Brüel & Kjær

Technical University of Denmark

REST based protocol parser

Prototype for future embedded implementation

s042067 – Clausen, Per Boye

DTU Informatics
Department of Informatics and Mathematical Modelling

IMM-B.Eng-2013-4

Delivery: May 27th, 2013
DTU supervisor: Stig Høgh
B&K supervisor: Helge Egelund Rasmussen

Technical University of Denmark
Informatics and Mathematical Modeling
Building 321, DK-2800 Kongens Lyngby,
Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk IMM-B.Eng-2013-4

Summary (English)

This report describes the development of a PC prototype of a parser which shall eventually run under Windows Embedded Compact 7 on an ARM CPU.

The protocol which the parser is developed for is Web-XI – a REST based protocol developed specifically for use in the next-generation Brüel & Kjær equipment. Not all aspects of the protocol are regarded in this first prototype, but focus is in particular on setting and retrieving values in the data model of the hardware.

When a client sets or retrieves data through the parser prototype, the data is represented using JSON. The primary effort has been put into parsing and generating JSON. Data is stored in WebXiCache, a data structure which was provided and developed somewhat in parallel with this project.

As the prototype should later be able to run on the ARM CPU, options have been considered regarding how the prototype could be implemented on a PC in much the same way as it should be on the ARM – as an ISAPI extension. A custom web server was developed from a simple example which was provided, and modified based on an analysis of the ISAPI interface.

To verify the functionality of the parser, both internal and external tests were made. Internal tests run in the same scope as the parser prototype, and they can assert values directly on the data structure. The external tests use the API which will later be used in the PC software which shall communicate with the hardware to set and retrieve values through the parser prototype. This tests the entire chain from the connector API through the web server and parser prototype to the WebXiCache.

At the end of the project period, attempts were made by others to import the parser prototype into the actual development environment used for the ARM CPU. With a minimum of changes the code was able to compile for the ARM CPU, and first impressions were that the parser should be able to run properly without too much effort.

Summary (Danish)

Denne rapport beskriver udviklingen af en PC-prototype af en fortolker, som på sigt skal køre under Windows Embedded Compact 7 på en ARM CPU.

Protokollen, som fortolkeren er udviklet til, er Web-XI – en REST-baseret protokol, som er udviklet specifikt til brug i næste-generation af Brüel & Kjærs udstyr. Ikke alle aspekter af protokollen er betragtet i denne første prototype, men fokus har især været på at sætte og hente værdier i datamodellen på hardwaren.

Når en klient sætter eller henter data gennem fortolker-prototypen, bliver data repræsenteret vha. JSON. Det primære fokus har været på at fortolke og generere JSON. Data gemmes i WebXiCache, som er blevet stillet til rådighed og udviklet delvist i parallel med dette projekt.

Da prototypen senere hen skal kunne køre på en ARM CPU, er forskellige muligheder for, hvordan prototypen kunne implementeres på en PC på stort set samme måde som på ARM – som en ISAPI udvidelse – blevet overvejet. En speciel web server er udviklet ud fra et simpelt eksempel, som var blevet stillet til rådighed, og tilpasset baseret på analyse af ISAPI interfacet.

For at verificere fortolkerens funktionalitet er både interne og eksterne test blevet udviklet. De interne test kører i samme miljø som fortolker prototypen, og de kan direkte tilgå og vurdere værdier på datastrukturen. De eksterne test bruger det interface, som senere vil blive brugt til den PC software, som skal kommunikere med hardwaren, til at sætte og hente værdier gennem fortolker prototypen. Dette tester hele kæden fra interface gennem web server og fortolker prototypen til WebXiCache.

I slutningen af projektperioden forsøgte andre at importere fortolker prototypen i det rigtige miljo, som bruges til udvikling til ARM CPUen. Med et minimum af ændringer var det muligt at kompilere koden til ARM CPUen, og indtrykket var, at fortolkeren burde kunne køre uden de store ændringer.

Acknowledgements

I wish to thank the following for help and support during this project period:

Helge Egelund Rasmussen

Stig Høgh

Klaus Elk

Jeppe Kronborg

Lars Thestrup

Carsten Hansen

Special contributions

This project depends on some particular contributions deserving special recognition:

WebXiCache etc. – provided by *Helge Egelund Rasmussen*

Web server skeleton – provided by *Klaus Elk*

WebXi .NET/C# connector – provided by *Jeppe Kronborg*

Table of contents

Table of figures

Chapter 1

Introduction

This chapter gives a brief introduction to Brüel & Kjær and the line of products relevant to this project.

The introduction does not cover Brüel & Kjær entirely, but focuses on introducing the aspects relevant to this project. An insight into the Brüel & Kjær history and product legacy can be found in the book Journey to Greatness – the Story of Brüel & Kjær[1].

As this project concerns a protocol parser – a small (but vital) component residing deep within the system – some effort is put into placing the parser in the system, including descriptions of the surroundings, before the actual project is presented.

A glossary is located in Chapter 9.

About Brüel & Kjær 1.1

Brüel & Kjær was founded in 1942 by Per V. Brüel and Viggo Kjær. The company has been leading on the professional market for sound and vibration measurement – but has also been active in other areas such as long-term condition monitoring of machinery and medical instruments.

In 1992, following financial trouble, the company was sold to the German AGIV and split into several more focused companies, one of which is Brüel & Kjær Sound & Vibration Measurement A/S, which became part of the Spectris Division of AGIV. This division was later sold to the British Fairey Group Ltd. but was ultimately kept under the name of Spectris Plc. – still maintaining the Brüel & Kjær Sound & Vibration Measurement name within.

B&K has for most of its history had the head quarters in Nærum. This is also where much of the development is still being done.

The products developed by Brüel & Kjær Sound & Vibration Measurement A/S include the LAN-XI modules which perform data acquisition – measuring from a variety of different sources, ie. microphones, accelerometers etc. – and do some processing of the input. Various connection options are achieved through interchangeable fronts. A LAN-XI module with two detached fronts is shown in figure 1.1. The modules are

Figure 1.1: A LAN-XI module with two detached fronts

connected to an ordinary LAN, and the measurements are processed and presented on a PC using the PULSE software developed by B&K. Several LAN-XI modules can be used simultaneously, making the system very flexible and expandable.

Furthermore, the product range includes 2250 range handheld analyzers which are stand-alone PDA-like devices used to perform and present acoustic or vibration measurements, systems for environmental monitoring of airports, roads, building

sites etc. – as well as transducers; microphones and accelerometers for vibration measurement.

1.1.1 The Instrumentation group

The Instrumentation group is part of the R&D department. The group develops much of the new hardware and embedded software used within the company and the products. This includes the LAN-XI modules and the 2250 range handheld devices.

Development of a new generation of LAN-XI modules is beginning. This project covers part of this development.

LAN-XI hardware 1.2

LAN-XI is the name of a series of measurement equipment made by Brüel & Kjær. The first LAN-XI model was released in 2008, and since then many more have followed with different configurations of inputs and outputs etc. All 1st generation LAN-XI (LAN-XI G1) modules share the same form factor shown in figure 1.1.

The LAN-XI modules generally perform data acquisition – measuring from a variety of different sources – transducers, ie. microphones, accelerometers etc. – and do some processing of the input (most processing is, however, done on a PC using B&K PULSE software).
Some models are equipped with generators for outputting signals to speakers, shakers etc.

Connection to the LAN-XI module is made through an ordinary network, and a PC can use several LAN-XI modules simultaneously – even different models. This makes the LAN-XI system very versatile and makes it possible to place the modules closer to the source when measurements are done on large objects such as an airplane. The modules synchronize their timing through the network as well, ensuring that the PC will be able to get very accurate timing between the modules even though they may each produce very different latency due to switches etc.

The modules can be inserted into frames, combining several modules into one unit which may, depending on frame type, be mountable in a standard 19" rack. LAN-XI modules connected using frames are shown in figure 1.2. The frames also enable the addition of battery modules, which makes the system usable in the field without complicated setups with many separate units and cables. LAN-XI frames are also network connected. The flexibility of the LAN-XI system is proven by the fact that a module may be the backbone in an in-car system monitoring a few parameters while driving (such as identifying unwanted noise and maybe even the source of it) – but

Figure 1.2: A single LAN-XI module, a 5-module frame and an 11-module frame connected to a network switch.

many of the same modules (typically in frames) set up in a large network can measure on several hundred channels at once spread over a large structure such as a plane, turbine etc.

A key feature of the LAN-XI platform is the detachable fronts of the modules. A large collection of different fronts is available facilitating different number of connectors and different connector types – BNC, LEMO etc, and the range is continually expanding to meet new customer demands.

LAN-XI G2 1.3

The second generation of the LAN-XI platform – LAN-XI G2 – is now under development. This uses many of the concepts from the first generation, but it will also incorporate many of the lessons learned over the years. The changes include e.g. using Ethernet for internal communication between components rather than custom made hardware interfaces as in LAN-XI G1 – and using a new REST based protocol – Web-XI – for controlling each module instead of a custom protocol on top of TCP.

These changes should make some aspects of development easier by using known technologies, and may provide added options for future use, such as external access to control the module for developers outside B&K. Connections to a LAN-XI G2 module is done as with the G1 module – as illustated in figure 1.3.

1.3.1 Web-XI

Web-XI is the name of the new REST based protocol to be used for LAN-XI G2. The protocol introduces a well-defined structure and aims at completely replacing previous somewhat fragmented protocols with a single dynamic and well-documented one.

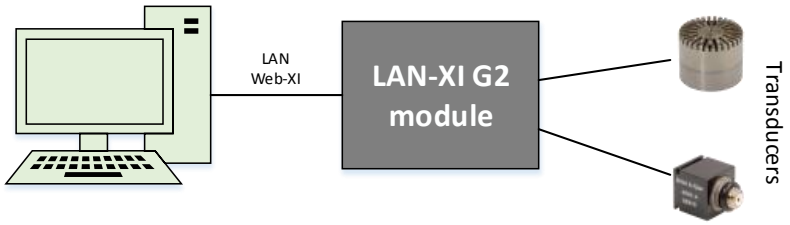


Figure 1.3: Illustration of the connection between a PC, a LAN-XI G2 module and transducers.

Web-XI uses JSON for representation of data for controlling the operation of a LAN-XI G2 module. The protocol is described in detail, and this project focuses on implementing the specification.

1.3.2 Architecture

The basic architecture has not changed much from G1 to G2. Each LAN-XI module basically consists of the following[1] – as illustrated in figure 1.4:

Input/output hardware providing connections for transducers etc. and performing analog/digital conversion.

DSP doing the necessary in-module signal processing.

ARM CPU providing the interface to the client over an ordinary LAN.

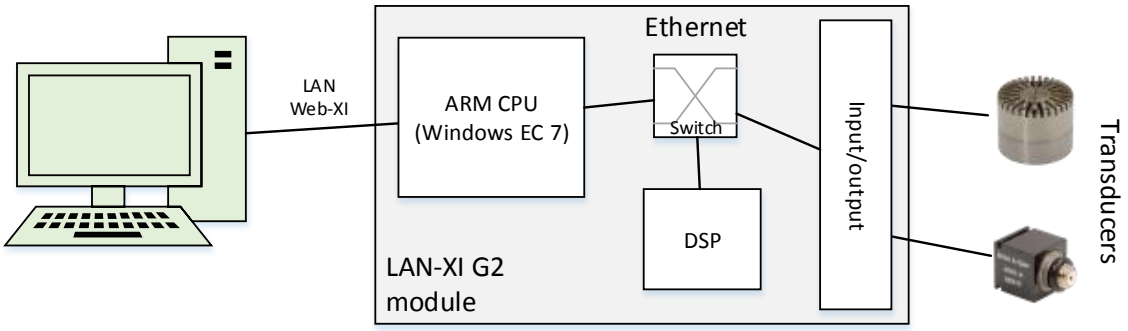


Figure 1.4: Simplified illustration of the LAN-XI G2 architecture.

When measuring, the signal will enter the module in the input/output hardware. Here the signal is digitized and passed on to the DSP.
The DSP applies any configured filters and sends the resulting signal to the ARM CPU

[1]The model is simplified to give a feel for the components involved for the use in this project

which handles further distribution of the signal to the client(s).
Any client can request access to a data stream on the ARM CPU. It will then pass the signal received from the DSP on to the client, completing the basic acquisition chain.

When a generator is present in the LAN-XI module, the generator signal path is reversed compared to the input. This is of course a very simplified model with the sole purpose of describing the signal path. Many more aspects exist in the acquisition process, but they are not relevant to the understanding needed for this project.

1.3.3 The ARM CPU

The ARM CPU in the LAN-XI module performs most of the tasks directly visible to the client (both PC and user), such as providing communication interface over the external network connection and controlling the display on the device – ie. an arbiter role.

This description is entirely focused on the tasks of the ARM CPU concerning this project – the web server, parser and cache. The relevant internals of the ARM CPU have been illustrated in figure 1.5.

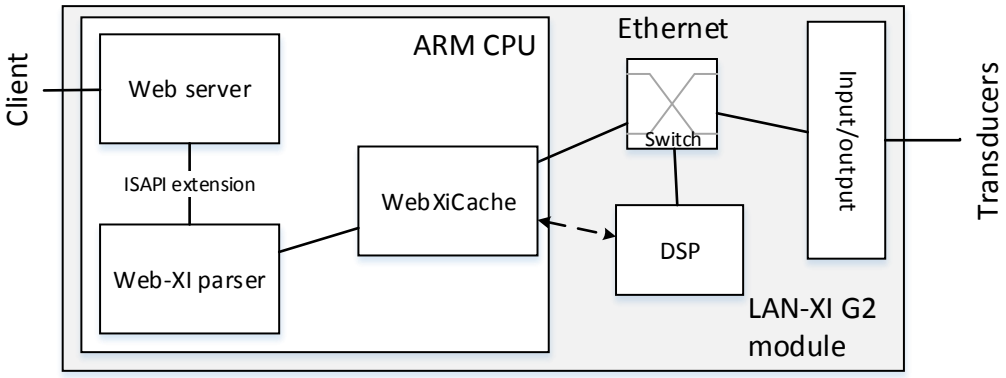


Figure 1.5: The blocks "inside" the ARM CPU relevant to this project and their connections.

Operating system

The ARM CPU runs Windows Embedded Compact (EC) 7 – formerly known as Windows CE. This is the ultimate target of the development made in this project, but initially, a prototype is developed on a PC. This prototype must be aimed at future implementation on the ARM CPU, but may also provide other benefits, which is discussed later in Chapter 7.

1.3.4 Web server

Windows EC 7 includes Internet Information Services (IIS), which has been used for previous web server and protocol implementations for LAN-XI G1.

1.3.5 Parser

The parser runs as an ISAPI extension on the web server. All requests relevant to Web-XI will be handled by this extension. The parser will bridge the gap between the client (Web-XI protocol) and the local cache data structure (WebXiCache).

1.3.6 Cache

The WebXiCache has been provided for use in this project.

WebXiCache will ultimately facilitate the connection between the Web-XI parser and the DSP.

Chapter 2

Analysis

This chapter focuses on the analysis of the project – describing the project itself and the entities surrounding it.

Web-XI 2.1

During operation, many parameters can be set on the LAN-XI module. These paramaters may be related to the analog input/outputs, signal processing – setting filters and measuring settings etc. These parameters are either related to the operation of the DSP or controlled by it.

Web-XI is based on REST and defines a tree of parameters which can be changed. Data is carried in JSON format.

The Web-XI protocol is specified in the *Design of Web-XI Communications Protocol for G2 devices* document which can be found in appendix D. This specifies the main requirements for the Web-XI parser, but not everything specified in the document is covered by this project.

2.1.1 REST

Representational State Transfer (REST) is a concept for communication between clients and a server. Clients can retrieve or change a representation of data using an intermediate representation – the client cannot directly access the underlying data.

In this case, JSON is used to describe parameters for the DSP.

2.1.2 JSON

In Web-XI, JSON is used to represent the parameters set or retrieved by the client. JSON uses a simple structure for describing a tree data structure, which is very compact.

An important objective of this project is for the parser to be able to parse and generate JSON from the cache structure.

The following description of the JSON structure has been partly derived from [2].

JSON operates with the term *object* which contains sets of any number of *name:value* pairs, separated by commas (,).
name is always a *string*, and value may be another *object*, or *string*, *number* (which covers both integer or floating point), *boolean* (true/false), *null* or *array* of *values*.

A more comprehensive illustration of the JSON structure is illustrated in figure 2.1.

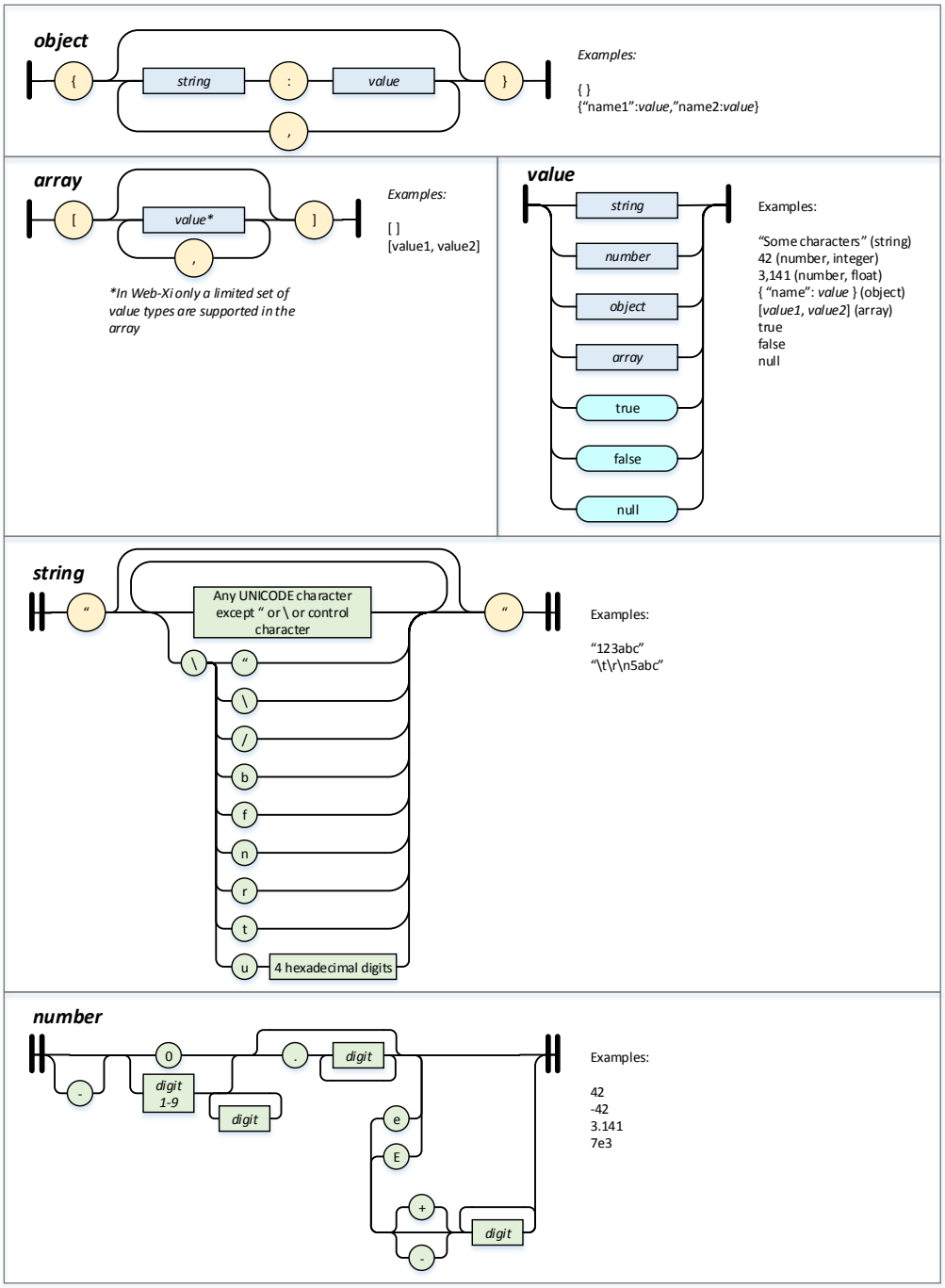
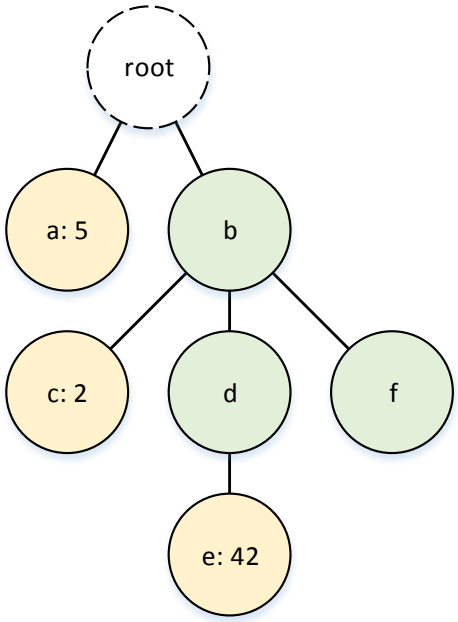


Figure 2.1: Illustration of the JSON structure. The illustration has been recreated on basis of [2].

```
{
  "a": 5,
  "b": {
    "c": 2,
    "d": {
      "e": 42
    },
    "f": {}
  }
}
```

(a) Simple JSON example.



(b) Tree representation. The *root* node is the outer object container, green nodes are *objects* and yellow nodes are *name:value* pairs.

Figure 2.2: JSON example and corresponding tree interpretation.

An example of a simple JSON structure is shown in figure 2.2. The JSON describes an object containing *a* – with value 5 – and *b* which itself is an object. *b* contains *c* which has a value of 2, and the objects *d* and *f*. *d* contains *e* with a value of 42, and *f* is empty. The figure also shows how the JSON structure can easily be represented as a tree structure.

2.1.3 Object model

A key element in Web-XI is the object model, which is a tree structure including all the parameters that can be set on the module.

The tree has a root node which defines the entry point for every operation on the tree. From the root node there may be any number of branch nodes leading down to leaf nodes. Branch nodes cannot contain any data apart from their children – other branch or leaf nodes – and only leaf nodes contain actual values.

An example object model is shown in figure 2.3. It is not an actual object model to be used in a finished LAN-XI G2 system; it is designed to illustrate the concepts of the Web-XI in a fairly compact tree. The example object model is used as a basis for development and testing throughout the project.

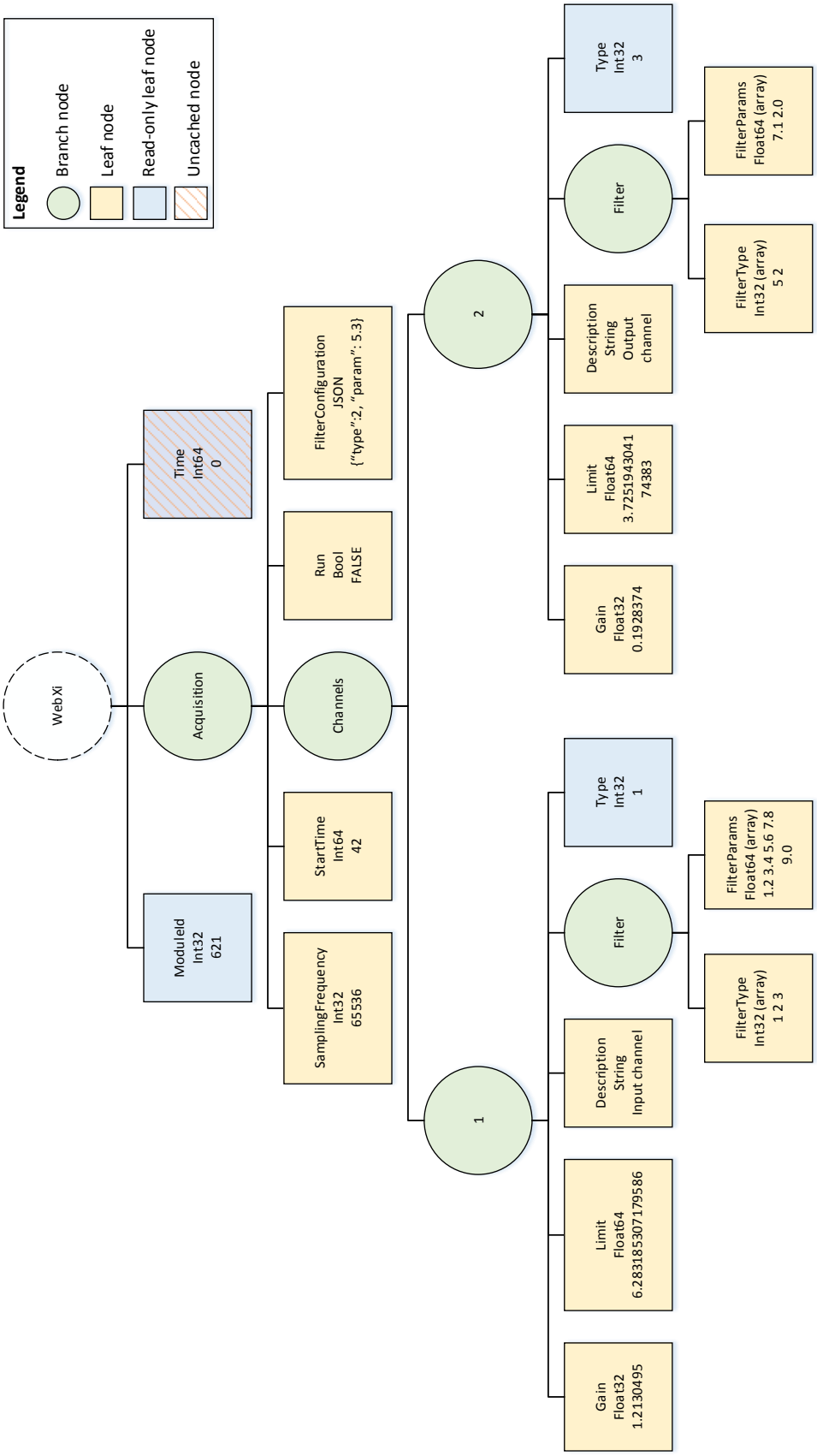


Figure 2.3: Illustration of an example object model.

The object model is published to the cache by the DSP at startup, and only the DSP can add or remove nodes in the object model. The client can – using Web-XI – change values of nodes in the object model.

Node value types Leaf nodes in the object model each have a data type assigned (The names in parenthesis are shorthand names used throughout this report):

- 32-bit Integer (`Int32`) – arrays supported
- 64-bit Integer (`Int64`)
- 32-bit Floating point (`Float32`)
- 64-bit Floating point (`Float64`) – arrays supported
- Boolean (`Bool`)
- `String`
- `JSON`

Apart from the value types on each node, integer and floating point nodes may also contain arrays. The value type of array nodes is the same as for scalar nodes. Each node holds information about the current and maximum length of the value, indicating whether it is a scalar or an array.

JSON nodes The JSON value type is used for dynamic trees. Rather than creating nodes for every combination possible in a subtree[2], or letting the client create and delete nodes[3], a node may itself carry a JSON data structure for the DSP to parse.

Read-only nodes Some nodes in the object model may be informational only, meaning that they must not be changed by the client. These nodes carry a read-only flag.

[2] A thought example could be an input channel which can be set up with any combination of several filters which can each have a combination of functions and parameters. This could result in e.g. 16 different filters × 10 possible parameters per filter = 160 nodes per channel – of which just a few parameters will be used at any time.

[3] This would make the object model more dynamic and unpredictable, making it much more difficult to manage the limited resources such as memory consumption.

Cached and uncached nodes Most nodes will only change value when the client requests it and may be cached on the ARM CPU, relieving the DSP when nodes are requested by the client. There are, however, some nodes that cannot be cached and need to be refreshed from the DSP every time their values are requested[4]. This is shown by a flag on the nodes.

2.1.4 Protocol

The Web-XI protocol defines the relation between HTTP request, JSON data and the object model.

The **path** given in any HTTP request is used to address a node in the object model. This may be any node, branch or value, and will be the root node of further processing.

The **method** of a request determines the desired action. For the part of Web-XI covered in this project, *GET* and *PUT* are covered. If a *GET* request is made, the requested contents of the object model should be compiled into JSON notation and sent to the client. The client can specify the query field *recursive* to control whether the entire subtree of a node should be returned, or just itself/its children. Recursion defaults to being false.
In a *PUT* request, the client sends a JSON structure containing the structure (values) to write to the object model.

Method examples for getting/setting data

Making a **GET** request on a **leaf node** with the path */WebXi/Acquisition/Channels/1/Gain* should return a JSON object with the *Gain* node and its value:

```
{
  "Gain": 1.2130495
}
```

A **GET** request made on a **branch node** with the path/query */WebXi/Acquisition/Channels/1?recursive=false* should return the children of the node named "1". The JSON structure returned to the client would be as follows (note that *Filter* is shown with the value *null*. This is used to indicate that there is a node called Filter, and that it may have children.):

[4] An example could be a node carrying the DSP time or number of samples processed by the DSP – which are values that (may) change without the client doing anything.

```
{
  "Gain": 1.2130495,
  "Limit": 6.283185307179586,
  "Description": "Input channel",
  "Filter": null,
  "Type": 1
}
```

Sending much the same request, but with *recursive=true* should result in the JSON below being returned. Now the children of *Filter* are included in the representation.

```
{
  "Gain": 1.2130495,
  "Limit": 6.283185307179586,
  "Description": "Input channel",
  "Filter": {
    FilterType: [1, 2, 3],
    FilterParams: [1.2, 3.4, 5.6, 7.8, 9.0]
  },
  "Type": 1
}
```

Sending a **PUT** request to the path */WebXi/Acquisition/Channels/1* with the body shown below would update the values of *Gain*, *Description* and *Filter/FilterType* to 2.0, "Main microphone" and [5 2] respectively. It is not necessary to specify values for all children of the requested node, as shown (*Limit* etc. have been left out).

```
{
  "Gain": 2.0,
  "Description": "Main microphone",
  "Filter": {
    FilterType: [5, 2]
  }
}
```

Action and time related queries

PUT requests can be used to perform certain actions on nodes. This is not true REST, but it is hard to do without for the LAN-XI modules. This may be used to ask a channel (represented as a node in the object model) to detect any transducer connected to it. Actions are set by the *Action* query field, and the value is a string, which must match a valid action for the node requested. Actions can have an argument, set by the query field *Argument*.

Sending a PUT request with the path/query */WebXi/Acquisition/Channels/1?Action=Detect&Argument=All* should run the *Detect*

action on the Channel 1 node. Actions are not actually doing anyting in the prototype, but this action could mean "Detect transducer connected to channel 1". AwaitTrigger, Time and Delay fields could be added to the query.

Time and triggers

It is possible to schedule PUT (value setting and action performing) operations to be executed later, using the query fields *AwaitTrigger*, *Time* and *Delay*.

AwaitTrigger takes a number value from 0 to 7. If this is set to a non-zero value, the action specified in the request will not be performed until the specified trigger is executed. Several actions can be assigned to the same trigger and thus be executed at once. The trigger can be executed by sending a PUT request with the query *Action=PerformTrigger&Argument=<TriggerId>*.

Time takes a 64-bit unsigned integer as value. When set to a non-zero value, the action specified in the request will not be performed until the specified absolute time has been reached on the DSP. The time is described in section 6 of appendix D.

Delay takes an unsigned integer as value. When set to a non-zero value, the action specified in the request will be performed after the specified number of milliseconds have elapsed after the request has been received in the module.

Specifying both *Time* and *Delay* in the same request is not allowed, but one of them may be combined with *AwaitTrigger*. In this case the action is performed when the first (either *AwaitTrigger* or *Time*/*Delay*) condition is met.

2.1.5 Versioning

In order to identify and support different versions of the Web-XI protocol in the future, the HTTP header fields *Content-Type* and *Accept* can be used to identify the provided and requested protocol/version.

This can ensure that – in the future, when possibly more versions of the protocol exist – clients and modules can agree on the correct version to use, rather than having possibly faulty assumptions.

2.1.6 Client-side caching

The *Cache-Control* response header field can be used to tell the client if – and for how long – a response may be cached. This is useful for reducing the number of requests necessary for the client to make while still maintaining good synchronization between client and module.

A response with the header-field `Cache-Control: no-cache` must not be cached, and if the client needs to use any data from the response at a later point, the request must be repeated to ensure an up-to-date version. A simple example could be data containing the absolute time on the module, which is regularly updated.

The header-field may also read e.g. `Cache-Control: max-age=3600`, telling the client that the data may be cached and used for 3600 seconds (1 hour). This could be used for data which is not regularly updated, such as version information or data exclusively set and controlled by the client.

2.1.7 HTTP status

To indicate the result of an operation standard HTTP status codes are used. These are described below. When an error status is returned, the response body may contain information regarding the error that has occurred.

200 ("OK") Operation succeeded. On GET requests the body contains the document returned.

400 ("Bad Request") Generic client-side error used when no other 4xx status is appropriate. E.g. used when the client makes a PUT request with semantic or value type errors.

404 ("Not Found") Indicates that a resource could not be found.

405 ("Method Not Allowed") The HTTP method used in the request is not allowed. May be used if the client attempts to PUT values to a read-only node.

413 ("Request Entity Too Large") The request is too large for the server (or parser) to handle.

Project definition 2.2

This project definition gives an overview of the goals of the project which are later further elaborated. The project was initially defined some time before the project period. This resulted in the *Initial project considerations* document which is included in appendix B.

The project is basically the same, but many concepts and goals have changed, as much has changed since the initial considerations were written.

A few points were changed or removed from the initial considerations at the beginning of the project. One of these things is porting the DSP unit tests developed during the internship (see appendix C) to use network and the Web-XI parser to perform the tests. Another was considerations regarding using another programming language and/or framework for developing the parser, as this would introduce added complexity, more unknown factors, and possibly a new programming language which noone at B&K is familiar with, to the system. The programming language of choice is thus C.

2.2.1 Prototype

The prototype should implement the command part of the Web-XI protocol, described previously. Versioning and client-side caching are initially not considered. The parser shall – as previous versions – be installed as an ISAPI extension.

Attempts should be made to be able to communicate with the prototype using the actual Web-XI API being developed for use in the PULSE PC software which will ultimately be used to control the LAN-XI G2 modules.

Thus, the prototype should include a web server capable of receiving requests from a PC, and as the Web-XI parser will eventually be installed as an ISAPI extension on the ARM CPU, the web server shall at least emulate ISAPI to the extent needed to implement the prototype, and without making the transition to the embedded web server on the LAN-XI G2 module too problematic.

With the WebXiCache, a DSP simulator has been provided to receive and handle value updates from the cache, and a custom object model, such as the one shown in Figure 2.3 on page 13, can be used to verify functionality – but this will not be an actual object model for some future application of LAN-XI G2, but rather an object model which can give an idea of the functionality of the parser and be used in validation.

As stability and predictability are key aspects, it is important to ensure that correct data is committed to the cache. It is important to incorporate some transaction handling, so that in any errors in the data sent to the parser (semantic, data type

etc.) do not result in some values to be applied to the DSP and others to be skipped. In this case an error should be returned and the state of the system should be reverted to before the request was made.

2.2.2 PC implementation

Implementing the prototype on a PC makes development and initial verification much easier, requiring no special hardware and making it possible to quickly see the effects of changes.

However, as the target is ultimately IIS on the ARM CPU, efforts should be made to comply with the limitations it brings. As an effect of this, the parser should be implemented with the fact in mind that resources are limited, especially memory should be in focus and potentials for memory fragmentation should be reduced.

The optimal outcome would be an implementation which – using the same code – can operate on both a PC and the ARM CPU.

2.2.3 Use cases

The use case diagram for the project is displayed in figure 2.4.

A client shall be able to *Get parameters from the object model* in the module. This includes *getting data and generating a JSON representation*.
The client shall also be able to *Set parameters in the object model* in the module, which includes *parsing JSON data and storing values*. If any errors occur, the system should be *returned to a stable state*.
Finally, it should be possible for the client to *perform actions* on nodes in the object model. All use cases include *Finding a node in the object model*.

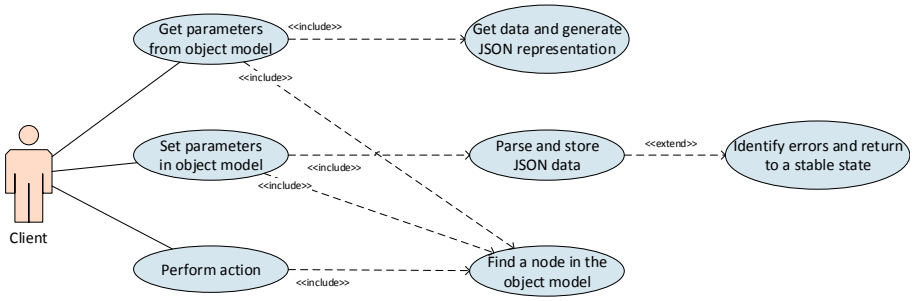


Figure 2.4: Use case diagram for the project.

Requirements 2.3

The "*Design of Web-XI Communications Protocol for G2 devices*" document in appendix D specifies the requirements for the protocol implementation. As only the command part of the protocol is considered, a breakdown of the requirements relevant to this project is presented here.

The requirements are divided into categories concerning the environment and prototype implementation and the actual parser related requirements, and some are further marked as A-, B- or C- requirements, where A requirements are success criteria, B requirements should be looked into, but are not essential, and C requirements are not initially part of this project, but should be developed in the future.

2.3.1 Environment

1. (A) The parser shall initially be implemented as a prototype running on a PC.
2. (A) Ultimately, the parser shall be ported to run on the ARM CPU, and the parser must be prepared for this transition.
3. (A) On the ARM CPU, the parser shall run as an ISAPI extension, so a suitable development solution should be found for the prototype, allowing the parser to run on a PC while being easily converted into an ISAPI extension.
4. (A) Everything should be developed in C/C++.
5. (A) Visual Studio 2010 with Visual C++ should be used for developing the system.
6. (A) A provided data structure – WebXiCache – must be used to store and retrieve data. This is the interface towards the inner parts of the LAN-XI G2 module.
7. (B) For future flexibility, the parser should be able to run on both the ARM CPU and a PC using the same code and a minimum of changes.
8. (B) A suitable web server should be used/implemented on the PC prototype to enable the PC to mimic the LAN-XI G2 module properly.

2.3.2 Web-XI parser

This is a breakdown of the relevant parts of the command protocol as described in appendix D.

1. (A) Request paths should be parsed:
   (a) (A) Find the node in WebXiCache corresponding to the request path.
   (b) (A) If the requested node does not exists, return an error.
2. (B) All node names are treated as case insensitive.
3. (A) Proper HTTP Status codes should always be returned. In the event of an error, the status code should be relevant to the error.
4. (A) Query fields should be parsed and their values extracted. Names are case insensitive:
   (a) (A) Recursive: true or false, default false.
   (b) (B) Action: String data, default empty.
   (c) (B) Argument: String data, default empty.
   (d) (B) AwaitTrigger: Int32, default 0.
   (e) (B) Time: Int64, default 0.
   (f) (B) Delay: Int64, default 0.
5. (A) GET requests should return JSON data to the client:
   (a) (A) If the requested node is a leaf node, a JSON object containing the name:value pair for the requested node should be returned.
   (b) (A) If the requested node is a branch node with recursive set false, a JSON object containing the name:value pairs of the children of the requested node should be returned. Branch nodes have the value null.
   (c) (A) If the requested node is a branch node with recursive set true, a JSON object containing the name:value pairs of all ancestors of the requested node should be returned. The value of a branch node is a JSON object with its children.
   (d) (B) All uncached nodes relevant to a request should be refreshed. The whole tree should not be refreshed every time if not necessary.
   (e) (B) Make sure the object model is consistent while generating a JSON structure – make sure changes are not made to the cache while generating the response.
6. (A) PUT requests should perform an action or update values in WebXiCache from the JSON data sent by the client.

(a) (B) If Action query is set (ie. not empty), let the WebXiCache perform the action (passing the value on as an argument) with the Argument, AwaitTrigger and Time data from the query. Nothing further should be done if Action is set.

(b) (A) Parse JSON data sent by the client and set the corresponding values in WebXiCache.

(c) (B) If a node with value type JSON is encountered, halt parsing and send the value (whilch should be a JSON object) into the node in WebXiCache as a string.

(d) (B) Make sure only one client makes changes at any time.

(e) (A) Enforce value types of nodes. E.g. trying to set a string to an integer node should result in an error.

(f) (B) Enforce JSON semantics. A JSON structure which cannot be interpreted as valid JSON should cause an error.

(g) (B) If an error occurs during processing, make sure no changes are committed to the WebXiCache and return it to the previous state.

(h) (B) If AwaitTrigger and/or Time/Delay is set, the values committed to WebXiCache should be postponed until the trigger or absolute time/delay time is reached.

(i) (B) Time and Delay may not be set at the same time. This should result in an error and no action should be taken from the request.

7. (C) Versioning shall at some point be implemented.

8. (C) Client-side cache control headers shall at some point be implemented.

2.3.3 Risks

Being done in a very early stage of the LAN-XI G2 development, there are uncertainties in many aspects of the development to take care of. Some identified risks are regarded here, including evaluation of their effect.

Resulting parser is not ARM compatible Probability: Medium – Effect: Minor

As the right tools for developing the ISAPI extension for the ARM CPU are not available during development, the outcome may be a PC parser prototype which cannot be ported to the ARM CPU. For the project the impact is minor, as the target is a PC prototype – but for further development it may be a greater problem, depending on the exact cause of the incompatibility.

Changes in requirements Probability: High – Effect: Medium

As many requirements, descriptions and systems for the parser prototype are in development during the project period, requirements may change while implementation is going on. Mostly, actual changes should have a minor impact,

but a flexible development strategy is important to be able to cope with these changes.

Lack of support in surrounding systems Probability: Medium – Effect: Medium

As this project concerns a small part in an chain of systems, this project depends very much on the other systems providing the needed supported for the things to implement. Some of these other systems are under ongoing development during the project period. This means that there may be shortcomings encountered – but help is at hand when they are identified.

ISAPI misunderstandings Probability: Medium – Effect: Minor

As development is done without the right tools for developing ISAPI extensions at hand, there is a real risk of misunderstandings leading to an incompatible implementation. The effect of these misunderstandings are, however, minor, as knowledge about ISAPI extensions in use has been gathered from previous implementations, and if anything should be wrong, the data used in the parser is very generic and easily substituted.

ISAPI 2.4

Internet **S**erver **A**pplication **P**rogramming **I**nterface (ISAPI) is an API of Internet Information Service (IIS), Microsoft's collection of web server services.

Based on studies of existing ISAPI extension implementations on the LAN-XI G1 hardware the entry point of the prototype has been determined.

An ISAPI extension is in IIS associated with a root path. For Web-XI this is /WebXi, meaning that any request going to /WebXi/* (on the web server port, default 80) on the device is being handled by the Web-XI parser. This makes it possible for the custom Web-XI parser to run on the same web server and port as the web interface and possibly older protocol implementations. Figure 2.5 shows a web server running a module website as default application at *http://[ip]*, a WebXi protocol attached to *http://[ip]/WebXi*, and an old legacy protocol attached to *http://[ip]/LANXI*. Any request made to *http://[ip]/WebXi/...* would be forwarded to the Web-XI plugin.

2.4.1 Entry point

The entry point of the ISAPI extensions is the function with the following signature:
`DWORD`[5] `WINAPI`[6] `HandleRequest(EXTENSION_CONTROL_BLOCK*)`

[5] DWORD: A 32-bit unsigned integer.
[6] Specifies the calling convention – how the function is called internally.

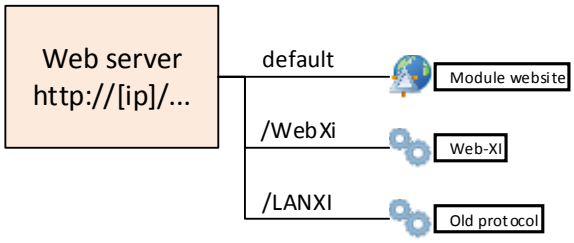


Figure 2.5: Illustration of a web server running a website and 2 plugins (ISAPI extensions). The example only for illustrative purpose, and is not related to any real setup.

This is invoked by the web server when the correct path has been requested by a client. Control is then handed to the `HandleRequest` funtion by the web server.

2.4.2 EXTENSION_CONTROL_BLOCK

The `EXTENSION_CONTROL_BLOCK` structure is defined in the `httpext.h` library and is central to parsing. It contains the following members (only the members relevant to this project are included[7]):

`ConnID` (HCONN[8]) – Refers to the associated connection. This is set and maintained by the HTTP server and used to distinguish between several different simultaneous connections being handled by the web server.

`lpszMethod` (LPSTR[9]) – A string containing the HTTP method (GET, PUT etc.)

`lpszQueryString` (LPSTR) – Any query information the URL, ie. the part of the URL after the question mark, if any.

`lpszPathInfo` (LPSTR) – String containing the query path – ie. the URL part before any question mark.

`cbTotalBytes` (DWORD) – The total number of bytes received in the input buffer.

`cbAvailable` (DWORD) – The number of bytes left to retrieve from the input buffer.

`lpbData` (LPBYTE[10]) – Pointer to the input buffer containing any request body sent by the client.

`lpszContentType` (LPSTR) – The content type of the request body.

`(WINAPI * WriteClient) ()` (BOOL) – Pointer to a function for writing data to the client.

[7] See `http://msdn.microsoft.com/en-us/library/ms525658.aspx` for the full documentation.
[8] HCONN: A unique identifier for an HTTP connection.
[9] LPSTR: Pointer to an array of characters
[10] LPBYTE: Pointer to a BYTE value

The `WriteClient` function has the signature
`BOOL WriteClient(HCONN ConnID, LPVOID`[11] `Buffer, LPDWORD`[12] `lpdwSizeofBuffer, DWORD dwSync)`.

`ConnID` is a connection identifiers for identifying the connection to use.
`Buffer` is a pointer to the buffer containing the content to send.
`lpdwSizeofBuffer` is a pointer to the number of bytes to send from `Buffer`.
`dwSync` is used to specify whether to run synchronous or asynchronous.

Example

This example illustrates how the `EXTENSION_CONTROL_BLOCK` is populated from a HTTP request. If the web server receives a request with the following contents:

```
PUT /WebXi/Acquisition/Channels/1?AwaitTrigger=3&Time=5000 HTTP/1.1
Content-type: text/json
Content-Length: 54

{
  "Gain": 5.3,
  "Description": "Monitor channel"
}
```

– the corresponding `EXTENSION_CONTROL_BLOCK` will have the following contents:

`ConnID`	[assigned by HTTP server]
`lpszMethod`	`PUT`
`lpszQueryString`	`AwaitTrigger=3&Time=5000`
`lpszPathInfo`	`/WebXi/Acquisition/Channels/1`
`cbTotalBytes`	`54`
`cbAvailable`	`54`

`lpbData`	`{`
	`  "Gain": 5.3,`
	`  "Description": "Monitor channel"`
	`}`

`lpszContentType`	`text/json`

[11] LPVOID: Pointer to a value of unspecified type.
[12] LPDWORD: Pointer to a DWORD value

2.4.3 Developing the ISAPI extension

ISAPI extension templates and tools have been removed from Visual Studio prior to the 2010 version. Furthermore, the add-on environment and compiler for developing for the ARM CPU are also discontinued.

Therefore, the entire development of the parser has been based on the knowledge summarized here. There has not been put a great effort into examining ISAPI extensions in detail, as the parser needs very basic information about the request such as path/query and body – and in case of problems, changing the sources of that information should be straight forward. The effort has instead been put into developing the parser.

Web server 2.5

The lack of tools to directly implement ISAPI extensions raises the question whether the PC prototype should run as an actual ISAPI extension, or if some custom solution should be used instead – of course to the greatest extent possible mimicking an ISAPI extension.

ISAPI-Capable web server Running the parser as an ISAPI extension obviously has an advantage when it comes to determining whether or not the final implementation can run. There are some possible differences between the platforms, ie. Windows 7 on a PC and Windows EC7 on an ARM platform, but the ISAPI interface should be the same.
The built-in IIS variant in Windows can load ISAPI extensions. The open source Apache server also has that capability.

Custom web server Some drawbacks of using a complete web server – apart from possible complications due to the lack of the right templates etc. – include lack of control. It might not be possible to directly access the stdin/out console and see debug prints generated by the software, and every time the ISAPI extension is compiled the web server extension/configuration has to be reloaded. A custom solution can be part of the parser development project, and compiled at the same time as the parser. Also, the custom server can initialize specific things only relevant to the prototype (such as the data structure in WebXiCache and other helpers).

Conclusion From the considerations it has been decided to develop the Web-XI parser with a custom web server. For this web server, a very simple example has been provided, which can be used as a basis for the development of the server. As the web

server is only part of the PC prototype of the parser, and not the final implementation on the LAN-XI G2 module, the requirements for the quality of the web server are reduced, and a greater degree than normal of hard-coded workarounds is allowed. Support for multiple clients is also optional.

The web server must meet the following demands:

1. Initialize WebXiCache, DSP simulator etc.
2. Listen for TCP connections.
3. Receive an HTTP request from a client[13].
4. Parse HTTP requests and generate the corresponding `EXTENSION_CONTROL_BLOCK` structure.
5. Invoke the parser.
6. Use the response from the parser to form and send a response to the client.
7. Close connection and set up for a new request.

WebXiCache 2.6

WebXiCache is the data structure where the parser stores and retrieves values. WebXiCache also acts as the interface between the parser and the DSP in the LAN-XI G2 module. It is developed in C++ and has been provided for this project. Development of WebXiCache has been ongoing during the parser project period, and some features have been implemented as needed by the parser.

2.6.1 Structure

WebXiCache models a tree data structure. The tree is realized by each node pointing to its parent, first child and next sibling. If any of those do not exist, the value will be NULL. This is illustrated in figure 2.6.

It is thus worth noting that a node does not point to all its children. This would require a dynamic structure with an unpredictable size for each node, causing overhead on memory consumption and possible fragmentation.

WebXiCache interacts with the *Kiss*[14] operating system on the DSP. Therefore, WebXiCache uses the Kiss names for simple data types. Nodes in WebXiCache

[13] No need to support multiple simultaneous connections.

[14] Kiss is the operating system of the DSP. It was developed in-house several years ago, and has since then been adapted for use in new platforms several times.

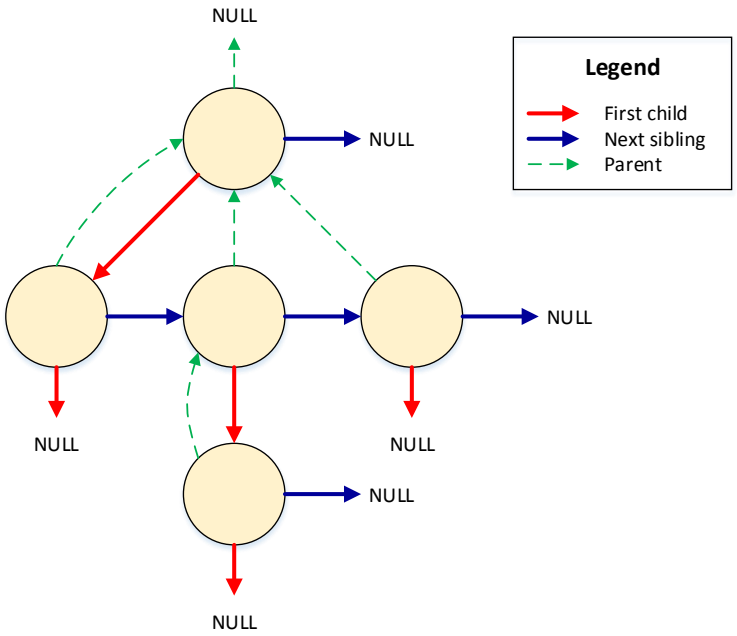


Figure 2.6: Illustration of the implemented tree structure in WebXiCache.

each have an associated value type, which is indicated by values from the `WebXiValueType_t` enum. These data types and their C and `WebXiValueType_t` representations are shown in table 2.1. The Kiss data types will also be used extensively in the Web-XI parser.

For boolean values, which are represented as 32-bit integers, *FALSE* is defined as 0 and *TRUE* is 1.

Nodes Nodes are modeled by the `WebXiNode_t` struct (only the members relevant to the parser are included):

`nodeId` (u32_t) – Unique ID for the node.

`name` (char*) – Name of the node.

`parent` (WebXiNode_t) – Parent node (NULL for the top node).

`nextSibling` (WebXiNode_t) – Next sibling in the chain. NULL on the last sibling.

`firstChild` (WebXiNode_t) – First child of the node. NULL on leaf nodes.

`isCached` (u8_t) – TRUE (1) if the value may be cached on the ARM CPU.

`isReadOnly` (u8_t) – TRUE (1) if the value of the node may only be modified by the DSP.

`maxVectorLength` (u16_t) – Maximum length of array. 0 indicates a scalar, greater than 0 indicates an array node.

Kiss data type	C data type	WebXiValueType_t value
Unknown		WebXiValueType_Unknown
void	void	WebXiValueType_None
bool_t	int	WebXiValueType_Bool
Unsigned integers		
u8_t	unsigned char	—
u16_t	unsigned short	—
u32_t	unsigned int	—
u64_t	unsigned __int64	—
Signed integers		
s8_t	char	—
s16_t	short	—
s32_t	int	WebXiValueType_Int32
s64_t	signed __int64	WebXiValueType_Int64
Floating point		
f32_t	float	WebXiValueType_Float32
f64_t	double	WebXiValueType_Float64
String etc.		
char*	char*	WebXiValueType_String
JSON metatype		WebXiValueType_JSON

Table 2.1: Kiss data types and their corresponding C data types.

`vectorLength` (u16_t) – Actual length of array.

`valueType` (WebXiValueType_t) – Type of the value.

`valueSize` (u16_t) – Total size of the value in bytes (value type size × number of values).

`value` (void*) – Pointer to value(s) of the node.

2.6.2 `WebXiCache` methods

The WebXiCache is instantiated and managed by the class `WebXiCache`. The following methods are available (methods that are not relevant to the parser are omitted):

`WebXiCache(int maxCount)` The constructor for the cache. `maxCount` sets the maximum number of nodes available in the data structure.

`~WebXiCache()` Destructor for the cache. Important to use to ensure resources being freed if the cache is to be taken down.

`WebXiNode_t* GetTopNode()` Returns a pointer to the top node of the WebXiCache.

`WebXiNode_t* FindNode(u32_t nodeId)` Returns a pointer to the node with the nodeId specified. If the desired node does not exist, NULL is returned.

`int SetValue(WebXiNode_t* node, void* value, u32_t valueSize)` Sets the value of a node. The value is not committed to the node until `Commit` is called. Several `SetValue` calls can be stacked before committing.
The method returns E_OK (0) if the operation is successful, otherwise an error code (non-zero).

`int Commit(u32_t triggerId, u64_t time)` Commits values set using `SetValue` to the cache. If `triggerId` and/or `time` is not 0, the commit will not take effect until the specified trigger or time has occurred.
The method returns E_OK (0) if the operation is successful, otherwise an error code (non-zero).

`void Rollback()` Cancels all `SetValue` calls stacked since last `Commit`. Used when a request sets a number of values and encounters an irrecoverable error. `Rollback` is then used to return to a known "good" state.

`int PerformAction(WebXiNode_t* node, char* action, char* argument,`

`u32_t triggerId, u64_t time)` Performs an *action* on the specified node with the specified argument. If `triggerId` and/or `time` is not 0, the action will not be performed until the specified trigger or time has occurred.
The method returns E_OK (0) if the operation is successful, otherwise an error code (non-zero).

`int RefreshValues(WebXiNode_t* node, bool_t refreshChildren)` Refreshes the values of the node given (if `refreshChildren` is FALSE), or for the node given and any descendants (if `refreshChildren` is TRUE). This ensures that the values for uncached nodes are up-to-date.
The method returns E_OK (0) if the refresh is successful, otherwise an error code (non-zero).

`void Lock()` Takes a lock, ensuring that no other process can change anything in the cache[15]. If the lock has already been taken, the method waits until it is released.

`void Unlock()` Releases the lock, enabling other processes to alter the cache.

`void LoadFile(char* path)` Load the object model file at the path specified into the cache. This is for test/prototype use only, as the object model will ultimately be published by the DSP.

`void StartKissPoller()` Starts the Kiss polling process, which manages the communication with the DSP for getting updates and values for the cache.

`void StopKissPoller()` Stop the Kiss polling process.

[15] Assuming every user of the cache uses the lock correctly.

2.6.3 Use

When the WebXiCache object has been initialized, the Kiss Poller should be started to enable the DSP to publish the object model to the cache. For the prototype, `LoadFile(...)` should be used to populate the cache.

Once ready, the cache can be used. A node can be found by getting the top node with `GetTopNode()`, and from there, a specific child can be found by accessing `firstChild` on the top node, and the children of the top node can be parsed through by following the `nextSibling` on the selected node until the desired node is found (`name` field matches the one being searched for) – or a NULL node is reached (no more siblings, the desired node does not exist). A simplified sequence diagram of this process is shown in figure 2.7.

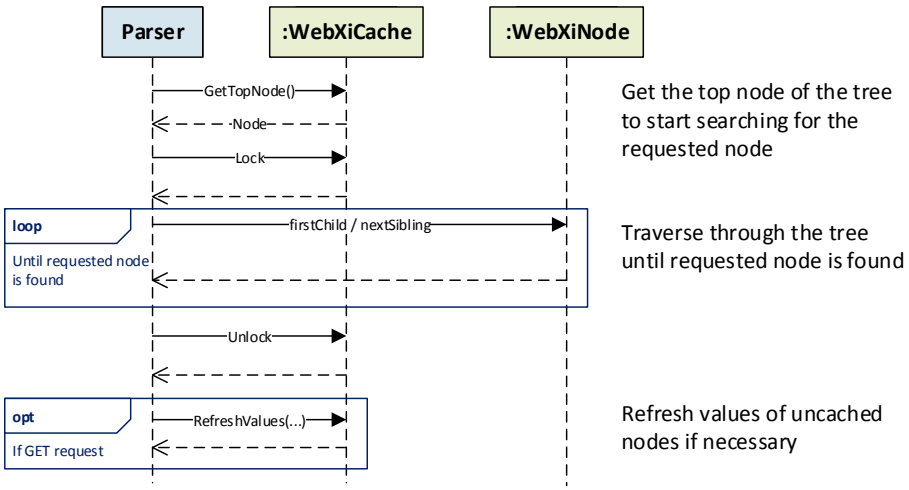


Figure 2.7: Simplified sequence diagram showing the interaction between the parser and WebXiCache when searching for a given node in the object model.

When it is important for the values in the cache to be current (ie. uncached nodes are up-to-date), `RefreshValues(...)` can be used. This can be done on the node or subtree which should be refreshed – depending on the `refreshChildren` value.

Node values can be accessed directly through the `value` field. The type of the value is determined by the `valueType` field. If the `maxVectorLength` value is greater than 0, the node holds an array.

For setting the value of a node `SetValue(...)` is used. The method needs a pointer to the value to be set, and its size in bytes (the size is the size of each value × the number of values – which is 1 for scalars).

`SetValue(...)` can be called for each value to be set. When all desired values have been set, they can be committed to the DSP with `Commit(...)`. If the `triggerId` and

`time` of the call are both 0, the values are committed immediately. Otherwise, the values will not be committed until the specified trigger ID and/or time has occurred.

During operations where it is important to ensure consistency – that no values are changed while the operations are going on – the lock can be taken using `Lock()`. This call will wait until the lock is available (if it is already taken). When the operations are done, the lock can be released using `Unlock()`.

An illustration of the general flow for setting values in WebXiCache is shown in figure 2.8.

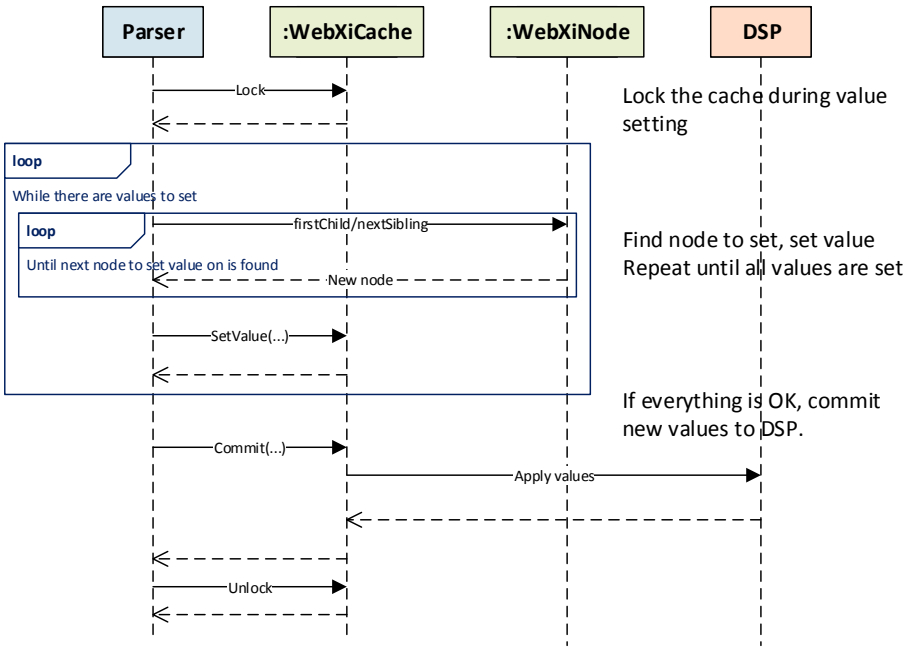


Figure 2.8: Simplified sequence diagram showing the interaction between the parser, WebXiCache and DSP setting node values.

2.6.4 Shortcomings

In part due to the lack of an actual DSP, the support for actions, triggers, time and delay is very limited. It will be possible to implement all but delay, but the there will be nothing or very little to see when they are used, limiting the verification/testing possibilities.

DSP Simulator 2.7

For the prototype, a simulation of the connection to a DSP is provided alongside the cache. This provides a very simple simulation of the handling of messages done by the DSP.

The DSP Simulator is used within the cache, and the parser shall not use it directly. It is, however, necessary to initialize the simulator for it to work. This is done by calling the `StartDspSimulator()` function. Stopping it is done using `StopDspSimulator()`.

Chapter 3

Design

This chapter describes in further detail the strategies and structure behind the prototype implemented.

Flow charts are used extensively to illustrate algorithms and structure of the solution, rather than long explanations. The diagrams make use of color coding to make it easier to identify where an operation takes place, or what kind of operation is being performed. The color coding is not the same in all diagrams, but in those diagrams where the color coding serves a specific purpose, a legend is displayed.

Overview 3.1

Splitting the description and development of the prototype into smaller logic entities is crucial to keeping an overview.

As an actual ISAPI template has not been available for development it has been necessary to seek a flexible solution which makes it easy to modify the sources of certain information – such as the HTTP request body – without having to rewrite the entire parser.

The approach taken in this development has been from the outside and in, ie. first examining and developing the simple prototype web server before the parser development is started. This has further been necessary as the WebXiCache has been under ongoing development parallel to the parser, meaning that connecting the parser to the cache has not been an option from the beginning.

Development of the parser is split into 3 parts. One is the entry point of the parser – the ISAPI extension entry point – another is retrieving data from the cache (GET requests), and the last is setting values in the cache (PUT requests). The 3 parts do not have much functionality in common.
The entry point manages parsing of the HTTP request, determining what is to be done and mapping the requested path to a node in the cache. It then calls one of the two other parts which either retrieve nodes and values from the cache and returns it represented as JSON, or parses through JSON data and sets values accordingly in the cache.

Web server 3.2

The web server is needed only for the PC prototype, and the focus on stability and absolute correctness is reduced. Only the features explicitly needed to make the prototype behave sufficiently like the final implementation in LAN-XI G2 are implemented, and not much effort is put into making the implementation "pretty" or easy to maintain.

For development of the web server a simple example has been provided. This web server sets up a socket and listens for a connection. When a connection is established, it receives one chunk of data from the receive buffer containing the HTTP request, writes back a fixed response header and body, and closes the connection. The web server can handle multiple simultaneous requests.

For the parser prototype, the web server is modified to instantiate the WebXiCache with the development object model from Figure 2.3 on page 13, parse the

received HTTP request and populate an EXTENSION_CONTROL_BLOCK structure with the contents.
The parser is called with this structure as an argument, and when parser execution is ended, the connection to the client is shut down and the web server listens for new connections.

In order to enable the parser to write data back to the client, a WriteClient function is needed as specified in the EXTENSION_CONTROL_BLOCK.

3.2.1 Parsing an HTTP request into an EXTENSION_CONTROL_BLOCK

Again, the following request sent by a client is considered:

```
PUT /WebXi/Acquisition/Channels/1?AwaitTrigger=3&Time=5000 HTTP/1.1
Content-type: text/json
Content-Length: 54

{
  "Gain": 5.3,
  "Description": "Monitor channel"
}
```

Parsing of the header is done line-by-line. The first line has a special format, while the rest of the lines each follow the same structure.

The first line is split at spaces. The first part is the HTTP Method, and is stored in the lpszMethod field in the EXTENSION_CONTROL_BLOCK.
The next part is the full request part and optional query. This is split at the questionmark (?), if it exists, and the first part is stored in lpszPathInfo, while the other part is stored in lpszQueryString. NULL is stored if there is no questionmark.
The third and final part of the first line is the protocol and version. This is ignored here.

Subsequent lines follow the *[name]:[value]* convention. Only the *Content-Type* field is to be supported.
For each line, the line is split at colon (:). The first part is then compared to *Content-Type* – and if it matches, the value (second part of the line) is stored in lpszContentType.

Header parsing continues until two consecutive line breaks (\r\n) are encountered. The rest – terminated by a NULL character – is the body of the request. This is stored in lpbData. The request may have no body. In this case, the double line breaks are followed immediately by the NULL character.

3.2.2 Getting full request data from the client

When a client sends a request containing a body (in this case it would be a PUT request with JSON data attached), the client often sends two chunks to the web server: 1) The header and 2) The body.

Under certain circumstances the server will receive the header and fetch it before the body is received, meaning that the body is not received in the first try – otherwise everything is received before processing begins.

To make sure the entire request is received before processing begins, the *Content-Length* field in the header should be examined – if it exists. The value of Content-Length is evaluated.

If Content-Length is greater than 0, there must be a body in the request. The delimiter between header and body (double line break) is found, and if the next character is NULL, no body was sent in the first chunk, and the body should be received separately and appended.

3.2.3 HTTP response

The HTTP response to a request is slightly different to the request. Instead of HTTP Method, path/query and protocol/version, the first line contains protocol/version and a status code/message (See the ones used in this context in section 2.1.7).
The following lines and optional body resemble the request format:

```
HTTP/1.1 200 OK
Content-Type: application/json
Content-Length: 22

{
  "ModuleId": 621
}
```

The header tells that the request was successful (200 OK status), and that the returned body contains JSON and has a length of 22 bytes. The body follows after a double linebreak and shows that the value of the *ModuleId* node is 621.

Parser entry 3.3

The parser entry is the implementation of the ISAPI extension and has the signature `DWORD WINAPI HandleRequest(EXTENSION_CONTROL_BLOCK*)`

The `HandleRequest` function manages the tasks that are common to all requests sent to the parser, regardless of type. This includes – as illustrated in figure 3.1:

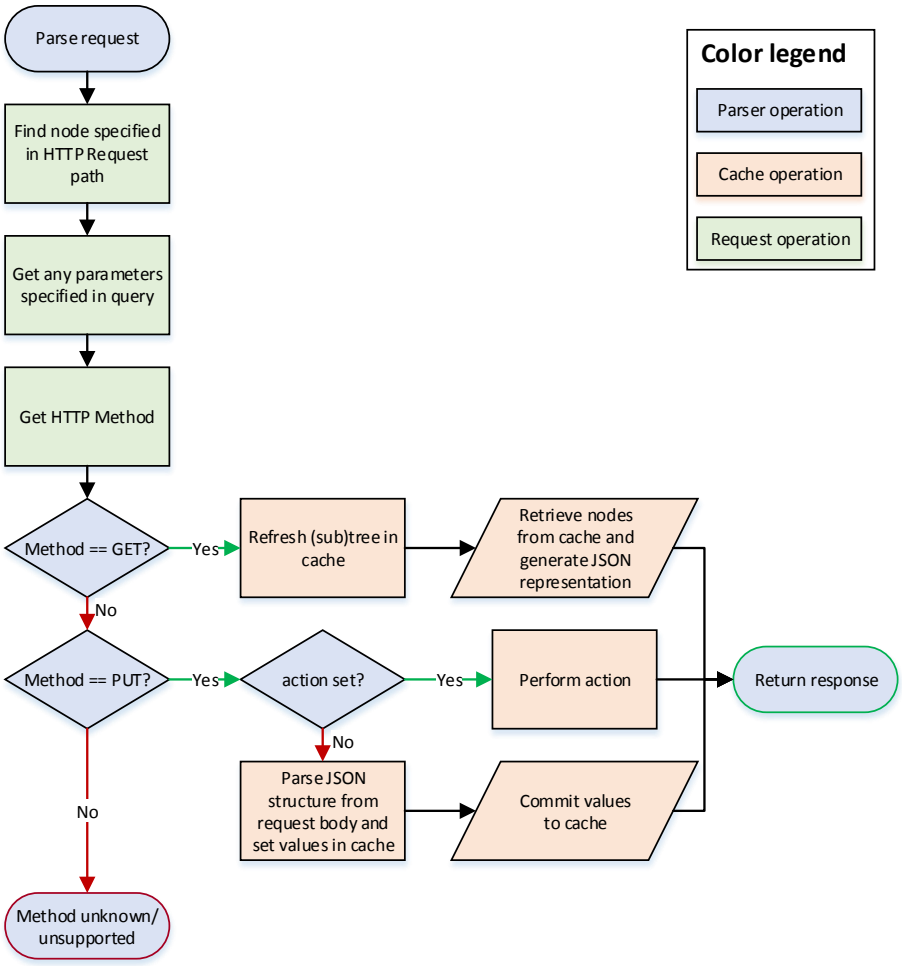


Figure 3.1: Rough overview of the flow of the parser.

1. Find the requested node from the path in the HTTP request.

   Verify that the requested node exists.

2. Parse query fields:

   `recursive` flag (true/false, default false).

   `action` (string, default empty).

   `argument` (string, default empty).

`awaittrigger` (integer, default 0).

`time` (64-bit integer, default 0).

`delay` (64-bit integer, default 0).

3. Parse HTTP Method (GET/PUT/...).

   If method is GET, make sure the requested nodes are up-to-date and handle control to the GET handler.

   If method is PUT, and an *action* is set, perform that action with any *argument* set.

   If method is PUT, and an action is not set, handle control to the PUT handler. Afterwards, if everything has succeeded, commit new values to the cache, otherwise rollback to return to the previous state.

   If method is neither GET or PUT, return an error.

4. Send a response to the client.

3.3.1 Find the requested node (`RequestPathToNode(...)` function)

Finding the requested node is done in a function with the signature
`WebXiNode_t* RequestPathToNode(char* requestPath)`

Given the request path (e.g. */WebXi/Acquisition/Channels/1*) the corresponding node in the object model should be found. This is done by splitting the path into parts at every slash (/), first returning "WebXi", next "Acquisition" etc.
On the object model, the top node of the tree is used as the point of reference. The approach is illustrated in figure 3.2.

The name of the top node of the object model is compared to the first part of the path, and then, for each part of the path, the children of the previous node selected in the object model are walked through until a name matches the part of the path.

If at any point no children match the corresponding part, the requested node does not exist in the object model, and `NULL` should be returned to indicate this.
When the requested node is found, indicated by no more parts of the path left to evaulate, the node found (current node) is returned.

3.3.2 Parse query fields (`GetQueryParams(...)` function)

The query is a string of field-value pairs separated by ampersands (&): Each field and value is separated by an equals sign (=), but the equals sign and the value may be omitted.
Example: `action=Detect&argument=All&time=512`

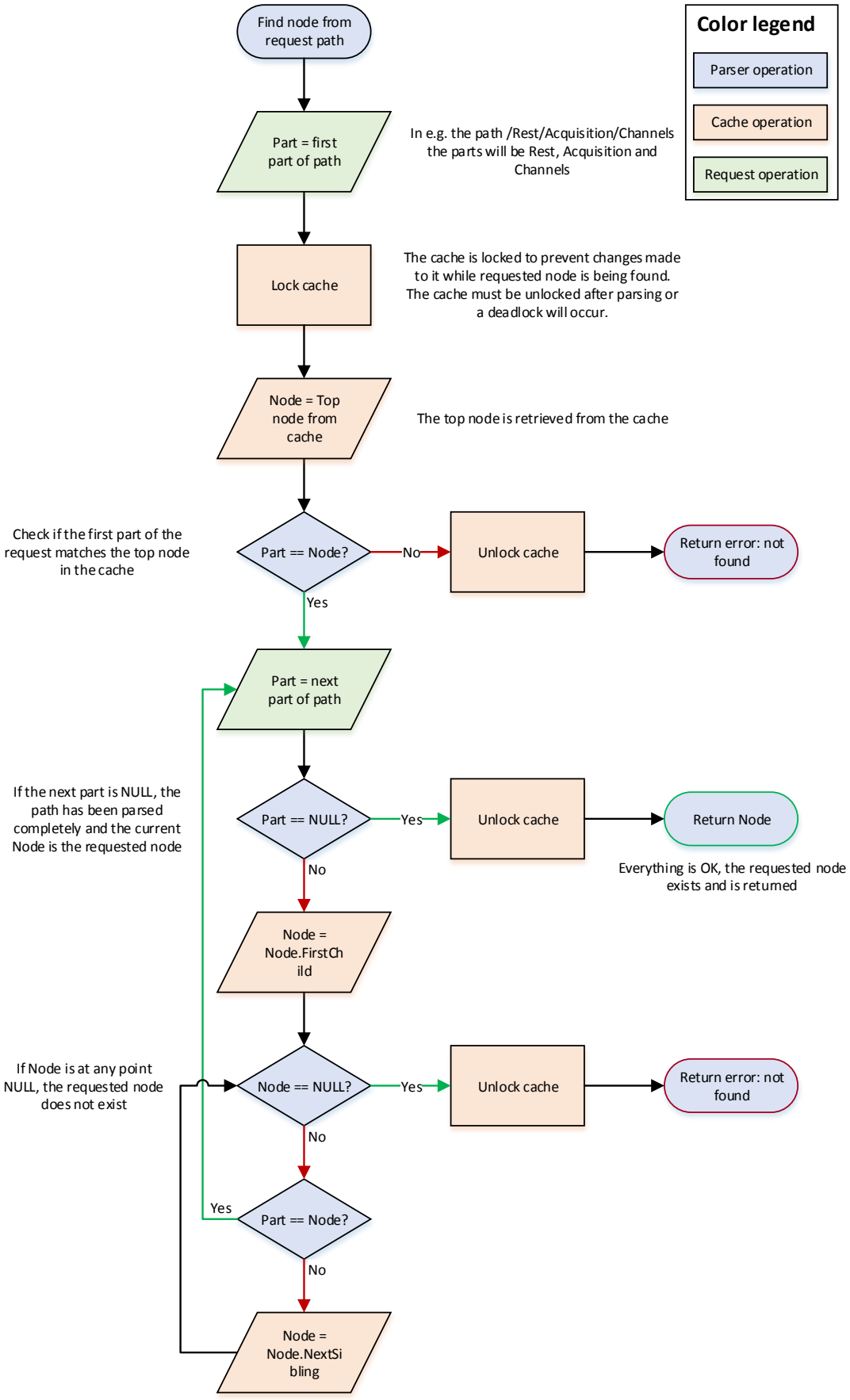


Figure 3.2: Flow chart of parsing the request path to a node in the object model.

The parameters to be identified are:

recursive (bool) – set to true if value is "true" or unset (the query would be ?recursive).

action (string)

argument (string)

awaittrigger (Int32)

time (Int64)

delay (Int64)

Due to the limited number of fields to support, the `GetQueryParams` function takes pointers to each variable as arguments for returning the values – as well as a pointer to the query string. The signature of the function is
`void GetQueryParams(char* query, bool_t* pRecursive, char* pAction, char* pActionArgument, u32_t* pTriggerId, u64_t* pTime, u64_t* pDelay)`

- `query` is a pointer to the query string to parse.
- `pRecursive` is a pointer to the boolean where an identified recursive value should be stored.
- `pAction` is a pointer to the string where the action field value should be stored.
- `pActionArgument` is a pointer to the string where the argument field value should be stored.
- `pTriggerId` is a pointer to the integer where the awaittrigger field value should be stored.
- `pTime` is a pointer to the integer where the time field value should be stored.
- `pDelay` is a pointer to the integer where the delay field value should be stored.

Parsing the query is done by examining each field-value pair. As only a limited number of different fields are necessary for the parser to understand, a simple construction can be used, comparing each field with the ones to be supported. Any unsupported fields will be ignored. All evaluations are case insensitive.

When a supported field is identified, the value is parsed separately, according to the field. The values are in some cases passed on as strings, and in other cases converted to integers. The values found are stored at the locations pointed to by the function arguments. The values must be initialized with their default values prior to running `GetQueryParams(...)`, as only values identified in the query are set.

The flow of `GetQueryParams(...)` is illustrated in figure 3.3.

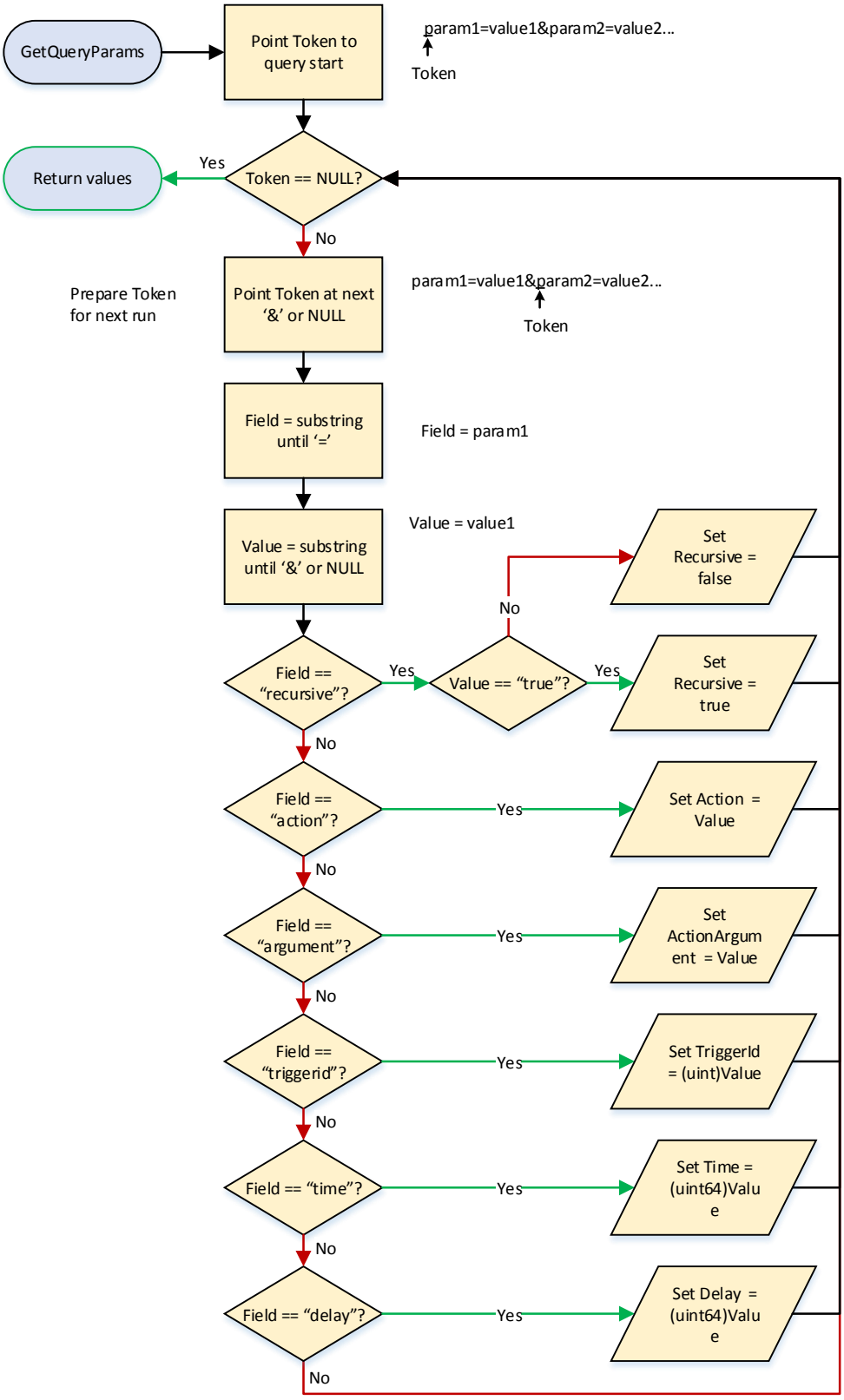


Figure 3.3: Flowchart for the operation of `GetQueryParams(...)`. Recursive, Action, ActionArgument, TriggerId, Time and Delay are all passed to the function as pointers, and are thus set directly in the calling scope.

3.3.3 Parse HTTP Method

As only GET and PUT methods are supported by the parser, a string comparison is done to determine the action to take.

If an unsupported or unknown method is encountered, an HTTP error status *405 METHOD NOT ALLOWED* is returned and no action is taken. For GET and PUT, the appropriate action is taken, preparing and handing over control to the appropriate handler.

Parsing GET requests 3.4

The GET handler takes a root node as argument and returns its output in a `char*` buffer. The signature is
`void HandleGET(WebXiNode_t* rootNode, bool_t recursive, char* pResponse)`

- `rootNode` A pointer to the root of the subtree – or the leaf – to be handled. This typically comes from `RequestPathToNode(...)`.
- `recursive` If TRUE, and if `rootNode` points to a branch node, the entire subtree from the root node should be returned. Otherwise, only the children.
- `pResponse` The string buffer for storing the response to the client.

Depending on the request, there are three different main scenarios to cover, depending on the request (this is also covered in Section 2.1.4, including examples):

1. The requested node is a leaf node.
2. The requested node is a branch node, and recursive is set false.
3. The requested node is a branch node, and recursive is set true.

In the first case – with a leaf node requested – a JSON object containing a name:value pair consisting of the node name and its value should be returned. This can be done directly.

In the second case the children of the requested node should be returned in a JSON object. This is done by entering the child of the requested node, opening a JSON object (), and for each sibling output its name:value representation (and a delimiter (,) if more siblings exist). The output is terminated by closing the JSON object ().

The third case is more complex. A DFS[16] approach is used to cover all descendants of the requested node in the right order.

Due to the limited resources and lack of dynamic data structures in the environment, it is not feasible to keep track of which nodes have been visited during the run. It is, however, possible to do a systematic run through the nodes and keep track of the "direction of travel" – whether the algorithm is processing upwards or downwards in the tree.

The flow of the GET handler is illustrated in figure 3.4 and 3.5.

3.4.1 Formatting value output

As value formatting is used in many places when generating JSON representation, it has been extracted into a separate function:
`void NodeValueToString(WebXiNode_t* node, char* pOutput)`

- `node` The node whose value should be formatted.
- `pOutput` Pointer to the output buffer where the formatted value should be appended to.

For scalar nodes, the node value is simply appended to `pOutput` with formatting matching the node value type. Ie. string nodes have the value printed as a string surrounded by quotes etc.

Floating point nodes are printed as decimals with 7 decimal digits for 32-bit floats and 16 for 64-bit floats[17].

For array values, the values are formatted by making a pointer to the value with the correct type. When this pointer is incremented it will point to the next array value, and this is done until the `vectorLength` times has been reached. A comma (,) is put between each value, and square brackets ([and]) encase the array.

[16] **D**epth-**F**irst **S**earch (DFS) is an algorithm for traversing a tree structure, starting at the root and exploring as far along each branch as possible before returning back and trying different branches.

[17] These representations should match the precision of the IEEE 754 32- and 64-bit representations: https://en.wikipedia.org/wiki/IEEE_floating_point

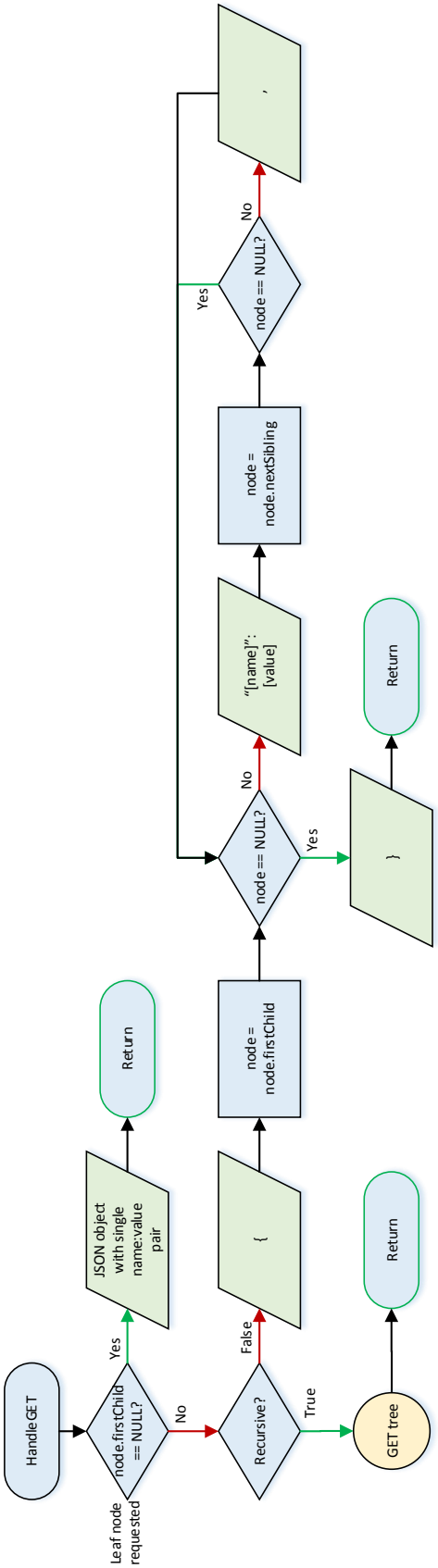


Figure 3.4: Main flow of the GET handler. Determines which of the three cases of output types to apply. The *branch node, recursive true* case is shown separately in figure 3.5.

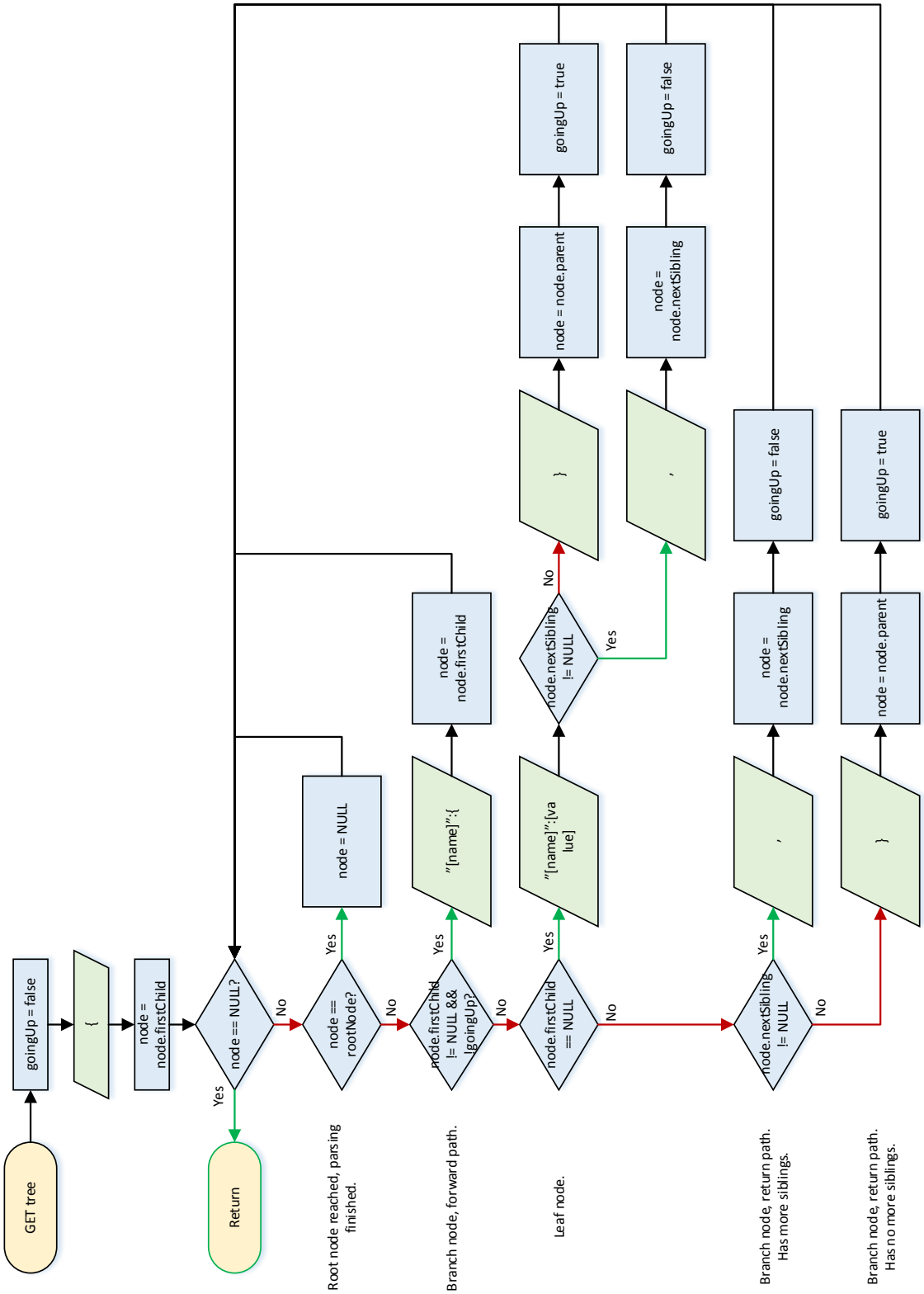


Figure 3.5: Flow of the GET handler for requested branch node with recursive set true. Referred to in figure 3.4.

Parsing PUT requests 3.5

For parsing the PUT requests, a lexical scan approach is used. The solution is custom-built from scratch to enable as direct interaction with the WebXiCache as possible.

A tokenizer will walk through the input JSON data until it recognizes a single character or sequence of characters (token). It will then return the kind of token – and if it is a value carrying token, the value. Values are parsed into their native types, ie. Int32, Float64, char* etc.

The parser can now handle the token found, set a value in the cache, select a node etc. Afterwards, the process is repeated. The tokenizer is set to scan for tokens from where it left off and so on.

3.5.1 Tokens

The tokens identified by the tokenizer have some characteristics, which are described here. Each token type has limitations for the sequence – which token types are allowed before and after – as well as value characteristics.

Token types are defined in the enum `JSONToken_t`:

- `JSONToken_BrL` – {
- `JSONToken_BrR` – }
- `JSONToken_SqL` – [
- `JSONToken_SqR` –]
- `JSONToken_Colon` – :
- `JSONToken_Comma` – ,
- `JSONToken_String` – "Some string"
- `JSONToken_JSON` – {"name":value}
- `JSONToken_Int` – 42
- `JSONToken_Float` – 3.141
- `JSONToken_True` – true
- `JSONToken_False` – false
- `JSONToken_Null` – null

More details can be found in appendix E.

3.5.2 Tokenizer

The tokenizer uses a pointer to walk through the JSON structure character by character. When it encounters a recognizable token it is evaluated and processed.

Conversion from the character data (e.g. a sequence of number characters) in JSON to the appropriate data type (e.g. an integer) is handled by the tokenizer. To make sure the right data types are returned (an *Integer* can be both `Int32` and `Int64` – and even `Float32` and `Float64`), the tokenizer must know something about the value type expected to make sure the tokenizer returns the value in the correct format, avoiding the need for unnecessary double conversions.

The tokenizer returns a JSON token type, a value (which may be nothing) and a value length (size in bytes). Furthermore, a status is returned, enabling the tokenizer to tell when it has reached the end of the JSON structure, if an error has occurred, such as invalid types for some expected value type etc.

The signature for the tokenizer is
```
JSONStatus_t GetToken(char** ppInput, JSONToken_t* pType,
WebXiValueType_t valueType, void* pValue, u32_t* pLength)
```

- `ppInput` A double-pointer used to indicate where the tokenizer should start scanning. The pointer is updated during tokenizer execution, so *ppInput after execution points to the point where the tokenizer left off.
- `pType` Pointer to the variable where the token type should be returned.
- `valueType` The value type expected for the node currently being processed.
- `pValue` Pointer to the value output buffer where any value identified in the token found is returned.
- `pLength` Pointer to the variable where the length of any variable found will be returned.

Example

The following small piece of JSON is examined:

```
{
  "Gain": 1.2130495
}
```

The tokenizer is called with `*ppInput` pointing to the first character ({). It immediately recognizes the character and returns the value type `JSONToken_BrL`, and

`*ppInput` is pointed at the next character (the line break). The tokenizer returns `JSONStatus_TOKEN_FOUND` as status. This is done under normal operation (no errors occurring) until the JSON data is terminated.

Next time called, the tokenizer ignores the line break characters and the spacing of the secondline. The first recognized character is the quote ("). This is recognized as being a string, the `pValue` is populated with "Gain" (without the quotes). `pLength` is set to 4, and the returned value type is `JSONToken_String`. The `*ppInput` pointer now points to the colon (:).

The next token found by the tokenizer is the colon (:), and the `JSONToken_Colon` type is returned.

Next run returns a floating point number. The `valueType` argument determines if it is `Float32` or `Float64`. `pValue` will contain the 32- or 64-bit floating point number and the `pType` will be `JSONToken_Float`.

Next and last tokenizer run will return the token type `JSONToken_BrR`, and as the input is terminated at the new `*ppInput` character, the tokenizer returns the status `JSONStatus_EOF`.

3.5.3 Parsing JSON

Parsing of the JSON structure follows the procedure described in Section 2.1.2. Values are set directly in the cache without an intermediate representation, saving resources and removing an additional need for dynamic memory allocation, and error handling relies on the rollback function of the cache.

Throughout the parser operation, a pointer is always maintained to the *current* node. This pointer walks through the cache in parallel with the nodes identified in the JSON data.

The tokenizer is used to find the first token in the request body. This should be an opening bracket – `JSONToken_BrL`. The root node (as specified by the request) is set as the current node.
The tokenizer is used again, and the desired action is decided based on the new token type and the previous one (which would be `JSONToken_BrL` from the first run).

For each run of the tokenizer, and based on the token type, it is determined if the token type is valid (sequence is valid JSON), and any actions that should be done are performed.
The semantic sequence allowed by the parser is shown in figure 3.6.

All valid successions are shown in the diagram. If a token type is succeeded by one which is not connected, a semantic error has occurred and the JSON data given is not valid.

Terminator nodes are not actual token types, but rather tokenizer states.
Control nodes are mostly used for semantic validation. The parser could be designed to ignore the comma (,) and colon (:) token types completely and still work as intended, but the JSON semantics might then be incorrect, and potentially the results might not be quite as intended by the client.
Node selection nodes show token types where the current node pointer may be changed, ie. to a child, sibling or parent.
The *value* nodes are nodes where the tokenizer returns a value. At these points a value may be set in the cache. Note that the *String* node has a dual function, both acting as an identifier and as a value. The role of the node is determined by the sequence; if it is preceded by a *Colon* (:), it is a value, otherwise it is an identifier.

The diagram illustrates only one check to secure the sanity of the input. Some sequences which may appear valid in the diagram may be invalid, due to wrong context. E.g. a String token may only be followed by a Colon (:) token if the String was used as a node selector – and an Int must only be followed by closing square bracket (]) if the Int was part of an array.

Setting a value in the cache includes checking if the node is read only, and that – for arrays – the array length does not exceed that allowed in the cache node.

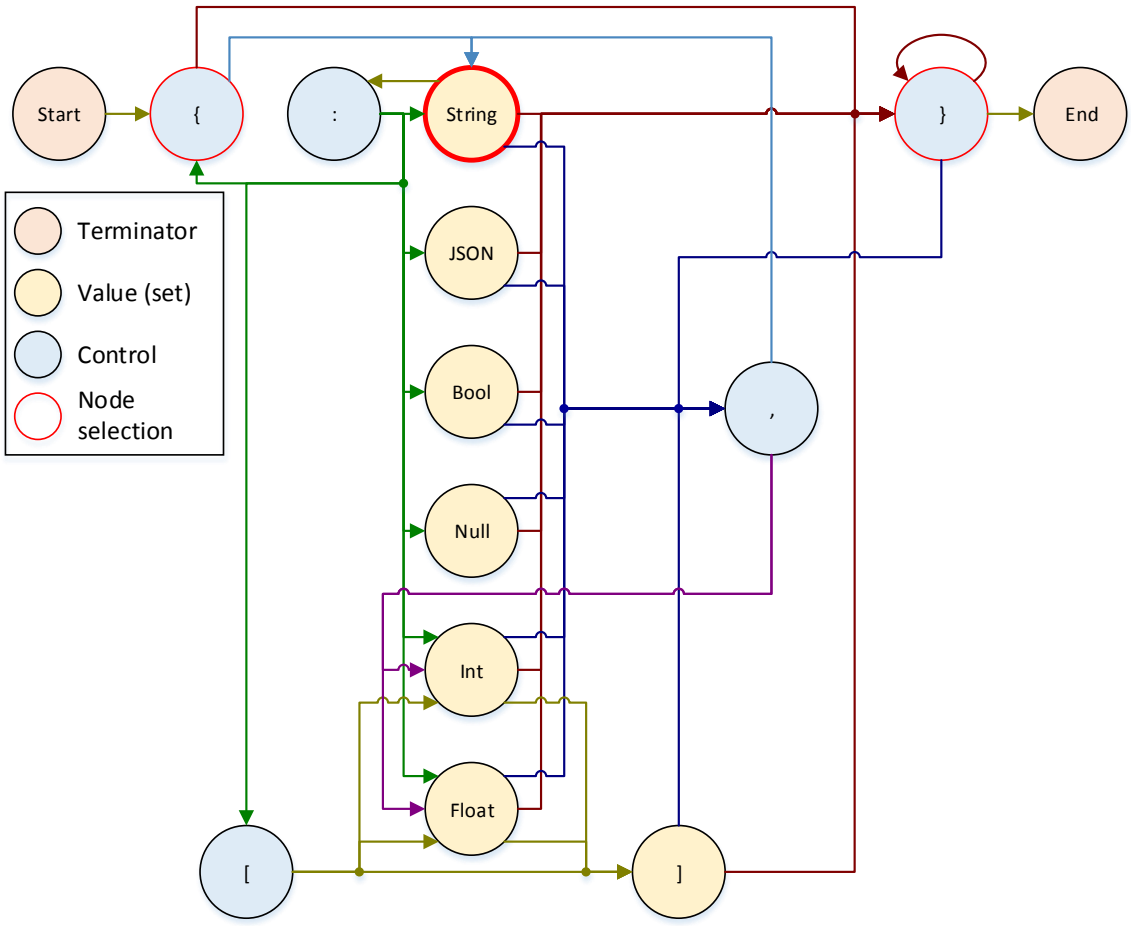


Figure 3.6: Diagram illustrating the valid sequences of JSON tokens, and categorization of the actions taken on each node.

Chapter 4

Implementation

This chapter aims at giving an idea of the structure of the implementation of the project, including the distribution of entities/tasks, some implementing strategies and other points.

As development has been done using lower-level C code, most of the code is rather generic. Only a few condensed code chunks have been included in the description to illustrate the solution, as full examples would require very large chunks of code, harming the overview.

General implementation strategies 4.1

Some general concepts have been used throughout the project – such as the development setup and distribution of tasks and code structure.

Due to the limited resources available on the ARM, and to avoid any unnecessary memory fragmentation, the parser has been implemented without use of `malloc`[18]. Instead, static allocations have been used with fixed sizes. This may result in more memory being used, but it is predictable and not as prone to fragmentation as dynamic allocation.

4.1.1 Visual Studio solution

The parser is implemented in a single solution in Visual Studio, containing several projects. Some of these projects compile into an executable program, others provide libraries of data structures and functions.

`Cache` (Provided) The WebXiCache implementation.

`CacheTest` (Provided) A small program for testing the WebXiCache (has been used for development of the cache).

`DSPSim` (Provided) The DSP Simulator.

`Kiss` (Provided) Data types etc. used for supporting the DSP communication.

`Parser` The actual Web-XI parser ISAPI extension.

`ParserTest` A small program for running pseudo-unit test on the parser. Covered in Section 5.1.

`TestServer` The custom web server implementation used to emulate the Windows EC IIS web server on the PC.

The Parser project contains the following implementation files:

WebXiParser.cpp The ISAPI entry point.

HandleGET.cpp Handles GET requests, parsing through the cache and generating a JSON representation.

[18] `malloc` can be used to dynamically allocate memory on the heap. Spaces allocated must be explicitly freed again when they are no longer used, introducing the potential for memory leaks. May also fragment the memory.

HandlePUT.cpp Handles PUT requests, parsing through the JSON structure given and setting values in the cache.

JSONToken.cpp Tokenizer for JSON.

4.1.2 Code documentation

All code is documented using comments throughout the code to ease the understanding. Furthermore, specially formatted comments have been made for every function and data structure for use with Doxygen[19]. Doxygen comments have also been used in the code provided by others.

Doxygen documentation is useful for finding out where a certain function is implemented, getting information about a function, or getting an idea of the structure of the system developed.

4.1.3 Code structure

In general, switch-case statements have been sought used rather than long if-else constructions. When applicable, the switch-case structure gives much better readability of the code, and the limits are easier to find.

Request path to node 4.2

Parsing a request `path` to the corresponding node in the cache is implemented as follows:

```
// Lock the cache to prevent nodes from being changed while finding the requested node
cache->Lock();

// Verify the first part of the requested path matches the root node
char* requestName = strtok_s(path, "/", &path);
if (_stricmp(requestNode->name, requestName) != 0)
{
    // First part of the requested path does not match the root node
    // Unlock cache and return null
    printf("ERROR: Root node does not match. First level requested: %s\n", requestName);
    cache->Unlock();
    return NULL;
} // if

// Process through the rest of the path
requestName = strtok_s(path, "/", &path);
while (requestName != NULL)
{
```

[19] `www.doxygen.org` – Doxygen can read through code in many programming languages and generate a complete code documentation with cross references etc.

```
    // Enter the first child of node
    requestNode = requestNode->firstChild;
    // Go through the siblings until the name of the node matches the part of the requested path
        or no more siblings exist
    while (requestNode != NULL && _stricmp(requestNode->name, requestName) != 0)
    {
        requestNode = requestNode->nextSibling;
    } // while
    if (requestNode == NULL)
    {
        // No more siblings exist. The requested name does not exist. Processing cannot proceed.
        // Unlock cache and return null
        printf("ERROR: Request node not found\n");
        cache->Unlock();
        return NULL;
    } // if
    // Proceed with next part of the path.
    requestName = strtok_s(path, "/", &path);
} // while

// Unlock cache, allowing the subtree to be refreshed
cache->Unlock();

return requestNode;
```

`strtok_s` is used to split the request path at the slashes (/) from left to right, giving a single node name each time it is used. This name is compared to each sibling in the level of the cache reached, and if a match is found, the examined part has been found and the process can be repeated.
The requested node has been found when no more parts exist in the request path, indicated by `strtok_s` returning a NULL pointer instead of a pointer to a string.

Locking and unlocking of the cache has also been shown in the code fragment. Note that all paths in the code resulting in a return statement has an `Unlock()` operation. If not, deadlocks may occur at runtime.

GET handler 4.3

The GET handler uses an if-else construction to determine if a leaf node is requested, or a branch node is requested, and if recursive is set or not.

When a branch node is requested with recursive set TRUE, a while loop with another large if-else construction comes into play. This if-else construction implements the DFS without any state information:

```
bool_t goingUp = FALSE;
while (node != NULL)
{
  if (node->nodeId == rootNode->nodeId) // We've reached the top node after parsing
  {
    node = NULL;
  }
  else if (node->firstChild != NULL && !goingUp) // Branch node (on forward path)
  {
```

```
    ...
    node = node->firstChild;
    goingUp = FALSE;
  }
  else if (node->firstChild == NULL) // Leaf node
  {
    ...
    if (node->nextSibling == NULL) // Last leaf in branch
    {
      ...
      node = node->parent;
      ...
      goingUp = TRUE;
    }
    else // More nodes exist at this level
    {
      ...
      node = node->nextSibling;
      goingUp = FALSE;
    } // if
  }
  else if (node->nextSibling != NULL) // Branch node (on return path), has more siblings
  {
    ...
    node = node->nextSibling;
    goingUp = FALSE;
  }
  else // Branch node (on return path), no more siblings
  {
    ...
    node = node->parent;
    goingUp = TRUE;
  } // if
} // while
```

The cases identified in the construction are:

1. Root node reached (parsing has reached the end).
2. Node has children, and the parsing is on a downward path (ie. the children have not yet been visited).
3. Node is a leaf node – with two specializations:

   Node has no next sibling, ie. all the siblings have been handled.

   Node has a next sibling, ie. more siblings need to be handled.
4. Node is a branch node with a next sibling, and the parsing is on the return path.
5. Any uncovered case (branch node on return path without a next sibling).

Tokenizer 4.4

The tokenizer evaulates the `ppInput` string character-by-character. Each character is evaluated to see if it matches a know token og beginning of a token, and processing is done accordingly, as shown below (much has been removed to better show the overall structure of the tokenizer):

```
// Walk through the string buffer char by char until it ends or a token has been found.
while (**ppInput != '\0' && !found)
{
    // Default value for found - it is set false later if no token is found at this character.
    found = true;
    // Evaluate the current character position in the string buffer.
    switch (**ppInput)
    {
        case '{': // Opening bracket found.
            if (valueType == WebXiValueType_JSON)
            {
                *pType = JSONToken_JSON;
                // Find JSON value and set pValue etc.
                ...
            }
            else
            {
                *pType = JSONToken_BrL;
            } // if
            break;
        case '}': // Closing bracket found.
            *pType = JSONToken_BrR;
            break;
        ...
        default:
            // Character not recognized - whitespace or other ignored.
            // Set found to false to search through next character.
            found = false;
    } // switch

    // Continue to next character
    (*ppInput)++;
    if (found)
        break;
} // while
```

Value conversion from string to integer and floating point numbers is done using `atof(...)` for floating point, and `atoi(...)`/`_atoi64(...)` for 32-bit/64-bit integers. These functions do not support exponential representation of numbers (e.g. 3E8). Support for exponential number number representation has been left out initially.

PUT handler 4.5

The main structure of the `HandlePUT` function is very similar to that of the tokenizer. The example below illustrates the main loop of the PUT handler. In every run, the tokenizer is used to find the next token. Based on the token type found, different actions are taken. These actions include semantic checks by evaluating the previous token (which is always stored at the end of the do-while block). The loop runs as long as no error occurs, and as long as tokens are found:

```
do // while (tokenStatus == JSONStatus_TOKEN_FOUND && setValueStatus == SetValueStatus_OK)
{
    // Get the next token
    tokenStatus = GetToken(&input, &token, (currentNode != NULL ? currentNode->valueType :
        WebXiValueType_None), pValue, &valueSize);
    switch (token)
    {
        case JSONToken_BrL: // {
            switch (previousToken)
            {
                case JSONToken_Start:
                    currentNode = rootNode;
                    break;
                case JSONToken_Colon:
                    break;
                default:
                    // Illegal token sequence
                    sprintf_s(pResponse, RESPONSE_MAX_LENGTH, "{\n\t\"Error\": \"Unexpected {\"\n
                        }");
                    return HTTPStatus_Bad_Request;
            } // switch
            break;
        case JSONToken_BrR: // }
            switch (previousToken)
            {
                case JSONToken_BrL: // Valid previous tokens
                case JSONToken_BrR:
                case JSONToken_JSON:
                case JSONToken_String:
                case JSONToken_Int:
                case JSONToken_Float:
                case JSONToken_True:
                case JSONToken_False:
                case JSONToken_Null:
                case JSONToken_SqR:
                    currentNode = currentNode->parent;
                    break;
                default: // Invalid previous token
                    // Illegal token sequence
                    sprintf_s(pResponse, RESPONSE_MAX_LENGTH, "{\n\t\"Error\": \"Unexpected }\"\n
                        }");
                    return HTTPStatus_Bad_Request;
            } // switch
            break;
    ...
    }
    previousToken = token;
} // do
// Continue as long as tokens are found, and values are set without errors
while (tokenStatus == JSONStatus_TOKEN_FOUND && setValueStatus == SetValueStatus_OK);
```

Chapter 5

Test

Testing of the Web-XI parser has been done using several strategies. The focus has been on targeting the Web-XI parser itself, and the cache and test server have not been targeted, although they are to some extent included when testing the parser.

The tests performed include pseudo-unit test performed in the C environment and external tests validating the entire web server and parser implementation.

During development manual, ie. non-scripted, tests have also been used.

Internal tests 5.1

The internal tests are part of the main parser solution and have direct access to functions etc. within the parser, enabling unit tests examining the inner parts of the system separately. The connections are illustrated in figure 5.1.

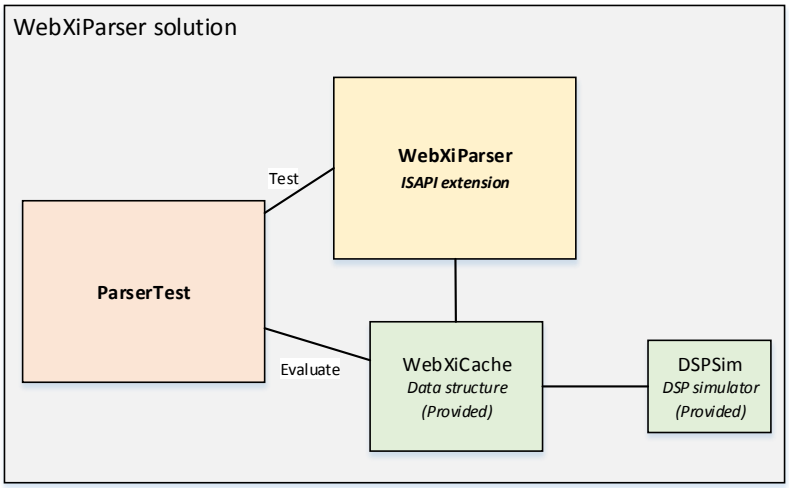


Figure 5.1: The logic connections in the internal test. The prototype web server is not part of the test chain, and the `ParserTest` is able to evaluate data in `WebXiCache` directly.

Attempts to find a suitable unit testing framework for the parser have not been successful. Therefore, a simple pseudo-unit testing framework has been developed for asserting values from nodes in the cache against specified expected values. These functions print out an error if an assertion fails. Not all internal tests developed can use the limited assert-functions. Some tests just print out the result of an operation, and it is then up to the developer to assess if the result is correct.

Testing in this environment is difficult to comprehend, as the number of lines needed for a test is rather large, and the test run output is in a text console.

5.1.1 Tests implemented

4 functions have been implemented in the `ParserTest.cpp` file, along with a few helper functions. Tests are executed by calling the function in the `Main` function and executing the ParserTest project.

The helper functions are `InitCache()`, which sets up the WebXiCache with the standard test tree and starts the DSP Simulator etc. `Stop()` tears down the cache.
A modified `WriteClient(...)` function has also been implemented to replace the one

from the TestServer. The modified version does not send anything, but prints what it was asked to send.

The `Main(...)` function runs the test sequence specified in the function body.

TestPUT() test This test tests the `HandlePUT(...)` method of the parser by first asserting the values of several nodes directly in the test tree, then calling `HandlePut` with a JSON structure, committing the values, and asserting the values again to verify that the new values have been applied. This is repeated a few times with different value sets and root node.

If any errors occur, they will be printed in the console.

TestPUTRequest() test The `HandleRequest(...)` function is tested for a PUT request by creating a custom `EXTENSION_CONTROL_BLOCK` structure with a JSON structure to apply to the cache. The affected values are asserted before and after `HandleRequest` is called.

If any errors occur, they will be printed in the console.

TestGET() test Several combinations of root node and recursive flag values are fed into the `HandleGET(...)` function, and the response generated is printed in the console. The cases covered are:

- Full tree (top node), recursive
- Full tree (top node), non-recursive
- Subtree (branch node), recursive
- Subtree (branch node), non-recursive
- Leaf node

This test does not automatically assert the responses generated, but execution will halt between each `HandleGET` call, and will not continue until ENTER is pressed.

TestGETRequest() test Similarly to the `TestPUTRequest` test, an `EXTENSION_CONTROL_BLOCK` is constructed to emulate a GET request to `HandleRequest(...)`. No automatic assertions are made, but the response from `HandleRequest` is printed for visual inspection.

External request tests 5.2

Requests have been tested by an external test, sending PUT and GET requests and evaluating the outcome. This test strategy does not facilitate direct assertions on the cache, and it is not possible to intercept values or calls anywhere in the parser, so only the parser as a complete black box is tested – including the web server.

In order to aid the test and ensure perfect repeatability, a small addition has been made to the prototype web server. If the path */reset* is requested, the web server will reset the cache, ensuring that the initial values have not been overwritten.

Conveniently, the connector for use in the PULSE PC software has been developed to a point where it can be used. This not only simplifies the creation of the test environment, but it also verifies the function of the parser against the client to be used in the future.

The WebXi connector uses the built-in .NET HTTPWebRequest framework to handle the requests, and Json.NET[20] to manage JSON data.

The request tests have been developed in a separate solution – `WebXiParserTest` – using .NET/C#. NUnit[21] has been used as unit testing framework. These choices were made as the connector is implemented in .NET/C#, and because NUnit is used throughout B&K for software unit test. An illustration of the connection between the external test and parser elements is shown in figure 5.2.

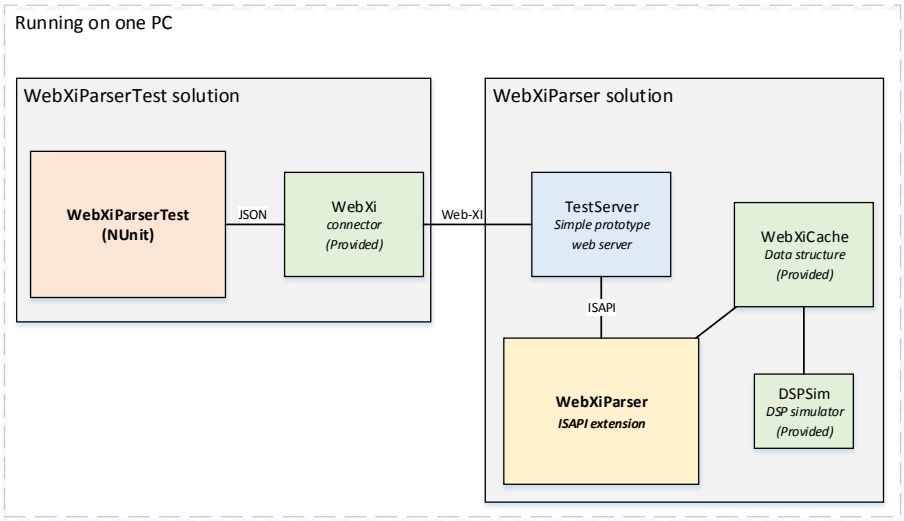


Figure 5.2: Illustration of the connections between the external unit test and the parser. Everything is described as running on a single PC, but the test and the parser may run on different computers.

[20] Json.NET is developed by James Newton-King – `www.james.newtonking.com/pages/json-net.aspx`
[21] `www.nunit.org` – A unit-testing framework for all .NET languages.

Being separate from the implementation of the parser, this test may with a minimum of changes be able to form the basis in a protocol accept test, which can be used both during implementation and maintenance on the LAN-XI G2 modules to verify that the parser and cache have not been broken.

5.2.1 Test sequence

The tests are divided into 5 fixtures, each handling a different aspect of the protocol:

GET Tests retrieving data from the prototype by issuing GET requests.

PUTTypesSuccess Tests setting values in the prototype by issuing value PUT requests with JSON data.

PUTTypesMismatch Verifies that errors are returned when the client attempts to apply the wrong data types to nodes.

Semantics Verifies that errors are returned when the client attempts to send semantically incorrect JSON to the parser.

Misc Tests various aspects not covered by the other tests, such as query parameters.

Certain aspects of the test setup has been placed in a separate class – `Setup` – which is used to set up the test fixtures. Here the hostname of the UUT (the parser instance) can be specified. It is possible to run the test server with the parser manually before running the tests – but if the `RUN_TESTSERVER` flag is set to true, the test will start the test server and stop it before and after each test[22]. This requires the `WebXiParser` solution to reside in the same directory as `WebXiParserTest` – and that the `WebXiParser` solution is compiled properly.

Each test fixture begins with a cache reset to ensure predictability.

GET

The GET request test tests different combinations of recursive flag use, and requests on the root, branches and leafs.

All tests validate each node returned from the requests with the default values of the cache.

The following cases are covered:

[22] This assumes that the `WebXiParser` and `WebXiParserTest` solution reside in the same directory. Otherwise, the path may be changed to match the correct path.

- Requesting the *top node* (WebXi) with the *recursive* flag set *true*, returning the full object model tree.
- Requesting the *top node* with *recursive* set *false*, returning the children of WebXi.
- Requesting a *branch node* with *recursive* set *true*, returning a full subtree.
- Requesting a *branch node* with *recursive* set *false*, returning the children of the requested node.
- Requesting one *leaf node* of each data type.
- Various requests, evaluating an *uncached node*, verifying that its value is updated properly – and only when it is recovered in a request.

PUTTypesSuccess

In this test fixture, setting values of each supported value type is tested by PUTting a value on an appropriate node, GETting the same node and verifying that the new value was set.

Read only behavior is also verified by attempting to set a value on a read-only node, verifying the HTTP Status returned to be an error, and verifying by retrieving the node value that the value is unchanged.

The following cases are covered:

- Setting an assorted tree of values (all value types covered).
- Setting each of the different types of value.
- Setting each of the supported types of array with values.
- Setting each of the supported types of array with empty arrays.
- Attempting to set a value on a read-only node.

PUTTypesMismatch

This test fixture tries to set values to nodes where the data types mismatch. The HTTP status is evaluated in all the cases, and for each node, the value is afterwards retrieved and verified to be unchanged from the default value.

The combinations tried are shown in the matrix in Table 5.1. X'es mark illegal illegal combinations of node value type and JSON data type which are tested.

Node type	JSON data type						
	Bool	Int	Float	String	Null	Int[]	Float[]
Boolean		X	X	X	X	X	X
Int32	X		X	X	X	X	X
Int64	X		X	X	X	X	X
Float32	X			X	X	X	X
Float64	X			X	X	X	X
String	X	X	X		X	X	X
JSON	X	X	X	X	X	X	X
Int32 array	X		X	X	X		X
Float64 array	X	X		X	X		

Table 5.1: Matrix showing the combinations of illegal value type assignments covered in the `PUTTypesMismatch` test fixture.

Semantics

Handling of semantic errors is covered in this test fixture. Various bits of JSON are sent to the parser, each with a particular semantic error, and the HTTP status is evaluated. The semantic errors are constructed around a JSON structure trying to set data on one or two specific nodes. The values of these nodes are checked at the end of each test, and at the end of the test fixture, the entire tree is validated. This is done using test code from the *GET* test fixture.

The semantic tests are not complete, but they cover many cases of missing/misplaced tokens. Focus has been on covering the errors that may be thought to cause problems by either being faultily accepted or to cause the parser to enter an illegal state.

One example of such an illegal state could be a valid JSON structure, which has an extra } appended. When the "legal" part of the JSON is parsed, the parser has returned to the root node of the request. The extra } would then cause the parser to jump to the parent of the root node. Any further JSON after the stray } would then potentially be handled, and the parser would operate outside the subtree requested.

Another possibly worse problem would be if the top node of the object model (*WebXi*) was requested. An extra } would cause the parser to jump to the parent of *WebXi*, which is null. This could lead to the parser crashing.

Misc

This test fixture contains single tests of various scenarios:

SuccessfulPUTAction Due to very limited support of the action query in the cache, only a very sparse test of Actions is implemented, verifying that the return status is *200 OK* when a valid action is sent.

TimeSet A request is made with time set. This should be accepted by the parser.

DelaySet A request is made with delay set. This should be accepted by the parser.

TimeDelayConflict A request is made with both time and delay set. This should result in an error.

5.2.2 Outcome

A total of 53 test cases have been implemented. All tests run properly and report success.

This test setup has aided in improving the prototype web server and fixing small problems in the parser. It has been a useful tool for verifying the functionality of small changes made toward the end of the project period – being able to quickly run a batch of tests frequently to establish that everything still works, and in case of errors, be able to pinpoint the change that led to the error.

Misc. tests 5.3

During the development many small non-scripted tests have been run. The methods used vary a lot.

For testing HTTP GET requests, internet browsers[23] have been used. A nice feature when using HTTP and JSON is that a browser can send the request and get the response – and the response is readable without any need for special parsing.
To get values using a browser, it should be pointed to e.g.
http://localhost/WebXi/Acquisition?recursive=true – assuming that the prototype and the browser both run on the same computer.

For testing HTTP PUT requests, cURL[24] has been used.
Running cURL in a command prompt with the following arguments will send the contents of the *PUT.json* file to the specified URL in a PUT request:
curl -H "Accept: application/json" -H "Content-type: application/json" -X PUT -d @PUT.json -v http://localhost/WebXi/Acquisition/Channels/1

The response from the server is written in the console. For a successful PUT request, the output may look like this:

[23] Google Chrome has been the preferred browser, as it displays the JSON text in the browser window when retrieving values – Internet Explorer instead wants to download a .json file.

[24] cURL (`www.curl.haxx.se`) is a command line tool for transferring data, e.g. performing PUT requests.

```
* About to connect() to localhost port 80 (#0)
*   Trying 127.0.0.1...
* Connected to localhost (127.0.0.1) port 80 (#0)
> PUT /WebXi/Acquisition HTTP/1.1
> User-Agent: curl/7.29.0
> Host: localhost
> Accept: application/json
> Content-type: application/json
> Content-Length: 301
>
* upload completely sent off: 301 out of 301 bytes
* HTTP 1.0, assume close after body
< HTTP/1.0 200 OK
< Content-Length: 0
< Content-Type: application/json
<
* Closing connection 0
```

Chapter 6

Process

The project has been developed using an iterative process. From the beginning a set of rough tasks were laid out and their development times were estimated. Each task was intended to be examined closer when they would begin, and they should each result in a prototype, meaning that each task would result in running code. The planning was done using a GANTT chart, which is shown in figure 6.1.

In the plan, documentation was set as an ongoing activity throughout the project period, but a large buffer was also left at the end of the project for misc. activities such as testing, bug fixing and report finishing. The buffer would also leave room for the compulsory colloquium which took place on April 22nd – and the preparations it required.

The original plan worked well to get going, but as the work progressed, many changes would be made to the it. Time wise the plan would hold, but when a task was examined it would often turn out to be better to integrate another task into the solution, eventually changing the sequence of the original plan quite a lot.

Despite the changes, the original plan would still provide a good baseline for assessing the progress of the project, as the tasks and their (partial) completion would still sum up to give an idea of how much was still left.

As the WebXiCache – which the parser depends on – has been developed somewhat in parallel with the parser, the focus on the parser implementation has occasionally been dictated by the that. In the beginning the WebXiCache was not available at all, which forced an approach from the outside in, beginning with the web server and considerations regarding it.

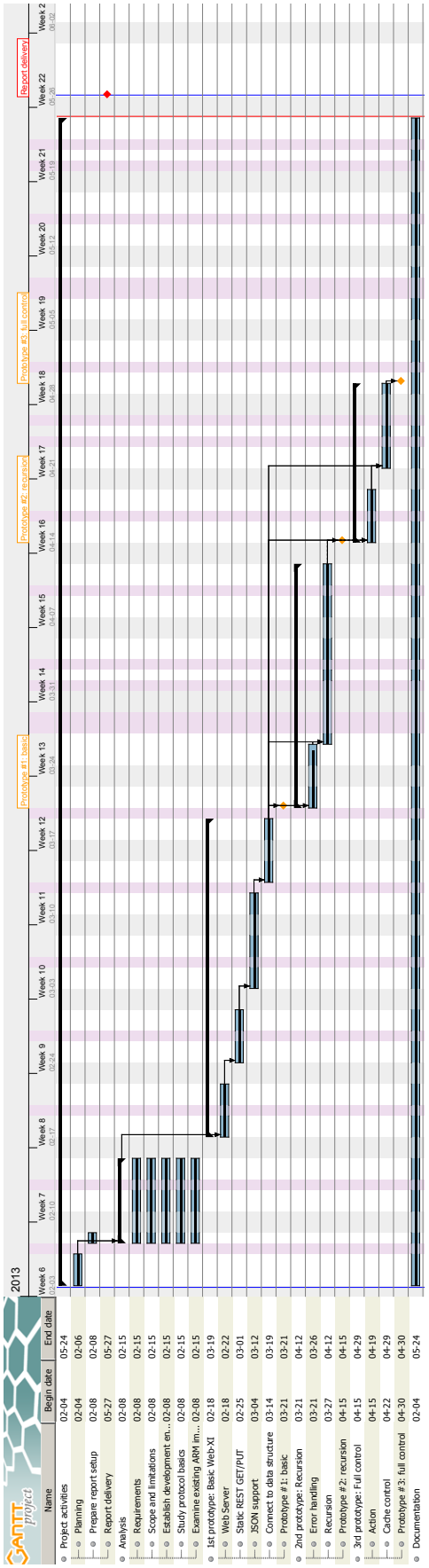


Figure 6.1: GANTT chart used for initial planning of the project.

Chapter 7

Perspective

This project has introduced a different approach to developing for the ARM system compared to previous development by developing a prototype on an ordinary PC with the fact in mind that it should eventually be able to run on the ARM CPU.

The project has given new experience and led to many new options for development and testing. These have been described briefly here.

Implementing the parser on the ARM CPU 7.1

At the end of the project period, attempts were made to import the Web-XI parser and cache into the development environment used for the ARM. This was done by others familiar with the tools.

The outcome of the initial attempt was, that the code was successfully imported and compiled with minor changes. Attempts have not yet been made to run the code on an ARM CPU.

The preliminary conclusion was, that a shared code base between PC prototype and the final ARM implementation seemed feasible, and that it seemed like the it would be possible to make the system work on the ARM platform.

Future development 7.2

The implemented Web-XI parser does not completely implement the Web-XI protocol. The parser itself lacks the version and caching support towards the client, and streaming of data has not been considered.

The WebXiCache and other helpers will eventually also need to have the correct interface for communicating with the DSP – and the Action/AwaitTrigger/Time/Delay fields shall be fully implemented and verified.

At the end of the project period major changes were made to the Web-XI connector used for the external tests. This happened after the tests were developed, and thus the tests in this project still use the old connector. To give a valid base line throughout the chain of communications, this could be changed, but the existing tests are still valid for verifying the web server/parser implementation.

New potentials 7.3

The new approach to development provides a lot of opportunities for easing the development process for LAN-XI G2.

7.3.1 Embedded development

Developing new larger subsystems can be done using PC prototypes. Targeting the PC will in some cases give advantages of more streamlined development, as things can be compiled and examined immediately without bringing in special hardware. Running the systems on a PC also typically gives access to easier debugging than when running on an external system.

For continued development of existing implementations PC prototypes are, however, not feasible, as this would mean having to port the entire existing application to a PC platform, and in many cases only a limited number of changes would have to be made to the existing system.

7.3.2 Hardware development

Since the LAN-XI G2 platform uses Ethernet for internal communication, a "shoebox model" can be used for development. The shoebox model is a concept where a complete module can be built without having the final parts. A lot of development can be done with each part – I/O board, DSP and ARM CPU – individually, but not the communication between them.

The following substitutions could be regarded as illustrated in figure 7.1 – connected to each other using an ordinary network switch:

I/O board A LAN-XI G1 module with a few modifications making the LAN-interface mimic the final G2 I/O board.

DSP A DSP evaluation board with a DSP and a network port.

ARM CPU A PC running the Web-XI parser prototype – with some changes/extensions – or an ARM evaluation board running Windows EC7 and the final Web-XI parser.

Using such a setup, it would theoretically be possible to implement the DSP completely without any of the final hardware available – and with only a few modifications switch to the final DSP when all hardware is ready.

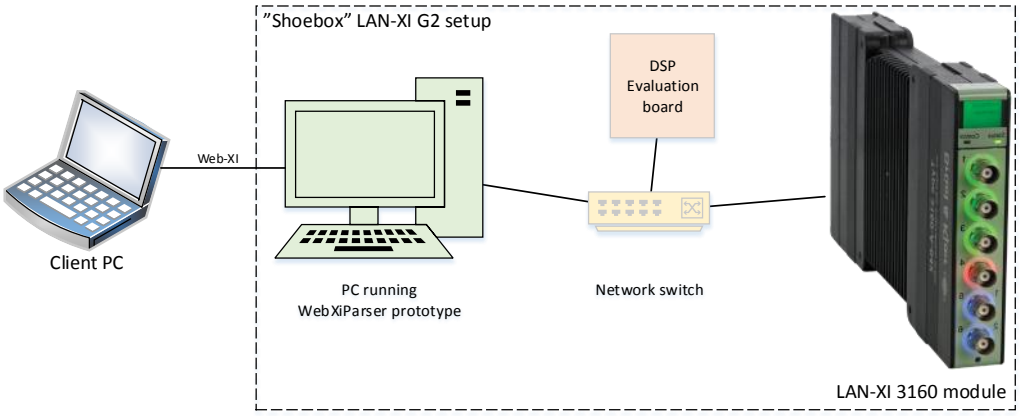


Figure 7.1: Illustration of the "shoebox" model with the ARM CPU replaced by a PC running the prototype Web-XI parser, a DSP evaluation board and a LAN-XI G1 module acting as input/output hardware.

The shoebox model may also be used to develop a set of tests for verifying the function of the modules. Such a test could then be used when implementing the final hardware modules to quickly tell if things work as intended, and thus create a valuable baseline for development – possibly exposing errors that would otherwise stay hidden for long during the final development phases, or even beyond.

7.3.3 Software development

When new hardware is released, it is necessary to support it in the associated software. Thus, software developers need access to the hardware during development – while the hardware is also being developed.

A simulator based on the parser prototype could suffice in many aspects and could reduce the need for the developers to have the vastly more expensive real hardware.

Chapter 8

Conclusion

A PC-based prototype protocol parser for Web-XI has been developed covering most of the command protocol.

Being able to interpret JSON and store the results in the provided WebXiCache data structure – and retrieve data from the cache and present it as JSON – was a major goal, and the parser is able to do that.

An attempt has been made to implement the parser as an ISAPI extension without having access to the correct tools, but rather by examining the ISAPI interface and data structure. The parser should also ultimately be able to run on Windows EC7 on an ARM platform. These things seem to have been quite successful, as the system (parser and cache) was imported into the correct environment and would compile with a minimum of changes.

The PC prototype has been set up against the connector to be used in the PC software, and data exchange works fine. For this, a unit test has been developed for externally testing the functionality of the parser – which includes the whole chain from connector to the web server, through the parser and into the cache.

There are still things to be done before a complete prototype for the Web-XI protocol is a reality, but the concepts selected for this first prototype have all been implemented. Some of the things implemented have not really been verified, as there is little or no support for their function in the cache. This includes actions, triggers and time/delay parameters. These should, however, be fairly simple and might mostly work as they are implemented now, otherwise changes should be straight forward to make.

This project has given opportunities to consider new ways of developing hardware, which may lead to quicker development, better utilization of resources, and/or better quality.

If this parser prototype is eventually fully developed to support all Web-XI features, it may be used as a basis for a PC-based LAN-XI simulator, enabling software developers to target a new device before the hardware is ready. It may also be used to replace the ARM CPU for development, making it possible to test DSP and/or I/O board implementations before the ARM CPU is ready to use.

All in all, this project realizes a part of a mass of new considerations and optimizations for the entire development process – both regarding hardware and software.

Chapter 9

Glossary

In order to provide a better understanding and avoid a lot of descriptions, this glossary is provided to give a quick definition of some key expressions commonly used through this report.

Many of the terms are described in more detail elsewhere.

ARM (**A**dvanced **R**isc **M**achine) is a computer processor architecture used in many embedded applications such as mobile phones. LAN-XI devices incorporate an ARM CPU running an embedded Windows version for managing communication with client computers and controlling the internal devices of the LAN-XI module.

B&K Shorthand for Brüel & Kjær (Sound & Vibration A/S in this context).

Branch node A tree node with children. In this case, a branch holds no value.

Client In this scope a client is typically a computer receiving measurements done with a LAN-XI module. The client runs B&K Pulse software for acquiring and processing the measurements.

DSP **D**igital **S**ignal **P**rocessing – the DSP unit handles processing of e.g. the measured signals before they are sent to the client PC.

Ethernet The physical layer of the standard local area network.

(HTTP Request) Body Data part of a request sent by a client to the HTTP server.

(HTTP Request) Method Identifies the desired action. May be GET, PUT, POST or DELETE (more do exist) where GET shows a request to get data, PUT desires to update data on the server, POST will add data to the server and DELETE will delete data from the server. The actual use of these methods depends on the server application running.

(HTTP Request) Path Usually points to a (sub) directory on a web server, but may be used to identify other resources. The path is on the form /Rest/Aquisition/Channels/1/Gain.

(HTTP Request) Query An optional set of field-value parameter to send with the path in an HTTP Request. The query follows the path, separated by a questionmark (?), field-value instances are separated byan ampersand (&), ie. if /Rest/Aquisition?parameter1=value1& parameter2=value2 is requested, the query is two field-value instances (parameter1,value1) and (parameter2,value2).

(HTTP) Request The term covers any HTTP requests received. A request consists of a header containing among other things a *path*, optional *query*, a *method* and optional **body**.

ISAPI **I**nternet **S**erver **A**pplication **P**rogramming **I**nterface – an API of Internet Information Services (IIS), Microsoft's web server.

ISAPI extension An application that runs on IIS. The interface is examined in section 2.4.

Kiss The operating system running on the DSP in e.g. LAN-XI modules. Kiss is an abbreviation of "Keep It Simple, Stupid", which refers to the simple structure of the operating system.

LAN **L**ocal **A**rea **N**etwork – An ordinary Ethernet network for connecting computers and other devices.

LAN-XI A series of acquisition hardware centered around network-attached modules to which transducers can be attached. Described further in section 1.2.

LAN-XI G1 The first (and current) generation of LAN-XI hardware first released in 2008.

LAN-XI G2 The next generation of LAN-XI hardware which this project is a part of. Described in section 1.3.

Leaf node A tree node without children. Leaf nodes hold values.

Module When dealing with LAN-XI system, module refers to a single LAN-XI unit.

Parent node In a tree data structure, a parent is the node above the current node.

Parser Interprets a language and performs actions based on it. Refers to the Web-XI parser which is the central element in this project.

PULSE The software suite developed by B&K used to work with the LAN-XI hardware. Ultimately, Pulse is going to use the Web-XI protocol to communicate with the WEb-XI parser in the LAN-XI G2 modules.

REST **RE**presentation **S**tate **T**ransfer – a design model for a client-server web API used in Web-XI.

Root node The top node of a tree data structure. The root node has no parent and no siblings. In some cases a root node may refer to the top node of a subtree.

Sibling In the tree data structure, a sibling is a node at the same level as the current one which has the same parent.

Subtree A part of a tree constisting of a node and all its successors.

Tree A data structure consisting of a single root node and multiple branch and leaf nodes.

Transducer A measuring device such as a microphone or accelerometer.

UUT **U**nit **U**nder **T**est – When testing UUT refers to the system, function etc. being tested.

Web-XI The new HTTP REST based protocol to be used for communication with LAN-XI G2 hardware. Described in section 1.3 and section 2.1.

Windows CE The previous embedded version of the Windows Operation System. LAN-XI G1 runs Windows CE 5 on the ARM CPU.

Windows Embedded Compact (Windows EC) The new name for the Windows CE series. Windows EC 7 is used in LAN-XI G2.

Bibliography

[1] Jackson Mowry and Ghita Borring. *Journey to Greatness – The Story of Brüel & Kjær*. Acoustical Publications, Inc., 2012. ISBN 978-0-9769816-3-3.

[2] Introducing json. http://www.json.org.

Appendix

Table of contents

Appendix A

Compiling and running the parser

The Web-XI parser and associated support programs and tests have been developed using Microsoft Visual Studio 2010 with Visual C# and Visual C++. Other versions may be used, and compiling and running may be possible without Visual Studio, but this walkthrough only covers Visual Studio. Knowledge on how to use Visual Studio is assumed.

The project is assumed to be structured like this – with the major directory structure and key locations pointed out:

WebXiParser – The Visual C++ solution containing the parser, internal test and prototype web server.

- **Debug** – Contains the compiled libraries and programs.
- **ParserTest** – Console program project running internal tests on the parser.
- **TestServer** – Project implementing a simple prototype web server.

WebXiParserTest – .NET 4/C# solution defining external tests. Requires projects from the WebXi solution.

- **WebXiParserTest.nunit** – NUnit project for running the external tests.
- **WebXiParserTest\BK** – The B&K WebXi connector DLLs (*BK.Pulse.FrontendDrivers.RestClient.dll* and *BK.Pulse.FrontendDrivers.WebXi.dll*) should be here.
- **WebXiParserTest\JSON** – An appropriate Json.NET library (*Newtonsoft.Json.dll*) should be here.
- **WebXiParserTest\NUnit** – An appropriate NUnit test library (*nunit.framework.dll*) should be here.
- **Setup.cs** – A class containing common parameters concerning the connection to the parser, such as hostname and functionality to start the test web server automatically.

WebXiParser solution A.1

The solution contains 2 executable projects; *ParserTest* and *TestServer*.

Compiling the solution requires Microsoft Visual Studio 2010 with Visual C++ including runtime.

As there are two executable projects in the solution, it is important to be aware of which one is being executed, ie. marked as StartUp project.

An alternative is to run already compiled versions. These are found in the *Debug* folder of the WebXiParser solution directory.

A.1.1 ParserTest

The ParserTest runs the internal test of the parser, described in section 5.1.

The test runs in a console, and pauses itself several times during the run so that the output can be evaluated. Pressing ENTER continues execution.

A.1.2 TestServer

The prototype web server runs in a console as with the ParserTest. It will run in an infinite loop and will not shut down unless a critical error occurs or it is forced to close.

The server can be terminated by using the normal close button (X) in the upper-right corner of the window.

WebXiParserTest solution A.2

The solution requires the *BK.Pulse.FrontendDrivers.RestClient.dll, BK.Pulse.FrontendDrivers.WebXi.dll, Newtonsoft.Json.dll*[25] and *nunit.framework.dll*[26].

Running the tests requires NUnit 2.6.2 and Microsoft .NET Framework 2.0.

The tests included in the solution are described in section 5.2.

[25] Version 5.0.4.16025 of the DLL has been used during development.
[26] Version 2.6.2.12296 of the DLL has been used during development.

It may be necessary to disable strong naming checks for the WebXi connector. This is done by issuing the following command in a command prompt[27]:
*sn.exe -Vr *,d64412599724c860*

Parameters such as which hostname to use when testing and whether or not the test web server should be automatically started are located in the `Setup` class in *Setup.cs*.

When the WebXiParserTest project is compiled, a DLL is generated. This contains the NUnit tests to run. The *WebXiParserTest.nunit* file in the WebXiParserTest solution directory is a NUnit project file containing the setup for the tests. This can be loaded in NUnit, and tests can be run.

[27] If the sn.exe utility is not in the system PATH, navigate to the folder in which it resides, e.g. C:\Program Files\Microsoft SDKs\Windows\v6.0A\Bin\x64\ for a 64-bit Windows 7

Appendix B

Initial project considerations

These are the considerations used in the initial definition of the project. The actual project has been derived from this.

Some things may not be completely accurate in the considerations, as several factors have changed from the considerations were made till the project work was actually started.

Scope B.1

The LAN-XI G2 platform will incorporate a new protocol based on REST. This means that both hardware and software needs to be implemented with support for the new protocol. This project focuses on the hardware-side implementation of the new protocol. The figure below shows roughly how communication between the PC and the components of the LAN-XI device should work.

The PC communicates with the LAN-XI module using REST messages on an ordinary network connection. On the LAN-XI module, the ARM processor (running Windows CE) will receive the REST messages via its web server and do some parsing.

It is currently unclear, how much of the parsing will be performed on the ARM CPU. The ARM CPU will handle some of the commands received itself, but other commands should be forwarded (possibly somewhat unprocessed) to other onboard devices such as the onboard DSP or the FPGA. The communication is illustrated in fig. B.1 on page A-8.

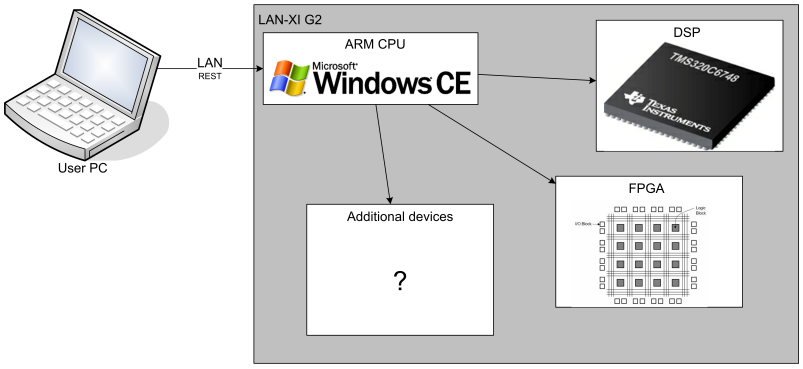


Figure B.1: Illustration of the communication chain between a PC and the devices within the LAN-XI module

The REST protocol command set - part of which is often referred to as "the new protocol" or "Web-XI" - is defined prior to this project, so the project concerns implementing the protocol efficiently.

Requirements B.2

These requirements are part of the initial input for defining the project and will be elaborated more during the analysis phase of the project.

- The development of the system must be based on C, as this is used on the ARM CPU.
- The system must be able to receive the specified REST commands through a network connection. Based on the contents of a received command, it can be handled on the Windows CE system or forwarded to another onboard device, i.e. the DSP or FPGA.
- Parsing of commands must be efficient and easy to manage for future changes. Huge if-else if-else blocks should be avoided, and efficient table setups should be sought.
- Tests developed during the internship must be ported to run on the new setup.
- The system must be flexible, communication with e.g. the DSP should be easily substituted, as the final means of communication with other devices are not yet clear.

Additional tasks B.3

- A higher-level programming language (i.e. Python) can be considered, but the advantages must ratify any added load or memory consumption.
- As an intermediate step, the REST interpreter can be implemented on a PC, using already known methods of communicating with the DSP.
- As commands received on the ARM CPU may effectively have different - or maybe even multiple - recipients, there should be a flexible and clear system for handling these individual cases.

Process thoughts B.4

Currently, following the internship project, a PC can control the DSP using specially formatted Kiss Items[28] through a JTAG connection. Tests have been implemented for verifying functionality on the DSP. This is illustrated in figure B.2.

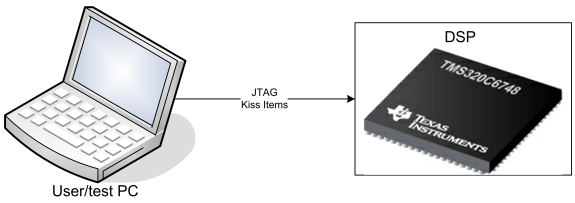


Figure B.2: Internship product – communication between PC and DSP

The first step of development can be implementing the REST interpreter on a PC, using the existing JTAG connection to the DSP – as illustrated in figure B.3 on page A-10. The existing tests can be rewritten to use the REST protocol. For simplicity, the REST protocol can be used to wrap Kiss Items initially. These Items will later on be substituted with the actual final commands.

Implementing on a PC makes debugging and development easier and quicker, as well as it can provide a good benchmark for implementation and evaluation of the REST protocol.

When functionality of the REST interpreter has been verified, the interpreter can be implemented on Windows CE on the ARM CPU. Initially, the DSP might be the only target for commands, and communication may still consist of wrapped Kiss Items. The existing tests could still be used for verifying the functionality of the interpreter. This is illustrated in figure B.4 on page A-10.

[28] Kiss Items are the data structures used for communication internally in the LAN-XI module, ie. on the ARM CPU and the DSP.

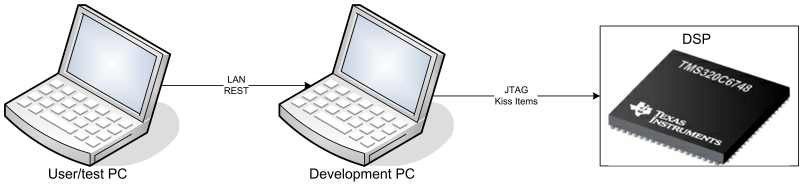


Figure B.3: Possible prototype setup with the ARM CPU substituted by a PC

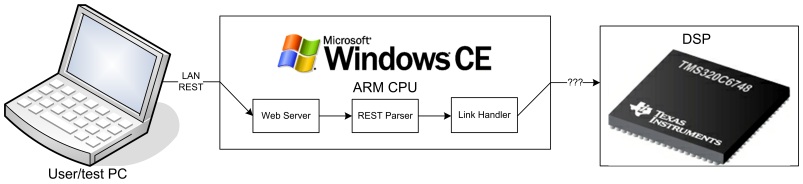


Figure B.4: Final setup with ARM CPU

With a basic command chain running, the full command set can be implemented, and more onboard devices can be added in addition to the DSP.

Appendix C

Internship report

My internship was completed one year prior to this project. The main internship project involved creating a unit testing framework for the DSP to be used with LAN-XI G2. One consideration regarding the Web-XI Parser project was to change the internship project so use the Web-XI parser.

Brüel & Kjær

Internship report

Testing DSP implementation

Made by:

Handed in via CampusNet

s042067 - Clausen, Per Boye

DTU Informatics
Department of Informatics and Mathematical Modelling

Delivered: July 18th, 2012
This report contains 10 pages incl. front page

Abstract

This report describes my internship stay at Brüel & Kjær Sound & Vibration Measurement A/S, in the Instrumentation group of the R&D department.

The internship was carried out during the Spring semester of 2012 and was focused on testing tasks for a DSP for next-generation products. These tasks would include developing a means of measuring and comparing performance of different routines on the DSP, debugging of a faulty routine for formatting a float when writing to a serial console, as well as the development of a framework for unit testing the DSP implementation.

The unit testing framework had to meet several requirements regarding the implementation into existing routines and systems at Brüel & Kjær, and tests would be written for these.

Table of contents

Chapter 1

Introduction

This report describes briefly my internship stay in the Instrumentation group at Brüel & Kjær Sound & Vibration A/S during the Spring semester of 2012 – February 1st until July 2nd.

Having some experience with Brüel & Kjær measuring equipment from previous interests and studies, the company was my number one choice for the internship. Throughout my studies, I have been focused towards the lower levels of the IT studies – hardware and embedded systems – while gaining experience in higher-level development. This broad spectrum was the entry point for the planning of the internship project.

About Brüel & Kjær 1.1

Brüel & Kjær was founded in 1942 by Per V. Brüel and Viggo Kjær. The company has been leading on the professional marked for sound and vibration measurement – but has also been active in other areas such as long-term condition monitoring of machinery and medical instruments.

In 1992, following financial trouble, the company was sold to the German AGIV and split into several more focused companies, one of which is Brüel & Kjær Sound & Vibration Measurement A/S, which was part of the Spectris Division of AGIV. This division was later sold to the British Fairey Group Ltd. but was ultimately kept under the name of Spectris Plc. – still maintaining the Brüel & Kjær Sound & Vibration Measurement name within.

The products developed by Brüel & Kjær Sound & Vibration Measurement A/S include the LAN-XI modules which perform data aquisition – measuring from a variety of different sources, ie. microphones, accelerometers etc. – and do some processing of the input. Various connection options are achieved through interchangeable fronts. A LAN-XI module with two detached fronts is shown in figure 1.1.

Figure 1.1: A LAN-XI module with two detached fronts

The modules are connected to an ordinary LAN, and the measurements are processed and presented on a PC using the PULSE software. Several LAN-XI modules can be used simultaneously, making the system very flexible.

Furthermore, the product range includes 2250 range handheld analyzers which are stand-alone PDA-like devices used to perform and present acoustic or vibration measurements – as well as transducers; microphones and accelerometers for vibration measurement.

1.1.1 The Instrumentation group

The Instrumentation group is part of the R&D department. The group develops much of the new hardware and embedded software used within the company and the products. This includes the LAN-XI modules and the 2250 range handheld devices.

Presently, the group is beginning development of a new generation of LAN-XI modules, which is where my work has been focused.

Chapter 2

Internship Project

During the internship at Brüel & Kjær, I have been working with a new DSP implementation for next-generation products. I have not done any actual development on the DSP software, but rather been performing testing tasks, which includes performance measurements, debugging and development of a unit testing framework.

Performance measuring 2.1

The first task was to develop means of measuring an approximate number of CPU cycles used to perform various tasks. This was used to compare the cost of various implementations of a Sine function.

The performance measurements are performed by placing small bits of code before and after the code, logging internal CPU cycle count values[1]. These values are passed to a performance monitor process, which gathers and summarizes data, and outputs a summary to a serial console.

[1]The CPU cycle count is an absolute value. Therefore, if the monitored code region allows another process to run (the DSP operating system uses non-preemptive scheduling), the cycle count will include the cycles used by the other process(es). Furthermore, caching will affect subsequent runs of a code region, but when used correctly, this can also be utilized to assess the performance when caching is applied.

Debugging 2.2

When printing floats to the serial console, a special formatting function is used to convert a IEEE 754 formatted (sign-exponent-mantissa) to integer and decimal parts. This function would in some cases produce the desired output, but in many cases it would seemingly round up the number to the next integer.

The function is written in Assembly, and the debugging involved understanding the method of the function, including examining contents of registers at each step of the execution for certain start conditions.

The error was found to be in a right-shift operation of what was to be the decimal part of the number. Here, a sign-extended right-shift was used, which would in many cases pad the number with 1's, giving a large (unsigned) decimal part, which would in turn round up to the next integer. Replacing the right-shift with a zero-extended right-shift solved the problem.

Unit testing framework 2.3

The majority of the internship was spent developing a unit testing framework for the DSP.

2.3.1 Entry point

The DSP is connected to a PC through a JTAG socket and a USB emulator, and the DSP operating system was already fitted with a *LinkHandler* process for exchanging messages through JTAG. A small set of Java methods was also provided, which used Java libraries provided by Texas Instruments to interface with the DSP. These methods were previously used from Matlab to create test scripts to verify the functionality of the DSP algorithms. All communication between the processes on the DSP uses simple message passing with messages called *Items*. The Java methods would use a similar Item structure to represent data.

Within Brüel & Kjær, C#/.NET is used for development, and NUnit for unit testing. Thus, an objective of the project was to enable unit tests to be run on the DSP from NUnit – and to put as much of the functionality as possible in a C# environment.

Futhermore, Jenkins CI[2] is being introduced for managing automated test runs. Another objective was to use Jenkins for managing nightly test routines and presenting the results.

The final setup is illustrated in figure 2.1 on the next page.

[2]Jenkins CI, `http://jenkins-ci.org/` is an automated build/test system with high extensibility through plugins

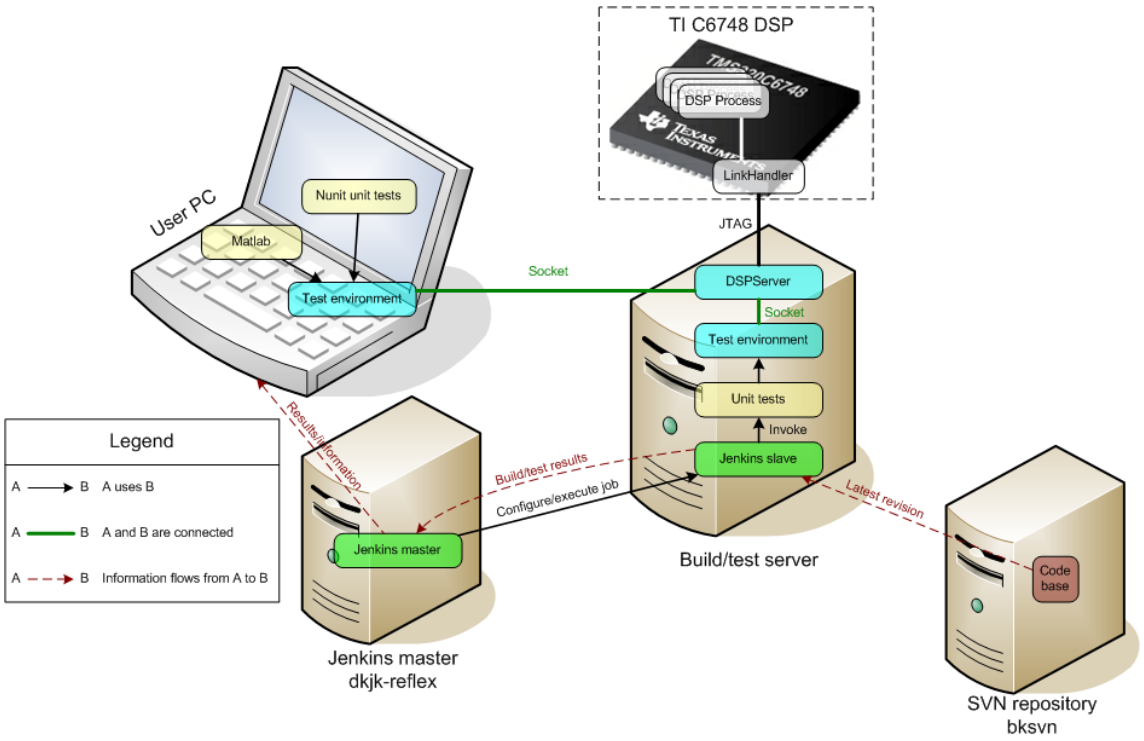


Figure 2.1: Illustration of the final setup.

2.3.2 Framework

The system needed to bridge the gap between the Texas Instruments Java libraries and a C# environment. This has been done by developing a small server application in Java – *DSPServer*. This is used by the C# *Test environment* through a socket connection and a self-developed pseudo-RPC approach, which means that the C# environment controls the DSP using roughly the same methods as the Java libraries provide.

On top of the connection to the DSPServer, the C# environment consists of 3 layers, each raising the level of abstraction.
The lower layer provides basic Item functions and requires the programmer to manage polling when exchanging Items with the DSP.
The middle layer does the polling and introduces queues of Items, from which Items matching certain conditions can be requested.
The top layer is based on data and signals and avoids the use of Items. The layers are interchangeable, and in the future, the two bottom layers may be replaced by other means of communication with the DSP[3], when development reaches a later stage.

2.3.3 Usage

The framework is still usable from Matlab, and several Matlab scripts have been developed for providing a graphical representation of the data to be evaluated in the

[3]In particular, the network connection which will be present on the final product.

unit tests. Furthermore, the Matlab connection is being used for an early interface for evaluating the functionality of the DSP without needing any other parts, e.g. PC software and embedded PC for controlling the DSP.

NUnit can use the framework directly, and a number of *Unit tests* have been developed for verifying the functionality of the DSP software and algorithms.

The test environment can access the DSPServer over a network connection, enabling developers to share one DSP – as long as they do not try to connect at the same time.

2.3.4 Automation

A *build/test server* has been set up. This computer is connected to a development board and runs as a *Jenkins slave*.

The *Jenkins master* server has jobs defined for various tasks, which are executed on the build/test server:

- (Re)configure and (re)start the Java server, compiling the program from the latest source available on the SVN server
- Compile and publish documentation of source code from SVN using Doxygen[4] and a local web server on the build/test server
- Compile DSP software, C# framework and unit tests from latest source available in the SVN repository and execute the unit tests

These tasks are managed by the Jenkins master and Windows Command scripts committed to the SVN repository. The first two are set up to automatically check for new SVN revisions every 5 minutes and execute if new revisions are detected, whereas the unit test job is executed every night.

For the unit test job, Jenkins gathers the test results and displays results and trends for every run, which makes it easy to monitor the progress of the development, as well as verify that no previously developed features are broken.

[4] `www.doxygen.org` - tool for generating code documentation from source code with specially formatted comments

Chapter 3

Conclusion

During the internship stay at Brüel & Kjær, I have learned a lot about daily work in a development department. In particular, I have learned to be aware of prioritizing and balancing the tasks for productivity rather than absolute correctness.

Another important experience has been working on a project alone, without others doing the same work at the same time. This requires problems to be either solved by myself, or turned into some more general problem which others are able to understand.

I have enjoyed having a very broad area of work, using 6 different programming/scripting languages spread across systems ranging from Assembly and C in the embedded operating system on the DSP over developing the framework in Java and C# to unit testing and Matlab scripting – and finally automating executions using Windows command scripting. Furthermore, I have had the opportunity to work with many different tools during development and had to learn how to use each of them efficiently.

Getting to know the new systems and routines – especially familiarizing myself with the DSP – was greatly helped by a gradual beginning with the smaller performance monitoring and debugging tasks, meaning that I had a good understanding of the basics and uses of the DSP and its operating system before the greater task of implementing the testing framework was initiated.

The studies at DTU – as well as my own projects through the years – have given me a lot of skills, many of which I have had the chance to use, ranging widely among the courses I have attended. Seeing a constructive use for these skills has been very rewarding.

Appendix D

Design of Web-XI Communications Protocol for G2 devices

This document describes the design of the Web-XI protocol and is the main source for the definition of the project requirements.

The document has been modified slightly to be presented in this appendix, but no points relevant to this project have been altered.

Design of
Web-XI Communications Protocol for G2 devices

Revision History:

Initials	Date	Revision	Comments
HELGERAS	14-01-2010	1	First Version started
HELGERAS	18-03-2010	2	First Version complete
KONS_LMJORGENSEN	06-03-2012	3	Update after first implementation
HELGERAS	09-10-2012	4	Updated with suggestions/clarifications from CHANSEN
HELGERAS	12-09-2012	5	Changed Signaldata message type to include interpretation data. Removed Interpretation message type. Introduced padding to multiples of 4 bytes.
HELGERAS	13-11-2012	6	Changed to used WebSocket protocol for streaming.
CHANSEN	18-12-2012	7	Minor edits
HELGERAS	15-01-2013	8	Added section about "Triggered actions and PUTs"
HELGERAS	20-02-2013	9	Added information about timed actions and PUTs.
HELGERAS	08-05-2013	10	Fixed a few inconsistencies in the description of actions and arguments.

Contents

1 Purpose

This Design Document is based on the requirement Specification for the Web-XI protocol, and defines an architecture that fulfils these requirements.

2 Terminology

Term/Acronym/Abbreviation	Description
BIST	Built In Self-Test
CHESS	Channel Embedded System Sheet: A description of a channel that contains information about settings and possible values for these.
GISP	General Instrumentation Solution Platform
GISP device	A device that implements GISP. Such a device will have a IP/Ethernet connection and must implement the GISP protocol.
Web-XI protocol	The new name for the "GISP protocol" for G2 devices
IDAe protocol	The communications protocol used for older B&K acquisition hardware. The protocol is implemented on top of TCP/IP. The IDAe protocol has a command that can send a KISS item to a device.
IVI driver	A driver for test instruments that conforms to the standard set forth by the IVI Foundation. See http://www.ivifoundation.org/ for details.
KISS	B&K Proprietary DSP Real-time operating System
KISS item	A message in the KISS operating system. A lot of commands for IDAe modules are described as KISS items.
PTP	Precision Time Protocol
REST	REST (representational state transfer) is an approach for getting information content from a Web site by reading a designated Web page that contains an XML (Extensible Mark-up Language) or JSON (JavaScript Object Notation) file that describes and includes the desired content.
SLM	Sound Level Meter
PULSE Labshop	This is the "old" PULSE including the PULSE Time Data Recorder.
Client	Software that communicates with a GISP device such as a web browser running NOTAR, PULSE Reflex, PULSE Labshop.
Websocket protocol	A protocol for a two-way streaming protocol. The protocol is documented in RFC 6455. See also http://www.websocket.org/

3 Introduction

This document describes the design for the new Web-XI protocol.

This is the protocol that will be implemented for LAN-XI G2 devices.

The document was written before the name Web-XI was invented, so it often talks about the GISP protocol and about GISP devices. This is synonymous to Web-XI protocol and to G2 devices.

The protocol will be divided in two parts. First part is the command protocol. This protocol is used to issue commands to a GISP device, receive events, and receive data for monitoring purposes. The other part is the *Streaming Protocol*. This is used to receive a stream of sample data and of relevant supporting data such as data quality information, certain events etc.

The first chapters of this document contain general design decisions, such as the structure of commands, how asynchronous events are handled etc.
The later chapters contain descriptions of the actual commands in the protocol.

4 General guidelines for command protocol

The command protocol is used to issue commands to a GISP device, receive events, and receive data for monitoring purposes.

4.1 REST

The command protocol will be based on REST. See "Restful Web Services" by Leonard Richardson and Sam Ruby for details.

Guidelines for using the 5 HTTP methods:

- **GET**: Used to get information. There must be no side effects
- **DELETE**: Delete something.
- **PUT**: Used to set/change information on existing resource.
- **POST**: Used to create a resource and possibly set values on it. Typically the POST command will return the address of the new resource, so it can be accessed and maybe deleted later.
- **OPTIONS**: Used to get information about what's possible to do on the resource.

In general we are going to place as much detail in the URL as possible, only properties and values are left out.

4.2 JSON

The body part of REST commands and of replies will be transmitted as JSON (see http://www.json.org). The encoding is ***Utf-8***.

It is possible to request a recursive data structure by setting "recursive" to true on the URI.:

```
GET /rest/a/b?recursive=<Boolean>
```

Where <*Boolean*> is either **true** or **false** (the default is **false**).

If *recursive* is true, then all child nodes are returned fully unless otherwise stated:
Certain node types may be left out of a recursive GET, the user will have to get those explicitly. This will be documented on the relevant resources

The body for a leaf node contains an object with the name and value for this node.
The body for a non-leaf node contains values for all immediate children. Child leaf nodes are returned fully, child non-leaf nodes will only contain the names of the nodes, the value will be null.

On **SET** methods the body may (but is not required to) contain information about children. The information will be set if given.
Resources may be left out when setting values; only resources specifically mentioned will be modified.

See the example below.

4.2.1 Example

Assume we have the following REST structure:

/rest/a/
/rest/a/b (value 2)
/rest/a/c
/rest/a/c/d (value 4)

GET /rest/a returns:

```
{
    "b" : 2,
    "c" : null
}
```

GET /rest/a?recursive=true:

```
{
    "b":  2,
    "c" : { "d" :  4 }
}
```

GET /rest/a/b returns:

```
{ "b" : 2 }
```

GET /rest/a/b?recursive=true:

```
{ "b" : 2 }
```

GET /rest/a/c returns:

```
{ "d" : 4 }
```

GET /rest/a/c?recursive=true returns:

```
{ "d" : 4 }
```

GET /rest/a/c/d returns:

```
{ "d" : 4 }
```

GET /rest/a/c/d?recursive=true returns:

```
{ "d" : 4 }
```

PUT on /rest/a that sets the b's value:

```
{ "b" : 22 }
```

PUT on /rest/a that changes b and d:

```
{
    "b" :  22,
    "c" : { "d" : 44 }
}
```

PUT on /rest/a that only changes d:

```
{
   "c" : { "d" : 44 }
}
```

4.3 Actions

Actions, such as resetting the device, doing a TEDS detect or starting the generator is done with a PUT command using the ?= notation in the URI:

```
PUT <URI>?Action=<WantedAction>
```

Where <*URI*> is the address of the relevant REST node, and <*WantedAction*> describes what to do. It is possible to specify an argument using "&Argument=<argument>"

Example:
TEDS detect might be initiated on all commands with

```
PUT /rest/channels/all/transducer?Action=Detect
```

4.4 Triggered actions and PUTs

Actions and assigning values to nodes can be synchronized by stating that the PUTs and/or actions should be performed at a trigger.
When this is done, the action/PUTs are not executed until a module is asked to activate the trigger. The system then guarantees that the actions and PUTs are executed at the same time.

This is the syntax for stating that a PUT should be triggered:

`PUT` *<URI>*`?AwaitTrigger=`*<TriggerId>*

All puts to nodes listed in the PUT will await the given trigger.

The syntax for stating that an action should be triggered is:

`PUT` *<URI>*`?Action=`*<WantedAction>*`&Argument=`*<arg>*`&AwaitTrigger=`*<TriggerId>*

Here <TriggerId> is a number from 0 to 7.

The trigger is then executed by sending the command "PerformTrig" to module:

`PUT` */Rest/Device*`?Action=`*PerformTrigger*`&Argument=`*<TriggerId>*

This command can be sent to any of the modules in the domain.

The module receiving this action is then responsible for multicasting a UDP trigger package to all modules.

It is also possible to specify a time for the execution of an action or a PUT. This is done by specifying:

`&Time=`*<Time>*

Where *<Time>* is an absolute time.

It is also possible to specify a delay:

`&Delay=`*<Delay in milliseconds>*

If this is used, then the command will be executed this amount of time after the command has been received in the module.
It is illegal to specify both an absolute time and a delay.

You can specify the time or delay together with "`&AwaitTrigger=`" or without it. If both are specified, then the action is done when the first of the two conditions are fulfilled.

4.5 Versioning

New firmware should work together with old pc software. This means that the pc should inform the device what version of the protocol it is using. The device will then either handle the request, or report an error.

In general all changes to the protocol should be backwards compatible (e.g. by only adding optional resources to the protocol).
If a more complex change is needed (which we strive to avoid as much as possible), then the version of the protocol will be changed.

Versioning is handled using content negotiation:

- The client sends a request, filling out the *Accept* header.
- This tells the server which version of the protocol the client understands. If no header is given, then the server will use the latest version.
- The server returns the relevant answer and sets the *Content type* header accordingly

Note that the version is optional in the request; this is to allow for a normal web browser to query the module**. All application should, however, specify the version in the request to be more stable for version changes.**

Example:

```
===>
GET /rest/Inputs/1/SampleRate HTTP/1.1
Accept: application/GISP2
<===
HTTP/1.1 200 OK
Content-Type: application/GISP2

..data in version 2 format here..
```

4.6 Caching

Use of the *Cache-Control* header is important for use of the applications.

In general caching of a response can be turned off by using the following header:

```
Cache-Control: no-cache
```

Allowing caching of a response for a certain number of seconds can be done with:

```
Cache-Control: max-age=3600
```

The rules for caching and disallowing caching are not set in stone, but here are some examples of what to disable caching on:

- Any URI containing a query string (?=). E.g. *…/generator1?action=start*
- Any resource containing volatile information, such as a property that another client might change, or a property that is changed by the device.
- Any URI that has a side effect when GET is used. (We shouldn't have any of these)

It is ok to uses cache for

- Resources that change slowly such as CHESS information, version numbers etc.

4.7 Guidelines for representing object model in REST

4.7.1 Selection of a branch in the object model

If only one branch can be active at the same time:

When a node only can has several child nodes, where only one of these can be active at a time, then the active node is chosen by a leaf node with the name "`Select`". Its value is the name of the active node.

Example:

`../Functions/Select`	(value "`Preamp`")
`../Functions/Direct`	(has child nodes)
`../Functions/Preamp`	(has child nodes)
`../Functions/CCLD`	(has child nodes)

In this example the active function is "`Preamp`", the other possibilities ("`Direct`" and "`CCLD`") are not active in the hardware. The user may set values on these nodes, but these values will not be used until the relevant node is selected.

If more than one branch can be active at the same time:

When a node can have several child nodes, where more than one can be active at a time, then the child nodes all have a leaf node with the name "`Active`". Its value is `true` for the active nodes, `false` for others.

Example:

`../Analog1/Input1`	(has child nodes)
`../Analog1/Input1/Active`	(value "`true`")
`../Analog1/Input2`	(has child nodes)
`../Analog1/Input2/Active`	(value "`false`")
`../Analog1/Input3`	(has child nodes)
`../Analog1/Input3/Active`	(value "`true`")

"Input1" and "Input3" are active in this example.

4.7.2 PUT errors and constraint handling

If there are constraints between properties (such as the sum of the values must be less than something), then an error (409 "Conflict") should be issued when the constraint is broken. The device should not try to modify values to satisfy the constraint.

If a PUT fails, then it will return one of the following errors:

- **403 ("Forbidden"):** The property cannot bet set at this time (because of e.g. wrong state).
- **409 ("Conflict"):** A constraint between several values would be broken if the property is set.
- **400 ("Bad Request"):** The property cannot be set to the given value.

If one of these error codes is returned, then the answer contains the following information:

- Whether the request was partially performed or it was not performed at all.
- The address of the resource where the update failed.
- A message in English describing what went wrong.

This is a JSON object:

```
{
      "Partial" : <true or false>,
      "URI": <uri>,
      "Message" : <Error message>
}
```

The "Partial" field is set to true if this is a compound update (more than one property is set), and the update was partially performed. It will be false if none of the properties were changed.

5 CHESS

CHESS stands for "Channel Embedded System Sheet" and is the idea of having a way to query a device of its capabilities.
This includes stuff like: Number of channels, available sample frequencies, possible attenuator settings etc.

Chess information about a resource can be requested using the OPTIONS method.

This answer to such a request should list the allowable methods in the *Allow* header, e.g.:¨

```
Allow: GET, PUT
```

The answer should contain GISP specific CHESS information.

We have not yet decided the format, but the following types of information should at least be included:

- A description of the resource in English.
- A list of child nodes.
- If the URI is a leaf node, then there should be a specification of the value domain.
- If actions are legal on this node, then the actions should be listed (GET, PUT etc.).
- A list of the states where its possible to update the node)

The value domain can be either an enumeration or an interval:
An **enumeration** lists a set of possible values and corresponding descriptions (in English) of these values, furthermore it defines a Unit and a data type.
An **interval** is defined by a minimum and maximum value (both min and max are included in the interval) plus a resolution. Furthermore it defines a Unit and a data type.

TBD: Find another name for this. It is not just for channels.

6 Time

We are going to use the concepts from the PULSE Reflex time format in the GISP protocol, but expand it to be an absolute time

6.1.1 The PULSE Reflex Relative time

The relative time in PULSE Reflex is contained in a 64 bit integer, and contains a number of "ticks". There are $2^{22} * 3^2 * 5^3 * 7^2$ ticks per second, so each tick is approximately 4.33 picoseconds. The maximum duration of this time is approximately 461 days.

We can specify the following sample frequency families with this:

Name	Frequency	Frequency Range
65 kHz family	$2^n * 3^0 * 5^0 * 7^0$	From 1 to 2^{22} Hz (4 MHz)
51,2 kHz family	$2^n * 3^0 * 5^2 * 7^0$	From 25 to $2^{22} *$ 25 Hz (105 MHz)
256 kHz family	$2^n * 3^0 * 5^3 * 7^0$	From 125 to $2^{22} *$ 125 Hz (524 MHz)
48 kHz family	$2^n * 3^1 * 5^3 * 7^0$	From 375 to $2^{22} *$ 375 Hz (1.6 GHz)
44,1 kHz family	$2^n * 3^2 * 5^2 * 7^2$	From 11025 to $2^{22} *$ 11025 Hz (46 GHz)

Here a family is defined as a group of frequencies which are equal except for a factor of 2^n.
The frequency listed in the family "name" is just a typical frequency within the family.

Note that this time wraps in 64 bit after 461 days. We are not satisfied with such a short wrap time, and we need an absolute time, so this leads to the format described in the next section.

6.1.2 The GISP absolute time

The time for GISP is based on the ideas in the PULSE Reflex time described above.

It is a 12 byte quantity in 2 parts:

The first part is the Family; it is a 32 bit quantity defining the exponents used for 2, 3, 5 and 7 (one byte for each). This defines the size of a clock tick:

Name	Description	Size in bytes
K	Exponent for 2	1
L	Exponent for 3	1
M	Exponent for 5	1
N	Exponent for 7	1

This family corresponds to a tick size of:

$$2^{-k} * 3^{-l} * 5^{-m} * 7^{-n}$$ **seconds**

The second part is the Count; it contains the number of ticks since 0 hours on 1 January 1970 (Modified Julian Day 40 587.0).
This is a 64 bit number.

6.1.3 Important notes about the time format

Compatibility with PULSE Reflex

Note that PULSE Reflex signal analysis only support part of the values that this time format supports.

PULSE Reflex sample frequencies that can be expressed by:

$$2^k * 3^l * 5^m * 7^n \text{ Hz}$$

Where

$$k \leq 22, l \leq 2, m \leq 3, n \leq 2$$

Wrap

To avoid wraps the family should be chosen so that the time to wrap, which is:

$$\frac{2^{64}}{2^k * 3^l * 5^m * 7^n} \text{ seconds}$$

is longer than 100 years.

The following table contains suggested values for n that allows for this:

Name	Family	Max Frequency	Time until wrap
65 kHz family	$2^{32} * 3^0 * 5^0 * 7^0$	4.2 GHz	136 years
51,2 kHz family	$2^{27} * 3^0 * 5^2 * 7^0$	3.3 GHz	174 years
256 kHz family	$2^{25} * 3^0 * 5^3 * 7^0$	4.2 GHz	139 years
48 kHz family	$2^{23} * 3^1 * 5^3 * 7^0$	3.1 GHz	186 years
44,1 kHz family	$2^{18} * 3^2 * 5^2 * 7^2$	2.9 GHz	202 years

PTP and LXI Compatibility

The PTP timestamp format (as defined by IEEE 1588) is different from the above proposed format:

Name	Data type	Description
Seconds	6 bytes	Time in seconds. In LXI this value is split in two parts: The 4 byte "S*econds*" field and the 2 byte "*Epoch*" field in the header
Nanoseconds	6 bytes	In LXI this is divided in the 32 bit "*Nanoseconds"* field and the 16 bit "*Fractional_Nanoseconds*" field.

Zero time for this is 0 hours on 1 January 1970 (Modified Julian Day 40 587.0), which is the same as proposed for GISP format above.

The GISP application will have to convert to and from this format when implementing LXI and PTP specific protocols.

7 Data from the device

The device generates data of different kinds:

- **Results** are for instance measured data or the result of some signal analysis done on the device. Results come from a "Signal" in the object model.
- **Data Quality** is also a kind of result, but is classified separately here because it is handled a bit different. (Value changes are transmitted instead of all values)
- **Events** describe something that has happened on the device, such as a state change, a trigger or a configuration change.

Such data is sent to the client as **Messages** and can be sent with one of the following methods:

- **Streaming:** The Client can request that certain message types be streamed to it**.** A stream can contain all message types.
- **EventManager:** The client can request messages containing events that have happened since a client specified time. Only events can be requested this way.
- **Monitor:** The client can request Results from a signal for monitoring purposes. The client is not guaranteed to receive all data if using this method.

This chapter first describes the different methods for sending data to the client.
The final sections of the chapter describe the different types of messages that can be sent.

7.1 Streaming

The device can deliver results, quality data and events over a Websocket[1] connection. It is also possible to request that the data be written to storage on the device (such as a SD card).
This is called streaming. The protocol tries to transfer all requested data and guarantees that any data loss will be reported as a status event.

The protocol allows for several clients to request data streaming. The first implementation will probably only will allow the controlling client to create a single stream.

7.1.1 Managing streaming

The client configures the data streaming by using REST.
This is done by executing a POST:

```
===>
POST /rest/StreamManager HTTP/1.1

{
    "Destination" : <Destination>,
    "Name" : <Name>,
    "Signals" :
    [
       <SignalId>, <SignalId>, …
    ]
    "MessageTypes" :
    [
       <MessageType>, <MessageType>, …
    ]
}
<===
HTTP/1.1 200 OK

URI to new Stream Resource
```

[1] Websocket is a standard internet protocol defined in RFC 6455

Here <Destination> is either "WebSocket" for streaming to a Web Socket, "Broadcast" for UDP broadcast or "SD" for streaming to SD card. Version 1 will probably only implement "WebSocket".

<Name> is the name of the stream, on SD cards it will be the name of the file (with optional added extra characters to make it unique).

<SignalId> identifies which signals to stream. Id 0 should never be used for a signal.

<MessageType> lists what message types to stream.

The following commands are available on the returned stream resource:

GET <URI>/Destination: Returns the destination
GET <URI>/Name: Returns the name
GET/PUT <URI>/MessageTypes: Get/set the message types that will be written to this stream.
GET/PUT <URI>/Signals: Get/set the signals that will be written to this stream.
PUT *<URI>*?Action=*OpenWebSocket:* Open WebSocket connection on this URI

NOTE: Maybe "Action=OpenPort" instead of OpenWebSocket if it is too difficult to implement Websockets.

DELETE <URI>: Closes the stream and deletes the resource. This also happens automatically when the WebSocket connection to the stream is closed or if no connection is made within a minute.

7.1.2 Timing of messages in the stream

Each message in the stream contains a time stamp, this is however not enough information for the client to efficiently handle the data.

For instance, the client will need to

- Match quality data to sample data
- Be sure to not discard any data before knowing if a trigger event is happening in the data.

The device must follow certain guidelines to help the application with these tasks:
- Data describing an input channels data must be transmitted before the data they describe (e.g. quality of channel 1 before channel 1.)
- General status data must be sent before any of the input data.

The problem is that the client needs to know when it is ok to discard old data.

To facilitate this, the device should emit a "heartbeat" in the data stream at certain intervals.
The device guarantees that no more data will be sent with a timestamp before the one given in the heartbeat message.

7.1.3 Data types and value domains

Data types below are typically specified as being signed, even when the value domain obviously is in fact unsigned (such as the number of values in a message). The reason for this is to be compliant with the Microsoft "Common Language Specification" (CLS).

It will of course be an error to specify a negative number of values for a count even though it in fact is possible.

Ids (such as **SignalId**) are also signed, in this case all values except for 0 are valid ids unless otherwise specified.

The value 0 is used to indicate an unknown Id.

7.1.4 Strings

Strings are in UTF-8 format. They are represented as a byte count followed by that number of bytes when part of binary data:

Name	Length (in bytes)	Contents
Count	2	The number of bytes (not characters) in the UTF8 string as an Int16, excluding padding bytes (see below). If Count is 0, then the string is empty. Count may not be negative.
Bytes	Count rounded up to a multiple of 4 bytes	The actual content of the string. (Note that Count may not equal the number of characters in the string since a single character may be from 1 to 4 bytes in length). Note that the number of bytes transmitted is rounded up to a multiple of 4 bytes, so any remaining bytes should be set to 0

With 4 byte padding rule, the **Count** is not necessarily the number of bytes following, it is the number of relevant bytes. The actual number of bytes is **Count** rounded up to a multiple of 4.

7.1.5 Streaming message format

The stream consists of a *messages* transmitted after each other. The format of a message is a header followed by content:

Name	Length (in bytes)	Contents
Magic	2	The ASCII characters "BK".
HeaderLength	2	The length of the rest of the header up to but not including the content length. Currently this is 20[2].
MessageType	2	Identifies the content of the message
Reserved1	2	For future use. Set to 0
DebugSupport	4	Reserved for debugging
Timestamp	12	Time of message
ContentLength	4	Length of the message content in bytes

This header is followed by the content part of the message:

Content	ContentLength	Depends on the message type.

The actual content depends on the message type. See the chapter on Message Types below.

All multi-byte values in a message are transmitted in little endian byte order.

[2] If new fields are needed in the header, then they should be appended after the "Timestamp" field. Existing fields must never be removed from the header, and the length of the header should always be a multiple of 4 bytes. If these rules are followed, then the header length will always increase for each new header version. The client can use HeaderLength as a kind of Header version field.

7.2 Event Manager

In order to receive events, the client will have to send a request to an event manager.
The request looks like this:

```
===>
GET /rest/EventManager HTTP/1.1

{
   "Time" : <RequestedTime>,
   "EventTypes" :
   [
       <MessageType>, <MessageType>, …
   ]
}
<===
HTTP/1.1 200 OK

{
    "Time" : <ActualTime>,
    "EventsMissed" : <Boolean>,
    "Events" :
    [
       <Message>, <Message>, …
    ]
}
```

Here <Message> has the format:

```
{
    "MessageType" : "<Type of Message>",
    "TimeStamp" :
    {
       "Family": <Time Family>
       "Count": <Time Count>
    },
    <Message data as described in the following sections>
}
```

The client specifies that it wants events after the given time, and which events it wants.
It is not possible to request SignalData and DataQuality messages from the event manager.
Signal data can be requested from Monitor nodes, and if the application request status events, then it will receive events about data quality as well.

The reply contains the current actual time of the device; the reply contains events up to this time. "EventsMissed" will be true if there was an event overrun, and finally the reply contains an array of the actual events.

7.3 Monitoring a signal

The client may obtain a number of results from a given channel by issuing a GET to a monitor resource.

The client specifies that it wants a certain number of values. The count is optional; it defaults to 1 if not specified.

```
===>
GET /rest/…/Signal7/Monitor HTTP/1.1

{
   "Count" : 1024
}
```

The reply contains a *SignalData* message (see below) with the requested values.

The device does **NOT** guarantee that 2 requests sent after each other will result in consecutive data; there may be a gap.

7.4 Message Types

This section describes the different types of messages that can be sent to the client.

Message Type	Value	Description
SignalData	1	Data values from a signal. This cannot be requested from the event manager. It is available via the stream protocol and from Monitor resources.
DataQuality	2	Indicates the data quality of a certain signals data. The quality message is typically only generated when the quality of a signal changes. It is only available via the stream protocol.
State	3	Indicates the state of the device.
Status	4	Status event, such as PTP sync lost.
Trigger	5	Trigger events from a signal
Configuration	6	Sent when the device's configuration has changed.
Heartbeat	7	Sent from the device. It is guaranteed that there will be no further messages with a timestamp before the heartbeat message's timestamp. It is only available via the stream protocol.
Interpretation	8	Describes how to interpret Signal Data. It is only available via the streaming protocol
DebugMessage	9	A message containing debug information

Each message type has an associated structure that is sent whenever the event happens.

The following sections describe the structures. The description covers both the streaming format and the REST event format.

7.4.1 SignalData

Messages of this type contain data values from a signal. It will not be available as an event via REST; it is only available via the stream protocol. It, however, will be possible to monitor signals via the REST protocol by issuing a GET to a monitor resource under the individual channel.

The message content is:

Name	Stream type	REST type	Description
NumberOfSignals	Int16	Number	Number of signals with data in this message.[3] The number of signals in the message must be greater than zero.
Reserved	Int16		Reserved. Set to 0

The following structure is then repeated "NumberOfSignal" times:

Name	Stream type	REST type	Description
SignalId	Int16	Number	Identifies the signal that produced the following values.
NumberOfValues	Int16	Number	Number of values. The number of values must be greater than zero.
Values	Array of values	Array of Number	Data from the signal. In the case of a vector a value means a single vector. This means that the actual number of scalar values (e.g. floats) transmitted is NumberOfValues * VectorLength from the interpretation message.

[3] It is up to the device to decide whether it wants to group signals into one message or not.

7.4.2 DataQuality

Messages of this type contain quality information about a certain signal. The quality message is typically only generated when the quality of a signal changes.

The message content is:

Name	Stream type	REST type	Description
NumberOfSignals	Int16	Number	Number of signals with data in this message. The number of signals in the message must be greater than zero.

The following structure is repeated "NumberOfSignal" times:

Name	Stream Type	REST type	Description
SignalId	Int16	Number	Identifies the signal that produced the quality.
Validity	DataValidity flags	String	Quality info
SettlingLevel	SettlingLevel enum	String	Settling level

DataValidity flags (Int16)

The DataValidity flags (Int16) is exactly as it is in Reflex. The values are "flags" that can be OR'ed together if more than one condition exist.

Name	Value	Description
Valid	0	Data is valid
Unknown	1	Data quality is unknown
Clipped	2	Data is clipped
Settling	4	Data is settling
Invalid	8	Data is invalid
Overrun	16	Overrun happened right before this value.

SettlingLevel enum (Int16)

The SettlingLevel enum (Int16) gives more detail about settling than the Settling flag above.

Name	Value	Description
Unsettled	0	Unsettled
Above10	1	Settling level is at or above 10 %
Above50	2	Settling level is at or above 50 %
Above90	3	Settling level is at or above 90 %
Settled	4	Settling done.

When settling starts, the device always should output the Unsettled level, and end with the Settled level.
The value of the Settling flag in the DataValidity enum is false when (and only when) Settling level is "Settled".

7.4.3 State

This message contains the state of the device. It is typically only sent when the state changes.

The message content is:

Name	Stream type	REST type	Description
State	State enum	String: Name from state enum	16 bit enumeration of the possible states.

State enum (Int16)

Possible values for "State":

Name	Value	Description
Unknown	0	State not set (should never happen)
?? Idle		
?? Measuring		
?? Armed		
??		

The size of the enum is Int16.
TBD: The possible states will be determined later.

7.4.4 Status

Status event, such as PTP sync lost.

The message content is:

Name	Stream type	REST type	Description
SignalId	Header	Number	Optionally Identifies the signal that produced the event. (0 if irrelevant)
Status	Status enum	String: Name from Status enum	16 bit enumeration of the possible stati.
Value1	Int32	Number	Content depending on "Status"
Value2	Int32	Number	Content depending on "Status"
String	See Strings7.1.4	string	Optional String describing the status.

Status enum (Int16)

Possible values for "Status":

Name	Value	Contents of the Value fields	Description
Unknown	0	Not used	Status not set (should never happen)
CCLDOverload	1	**Value1**: Overload enum	Indicates change in CCLD overload status.
CVLDOverload	2	**Value1**: Overload enum	Indicates change in CVLD overload status.
CommonModeOverload	3	**Value1**: Overload enum	Indicates a change in Common mode overload status.
InputProtection	4	**Value1**: 1: Input protection on, 0: Input protection off	Indicates a change in Input protection status.
CableBreak	5	**Value1**: Overload enum	Indicates change in Cable Break status
Fan	6	**Value1**: Fan speed in %	Fan event
Temperature	7	**Value1**: Alarm level (0-5). **Value2**: Temperature in degree Celcius	If Value is non-zero, then temperature is high, otherwise temperature is ok.
Power	8	**Value1**: PowerSource enum in top 16 bit, PowerMode enum in low 16 bit. **Value2**: Current estimated power consumption in mW.	Power status

PowerAlarm	9	**Value1:** Estimated remaining time before power is gone in seconds	This status is sent when power is getting low
Synchronization	10	**Value1**: Precision in nanoseconds. **Value2**: SynchronizationState enum	Synchronization status
Command completed	11	**Value1:** Completion status (0: Ok) **String:** The command that completed	This status is sent when an asynchronous command has completed

Overload enum (Int16)

The Overload enum (Int16) contains:

Name	Value	Description
Unknown	0	Overload status is unknown.
Ok	1	Value is ok
Low	2	Value is to low
High	3	Value is too high
OutOfRange	4	Value is overloaded, we do not know if its too low or high

PowerSource enum (Int16)

The PowerSource enum (Int16) contains:

Name	Value	Description
Unknown	0	Unknown power source. Should not be used.
PoE	1	Power over Ethernet
DC	2	
AC	3	
Battery	4	
USB	5	
Backup	6	

PowerMode enum (Int16)

The PowerMode enum (Int16) contains:

Name	Value	Description
Unknown	0	Unknown mode.
Normal	1	Normal operations mode
PowerSave	2	Power save mode.
Off	3	Power is off (after this message)

SynchronizationState enum (Int16)

The SynchronizationState enum (Int16) contains:

Name	Value	Description
Unknown	0	Unknown synchronization state
OutOfLock	1	Device is not locked to anything
Locking	2	The device is trying to lock
Locked	3	The device has locked synchronization

7.4.5 Trigger

Trigger events from a signal (input or output)

The message content is:

Name	Stream Type	REST type	Description
SignalId	Header	Number	Identifies the signal that produced the trigger.
TriggerType	TriggerType enum	Number	Identifies the trigger.

TriggerType enum (Int16)

The TriggerType enum (Int16) contains:

Name	Value	Description
Unknown	0	Unknown trigger
Level	1	Level trigger
Start	2	Waveform start on output signal.
??	3	
??	4	

TBD: To be determined when implementing

7.4.6 Configuration

This event is generated when the device's configuration has changed.

The message content is:

Name	Stream type	REST type	Description
URI	See Strings7.1.4	String	URI for the resource that was changed.
DataType	ContentDataType enum	String	Type of the value
Value	Depends on "DataType"	Depends on "DataType"	The new value for the resource identified by URI.

The `ContentDataType` enum is described in section ContentDataType enum (Int16).

7.4.7 Heartbeat

This event is generated at certain intervals (defined by the device). The device guarantees that there won't be any messages with a timestamp earlier than the timestamp in the heartbeat message.

There is no message content.

7.4.8 Interpretation

This event is generated when information about one or more signals changes.

A piece of information about a signal, such as for instance its unit, is called a descriptor.

All relevant descriptors are sent automatically when a new stream is opened.

The message contains one or more descriptors.

The content of a single descriptor is as follows:

Name	Stream type	REST type	Description
SignalId	Int16	Number	Identifies the signal that this descriptor refers to. If SignalId is 0 then the descriptor is for all signals.
DescriptorType	DescriptorType enum	String	Identifies the descriptor, see table of possible descriptor types below
Reserved	Int16		Reserved for future use, set to 0
ValueLength	Int16	Number	Length of value in bytes, not including any padding that may have been added to Value to make it a multiple of 32 bit word
Value	Depends on descriptor type	Depends on descriptor type	The value of the descriptor. This value must be a multiple of 32 bit word

DescriptorType enum (Int16):
The DescriptorType enum (Int16) contains:

Name	Value	Description	Type of corresponding descriptor value	Default value
DataType	1	Data type of a single value in signal	ContentDataType enum	None
ScaleFactor	2	Scale factor to multiply on each value in signal data to obtain a value in the specified unit.[4]	Float64	1.0
Offset	3	Offset to add to each value in signal data to obtain a value in the specified unit.	Float64	0.0
PeriodTime	4	Time between 2 consecutive values of the signal.	Timestamp	None
Unit	5	The SI unit of the signal. The corrected value (after applying scale factor and offset) will be in this unit.	See Strings 7.1.4	Empty string
VectorLength	6	Length of one value. 0 means scalar.	Int16	0

This table is certain to grow in the future. Applications that do not support a given descriptor should just ignore it and skip to the next descriptor in the message.

[4] The scale factor is applied before the offset (CorrectedValue = ScaleFactor * signalValue + Offset)

ContentDataType enum (Int16)

The ContentDataType (Int16) enum has the following possibilities:

Name	Value	Description
Unknown	0	ContentDataType not set (should never happen)
Byte	1	8 bit byte
Int16	2	16 bit integer
Int24	3	24 bit integer
Int32	4	32 bit integer
Int64	5	64 bit integer
Float32	6	32 bit float
Float64	7	64 bit float
Complex32	8	32 bit complex float
Complex64	9	64 bit complex float
String	10	UTF8 string. Content is an Int16 length followed by that number of bytes. See Strings 7.1.4

7.4.9 DebugMessage

This event is generated when the device wants to output a debug message.

The message content is:

Name	Stream type	REST type	Description
Severity	Severity enum	String	Severity of message
Domain	Int16	Number	Id indicating what subsystem of the device that reported the message.
String	See Strings7.1.4	String	Text message
Count	Int16	Number	Number of Type+Value pairs below.
TypeAndValue	ContentDataType enum followed by a value	String, String or Number	Data type followed by a value of that data type

Domain (Int16)

The Domain can have the following values:

Name	Value	Description
Unknown	0	Domain is not specified
???	**1**	**TBD**

Severity enum (Int16)

The Severity enum (Int16) has the following possibilities:

Name	Value	Description
Information	0	Message is informational
Warning	1	Message is a warning message
Error	2	Message is an error message

8 Error code usage

This chapter describes which error codes the device should return, and what they mean

8.1 101 ("Switching Protocol")

The host has accepted a WebSocket connection.
After this is received the WebSocket connection is established, and the connection is in the OPEN state.
The WebSocket protocol is described in RFC 6455 (See http://tools.ietf.org/html/rfc6455)

8.2 200 ("Ok")

Operation succeeded. On GET the resulting document is the body of the message.

Entity-body: For GET requests, a representation of the resource the client requested. For other requests, a representation of the current state of the selected resource, or a description of the action just performed.

GISP: This is the default ok response. Use this unless 201 ("Created") or 202 ("Accepted") should be used. Place the answer in JSON format in the Entity-Body.

8.3 201 ("Created")

The server sends this status code when it creates at the client's request.

Response headers: The *Location* header should contain the canonical URI to the new resource.

Entity-body: Should describe and link to the newly created resource. A representation of that resource is acceptable, if you use the *Location* header to tell the client where the resource actually is.

GISP: Use this answer when the user uses POST to create a child node (e.g. creating a Measurement). Return the URI for the child node

8.4 202 ("Accepted")

The client's request can't or won't be handled in real time. It will be processed later
The pending request should be exposed as a resource so the client can check up on it later.

Response header: The pending request should be exposed as a resource so the client can check up on it later. The *Location* header can contain the URI to this resource.

Entity-body: If there's no way for the client to check up on the request later, at least give an estimate of when the request will be processed.

GISP: Use this answer when the user uses POST to create a child node for something that is not completed at once. The user should be able to wait for completion by accessing the child node:

```
GET <child URI>?action=WaitForCompletion
```

8.5 400 ("Bad Request")

This is the generic client-side error status, used when no other 4xx error code is appropriate. It's commonly used when the client submits a representation along with a PUT or POST request, and the representation is in the right format, but it doesn't make any sense.

Entity-body: May contain a document describing why the server thinks there's a client-side error.

GISP: Use this to report a malformed or otherwise bad request. Use one of other 4xx messages instead if one is applicable.

8.6 401 ("Unauthorized")

The client tried to operate on a protected resource without providing the proper authentication credentials.
If the server doesn't want to acknowledge the existence of the resource to unauthorized users, it may lie and send a 404 ("Not Found") instead of a 401.

Response headers: The *WWW-Authenticate* header describes what kind of authentication the server will accept.

Entity-body: A document describing the failure: why the credentials (if any were provided) were rejected, and what credentials would be accepted. If the end user can get credentials by signing up on a web site, or creating a "user account" resource, a link to the sign up URI is useful.

GISP: Use this to tell the user that he isn't authorized to perform the action. Examples of use: The user tries to change the setup, but is not the controlling client.

8.7 403 ("Forbidden")

The client's request is formed correctly, but the server doesn't want to carry it out. This is not merely a case of insufficient credentials: that would be 401 ("Unauthorized"). This is more like a resource that is only accessible at certain times, or from a certain IP address.

Entity-body: Optionally, a document describing why the request was denied.

GISP: Use this if the state of the device forbids execution of the request. Example: Requesting a TEDS detect while measuring.

8.8 404 ("Not Found")

404 indicates that the server can't map the client's URI to a resource.

GISP: Use this response if the user specifies an URI that we do not recognize.

8.9 405 ("Method Not Allowed")

The client tried to use an HTTP method that this resource doesn't support. For instance, a read-only resource may support only GET and HEAD. Another resource may allow GET and POST, but not PUT and DELETE.

Response headers: The *Allow* header lists the HTTP methods that this resource does support. The following is a sample header:

```
Allow: GET, POST
```

GISP: Use if the client uses a method that is not supported at this resource.

8.10 408 ("Request Timeout")

If an HTTP client opens a connection to the server, but never sends a request (or never sends the blank line that signals the end of the request), the server should eventually send a 408 response code and close the connection.

GISP: I assume that the web server handles this.

8.11 409 ("Conflict")

This is sent when the client tries to perform an operation that would leave one or more resources in an inconsistent state.

Response headers: If the conflict is caused by the existence of some other resource, the *Location* header should point to the URI of that resource: that is, the source of the conflict.

Entity-body: Should contain a document that describes the conflicts, so that the user can resolve them if possible.

GISP: Use this in cases where a constraint between properties would be broken.

8.12 411 ("Length Required")

Content-Length header field expected but wasn't given.

GISP: Probably handled by the web server.

8.13 413 ("Request Entity Too Large")

The request is too large for the server to handle.

GISP: Probably handled by the web server.

8.14 415 ("Unsupported Media Type")

The server sends this status code when the client sends a representation in a media type it doesn't understand.

GISP: We us the media type field for protocol versioning, so use this error code if the device does not support the requested version. Be sure to place a description in the entity body.

8.15 500 ("Internal Server Error")

This is the generic server error response. Most web frameworks send this status code if they run request handler code that raises an exception.

GISP: There is a serious problem on the device. There is a text in the body describing the problem. This should be used for non-recoverable errors such as asserts etc.

8.16 503 ("Service Unavailable")

This status code means that the HTTP server is up, but the underlying web service isn't working properly. The most likely cause is resource starvation: Too many requests are coming in at once for the service to handle them all.

Since repeated client requests are probably what's causing the problem, the HTTP server always has the option of refusing to accept a client request, rather than accepting it only to send a 503 response.

GISP: We may need to deny requests from monitoring clients if the device is very busy serving the controlling client. This answer can be used in this case.

Appendix E

Tokens used in the JSON tokenizer

The following is a description of the token types identified by the JSON tokenizer described in Section 3.5.2. They are all part of the enum `JSONToken_t`

Each token type is presented with valid predecessors and successors, as well as information regarding any value they may return.

JSONToken_Start – Beginning of JSON
Used as initial state when parsing begins.

Valid predecessors	None
Valid successors	JSONToken_BrL
Value	No value associated

JSONToken_BrL – Left bracket ({)
The token signals the beginning of a JSON object.

Valid predecessors	JSONToken_Start, JSONToken_Colon
Valid successors	JSONToken_BrR, JSONToken_String
Value	No value associated

JSONToken_BrR – Right bracket (})
Signals the end of a JSON object.

Valid predecessors	JSONToken_BrL, JSONToken_BrR, JSONToken_SqR, JSONToken_JSON, JSONToken_String, JSONToken_Int, JSONToken_Float, JSONToken_True, JSONToken_False, JSONToken_Null
Valid successors	JSONToken_BrR, JSONToken_Comma
Value	No value associated.

JSONToken_SqL – Left square bracket ([)
Starts an array of values. From this point the parser should be in an array building state.

Valid predecessors	JSONToken_Colon
Valid successors	JSONToken_SqR, JSONToken_Int, JSONToken_Float
Value	No value associated.

JSONToken_SqR – Right square bracket (])
Ends an array of values. The array parsed has been built, and the value should be set on the selected node. From this point the parser should return to normal mode.

Valid predecessors	JSONToken_Int, JSONToken_Float
Valid successors	JSONToken_BrR, JSONToken_SqL, JSONToken_Comma
Value	The array of values built since JSONToken_SqL.

JSONToken_Colon – Colon (:)
Delimiter between an identifier (string, node name) and a value.

Valid predecessors	JSONToken_String
Valid successors	JSONToken_BrL, JSONToken_SqL, JSONToken_JSON, JSONToken_String, JSONToken_Int, JSONToken_Float, JSONToken_True, JSONToken_False, JSONToken_Null
Value	No value associated.

JSONToken_Comma – Comma (,)
Separator between name:value pairs and between values in an array.

Valid predecessors	JSONToken_BrR, JSONToken_SqR, JSONToken_JSON, JSONToken_String, JSONToken_Int, JSONToken_Float, JSONToken_True, JSONToken_False, JSONToken_Null
Valid successors	JSONToken_String, JSONToken_Int, JSONToken_Float
Value	No value associated.

JSONToken_String – Character string ("some string")
A sequence of ASCII characters (including escaped characters). Recognized by quotes before and after the string value. Used as an identifier and as a value.

Valid predecessors	JSONToken_BrL, JSONToken_Comma, JSONToken_Colon
Valid successors	JSONToken_BrR, JSONToken_Comma, JSONToken_Colon
Value	Character string without the quotes.

JSONToken_JSON – JSON data ({"name1":value1,"name2":value2})
A JSON structure in a string. This token type only applies when a node in the cache is of the JSON type.

Valid predecessors	JSONToken_Colon
Valid successors	JSONToken_BrR, JSONToken_Comma
Value	Character string with the begin and end brackets.

JSONToken_Int – Integer (42)
Positive or negative number. If the parser is in array building mode, the integer is the next (or first) value in the array, otherwise it is a scalar.

Valid predecessors	JSONToken_SqL, JSONToken_Comma, JSONToken_Colon
Valid successors	JSONToken_BrR, JSONToken_SqR, JSONToken_Comma
Value	The numeric value of the JSON value.

JSONToken_Float – Float (3.141)
Positive or negative floating point number. If the parser is in array building mode, the float is the next (or first) value in the array, otherwise it is a scalar. An integer JSON value may be parsed as a float if the selected node is a float type.

Valid predecessors	JSONToken_SqL, JSONToken_Comma, JSONToken_Colon
Valid successors	JSONToken_BrR, JSONToken_SqR, JSONToken_Comma
Value	The floating point value of the JSON value.

JSONToken_True – Boolean true (true)

Valid predecessors	JSONToken_Colon
Valid successors	JSONToken_BrR, JSONToken_Comma
Value	TRUE (1)

JSONToken_False – Boolean False (false)

Valid predecessors	JSONToken_Colon
Valid successors	JSONToken_BrR, JSONToken_Comma
Value	FALSE (0)

JSONToken_Null – Null value (null)
The null value type is only set on branch nodes, and it may be used to set an empty tree on a JSON node.

Valid predecessors	JSONToken_Colon
Valid successors	JSONToken_BrR, JSONToken_Comma
Value	NULL