Uei-Bridge application

Design, decisions, detailed specification.

Contents

[Some project issues. 1](#_Toc143689083)

[ICD 1](#_Toc143689084)

[Names 1](#_Toc143689085)

[Digital card DIO403 2](#_Toc143689086)

[Block sensor 2](#_Toc143689087)

[ToDo 2](#_Toc143689088)

[Converters 2](#_Toc143689089)

[Device table 2](#_Toc143689090)

[Device list 2](#_Toc143689091)

[Project modules 3](#_Toc143689092)

[Config module 3](#_Toc143689093)

[Startup 3](#_Toc143689094)

[Main program 3](#_Toc143689095)

[Startup 3](#_Toc143689096)

[Build per-device-objects for device/slot 3](#_Toc143689097)

[Uei-Bridge-Setup. (setup editor) 3](#_Toc143689098)

[Use case 1: App startup A 3](#_Toc143689099)

[Use case 2: Open existing setup file. 4](#_Toc143689100)

[Use case 3: Create empty setup file. 4](#_Toc143689101)

[Use case 3: Create default setup file. 4](#_Toc143689102)

[Use case ?: Add configuration entry for specific device 5](#_Toc143689103)

[Software guidelines 5](#_Toc143689104)

# Some project issues.

## ICD

Ethernet => Device (downstream), starts with {aah, 55h).

Device => Ethernet (upstream), starts with {55h, aah).

## Names

1. Device (real) name is determines by as Uei api: GetDeviceName()
2. Block sensor device name is **“BlockSensor”**
3. Slot number is given by Device.GetIndex().

## Digital card DIO403

48 bits. Might be configured as input or output

Read/Write is done with an array of Int16. The 8 higher bits are ignored.

Currently, lower 24 bits are considered output, and upper 24 bits are considered input.

When **block sensor** is active, this card output is redirected to block sensor.

## Block sensor

~~Block-sensor depends on A0308Manager for outputting analog value.~~

BlockSensor uses analog output and digital input cards. Thus, when creating a default setup file, an entry for block sensor shall be created only if there is a digital input card (403) and analog output card (308). The entry of the block sensor shall refer to the slots of those cards.

The slot number of the block sensor itself shall be 32, and the device id shall be 32.

## ToDo

1. Unit test on EthMessage
2. Maybe it is better **not** to use Activator.CreateInstance() For device managers, you earn one and loose two. Something like DeviceManagerFactory might be more efficient.

## Converters

**~~To do 1~~**~~: Dedicated converter for each card looks overwhelming. There should be Analog-Convertor which unites 308 and 201 cards, and Digital-Convertor with unites 403 and 470 cards. All others (serials) might be handled be the device manager itself.~~ Done.

**~~To do 2~~**~~: Use~~

~~public interface~~ **~~IConvert2<T>~~**

~~{~~

~~T DownstreamConvert(byte[] messagePayload);~~

~~byte[] UpstreamConvert(T dt);~~

~~string DeviceName { get; }~~

~~}~~

~~Instead of IConvert,~~ Done.

**~~To do 3~~**~~: It is too heavy to enforce a convert to know about the setup of the program. Just give the needed parameters! (min voltage, max voltage, etc…)~~ Done

## Device table

### Device list

List< PerDeviceObjects> ~~\_deviceObjectsTable~~ \_deviceObjectsList

There might be more the one entry per slot.

class PerDeviceObjects

{

DeviceName

SlotNumber

CubeUrl

DeviceManager (in or out)

UdpReader (for output device only)

~~SL598Session~~SessionAdapter

}

# Project modules

## Config module

Roles & responsibilities:

Holds the persistent config.

This is a “half singleton” class (instantiation is done from outside).

Can generate default config according to device names.

Can read config from file.

Can write config to file.

### Startup

1. Upon startup, the main program checks to see if config file exists, if exists, load it.
2. If config files does not exists, the main program asks the Config modules to create default config according to device names, and saves it to file. This config shall be used for current session.
3. If the loading of config file fails, the program terminates. (generation of default config is done only if there is no config file).
4. **Todo**: change Config2 class according to above.

## Main program

### Startup

1. App search for cubes according to file cubelist.txt which contains simple list of cube url's. (If this file doesn't exist, it starts scanning the nearby network.)
2. App inquire each cube for its device list and fills one-dimensional list with 'Device' and cube url. This is real-device-list.
3. App emit to logger a list of the devices.
4. For each entry in read-device-list:
   1. Is there a device manager for the device is this slot? If not, emit log and continue to next entry.
   2. Is there a config entry for this entry? If not, emit log and continue to next entry.
   3. Is the config entry type match the device? If not, emit log and continue to next entry
   4. Build per-device-objects (see use case) and add them to \_deviceManagerList

### Build per-device-objects for device/slot

1. List<PerDeviceObjects> BuildDeviceObjects( device, cubeUrl)
2. If it is serial, first create sessionObject
3. If DIO403, create both input and output manger.
4. If AO308, create output manager
5. If 470, create output manger.
6. If AI, create input manager
7. If serial, create both input and output manger.
8. After all that, add block-sensor. If block sensor enabled, redirect output of DIO403/input to block sensor and disable inputs to AO308 which is know owned by block sensor.

## Uei-Bridge-Setup. (setup editor)

### Use case 1: App startup A

Preconditions:

#### Steps:

1. This use case starts right after user launch Bridge-Setup app.
2. App try to read default setup file.
3. App verifies file validity
   1. The device with same slot
   2. Unknown device name
   3. …
4. If default file doesn’t exist or not valid, A warning is shown on upper panel: “*no valid setup file*”. left panel remains blank. Use case end.
5. Populate view entries:
6. If file exists for each cube found in setup file, app shows icon in left panel.
7. If cube not connected, cube icon is displayed with red background.
8. User might expand each cube and see the list of devices:
   1. Slot0: AO308
   2. Slot1: DIO403
   3. …
9. For each slot/device entry, the app checks to see against physical cube if such device actually resides in specified slot, if not, the device is shown with red background. (No red background if there isn’t connection with cube).
10. If no cube connected, a warning is shown on upper panel: “*no cube connected*”.
11. Use case end.

### Use case 2: Open existing setup file.

Preconditions: Bridge-Setup is up.

#### Steps:

1. User select: Menu>File>Open setup file.
2. If file is valid, continue with use case 1 - Populate view entries
3. If not, use case end.

### Use case 3: Create empty setup file.

Preconditions: Bridge-Setup is up.

#### Steps:

1. User select: Menu>File>Create empty config file
2. App asks for file name and creates an xml file with one empty cube setup:

<UeiBridgeCubeSetup>

<UeiCubes>

<CubeSetup uri="pdna://192.168.100.2/">

</CubeSetup>

</UeiCubes>

</UeiBridgeCubeSetup>

1. continue with use case 1 - Populate view entries

### Use case 3: Create default setup file.

Preconditions: Bridge-Setup is up. At least one cube connected.

#### Steps:

1. User select: Menu>File>Create **default** config file
2. App asks for file name and creates an xml file based on connected cubes and default value.
3. continue with use case 1 - Populate view entries

### Use case ?: Add configuration entry for specific device

Description: The user wants to add/update configuration for specific device

Preconditions: config file already exists.

1. The system loads the existing setup file and opens the default view..
2. If user wants to change the device name, he’ll get a drop-down list from which he might choose.
3. (no limit on slot number)
4. If a uei cube is connected, the app search for such device. The app indicates:
   * No connected devices.
   * Such device does not reside in any of the connected cubes.
   * This device resides on cube2/slot3, cube2/slot4 etc…
5. The app shows a form which is suitable to the selected device.
6. The use might select auto-fill, or fill each field.

## Software guidelines

1. Output-Device-Managers must set \_isDeviceReady flag upon successful opening. (this flag is not needed in input-mangers).
2. Output-Device-Manager Dispose:
   1. Set \_inDisposeState = true. (Signal writer not to handle incoming requests)
   2. Dispose writer(s).
   3. Dispose session -\_ueiSession.Dispose(). **NOT in Serial/CAN**.
   4. call base.TerminateMessageLoop();
   5. show debug message
3. Input-Device-Manger Dispose:
   1. Serial/CAN: Set \_inDisposeState
   2. Session.Stop()
   3. Serial/CAN: Wait for all readers to finish (by timeout).
   4. Digital/analog: Dispose sampling timer. NOT in Serial/CAN
   5. Dispose reader(s).
   6. Dispose session
   7. Dispose Target-Consumer
   8. show debug message
4. protected ISession \_ueiSession; Shall exist in higher level. Same for input and output.
5. Udp readers are disposed by ProgramObjectBuilder. Udp writers are disposed by the device mangers that use them.
6. *protected ISend<SendObject> \_targetConsumer*; For input managers. Shall exist in higher level
7. all managers shall hold their private setup if **\_thisSetup** field.

# New architecture (Aug 23)

## Use cases

### Use case 1: PowerBridge (GUI) startup

#### Description

User launch PowerBridge and select which cubes/racks it should manage.

#### Actors

User.

PowerBridge application.

#### Preconditions

Setup files exists for each cube type that user wants to connect.

#### Postconditions

PowerBridge connected to selected cubes, up and running.

#### Steps

1. User launch PowerBridge.
2. UeiBridge search for all cubes in local network and maps the ‘busy’ ones.
   1. 'busy' means that a file called "cube<N>.token.json" exists in global folder.

Cube<N>.token.json

{

CubeAddress:

HostName:

StartTime:

}

1. PowerBridge show a list[[1]](#footnote-1) of cube ip’s and let user select which cube(s) should be connected in current session.
2. User selects cubes and click ‘connect’.
3. For each selected cube, UeiBridge search for suitable setup file. If there is more than, the user shall be asked to select, otherwise, the setup files will be loaded automatically.
4. (‘suitable’ means that there is a match of slot/device between the cube and the setup file. The name of the setup file does not matter)
5. PowerBridge:
   1. Opens and activate the selected cubes.
   2. Create a file "cube<N>.token.json" per each cube in global folder. Every 10 seconds, the existence of this file is verified.
   3. Add log line in global log location: " PowerBridge on machine <XYZ> activated cubes 192.168.100.3, 192.168.100.13.
   4. wait for incoming messages by multicast, and forward them to output devices
   5. read message from input devices and sends them through multicast to consumer.
6. Upon need, user close PowerBridge
7. PowerBridge:
   1. Delete files "cube<N>.token.json" per each cube in global folder.
   2. Add log line in global log location: " PowerBridge on machine <XYZ> closed cubes 192.168.100.3, 192.168.100.13

### Use case 2: Accept new cube

#### Description

When a new cube/rack is accepted from vendor, new unique ip must be assigned to it.

#### Actors

User.

PowerDNA explorer application.

CubeNet app.

#### Preconditions

The new cube is connected.

#### Postconditions

New ip assigned to new cube.

File CubeRepository.json (in ‘global folder’) is updated details of new cube/rack.

#### Steps

1. User launch CubeNet app and clicks “***Generate number for new cube***”
2. CubeNet checks in file CubeRepository.json what are the available numbers.
3. CubeNet display cube number.
4. User click ‘**Accept’**.
5. CubeNet display the ip of the new cube and ask user to user PowerDNA
6. User launce PowerDNA Explorer and sets the new ip for the new cube.
7. User clicks: ‘**check cube**’
8. CubeNet validates that the new cube is responding.
9. CubeNet updates file CubeRepository.json:
   1. Read new cube type string
   2. If the new cube type is already defined in CubeRepository.json file, the new cube ip is added to existing cube-type entry.
   3. If the new cube type is now known, CubeNet asks user for short name and description for the new cube-type.
   4. Add CubeType entry to file CubeRepository.json

“CubeType”: {

TypeId: “DIO403- SL508- SL508- SL508- SL508- SL508- DIO452- DIO452-CAN503”

NickName: “mcc”

“Desc”: “this cube should connect to mcc”

CubeList: [“192.168.100,3”, “192.168.100,15”, “192.168.100,17”]

}

“CubeType”: {

TypeId: “AO308- DIO430- SL508”

NickName: “ins”

“Desc”: “This cube should connect to ins”

CubeList: [“192.168.100,3”, “192.168.100,15”, “192.168.100,17”]

}

### Use case 3: Create new setup file

#### Description

Creates default setup file for specific cube type.

#### Preconditions

Access to updated CubeRepository file in global folder.

#### Postconditions

New setup file created. (example “CubeSetup.**fcc**.config”).

#### Actors

User

CubeDesign app

#### Steps

1. User launch CubeDesign app.
2. User clicks button ‘create new cube setup’.
3. Bridge-Design reads CubeRepository and display list of cube types.
4. User select one entry from cube type list.
5. CubeDesign generates default config and populate fields.
6. User edits setup.
7. User click ‘save as’, enters file name and save it.
8. Use case end.

### Use case 4: Edit existing setup file

#### Description

After user generated default setup file, he wants to edit it using this use case.

#### Preconditions

#### Postconditions

Setup file updated.

#### Steps

1. User launch CubeDesign app.
2. User clicks button ‘open cube setup file’.
3. CubeDesign open a file-select-dialog.
4. User select setup file
5. CubeDesign populate fields according to file.
6. User edit fields.
7. User click ‘save’.
8. Use case end.

1. Might consult CubeRep files, and show details about selected cube. [↑](#footnote-ref-1)