTxQ3 & ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TRCSTS_MASK;

      txQ0NotEmptyStatus = regValTxQ0 & ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TXQSTS_MASK;
      txQ1NotEmptyStatus = regValTxQ1 & ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TXQSTS_MASK;
      txQ2NotEmptyStatus = regValTxQ2 & ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TXQSTS_MASK;
      txQ3NotEmptyStatus = regValTxQ3 & ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TXQSTS_MASK;


      /* #20 Check the appropriate bits if transmission is complete */
      if((txQ0ReadCtrlStatus != ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TXQ_READ_STATE) &&
         (txQ1ReadCtrlStatus != ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TXQ_READ_STATE) &&
         (txQ2ReadCtrlStatus != ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TXQ_READ_STATE) &&
         (txQ3ReadCtrlStatus != ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TXQ_READ_STATE) &&
         (txQ0NotEmptyStatus == ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TXQ_EMPTY) &&
         (txQ1NotEmptyStatus == ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TXQ_EMPTY) &&
         (txQ2NotEmptyStatus == ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TXQ_EMPTY) &&
         (txQ3NotEmptyStatus == ETH_30_TC3XX_REG_MTL_TXQ_X_DEBUG_TXQ_EMPTY))
      {
        /* #210 Set return value to E_OK if completed for every TX queue,
                otherwise check again if timeout cycle counter not exceeded */
        retVal = E_OK;
        break;
      }
    }

    return retVal;
  } /* Eth_30_Tc3xx_LL_IsTransmissionComplete() */

  /***************************************************************************************************
```

Example of a VectorCAST code coverage report

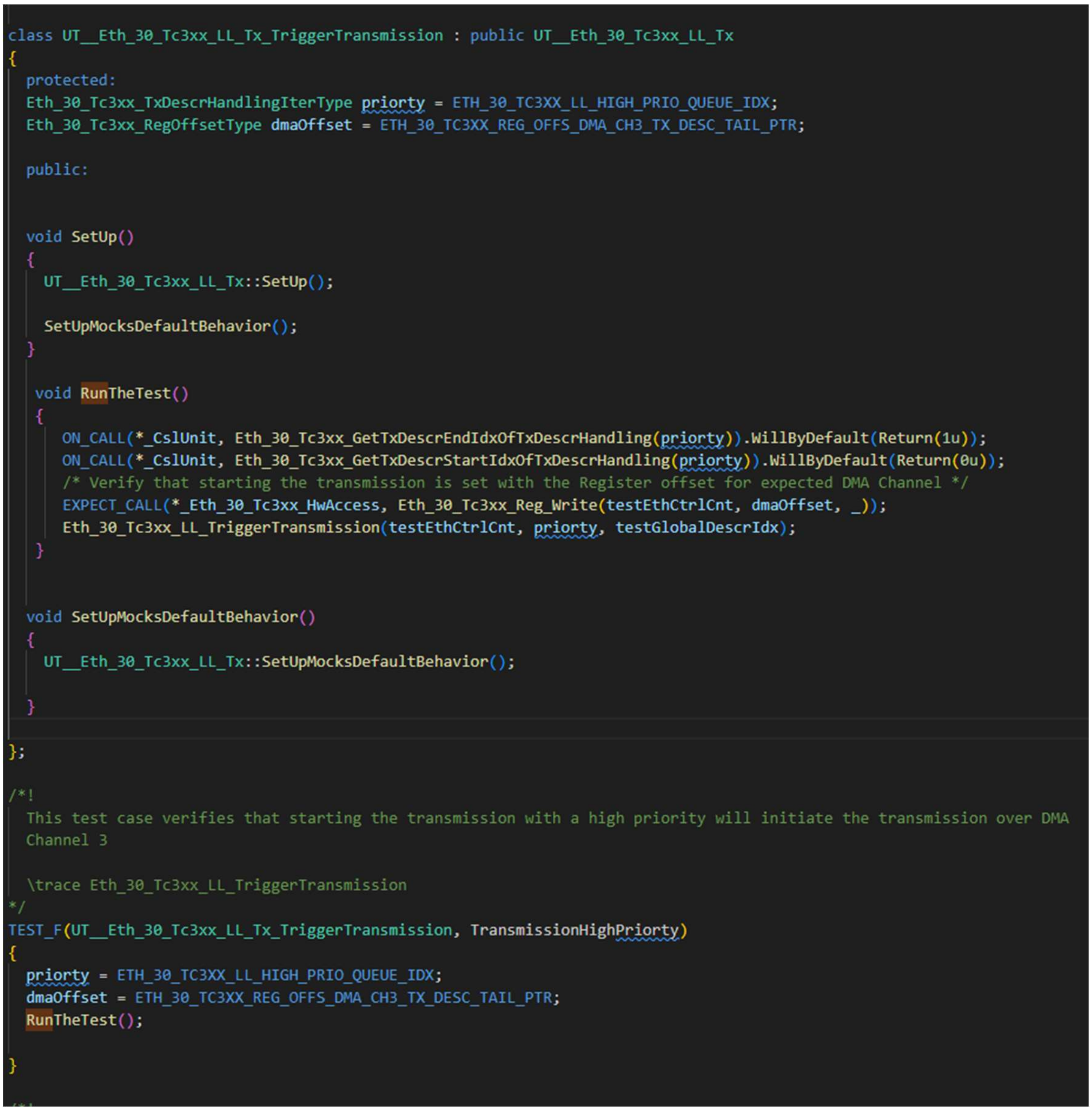
4.2.2 The soultion

After discussing this problem with our head department's senior safety coach, we came to a conclusion that will be included in the next version of our test working guide: that only parameters that are considered to be in an equivalence class of the function can be used, so they don't need to be explained in the test specification. Another solution is to create a new fixture for a helper function that contains most of the code, it could be the setup or teardown function of the fixture(Explained in the next champter), but a normal function improves readability. A new fixture is not required, but it does reduce clutter, so in the opinion of the safety coach it is preferable. As a result, only the variables need to be changed in a test case, and the test specification can be used as usual as a solution if it cannot be broken down into an equivalence class. Compared to our old approach, this approach reduces code duplication and increases test case maintainability.



```
This test case verifies that if the API is called and the transmission is not completed due to a Queue beeing in
ReadState it will return E_NOT_OK

Test steps:
 - Arrange that all Queues are used
 - Stimulate that one Queue is in read state
 - Stimulate that the other Queues are empty
 - Ensure the Timeout is reached soon
 - Call API will valid inputs
 - Assert that API returns E_NOT_OK
```

Example of a test description and the steps of a parametrised test.

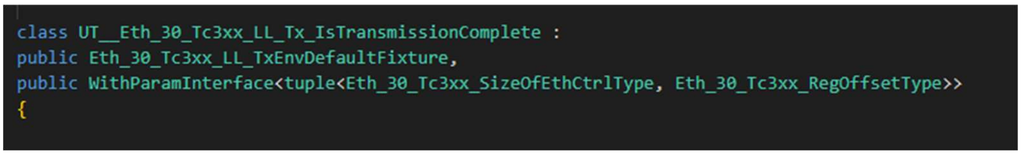


```
class UT__Eth_30_Tc3xx_LL_Tx_TriggerTransmission : public UT__Eth_30_Tc3xx_LL_Tx
{
  protected:
  Eth_30_Tc3xx_TxDescrHandlingIterType priorty = ETH_30_TC3XX_LL_HIGH_PRIO_QUEUE_IDX;
  Eth_30_Tc3xx_RegOffsetType dmaOffset = ETH_30_TC3XX_REG_OFFS_DMA_CH3_TX_DESC_TAIL_PTR;

  public:


  void SetUp()
  {
    UT__Eth_30_Tc3xx_LL_Tx::SetUp();

    SetUpMocksDefaultBehavior();
  }

   void RunTheTest()
   {
      ON_CALL(*_CslUnit, Eth_30_Tc3xx_GetTxDescrEndIdxOfTxDescrHandling(priorty)).WillByDefault(Return(1u));
      ON_CALL(*_CslUnit, Eth_30_Tc3xx_GetTxDescrStartIdxOfTxDescrHandling(priorty)).WillByDefault(Return(0u));
      /* Verify that starting the transmission is set with the Register offset for expected DMA Channel */
      EXPECT_CALL(*_Eth_30_Tc3xx_HwAccess, Eth_30_Tc3xx_Reg_Write(testEthCtrlCnt, dmaOffset, _));
      Eth_30_Tc3xx_LL_TriggerTransmission(testEthCtrlCnt, priorty, testGlobalDescrIdx);
   }


  void SetUpMocksDefaultBehavior()
  {
    UT__Eth_30_Tc3xx_LL_Tx::SetUpMocksDefaultBehavior();

  }

};

/*!
  This test case verifies that starting the transmission with a high priority will initiate the transmission over DMA
  Channel 3

  \trace Eth_30_Tc3xx_LL_TriggerTransmission
*/
TEST_F(UT__Eth_30_Tc3xx_LL_Tx_TriggerTransmission, TransmissionHighPriorty)
{
  priorty = ETH_30_TC3XX_LL_HIGH_PRIO_QUEUE_IDX;
  dmaOffset = ETH_30_TC3XX_REG_OFFS_DMA_CH3_TX_DESC_TAIL_PTR;
  RunTheTest();

}
```

Example of a test case where the parameter is in no equivalence class thus no Parameterized test could be used

4.2.3 How to implement Parameterized tests

First of all, the parameters need to be declared in the class definition of the fixture. [6] Fixtures are described in more detail in the next chapter.

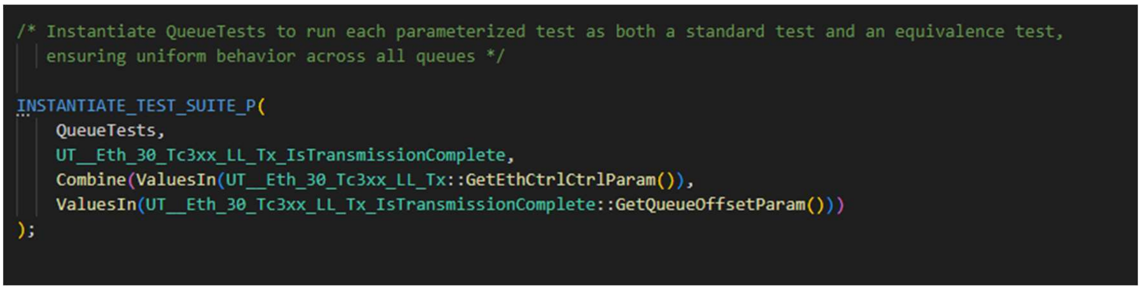


```
class UT__Eth_30_Tc3xx_LL_Tx_IsTransmissionComplete :
public Eth_30_Tc3xx_LL_TxEnvDefaultFixture,
public WithParamInterface<tuple<Eth_30_Tc3xx_SizeOfEthCtrlType, Eth_30_Tc3xx_RegOffsetType>>
{
```

Example of class definition of the fixture for a parameterized test

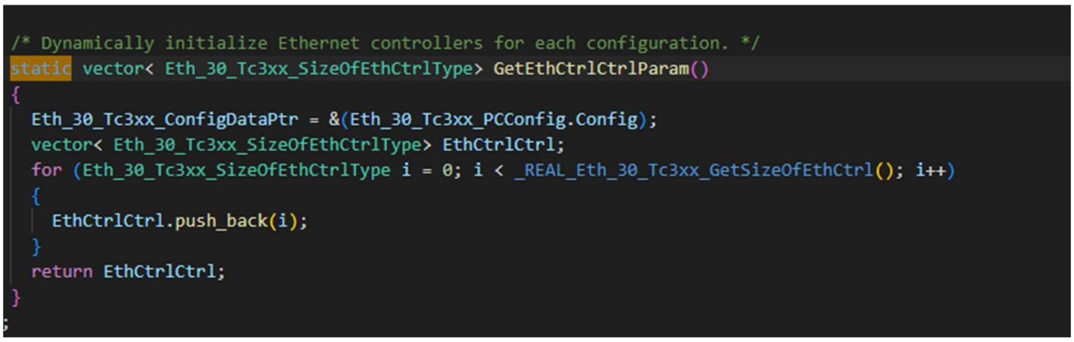
A fixture requires a parameter list. This is initialised when using the framework's TEST_P macro. [4] A common use case in our team for unit testing is to use the

Ethernet controller index as a parameter. Another use case for this could be to use different queues.



```
/* Instantiate QueueTests to run each parameterized test as both a standard test and an equivalence test,
   ensuring uniform behavior across all queues */

INSTANTIATE_TEST_SUITE_P(
    QueueTests,
    UT__Eth_30_Tc3xx_LL_Tx_IsTransmissionComplete,
    Combine(ValuesIn(UT__Eth_30_Tc3xx_LL_Tx::GetEthCtrlCtrlParam()),
    ValuesIn(UT__Eth_30_Tc3xx_LL_Tx_IsTransmissionComplete::GetQueueOffsetParam()))
);
```

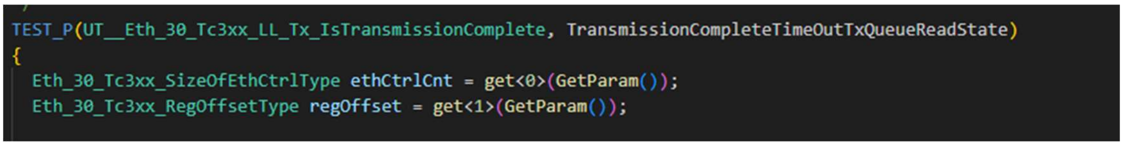
Example of an initiation of a parameter list



```
/* Dynamically initialize Ethernet controllers for each configuration. */
static vector< Eth_30_Tc3xx_SizeOfEthCtrlType> GetEthCtrlCtrlParam()
{
  Eth_30_Tc3xx_ConfigDataPtr = &(Eth_30_Tc3xx_PCConfig.Config);
  vector< Eth_30_Tc3xx_SizeOfEthCtrlType> EthCtrlCtrl;
  for (Eth_30_Tc3xx_SizeOfEthCtrlType i = 0; i < _REAL_Eth_30_Tc3xx_GetSizeOfEthCtrl(); i++)
  {
    EthCtrlCtrl.push_back(i);
  }
  return EthCtrlCtrl;
}
;
```

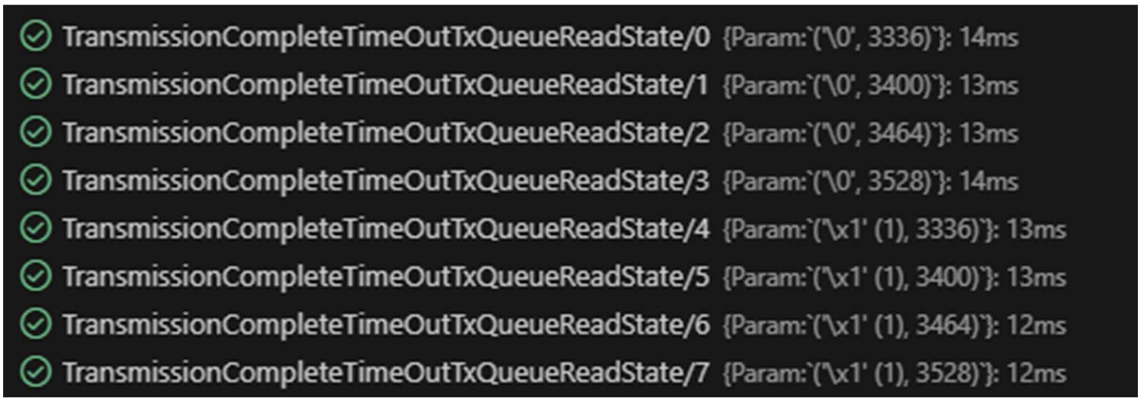
Example of a helper function to dynamically initialize ethernet controller based on the current configuration.

The parameter can then be retrieved for a test using the get parameter method. In the examples used in this section two parameters are combined in a C++ tuple. The first is the Ethernet controller count and the second is the register offset of a queue.



```
TEST_P(UT__Eth_30_Tc3xx_LL_Tx_IsTransmissionComplete, TransmissionCompleteTimeOutTxQueueReadState)
{
  Eth_30_Tc3xx_SizeOfEthCtrlType ethCtrlCnt = get<0>(GetParam());
  Eth_30_Tc3xx_RegOffsetType regOffset = get<1>(GetParam());
```

Example get parameter.



```
TransmissionCompleteTimeOutTxQueueReadState/0 {Param:`('\0', 3336)`}: 14ms
TransmissionCompleteTimeOutTxQueueReadState/1 {Param:`('\0', 3400)`}: 13ms
TransmissionCompleteTimeOutTxQueueReadState/2 {Param:`('\0', 3464)`}: 13ms
TransmissionCompleteTimeOutTxQueueReadState/3 {Param:`('\0', 3528)`}: 14ms
TransmissionCompleteTimeOutTxQueueReadState/4 {Param:`('\x1' (1), 3336)`}: 13ms
TransmissionCompleteTimeOutTxQueueReadState/5 {Param:`('\x1' (1), 3400)`}: 13ms
TransmissionCompleteTimeOutTxQueueReadState/6 {Param:`('\x1' (1), 3464)`}: 12ms
TransmissionCompleteTimeOutTxQueueReadState/7 {Param:`('\x1' (1), 3528)`}: 12ms
```

Example results of a parametrised test with 2 Ethernet Controller and 4 different Queues.

4.3 Tx-Unit Test Structure

4.3.1 The Problem

Another important task I undertook was to design a slightly different test structure for the TX unit. This was due to the extreme complexity of this unit and the restriction in our working guide which prohibits stubbing within a unit.

Our usual structure consists of a CPP file for all tests within a unit and a header file. In this CPP file, we will typically have a fixture for each of the unit's public APIs. In a typical unit, we will have about three to ten public APIs, with only one or two internal helper functions that are usually only used within one API.

However, there are seven public APIs in the TX unit. At the time we designed the structure, we had 31 internal helper functions. These were called sequentially in different public APIs. In addition, each internal helper function was sometimes twice or more the size and complexity of a normal public API in another unit.

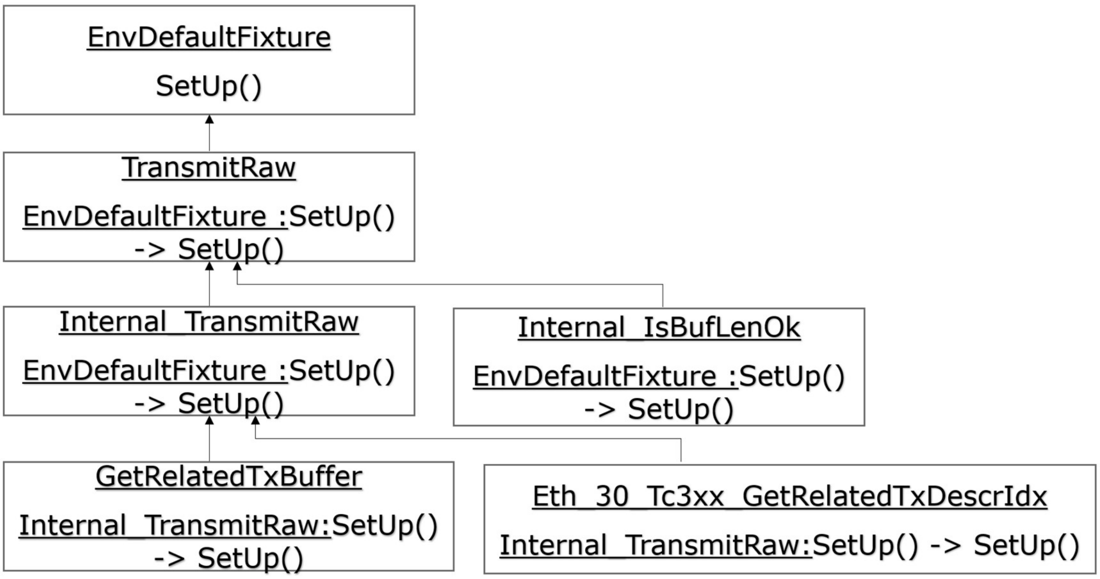
Our team decided to use a single CPP file for a single public API to reduce clutter and improve organisation. While this strategy may seem sufficient for less complex APIs, it resulted in an extremely confusing and unwieldy test for more complex APIs. The fixtures contained numerous helper functions that were not used in every part of the code, making the fixture itself bloated. We also ended up unnecessarily testing the code multiple times because the internal functions were called in multiple public APIs.

4.3.2 Using Fixture in Google Test as objekts

In Google Test, fixtures are basically CPP classes that inherit a Google Test framework class that defines all test macros. Additional features used in the tests, such as parameterised tests, are also controlled by inheritance from other classes. Only functions and variables within the scope of the fixture could be used in a test. In addition, at the end of each test run, these variables are reset for the next test run. [4]

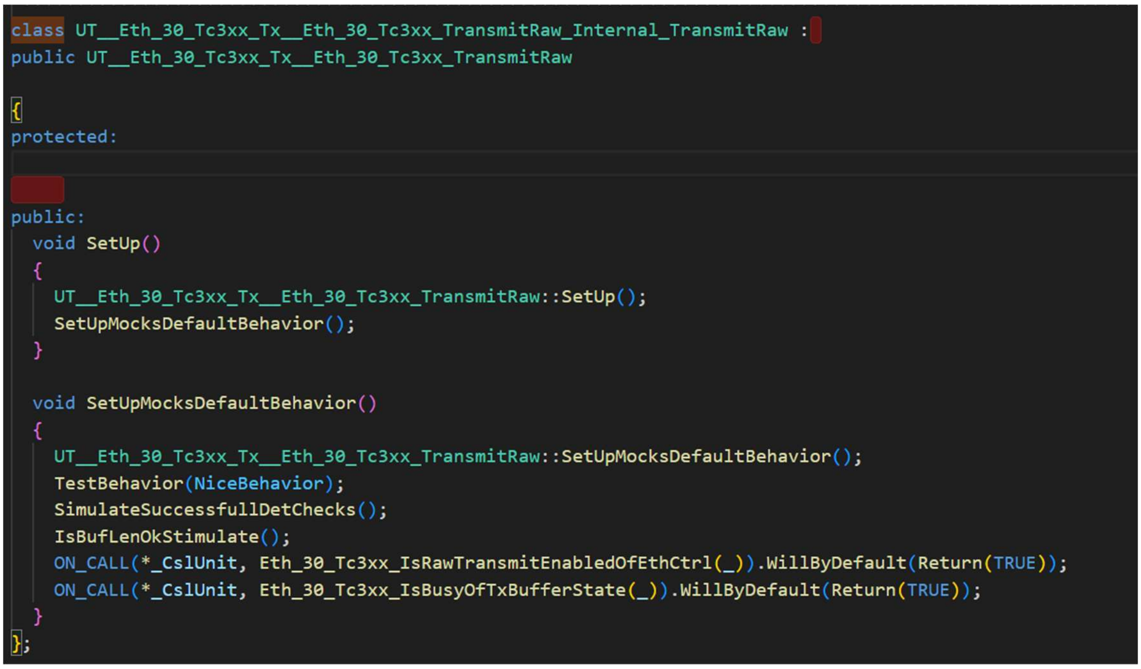
4.3.3 My soultion

My solution to this problem was to create a fixture for each internal C function within a public API that isn't tested in another CPP file, then chain my setup function in the order that these internal functions are called in the public API, ensuring that access to the internal function is the fixture is ensured.


EnvDefaultFixture
SetUp()
TransmitRaw
EnvDefaultFixture :SetUp() -> SetUp()
Internal_TransmitRaw
EnvDefaultFixture :SetUp() -> SetUp()
Internal_IsBufLenOk
EnvDefaultFixture :SetUp() -> SetUp()
GetRelatedTxBuffer
Internal_TransmitRaw:SetUp() -> SetUp()
Eth_30_Tc3xx_GetRelatedTxDescrIdx
Internal_TransmitRaw:SetUp() -> SetUp()

Part of a dummy Uml diagram for the TransmitRaw API

Another advantage of this approach is that when a small or medium change is made to the code, only the Fixture functions and test cases where the code has changed need to be changed, as well as the Setup function of the subsequent Fixture.Also, only variables or helper functions that are needed within the scope of the internal functions can be used in the Fixture whose internal function is being tested, reducing the bloat of a large Fixture. This test structure has also been discussed with and approved by our senior safety coach from our head department.



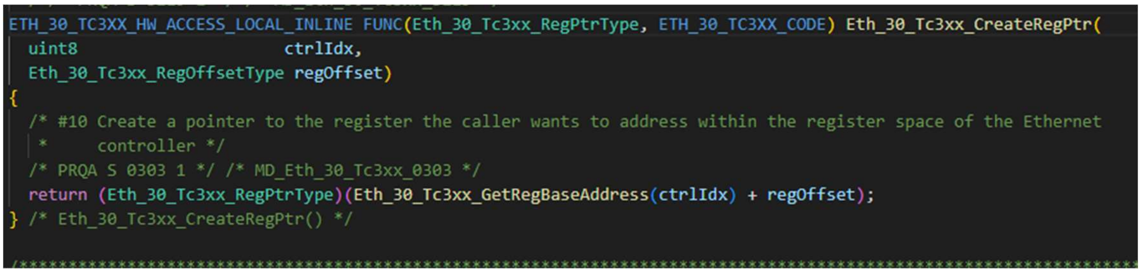
```
class UT__Eth_30_Tc3xx_Tx__Eth_30_Tc3xx_TransmitRaw_Internal_TransmitRaw :
public UT__Eth_30_Tc3xx_Tx__Eth_30_Tc3xx_TransmitRaw

{
protected:


public:
  void SetUp()
  {
    UT__Eth_30_Tc3xx_Tx__Eth_30_Tc3xx_TransmitRaw::SetUp();
    SetUpMocksDefaultBehavior();
  }

  void SetUpMocksDefaultBehavior()
  {
    UT__Eth_30_Tc3xx_Tx__Eth_30_Tc3xx_TransmitRaw::SetUpMocksDefaultBehavior();
    TestBehavior(NiceBehavior);
    SimulateSuccessfullDetChecks();
    IsBufLenOkStimulate();
    ON_CALL(*_CslUnit, Eth_30_Tc3xx_IsRawTransmitEnabledOfEthCtrl(_)).WillByDefault(Return(TRUE));
    ON_CALL(*_CslUnit, Eth_30_Tc3xx_IsBusyOfTxBufferState(_)).WillByDefault(Return(TRUE));
  }
};
```

Example Fixture for Internal_ TransmirRaw

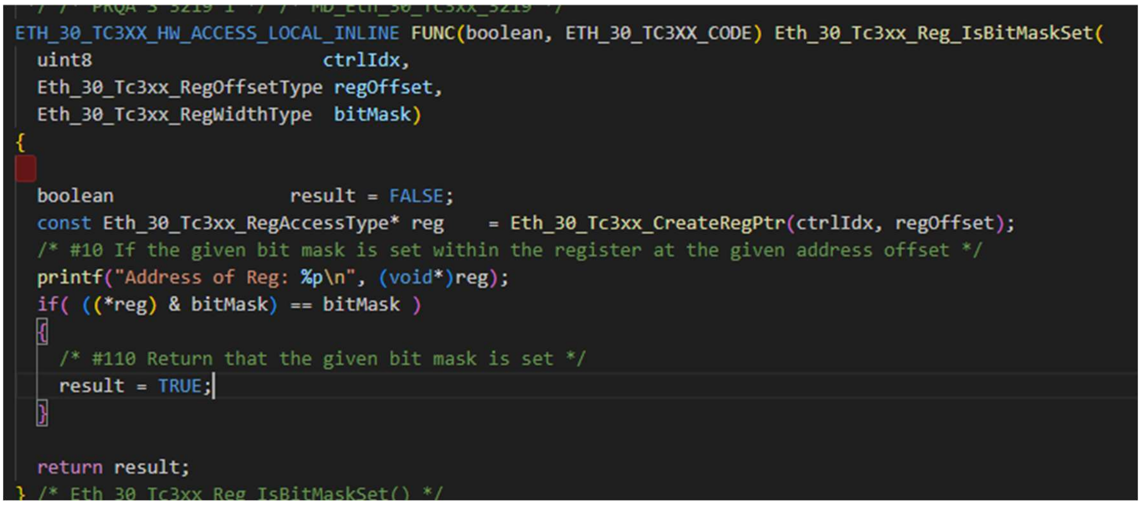
4.4 32 Bit Compiler Support in the Toolchain

Another task that arose while working on this project was troubleshooting problems within the hardware access unit. Initially I inherited this unit from a student who had left the company.We ran into a problem where most tests resulted in segmentation faults, but only when the test plan was run. However, in our Visual Studio Code working environment, we didn't get any segmentation faults.

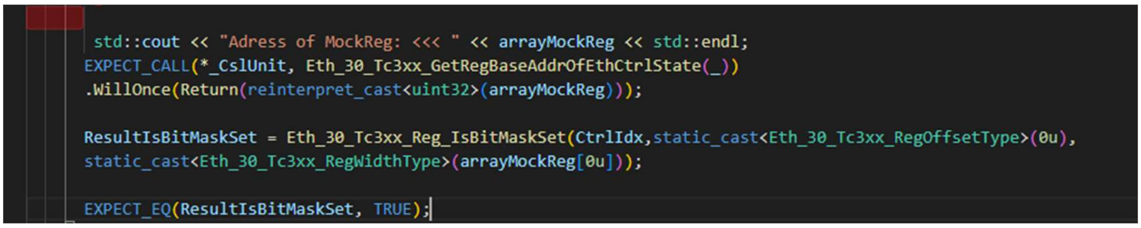
To address this issue, I isolated a specific function. Segmentation faults typically result from invalid memory accesses. However, debugging in the test plan proved challenging as everything is built in a release mode. So I isolated the executable for this unit in the temporary build file. By running this executable from the terminal, I was able to print out all the addresses that I had mocked and provided in my tests, as well as addresses accessed in the C code using cout and printf.



```
ETH_30_TC3XX_HW_ACCESS_LOCAL_INLINE FUNC(Eth_30_Tc3xx_RegPtrType, ETH_30_TC3XX_CODE) Eth_30_Tc3xx_CreateRegPtr(
  uint8                      ctrlIdx,
  Eth_30_Tc3xx_RegOffsetType regOffset)
{
  /* #10 Create a pointer to the register the caller wants to address within the register space of the Ethernet
   *     controller */
  /* PRQA S 0303 1 */ /* MD_Eth_30_Tc3xx_0303 */
  return (Eth_30_Tc3xx_RegPtrType)(Eth_30_Tc3xx_GetRegBaseAddress(ctrlIdx) + regOffset);
} /* Eth_30_Tc3xx_CreateRegPtr() */

/*****************************************************************************************************************
```



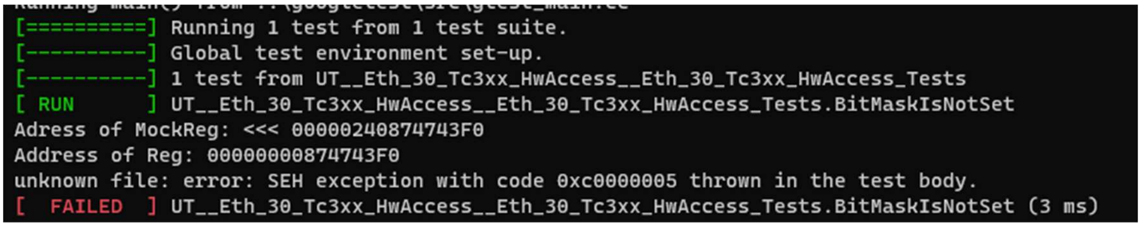
```
ETH_30_TC3XX_HW_ACCESS_LOCAL_INLINE FUNC(boolean, ETH_30_TC3XX_CODE) Eth_30_Tc3xx_Reg_IsBitMaskSet(
  uint8                      ctrlIdx,
  Eth_30_Tc3xx_RegOffsetType regOffset,
  Eth_30_Tc3xx_RegWidthType  bitMask)
{

  boolean                    result = FALSE;
  const Eth_30_Tc3xx_RegAccessType* reg    = Eth_30_Tc3xx_CreateRegPtr(ctrlIdx, regOffset);
  /* #10 If the given bit mask is set within the register at the given address offset */
  printf("Address of Reg: %p\n", (void*)reg);
  if( ((*reg) & bitMask) == bitMask )
  {
    /* #110 Return that the given bit mask is set */
    result = TRUE;
  }

  return result;
} /* Eth_30_Tc3xx_Reg_IsBitMaskSet() */
```



```
  std::cout << "Adress of MockReg: <<< " << arrayMockReg << std::endl;
 EXPECT_CALL(*_CslUnit, Eth_30_Tc3xx_GetRegBaseAddrOfEthCtrlState(_))
 .WillOnce(Return(reinterpret_cast<uint32>(arrayMockReg)));

 ResultIsBitMaskSet = Eth_30_Tc3xx_Reg_IsBitMaskSet(CtrlIdx,static_cast<Eth_30_Tc3xx_RegOffsetType>(0u),
 static_cast<Eth_30_Tc3xx_RegWidthType>(arrayMockReg[0u]));

 EXPECT_EQ(ResultIsBitMaskSet, TRUE);
```

It was through this process that I realised that the problem was due to the difference in the compiler. The compiler used in the test plan was the MSVC 64-bit compiler. Our current code was only x86 compliant. In this particular case, a 64-bit long address was being cast into a 32-bit long variable in the C code. This was causing the segmentation faults.

To understand this problem better, Google Test, and in particular Google Mock, is extremely useful in our department. This is because the C code we are testing runs on a microcontroller. We can create mock registers as arrays to test something that would normally be done in registers, but without accessing actual hardware addresses. This allows the C code we are testing to access valid memory, which facilitates effective testing in our environment.



```
Running main() from ..\googletest\src\gtest_main.cc
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from UT__Eth_30_Tc3xx_HwAccess__Eth_30_Tc3xx_HwAccess_Tests
[ RUN      ] UT__Eth_30_Tc3xx_HwAccess__Eth_30_Tc3xx_HwAccess_Tests.BitMaskIsNotSet
Adress of MockReg: <<< 00000240874743F0
Address of Reg: 00000000874743F0
unknown file: error: SEH exception with code 0xc0000005 thrown in the test body.
[  FAILED  ] UT__Eth_30_Tc3xx_HwAccess__Eth_30_Tc3xx_HwAccess_Tests.BitMaskIsNotSet (3 ms)
```

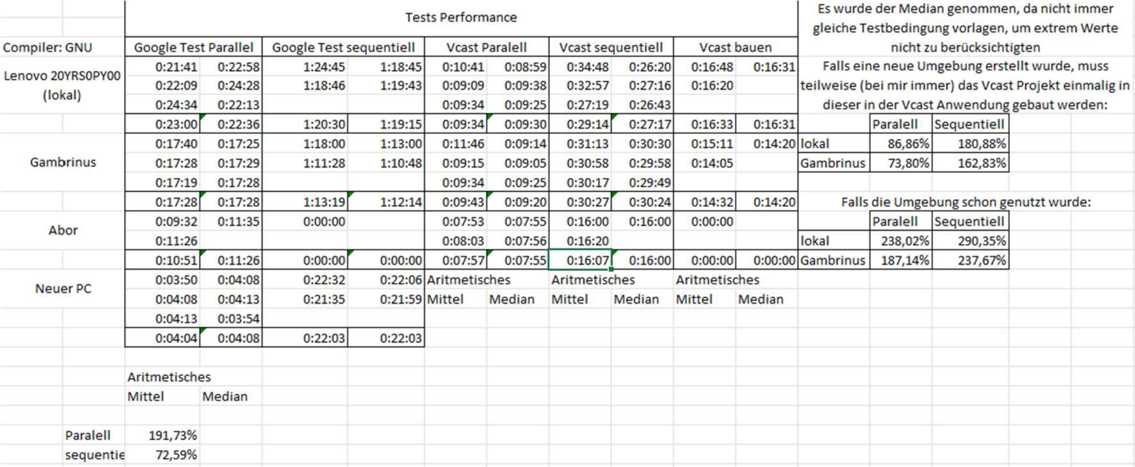
To solve this problem, I created a ticket to the CDK team to add a feature that would allow us to use the x86 compiler platform as an option in the test plan.

4.5 Performance Tests

Another of my tasks was to test the performance between the Google Test Framework and the VectorCAST framework. I measured the time taken to complete the build and run the code coverage for the entire component. I then calculated the arithmetic mean square and median of these times, from which I could calculate the percentage performance of the Google Test Framework compared to VectorCAST.

To build and run the tests, I used two methods. The first was a sequential mode where variants were built sequentially on a single core of the CPU. At the time, this was considered to be the safer and standard method. The second method was to build and run each variant in parallel. This used more cores per CPU. However, this feature was still in beta. We couldn't fully rely on it at the time.

After the comparison of VectorCAST and Google Test, we obtained new remote PCs and compared the performance of the new remote PCs in terms of code coverage execution against our old remote PCs. Although the single clock speed of a core in the CDU of the new PCs was higher than that of the old PCs, the performance of the sequential method was worse on the new PCs than on the old PCs. We presented this result to our internal IT department and they investigated it further.



Tests Performance

Compiler: GNU	Google Test Parallel		Google Test sequentiell		Vcast Paralell		Vcast sequentiell		Vcast bauen	
Lenovo 20YRS0PY00 (lokal)	0:21:41	0:22:58	1:24:45	1:18:45	0:10:41	0:08:59	0:34:48	0:26:20	0:16:48	0:16:31
	0:22:09	0:24:28	1:18:46	1:19:43	0:09:09	0:09:38	0:32:57	0:27:16	0:16:20	
	0:24:34	0:22:13			0:09:34	0:09:25	0:27:19	0:26:43		
	0:23:00	0:22:36	1:20:30	1:19:15	0:09:34	0:09:30	0:29:14	0:27:17	0:16:33	0:16:31
Gambrinus	0:17:40	0:17:25	1:18:00	1:13:00	0:11:46	0:09:14	0:31:13	0:30:30	0:15:11	0:14:20
	0:17:28	0:17:29	1:11:28	1:10:48	0:09:15	0:09:05	0:30:58	0:29:58	0:14:05	
	0:17:19	0:17:28			0:09:34	0:09:25	0:30:17	0:29:49		
	0:17:28	0:17:28	1:13:19	1:12:14	0:09:43	0:09:20	0:30:27	0:30:24	0:14:32	0:14:20
Abor	0:09:32	0:11:35	0:00:00		0:07:53	0:07:55	0:16:00	0:16:00	0:00:00	
	0:11:26				0:08:03	0:07:56	0:16:20			
	0:10:51	0:11:26	0:00:00	0:00:00	0:07:57	0:07:55	0:16:07	0:16:00	0:00:00	0:00:00
Neuer PC	0:03:50	0:04:08	0:22:32	0:22:06	Aritmetisches		Aritmetisches		Aritmetisches	
	0:04:08	0:04:13	0:21:35	0:21:59	Mittel	Median	Mittel	Median	Mittel	Median
	0:04:13	0:03:54								
	0:04:04	0:04:08	0:22:03	0:22:03						
	Aritmetisches									
	Mittel	Median								
Paralell	191,73%									
sequentie	72,59%									

Es wurde der Median genommen, da nicht immer gleiche Testbedingung vorlagen, um extrem Werte nicht zu berücksichtigten

Falls eine neue Umgebung erstellt wurde, muss teilweise (bei mir immer) das Vcast Projekt einmalig in dieser in der Vcast Anwendung gebaut werden:

	Paralell	Sequentiell
lokal	86,86%	180,88%
Gambrinus	73,80%	162,83%

Falls die Umgebung schon genutzt wurde:

	Paralell	Sequentiell
lokal	238,02%	290,35%
Gambrinus	187,14%	237,67%

Excel Spread sheet of my Google Test and VectorCAST comparison.

Another benefit of this task was that I had to set up our testing environment on a lot of fresh PCs, so I streamlined the process for the minimal setup for our testing environment. This was followed by documentation on our department's internal wiki.

4.6 Project plan

Our component for the Infineon TC3XX consists of two main parts: the core and the lower layer. The core contains the logic that is consistent across different microcontrollers, while the lower layer contains the register abstraction and logic that is unique to each microcontroller. In its current design, the core is only able to run in conjunction with a valid lower layer.

When I joined the team there were 13 units, seven of which were complete at the time, these units came from the core which had a total of 12 units. All the code for the lower layer was contained in a lower layer unit. During my time on the project, however, this structure changed. We started splitting the lower layer unit into corresponding core units, making 22 units. By the time I left the project, 20 units had been completed. There were only two unfinished units, both of which were part of the core.

The plan for the project after my departure is the completion of the two unfinished units in the core. Then they will move on to a new controller, such as the Marvel Fiver, for example. For a new controller, they can reuse all the tests that were written for the core. But they will have to write new test cases for the lower layer units.

5 Summary and Outlook

This project was originally started by students a year and a half ago. At the time, it was uncertain whether Google Test was superior to Vector Cast. However, Google Test proved to be the better testing framework for our application. Currently, the first tests for the core units have been deployed on our production system.

Looking ahead, this project will continue for many years to come, with at least four more microcontrollers to migrate to Google Test. In retrospect, the team probably didn't expect to make this much progress by the end of my internship, in terms of total units completed.

The impact of parameterised testing on unit testing is probably minimal, but could be significant for component testing. Unfortunately, my TX unit test structure wasn't able to be thoroughly tested during my placement, as we were forced to stop test case development due to a major rewrite of the entire unit in the main team, so we had to move to other units.

The feedback on the test structure has been positive, but it needs to be put into practice to see if it has the desired impact. At the moment, the impact may not be significant, but it is laying a good foundation for the next batch of students. In retrospect, it was fortunate that we discovered the 32-bit problem in the hardware access unit because it would have been a challenge in the implementation of the lower layer units.

Bibliography

[1] Wikipedia: Vector Informatik https://de.wikipedia.org/wiki/Vector_Informatik Stand:11.03..2024

[2] Instruction Sessions on context of my mentoring by tao chen

[3] Wikipedia: Google Test https://en.wikipedia.org/wiki/Google_Test
Stand:11.03..2024

[4] GoogleTest User's Guide: https://google.github.io/googletest/:11.03..2024

[5] Software Engineering 2 Script Stand SS 23

[6] Work Guideline Component Test – based on Test.2020
https://intranet.vg.vector.int/GlobalContent/Processes/Process%20Documents/Development%20Embedded/Development%20Embedded%20Product/WG_Component_Test_2020.pdf#search=WG%5FComponent%5FTest Stand 11.03.2024

[7] Clean Code: A Handbook of Agile Software Craftsmanship, Robert C.Martin Publication: 1. August 2008

List of abbreviations

ECU: Electronic Control Unit
CDK: Component Development Kit
TX: Transmit
CDD: Component Detailed Design
API: Application Programming Interface
CPU: Central Processing Unit