DOKUZ EYLÜL UNIVERSITY

FACULTY OF ENGINEERING

INTERNSHIP REPORT

STUDENT'S

DEPARTMENT	**:**	**COMPUTER ENGINEERING**
NAME SURNAME	**:**	**AKİF SELİM ARSLAN**
NUMBER	**:**	**2020510009**

İZMİR

TABLE OF CONTENTS

LIST OF FIGURES

SECTION 1 INTRODUCTION

1.1 Project Description

Phinia Delphi Turkey, where I did my internship, is a leading organization in its field, developing innovative systems and components for the automotive industry. In this dynamic environment, as part of the software team, I had the opportunity to work on a project that aimed to fundamentally improve engineers' daily interactions with Electronic Control Units (ECUs). In the current situation, an engineer who wanted to develop or test on an ECU had to use a CAN-USB adapter to listen to the ECU's CAN (Controller Area Network) traffic, a separate UART-USB converter to view text-based diagnostic logs, and sometimes a different hardware interface for special configurations. This scattered structure not only slowed down the setup of test rigs, but also reduced efficiency by requiring constant switching between different software. The main mission of my project was to eliminate the need for these multiple devices and cables and combine all these critical communication functions under a single smart USB device.

For this task, I was asked to use the Cypress PSoC 5LP microcontroller. With the software I developed, I transformed the microcontroller into a sophisticated gateway that introduced itself as a Composite USB Device with multiple functions when connected to the computer. This architecture formed the basis of the project, allowing a USB socket to become a virtual function hub. In this way, I was able to activate three different and independent communication channels at the same time.

One The first of these channels was a gateway for the CAN bus, which is the main communication language of the ECU. This channel not only transmits CAN messages from the ECU to the PC, but also allows simulating other nodes or running test scenarios by writing messages sent from the PC directly to the CAN line. In the second part, I added the Bulk Transfer Protocol to the project. This protocol uses structured packets consisting of a header, command ID, data length and data fields. and by adding a CRC16 checksum to each packet, I aimed to improve communication reliability by ensuring that the data transfer is error-free. As the third and last channel, I added a CDC interface that acts as a standard virtual COM port (USBUART). This interface is especially used for receiving instant readable debug logs or sending simple text-based commands.

In order to present all these capabilities of the system in a user-friendly way, I developed a comprehensive desktop application on the PC side using the C# language and the .NET platform. By integrating the CyUSB library, I ensured that the PSoC device was automatically recognized when it was plugged into the USB and that all the interfaces it offered (Bulk, CAN, CDC) were managed correctly by the application. Engineers could now read and write the ECU's internal registers or follow the log flow from the UART terminal by sending special Bulk commands, while monitoring the CAN messages in real-time lists through this single application. In order to ensure that the user interface remained fluid even under heavy data traffic, I designed the data-intensive modules, especially CanHandler, to run in background threads. Finally, I added "echo" test modules to the application that work separately for each communication channel. These tests measured the round-trip time (latency) of a given data packet in milliseconds and the throughput in Mbps, providing concrete, quantitative data about the stability of the system. At the end of my project, I believe I have succeeded in transforming a messy and complex testbed into an efficient, powerful, and holistic engineering tool that can be managed with a single cable.

1.1.1 Week 1

The first week of my internship was spent getting to know the company's work culture and projects, and laying the foundation for the task I would be working on. On the first day, my team leader and mentor gave a detailed presentation about Phinia's current projects, especially the software development and testing processes carried out on Electronic Control Units (ECUs). During this presentation, it was explained that my internship project was to develop an integrated USB communication tool that would facilitate engineers' communication with ECUs. The outline of the project and the expected outputs from me were clearly stated. Then, the main development platform I would use in the project, Cypress PSoC Creator IDE, was introduced and its basic features were shown. The rest of the day was spent with administrative procedures such as preparing my work computer and initiating the intern authorization processes for the necessary software and hardware access.

The second and third days of the week focused on deeply understanding the USB Bulk Transfer concept, which forms the technical basis of my project. With the guidance of my mentor, I reviewed various technical documents and articles explaining how this transfer type of the USB protocol works, its differences from other transfer types (Interrupt, Isochronous), and why it is ideal for high-volume, fault-tolerant applications. In order to put this theoretical knowledge into practice, I opened an empty project on PSoC Creator and started configuring the USBFS (Full-Speed USB) component. In particular, I took the first steps of defining a Bulk IN and a Bulk OUT endpoint and how these endpoints will be managed by the PSoC software. In addition to these technical studies, I also completed the company's mandatory Occupational Health and Safety (OHS) training in this first week of my internship and learned about the work environment and safety procedures. This week was a productive start that allowed me to fully understand the requirements of the project and to start creating the necessary technical infrastructure.

1.1.2 Week 2

In the second week of my internship, I started to transform the theoretical knowledge I gained in the first week into concrete steps. My main goal this week was to establish the basic USB communication line between the PSoC and the computer. I started my work on the PSoC side and focused on the configuration of the USBFS (Full-Speed USB) component in the PSoC Creator IDE. I carefully edited the Device Descriptor and Configuration Descriptor required for the device to be recognized as a "Vendor-Specific Device". As the most critical step, I defined two endpoints, one Bulk IN (EP2) and one Bulk OUT (EP1), for the Bulk transfer that would form the backbone of my project, and set the maximum packet sizes of these endpoints to 64 bytes. After these configurations, I started writing the C code that would manage these endpoints. In the main.c file, I wrote a basic skeleton code that constantly listens to the Bulk OUT endpoint with the USB_GetEPState() function in the main for(;;) loop, reads this data into a buffer with USB_ReadOutEP() when data arrives, and for now only calls the USB_LoadInEP() function to send back the incoming data.

I In order to test this basic software that I developed on the PSoC side, I also needed a control tool on the PC side. For this purpose, I developed a simple but functional prototype interface using the C# Windows Forms platform. As can be seen in the shared screenshot, this first interface aimed to connect to the device with the "Connect" button, send data with the "Send" button, and display the communication logs in the "Logs" area. At this stage, my aim was to find answers to the questions "Can I basically connect to the device?" and "Can I send a data packet and receive a response?" rather than going into details such as CRC checks or complex packet structures. In the background of this interface, I wrote the codes that find the PSoC device with the VID/PID values that I specified, and access the Bulk endpoints that I defined, using the CyUSB.dll library provided by Cypress.

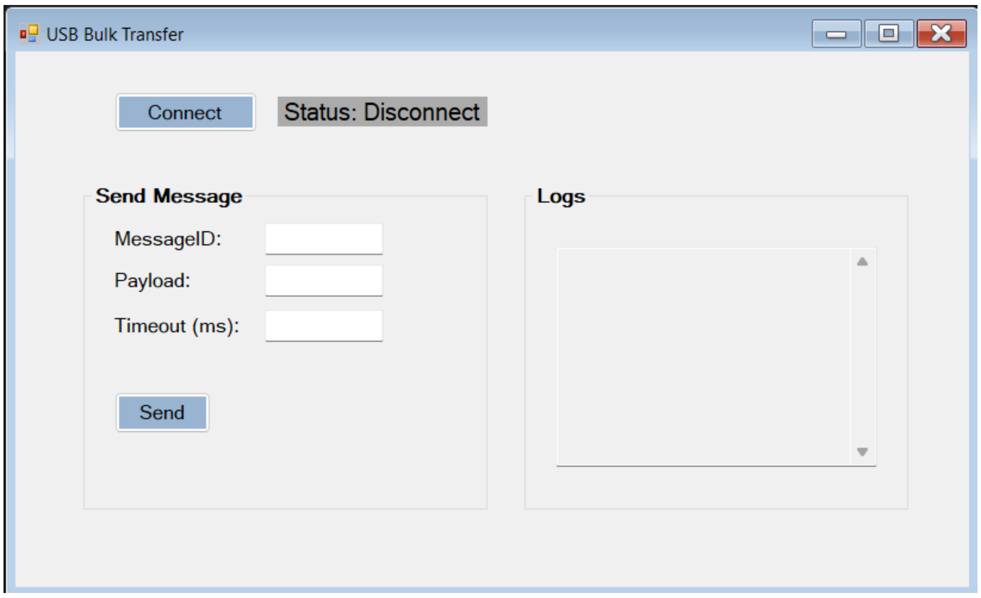


Figure 1.1 First prototype of interface

By the end of the week, the C# application I developed was not yet able to detect the PSoC device when it was plugged into the USB port. Since the device connection could not be established, when I tried to send data, the PSoC could not receive data or the response sent from the PSoC did not reach the PC. This was due to a driver problem. Windows did not recognize the prepared device.

1.1.3 Week 3

As I began my third week, I focused on digging into the root cause of the "not connecting" problem I had been experiencing since the previous weekend. Upon initial investigation, I realized that the problem was more fundamental than the code logic on the PSoC or C# side: Windows was not recognizing my PSoC device as a valid USB device. It was listed as an "Unknown Device" in Device Manager, making any attempt to exchange data impossible from the start. This critical hurdle shifted the focus of the week entirely to resolving driver issues. I started by downloading and installing the official Cypress USB drivers, which was the most obvious solution, but unfortunately this standard approach did not work.

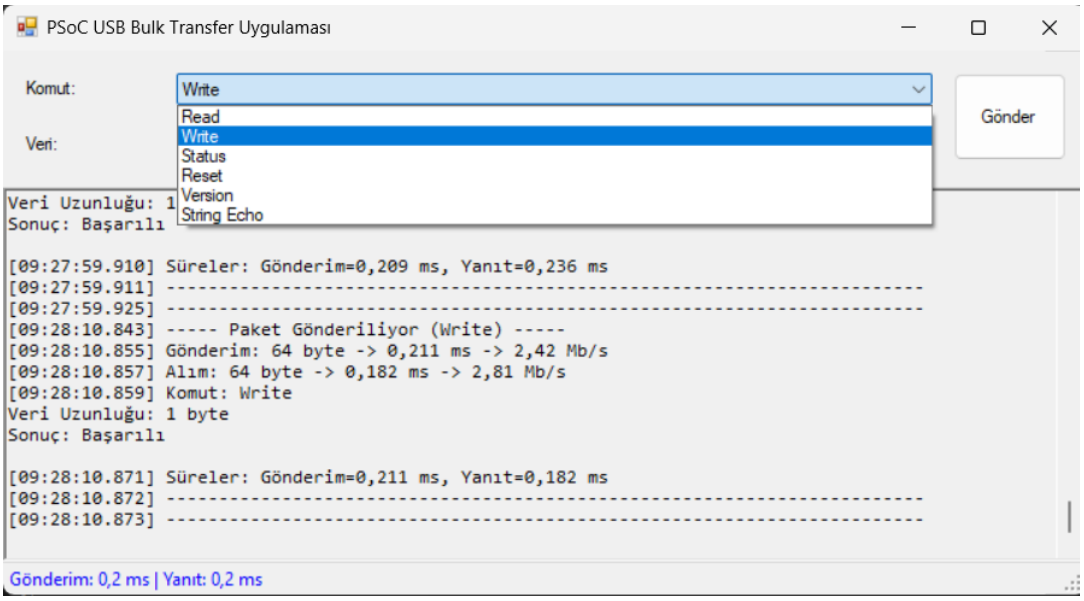
When the problem persisted, I re-examined the configuration of the USBFS component in my PSoC project; I tried different combinations of endpoint addresses and device identifiers (VID/PID), but none of them allowed Windows to recognize the device correctly. This confirmed that the problem was not with the hardware configuration of the PSoC, but rather on the PC side, where the operating system was not assigning the appropriate driver to the device. At this point, I decided to try a more radical solution and tried to write my own custom driver .inf file. In this file, I specified the VID and PID of my PSoC device, aiming to tell Windows to "recognize this device as a generic USB device". I tried to load this custom driver by temporarily disabling Windows' driver signing enforcement, but due to my inexperience in writing drivers, this attempt also failed.

Towards the end of the week, I came up with a solution using a different strategy. I discovered that Cypress's EZ-USB FX3 SDK contains working, signed USB drivers for various Cypress devices. I opened the .inf configuration file of one of these drivers with a text editor and added a new line containing the VID and PID of my PSoC device. After installing the driver in this modified form, my PSoC device was finally recognized properly when I refreshed Device Manager.

1.1.4 Week 4

In the fourth week, based on feedback from my mentor, I needed to move from the simple "send/receive data" prototype to a much more flexible and powerful structure. Accordingly, I redesigned the C# interface and created a command-based communication architecture. I added a ComboBox (drop-down list) to the interface, where the command to be sent could be selected.

I also made a major update to the C code on the PSoC side to support this new command structure. I wrote a main process function (ProcessPacket) that reads the command ID (commandId) of the incoming USB packet and calls the relevant function in a switch-case structure according to this ID. For each command, I created code blocks that perform the necessary action, prepare a response packet and send this packet back to the PC.



```
Veri Uzunluğu: 1
Sonuç: Başarılı

[09:27:59.910] Süreler: Gönderim=0,209 ms, Yanıt=0,236 ms
[09:27:59.911] -----------------------------------------------------------------------
[09:27:59.925] -----------------------------------------------------------------------
[09:28:10.843] ----- Paket Gönderiliyor (Write) -----
[09:28:10.855] Gönderim: 64 byte -> 0,211 ms -> 2,42 Mb/s
[09:28:10.857] Alım: 64 byte -> 0,182 ms -> 2,81 Mb/s
[09:28:10.859] Komut: Write
Veri Uzunluğu: 1 byte
Sonuç: Başarılı

[09:28:10.871] Süreler: Gönderim=0,211 ms, Yanıt=0,182 ms
[09:28:10.872] -----------------------------------------------------------------------
[09:28:10.873] -----------------------------------------------------------------------
```

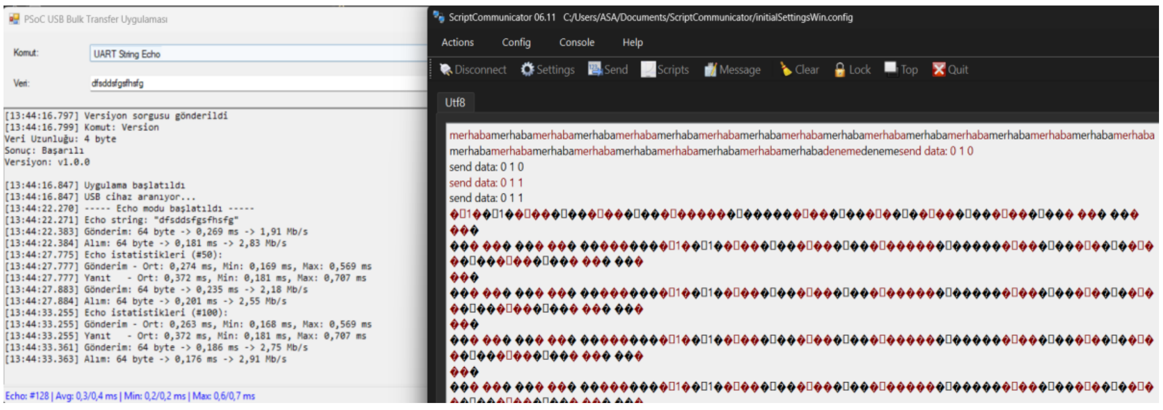
Gönderim: 0,2 ms | Yanıt: 0,2 ms

Figure 1.2 New design and features of interface

On the last day of the week, in addition to the existing Bulk transfer interface, I was asked to add a virtual COM port (UART/CDC) functionality to the system. This meant that the PSoC could handle both special command traffic and standard text-based serial communication over a single USB cable. I made the necessary preliminary preparations by examining technical documents and researching sample projects on how to establish this "Composite Device" structure on the PSoC, how to manage the endpoints of two different interfaces without conflicting with each other and how to access these two different interfaces simultaneously on the PC side.

1.1.5 Week 5

The main focus of my fifth week was to take the communication capabilities of the project one step further by adding a virtual COM port (UART/CDC) interface that would work in parallel with the existing custom Bulk transfer protocol. I restructured the USBFS component of my project in PSoC Creator, adding a standard CDC (Communications Device Class) interface next to the existing "Vendor-Specific" interface. This meant that the PSoC would now work as a "Composite Device". I carefully defined the IN, OUT and Interrupt endpoints required for the CDC interface, making sure they did not conflict with the Bulk transfer endpoints. I added functions to the C code on the PSoC side, which receive the data coming from the PC over the CDC interface (USB_GetAll()) and return the data as is (USB_PutData()). On the C# side, I added a new tab to my application, developed a terminal interface that lists the virtual COM ports in the system, allows the user to connect to the desired port, and can start a "UART Echo" test. As a result of these developments, I was able to send and receive special Bulk commands from my C# application and simultaneously exchange text-based data via the UART terminal from the other tab. This was concrete proof that I had achieved the "multiple communication over a single cable" goal at the very beginning of the project.

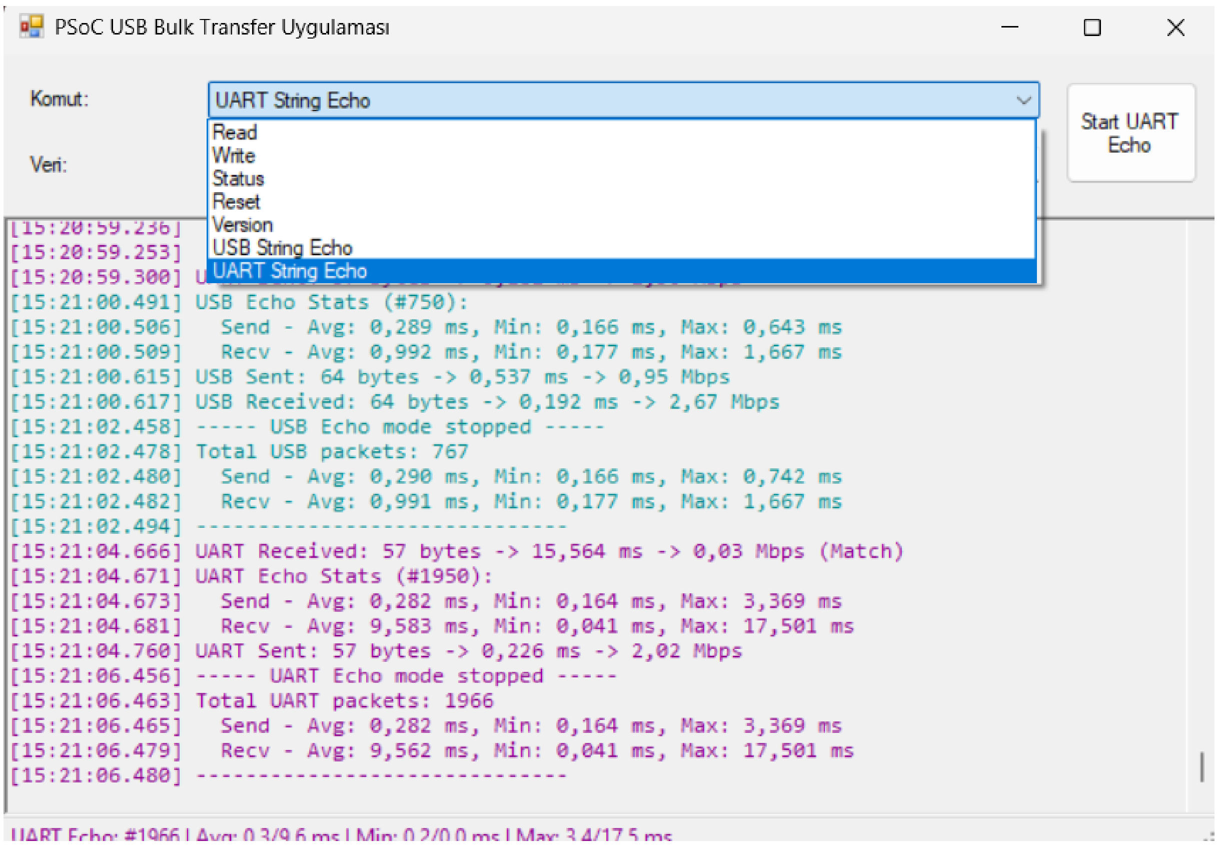

PSoC USB Bulk Transfer Uygulaması
Komut:
UART String Echo
Veri:
dfsddsfgsfhsfg
ScriptCommunicator 06.11 C:/Users/ASA/Documents/ScriptCommunicator/initialSettingsWin.config
Actions
Config
Console
Help
Disconnect
Settings
Send
Scripts
Message
Clear
Lock
Top
Quit
Utf8

```
[13:44:16.797] Versiyon sorgusu gönderildi
[13:44:16.799] Komut: Version
Veri Uzunluğu: 4 byte
Sonuç: Başarılı
Versiyon: v1.0.0

[13:44:16.847] Uygulama başlatıldı
[13:44:16.847] USB cihaz aranıyor...
[13:44:22.270] ----- Echo modu başlatıldı -----
[13:44:22.271] Echo string: "dfsddsfgsfhsfg"
[13:44:22.383] Gönderim: 64 byte -> 0,269 ms -> 1,91 Mb/s
[13:44:22.384] Alım: 64 byte -> 0,181 ms -> 2,83 Mb/s
[13:44:27.775] Echo istatistikleri (#50):
[13:44:27.777] Gönderim - Ort: 0,274 ms, Min: 0,169 ms, Max: 0,569 ms
[13:44:27.777] Yanıt   - Ort: 0,372 ms, Min: 0,181 ms, Max: 0,707 ms
[13:44:27.883] Gönderim: 64 byte -> 0,235 ms -> 2,18 Mb/s
[13:44:27.884] Alım: 64 byte -> 0,201 ms -> 2,55 Mb/s
[13:44:33.255] Echo istatistikleri (#100):
[13:44:33.255] Gönderim - Ort: 0,263 ms, Min: 0,168 ms, Max: 0,569 ms
[13:44:33.255] Yanıt   - Ort: 0,372 ms, Min: 0,181 ms, Max: 0,707 ms
[13:44:33.361] Gönderim: 64 byte -> 0,186 ms -> 2,75 Mb/s
[13:44:33.363] Alım: 64 byte -> 0,176 ms -> 2,91 Mb/s

Echo: #128 | Avg: 0,3/0,4 ms | Min: 0,2/0,2 ms | Max: 0,6/0,7 ms
```

```
merhabamerhabamerhabamerhabamerhabamerhabamerhabamerhabamerhabamerhabamerhabamerhabamerhabamerhabamerhabamerhabamerhaba
merhabamerhabamerhabamerhabamerhabamerhabamerhabamerhabamerhabadenemedenemesend data: 0 1 0
send data: 0 1 0
send data: 0 1 1
send data: 0 1 1
�□1��□1��□����□����□����□����□������□�������□����□����□��□��□����□����□����□���� ��� ���
���
��� ��� ��� ��� ���������□1��□1��□����□����□����□����□�������□�������□����□����□��□��□�
��□����□����□���� ��� ���
���
��� ��� ��� ��� ���������□1��□1��□����□����□����□����□�������□�������□����□����□��□��□�
��□����□����□���� ��� ���
���
��� ��� ��� ��� ���������□1��□1��□����□����□����□����□�������□�������□����□����□��□��□�
��□����□����□���� ��� ���
���
��� ��� ��� ��� ���������□1��□1��□����□����□����□����□�������□�������□����□����□��□��□�
```

Figure 1.3 UART echo mode testing with my interface and also test with ScriptCommunicator

Name Surname : Akif Selim ARSLAN		Company Name : PHINIA DELPHI TURKEY	
Internship Type: Software	Start Date : 24.04.2025	End Date : 13.06.2025	Page : 13



```
[15:20:59.236]
[15:20:59.253]
[15:20:59.300] U
[15:21:00.491] USB Echo Stats (#750):
[15:21:00.506]   Send - Avg: 0,289 ms, Min: 0,166 ms, Max: 0,643 ms
[15:21:00.509]   Recv - Avg: 0,992 ms, Min: 0,177 ms, Max: 1,667 ms
[15:21:00.615] USB Sent: 64 bytes -> 0,537 ms -> 0,95 Mbps
[15:21:00.617] USB Received: 64 bytes -> 0,192 ms -> 2,67 Mbps
[15:21:02.458] ----- USB Echo mode stopped -----
[15:21:02.478] Total USB packets: 767
[15:21:02.480]   Send - Avg: 0,290 ms, Min: 0,166 ms, Max: 0,742 ms
[15:21:02.482]   Recv - Avg: 0,991 ms, Min: 0,177 ms, Max: 1,667 ms
[15:21:02.494] ------------------------------
[15:21:04.666] UART Received: 57 bytes -> 15,564 ms -> 0,03 Mbps (Match)
[15:21:04.671] UART Echo Stats (#1950):
[15:21:04.673]   Send - Avg: 0,282 ms, Min: 0,164 ms, Max: 3,369 ms
[15:21:04.681]   Recv - Avg: 9,583 ms, Min: 0,041 ms, Max: 17,501 ms
[15:21:04.760] UART Sent: 57 bytes -> 0,226 ms -> 2,02 Mbps
[15:21:06.456] ----- UART Echo mode stopped -----
[15:21:06.463] Total UART packets: 1966
[15:21:06.465]   Send - Avg: 0,282 ms, Min: 0,164 ms, Max: 3,369 ms
[15:21:06.479]   Recv - Avg: 9,562 ms, Min: 0,041 ms, Max: 17,501 ms
[15:21:06.480] ------------------------------
```

UART Echo: #1966 | Avg: 0.3/9.6 ms | Min: 0.2/0.0 ms | Max: 3.4/17.5 ms

Figure 1.4 Paralel working of UART echo mode and Bulk transfer mode

In the middle of the week, during a meeting with my mentor, it was stated that a new requirement had been added to the scope of the project: CAN (Controller Area Network) communication. The project was now also required to function as a USB-CAN gateway. However, the PSoC development kit I had been using up until that point did not have a physical CAN transceiver and pinouts. To overcome this technical limitation, I switched to a new development kit that used the same PSoC 5LP chip family but had built-in CAN hardware and physical connection ports. I spent the rest of the week transferring all the Bulk and UART/CDC codes I had written up until that point to this new development kit and making sure they worked without any problems. After successfully completing this transition, I began an intensive research and documentation review process on how to configure the CAN component of the PSoC, how to package and send CAN messages to the PC over USB, and how to process this data in C#.

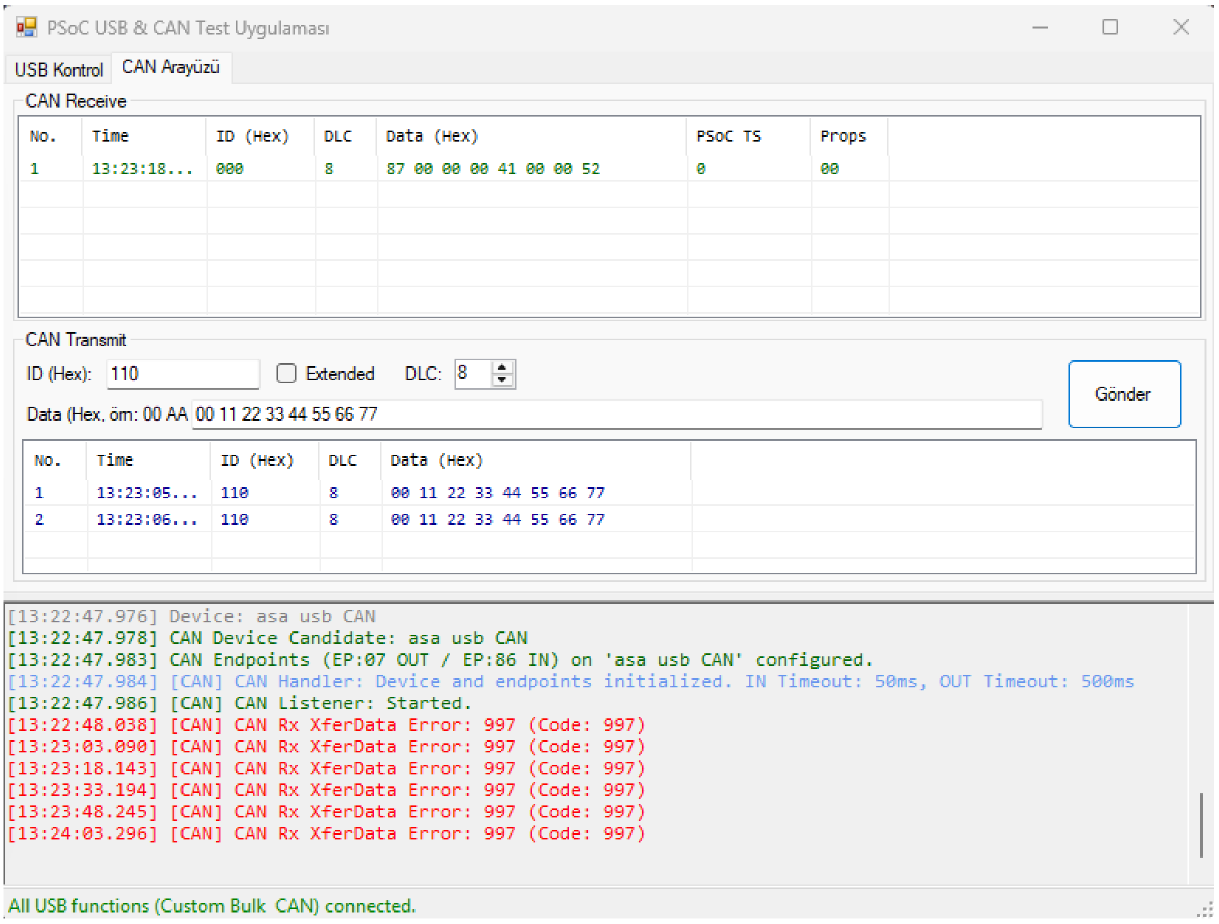
Internship Advisor: Software Technical Leader İbrahim Lütfullah METE	Signature:

1.1.6 Week 6

My sixth week was spent focusing on implementing the most important and industry-specific module of the project, the CAN-USB gateway function, both on the embedded software and PC application side. In light of the research I did the previous week, I added a new Bulk IN (EP6) and Bulk OUT (EP7) interface that I reserved specifically for CAN communication to the USBFS component of my PSoC project. Thus, my device has now become a full-fledged composite device that can provide three different functions, Bulk, UART/CDC and CAN, over a single USB connection. On the PSoC side, I initialized the CAN component and wrote an interrupt service routine (CAN_ISR_Handler). This routine is automatically triggered when a new message arrives on the CAN line, reads the incoming message ID, data length (DLC) and 8 bytes of data, converts them to a special 18-byte USB packet format that I have previously determined, and sends it to the PC via the CAN-specific Bulk IN endpoint. In the reverse direction, I implemented a function (CAN_Process_USB_Message) that parses the 18-byte USB packet coming from the PC and converts it into a CAN message and sends it directly to the CAN line with the CAN_Send_Message() function.

Simultaneously, I added a new tab called "CAN Interface" to the interface of my C# desktop application. This tab was designed to provide the user with the basic features of a professional CAN analysis tool. I created fields where users can enter the ID (both in standard 11-bit and extended 29-bit formats), DLC and data bytes of the CAN message they want to send. I also designed a logging interface where all messages received and sent over the CAN line are displayed in real time in separate lists with their timestamp, direction (Rx/Tx), ID and data. In order for this complex structure not to block the user interface, I carried out the CAN listening and sending operations in background threads within a class I named CanHandler. In this way, the application remained fluid and responsive even when hundreds of CAN messages were flowing per second.

While connecting one end of the system I developed to the PC, I also connected the CAN pins of the PSoC to an external USB-CAN adapter. I ran the industry standard PCAN-View software on this adapter. In the tests I conducted, I saw that a CAN message I sent from the C# interface I developed appeared on the PCAN-View screen immediately, and a message I sent via PCAN-View also appeared in the "Received Messages" list of my own application. This successful bidirectional communication test showed that the project's CAN gateway module worked as expected and worked flawlessly in parallel with the other two communication channels (Bulk and UART). The project reached a state where it could perform all the main functions it aimed to.



No.	Time	ID (Hex)	DLC	Data (Hex)	PSoC TS	Props
1	13:23:18...	000	8	87 00 00 00 41 00 00 52	0	00

ID (Hex): 110 ☐ Extended DLC: 8 Gönder

Data (Hex, örn: 00 AA 00 11 22 33 44 55 66 77

No.	Time	ID (Hex)	DLC	Data (Hex)
1	13:23:05...	110	8	00 11 22 33 44 55 66 77
2	13:23:06...	110	8	00 11 22 33 44 55 66 77

```
[13:22:47.976] Device: asa usb CAN
[13:22:47.978] CAN Device Candidate: asa usb CAN
[13:22:47.983] CAN Endpoints (EP:07 OUT / EP:86 IN) on 'asa usb CAN' configured.
[13:22:47.984] [CAN] CAN Handler: Device and endpoints initialized. IN Timeout: 50ms, OUT Timeout: 500ms
[13:22:47.986] [CAN] CAN Listener: Started.
[13:22:48.038] [CAN] CAN Rx XferData Error: 997 (Code: 997)
[13:23:03.090] [CAN] CAN Rx XferData Error: 997 (Code: 997)
[13:23:18.143] [CAN] CAN Rx XferData Error: 997 (Code: 997)
[13:23:33.194] [CAN] CAN Rx XferData Error: 997 (Code: 997)
[13:23:48.245] [CAN] CAN Rx XferData Error: 997 (Code: 997)
[13:24:03.296] [CAN] CAN Rx XferData Error: 997 (Code: 997)
```

All USB functions (Custom Bulk CAN) connected.

Figure 1.5 CAN communication interface

1.1.7 Week 7

In this last week, I first tested all modules of the project intensively under different scenarios. During these tests, I detected some minor errors and exceptions that could occur especially under high data traffic or as a result of unexpected user inputs. For example, in the C# interface, I made the application give more informative error messages when invalid format data was entered in the CAN ID or data fields, and on the PSoC side, I tried to make the system more resilient against possible errors by making the length checks of incoming USB packets more robust.

Again this week, I prepared a comprehensive technical documentation to ensure that the project can be easily understood and developed by other engineers after me. I added explanatory comments to the code for both the PSoC embedded software and the C# desktop application. In addition, I prepared a detailed report explaining the general architecture of the project, the functions of the USB endpoints used, the special Bulk packet structure and how CAN messages are tunneled over USB. This document includes all the information, starting from the installation steps of the project and how to use each feature.

On the last day of my internship, I presented this project, which I worked on for seven weeks, to the engineers and managers in the Phinia Delphi Turkey software department. In my presentation, I showed the starting point of the project, the difficulties I encountered (especially the USB driver problem), the solutions I developed, and the practical benefits the project would provide to the team with a live demo. I explained how being able to both monitor CAN traffic and send special commands with a single USB cable would simplify the testing processes.

1.2 Conclusion

This internship project aimed to solve an operational challenge encountered in the software development processes of Phinia Delphi Turkey. Using multiple different adapters and cables to communicate with Electronic Control Units (ECU) for test and diagnostic purposes complicated the setup of the test setups and reduced efficiency. The main goal of the project was to eliminate this scattered structure and develop a holistic hardware and software system that can handle CAN, UART and custom command-based communication over a single USB connection. In line with this goal, an integrated solution consisting of Cypress PSoC 5LP microcontroller-based hardware and a C#-based desktop application managing this hardware was successfully implemented. PSoC was configured as a Composite USB Device, enabling it to simultaneously provide three independent communication channels (CAN-USB Gateway, CRC-protected custom Bulk Protocol and standard UART/CDC) over a single physical connection. The developed desktop application provides a central control and monitoring interface for all of these channels, providing a holistic experience to the user.

As a result of the project, a complex process that previously required multiple hardware and software was reduced to a single, easy-to-use tool. This successfully completed system was presented to the software department in the last week of the internship and it was verified that it fulfilled all the targeted functions flawlessly. This developed tool has the potential to significantly simplify and accelerate engineers' testing and verification processes.

During the seven-week internship, the entire life cycle of a project was experienced, starting from the identification of an engineering problem to the concept development, implementation, testing, documentation and presentation stages. While the technical difficulties encountered (especially USB driver compatibility) improved systematic problem-solving skills, the successful completion of the project provided valuable experience in transforming theoretical knowledge into practical applications.

SECTION 2 INTERNSHIP

2.1 Project Overview and Objectives

2.1.1 Definition of the Current Problem and Purpose of the Project

Modern Electronic Control Units (ECUs) developed in the automotive industry are multi-layered and complex systems. The testing, verification and debugging processes of these systems require simultaneous interaction over multiple communication protocols. In the current workflow, an engineer usually needs more than one independent hardware tool to fully analyze an ECU. For example, he may need to use a CAN-USB adapter to monitor in-car network traffic, a UART-USB converter to capture text-based diagnostic logs (debug logs), and a separate programmer or interface device to send special configuration data.

The This fragmented approach leads to several operational inefficiencies:

- **Setup Complexity:** Test rigs take longer to set up and create a mess of cables in the work area.
- **Software Clutter:** Each hardware tool often requires its own driver and control software, forcing engineers to constantly switch between different software interfaces.
- **Inefficiency:** Managing different tools and software wastes time in development and testing cycles.
- **Reliability:** Each individual piece of hardware in the system represents a potential point of failure.

The main objective of this project is to develop a multifunctional USB communication gateway that combines all necessary communication channels with an ECU on a single platform, eliminating the operational inefficiencies described above. This integrated solution aims to perform the following basic functions by connecting to a computer via a single USB cable:

- **CAN-USB Gateway:** Working like a standard CAN analyzer, listening to traffic on the CAN line where the ECU is located, transmitting these messages to the PC for analysis, and sending commands from the PC to the line as CAN messages.
- **Bulk Transfer Channel:** Designed for situations where the CAN protocol is not suitable (e.g. large data blocks, firmware updates, secure configuration commands), providing a high-performance and reliable data transfer channel with CRC (Cyclic Redundancy Check) data integrity guarantee.
- **Virtual COM Port (UART/CDC):** Acting like a standard serial port, allowing simple text-based diagnostic data or commands to be easily received and sent.

The ultimate goal is to deliver a professional engineering tool that eliminates hardware clutter on engineers' test benches, meets all communication needs through a single hardware and a single software interface, thus significantly speeding up and simplifying test and development processes.

2.1.2 Scope of the Project and Defined Success Criteria

The The scope of this project includes the end-to-end development of a hardware and software system to solve the defined problem. The project boundaries are determined by the following items:

Embedded Software (Firmware):

- A firmware will be developed in C language on the Cypress PSoC 5LP microcontroller.
- The firmware will enable the PSoC to operate as a Composite USB Device and will include three different interfaces:
- Special Bulk Transfer Interface (Vendor-Specific Bulk Interface)

- CAN-USB Gateway Interface (via a second Bulk Interface)
- Virtual COM Port Interface (CDC/ACM Interface)
- A command-based packet protocol (UsbPacket) that performs data integrity checks with CRC16 checksum for bulk transfer will be implemented.
- The CAN module will convert and process incoming and outgoing CAN messages to USB format.

Desktop Software (PC Application):

- A PC interface application will be developed using the C# language and .NET Windows Forms platform.
- The application will automatically recognize the PSoC device using the CyUSB.dll library and manage all communication interfaces.
- The user interface will include separate and functional tabs for each communication channel (Bulk, CAN, UART).
- The application will log data from all channels in real time and present it to the user.database.

The following measurable and testable criteria have been determined to verify the successful completion of the project:

1. *Device Recognition and Connection:* The developed C# application should be able to recognize the device and establish a connection without error via the specified VID/PID when the PSoC device is plugged into the USB port.
2. *Bulk Protocol Functionality:* PSoC should return a version response (v1.0.0) in the correct format and content to a CMD_VERSION command sent via the application.
3. *CAN Gateway Verification:* A CAN message sent from a C# application should be observed with the correct ID and data content on an external CAN analyzer (e.g. PCAN-View). Conversely, a message sent from an external analyzer should be listed correctly on the interface of the C# application.
4. *UART/CDC Functionality:* A text string sent from the application's virtual COM port terminal should be received by the PSoC and returned without modification and displayed correctly on the terminal screen (Echo Test).

5. Simultaneous Operation: While three communication channels (Bulk, CAN, UART) are actively working at the same time, there should be no crashes, data loss or performance degradation in the system.
6. Project Delivery: The project, along with all developed source codes, technical documentation and this internship report, should be submitted to the Phinia Delphi Turkey software department and approved.

2.1.3 Main Technologies and Development Tools Used

From the beginning to the end of the project, various hardware, software and tools commonly used in modern embedded systems and desktop application development processes were utilized. The main technologies used are listed below:

- **Cypress PSoC 5LP Development Kit (CY8CKIT-059):** It was used as the hardware platform of the project. It was preferred because it has an ARM Cortex-M3 core, programmable digital and analog blocks and especially includes USBFS and CAN controllers.
- **Cypress PSoC Creator IDE:** It was used as an integrated development, component-based hardware configuration, coding and debugging environment for PSoC microcontrollers.
- **C Programming Language:** PSoC firmware was written in this language because it is the industry standard for embedded system programming and offers low-level access to hardware.
- **PSoC Components:** USBFS, CAN and UART components provided ready-made in PSoC Creator were used as the basic building blocks in the hardware layer abstraction of the project.
- **Microsoft Visual Studio IDE:** It is the main integrated development environment used for the development, design and debugging of C# desktop applications.
- **C# Programming Language:** It is preferred for PC interface software because it is an object-oriented, modern and powerful language that runs on the .NET platform.
- **Microsoft .NET Framework & Windows Forms:** Platform and technology used to develop graphical user interfaces (GUI) quickly.

- **Cypress CyUSB.dll Library:** It is the library that provides easy and effective communication between Cypress USB devices and the PC, forming the USB communication backbone on the PC side of the project.
- **PCAN-View:** It is a third-party, industry-standard CAN analysis software used to test and verify the developed CAN-USB gateway function.
- **PCAN-USB Adapter:** It is an external USB-CAN hardware adapter that allows the PCAN-View software to be physically connected to the CAN line.

2.2 Theoretical Infrastructure of Communication Protocols

2.2.1 Universal Serial Bus - USB

The The USB architecture is based on a master-slave or host-centric topology, where the host plays a central role. In this structure, all communication is controlled by the host computer, which manages the data path and initiates all data transfers. Peripheral devices respond to requests from the host computer, but cannot initiate a transfer on their own. This centralized control simplifies data path management and prevents conflicts. One of the most distinctive features of USB is its "Hot-Swapping" capability, which allows devices to be plugged in and out during operation, and "Plug-and-Play", which is the process of automatically recognizing, querying, and loading the appropriate driver for the plugged device by the host computer. This process consists of a series of standard steps known as "enumeration", in which the device reports its identity and capabilities to the host computer [1].

In addition, the USB standard is designed not only for data transfer, but also for powering low-power devices. This feature has enabled many peripherals to be powered directly via the USB port without the need for an external power adapter. The standard has evolved over time, offering exponential increases in data transfer speeds from USB 1.1 to USB 2.0 (High-Speed) to USB 3.x (SuperSpeed). The reason for choosing USB as the basis for this project is its standardized, flexible, power-supplied and, most importantly, the ability to manage multiple, independent data streams over a single physical interface thanks to its Composite Device architecture. This capability has provided an ideal basis for achieving the project's goal of being a multifunctional engineering tool [1].

2.2.1.1 USB Transfer Types and Bulk Transfer

The USB standard defines four basic transfer types designed for different needs [2]:

- **Control Transfer:** Used for basic device management operations such as configuring the device, querying its status, and sending commands. It is mandatory for all USB devices to support it.
- **Interrupt Transfer:** Designed for small and periodic data-sending, delay-sensitive devices such as keyboards and mice. Data transfer is guaranteed at certain intervals.
- **Isochronous Transfer:** Used for applications that require continuous and real-time data flow, such as audio or video streaming. On-time delivery is more important than data integrity; therefore, error checking is not performed.
- **Bulk Transfer:** Designed for situations where large amounts of data must be transferred without errors. Data integrity is guaranteed by CRC (Cyclic Redundancy Check) mechanisms, but the timing of the transfer is not guaranteed; it uses the entire available bandwidth when the data path is not busy.

In this project, the Bulk Transfer type was preferred for both the special command protocol and tunneling of CAN messages. The main reason for this is that this type of transfer supports large data packets and, most importantly, provides a reliable communication channel by guaranteeing data integrity [3].

2.2.1.2 USB Descriptors and Endpoints

When a USB device is connected to a host, the host performs a series of queries to understand what the device is and what capabilities it has. This process is called "enumeration". The device responds to these queries with standard data structures called Descriptors [4]. The main identifiers are:

- **Device Descriptor:** Contains general information about the device (e.g. VID-Vendor ID, PID-Product ID). Each device has one.
- **Configuration Descriptor:** Contains information such as the power requirements of the device and the number of interfaces it offers.

- **Interface Descriptor:** Describes each function offered by the device (e.g. keyboard, sound card, custom interface). Each interface can belong to a specific "Device Class".
- **Endpoint Descriptor:** Identifies Endpoints, which are buffer memory areas where data actually flows between the device and the host. Endpoints have properties such as direction (IN or OUT), transfer type (Bulk, Interrupt, etc.), and maximum packet size [5].

2.2.1.3 Composite USB Devices Architecture

The When a USB device offers multiple interfaces (and therefore multiple functions) simultaneously over a single physical connection, it is called a Composite Device architecture. This is achieved by hosting multiple Interface Descriptors under a single Device Descriptor. The host computer treats each interface as a separate logical device and loads the appropriate driver for each [6]. By configuring the PSoC device as a Composite Device, this project was able to offer a dedicated Bulk interface, a CAN gateway interface, and a CDC/virtual COM port interface over a single USB cable. This was the key architecture used to achieve one of the main goals of the project.

2.2.2 Controller Area Network (CAN) Protocol

The Developed by Robert Bosch GmbH in the 1980s, the CAN protocol is a serial communication protocol designed to allow microcontrollers and devices, particularly for the automotive industry, to communicate with each other robustly and efficiently without a central computer [8].

2.2.2.1 CAN Bus Fundamentals and Physical Layer

CAN is a message-based protocol that operates in a multi-master structure. The physical layer is usually based on a two-wire structure (CAN_H and CAN_L) using differential signaling, which provides high immunity to external electromagnetic noise. The data path should be terminated with 120 Ohm termination resistors at both ends to prevent signal reflections [9].

2.2.2.2 CAN Message Frame Structure: Standard and Extended ID

There is no addressing in the CAN protocol; instead, each message carries a Message Identifier (ID) that identifies its content and priority. There are two types of message frames:

- Standard CAN (CAN 2.0A): Uses an 11-bit identifier.
- Extended CAN (CAN 2.0B): Uses a 29-bit identifier, allowing for a much larger number of unique message identifiers. A data frame consists of the Start Bit (SOF), Identifier (ID), Control Field (containing DLC-Data Length Code), Data Field from 0 to 8 bytes, CRC Field, Acknowledgement (ACK) Field, and End of Frame (EOF) [10].

2.2.2.3 Data Path Arbitration and Error Detection Mechanisms

One of the most powerful features of CAN is its arbitration mechanism, which elegantly resolves data path conflicts. If multiple nodes start sending messages at the same time, a bit-by-bit arbitration process is initiated based on the message ID. Since the '0' bit is dominant over the '1' bit, the node with the numerically smaller ID wins the data path and sends its message without interruption, while the other nodes silently listen. Thanks to this mechanism, no data is lost and the highest priority (lowest ID) message always takes precedence [11]. CAN also offers highly reliable communication thanks to five different built-in error detection mechanisms: Bit Watch, Bit Stuffing, Frame Check (CRC), Acknowledgement Check, and Form Check [12].

2.2.3 UART ve USB CDC (Communications Device Class)

2.2.3.1 Asynchronous Serial Communication (UART) Principles

UART (Universal Asynchronous Receiver-Transmitter) is one of the most basic serial communication protocols that does not require the sharing of a clock signal. Communication usually occurs over two lines (TX - Transmit, RX - Receive). Data is sent in packets that contain a Start Bit, typically 5 to 8 data bits, an optional Parity Bit, and one or more Stop Bits at a predetermined rate (baud rate). This simple and effective structure is widely used for short-distance, point-to-point communication [13].

2.2.3.2 USB CDC and Virtual COM Port (VCP) Concept

USB CDC (Communications Device Class) is a standard device class designed to emulate traditional serial communication interfaces (such as RS-232) over the USB protocol. When a USB device presents itself as a CDC device to the host computer, modern operating systems such as Windows, macOS or Linux automatically create a Virtual COM Port (VCP) for this device without the need for any special drivers [Source 14]. This allows PC software traditionally written to communicate over a COM port to communicate with this modern USB device without any modification. Using the CDC interface in this project provided engineers with the flexibility to easily send and receive text-based diagnostic data via standard terminal programs (e.g. PuTTY, Tera Term) or with the built-in terminal in a C# application [15].

2.3 Design and Development of the System

In this section, the architectural decisions, technical implementation details, and challenges encountered during the development process on both the embedded software and PC interface sides of the project are discussed in detail.

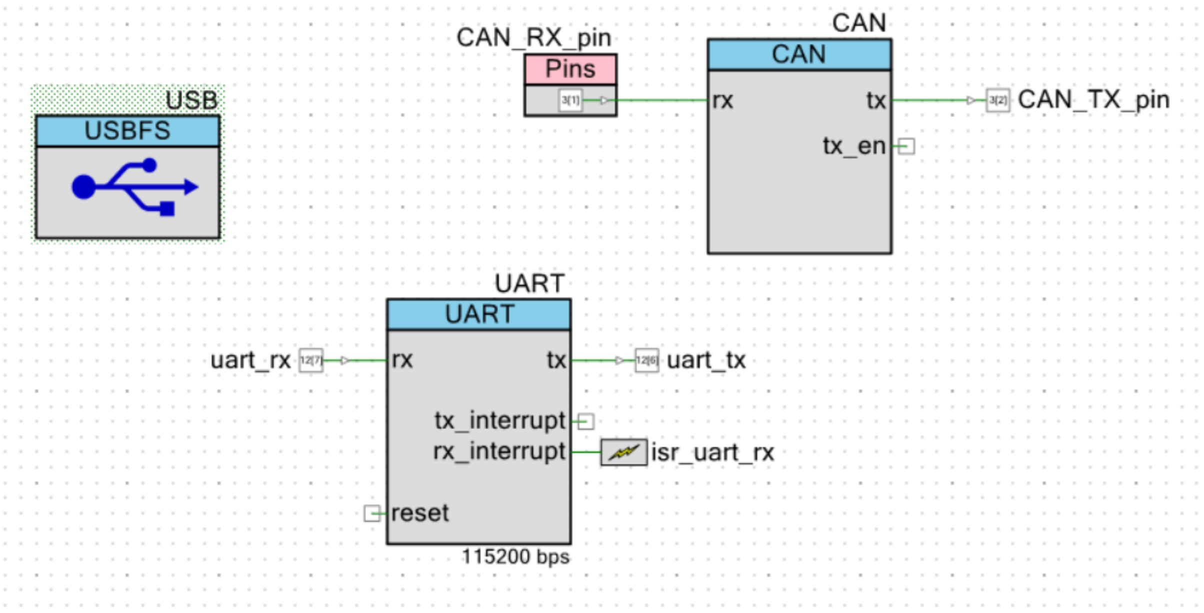
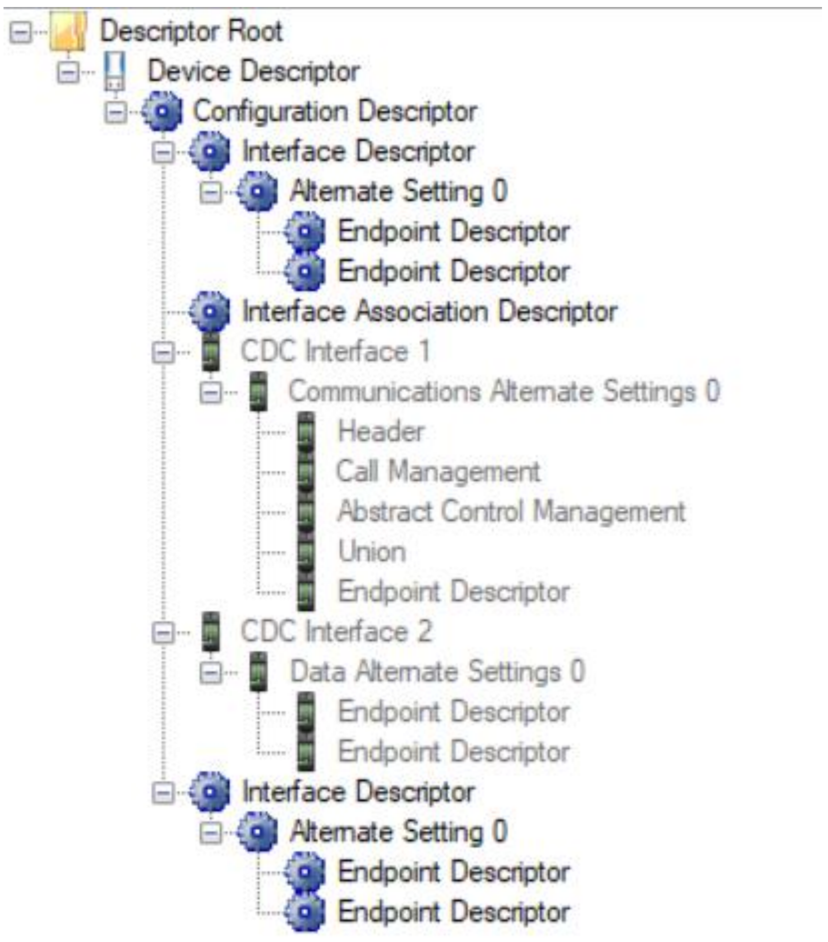


Figure 2.1 General design of microcontroller firmware

2.3.1 PSoC Embedded Software Architecture (Firmware Architecture)

The software architecture on the PSoC side is designed to manage hardware resources efficiently, respond to incoming USB requests simultaneously, and provide a modular structure. An event-based (polling) approach was adopted within the main for(;;) loop, and data from different USB interfaces were continuously checked and directed to the relevant processing functions.

In order to realize one of the main goals of the project, "multiple communication over a single cable", the USBFS (Full-Speed USB) component of the PSoC was configured as a Composite Device. This architecture was created by defining multiple Interface Descriptors under a single Device Descriptor using the graphical interface of the PSoC Creator IDE.



```
Descriptor Root
    Device Descriptor
        Configuration Descriptor
            Interface Descriptor
                Alternate Setting 0
                    Endpoint Descriptor
                    Endpoint Descriptor
            Interface Association Descriptor
            CDC Interface 1
                Communications Alternate Settings 0
                    Header
                    Call Management
                    Abstract Control Management
                    Union
                    Endpoint Descriptor
            CDC Interface 2
                Data Alternate Settings 0
                    Endpoint Descriptor
                    Endpoint Descriptor
            Interface Descriptor
                Alternate Setting 0
                    Endpoint Descriptor
                    Endpoint Descriptor
```

Figure 2.2 USB device descriptor root

When connected to the PC, the device presents itself as a unit with three different functions:

Interface 0: Bulk Transfer Interface (Vendor-Specific):

Endpoints:

- EP1 OUT: Used to receive command packets from the PC to the PSoC. (Address: 0x01)
- EP2 IN: Used to send response packets from the PSoC to the PC. (Address: 0x82)

Purpose: Reliable transmission of special commands and responses in the UsbPacket format.

Interface 1: Virtual COM Port Interface (CDC/ACM):

Endpoints:

- EP4 IN (Data): To send serial data from PSoC to PC.
- EP5 OUT (Data): To receive serial data from PC to PSoC.

EP3 IN (Interrupt): For CDC status notifications (standard requirement).

Purpose: To provide an easily accessible text-based diagnostic (debug) channel that works with standard drivers.

Interface 2: CAN-USB Gateway Interface (Vendor-Specific):

Endpoints:

- EP7 OUT: Used to receive CAN messages to be sent from the PC to the PSoC. (Address: 0x07)
- EP6 IN: Used to send messages read from the CAN line on the PSoC to the PC. (Address: 0x86)

Purpose: To tunnel CAN messages over USB. Using a different interface and endpoint set allows CAN traffic to be completely isolated from other command traffic.

This configuration allows the operating system to create separate logical devices for each interface and load the corresponding drivers (CyUSB.sys for custom interfaces, usbser.sys for CDC).

In order to create a reliable and structured communication channel, a special packet protocol called UsbPacket was designed and implemented in C language. This protocol was designed to be fully compatible with the 64 byte Bulk transfer packet size.

Packet structure is as follows:

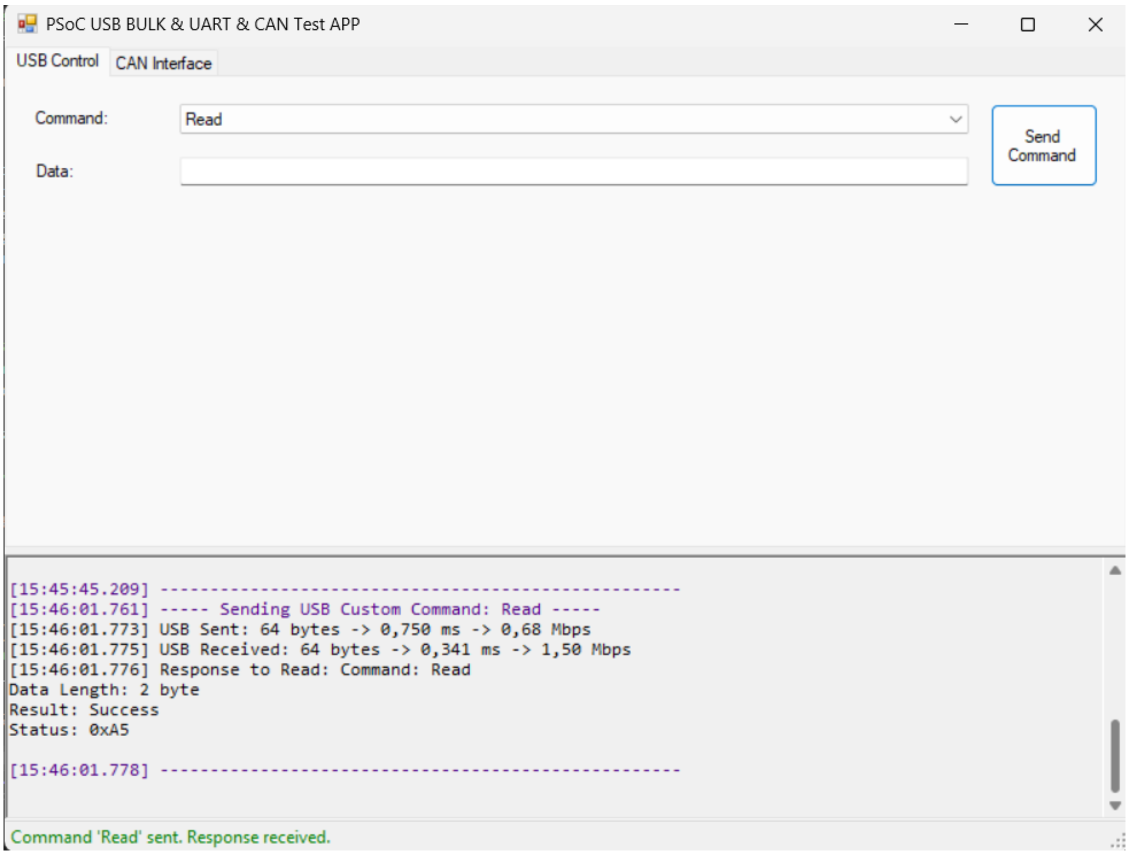
- Header (2 bytes): A fixed signature indicating the beginning of the packet (0xAA, 0x55).
- Command ID (1 byte): Indicates the type of command sent (CMD_READ, CMD_WRITE etc.).
- Data Length (1 byte): Indicates the length of the data field. (Maximum can be 58 bytes).
- Data (0-58 bytes): Carries command-specific data.
- Checksum (2 bytes): CRC-16-CCITT checksum of the part from the header of the packet to the end of the data field. This guarantees data integrity.

The basic functions that manage this protocol on the PSoC side are as follows:

ProcessPacket(): This is the main function that processes an incoming rxPacket. First, it checks the packet's header and CRC with *ValidatePacket().* In case of CRC error or invalid header, it creates an error packet. If the packet is valid, it executes the relevant command in a switch-case structure according to the commandId value and prepares a response packet (txPacket).

BytesToPacket() and *PacketToBytes():* These two functions perform serialization and deserialization operations between raw byte arrays and the UsbPacket C structure. *BytesToPacket* converts the 64-byte custom_outBuffer array read from EP1 into an rxPacket structure; *PacketToBytes* converts the prepared txPacket structure into a custom_inBuffer array to be loaded into EP2.

```
void ProcessPacket() {
    uint8 result = ValidatePacket(&rxPacket);
    uint8 responseData[MAX_DATA_SIZE];
    uint8 responseLen = 0;

    if (result != RESULT_OK) {
        responseData[0] = result;
        PrepareResponsePacket(&txPacket, &rxPacket, 0xFF, responseData, 1);
        return;
    }
    switch(rxPacket.commandId) {
        case CMD_READ:
            responseData[0] = RESULT_OK;
            responseData[1] = deviceStatus;
            responseLen = 2;
            break;
        case CMD_WRITE:
            if (rxPacket.dataLength > 0) {
                deviceStatus = rxPacket.data[0];
                responseData[0] = RESULT_OK;
                responseLen = 1;
            } else {
                responseData[0] = RESULT_ERROR;
                responseLen = 1;
            }
            break;
        case CMD_STATUS:
            responseData[0] = RESULT_OK;
            responseData[1] = deviceStatus;
            responseData[2] = (uint8)(packetCounter & 0xFF);
            responseData[3] = (uint8)((packetCounter >> 8) & 0xFF);
            responseData[4] = (uint8)((packetCounter >> 16) & 0xFF);
            responseData[5] = (uint8)((packetCounter >> 24) & 0xFF);
            responseLen = 6;
            break;
        case CMD_RESET:
            deviceStatus = 0;
            packetCounter = 0;
            responseData[0] = RESULT_OK;
            responseLen = 1;
            break;
        case CMD_VERSION:
            responseData[0] = RESULT_OK;
            responseData[1] = DEVICE_VERSION[0];
            responseData[2] = DEVICE_VERSION[1];
            responseData[3] = DEVICE_VERSION[2];
            responseLen = 4;
            break;
        case CMD_ECHO_STRING:
            if (rxPacket.dataLength > 0) {
                responseData[0] = RESULT_OK;
                uint8 i;
                for (i = 0; i < rxPacket.dataLength; i++) {
                    responseData[i + 1] = rxPacket.data[i];
                }
                responseLen = rxPacket.dataLength + 1;
            } else {
                responseData[0] = RESULT_ERROR;
                responseLen = 1;
            }
            break;
        default:
            responseData[0] = RESULT_INVALID_CMD; kodu.
            responseLen = 1;
            break;
    }
    PrepareResponsePacket(&txPacket, &rxPacket, rxPacket.commandId, responseData,
responseLen);
    packetCounter++;
}
```



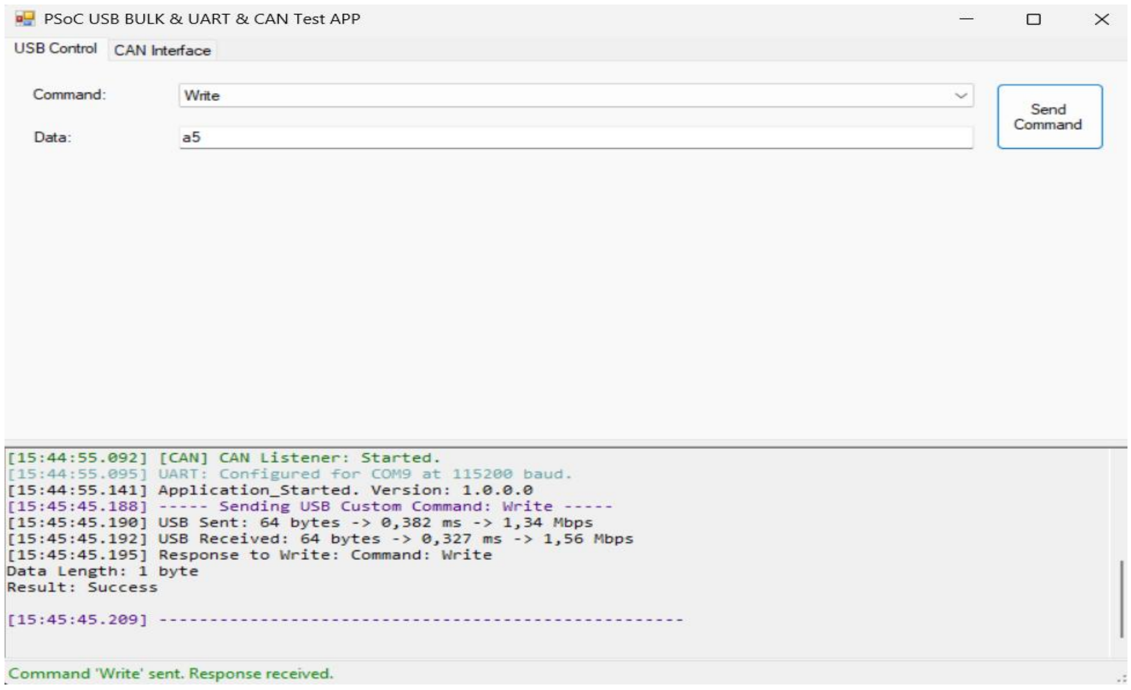
Command: Read

Data:

```
[15:45:45.209] ---------------------------------------------------
[15:46:01.761] ----- Sending USB Custom Command: Read -----
[15:46:01.773] USB Sent: 64 bytes -> 0,750 ms -> 0,68 Mbps
[15:46:01.775] USB Received: 64 bytes -> 0,341 ms -> 1,50 Mbps
[15:46:01.776] Response to Read: Command: Read
Data Length: 2 byte
Result: Success
Status: 0xA5

[15:46:01.778] ---------------------------------------------------
```

Command 'Read' sent. Response received.

Figure 2.3 USB bulk transfer read command



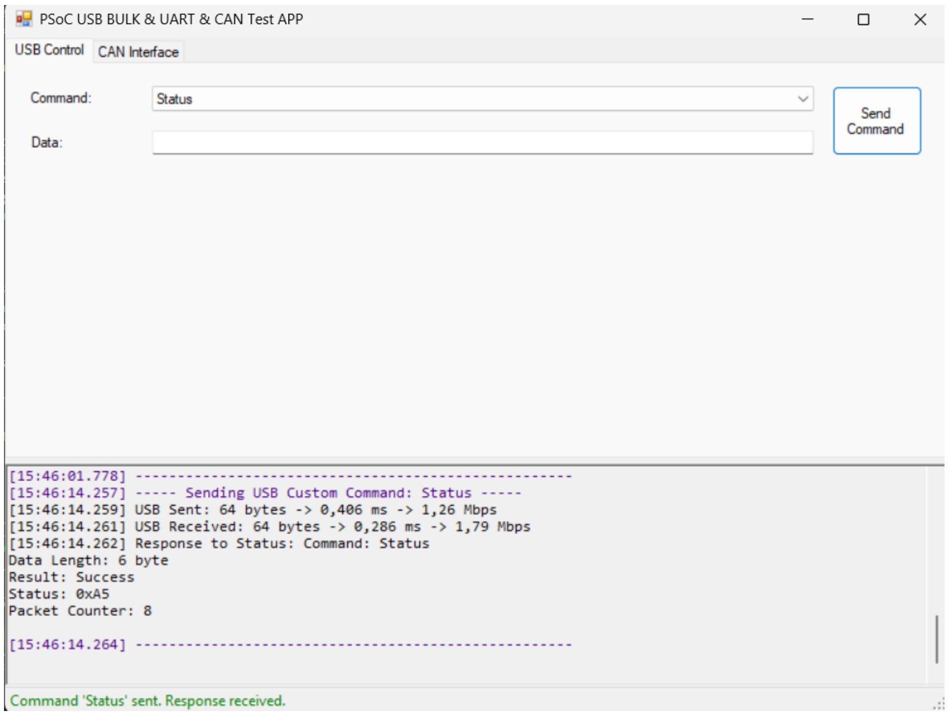
Command: Write

Data: a5

```
[15:44:55.092] [CAN] CAN Listener: Started.
[15:44:55.095] UART: Configured for COM9 at 115200 baud.
[15:44:55.141] Application_Started. Version: 1.0.0.0
[15:45:45.188] ----- Sending USB Custom Command: Write -----
[15:45:45.190] USB Sent: 64 bytes -> 0,382 ms -> 1,34 Mbps
[15:45:45.192] USB Received: 64 bytes -> 0,327 ms -> 1,56 Mbps
[15:45:45.195] Response to Write: Command: Write
Data Length: 1 byte
Result: Success

[15:45:45.209] ---------------------------------------------------
```

Command 'Write' sent. Response received.

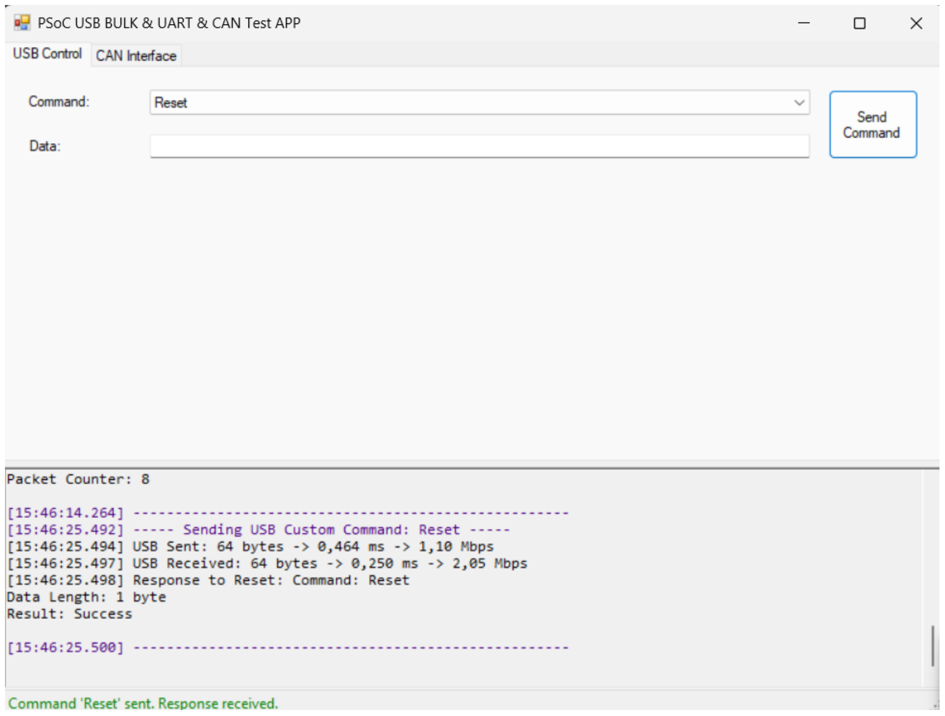
Figure 2.4 USB bulk transfer write command



```
[15:46:01.778] ----------------------------------------------------
[15:46:14.257] ----- Sending USB Custom Command: Status -----
[15:46:14.259] USB Sent: 64 bytes -> 0,406 ms -> 1,26 Mbps
[15:46:14.261] USB Received: 64 bytes -> 0,286 ms -> 1,79 Mbps
[15:46:14.262] Response to Status: Command: Status
Data Length: 6 byte
Result: Success
Status: 0xA5
Packet Counter: 8

[15:46:14.264] ----------------------------------------------------
```

Command 'Status' sent. Response received.

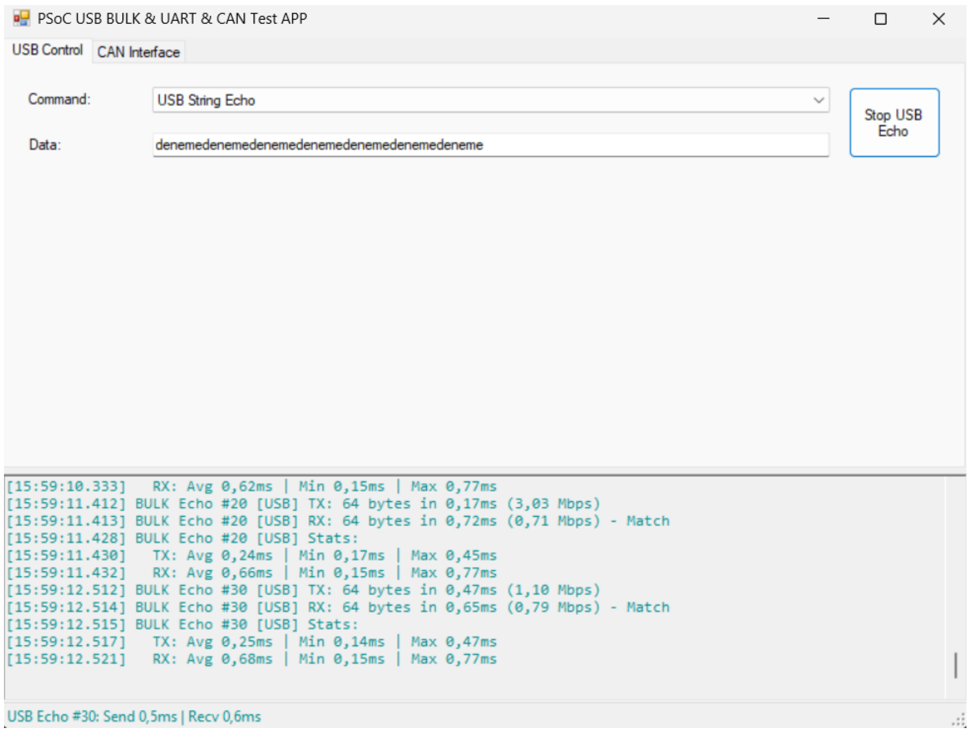
Figure 2.5 USB bulk transfer status command



```
Packet Counter: 8

[15:46:14.264] ----------------------------------------------------
[15:46:25.492] ----- Sending USB Custom Command: Reset -----
[15:46:25.494] USB Sent: 64 bytes -> 0,464 ms -> 1,10 Mbps
[15:46:25.497] USB Received: 64 bytes -> 0,250 ms -> 2,05 Mbps
[15:46:25.498] Response to Reset: Command: Reset
Data Length: 1 byte
Result: Success

[15:46:25.500] ----------------------------------------------------
```

Command 'Reset' sent. Response received.

Figure 2.6 USB bulk transfer reset command



Command: USB String Echo

Data: denemedenemedenemedenemedenemedenemedeneme

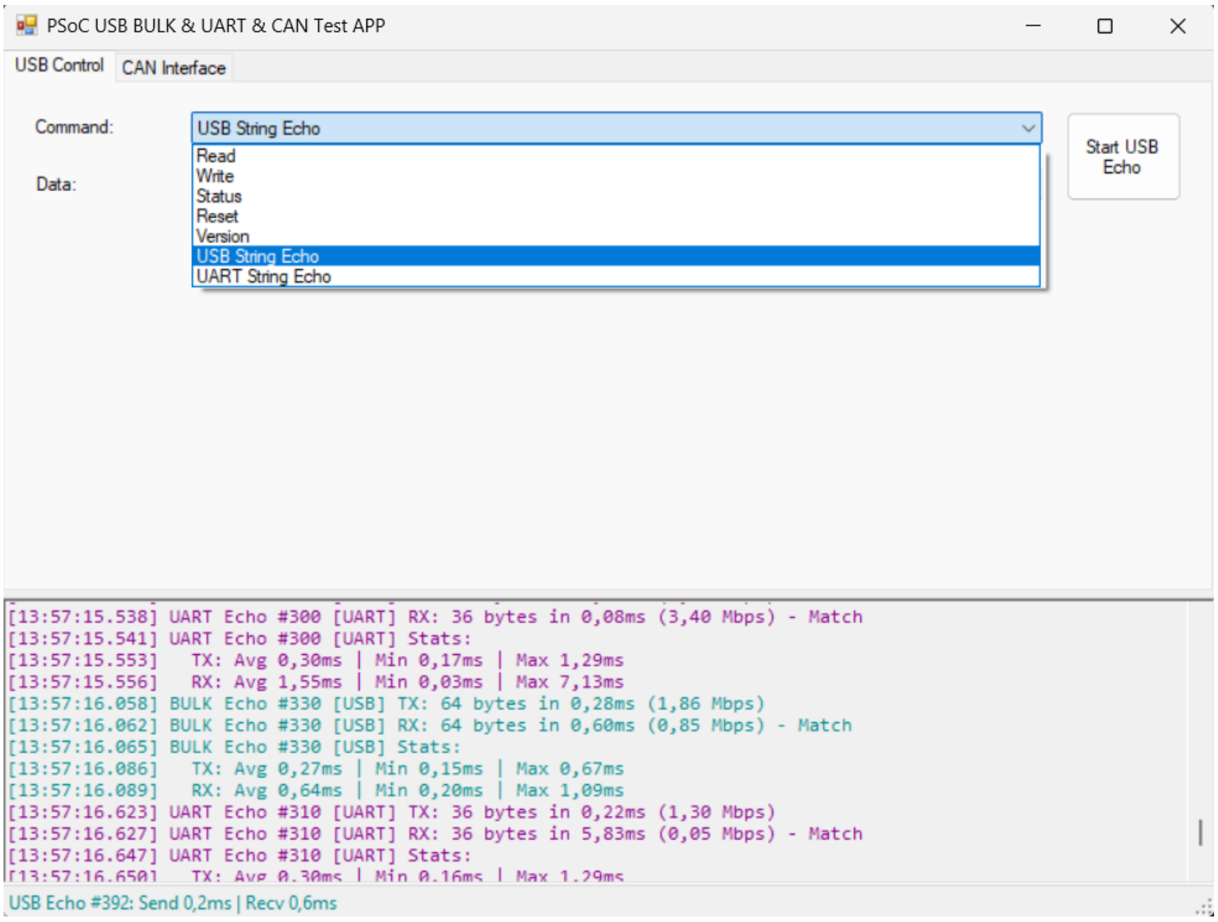
```
[15:59:10.333]   RX: Avg 0,62ms | Min 0,15ms | Max 0,77ms
[15:59:11.412] BULK Echo #20 [USB] TX: 64 bytes in 0,17ms (3,03 Mbps)
[15:59:11.413] BULK Echo #20 [USB] RX: 64 bytes in 0,72ms (0,71 Mbps) - Match
[15:59:11.428] BULK Echo #20 [USB] Stats:
[15:59:11.430]   TX: Avg 0,24ms | Min 0,17ms | Max 0,45ms
[15:59:11.432]   RX: Avg 0,66ms | Min 0,15ms | Max 0,77ms
[15:59:12.512] BULK Echo #30 [USB] TX: 64 bytes in 0,47ms (1,10 Mbps)
[15:59:12.514] BULK Echo #30 [USB] RX: 64 bytes in 0,65ms (0,79 Mbps) - Match
[15:59:12.515] BULK Echo #30 [USB] Stats:
[15:59:12.517]   TX: Avg 0,25ms | Min 0,14ms | Max 0,47ms
[15:59:12.521]   RX: Avg 0,68ms | Min 0,15ms | Max 0,77ms
```

USB Echo #30: Send 0,5ms | Recv 0,6ms

Figure 2.7 USB bulk transfer echo mode

The integration of the UART/CDC interface is simpler compared to other modules because it is well defined by the USB standards. In the main loop, the USB_DataIsReady() function checks if data is coming from the PC. If data is available, all data is put into the uart_rx_buffer with USB_GetAll(). In this project, a simple "echo" function is implemented: the received data is sent back to the PC without any changes with the USB_PutData() function.

```
if (USB_DataIsReady() != 0u) {
            uart_count = USB_GetAll(uart_rx_buffer);
            if (uart_count > 0) {
                while (USB_CDCIsReady() == 0u);
                USB_PutData(uart_rx_buffer, uart_count);
                if (UART_BUFFER_SIZE == uart_count)
                    {
                        while (0u == USB_CDCIsReady());
                        USB_PutData(NULL, 0u);
                    }
            }

   }
```



```
[13:57:15.538] UART Echo #300 [UART] RX: 36 bytes in 0,08ms (3,40 Mbps) - Match
[13:57:15.541] UART Echo #300 [UART] Stats:
[13:57:15.553]   TX: Avg 0,30ms | Min 0,17ms | Max 1,29ms
[13:57:15.556]   RX: Avg 1,55ms | Min 0,03ms | Max 7,13ms
[13:57:16.058] BULK Echo #330 [USB] TX: 64 bytes in 0,28ms (1,86 Mbps)
[13:57:16.062] BULK Echo #330 [USB] RX: 64 bytes in 0,60ms (0,85 Mbps) - Match
[13:57:16.065] BULK Echo #330 [USB] Stats:
[13:57:16.086]   TX: Avg 0,27ms | Min 0,15ms | Max 0,67ms
[13:57:16.089]   RX: Avg 0,64ms | Min 0,20ms | Max 1,09ms
[13:57:16.623] UART Echo #310 [UART] TX: 36 bytes in 0,22ms (1,30 Mbps)
[13:57:16.627] UART Echo #310 [UART] RX: 36 bytes in 5,83ms (0,05 Mbps) - Match
[13:57:16.647] UART Echo #310 [UART] Stats:
[13:57:16.650]   TX: Avg 0,30ms | Min 0,16ms | Max 1,29ms
USB Echo #392: Send 0,2ms | Recv 0,6ms
```

Figure 2.8 BULK and UART communication on paralell

The CAN-USB gateway module is the most critical industrial component of the project. This module acts as a bidirectional bridge between the CAN hardware and the USB interface.

CAN to USB (Rx Direction): This operation is performed interrupt-based. The CAN component of the PSoC triggers an interrupt when it detects a new message on the CAN line. The interrupt service routine (ISR) named CAN_ISR_Handler in the can_help.c file catches this event.

The main for(;;) loop constantly checks the can_msg_received flag. When the flag is 1, the message is received with the CAN_Receive_Message() function, converted to the 18-byte USB format with CAN_Prepare_USB_Message() and sent to the PC with the USB_LoadInEP(6, ...) command.

USB to CAN (Tx Direction): The main loop listens to the CAN specific EP7 with USB_GetEPState(7). When an 18 byte packet arrives from the PC, it is read with USB_ReadOutEP(7, ...). Then the CAN_Process_USB_Message() function parses this 18 byte raw data into a CAN_Message_t structure and sends it to the physical CAN line via the CAN_Send_Message() function.

```
CY_ISR(CAN_ISR_Handler)
{
    // ... interrupt source is checked ...

    // ID, DLC and data bytes of the incoming CAN message are read
    // and saved in a global structure 'can_received_msg'.
    can_received_msg.id = ...;
    can_received_msg.length = CAN_GET_DLC(0);
    can_received_msg.data[0] = CAN_RX_DATA_BYTE1(0);
    // ... (other data bytes) ...

    can_msg_received = 1; // The main loop is notified that there is a new message.

    CAN_INT_SR_REG.byte[1] = CAN_RX_MESSAGE_MASK; // Interrupt flag is cleared.

   }
```

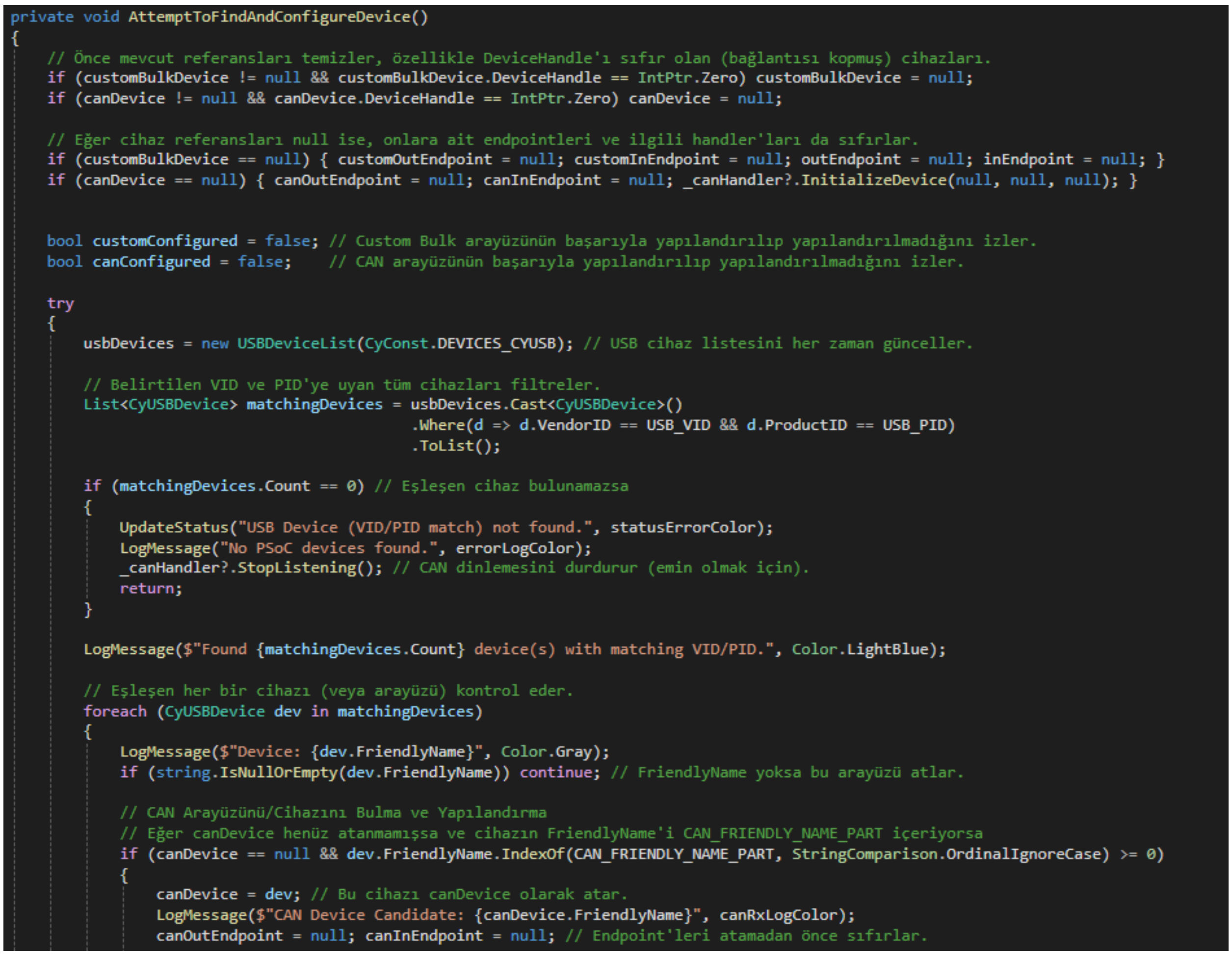
2.3.2 C# Application Architecture

The C# application on the PC side is designed in a layered structure to combine all the capabilities offered by PSoC in a user-friendly interface and to manage complex background processes.

In order for the application to communicate with the PSoC device, the CyUSB.dll library provided by Cypress is used. Device detection and management is performed with the following steps:

- Device Listing and Filtering: When the application is started or periodically, all CyUSB compatible devices connected to the system are listed with the USBDeviceList class. This list is filtered according to the project's VID (Vendor ID: 0x04B4) and PID (Product ID: 0xF001) values.
- Separation of Interfaces: Since PSoC is a Composite Device, it offers multiple interfaces with the same VID/PID. The AttemptToFindAndConfigureDevice() function checks the FriendlyName property of the found devices. The interface containing the name "asa usb bulk" is used as a special command channel; The interface with the name "asa usb CAN" is assigned as the CAN gateway channel.

- Endpoint Assignment: After the correct interfaces are found, the endpoints belonging to these interfaces are scanned. The addresses of the endpoints (0x01, 0x82, 0x86, 0x07) are checked and the relevant CyBulkEndPoint objects (customOutEndpoint, canInEndpoint, etc.) are created and their properties such as TimeOut are set. This dynamic recognition process allows the application to work flexibly without being tightly connected to the hardware.

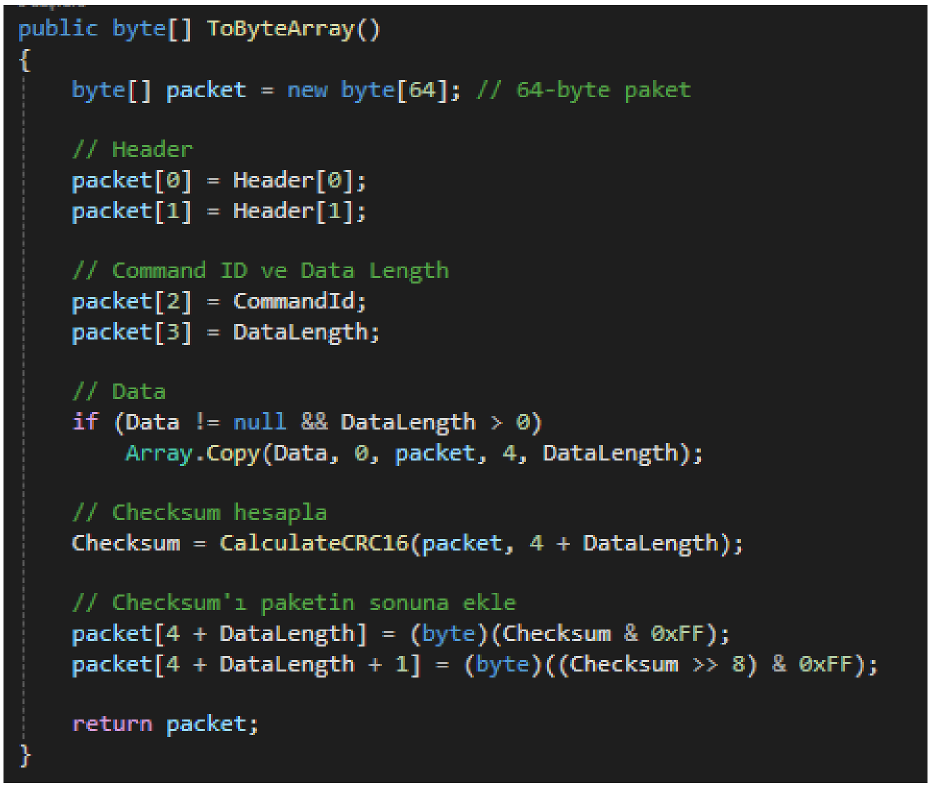


```
private void AttemptToFindAndConfigureDevice()
{
    // Önce mevcut referansları temizler, özellikle DeviceHandle'ı sıfır olan (bağlantısı kopmuş) cihazları.
    if (customBulkDevice != null && customBulkDevice.DeviceHandle == IntPtr.Zero) customBulkDevice = null;
    if (canDevice != null && canDevice.DeviceHandle == IntPtr.Zero) canDevice = null;

    // Eğer cihaz referansları null ise, onlara ait endpointleri ve ilgili handler'ları da sıfırlar.
    if (customBulkDevice == null) { customOutEndpoint = null; customInEndpoint = null; outEndpoint = null; inEndpoint = null; }
    if (canDevice == null) { canOutEndpoint = null; canInEndpoint = null; _canHandler?.InitializeDevice(null, null, null); }


    bool customConfigured = false; // Custom Bulk arayüzünün başarıyla yapılandırılıp yapılandırılmadığını izler.
    bool canConfigured = false;    // CAN arayüzünün başarıyla yapılandırılıp yapılandırılmadığını izler.

    try
    {
        usbDevices = new USBDeviceList(CyConst.DEVICES_CYUSB); // USB cihaz listesini her zaman günceller.

        // Belirtilen VID ve PID'ye uyan tüm cihazları filtreler.
        List<CyUSBDevice> matchingDevices = usbDevices.Cast<CyUSBDevice>()
                                      .Where(d => d.VendorID == USB_VID && d.ProductID == USB_PID)
                                      .ToList();

        if (matchingDevices.Count == 0) // Eşleşen cihaz bulunamazsa
        {
            UpdateStatus("USB Device (VID/PID match) not found.", statusErrorColor);
            LogMessage("No PSoC devices found.", errorLogColor);
            _canHandler?.StopListening(); // CAN dinlemesini durdurur (emin olmak için).
            return;
        }

        LogMessage($"Found {matchingDevices.Count} device(s) with matching VID/PID.", Color.LightBlue);

        // Eşleşen her bir cihazı (veya arayüzü) kontrol eder.
        foreach (CyUSBDevice dev in matchingDevices)
        {
            LogMessage($"Device: {dev.FriendlyName}", Color.Gray);
            if (string.IsNullOrEmpty(dev.FriendlyName)) continue; // FriendlyName yoksa bu arayüzü atlar.

            // CAN Arayüzünü/Cihazını Bulma ve Yapılandırma
            // Eğer canDevice henüz atanmamışsa ve cihazın FriendlyName'i CAN_FRIENDLY_NAME_PART içeriyorsa
            if (canDevice == null && dev.FriendlyName.IndexOf(CAN_FRIENDLY_NAME_PART, StringComparison.OrdinalIgnoreCase) >= 0)
            {
                canDevice = dev; // Bu cihazı canDevice olarak atar.
                LogMessage($"CAN Device Candidate: {canDevice.FriendlyName}", canRxLogColor);
                canOutEndpoint = null; canInEndpoint = null; // Endpoint'leri atamadan önce sıfırlar.
```

Figure 2.9 Find the device matching with VID/PID

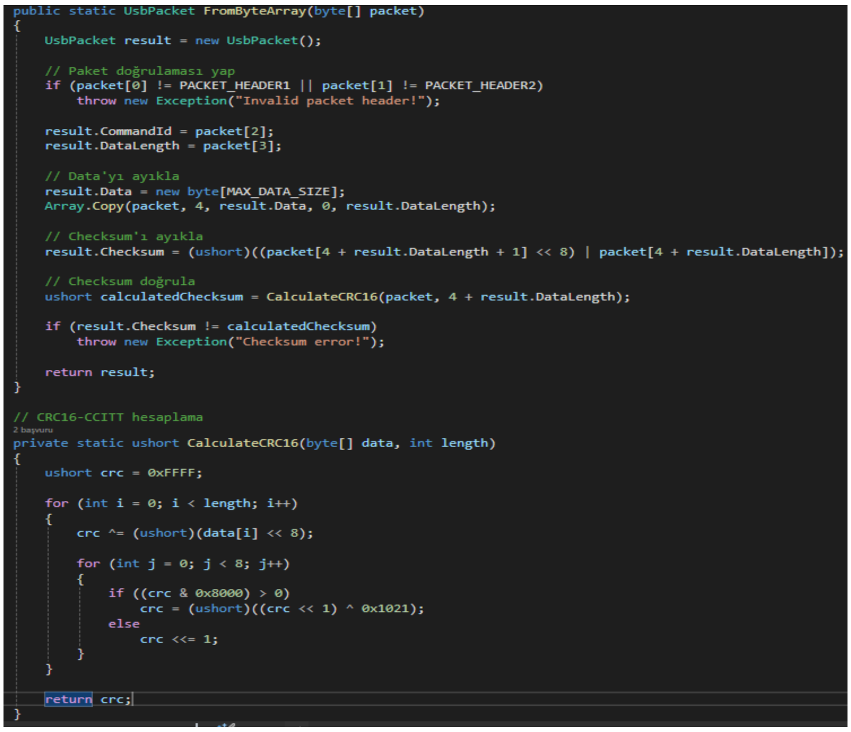
A UsbPacket.cs class with the same fields has been created to manage the UsbPacket structure in the PSoC on the PC side.

- ToByteArray(): This method converts a UsbPacket object into a 64-byte byte[] array ready to be sent to the PSoC. During this process, it also calculates the CRC16 checksum and adds it to the end of the packet.
- FromByteArray(byte[] packet): Receives the raw byte[] array from the PSoC, performs header checks, parses the fields, and most importantly verifies the data integrity by comparing the incoming CRC with the CRC it calculates. If it fails, throws an Exception.
- ParseContent(): Converts the content of an incoming response packet into a readable text (string) to be displayed to the user, interpreting it according to the command type. For example, for the CMD_VERSION response, it produces the text "Version: v1.0.0".



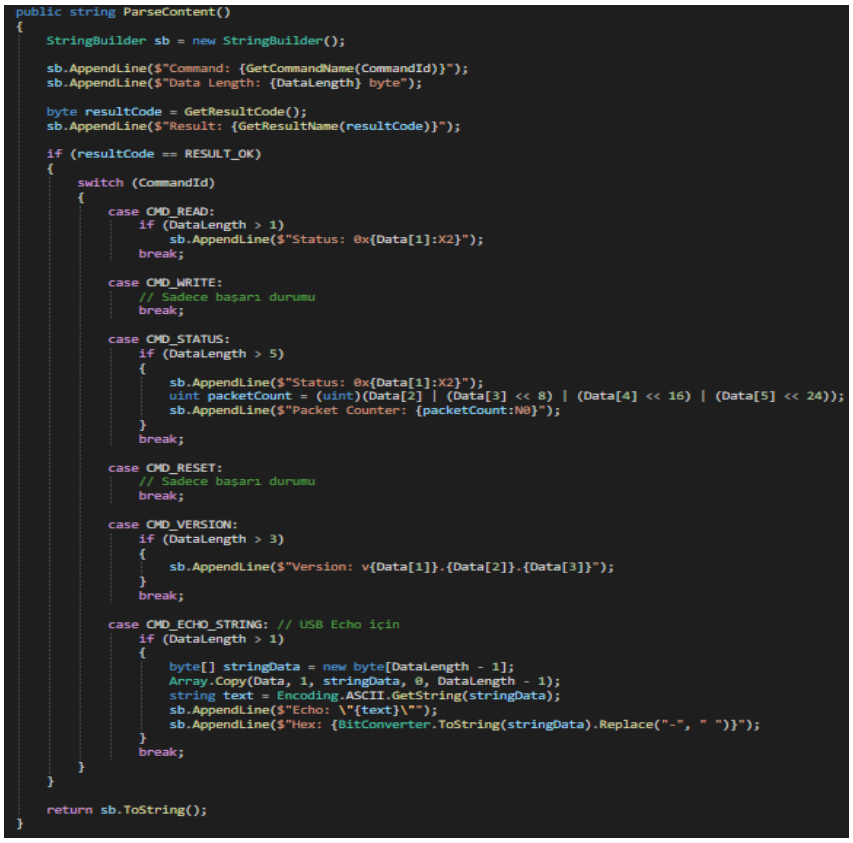
```
public byte[] ToByteArray()
{
    byte[] packet = new byte[64]; // 64-byte paket

    // Header
    packet[0] = Header[0];
    packet[1] = Header[1];

    // Command ID ve Data Length
    packet[2] = CommandId;
    packet[3] = DataLength;

    // Data
    if (Data != null && DataLength > 0)
        Array.Copy(Data, 0, packet, 4, DataLength);

    // Checksum hesapla
    Checksum = CalculateCRC16(packet, 4 + DataLength);

    // Checksum'ı paketin sonuna ekle
    packet[4 + DataLength] = (byte)(Checksum & 0xFF);
    packet[4 + DataLength + 1] = (byte)((Checksum >> 8) & 0xFF);

    return packet;
}
```

Figure 2.10 C# ToByteArray()



```
public static UsbPacket FromByteArray(byte[] packet)
{
    UsbPacket result = new UsbPacket();

    // Paket doğrulaması yap
    if (packet[0] != PACKET_HEADER1 || packet[1] != PACKET_HEADER2)
        throw new Exception("Invalid packet header!");

    result.CommandId = packet[2];
    result.DataLength = packet[3];

    // Data'yı ayıkla
    result.Data = new byte[MAX_DATA_SIZE];
    Array.Copy(packet, 4, result.Data, 0, result.DataLength);

    // Checksum'ı ayıkla
    result.Checksum = (ushort)((packet[4 + result.DataLength + 1] << 8) | packet[4 + result.DataLength]);

    // Checksum doğrula
    ushort calculatedChecksum = CalculateCRC16(packet, 4 + result.DataLength);

    if (result.Checksum != calculatedChecksum)
        throw new Exception("Checksum error!");

    return result;
}

// CRC16-CCITT hesaplama
2 başvuru
private static ushort CalculateCRC16(byte[] data, int length)
{
    ushort crc = 0xFFFF;

    for (int i = 0; i < length; i++)
    {
        crc ^= (ushort)(data[i] << 8);

        for (int j = 0; j < 8; j++)
        {
            if ((crc & 0x8000) > 0)
                crc = (ushort)((crc << 1) ^ 0x1021);
            else
                crc <<= 1;
        }
    }

    return crc;
}
```

Figure 2.11 C# FromByteArray and Calculate CRC functions

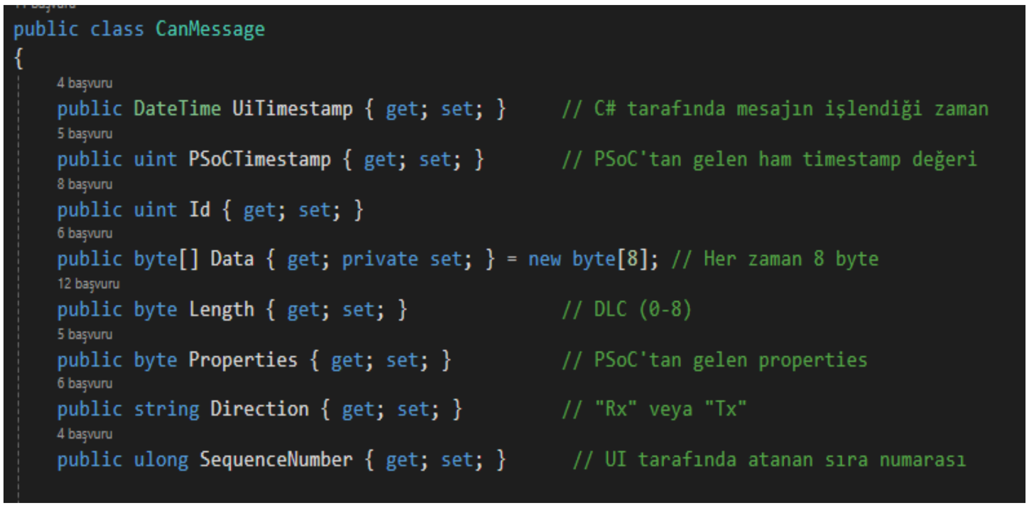


```
public string ParseContent()
{
    StringBuilder sb = new StringBuilder();

    sb.AppendLine($"Command: {GetCommandName(CommandId)}");
    sb.AppendLine($"Data Length: {DataLength} byte");

    byte resultCode = GetResultCode();
    sb.AppendLine($"Result: {GetResultName(resultCode)}");

    if (resultCode == RESULT_OK)
    {
        switch (CommandId)
        {
            case CMD_READ:
                if (DataLength > 1)
                    sb.AppendLine($"Status: 0x{Data[1]:X2}");
                break;

            case CMD_WRITE:
                // Sadece başarı durumu
                break;

            case CMD_STATUS:
                if (DataLength > 5)
                {
                    sb.AppendLine($"Status: 0x{Data[1]:X2}");
                    uint packetCount = (uint)(Data[2] | (Data[3] << 8) | (Data[4] << 16) | (Data[5] << 24));
                    sb.AppendLine($"Packet Counter: {packetCount:N0}");
                }
                break;

            case CMD_RESET:
                // Sadece başarı durumu
                break;

            case CMD_VERSION:
                if (DataLength > 3)
                {
                    sb.AppendLine($"Version: v{Data[1]}.{Data[2]}.{Data[3]}");
                }
                break;

            case CMD_ECHO_STRING: // USB Echo için
                if (DataLength > 1)
                {
                    byte[] stringData = new byte[DataLength - 1];
                    Array.Copy(Data, 1, stringData, 0, DataLength - 1);
                    string text = Encoding.ASCII.GetString(stringData);
                    sb.AppendLine($"Echo: \"{text}\"");
                    sb.AppendLine($"Hex: {BitConverter.ToString(stringData).Replace("-", " ")}");
                }
                break;
        }
    }

    return sb.ToString();
}
```

Figure 2.12 C# ParseContent function

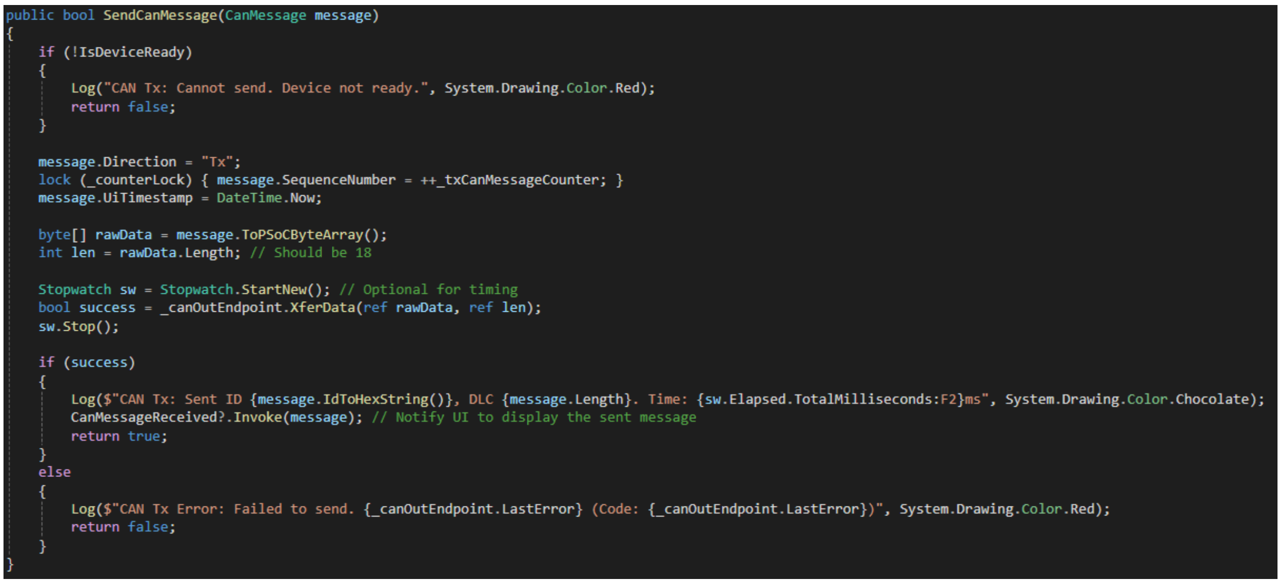
CAN communication can have a high-frequency data flow that can contain hundreds of messages per second. To prevent this flow from clogging the main UI thread, all CAN operations are grouped into a separate class called CanHandler.cs and managed using background threads.

- CanHandler Class: This class receives and manages CAN-specific endpoints (_canInEndpoint, _canOutEndpoint).
- ListenLoop() Method: When StartListening() is called, this method starts working in a new background thread with Task.Factory.StartNew. It waits for data to arrive from the PSoC by calling the _canInEndpoint.XferData() method in an infinite loop. When an 18-byte packet arrives, it converts this data into a CanMessage object with the CanMessage.FromPSoCByteArray() method.
- UI Update (Event-Based Architecture): The CanMessage object created in ListenLoop does not directly access the main interface. Instead, it triggers an event called CanMessageReceived. MainForm subscribes to this event (_canHandler.CanMessageReceived += ...). When the event is triggered, the event handler in MainForm safely posts the UI update operation (e.g. adding a new row to the ListView) to the main UI thread using this.BeginInvoke(). This design keeps the application fluid and responsive even under high CAN traffic.
- SendCanMessage() Method: This method is called when the user wants to send a CAN message from the interface. It creates the required CanMessage object, serializes it with ToPSoCByteArray() and sends it to the PSoC with _canOutEndpoint.XferData().



```
public class CanMessage
{
    4 başvuru
    public DateTime UiTimestamp { get; set; }     // C# tarafında mesajın işlendiği zaman
    5 başvuru
    public uint PSoCTimestamp { get; set; }       // PSoC'tan gelen ham timestamp değeri
    8 başvuru
    public uint Id { get; set; }
    6 başvuru
    public byte[] Data { get; private set; } = new byte[8]; // Her zaman 8 byte
    12 başvuru
    public byte Length { get; set; }              // DLC (0-8)
    5 başvuru
    public byte Properties { get; set; }          // PSoC'tan gelen properties
    6 başvuru
    public string Direction { get; set; }         // "Rx" veya "Tx"
    4 başvuru
    public ulong SequenceNumber { get; set; }      // UI tarafında atanan sıra numarası
```

Figure 2.13 CAN message structure



```
public bool SendCanMessage(CanMessage message)
{
    if (!IsDeviceReady)
    {
        Log("CAN Tx: Cannot send. Device not ready.", System.Drawing.Color.Red);
        return false;
    }

    message.Direction = "Tx";
    lock (_counterLock) { message.SequenceNumber = ++_txCanMessageCounter; }
    message.UiTimestamp = DateTime.Now;

    byte[] rawData = message.ToPSoCByteArray();
    int len = rawData.Length; // Should be 18

    Stopwatch sw = Stopwatch.StartNew(); // Optional for timing
    bool success = _canOutEndpoint.XferData(ref rawData, ref len);
    sw.Stop();

    if (success)
    {
        Log($"CAN Tx: Sent ID {message.IdToHexString()}, DLC {message.Length}. Time: {sw.Elapsed.TotalMilliseconds:F2}ms", System.Drawing.Color.Chocolate);
        CanMessageReceived?.Invoke(message); // Notify UI to display the sent message
        return true;
    }
    else
    {
        Log($"CAN Tx Error: Failed to send. {_canOutEndpoint.LastError} (Code: {_canOutEndpoint.LastError})", System.Drawing.Color.Red);
        return false;
    }
}
```

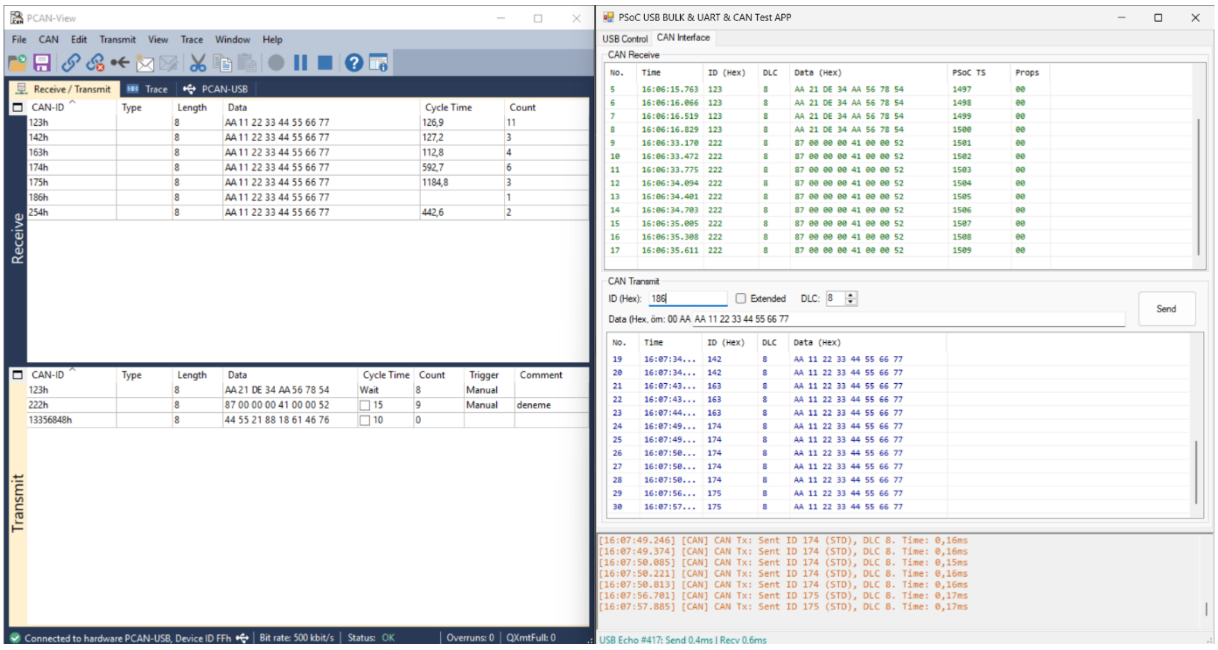
Figure 2.14 SendCanMessage function

The user interface of the application is designed to present complex functions in a simple and understandable way:

- Tabbed Interface: Each main function (USB Control and CAN) is grouped under a separate tab.
- Real Time Logging: All important events (device connection, command sending, errors) are written to the main log screen with a time stamp. Log messages are colored

according to their importance (usbEchoLogColor, errorLogColor, etc.) to increase readability.

- CAN Interface: Separate ListViews are used for received (Rx) and sent (Tx) messages. These lists show messages with columns such as sequence number, time, ID, DLC and data. This mimics the appearance of an industry standard CAN analyzer.
- Dynamic Controls: The ComboBox in the special command tab lists all commands that can be sent. The text of the "Send" button changes dynamically depending on the selected command (e.g. "Start USB Echo").



CAN-ID	Type	Length	Data	Cycle Time	Count
123h		8	AA 11 22 33 44 55 66 77	126,9	11
142h		8	AA 11 22 33 44 55 66 77	127,2	3
163h		8	AA 11 22 33 44 55 66 77	112,8	4
174h		8	AA 11 22 33 44 55 66 77	592,7	6
175h		8	AA 11 22 33 44 55 66 77	1184,8	3
186h		8	AA 11 22 33 44 55 66 77		1
254h		8	AA 11 22 33 44 55 66 77	442,6	2

CAN-ID	Type	Length	Data	Cycle Time	Count	Trigger	Comment
123h		8	AA 21 DE 34 AA 56 78 54	Wait	8	Manual	
222h		8	87 00 00 00 41 00 00 52	☐ 15	9	Manual	deneme
13356848h		8	44 55 21 88 18 61 46 76	☐ 10	0		

Connected to hardware PCAN-USB, Device ID FFh | Bit rate: 500 kbit/s | Status: OK | Overruns: 0 | QXmtFull: 0

No.	Time	ID (Hex)	DLC	Data (Hex)	PSoC TS	Props
5	16:06:15.763	123	8	AA 21 DE 34 AA 56 78 54	1497	00
6	16:06:16.066	123	8	AA 21 DE 34 AA 56 78 54	1498	00
7	16:06:16.519	123	8	AA 21 DE 34 AA 56 78 54	1499	00
8	16:06:16.829	123	8	AA 21 DE 34 AA 56 78 54	1500	00
9	16:06:33.170	222	8	87 00 00 00 41 00 00 52	1501	00
10	16:06:33.472	222	8	87 00 00 00 41 00 00 52	1502	00
11	16:06:33.775	222	8	87 00 00 00 41 00 00 52	1503	00
12	16:06:34.094	222	8	87 00 00 00 41 00 00 52	1504	00
13	16:06:34.401	222	8	87 00 00 00 41 00 00 52	1505	00
14	16:06:34.703	222	8	87 00 00 00 41 00 00 52	1506	00
15	16:06:35.005	222	8	87 00 00 00 41 00 00 52	1507	00
16	16:06:35.308	222	8	87 00 00 00 41 00 00 52	1508	00
17	16:06:35.611	222	8	87 00 00 00 41 00 00 52	1509	00

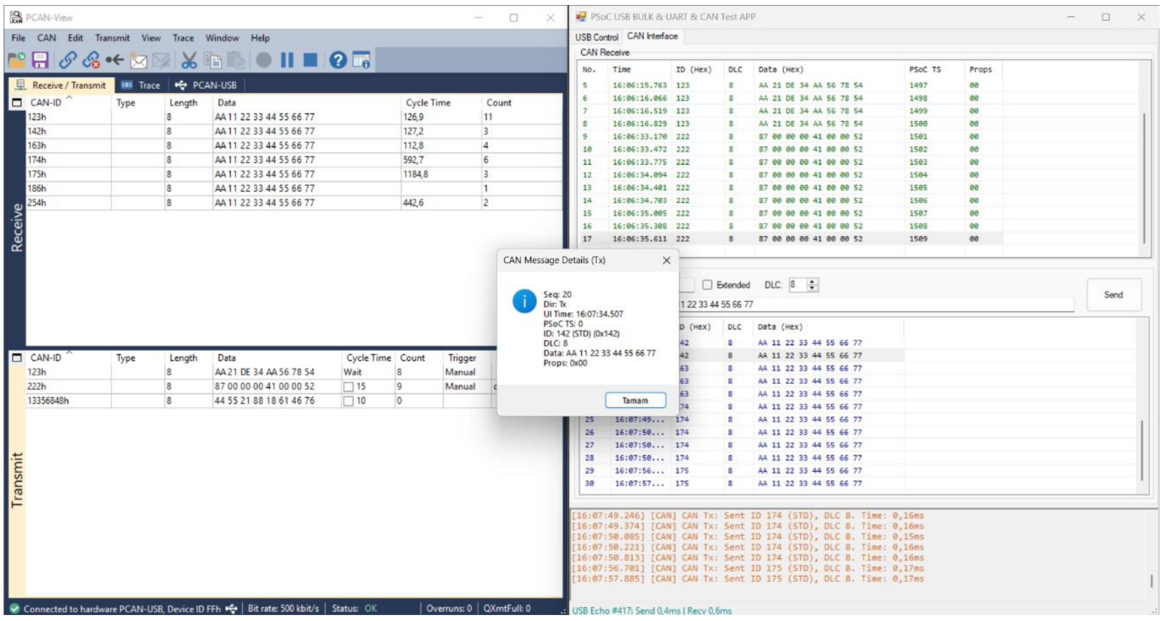
ID (Hex): 186 ☐ Extended DLC: 8
Data (Hex, öm: 00 AA AA 11 22 33 44 55 66 77

No.	Time	ID (Hex)	DLC	Data (Hex)
19	16:07:34...	142	8	AA 11 22 33 44 55 66 77
20	16:07:34...	142	8	AA 11 22 33 44 55 66 77
21	16:07:43...	163	8	AA 11 22 33 44 55 66 77
22	16:07:43...	163	8	AA 11 22 33 44 55 66 77
23	16:07:44...	163	8	AA 11 22 33 44 55 66 77
24	16:07:49...	174	8	AA 11 22 33 44 55 66 77
25	16:07:49...	174	8	AA 11 22 33 44 55 66 77
26	16:07:50...	174	8	AA 11 22 33 44 55 66 77
27	16:07:50...	174	8	AA 11 22 33 44 55 66 77
28	16:07:50...	174	8	AA 11 22 33 44 55 66 77
29	16:07:56...	175	8	AA 11 22 33 44 55 66 77
30	16:07:57...	175	8	AA 11 22 33 44 55 66 77

```
[16:07:49.246] [CAN] CAN Tx: Sent ID 174 (STD), DLC 8. Time: 0,16ms
[16:07:49.374] [CAN] CAN Tx: Sent ID 174 (STD), DLC 8. Time: 0,16ms
[16:07:50.085] [CAN] CAN Tx: Sent ID 174 (STD), DLC 8. Time: 0,15ms
[16:07:50.221] [CAN] CAN Tx: Sent ID 174 (STD), DLC 8. Time: 0,16ms
[16:07:50.813] [CAN] CAN Tx: Sent ID 174 (STD), DLC 8. Time: 0,16ms
[16:07:56.701] [CAN] CAN Tx: Sent ID 175 (STD), DLC 8. Time: 0,17ms
[16:07:57.885] [CAN] CAN Tx: Sent ID 175 (STD), DLC 8. Time: 0,17ms
```

USB Echo #417: Send 0.4ms | Recv 0.6ms

Figure 2.15 CAN message transmit and receive with PCAN-View app



CAN-ID	Type	Length	Data	Cycle Time	Count
123h		8	AA 11 22 33 44 55 66 77	126,9	11
142h		8	AA 11 22 33 44 55 66 77	127,2	3
163h		8	AA 11 22 33 44 55 66 77	112,8	4
174h		8	AA 11 22 33 44 55 66 77	592,7	6
175h		8	AA 11 22 33 44 55 66 77	1184,8	3
186h		8	AA 11 22 33 44 55 66 77		1
254h		8	AA 11 22 33 44 55 66 77	442,6	2

CAN-ID	Type	Length	Data	Cycle Time	Count	Trigger
123h		8	AA 21 DE 34 AA 56 78 54	Wait	8	Manual
222h		8	87 00 00 00 41 00 00 52	☐ 15	9	Manual
13356848h		8	44 55 21 88 18 61 46 76	☐ 10	0	

Connected to hardware PCAN-USB, Device ID FFh | Bit rate: 500 kbit/s | Status: OK | Overruns: 0 | QXmtFull: 0

CAN Message Details (Tx)

Seq: 20
Dir: Tx
UI Time: 16:07:34.507
PSoC TS: 0
ID: 142 (STD) (0x142)
DLC: 8
Data: AA 11 22 33 44 55 66 77
Props: 0x00

No.	Time	ID (Hex)	DLC	Data (Hex)	PSoC TS	Props
5	16:06:15.763	123	8	AA 21 DE 34 AA 56 78 54	1497	00
6	16:06:16.066	123	8	AA 21 DE 34 AA 56 78 54	1498	00
7	16:06:16.519	123	8	AA 21 DE 34 AA 56 78 54	1499	00
8	16:06:16.829	123	8	AA 21 DE 34 AA 56 78 54	1500	00
9	16:06:33.170	222	8	87 00 00 00 41 00 00 52	1501	00
10	16:06:33.472	222	8	87 00 00 00 41 00 00 52	1502	00
11	16:06:33.775	222	8	87 00 00 00 41 00 00 52	1503	00
12	16:06:34.094	222	8	87 00 00 00 41 00 00 52	1504	00
13	16:06:34.401	222	8	87 00 00 00 41 00 00 52	1505	00
14	16:06:34.703	222	8	87 00 00 00 41 00 00 52	1506	00
15	16:06:35.005	222	8	87 00 00 00 41 00 00 52	1507	00
16	16:06:35.308	222	8	87 00 00 00 41 00 00 52	1508	00
17	16:06:35.611	222	8	87 00 00 00 41 00 00 52	1509	00

1 22 33 44 55 66 77

No.	Time	D (Hex)	DLC	Data (Hex)
		42	8	AA 11 22 33 44 55 66 77
		42	8	AA 11 22 33 44 55 66 77
		63	8	AA 11 22 33 44 55 66 77
		63	8	AA 11 22 33 44 55 66 77
		63	8	AA 11 22 33 44 55 66 77
		74	8	AA 11 22 33 44 55 66 77
25	16:07:49...	174	8	AA 11 22 33 44 55 66 77
26	16:07:50...	174	8	AA 11 22 33 44 55 66 77
27	16:07:50...	174	8	AA 11 22 33 44 55 66 77
28	16:07:50...	174	8	AA 11 22 33 44 55 66 77
29	16:07:56...	175	8	AA 11 22 33 44 55 66 77
30	16:07:57...	175	8	AA 11 22 33 44 55 66 77

```
[16:07:49.246] [CAN] CAN Tx: Sent ID 174 (STD), DLC 8. Time: 0,16ms
[16:07:49.374] [CAN] CAN Tx: Sent ID 174 (STD), DLC 8. Time: 0,16ms
[16:07:50.085] [CAN] CAN Tx: Sent ID 174 (STD), DLC 8. Time: 0,15ms
[16:07:50.221] [CAN] CAN Tx: Sent ID 174 (STD), DLC 8. Time: 0,16ms
[16:07:50.813] [CAN] CAN Tx: Sent ID 174 (STD), DLC 8. Time: 0,16ms
[16:07:56.701] [CAN] CAN Tx: Sent ID 175 (STD), DLC 8. Time: 0,17ms
[16:07:57.885] [CAN] CAN Tx: Sent ID 175 (STD), DLC 8. Time: 0,17ms
```

USB Echo #417: Send 0,4ms | Recv 0,6ms

Figure 2.16 CAN communication with PCAN-View app

2.3.3 Challenges Encountered and Solutions Implemented

The most critical challenge encountered in the third week of the project was that the Windows operating system did not recognize the developed PSoC device. The device, listed as "Unknown Device" in Device Manager, could not be communicated with in any way.

It was determined that the problem was not caused by the PSoC or the C# code, but rather by the operating system not knowing which driver to assign to this special VID/PID combination. Since the PSoC was configured as a "Vendor-Specific" device, it did not have a standard driver.

First, Cypress's standard driver package was installed, but this package did not work because it did not contain the project's special VID/PID.

A custom .inf driver file was attempted to be written from scratch, but this attempt failed due to inexperience in driver signing and configuration.

As a more practical and effective solution, the .inf file of a signed and working driver included in Cypress's EZ-USB FX3 SDK was opened with a text editor. A new line was manually added to the list of hardware IDs of this file, containing the project's VID (0x04B4) and PID (0xF001). When the driver was reinstalled from this modified .inf file, Windows now correctly recognized the device and successfully assigned the CyUSB.sys driver. This solution allowed to overcome the problem by modifying an existing and reliable infrastructure without going into the complex driver development process. Example .inf file. The red rows are the devices we want to use:

```
;for x86 platforms
[Device.NTx86]
...
%VID_04B4&PID_5219&MI_03.DeviceDesc%=CyUsb3, USB\VID_04B4&PID_5219&MI_03
%VID_04B4&PID_00FB&MI_02.DeviceDesc%=CyUsb3, USB\VID_04B4&PID_00FB&MI_02
%VID_04B4&PID_0033&MI_01.DeviceDesc%=CyUsb3, USB\VID_04B4&PID_0033&MI_01
%VID_04B4&PID_F001&MI_00.DeviceDesc%=CyUsb3, USB\VID_04B4&PID_F001&MI_00
%VID_04B4&PID_F001&MI_03.DeviceDesc%=CyUsb3, USB\VID_04B4&PID_F001&MI_03

;for x64 platforms
[Device.NTamd64]
...
%VID_04B4&PID_00FB&MI_02.DeviceDesc%=CyUsb3, USB\VID_04B4&PID_00FB&MI_02
%VID_04B4&PID_0033&MI_01.DeviceDesc%=CyUsb3, USB\VID_04B4&PID_0033&MI_01
%VID_04B4&PID_F001&MI_00.DeviceDesc%=CyUsb3, USB\VID_04B4&PID_F001&MI_00
%VID_04B4&PID_F001&MI_03.DeviceDesc%=CyUsb3, USB\VID_04B4&PID_F001&MI_03

[Strings]
CYUSB3_Provider    = "Cypress"
CYUSB3_Company     = "Cypress Semiconductor Corporation"
CYUSB3_Description = "Cypress Generic USB3.0 Driver"
VID_04B4&PID_F650&MI_00.DeviceDesc="CCG3 I2CM Bridge Vendor"
VID_04B4&PID_0033&MI_01.DeviceDesc="USB-Serial (Single Channel) Vendor MFG"
VID_04B4&PID_F001&MI_00.DeviceDesc="asa usb bulk"
VID_04B4&PID_F001&MI_03.DeviceDesc="asa usb CAN"
```

The project had to manage three different communication channels simultaneously. This had the potential to negatively affect the performance of both the PSoC and the PC application, especially when processing high-frequency CAN data.

Since no real-time operating system (RTOS) was used in PSoC, a simple and effective "super loop" and "event polling" architecture was adopted. The main for(;;) loop checks the status of each interface (Bulk, CAN, UART) in turn and very quickly. If there is data in an interface, the relevant operation is performed and the loop continues. This non-blocking approach ensures that PSoC manages all tasks fairly on a single thread and that no thread keeps the other waiting for a long time. Time-critical events such as CAN data are caught with interrupts and prioritized.

In the PC application, the biggest risk was that the high data flow would freeze the user interface (UI). To solve this problem, a multi-threading architecture was used. The CanHandler class was designed to execute all CAN listening operations on a background Task independent of the main UI thread. This background thread receives the data from the PSoC and processes it. When the processed data needs to be displayed on the UI (e.g. ListView), the Control.BeginInvoke() method was used to ensure thread safety and prevent conflicts. This method safely adds the UI update request to the message queue of the main UI thread. In this way, the application remained fluent and instantly responsive to user commands, even when processing hundreds of CAN messages per second.

2.4 Conclusion and Evaluation

This internship project was initiated with the aim of simplifying the interaction of the software team with Electronic Control Units (ECU) and making the testing processes efficient. The main objective of the project; to develop an integrated hardware and software solution that can manage CAN, custom command-based Bulk transfer and standard UART/CDC communication over a single USB connection has been successfully achieved.

The success criteria defined at the beginning of the project were completely met:

- Multi-Function Communication: Cypress PSoC 5LP was successfully configured as a Composite USB Device offering three independent interfaces (CAN gateway, custom Bulk protocol, UART/CDC). All these channels could be operated simultaneously without any conflict or performance degradation in the system.
- Reliable Data Transfer: The UsbPacket protocol designed for custom Bulk transfer guaranteed data integrity thanks to the CRC16 checksum, ensuring that commands and responses were transmitted without errors.
- Industry Standard Compatibility: The developed CAN-USB gateway module has proven to communicate seamlessly bidirectionally with an industry standard analysis tool such as PCAN-View. Similarly, the UART/CDC interface has also been compatible with standard terminal programs.
- User-Friendly Centralized Management: The developed C# desktop application automatically recognizes the PSoC device and gathers all these complex functions under a single and centralized interface. The use of background threads has ensured that the application remains fluid and responsive even under high data traffic.

As a result, the project has transformed test environment into an efficient tool that can be managed with a single cable and software.

REFERENCES

1. Infineon Technologies AG. (2018). *AN57294: USB 101: An Introduction to Universal Serial Bus 2.0*.
2. Axelson, J. (2015). *USB Complete: The Developer's Guide* (5). Lakeview Research LLC.
3. Infineon Technologies AG. (2022, February). *AN56377: PSoC® 3 and PSoC® 5LP Introduction to Implementing USB Data Transfers*.
4. Massachusetts Institute of Technology. (t.y.). *USB in a Nutshell*.
5. USB Implementers Forum. (2000). *Universal Serial Bus Specification, Revision 2.0*.
6. Microsoft. *USB Composite Devices*. Microsoft Docs.
7. Infineon Technologies AG. (2022, Mart). *AN75705: Getting Started with EZ-USB™ FX3™*.
8. Bosch. (1991). *CAN Specification Version 2.0*. Robert Bosch GmbH.
9. Texas Instruments. (2016, Mart). *Introduction to the Controller Area Network (CAN)*.
10. Kvaser. (t.y.). *The CAN Protocol*.
11. Microchip Technology Inc. (2002). *AN754: Understanding Microchip's CAN Module Bit Timing*.
12. CAN in Automation (CiA). (t.y.). *CAN error handling*.
13. SparkFun Electronics. (t.y.). *Serial Communication*. SparkFun Learn.
14. USB Implementers Forum. (1998). *Class Definitions for Communication Devices, Version 1.1*.
15. Infineon Technologies AG. (2018). *AN61223: PSoC® 3 and PSoC 5LP - Implementing a Virtual COM Port using USB-UART*.