POLITECNICO DI TORINO

Department of Electronics and Telecommunications

Master's Degree Programme in

Mechatronic Engineering

Master's Degree Thesis

Analysis of Simulink External Mode using XCP Protocol via Serial and CAN Bus communication channels

Supervisor:

Prof. Massimo VIOLANTE

Candidate:

Valentina TAGARELLI

Company Supervisor:

Dr. Massimiliano CURTI

ACADEMIC YEAR 2025/2026

Abstract

This thesis analyzes the use of Simulink External Mode for the execution, monitoring, and calibration of embedded system models, with a focus on the Universal Measurement and Calibration Protocol (XCP) for host-target interaction during model execution.

Adopting a protocol- and architecture-oriented approach, the work examines how External Mode structures the interaction between the host environment and the embedded application. XCP is framed within the ISO/OSI model to highlight the separation of communication layers, considering serial communication and CAN bus as transport mechanisms.

The study explores the integration of XCP within automatically generated code, distinguishing between hardware-independent and hardware-dependent layers, and identifying the interfaces that enable adaptation to different hardware targets. Simulink models are implemented for the Nucleo STM32F4 with STM32F439ZI microcontroller using the Embedded Coder Support Package and a configuration file generated through STM32CubeMX.

Validation demonstrates that, with proper hardware configuration settings, the same model can be executed, monitored, and calibrated across multiple host environments, including Simulink, using Monitor & Tune over both serial and CAN communication, and third-party tools, such as Intrepid Vehicle Spy 3, using XCP over CAN and the automatically generated A2L file.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Context

As embedded control systems become more complex, there is a greater need for reliable ways to develop, test, and validate them. As a consequence of this, the related software expands, becoming more sophisticated in terms of the many internal parameters, which must be adjusted to keep the overall systems working in the desirable manner in different operational situations. Calibration meets this need by tuning the Electronic Control Unit (ECU) parameters through organized hardware and software testing. During calibration, internal variables and sensor data are collected and analyzed, and final parameter values are adjusted and set based on this analysis [5].

Calibration is a key step that connects control algorithm design to system performance and reliability. For calibration to work properly, there must be clear access to internal variables and parameters.
In the past, communication between embedded systems and external tools was often based on custom or proprietary communication protocols, making it difficult to share tools and processes throughout all design and test phases. Thus, the increased complexity of the systems calls for the need for standard measurement and calibration protocols. Standardization gives a common interface between embedded systems and calibration tools, making it easier to use different tools and follow consistent processes. Standard protocols also help ensure that calibration can be repeated across different tools, transport technologies, and target platforms.

Concerning the automotive industry, the CAN Calibration Protocol (CCP) [4] was one of the first standards for ECU measurement and calibration over Controller Area Network (CAN) [1, 2] bus, a vehicle bus standard designed to enable communication among microcontrollers in a distributed system.
However, CCP relied on only one specific transport technology and had limited features, which showed the need for a more flexible solution [5].
This was the pivotal point for which the need for abstraction led to the creation

of the Universal Measurement and Calibration Protocol (XCP) [3], actually built on the existing CCP. According to the ASAM MCD-1 XCP standard, XCP keeps CCP's main ideas, but adds a layer of abstraction between the calibration protocol and the transport layer. This setup allows for consistent measurement and calibration over different physical interfaces without changing the application layer. As a result, XCP, which is standardized by ASAM [3], is now widely used for measurement and calibration in automotive ECUs and can be used in many types of embedded control systems.

1.2 Calibration, Model-Based Design, and External Mode

As calibration processes became more structured, they were progressively integrated within model-based development workflows. In this context, Model-Based Design allows control algorithms to be developed and validated at the model level, while calibration activities are performed by adjusting model parameters that directly correspond to the behavior of the embedded application. This approach supports a consistent development flow in which modeling, calibration, and validation are closely linked.

Within Model-Based Design, calibration requires a communication interface that connects the simulation environment to the executing embedded application. This connection enables the exchange of measurement data and calibration parameters between the host system and the target hardware, allowing the behavior of the embedded software to be observed and adjusted during execution.

In the MATLAB and Simulink environment, this interaction is supported through a host–target communication channel, used by External Mode to establish the communication between the Simulink model, which runs on the host computer, and the application executing on the embedded target [7]. Through this mechanism, signals can easily be monitored, as well as the parameters of the embedded application can be accessed and modified from the Simulink environment, supporting calibration, monitoring and verification activities maintaining consistency between the model and the deployed code.

1.3 External Mode and XCP Adoption

Prior to the availability and official support of the XCP [3] protocol as a communication option, External Mode operated using what is commonly referred to as the classic External Mode configuration. In this configuration, the host environment communicated with the embedded target directly over the selected transport interface, without employing any standardized measurement and calibration protocol.

Classic External Mode could operate over several types of connections, including serial [9] and TCP/IP [10].
For example, when using serial communication, External Mode relied on a direct link—usually through a virtual COM port over USB or, in some cases, a straightforward RS-232 connection. Data moved across this serial path, allowing engineers to monitor signals, adjust parameters, and log data as the model ran. But because the communication was built into the hardware setup, it tended to be tightly coupled to the specific physical interface being used.

For targets with networking capabilities, communication could instead happen over a TCP/IP link. In this setup, data was exchanged through a socket connection over Ethernet. It offered more flexibility in how the systems were physically connected, but the calibration and measurement logic still lived entirely at the transport layer and there was no abstraction to unify or simplify the process.

CAN-based communication was another option. In those cases, External Mode used the CAN Calibration Protocol (CCP) [4] to provide, even in this instance, live access to signals and tunable parameters on the target system. While this worked effectively for CAN networks, it remained a solution specific to that transport type and couldn't be generalized to others.

In general, classic External Mode provided the core features developers needed, such as live monitoring, tuning, and logging within the Simulink environment. On the other hand, it was built around specific hardware interfaces and tools, lacking flexibility. This limitation played a big role in the move towards integrating XCP [7], which, as stated in the previous section, brought a more standardized, transport-independent approach that could work with a range of communication technologies.

Chapter 2

State of the Art

2.1 Calibration and Measurement

In the development of embedded control systems, calibration and measurement are said to function in unison. The measurement involves monitoring signals and assessing the application's behavior during its execution. Calibration involves adjusting ECU parameter values to meet specific performance criteria under different conditions. Calibration occurs through an iterative procedure in which engineers monitor specific signals and variables. They then compare them with desired behavior and adjust specific parameters to meet the desired criteria [5].

2.1.1 Calibration and Measurement in Embedded Control Systems

Under contemporary methodologies, adjusting parameters while the system is in operation and without altering the fundamental architecture of the software code becomes necessary. As a result, control logic and calibration data are said to function separately. The control logic remains constant while calibration data varies to accommodate different hardware versions, test setups and constraints. Under the model-based development methodology, specific parameters and signals are said to function as part of the modeling stage [11].

The code generation ensures controlled access during system execution. Calibration configurations necessitate repeatable measurement sessions. Logging a defined set of signals, relating them to the current calibration state, and ensuring experiments can be reproduced is essential to shorten iteration cycles and make results comparable. For this reason, calibration is rarely an isolated action: it is part of a toolchain in which model design choices, communication mechanisms, and ECU descriptions must remain consistent.

2.1.2 Roles and toolchain responsibilities

The common pattern of the calibration process is divided among different roles that contribute different tasks [11].

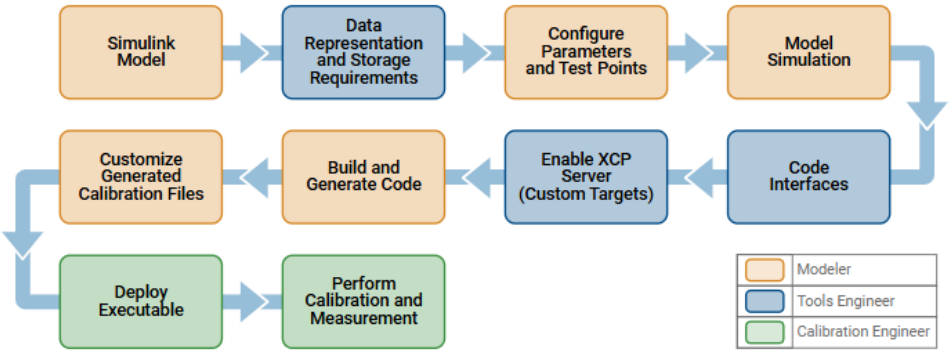


Fig. 2.1: Calibration Workflow [11]

Fig.2.1 shows a diagrammatic representation of calibration, which involves the following three roles:

- **Modeler:** Includes developing the Simulink model, simulating the model, generating the code (along with the calibration files if required);
- **Tools Engineer:** Includes determining the storage requirements and the code interfaces, as well as the connectivity aspects (such as the XCP server configuration) according to the target requirements;
- **Calibration Engineer:** Includes executing the target code and performing the calibration along with the measurement using the appropriate tools;

The above division of tasks is based on the current best practice because the runtime calibration heavily depends on the initial modeling decisions, the architecture of the code, and the final measurement process.

2.1.3 Data representation, storage, and observability

For a calibration-capable model, it is necessary to ensure that relevant information is available after code generation along with associated optimizations. In Simulink-based workflows, it is possible to store information related to models, including parameters, signals, states, data stores, and constants, with varying scopes depending on requirements related to their lifetimes and information sharing [11]. Typical information storage mechanisms include:

- **MATLAB base workspace:** Useful for temporary information storage;
- **Model workspace:** Local to a particular model;
- **Data dictionaries:** Provide global storage with change tracking;

These choices influence traceability and the stability of identifiers used in calibration setups.
The above mechanisms are related to traceability and stability of associated identifiers used in calibration.
Observability is another important aspect, for which Simulink offers test points, or signals that can be explicitly marked for observation during simulation execution. Marking a signal as a test point ensures that it is not removed or inaccessible due to associated optimizations or code generation [11].

2.1.4 Calibration Workflows Based on External Mode

In the case of MATLAB/Simulink-based development environments, the different types of calibration workflows can be distinguished according to the communication method used between the host and the target application in operation [11].
At a high level, the different types of calibration workflows can be distinguished between classical External Mode-based and XCP-based workflows.

2.1.4.1 Classical External Mode

In the classical External Mode, Simulink establishes a proprietary host-target communication channel, which enables the tuning and monitoring of parameters within the Simulink environment. The standard workflow is based on the build, deploy, connect, and start sequence, after which the tuning of the parameters is possible in real time, and the signals can be sent back for the purpose of visualization. In this mode, Simulink acts both as the execution control interface and the front end for the calibration.

2.1.4.2 XCP-Based External Mode

In XCP-based workflows, the communication between the host and the target is standardized in compliance with the ASAM MCD-1 XCP [3] specification. The generated target application includes an XCP server, while the host (client) is an XCP server. Such demarcation is helpful for the integration with different host tools.
Two major calibration and measurement scenarios are identified in this context:

- **XCP with Simulink as Host:** Simulink acts as the XCP client, also known as the master, which provides the functionality of the Monitor & Tune feature through the standardized interface of XCP. Parameters are updated and signal acquisition is performed through the services provided by XCP, all within the Simulink environment.
- **XCP with Third-Party Calibration Tools:** The generated application is provided with an ASAP2 (A2L) [13] description file that associates symbolic variables with physical memory addresses. Third-party tools, e.g., Intrepid Vehicle Spy, use this A2L file to interpret calibration parameters and measurement signals during communication with the target system using XCP over CAN, Ethernet, Serial, or other transport layers.

While these workflows differ in the specific host tool used as well as the associated connectivity infrastructure, they all follow the same architectural principle of having a target application that is runnable with tunable parameters as well as measurable signals, as well as a host tool that is able to interpret these values. The difference in the two methods is in the use of a proprietary host-target interface as opposed to the use of the standardized XCP protocol in conjunction with the A2L description mechanism.

2.1.5 Calibration data description and tool interoperability

The interoperability of the ECU software with the calibration tools depends on two different parts:

- **Protocol:** This defines the mechanism used for the exchange of information or data;
- **Description:** This defines the measurable or calibratable variables;

In the case of ASAM-based workflows, the description is usually provided by the A2L file format (ASAM MCD-2 MC) [13], while the protocol layer is provided by XCP (ASAM MCD-1 XCP) [3]. This distinction between the two parts allows different tools to communicate with the same ECU implementation if the description is the same and the protocol is the same.

2.2 The XCP Protocol

The Universal Measurement and Calibration Protocol (XCP), which is standardized as an application-layer protocol by ASAM [3], allows structured access to the internal memory of a car's ECU for measurement, calibration, stimulation, and programming in the course of runtime. XCP is an evolved version of the CAN Calibration Protocol (CCP) [4], which eliminated the dependency on the CAN bus and provided transport independence.
The fundamental idea behind XCP is that there is always a clear distinction between the protocol itself and its transport [14].

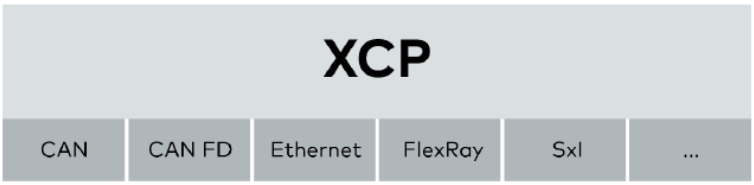


Fig. 2.2: Separation of the XCP Protocol into Protocol and Transport layers [14]

The protocol itself defines how commands are structured, what service classes are available, how memory access is handled, how data is acquired, how synchronization is handled, and how resources can be protected. The transport layer is in charge of framing, packet boundaries, addressing, and transmission of the protocol messages over the chosen transport media, which can be CAN, CAN FD, Ethernet, Serial, or FlexRay (Fig.2.2).
The logical calibration interface is always the same regardless of the physical network that is present. Calibration tools can function in the same way regardless of the communication setup that is in place, while the ECU-side implementation has the same application-layer protocol.

2.2.1 XCP in The ISO/OSI Communication Model

The difference in XCP's protocol layer and transport layer pointed out in the previous section, is best understood in relation to the ISO/OSI reference model. This model defines seven layers, each of which carries out communication-related tasks ranging from the physical medium itself up through application logic (Fig.2.3).

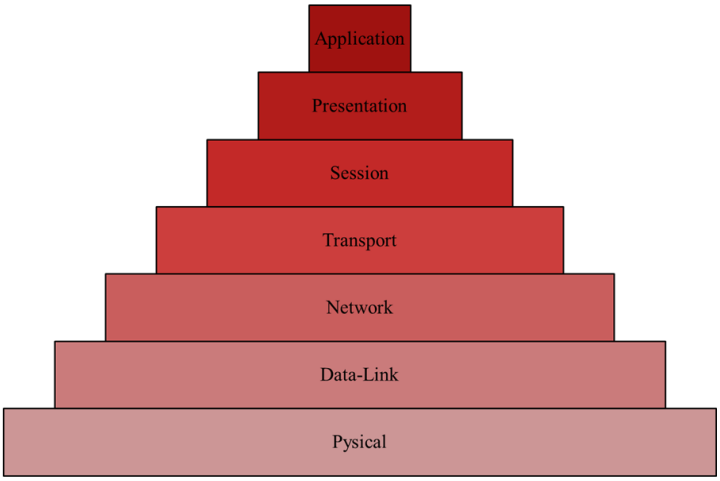


Fig. 2.3: ISO/OSI Model

The model consists of the Physical, Data Link, Network, Transport, Session, Presentation, and Application layers, progressing from the lowest to the highest level of abstraction [17]. The lower strata, the Physical and Data Link layers, are responsible for signal transmission, framing, and medium access control. The intermediate strata, the Network and Transport layers, are responsible for routing, segmentation, reliability, and end-to-end communication control. The Application layer provides standardized services, which are used directly by software applications.

XCP is specified as an application protocol that identifies the services that are provided in terms of measurement, calibration, memory access, programming, and synchronization, irrespective of the underlying network technology. On the other hand, the transport layer is responsible for the encapsulation of XCP packets in a corresponding communication frame, e.g., CAN frames, Ethernet frames, FlexRay slots, or a serial stream.
This layered structuring ensures that the logical calibration interface remains invariant with regard to the underlying physical communication medium.

2.2.2 Master–slave architecture and session control

XCP employs a single master-multi slave relationship for communication, as shown in Fig.2.10. The tool that performs the external calibration or measurement is designated as the master, and the ECU-side implementation is designated as the slave [14].

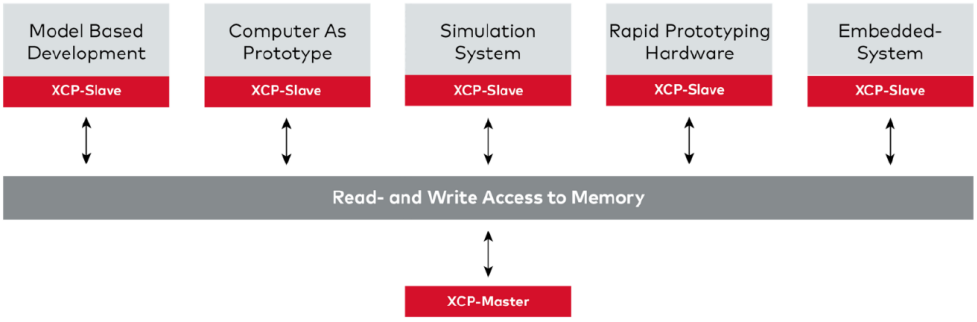


Fig. 2.4: End-to-End Universal Access Across the Development Process [14]

All communication begins at the master end. This makes XCP communication deterministic and prevents any unsolicited communication from the ECU. Each command sent by the master is contained within a Command Transfer Object (CTO). The slave processes the command and sends back one of three possible objects: a positive response (RES), a negative response (ERR), or an event message (EV). This request-response relationship makes tracking states and resources straightforward.

A standard XCP communication session begins with a CONNECT command. The slave sends the slave's version of the protocol, resources supported (DAQ, STIM, programming), address granularity, and maximum sizes for communication objects (MAX_CTO and MAX_DTO) via the CONNECT command [14]. These values define the operational limits for the communication session and ensure that both the master and slave are compatible.

2.2.3 Communication modes

The behavior observed in the given scenario, where the master transmits a command and waits for the response of the slave before proceeding to send the next command, is similar to the standard request response communication mode of XCP (Fig.2.5). This mode of communication allows only one command to be active at a given time. It waits for the response of the slave before proceeding to send the next command to the slave device [14].

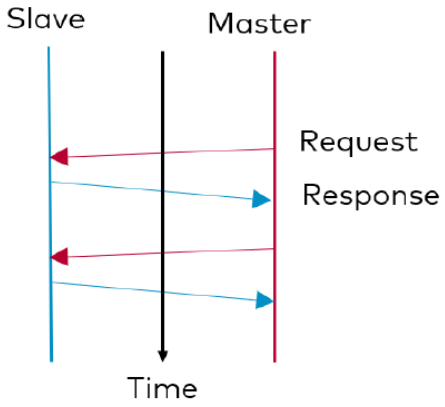


Fig. 2.5: Standard CTO exchange [14]

2.2.3.1 Master Block Transfer

In Master Block Transfer mode, the master transmits a series of command packets sequentially without any pause to wait for the response from the slave device.

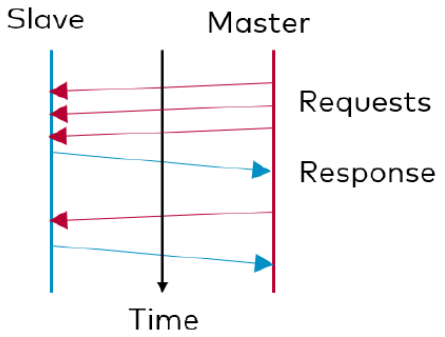


Fig. 2.6: Master Block Transfer Mode [14]

This method is often used for block download operations like programming or moving large sets of calibration data. This reduces handshake overhead significantly [14].

2.2.3.2 Slave Block Transfer

Slave Block Transfer mode is mainly used to upload blocks. Once the master sends the request, the slave sends a series of data packets in order without the need to send a new request for the next step.

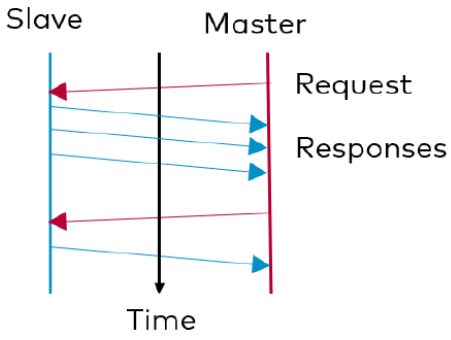


Fig. 2.7: Slave Block Transfer Mode [14]

This method improves the throughput for large memory blocks [14].

2.2.3.3 Interleaved Communication Mode

In Interleaved Communication Mode, there is the ability to have several commands in flight. This means the master does not have to wait for the reply before it can send the next command. The replies have identification information so they can be correlated to the request. This makes the bus more efficient and reduces idle time [14].

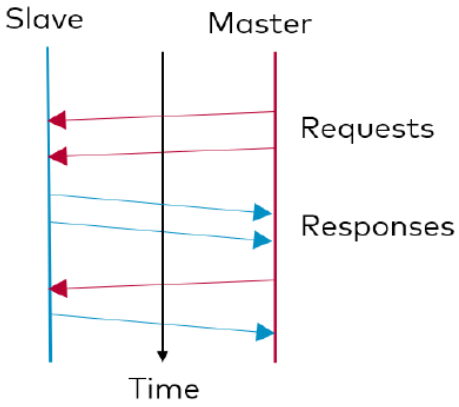


Fig. 2.8: Interleaved Communication Mode [14]

On the other hand, the complexity of the interleaved communication is higher.

2.2.4 XCP Packet structure

From a structural perspective, an XCP packet defined at the protocol level begins with a Packet Identifier (PID) and then the payload (Fig.2.9).
The PID specifies how the payload should be interpreted: whether it's a command, a response, an error, or measurements. There is no specific header and trailer defined in the protocol; these may be provided by the underlying transport layer.

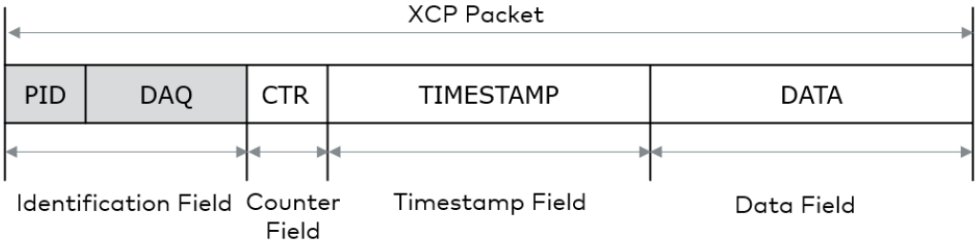


Fig. 2.9: XCP Packet [14]

Depending upon the specific transport used (CAN, Ethernet, FlexRay, UART, etc.), specific header and trailer fields may be included in the transport layer (Section 2.2.12). This may include length indicators, counters, timestamps, and checksums. These fields are not part of the XCP protocol stack but are included to ensure proper framing and synchronization of the communication over the specific physical network. [14].

2.2.5 XCP Communication Objects

As discussed in Section 2.2.2, XCP communication distinguishes between two main object types: Command Transfer Objects (CTO) and Data Transfer Objects(DTO) [14].

Command Transfer Objects are used for command-oriented communication between the master and slave. They are used for communication related to protocol services such as session management (CONNECT, DISCONNECT), memory access (UPLOAD, DOWNLOAD), and configuration of DAQ lists.
A CTO starts with the PID and then contains command-specific parameters.

Data Transfer Objects are optimized for high-speed data transfer. They are mainly used for Data Acquisition mode (Section 2.2.8) for transferring measurement data from the slave to the master and Stimulation mode (Section 2.2.9) for transferring input data from the master to the slave. In Data Acquisition (DAQ) mode, measurement variables are structured into Data Acquisition (DAQ) Lists and Object Descriptor Tables (ODTs). These Data Acquisition Lists and Object Descriptor Tables define the way individual DTOs are formed [14].

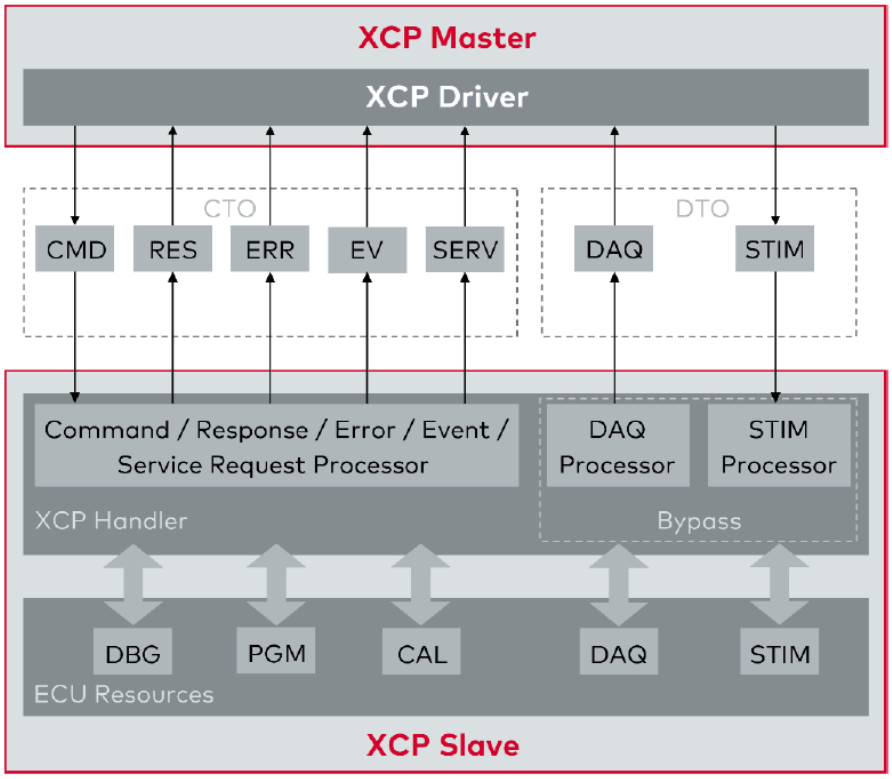


Fig. 2.10: XCP communication model with CTO and DTO [14]

The upper bounds for CTO and DTO packet sizes are determined by the parameters MAX_CTO and MAX_DTO, respectively. These are negotiated during the connection establishment phase. These are dependent on the capabilities of the transport layer and impact data segmentation and packing efficiency and measurement throughput.

2.2.6 Memory access and the MTA mechanism

The direct memory access in XCP is based on the Memory Transfer Address (MTA) [14]. Instead of embedding full memory addresses in every transfer command, XCP uses a pointer-based mechanism.

Explanation of how it works: The master initializes a memory reference with SET_MTA. Once that is set up, UPLOAD (read) and DOWNLOAD (write) commands operate relative to the memory reference that is automatically incremented after each transfer. This minimizes communication overhead and enables efficient block data movement.

The abstraction of memory transfer addresses also allows for the decoupling of symbolic variable names from actual physical memory locations. These

variable names in the A2L file are translated to actual memory locations by the calibration tool, which in turn uses memory access based on the MTA abstraction.

2.2.7 Calibration page management

XCP allows for memory segmentation via calibration pages. This allows for a distinction between reference and work parameter sets. This allows for parameter changes without immediately affecting the current active control in real-time. Switching between pages is done via commands like SET_CAL_PAGE and GET_CAL_PAGE [14]. This page-based calibration improves safety and allows atomic activation of updated parameter sets.

2.2.8 Data Acquisition (DAQ) architecture

XCP has a structured Data Acquisition (DAQ) system to facilitate the measurement of various variables in real-time [14]. Instead of using memory polling to send signals to the ECU, DAQ allows signals to be sent in an event synchronous manner. The DAQ system is structured into DAQ lists, Object Descriptor Tables (ODTs), and individual ODT entries. An ODT entry contains memory address details, length of the data to be transferred, and an extension of the memory address. ODTs are grouped into DAQ lists, with each DAQ list referencing an event channel. Event channels represent specific execution locations in the ECU, typically periodic control functions or interrupt service routines. Once an event is triggered in the ECU, it sends all ODT entries in DTO packets to the master without any request/response chatter. The DAQ system is deterministic in nature and is based on the ECU scheduling model. The DAQ system can operate at high speeds depending on the rate of occurrence of the event, maximum size of the DTO packet, transport layer bandwidth, and buffering capacity of the system.

2.2.9 Stimulation (STIM)

In addition to DAQ, XCP also supports Stimulation, or STIM. This allows the master to write variables to the ECU synchronously, utilizing DTOs to transfer the new values [14]. These are then applied at specified event triggers, so they remain time-synchronized with the control execution.
STIM is especially well-suited to rapid control prototyping and test.

2.2.10 Programming services

Apart from calibration and measurement, XCP also provides programming services (PGM) to assist with reprogramming of the flash memory [14]. These services include sector erase, block download, checksum, and activation.
Programming services are also important in high-bandwidth environments, including Ethernet.

2.2.11 Resource protection and security

Protected operations use a Seed-and-Key authentication method. In this method, the master asks for a seed and then uses a proprietary algorithm to calculate a key based on that seed and sends it for authentication [14]. Access to calibration and programming resources is allowed only after authentication. This method prevents unauthorized attempts to access calibration and programming services.

2.2.12 Transport layers in XCP

Although the actual protocol remains the same, the method of selecting the transport layer has a great influence on the performance.

2.2.12.1 XCP on CAN

In the case of XCP using CAN, the data packets are sent in CAN frames, and the data transfer occurs as specified in the ISO 11898 standard [1, 2]. A Classical CAN frame consists of the arbitration field, where the CAN identifier (11-bit standard identifier and 29-bit extended identifier) is sent, the control field, the data field, which can carry a maximum of 8 bytes, the cyclic redundancy check (CRC) field, and the acknowledge and end of frame bits. The XCP data packet (Fig.2.11) is embedded in the data field of the CAN frame, and the first byte of the data field represents the XCP Packet Identifier (PID), followed by the payload.

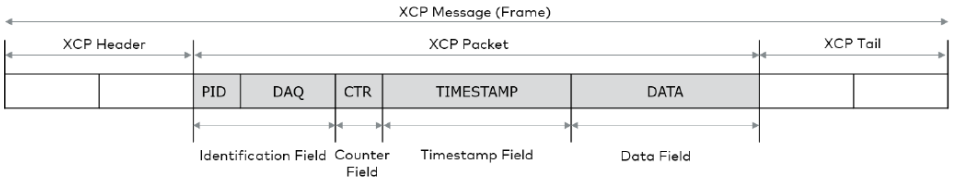


Fig. 2.11: XCP Frame [14]

The 8-byte data limit of the Classical CAN, relevant to MAX_DTO, creates a hindrance in the efficient packing of the signal data. In a DAQ system, where the data acquisition process occurs on a large scale, it may be necessary to use more than one CAN frame, thereby increasing the traffic on the CAN bus [14]. Even though the CAN bus provides deterministic access to the data, the measurement jitter depends on the traffic on the CAN bus.

2.2.12.2 XCP on Ethernet

Ethernet is a commonly used network protocol that operates at the physical and data link layers of the ISO/OSI stack (Section 2.2.1). In turn, TCP and UDP are transport-layer protocols that are based on the Internet Protocol. Payload bottlenecks are eliminated by Ethernet transport, resulting in much greater throughput [14]. In the case of the Transmission Control Protocol (TCP), reliability is ensured via acknowledgments and retransmission, while in the User Datagram Protocol (UDP), less latency is experienced due to the lack of retransmission.

In the context of an Ethernet-based implementation of the XCP protocol, the XCP packet is included in a TCP/IP packet or a UDP/IP packet (Fig.2.12). In accordance with the transport layer protocol specification, the XCP packet includes a four-byte control header that contains a length field (LEN) and a counter field (CTR) in Intel byte order. The LEN field specifies the length of the XCP packet, while the CTR field enables the detection of packet loss. At the end of the packet, a maximum of three bytes of padding may be included if required for alignment.

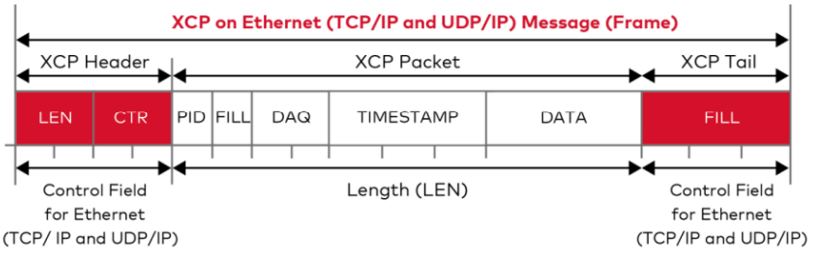


Fig. 2.12: XCP Packet with TCP/IP or UDP/IP [14]

An Ethernet frame may contain multiple XCP packets, but a single XCP packet may not exceed the maximum size of a UDP/IP packet [14].

2.2.12.3 XCP on Serial

Serial transport, typically implemented as UART-based asynchronous communication (SCI) or SPI-based serial communication (collectively referred to as SxI), enables a direct byte stream between two endpoints [14]. The continuous nature of the byte stream makes it necessary to control the length of the XCP packets.

In XCP running on serial interfaces (Fig.2.13), packets are preceded by a control header that includes a length field designated as LEN and a counter field designated as CTR. In the control header, the LEN field indicates the number of bytes in the XCP packet, while the CTR field is used to monitor packet loss. Depending on the configuration mode, whether byte, word, or double word transmission mode, additional bytes may be added to packets [14].

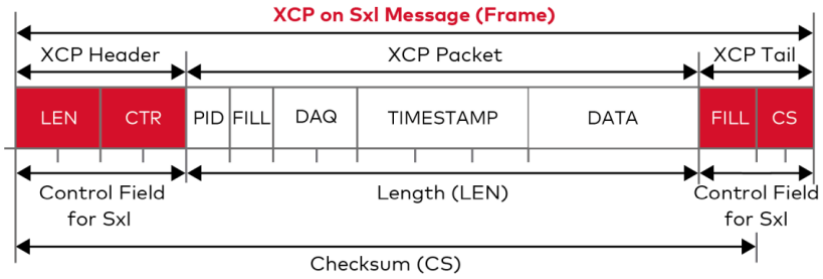


Fig. 2.13: XCP on Serial Package [14]

Even though serial transmission modes have slower bandwidth compared to Ethernet, their simplicity and one-to-one communication model are beneficial in embedded systems.

2.2.12.4 XCP on FlexRay

FlexRay is a deterministic protocol used in automotive communication, which is based on a time division multiple access scheme. It is based on communication slots within a cycle, as per the protocol specification [15]. The transport layer is based on a time-triggered scheduling mechanism; thus, the slots must be allocated in synchronization with the schedule of the FlexRay network.

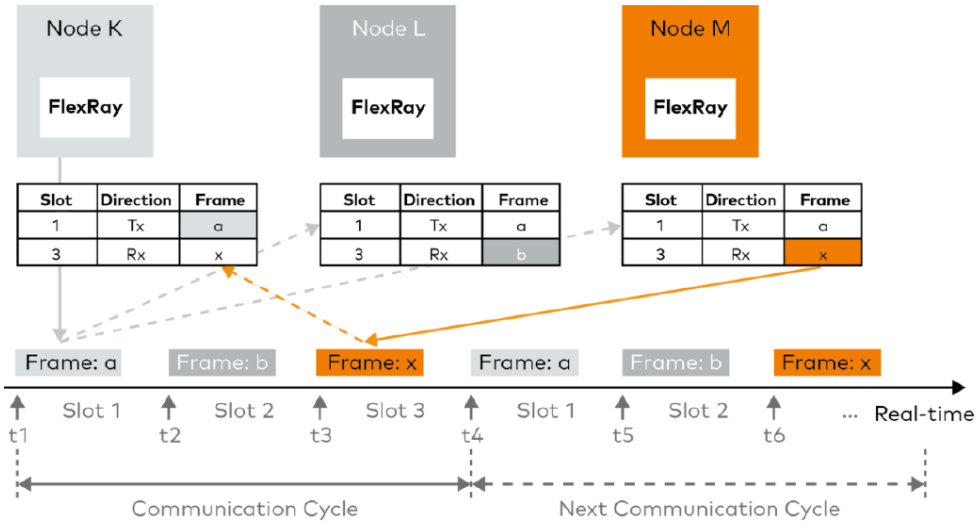


Fig. 2.14: FlexRay communication via slot [14]

This means that the XCP configuration is to be synchronized with the time artifacts of FlexRay [14]. The advantage is determinism, but the complexity is also increased by the integration of time.

2.3 A2L and the ASAM MCD-2 MC Standard

The A2L file format, as described in the ASAM MCD-2 MC standard (also called ASAP2) [13], provides a structured information model for internal variables in the context of measurements/calibrations. In addition, in an XCP-based tool chain (Section 2.2), the A2L file acts as a calibration database: it conveys to the host tool information regarding the objects that exist, where they exist, and what they mean [12].

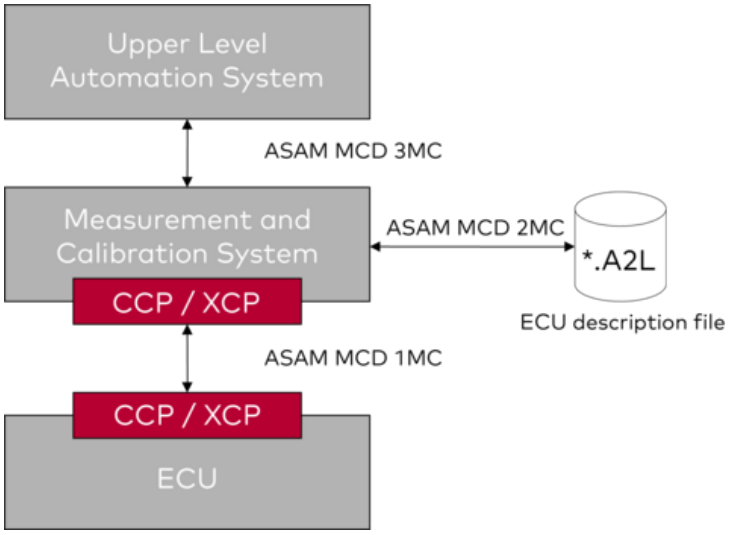


Fig. 2.15: A2L in an XCP-based calibration architecture [14]

2.3.1 A2L as a calibration database: semantics, addressing, and scaling

A2L is useful because it decouples the access from meaning. In fact, calibration tools do not need the ECU source code, but they do need to be able to find variables and parameters in memory, as well as interpret the raw data into meaningful engineering units. For these reasons, A2L often includes [12, 13]:

- **Measurement definitions:** Measurement definitions: Recordable ECU-internal variables, including addressing, data types, byte order, limits, and conversion procedures;
- **Characteristic definitions:** Tunable parameters, including scalar data, curves, and maps/lookup tables. Information given may cover addressing, limits, and conversion procedures;

- **Event definitions:** Event channels used to synchronize acquisition and stimulation with timers or operating conditions;
- **Conversion and units:** Methodologies to transform raw memory representations into physical data;

2.3.2 Structural organization of an A2L file

Apart from its semantics, the ASAM MCD-2 Measurement and Calibration standard also defines a very specific hierarchical structure for A2L files [13]. The file is made up of keywords, parameters and comments.

Keywords are the basic elements of this file. Keywords can be made up of parameters and aggregated sub-keywords, resulting in a hierarchical structure. However, some keywords are also enclosed in a delimiter defined by 'begin' and 'end' to avoid ambiguity in optional or repeated elements. The standard clearly defines what parameters are mandatory or optional and their multiplicity [13]. An A2L file consists of four structural levels (Fig.2.16):

- **Project:** This is the root that holds everything for a particular calibration project;
- **Module:** This stands for one ECU description within the above project;
- **Primary Keywords:** These identify the main objects within the ECU and the main sections within the configuration;
- **Secondary keywords:** These provide additional structural refinement;

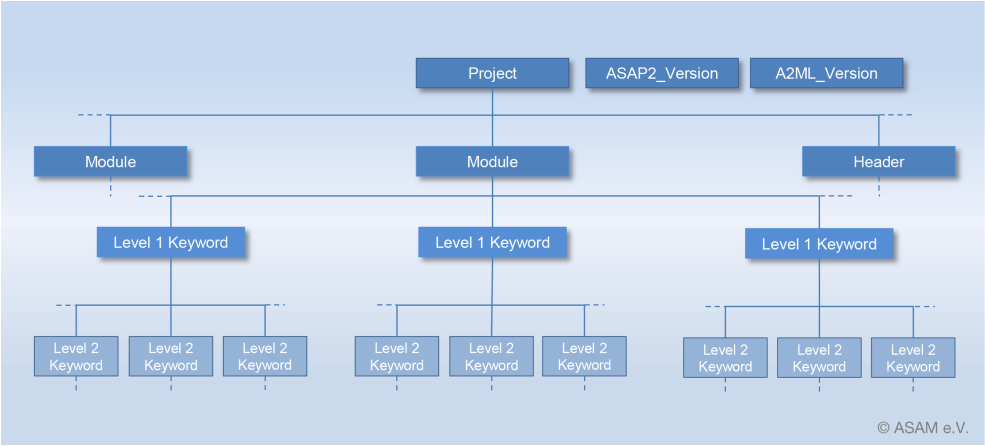


Fig. 2.16: Structure of an A2L-file [13]

The 'Project' keyword encapsulates global information (e.g., via 'Header' which contains project number, version and a description). Each 'Module' contains the complete description of one ECU, including its measurable and calibratable objects. Although multiple modules are allowed by the standard, most MC systems typically process one module per A2L file [13].

As part of the module, the basic data model is constructed based on primary keywords like MEASUREMENT, CHARACTERISTIC, COMPU_METHOD, RECORD_LAYOUT, MOD_PAR, and MOD_COMMON. The secondary keywords further specify the definition of the primary keywords. This is done with the descriptions of the axes (AXIS_DESCR), units (UNIT), memory segment definition (MEMORY_SEGMENT), and the configuration of the protocol sections (IF_DATA). The actual grammar of the communication parameters is defined in the A2ML block. This block defines the necessary structure for the information related to the protocol.

2.3.3 Measurements, characteristics, and events in practice

From the point of view of calibration, the three object classes described above 2.3.1 are analogous to the way in which an engineer might interact with an ECU in operation [12].

2.3.3.1 Measurements

Measurements are internal variables that are available for recording during runtime. The description may include additional information, such as byte ordering, scaling, and limits, as well as the possibility of interpreting the data at the bit level. Measurement objects may, in many toolchains, also be writable, allowing stimulation of the ECU during runtime, as described in the MathWorks A2L reference.

2.3.3.2 Characteristics

Characteristics are variables that are tunable. They may be single variables, strings, or structured data types, such as look-up tables, depending on the application. The A2L description contains the information that is necessary for the tool to provide an appropriate user interface, applying the appropriate conversion with consideration of the specified limits.

2.3.3.3 Events

Events provide the time basis for synchronous communication, allowing acquisition and stimulation to be coordinated with periodic tasks or operating condition changes. This is an essential part of the overall DAQ system, particularly in the case of deterministic DAQ configurations, where the ECU's execution schedule must be taken into account.

2.3.4 Protocol and Transport information in A2L-based setups

Apart from data objects, it may also hold information related to communication protocols necessary to configure communication between the host tool and the device. For instance, it may hold information such as MAX_CTO and MAX_DTO. As stated in Section 2.2.5, MAX_CTO refers to the Command Transfer Object limit, designating the maximum size of command objects transferred between master and slave devices. This is used in synchronous communication where commands are sent to a slave device to read or write parameter values. MAX_DTO refers to the Data Transfer Object limit. This specifies the maximum size of data objects used to transmit measurement data. This depends on the transport layer used. The transport layers used are usually CAN or Ethernet.

It may also hold information on byte order, granularity of addresses, DAQ (Data Acquisition) capabilities (e.g. the number of lists or event channels), and transport layers (e.g. CAN ID or IP address and port used by XCP over Ethernet). In essence, it not only specifies what can be accessed on a device but also how communication with the device is structured at both protocol and transport layers.

Chapter 3

Simulink External Mode

Simulink External Mode represents a model-based development approach that allows for real-time communication between a Simulink model running on a development system and the target application running on the target system. In contrast to the traditional approach of simulating the model, running the entire model within the development environment, the External Mode allows the target application, which has been compiled and executed on the target system, to communicate bidirectionally with the Simulink model running within the development environment.

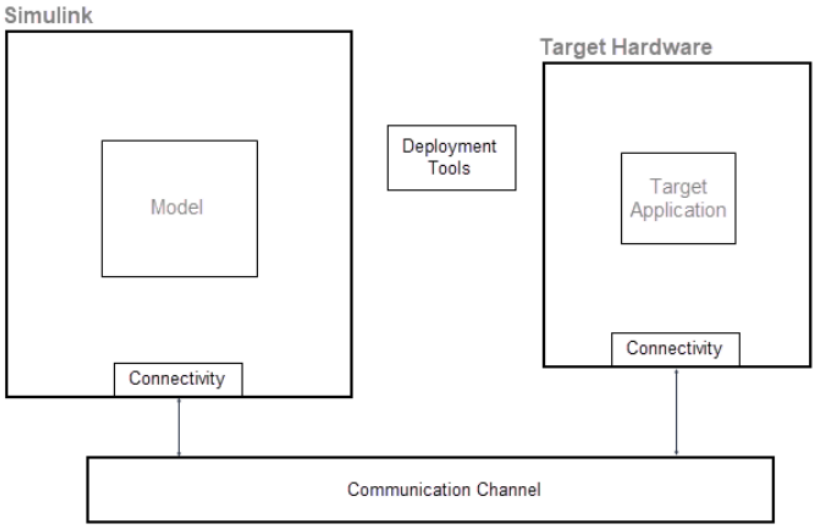


Fig. 3.1: External Mode components [16]

Using the communication channel, the engineer is able to monitor the internal signals, tune the parameters, and view the dynamic behavior of the control algorithm without interrupting the normal operation of the target application. This is especially useful when developing applications for the embedded control system, as the validation and tuning of the algorithm must be performed under realistic conditions, including the constraints of the target system.

For XCP-based Simulink External Mode, the communication between Simulink and the target application is achieved through the ASAM XCP [3] communication protocol. The code generated has an XCP server, while Simulink has

an XCP master. During operation, Simulink is able to send parameter update requests and receive measurement data from the target application.
The whole process begins with the development of a model to be used in monitoring, followed by the deployment of the generated code to the target system, the configuration of the XCP connection, and the execution of the system. After the connection has been made, it is possible to adjust parameters in real-time, as well as to capture and view the signal using Simulink scopes and the Simulation Data Inspector. The workflow of the Monitor & Tune mode allows the calibration and validation to be done in an iterative manner. It does not require the regeneration of the code in case the parameters are changed [8]. Simulation modes will be further explored in Chapter (xxx).

3.1 XCP Integration within Simulink External Mode

For the model-based development process, the Simulink External Mode simulation process utilizes the XCP protocol (Section 2.2) as the basis for its communication process. In the Simulink External Mode simulation process, the XCP protocol operates in a client-server setup, wherein the Simulink client on the development computer operates as the XCP client, also referred to as the master, and the target device operates as the XCP server, also referred to as the slave, and exchanges data with the Simulink client [16].
The XCP protocol in the context of Simulink External Mode provides a clean and organized way of parameter tuning, signal viewing, and code execution on a target device in a deterministic way.

3.1.1 Layered Architecture of External Mode Connectivity

The External Mode target connectivity software is organized in a layered architecture consisting of four main parts [16]:

- External Mode Abstraction Layer;
- XCP Server Protocol Layer,
- XCP Server Transport Layer;
- XCP Platform Abstraction Layer;

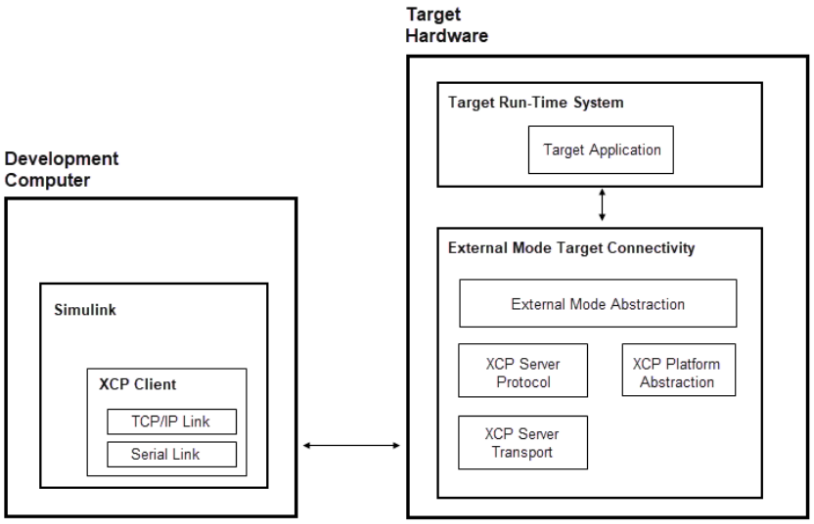


Fig. 3.2: XCP Master Slave External Mode Architecture [16]

This layered organization depicted in Fig.3.2 is based on the conceptual separation described in the ASAM XCP [3] specification and is extended to support integration with the Simulink code generation tool.

3.1.2 Parallelism between the ISO/OSI Model and XCP-Based External Mode

The multi-layer model of the XCP-based External Mode connectivity can be understood with reference to the ISO/OSI [17] reference model. It is clear that while the XCP protocol does not support the seven-layer OSI model directly, it can be mapped to the OSI model in a conceptual sense.

At the top of the OSI stack (Layer 7 – Application) is the XCP Protocol Layer. This is the layer that supports the ASAM MCD-1 XCP [3] protocol specification. It is the layer that defines the services used for communication in a calibration environment. Layers 5 and 6 (Session and Presentation) are not implemented as separate layers but can be considered logically integrated within the services offered by the XCP protocol. This is because session services are controlled through the use of the CONNECT and DISCONNECT commands. On the other hand, presentation services such as scaling, offsetting, and typing can be described using the ASAP2 (A2L) file format. This file format is used to describe the interpretation of raw memory values. The "Transport" layer, or Level 4 of the model, is equivalent to the XCP Transport Layer. This layer prepares the XCP protocol packets for transmission over a chosen communication medium by adding transport-specific framing information that

may include a packet length, a counter, a CAN ID, or a UDP or TCP header. The lower three layers of the model, "Physical," "Data Link," and "Network," are provided by the chosen communication media. These may include CAN (ISO 11898), Ethernet (TCP/IP or UDP/IP), Serial (UART/SPI), or FlexRay. These lower layers are responsible for electrical signal management, medium access control, and physical data transfer.

The previously described conceptual relationship can be briefly summarized in terms of a layered correspondence as depicted in Table 3.1.

OSI layer	**XCP element**	**Meaning in practice**
7 – Application	XCP protocol layer	ASAM-defined services and packet types: CTO for commands and DTO for synchronous DAQ/STIM data
6 – Presentation	A2L description (ASAP2)	Definition of how raw ECU values are interpreted as physical quantities (data types, scaling, offsets)
5 – Session	Connection management	Session state between master and slave (e.g., CONNECT / DISCONNECT) and related control services
4 – Transport	XCP transport layer	Adaptation of XCP packets to the selected medium (e.g., CAN mapping, TCP/UDP encapsulation, transport framing)
1–3 – Lower Layes	Network / Data-link / Phisical layer	Underlying communication technology (XCP on CAN, XCP on Ethernet, XCP on FlexRay, or XCP on Serial/UART)

Table 3.1: Conceptual Mapping between ISO/OSI Layers and XCP-Based External Mode

The layered model of XCP and its relation to Simulink's External Mode implementation is similar to the internal software architecture of XCP, in that the XCP Server Protocol Layer is equivalent to the OSI application layer, the XCP Server Transport Layer is equivalent to the OSI transport layer, and the abstraction and hardware drivers are equivalent to the OSI lower layers of a network protocol stack.

3.1.3 External Mode Abstraction Layer

The External Mode Abstraction Layer acts as the interface between the model-generated code and the XCP communication stack. In order to communicate

with Simulink, the target application has to call the chosen set of external mode APIs [16].
The general steps involved in the execution are:

1. Parsing command-line arguments for external mode;
2. Initializing the generated model code;
3. Setting up connectivity for the external mode target;
4. Waiting for host-initiated start;
5. Execution of the model step functions;
6. Triggering events in external mode;
7. Background communication tasks;
8. Handling stop or reset requests;

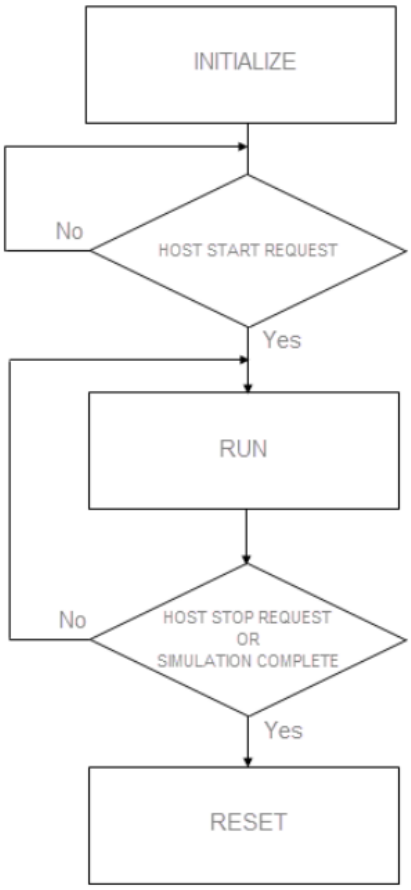


Fig. 3.3: Target–Simulink communication Flow Chart [16]

In the execution process (Fig.3.3), a minimum of two logical execution contexts must exist: a periodic task responsible for the execution of the model step function, and a background task responsible for XCP communication, packet transmission, and processing. The use of two logical execution contexts ensures the deterministic execution of the model [16].

3.1.4 XCP Server Protocol and Transport Layers

Under the External Mode architecture, the XCP Server Protocol Layer is responsible for interpreting XCP commands according to the ASAM MCD-1 XCP standard [3]. It manages memory access operations, DAQ/STIM services, session control, and command-response handling [16].

The XCP Server Transport Layer is responsible for XCP message transmission and reception via a chosen physical interface. In host-based external mode simulations, it is normally implemented by relying on the rtiostream API for handling details such as TCP/IP and serial communication. This ensures that the principle of transport independence is maintained by XCP while allowing for flexibility in implementation across different hardware platforms [16].

3.1.5 Platform Abstraction Layer and Hardware Adaptation

The XCP platform abstraction layer provides hardware-specific services that are essential for the functioning of the XCP server stack. These include:

- Raw data transmission drivers;
- Dynamic memory allocation;
- Mutual exclusion primitives;
- Sleep and timing functions;
- Address conversion logic;

Of particular interest is the address conversion service. The XCP standard defines memory locations as 32-bit addresses and an address extension field. The platform abstraction layer is expected to map these memory representations to valid memory locations in the target system environment [16].
Furthermore, structure packing, memory alignment, and concurrency mechanisms must be provided in conformance with the requirements of the target

compiler and operating environment. These factors demonstrate the point that, although XCP is a transport-independent application-layer protocol, its use in embedded systems requires careful adaptation to hardware-specific limitations.

3.1.6 Determinism and Runtime Interaction

The inclusion of the XCP implementation within the Simulink External Mode serves to illustrate the interrelation of the execution of the model and the communication services. The periodic execution of the step function of the model must be temporally deterministic, with background communication threads serving to process the XCP requests and communicate the DAQ information. This separation of the architecture serves to prevent the interference of the parameter tuning and signal monitoring with the real-time behavior of the control algorithm. In this manner, the External Mode represents a concrete implementation of the theoretical specification of the XCP [16].

Chapter 4

XCP Architecture in External Mode: Hardware-Independent Layers

Chapter 3 presented the concept of XCP-based External Mode as an architectural connectivity solution that allows the host (Simulink) to access signals and tune the model parameters while the target application is running on the embedded platform. Based on the conceptual presentation of the XCP-based External Mode solution, the current chapter aims at providing an elucidation of the implementation of the layered solution (Fig.4.1) in the generated target code, with special focus on the hardware-independent layers included in the External Mode Target Connectivity block, such as External Mode Abstraction, XCP Server Protocol, XCP Server Transport, XCP Frame Handler, and XCP Platform Abstraction.

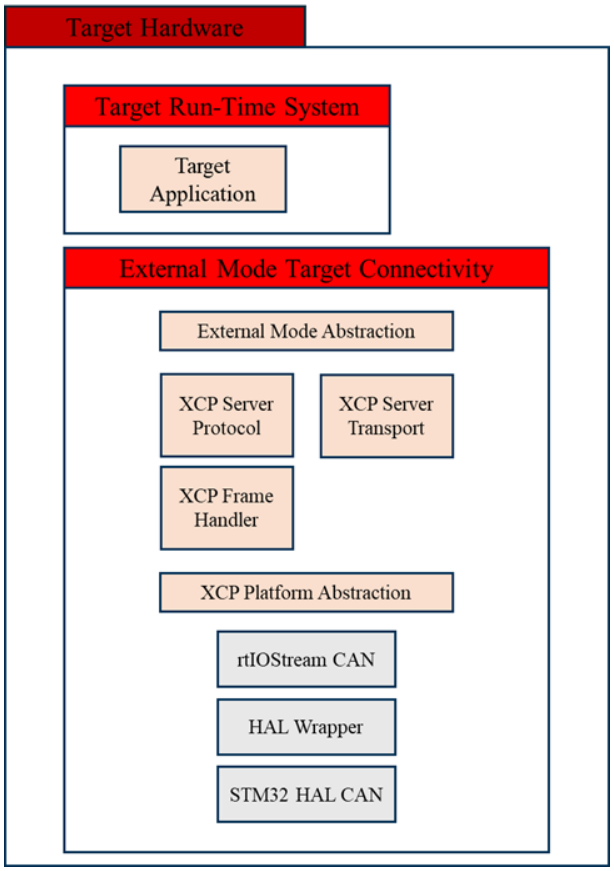


Fig. 4.1: Layered Architecture of External Mode over XCP (Hardware-Independent) – CAN

This thesis uses MATLAB R2025a and examines two physical communication media for External Mode connectivity, namely Controller Area Network (CAN) and Serial (UART). The architectural principle is the same for both communication media, working on abstract XCP packets for the upper layers, while the medium-specific behavior is contained within the frame handler for each medium, as well as hardware-dependent behavior contained within the rtIOStream backend, as discussed in Chapter 5.

4.1 Code Organization: Hardware-Independent vs Hardware-Dependent Sources

For instance, for MATLAB R2025a, the External Mode XCP stack is conceptually and physically partitioned across two different code bases. The hardware-independent components, such as the External Mode abstraction, XCP protocol/transport, generic driver model, and frame handler, are supplied within the MATLAB codebase and are reused across targets, while the hardware-dependent communication backends are supplied within the target hardware Support Package and implement the rtIOStream interface for a specific communication medium, e.g., CAN/UART on STM32 Nucleo Board.

This partitioning makes explicit the boundary for portability, as changing the physical channel or target is essentially a matter of choosing a different implementation of the `rtIOStream` backend and its accompanying frame handler, while the XCP layers are left unchanged.

4.2 Target Run-Time Context

In the case of the execution environment in External Mode, there are two different environments that need to be considered:

- one for the execution of the step function defined in the model, which runs periodically;
- another for the execution of the communication services, which runs in the background;

This dichotomy must be considered essential because, due to possible latencies that may appear during the execution of the communication protocols, the determinism of the control algorithm must not be compromised. This separation is directly reflected in the code structure, where events are sent by the

periodic part and the background part handles the communication stack. The interface layer that is automatically generated includes the periodic function calling `extmodeEvent(...)` to handle sample time events, while packet processing is handled by a background function `extmodeBackgroundRun()`, which internally handles XCP stack iteration. The interface layer is implemented in `xcp_ext_mode.c`.

4.3 External Mode Abstraction Layer

The External Mode Abstraction Layer is the highest-level component of the External Mode Target Connectivity block. It essentially serves to expose the standard external mode application programming interface used by the generated code and to map external mode operations (such as start/stop, events, and background operations) to equivalent XCP stack operations.

4.3.1 Initialization and Host Synchronization

During this process, three main functions are performed by the abstraction layer. First, External Mode arguments are parsed. For example, `extmodeParseArgs` parses the External Mode arguments, such as the simulation time and host-start, and stores them as global variables. This provides a uniform method for setting External Mode parameters without modifying application code.

Second, compatibility data is prepared, which is later utilized by the host-side code. The model checksum fields are retrieved from the `RTWExtModeInfo` data structure and stored as global variables, such as `xcpModelChecksum0..3`. This is a critical step for ensuring that the host connects to the correct executable.

Third, the XCP stack is initialized by invoking the platform-common initialization function, `xcpExtModeInit`, which then configures the transport and protocol stacks. This initialization pipeline is defined within the `xcp_ext_common.c` source file.

Once this process is complete, communication with the host is synchronized by the function `extmodeWaitForHostRequest`, which suspends (or polls with a specified timeout) until a start request is received from the host, while continuing to process background communication requests during this period.

4.3.2 Event Notification and Background Processing

The function `extmodeEvent(eventId, simulationTime)` is the crucial interface between the deterministic model execution and communication, as it updates the internal time tracking and notifies the XCP stack about the occurrence of events, thereby providing the support for synchronous data transfer (DAQ) according to the sample times of the model.

The process is advanced by `extmodeBackgroundRun()`, which runs a background engine implemented in `xcp_ext_common.c`. The routine is a structured cycle consisting of:

1. Attempt to receive a new packet from the transport layer;
2. Process it in the protocol layer;
3. Transmit pending packets from the transport queues;

This cycle is realized by `xcpExtModeRunBackground`, which in turn calls `xcpTransportRx()`, `xcpRun()`, and `xcpTransportTx()` in sequence. The observed behavior corresponds to the architecture presented in Fig. 4.1, where the orchestration of the stack is done by the External Mode Abstraction, and the execution of the packet management and transport services is done by the subordinate layers.

4.4 XCP Server Transport Layer

The XCP Server Transport Layer is responsible for moving XCP packets between the protocol layer and the underlying framing/driver backend. In practice, its main job is not to interpret XCP semantics, but to manage buffers, queueing, and scheduling of TX/RX operations.
This layer is implemented in `xcp_transport.c`.

4.4.1 Decoupling Through Queues and FIFO Infrastructure

Another design choice involves the use of FIFO queues to remove the coupling between packet generation and transmission. The transport layer has its own transmit (TX) and receive (RX) FIFO structures. The FIFO design is very simple and efficient; it supports enqueue, dequeue, enqueue at head, and splice operations as provided in `xcp_fifo.c`.

This separation is crucial for the External Mode because of the following reasons:

- The protocol layer is capable of generating responses immediately and queuing them up;
- The actual transmission over the bus takes place later, controlled by a background task;
- The management of the priority can be done without interfering with the execution of the protocol;

4.4.2 Packet Lifecycle in the Transport Layer

The transport layer provides a complete life cycle for packets, which includes the following steps:

- **Packet Enqueue (TX):** The `xcpTransportTxPacketSet` function creates a message frame and places the packet in the appropriate TX queue;
- **Packet Transmit:** The `xcpTransportTx` function selects the packets according to their priority level and passes the send request to the frame handler;
- **Packet receive:** The `xcpTransportRx` function requests a new message from the frame handler and places the packet in the RX FIFO queue;
- **Packet extraction (RX):** The `xcpTransportRxPacketGet` function dequeues a received message from the RX FIFO queue and retrieves the XCP packet size and offset for the protocol layer;

The fundamental architectural principle asserts that the transport layer retains ownership of message buffers while they are queued and releases or re-queues the message based on the success of the transmission. This ensures that all three layers of the protocol, transport, and frame have consistent memory ownership.

4.5 XCP Frame Handler (CAN and Serial framing)

In the structure presented in Fig.4.1, the XCP Frame Handler is positioned at the lower layer than the protocol and transport layers. It is responsible

for converting the abstract packet to a transport-compliant representation and vice versa.
In the context of this thesis, the following two frame handlers, which correspond to the transport mediums of interest, will be considered:

- **CAN:** located in `xcp_frame_can.c`;
- **Serial:** located in `xcp_frame_serial.c`;

Both transport transport layers operate on XCP packets (CTO/DTO), but the underlying medium he transport medium has its own restrictions and framing standards. The role of the frame handler is to bridge this gap by:

- Creating a message suitable for the transport medium;
- Verifying the received message and identifying packet boundaries;

4.5.1 CAN Virtual Header Concept

In CAN-based transport, the frame handler uses a virtual header to hold the length of the packets. This is done by placing the length of the packets in the first byte of the internal message frame and then shifting the pointer so that the length field is not transmitted on the CAN bus. This follows the need to have a uniform internal message representation within the XCP stack and ensures that only bus-compliant bytes are transmitted on the CAN bus.

4.5.2 Serial Length-based Framing, Optional Counter, and Checksum

For Serial transport, the frame handler deals with a different problem. In a byte stream-based transport link, frame boundaries are not automatically defined. The Serial frame handler specifies a frame format as follows:

- **Header** containing at least the XCP payload length;
- **Counter** (Optional) for sequencing (`XCP_SERIAL_USES_CTR`);
- **Tail Checksum** (Optional)(`XCP_SERIAL_USES_CS`);

The process of staged reception in `xcpFrameMsgRecv` starts by reading a header of fixed size to identify the length of the received packet. It then allocates a complete message buffer, which is sized to header + payload + tail. Once frame reception starts, all subsequent reads are forced to be blocking (via

`xcpDrvIoctl(XCP_DRV_FORCE_BLOCKING)`) until payload and tail are completely received. Once frame reception starts, the standard blocking/non-blocking configuration is reinstated.
During the process of receiving the frame, `xcpFrameExtractPacket` validates the size of the received message. It also performs additional validation as follows:

- **Sequence integrity** checking for lost or out-of-sequence packets via counter;
- **Data integrity** via checksum computed over header and payload;

This architectural approach is transport robust and maintains identical upper-layer protocol/transport interaction for CAN and Serial.

4.6 XCP Platform Abstraction (Hardware-Independent Boundary)

As can be seen in Figure 4.1, the XCP Platform Abstraction layer represents the last block before the hardware-dependent back-end. From the given codebase, this boundary is represented by a driver implemented on top of the `rtIOStream` interface. The implementation of this abstraction can be found in `xcp_drv_rtiostream.c`.

4.6.1 Driver Responsibilities and Error Semantics

The driver layer defines how a byte is sent and received but does so indirectly through `rtIOStream` abstraction without direct access to a peripheral device. The functions `xcpDrvOpen`, `xcpDrvSend`, `xcpDrvRecv`, and `xcpDrvRecvUnknownSize` encapsulate:

- The initialization and management of a single active connection (single-master session behavior);
- Blocking versus non-blocking behavior (with a forcing mechanism provided by `xcpDrvIoctl`);
- Bounded timeout and retry delay for send and receive operations;

From a design perspective, this means that the higher layers (frame, transport, protocol, and external mode abstraction) can assume uniform error codes such as `XCP_SUCCESS`, `XCP_EMPTY`, `XCP_BUSY`, without making any assumption about a particular hardware implementation.

4.6.2 Transport Configuration Parameters

Finally, the initialization parameters for the transport layer are collected and sent using a special parameter interface. For the CAN protocol, this process is implemented in the file `xcp_ext_param_default_can.c`. A similar process exists for the Serial transport protocol, to ensure that the choice of transport protocol remains a configuration issue rather than an alteration to the protocol layer.

4.7 Summary of Hardware-Independent Contribution

The hardware-independent layers implement the full communication logic required for the External Mode, which includes:

- The External Mode Abstraction is in charge of lifecycle management and time-related aspects;
- the Protocol Layer provides ASAM XCP services;
- The Transport Layer is in charge of buffering and scheduling;
- The Frame Handler is in charge of transport-specific formatting (CAN vs. Serial);
- The Platform Abstraction provides the uniform driver interface, as well as uniform timing and error models;

In the following chapter, the hardware-dependent backends for the implementation of the `rtIOStream` over an STM32 Nucleo Board using CAN and Serial (UART/LPUART) communication channels, will be presented, finalizing the architecture description.

Chapter 5

XCP Architecture in External Mode: Hardware-Dependent Layers

Chapter 4 described the hardware-independent part of the External Mode Target Connectivity stack, up to the XCP Platform Abstraction boundary. This chapter focuses on the hardware-dependent strata (Fig.5.1) in the architecture, as shown in the figure below, namely the `rtIOStream` backends, their interaction with the low-level drivers of the STM32 Nucleo Board, and, as already mentioned, the two communication media used in this thesis, CAN and UART. All the components that will be investigated in this chapter represent the actual interaction with the peripherals, while the independence of the transport medium is still maintained in the upper XCP layers.

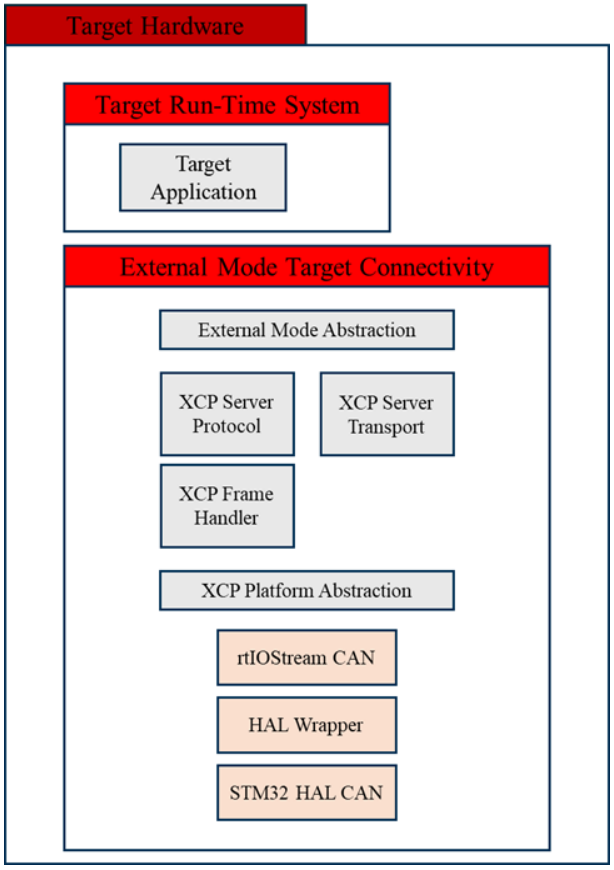


Fig. 5.1: Layered Architecture of External Mode over XCP (Hardware-Dependent) – CAN

5.1 Role of rtIOStream as Hardware Backend

The XCP driver, as presented in Chapter 4, exclusively uses the `rtIOStream` API to create a channel and exchange bytes. The architectural decision to use this API to create a channel and exchange bytes isolates the XCP stack from implementation details of the physical transport. As long as the `rtIOStreamOpen/Send/Recv/Close` interface is implemented, other components are not impacted.

From an architectural perspective, `rtIOStream` is the first "genuine" hardware-dependent component. It maps abstract send and/or receive requests to peripheral operations and configures bus-dependent parameters (e.g., CAN IDs and filters, or DMA settings and buffers for UART).

5.2 CAN Backend: rtIOStream CAN and STM32 HAL Integration

For this particular implementation set, the CAN-specific backend implementation is contained within the file `rtiostream_can.c`. With respect to the initial stages of External Mode, the function `xcpDrvOpen` calls `rtIOStreamOpen` to open the communication channel. The CAN backend carries out the relevant hardware initialization code here. The main architectural contribution is that:

- The initialization of peripherals is handled entirely within this component;
- The handle/state is stored internally within the `rtIOStream` layer;
- The XCP driver operates at a level above this component, interacting solely with an integer channel identifier, `drvID`;

This is the concrete realization of the hardware abstraction boundary shown in the diagram (Fig.5.1).

5.2.1 Transmit Path: From XCP Packet to CAN Peripheral

The path that the transmission takes through the system layers is as follows:

1. The protocol/transport layers enqueue the message;

2. The frame handler provides the payload to be sent (excluding the virtual header);

3. The XCP driver calls `rtIOStreamSend`;

4. The CAN backend sends a CAN frame using the HAL wrapper;

5. STM32 HAL interacts with the peripheral;

The frame handler specifically ensures that the internal CAN header is not sent by advancing the buffer pointer over the virtual header before calling `xcpDrvSend`. This point is interesting because it shows that the upper layers have an internal representation of the message that is uniform, yet the hardware backend sends only what complies with the bus protocol.

Looking at the driver level, the function `xcpDrvSend` has bounded retries and timeouts. This is not specific to CAN; this is an architectural constraint to ensure that the system does not hang indefinitely if the backend is temporarily unable to send the data. This implies that the hardware-dependent layer must be able to send the data immediately or have sent some of the data to allow the retry mechanism to work.

5.2.2 Receive Path: From CAN Peripheral to XCP Packet

In terms of reception, this is symmetric in principle; however, there is a further complication. With CAN, there is no management of the receive size as a stream with a known length. The frame handler deals with this problem by using xcpDrvRecvUnknownSize, which reads to a maximum and returns the number received. `xcpDrvRecvUnknownSize`, which reads to a maximum and returns the number received. xcpDrvRecvUnknownSize, which reads to a maximum and returns the number received. Upon receipt of the bytes, the frame handler allocates a message buffer, writes the virtual length header, copies the payload, and finally passes the responsibility to the transport layer. This process serves as a good example of the design motivation for compartmentalizing the process as follows:

- The `rtIOStream CAN` backend has the responsibility of receiving raw CAN payload bytes;

- The frame handler has the responsibility of structuring the received bytes into a consistent internal message format;

- The transport layer has the responsibility for queuing and scheduling the packet for protocol processing;

5.2.3 HAL Wrapper and STM32 HAL CAN

The next layer below the `rtIOStream CAN` interface is the HAL wrapper layer. This layer is conventionally used to insulate the MathWorks code from vendor-dependent HAL code. From an architectural point of view, this function is explicitly indicated in the layered model as a separate block.

The STM32 HAL CAN driver performs the actual register-level access along with interrupt handling and FIFO management at the lowest level. From an architectural point of view, the key factor here is not what exactly the STM32 HAL calls are but the fact that they are confined to these layers. From the code point of view, the STM32 HAL module corresponds to `stm_can_hal.c`, which source code is provided by the STM Support package for Embedded Coder.

5.3 Serial Backend: rtIOStream UART and Buffer/DMA Integration

For the case of Serial, the hardware-dependent backend is implemented in `rtiostream_serial.c`. The XCP driver interface remains unchanged, but now `rtIOStreamOpen/Send/Recv/Close` map onto the STM32 UART services.

5.3.1 Open and configuration: UART vs LPUART selection

During the initialization process, the backend makes a choice between the standard UART and LPUART depending on the base address of the configured peripheral. This process is accomplished through the `MW_LPUART_RTIOSTREAM_USED` compile-time check. The `rtIOStreamOpen` function maps the configured RX buffer and the optionally provided TX buffer, which in turn calls either `MW_LPUART_Initialize` or `MW_UART_Initialize`.

The backend uses a structured `uartObj` configuration that includes pointers for peripherals, DMA stream selection, and runtime buffer pointers. RX buffering

is always enabled (default `MW_SERIAL_RX_BUFFER_SIZE`), while TX buffering is conditionally enabled depending on the TX DMA configuration.

5.3.2 Transmit: DMA vs polling

The `rtIOStreamSend` function operates in in two different modes depending upon whether `MW_CONNECTIVITY_TX_DMA` is defined or not during compile time. If so, then DMA-based UART services (e.g. `MW_UART_TransmitUsingDMA` or the corresponding LPUART variant) are used to send the data. Otherwise, a polling-based method (e.g., `MW_UART_TransmitUsingPolling`) is used, which incorporates a finite timeout. The XCP stack is unaware of this difference. The driver simply calls a generic send function that returns information about the number of bytes sent in `sizeSent` and a standard error code set.

5.3.3 Receive: buffered extraction

In contrast to CAN reception, hich is message-based at the bus level, serial reception is a stream-based process that requires buffering. The `rtIOStreamRecv` function checks the internal receive buffer using `getBytesToBeReadFromBuffer` before consuming the available bytes using `MW_UART_ReceiveUsingBuffer` (or the LPUART equivalent). This design allows for reading from the background communication task without causing blocking behavior within the model execution context.

5.3.4 Interaction with the Serial frame handler

For the case of Serial transport, the module `xcp_frame_serial.c`ensures that the boundaries of the message are maintained by sending the frame as a sequence with a header, payload, and a tail. The frame handler implements a blocking behavior only after the header step has successfully completed, thereby ensuring that a partially received frame is completed in a consistent way before being sent upwards.

5.4 Consistency with Deterministic Execution

The hardware-dependent backend must be able to safely interact with the concurrency model that was introduced in Chapter 3. This entails a background task for communication, as well as a periodic task for model execution. This translates to the following:

- Peripheral transmit/receive must be callable from the background context;
- Buffering must prevent blocking the real-time loop;
- Error handling must not leak up through the protocol state;

The following elements combine to ensure that variable communication latency does not interfere with the deterministic nature of the target application:

- FIFO buffering in the transport layer;
- Framing isolation in the frame handler (CAN virtual header vs Serial header/counter/checksum);
- Bounded retry/timeout semantics in the XCP driver;
- Aperipheral buffering/DMA decisions at `rtIOStream`;

5.5 Portability to Unsupported Hardware Platforms

The layered structure described in this chapter makes it easier to apply External Mode on hardware systems where there is no official support package. The key idea here is that hardware dependencies are carefully contained within the `rtIOStream` backend layer.

In such a scenario, it is expected that most parts of the External Mode code base will remain unchanged. This is because all hardware-independent components, as described in Chapter 4, will be applicable, such as the External Mode abstraction, XCP protocol layer, XCP transport layer, frame handler, and platform abstraction layer. The distal communication interface will, however, have to be modified to suit the hardware platform.

In terms of implementation, it means developing a new `rtIOStream` backend that interfaces with the communication peripherals of the target system. This adaptation is typically confined to four key functions:

- `rtIOStreamOpen()`, responsible for initializing the communication peripheral and configuring transport-specific parameters;
- `rtIOStreamSend()`, which converts a raw XCP buffer into a transmission request for the platform driver;

- `rtIOStreamRecv()`, which retrieves incoming frames from the communication interface and forwards the payload to the XCP stack;
- `rtIOStreamClose()`, which releases resources and shuts down the communication channel.

These functions act as an adaptation layer between the generic XCP driver and the platform-specific low-level drivers, e.g., the vendor CAN or UART driver. Hence, the process of porting the External Mode on a new target platform will involve changes to the `rtIOStream` backend only, leaving the entire communication stack in place. This has greatly simplified the portability of the XCP-based External Mode on different hardware platforms.

5.6 Summary of Hardware-Dependent Contribution

The layered architectural paradigm is obviously reflected in the organization of the generated and support code. As has been shown throughout this chapter, the components that directly interface with the hardware are limited to the `rtIOStream` backends and to the associated low-level drivers responsible for peripheral access. With regard to the communication media under investigation in this study, the hardware-dependent implementation includes the following components:

- **CAN:** `rtIOStream CAN` (`rtiostream_can.c`), the HAL wrapper layer, and the STM32 HAL CAN driver (`stm_can_hal.c`);
- **Serial:** `rtIOStream Serial` (`rtiostream_serial.c`) together with the underlying UART/LPUART low-level services responsible for buffered or DMA-based communication;

In other words, all components of higher layers, which have been analyzed in Chapter 4, such as the External Mode abstraction, XCP protocol and transport layers, generic driver semantics, and frame management, remain reusable and completely decoupled from the particular communication peripheral and underlying hardware.

The separation between protocol logic and hardware access is considered to be a key architectural property of the External Mode connectivity infrastructure. By containing all platform-specific code within the `rtIOStream` backend,

reusability is ensured across heterogeneous targets and communication media. In fact, it has been observed that when it comes to porting External Mode onto new hardware, it is typically necessary to touch only the `rtIOStream` implementation and associated communication peripheral drivers, with the rest of the XCP-based communication infrastructure remaining unaffected.

Chapter 6

Development Toolchain and Runtime Integration on STM32 NUCLEO-F439ZI

This chapter describes the entire development environment used within this thesis work, based on MATLAB R2025a and targeting the STMicroelectronics STM32F439ZI microcontroller device. As a natural continuation of the topics discussed within the previous chapters of this work, which introduced the XCP-Protocol based External Mode architecture and discussed the split of the stack into hardware-independent logic (protocol, transport, framing, and driver semantics) and hardware-dependent backends (rtIOStream and low-level peripheral APIs), this chapter aims to provide insight into how this architecture is supported by the accompanying development tools and the STM32 microcontroller device ecosystem.
The development environment used within this work is as follows:

- **Simulink and Embedded Coder (MATLAB R2025a)**, For model-based code generation and External Mode integration;
- **Embedded Coder Support Package for STMicroelectronics STM32 Processors**: For target-specific build infrastructure, startup/linker integration, and rtIOStream backends;
- **STM32CubeMX and STM32CubeIDE**: Assisting with clock tree and peripheral configuration as well as maintaining a standard project structure for STM32 devices;
- **GNU Tools for Arm Embedded Processors**: Compilation and linking toolchain used for building the executable;

One of the main aspects of this work is the evaluation of External Mode connectivity on the same target device via two physical media: CAN and Serial (UART). The split of the stack as presented within Chapters 4 and 5 of this work allows for this feature: the XCP-Protocol logic is common to all supported physical media, as is the rtIOStream backend.

6.1 Target Hardware Context: STM32F439ZI

The target microcontroller, in particular, is the STM32F439ZI, a member of the STM32F4 family of high-performance microcontrollers. From the viewpoint of the present study, the STM32F439ZI is a particularly appropriate choice because it has the following characteristics:

- It has a sufficient level of computational power to execute models in real time;
- It has a set of peripherals that can support, in a natural way, both CAN-based and UART-based XCP;

The microcontroller core is based on the ARM Cortex-M4 processor, accompanied by a single precision FPU (Floating Point Unit). This is a significant feature, from the viewpoint of model-based control, because many Simulink models, generated for a variety of purposes, rely on floating-point arithmetic.

Finally, the memory resources of the microcontroller, including Flash memory for code and constants, and SRAM for variables, are sufficient to accommodate the state of the model, together with the necessary buffers for the communication channels of the External Mode.
From the viewpoint of integration, the most relevant peripherals of the STM32F439ZI are:

- **CAN controller**: To support XCP via CAN, in which parameters such as prescaler and time segments, as well as filtering, are significant in defining the actual CAN bus behavior;
- **USART/UART peripherals**: To support XCP via Serial, in which parameters such as baud rate, DMA, and buffering play a significant role in defining the actual Serial communication behavior;
- **DMA controllers and NVIC**: To facilitate non-blocking execution and interrupt-based execution, in a manner consistent with real-time scheduling;

6.2 Embedded Coder and the Generated Application Layer

The role of the Embedded Coder is to generate the deployable code from the Simulink model, but more importantly, to add the integration hooks that the

External Mode operation requires. It should be noted that the generated code artifact includes more than just the step function of the model. It includes the initialization function, the termination function, the timing structures, the parameter and signal access structures, and the glue code to integrate the model with the runtime environment on the microcontroller.
When the External Mode operation is enabled, the code generator adds the External Mode API to the generated code. It does this in two ways:

- **Periodic execution synchronization**: During each base rate step, the generated code executes the `extmodeEvent` function, thus maintaining the timing relationship between the host-based measurement acquisition (DAQ) operation and the model's sample time;
- **Background communication servicing**: Along with the periodic execution of the model, the generated code includes the `extmodeBackgroundRun` function to process incoming commands, service protocol requests, and send responses without interrupting the deterministic control step;

The generated code for the model and the entry-point code generated by the Embedded Coder tool are placed within the build output directory, which is typically the `model_ert_rtw` folder. This folder represents the application layer of the final embedded system. It includes the model code and the runtime structure that describes the execution of the model.

6.3 Support Package Responsibilities: Target Integration and rtIOStream Backends

The Embedded Coder Support Package for STMicroelectronics STM32 Processors provides the necessary infrastructure to compile the generated code and execute the application on the STM32 platform. This tool acts as a bridge between the generic generated code and the actual STM32 execution platform.

6.3.1 HAL integration and peripheral abstraction

The support package includes the STM32 HAL (and, for certain implementations, also the LL) driver library, as well as a common solution for the incorporation of the initialization code generated by the CubeMX tool into the final build process. In other words, it ensures that the CAN bit timing, UART

baud rates, DMA streams, and GPIO settings specified with the CubeMX tool are actually used by the generated application.

6.3.2 rtIOStream backends for CAN and Serial

Another key aspect of the support package with regard to the External Mode architecture is the implementation of hardware-dependent `rtIOStream` backends. In the current development, the following backends are of specific interest:

- **CAN backend** (e.g., `rtiostream_can.c`): Responsible for mapping `rtIOStreamSend/Recv` to CAN transmit/receive operations via the HAL wrapper and CAN peripheral services;
- **Serial backend** (e.g., `rtiostream_serial.c`), Responsible for initializing UART, binding RX/TX buffers, and transmits data, typically via DMA or polling, with buffered reception;

This is the concrete representation of the boundary that has been described in Chapter 4, with the XCP driver making use of a uniform rtIOStream interface and the support package providing the physical implementation.

6.3.3 Build infrastructure using GNU Arm toolchain

The support package, in the end, produces or configures the necessary build artifacts to compile and link the code with the GNU Tools for Arm Embedded Processors. In other words, it provides the necessary Makefiles, compiler flags, and include paths for the model code and the HAL code, as well as the necessary object file and library linking to produce a final ELF executable, which is then converted to a binary for flashing purposes.

6.4 STM32CubeMX and STM32CubeIDE: Peripheral Configuration

STM32CubeMX is utilized to define the low-level hardware configuration of the target device. In the methodology of this thesis, CubeMX is considered to be the single source of truth for clock and peripheral configuration. This is an explicit decision to avoid manual, register-level initializations, as it is ensured that the configuration is explicit, reproducible, and consistent with the HAL and LL layers supported by the vendors. CubeMX is considered to be responsible for the generation of the following:

- `SystemClock_Config()`: Responsible for oscillator, PLL, bus prescaler, and tick timer configuration;
- `MX_..._Init()` function for each peripheral, such as CAN, UART, DMA, and GPIO, wherein parameters such as CAN time segments and UART baud rates are concretely realized as part of the HAL and LL layers;
- DMA stream configuration and NVIC priority configuration, which is considered to be critical for non-blocking communication backends;

The STM32CubeIDE project structure is considered to be responsible for the standardized placement of the initialization code, as well as the code regions intended for the user, which is preserved across code regeneration.

6.5 GNU Tools for Arm Embedded Processors: Compilation and Linking

The compilation and linking are done through the GNU Tools for Arm Embedded Processors. From the thesis writing viewpoint, it is useful to recognize the benefits of understanding the toolchain as the level at which the different layers integrate into one executable artifact:

- Model code originating from `model_ert_rtw`;
- XCP stack (hardware-independent sources);
- rtIOStream backend (support package sources);
- CubeMX-generated HAL initialization and drivers;
- Startup file and linker script;

The linker script is particularly relevant for the External Mode, which determines the amount of SRAM allocated for the buffers (transport queues, RX/TX buffers) while ensuring safe stack operation during interrupt scheduling.

6.6 Entry-Point Integration: `ert_main.c` vs CubeMX/CubeIDE `main.c`

In an STM32 workflow that involves code generation based on a model and configuration based on CubeMX, there are two "main" files that are significant

and perform unique functions. Understanding the purpose of each file is vital for an accurate interpretation of the target's execution architecture.

6.6.1 Embedded Coder entry point: `ert_main.c`

The file `ert_main.c` is a model-centric code that is generated within the `model_ert_rtw` directory. It is the Embedded Coder's main function that determines the execution architecture of the model on the target device by defining the scheduling strategy, overrun protection, inclusion of External Mode, and run lifecycle from initialize to reset.

One of the interesting aspects of this file is the inclusion of a one-step function, which is represented by `rt_OneStep` in this code and is expected to be invoked by a timer interrupt via SysTick. The overrun protection strategy is implemented by checking for the arrival of a new tick before the completion of the current step by checking the `OverrunFlag`variable. This strategy reflects the real-time assumption that the execution period of the step function should always be shorter than the period of the base-rate period. During each step, the model time is extracted from the created timing structure (e.g., `Timing.taskTime0`), the step function of the model is invoked, and subsequently, an External Mode event is triggered via the function `extmodeEvent`. This event is vital for the Monitor & Tune process because it synchronizes the data acquisition sampling of the host with the embedded model's timeline.

The `main()` function created by the code generator for `ert_main.c` follows a clear sequence:

- Hardware initialization occurs. In the present case, this involves explicit calls to initialization functions created by the CubeMX code generator for the present thesis setup, such as `HAL_Init()`, `SystemClock_Config()`, and `MX_..._Init()` for CAN/UART/DMA/GPIO);
- It initializes the model and the External Mode stack (`extmodeInit`);
- It waits for the host start request (`extmodeWaitForHostRequest`);
- Periodic stepping is enabled (in the present case, this involves setting up the SysTick timer according to the model's base rate);
- A loop is entered where `extmodeBackgroundRun` manages the XCP stack while the periodic interrupt invokes `rt_OneStep`;

- It terminates the model and resets External Mode resources;

This structure makes `ert_main.c` the central control point of the code's execution flow and scheduling, while the hardware-specific code remains provided by CubeMX.

6.6.2 CubeMX/CubeIDE entry point: `main.c`

The CubeMX-generated `main.c` is hardware-oriented. Its purpose is to offer a canonical STM32 project template and to implement the chosen configuration in CubeMX as initialization code. The file is marked by the following features:

- Protected user-code areas (`USER CODE BEGIN/END`) which protect the code written by the user and prevent it from being overwritten during regeneration;
- A detailed `SystemClock_Config` function, which covers oscillator sources, PLL, prescaler, and system tick configuration;
- One `MX_..._Init` function per peripheral, including specific CAN bus timing parameters (prescaler, time segment, SJW) and UART-specific parameters (baud rate, GPIO alternate function, DMA);
- Standard error handling and optional asserts;

For example, in this case of study, `MX_CAN1_Init` function specifies CAN bus bit timing and mode parameters and calls `HAL_CAN_Init`. Similarly, `MX_USART3_UART_Init` configures the UART peripheral, including GPIO and DMA parameters for reception. The code is not just boilerplate; it is essential to the correct and reliable functioning of the rtIOStream backends, as described in Chapter 5. The backend assumes that baud rate, DMA, and interrupt priorities have been set to the expected values.

6.6.3 Integration strategy used in this thesis

In this work, CubeMX is used as the authoritative source for hardware configuration, and Embedded Coder is used as the authoritative source for runtime structure. This is achieved by calling the initialization routines generated by CubeMX from within the main function generated by ERT, as seen in the dedicated section of `ert_main.c` that calls `HAL_Init()`, `SystemClock_Config()`, and the `MX_..._Init` functions.
This arrangement provides a clean separation of concerns between the tools:

- Microcontroller configuration is defined by CubeMX (pins, clocks, CAN settings, UART baud rate, DMA streams);
- Execution model and External Mode control are defined by Embedded Coder (including periodic stepping, overrun handling, host synchronization, and background communication handling);
- The XCP stack, sitting on top of the hardware backend, is invariant, allowing the same structure to be used for XCP on CAN and XCP on Serial by selecting the correct frame handler and rtIOStream backend;

6.7 Operational Modes: Simulation, Monitor & Tune, and Build for Monitoring

The chosen toolchain supports various modes of operation. All modes utilize the same model; however, the modes vary with regard to the execution point and the entity that plays the role of the XCP master.

6.7.1 Simulation (host-only execution)

In the simulation mode, the model executes completely on the host side with the assistance of Simulink. This mode offers the greatest level of visibility, allowing for easy debugging; however, it does not account for hardware timing effects, interrupt effects, peripheral effects, or transport layer effects. This mode is mainly used for the initial validation of the model.

6.7.2 Monitor & Tune (External Mode with Simulink as master)

In Monitor & Tune mode, the model executes on the STM32F439ZI, while the XCP master remains connected with the Simulink model. This mode offers the capability for real-time parameter tuning and signal monitoring. From the architectural point of view, this mode makes use of the complete toolchain presented in the paper, which includes the following components:

- Periodic stepping of the model on target (`rt_OneStep`);
- Synchronous event notification to the host (`extmodeEvent`);
- Background servicing of communication (`extmodeBackgroundRun`);

- CAN or Serial transport selected through the model hardware settings;

This mode has the greatest efficiency during the development process, as it allows for the tuning of parameters without the need for re-flashing the model.

6.7.3 Build for Monitoring (A2L-based calibration workflow)

In the Build for Monitoring mode, the model can be executed independently on the target device, while the third-party calibration tool plays the role of the XCP master. In this case, the A2L file is automatically generated containng the information that the model needs for interpreting the memory objects, which can be used for the interpretation of measurements and parameters. This mode differs from the Monitor & Tune mode, as it does not require the execution of the model with the assistance of Simulink.

6.8 Summary

This chapter seeks to illustrate how the toolchain and the STM32 ecosystem enable the External Mode architecture that has been presented in the earlier chapters. The toolchain, comprising MATLAB 2025a and Embedded Coder, generates the model code and the runtime execution structure, while the STM32 Support Package provides the rtIOStream backends and build integrations. The CubeMX and CubeIDE provide reproducible and vendor-supported hardware configuration, and the GNU Arm toolchain provides the final executable.

The explicit decoupling of model-centric execution (`ert_main.c`) from hardware-centric configuration (the initialization code generated by the CubeMX tool and `main.c`) is one of the primary enablers of portability, as the External Mode stack can be made to run over the target platform's CAN or Serial communication infrastructure by modifying the backend and its configuration alone, while leaving the rest of the scheduling and protocol logic intact.

Chapter 7

Experimental Validation of XCP-Based External Mode

As a natural continuation of the discussion on the development toolchain and runtime integration as introduced and explained in Chapter 6, this chapter will present the experimental validation of the External Mode with XCP on the STM32F439ZI microcontroller. The goal will be to demonstrate that the proposed layered architecture as introduced and discussed in Chapters 4 and 5, and the proposed implementation thereof as detailed and discussed in Chapter 6, indeed delivers a working system that supports both measurement and calibration during the execution of the embedded application.
The experiments that will be presented in this chapter are structured as follows:

- **Experiment 1**: Serial (UART) Monitor & Tune with ADC Measurement (Simulink as XCP master);
- **Experiment 2**: CAN Monitor & Tune (Simulink as XCP master);
- **Experiment 3**: CAN Build for Monitoring (Vehicle Spy by Intrepid as XCP master);

The first experiment is aimed at validating the External Mode using the serial protocol, where Simulink is used as the XCP Master. The second experiment is aimed at validating the External Mode using the CAN protocol. In these experiments, the Simulink Monitor & Tune facility will be used, which allows online calibration and measurement directly from the Simulink model. The third experiment is aimed at validating the calibration workflow based on the A2L file, where Simulink is used only to create the application code and the A2L file, which can then be used for calibration using an alternative calibration tool, namely Intrepid Vehicle Spy.

7.1 Experiment 1: Serial (UART) Monitor & Tune

7.1.1 Scope and objectives of the experimental validation

As earlier stated, the validation of this case study is done under Monitor & Tune mode, in which Simulink acts as the master on the host PC and the STM32 NUCLEO-F439ZI Board executes the generated model code as the slave device. This mode is very useful for development and debugging, as it allows for a brief iteration process where parameters can be updated without re-flashing the application and signals can be monitored without having to add any additional code for monitoring.
Therefore, the objectives of this section are:

- Documenting, with reproducible configuration, a set of steps that can be replicated and show how Serial External Mode is enabled for the STM32F439ZI micro-controller setup;
- Validating the "Tune" feature of Simulink by changing one of its parameters and showing deterministic effects on the output of the target device;
- Validating the Simulink "Monitor" feature by monitoring physical quantities, using the Nucleo's integrated ADC sensor, parameters, and LED behaviors, displaying it on Dashboard Scopes and Data Inspector of the Simulink host environment;

7.1.2 Operational mode: Run on Board and Monitor & Tune

The experiment is carried out through the Simulink process as shown in Figure 7.1. The selected hardware platform is STM32F4xx-based board, and the model is executed on the embedded target using the Run on Board feature. In the process, the Monitor & Tune tool is used to establish a connection through External Mode, thus facilitating two-way interaction between the host and running application.

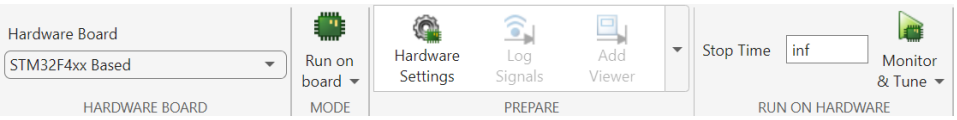


Fig. 7.1: Simulink workflow: board selection and *Monitor & Tune* execution.

From the runtime perspective discussed in Chapter 6, the Monitor & Tune is a concrete sequence of execution and communication. The code generator produces an application with the following characteristics:

- The model is advanced on the target at some periodic rate (base rate scheduling, e.g., SysTick calling `rtOneStep`);
- The host is notified of each advance via a synchronizing event sent on each step of the model using `extmodeEvent`;
- Incoming communication is serviced in the background via `extmodeBackgroundRun`;

This is a critical partitioning of functionality because it preserves a deterministic control step while servicing communications asynchronously, a concern especially relevant for Serial channels because of potential buffer and timing jitter issues with this media compared to CAN.

7.1.3 External Mode configuration for Serial transport

7.1.3.1 Serial selection and transport semantics

The configuration of the External Mode communication interface was explicitly set to Serial, as depicted in Figure 7.2. This configuration step directly influences the hardware-dependent backend chosen by the support package and, therefore, impacts the `rtIOStream` implementation that is eventually linked into the resulting executable.

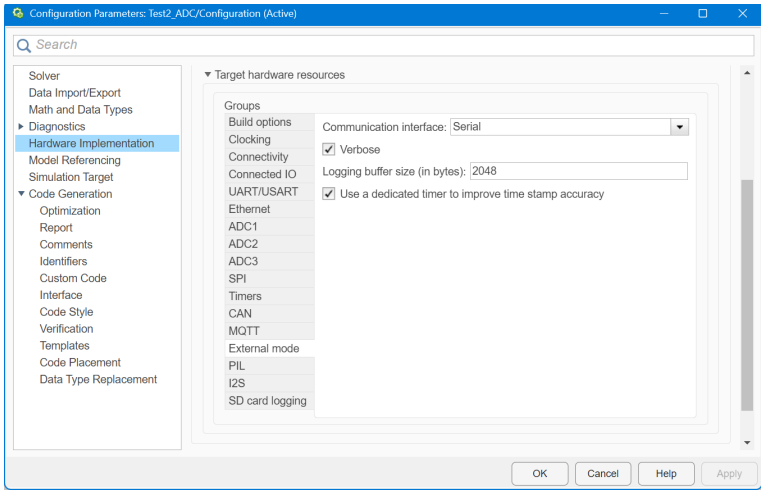


Fig. 7.2: External Mode configuration: Serial communication interface selected

When Serial is selected, the produced code and the External Mode stack will ultimately transfer XCP frames to each other with a UART-based `rtIOStream`

backend. At the architectural boundary defined in Chapter 6, the hardware-independent components of the stack will be the same, while the support package will provide the physical read/write operations. This means that the XCP driver will call `rtIOStreamSend/Recv` and the Serial backend will map these calls to the UART transmit and receive operations on USART3, leveraging the HAL infrastructure provided by CubeMX.

7.1.3.2 Physical link: USB, ST-LINK, and Virtual COM Port

In the employed device of NUCLEO-F439ZI, the STM32 board is connected to the host computer through the onboard ST-LINK USB interface. This allows for power supply, flashing/debugging, and Virtual COM Port (VCP) connectivity to the host OS. This single cable is utilized for communication and is the physical endpoint for Simulink to connect and create the Serial channel for External Mode operation.

From the perspective of the Serial channel, it is apparent that the physical layer is defined between the host and target using the VCP. Simulink sends XCP commands over the COM port, and these are relayed through the ST-LINK bridge and onto the UART pins connected to the target. These commands are received and processed by the target application using the Serial `rtIOStream` backend and USART3. Similarly, measurements and replies are sent back over the target UART and relayed back to the host over the same physical endpoint. This allows for a deterministic setup and avoids the need for additional USB-to-UART converters.

7.1.4 Code generation requirements for calibration: Tunable parameters

A necessary condition for the Tune component of Monitor & Tune is the ability to change parameters at runtime in the generated code. The fundamental configuration for the generated code will be the Default parameter behavior, which was set to Tunable as shown in Figure 7.3. Setting parameters to the "Tunable" state ensures that they are not inlined or optimized away during compilation. As a result, these parameters are preserved as memory-resident objects, typically global variables stored in memory (SRAM).

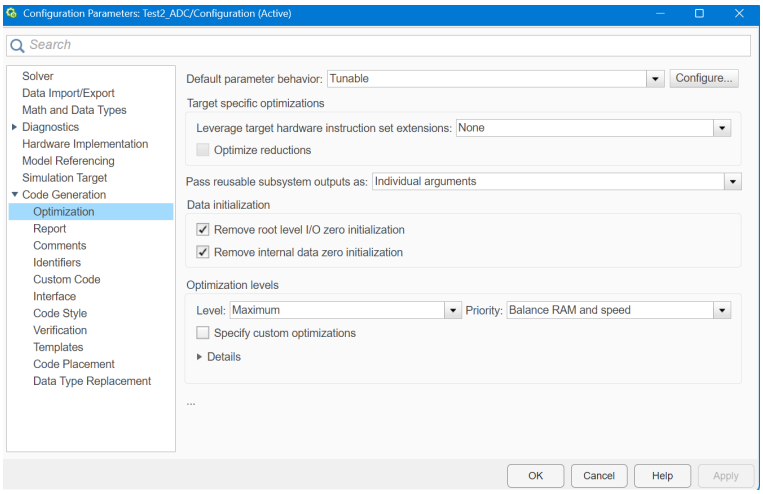


Fig. 7.3: Configuration Parameters: default parameter behavior set to Tunable

This preservation is critical for XCP-based calibration, as the host tool must be able to update new values at well-defined memory addresses via services provided by the protocol. Allowing inlining would cause the code generator to overwrite references to parameters with constant values, eliminating the calibration point.

The Model Data Editor offers an opportunity to gain insight into parameters comprising the tunable set used in the model (Figure 7.4). Within the context of the current experiment, the tunable set includes calibration gains used to manage the logic used to enable the LEDs and breakpoint vectors used in the scaling of the lookup table.

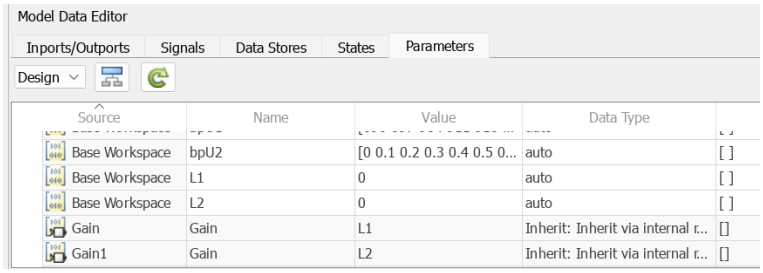


Fig. 7.4: Model Data Editor view of exported parameters

7.1.5 Target hardware configuration with STM32CubeMX

7.1.5.1 ADC1 configuration: temperature sensor as measurable source

The experiment makes use of a measurable value provided by the embedded target. In this context, ADC1 was configured to use the internal temperature

sensor channel, as indicated in Figure 7.5. In doing so, a stable and easily reproducible signal source that does not require external sensors or wiring is provided to validate the measurement path for External Mode.

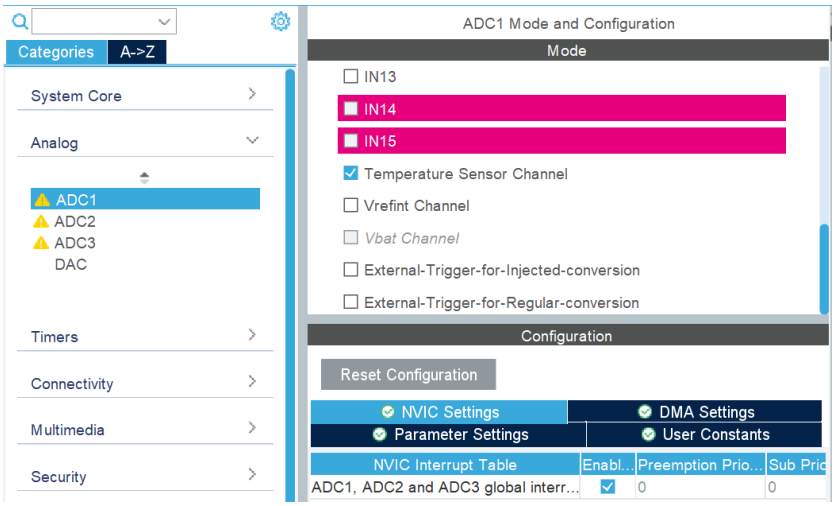


Fig. 7.5: STM32CubeMX: ADC1 configuration with internal temperature sensor channel enabled

7.1.5.2 USART3 configuration: UART backend for External Mode

Serial external mode communication is enabled through USART3, operating in asynchronous mode, as shown in Figure 7.6. The parameters required for the UART configuration, which include the baud rate, word length, and the number of stop bits, as well as the option for the use of DMA and interrupt, are all generated by CubeMX and inserted into the `MX_USART3_UART_Init()` function, which corresponds with the integration strategy as discussed in Chapter 6.

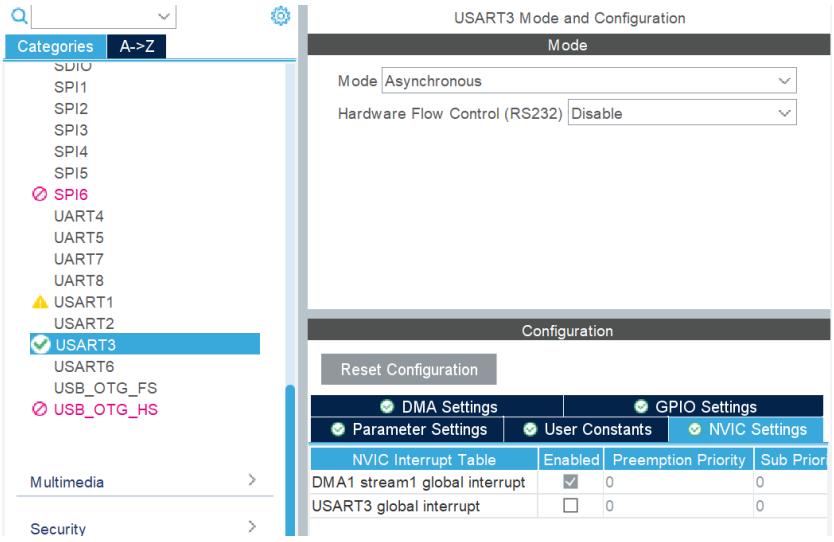


Fig. 7.6: STM32CubeMX: USART3 configuration used for Serial connectivity

The pinout diagram for this configuration is shown in Figure 7.7. Although it is abstracted under `rtIOStream` under External Mode, correct pinout and clock configuration are critical. Any mismatch at the CubeMX level would make the backend layer non-functional, even though the higher layers might be correct.

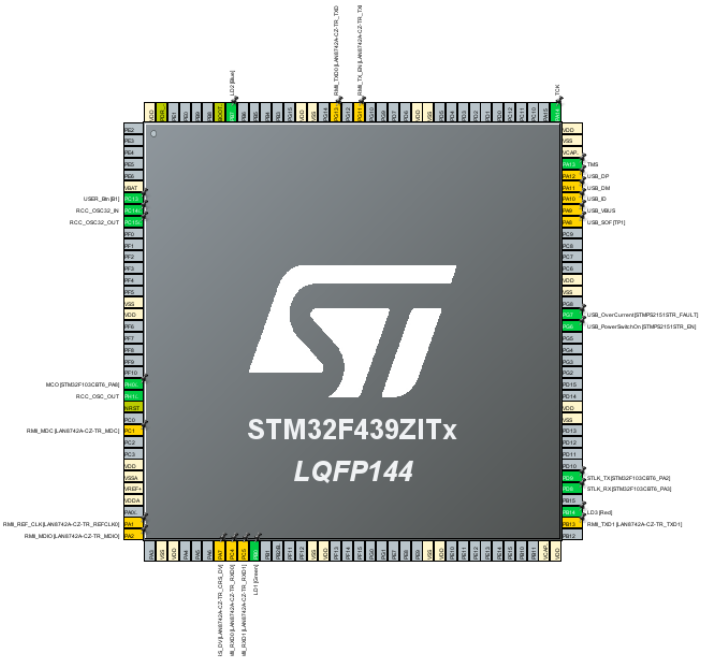


Fig. 7.7: STM32CubeMX: pinout view associated with ADC and USART configuration

7.1.6 Simulink model design for combined measurement and calibration

7.1.6.1 Model structure

The Simulink model utilized in the experiment is shown in Figure 7.8. The model is intentionally simple yet representative of a design that incorporates a sensor measurement, a scaling stage, and a logic block controlling physical outputs. The execution sequence is as follows. First, ADC counts are obtained from the configured ADC channel. However, as embedded ADC drivers are likely to return integer counts, a cast to double is applied for smoother execution of subsequent Simulink blocks. Next, a one-dimensional Lookup Table is applied for the measurement, mapping the ADC domain to a normalized domain for LED modulation. Lastly, there are three branches for generating output for the LEDs. Among these, two branches use tunable gain parameters (L1 and L2), making them runtime adjustable and thus making it possible to use them for calibration.

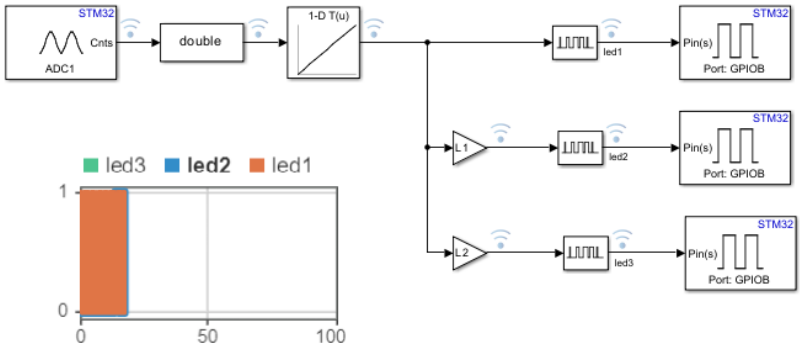


Fig. 7.8: Simulink model architecture for Serial Monitor & Tune validation

This model concurrently exercises both halves of Monitor & Tune:

- **Monitor**: ADC and LED output signals are streamed to the host;
- **Tune**: Runtime modification of gain parameters and immediate effect on GPIO outputs.

7.1.6.2 Lookup table scaling based on experimental breakpoints

The lookup table has been set up based on the experimentally measured values at ambient temperature. Figure 7.9 shows the parameter script used to specify the breakpoint vectors and the gain parameters as Simulink.Parameter. The storage class for the parameters has been specified as `ExportedGlobal`. The parameters will be allocated as global variables and will be accessible to the generated code as well as to other tools.

```
% Breakpoints  X: valori ADC
bpU1 = Simulink.Parameter(890:7:960);
bpU1.StorageClass = 'ExportedGlobal';

% Breakpoints  Y:
bpU2 = Simulink.Parameter([0,0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1]);
bpU2.StorageClass = 'ExportedGlobal';

% Led parameters
L1 = Simulink.Parameter(1); % LED 2
L1.StorageClass = 'ExportedGlobal';
L2 = Simulink.Parameter(1); % LED 3
L2.StorageClass = 'ExportedGlobal';
```

Fig. 7.9: MATLAB parameter script: breakpoint vectors and LED gain parameters

However, a significant limitation, although appropriate for the purpose of this test, is that the lookup table is not defined across the full range of the ADC but is only defined for the range of ADC counts observed at ambient temperature on the target device. This is a common real-world scenario in which the calibration data is valid within a specific operating range.

7.1.7 Runtime calibration procedure and observed effects

7.1.7.1 Parameter update through Monitor & Tune

During the execution of the model in External Mode, parameters were changed directly from the host environment. Figure 7.10 shows the modification of the parameter L1, which commands the behavior of LED2. This process represents a calibration step since the calibration tool, which is Simulink, sends an XCP write command to write the new value into the memory location where the parameter resides.

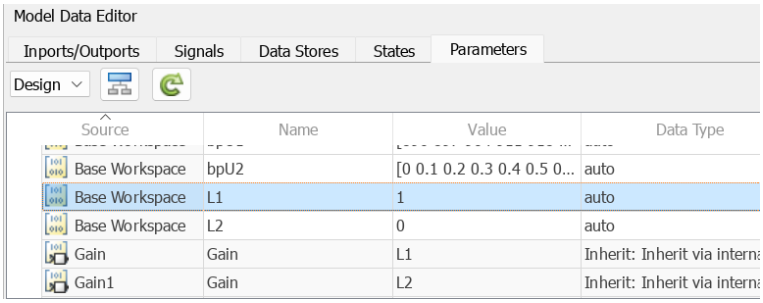


Fig. 7.10: Runtime calibration: modification of parameter `L1` in the Model Data Editor while the target is running

The propagation of the revised parameter values to the target was initiated using the "Update All Parameters" command, as shown in Figure 7.11. In terms of the system model, this command causes the host to send the current set of parameters to the target, so that the embedded application can use the most recent parameter set.

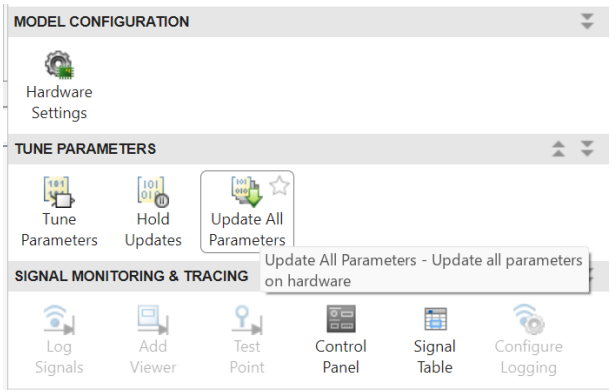


Fig. 7.11: Update All Parameters triggers runtime update on the target

7.1.7.2 Effect on the embedded outputs

This calibration effect can be observed through two ways. First, the Simulink signals are monitored, showing changes in the output signals of the LED.

Second, and more importantly, the calibration effect can be observed on the physical board, thus validating the fact that the update does indeed affect the running embedded code, rather than just affecting the visualization on the host side. The monitoring of the output behavior of one of the LEDs on the host side is shown in Figure 7.12.

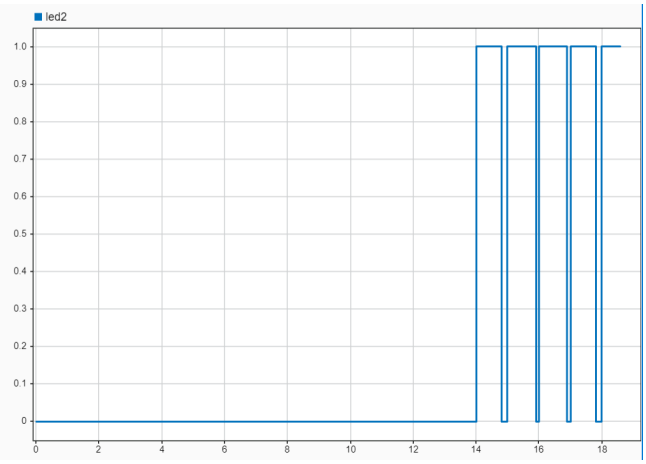
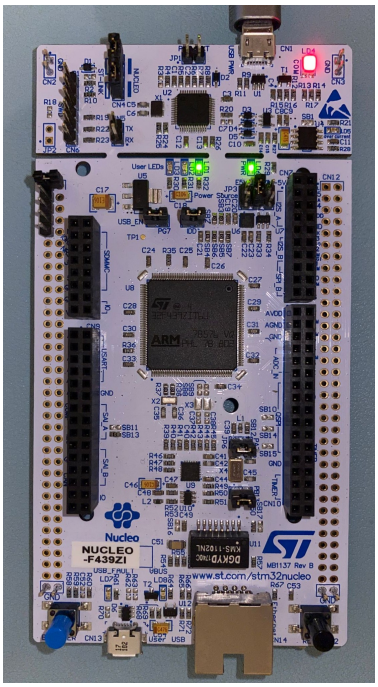
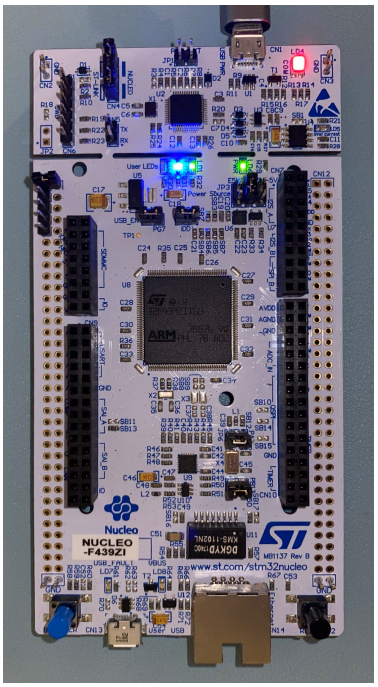


Fig. 7.12: Host-side visualization: LED output signal behavior after runtime calibration

On the board side, Figure 7.13 shows the visual comparison of the physical hardware state before and after the occurrence of the calibration action.



(a) Initial state: only LED1 active



(b) After calibration: LED1 and LED2 active

Fig. 7.13: Physical board behavior before and after runtime parameter update

Before the modification, only LED1 is active, with its duty cycle controlled by the ADC measurement, which is scaled using the lookup table. When the modification to L1 is made, the duty cycle modulation is the same for LED2 as that of LED1. LED3 is kept off because its related parameter, namely L2, hasn't been modified. This is the main observation on which the concept of runtime calibration is based, as the value set through External Mode affects the running algorithm and influences the physical output without the need to reflash and/or restart the application.

7.1.8 Measurement (Monitor) procedure and results

7.1.8.1 Signal streaming to the host: Simulation Data Inspector

The Monitor component of Monitor & Tune was evaluated through streaming internal signals and output signals to the host and displaying these signals in the Simulation Data Inspector (SDI).

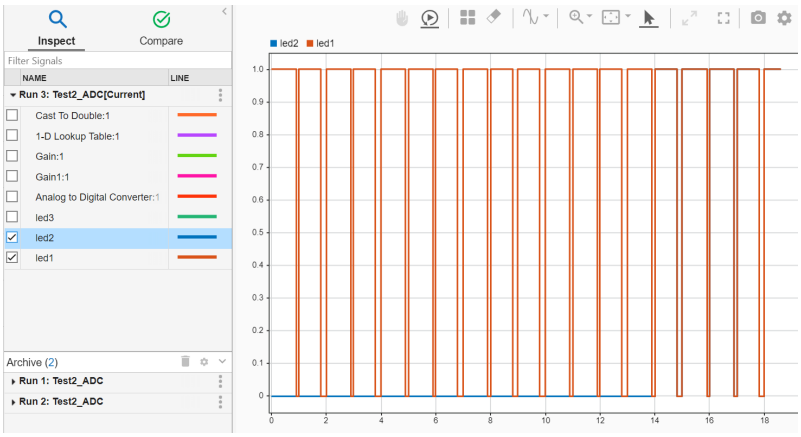


Fig. 7.14: Simulation Data Inspector: monitoring of LED1 & LED2 signals during Serial Monitor & Tune execution

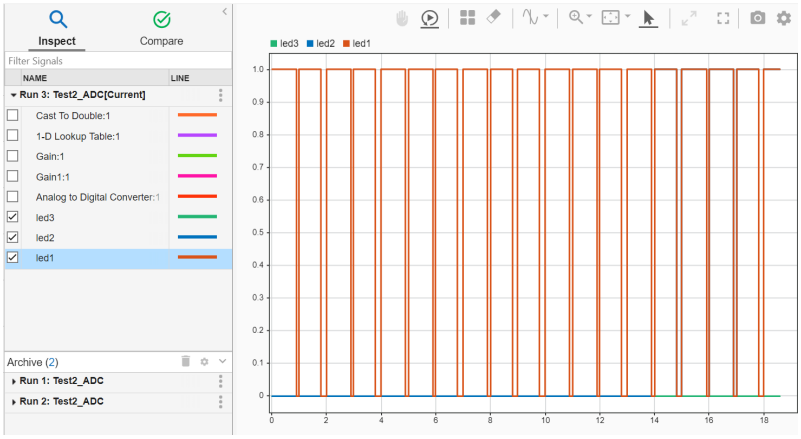


Fig. 7.15: Simulation Data Inspector: simultaneous monitoring of all three LEDs

Figure 7.14 illustrates the SDI view of two LED signals, and Figure 7.15 illustrates the monitoring of three LED output signals concurrently. These repeated transitions validate that the output signals are indeed generated on the target and correctly streamed to the host through the Serial channel.

7.1.8.2 ADC measurement streaming

To validate the system's capability to measure a physical quantity in the real world, ADC counts from the internal temperature sensor were streamed to the host. The results are shown in Figure 7.16. The figure shows that the ADC values change in a limited range. This is because of the stable conditions in the environment, as expected from the selected signal source.

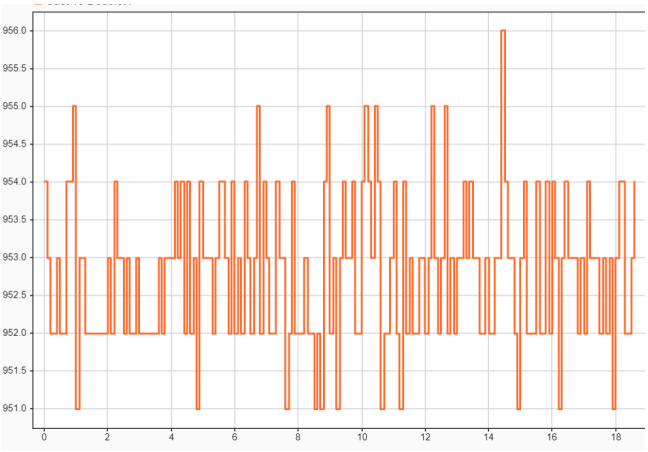


Fig. 7.16: ADC counts acquired from the internal temperature sensor and streamed to the host via Serial External Mode

This result verifies the integrity of the full measurement chain, including the ADC acquisition on the STM32F439ZI, the propagation within the model, and the transfer to the host using the Serial `rtIOStream` backend.

7.1.9 Summary and Conclusion

The results of the experiments conducted in this chapter can be directly related to the runtime architecture discussed in Chapter 6. The periodic execution of the model can be observed in the results as it follows the generated scheduling structure (`rt_OneStep`), while External Mode synchronization is maintained by `extmodeEvent`. The ability to stream signals to the Simulink Data Inspector is maintained by the background servicing loop in the function `extmodeBackgroundRun`, which handles the incoming requests and outgoing data transfers without disrupting the deterministic base-rate step.

Similarly, the execution of the "Tune" functionality can be observed in the results as it follows two conditions: the parameters being maintained as runtime objects as a result of the Tunable configuration, as well as the ability of the host to write new values to the target via the XCP commands sent over the Serial transport protocol. The activation of the LED2 signal as a result of the modification of the value of L1 in the results confirms that the memory write operation can be propagated from the host to the running target algorithm. Finally, the execution of the "Monitor" functionality can be observed in the results as it streams the ADC values as well as the digital output to the host. The results of the experiments conducted in this chapter can be directly related to the toolchain and architecture discussed in Chapter 6.

7.2 Experiment 2: CAN Monitor & Tune (Simulink as XCP master)

7.2.1 Scope and objectives of the experimental validation

As in Experiment 7.1, validation is done in Monitor & Tune mode, where Simulink is configured as XCP master on the host PC and the STM32 NUCLEO-F439ZI runs the generated application as XCP slave. However, in this experiment, the medium of transport is different from Experiment 7.1, where `rtIOStream` is a UART-based transport medium for External Mode. In this experiment, instead, the CAN transport layer is used for External Mode validation. The objectives of this experiment are:

- Enabling and documenting a reproducible CAN-based External Mode configuration on STM32F439ZI;
- Verifying "Tune" functionality by modifying exported calibration parameters during execution and observing deterministic effects on physical LEDs;
- Verifying "Monitor" functionality by streaming internal signals to the host (SDI) while the target runs on the board via CAN;

7.2.2 Operational mode: Run on Board and Monitor & Tune

The operational workflow is the same as in Experiment 7.1, where we create the model and deploy it with Run on Board, followed by the creation of the bidi-

rectional connection with Monitor Tune. Examining the runtime picture from Chapter 6, we can observe that the deterministic base rate execution of the code, which is based on the generated scheduler (e.g., `rt_OneStep`), while External Mode communication is serviced asynchronously (e.g., `extmodeEvent` for synchronization and `extmodeBackgroundRun` for background servicing). The difference now is that the XCP frames use the CAN protocol, which affects the physical layer, framing, and the host interface for the device.

7.2.3 External Mode configuration for CAN transport

7.2.3.1 CAN selection and host interface

The External Mode interface was configured for CAN, while Simulink continued to function as the host interface, as shown in Figure 7.17. This selection determines which CAN backend is used by the support package for its low-level send/receive implementation, while the hardware-independent XCP layers remain unchanged.

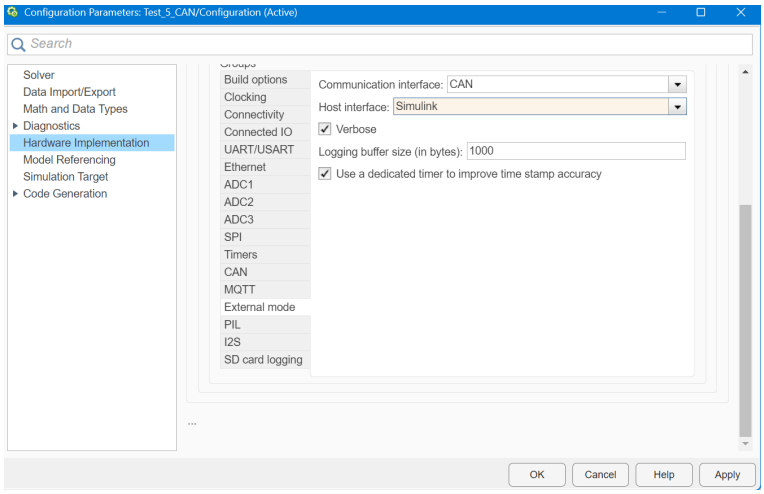


Fig. 7.17: External Mode configuration: CAN interface selected

7.2.3.2 CAN device binding and addressing

After selecting CAN, the CAN interface on the host side gets bound to a particular vendor, device, and channel, along with message identifiers. In this configuration, the CAN vendor is Vector, and it's a VN1630A device, and the channel is specified along with the CAN command and response identifiers, as shown in Figure 7.18. These identifiers determine how XCP master and slave communicate with each other through commands and responses, over the CAN bus during the External Mode session.

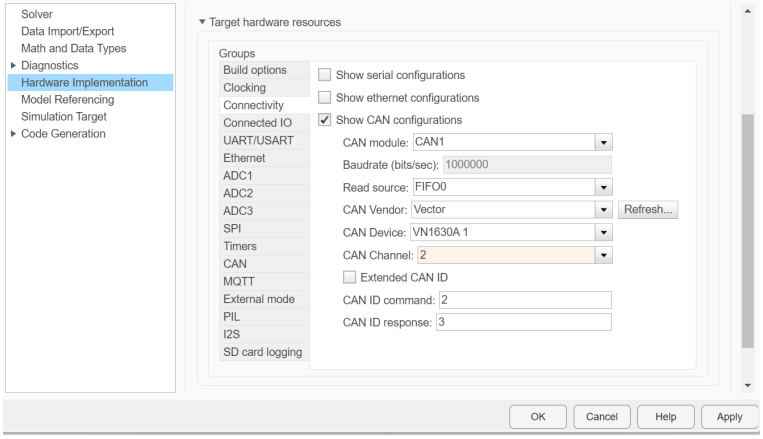


Fig. 7.18: CAN settings for External Mode: Vector VN1630A device selection, channel selection, and CAN command/response IDs

7.2.4 Target hardware configuration with STM32CubeMX

7.2.4.1 CAN1 activation and bit timing

CAN communication on the target can be initiated by enabling the CAN1 peripheral and setting the bit timing parameters according to the chosen baud rate. Figure 7.19 indicates the CubeMX configuration for the CAN1 peripheral, including the parameters for the chosen baud rate.

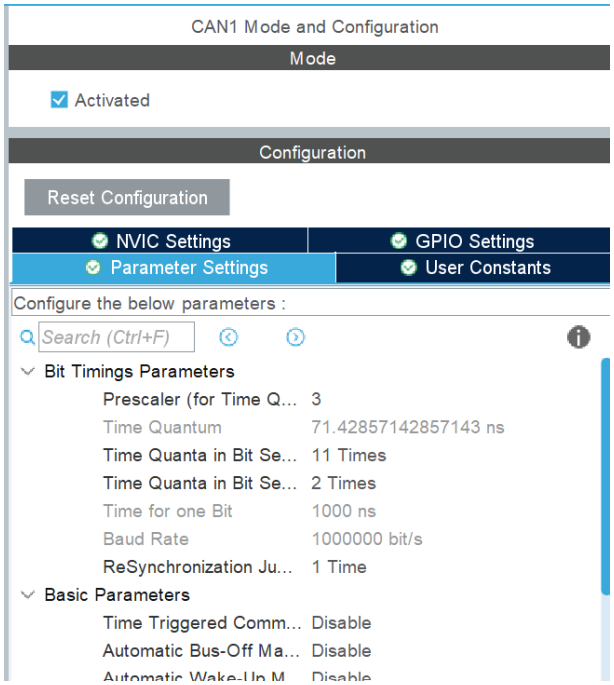


Fig. 7.19: STM32CubeMX: CAN1 configuration

7.2.4.2 CAN1 GPIO mapping: PD0 (RX) and PD1 (TX)

On the STM32F439ZI, CAN1 is mapped to **PD0** (CAN1_RX) and **PD1** (CAN1_TX) in alternate function mode (Fig. 7.20).

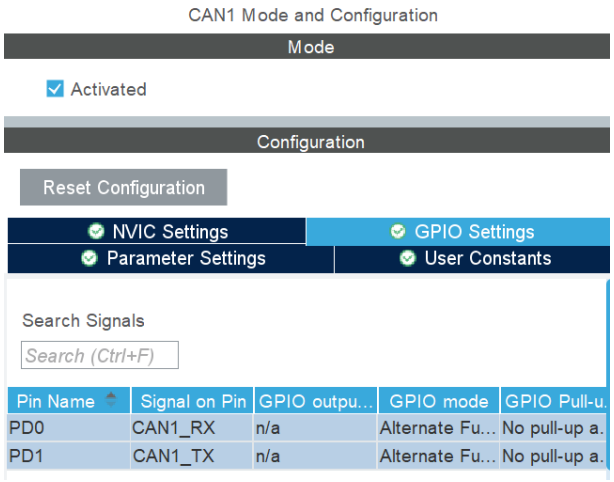


Fig. 7.20: STM32CubeMX GPIO mapping for CAN1: PD0 configured as CAN1_RX and PD1 configured as CAN1_TX.

The configuration must be consistent across CubeMX pin assignment, generated initialization code, and the physical wiring to the external transceiver. For completeness, Figure 7.21 shows the CAN-related pins in the overall MCU pinout view.

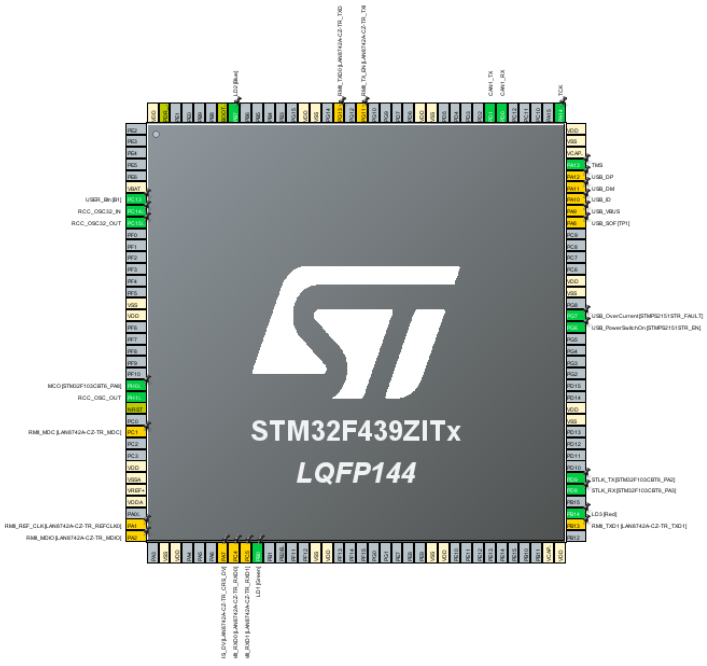


Fig. 7.21: STM32CubeMX: MCU pinout view associated with the CAN-related pins

7.2.5 Hardware setup for CAN External Mode

Unlike in the Serial case, which uses the on-board ST-LINK VCP, CAN External Mode uses a dedicated CAN physical layer interface in both the host and target sides:

- On the **host** side, a **Vector VN1630A** provides the CAN/USB interface (PC ↔ CAN bus).
- On the **target** side, the STM32 MCU provides only the CAN controller; a **CAN transceiver** is required to translate MCU logic-level CAN_TX/CAN_RX to the differential bus signals **CAN_H** and **CAN_L**;

Figure 7.22 presents the entire setup used in the experiment: VN1630A CAN/USB bridge connects to the host PC via USB, CAN transceiver module on a breadboard, and NUCLEO-F439ZI board connects to CAN transceiver via jumper wires.

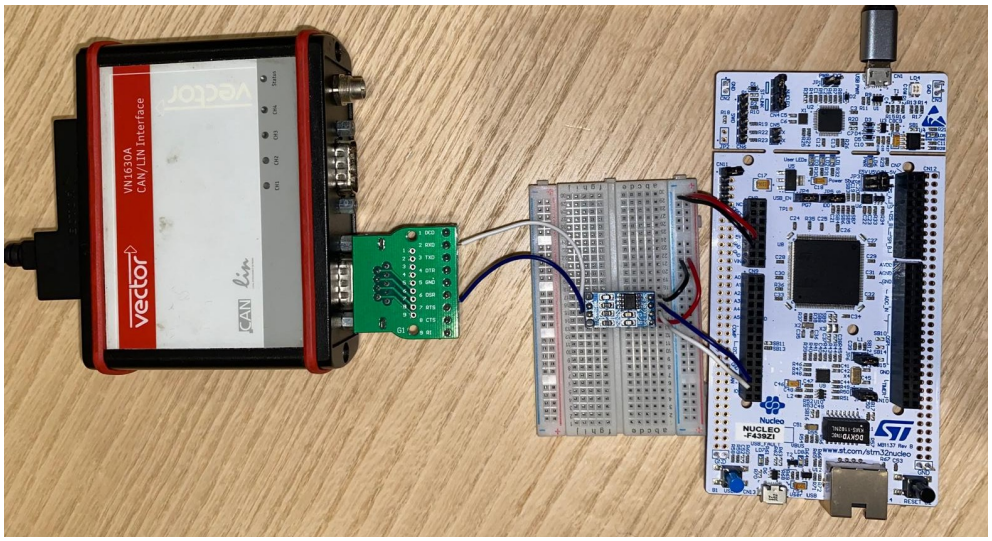


Fig. 7.22: Hardware overview: Vector VN1630A (host CAN/USB interface), CAN transceiver module on breadboard, and STM32 NUCLEO-F439ZI target

7.2.5.1 Board connector mapping: locating PD0/PD1 on the NUCLEO headers

To properly connect it, it is important to ensure that CAN1 uses the PD0 and PD1 pins of the Nucleo board header. Figure 7.23 indicates how the header pins are mapped out with PD0 and PD1 properly exposed and identified for CAN1 receive and transmit operations.

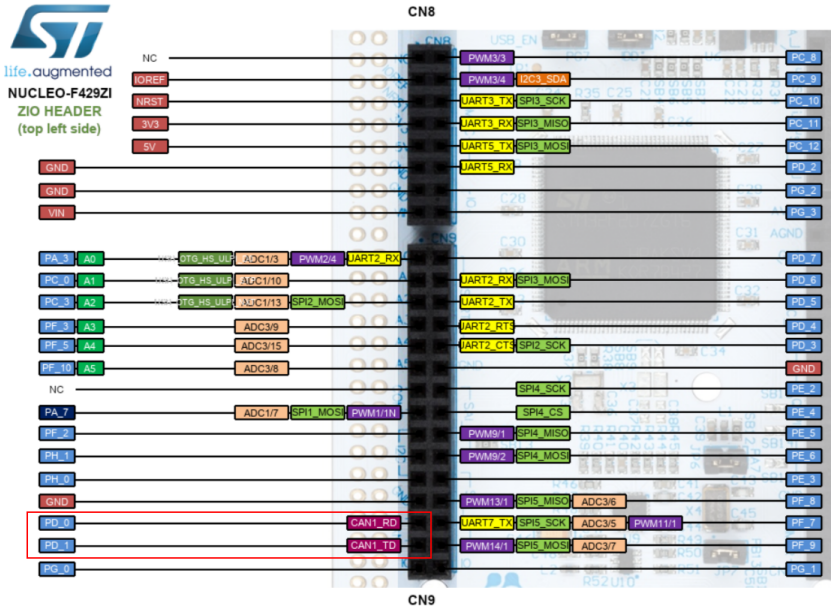


Fig. 7.23: NUCLEO header mapping used to locate PD0 and PD1 for CAN1: PD0 (CAN1_RD / CAN1_RX) and PD1 (CAN1_TD / CAN1_TX).

7.2.5.2 Electrical wiring: MCU → Transceiver → CAN Bus

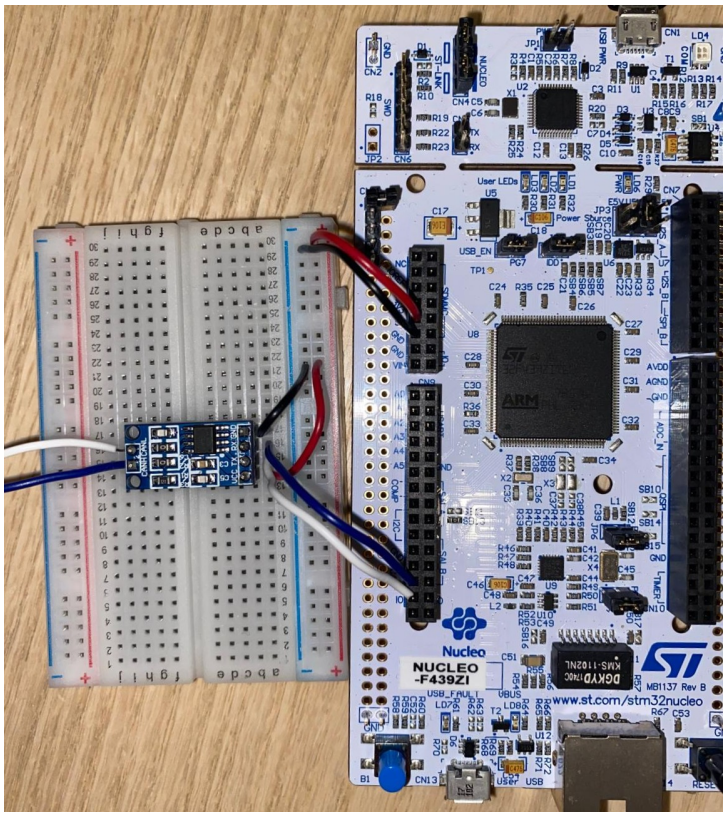


Fig. 7.24: Target-side close-up: wiring between NUCLEO-F439ZI headers (PD0/PD1) and the CAN transceiver module on the breadboard

The wiring matches the standard CAN physical layer partitioning:

- **MCU-side (logic):** The pin PD1 connects to the transceiver's TXD, and the transceiver's RXD connects to the pin PD0;
- **Bus-side (differential):** The CAN_H and CAN_L lines of the transceiver connect to the CAN channel provided by the VN1630A throuh CAN interface;
- **Reference and supply:** The NUCLEO board and the transceiver must have a common ground reference, and the transceiver must be supplied with power (connected to the PC's USB port);

Figure 7.24 above provides a close-up view of how the wiring on the target side is routed from the NUCLEO headers to the CAN transceiver module on the breadboard.

7.2.6 Simulink model design for Monitor & Tune over CAN

The model used in Experiment 2, as shown in Figure 7.25, is simple and has a few parameters. There are three exported parameters, Par_Led1, Par_Led2, and Par_Led3, which are used to determine the logic used to drive the LEDs. This is great for playing with Tune by adjusting these parameters and at the same time observing Monitor in action by streaming the LEDs back to the host.

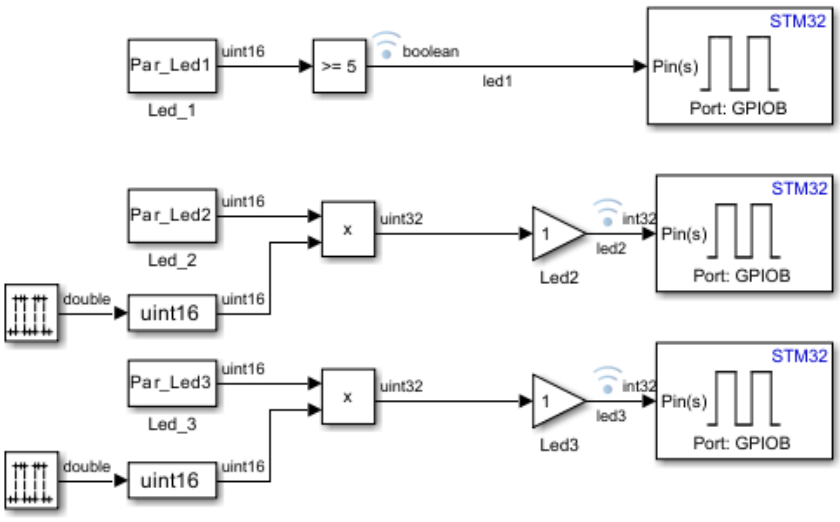


Fig. 7.25: Simulink model architecture for CAN Monitor & Tune validation

7.2.7 Code generation requirements for calibration: Tunable parameters

The Tune feature is based on the same assumption as before: calibration parameters must remain as memory-resident data, not inlined or optimized away. Hence, model parameters controlling LED behavior are specified as exported, typed runtime parameters. In this code snippet, exported parameters are specified as Par_Led1, Par_Led2, and Par_Led3. They are specified as Simulink.Parameter types with `StorageClass = ExportedGlobal`, as shown in Figure 7.26.

```
Par_Led1 = Simulink.Parameter(10);
Par_Led1.DataType = 'uint16';
Par_Led1.StorageClass = 'ExportedGlobal';

Par_Led2 = Simulink.Parameter(1);
Par_Led2.DataType = 'uint16';
Par_Led2.StorageClass = 'ExportedGlobal';

Par_Led3 = Simulink.Parameter(1);
Par_Led3.DataType = 'uint16';
Par_Led3.StorageClass = 'ExportedGlobal';
```

Fig. 7.26: MATLAB parameter script: exported global parameters `Par_Led1`, `Par_Led2`, and `Par_Led3`.

7.2.8 Runtime calibration procedure and observed effects

7.2.8.1 Parameter update through Monitor & Tune

As shown in Figure 7.27, the parameter setting in the Model Data Editor can be seen at the beginning of the run. During the execution of the run, the host updates the parameter values, and the "Update All Parameters" action sends the updated calibration set to the target via XCP on CAN.

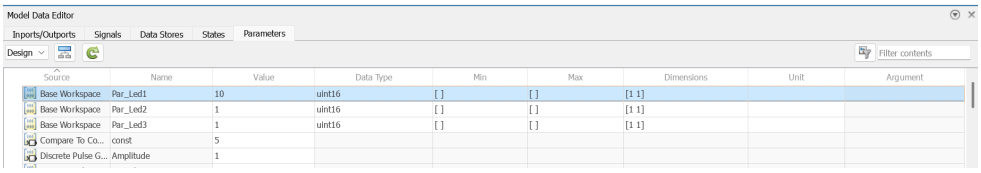


Fig. 7.27: Initial parameter set in the Model Data Editor

A representative tuning action is shown in Figure 7.28, where `Par_Led1` is modified while the application is running.

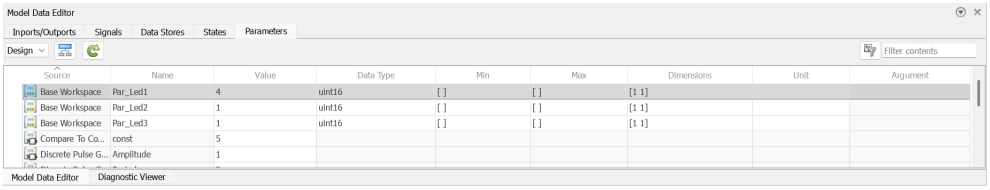


Fig. 7.28: Runtime calibration: modification of `Par_Led1` during execution (CAN Monitor & Tune).

A second tuning action is shown in Figure 7.29, where `Par_Led3` is set to a different value to force a deterministic change on the corresponding LED output branch.

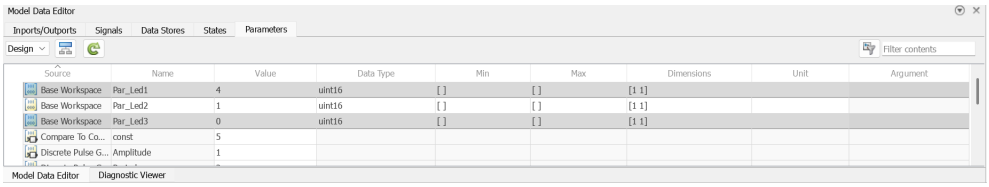
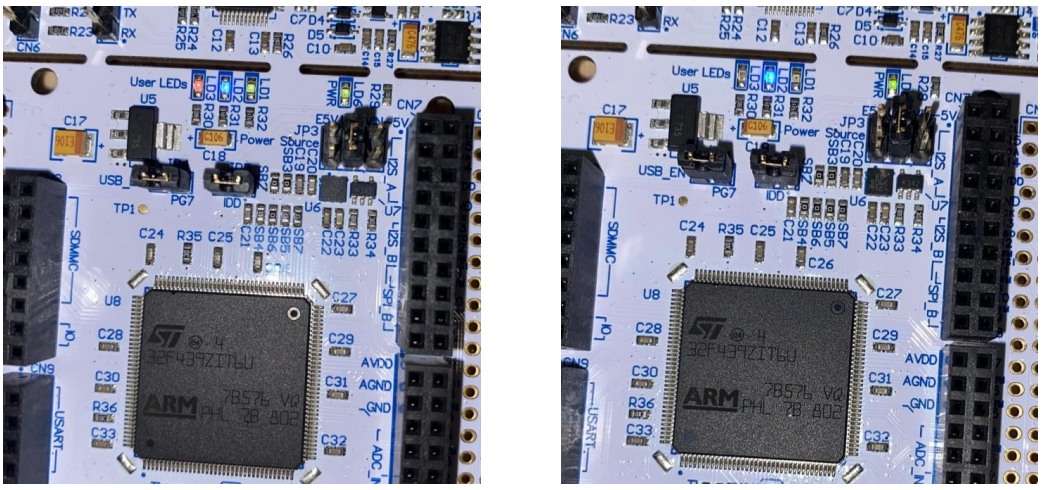


Fig. 7.29: Runtime calibration: modification of `Par_Led3` during execution (CAN Monitor & Tune).

7.2.8.2 Physical effect on the embedded outputs

The key validation for this is that not only do the new parameters change the host-side visuals, but they also change the way the code embedded within the system runs. This can be validated by watching the LEDs on the NUCLEO board itself.



(a) Before tuning action

(b) After complete tuning action

Fig. 7.30: Hardware-side evidence of tuning: physical LED state changes after runtime parameter update over CAN

7.2.9 Measurement (Monitor) procedure and results

The "Monitor" component is validated by streaming the LED signals back to the host and checking them in the Simulation Data Inspector (SDI).

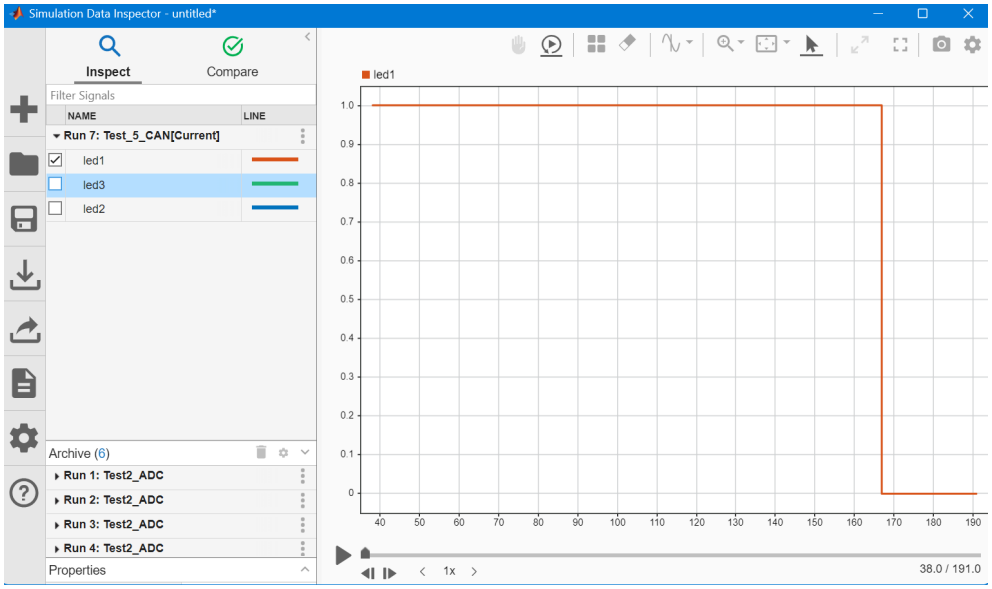


Fig. 7.31: Simulation Data Inspector: monitored `led1` signal during CAN Monitor & Tune execution

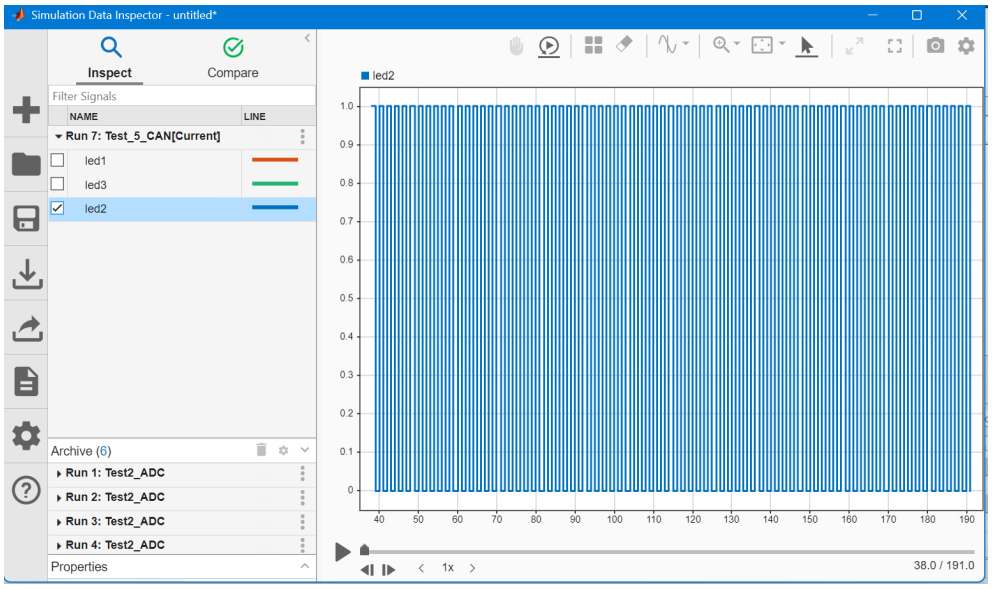


Fig. 7.32: Simulation Data Inspector: monitored `led2` signal during CAN Monitor & Tune execution

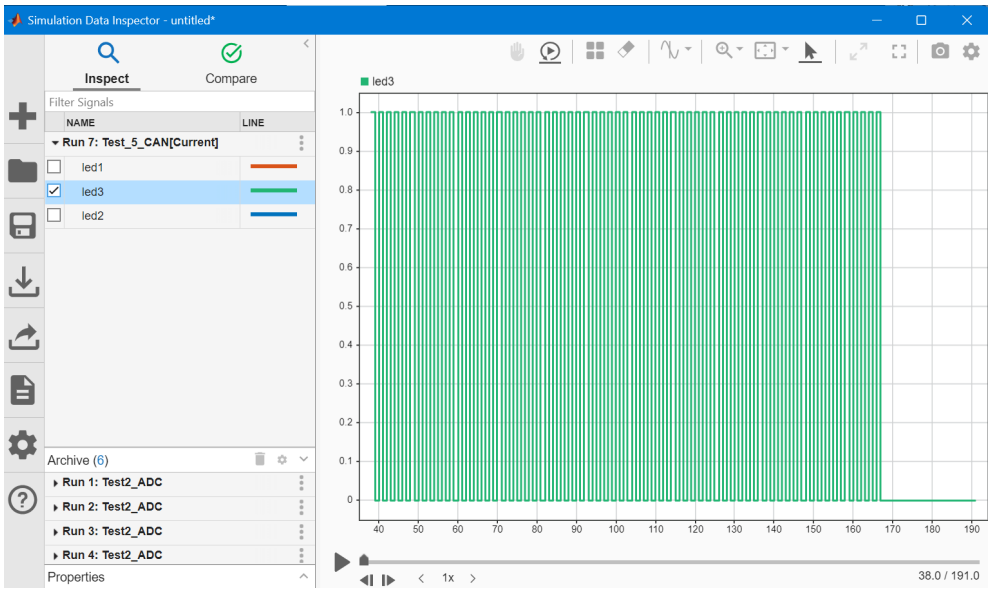


Fig. 7.33: Simulation Data Inspector: monitored `led3` signal during CAN Monitor & Tune execution

7.2.10 Summary and Conclusion

Experiment 2 also demonstrates that the External Mode runtime integration, as described in Chapter 6, still functions when the transport protocol is switched from Serial to CAN. What this demonstrates is that our updates to the parameters that we are sending over to the target hardware are indeed being received. We can see this with the LED lights, for instance. The LEDs turn off right after receiving the updates through the XCP protocol over CAN bus communication. In particular, LED1 and LED3 turn off, but LED2 continues to blink since its parameter has not been updated and continues to operate in the same manner as it was modeled to operate in. The SDI traces also indicate that the monitor stream back to the host from the application has not been affected at all.

7.3 Experiment 3: CAN Calibration using a Third-Party Tool (Vehicle Spy)

7.3.1 Scope and objectives of the experimental validation

The third experiment follows the validation framework developed in the previous experiments and illustrates how the calibration workflow works in conjunction with a third-party professional tool, rather than the Simulink Monitor & Tune interface. In Experiments 1 (section 7.1) and 2 (section 7.2), Simulink

was utilized as the XCP master and communicated directly to the target via External Mode. In this scenario, the embedded application is accessed via the external calibration environment Vehicle Spy, developed by Intrepid Control Systems. In this scenario, the STM32 NUCLEO-F439ZI board will operate as the XCP slave, and Vehicle Spy will operate as the XCP master. The interface between the host and the embedded application will be via the CAN bus, utilizing the ValueCAN interface from Intrepid Control Systems. As a note, the Simulink model utilized in this scenario will be the exact same model utilized in Experiment 2. The architecture of the Simulink model, the calibration parameters, and the signals will be the same, only the deployment scenario and the tool utilized for runtime calibration and monitoring will change. The objectives of this experiment will be to:

- Validate the usage of the generated embedded application in conjunction with a third-party calibration tool;
- Validate the proper creation and utilization of the A2L file;
- Demonstrate the utilization of runtime monitoring and calibration via XCP on the CAN bus;
- Validate the utilization of the runtime start feature, utilizing the variable `xcpModelStartRequest`;

7.3.2 External Mode configuration for third-party calibration tools

To enable an external calibration setup, the settings related to External Mode must be adjusted in the hardware implementation part of the Simulink model.

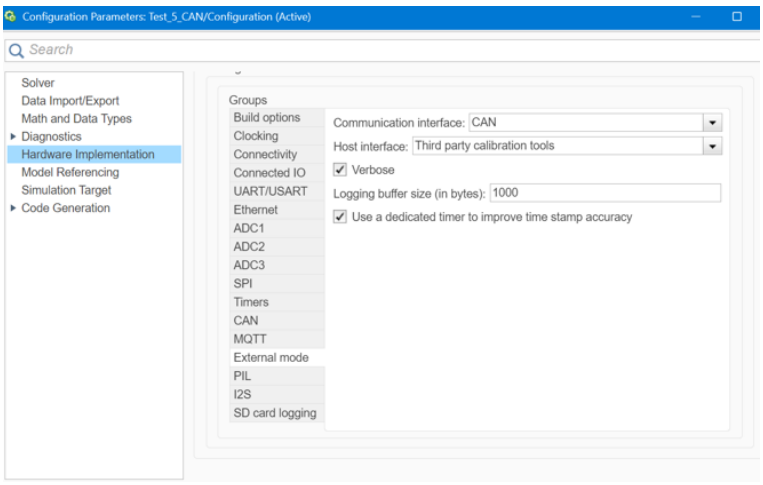


Fig. 7.34: External Mode configuration to enable communication with Third-party tools

The interface type should remain set as CAN, and the interface type for the Host should be Third-party calibration tools, as shown in Figure 7.34.

7.3.3 Code generation workflow: Build for Monitoring

In contrast to Experiments 1 and 2, the model is not executed using the "Monitor & Tune" workflow in the current case. Instead, the executable application is created using the "Build for Monitoring" option provided in the "Run on Board" interface for the deployment of the model. This path results in the generation of the instrumented version of the deployed application, which is intended to allow the exposure of the variables for the purpose of calibration and measurement through the XCP protocol. The executable application, as just said, is also accompanied by the A2L file that contains the information regarding the parameters that can be used for the purpose of calibration as well as the measurements that can be performed. Figure 7.35 shows the Simulink interface where the *Build for Monitoring* option is selected.

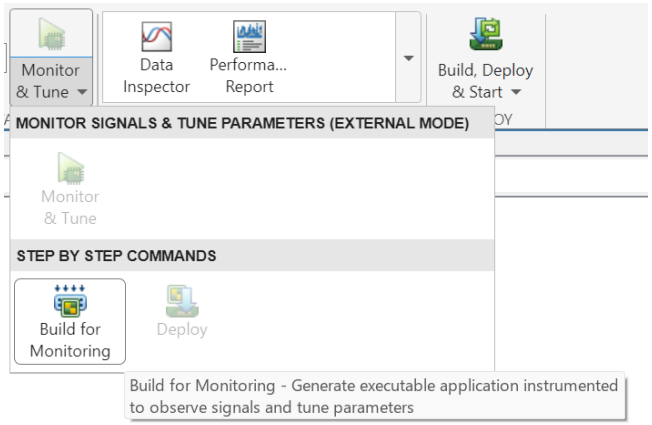
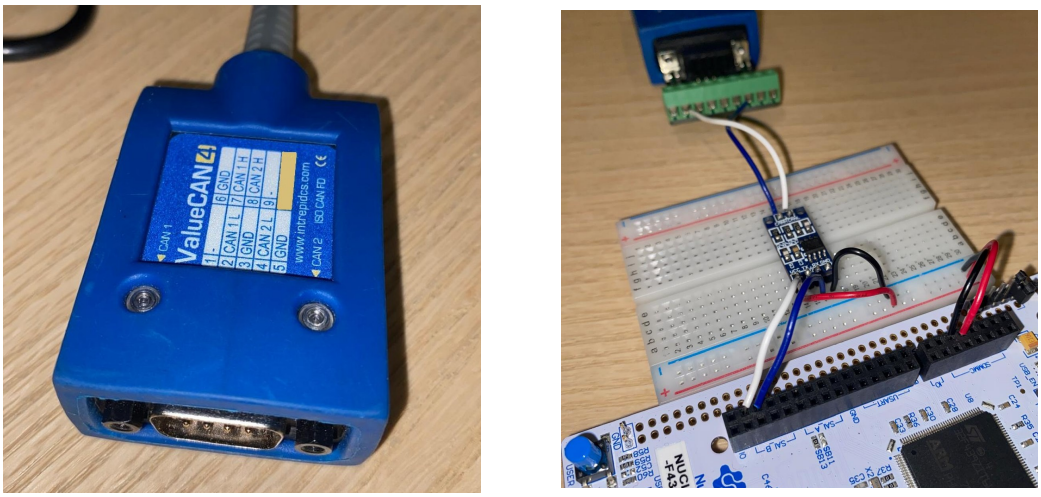


Fig. 7.35: Simulink deployment interface showing the Build for Monitoring option

Once the build is complete, the executable is ready to be flashed onto the STM32 target board. This operation can be done directly via Simulink. The monitoring and calibration tasks will be carried instead using Vehicle Spy.

7.3.4 Hardware setup and CAN communication interface

The hardware configuration is essentially the same as used in Experiment 2, with a major exception: the interface to the host computer via CAN. Rather than the Vector interface used before, the CAN network is connected to the host PC via an Intrepid ValueCAN interface, supported by the Vehicle Spy calibration environment.



(a) Intrepid ValueCAN Interface

(b) View of the CAN transceiver wiring

Fig. 7.36: Close-up view of the CAN transceiver wiring used in the experimental setup

The setup consists of:

- STM32 NUCLEO-F439ZI board executing the generated application;
- External CAN transceiver mounted on a breadboard;
- Intrepid ValueCAN interface connected to the host PC;

The CAN controller integrated in the STM32 communicates with the external transceiver via CAN1 peripheral pins PD0 and PD1.

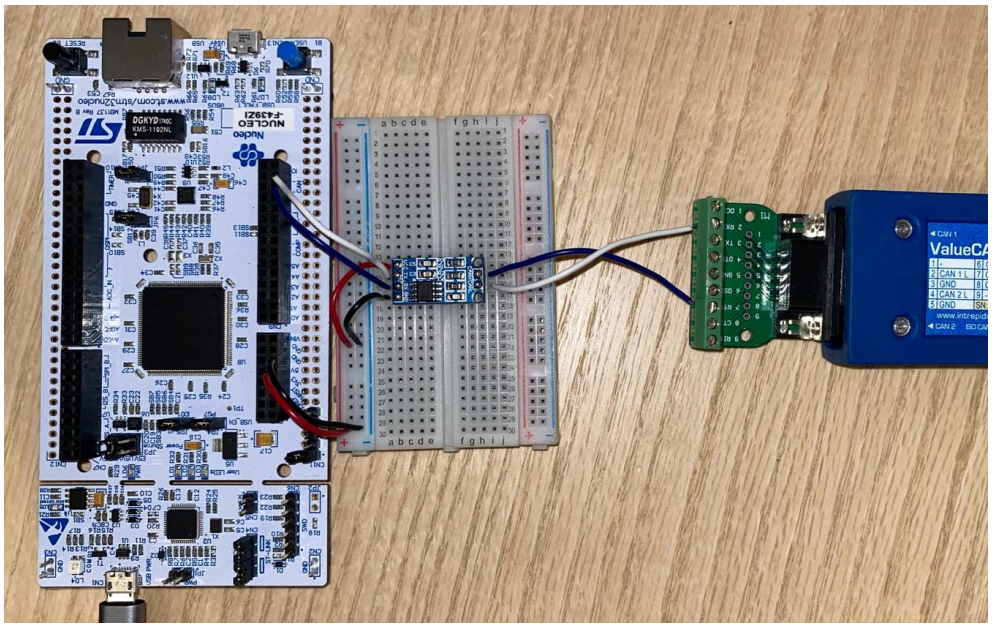


Fig. 7.37: Complete Experimental Setup

The transceiver converts the logic-level signal from the microcontroller into the differential CAN bus signal required by the physical layer, namely the CAN_H and CAN_L signals.

7.3.5 Vehicle Spy configuration and A2L import

After the embedded application has been deployed onto the target board, the generated A2L file is launched into the Vehicle Spy calibration environment. Figure 7.38 shows a screenshot of the Vehicle Spy interface after a successful ECU connection via the CAN interface.

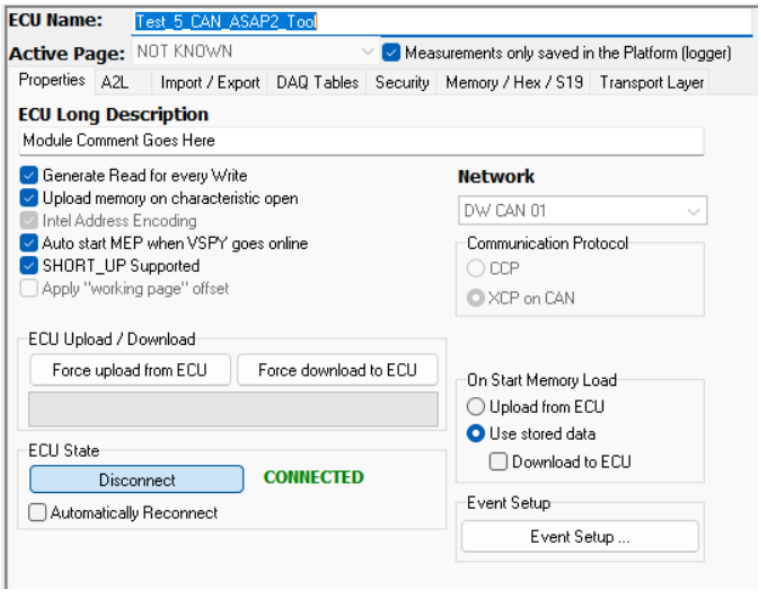


Fig. 7.38: Vehicle Spy interface showing the ECU configuration and XCP connection status

The XCP commands and their corresponding responses use special CAN identifiers that are identical to those used in the Simulink environment to set up the CAN configuration.

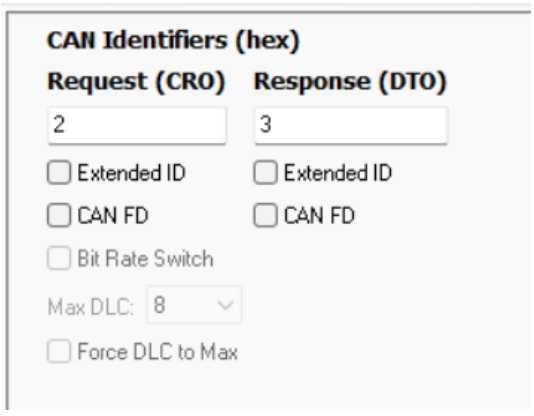


Fig. 7.39: CAN identifiers used for XCP command and response messages

Once the A2L file has been imported into Vehicle Spy, it automatically detects the calibration parameters and measurement signals provided by the application.

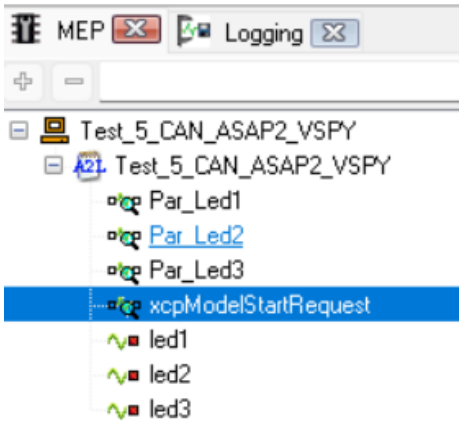
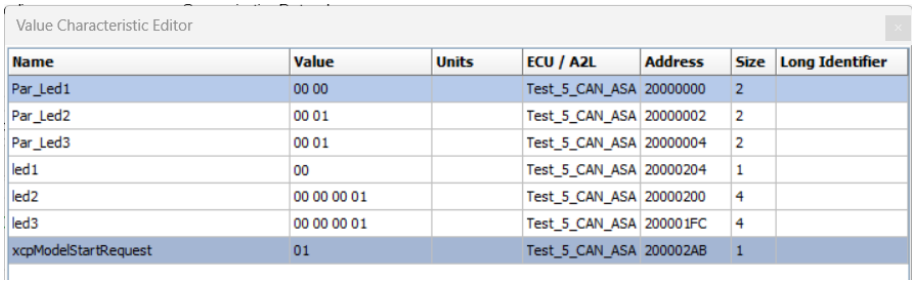


Fig. 7.40: List of calibration parameters and measurement signals detected from the A2L file.

The application parameters and measurement signals can be accessed and changed through the calibration interface.



Name	Value	Units	ECU / A2L	Address	Size	Long Identifier
Par_Led1	00 00		Test_5_CAN_ASA	20000000	2	
Par_Led2	00 01		Test_5_CAN_ASA	20000002	2	
Par_Led3	00 01		Test_5_CAN_ASA	20000004	2	
led1	00		Test_5_CAN_ASA	20000204	1	
led2	00 00 00 01		Test_5_CAN_ASA	20000200	4	
led3	00 00 00 01		Test_5_CAN_ASA	200001FC	4	
xcpModelStartRequest	01		Test_5_CAN_ASA	200002AB	1	

Fig. 7.41: Vehicle Spy value editor showing calibration parameters and memory addresses

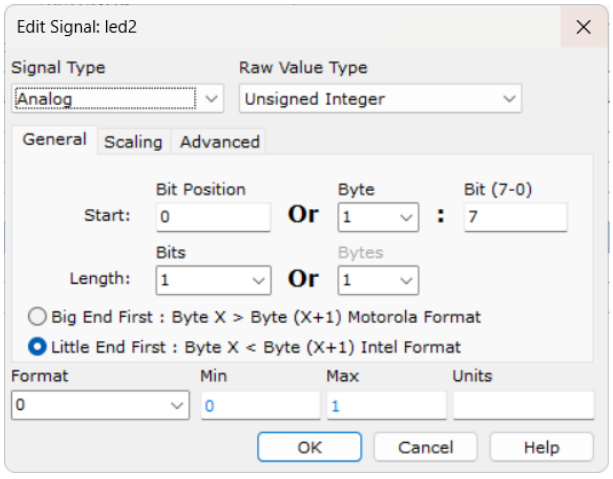


Fig. 7.42: Signal configuration window used to define the encoding of monitored signals.

Figure 7.43 shows the full Vehicle Spy window once the A2L file has been imported and the ECU connection has been successfully established.

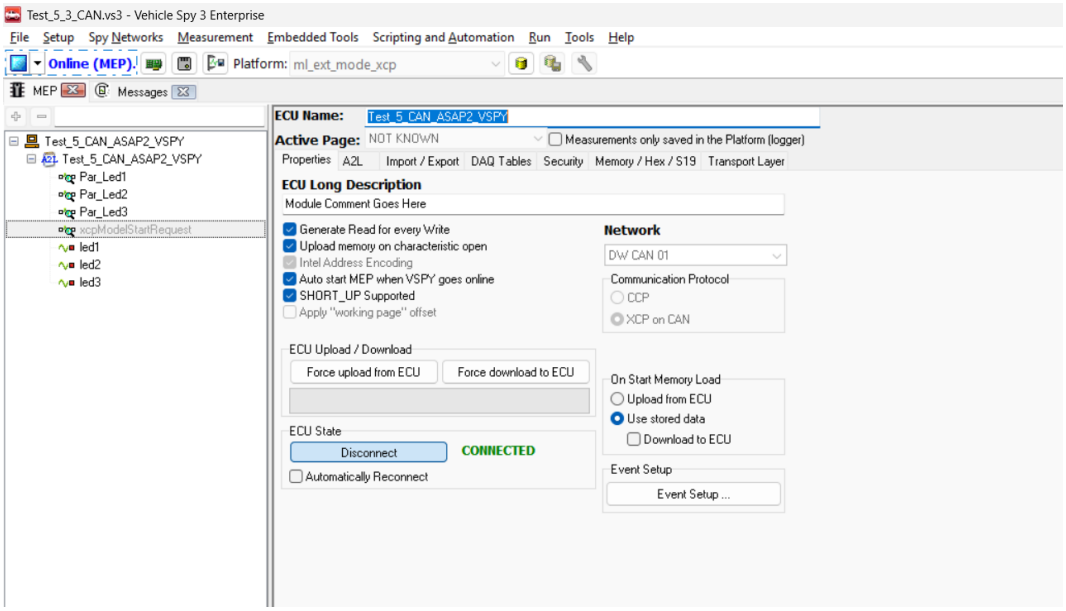


Fig. 7.43: Vehicle Spy complete interface

7.3.6 Runtime model start using `xcpModelStartRequest`

It's worth noting that when the generated application boots on the target board, it doesn't start executing the model immediately. Instead, it waits and expects a start request from the host calibration tool.
This can be seen in the generated code for the `main()` function, wherein after the initialization of External Mode, there is the following instruction:

```
extmodeWaitForHostRequest(EXTMODE_WAIT_FOREVER);
```

By default, this variable will have a zero value. The model will only be started when the value of the variable `xcpModelStartRequest` is set to 1.
The address of this variable will not be found in the A2L description file. Instead, you need to obtain it from the linker map file, which is created during compilation. From this file the variable address can be identified as:

```
0x200002AB
```

This address corresponds to a RAM memory location of the STM32 microcontroller. Figure 7.44 shows the system state before issuing the start request.

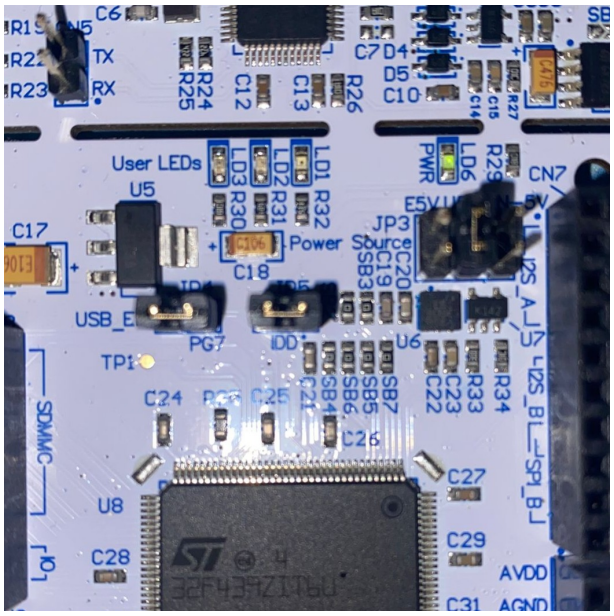


Fig. 7.44: System state before the model start request is issued.

Thus, in order to start the model execution, as just explained, the variable value must be changed from:

$$\texttt{0x00} \rightarrow \texttt{0x01}$$

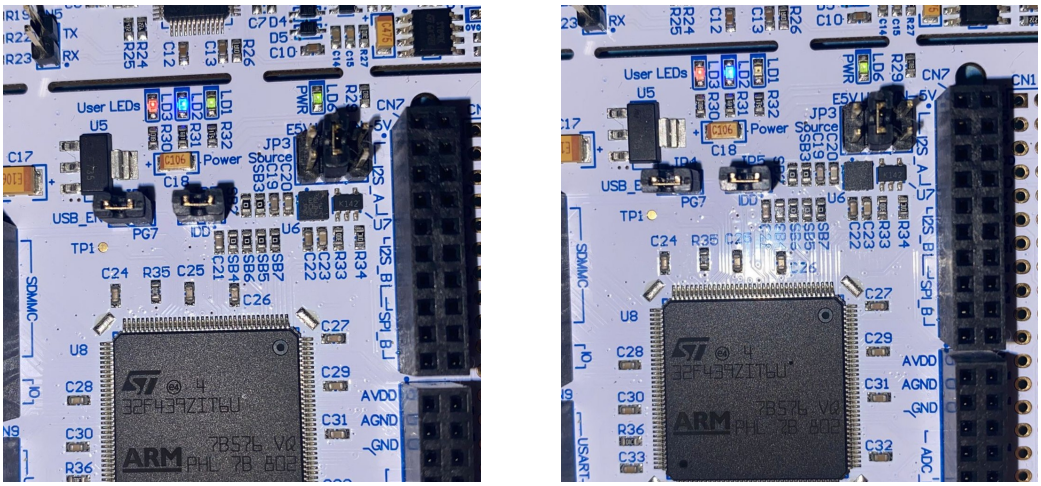
7.3.7 Runtime calibration procedure

Once the model run has started, it is also possible to modify the parameters using the Vehicle Spy interface. In the test, the parameter Par_Led1 is updated to a zero value when the application is running live. The new value is directly written into the memory location of the parameter using the XCP protocol. The following effects can be seen:

- LED1 stops blinking;
- LED2 and LED3 continue blinking;

7.3.8 Hardware observation of calibration effects

The effect of the calibration operation can be directly observed on the physical LEDs of the NUCLEO board.



(a) LED behaviour before modifying the calibration parameter

(b) LED behaviour after modifying Par_Led1

Fig. 7.45: Hardware-side evidence of tuning: physical LED state changes after calibration action

After the parameter is updated, the blinking of LED1 stops, while the blinking of LED2 and LED3 continues, showing that the change of the calibration parameter has been applied correctly to the running embedded application.

7.3.9 Summary and Conclusion

In Experiment 3, the calibration process is demonstrated by using a professional third-party setup, as opposed to the Simulink Monitor & Tune interface. In the experiment, the application generated for the embedded system in Simulink can be successfully integrated with external calibration tools through the XCP protocol over the CAN bus. The runtime monitoring and calibration of the application are performed by Vehicle Spy, which acts as the XCP master.

The tweaking of the parameter Par_Led1 and the observation of the change in the behavior of the LED prove that the calibration process is working correctly and that the application is fully compatible with external ECU calibration tools.

Chapter 8

Conclusions

The thesis aims at investigating the integration of the Universal Measurement and Calibration Protocol (XCP) into the Simulink External Mode, with the objective of revealing the underlying principles that govern the measurement and calibration of control applications running on embedded systems. The thesis undertakes an investigation that combines theoretical analysis of the communication architecture with practical evaluation, focusing on the XCP protocol over Serial and CAN transport layers.

In the context of model-based control system development, control algorithms are usually developed, validated, and then automatically generated into executable code for deployment on target hardware. While the simulation phase is an important first step for the validation of system behavior, the capacity to interact with the running application on target hardware is equally important for the debugging, validation, and calibration of the control application. Measurement allows for the observation of internal variables of the running control application, while calibration allows for the adjustment of parameters during runtime execution of the control application [11]. Simulink External Mode is the communication infrastructure that enables the interaction between the host environment and the running control application.

A key component of this communication infrastructure is the XCP protocol, which represents the industry-standard solution for ECU measurement and calibration procedures. XCP specifies a standard set of services for accessing memory locations, reading measurement signals, and updating calibration parameters of the embedded application [3]. Because of the distinction between the protocol layer and the transport layer, it is possible for the same calibration services to be performed using different communication media, with no changes to the protocol logic. This modular structure of the protocol makes it appropriate for incorporation into heterogeneous development environments and communication infrastructures.

Within this context, the current research aims to investigate the implementation of the Simulink External Mode communication infrastructure, as sup-

ported by the XCP protocol, using a layered communication architecture with distinct protocol services, transport services, and hardware interface components. This layered structure of the communication infrastructure is essential for achieving the portability of the communication services to heterogeneous hardware platforms. The layered structure of the External Mode communication system is reflected by the functional blocks described by Table 8.1, which emphasizes the distinction between protocol, transport, and hardware interface layers.

Layer	**Component**	**Role in the System**
Host Environment	Simulink External Mode Client	Provides the interface that allows the host computer to monitor signals and tune parameters of the running embedded application.
Protocol Layer	XCP Server	Implements the standardized command set defined by the XCP protocol for measurement and calibration operations.
Transport Layer	XCP Transport (Serial / CAN)	Adapts protocol messages to the selected communication medium and manages message transmission between host and target.
Communication Interface	`rtiostream`	Provides the abstraction layer used by External Mode to exchange data between the communication stack and the target hardware drivers.
Hardware Interface	Target communication drivers	Interfaces directly with the physical communication peripherals of the embedded platform.

Table 8.1: Layered architecture of the XCP-based External Mode communication framework

The experimental activities developed in this thesis validate the applicability of the proposed External Mode communication infrastructure for the target system. Indeed, the experiments have shown that the control algorithm's generated signals can be sent to the host environment in real-time, making the visualization of the results possible in the development environment. Moreover, the possibility of changing the tunable parameters defined in the

Simulink model during the runtime execution of the target system, through XCP commands, allows the developer to immediately view the changes in the behavior of the system.

A significant emphasis was placed on the applicability of the proposed transport protocols. As mentioned, the experimental analysis developed in this thesis focuses on the Serial and CAN communication channels. As mentioned, the Serial communication channel is an appropriate solution for the connection between the host environment and the target system, due to its flexibility. On the other hand, the CAN communication channel is an appropriate solution for the automotive environment, considering that the calibration activity must coexist with other messages in the vehicle communication bus [1, 2]. Indeed, the experiments developed in this thesis have shown that the XCP communication infrastructure can operate over the two communication channels, offering the same measurement and calibration services.

Another important aspect that this thesis emphasizes concerns the relationship between the model-based development environment, the Simulink environment, and the third-party calibration environment. As mentioned, the Simulink environment, through the External Mode, allows the developer to interact with the target system. However, the XCP protocol, which is used for the communication infrastructure, is also applicable for third-party calibration tools, which are commonly used in the automotive environment.

Specifically, the ASAM MCD-3 MC standard, which defines the XCP protocol, is applicable for third-party calibration tools, such as the CANape or the Intrepid Vehicle Spy, which interpret the memory structure of the target system through the A2L files [13]. Therefore, the communication infrastructure developed in this thesis is not only applicable for the debugging environment but also for the broader context of the vehicle software development process.

Overall, the results obtained in this study suggest that the integration of model-based design tools with standardized calibration procedures results in an effective framework for the development of embedded systems. Indeed, the interactive communication between the development environment and the executing embedded application, facilitated by the XCP-based External Mode, significantly improves the efficiency of debugging, validation, and calibration activities.

8.1 Future Work

The focus of the future work could be the extension of the portability of the External Mode communication framework to other embedded platforms through the implementation of custom hardware targets. The MATLAB framework offers a dedicated framework that allows developers to configure custom targets through the classes provided within the `target` namespace [16]. This helps in the configuration of the components that facilitate the creation of the External Mode communication between the host and the target platform.

The custom target configuration is initiated through the definition of the target hardware platform through the creation of a `target.Board` object, which defines the target platform and associates it with the processor architecture used in the code generation workflow. Subsequently, the mechanism through which the execution of the target application is initiated must be defined. This is achievable through the `target.ExecutionTool` interface, which offers the essential services to start, stop, and control the execution of the target application.

Another key element of the customization process is the definition of the customization process is the definition of the communication interface used within the External Mode infrastructure. This is achieved through the definition of the `target.CommunicationInterface` object that specifies the communication channel and the implementation of the `rtiostream` APIused for the communication between the host and the target application running in the embedded system. The `rtiostream` layer is the layer that provides the abstraction between the External Mode communication stack and the hardware communication stack of the target system. By implementing the appropriate functions within the `rtiostream` functions, developers can extend the communication infrastructure to support the communication with the embedded systems that do not have the official support packages available.

Based on the definition of the communication interface, the XCP protocol stack may be incorporated into the overall structure of the External Mode environment by configuring the platform abstraction layer and the transport layer, which is responsible for the transmission of the protocol messages via the selected communication interface. These components may then be

consolidated into an overall structure of the External Mode, which enables communication for measurement and calibration purposes with the host application and the target application.

The development of custom targets, therefore, represents a significant extension of the overall structure of the External Mode, allowing developers to utilize the same measurement and calibration environment for a wide range of embedded platforms. By modifying the lower layers of the communication stack, the overall architecture of the protocol stack discussed in this thesis may be generalized, as discussed, with regards to the hardware platform used in the experimental environment, allowing for a more flexible approach to embedded development environments.

Bibliography

[1] International Organization for Standardization, "Road vehicles — Controller Area Network (CAN) — Part 1: Data link layer and physical signalling," 2024. [Online]. Available: https://www.iso.org/standard/86384.html

[2] International Organization for Standardization, "Road vehicles — Controller Area Network (CAN) — Part 2: High-speed physical layer," 2024. [Online]. Available: https://www.iso.org/standard/85120.html

[3] ASAM e.V., "MCD-1 XCP — Universal Measurement and Calibration Protocol," 2023. [Online]. Available: https://www.asam.net/standards/detail/mcd-1-xcp/

[4] ASAM e.V., "MCD-1 CCP — CAN Calibration Protocol," 2005. [Online]. Available: https://www.asam.net/standards/detail/mcd-1-ccp/

[5] ASAM e.V., "MCD-1 XCP — History and Background," 2023. [Online]. Available: https://www.asam.net/standards/detail/mcd-1-xcp/wiki/

[6] Vector Informatik GmbH, "XCP Measurement and Calibration Protocol," 2022. [Online]. Available: https://www.vector.com/int/en/know-how/protocols/xcp-measurement-and-calibration-protocol/

[7] The MathWorks, Inc., "Host-Target Communication with External Mode Simulation," MATLAB R2025b Documentation, 2025. [Online]. Available: https://it.mathworks.com/help/ecoder/armcortexa/ug/set-up-and-use-hosttarget-communication-channel.html

[8] The MathWorks, Inc., "External Mode Simulation by Using XCP Communication," MATLAB R2025b Documentation, 2025. [Online]. Available: https://it.mathworks.com/help/ecoder/ug/external-mode-simulation-with-xcp-communication.html

[9] The MathWorks, Inc., "External Mode Simulation with TCP/IP or Serial Communication," MATLAB R2025b Documentation, 2025. [Online]. Available: https://it.mathworks.com/help/ecoder/ug/external-mode-simulation-with-tcpip-or-serial-communication.html

[10] The MathWorks, Inc., "Create a Transport Layer for TCP/IP or Serial External Mode Communication," MATLAB R2025b Documentation, 2025. [Online]. Available: https://it.mathworks.com/help/ecoder/ug/creating-a-tcp-ip-transport-layer-for-external-communication.html

[11] The MathWorks, Inc., "Calibration," MATLAB R2025b Documentation, 2025. [Online]. Available: https://it.mathworks.com/help/rtw/ug/calibration.html

[12] The MathWorks, Inc., "Get Started with A2L Files," MATLAB R2025b Documentation, 2025. [Online]. Available: https://it.mathworks.com/help/vnt/ug/get-started-with-a2l-files.html

[13] ASAM e.V., "MCD-2 MC — ASAP2 (A2L) File Format," 2022. [Online]. Available: https://www.asam.net/standards/detail/mcd-2-mc/

[14] Vector Informatik GmbH, *XCP — The Standard Protocol for ECU Development*, Version 1.5, 2018. [Online]. Available: https://cdn.vector.com/cms/content/application-areas/ecu-calibration/xcp/XCP_Book_V1.5_EN.pdf

[15] FlexRay Consortium, *FlexRay Communications System Protocol Specification, Version 3.0.1*, 2010. [Online]. Available: https://www.flexray.com

[16] The MathWorks, Inc., "Customize XCP Server Software," MATLAB R2025b Documentation, 2025. [Online]. Available: https://it.mathworks.com/help/rtw/ug/customize-xcp-server-software.html

[17] International Organization for Standardization, "ISO/IEC 7498-1 — Open Systems Interconnection — Basic Reference Model," 1994. [Online]. Available: https://www.iso.org/standard/20269.html