Universal Internet of Things Connector (UIC) Specification

UIC® software specification for embedded hardware products

Version 1.0
March 12, 2018

Revision History

Revision	Date	Change	Author
0.1	July 2, 2017		J. Uhlig
0.2	July 24, 2017	Changes to the introduction	W. Eisenbarth
0.3	July 27, 2017	Added Technical Interface Description	J. Uhlig
0.4	August 20, 2017	Added Program Flow Sequence Diagram	J. Uhlig
0.5	August 21, 2017	Added logo and text correction §1	W. Eisenbarth
		ADDED: EDM Callback Flowchart CHANGED: Startup Flowchart CHAGNED Interface and UIC Startup PseudoCode • CloudAgent CommandAction • EDM SetXXXCAllbackMehtods • UniversallotConnector to set Callback mehtods	J. Uhlig
06	January 28, 2018	Adapt to changes resulting from the UIC.NET reference implementation	J. Uhlig
0.7	February 5, 2018	Clean up of structure and text review	W.Eisenbarth
0.8	February 13,2018	Text review	C. Rebmann
0.85	February 19, 2018	Proofread	M. Swiecicki
0.9	February 22, 2018	Finalization	J. Uhlig
0.95	February 25, 2018	Finalization	J. Uhlig
0.96	March 9,2018	Proofread / Clean up of open comments Release version generated	C. Rebmann
0.97	March 12, 2018	Updated Sequence diagramm start up and shut down	J. Uhlig
1.0	March 26, 2018	Final release by SGET board	W.Eisenbarth

Table of Content

Table of Figures

1. Introduction

1.1 Legal

Note that some content of this SGET document may be legally protected by patent rights not held by SGET. SGET is not obligated to identify the parts of this specification that require licensing or another legitimization. The contents of this SGET document are advisory only. Users of SGET documents are responsible for protecting themselves against liability for infringement of patents. All content and information within this document are subject to change without prior notice.

SGET provides no warranty regarding this SGET document or any other information contained herein and hereby expressly disclaims any implied warranties of merchantability or fitness for any particular purpose with regard to any of the foregoing. SGET assumes no liability for any damages incurred directly or indirectly from any technical or typographical errors or omissions contained herein or for discrepancies between the product and this SGET document. In no event shall SGET be liable for any incidental, consequential, special, or exemplary damages, whether based on tort, contract or otherwise, arising out of or relating to this SGET document or any other information contained herein or the use thereof.

1.2 Copyright Notice

SGET has made every attempt to ensure that the information in this document is accurate yet the information contained within is supplied “as-is”.

1.3 Trademarks

Microsoft®, Windows® and Azure® are registered trademarks of Microsoft Corporation. All product names and logos are property of their owners.

1.4 Special Word Usage

Mandatory features are indicated by the use of the word ***“shall”***.

Recommended features are indicated by the use of the word ***“should”***.

Optional features are indicated by the use of the word ***“may”***.

1.5 Purpose of this Document

The Universal IoT Connector (UIC) is a software specification to define a common approach for embedded hardware supplier in providing access to the sensor level devices onboard or thru additional interfaces. A main goal is also the independence of supplier specific software interfaces to achieve a high compatibility between different vendors to the benefit of the application programmers.

1.6 Applicable Documents and Standards

- ***eAPI*** (embedded API) PICMG.org Specification
  https://www.picmg.org/wp-content/uploads/COM_EAPI_R1_0.pdf
- ***Smart Mobility Architecture Hardware Specification***, V 2.0, June 2, 2016 © SGET (Standardization Group for Embedded Technologies) www.sget.org
  https://www.sget.org/fileadmin/user_upload/SMARC_DG_V2.pdf
- ***Qseven Specification***, V2.1, March 23, 2016 © SGET
  (Standardization Group for Embedded Technologies) www.sget.org
  https://www.sget.org/fileadmin/file_management/SDT02/Qseven-Spec_2.1.pdf

1.7 Statement of Compliance

Statements of compliance with this specification take the form specified in the SGET Policies and Procedures for Specification Development:

“This product provides software support for SGET® UIC Revision 1.0” may be referenced by any SGET member for their complying products and shall inform customers about the existence of the support as described in the actual specification.

Because the specification provides for many recommended features beyond the mandatory minimum set and a wide range of performance capabilities, more complete descriptions of product compliance by the respective vendor is encouraged.

2. General

The Internet of Things (IoT) and digital transformation put huge pressure on many traditional companies and of course on hardware and embedded system manufacturers by shifting business cases towards service models. Even though software is nothing new for these companies, new requirements of the product life cycle result in massive challenges in evolving the software stack and its interoperability.

On the other side conventional IT practitioners are forced to interact with new smart assets to offer feature rich end-to-end solutions to their customers. Although software is nothing new to them either, writing embedded software and accessing hardware peripherals is a complete new and unknown dimension.

Bringing both worlds together is the challenge. This is the integration of internet and cloud based IT with the distributed and local operational technology (OT), i.e. smart embedded systems.

With this specification, the SGeT defines a software architecture pattern that connects the embedded system environment with the Internet universe from an embedded system perspective. The core of the Universal Internet of Things Connector (UIC) architecture is a conceptual design of the involved software modules without any implementation details, but with the description of the interaction between these modules by defining common interfaces. This approach focusses on the processes and the interaction of well documented building blocks of IoT appliances. The data model as one aspect of the interface description is kept as lean and generic as possible to allow for future flexibility.

The following two principals guided all design decisions in the creation process of this specification:

- **Interoperability:** All parts, modules and subsystems interoperate with each other in a seamless way without vendor specific knowledge. In that way IT engineers can easily leverage different types and versions of embedded systems in a generic way. Likewise, embedded system integrators are able to utilize several applications with the same code base.
- **Interchangeability:** Another goal of interoperability is to avoid a vendor lock-in. Customers shall have the freedom to try new opportunities in new business models without the risk of investing in the wrong technology.

2.1 Anatomy of an Internet of Things appliance

A state of the art smart appliance consists of a new and complex technology stack. It covers the local peripherals (sensors and actors) at the one end and the mobile applications at the other end. The stack contains at least the following layers:

- Sensors and actors which control the environment on site
- Embedded systems as the local brain that control the local installation including different responsibilities such as
    - Hardware driver and interface access
    - Operating system
    - Communication with other systems and the IP-driven internet universe
    - Local user interface (UI)

- Connectivity
- Device- and Data-Management as part of a backend IT system
- Application Server
- Web Portals, mobile apps and rich client applications

All parts need to interact with each other to build a successful solution. As this specification is to be read in the context of embedded systems the technology stack can be distilled to the model in Figure 2.1, which shows the generic modules of the UIC-architecture.

Figure 2.1 generic UIC modules

Hardware Module: From the embedded system perspective as part of the whole end-to-end solution the main responsibility is to control the local **hardware**. This has been the main business for decades.

Application Module: A new requirement is to represent the embedded system in the internet and IP universe as a digital twin. In general, this will be an **application server,** either **on premises** or in the **cloud**.

Configuration Module: Another crucial requirement is to allow easy configuration of the embedded system. Optimally the same software base can be 100% reused and all customization is done through configuration.

UIC Module: Finally, we need a local orchestration authority that manages the other three modules. It is the core module and the heart of this specification: The **Universal IoT Connector – SGET UIC**.

Up to this point we only talked about the modules as generic building blocks of such a software architecture. As the scope of this specification is not to define the implementation details of these modules, the final implementation is up to the vendors and service providers. To reach the goal of full interoperability and interchangeability, defining the interaction between these modules is critical. Therefore, Figure 2.2 identifies the interfaces and the functional names of the generic modules.

The embedded system manufacturer will mainly take care of the EDM implementation while the communication agent may be maintained by the application provider. A system integrator will most likely implement the program-flow logic in the UIC module and the configuration mechanisms within the project agent.

In this way, an ecosystem may emerge and several libraries and implementations for the different modules may be built.

Figure 2.2 UIC interface overview

2.2 Module Overview (refer to Figure 2.2)

Please keep in mind that the implementation details are not part of this specification and are only used to explain the UIC architecture and interfaces.

2.2.1 UIC module - Universal Internet of Things Connector

Any software needs a authority which takes control over the progrm and information flow of the solution. This local brain in this UIC specification is the UIC module. IT manages the interaction between all the software modules on the embedded system. It is the local authority which starts up the solution, controls the regular processing and monitoring tasks and shuts down the program. This module should not depend on any implementation detail of the other UIC modules to guarantee a maximum of flexibility. In general the UIC is responsible for:

- Providing the starting point of the execution,
- managing the initialization of the local EDMs and ,
- monitoring of configured data points
- dispatch commands received through the Communication Agent to the appropriate EDM

2.2.2 EDM

An Embedded Driver Module (EDM) is the actual driver for a connected peripheral which, in general, will be sensors or actors. In more sophisticated scenarios the embedded system may be connected to other complex mashines via I/O buses, e.g. Fieldbuses.

The challenge is the heterogenous landscape of mashine communication protocols and the plurality of the peripherals. The EDM may need to interpret the raw data from the peripherals to some meaningful value. A good example for such a case is a multi-sensor peripheral read over an I^2C bus. Instead of providing the pure register value, the EDM has to preprocess the raw data, extract the actual information like temperature or a distance and publishes these values to the UIC.

2.2.3 Communication Agent

The Communication Agent is responsible to communicate with an application server or any other kind of connected system. These systems are resposnible to transfrom the edge data into valuable information for the customer and end user. In general, we see application servers in the IP universe but also machine to machine communication can be found in the market. Foremost the communication consists of sending sensor data to the customer application server and receiving data from it.

Another important task of the Communication Agent is to initialize the application server. The more complex IoT solutions get the more information is needed by the application server about the local appliances to provide the best possible User Experience to the end user. That’s why the Communication Agent shall initialize the application server on start up. In general, it sends detailed information about the actual configuration and local available datapoints. The application cloud can then use this information to change the user experience accordingly.

2.2.4 Project Agent

Using a generic code base over several appliances and products has always been an important goal in software engineering. The adjustments to a concrete system are realized over different sets of settings. In the past, these configurations were implemented in proprietary mechanisms including binary or xml files. In this SGET UIC specification, a common project configuration is needed that describes which EDMs to use and how to set up the Communication Agent.

Remote configuration is getting more and more important. Different approaches form a remote configuration server to a local file can be seen in the market. That's why the handling of the configuration data is encapsulated into the Project Agent. The agent keeps full control over the configuration and can load the project data from a local file (in JSON format), a remote server or both.

In general, the project configuration tells the embedded system:

- which peripheral to utilize,
- how to process the raw data and
- when/how often to forward the data to the Communication Agent.

2.3 Interface Overview

The UIC is the local brain of the embedded software stack. As a dispatcher, the UIC orchestrates the other modules by calling them actively and does not wait for a notification. Consequently, the UIC is the only module that knows about and interacts with the other modules. Therefore, the interfaces let the UIC module handle the different implementation of the Project Agent and Communication Agent without the UIC having to know any of their internal implementation details. The necessary interfaces are:

- **Project Agent Interface:** The most important function of this interface is a Get function for the current configuration.
- **EDM Interface:** Provides functions to the UIC module for initializing and disposing the EDM as well as reading sensor data and / or writing actor data to the connected peripheral.
- **Communication Agent Interface:** Provides functions to the UIC module for publishing data to the application host and receiving data from remote systems.

2.4 Reference UIC Solution Architecture

To sum up the previous explanation the following picture illustrates the architecture of an IoT solution based on a UIC compliant embedded system. Well defined responsibilities and the standardized interfaces lead to a large reusable code base across different products, vendors and solutions.

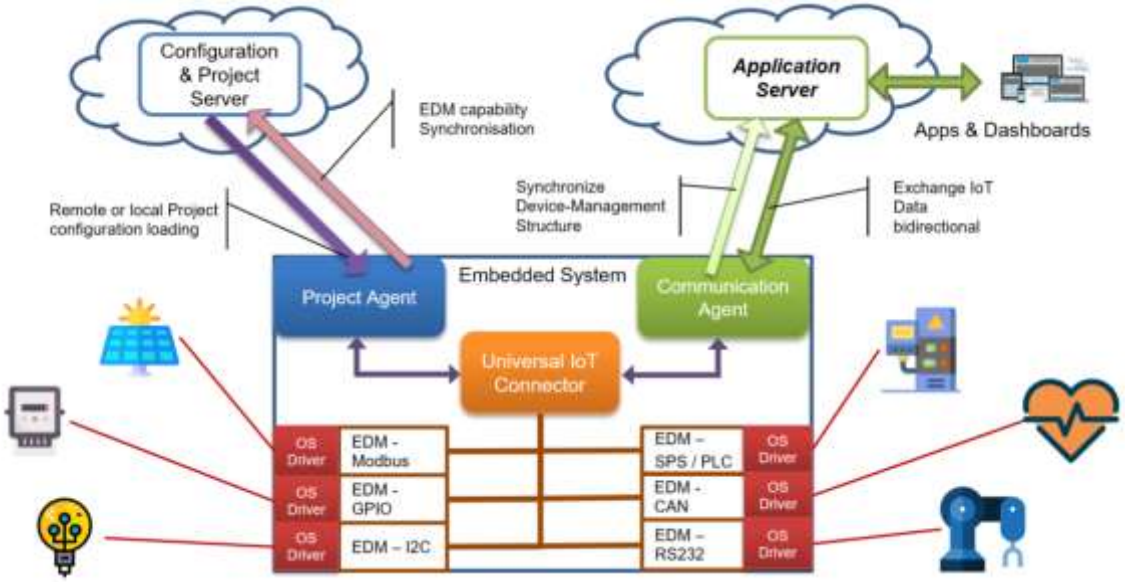
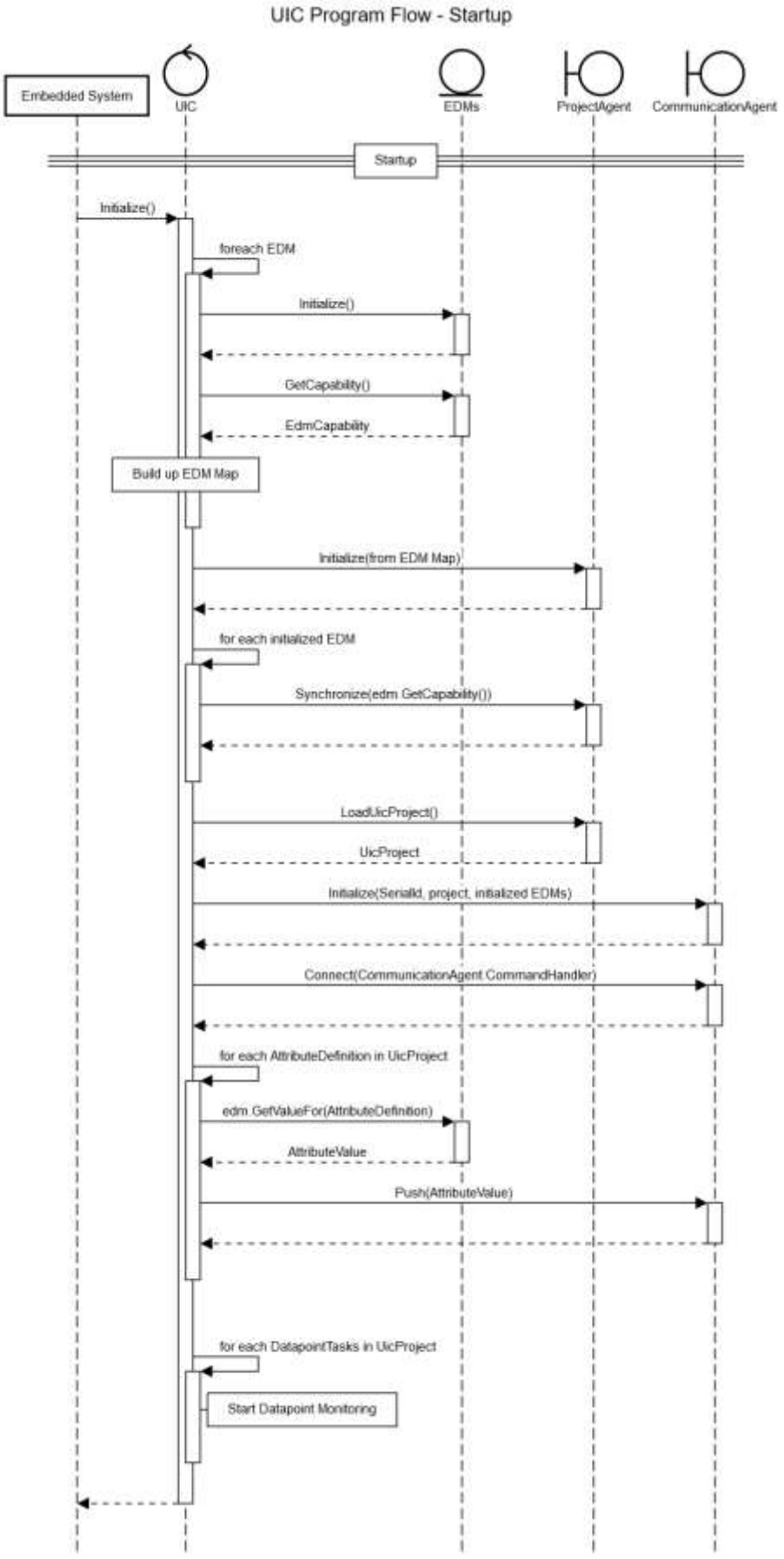


Figure 2.3 UIC application reference architecture model

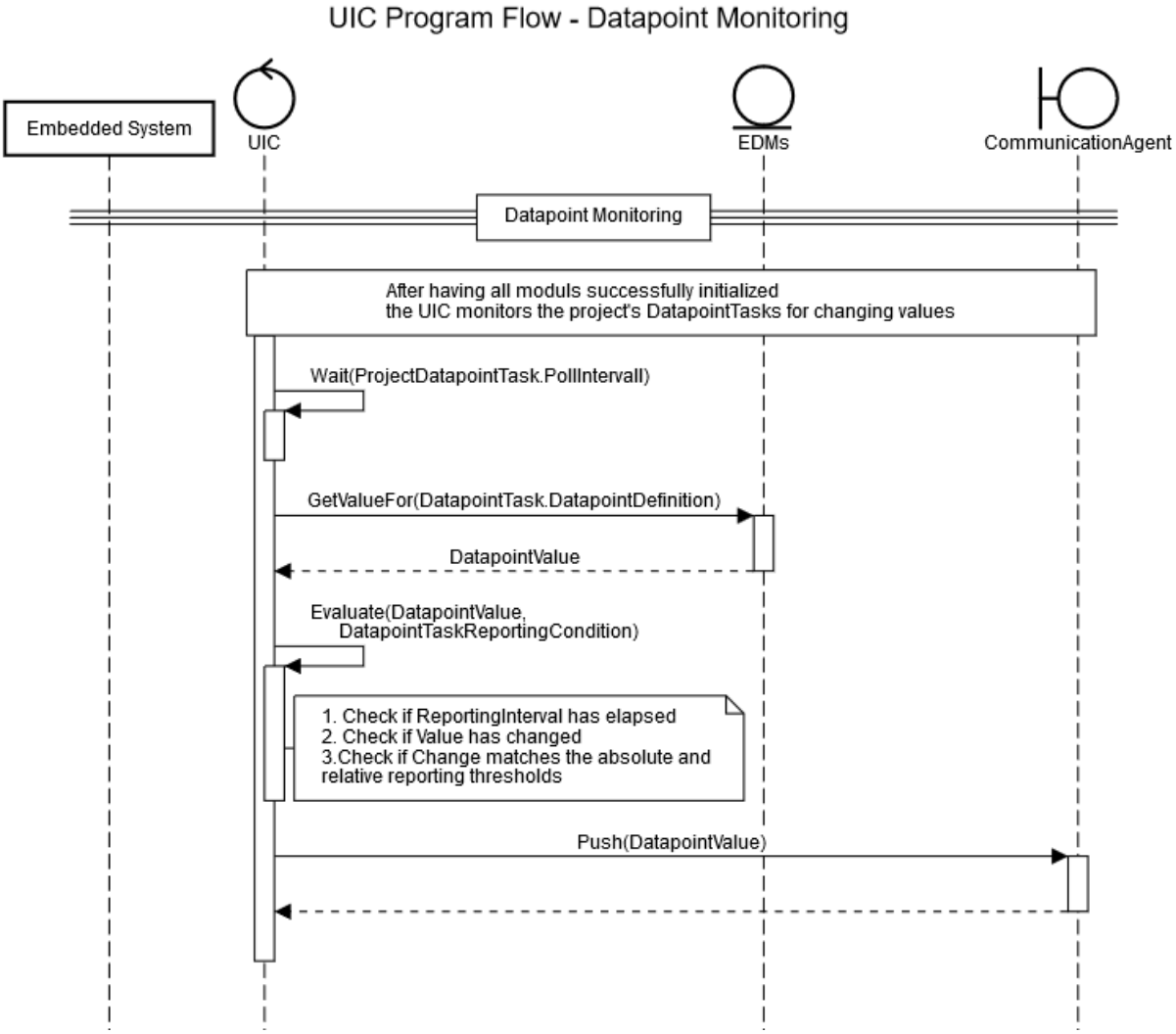
2.5 General UIC Program Flow

To understand the nature of the UIC interface design it is important to understand the programm and information flow of an IoT solution. In the following you will find sequence diagrams that illustrate the lifecycle of an IoT software solution on an embedded system from the “Startup”-phase to the ongoing “Monitoring”-phase until the “Shut Down”-phase.

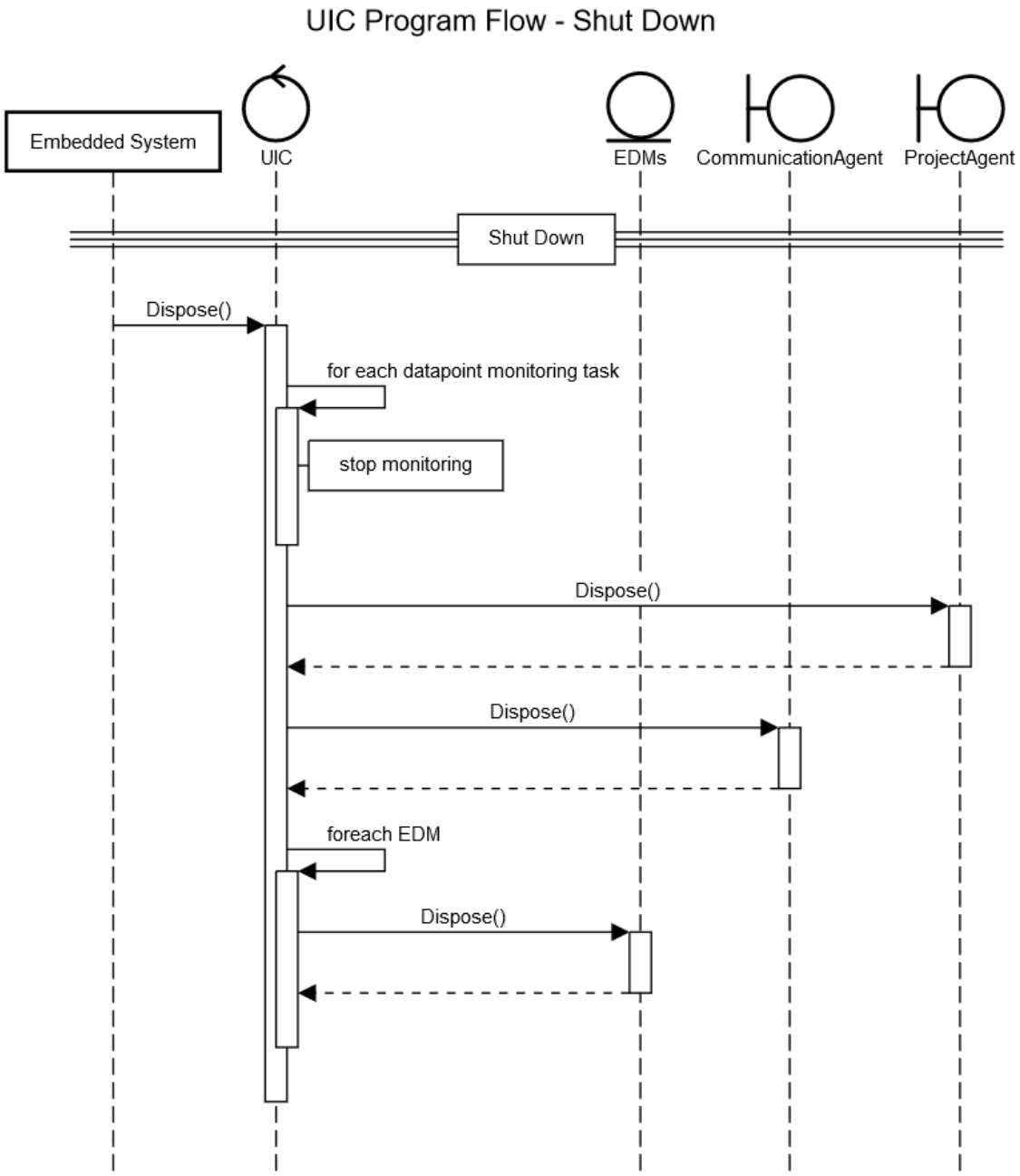
2.5.1 Startup Sequence



2.5.2 Datapoint Monitoring



2.5.3 Shut Down



3. EDM Interface

An Embedded Driver Module (EDM) is the actual driver for a connected peripheral which, in general, will be sensors or actors. In more sophisticated scenarios the embedded system may be connected to other complex mashines via I/O buses, e.g. Fieldbuses.

Beacause of the heterogenous landscape of mashine communication protocols and the plurality of the peripherals the EDM shall be designed to wrap the communication with a specific peripheral and translate the raw data according to the interfaces of the UIC architecture. This will result in many small and specific EDMs.

In general, the EDM gets invoked by the UIC module for providing current value for the supported data points. It should not actively call the UIC and it must not have any dependency to a Communication or Project Agent.

For that purpose a communicatoin Agent must provide the following interface to the UIC module.

```
interface EmbeddedDriverModule {
     EdmIdentifier Identifier { get; }

     void Initialize();
     void Dispose();

     EdmCapability GetCapability();

     DatapointValue GetValueFor(DatapointDefinition datapoint);
     AttributeValue GetValueFor(AttributeDefinition attribute);

     bool Handle(Command command);

     void SetDatapointCallback(ProjectDatapointTask, callback<DatapointValue>);
     void SetAttributeCallback(AttributeDefinition, callback<AttributeValue>);
 }
```

Initialize and **Dispose** are called on boot time and shut down.

The **EdmIdentifier** uniquely identifies the EDM.

GetCapability returns the EdmCapability data structure that describes the capability of the EDM in the form of definitions (command, attributes and data points)

The **GetValueFor** function accepts a definition and returns the corresponding actual value.

The **HandleCommand** is called by the UIC when a command is received from the Communication Agent.

The **Set…Callback** functions are optional and can be used by the UIC to enable the EDM to notify the UIC about changing attributes or data point values proactively.

The appropriate sub components are defined as follows:

```
interface EdmCapability
{
    EdmIdentifier GetIdentifier { get; }
    CommandDefinition[] CommandDefinitios { get; }
    AttribtueDefinition[] AttribtueDefinitions { get; }
    DatapointDefinition[] DatapointDefinitions { get; }
}


interface EdmIdentifier
{
    Guid Id { get; }
    string Uri { get; }
}


interface AttributeDefinition
{
    Guid Id { get; }
    string Label { get; }
    string Description { get; }
    UicDataType DataType { get; }
    string Uri { get; }

}

interface DatapointDefinition
{
    Guid Id { get; }
    string Uri { get; }
    string Label { get; }
    string Description { get; }
    UicDataType DataType { get; }
}


internal interface CommandDefinition
{
    Guid Id { get; }
    string Uri { get; }
    string Label { get; }
    string Command { get; }
    UicDataType DataType { get; }
    string[] Tags { get; }
    DatapointDefinition RelatedDatapoint { get; }
}


internal interface DatapointValue
{
    DatapointDefinition Definition { get; }
    object Value { get; }
}


interface AttributeValue
{
    AttributeDefinition Definition { get;  }
    object Value { get; }
}
```

4. Communication Agent Interface

The genuine responisbility of the Communication Agent is to wrap the communication with an application server or any other kind of connected system. In general, these systems collect data from edge devices and transform the raw data into valuable information for the customer and end user.

In general, the Communication Agent gets invoked by the UIC module for pushing new data points to the application server. It should only actively call the UIC if data was received from the application server. It must not have any dependency to a concrete EDM.

For that purpose a communicatoin Agent must provide the following interface to the UIC module.

```
interface CommunicationAgent
    {
        void Connect(Action<Command> commandHandler);
        void Dispose();
        void Initialize(serialId, UicProject, EmbeddedDriverModule[] edms);
        void Push(DatapointValue value);
        void Push(IEnumerable<DatapointValue> values);
        void Push(AttributeValue value);
        void Push(IEnumerable<AttributeValue> values);
    }
```

The simplicity of this interface shall enable a strong decoupling between the SW blocks.

The **Initialize** method tells the Communication Agent that the local appliance is about to start and the Communication Agent should synchronize the structural information with the application host. At this point, the agent should inform the application host that a device with certain capabilities is about to start and will soon send data.

The **Connect** function tells the Communication Agent that it should connect to the application host. The parameter is a reference to a callback function for incoming data.

The **Push** functions are used by the UIC to send data to the application host.

5. Project Agent Interface

The genuine responisbility of the Project Agent is to load a configuration and provide it to the UIC module in the form of an UIC project. In general, the configuration will be loaded from a remote configuration server or a local file.

Another important aspect is the option of publishing installed EDMs to a remote system. One possible scenario is to synchronize the local list of EDMs and its capabilities with a central EDM register. In that way, newly installed or deployed EDMs can be synchornized automatically with that registry and can then be utilized by other appliances.

The Project Agent gets invoked by the UIC module. It must not actively call the UIC and it must not have any dependency to an EDM or a Communciation Agent.

For that purpose a Project Agent must provide the following interface to the UIC module.

```
interface ProjectAgent
    {
        void Initialize(EmbeddedDriverModule[]);
        UicProject LoadProject(ProjectKey);
        void Synchronize(EdmCapability);
    }
```

The **initialize** method tells the Project Agent that it should prepare for loading the actual project configuration.

After initialization the UIC wants to obtain the actual configuration in form of a UicProject. Therefore, the UIC calls the **LoadProject** method. It is the responsibility of the Project Agent to provide the project, be it a request to a remote server or a read from the local drive.

The **Synchronize** method can be used to publish the capabilities of the local installed EDMs to an EDM function repository. This can be helpful in cases with complex remote configuration scenarios.

5.1 UicProject Interface

The UicProject defines the actual setup of the specific appliance. It contains information about the data points and EDMs to use and includes parameter to control the data preprocessing of the UIC module.

```
    interface UicProject
    {
        string ProjectKey { get; }
        string Name { get; }
        string Description { get; }
        string Owner { get; }
        Guid CustomerForeignKey { get; }

        IEnumerable<AttributeDefinition> Attributes { get; }
        IEnumerable<ProjectDatapointTask> DatapointTasks { get; }
    }
```

```
interface ProjectDatapointTask
{
    long PollIntervall { get; }
    string Description { get; }
    DatapointDefinition Definition { get; }
    DatapointTaskReportingCondition ReportingCondition { get; }
    DatapointTaskMetadata MetaData { get; }
}


interface DatapointTaskReportingCondition
{
    double ReportingThresholdInPercent { get; }
    double MinimalAbsoluteChange { get; }
    double MinimalPercentageChange { get; }
}


interface DatapointTaskMetadata
{
    double ExpectedMaximum { get; }
    double ExpectedMinimum { get; }
    double WarningThreshold { get; }
    double ErrorThreshold { get; }
    bool IsInverseThresholdEvaluation { get; }
    string Tags { get; }
}
```

The Project ProjectDatapointTask represents a DatapointDefinition that is configured to be monitored.

The DatapointTaskReportingCondition will be evaluatated by the UIC module to control when to push new values to the Communication Agent.

The DatapointTaskMetadata contains further information about the configured DatapointDefinition. In general, this information is meant for the application server to add some semantic meaning. One scenario could be to generate automatic dashboard from this meta data.

6. UIC / Universal-IoT-Connector

```
interface UniversalIotConnector : IDisposable
{
    string SerialId { get; }
    void Initialize(EmbeddedDriverModule[]);
    void Push(DatapointValue);
}
```

The UIC is the general program manager of the UIC-Framework. As the authority, this component only provides a public entry point to enable a launcher to start up the framework. The connector consumes all the previously defined interfaces to orchestrate the program flow.

Nevertheless, the startup process within the UIC is not trivial as you might have noticed in the startup sequence diagram. A .NET reference implementation may help you to understand the general process and provide you a jump start for the actual implementation.

The UIC.NET reference implementation can be found at https://github.com/sgetuic/UIC.net

7. Configuration

This standard describes the interfaces between the different building blocks of an IoT solution from an embedded system perspective. The implementation details are up to the community who wants to utilize this standard. With this approach a maximum of flexibility with minimal risk of technology lock-in is guaranteed.

But this also means that the configuration of the different modules is the responsibility of the respective implementation and cannot be part of this specification.

8. Application Cloud Requirements

For an open definition of the Cloud Agent Interface we previously examined the connection and data model requirements of some commonly used application clouds.

- AWS
- Microsoft Azure Cloud
- M2MGO's People System Things (PST)

	AWS	Microsoft Azure Cloud	M2MGO's People System Things (pst)
Protocol	MQTT, Rest	MQTT, AMQP, Rest	MQTT, Rest, Custom protocols, CoAP
Client Security Certificate	Must	Must	Optional
Essential Data Model	• Device Identifier • Sensor Key • Property Key • Device Type Key (optional)	• Device Identifier • Sensor Key	• Device Identifier • Sensor Key • Attribute Key (like Property Key) • Blueprint UUID (like Device Type Key)
Cloud to Device Data	• Supported	• Supported	• Supported
Visualization	• Custom Application	• Stream Analytics & Power BI • Custom Application	• Integrated App Creator Incl. Stream Analytics • 3rd Party Services

The UIC standard is written in order to allow the usage of all these cloud providers and more. The listing here is just as a reference.

9. Open Source Community – Github.com

To enable a strong community exchange, SGET created a repository for the UIC and demo implementations for Windows 10 (based on .net) and Linux (based on Mono) from the SGET are already available. You are welcome to contribute to and benefit from this open source project.

The SGET repository can be found at https://github.com/sgetuic .

The code is under MIT license conditions available for anybody.

---

10. Joint effort recognition

Many thanks to the team members of the SDT.04 for their engaged work and the special contribution of Jens Uhlig from M2MGO, Berlin as an editor for the specification.

Involved SGET members in alphabetical order: ADLINK, congatec, Kontron, iesy, Portwell, Seco.

11. Credits

Icons made by Freepik from www.flaticon.com is licensed by CC 3.0 BY

- Figure 2.3 UIC application reference architecture model
  - illumination.svg
  - solar-panel.svg
  - electric-meter.svg
  - robot-arm.svg
  - heartbeat.svg

The SGET Board