Enhancement Proposal

EECS 4314: Advanced Software Engineering
Dr. Zhen Ming (Jack) Jiang
Assignment 3 Report
March 24rd, 2023

Group Sudo:
Michael Koiku
Swapnilkumar Parmar
Henry Gu
Amin Mohammadi
Omar BaGunaid
Michelangelo Granato
Hieu (Aaron) Le
Vikramjeet Gill
Jai Ramotar

Abstract

This report proposes adding support for the Controller Area Network (CAN) bus protocol to the FreeBSD kernel. This enhancement would provide significant benefits for FreeBSD users by enabling communication with CAN bus devices commonly used in automotive, industrial, and other embedded applications. The report gives a high-level overview of the effects of these changes and discusses the necessary alterations to the kernel architecture, including adjustments to drivers, interfaces, and components. It also draws attention to the potential drawbacks of the suggested changes, such as potential performance hitches and compatibility problems with old drivers. A sequence diagram is included to illustrate the flow of a typical use case through the proposed architectural modifications. Overall, the report concludes that the benefits of adding support for the CAN bus protocol to the FreeBSD kernel outweigh the potential risks and will greatly enhance the usability of FreeBSD in embedded applications.

Introduction

The Controller Area Network (CAN) bus is a common communication protocol used in embedded systems and industries to communicate data between devices. It was developed in the 1980s to meet the need for a reliable and effective method of data transmission between microcontrollers in vehicles. The CAN bus is utilized in various applications today, from medical devices to industrial automation, and has established itself as a standard in the industry. By using a single pair of wires, the CAN bus protocol enables several devices to interact with one another on a single bus.

Controller Area Network (CAN) bus support for FreeBSD is a vital enhancement that allows FreeBSD to interface with CAN bus devices and opens up a wider range of applications, especially in the automotive and industrial automation industries. Modern embedded systems must have the CAN bus protocol since it is so commonly utilized in embedded systems. Without built-in support for the CAN bus protocol, FreeBSD is not a viable option for applications that need communication with CAN bus controllers. The FreeBSD CAN bus implementation will feature a CAN bus subsystem that supports different CAN controller hardware and offers socket-based communication and direct access to the CAN controller hardware. Applications will be able to interact with the CAN bus subsystem thanks to the proposed CAN bus, which will increase FreeBSD's value to users and developers who need to communicate with CAN bus devices.

The current state of the system relative to the enhancement

The CAN bus protocol was not integrated into the FreeBSD operating system. Users that needed to communicate with CAN bus devices were forced to create their own drivers or rely on third parties to do so. For individuals who are unfamiliar with device driver development, this could be time-consuming and even expensive.

FreeBSD was not an appropriate choice for applications that needed to communicate with CAN bus devices since it did not come with built-in support for the CAN bus protocol. Users have to employ specialized real-time operating systems that were created expressly for the automotive or industrial automation industries or choose other operating systems with built-in support for the CAN bus protocol, such as Linux or Windows Embedded.

The value & benefit of adding CAN Bus Support

The addition of CAN bus support to the FreeBSD kernel would provide several benefits:

Reliability: CAN bus is designed to be extremely reliable, using both hardware and software techniques to prevent errors and correct them if they occur. CAN bus is also designed to continue operating even under various fault conditions. By supporting CAN bus, FreeBSD can be used in applications that require extremely reliable communication.

Flexibility and Modularity: The CAN bus protocol is a message-based communication protocol that allows nodes to communicate without identifying information, enabling easy hot-plugging of nodes without requiring any software or hardware updates on the system. This results in a modular system that can be easily modified to meet specific requirements, and new electronic devices can be easily integrated into the CAN bus network without significant programming overhead. Therefore, if adopted by FreeBSD, the CAN bus protocol can provide the benefit of flexibility in terms of easy integration and modification of electronic devices.

Low Cost: The CAN bus protocol reduces the cost of wiring and labour by allowing all electronic systems, actuators, and sensors in a vehicle to connect via a single wire, which enables transmission over a shared medium. The low cost of implementation is one of the primary reasons for the widespread adoption of the CAN bus protocol. Adding CAN bus support to FreeBSD can provide similar benefits, enabling faster communication between electronic devices while reducing the cost of wiring and labour.

Compatibility/Integration: The CAN bus protocol is a well-established standard in the automotive and industrial automation sectors. Even in the future, it is expected to be the preferred networking technology for connecting electronic devices that require simple and frequent communication. Integrating CAN bus support into FreeBSD would simplify the development of custom solutions and applications that can be seamlessly integrated into existing systems. As a result, this would make FreeBSD a more appealing choice for both developers and users in these industries.

The changes required in the current architecture to support this enhancement

Minimal changes to the existing FreeBSD architecture are required. FreeBSD's network stack is highly modular and extensible. Our proposed architecture for CAN bus support is heavily

inspired by the SocketCAN architecture on Linux. While our proposal does not consider this, it may be possible for Linux binaries running on FreeBSD to take advantage of our CAN bus implementation since it is similar to the Linux implementation; this will require some but minimal translation layers.

FreeBSD already supports a wide variety of socket types, such as TCP, UDP, Unix Domain sockets, etc. Our proposal will add a new socket type for CAN bus use, applications creating sockets using the socket() syscall can specify our new CAN bus type to create a socket.

Similar to other sockets, they must be bound to an interface that is used for communication. Our CAN bus architecture would also create a new type of network interface to represent hardware CAN bus controllers. CAN bus driver implementers would register a new network interface of the CAN bus interface type. Applications can then use the bind() syscall to attach the socket to a CAN bus interface.

From here, applications can simply use the read() and write() syscalls to read and write CAN bus frames. Frames written to a CAN bus socket would get passed through the existing network architecture to the relevant driver, which would send the frame out on the physical CAN bus. Frames received on the bus by the driver implementation would be put into the incoming packet queue for the network interface.

By making use of the existing FreeBSD networking architecture for sockets, interfaces, packet queues, etc., and simply adding new socket and interface types, we minimize the amount of code that needs to be written and tested.

The architectural styles and design patterns needed to realize this enhancement

Depending on the needs, several architectural styles and design patterns could be used to realize this enhancement. Some of these can potentially include:

Modular architecture: Adoption of a modular architecture would allow for easy integration and modification of the CAN bus stack. This would enable developers to add and remove specific features without affecting the overall system, making it more flexible and customizable.

Observer pattern: The observer pattern would allow for the separation of the CAN bus driver from the rest of the kernel, enabling the driver to observe and respond to changes in the state of the CAN bus network. Thus, it enables efficient handling of events related to the CAN bus network.

Adapter pattern: The adapter pattern could be used to implement support for different types of CAN bus hardware. This pattern would allow the kernel to interface with different hardware interfaces through a common interface, enabling support for a wide range of hardware without requiring significant changes to the kernel.

Command pattern: The command pattern could be used to implement the communication between different components of the CAN bus stack. This pattern would allow the kernel to issue commands to the different layers of the stack, enabling efficient and reliable communication between devices on the CAN bus network.

The effects of the enhancement on the system

Maintainability

Adding CAN bus support to FreeBSD can increase the complexity of the system, as it requires the development and implementation of a custom kernel module. For maintainability, the developers will have to write additional code and configure the system, and doing so, will make it difficult to understand and maintain.

Other dependencies will have to be implemented and used for the system such as third-party libraries, or tools to be installed and maintained. Each of these dependencies will need to go through maintenance, it might need updates or configuration changes, which can further increase the complexity of maintaining the system.

Most importantly, documenting the newly implemented functionality of the system after making the changes. That includes configuration changes and troubleshooting guides.

Evolvability

The possibility for the system to be upgraded and adapted at ease after implementing the CAN bus protocol can be affected. Adding CAN bus support will provide a feature which will let the system support new hardware and communicate with other systems that need to use CAN bus protocol. These new functionalities that were implemented in the system can sometimes lead to some conflicts with the existing features, requiring extra effort to fix these issues. It might also require more changes to the system and make it harder to improve in the future. So adding CAN bus protocol should be done after careful planning to confirm that it does not impact the evolvability of the system.

Testability

Adding CAN bus support can make testing more difficult as it requires the development and implementation of new custom tools to help test the system. This will increase the complexity of the testing process, not only to test the new functionality that is brought by the CAN bus protocol but also to confirm that it does not impact other components of the current system. It also can bring new bugs to the system and other issues that might be difficult to test since the behaviour of the system might get affected.

Adding CAN bus support can provide benefits, but it's important to consider the impact on testability. By taking a well-analyzed and thoughtful approach to testing, developers can decrease the impact on the system and guarantee that the system functions as needed.

Performance of the system

System performance will get affected by adding the CAN bus protocol to the FreeBSD. The impact can be negative since the implementation requires more processing time which can bring additional latency and slow down the system. The impact can be measured by a variety of aspects, for example, it could impact the implementation of the CAN bus protocol, the hardware of the system, and the software configuration of the system.

To minimize the effect on the performance of the system, the developers have to optimize the implementation of the CAN bus protocol and decrease the amount of added processing time. This requires optimizing the CAN bus drivers to make them more efficient and less resource intensive.

Risks and limitations

CAN bus support is undoubtedly a reliable communication protocol that helps simplify and speed up the data transmission process among internal units within a system. However, the said protocol also has its limitations:

Limited bandwidth and length: Since the working principle of CAN bus relies on simplicity such as prioritizing transmission of only critical data, low wiring as well as only allowing a limited number of nodes to communicate at one time, transferring large amounts of data isn’t always guaranteed. Furthermore, since the protocol is only designed to work within a single system, the wire range is very limited and therefore not suitable for intersystem connection or any long-range data transmission

Potentially high maintenance cost: Although the CAN bus itself is a simple design and thus easy to implement initially, it can be difficult to upgrade and integrate new components into the network without compromising the simplicity of the design. For this reason, the upgrade and maintenance cost might be higher than developing a new protocol suitable for different system

Security issues: Based on the architecture and working principle of the CAN bus, all nodes have connections with each other via the only bus. This means any attack attempted on 1 node will affect the remaining ones and any node is as vulnerable as the others. Additionally, communication via the CAN bus most of the time are unencrypted in order to save the overhead cost of decryption and for this reason, the protocol is vulnerable to unauthorized access and different types of cyber attacks:

Message manipulation: This type of attack can inject the transmission flow with messages that scramble priorities or cause unexpected behaviours

Eavesdropping: Due to the interconnection between nodes, if the attacker manages to gain access to one node, they could easily retrieve unencrypted info from other nodes [8]

Denial of Service (DoS) attack: Similar to message manipulation, the attacker can input malicious code that jams the CAN communication channel, thus preventing data from being transmitted

Approaches to realizing this feature

Phase-Based Approach

In a phase-based approach, each phase builds upon the previous one, with a clear set of goals and deliverables for each phase. This approach can help to ensure that the development process is structured, organized, and focused on delivering a high-quality CAN bus support feature in FreeBSD.

A phase-based approach to implementing CAN bus support in FreeBSD would involve breaking the development process into distinct phases, each with a specific set of goals and deliverables. The following are the phases involved in a phase-based approach to implementing CAN bus support in FreeBSD:

Requirements Analysis: This would involve gathering and documenting the specific functional and non-functional requirements for the CAN bus support feature, as well as any constraints or dependencies that need to be considered.

System Design: With the requirements documented, the next phase would be to design the architecture of the CAN bus support feature. This would involve designing the APIs, messaging protocols, and other software components required to support CAN bus communication in FreeBSD.

Coding: With the architecture in place, the next phase would be to develop the code for the feature. This would involve implementing the device drivers, messaging protocols, and other software components required to support CAN bus communication in FreeBSD.

Testing: Once the code has been developed, it would need to be thoroughly tested to ensure that it meets the requirements and is compatible with a range of CAN bus devices and networks. This would involve both unit testing and integration testing, as well as testing with real-world CAN bus devices and networks.

In a phase-based approach, each phase builds upon the previous one, with a clear set of goals and deliverables for each phase. This approach can help to ensure that the development

process is structured, organized, and focused on delivering a high-quality CAN bus support feature in FreeBSD.

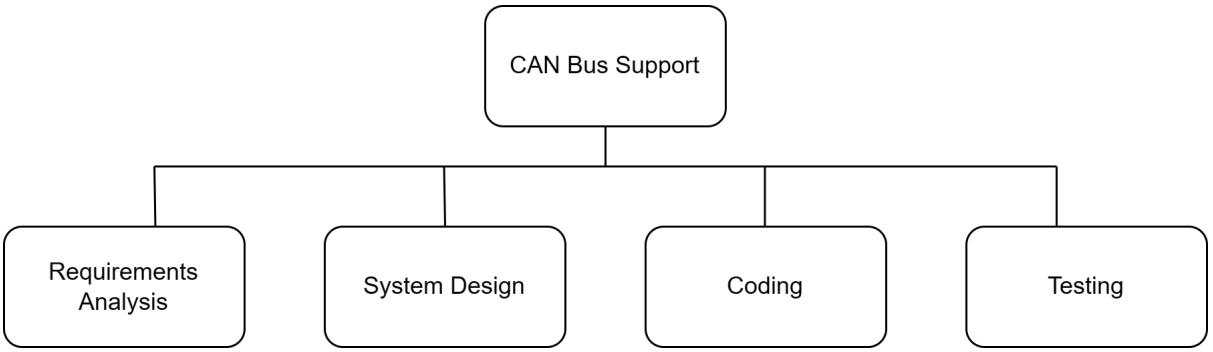

Figure 1: Phase-Based Approach to Implementing CAN bus support

Product Based Approach

To implement CAN bus support in FreeBSD using a product-based approach, several components would need to be developed. These components would include:

CAN bus controller driver: This driver would provide the interface between the CAN bus controller hardware and the FreeBSD kernel. It would need to support the specific hardware used in the target system and provide a standardized API for the rest of the system to use.

CAN bus messaging protocol: The messaging protocol defines the format and content of the messages sent over the CAN bus. The FreeBSD implementation would need to support a range of standard messaging protocols used in the industry, such as CANopen or J1939, as well as proprietary protocols used by specific devices.

CAN bus network interface: The network interface would provide the mechanism for applications and services in FreeBSD to interact with the CAN bus. It would need to provide a standardized API for sending and receiving messages, as well as support for filtering and prioritizing messages.

CAN bus configuration utility: A configuration utility would be needed to allow users to configure the CAN bus hardware and network settings. This utility would need to be user-friendly and provide a simple, intuitive interface for setting up and configuring the CAN bus.

Overall, the product-based approach to implementing CAN bus support in FreeBSD would require the development of a range of components, each with specific functionality and requirements. By approaching the development process in a structured, systematic way, it would be possible to create a robust and reliable CAN bus support feature that meets the needs of users and is compatible with a range of devices and networks.

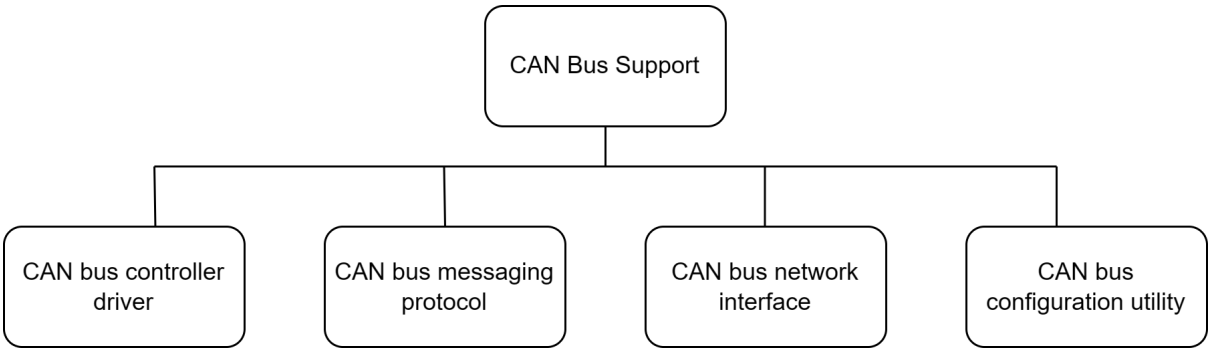

Figure 2: Product-Based Approach to Implementing CAN bus support

Uses cases, sequence diagrams, and state diagrams of this enhancement

CAN bus has many use cases in the vehicle industry, it can be applicable for automotive, aviation, boats, submarines, etc. CAN bus is widely used in the automotive industry to interconnect various electronic control units (ECU), such as engine management, transmission control, airbags, and other systems. It allows systems to communicate with each other through one centralized bus, which allows any other system to listen for or push commands. There are many applications for this: car safety features such as lane assist and collision avoidance, auto start and stop for idling, park assist, electric parking brakes, and more.

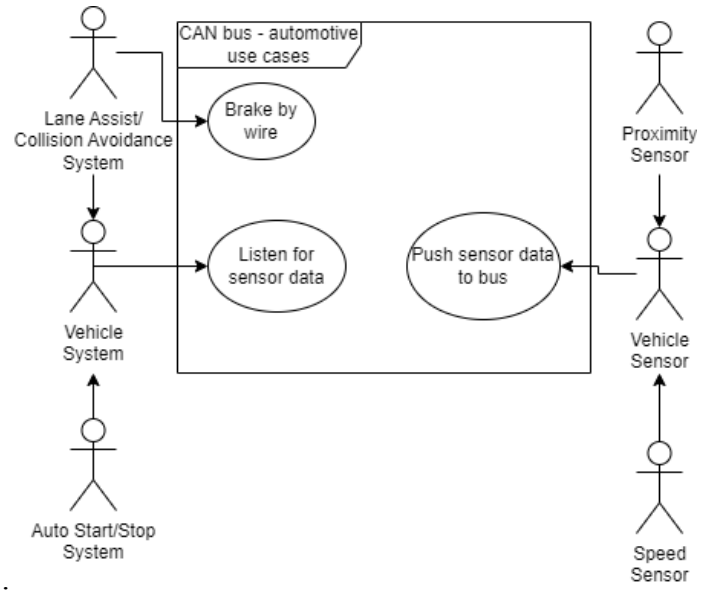

Figure 3: Automotive Use Cases

One specific use case is the collision avoidance system. For the system to make decisions on whether a collision is imminent and send the instructions to avoid it, it needs new sensor data constantly and with minimal latency. The CAN bus supports this, with all

relevant sensors (speed, proximity) sending data to the bus for the Collision avoidance system to listen for. When the computed sensor data results in a detected future collision, the avoidance system can send a brake-by-wire command through the bus, which the braking system will pick up, slowing down the vehicle. If no collision occurs and the sensor data reads normally, then the brakes can be eased the same way. This is all facilitated by the CAN bus. Many car safety features can be easily and safely facilitated using the CAN bus.

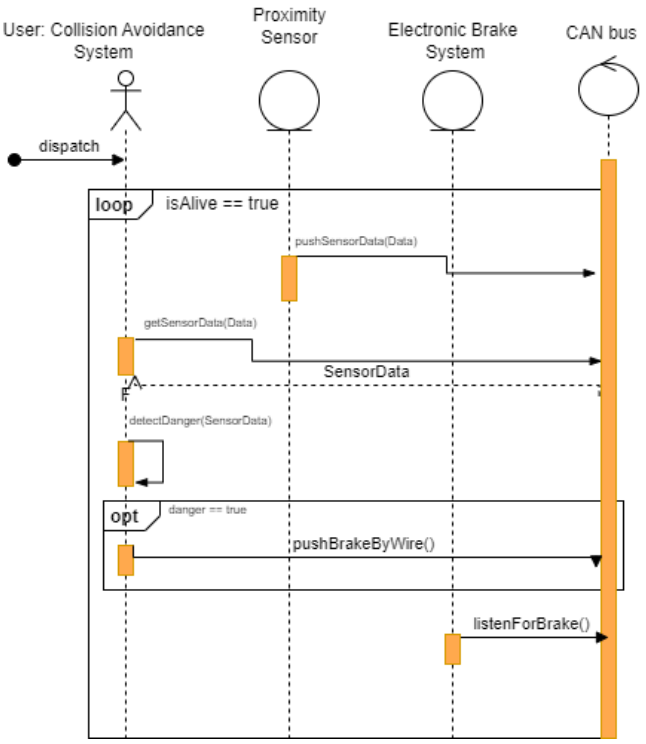

Figure 4: Collision Avoidance sequence diagram

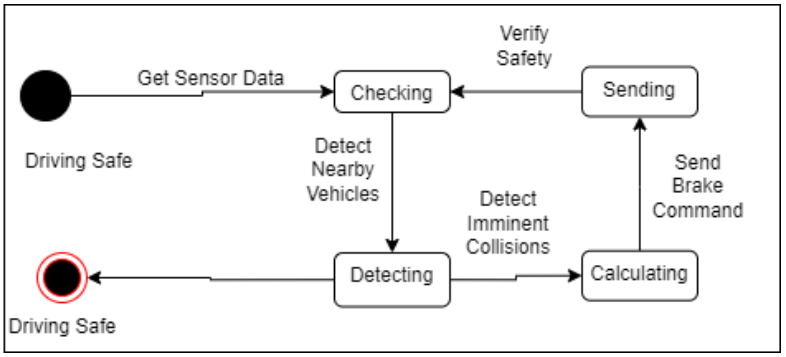

Figure 5: Collision Avoidance state diagram

Interactions with other features

One of the main ways CANbus interacts with the features of FreeBSD is through sending network problem notifications to the user. CANbus can detect errors on the physical and

MAC layer of FreeBSD via its built-in CAN controller and create an error message frame to notify the user application (the user application can request this error message frame using CAN filter mechanisms). In addition, CANbus also interacts with FreeBSD via its loopback capabilities. CANbus can perform a local loopback of the data sent on a given node after a successful FreeBSD transmission in order to ensure the correct amount of traffic is sent on that node. This is similar to the "echo" feature that is available on text telephone devices.

Plans for testing the impact of interactions between the proposed enhancement and other features

In order to test CANbus' error message frame feature, an error can be created on the MAC layer to test if CANbus delivers an effective error message frame that can allow the user to diagnose the problem with enough information to take the first steps to solve the problem. In order to test CANbus' loopback capabilities the *cansniffer* tool can be used on a single node for two identical FreeBSD applications to ensure that they both received the same information (which means the local loopback of CANbus did its job).

The effects of concurrency and team issues on the architecture

The effects of concurrency on the CAN bus protocol can be both positive and negative. On the one hand, concurrency allows for multiple nodes to transmit messages simultaneously, increasing the efficiency and speed of communication. On the other hand, when two different nodes try to concurrently send messages at the same time we can get collisions where the messages interfere with and corrupt each other which can cause delays and errors in the communication process.

The possibility of collisions necessitates using a process called arbitration which seeks to solve the problem of collisions by assigning each message a priority based on its unique identifier and whichever message has the highest priority is allowed through and the others have to wait to retry.

Conclusions & lessons learned

From our study and research, we learned that in order to implement this feature enhancement, we must understand the importance of research and planning when implementing a new feature. It is also important to understand the scope of the project, the resources available, and the potential risks associated with the project. In addition, we must also have a clear understanding of the hardware and software components involved in the project as this will help to ensure that the project is implemented correctly and without any unexpected issues.

References

[1] *Https://docs.freebsd.org/en/*. https://docs.freebsd.org/en/. (n.d.). Retrieved February 10, 2023, from https://docs.freebsd.org/

[2] McKusick, M. K. (2006). *The design and implementation of the 4.4BSD operating system*. Addison-Wesley

[3] FreeBSD. (n.d.). Retrieved from https://github.com/freebsd/freebsd-src

[4] https://www.totalphase.com/blog/2019/08/5-advantages-of-can-bus-protocol/

[5] https://www.gridconnect.com/blogs/news/can-network-protocol-advantages-disadvantages-application-examples

[6] https://ocsaly.com/can-bus-the-future-of-vehicle-communication-or-a-double-edged-sword-exploring-the-advantages-and-disadvantages/

[7] https://www.linkedin.com/pulse/security-issues-can-bus-attack-scenarios-risks-ella-mayoni

[8] http://ece-research.unm.edu/jimp/pubs/AsianHOST2017.pdf