Bremerhaven University of Applied Sciences

ASCII Acknowledgment response system for G4 product platforms

Master Thesis

Submitted in Fulfilment of the Requirements for Academic Degree

M.Sc. in Embedded Systems Design

Joe Joseph Nelson

Matriculation number: **38750**

Course of Study: Embedded Systems Design

First Examiner: Prof. Dr. Uwe Werner (Hochschule Bremerhaven)

Second Examiner: Dipl. Manfred Huber (Wenglor MEL GmbH)

Submission Date: 06.12 .2024

ACKNOWLEDGEMENTS

This Master Thesis would not have been possible without the help and support from several individuals. I would like to take this opportunity to express my gratitude and appreciation to everyone who contributed to the successful completion of my Master Thesis.

Firstly, I would like to express my sincere gratitude to my external supervisor Dipl. Manfred Huber, my Team leader Mahesh Ramesh Ingle, and my supervising support Jajwalya Waghmare for their guidance, support, and insightful feedback throughout the entire development process. Their expertise and dedication have been pivotal in shaping this thesis.

I am extremely grateful to my academic supervisor, Prof. Dr.-Ing. Uwe Werner, for supervising me and guiding me with his invaluable guidance, advice, and feedback at all stages of this thesis.

I extend my heartfelt thanks to Wenglor MEL GmbH for giving me this opportunity to write my master thesis with them and providing the necessary resources and facilities for my research. My colleagues and my Development (High precision) teammates at Wenglor have been always there for me whenever I was in need for help, and I thank them all for the helping hands offered.

My academic journey would have never been such a wonderful experience without the support of my family, girlfriend, and friends. I thank them all for their never-ending encouragement and belief in my abilities.

Above all, I thank God for giving me the strength and wisdom throughout this journey to complete this thesis.

Thank you all for being a part of this significant milestone in my academic journey.

ABSTRACT

The thesis explores the idea of an ASCII Acknowledgment System proposed for sensors under G4 product platform, a cutting-edge technology in 3D machine vision. These sensors integrate a central processing unit, a field-programmable gate array (FPGA) for real-time processing, sensors and peripherals.

The ASCII Acknowledgment System that is proposed in this thesis achieves its aim to create a robust modular acknowledgment response system for the sensors under G4 product platform. The system facilitates seamless interactions that enable users to effectively monitor the device's operations. Each functionality of the device after execution is communicated to the user in an XML Container format.

An XML-based acknowledgment format was adopted to enhance interoperability with external systems by providing a structured representation of command responses, including current value, user value, minimum and maximum value limits, execution time and command name tags. Through this thesis, a study and demonstration on how the ASCII Acknowledgment System could be developed and the factors that should be considered for its development so that the device works synchronously with the feedback system is showcased.

Firmware architecture modifications were implemented to make the systems more modular and reliable. This included a common command handler which acts as a centralized registry for all ASCII commands. The system was rigorously tested for performance, including unit testing, integration testing, and stress testing under real-world conditions. Results demonstrated high reliability, low latency, and error handling, with acknowledgment responses produced without delay.

This Master thesis work lays the foundation for scalable acknowledgment systems in embedded systems, with applications in industrial automation, sensor networks, and real-time control systems. Future enhancements include optimizing performance for large-scale systems, implementing logical steps for acknowledgments of functionalities which are difficult to predict, and adding more information parameters to the acknowledgment response that would help for debugging purposes.

DECLARATION OF ORIGINALITY

I Joe Joseph Nelson hereby declare that this thesis and the work reported herein is my own work. I conducted all the activities under supervision of my supervisor. All the information's taken from published and unpublished work of others have been acknowledged in the body and references are given in bibliography section at the end.

Place and Date:

Munich, 06.12.2024

Signature:

Joe Joseph Nelson

CONTENTS

LIST OF FIGURES

LIST OF ACRONYMS

FPGA	Field Programmable Gate Array
MLAS	Code name for sensor
MLBS	Code name for senor
G4	Code name for the platform
ASCII	American Standard Code for Information Interchange
XML	Extensible Markup Language
PS	Processing System
PL	Programmable Logic
SDK	Software Development Kit
CMD	Command
TCP	Transmission Control Protocol
IP	Internet Protocol
UDP	User Datagram Protocol
MQTT	Message Queuing Telemetry Transport
HTTP	Hyper Text Transfer Protocol
SMTP	Simple Mail Transfer Protocol
FTP	File Transfer Protocol
FDDI	Fiber Distributed Data Interface
VTAM	Virtual Telecommunication Access Method
SNA	System Network Architecture
COTS	Connection Oriented Transport Service
TELNET	Teletype Network
NFS	Network File Server
DNS	Domain Name Server
RPC	Remote Procedural Call
UNIX	UNiplexed Information Computing System
OS	Operating System
API	Application Programming Interface
FIFO	First In First Out
IPC	Inter-Process Communication
IPv4	Internet Protocol version 4
SYN	Synchronise
ACK	Acknowledgment
QoS	Quality of service
IDE	Integrated Development Environment
SSH	Secure Shell
SCP	Secure Copy Protocol
SFTP	Secure File Transfer Protocol
CRC	Cyclic Redundancy Check
nc	NetCat
strcmp	String comparison
CRLF	Carriage Return Line Feed

Chapter 1

INTRODUCTION

The Shape Drive G4 3D sensors are state-of-the-art devices in the field of 3D machine vision. They are known for their high-resolution and high-speed measurement capabilities. As pioneers in 3D vision technology, the Shape Drive G4 sensors are distinguished by their ability to convert physical objects into 3D digital models, which is essential for numerous applications, including quality inspection, and automation. The combination and integration of advanced features, such as high-speed Ethernet interfaces of up to 10 Gbit/s and onboard 3D point cloud processing using FPGAs, help in capturing highly efficient real-time data that is then collected and processed. This makes the G4 series an excellent solution for industrial applications that demand accuracy and speed like bin picking, precise meat cutting and similar applications.

The fundamental working principle of the 3D machine vision sensor Shape Drive G4 is based on **triangulation** and **structured light projection**, which together help in precise, non-contact measurement. The system works by projecting several patterns of light onto an object, which is then captured synchronously using a camera. Now these images which are the captured images are processed using sophisticated algorithms to generate a 3D point cloud representation of the object's surface. This structured approach to measurement ensures both accuracy and reliability, making the Shape Drive G4 sensors particularly effective for applications in robotics, automated production, and quality assurance.

The G4 sensors which are available as MLAS and MLBS also use the latest and the best available in processing technology. Both MLAS and MLBS differ only in their range of operation. The use of an onboard FPGA (Field Programmable Gate Array) programmable logic hardware to achieve rapid computation of 3D models showcases the advanced usage of technology directly within the device. This architecture allows for data to be processed almost instantaneously, providing users with precise 3D data without reliance on external computing resources. The combination of sensor technology and variable measuring ranges ensures that these sensors can always be tailored to meet specific user needs, enhancing the versatility of the platform for different industrial requirements.

Despite the sophisticated hardware, the effectiveness of the Shape Drive G4 (figure 1) is equally dependent on the software (processing system) that controls and configures it. The communication protocols and software allow the sensor to interact with users or external systems. To facilitate this efficient interaction and to ensure that the sensor works efficiently, an **ASCII acknowledgment response system is to be implemented as part of the G4 sensor's firmware**. This system provides a reliable method for verifying the successful execution of commands, thus improving the user-friendliness and reliability of the sensors.

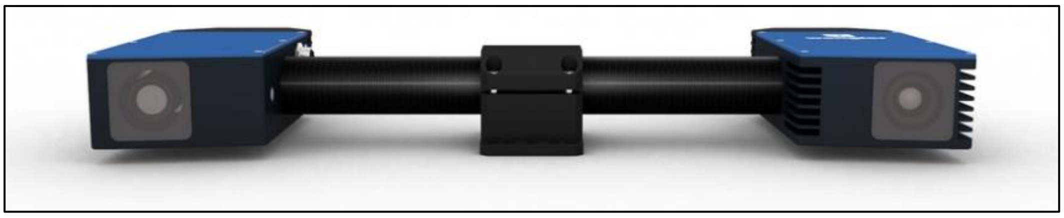


Figure 1 MLBS 3D Sensor [1]

1.1 Motivation and Problem Statement

With increasing demand for real-time data acquisition and flexible configurability, it is vital to ensure that complex 3D vision sensors, like the Shape Drive G4, can be easily integrated into different industrial environments. However, communication and performance of these devices are issues to be addressed and assured, especially when commands need to be reliably sent, received, and validated. The lack of acknowledgment mechanisms in previous architectures led to potential uncertainties about whether a given command was received and executed properly. This could lead to inefficiencies and, ultimately, a reduction in the reliability of the captured data for applications.

To overcome this limitation, the sensors under G4 platform require an acknowledgment mechanism that provides meaningful feedback when a command is successfully executed but also when incorrect parameters or unsuccessful executions occur. Now this was the main motivation behind the development of an acknowledgment response system that would provide a robust solution for monitoring and managing command execution of the sensors.

1.2 Problem Statement

“The lack of active acknowledgment responses in the current G4 product platform prevents users from knowing the real-time status of ASCII command execution, reducing the sensor’s user-friendliness and reliability. This project aims to develop an acknowledgment response system to address this issue.”

1.2.1 Definition of Problem Statement

The sensors under G4 product platform work efficiently to produce 3D point clouds using their calculations. As these sensors are currently on the market, it is very essential to ensure that they are user friendly, efficient, and free of bugs. The shape Drive G4 sensors have undergone extensive environmental testing before market release. All the functions that the G4 products exhibit are defined under ASCII Commands. The precise execution of these command logics is necessary for the sensor to fulfil its intended functionality.

The idea of creating acknowledgment responses came into play only when it was necessary to know the status of the commands to the user. Why it was necessary is a good question. The answer to this is to make sure that the right commands are executed with the insisted values and the operations take place as intended. The acknowledgment response would also help the user to debug critical problems by knowing what exactly happened to the sensor in case of both success and failure.

Before thinking about implementing an acknowledgment response system for ASCII Commands, let us understand what an acknowledgment is, and how important it is in today’s world.

An acknowledgment is regarded as the feedback to the input system from the output system after execution of its function. It is important to let the input system know whether the inputs sent from the input system have reached the desired target and whether the desired function has been executed by the target output system.

1.2.2 Impact of the requirement on real-time application

The products in the G4 lineup have various applications. This includes bin picking in the automotive industry, precise automation application in the food industry and so on.

The acknowledgment response system can be clearly useful for ensuring the efficiency of these applications and debugging. From setting the exposure time and subsampling, selecting the desired camera mode and starting acquisition, all these actions would have a clear-cut response back from the sensor for the user.

In case of misbehaviour, the acknowledgment responses could be checked, and debugging could be done easily based on the detailed response generated.

1.3 Definition of Requirements

This section is to clearly outline the functional and non-functional requirements necessary for the successful development and integration of the ASCII command acknowledgment system within the Shape Drive G4 firmware.

1.3.1 Functional Requirements

Functional requirements are the specific behaviour or functions that the system should exhibit to achieve the goal of this project.

1. **Command parsing and handling:** The current system architecture uses an if-else conditional check for parsing function. This approach has a disadvantage in terms of extending the cases as the number of commands increases and would also affect code readability and standardization. As a modification to this existing implementation, a more reliable approach must be implemented. This approach must ensure that the parsing function should be implemented in such a way that the command handling at each stage should be focused.
2. **Command Acknowledgment Implementation:** The system must generate an acknowledgment response for all ASCII commands currently supported by the G4 platform sensors as well as future commands. The system should be able to generate responses for successful execution and report indications in other cases for all the ASCII commands.
3. **Command Validation:** The system must validate every ASCII command before execution. In cases where the user tries to send an invalid command or value, and in case if the user sends valid commands, the system should respond accordingly in respective cases and give the indication to the user in the acknowledgment response.
4. **Command Classification for Acknowledgment:** The system must categorize commands into appropriate types, to achieve a common ground for command acknowledgment. This classification should help in guiding the specific handling and acknowledgment response for each command.
5. **Acknowledgment Format Standardization:** The acknowledgment responses should follow a format which is readable and flexible. The format agreed upon by the software architects is XML.
6. **Logging and Reporting:** The system must log the outcome of every command execution, including the acknowledgment response. A feature to configure the

acknowledgment response functionality should be implemented. The acknowledgment response should be displayed on the PC console or an appropriate user interface, such as the WeDevTool (a developer tool) and in the terminal as a debug message in the sensor logs.

1.3.2 Non-Functional Requirements

Non-functional requirements describe the overall qualities or attributes the system must have. These requirements do not define specific behaviours but rather how the system should perform under various conditions.

1. **Performance:** The acknowledgment system must process and respond to commands with minimal latency. Responses should be generated in real-time.
2. **Reliability and testing:** The system must reliably generate acknowledgments for every command, ensuring no feedback is missed. The acknowledgment handler must be implemented and tested so that it can withstand high traffic and complex command sequences without failure.
3. **Scalability:** The system should be scalable to accommodate future additions to the number of commands or changes in the functionality of the ShapeDrive G4 sensors. The architecture should allow for easy extension and integration of new commands.
4. **Security:** To prevent unauthorized access or manipulation of sensor functionality, the system should implement basic security measures, ensuring that only valid and authenticated command should trigger the acknowledgment responses.
5. **Usability:** The acknowledgment response system should be user-friendly, providing clear feedback to the users. The responses should aid in troubleshooting and ensure smooth operation of the sensors in diverse application environments.
6. **Compatibility:** The acknowledgment handler must be compatible with the existing system architecture.

Chapter 2

STRUCTURE OF THE THESIS

After introducing this thesis in Chapter 1, describing the context of the problem, its motivation, and the definition of problem statement, the next parts of the report are organized as follows:

- **Chapter 3:** presents a comprehensive overview of the Shape Drive G4 system, including its hardware, software, and the current architecture.
- **Chapter 4:** provides a literature review of communication protocols, parsing mechanisms, and acknowledgment handling techniques in embedded systems.
- **Chapter 5:** provides a comprehensive overview of the methodology followed to implement the acknowledgement response system.
- **Chapter 6:** discusses the implementation part of the command Acknowledgement Response System.
- **Chapter 7:** discusses the testing and analysis of the results generated.
- **Chapter 8:** Summarizes the findings and contributions of the thesis, concludes the work.
- **Chapter 9:** Summarizes the future outlook possible for the current implementation that presents future opportunities for extending the acknowledgment response functionality.
- **Chapter 10:** Bibliography: Lists all the sources cited in the thesis using an IEEE citation style.
- **Chapter 11:** Appendices: Includes the additional material that is relevant but too detailed for the main body of the report.

Chapter 3

SYSTEM OVERVIEW

This section will focus on the high-level design and current architecture of the project and then the possible modification to it for implementing the acknowledgment handler for the system.

3.1 Overall System

The sensors under G4 product platform especially the ShapeDrive G4 3D sensors are high-performance sensors in the field of resolution and measuring speed and are the pioneers among 3D machine vision sensors. Thanks to the integrated processing of the data into a 3D point cloud and a fast Ethernet interface with up to 10 Gbit/s, users can receive and process 3D point clouds in the shortest possible time. one [1]

The measuring principle of the 3D machine vision sensor ShapeDrive G4 is based on triangulation and structured light. Several patterns are projected onto the object via the illumination, while it is captured synchronously by a camera. Sophisticated algorithms are used to calculate a 3D point cloud from the various images. The internal calculation is carried out using the fastest electronics. The 3D sensors of the ShapeDrive G4 series, also known as snapshot sensors, are available with different measuring ranges or measuring volumes, so that a sensor can always be selected to suit the application. [1]

The main highlights of the ShapeDrive G4 sensors are:

1. **Processing Unit (PS):** Central processor unit for liquid command processing, control and communication. [1]
2. **Field Programmable Gate Array (PL):** Real-time processing unit for fast calculation of 3D point clouds in less than 250 milliseconds. [1]
3. **Memory:** Large (4 GB) and fast (19.2 Gbit/s) memories enable reliable processing of huge amounts of data. [1]
4. **Connectivity:** Fast transmission speeds, thanks to the integrated 1/10 Gigabit Ethernet interface. [1]

3.2 The Benefits of ShapeDrive G4

ShapeDrive G4 sensors are designed to the highest quality in all respects. All components of the ShapeDrive G4 series such as optics, mechanics, electronics, firmware and software must fulfil the highest quality requirements. This ensures a long service life with stable and reproducible results, as expected in industrial applications. Many different factors influence quality. The most important of these are given below. [1]

1. **Excellent 3D Point Cloud:** The 3D point cloud meets the highest demands thanks to the sophisticated algorithm combined with the quality of the hardware. This minimizes or eliminates noise and other artifacts. [1]
2. **Stability with Temperature Management:** In addition to high-quality components, all ShapeDrive G4 sensors have active temperature management. This temperature management ensures that the sensor delivers stable and reproducible results, even under fluctuating external conditions. [1]
3. **Reliability Through Calibration:** All ShapeDrive G4 sensors are calibrated and tested prior to delivery. This ensures that the sensor remains stable for years, with no recalibration required. The sensor can therefore be used permanently and without interruption. [1]
4. **Easy integration:** The ShapeDrive G4 sensors can be integrated via an SDK or a GigE Vision interface. The sensor is connected to the application and ready for immediate use in just a few steps. The integrated web server simplifies handling and configuration. [1]
5. **Regular firmware and software updates:** Improvements in firmware and SDK/GigE Vision free of charge for the user. Benefit from continuous product development with excellent extended functions. The ShapeDrive G4 sensors can be updated to the latest versions by the user. [1]
6. **Flexibility with wide range of Models:** There are various Shape Drive G4 sensors which cover measuring ranges from a few centimetres to over one meter. For extreme requirements, there are also sensors with a camera resolution of 12 MP that accurately capture even the smallest details. [1]

3.3 Existing Architecture

In this session we try and understand the Existing architecture for the firmware **melApp** and the developer application **WeDevTool** and how this can be modified to add the new feature of Acknowledgement Handler for generating the acknowledgement response without affecting the performance and stability of the system and allowing a seamless integration.

The figures 2 and 3 shows the melApp and WeDevTool architecture respectively.

3.3.1 Firmware (MelApp) Architecture

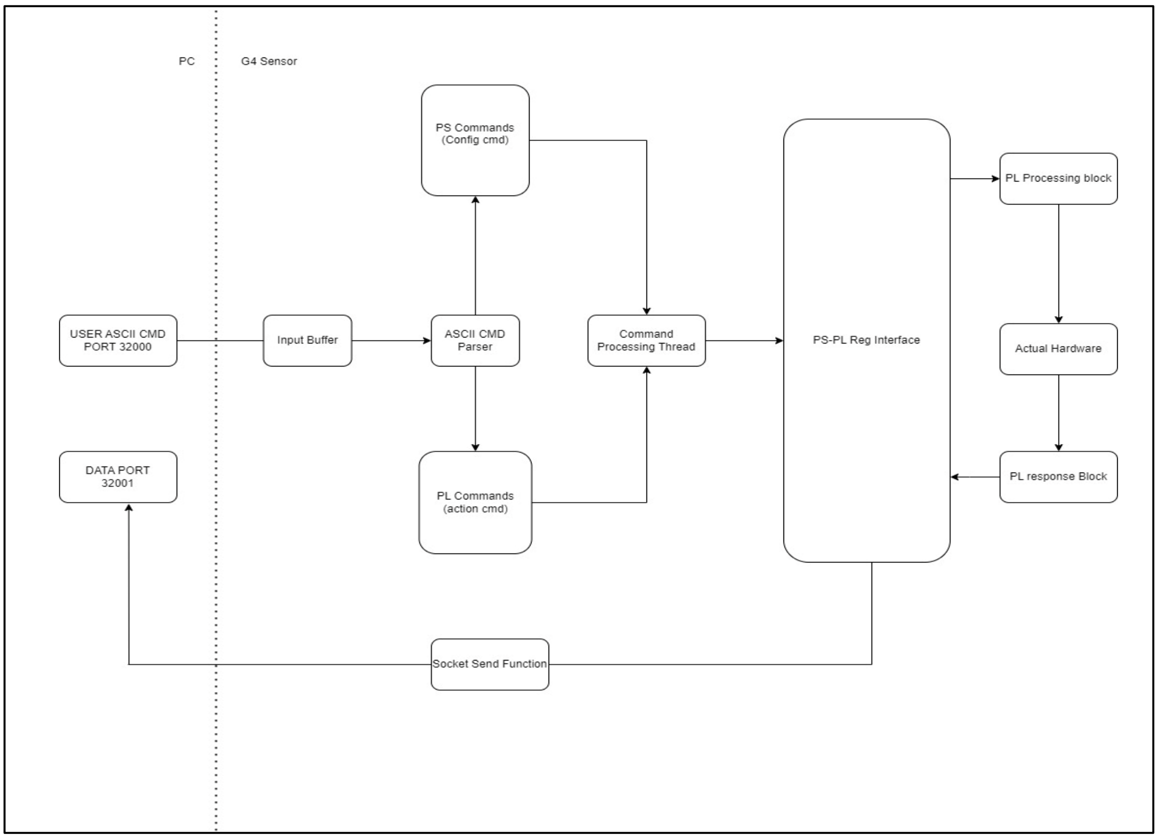


Figure 2 melApp Architecture

Figure 2 shown above represents the outline of current firmware (melApp) architecture with respect to the implementation of an acknowledgment response system.

1. The ASCII commands are sent through USER **ASCII CMD PORT 32000**.
2. The incoming ASCII commands are stored in **Input Buffer** and passed on to the **ASCII CMD Parser** for parsing procedure.
3. After Parsing the ASCII commands are handled as **PS Commands** and **PL commands**.
4. The PS Commands (configuration commands) are processed and executed in **Command Processing Thread**.
5. The PL Commands (action commands) are passed on to the **PL Processing Block** for its execution and the communication takes place through **PS-PL Register Interface**.
6. The execution PL Commands are reflected in the **actual hardware** including camera, LED and peripherals.
7. The response from the PL after the execution is generated by the PL response block and is communicated with PS through the **PS-PL Register Interface**.
8. The response data is sent back to PC Applications like WeDevTool through **DATA PORT 32001** using **socket send**.

The above architecture is to be modified as a part of the thesis. The Acknowledgement Response System should be integrated into this architecture without causing any disturbance to the existing functionalities.

3.3.2 WeDevTool Architecture

The WeDevTool is a developer application which is used primarily to send ASCII commands to the sensor and test its functionality. It is connected to the sensor by TCP communication. The two ports used are 32000 (ASCII Port) and 32001 (Data Port).

The WeDevTool consists of different functionalities which are activated accordingly. This includes a command sequence window, XML descriptor window, video output window, 3D point-cloud window and so on. There is also a log window for logging the command sequences sent to the sensor.

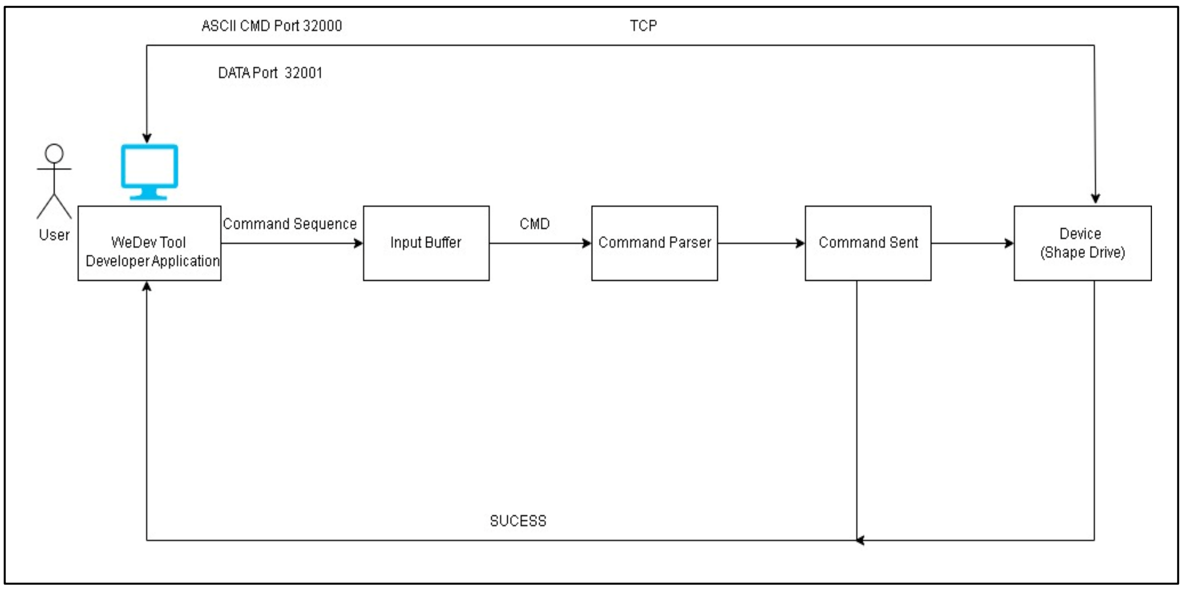


Figure 3 WeDevTool Workflow Diagram

Let’s understand the workflow of WeDevTool from figure 3:

The WeDevTool is connected to the device using an IP address and establishes a TCP Connection.

There are two ports through which communication takes place. They are:

ASCII Command port and **Data Port.**

- **ASCII Command port:** The ASCII Command port (32000) is used to send ASCII Commands to the Sensor.

- **Data Port:** The Data port (32001) is used for transmitting video stream data.

Workflow:

1. **Input Buffer:** The commands when sent in sequences are initially stored in an input buffer and then each command is sent to the command parser.

2. **Command parser:** The command parser is used to parse the command and the value which is then sent to the Sensor.
3. The log window displays a success message for each command which indicates that the ASCII Command has been sent to the sensor successfully.
4. **Device (Sensors):** The commands which are sent from the PC using WeDevTool are received by the sensor. The sensor also communicates with the WeDevTool with response data like video output and XML descriptor data as requested.

The next step is to propose an initial concept on how the acknowledgment response system should be developed. The idea should be developed by understanding the (melApp) firmware Architecture and by trying to improve this architecture without disturbing the current implementations. For moving on to the next step a background study is necessary about the existing acknowledgement methodologies.

Chapter 4

RELATED WORKS

The concept of how the sensor interacts with PC, what are the acknowledgment responses existing in today's world and what are the requirements necessary to build an acknowledgment response system must be studied and decided. There are several possible ways on how the sensor could interact with the host system like socket communication, client-server model, TCP/IP Model and use of communication protocols like TCP/IP, UDP, MQTT, HTTP and so on.

In Today's world of industrial automation and high- precision operations, 3D sensors play an important role in capturing the 3-dimensional data which serves as a reference for various applications. As these applications largely depend on the precision and reliability of the data, the communication between the sensors and user/control system must be robust and should be able to give some feedback to ensure the correctness and reliability of the data produced.

How this comes into picture when dealing Shape Drive G4 sensor is that, for these sensors which are used for applications like bin picking, and so on requires extreme accuracy in terms of their function execution and the captured data. It is necessary to ensure that the commands sent to these sensors for the functional operation are executed accurately and without error. Any miscommunication or failure to confirm command execution could lead to inefficiencies, or compromised product quality.

There is a significant gap in the current implementation of 3D sensor firmware which is the absence of built-in command acknowledgment response systems. These systems, which confirm the successful receipt and execution of commands at the application level, are critical in enhancing reliability. There are different fields such as IoT and robotics where acknowledgment mechanisms are commonly used to increase communication robustness and ensure precise control. These systems provide real-time feedback to confirm that the intended operations have been carried out as expected, reducing the chance for command misinterpretation, and helps in debugging.

Even though this implementation is a key towards achieving data authenticity the same has not been widely implemented in 3D sensor firmware particularly within the G4 product platform. This lack of acknowledgment functionality leaves a gap in the sensor's ability to guarantee command execution, limiting its effectiveness in high-reliability applications.

Implementation of such acknowledgment response systems requires an idea about the Network protocols and the current acknowledgment mechanisms already used in the world of embedded systems and in the next sections we focus on these in detail.

4.1 Overview of Network Protocols and Their Role in Embedded Systems

4.1.1 Network Protocols

Network protocols are sets of rules and conventions that determine how data is transmitted, received, and interpreted across networked devices. They usually represent the language for communication between devices. [2]

For embedded devices, which are often designed for specific applications reliable communication is crucial. Many embedded systems—like 3D sensors, industrial controllers, and medical devices—operate in environments that require precise, data exchange. Network protocols play a foundational role in these systems by ensuring that messages are transmitted accurately and efficiently. [2]

There are several protocols in the world that is used for communication. A few such protocols are TCP, IP, UDP, SMTP, FTP, and so on.

1. **Transmission Control Protocol/Internet Protocol (TCP/IP):** The TCP/IP is the foundational protocol suite of the internet, enabling reliable communication.
    - TCP: Ensures data is delivered reliably and in order.
    - IP: Routes data packets to their destination based on IP addresses. [2]
2. **Hypertext Transfer Protocol HTTP and HTTPS:** The protocols used for transmitting web pages.
    - HTTP**:** Unsecured communication.
    - HTTPS**:** Secured communication using SSL/TLS encryption. [2]
3. **Simple Mail Transfer Protocol (SMTP):** The Protocol for sending email. Works with other protocols like POP3 and IMAP for email retrieval. [2]
4. **File Transfer Protocol (FTP):** The Protocol for transferring files between computers.
    - Includes commands for uploading, downloading, and managing files on a remote server. [2]

4.1.2 Functional Overview of TCP/IP and its Usage

The Transmission Control Protocol/Internet Protocol or TCP/IP is the most used protocol for communication across networks. It is regarded as the backbone of the internet and plays a crucial role for reliable and efficient data transmission, making it ideal for applications requiring high reliability, such as sensor networks and industrial automation. [2]

- **TCP/IP Protocol Stack:** TCP/IP protocol stack contains four layers: [2]
    1. Physical layer
    2. Internet Protocol (IP) layer
    3. Transport layer: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP)
    4. Applications layer

- **Physical Layer:** At the bottom of the stack is the physical layer, which deals with the actual transmission of data over physical media such as serial lines, Ethernet, token rings, and FDDI rings. Messages can also be sent and received over other, non-physical access methods such as VTAM/SNA. [2]

- **Internet Protocol (IP) Layer:** Above the physical layer is the Internet Protocol (IP) layer, which deals with the routing of packets from one computer to another. The IP layer:
  1. Determines which lower-level protocol to use when multiple interfaces exist.
  2. Determines whether to send a packet directly to the host or indirectly to a relay host known as a router.
  3. When a packet is larger than the size supported by the physical medium, the IP layer breaks the packet into smaller packets, a process referred to as "fragmentation and reassembly".
  4. Provides some control services for packets and ensures that they are not sent from router to router indefinitely. However, the IP layer does not keep track of a packet after it is sent, nor does it guarantee that the packet will be delivered. [2]

- **Transport Layer:** Above the IP layer is the transport layer, which contains the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). [2]

  1. **Transmission Control Protocol (TCP):** The Transmission Control Protocol (TCP) guarantees that data sent by higher levels is delivered in order and without corruption. To accomplish this level of service, the TCP implementation on one computer establishes a session or connection with the TCP implementation on another computer devices (e.g., between a server and a client). This process is referred to as Connection Oriented Transport Service (COTS) and is critical for embedded systems. [2]. After a session is established, data is sent and received as a stream of contiguous bytes, each byte can be referenced by an exact sequence number. When data is received by the remote TCP, it sends an acknowledgment back to the local TCP advising it of the sequence number of the last byte of data received. If an acknowledgment is not received, or if an acknowledgment for previously sent data is received twice, the local TCP retransmits the data until it is all acknowledged. The remote TCP discards any bytes that are received more than once. The Transmission Control Protocol needs support to make itself reliable for data transmission and this is done with the help of an IP or Internet Protocol. [2] All data sent and received by TCP is validated for corruption using checksums. Whenever a checksum is incorrect, the bad data is discarded by TCP, and the correct data is retransmitted until it is accurately received. [2]
  2. **User Datagram Protocol (UDP):** Unlike the TCP, the User Datagram Protocol (UDP) transmits and receives data in packets (datagrams), and delivery is not guaranteed. The contents of the data can be sent with or without a checksum. The use of checksums varies widely from one implementation to another. [2]

- **Applications Layer:** Above the transport layer is the applications layer, which contains both general applications and function libraries for use by applications. [2]

  Some general applications that run over TCP include:

  1. File Transfer Protocol (FTP)
  2. Remote terminal emulation (TELNET in line mode, TN3270 in full screen)

3. Electronic Mail (SMTP)

Some general applications that run over UDP are:

1. Network File Server (NFS)
2. Domain Name Server (DNS). [2]

4.1.3 Interface with TCP and UDP

Function libraries provide routines to simplify the interface between applications and TCP/UDP of the Transport layer:

- The most common function library is known as Sockets, which allows an application written in C to access TCP as if it were just another stream input/output device.
- Another function library that is less commonly used is Remote Procedure Call (RPC), which allows applications to make calls to functions that are in another application on a different computer. [2]

The environment in which an application runs often dictates the interface used between it and TCP or UDP:

- Most UNIX, OS/2, and Windows applications are written in C and utilize a direct socket interface.
- On IBM mainframes and other systems based on the same architecture such as Fujitsu Technology Solutions, applications are often written in S/390 assembler and use either a pseudo-socket interface or an application program interface (API) to gain access to the TCP/IP protocol stack. [2]

4.1.4 Ports

The interface that exists between an application and TCP is referred to as a port. Ports are classified as server ports and client ports:

- Server ports are generally ports on which the application "listens" for incoming connections to be made. [2]
- Client ports are generally ports on which the application "connects" outwardly to a server port. [2]
- An application may control multiple client ports and server ports simultaneously.
- Each port is identified by a port number, which ranges from 1 to 65535.
- The port number used by client ports usually has no significance and is often assigned by TCP.
- Server port numbers, however, are usually required to be "well known"; that is, the client must know which port the server is listening on when it attempts to connect. Server port numbers usually are specified by the server application. [2]
- Ports used by g4 devices are **32000** and **32001**. They are ascii command port and data port respectively. [2]

4.1.5 Transmission Control Protocol (TCP) Benefits to Embedded Systems

TCP/IP is beneficial for embedded systems because it provides reliability through error detection, packet retransmission, and in-order packet delivery. For the device like Shape Drive G4 sensor, where commands sent to the sensor need to be confirmed to ensure proper operation, TCP's acknowledgment feature becomes essential. TCP's acknowledgment mechanism enables the sensor to send a confirmation back to the client device, ensuring that each command is successfully received and processed. This adds a layer of robustness to the system, reducing errors and increasing trust in the communication process. [2]

4.1.6 Advantages of TCP over others for the G4 Devices

The Need for Reliable Communication in Sensor Systems: In high-precision industrial applications, 3D sensors like the ShapeDrive G4 are often tasked with capturing detailed spatial data that is integral to functions like quality control, object detection, or autonomous navigation. Given the critical role of these sensors, any communication issues, such as missed commands or misinterpreted data, can lead to significant operational inefficiencies and even safety concerns. TCP/IP's reliability features make it well-suited for these contexts, as it reduces the chance of data loss and ensures that commands are accurately received and executed. [2]

- **Reliability:** TCP is a connection-oriented protocol, meaning that it ensures reliable data delivery. Given the importance of accurate sensor readings and commands, reliability is critical. Unlike UDP, which can drop packets, TCP provides built-in acknowledgment, error correction, and retransmission. [2]
- **Built-in Acknowledgment:** TCP handles acknowledgment automatically, reducing the need for custom acknowledgment mechanisms as required by UDP. This simplifies the communication design, reducing development time and complexity. [2]
- **Data Integrity:** TCP uses checksums to verify data integrity and ensures that messages are received in the correct order. This is crucial for sensor networks, where incorrect or out-of-order data can lead to inaccurate measurements or command execution. [2]
- **Ease of Implementation:** With TCP/IP, socket communication can be implemented easily in various programming languages, allowing seamless integration with embedded systems. Many libraries and frameworks support TCP, making it a widely used and well-documented protocol. [2]
- **Overhead Considerations:** While TCP introduces more overhead than UDP, the additional reliability and acknowledgment features justify the overhead in sensor communication systems where data accuracy is paramount. [2]

4.2 Overview of Sockets in communication systems

- A socket is one endpoint of a two-way communication link between two programs running on the network. The socket mechanism provides a means of inter-process communication (IPC) by establishing named contact points between which the communication takes place. [3]

- The socket provides a bidirectional **FIFO** Communication facility over the network. A socket connecting to the network is created at each end of the communication. Each socket has a specific address. This address is composed of an IP address and a port number. [3]
- Sockets are generally employed in client server applications. The server creates a socket, attaches it to a network port address then waits for the client to contact it. The client creates a socket and then attempts to connect to the server socket. When the connection is established, transfer of data takes place. [3]

There are two types of Sockets Datagram Socket and Stream Socket.

- **Datagram Socket:** Use UDP (User Datagram Protocol), which is faster but doesn't guarantee message delivery or order. This is a type of network which has connection-less points for sending and receiving packets. It is like a mailbox. The letters (data) posted into the box are collected and delivered (transmitted) to a letterbox (receiving socket). [3]

- **Stream Socket:** Use TCP to ensure reliable communication with error checking and retransmission. In Computer operating system, a stream socket is type of interprocess communication socket or network socket which provides a connection-oriented, sequenced, and unique flow of data without record boundaries with well-defined mechanisms for creating and destroying connections and for detecting errors. It is like phone. A connection is established between the phones (two ends) and a conversation (transfer of data) takes place. [3]

4.2.1 Client –Server Model

The Client-server model (figure 4) is a distributed application structure that partitions tasks or workloads between the providers of a resource or service, called servers, and service requesters called clients. In the client-server architecture, when the client computer sends a request for data to the server through the internet, the server accepts the requested process and delivers the data packets requested back to the client. Clients do not share any of their resources. Examples of the Client-Server Model are Email, World Wide Web, etc. [4]

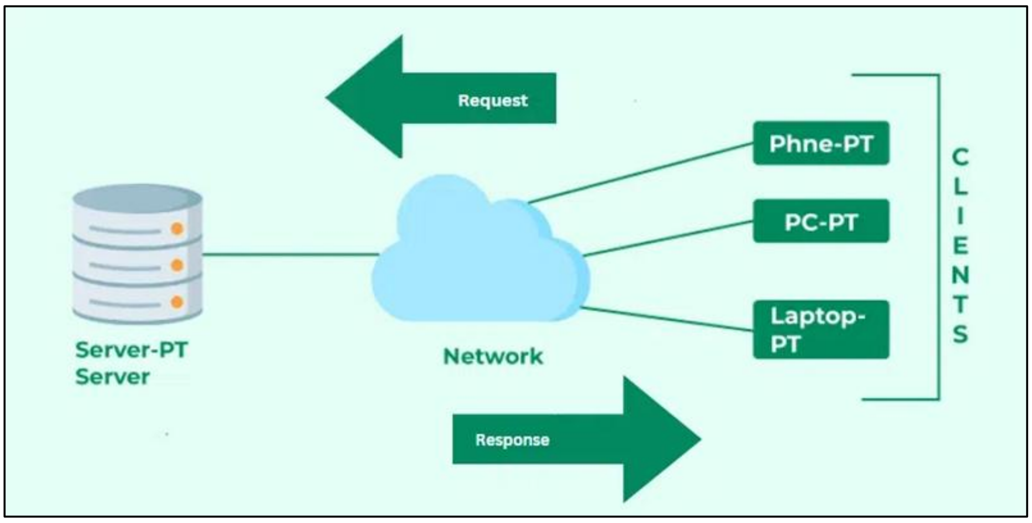


Figure 4 Client-Server communication [4]

4.2.1.1 Working of Client Server Model [4]

- **Socket Creation:** The server creates a socket using system calls or APIs, specifying parameters like the protocol IPv4, the socket type (stream or datagram), and the protocol (TCP or UDP).
- **Binding the Socket:** The server binds the socket to a specific IP address and port, preparing it to listen for incoming connection requests.
- **Listening for Connections:** The server socket is set to listen mode, allowing it to handle multiple incoming requests. In TCP communication, it will wait for a client to request a connection.
- **Establishing a Connection:** The client creates its own socket and initiates a connection to the server's IP address and port.
- For TCP communication, a three-way handshake occurs:
  - **SYN** (synchronize) packet is sent from the client.
  - **SYN-ACK** packet is sent by the server to acknowledge the client’s request.
  - **ACK** packet is sent by the client to acknowledge the server's response.
- Once this handshake is completed, the connection is established, and both client and server can send data.
- **Data Transfer:** Once the connection is established, data can be sent from the client to the server and vice versa. For TCP sockets, data is split into packets, ensuring reliability and order. For UDP sockets, data packets are sent individually and can arrive out of order or get lost.
- **Closing the Connection:** Once communication is complete, either the client or server can close the connection. For TCP, a four-way handshake occurs to close the connection, while in UDP, each endpoint can stop communication independently. [4]

4.2.2 Client-Server Model in G4 Sensors

The G4 Sensors follow the client-server model for their communication. Here the user acts as the client and the sensor acts as the server based on the use cases.

4.2.3 Aim of the Thesis with respect to the TCP/IP

The TCP/IP protocol supports the transport layer of the sensor communication but does not address application-level acknowledgment, which is something the acknowledgment response system aims to add.

4.3 Existing Command Acknowledgment Mechanisms

1. **HTTP Response Codes**: HTTP uses standardized status codes to acknowledge requests. For example, 200 OK indicates success, while 404 Not Found signals an error and there are many more status codes. While HTTP’s acknowledgment system is well-structured, it is not suitable for real-time systems due to the overhead it introduces. [5]

2. **MQTT QoS Levels**: Message Queuing Telemetry Transport (MQTT) is a protocol commonly used in IoT applications. It uses Quality of Service (QoS) levels to define how acknowledgment is handled. [6]

   - QoS Levels:
   - QoS 0: Fire-and-forget (no acknowledgment).
   - QoS 1: The message is guaranteed to be delivered at least once.
   - QoS 2: The message is guaranteed to be delivered exactly once. [6]

Chapter 5

METHODOLOGY

This chapter mainly outlines the methodology employed in this project. What are the approaches followed. This section also gives the insight to the experimental setup involve in this project. The goal is to establish a balance between efficiency and modularity of the system.

The aim of this section is to fulfil the requirements of implementing an acknowledgment response system and what are the methodologies involved and why.

5.1 Experimental Setup

This section focuses on the initial hardware and software experimental setups for preparing the ecosystem towards achieving the target functionality.

5.1.1 Hardware Setup

The first step towards the task of creating ASCII command acknowledgment responses was to set up the Hardware Shape Drive G4.

- The Hardware installation involves connecting the Shape Drive to the programmable power supply with input voltage of 24 V. The resulting optimal range of current is 0.6 - 0.7 A.
- A 10G Ethernet cable is connected to the device as they are used to support high-speed, low-latency data transfer in various applications, ranging from data centers and enterprise networks to high-performance computing and media broadcasting. They provide the necessary bandwidth and reliability for modern network infrastructures and help future-proof networks against increasing data demands.
- Now the device is powered ON, as the next step the latest firmware is flashed. This is done by uploading the latest firmware from latest releases in Jenkins. Once the firmware is downloaded it is flashed on to the hardware memory and the device is restarted.

5.1.2 Software Setup

The next step is to set up the software components necessary to start working towards the goal.

- First, a tool called WeDevTool is downloaded from the software kiosk and set up. The WeDevTool is a developer’s tool, and its current architecture allows users to send ASCII Commands to the sensor and test the implemented functionalities. It communicates with the sensor by following TCP/IP protocol. The two ports used are 32000 (ASCII Port) and 32001 (Data Port).

- Next the Linux environment is to be set up. This is done with the help of Virtual Box, which allows users to create and run virtual machines on computers. This helps in running multiple operating systems simultaneously. The whole development process takes place in the Linux environment.
- The next step is to install the Wenglor toolchain which helps in cross compiling and running the code.
- The Visual Studio Code is selected to work with as it is a user friendly and efficient IDE.
- The design decision is to use the C programming language for the development of the acknowledgment Response System as it is being currently used for firmware development for G4 sensors.

5.1.3 C Programming

C programming is a general-purpose programming language. It was created in the 1970s by Dennis Ritchie and remains very widely used and influential. By design, C's features cleanly reflect the capabilities of the targeted CPUs. It has found lasting use in operating systems code (especially in kernels), device drivers, and protocol stacks. C is commonly used on computer architectures that range from the largest supercomputers to the smallest microcontrollers and embedded systems. It was designed to be compiled to provide low-level access to memory and language constructs that map efficiently to machine instructions, all with minimal runtime support. Despite its low-level capabilities, the language was designed to encourage cross-platform programming. [7]

After organizing and setting up all the hardware and software components which are necessary for the project, the next step is to access the sensor using Secure Shell (SSH).

5.1.4 Secure Shell (SSH)

Secure Shell (SSH) is a widely used protocol for securely accessing and managing remote systems over an unsecured network. It provides encrypted connections, ensuring data confidentiality and integrity, making it ideal for remote system management, command execution, and secure data transfers. The uses case of ssh in connecting to the G4 Sensors is to have remote access and to help in file transfers:

- **Remote Access**: SSH allows administrators to remotely manage systems.
- **ssh user@sensor_ip**.
- **File Transfers**: Using SCP or SFTP, files can be securely transferred between local and remote systems.

5.2 Concept Development: Acknowledgement Handler

This section will focus on the next steps of what the initial ideas on how the Acknowledgment handler is to be developed and integrated was proposed. This was only possible after understanding the existing system overview and workflow. The first proposed structure of the Acknowledgement Handler is described below in figure 5:

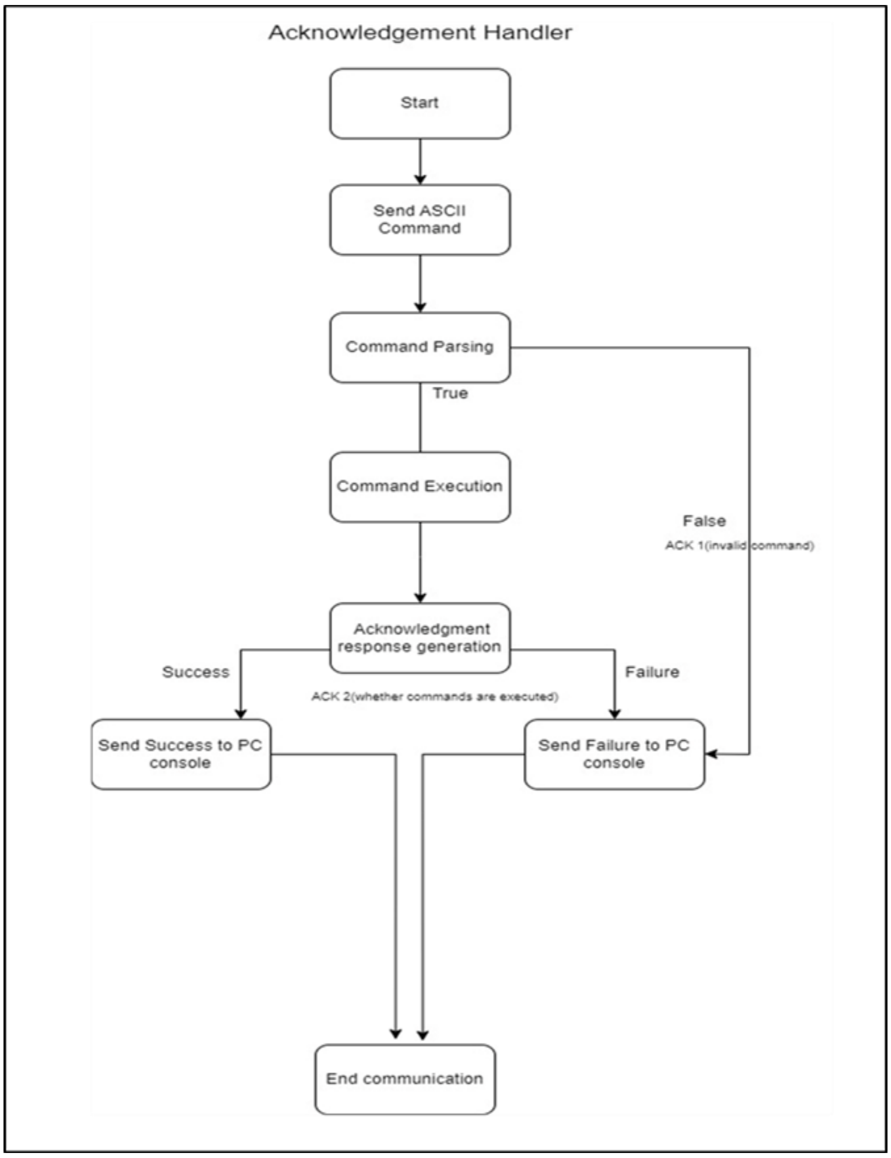


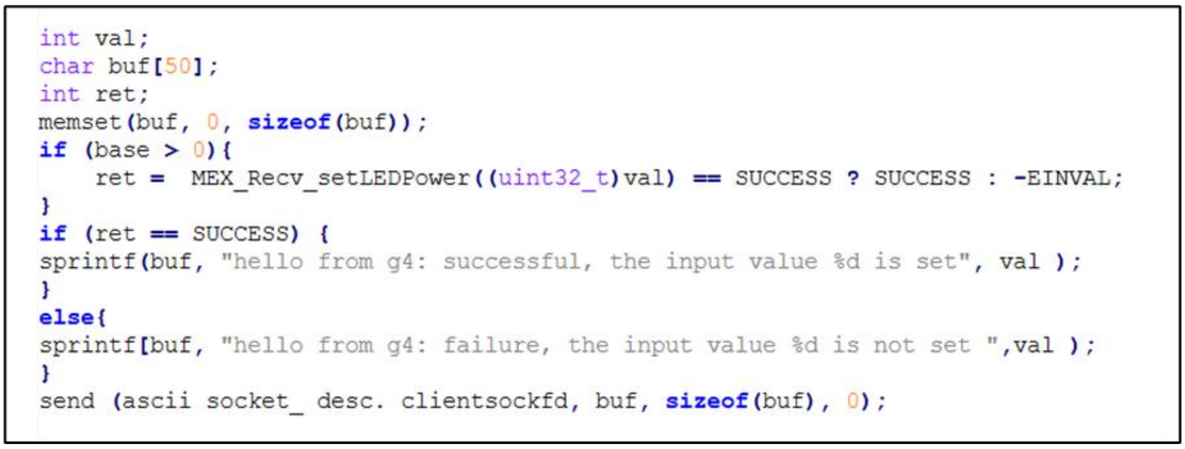
Figure 5 Acknowledgment handler (initial diagram)

1. As the first stage of concept development of the acknowledgment handler, the workflow of how the command acknowledgement should work was proposed as shown in the figure above.
2. The first block START shows the start of command acknowledgment process. As the next step the ASCII commands are to be sent for command validation. This is indicated by the next two blocks SEND ASCII COMMAND and COMMAND VALIDATION.
3. The command validation and command processing go hand in hand and is done for checking whether the commands that are sent are valid commands. This is done using a command parser which is the ParseAsciiCommand Function. The ParseAsciiCommand function takes in the command from the user together with its value for checking if they are in valid format. If they are not in valid format, an acknowledgment is sent via the acknowledgment response block. If they are valid then processing takes place in the command processing block and the commands are executed using command executor block.

4. A second acknowledgement is generated via the acknowledgment response block. This acknowledgment is to help the user to know whether the command execution was successful or not. If it's successful it sends success to PC console and if not, it sends failure to the PC console. These are shown using the two blocks SEND SUCCESS TO PC CONSOLE and SEND FAILUR TO PC CONSOLE. This is the end of communication shown by END COMMUNICATTION block.

5.2.1 Practical Work

The above idea has been implemented using an experiment setup with the existing architecture by adding the simple snippet code as shown below in figure 6.



```
int val;
char buf[50];
int ret;
memset(buf, 0, sizeof(buf));
if (base > 0){
    ret =  MEX_Recv_setLEDPower((uint32_t)val) == SUCCESS ? SUCCESS : -EINVAL;
}
if (ret == SUCCESS) {
sprintf(buf, "hello from g4: successful, the input value %d is set", val );
}
else{
sprintf[buf, "hello from g4: failure, the input value %d is not set ",val );
}
send (ascii socket_ desc. clientsockfd, buf, sizeof(buf), 0);
```

Figure 6 initial implementation code snippet

The code above generates an acknowledgement response with the message "**hello from g4; successful, the input value (value) is set**" if the LED Power setting is done successfully and generates a message "**hello from g4; successful, the input value (value) is not set**" if the LED Power Setting fails. The generated responses are sent over the socket with the send function by using an **ascii socket_desc.clientsockfd** which is an ascii socket file descriptor already implemented within the firmware of the sensor.

The initial concept of Acknowledgement Handler was designed as shown above and proposed. The idea was then subjected to peer review and after discussions it was found the idea needs to be modified in terms of readability, detailing, handling acknowledgments for different types of commands, modularity of command handles and acknowledgment and the format of acknowledgment responses.

The next step is to define the command classification for acknowledgments and behaviors, acknowledgement format, and details of responses to be generated.

5.3 Design Decisions

This section focuses on what are the design decisions to be followed to implement the Acknowledgement Response System.

5.3.1 Acknowledgment Format Definition

The acknowledgement response generated is defined to be in an XML container format. The reason for using XML container format is to standardize the acknowledgement format to make it easier for parsing and verification. The current firmware also uses the container format to ensure accurate parsing, verification and error checking because they are essential for real-time communication environments.

The XML Acknowledgement container serves as a structured and reliable format to send the acknowledgment responses from the sensor to the user space. It provides a way to hold the data in a readable and machine-processed format. The container acts as a wrapper to hold different elements present in the acknowledgment response.

The XML Container typically contains:

1. **Metadata:** This indicates the information about the data (data tags) which should be given back as response.
2. **Data fields:** Indicates the actual data being sent as the response associated with the Metadata.
3. **Attributes:** Additional information for describing data if necessary. [8]

5.3.2 Advantages of Using XML as a Container Format

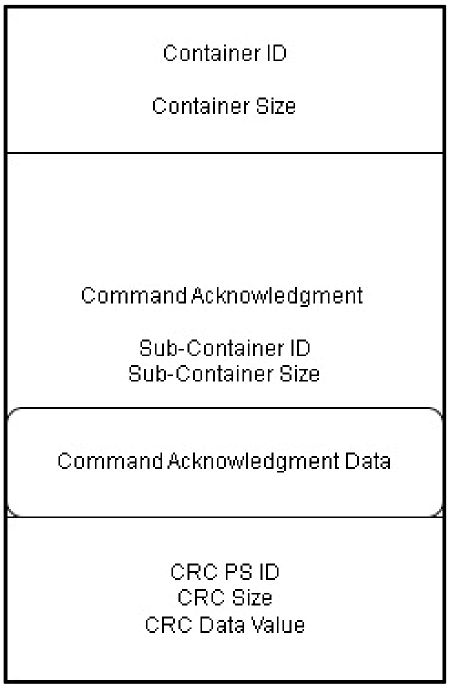


Figure 7 XML Container Definition

1. Human-Readable.
2. Extensible and Flexible.
3. Well-Defined Structure.
4. Self-descriptive using tags and attributes. Supports Metadata for adding contextual information.

Figure 7 shows the information about how the XML Acknowledgement Container is designed and the details behind the implementation.

The design of XML acknowledgment container involves following certain standards of implementation. These are shown in the above figure 7. They are:

5.3.3 Container ID and Structure

The Container ID is an ID which is used to uniquely identify the container being referred to. It is different for different containers. For the Acknowledgement container the container ID is chosen as requested**.** This ID allows the recipient to recognize and correctly parse the container. The Container Size is selected as needed. This is selected in such a way that the entire message is processed without any truncation or padding issues.

5.3.4 PS Command Acknowledgment Sub-Container

This sub-container is attached to the main container to hold specific acknowledgement data. It includes:

- **Sub-Container ID**: The Sub-Container ID is selected as needed and it uniquely identifies the sub-container type.
- **Sub-Container Size**: It specifies the size of the acknowledgement block of data.
- **Command Acknowledgment Data**: This block holds the detailed acknowledgment information including
    1. Current Value
    2. User Value
    3. Minimum Limit
    4. Maximum Limit
    5. Execution Time
    6. Command Name

5.3.5 CRC for Error-Checking

The Cyclic Redundancy Check (CRC) is majorly used to ensure data integrity.

It includes:

- **CRC PS ID**: The CRC PS ID is selected as required. This helps in identifying the CRC check block.
- **CRC Size**: Indicates the size of CRC data.
- **CRC Data Value**: The data value is selected such that it should match when computed over the acknowledgment data. This ensures any corruption during the data transmission is detected by the receiver end.

The CRC ensures robust error-checking which would help in maintaining the reliability of the acknowledgement data.

5.3.6 Command Acknowledgement Data

One of the most important parts in the creation of Acknowledgement Response Systems is defining the data to be included in the XML Container. After careful considerations and discussions, the data to be included in command acknowledgement are as follows:

- Current Value: The current value prints the value obtained from the sensor after the application of the corresponding ASCII command which is sent by the user to the sensor. This is the most important tag in the acknowledgment response as this value determines the correctness of the sensor. The tag used for XML acknowledgment response is **current**.
- User Value: The user value prints the input value which the user desires to apply to the sensor. The tag used for XML acknowledgment response is **user_value**.
- Minimum Value: The minimum value prints the minimum limit that the user can possibly set for a given ASCII command. The tag used for XML acknowledgment response is **min**.
- Maximum Value: The maximum value prints the maximum limit that the user can possibly set for a given ASCII command. The tag used for XML acknowledgment response is **max**.
- Execution Time: The execution time prints the time taken for executing the corresponding ASCII command. The tag used for XML acknowledgment response is **execution_time**.
- Command Name: The Command Name prints the name of the ASCII Command sent by the user to the sensor. The tag used for XML acknowledgment is **command**.

5.3.7 Command Classification

The ASCII commands are classified into different categories based on their value and their behavior towards incoming outbound/invalid values. The command classification was done to define the acknowledgment responses for different types of ASCII commands. This classification fulfils the criteria of following a modular approach of implementing the acknowledgment Handler to make the system flexible and future proof. After trying to find the possible command types by studying and understanding the behavior and value of commands and by following trial and error methods of classification, the number command types were down to three major types
The major command types include:

1. **COMMAND_TYPE_ACTION**
2. **COMMAND_TYPE_ENUM_PARAM**
3. **COMMAND_TYPE_RANGE_PARAM**
4. **COMMAND_TYPE_NO_ACK**
5. **COMMAND_TYPE_ACK_CHECK**

The next step after creation of different categories of command types, the next step is to identify and group which commands come under each type.

The process of grouping the commands under each category was done carefully by observing and understanding each command and their code structure.

5.3.8 Command Types with Examples

1. **COMMAND_TYPE_ACTION:** These commands are differentiated from other command types based on absence of parameter value.

   Command Examples:
   - SetAcquisitionStart
   - SetAcquisitionStop
   - SetTriggerSoftware
   - SetReboot

2. **COMMAND_TYPE_ENUM_PARAM:** These are Commands which have values of enumerated type.
   Command Examples:
   - SetSensorMode
   - SetTriggerSource
   - SetLedPattern
   - SetSubSampling

3. **COMMAND_TYPE_RANGE_PARAM:** These are Commands which have their values in range.

   Command Examples:
   - SetLedPower
   - SetGain
   - SetExposureTime
   - SetExposureTimeLimit

4. **COMMAND_TYPE_NO_ACK:** These are action Commands which do not require an explicit acknowledgement response.

   Command Examples:
   - SetReboot
   - GetXmlDescriptor
   - GetXmlDescriptorExpert

5. **COMMAND_TYPE_ACK_CHECK:** This command type is used for generating acknowledgment response for ASCII command which enables or disables the acknowledgment response feature.

   Command Example: SetAckResponseEnable

Necessity for Classification: The command classification was done in order to differentiate the acknowledgement responses for ASCII Commands of specific behaviors.

5.3.9 Command structure

A command structure is defined to manage various command profiles used for command processing, its execution and input parameters for the acknowledgment response generation. These structures contain a list of commands, each associated with certain parameters, types, and range limits, and they determine how the sensor should handle each command.

Command Structures: The ASCIiCommand structure defines each command. This structure contains:

- The command name.
- Command type (COMMAND_TYPE_ACTION, etc.).
- FPGA parameters (fpgapara) that control the hardware behaviour.
- The current value variable to hold the current set value of the functions.
- Limit values (min/max).
- The initial value function pointer for storing default value.
- The handler function pointer to hold the command implementation functions.

5.3.10 Parsing Function Modification

The Parsing Function is modified in such a way that it works together with the Command Structure defined. Both are dependent on each other. When the ASCII Command is received by the Sensor and passed on to the Parsing Function, each command Name entered by the user is compared to the command Name member of the ASCIICommand Structure using string comparison function along with index of the structure. Similarly, the function execution logic is also added as a member to this command structure so that the parsing and execution functions works together efficiently, and every behaviour of an ASCII command can be accessed at a single place.

5.3.11 Limit check Function for error handling

A decision is made to implement a limit check function before the execution of each ASCII command to prevent the application of outbound values to the sensor which may cause harm.

5.4 Workflow: Implementation for ASCII Command Acknowledgement

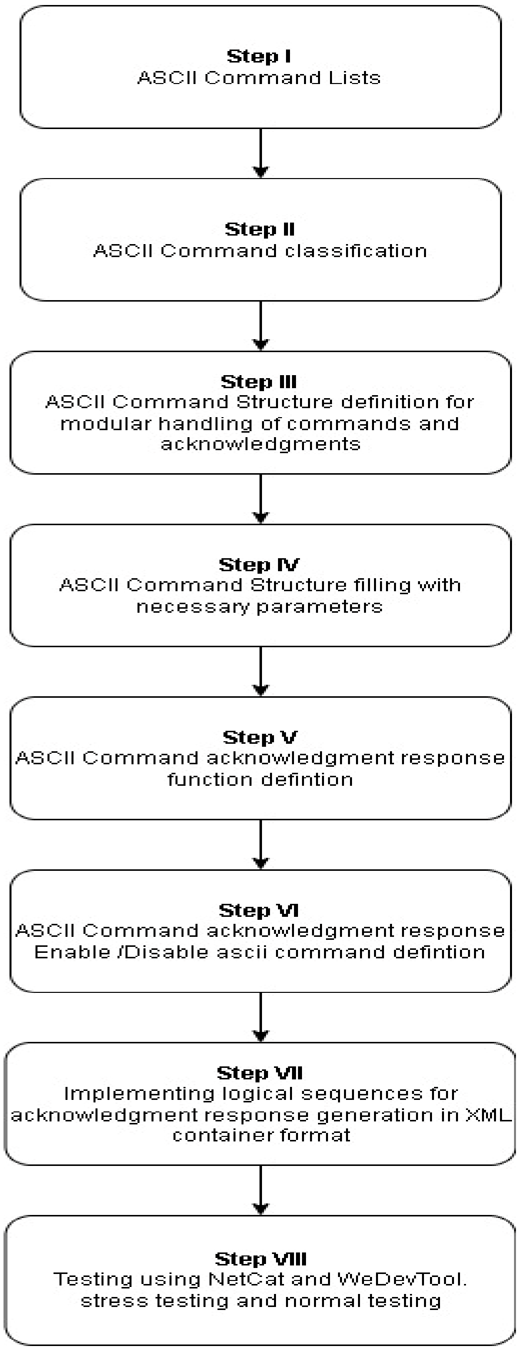

Step I
ASCII Command Lists
Step II
ASCII Command classification
Step III
ASCII Command Structure definition for modular handling of commands and acknowledgments
Step IV
ASCII Command Structure filling with necessary parameters
Step V
ASCII Command acknowledgment response function defintion
Step VI
ASCII Command acknowledgment response Enable /Disable ascii command defintion
Step VII
Implementing logical sequences for acknowledgment response generation in XML container format
Step VIII
Testing using NetCat and WeDevTool. stress testing and normal testing

Figure 8 workflow ASCII command acknowledgment implementation

The Figure 8 shows the workflow of ASCII command acknowledgment implementation. The steps required shown in the above flow chart are described below. They are:

Step I

As a first step of the final approach of creating an acknowledgment handler for ASCII Commands all the ASCII Commands present were studied and listed. The list of ASCII Commands is given at the end of the document.

These Command were listed in an excel sheet along with their predefined macro names and command types for documentation purposes.

Step II

The next step is to classify and group the ASCII commands under different command types to prepare them for acknowledgment generation.

Step III

The next step is to create a command structure which act as a centralized register point for all ASCII commands and their parameters. Then there is a need to modify the Parsing function which is the command parser function to integrate with the command structure.

Step IV

The next step after classifying commands under different categories is to fill the structure containing information necessary for acknowledgment response system with respective data. This includes command names, command types, minimum and maximum value limits, command handler function, default value functions and so on. The command execution logics are assigned to command handler functions and before each execution logic, a limit check functionality is implemented to prevent the outbound values from entering the execution loop.

Step V

The next step after filling the command structure with respective information is to create the ASCII Command acknowledgment handler function.

This function handles the acknowledgment responses for the ASCII Commands. The provided function is a function, which will be responsible for several actions that can be taken depending on the type of command passed to it. This function accepts several arguments: command type, input value, execution time and a bunch of other necessary parameters. In fact, this function deals with multiple types of commands. This function includes the logic required for implementing the desired functionality for different command types and sending the corresponding results to a new function which is the XML Container function. This is a function that is implemented for gathering the information as XML tags and sending them over the socket as an acknowledgment response to the user.

Step VI

The next step after implementing the acknowledgment generation function is to create an ASCII command which would be used to enable or disable the newly created acknowledgment response feature**.**

Step VII

The next step after implementing the acknowledgment ASCII Command is to create a logical sequence between all these procedures and create a workflow that would invoke the acknowledgment response feature and generate an acknowledgment response in the XML container format.

Step VIII

The final step after creating a logical sequence for generating the acknowledgement response is to test the response. There are two modes of testing to be done to ensure that the acknowledgment response system meets all the functional and non-functional requirements as discussed previously. They are **normal testing** and **stress testing**.

- **Normal testing:** This test is done just by firing ASCII Commands at random and checking its generated acknowledgment response.
- **Stress Testing:** This test is done to make the sensor run at high speeds and check the acknowledgement response and its stability. This can be achieved by firing certain ASCII Commands in a sequence and checking the network traffic simultaneously.

The two ways of testing the generated acknowledgment responses are using **NetCat** and **WeDevTool**.

- **Netcat:** Netcat, often abbreviated as '**nc**', is a versatile networking utility that offers a plethora of functionalities. nc permits users to establish connections, read and write data across networks, scan for open ports and perform network testing and analysis. **NetCat** Command Syntax: **nc [options] host ip port**. [9]
- **WeDevTool:** WeDevTool is a developer application which is used to connect to the sensor using TCP/IP and is modified as a part of design decisions to incorporate an acknowledgement window which would display the generated acknowledgement response when the acknowledgment feature is enabled.

5.5 melApp Architecture with Acknowledgement Handler

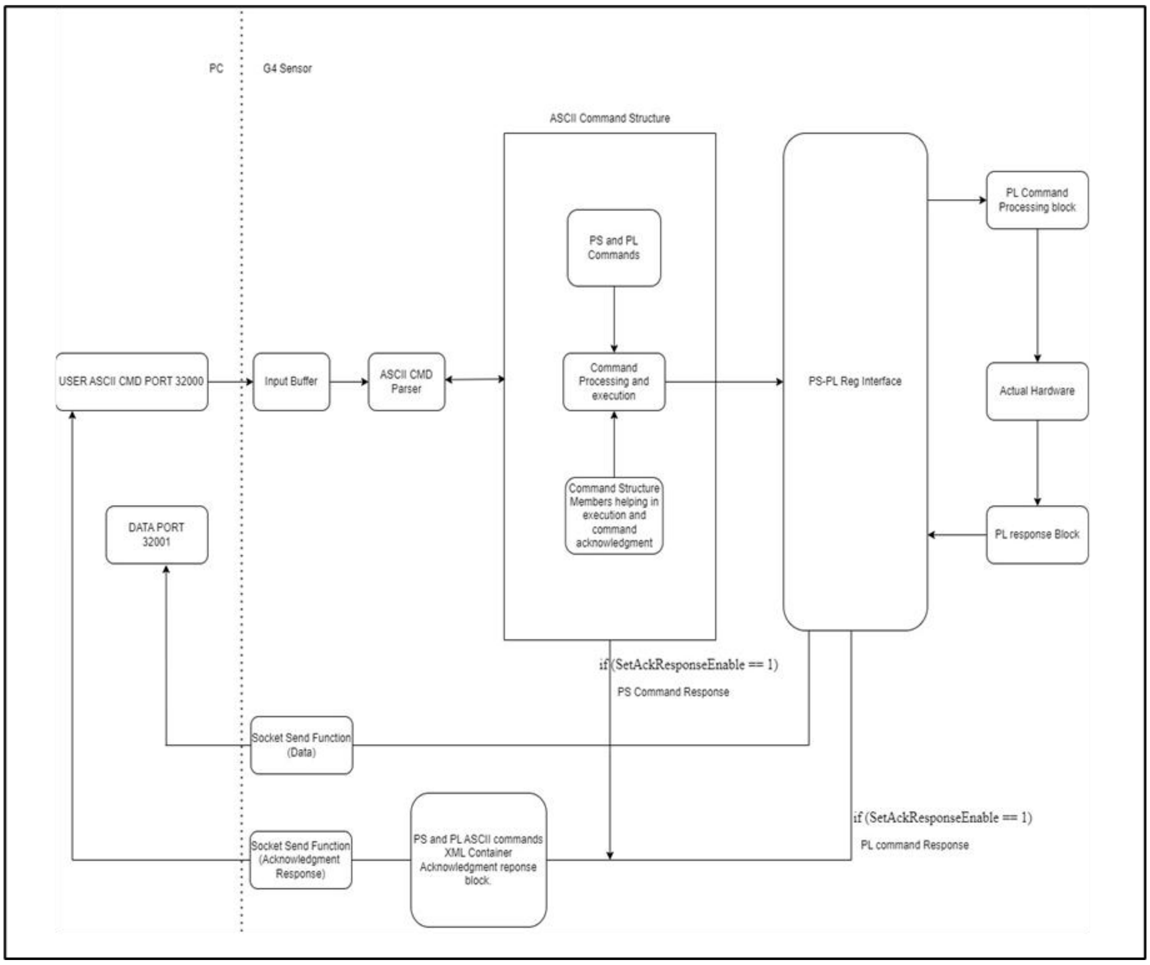

PC
G4 Sensor
ASCII Command Structure
PS and PL Commands
Command Processing and execution
Command Structure Members helping in execution and command acknowledgment
USER ASCII CMD PORT 32000
Input Buffer
ASCII CMD Parser
PS-PL Reg Interface
PL Command Processing block
Actual Hardware
PL response Block
DATA PORT 32001
if (SetAckResponseEnable == 1)
PS Command Response
Socket Send Function (Data)
Socket Send Function (Acknowledgment Response)
PS and PL ASCII commands XML Container Acknowledgment reponse block.
if (SetAckResponseEnable == 1)
PL command Response

Figure 9 melApp architecture (with acknowledgment handler)

Figure 9 shown above represents the outline of modified firmware(melApp) architecture with the inclusion of acknowledgment response system.

1. The ASCII commands are sent through **USER ASCII CMD PORT 32000**.
2. The incoming ASCII commands are stored in **Input Buffer** and passed on to the **ASCII CMD Parser** for parsing procedure.
3. The command parser and the ASCII Command structure work together to complete the command execution and prepare the command for generation of acknowledgement response by assigning corresponding parameters to the structure members.
4. The PS Commands (configuration commands) are processed and executed in **Command Processing and execution.**
5. The PL Commands (action commands) are passed on to the **PL Processing Block** for its execution and the communication takes place through **PS-PL Register Interface**.
6. The execution PL Commands are reflected in the **actual hardware** including camera, LED and peripherals.

7. The response from the PL after the execution is generated by the PL response block and is communicated with PS through the **PS-PL Register Interface**.
8. The acknowledgment response is generated if **SetAckResponseEnable** is set to 1, and the Acknowledgement generation which includes PS and PL command responses are generated and handled in the **PS and PL ASCII command XML Container Acknowledgement response block.**
9. The response data is sent back to PC Applications like WeDevTool through **DATA PORT 32001** using **socket send**.
10. The Acknowledgement response data is sent back to PC Applications like WeDevTool through **ASCII cmd PORT 32000** using **socket send**.

Chapter 6

IMPLEMENTATION

The primary objective of this thesis project is to develop and implement an ASCII command acknowledgment response system for the ShapeDrive G4 product platform, thereby improving its configurability and integration capabilities within the diverse industrial environments.

This section focuses on the detailed explanation of the steps involved in the creation of parsing function modifications, ASCII Command structure and structure array, implementation of limit check function, acknowledgment response system and XML acknowledgment generation.

The best way to explain the implementation of all these steps is to create a workflow diagram as shown below in figure 10.

6.1 Implementation: Workflow Diagram

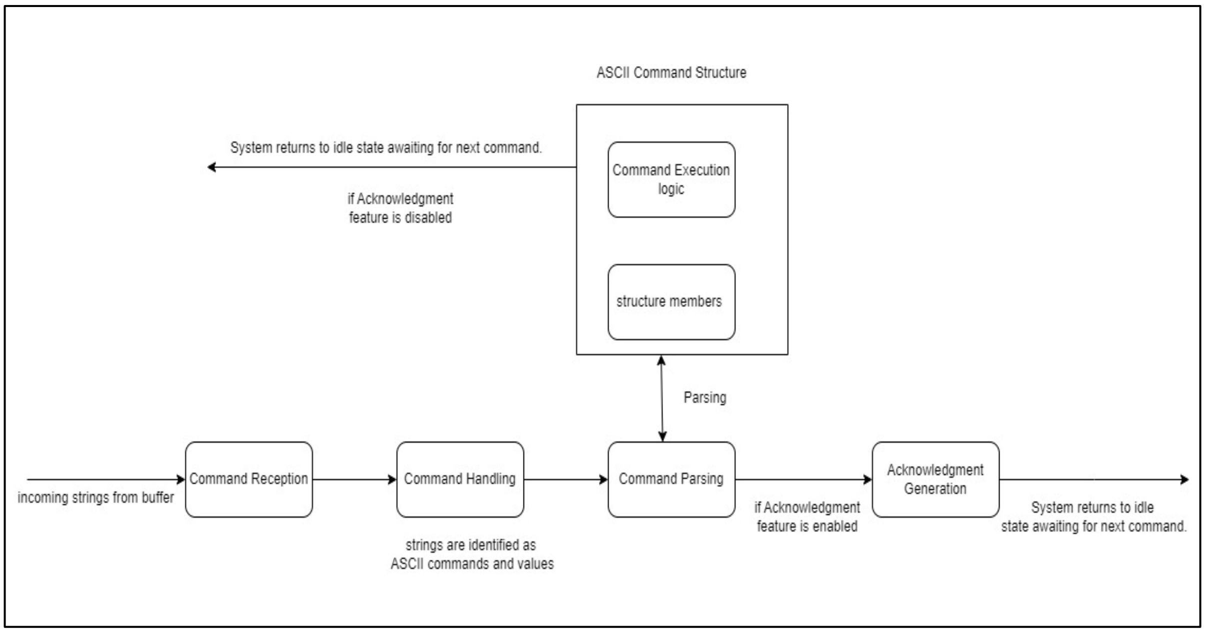

ASCII Command Structure
System returns to idle state awaiting for next command.
if Acknowledgment feature is disabled
Command Execution logic
structure members
Parsing
incoming strings from buffer
Command Reception
Command Handling
Command Parsing
if Acknowledgment feature is enabled
Acknowledgment Generation
System returns to idle state awaiting for next command.
strings are identified as ASCII commands and values

Figure 10 Implementation: workflow diagram

The ASCII command workflow is a process that starts from the moment a command is received by the sensor until the acknowledgment is generated and sent back. This process involves multiple stages, such as handling, parsing, validation, execution, and acknowledgment generation.

1. **Command Reception:** Here the sensor receives an ASCII command via a TCP/IP socket connection.
2. **Command Handling:** The received ASCII command is recognized and separated as the command name and value parameter.
3. **Command Parsing:** The command name is parsed with respect to existing commands in the command structure and is ready for validation.
4. **Command Execution:** The appropriate function handle in the structure is executed to carry out the command's task.
5. **Acknowledgment Generation:** The sensor generates an XML Container format acknowledgment response indicating the status of the command and its parameters.
6. **Completion:** After processing, the system returns to an idle state, awaiting the next command.

6.1.1 Command Reception

The sensor receives an ASCII command sent via a TCP/IP connection, which is initiated by a user interface. This is the first step in the command acknowledgment workflow.

The component involved in this process is TCP/IP Socket Layer.

- **TCP/IP Socket Layer:** The incoming command reaches the sensor through the network, specifically a TCP/IP socket connection. This ensures reliable transmission. It means that the command is guaranteed to reach the sensor. It is in form of a string message.

6.1.2 Command Handling

Once the command is received by the sensor, they are passed on to a handle function which does the job of identifying and separating command name and value parameter. This function recognizes the "=" sign as the major factor for classifying the command Name and value parameter separately. Then they are stored in a message buffer and are subjected to parsing function which is the **ParseAsciiCommand()** function.

- SetLEDPower: command name.
- 20: value parameter.
- **SetLEDPower = 20**
- This command means **"Set the LED Power of the Sensor to 20 percent"**.

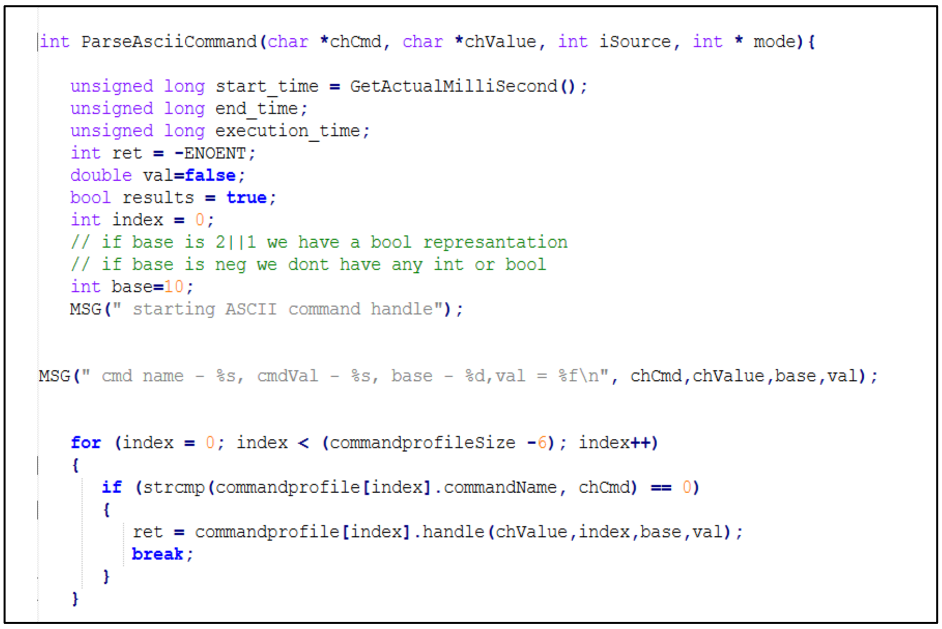
6.1.3 Command Parsing

Once the command is identified and separated as command name and value parameter, it is parsed using the ParseAsciiCommand() function. The incoming commands strings are compared against predefined command strings using **strcmp**. If a match is found, then the corresponding command execution and further acknowledgment generation takes place.

The ParseASCiiCommand Function block works in association with the command profile structure and this parsing function is modified and implemented as shown below in figures 11 and 12.

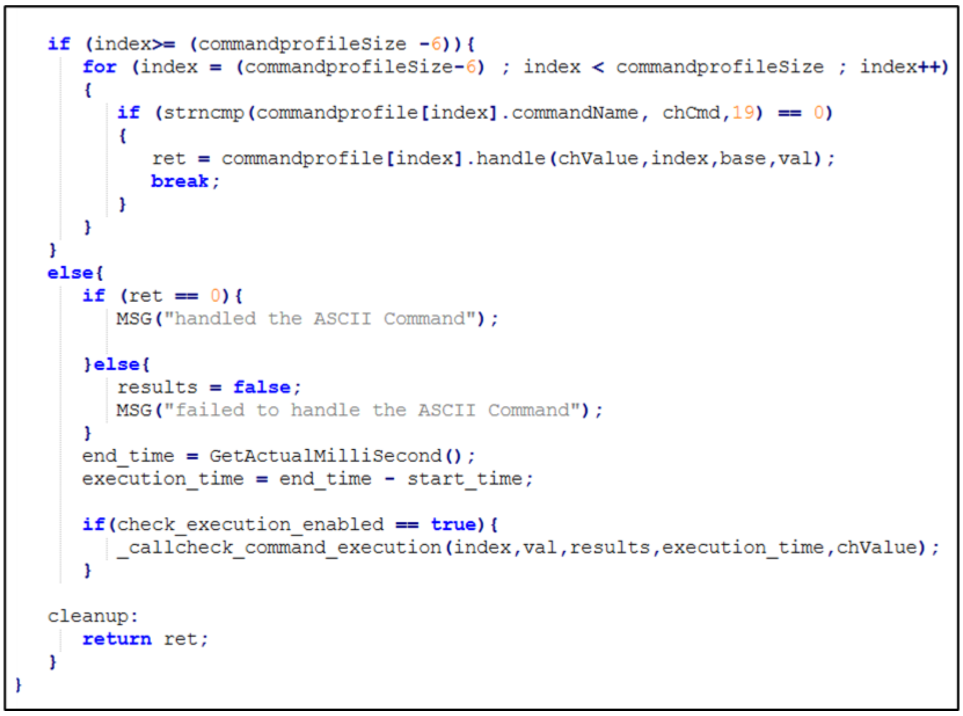
The incoming string from the user after being handled and identified as ASCII Command and the value parameter is then passed on to ParseAsciiCommand Function with arguments like chCmd, chValue, iSource, and mode.

- **chCmd** is the char pointer pointing towards the command Name.
- **chValue** is the char pointer pointing towards the value parameter.
- **iSource** is the parameter showing the incoming source of commands.
- **Mode** is the integer pointer pointing to modes of certain functionalities.



```
int ParseAsciiCommand(char *chCmd, char *chValue, int iSource, int * mode){

    unsigned long start_time = GetActualMilliSecond();
    unsigned long end_time;
    unsigned long execution_time;
    int ret = -ENOENT;
    double val=false;
    bool results = true;
    int index = 0;
    // if base is 2||1 we have a bool represantation
    // if base is neg we dont have any int or bool
    int base=10;
    MSG(" starting ASCII command handle");


MSG(" cmd name - %s, cmdVal - %s, base - %d,val = %f\n", chCmd,chValue,base,val);


    for (index = 0; index < (commandprofileSize -6); index++)
    {
        if (strcmp(commandprofile[index].commandName, chCmd) == 0)
        {
            ret = commandprofile[index].handle(chValue,index,base,val);
            break;
        }
    }
```

Figure 11 Parse ASCII Command Function part I



```
    if (index>= (commandprofileSize -6)){
        for (index = (commandprofileSize-6) ; index < commandprofileSize ; index++)
        {
            if (strncmp(commandprofile[index].commandName, chCmd,19) == 0)
            {
                ret = commandprofile[index].handle(chValue,index,base,val);
                break;
            }
        }
    }
    else{
        if (ret == 0){
            MSG("handled the ASCII Command");

        }else{
            results = false;
            MSG("failed to handle the ASCII Command");
        }
        end_time = GetActualMilliSecond();
        execution_time = end_time - start_time;

        if(check_execution_enabled == true){
            _callcheck_command_execution(index,val,results,execution_time,chValue);
        }

    cleanup:
        return ret;
    }
}
```

Figure 12 Parse ASCII Command Function part II

The function includes many variable declarations including ret, val, results, index, base.

- **ret** is the variable indicating the return value and is of integer type. It returns either 0 or error code indicating whether the command has been handled or not.
- **val** is the variable of type double and is used to store the incoming value parameters along with the ASCII Commands.
- **results** is the variable of type bool which returns true or false based on the value of ret variable and it is used as a helper variable for acknowledgment response function for generating acknowledgement depending on the command execution status
- **index** is the variable used for knowing the index of ASCII command from the commandprofile array.
- **base** denotes the type of the incoming value parameter like int, float, bool and so on.
- **commandprofileSize** This variable represents the total number of commands available in the commandprofile array. The loop will run from 0 to commandprofileSize - 1.
- **commandprofile** This is a structure array containing members which are identified and defined for each ASCII Command.
- **execution_time** There is a variable execution_time which is the difference between of end_time and start_time. This gives the time taken for an ASCII command to complete it execution starting from the parsing stage. This is then printed under one of the tag **execution_time** in the XML container acknowledgment response.

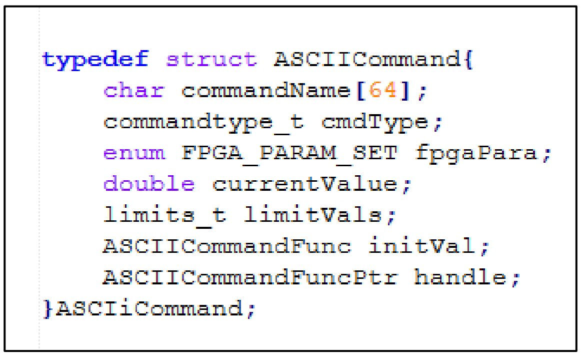
6.1.3.1 Logical workflow

1. The first for loop here that iterates over the elements of commandprofile, except the last 6 ASCII Commands as they are parsed differently due to need for using strncmp instead of strcmp for more accurate parsing. Inside the loop there is string comparison taking place based on the incoming command name from the user and the predefined command name defined inside the commandprofile structure array for proving the existence of the command.
2. Once the command existence is proven to be true, the handle function pointer which is one of the members of the commandprofile array which holds the command execution logic is invoked and command execution takes place and subsequently the sensor operation takes place. This handle function returns either 0 or an error code based on the state of command execution and this value is stored in ret.
3. Based on the value of ret a debug message is shown in the firmware side indicating the success or failure of the handling ASCII command.
4. This proves that the ParseAsciiCommand function is the function where the whole command handling takes place.

The ParseAsciiCommand function parses a given ASCII command and looks for it in the predefined commandprofile array, and if it finds a match it assigns corresponding the index to it, calls the corresponding function handle associated with that command. It passes the command's associated value and other parameters like index, base, and val to the handle function. Once the function is executed, the loop is exited.

6.1.4 Command Execution and Command Mapping

The ASCIICommand structure is the structure which defines the properties of each command. This structure contains fields for holding its members like the command name, command type, FPGA parameter, current value, limit values, and a function pointer that references the handler responsible for processing the command and another function pointer which holds the default values of commands. By encapsulating all these properties into a single structure array, the system ensures that commands are processed uniformly and efficiently. A common structure that defines and manages various command profiles will help for command mapping and would act as a reference point where the commands are handled. This handler manages a list of commands, each associated with certain parameters, types, and range limits, and they determine how the system should handle each command and generate acknowledgment response.



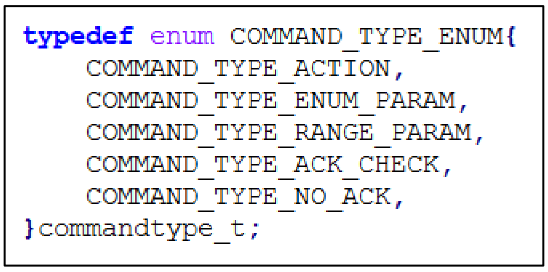
```
typedef struct ASCIICommand{
    char commandName[64];
    commandtype_t cmdType;
    enum FPGA_PARAM_SET fpgaPara;
    double currentValue;
    limits_t limitVals;
    ASCIICommandFunc initVal;
    ASCIICommandFuncPtr handle;
}ASCIiCommand;
```

Figure 13 ASCIICommand Structure definition

The ASCIICommand structure defines parameters for each command and is shown in the figure 13. This structure contains:

1. **char commandName:** This member present in the structure holds the command name of each command. It is of type char and is used for command parsing process and command acknowledgment process where each command entered by the user is compared using string comparison technique to prove its existence. This commandName is also used in the **command** tag of XML acknowledgment response.

2. **commandtype_t cmdType:**



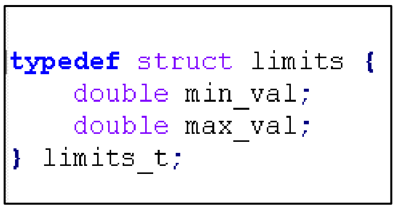
```
typedef enum COMMAND_TYPE_ENUM{
    COMMAND_TYPE_ACTION,
    COMMAND_TYPE_ENUM_PARAM,
    COMMAND_TYPE_RANGE_PARAM,
    COMMAND_TYPE_ACK_CHECK,
    COMMAND_TYPE_NO_ACK,
}commandtype_t;
```

Figure 14 commandtype_t structure

The figure 14 represents the member present in the structure that holds the command type for each command. It is a structure containing enumerated type definitions for command types and is named as commandtype_t. The **cmdType** is used for command acknowledgment process where the command acknowledgment function refers the respective command type member of each ASCII command and generates the corresponding acknowledgement response.

3. **double currentValue:** This member present in the structure holds the current value for each ASCII command after its execution. It is of type double and is used for command acknowledgment process where the values which are applied to the system are stored in respective currentValue members and are displayed under the tag **current** of XML acknowledgment response.

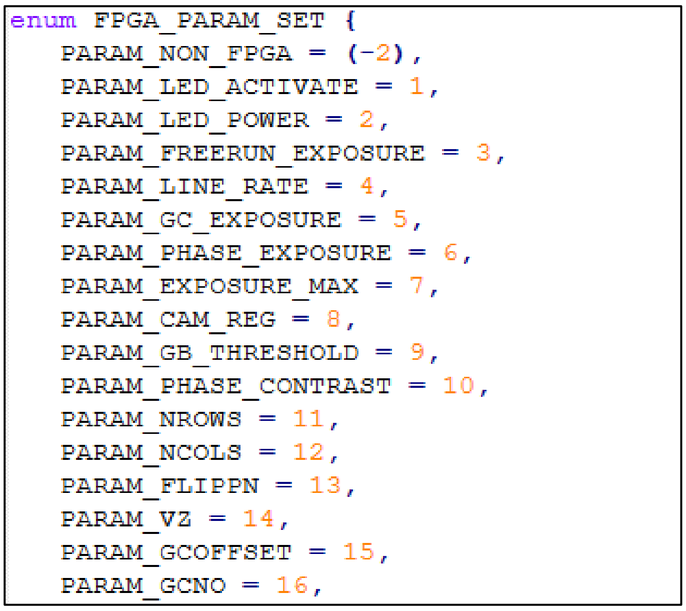
4. **limits_t limitVals:** The figure 15 represents the member present in the structure that holds the limit values for each ASCII command. It includes minimum and maximum value limits. It is a structure containing min_val and max_val of type double and is names as limits_t. This member is used for command validation and acknowledgement process. Each ASCII command after being parsed are checked against a function for validating the input values and executing the next steps accordingly. The limits_t are displayed under the tag **min** and **max** in the XML acknowledgment response.



```
typedef struct limits {
    double min_val;
    double max_val;
} limits_t;
```

Figure 15 limits_t structure

5. **enum FPGA_PARAM_SET fpgapara:** The figure 16 represents the member present in the structure that holds the enumerated type definitions for identifying the ASCII commands associated with FPGAs that control hardware functionalities and are used inside acknowledgment function as an argument for obtaining the current value(**currentValue**) of the corresponding FPGA command after its application on to the sensor.

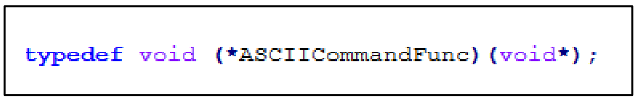


```
enum FPGA_PARAM_SET {
   PARAM_NON_FPGA = (-2),
   PARAM_LED_ACTIVATE = 1,
   PARAM_LED_POWER = 2,
   PARAM_FREERUN_EXPOSURE = 3,
   PARAM_LINE_RATE = 4,
   PARAM_GC_EXPOSURE = 5,
   PARAM_PHASE_EXPOSURE = 6,
   PARAM_EXPOSURE_MAX = 7,
   PARAM_CAM_REG = 8,
   PARAM_GB_THRESHOLD = 9,
   PARAM_PHASE_CONTRAST = 10,
   PARAM_NROWS = 11,
   PARAM_NCOLS = 12,
   PARAM_FLIPPN = 13,
   PARAM_VZ = 14,
   PARAM_GCOFFSET = 15,
   PARAM_GCNO = 16,
```

Figure 16 fpgapara structure

6. **ASCIICommandFunc initVal:**

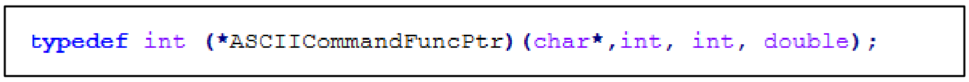
   The figure 17 represents the member present in the structure which is a function pointer for storing the default values of corresponding ASCII Commands when the sensor is turned on. The initVal is displayed inside the XML acknowledgement response under the tag **current** so that the user can know the default value of a particular functionality before applying the intended value.



```
typedef void (*ASCIICommandFunc)(void*);
```

Figure 17 initVal definition

7. **ASCIICommandFuncPtr handle:**



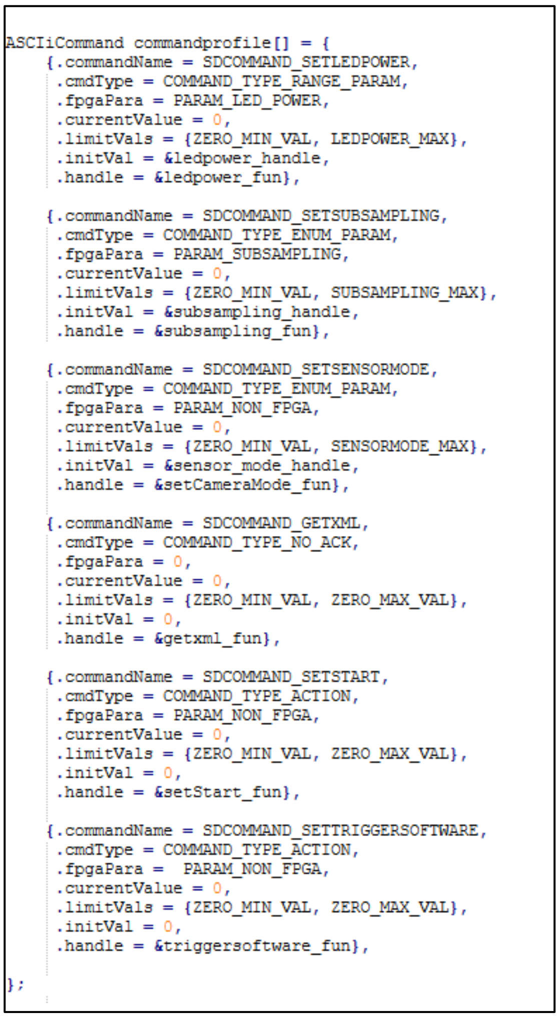
```
typedef int (*ASCIICommandFuncPtr)(char*,int, int, double);
```

Figure 18 handler function

The figure 18 represents the member present in the structure which is a function pointer that holds the execution logic of each ASCII command. Every ascii command after its parsing stage invokes the handle function and executes the logic implemented inside the handler function for the corresponding ASCII command.

After defining the ASCII Command structure the next step is defining the structure array commandprofile[] which creates a centralized definition of all commands, helping in both maintenance and extensibility.

6.1.4.1 commandprofile Structure Array Definition



```
ASCIiCommand commandprofile[] = {
    {.commandName = SDCOMMAND_SETLEDPOWER,
     .cmdType = COMMAND_TYPE_RANGE_PARAM,
     .fpgaPara = PARAM_LED_POWER,
     .currentValue = 0,
     .limitVals = {ZERO_MIN_VAL, LEDPOWER_MAX},
     .initVal = &ledpower_handle,
     .handle = &ledpower_fun},

    {.commandName = SDCOMMAND_SETSUBSAMPLING,
     .cmdType = COMMAND_TYPE_ENUM_PARAM,
     .fpgaPara = PARAM_SUBSAMPLING,
     .currentValue = 0,
     .limitVals = {ZERO_MIN_VAL, SUBSAMPLING_MAX},
     .initVal = &subsampling_handle,
     .handle = &subsampling_fun},

    {.commandName = SDCOMMAND_SETSENSORMODE,
     .cmdType = COMMAND_TYPE_ENUM_PARAM,
     .fpgaPara = PARAM_NON_FPGA,
     .currentValue = 0,
     .limitVals = {ZERO_MIN_VAL, SENSORMODE_MAX},
     .initVal = &sensor_mode_handle,
     .handle = &setCameraMode_fun},

    {.commandName = SDCOMMAND_GETXML,
     .cmdType = COMMAND_TYPE_NO_ACK,
     .fpgaPara = 0,
     .currentValue = 0,
     .limitVals = {ZERO_MIN_VAL, ZERO_MAX_VAL},
     .initVal = 0,
     .handle = &getxml_fun},

    {.commandName = SDCOMMAND_SETSTART,
     .cmdType = COMMAND_TYPE_ACTION,
     .fpgaPara = PARAM_NON_FPGA,
     .currentValue = 0,
     .limitVals = {ZERO_MIN_VAL, ZERO_MAX_VAL},
     .initVal = 0,
     .handle = &setStart_fun},

    {.commandName = SDCOMMAND_SETTRIGGERSOFTWARE,
     .cmdType = COMMAND_TYPE_ACTION,
     .fpgaPara =  PARAM_NON_FPGA,
     .currentValue = 0,
     .limitVals = {ZERO_MIN_VAL, ZERO_MAX_VAL},
     .initVal = 0,
     .handle = &triggersoftware_fun},

};
```

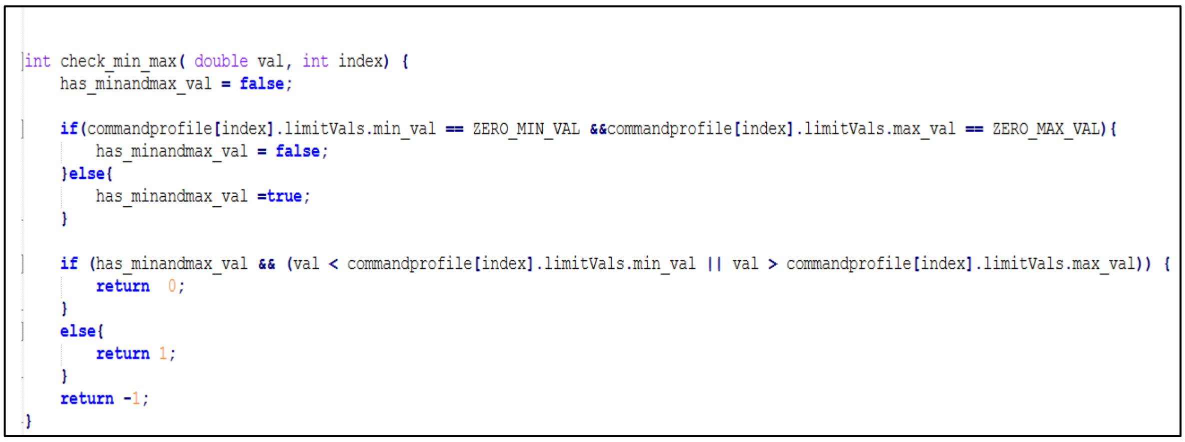
Figure 19 commandprofile structure array definition

- The commandprofile[] array shown in the above figure 18 is a collection of ASCIICommand structures where each element represents a specific command. The array shown above in figure 19 is a small part of the whole array and is used for explanation. Displaying the whole structure array is out of scope of this document. This array acts as a centralized reference point for all the supported ASCII commands and manages the command parameters dynamically.
- Each element in the array is an instance of the ASCIICommand structure. It is initialized with specific data and functions for the respective command. This centralized approach allows the system to dynamically identify, and process commands based on their commandName.
- This structure array acts as the reference point for generating the acknowledgment response. By using this structure and array, the acknowledgment system achieves modularity, making it straightforward to add or modify commands without significant changes to the overall architecture.
- The explanation of the implementation of each member of the structure array for an ASCII Command is described below to get a clear-cut picture. The example ASCII Command used is SetLEDPower.

1. **.commandName: SDCOMMAND_SETLEDPOWER.** It is the macro defined for the actual ASCII Command SetLEDPower.
2. **.cmdType: COMMAND_TYPE_RANGE_PARAM**. It means this command SetLEDPower is a command with value in range. This type also means if the user tries to set a value out of its valid range, the sensor will set its value to the nearest valid parameter.
3. **.fpgapara: PARAM_LED_POWER.** It is the enumerated definition used for getting the current value after the user tries to set a value for the SetLEDPower.
4. **.limitVals: {ZERO_MIN_VAL,LED_POWER_MAX}.** It is the macro defined for minimum and maximum possible values for SetLEDPower and they are 0 and 100 respectively.
5. **.initVal: &ledpower_handle** . This is a function pointer holding the default value of led power when the sensor is turned on.
6. **.handle: &ledpower_fun.** This is a function pointer holding the execution logic for the ASCII command.

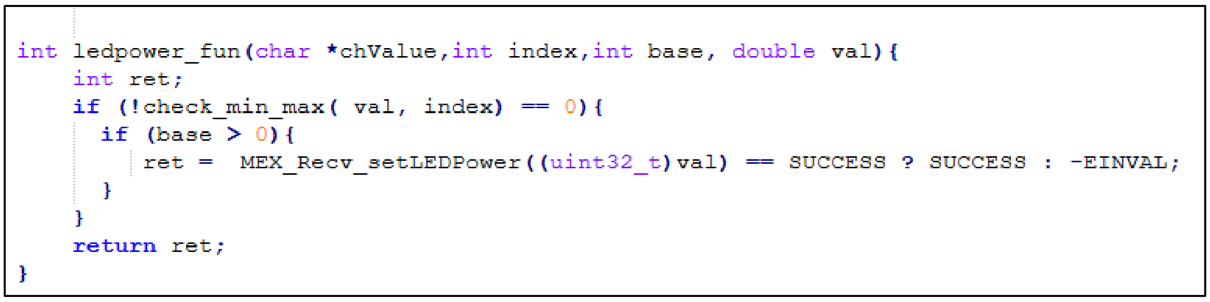
6.1.4.2 Limit check function

This function is implemented to check the validity of incoming value for each parameter and prevent the entry of invalid value to the execution logic. The function is defined and used as shown below in figures 20 and 21:



```
int check_min_max( double val, int index) {
    has_minandmax_val = false;

    if(commandprofile[index].limitVals.min_val == ZERO_MIN_VAL &&commandprofile[index].limitVals.max_val == ZERO_MAX_VAL){
        has_minandmax_val = false;
    }else{
        has_minandmax_val =true;
    }

    if (has_minandmax_val && (val < commandprofile[index].limitVals.min_val || val > commandprofile[index].limitVals.max_val)) {
        return  0;
    }
    else{
        return 1;
    }
    return -1;
}
```

Figure 20 limit check functionality definition



```
int ledpower_fun(char *chValue,int index,int base, double val){
    int ret;
    if (!check_min_max( val, index) == 0){
      if (base > 0){
        ret =  MEX_Recv_setLEDPower((uint32_t)val) == SUCCESS ? SUCCESS : -EINVAL;
      }
    }
    return ret;
}
```

Figure 21 limit check functionality usage

This function basically checks if the incoming value is out of range of the minimum or maximum value limits predefined for the ASCII Commands. If the value is out of range, then the command execution operation is limited and the value according to the command type would be applied to the sensor instead.

This ensures that only valid operations are performed on the hardware, thus protecting against potentially unsafe actions that could damage the hardware or produce wrong results.

6.1.4.3 Steps for adding new ASCII Commands to the structure array

1. Define the macro for new command.
2. Add the command name macro to the Structure array under the member **.commandName**.
3. Define the function for the new command with logic and assign it to the member **.handle**.
4. Initialize the rest of the parameters like cmdType, fpgapara, currentValue, initVal, and limitVals if they are present.

Now the implementation of ASCII command structure and command profile array is done, and the command execution takes place.

The next step is implementing the acknowledgement response function.

6.1.5 Acknowledgement Generation

The acknowledgment response system plays a vital role in ensuring the reliability and correctness of commands and their execution within the system. The acknowledgment response system designed and developed as a part of this thesis is an extension to the existing architecture of the system.

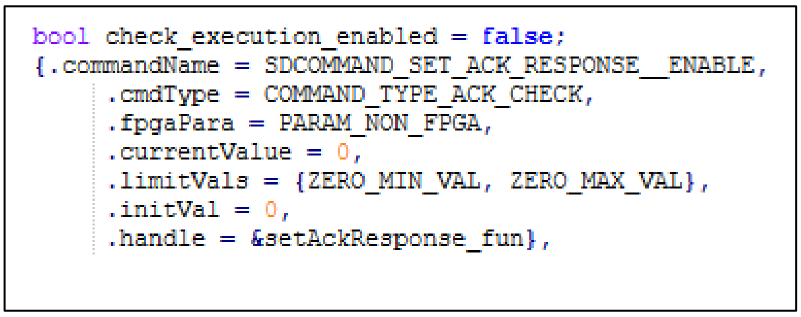
The acknowledgment response system implemented here aims to:

- Generates feedback on command execution and send it to the user, ensuring that every action, parameter change, or operation requested through the G4 sensor is properly acknowledged.
- Uses XML Container formats for acknowledgment, ensuring data readability and providing easy integration with various software environments.
- Incorporate validation mechanisms to prevent the entry of incorrect parameters, enhancing the safety and reliability of command execution.
- Enable configurability in the G4 series sensors to allow users to enable/disable the acknowledgment response functionality according to their use cases.
- Communicate execution results, including current value, user value, min and max values, execution time and the command name as responses.

6.1.5.1 Creation of SetAckResponseEnable ASCII Command.

The first step involved in implementing the acknowledgement response system and integrating this feature with the current firmware architecture is defining an ASCII Command which enables or disables this feature.

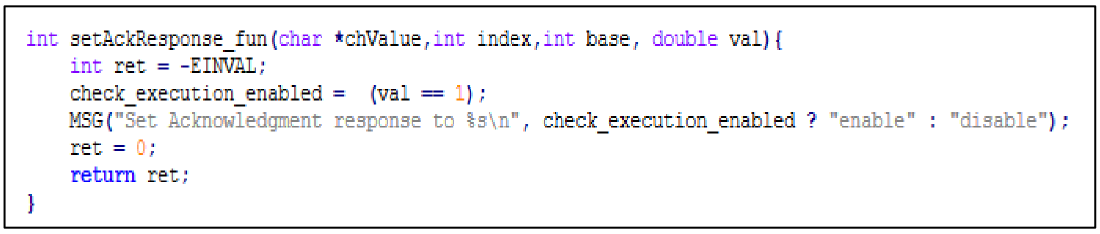
An ASCII Command **SetAckResponseEnable** is defined as follows:



```
bool check_execution_enabled = false;
{.commandName = SDCOMMAND_SET_ACK_RESPONSE__ENABLE,
     .cmdType = COMMAND_TYPE_ACK_CHECK,
     .fpgaPara = PARAM_NON_FPGA,
     .currentValue = 0,
     .limitVals = {ZERO_MIN_VAL, ZERO_MAX_VAL},
     .initVal = 0,
     .handle = &setAckResponse_fun},
```

Figure 22 SetAckResponseEnable in commandprofile Structure array.

The above figure 22 shows the definition of the **SetAckResponseEnable** inside the commandprofile structure array. A Boolean check_execution_enabled is also declared and initialized as false as default. This Boolean variable is assigned to true when the **SetAckResponseEnable** is assigned to value 1 and thus enabling functionality of feature acknowledgment response.



```
int setAckResponse_fun(char *chValue,int index,int base, double val){
    int ret = -EINVAL;
    check_execution_enabled =  (val == 1);
    MSG("Set Acknowledgment response to %s\n", check_execution_enabled ? "enable" : "disable");
    ret = 0;
    return ret;
}
```

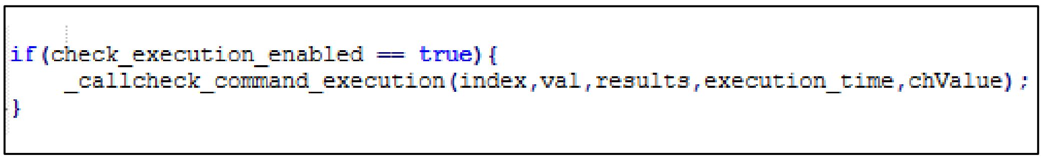
Figure 23 SetAckResponseEnable funciton handle definition

A function setAckResponse_fun is defined as shown in figure 23 The logic is defined such that this command enables the acknowledgment response which when user sends the ASCII Command **SetAckResponseEnable = 1.** A Boolean check_execution_enabled is assigned the value 1 and a debug message is printed in the logs as “Set Acknowledgment response to enable/disable”.

6.1.5.2 Workflow until Acknowledgment response is generated

Step 1: Enabling the acknowledgement response feature

When the ASCII Command **SetAckResponseEnable=1** is sent to the system. The acknowledgement response feature gets enabled. This is represented inside the ParseAsciiCommand function as a check condition. If check_execution_enabled is true, then a function _callcheck_command_execution is called which contains the acknowledgement response function. This is shown in figure 24.

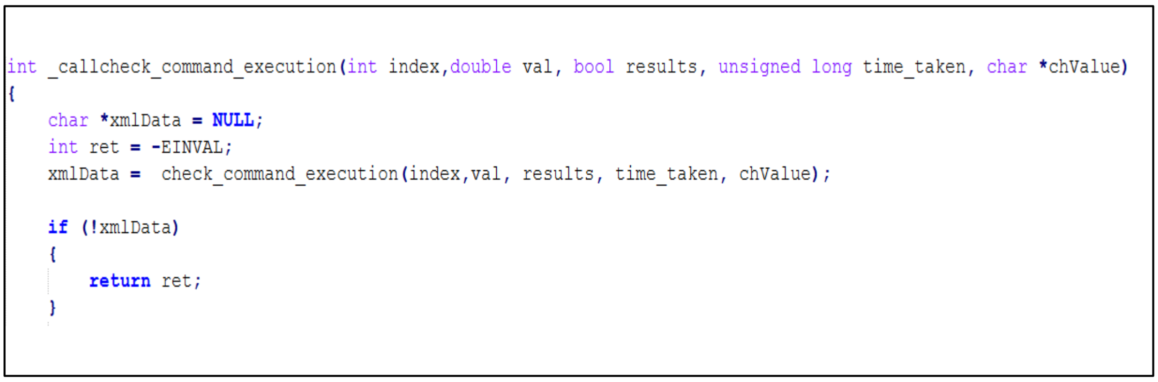


```
if(check_execution_enabled == true){
    _callcheck_command_execution(index,val,results,execution_time,chValue);
}
```

Figure 24 Acknowledgment response feature check condition

Step 2: Defining the _callcheck_command_execution function.

This function _callcheck_command_execution is a function that calls the main acknowledgement response function which is the check_command_execution and the function _CreateContainerSendOverSocket which sends the XML Container acknowledgement response generated over the socket to the user. This function even though is a single function. It can be divided into two parts for explanatory purposes. The first part calls the check_command_execution function and assigns it to xmlData variable as shown below. This function is of integer type takes in arguments like index, val, results, time_taken, and chValue. This is shown in the figure 25.

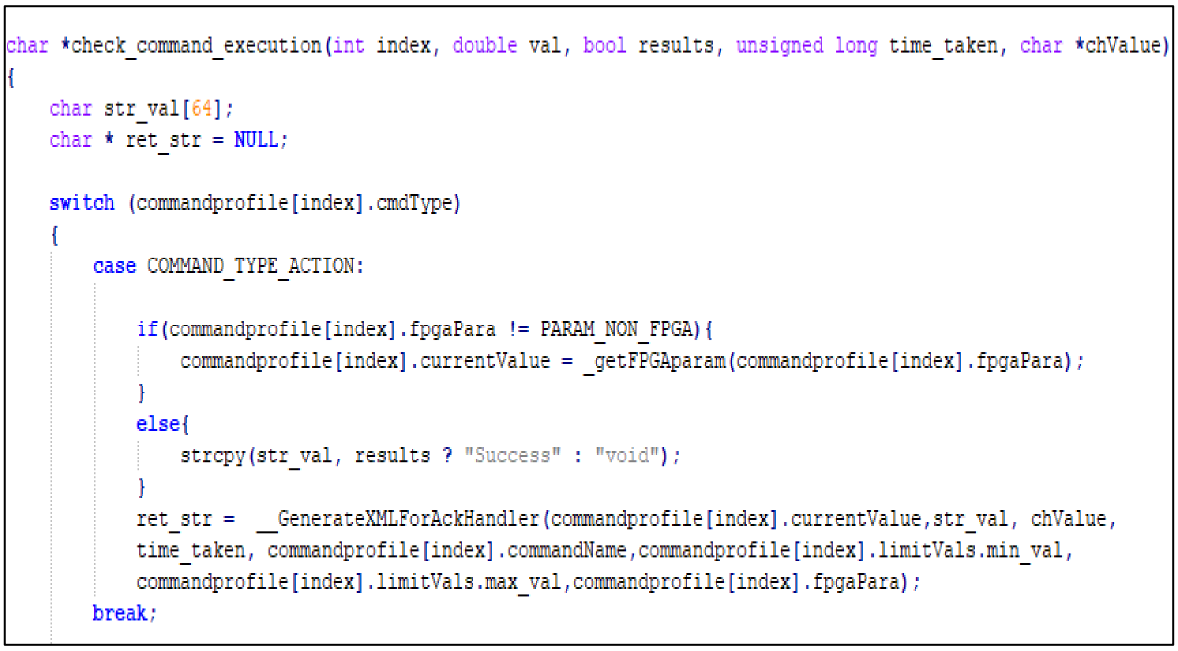


```
int _callcheck_command_execution(int index,double val, bool results, unsigned long time_taken, char *chValue)
{
    char *xmlData = NULL;
    int ret = -EINVAL;
    xmlData =  check_command_execution(index,val, results, time_taken, chValue);

    if (!xmlData)
    {
        return ret;
    }
```

Figure 25 _callcheck_command_execution part I

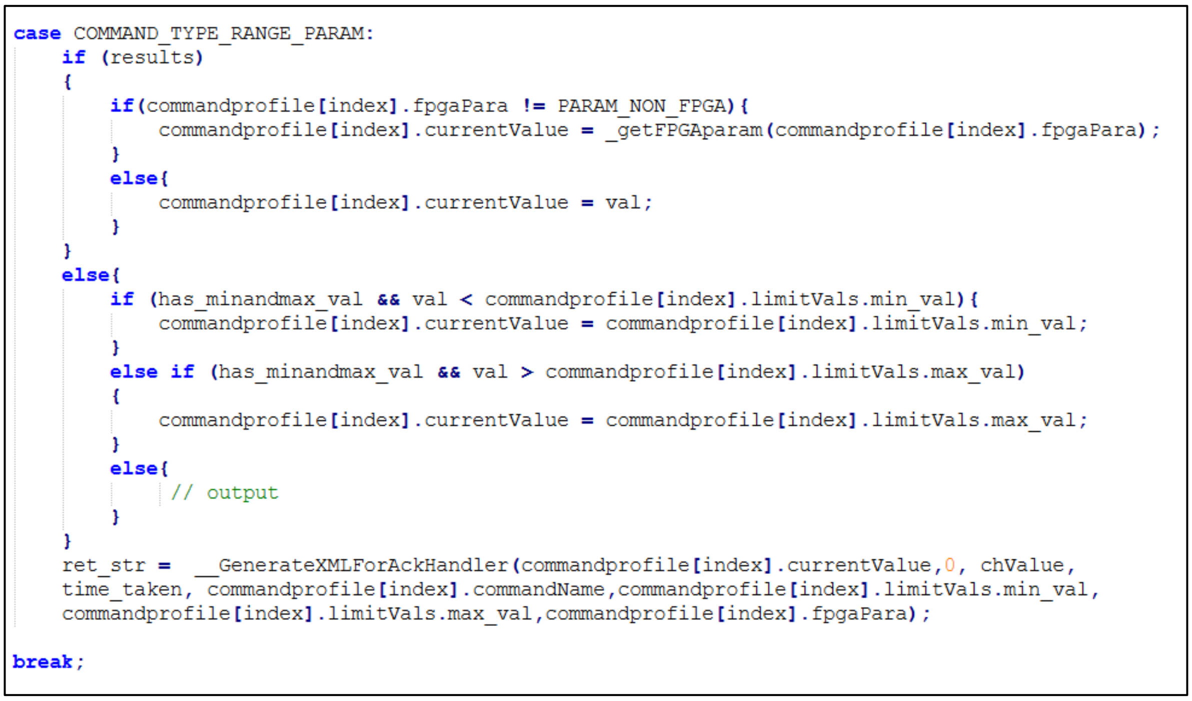
Step 3: Defining the check_command_execution function.

The function below in figure 26 takes in arguments like index, val, results, time_taken and chValue and executes the acknowledgment response functionality for different command types as shown below.



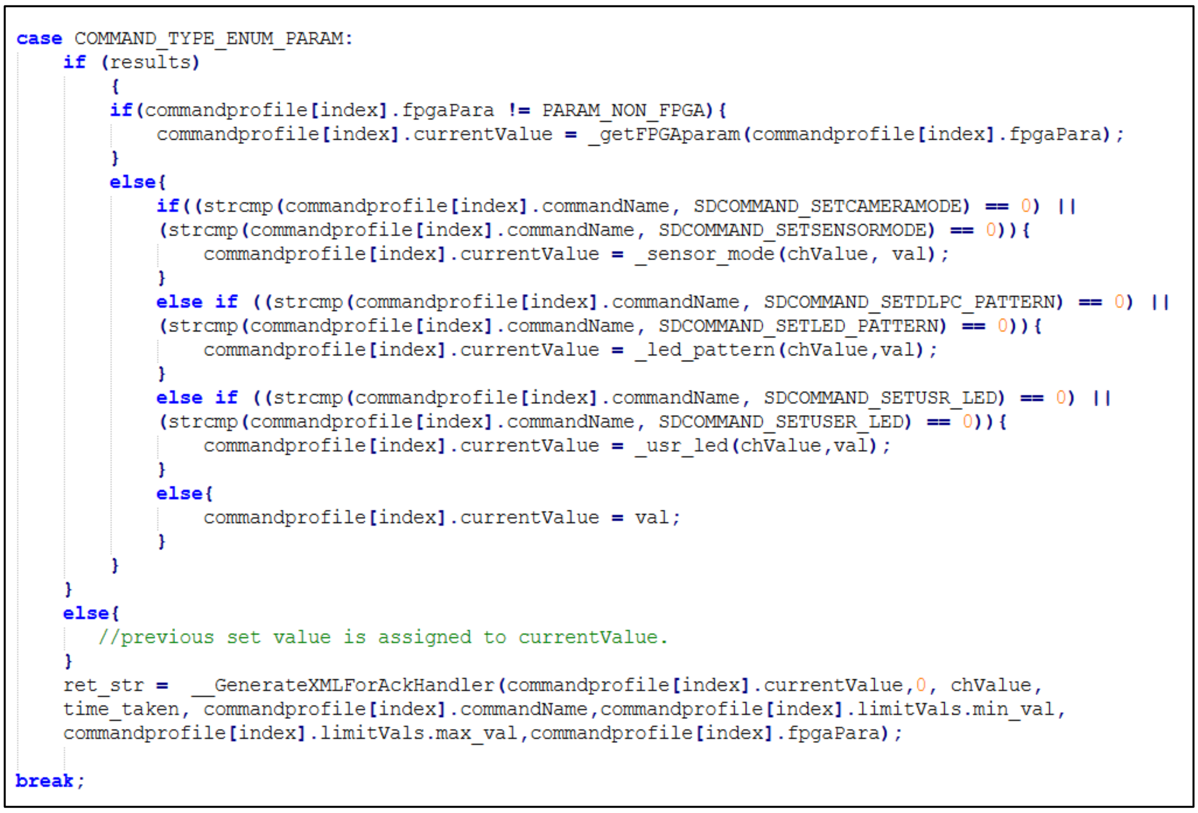
```c
char *check_command_execution(int index, double val, bool results, unsigned long time_taken, char *chValue)
{
    char str_val[64];
    char * ret_str = NULL;

    switch (commandprofile[index].cmdType)
    {
        case COMMAND_TYPE_ACTION:

            if(commandprofile[index].fpgaPara != PARAM_NON_FPGA){
                commandprofile[index].currentValue = _getFPGAparam(commandprofile[index].fpgaPara);
            }
            else{
                strcpy(str_val, results ? "Success" : "void");
            }
            ret_str =  __GenerateXMLForAckHandler(commandprofile[index].currentValue,str_val, chValue,
            time_taken, commandprofile[index].commandName,commandprofile[index].limitVals.min_val,
            commandprofile[index].limitVals.max_val,commandprofile[index].fpgaPara);
        break;
```

Figure 26 check_command_execution function code snippet part 1



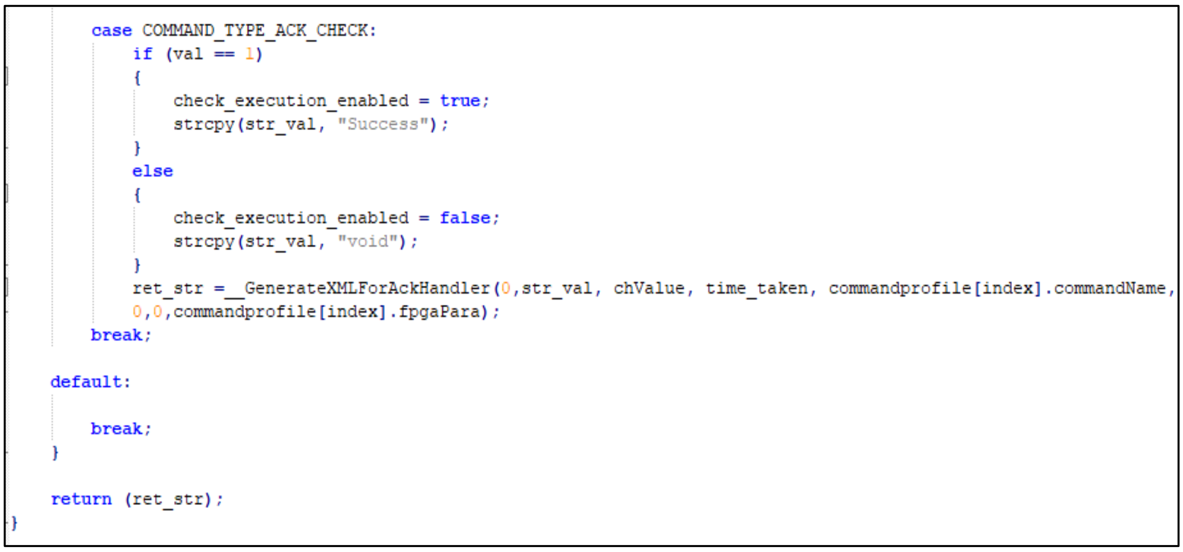
```c
case COMMAND_TYPE_RANGE_PARAM:
    if (results)
    {
        if(commandprofile[index].fpgaPara != PARAM_NON_FPGA){
            commandprofile[index].currentValue = _getFPGAparam(commandprofile[index].fpgaPara);
        }
        else{
            commandprofile[index].currentValue = val;
        }
    }
    else{
        if (has_minandmax_val && val < commandprofile[index].limitVals.min_val){
            commandprofile[index].currentValue = commandprofile[index].limitVals.min_val;
        }
        else if (has_minandmax_val && val > commandprofile[index].limitVals.max_val)
        {
            commandprofile[index].currentValue = commandprofile[index].limitVals.max_val;
        }
        else{
            // output
        }
    }
    ret_str =  __GenerateXMLForAckHandler(commandprofile[index].currentValue,0, chValue,
    time_taken, commandprofile[index].commandName,commandprofile[index].limitVals.min_val,
    commandprofile[index].limitVals.max_val,commandprofile[index].fpgaPara);

break;
```

Figure 27 check_command_execution code snippet part II



```
case COMMAND_TYPE_ENUM_PARAM:
    if (results)
        {
        if(commandprofile[index].fpgaPara != PARAM_NON_FPGA){
            commandprofile[index].currentValue = _getFPGAparam(commandprofile[index].fpgaPara);
        }
        else{
            if((strcmp(commandprofile[index].commandName, SDCOMMAND_SETCAMERAMODE) == 0) ||
            (strcmp(commandprofile[index].commandName, SDCOMMAND_SETSENSORMODE) == 0)){
                commandprofile[index].currentValue = _sensor_mode(chValue, val);
            }
            else if ((strcmp(commandprofile[index].commandName, SDCOMMAND_SETDLPC_PATTERN) == 0) ||
            (strcmp(commandprofile[index].commandName, SDCOMMAND_SETLED_PATTERN) == 0)){
                commandprofile[index].currentValue = _led_pattern(chValue,val);
            }
            else if ((strcmp(commandprofile[index].commandName, SDCOMMAND_SETUSR_LED) == 0) ||
            (strcmp(commandprofile[index].commandName, SDCOMMAND_SETUSER_LED) == 0)){
                commandprofile[index].currentValue = _usr_led(chValue,val);
            }
            else{
                commandprofile[index].currentValue = val;
            }
        }
    }
    else{
       //previous set value is assigned to currentValue.
    }
    ret_str =  __GenerateXMLForAckHandler(commandprofile[index].currentValue,0, chValue,
    time_taken, commandprofile[index].commandName,commandprofile[index].limitVals.min_val,
    commandprofile[index].limitVals.max_val,commandprofile[index].fpgaPara);

break;
```

Figure 28 check_command_execution code snippet part III

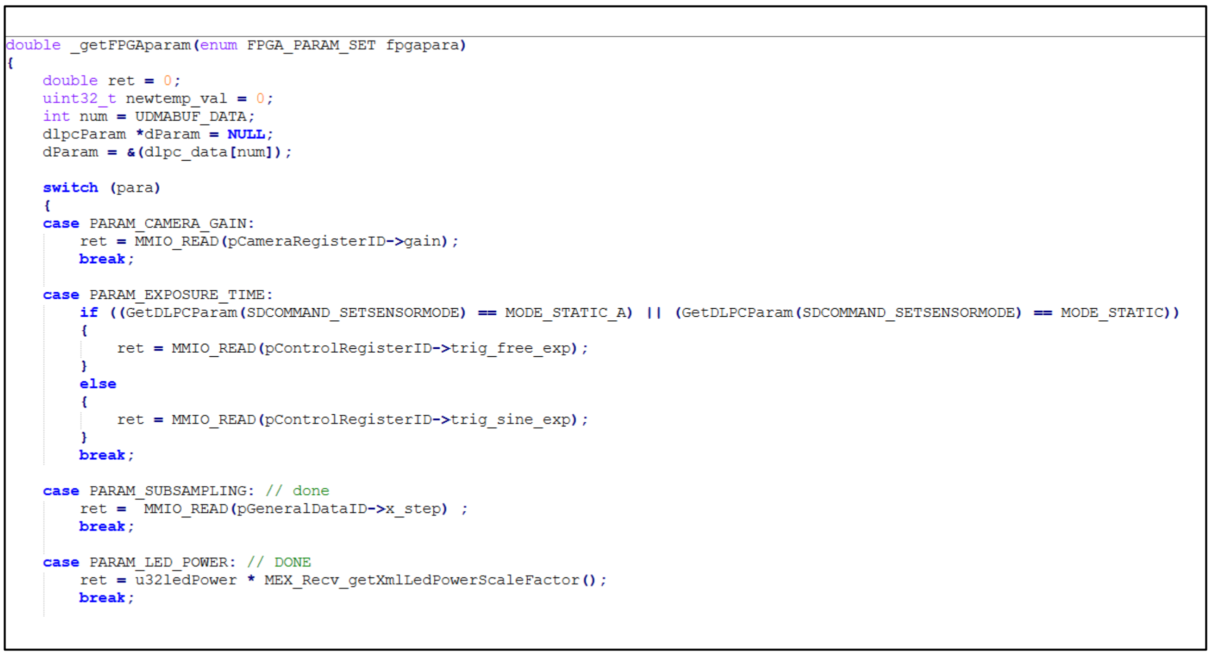


```
        case COMMAND_TYPE_ACK_CHECK:
            if (val == 1)
            {
                check_execution_enabled = true;
                strcpy(str_val, "Success");
            }
            else
            {
                check_execution_enabled = false;
                strcpy(str_val, "void");
            }
            ret_str =__GenerateXMLForAckHandler(0,str_val, chValue, time_taken, commandprofile[index].commandName,
            0,0,commandprofile[index].fpgaPara);
        break;

    default:

        break;
    }

    return (ret_str);
}
```

Figure 29 check_command_execution code snippet part IV

The sensor supports different types of commands, each with specific acknowledgment requirements. This is managed by the check_command_execution() function. Each ASCII command when fired will identify its corresponding command type from the structure array commandprofile using the parsing function and then based on this command type would invoke the cases in check_command_execution function and prepare the parameters for generating acknowledgment response. The different requirements to be met while implementing such acknowledgment functionalities is shown above in the figure 26,27,28 and 29 and are described in detail below:

- **Command Type Action (COMMAND_TYPE_ACTION):** These commands are executed and then acknowledged based on their completion status. The results are printed under the tag **current** in the XML Container format as either success or failure.
- **Range Parameters (COMMAND_TYPE_RANGE_PARAM):** Commands involving range validation (eg: SetExposureTime, SetLEDPower and so on) are validated to ensure they fall within predefined limits. Once the command execution takes place successfully, an acknowledgment response is generated for the true value condition. If they fall outside the allowable range or an error occurs during the command execution, then an acknowledgment response is generated which gives the current value (tag current in the XML Container format) as the nearest possible true value.
- **Enum Parameters (COMMAND_TYPE_ENUM_PARAM):** Commands with enumerated parameter values, such as (eg: SetSensorMode, SetLEDpattern and so on), are executed when the requested value is valid. Once the command execution takes place successfully acknowledgment response is generated for the true value condition. and if they fall outside the valid values or an error occurs during the command execution then an acknowledgment response is generated which gives the current value (tag **current** in the XML Container format) as the previous set true value. There are three commands currently handled specially under this command type as they have value parameters which are of type strings and integers. They had to handled separately because they must output the current value as an integer even if the input value was a string corresponding to it. These ASCII commands are SetSensorMode, SetLEDPattern, and SetUsrLED.
- **No ACK Parameter:** There are some ASCII commands SetReboot, GetXmlDescriptor, GetXmlDescriptorExpert and so on), which do not need to explicitly generate an acknowledgement response as they create visual impacts and indications when they are fired.
- **Acknowledgment function parameters (COMMAND_TYPE_ACK_CHECK):** This command type is similar to action command and are executed and then acknowledged based on their completion status. The results are printed under the tag **current** in the XML Container format as either success or failure. For each command type case in the check_command_execution function the parameters for acknowledgment response are assigned based on the success or failure of command execution. This is determined by the Boolean variable **results.**



```c
double _getFPGAparam(enum FPGA_PARAM_SET fpgapara)
{
    double ret = 0;
    uint32_t newtemp_val = 0;
    int num = UDMABUF_DATA;
    dlpcParam *dParam = NULL;
    dParam = &(dlpc_data[num]);

    switch (para)
    {
    case PARAM_CAMERA_GAIN:
        ret = MMIO_READ(pCameraRegisterID->gain);
        break;

    case PARAM_EXPOSURE_TIME:
        if ((GetDLPCParam(SDCOMMAND_SETSENSORMODE) == MODE_STATIC_A) || (GetDLPCParam(SDCOMMAND_SETSENSORMODE) == MODE_STATIC))
        {
            ret = MMIO_READ(pControlRegisterID->trig_free_exp);
        }
        else
        {
            ret = MMIO_READ(pControlRegisterID->trig_sine_exp);
        }
        break;

    case PARAM_SUBSAMPLING: // done
        ret =  MMIO_READ(pGeneralDataID->x_step) ;
        break;

    case PARAM_LED_POWER: // DONE
        ret = u32ledPower * MEX_Recv_getXmlLedPowerScaleFactor();
        break;
```

Figure 30 _getFPGAparam function code snippet

There are some ASCII commands which interacts directly with the hardware, specifically the FPGA, to ensure the current value of command results reflect the actual state of the hardware the function **_getFPGAparam** is implemented as shown in figure 30:

- **_getFPGAparam Function**: This function is used to get the current value of FPGA based ASCII commands. The switch-case block is used to determine which parameter is being accessed based on the value of fpgapara. This mechanism involves reading directly from hardware registers (MMIO_READ), ensuring the values are always up-to-date and they reflect the hardware's current state.

Step 4: XML Response Generation Function

The parameters which are obtained from the function check_command_execution is passed on the XML response generation function as arguments inorder to generate the acknowledgement response in XML container format.

The acknowledgment response system generates XML Container responses, and these contain the following data tags:

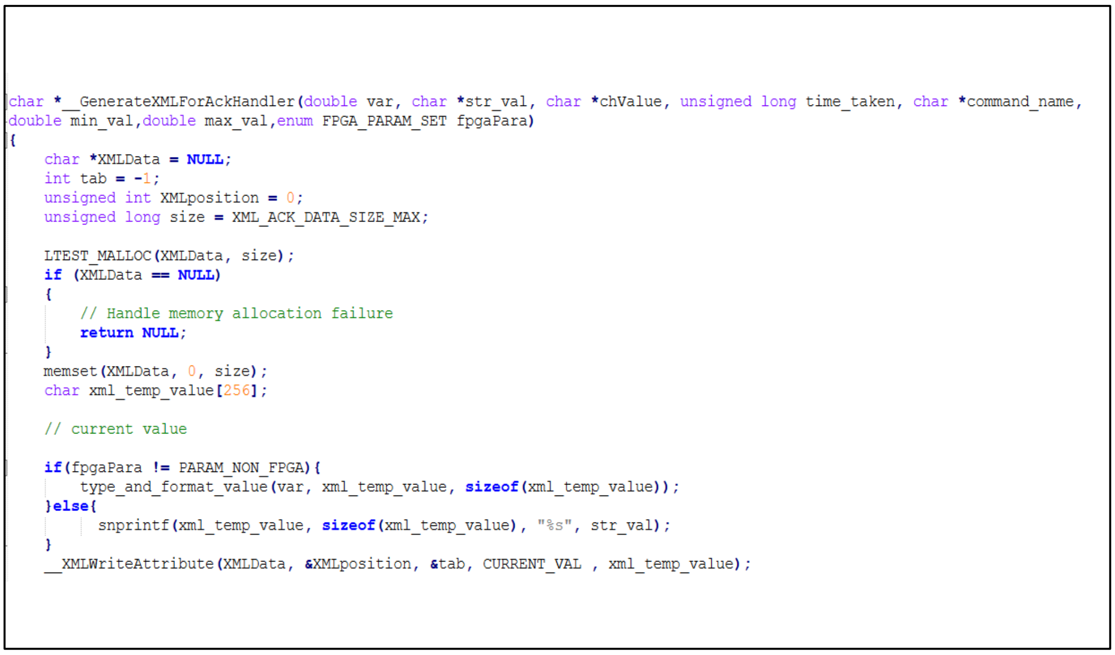
1. **current**
2. **user_value**
3. **min**
4. **max**
5. **execution_time**
6. **command**

This is managed using the __GenerateXMLForAckHandler function as shown below in figures.

Figure 31 shows the definition of __GenerateXMLForAckHandler function that generates parameters for XML data. The function takes arguments like var, str_val, chValue, time_taken, command_name, min_val, max_val, fpapara as inputs.

1. **str_val** is the char pointer that points to the current values of type string.
2. **var** is the variable that holds the current values of types int, float.
3. **chValue** is the char pointer that points to the user value.
4. **time_taken** is the variable holding the execution time value.
5. **command_name** is the variable holding the command name.
6. **min_val** is the variable holding the minimum value limit.
7. **max_val** is the variable holding the maximum value limit.
8. **fpgapara** is the variable holding the enum type of parameters for fpga ASCII commands.

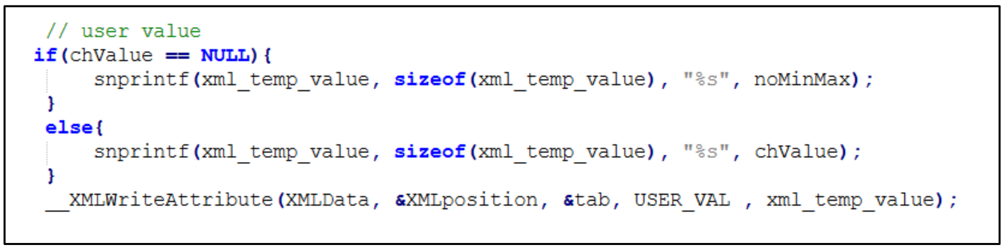
The function shown in figure 28 also shows how the parameter current value is converted to XML format to display under the tag **current.** This function constructs an XML document to include all relevant information related to the acknowledgment of a command. A function **type_and_format_value** is used to ensure the incoming current value is formatted appropriately for XML and matches with values in the XMLdescriptor generated. A dynamic memory allocation is necessary to store XMLData pointer and it is done using LTEST_MALLOC(XMLData, size).



```
char *__GenerateXMLForAckHandler(double var, char *str_val, char *chValue, unsigned long time_taken, char *command_name,
double min_val,double max_val,enum FPGA_PARAM_SET fpgaPara)
{
    char *XMLData = NULL;
    int tab = -1;
    unsigned int XMLposition = 0;
    unsigned long size = XML_ACK_DATA_SIZE_MAX;

    LTEST_MALLOC(XMLData, size);
    if (XMLData == NULL)
    {
        // Handle memory allocation failure
        return NULL;
    }
    memset(XMLData, 0, size);
    char xml_temp_value[256];

    // current value

    if(fpgaPara != PARAM_NON_FPGA){
        type_and_format_value(var, xml_temp_value, sizeof(xml_temp_value));
    }else{
          snprintf(xml_temp_value, sizeof(xml_temp_value), "%s", str_val);
    }
    __XMLWriteAttribute(XMLData, &XMLposition, &tab, CURRENT_VAL , xml_temp_value);
```

*Figure 31 __GenerateXMLForAckHandler code snippet for **current** XML generation*

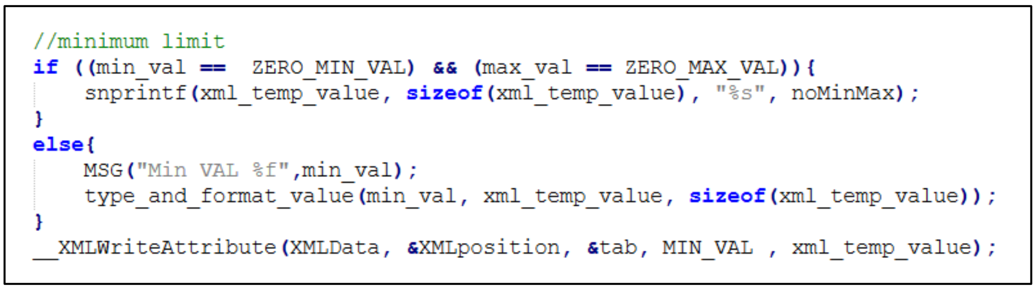
The function shown in figure 32 shows how the parameter user value is converted to XML format to display under the tag **user_vlaue**.



```
 // user value
if(chValue == NULL){
     snprintf(xml_temp_value, sizeof(xml_temp_value), "%s", noMinMax);
 }
 else{
     snprintf(xml_temp_value, sizeof(xml_temp_value), "%s", chValue);
 }
 __XMLWriteAttribute(XMLData, &XMLposition, &tab, USER_VAL , xml_temp_value);
```

*Figure 32 __GenerateXMLForAckHandler code snippet for **user_value** XML generation*

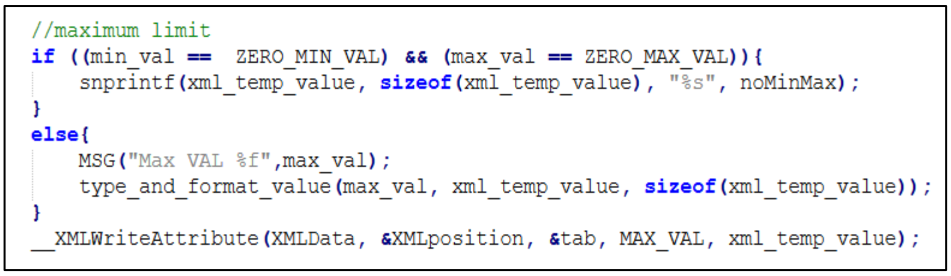
The function shown in figure 33 shows how the parameter minimum value limit is converted to XML format to display under the tag **min**.



```
//minimum limit
if ((min_val ==  ZERO_MIN_VAL) && (max_val == ZERO_MAX_VAL)){
    snprintf(xml_temp_value, sizeof(xml_temp_value), "%s", noMinMax);
}
else{
    MSG("Min VAL %f",min_val);
    type_and_format_value(min_val, xml_temp_value, sizeof(xml_temp_value));
}
__XMLWriteAttribute(XMLData, &XMLposition, &tab, MIN_VAL , xml_temp_value);
```

*Figure 33 __GenerateXMLForAckHandler code snippet for **min** XML generation*

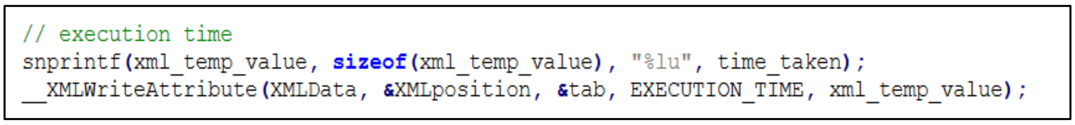
The function shown in figure 34 shows how the parameter maximum value limit is converted to XML format to display under the tag **max**.



```
//maximum limit
if ((min_val ==  ZERO_MIN_VAL) && (max_val == ZERO_MAX_VAL)){
    snprintf(xml_temp_value, sizeof(xml_temp_value), "%s", noMinMax);
}
else{
    MSG("Max VAL %f",max_val);
    type_and_format_value(max_val, xml_temp_value, sizeof(xml_temp_value));
}
__XMLWriteAttribute(XMLData, &XMLposition, &tab, MAX_VAL, xml_temp_value);
```

*Figure 34 __GenerateXMLForAckHandler code snippet for **max** XML generation*

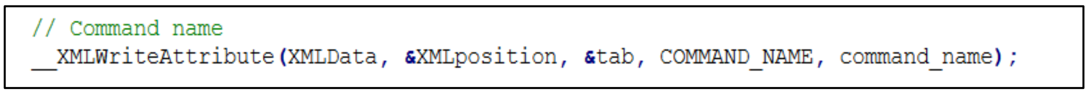
The function shown in figure 35 shows how the parameter execution time is converted to XML format to display under the tag **execution_time**.



```
// execution time
snprintf(xml_temp_value, sizeof(xml_temp_value), "%lu", time_taken);
__XMLWriteAttribute(XMLData, &XMLposition, &tab, EXECUTION_TIME, xml_temp_value);
```

*Figure 35 __GenerateXMLForAckHandler code snippet for **execution_time** XML generation*

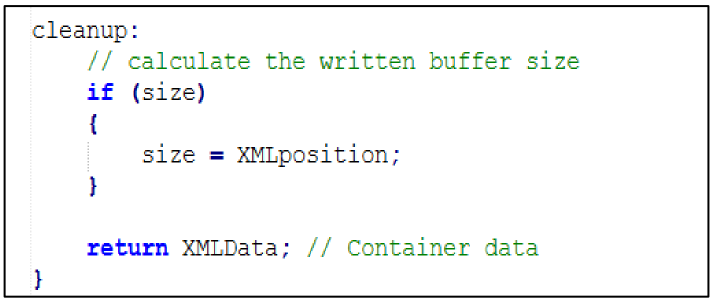
The function shown in figure 36 shows how the parameter command name is converted to XML format to display under the tag **command**.



```
// Command name
__XMLWriteAttribute(XMLData, &XMLposition, &tab, COMMAND_NAME, command_name);
```

*Figure 36 __GenerateXMLForAckHandler code snippet for **command** XML generation*

The function shown in figure 37 shows how the buffer size is calculated for the XMLData and how the XMLData is returned as a result for function call.

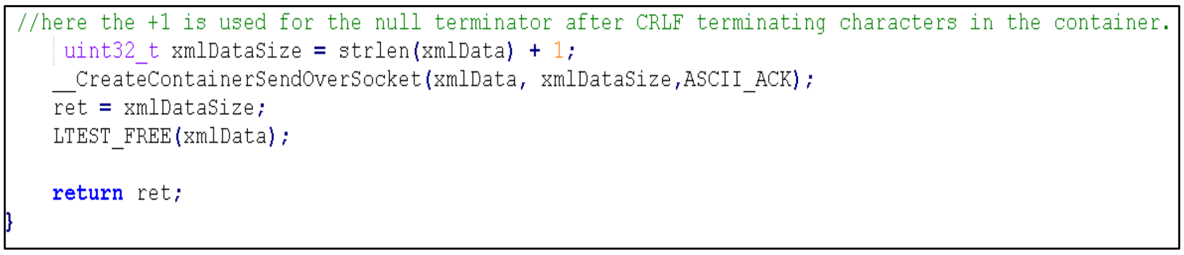


```
cleanup:
    // calculate the written buffer size
    if (size)
    {
        size = XMLposition;
    }

    return XMLData; // Container data
}
```

Figure 37 __GenerateXMLForAckHandler code snippet for returning XML data and calculating size

Step 5: XMLData to Generate Acknowledgment Response as XML Container

The XMLData created is then prepared to be send over the socket in XML Container format to the user end. This is shown in figure 38. As the first step the size of the XMLData is calculated and passed on to the function _CreateContainerSendOverSocket as an argument. Here 1 is added along with size for the null terminator after CRLF terminating characters in the container. The allocated memory is let free after the XMLData has been send over socket in container format to avoid segmentation fault due to memory overflow. This completes the process of Acknowledgement response generation in Container format.



```
//here the +1 is used for the null terminator after CRLF terminating characters in the container.
    uint32_t xmlDataSize = strlen(xmlData) + 1;
    __CreateContainerSendOverSocket(xmlData, xmlDataSize,ASCII_ACK);
    ret = xmlDataSize;
    LTEST_FREE(xmlData);

    return ret;
}
```

Figure 38 Final step: Sending the XMLData over the socket to user data port

Step 6: XML Container response generation with _CreateContainerSendOverSocket function.

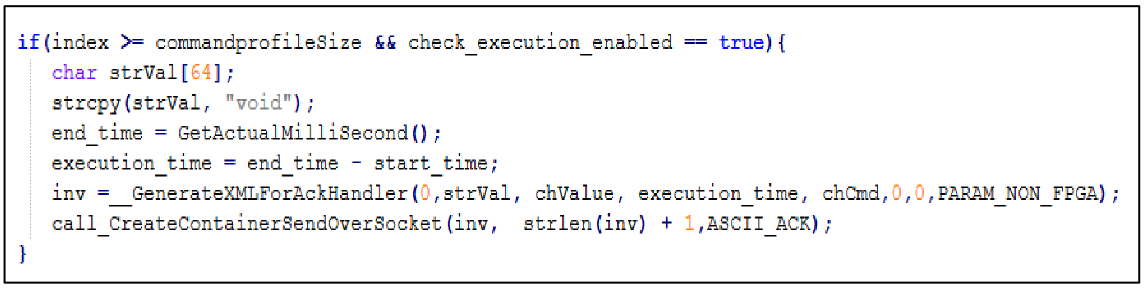
The function used here is already in use in the firmware for other containers. There are some modifications in the function which have been made to make it suitable for sending the XMLData containing the acknowledgement response in a container format.

The main change made were:

- Adding the subcontainer ID as one of the cases.
- Adding 1 to the XML Data Size to have a null terminator after CRLF terminating characters in the container, which is necessary.

6.1.5.3 Handling Invalid Commands

Each command in the system is associated with their valid command names and value types. In cases where the user tried to send random strings like hey, or some random characters and in some cases where the command name is valid, but the value is of invalid type then they are treated as invalid commands. These commands are also treated separately, and they generate an acknowledgement response. This case of handling invalid commands is shown below in figure 39:



```
if(index >= commandprofileSize && check_execution_enabled == true){
    char strVal[64];
    strcpy(strVal, "void");
    end_time = GetActualMilliSecond();
    execution_time = end_time - start_time;
    inv =__GenerateXMLForAckHandler(0,strVal, chValue, execution_time, chCmd,0,0,PARAM_NON_FPGA);
    call_CreateContainerSendOverSocket(inv,  strlen(inv) + 1,ASCII_ACK);
}
```

Figure 39 Handling acknowledgment response of invalid commands in the parsing function

From the figure 39 the incoming string is compared with ASCII Command structure and if it is not present inside the structure it is considered as an invalid command and then the current value is assigned as “void”, and the acknowledgment response is generated.

6.1.6 Command Completion

Once the acknowledgment is sent, the sensor returns to an idle state, awaiting the next command from the client. This ensures the system can handle multiple commands in sequence.

Chapter 7

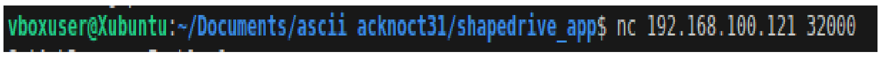
TESTING AND RESULTS

The acknowledgment response system must be tested and validated throughout the development process. This is done to ensure the correctness, efficiency and reliability of the system.

The testing of all the ASCII commands were done using different methodologies like NetCat and WeDevTool. Stress tests and normal tests were conducted on all ASCII commands and the corresponding acknowledgment responses were verified. Test results for ASCII commands of each type for valid and invalid cases are discussed below.

7.1 NetCat

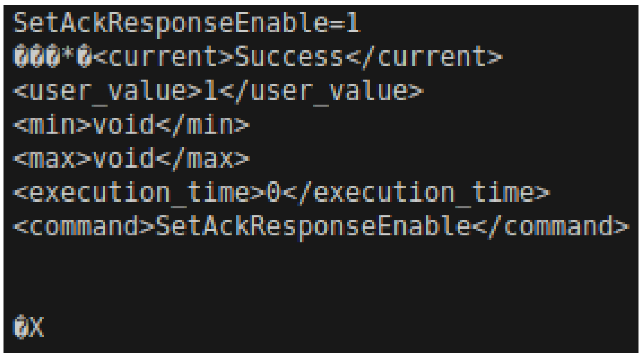
- Observations from the figure 40 is **nc 192.168.100.121 32000**. This establishes connection with the host ip 192.168.100.121 and port 32000. [9]



```
vboxuser@Xubuntu:~/Documents/ascii acknoct31/shapedrive_app$ nc 192.168.100.121 32000
```

Figure 40 NetCat Connection for testing

7.1.1 Acknowledgement response: SetAckResponseEnable (ACK type)



```
SetAckResponseEnable=1
���*�<current>Success</current>
<user_value>1</user_value>
<min>void</min>
<max>void</max>
<execution_time>0</execution_time>
<command>SetAckResponseEnable</command>


�X
```

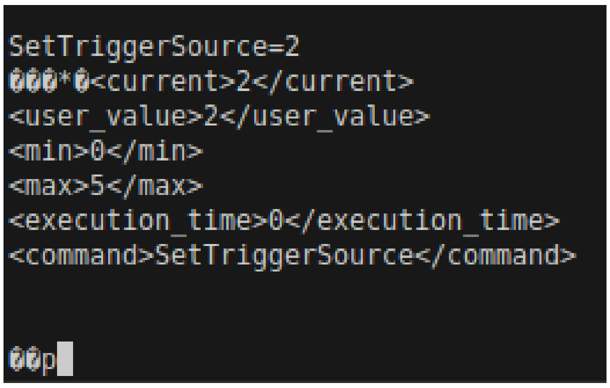
Figure 41 Observations for SetAckResponseEnable Acknowledgment response in XML-Container

- Figure 41 shown here is the observation of the implemented ASCII Acknowledgement response in XML Container format when tested using NetCat.
- This ASCII Command is used for enabling and disabling the acknowledgment response feature.
- The ASCII command "**SetAckResponseEnable"** is sent with value "**1**". This enables the acknowledgement response feature.

- The **current** tag shows "**Success**" which means the ASCII command execution is successful.
- The **user_value** tag shows "**1**" which is the input value from user.
- The **min** tag shows "**void**" which means the minimum value is not defined for this command and there is only one possible value.
- The **max** tag shows "**void**" which means the maximum value is not defined for this command and there is only one possible value.
- The **execution_time** tag shows "**0**" ms means it does not take much time for executing this command.
- The **command** tag shows the command name as "**SetAckResponseEnable**".
- The other garbage value found in the figure is the XML Container format parameter.

7.1.2 Acknowledgement response: SetTriggerSource (enum Type)

7.1.2.1 Valid Value Test

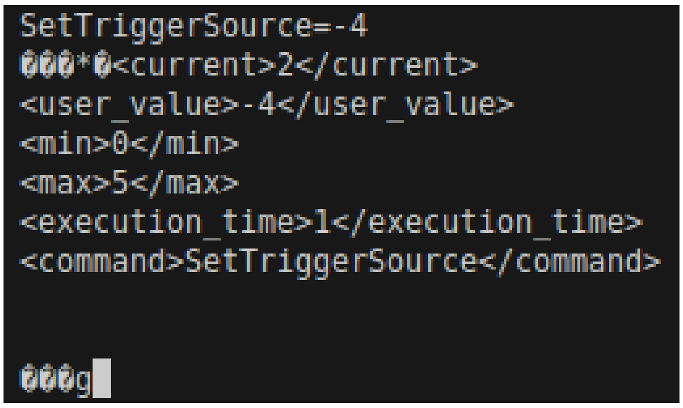


```
SetTriggerSource=2
���*�<current>2</current>
<user_value>2</user_value>
<min>0</min>
<max>5</max>
<execution_time>0</execution_time>
<command>SetTriggerSource</command>


��p
```

Figure 42 Observations for SetTriggerSource Acknowledgment response in XML-Container

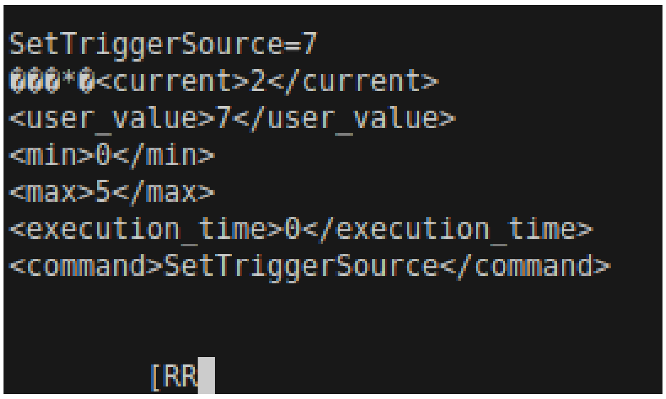
- Figure 42 shown here is the observation of the implemented ASCII Acknowledgement response in XML Container format when tested using NetCat.
- The ASCII command "**SetTriggerSource**" is sent with value "**2**". This sets the trigger source as "**2**".
- The **current** tag shows "**2**" which means the ASCII command execution is successful and applies value "**2**" to it.
- The **user_value** tag shows "**2**" which is the input value from user.
- The **min** tag shows "**0**" which means the minimum value is defined for this command is "**0**".
- The **max** tag shows "**5**" which means the maximum value defined for this command is "**5**".
- The **execution_time** tag shows "**1**" ms means it takes "**1**" millisecond for executing this command.
- The **command** tag shows the command name as "**SetTriggerSource**".
- The other garbage value found in the figure is the XML Container format.

7.1.2.2 Invalid Value test



```
SetTriggerSource=-4
���*�<current>2</current>
<user_value>-4</user_value>
<min>0</min>
<max>5</max>
<execution_time>1</execution_time>
<command>SetTriggerSource</command>


���g
```

Figure 43 Acknowledgment response for negative outbound value test for enum type command



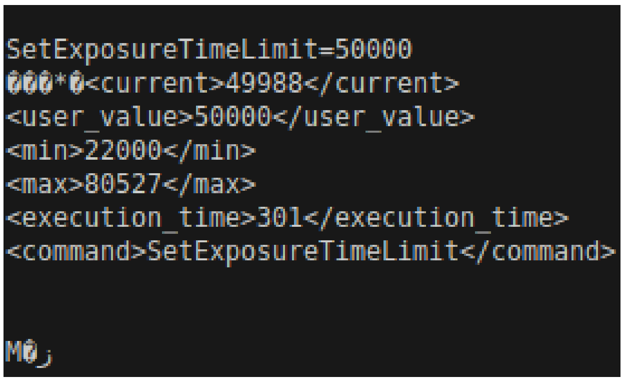
```
SetTriggerSource=7
���*�<current>2</current>
<user_value>7</user_value>
<min>0</min>
<max>5</max>
<execution_time>0</execution_time>
<command>SetTriggerSource</command>


        [RR
```

Figure 44 Acknowledgment response for positive outbound value test for enum type command

- The ASCII command “**SetTriggerSource”** is sent with negative and positive outbound values “**-4**” and “**7**” respectively as shown in figures 43 and 44.
- The **current** tag shows “**2**” in both cases, which means the **enum** type ASCII command “**SetTriggerSource”** retains its previous set value when the user tries to apply outbound values.
- The **user_value** tag shows “**-4**” and “**7**” which are the respective input values from user.
- The **min** tag shows “**0**” which means the minimum value is defined for this command is “**0**”.
- The **max** tag shows “**5**” which means the maximum value defined for this command is “**5**”.
- The **execution_time** tag shows “**0**” ms means it takes very less time for executing this command.
- The **command** tag shows the command name as “**SetTriggerSource**”.
- The other ‘garbage value’ found in the figure is the XML Container format parameters.

7.1.3 Acknowledgement response: SetExposureTimeLimit (range type)

7.1.3.1 Valid Value Test

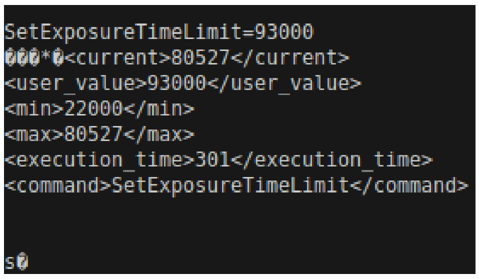


```
SetExposureTimeLimit=50000
���*�<current>49988</current>
<user_value>50000</user_value>
<min>22000</min>
<max>80527</max>
<execution_time>301</execution_time>
<command>SetExposureTimeLimit</command>


M�¡
```

Figure 45 Acknowledgment response Valid Value test for range type command

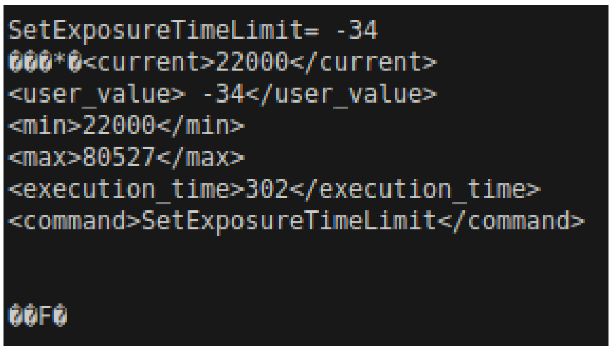
- Figure 45 shown here is the observation of the implemented ASCII Acknowledgement response in XML Container format when tested using NetCat.
- The ASCII command “**SetExposureTimeLimit**” is sent with a value of “**50000**”.
- The **current** tag shows “**49988**” which means the ASCII command execution is successful and applies value “**49988**” to it. This value is the result of internal calculations as part of the algorithm.
- The **user_value** tag shows “**50000**” which is the input value from user.
- The **min** tag shows “**22000**” which means the minimum value is defined for this command is “**22000**”.
- The **max** tag shows “**80527**” which means the maximum value defined for this command is “**80527**”.
- The **execution_time** tag shows “**301**” ms means it takes “**301**” millisecond for executing this command.
- The **command** tag shows the command name as “**SetExposureTimeLimit**”.
- The other ‘garbage value’ found in the figure is the XML Container format parameters.

7.1.3.2 Invalid Value Test



```
SetExposureTimeLimit=93000
���*�<current>80527</current>
<user_value>93000</user_value>
<min>22000</min>
<max>80527</max>
<execution_time>301</execution_time>
<command>SetExposureTimeLimit</command>


s�
```

Figure 46 Acknowledgment response for positive outbound value test for range type command



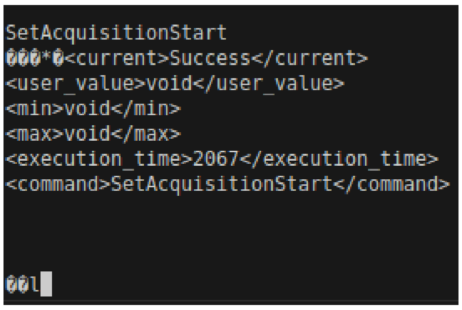
```
SetExposureTimeLimit= -34
���*�<current>22000</current>
<user_value> -34</user_value>
<min>22000</min>
<max>80527</max>
<execution_time>302</execution_time>
<command>SetExposureTimeLimit</command>


��F�
```

Figure 47 Acknowledgment response for negative outbound value test for range type command

- The ASCII command “**SetExposureTimeLimit”** is sent with negative and positive outbound values “**93000**” and “**-34**” respectively as shown in figures 46 and 47.
- The **current** tag shows “**80527**” and “**22000**” respectively, which means the **range** type ASCII command “**SetExposureTimeLimit”** applies its nearest possible valid values when the user tries to apply outbound values. These values are shown under **min** and **max** tags respectively.
- The **user_value** tag shows “**93000**” and “**-34**” which are the respective input values from user.
- The **min** tag shows “**22000**” which means the minimum value is defined for this command is “**22000**”.
- The **max** tag shows “**80527**” which means the maximum value defined for this command is “**80527**”.
- The **execution_time** tag shows “**301-302ms**” ms means it takes “**301-302**” millisecond for executing this command.
- The **command** tag shows the command name as “**SetExposureTimeLimit**”.
- The other ‘garbage value’ found in the figure is the XML Container format parameters.

7.1.4 Acknowledgement response: SetAcquisitionStart (action type)

7.1.4.1 Valid Value Test

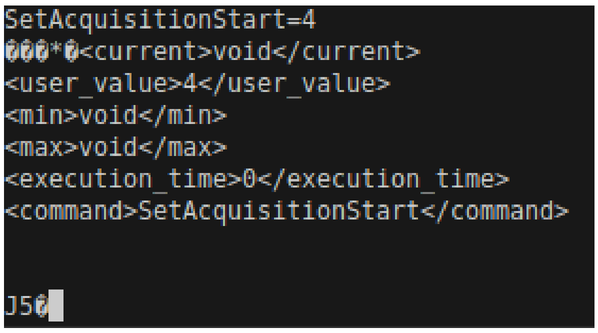


```
SetAcquisitionStart
���*�<current>Success</current>
<user_value>void</user_value>
<min>void</min>
<max>void</max>
<execution_time>2067</execution_time>
<command>SetAcquisitionStart</command>


��l
```

Figure 48 Acknowledgment response for Valid value test for action type command

- Figure 48 shown here is the observation of the implemented ASCII Acknowledgement response in XML Container format when tested using NetCat.
- The ASCII command "**SetAcquisitionStart**" is sent as it is without value.
- The **current** tag shows "**Success**" which means the action ASCII command execution is successful and the sensor starts projecting.
- The **user_value** tag shows "**void**" which means no input value from user.
- The **min** tag shows "**void**" which means there is no minimum value defined for this command.
- The **max** tag shows "**void**" which means there is no maximum value defined for this command.
- The **execution_time** tag shows "**2067**" ms means it takes "**2067**" millisecond or "**2**" seconds for executing this command.
- The **command** tag shows the command name as "**SetAcquisitionStart**".
- The other 'garbage value' found in the figure is the XML Container format parameters.

7.1.4.2 Invalid Value Test



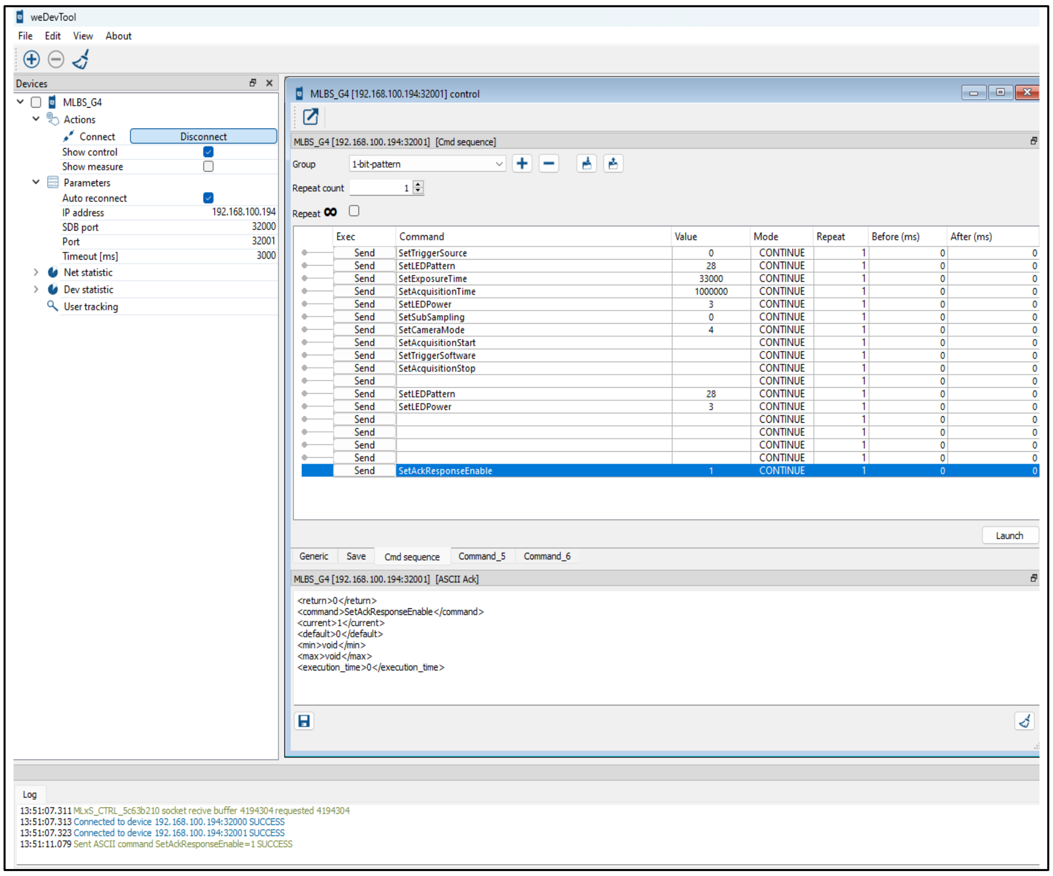
```
SetAcquisitionStart=4
���*�<current>void</current>
<user_value>4</user_value>
<min>void</min>
<max>void</max>
<execution_time>0</execution_time>
<command>SetAcquisitionStart</command>


J5�
```

Figure 49 Acknowledgment response for invalid value test for action type command

- Figure 49 shown here is the observation of the implemented ASCII Acknowledgement response in XML Container format when tested using NetCat.
- The ASCII command "**SetAcquisitionStart**" is sent as it is with value "**4**".
- The **current** tag shows "**void**" which means the action ASCII command execution is not successful as the input value is invalid.
- The **user_value** tag shows "**4**" which means the input value from user is "**4**".
- The **min** tag shows "**void**" which means there is no minimum value defined for this command.
- The **max** tag shows "**void**" which means there is no maximum value defined for this command.
- The **execution_time** tag shows "**0**" ms as the action does not take place.
- The **command** tag shows the command name as "**SetAcquisitionStart**".
- The other 'garbage value' found in the figure is the XML Container format parameters.

7.2 WeDevTool Test

- The acknowledgment response test is also done using WeDevTool after the tool was modified to have an acknowledgment response window.
- The observations are shown in the figures below



Show control	✓
Show measure	☐
Auto reconnect	✓
IP address	192.168.100.194
SDB port	32000
Port	32001
Timeout [ms]	3000

MLBS_G4 [192.168.100.194:32001] [Cmd sequence]

Group: 1-bit-pattern
Repeat count: 1
Repeat ∞ ☐

Exec	Command	Value	Mode	Repeat	Before (ms)	After (ms)
Send	SetTriggerSource	0	CONTINUE	1	0	0
Send	SetLEDPattern	28	CONTINUE	1	0	0
Send	SetExposureTime	33000	CONTINUE	1	0	0
Send	SetAcquisitionTime	1000000	CONTINUE	1	0	0
Send	SetLEDPower	3	CONTINUE	1	0	0
Send	SetSubSampling	0	CONTINUE	1	0	0
Send	SetCameraMode	4	CONTINUE	1	0	0
Send	SetAcquisitionStart		CONTINUE	1	0	0
Send	SetTriggerSoftware		CONTINUE	1	0	0
Send	SetAcquisitionStop		CONTINUE	1	0	0
Send			CONTINUE	1	0	0
Send	SetLEDPattern	28	CONTINUE	1	0	0
Send	SetLEDPower	3	CONTINUE	1	0	0
Send			CONTINUE	1	0	0
Send			CONTINUE	1	0	0
Send			CONTINUE	1	0	0
Send			CONTINUE	1	0	0
Send	SetAckResponseEnable	1	CONTINUE	1	0	0

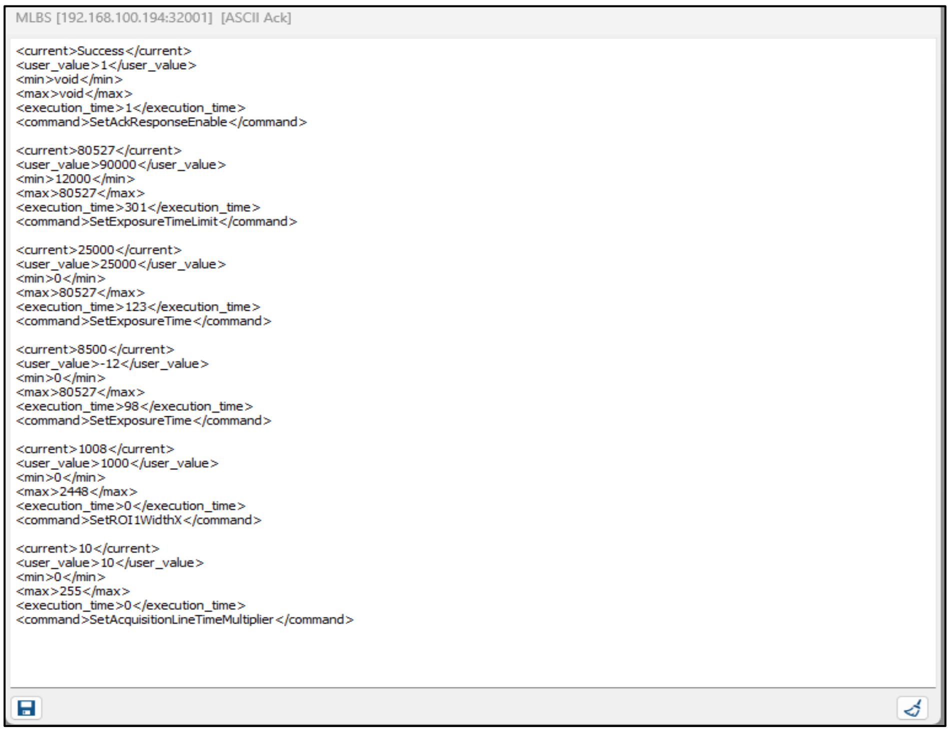
MLBS_G4 [192.168.100.194:32001] [ASCII Ack]

```
<return>0</return>
<command>SetAckResponseEnable</command>
<current>1</current>
<default>0</default>
<min>void</min>
<max>void</max>
<execution_time>0</execution_time>
```

```
13:51:07.311 MLxS_CTRL_5c63b210 socket recive buffer 4194304 requested 4194304
13:51:07.313 Connected to device 192.168.100.194:32000 SUCCESS
13:51:07.323 Connected to device 192.168.100.194:32001 SUCCESS
13:51:11.079 Sent ASCII command SetAckResponseEnable=1 SUCCESS
```

Figure 50 Testing using WeDevTool acknowledgment response window

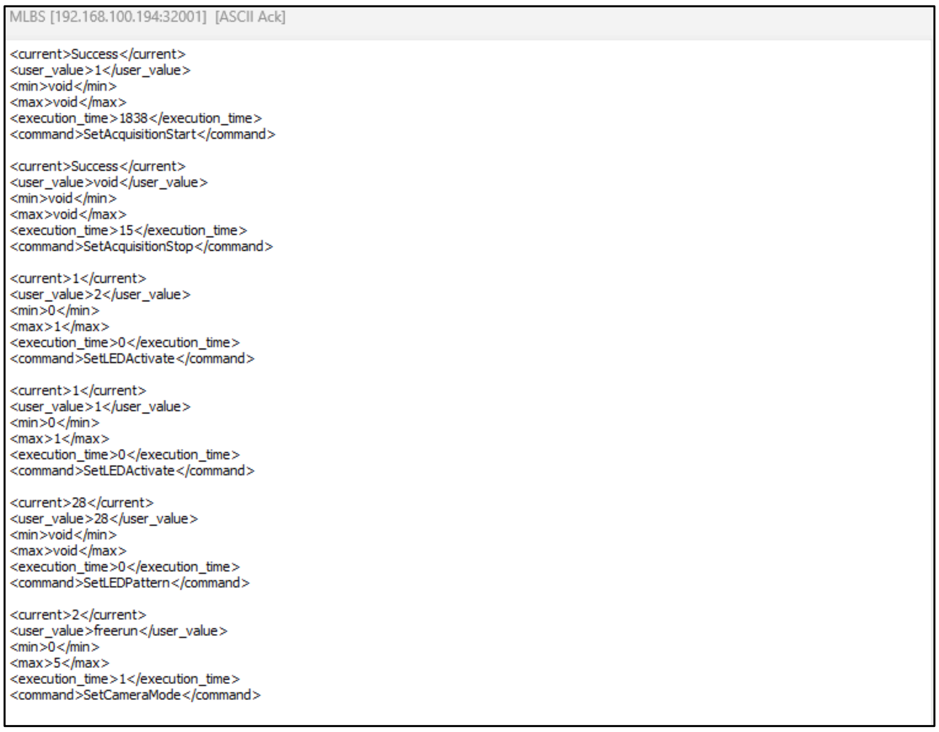
- The figure 50 shows the WeDevTool window when the **SetAckResponseEnable** ASCII command.
- The figures 51 and 52 shows the test results of generated acknowledgment responses in the WeDevTool acknowledgment window for some of the ASCII Commands.
- The figure 51 contains acknowledgment responses for the following ASCII Commands:
  - SetAckResponseEnable
  - SetExposureTimeLimit
  - SetExposureTime
  - SetROI1WidthX
  - SetAcquisitionLineTimeMultiplier



```
<current>Success</current>
<user_value>1</user_value>
<min>void</min>
<max>void</max>
<execution_time>1</execution_time>
<command>SetAckResponseEnable</command>

<current>80527</current>
<user_value>90000</user_value>
<min>12000</min>
<max>80527</max>
<execution_time>301</execution_time>
<command>SetExposureTimeLimit</command>

<current>25000</current>
<user_value>25000</user_value>
<min>0</min>
<max>80527</max>
<execution_time>123</execution_time>
<command>SetExposureTime</command>

<current>8500</current>
<user_value>-12</user_value>
<min>0</min>
<max>80527</max>
<execution_time>98</execution_time>
<command>SetExposureTime</command>

<current>1008</current>
<user_value>1000</user_value>
<min>0</min>
<max>2448</max>
<execution_time>0</execution_time>
<command>SetROI1WidthX</command>

<current>10</current>
<user_value>10</user_value>
<min>0</min>
<max>255</max>
<execution_time>0</execution_time>
<command>SetAcquisitionLineTimeMultiplier</command>
```

Figure 51 Acknowledgment response observations in the WeDevTool acknowledgment window part I

- The figure 52 contains acknowledgment responses for the following ASCII Commands:
    - SetAcquisitionStart
    - SetAcquisitionStop
    - SetLEDActivate
    - SetLEDPattern
    - SetCameraMode



```
<current>Success</current>
<user_value>1</user_value>
<min>void</min>
<max>void</max>
<execution_time>1838</execution_time>
<command>SetAcquisitionStart</command>

<current>Success</current>
<user_value>void</user_value>
<min>void</min>
<max>void</max>
<execution_time>15</execution_time>
<command>SetAcquisitionStop</command>

<current>1</current>
<user_value>2</user_value>
<min>0</min>
<max>1</max>
<execution_time>0</execution_time>
<command>SetLEDActivate</command>

<current>1</current>
<user_value>1</user_value>
<min>0</min>
<max>1</max>
<execution_time>0</execution_time>
<command>SetLEDActivate</command>

<current>28</current>
<user_value>28</user_value>
<min>void</min>
<max>void</max>
<execution_time>0</execution_time>
<command>SetLEDPattern</command>

<current>2</current>
<user_value>freerun</user_value>
<min>0</min>
<max>5</max>
<execution_time>1</execution_time>
<command>SetCameraMode</command>
```

Figure 52 Acknowledgment response observations in the WeDevTool acknowledgment window part II

Chapter 8

CONCLUSION

8.1 Summary

In this thesis work, an acknowledgment response system is implemented for ASCII based functionalities for the sensors under G4 product platforms. The acknowledgment response generated

The contributions of this thesis are as follows:

- A modular command handler mechanism for 3D and 2D sensors that between the ShapeDrive G4 sensors and user applications.
- Development of a modular firmware extension that provides real-time acknowledgment of commands, including responses in XML format for easy integration.
- Implementation of safety features, including parameter range checks, which contribute to the robustness of the ShapeDrive G4 platform.
- Testing and verification of the acknowledgment response system in various use-case scenarios to validate its effectiveness and performance.

8.2 Goals Achieved

- Implemented acknowledgment response for all existing ASCII Commands.
- Created an ASCII Command **SetAckResponseEnable** for configuration purpose. This enables the user to enable or disable the acknowledgment response feature.
- The acknowledgment Format in which the responses are generated is XML. The XML based acknowledgment response is handled and sent using a container to ensure the data integrity.
- Implemented a command handler, which act as a central registry for all ASCII commands and is seamlessly integrated to the firmware. Along with this the command parsing functionality is also modified which is now working in hand in hand with the command handler.
- Added a limit check functionality for all ASCII Commands inorder to ensure safety of the device.
- Tested and verified the acknowledgment response feature using NetCat and WeDevTool (developer application) and ensured the reliability and responsiveness of the implemented acknowledgment response system.
    - **Stress testing**
    - **Normal testing**

Chapter 9

FUTURE OUTLOOK

The acknowledgment response system implemented as a part of this thesis works efficiently and fulfils all its requirements. However, there are certain recommendations which could be added in future to enhance the functionality of the system. They include:

- Adding the **GET** ASCII Commands corresponding to the existing SET ASCII commands inside the centralized command handler and generate acknowledgment response for these. These **GET** ASCII Commands are those commands that when fired generate an acknowledgment response which could help user know the current state of any corresponding **SET** ASCII Command before applying any value to it.
- Adding a new XML tag inside the acknowledgement response which could reveal more information regarding the ASCII Command sent to the system. For example, adding error codes in case of all possible invalid cases would allow the user to easily understand what exactly the reason for error is.
- Adding automated python test programs which could enhance the possibility make the tests easier compared to manual tests which are time consuming. This test also generates test reports which are helpful in verifying the authenticity of the responses generated.

BIBLIOGRAPHY

[1] [Online]. Available: https://www.wenglor.com/en/Product-Highlights-3D-Sensors/s/Produkthighlights-3D-Sensoren.

[2] [Online]. Available: https://documentation.softwareag.com/adabas/wcp652mfr/wtc/wtc_prot.htm#wtc_prot.

[3] [Online]. Available: https://www.geeksforgeeks.org/socket-in-computer-network/?ref=header_outind.

[4] [Online]. Available: https://www.geeksforgeeks.org/client-server-model/?ref=header_outind.

[5] [Online]. Available: https://en.wikipedia.org/wiki/List_of_HTTP_status_codes.

[6] [Online]. Available: https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/.

[7] [Online]. Available: https://en.wikipedia.org/wiki/C_(programming_language).

[8] [Online]. Available: https://www.computernetworkingnotes.com/ccna-study-guide/data-encapsulation-and-de-encapsulation-explained.html?utm_source.

[9] "NetCat," [Online]. Available: https://linuxize.com/post/netcat-nc-command-with-examples/.

Chapter 11

APPENDIX

I. Type and format value function

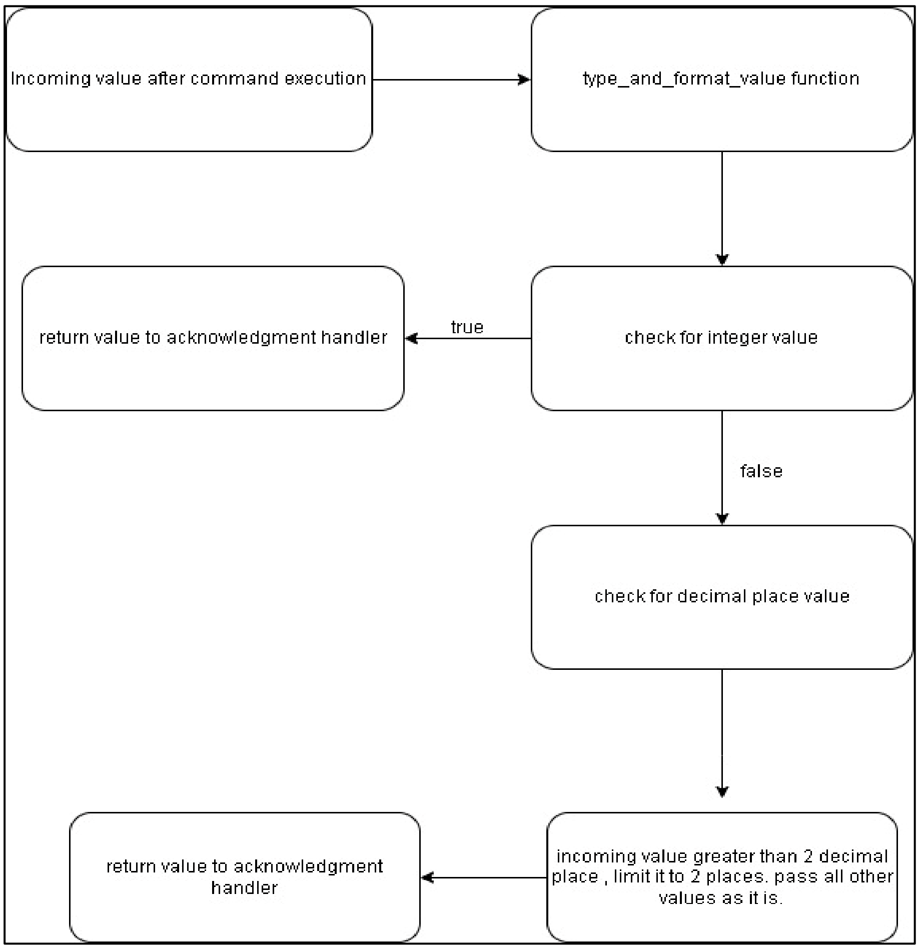

Incoming value after command execution
type_and_format_value function
return value to acknowledgment handler
true
check for integer value
false
check for decimal place value
return value to acknowledgment handler
incoming value greater than 2 decimal place , limit it to 2 places. pass all other values as it is.

Figure 53 type_and_format_value function

II. Sensor mode code for acknowledgment response

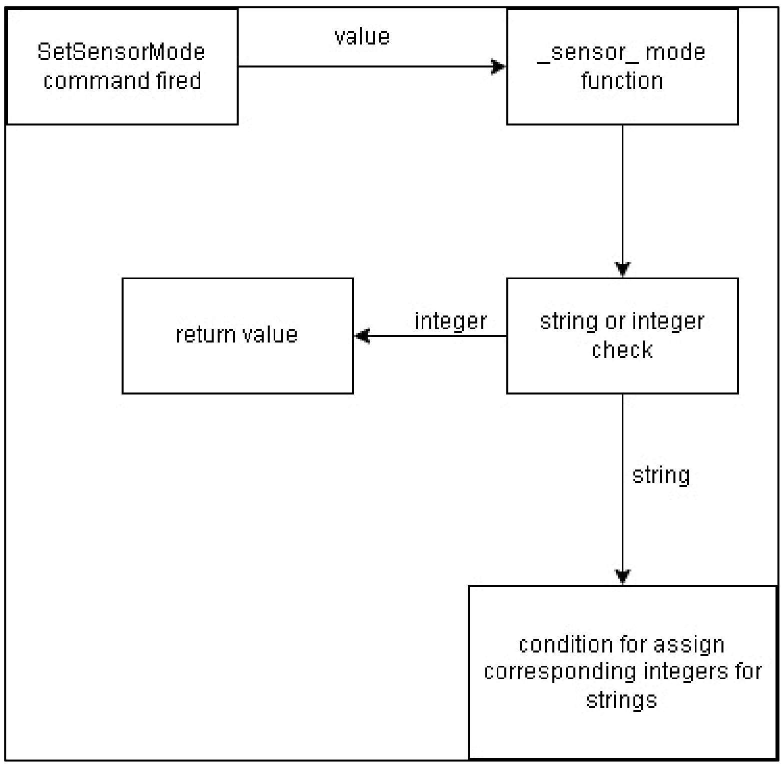

SetSensorMode command fired
value
sensor mode function
return value
integer
string or integer check
string
condition for assign corresponding integers for strings

Figure 54 _sensor_mode function

III. LED pattern function for acknowledgement response

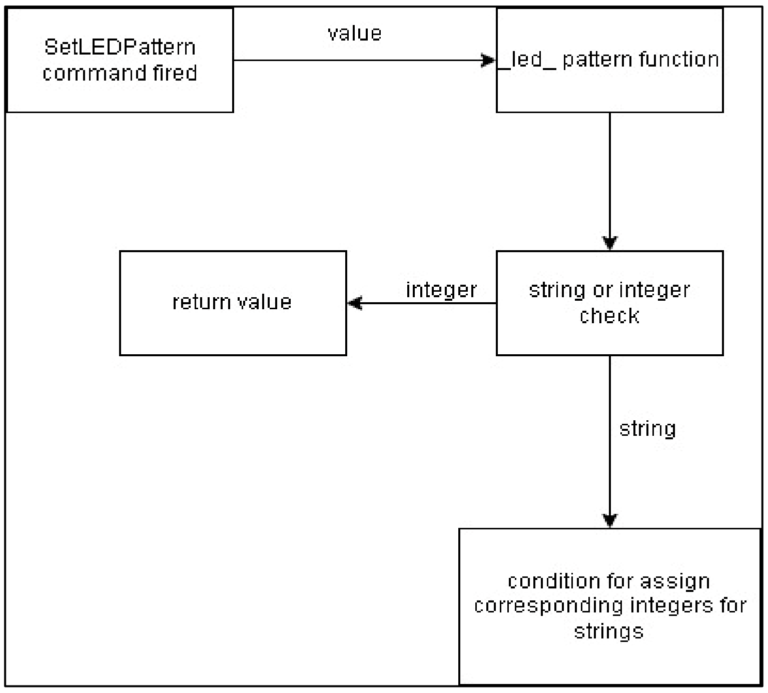

SetLEDPattern command fired
value
led pattern function
return value
integer
string or integer check
string
condition for assign corresponding integers for strings

Figure 55 _led_pattern function

IV. Get FPGA parameter function

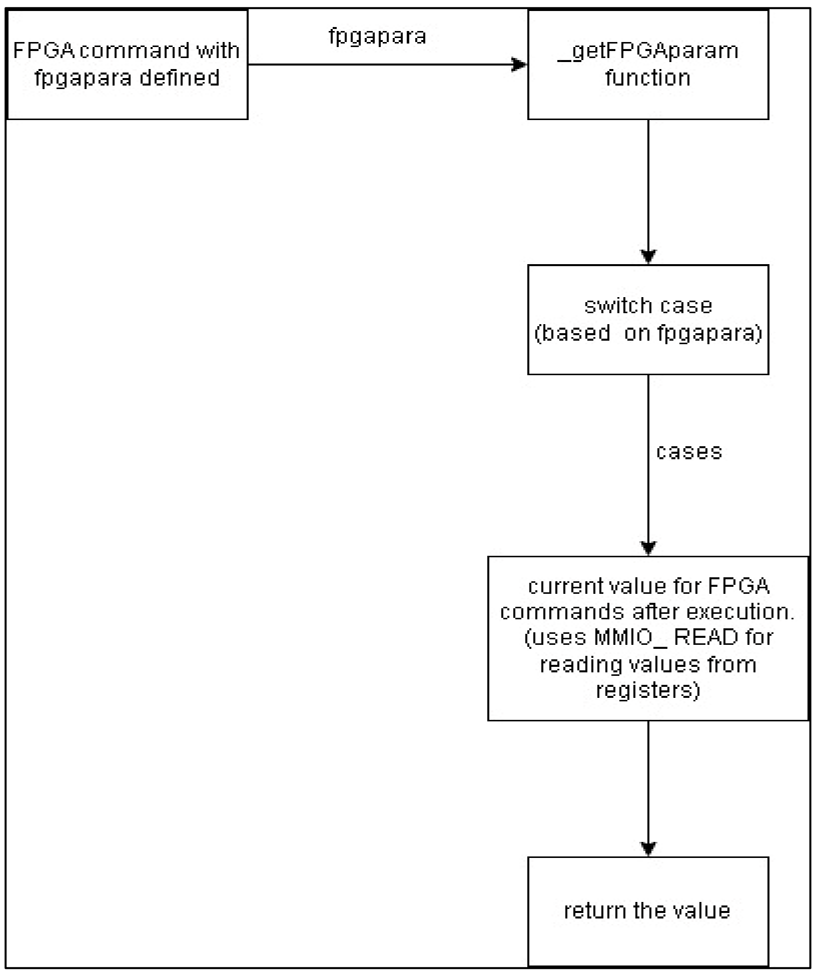

FPGA command with fpgapara defined
fpgapara
_getFPGAparam function
switch case (based on fpgapara)
cases
current value for FPGA commands after execution. (uses MMIO_ READ for reading values from registers)
return the value

Figure 56 _getFPGAparam function

V. ASCII Command List with command types (part I)

ASCII Commands	Command Type
SetAckResponseEnable	COMMAND_TYPE_ACK_CHECK
SetBlackImageErrorReset	COMMAND_TYPE_ENUM_PARAM
uspace_do_reset	COMMAND_TYPE_NO_ACK
SetSensorMode	COMMAND_TYPE_ENUM_PARAM
SetAcquisitionStart	COMMAND_TYPE_ACTION
SetAcquisitionStop	COMMAND_TYPE_ACTION
SetDLPCUsed	COMMAND_TYPE_ENUM_PARAM
SetDLPCPattern | SetLEDPattern	COMMAND_TYPE_ENUM_PARAM
SetUsrLED	COMMAND_TYPE_ENUM_PARAM
SetTriggerSoftware	COMMAND_TYPE_ACTION
GetAppVersion	COMMAND_TYPE_ACTION
SetCameraGainI	COMMAND_TYPE_RANGE_PARAM
SetCameraGain	COMMAND_TYPE_RANGE_PARAM
SetSubSampling	COMMAND_TYPE_ENUM_PARAM
SetROI1WidthX	COMMAND_TYPE_RANGE_PARAM
SetROI1OffsetX	COMMAND_TYPE_RANGE_PARAM
SetROI1StepX	COMMAND_TYPE_ENUM_PARAM
SetAcquisitionTime	COMMAND_TYPE_RANGE_PARAM
SetROI1HeightY	COMMAND_TYPE_RANGE_PARAM
SetROI1OffsetY	COMMAND_TYPE_RANGE_PARAM
SetROI1StepY	COMMAND_TYPE_ENUM_PARAM
SetAcquisitionLineTimeMultiplier"	COMMAND_TYPE_ENUM_PARAM
SetFreerunExposure	COMMAND_TYPE_RANGE_PARAM
SetGCExposure	COMMAND_TYPE_RANGE_PARAM
SetPhaseExposure	COMMAND_TYPE_RANGE_PARAM
SetDLPCExposureTime	COMMAND_TYPE_RANGE_PARAM
SetExposureTimeLimit	COMMAND_TYPE_RANGE_PARAM
SetExposureTime	COMMAND_TYPE_RANGE_PARAM
SetLEDActivate	COMMAND_TYPE_ENUM_PARAM
SetLEDPower	COMMAND_TYPE_RANGE_PARAM
SetEnableDebugData	COMMAND_TYPE_ENUM_PARAM
SetEnableMedianFilter	COMMAND_TYPE_ENUM_PARAM
SetGBThreshold	COMMAND_TYPE_RANGE_PARAM
SetContrastComparisonFilterMinPhase	COMMAND_TYPE_ENUM_PARAM
SetNRows	COMMAND_TYPE_RANGE_PARAM
SetNCols	COMMAND_TYPE_RANGE_PARAM
SetFlipPN	COMMAND_TYPE_RANGE_PARAM
SetVz	COMMAND_TYPE_RANGE_PARAM
SetGCOffset	COMMAND_TYPE_RANGE_PARAM
SetGCNo	COMMAND_TYPE_RANGE_PARAM
SetOverTrigSafeEnable	COMMAND_TYPE_ENUM_PARAM
SetLEDVsyncOffset	COMMAND_TYPE_RANGE_PARAM
SetHDREnable |SetHDRMode	COMMAND_TYPE_ENUM_PARAM
SetHDR1Exposure	COMMAND_TYPE_RANGE_PARAM
SetHDR2Exposure"	COMMAND_TYPE_RANGE_PARAM
SetCorrectionFactor	COMMAND_TYPE_RANGE_PARAM
SetTargetTempLed	COMMAND_TYPE_RANGE_PARAM
SetFan0KpLED	COMMAND_TYPE_RANGE_PARAM

Figure 57 List of ASCII Commands and types part 1

ASCII Command List with command types (part II)

SetFan0KiLED	COMMAND_TYPE_RANGE_PARAM
SetFan0KdLED	COMMAND_TYPE_RANGE_PARAM
SetFan0KpLeBody	COMMAND_TYPE_RANGE_PARAM
SetFan0KiLeBody	COMMAND_TYPE_RANGE_PARAM
SetFan0KdLeBody	COMMAND_TYPE_RANGE_PARAM
SetFan1KpMain	COMMAND_TYPE_RANGE_PARAM
SetFan1KiMain	COMMAND_TYPE_RANGE_PARAM
SetFan1KdMain	COMMAND_TYPE_RANGE_PARAM
SetHeatKpLeBody	COMMAND_TYPE_RANGE_PARAM
SetHeatKiLeBody	COMMAND_TYPE_RANGE_PARAM
SetHeatKdLeBody	COMMAND_TYPE_RANGE_PARAM
SetTargetTempLeBody	COMMAND_TYPE_RANGE_PARAM
SetTargetTempMain	COMMAND_TYPE_RANGE_PARAM
SetResetCalcPIDConstants	COMMAND_TYPE_RANGE_PARAM
SetCountersReset	COMMAND_TYPE_ENUM_PARAM
SetPLReset	COMMAND_TYPE_ENUM_PARAM
SetPSReset	COMMAND_TYPE_ENUM_PARAM
SetDLPCReset	COMMAND_TYPE_ENUM_PARAM
SetCameraReset	COMMAND_TYPE_ENUM_PARAM
SetReboot	COMMAND_TYPE_NO_ACK
get_xmldescriptorexpert /GetXmlDescriptorE	COMMAND_TYPE_NO_ACK
get_xmldescriptor / GetXmlDescriptor	COMMAND_TYPE_NO_ACK
SetHeartBeat	COMMAND_TYPE_NO_ACK
SdbDownload	COMMAND_TYPE_NO_ACK
SdbUpload	COMMAND_TYPE_NO_ACK
GetSensorStatus	COMMAND_TYPE_ACTION
GetSensorSettings	COMMAND_TYPE_ACTION
SetBoundingBoxEnabled	COMMAND_TYPE_ENUM_PARAM
SetEnableExtendedMeasuringRange	COMMAND_TYPE_ENUM_PARAM
SetDbgStandbyCmd	COMMAND_TYPE_ENUM_PARAM
SetDbgMeasRangeX	COMMAND_TYPE_RANGE_PARAM
SetDbgMeasRangeY	COMMAND_TYPE_RANGE_PARAM
SetDbgMeasRangeZ	COMMAND_TYPE_RANGE_PARAM
loadSdbPL	COMMAND_TYPE_NO_ACK
DumpBuffer	COMMAND_TYPE_NO_ACK
SetTemperatureAutoControl	COMMAND_TYPE_RANGE_PARAM
SetFanRPM	COMMAND_TYPE_RANGE_PARAM
SetUserio1Mode1	COMMAND_TYPE_RANGE_PARAM
SetUserio1Mode2	COMMAND_TYPE_RANGE_PARAM
SetUserio1OutputFunction	COMMAND_TYPE_RANGE_PARAM
SetUserio1InputFunction	COMMAND_TYPE_RANGE_PARAM
SetUserio1OutputFunctionReset	COMMAND_TYPE_RANGE_PARAM
SetUserio1InputFunctionReset	COMMAND_TYPE_RANGE_PARAM
SetUserio2Mode1	COMMAND_TYPE_RANGE_PARAM
SetUserio2Mode2	COMMAND_TYPE_RANGE_PARAM
SetUserio2OutputFunction	COMMAND_TYPE_RANGE_PARAM
SetUserio2InputFunction	COMMAND_TYPE_RANGE_PARAM
SetUserio2OutputFunctionReset	COMMAND_TYPE_RANGE_PARAM
SetUserio2InputFunctionReset	COMMAND_TYPE_RANGE_PARAM

Figure 58 List of ASCII Commands and types part 2

ASCII Command List with command types (part III)

SetUserio3Mode1	COMMAND_TYPE_RANGE_PARAM
SetUserio3Mode2	COMMAND_TYPE_RANGE_PARAM
SetUserio3OutputFunction	COMMAND_TYPE_RANGE_PARAM
SetUserio3InputFunction	COMMAND_TYPE_RANGE_PARAM
SetUserio3OutputFunctionReset	COMMAND_TYPE_RANGE_PARAM
SetUserio3InputFunctionReset	COMMAND_TYPE_RANGE_PARAM
SetUserio4Mode1	COMMAND_TYPE_RANGE_PARAM
SetUserio4Mode2	COMMAND_TYPE_RANGE_PARAM
SetUserio4OutputFunction	COMMAND_TYPE_RANGE_PARAM
SetUserio4InputFunction	COMMAND_TYPE_RANGE_PARAM
SetUserio4OutputFunctionReset	COMMAND_TYPE_RANGE_PARAM
SetUserio4InputFunctionReset	COMMAND_TYPE_RANGE_PARAM
SetCrcControlEnabled	COMMAND_TYPE_ENUM_PARAM
SetBlackImageDebugEnabled	COMMAND_TYPE_ENUM_PARAM
SetResetFactorySettings	COMMAND_TYPE_ENUM_PARAM
SetSyncOutTime	COMMAND_TYPE_RANGE_PARAM
SetSyncOutStartDelay	COMMAND_TYPE_RANGE_PARAM
SetSensorEnable	COMMAND_TYPE_ENUM_PARAM
SetTimer0Source	COMMAND_TYPE_ENUM_PARAM
SetEA1LineMode	COMMAND_TYPE_ENUM_PARAM
SetEA2LineMode	COMMAND_TYPE_ENUM_PARAM
SetEA3LineMode	COMMAND_TYPE_ENUM_PARAM
SetEA4LineMode	COMMAND_TYPE_ENUM_PARAM
SetEA1Function	COMMAND_TYPE_ENUM_PARAM
SetEA2Function	COMMAND_TYPE_ENUM_PARAM
SetEA3Function	COMMAND_TYPE_ENUM_PARAM
SetEA4Function	COMMAND_TYPE_ENUM_PARAM
SetEA1CameraEnable	COMMAND_TYPE_ENUM_PARAM
SetEA2CameraEnable	COMMAND_TYPE_ENUM_PARAM
SetEA3CameraEnable	COMMAND_TYPE_ENUM_PARAM
SetEA4CameraEnable	COMMAND_TYPE_ENUM_PARAM
SetEA1LineSource	COMMAND_TYPE_ENUM_PARAM
SetEA2LineSource	COMMAND_TYPE_ENUM_PARAM
SetEA3LineSource	COMMAND_TYPE_ENUM_PARAM
SetEA4LineSource	COMMAND_TYPE_ENUM_PARAM
SetEA1FunctionSyncOut	COMMAND_TYPE_ENUM_PARAM
SetEA2FunctionSyncOut	COMMAND_TYPE_ENUM_PARAM
SetEA3FunctionSyncOut	COMMAND_TYPE_ENUM_PARAM
SetEA4FunctionSyncOut	COMMAND_TYPE_ENUM_PARAM
SetTriggerSource	COMMAND_TYPE_ENUM_PARAM
SetEncoderSourceA	COMMAND_TYPE_ENUM_PARAM
SetEncoderSourceB	COMMAND_TYPE_ENUM_PARAM
SetEncoderResetSource	COMMAND_TYPE_ENUM_PARAM
SetEncoderResetActivation	COMMAND_TYPE_ENUM_PARAM
SetEncoderResetSoftware	COMMAND_TYPE_ENUM_PARAM
SetPSPLRegsDebugEnable	COMMAND_TYPE_ENUM_PARAM
GetLinuxPackagesVersion	COMMAND_TYPE_ACTION
SetEncoderDebounceDelay	COMMAND_TYPE_RANGE_PARAM
SetLEDAutoControl	COMMAND_TYPE_ENUM_PARAM
SetLEDROIEnable	COMMAND_TYPE_ENUM_PARAM
SetLEDMaskEnable	COMMAND_TYPE_ENUM_PARAM
SetLEDCutoffEnable	COMMAND_TYPE_ENUM_PARAM
LEDMaskUploadAndSave	COMMAND_TYPE_NO_ACK
LEDMaskUpload	COMMAND_TYPE_NO_ACK
SetLEDPowerScale	COMMAND_TYPE_ENUM_PARAM
SetImagePixBitSize	COMMAND_TYPE_ENUM_PARAM
SetNetLinkSpeed	COMMAND_TYPE_ENUM_PARAM
SetLima	COMMAND_TYPE_ENUM_PARAM
SetEthernetSend	COMMAND_TYPE_ENUM_PARAM

Figure 59 List of ASCII Commands and types part 3