Journal of Theoretical and Applied Information Technology
31st January 2013. Vol. 47 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

RESEARCH ON CAN BUS AUTOMATED TEST ENVIRONMENT

FENG LUO, CHU LIU
Clean Energy Automotive Engineering Center, College of Automotive Engineering,
Tongji University, Shanghai 201804, China, +86-021-69583892
E-mail: luo_feng@tongji.edu.cn , liuchu1985@126.com

ABSTRACT

Test and validation are so important that each type of automotive ECU (Electronic Control Unit) should be qualified before entering the market. In this article, a fully scaled and customized automotive CAN Bus test environment is introduced, including a test hardware, a driver and its software platform, in which a test kernel is implemented, which is able to parse CAN databases and automatically executes test cases generated by the test case generator. Programming for a specific test case is also supported. Log files are created during the test execution to log test steps as well as CAN Bus messages, which are used by the test report generator to generate the final test report. Problems are detected during the whole test process and reproduced later in the laboratory environment, which enhances the debugging process. With the help of the testing environment, the quality assurance of the ECU can be better achieved.
Keywords: *CAN Bus, Automated Testing, Database Engine, Test Cases*

1. INTRODUCTION

With the increasing demands on automotive CAN Bus application, problems regarding the stability and reliability of in-vehicle software should be considered [1]. According to the report published by ADAC (Germany and Europe's largest automobile club), the primary cause of automotive breakdown (41.2%) was directly due to electronics [2]. Nowadays most developments of automotive ECUs are model based, such as traditional V-model development [3] or AUTOSAR [4]. The internal behaviors of the software-components are required to be tested during the whole development process, from the initial simulation, HIL (Hardware in the Loop) simulation, to the final validation. These tests are typically performed on a PC [5]. However, to build such test equipment may encounter problems due to the complexity of the system and the performance limit of the computer. The objective of this study is to introduce a solution for the testing of CAN network devices based on PC with high efficiency, the steps and components required to build up such a test environment are discussed.

A test can be performed by two methods: the "White Box" method and the "Black Box" method; while "White Box" method requires a connection to the ECU with debugging tools, which is used mainly by the developer of the ECU, thus bugs and malfunctions are solved at an early age during the development. Compared with "White Box" method, "Black Box" is also indispensable; the behaviors of the DUT such as the functionality, response and timing are checked. The test environment discussed in this paper is built based on the "Black Box" method.

The test environment involves two parts: the hardware communicating with the ECU to be tested (DUT, the device under test) on the CAN Bus; and the test tool (Tester) running on Windows for test case creation, test sequence execution and report generation. Windows platform is adopted due to its wide-spread usage all over the world and rich development tools support.

The flow of the upcoming sections is presented in the following manner:

- The concept of the automated testing
- Requirements to build a testing environment
- The structure of the whole test environment
- The realization of each software component
- The conclusion and validation

2. RESEARCH METHOD

2.1. Research Design of the Test Environment

2.1.1. Test development stages

The development of CAN Bus automated test environment is made up of the following four important stages, please see figure 1.

Journal of Theoretical and Applied Information Technology
31st January 2013. Vol. 47 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

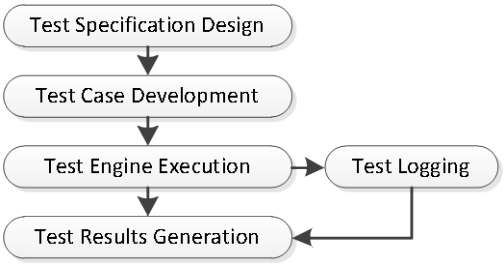

Figure 1. CAN Bus Test stages

Test specification design is the first stage, in which the ECU specific testing requirements are defined. This is usually provided by the manufacturer of the ECU.

In the second stage, test cases are developed according to the previously defined test specification, which can be generated using the graphical interface of the test case generator, or programmed by the test engineers. Various validations should be made on the test cases, so as to ensure that these test cases are fully consistent with the specifications.

The execution of the test engine requires hardware with at least two CAN channels, so that regular test including gateway test requirements is met. Test cases are read out by the test engine one after another, which generates test sequences on the CAN Bus, reactions of the DUT are compared with the corresponding pre-defined values in real-time. All related messages are stored into log files at the meantime.

The result of a test case is marked with "OK" if it passes the testing criteria; otherwise, "NOK" (not OK) is marked instead. Results are stored into an XML file during the test process, which can be used to generate the final test report.

2.1.1. Testing requirements

To ensure the ease of use of the test environment, as well as the performance and scalability, the following features are required:

2.1.1.1. Graphical user interface of the test case generator

The generator should be able to create pre-defined test cases without one line of code based on the built-in test module, take a test case named "Signal Value Range Check" for example, which automatically checks the signal value according to the min-max value defined in the CAN database, and does not require coding.

2.1.1.2. Fast reaction speed of the environment

The test is usually performed based on the hardware communication with the DUT during HIL simulation. This requires that the Tester act the same way as the real ECUs while communicating with the DUT on the CAN Bus. Delay time is critical and should be minimized during data request of the DUT, such as diagnostics session and so on. Take remote frame request of the DUT as an example, in regular CAN Bus communication, the delay time is defined in Equation 1 (considering no arbitration on the bus):

$$T_{Delay} = T_{Calc} + T_{CANTX} \quad (1)$$

Where T_{Delay} is the allowed delay time; T_{Calc} is the internal calculation delay of the ECU being requested, which can be treated as 0; T_{CANTX} is the transmit time of the requested data frame on the CAN Bus, usually 200 microseconds under 500 kbps. However, in the test environment, the ECU being requested is the Tester on a PC, so the delay time is:

$$T_{Delay} = T_{USBRX} + T_{Calc} + T_{USBTX} + T_{CANTX} \quad (2)$$

Where T_{USBRX} is the delay of receiving frame data from the Tester hardware; T_{Calc} is the internal calculation delay on the PC, which can be treated as 0; T_{USBTX} is the data transfer time from PC to the Tester hardware; and T_{CANTX} is the transmit time of the requested data frame. So the additional delay time (T_{USBRX} + T_{USBTX}) should be minimized. Experiments show that this value should be lower than 3ms so as to deal with all kinds of situations in most cases.

2.1.1.3. Interfacing capability

Users are allowed to build their own test cases based on the built-in test module using graphical user interface, however, this is not enough under certain circumstances. Take a test case named "Controller Bus Off" as an example, this requires operating on an external CAN Bus disturbance generator, along with the error handler if the DUT fails to react as expected, which cannot be realized within the internal built-in test module, because the types of CAN Bus disturbance generator may differ; the methods dealing with the failure of the DUT may differ, such as email notification and SMS message transmission.

A plugin manager of the test environment is required, which enables the third-party software developers to create customized test modules for the environment by any programming language, using the "_stdcall" calling convention. The plugin

Journal of Theoretical and Applied Information Technology
31st January 2013. Vol. 47 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

manager loads each test module plugin at startup, collects the test cases, which can be dynamically invoked during the test.

2.1.2. Test Environment Structure

The structure of the test environment is presented in Figure 2.

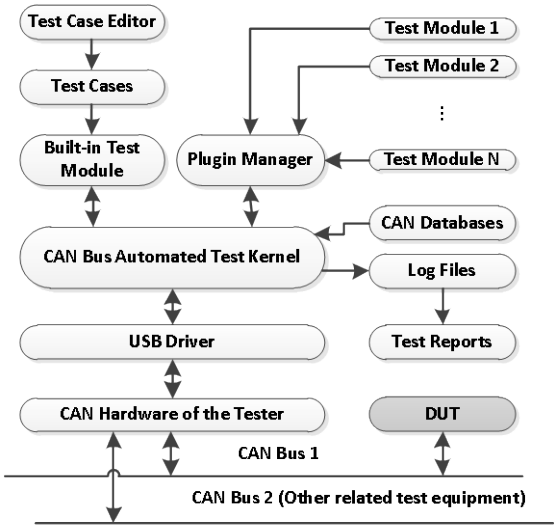

Figure 2. CAN Test Environment Structure

There are two CAN Buses connected to the Tester; "CAN Bus 1" is connected to the DUT at the same time, so that all the remaining bus nodes can be simulated by the Tester. No more real CAN nodes are allowed to be connected on "CAN Bus 1" because all the necessary messages of the DUT should be transmitted by the Tester, so as to avoid any problem introduced by other ECUs. "CAN Bus 2" is used to test gateways, or to interfere with the DUT by other related test equipment like CAN Bus disturbance generator, or relay boards.

Test cases can be either created using the "Test Case Generator", or directly created by other programming language, which are loaded into the test modules by the CAN Bus automated test kernel, and executed one by one to generate test logs during runtime and test reports at the end.

CAN databases are also loaded by the test kernel to generate symbolic names during test runtime and test logging.

2.2. The Realization of the Test Environment

2.2.1. The hardware and communication

The Tester hardware is required to operate at a higher bus speed with lower latency. The Freescale automotive controller MC9S12XEP100 is used as the main controller, with the main processor running at 50MHz frequency and co-processor running at 100MHz. SJA1000 is used in this test environment as the individual CAN Bus controller, which communicates with the DUT or other devices through CAN transceiver TJA1050.

To realize high-speed USB communication with minimum latency, Cypress USB 2.0 controller CY7C68013A is used as a FIFO to connect the microcontroller to the PC [6]. A customized USB driver is developed using Windows Driver Foundation (WDF). Bulk transfer is realized in the communication; approximately 4000 times of transfer containing 512 bytes in each packet within a second is achieved in this transfer type [6], which satisfies the requirements of the communication.

2.2.2. The Database Engine

The database engine is integrated in the test kernel, which loads the databases from local disk and parses them to build the CAN Bus symbols in the memory.

During runtime, the CAN database engine checks each CAN message's identifier, looks it up in the symbol tree to find a proper name that matches the identifier within the specified network. The flowchart of the database engine is shown in Figure 3.

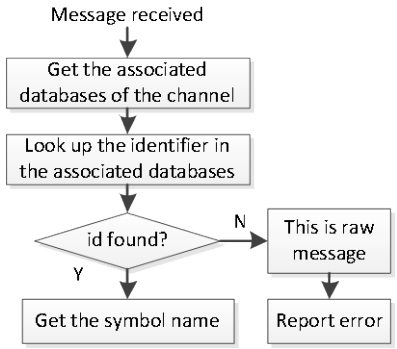

Figure 3. Database Engine in Runtime

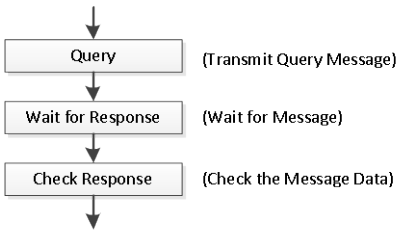

Figure 4. Basic Test Method

If the identifier received during the test is not found in the database, this could be a problem of the DUT, because it transmits the un-defined message, which is not allowed in the

Journal of Theoretical and Applied Information Technology
31st January 2013. Vol. 47 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

communication. The engine automatically reports this kind of error.

2.2.3. The test kernel

The test kernel is realized based on the "query – read – check" Black Box test method on the CAN Bus. The test kernel first sends a query message to the DUT on the CAN Bus, and wait for a certain time for the response message; if no response message received, time-out error occurred, otherwise, the response value is checked against the desired value, as shown in Figure 4.

Two high-priority threads are implemented in the test kernel: kernel timer thread and test case thread.

The kernel timer thread plays an important role in the whole test environment, which reads the received CAN messages (RX messages) from USB driver and stores them into the local message buffer. Each RX message is processed first by the "event callback interface". The interaction with the test environment is enabled through such interface; for example: message id counting, message selective logging, and additional signal checking. Callback functions can be registered by external libraries. Figure 5 shows the main steps in the kernel timer.

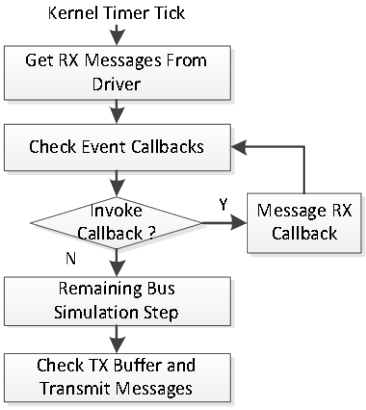

Figure 5. The Kernel Timer Thread

The kernel timer ticks at an interval of 500us under Windows system, so as to ensure the accuracy of the simulation and test. To achieve this, Windows kernel object – "Waitable Timer" is used, thus the real-time requirements of a high resolution timer is satisfied [7].

During the timer tick, an important step is to simulate all the possible ECUs' messages that are received by the DUT, which is called the "remaining bus simulation". The scheduled time for a message transmission is determined in this step. If a periodical message is scheduled to be sent, it will be first pushed into the transmit buffer. At the end of one kernel timer tick, all the messages in the transmit buffer are sent to the USB driver for transmission on the CAN Bus to the DUT.

The test case thread executes each test case in a pre-defined sequence, which runs in parallel with the kernel timer thread. A test case is divided into test steps. Each test step represents a "check" against certain CAN Bus objects such as signal value, signal range, data length. These "checks" are represented by set of APIs (application programming interface), which are implemented either in the "built-in test module", or in the "customized test modules". Figure 6 depicts the API calling during the test case execution within the thread, in which the thread first transmit messages, then get RX message from the buffer, and then invoke other kernel API functions.

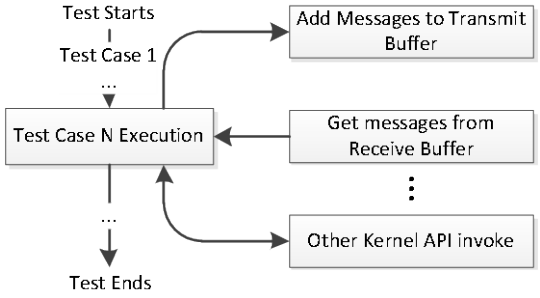

Figure 6. The Test Case Execution Thread

2.2.4. The Logging of Test Data

2.2.4.1. The Logging of Test Steps

In order to build the test report, all steps within a test case is required to be logged. Usually an XML file is used to store these steps. That is because further manipulation can be done on such file to form a user-friendly test step sequence.

The required fields for a node in XML file are listed below:

- Timestamp of the test step
- Name of the current operation
- Comment of the current operation

2.2.4.2. The Logging of the Message Objects

All the CAN messages transmitted or received by the DUT are required to be logged into a file in local disk for future analysis. ASCII file format is used instead of binary format, so that the log file can be opened by any tool that supports text format; in addition, based on the principle of one message per line, the message contained in the log file will be easily referenced in the final test report.

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

The required columns for a message are listed below, which describe the properties of the associated CAN message:

- Timestamp of the message in microsecond
- Channel index of the message
- Message direction (TX or RX)
- Message identifier
- Message type
- Message data length (DLC)
- Message data bytes

In the test environment, an individual logging thread is implemented and running in background, which maintains a message buffer with messages being added from the test kernel. Messages are stored in the disk when certain amount of messages is filled in the buffer.

Besides, descriptions of important operations are logged into the same log file in the meantime. By means of log file, rich information can be provided in the test report.

2.2.5. Test Report Generation

The test step log file is stored in XML file format, which concentrates on the structure of the information in a file and not its appearance. To display the test result in a more readable file format, these XML files should be formatted. In practice, XSLT style sheet is used to transform the XML test step log file into the HTML or PDF file.

An XSLT style sheet is created especially for the test environment, during the transformation, the test step log XML file and message log file are combined into the destination HTML file format. Hyperlinks are added for each test step so that the logged messages with timestamp can be view with ease.

2.2.6. The Design of the Test Case Generator

2.2.6.1. Test Case Generator GUI

Graphical User Interface (GUI) enables the user to create test cases without programming. With the interaction of the GUI, test steps are constructed in a pre-defined order; each test step has its own properties like “step name”, “delay time”, “timeout value” and so on, which can be modified on the GUI. The structure of the GUI is in Figure 7.

All the built-in functions are implemented in the “Test Checks Repository”, which are added dynamically into the “Test Sequence Editor”. CAN database symbols are parsed to generate names for signals and messages, and listed in the editor for selection.

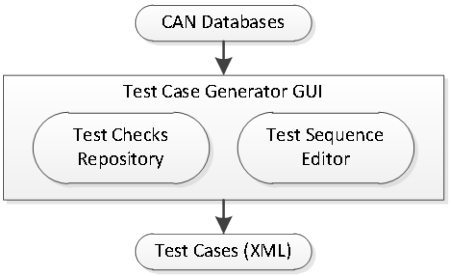

Figure 7. Structure of the Test Case Editor GUI

The configuration is saved into an XML file, which can be loaded by the test kernel for automated test, or by the GUI to edit.

2.2.6.2. Programming Support

External test cases are implemented inside a dynamic link library (DLL), which are loaded by the test kernel and treated as an individual test module.

To achieve the DLL development of external test module, a header file of the test environment is provided for each kind of DLL. Basic system functions in the environment can be invoked by the user so as to perform operations like message transmission and reception, database symbol lookup and so on.

During the initialization of each library, test cases are registered into the test kernel; this is done by an API call named “Register_Testcases”. After the process of registration, all the test cases are visible in the test environment.

3. RESULTS AND ANALYSIS

The study presents a solution for the realization of automated testing of CAN devices on the PC. Experiments are performed to validate that the test environment is able to handle the test case creation, test case execution and test report generation with high efficiency. The complexity and performance issues for building such a test environment are solved.

3.1. The Efficiency of the Test Environment

The test environment is built with the hardware, the test kernel, the test case generator and the test report generator; all the software components are realized in a modular manner. To test a specific ECU, the test case can be created by the test case generator, and is automatically loaded by the test kernel to generate the test report.

Users are only required to create or modify each test case so as to perform the test. The test report can be used by the user to detect the problems or

Journal of Theoretical and Applied Information Technology
31st January 2013. Vol. 47 No.3

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

malfunctions of the DUT. Thus the complexity to build or maintain the test environment is minimized, and the problems can be automatically identified at an early stage during the development.

3.2. The Performance of the Test Kernel

A test is executed to validate the performance of the test environment: A tool is programmed to send a remote frame every one millisecond; a response data frame within a test case of the environment will be sent back when the remote frame is received. The delay time ($T_{USBRX} + T_{Calc} + T_{USBTX}$) taken for reaction by the test environment is measured on an oscilloscope, in Figure 8.

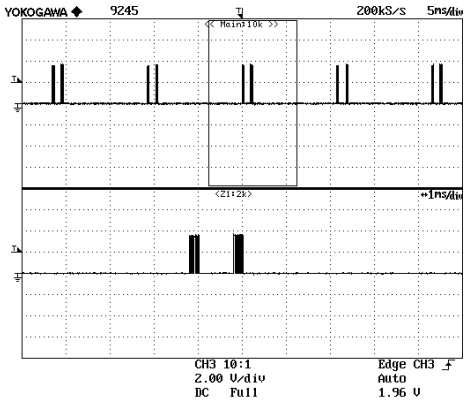

Figure 8. Response Delay of the Test Environment

From the experiment we can conclude that the response delay time of the test environment is around 800 ~ 1000 microseconds, which is fast enough for ECU tests in most cases.

4. CONCLUSION AND FUTURE WORK

An automated test environment is built for ECU testing on CAN Bus, along with a test case generator and logging & reporting feature. With the test cases and databases provided, the test environment is able to configure the testing sequences and perform the test steps automatically with a single click on the user interface. The files logged during test can be generated to form the test reports in HTML or PDF file formats. The requirements of an ECU testing are satisfied by the test environment, which greatly enhances the testing efficiency of the automotive CAN network.

Since the test environment is built on the PC, the limitation is that the reaction speed of the environment cannot be further decreased, according to the response delay in Figure 8, the time 800 ~ 1000 microseconds is wasted during the data exchange between the PC and the hardware. The goal for the future is to implement a script engine in the hardware of the test environment, which parses and executes the test scripts pre-compiled by the PC, so as to achieve instant reaction to the CAN messages and other events during the test.

Another improvement is to automate the test kernel of the test environment using Windows COM Object, with the automation interface of the test kernel registered in the Windows system, the test kernel can be utilized by external programs such as VB Scripts, command line scripts to achieve flexible scheduling of each test case.

REFERENCES

[1] Transportation Research Board. the safety Promise and challenge of Automotive electronics. Washington, D.C.: National Research Council of the National Academies. 2012.

[2] Allgemeiner Deutscher Automobil-Club. ADAC Pannenstatistik 2011. München: ADAC Fahrzeugtechnik. 2012

[3] Dr. Richard Turner. Toward Agile Systems Engineering Processes. CrossTalk the Journal of Defense Software Engineering. 2007; 11-15 (April 2007).

[4] Guido Sandmann, Richard Thompson. Development of AUTOSAR Software Components within Model-Based Design. Michigan. 2008 SAE World Congress. 2008; SAE Paper 2008-01-0383.

[5] Nigel Tracey, Ulrich Lefarth, Hans-Jörg Wolff, Ulrich Freund. ECU Software Module Development Process Changes in AUTOSAR. Stuttgart: ETAS GmbH. 2007.

[6] Feng Luo, Chu Liu. Implementation of CAN bus real-time simulation kernel based on windows platform. Advances in Computer, Communication, Control and Automation. Shanghai. 2012; 121: 61-68

[7] Johannes Petrus Grobler. Design and Implementation of a High Resolution Soft Real-Time Timer. Thesis. Pretoria: University of Pretoria; 2006