High Level Design (HLD)

Revision 0.25

UWC <Product Version #>

High Level Design

<Name of the kit/ Customer name, if any>

Apr 2021 (Draft)

Approvals

<This section is optional. This section can be used for Customer Collateral. Indicate the stakeholders who have approval authority for this document. Typcially that list should include: Receiving Project’s engineering manager, developing group’s engineering manager, and IXA Reuse Tech Oversight chairperson.>

| Role | Name | Signature | Date |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |

INFORMATION IN THIS DOCUMENT IS PROVIDED IN CONNECTION WITH INTEL® PRODUCTS AND SERVICES. EXCEPT AS PROVIDED IN INTEL’S TERMS AND CONDITIONS OF SALE FOR SUCH PRODUCTS AND SERVICES, INTEL ASSUMES NO LIABILITY WHATSOEVER, AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO SALE AND/OR USE OF INTEL PRODUCTS, AND SERVICES, INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT, OR OTHER INTELLECTUAL PROPERTY RIGHT.

Intel Corporation may have patents or pending patent applications, trademarks, copyrights, or other intellectual property rights that relate to the presented subject matter. The furnishing of documents and other materials and information does not provide any license, express or implied, by estoppel or otherwise, to any such patents, trademarks, copyrights, or other intellectual property rights.

Intel products are not intended for use in medical, life saving, life sustaining, critical control or safety systems, or in nuclear facility applications.

Intel may make changes to specifications and product descriptions at any time, without notice.  
<Include the following text enclosed in curly braces for example designs, interim micro blocks, and   
pre-release/ preliminary documents:>

**{**This document contains information on products and services in the design phase of development. The information here is subject to change without notice. Do not finalize a design with this information.**}**

<Include the following for IA processors, chipsets, and specification updates, if applicable:>

**{**The <product name> may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Current characterized errata are available on request.**}**

\*Third Party trademarks are the property of their respective owners.

Unless otherwise agreed by Intel in writing, you may not remove or alter this notice or any other notice embedded in Materials by Intel or Intel’s suppliers or licensors in any way.

This High Level Design and the software described in it are furnished under license and may only be used or copied in accordance with the terms of the license. The information in this document is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Intel Corporation. Intel Corporation assumes no responsibility or liability for any errors or inaccuracies that may appear in this document or any software that may be provided in association with this document. Except as permitted by such license, no part of this document may be reproduced, stored in a retrieval system, or transmitted in any form or by any means without the express written consent of Intel Corporation.

\* Other brands and names are the property of their respective owners.

Copyright © 2021, Intel Corporation. All Rights Reserved.

Contents

[1 Introduction 13](#_Toc72925840)

[1.1 Purpose and Scope of this Document 13](#_Toc72925841)

[1.2 Terminology 13](#_Toc72925842)

[1.3 Reference Documents 13](#_Toc72925843)

[2 Detailed Requirements Correlation 15](#_Toc72925844)

[3 Assumptions, Dependencies, and Risks 16](#_Toc72925845)

[3.1 Assumptions 16](#_Toc72925846)

[3.2 Dependencies 16](#_Toc72925847)

[3.3 Risks and Open Issues 16](#_Toc72925848)

[4 UWC 17](#_Toc72925849)

[4.1 System Overview 17](#_Toc72925850)

[4.2 UWC S/W Block Diagram 18](#_Toc72925851)

[4.3 Important Terms 19](#_Toc72925852)

[5 Platform Design 21](#_Toc72925853)

[5.1 Platform Overview 21](#_Toc72925854)

[5.1.1 Platform SW Architecture 21](#_Toc72925855)

[5.2 Considerations for real time processing 21](#_Toc72925856)

[5.2.1 System Operations 23](#_Toc72925857)

[5.2.2 Building blocks 23](#_Toc72925858)

[5.3 System Design 24](#_Toc72925859)

[5.3.1 Polling Operation 24](#_Toc72925860)

[5.3.2 On-Demand Read and Write Operations 26](#_Toc72925861)

[5.3.3 Sequence diagram for operations 27](#_Toc72925862)

[5.3.4 Timing Measurement 28](#_Toc72925863)

[5.4 Modbus TCP/RTU Container Application 30](#_Toc72925864)

[5.4.1 Reading configuration files 30](#_Toc72925865)

[5.4.2 On demand operations 30](#_Toc72925866)

[5.4.3 Polling operation 30](#_Toc72925867)

[5.4.4 Response processing in application 31](#_Toc72925868)

[5.4.5 Processing within Modbus Stack (master) 31](#_Toc72925869)

[5.4.6 Hex data conversion and Scaling 34](#_Toc72925870)

[5.5 MQTT Bridge Container 36](#_Toc72925871)

[5.5.1 Design 36](#_Toc72925872)

[5.5.2 Block Diagram 36](#_Toc72925873)

[5.5.3 On demand operations 37](#_Toc72925874)

[5.5.4 Polling operation 37](#_Toc72925875)

[5.6 Threads and Scheduler 37](#_Toc72925876)

[5.7 Timer for polling 41](#_Toc72925877)

[5.7.1 Considerations 41](#_Toc72925878)

[5.7.2 Design 41](#_Toc72925879)

[5.8 Sparkplug-Bridge Container 46](#_Toc72925880)

[5.8.1 Purpose 46](#_Toc72925881)

[5.8.2 SparkPlug Payload Types 46](#_Toc72925882)

[5.8.3 SparkPlug MQTT Topic 47](#_Toc72925883)

[5.8.4 SparkPlug Message Mapping 47](#_Toc72925884)

[5.8.5 Design 51](#_Toc72925885)

[5.9 KPI App Container 61](#_Toc72925886)

[5.9.1 Design 61](#_Toc72925887)

[5.9.2 Block Diagram 61](#_Toc72925888)

[5.9.3 Control Loop Operation 63](#_Toc72925889)

[5.10 Common Library 64](#_Toc72925890)

[5.10.1 Design 64](#_Toc72925891)

[5.11 Message formats 65](#_Toc72925892)

[5.12 Key Design Decisions and Alternatives 65](#_Toc72925893)

[5.12.1 Design Alternative <1, 2, …> 65](#_Toc72925894)

[5.12.2 Open Issues or Unresolved Trade-off Decisions 66](#_Toc72925895)

[5.13 Project Configuration Parameters 66](#_Toc72925896)

[5.13.1 Configuration Options 66](#_Toc72925897)

Figures

[Figure 1: Block Diagram (Ref: Proposal document) 15](#_Toc72925814)

[Figure 2: System Overview 17](#_Toc72925815)

[Figure 3: UWC s/w block diagram 18](#_Toc72925816)

[Figure 4: Architecture with MQTT and Sparkplug-Bridge Support 21](#_Toc72925817)

[Figure 5: Polling operation - TCP 25](#_Toc72925818)

[Figure 6: Polling operation - RTU 25](#_Toc72925819)

[Figure 7: On-Demand operation - TCP 26](#_Toc72925820)

[Figure 8: On-Demand operation - RTU 27](#_Toc72925821)

[Figure 9: Generic sequence diagram flow 28](#_Toc72925822)

[Figure 10: Timing Measurement 30](#_Toc72925823)

[Figure 11: Modbus TCP Stack Init Seq diagram 32](#_Toc72925824)

[Figure 12: Modbus TCP Request Init Seq diagram 32](#_Toc72925825)

[Figure 13: Modbus RTU Stack Init Seq diagram 33](#_Toc72925826)

[Figure 14: Modbus RTU Request Init Seq diagram 34](#_Toc72925827)

[Figure 15: Hex Conversion and Scaling flow chart diagram 35](#_Toc72925828)

[Figure 15: MQTT Bridge Block diagram 37](#_Toc72925829)

[Figure 16: Class diagram for timer mechanism 45](#_Toc72925830)

[Figure 17: Sparkplug-Bridge High Level Flow 46](#_Toc72925831)

[Figure 18: Sparkplug-Bridge Block Diagram 52](#_Toc72925832)

[Figure 19: SCAD-RTU Operation Processing: Internals 54](#_Toc72925833)

[Figure 20: Class diagram Sparkplug-Bridge 57](#_Toc72925834)

[Figure 22: DCMD and DDATA flow for Hex data conversion 60](#_Toc72925835)

[Figure 21: Sequence diagram – KPI App – ZMQ communication 62](#_Toc72925836)

[Figure 22: Sequence diagram - KPI App – MQTT communication 62](#_Toc72925837)

[Figure 23: Block diagram – KPI App – ZMQ communication 63](#_Toc72925838)

[Figure 24: Block diagram – KPI App – ZMQ communication 63](#_Toc72925839)

Tables

[Table 1: List of terms 13](#_Toc68889737)

[Table 2: Reference Documents 13](#_Toc68889738)

[Table 3: Assumptions 16](#_Toc68889739)

[Table 4: Dependencies 16](#_Toc68889740)

[Table 5: Thread Priority 37](#_Toc68889741)

[Table 6: Classes: Timer design 42](#_Toc68889742)

[Table 7: Supported SparkPlug Message Details 46](#_Toc68889743)

[Table 8: Sparkplug-Bridge Actions on connection lost 53](#_Toc68889744)

[Table 9: Classes: Sparkplug-Bridge Design 56](#_Toc68889745)

[Table 10: Classes: Common Library 62](#_Toc68889746)

Revision History

| Revision | Description | Date | Author |
| --- | --- | --- | --- |
| 0.1 | First draft | 14-Nov-2019 | Softdel |
| 0.2 | Second draft | 29-Nov-2019 | Softdel |
| 0.3 | Third draft | 13-Dec-2019 | Softdel |
| 0.4 | Fourth draft | 24-Jan-2020 | Softdel |
| 0.5 | Fifth draft | 20-Feb-2020 | Softdel |
| 0.6a | Work in progress draft | 03-Mar-2020 | Softdel |
| 0.8 | Added RT design and response handdling | 19-Mar-2020 | Softdel |
| 0.9 | Updated various operation flows based on discussion | 04-Apr-2020 | Softdel |
| 0.10 | Added information about timer mechanism (section 5.7)  Updated as per new TCP stack design | 24-Apr-2020 | Softdel |
| 0.11 | Addressed Dean’s review comments  Added a class diagram | 06-May-2020 | Softdel |
| 0.12 | Added introduction for SCADA-RTU  Udpated stack design for combining RTU containers | 16-Jun-2020 | Softdel |
| 0.13 | Added data for SCADA-RTU | 25-Jun-2020 | Softdel |
| 0.14 | Added data for SCADA-RTU | 07-Sep-2020 | Softdel |
| 0.15 | Added data for SCADA-RTU | 21-Sep-2020 | Softdel |
| 0.16 | Added details for SCADA-RTU | 05-Oct-2020 | Softdel |
| 0.17 | Added details about KPI application and common library | 19-Oct-2020 | Softdel |
| 0.18 | Added details about common library | 02-Nov-2020 | Softdel |
| 0.19 | Added more details | 27-Nov-2020 | Softdel |
| 0.21 | Added more details | 22-Dec-2020 | Softdel |
| 0.22 | Added details about Sparkplug template definition handling | 01-Mar-2021 | Softdel |
| 0.23 | Added NCMD handlig | 22-Mar-2021 | Softdel |
| 0.24 | Changed “SCADA-RTU” by “Sparkplug-Bridge” and “MQTT-Export” by “MQTT-Bridge”. | 09-Apr-2021 | Softdel |
| 0.25 | Changes done for Hex data conversion and scaling feature | 26-May-2021 | Softdel |

# Introduction

Intel’s IoTG Intelligent Solutions Group (ISG) wants to build a reference design for gateway platform (software + hardware) for Universal Wellhead and Wellpad Controller (UWC) for Oil and Gas (O&G) industry.

Solution is targeted to address multiple pain areas O&G industry is facing in day-to-day operations. These pain areas are further restricting O&G industry to get benefitted from technology advancements resulting from cloud-based services and applications for business intelligence (BI), analytics, dashboards, etc. There is a need to provide a uniform mechanism to connect, monitor and control various devices in an O&G well-site adhering to real-time nature of the industry.

The Intel® UWC is a reference design for a secured management platform that gives third party application developers an easy access to data from O&G well-site.

## Purpose and Scope of this Document

This document explains design of various modules developed for UWC platform.

## Terminology

Table 1 lists the terms used in this document.

Table : List of terms

| Term | Description |
| --- | --- |
| SAS | Software Architecture Specification |
| UWC | Universal Wellpad (or Wellhead) Controller |
| EIS | Edges Insights Softwae |
| TC | Turtle Creek |
| OEM/ODM | Original Equipment Manufacturer / Original Design Manufacturer |
| O&G | Oil and gas |
| ISG | IoTG Intelliegent Solution Group |
| BI | Business Intelliegence |
| SCADA | Supervisory control and data acquisition |

## Reference Documents

Table : Reference Documents

| Ref | Document Name | Version | File/Location |
| --- | --- | --- | --- |
| [SAS] | <Project Code Name> *SAS* | + | <Insert hyperlink to location here.> |
| [PRD] | <Project Code Name> *PRD* | + | <Insert hyperlink to location here.> |
|  | SparkPlug reference |  | <https://www.eclipse.org/tahu/spec/Sparkplug%20Topic%20Namespace%20and%20State%20ManagementV2.2-with%20appendix%20B%20format%20-%20Eclipse.pdf> |
|  | Sparkplug-Bridge Flows.xlsx | V0.9 |  |
|  |  |  |  |

# Detailed Requirements Correlation

Advancements in drilling operations in the O&G segment have spurred the evolution of complex multi-well pads from simplistic single-well pad fields. Higher data and control requirements are driving the need for a modular, scalable control architecture.

At present, O&G industry is facing certain challenges while in modernizing their existing fields. Examples of these challenges are – devices are not interoperable, difficulty in determining how to get data from current devices and processes, difficulty in maintaining legacy hardware due to availability of parts, security vulnerabilities in current legacy devices, etc.

Intel plans to develop a gateway platform to address above mentioned pain areas thereby providing benefits to O&G industry such as – elimination of vendor lock-in, reduce automation costs, reduce complexity in deployment and maintenance of the solution, increase flexibility (i.e., modularity) and scalability, increase speed of innovation, etc.

Intel plans to achieve above mentioned objectives by decoupling hardware from software, using open architecture, using docker container, modular and incremental development approach, following Intel’s processes and guidelines (for – security, performance, architecture, development, testing and documentation).

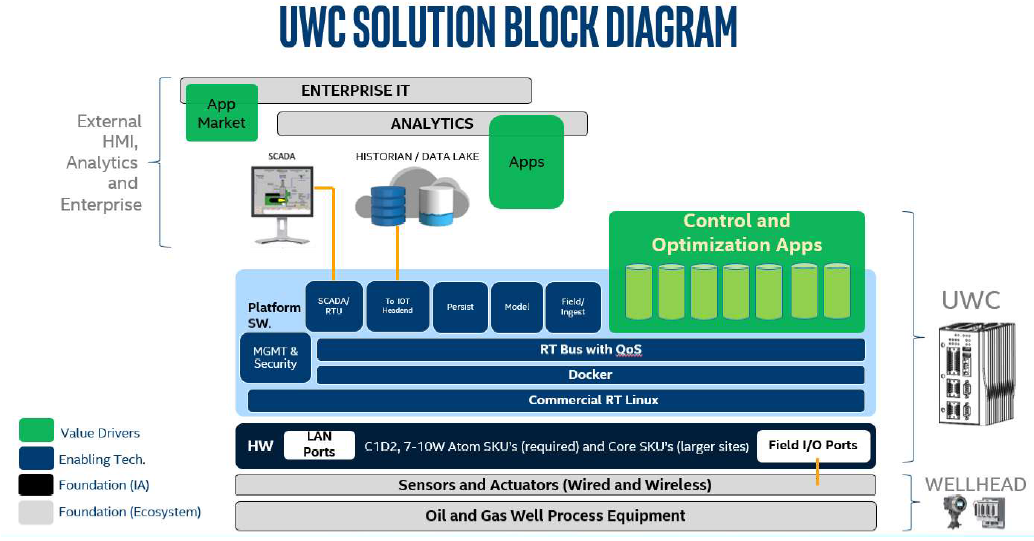


Figure : Block Diagram (Ref: Proposal document)

# Assumptions, Dependencies, and Risks

## Assumptions

To be updated.

Table : Assumptions

| Assumption # | Assumption Name | Detailed Description |
| --- | --- | --- |
| < # > | < Assumption 1> | < Description 1 > |

## Dependencies

To be updated.

Table : Dependencies

| Dependency # | Dependency Name | Detailed Description |
| --- | --- | --- |
| < # > | < Dependency 1> | < Description 1 > |

## Risks and Open Issues

To be updated.

# UWC

## System Overview

The UWC is one of a few initiatives where Oil and Gas industries are working to modernize and IOT-enable their Infrastructure.



Figure : System Overview

## UWC S/W Block Diagram

The objective of the proposed system is to develop a dynamically configurable solution which is compatible with the target deployment environment and communicates with field devices using Modbus protocols (Modbus TCP and Modbus RTU). A common data model (i.e. message format) will be defined to expose platform accessible to client applications. The platform should meet the real time requirement i.e., control loop execution.

The platform should also provide an ability to expose the system to SCADA Master over Modbus TCP.

The platform can also be updated (i.e., component update or configuration update) whenever needed locally.

The platform also provides in interface for SCADA Master using Eclipse Foundation’s SparkPlug standard. The SCADA-RTU module implements this interface. SCADA-RTU is now known as “Sparkplug-Bridge”.

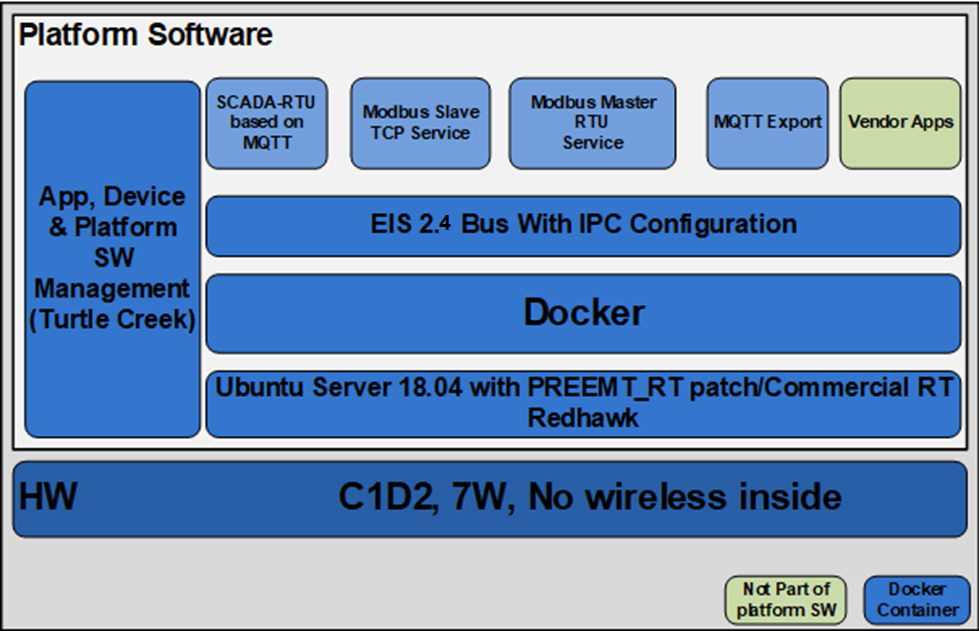


Figure : UWC s/w block diagram

## Important Terms

Following terms are used throughout this document:

**Modbus point** or **Data point** or **point**: It is generically used to denote a Modbus register, coil, etc. present in a Modbus slave.

**RT**: It means realtime operation.

**Non-RT**: It means “non realtime” or “normal priority” operation

**Polling interval**: It means an interval (defined in milliseconds) at which Modbus application container will read data from Modbus points. Polling intervals will be decided by platform user and mentioned in data-points configuration file. Example values for field trial are 250, 500, 1000, etc.

**Cutoff interval**: This interval is in milliseconds and it is associated with every polling interval. It is defined in terms of certain fraction of polling interval, say 90%. Hence, if polling interval is 500 msec, then associated cutoff interval at 90% is 450 msec. Modbus application container will publish a dummy error response at cutoff interval if no response is received from protocol stack for a request sent at polling interval.

**Cutoff interval percentage**: It is the “fraction” used to calculate cutoff interval. Default is 90%.

# Platform Design

This section describes various modules of the system. These are mainly Modbus containers, MQTT-Bridge and Sparkplug-Bridge. There is a specific requirement with respect to realtime behavior of the system. Modbus and MQTT-Bridge modules fall under realtime path. Sparkplug-Bridge does not fall under realtime path.

Next sections describe modules concerned with realtime path. Section 5.8 describes Sparkplug-Bridge module.

Sparkplug-Bridge module was earlier known as SCADA-RTU.

MQTT-Bridge module was earlier known as MQTT-Export.

## Platform Overview

This section describes various modules of the system.

### Platform SW Architecture

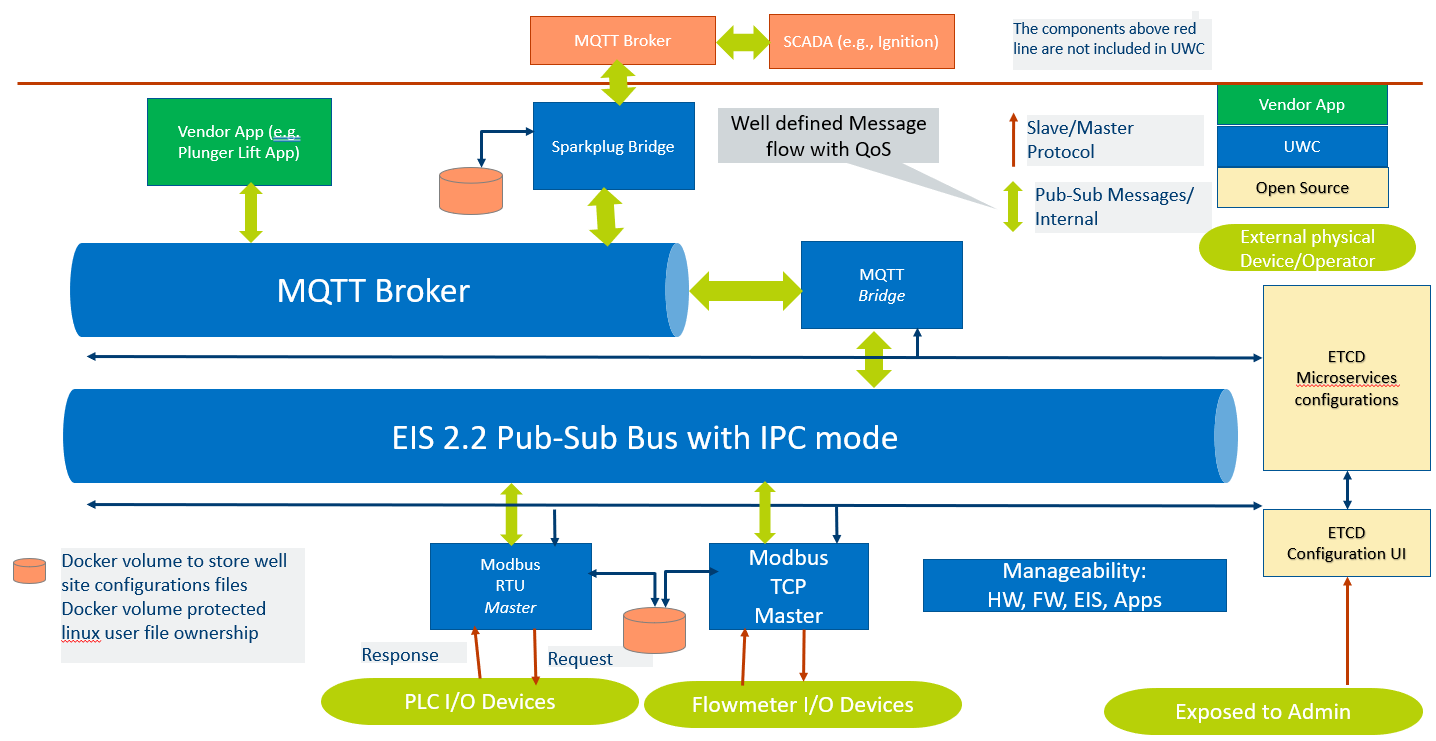


Figure : Architecture with MQTT and Sparkplug-Bridge Support

## Considerations for real time processing

Considering need of oil and gas applications, it is decided to provide soft realtime environment. Hence Ubuntu OS with RT patch will be applied. The next sections describe the design of individual containers (i.e., functional blocks) of UWC platform.

This section explains the overall design thought process.

Following are considerations:

1. At present, only Modbus and MQTT-Bridge modules come under realtime consideration. Sparkplug-Bridge does not come under realtime path.
2. How to decide RT path:
   1. JSON payload for on-demand read and write requests will have a field called “realtime” which denotes whether the request to be executed in realtime mode. Based on this field, RT or Non-RT path for request and response for that instance will be decided.
   2. For polling, datapoints YML will contain a field called “realtime”. Based on this field, RT or Non-RT path for polling of that point will be decided.
3. On ZMQ, different generic topics will be used for RT and Non-RT flow. E.g., “RTPolledData” and “PolledData”.
4. For MQTT,
   1. Topic names are point specific
   2. Topic name will not denote RT or Non-RT differentiation
5. RT and non-RT paths will merge at Modbus stack level. Within Modbus stack, priority queue will be used to prioritize RT processing of different operations.
6. “Good” response will be sent once available.
7. If a response is not received within cutoff interval, then last known response will be sent if available along with bad status.
8. For every request sent to an external Modbus device, a response timeout period will be associated. This response timeout will be configurable at container level. Protocol stack will track response timeout period.
9. In case of response timeout scenario, the protocol stack will inform application container. In such a scenario, request may be retried by the application container. This retry-count will be configurable. The retry will be configurable at operation level and application container will perform the retry after receiving a response timeout notification from protocol tack.
10. No request will be discarded from stack, if a request cannot be sent within polling interval. At the time of sending a new request at next polling interval, if response for previously sent polling request is not received, then application:
    1. will not initiate a new request
    2. publish a bad response with last known good value
11. There will be a generic setting at MQTT Bridge level which defines “QoS” value to be used for different operations when publishing a message on MQTT.
12. IPC mode of ZMQ will be used for all message communication over ZMQ.

### System Operations

UWC as a platform supports following operations. In general, there is a priority associated with these functions which further guides the system to pick up a task to execute when there are number of tasks.

1. On demand write request: Application can send a write request to a Modbus point.
2. On demand read request: Application can send a read request to a Modbus point.
3. RT data polling: UWC platform will give higher priority to these Modbus points while collecting data at set interval.
4. Normal data polling: UWC platform will poll these Modbus points as per set interval.

Priority of these operations is configurable. These priorities define thread priorities in RT and Non-RT flows and priority of messages in priority queue in Modbus stack.

#### Configuration

Following will be configurable for these operations:

Default realtime behavior

Following will be configurable for each realtime and non-realtime type of operations:

* QOS: MQTT-Bridge will use this to publish on MQTT
* Operation priority: Defines priority of said operation among all operations.
* Retries: Defines number of times a request to be retried on Modbus network sent if no response is received.

### Building blocks

Following are building blocks of the design:

1. RT patch: To meet timing requirement, it is recommended to use RT patch on Ubuntu OS.
   1. Scheduler: RT patch allows to select a realtime scheduler.
   2. Priority of threads: There are number of threads in the system executing some unique tasks. A priority should be assigned to these threads for execution considering the overall context of the system and priority of operations.
2. Priority of a message: This will be used inside Modbus stack to prioritize message execution.

## System Design

This section describes flows for operations through different containers. Different ZMQ topics are considered for RT and Non-RT. Hence, with this approach, there will be two generic ZMQ topics for one operation like “PolledData” or “PolledDataRT”.

This helps to separate RT/Non-RT paths.

Modbus container will publish on following topics:

1. Polled data
2. On demand write response
3. On demand read response
4. RT Polled data
5. RT On demand write response
6. RT On demand read response

Similarly, MQTT-Bridge container will publish on following topics:

1. On demand write request
2. On demand read request
3. RT On demand write request
4. RT On demand read request

Subscriber application will have to create corresponding listener threads for these ZMQ topics. Hence, each Modbus container needs to create 4 listeners to receive on-demand read/write RT/Non-RT requests.

MQTT-Bridge needs to create 6 listeners (On-demand read/write RT/Non-RT response and polling data RT/Non-RT) for each Modbus container.

Following diagram shows system level flow for these operations. These diagrams focus on application-level flows.

### Polling Operation

Following is end-to-end flow for Polling operation. First diagram is for TCP communication and later one is for RTU communication.

Legends in following diagram:

* Polling operation – RT flow
* Polling operation – Non-RT flow

In following diagram, the timer thread from Modbus application, identifies queue to be used based presence of “realtime” field in data point YML configuration file for given point being polled at set interval.



Figure : Polling operation - TCP



Figure : Polling operation - RTU

### On-Demand Read and Write Operations

Following is end-to-end flow for On-Demand request (read/write) operation. First diagram is for TCP communication and later one is for RTU communication.

Legends in following diagram:

* On-Demand Write – RT flow
* On-Demand Write – Non-RT flow
* On-Demand Read – RT flow
* On-Demand Read – Non-RT flow

In following diagrams, MQTT listener (i.e., subscriber) in MQTT-Bridge, identifies queue to be used based on topic name (i.e., read or write) and presence of “realtime” field in JSON payload.



Figure : On-Demand operation - TCP



Figure : On-Demand operation - RTU

### Sequence diagram for operations

Following image depicts how communication is initiated by various modules. Please note in following diagram “Application” can be any application developed on top of UWC. Modbus device is any TCP or RTU device present in network.

The diagram shows how on-demand operation and on-demand-write operation can be performed provided all configurations are proper.

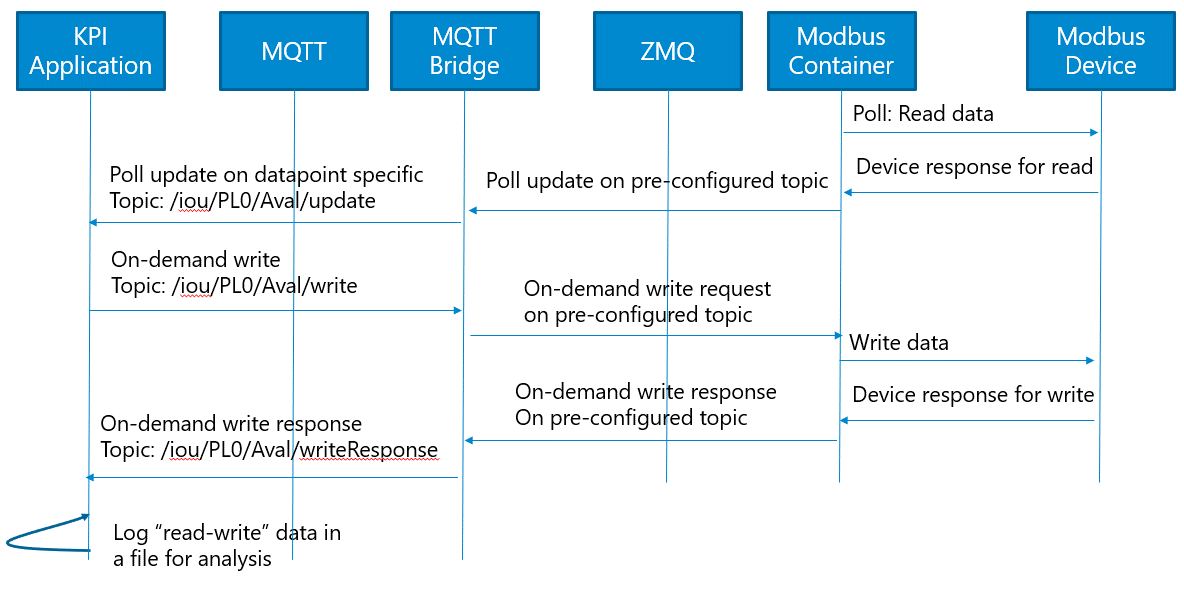


Figure : Generic sequence diagram flow

### Timing Measurement

At various level, timestamps are captured so that inter-module data transition time can be measured. Following diagram shows how these timestamps are measured.

Description of these timestamps is available in user guide.

UWC Platform

Application Container

MQTT

MQTT-Bridge

ZeroMQ

Modbus Driver

Application

Modbus Stack

Modbus Device

Publish topic: /flowmeter/PL0/Flow/write

/flowmeter/PL0/Flow/write

Publish topic1: MQTT\_Export\_WrReq

MQTT\_Export\_WrReq

Modbus Communication:

Request is sent to device

Modbus Communication:

Response is sent by device

Publish topic1: TCP\_WrResp

TCP1\_WrResp

Publish topic: /flowmeter/PL0/Flow/writeResponse

tsMsgRcvdFromMQTT

tsMsgPublishOnEIS

reqRcvdByApp

reqRcvdInStack

reqSentByStack

respRcvdByStack

respPostedByStack

usec

Timestamp captured at given level

/flowmeter/PL0/Flow/writeResponse

Note 1: This is a topic used for non-realtime flow. For realtime flow, a different topic is used.

Figure : Timing Measurement

## Modbus TCP/RTU Container Application

Modbus TCP/RTU container application listens to ZMQ to receive on-demand read/write requests. The application also performs data polling operation as per configured interval and publishes data to ZMQ. The application also publishes response of on-demand request over ZMQ.

Modbus TCP/RTU container application gets information about Modbus slave devices, points, wellsite, etc. from YML configuration files. Modbus container then sets required contexts, listeners, data structures, etc. needed for performing on-demand and polling operations (RT and Non-RT).

At present, Modbus TCP application and Modbus RTU application are handled in different independent containers.

Further, there can be multiple images of these containers. There is a separate Modbus RTU container or each Modbus RTU communication port. Similarly, note that there could be multiple Modbus TCP application containers.

### Reading configuration files

On startup, Modbus TCP/RTU container application parses YML configuration files which are having global configuration, well sites, device and device data points configuration information. This parsed information is updated into data structures.

YAML parser is a 3rd party library. YML files are stored in a data volume mounted in a container.

### On demand operations

For on-demand requests, the application listens on different EII contexts in separate threads for RT and Non-RT flows. Hence there are 4 threads – one RT and Non-RT thread each for on demand write and another for on demand read.

These threads process a received request and send it to stack for further processing.

A priority is assigned to a message based on global configuration for that operation type.

### Polling operation

Polling operation is classified into polling for RT points and polling for non-RT points. For these, separate queue and separate threads are maintained to initiate requests.

There is a timer thread which identifies points to be polled at any given time.

RT/Non-RT polling threads process the received request and sends it to stack for further processing.

A priority is assigned to a message based on global configuration and polling frequency.

### Response processing in application

Stack sends received responses to application. These responses are stored in different queues based on operation type. There is a thread listening on each queue which processes responses. Finally, a JSON is published over ZMQ.

### Processing within Modbus Stack (master)

#### Processing within TCP Stack

Processing within stack is based on priority. For each device, a separate session thread is maintained which takes care of next priority message to execute and manages socket connection for that device. Each device session thread is independent of each other and hence polling requests for multiple devices will happen independently. In such a case, a sequence at which requests are initiated, for given polling interval for multi-device scenario, may vary.

These session threads are created explicitly at the beginning as a part of configuration. A context is associated with each device-specific session thread. The application uses this context while sending a request to a specific device.

For responses, after receiving raw data for a complete response, the associated request and priority are identified. Finally, the matched response is passed to a response dispatcher thread through a priority queue to post the response to application.

A response timeout tracker thread keeps track of time counting. A response timeout action thread checks whether for current time counter, a timeout response needs to be initiated for earlier sent requests.

##### Sequence Diagrams For TCP

Above section explained the use of various threads. This section explains when these threads are created. In following diagrams, for better understanding, “Modbus stack APIs” is shown as an independent entity.

###### Stack initialization

On calling stack init API, at the beginning various threads are created. Please note device specific session thread is not created. This thread will be created when API to get new context is executed by application.

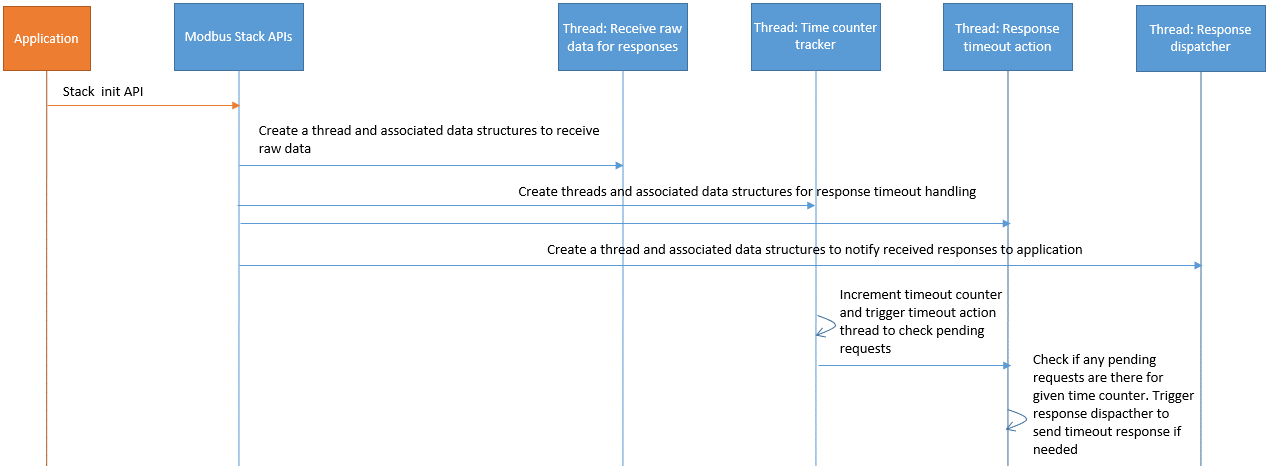


Figure : Modbus TCP Stack Init Seq diagram

###### Getting device context and request processing

Following diagram shows device specific session creation and then request processing.

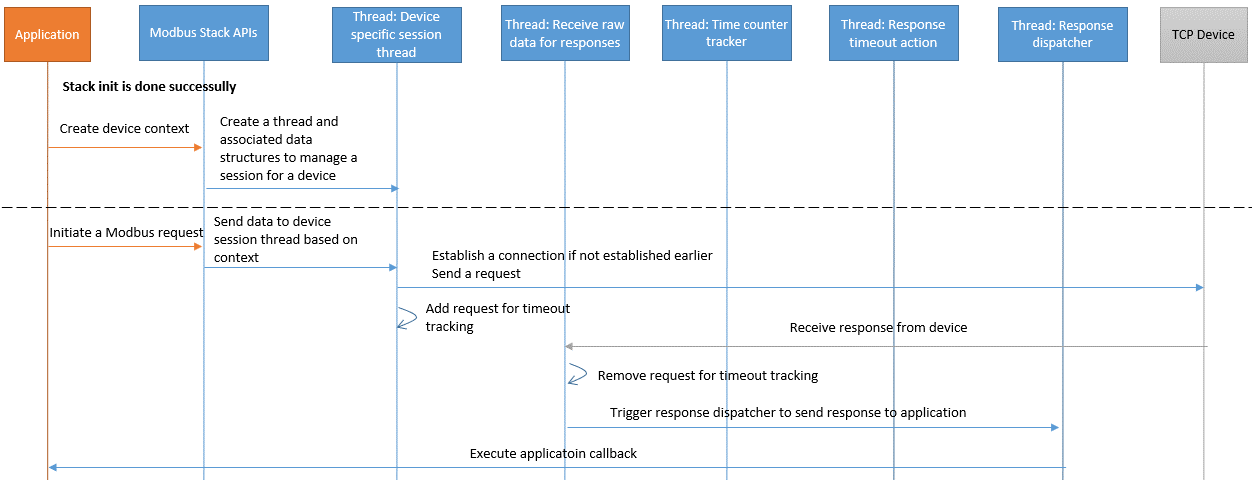


Figure : Modbus TCP Request Init Seq diagram

In case of timeout, no response will be received from device. On timeout completion, timeout action thread will trigger response dispatcher thread to send timeout response.

#### Processing within RTU Stack

In case of Modbus RTU only one request can be sent on one Modbus RTU network at one time and next request cannot be sent unless a response is received, or a timeout occurs. A priority queue is used to decide request to be processed. Response, once received, is passed to a response dispatcher thread through a priority queue to post the response to application.

Communication with multiple RTU networks can be maintained at any given time. A separate thread is created for each different RTU network. Each RTU network communication thread is independent of each other and hence polling requests for multiple devices will happen independently. In such a case, a sequence at which requests are initiated, for given polling interval for multi-RTU-network scenario, may vary.

##### Sequence Diagrams For RTU

Above section explained the use of various threads. This section explains when these threads are created. In following diagrams, for better understanding, “Modbus stack APIs” is shown as an independent entity.

###### Stack initialization

On calling stack init API, at the beginning various threads are created. Please note RTU network specific session thread is not created. This thread will be created when API to get new context is executed by application.

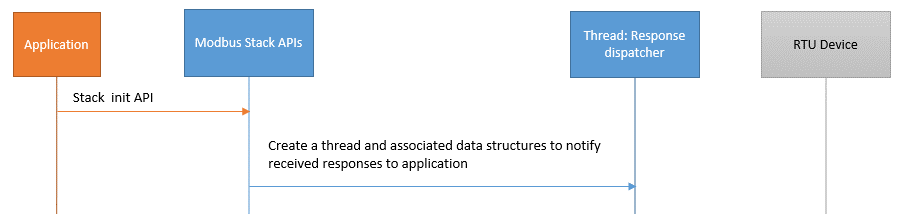


Figure : Modbus RTU Stack Init Seq diagram

###### Getting device context and request processing

Following diagram shows RTU network specific session creation and then request processing.

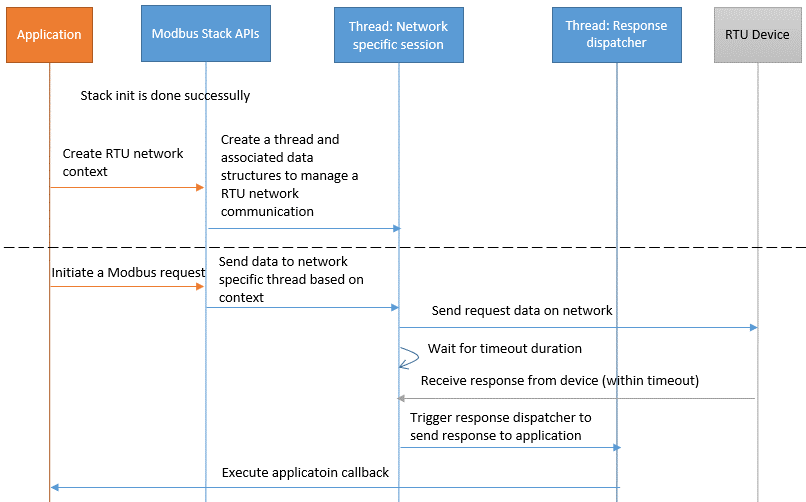


Figure : Modbus RTU Request Init Seq diagram

In case of timeout, no response will be received from device. On timeout completion, response dispatcher thread will be triggered to send timeout response.

### Hex data conversion and Scaling

Modbus devices communicates data in Hex String format. In Modbus containers, this feature converts data received and scales up/down based on the source of the data. Data source can be Modbus devices or ZMQ (E.g., From Ignition). The data in these data sources are handled either in Polling and On-Demand Read or On-Demand Write which are explained in below section.

#### Polling and On-Demand Read

In response to polling and On-Demand Read, Modbus device responds data in Hex string format. Modbus device responded Hex string data is converted into data based on its data type specified in datapoint.yml. This converted data is scaled up using scale factor. Json response message is formed for “value”, “scaledValue”, “datatype”. Sample output message/Json response formed is as below for scale factor as 1

**Output message:** {"**datatype**":"int", “status":"Good","wellhead":"PL0","**scaledValue**":30, **"value":"0x001E”**,"data\_topic":"/flowmeter/PL0/INT16\_MF10/update","metric":"INT16\_MF10", Version":"2.0","realtime":"0”}

For more details, please refer Hex data conversion and scaling flow chart diagram.

Note: Conversion and scaling up is not required for String data type.

#### On-Demand Write

When On-Demand Write message is received from ZMQ, the message can be from Ignition system or vendor App. Processing is done only for the messages received from Ignition system i.e., message having “scaledValue”. Reverse conversion on “scaledValue” is performed using scale factor, data type and then converted into Hex string. This Hex string is sent to Modbus device. When input message is having “value” then don’t perform any processing on it i.e., message is received from vendor App.

**Sample input message:** {"wellhead": "PL0","command": "INT16\_MF10","version": "2.0","realtime": "0","**scaledValue**": 12} realtime":"0”}

For more details, please refer Hex data conversion and scaling flow chart diagram.

Note: Reverse conversion and scaling down is not required for String data type.



Figure : Hex Conversion and Scaling flow chart diagram

#### Scale Factor Evaluation

Scale factor is either read from either datapoint.yml or Global\_Config.yml. First scale factor is read from datapoint.yml. If scale factor is not available in datapoint.yml then read the scale factor from Global\_Config.yml. If scale factor is not available in Global\_Config.yml then use default scale factor 1.0. If evaluation result of scale factor is result in 0 then use default scale factor as 1.0. Zero scale factor gives result nothing.

Scale factor can have positive, negative, decimal values except 0.

## MQTT Bridge Container

MQTT Bridge container application listens to MQTT to receive on-demand read/write requests. The application listens to ZMQ to receive polling data and response of on-demand requests. The application sends data from MQTT to ZMQ and vice versa.

### Design

MQTT Bridge container application sets required contexts, listeners, MQTT subscriber and listener, etc. needed for sending data to and fro for both ZMQ and MQTT.

Topic to be used for publishing data on MQTT is mentioned in a message received from ZMQ.

Separate generic topics are used to publish data on ZMQ for on-demand requests for RT and Non-RT.

MQTT Bridge listens to RT and Non-RT topics from all Modbus containers.

MQTT-Bridge reads global configuration at the start up to define the operation and thread priority.

### Block Diagram

The MQTT client uses port 11883 (secure port) for MQTT communication. For ZMQ IPC is used

For ZMQ to MQTT communication, the MQTT Bridge module receives MQTT topic name as a part of JSON payload. This MQTT topic is used to publish a message on MQTT.

For MQTT to ZMQ communication, the MQTT Bridge module publishes messages on generic topics on EII.



Figure : MQTT Bridge Block diagram

### On demand operations

For on-demand requests, the application listens on MQTT topics, pushes the message on appropriate queue based on topic type (read / write) and message type (RT and Non-RT).

Separate threads are listening on these queues to process the request and to send it to ZMQ using appropriate RT/Non-RT read/write request topic.

Similarly, separate listener threads are created to read data from ZMQ topics to receive responses. These threads process responses and publish it to MQTT. There are separate threads for RT/Non-RT response for read/write.

### Polling operation

Separate listener threads are created to read data from ZMQ topics to receive polling data. These threads process polling data and publish it to MQTT. There are separate threads for RT/Non-RT polling.

## Threads and Scheduler

With RT patch, a different scheduler and priority can be configured for thread processing. This section describes the priorities considered. For Sparkplug-Bridge, threads are executed in normal mode. Following section applies to Modbus and MQTT-Bridge modules.

Table : Thread Priority

Following table shows priorities that can be considered for various threads throughout the system.

Scheduler to be used: RR

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Modbus container | | | MQTT-Bridge container | | | Priority | Comments |
|  | Module / Thread | Count | Total | Module / Thread | Count | Total |
| RT On-demand write | Req processing | 1 | 2 | Req processing | 1 | 1 + N | 60 | Priority for RT operations: On-demand write > On-demand read > Polling |
| Resp processing | 1 | Resp processing | 1 x Number of Modbus containers (N) |
| RT On-demand read | Req processing | 1 | 2 | Req processing | 1 | 1 + N | 55 |
| Resp processing | 1 | Resp processing | 1 x Number of Modbus containers |
| RT Polling | Req init | 1 | 3 |  |  |  | 50 |
| Resp handling Cutoff handling | 1 1 | Polled data handler | 1 x Number of Modbus containers | N |
| Non-RT On-demand write | Req processing | 1 | 2 | Req processing | 1 | 1 + N | 15 | Priority for Non-RT operations: On-demand write > On-demand read > Polling |
| Resp processing | 1 | Resp processing | 1 x Number of Modbus containers |
| Non-RT On-demand read | Req processing | 1 | 2 | Req processing | 1 | 1 + N | 10 |
| Resp processing | 1 | Resp processing | 1 x Number of Modbus containers |
| Non-RT Polling | Req init | 1 | 3 |  |  |  | 5 |
| Resp handling Cutoff handling | 1 1 | Polled data handler | 1 x Number of Modbus containers | N |
| Timer | Timer thread | 1 | 1 |  |  |  | 65 | Timer tick is given highest priority to ensure polling interval is measured correctly. |
| Modbus RTU stack | RTU communication | 1 x RTU Network count (R) | 1 + R |  |  |  | 30 | Stack operations are given higher priority than Non-RT operations and lower than RT operations |
| Resp dispatcher | 1 |  |  |  |
| Modbus TCP stack | Device session | 1 x TCP device count (M) | 3 + M |  |  |  | 30 | Stack operations are given higher priority than Non-RT operations and lower than RT operations.  We can have a range defined here. E.g. 30-40 |
| Resp dispatcher | 1 |  |  |  |
| Resp timeout tracker | 1 |  |  |  |
| Resp timeout action | 1 |  |  |  |

## Timer for polling

This section describes the mechanism used for implementing a timer needed for polling operation.

### Considerations

There will be number of polling intervals and associated cutoff intervals defined in milli-seconds. Following are considerations:

A different timer for each of polling and cutoff interval, might result in too many timers which need to be monitored. This approach could also lead to having more number of threads.

Looking at above mentioned scenario, a single timer is proposed for overall polling operation.

#### Timer selection

Over the period, following approaches were tried/evaluated for timer for polling operation in Modbus application container. Please note timer operation in each container (e.g., Modbus TCP, Modbus RTU) is independent of each other.

* Timer from boost library: This timer was not accurate and there was some lag in getting tick. This was first used in sprint 1.
* Interrupt based timer: This mechanism gave accurate ticks but for each tick a separate interrupt was generated. Interrupts caused system to become unstable. This was first used in sprint 5.
* usleep based timer: Linux API “usleep” was used to sleep between timer intervals. Sometimes, this mechanism did not give required accuracy. This was used in sprint 9.
* timerfd based timer: This is based on Linux’s timerfd\_create() mechanism. A consistency was not observed during testing with this timer. This was used in sprint 11.
* High resolution timer: High resolution timer upto nanosecond precision can be implemented using Linux using APIs “clock\_nanosleep()” function. This is implemented in sprint 12.

High resolution timer is selected as it gave consistent and accurate results.

### Design

This section provides details of design for timer.

#### Timer algorithm

Following is a high-level algorithm for timer using high resolution timer:

Step 1: Get current monotonic time (say T1) using clock\_gettime()

Step 2: Add interval (say In1) after which function needs to be triggered.

T2 = T1 + In1

Step 3: Use function clock\_nanosleep() with monotonic clock and TIMER\_ABSTIME flag to sleep till T2 time is reached.

Step 4: After sleep, perform operations related to polling

Step 5: Assign T2 to T1 i.e., T1 = T2

Step 6: Go to step 2.

#### Polling timer algorithm

There will be number of polling intervals and associated cutoff intervals defined in milli-seconds. e.g., 250, 500, 1000, 2000, etc. Polling requests should be initiated at the polling interval.

As presented earlier, a single timer is proposed to handle all different polling and cutoff intervals. Following is a high-level algorithm for this based on a concept of a counter:

Step 1: Calculate greatest common factor (say INTERVAL) of all polling and cutoff intervals. Smallest possible common factor i.e., INTERVAL is 1 msec.

Step 2: Identify the highest value of polling interval (say MAX\_INTERVAL)

Step 3: Set counter value for each polling interval assuming counter starts from 0.

E.g., For polling interval of 250, counter value will be 250.

For polling interval of 1000, counter value will be 1000.

Step 4: Set counter tracker (say CTR\_TRACKER) to 0.

Step 5: Start timer with sleep-interval as INTERVAL

Step 6: After sleep, increment counter:

CTR\_TRACKER = CTR\_TRACKER + INTERVAL

Step 7: Check if any of polling interval’s counter value is equal to CTR\_TRACKER

Step 7a.1: If yes, trigger request initiation for eligible points

Step 7a.2: Set next counter value by adding polling interval to current counter value.

If counter value for polling interval exceeds MAX\_INTERVAL, then roll over.

Step 8: If CTR\_TRACKER value exceeds MAX\_INTERVAL, then roll over.

Step 9: Go to step 6.

#### Timer design

On a high level, following classes participate in managing polling operation. Other classes are not listed for simplicity.

Table : Classes: Timer design

|  |  |
| --- | --- |
| Class name | Description |
| CTimeRecord | This class maintains a list of points to be polled against certain polling interval.  This class maintains following:   * Polling interval * Cutoff interval * List of RT and Non-RT points against the polling interval |
| StPollingTracker | This structure maintains polling interval, reference to CTimeRecord object and next counter to be used for polling. |
| CTimeMapper | This class maintains a list of CTimeRecord objects and tracks polling for these objects.  This class maintains following:   * List of CTimeRecord objects * List of StPollingTracker objects |
| CRequestInitiator | This class maintains threads, data structures to initiate a Modbus request as per polling interval and to send error responses at cutoff interval. This class maintains data for both RT and Non-RT threads and operations.  The class also maintains a data structure to map a response with an initiated polling request. |
| CRefDataForPolling | This class represents a point for which polling is needed. It has following:   * a reference of other class representing actual unique point * last polling status * current polling timestamp * data structure containing Modbus request information * reference of ZMQ context used for publishing data of this point |

Following is a high-level class diagram which depicts the association between these classes. For simplicity, only relevant class members are shown in following diagram.



Figure : Class diagram for timer mechanism

## Sparkplug-Bridge Container

Sparkplug-Bridge provides an interface to SCADA Master using Eclipse Foundation’s SparkPlug standard. Sparkplug-Bridge will publish data over MQTT for SCADA Master. MQTT Topics and Payload standardization is must for Sparkplug-Bridge interoperability with multi-vendor SCADA Master.

UWC selected Eclipse Foundation’s Sparkplug B specification for interoperability of Sparkplug-Bridge implementation with multi-vendor SCADA Master (e.g., Ignition, Autosol eACM, Wonderware, Cloud based deployments).

Sparkplug-Bridge does not fall under realtime path.

### Purpose

This module exposes the data on the platform to an external, centralized, master system for the SCADA:

* Data from base UWC platform
* Mechanism to expose data from Apps running on UWC

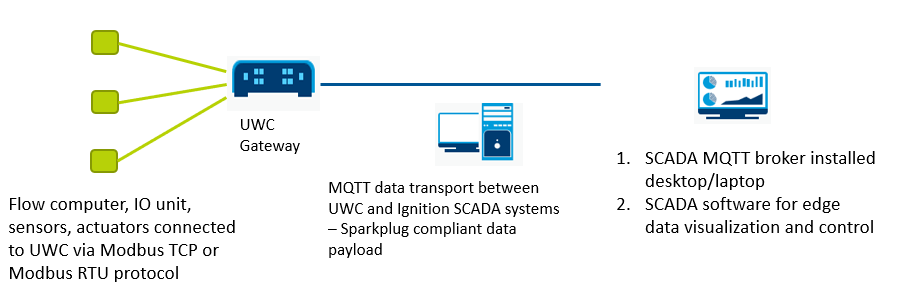


Figure : Sparkplug-Bridge High Level Flow

### SparkPlug Payload Types

Following are the payload types defined by SparkPlug standard:

* Edge node birth (NBIRTH)
* Edge node death (NDEATH)
* Device birth (DBIRTH)
* Device death (DDEATH)
* Node data Message (NDATA)
* Node command (NCMD)
* Device data Message (DDATA)
* Device command (DCMD)
* SCADA Master application stat message (STATE)

Refer SparkPlug standard for more information.

### SparkPlug MQTT Topic

Following is the topic format:

spBv1.0/group\_id/message\_type/edge\_node\_id/[device\_id]

“group\_id” and “edge\_node\_id” are specified under global-configuration. For “edge\_node\_id”, to make it unique, MAC id of the gateway is appended. If the MAC-id is absent, then a fixed string “00” is used.

Device id is mentioned like “device\_id-device\_group\_id” e.g. iou-PL0.

The value of the “group\_id”, “edge\_node\_id” and “device\_id” can be valid UTF-8 alphanumeric string. The string shall not use the reserved characters of ‘+’ (plus), ‘/’ (forward slash), and ‘#’ (number sign).

### SparkPlug Message Mapping

This section details the mapping for currently supported messages (listed below).

Table : Supported SparkPlug Message Details

|  |  |  |
| --- | --- | --- |
| Message Type | Support for real device | Support for virtual device (apps) |
| NBIRTH | Supported. This is an edge level message. | |
| NDEATH | Supported. This is an edge level message. | |
| DBIRTH | Supported. Data is taken from YML file. | Supported. Vendor app should publish data on “BIRTH” topic. |
| DDATA | Supported. Data from Poll-update messages is taken to determine change in data for publishing a DDATA message | Supported using RBE (Report By Exception). Vendor app should publish data on “DATA” topic. |
| DCMD | Supported. A corresponding On-Demand-Write request message is published on internal MQTT for other UWC containers to process a request. | Supported. A corresponding CMD message is published on internal MQTT for vendor app. |
| DDEATH | Supported. Data from Poll-update messages is taken to determine change in data for publishing a DDEATH message in case of error scenarios | Supported. Vendor app should publish data on “DEATH” topic. |
| NDATA | Not Supported |  |
| NCMD | Supported for “Node Control/Rebirth” control | |
| STATE | Not Supported | |

#### NBIRTH

This message is published on start-up after establishing a connection with MQTT broker.

1. On start-up, Sparkplug-Bridge reads data about Modbus devices (both TCP and RTU) from YML files.
2. It identifies all datapoints YML files. E.g., iou\_datapoints.yml, flowmeter\_datapoints.yml, etc.
3. Each YML file corresponds to one SparkPlug template definition.
   1. Name of YML file is used as a name of SparkPlug template.
   2. Version of YML file is used as a version of SparkPlug template.
   3. Protocol type is added as a parameter for this template definition.
   4. Metrics in YML file are used as metrics inside template.
      1. For each of these metrics, following fields from YML file are mapped as properties:
         1. Pollinterval
         2. Realtime

#### NDEATH

This message is published by MQTT broker when a connection with Sparkplug-Bridge is broken. For this to work, a proper Will payload is set by Sparkplug-Bridge while establishing a connection with MQTT broker.

#### DBIRTH

##### DBIRTH for real devices

This message is published on start-up after NBIRTH is sent.

1. On start-up, Sparkplug-Bridge reads data about Modbus devices (both TCP and RTU) from YML files.
2. A DBIRTH message is prepared for each Modbus device and published over MQTT.
3. For a Modbus device, there is one major metric of type template.
   1. A corresponding datapoints YML file is used as a template for a Modbus device definition.
   2. Device id is name of template type metric.
   3. Reference of template is corresponding datapoints YML file.
4. Each datapoint from a device is mapped as a metric inside template in DBIRTH message for respective device.
5. For each of these metrics, following fields from YML file are mapped as properties:
   1. Pollinterval
   2. Realtime
   3. Scale factor

Please note, in current design, when the gateway starts, the Sparkplug-Bridge module publishes DBIRTH messages for Modbus devices with default empty values for metrics. This is a static message published based on data available in YML files.

##### DBIRTH for virtual devices (i.e. Apps)

DBIRTH message is published in following cases when a BIRTH message is received.

1. It is a first BIRTH message for the virtual device.
2. There is a change in metrics reported earlier in 2 BIRTH messages for the virtual device.
3. Previously a DDEATH message is published and then a BIRTH message with no changes is received for the device.

Please refer document “Sparkplug-Bridge Flows.xlsx” for detailed flows.

#### DDATA

##### DDATA for real devices

DDATA message is published in following cases when a polling update message is received for a real device:

1. There is a change in metric scaled value compared to the one reported earlier and there is no error scenario.

Note that SparkPlug template format is used for publishing DDATA message.

##### DDATA for virtual devices (i.e. Apps)

DDATA message is published in following cases when a DATA message is received.

1. There is a change in metric value compared to the one reported earlier in BIRTH or DATA message.

DBIRTH message is published in following cases when a DATA message is received.

1. Previously a DDEATH message is published and then a DATA message is received for the device.

Please refer document “Sparkplug-Bridge Flows.xlsx” for detailed flows.

#### DDEATH

##### DDEATH for real devices

DDATA message is published in following cases when a polling update message is received for a real device having error code:

1. If error code is a specific one which means device is not reachable, then a DDEATH message is published for a rea device.
2. For a real device, once a DDEATH message is published, then device poll updates are monitored to identify a change in state of a device. When a proper data is received, a DBIRTH message is sent for the real device.

##### DDEATH for virtual devices (i.e. Apps)

DDEATH message is published in following cases:

1. When a DEATH message is received for a vendor app
2. When a new BIRTH message is received for a virtual device

Please refer document “Sparkplug-Bridge Flows.xlsx” for detailed flows.

#### TemplateDef

This is a special message used by vendor app to provide definition of a Sparkplug Template i.e., UDT to Sparkplug-Bridge. On receiving this message, Sparkplug-Bridge identifies that a new UDT definition is received. UDT definitions are sent to SCADA master as a part of NBIRTH message. Hence, on receiving UDT definition, the messages are published in the sequence: NDEATH and then NBIRTH.

##### Using UDT in BIRTH message

A metric can be of type UDT. These metrics are published in BIRTH message. The UDT metric should provide reference of UDT which was earlier published using TemplateDef message.

#### START\_BIRTH\_PROCESS

This is a special message used by Sparkplug-Bridge to inform vendor apps that Sparkplug-Bridge module is functional now. On receiving this message, a vendor app shall send all Sparkplug Template definitions it intends to use, using TemplateDef message. The vendor app shall also send app device details using BIRTH message.

#### NCMD

SCADA master can initiate NCMD message asking for rebirth of node. In this particular case, the node shall publish NBIRTH, DBIRTH messages.

### Design

Following is a high level block diagram.

External MQTT connecting to SCADA Master

Sparkplug-Bridge

MQTT Handler (uses Paho Library)

SparkPlug Standard handler (using Tahu Library)

Sparkplug-Bridge/ EdgeNode functions:

1. Build network info
2. Establish MQTT connection
3. Publish messages

YML Parser

Internal MQTT connecting to vendor apps and MQTT-Bridge

Figure : Sparkplug-Bridge Block Diagram

In terms of SparkPlug, UWC gateway is an edge-node. The Sparkplug-Bridge module implements the SparkPlug standard.

The Tahu library from Eclipse provides APIs required to build SparkPlug data structures, message encoding / decoding, etc. For testing purpose, SparkPlug encoded messages can be viewed in MQTT.Fx, a generic 3rd party MQTT client.

The Paho library is used to publish messages on MQTT broker. Please note this is an external MQTT broker. This is set up by SparkPlug compliant SCADA Master. There could be multiple MQTT brokers. At present, this design only considers one MQTT broker.

Sparkplug-Bridge module performs following:

1. Establishes connection with MQTT broker. Configures Will payload such that MQTT broker can publish a proper NDEATH message when a connection is broken.
2. Publishes NBIRTH message.
3. Builds network information using YML file. This information is used to build DBIRTH messages.
4. Publishes DBIRTH message for each Modbus device based on information available in YML file.
5. Subscribes to:
   1. virtual device messages (BIRTH/#, DATA/#, DEATH/#, TemplateDef)
   2. real device poll update messages
6. Publishes START\_BIRTH\_PROCESS to vendor app.
7. For vendor app (i.e., virtual devices):
   1. Publishes DBIRTH message when appropriate on receiving BIRTH message from app
   2. Publishes DDEATH message when appropriate on receiving DEATH or BIRTH message from app
   3. Publishes DDATA message when appropriate on receiving DATA message from app
8. For real devices:
   1. Publishes a DDATA message when there is a change in value of the metric
   2. Publishes a DDEATH message when an error code meaning “unreachable device” is received
   3. Publishes a DBIRTH message when “unreachable device” changes status to “reachable”

#### Operation Processing

Following is the design approach:

Sparkplug-Bridge

External MQTT connecting to SCADA Master

Thread: To process a message from queue

Queue to store (BIRTH, DATA, DEATH) messages

Internal MQTT connecting to vendor apps and MQTT-Bridge

Thread: To process a message from queue

A

Queue to store DCMD messages

Figure : SCAD-RTU Operation Processing: Internals

Messages from internal MQTT are stored in a queue. There is a thread which listens to this queue and processes messages received from queue to publish data on external MQTT in SparkPlug format.

Similarly, there is a queue which stores data received from external MQTT. These are DCMD messages. There is a thread which listens on this queue and publishes CMD or On-Demand-Write operation.

Please note all messages are processed sequentially from given queue for following reasons:

1. The status of devices to be published depends on the last status published which in turn depends on some other factors. E.g., If last message published was DDEATH, then next message to be published should be DBIRTH.
2. RBE for DDATA should be calculated with respect to last reported value.

MQTT connection handling: Following threads are created to process notifications from MQTT broker. Please note these threads are created for both internal MQTT handler and external MQTT handler:

Table : Sparkplug-Bridge Actions on connection lost

|  |  |  |
| --- | --- | --- |
| Notification | Internal MQTT Handler | SCADA MQTT Handler |
| Connection established | * Subscribes to topics * If the connection is established, then a signal is sent to a thread\* which publishes DBIRTH message on SCADA-MQTT connection for all up devices | * Subscribes to topics * Publishes NBIRTH and DBITH messages |
| Disconnected | * Monitors for given time (1 min) to see whether a connection is established. * If the connection is not established, then a signal is sent to a thread\*\* which publishes DDEATH messages on SCADA-MQTT connection | - |
| Message received | Above block diagram shows these threads. It processes device update messages. | Above block diagram shows these threads. It processes DCMD messages. |

\*There is a separate thread which sends DBIRTH messages for known online devices whenever internal MQTT connection is established.

\*\*There is a separate thread which sends DDEATH messages whenever internal MQTT connection is lost for given timeout.

#### Class Diagram

Following is a conceptual class diagram:



Figure : Class diagram Sparkplug-Bridge

Table : Classes: Sparkplug-Bridge Design

|  |  |
| --- | --- |
| Class name | Description |
| CSparkPlugDev | This class represents a real Modbus device or a virtual app subclass. It contains a reference to classes derived from YML file parsing. For real Modbus device, it refers CUniqueDataDevice.  This class maintains following:   * List of metrics * Last published device/message status |
| CIfMetric | This class is a base class for a Sparkplug metric of base type or template type. |
| CMetric | This class is derived from CIfMetric and it represents a real Modbus device data-point or a datapoint from a virtual app subclass. It contains a reference to classes derived from YML file parsing. For real Modbus device datapoint, it refers CUniqueDataPoint.  This class maintains following:   * Value * Last timestamp for this property |
| CUDT | This class is derived from CIfMetric and it represents a Sparkplug Template. It represents both – UDT definition and UDT instance. This class implements a composite design pattern.  The class maintains Metric list of UDT. |
| CSparkPlugUDTManager | This class manages UDT definitions. It provides interface to parse UDT definition messages, to access UDT definitions.  This is a singleton class.  This class maintains UDT using 2 primary keys – major key is UDT-name, minor key is UDT version.  These fields are present in TemplateDef message used to define a UDT.  Each UDT instance refers to these fields to uniquely identify UDT definition to be referred. |
| CValObj | This class represents a value of a metric. The value can be of different datatypes.  This class maintains following:   * Datatype * Value as per datatype |
| CVendorApp | This class represents a vendor app. This class contains a list of subclasses represented as CSparkPlugDev. |
| CVendorAppList | This class maintains list of all CVendorApp objects. |
| CSparkPlugDevManager | This class maintains all CSparkPlugDev objects. This class parses all messages received from internal and external MQTT and accordingly updates CSparkPlugDev objects and decides action to be taken. E.g., when a DEATH message is received for a CVendorApp objects, this class identifies that DDEATH message needs to be published for all CSparkPlugDev objects representing subclass associated with that vendor app.  This is a singleton class.  This class also implements a factory method to determine which metric type to create whether base type or template type. |

#### Hex data conversion support

##### DDATA

For supporting hex data conversion feature, Sparkplug-Bridge has to identify whether message has received from real device or virtual device (Vendor App).

When message is received from real device, the message is having value as well as scaledValue. Sparkplug-Bridge has to publish DDATA with only scaledValue to external broker. Hex data conversion is applicable for only real devices and not for virtual devices.

When message is received from Vendor App, the message is having only value. Sparkplug has to publish DDATA with value to external broker. No change in the existing flow.

For more detail, please refer below data flow diagram.

##### DCMD

First Sparkplug-Bridge has to identify whether received DCMD is for real device or virtual device (Vendor App).

When received DCMD is for real device then change value key with scaledValue key and publish it to the internal broker.

When received DCMD is for virtual device then don’t do any change, keep existing implementation as it is.

For more detail, please refer below data flow diagram.

##### DBIRTH

In DBIRTH message for real device, added new field “scale” for each of the metrics.



Figure : DCMD and DDATA flow for Hex data conversion

Legends: Green lines indicates existing implementation is having change. Red lines indicate existing implementation don’t have change.

## KPI App Container

“KPI Application” represents a sample application which is provided to depict how one can develop an application on UWC platform. This is a simple application which demonstrates how “single input, single output” control loop can be implemented.

A control loop is executed continuously to monitor certain parameter and the adjust other parameters. Thus, a control loop consists of one read operation and one write operation. In this sample application, polling mechanism of UWC platform is used to receive values of parameters as per polling interval. The application uses “on-demand-write” operation on receiving data from polling.

This KPI application can either be executed based on MQTT communication or based on ZMQ communication based on communication.

The KPI Application also logs all data received as a part of control loop application in a log file. This data can be used for measuring performance of the system.

### Design

For ZMQ-based communication, the KPI application

1. uses generic topics:
   1. uses RT or Non-RT topics (based on configuration) for getting updates from polling and write responses
   2. uses RT or Non-RT topics (based on configuration) for sending write requests
2. sets required ZMQ contexts, ZMQ listeners, etc.
3. sets value of realtime field in write request payload depending on RT/NON-RT communication

For MQTT-based communication, the KPI application

1. uses point-specific topics
2. sets required MQTT listeners, etc.
3. sets value of realtime field in write request payload depending on RT/NON-RT communication

### Block Diagram

The MQTT client uses port 1883 (non-secure port) for MQTT communication. For ZMQ IPC is used

Flow diagram for ZMQ based communication:



Figure : Sequence diagram – KPI App – ZMQ communication

Flow diagram for MQTT based communication:

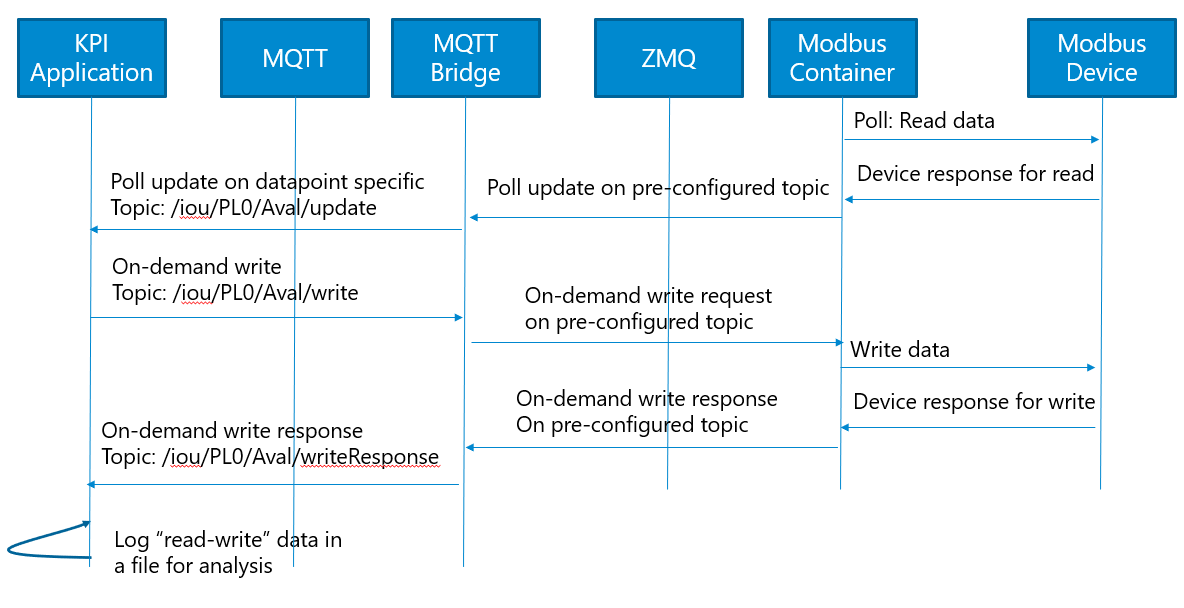


Figure : Sequence diagram - KPI App – MQTT communication

Block diagram for ZMQ:

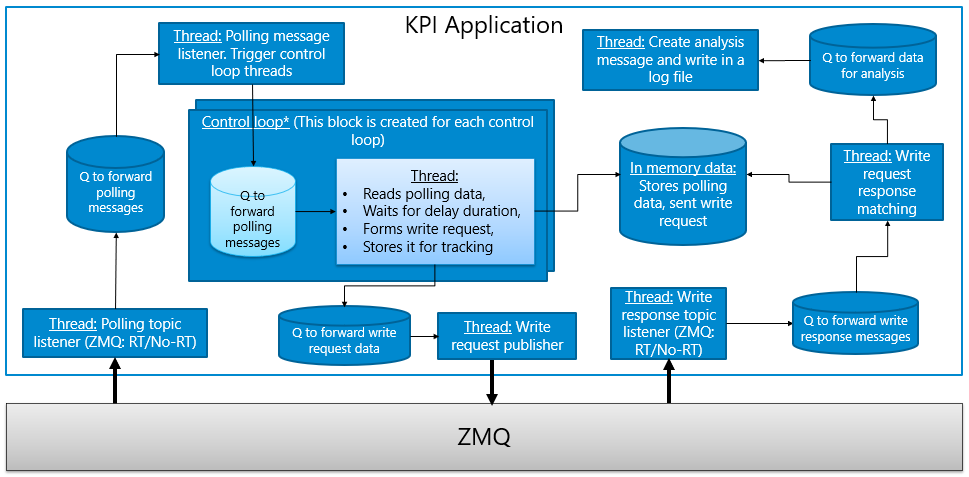


Figure : Block diagram – KPI App – ZMQ communication

Block diagram for MQTT:

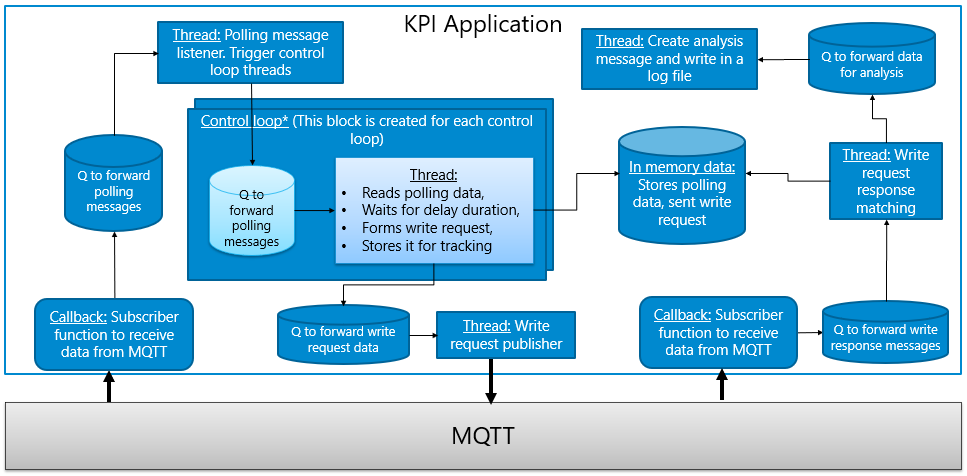


Figure : Block diagram – KPI App – ZMQ communication

### Control Loop Operation

In control loop, once a polling data is received, before sending a write request a pre-configured delay needs to be added. For this, a separate thread is created for each control loop. In that thread, once a polled data is received, a write request is created after adding a specific delay. This created request write is sent to another thread to post it to ZMQ or MQTT.

The write request data and associated polling data are stored in a map for creating a log-record once a write response is received.

If new polling data is received before receiving write response of previously initiated write request, then an error is logged.

#### ZMQ based communication

For ZMQ based communication, separate threads are created for each generic topic based on RT or Non-RT configuration on polling data and write response. These threads will store data in queue for further processing.

#### MQTT based communication

For MQTT based communication, callback functions are triggered by MQTT library. These threads will store data in queue for further processing.

#### Thread operations

Control loop operation handling design is independent of ZMQ and MQTT communication.

There is one queue and one thread which receives polling data for all points. This thread posts data to individual control loop handlers.

Each control loop handler has one queue and one thread. The thread performs the operation as described above.

There is one queue and one thread which receives write response data for all points. This thread does request response matching and sends data to another thread for logging into a log file for analysis.

## Common Library

UWC platform contains different containers. These modules perform certain operations which are similar in nature. For this, a common library is built which is used by all modules.

### Design

Following are the details of classes/namespaces present in this common library:

Table : Classes: Common Library

|  |  |
| --- | --- |
| Class name | Description |
| CLogger | Implements logging mechanism. It uses log4cpp library. This class is used by all modules to create logging data. |
| network\_info | It is a namespace which hold number of classes used to store data parsed from YML files about wellhead, device, datapoint, etc. |
| globalConfig | It is a namespace which holds data about Global\_Config.yml file |
| CfgManager | This class is used to store handles of ETCD like config manager and env config |
| CcommonEnvManager | This class is used to store information about dev mode and app name. |
| EnvironmentInfo | This class is used to parse environment variables and store the values. It provides interface to access these environment variables on need basis. |
| CQueueHandler | This class implements a queue to store data and inform others when a data is received. |
| zmq\_handler | It is a namespace which contains data structures which are used to store ZMQ related contexts for pub and sub mechanism. |
| CMQTTPubSubClient | It is a class which implements a publisher and subscriber for MQTT |
| CMQTTBaseHandler | It is a base class used to handle callbacks from MQTT. It provides a base framework to operate MQTT handling. This class is then extended by MQTT handlers in different modules. |

## Message formats

Refer user guide for message formats.

## Key Design Decisions and Alternatives

Sample text

### Design Alternative <1, 2, …>

Sample text

### Open Issues or Unresolved Trade-off Decisions

Sample text.

## Project Configuration Parameters

### Configuration Options

Sample text