HISTORY AND REVISIONS

|  |  |  |
| --- | --- | --- |
| **Date** | **Author** | **Note** |
| September 6, 2019 | Gianluca Brunelli | **First release (draft)** |

Sommario

[2. Long term goal 3](#_Toc18684620)

[3. General software architecture 3](#_Toc18684621)

[4. SMU 5](#_Toc18684622)

[1. Key feature 5](#_Toc18684623)

[2. Architectural details 5](#_Toc18684624)

[3. Source code structure 7](#_Toc18684625)

[1. A note about the code of the beta v1 7](#_Toc18684626)

[2. Code for beta v2 7](#_Toc18684627)

[3. Library rheaCommonLib 7](#_Toc18684628)

[4. Library CPUBridge 9](#_Toc18684629)

[5. Library SocketBridge 9](#_Toc18684630)

[6. Application GPUServer 9](#_Toc18684631)

[5. Fusion Beta V1 GUI and Javascript interface 10](#_Toc18684632)

[4. How to 10](#_Toc18684633)

# Long term goal

The long term goal for this project is a complete revamp of the current software architecture in order to obtain a better and more maintainable using a different and more flexible approach so to prepare the VMC to face and win the challenges of the future.

The main idea is to move most of the functionality that today are handled by the CPU to what we today call the GPU. The GPU has a better processor, more RAM, more storage space, has on OS, better debugging tools so it’s the natural candidate for managing many of the functionality that, for historical reasons, are today implemented in the CPU.

The CPU will mostly deal with the actuators (i.e.: mixers, valve, variflex…) and will be the main interface for the MDB payment systems; the GPU will handle all the logic, will keep all the recipes for selections, the machine configuration, EVA-DTS information, counters, statistics and so on.

In order to achieve this results, we can’t simple discard all the works done so far and redo everything from scratch in one go; we still have to regularly release updates in terms of new functionality and bug fixing and still have to support all the machines that are already on the field. Having said that, the sw transformation will happens gradually, in steps, each one closer to the final goal.

# General software architecture

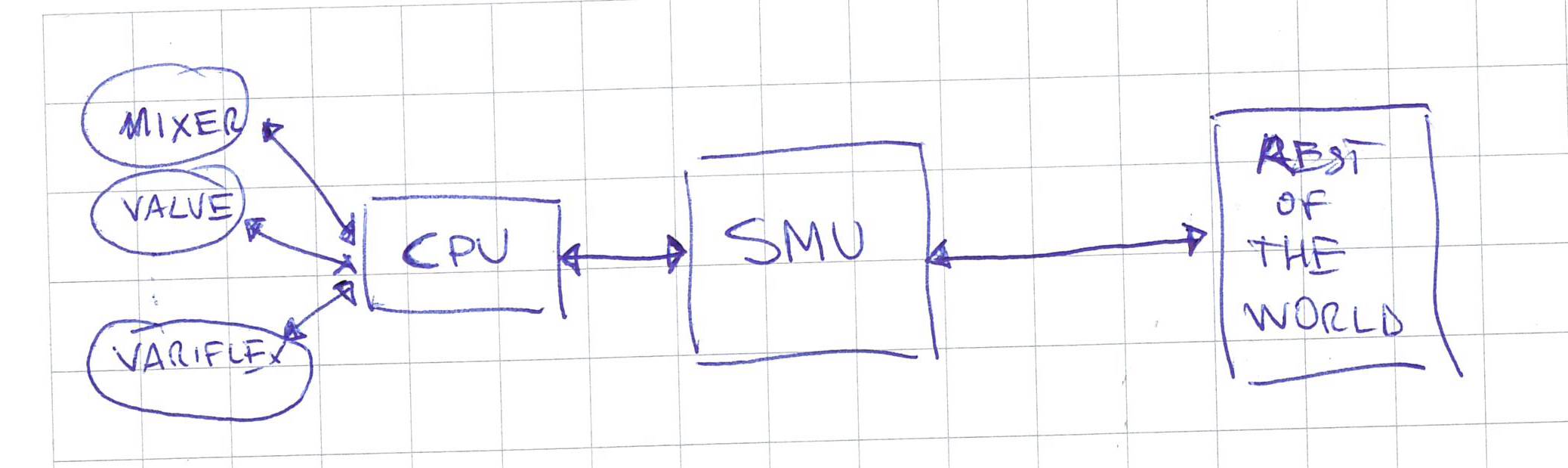


Figure 1

As seen in fig. 1, from a high level point of view, the new VMC will be composed of 2 big independent software, the CPU and the SMU.

The CPU is the one that will handle all the devices needed to actually deliver a recipe (mixer, valve, variflex…). Starting from the current CPU, the idea is to gradually remove features and at the same time introduce them in the SMU.

The SMU is the core of the logic of the VMC. The SMU is the only one that can talk to the CPU; the SMU will send orders to the CPU in order to:

* Deliver recipes
* Gather information relative to the devices linked to the CPU

The SMU does not care “how” the CPU achieve a specific tasks. This means that the SMU and CPU are completely independent from each other, they only share a “communication channel” and a “communication protocol”. By formatting messages according to the protocol and sending them thought the channel, SMU and CPU can communicate and work together ignoring how each other does what it does.

The SMU does not have any graphic or textual interface at all. It’s a process (or demon in Linux terminology) that starts silently and then stay resident, waiting for something to happens. The SMU establish a connection with the CPU and keep the connection alive by periodically sending messages and receiving answer that will report the status of the CPU and its attached devices.

On top of that, the SMU also expose one and only one interface to the rest of world; every application that wants to interact with the VMC, will have to do it by communicating through this interface.

The “rest of the world” (i.e.: GUI, new Rheamedia, new Rheaction (if any), website, GPU…) can connect to the SMU and, by using a specific protocol, issue commands and query.

As of today, the SMU open a TCP/IP socket on port 2280; every application that can reach this port and establish a connection, is able to issue commands and query to the VMC (this means that, already today, you could command the machine from a remote website, as soon as the VMC is open to the internet and has a reachable IP address).

This is exactly how the new GUI works. The GUI open a (web)socket on localhost, port 2280 and perform a handshake. If everything goes as expected, the GUI is now able to send messages through the socket so to query the VMC about its state (i.e.: selection availability, selection prices,) or issue command (i.e.: start-selection-number-7).

Once connected, the GUI can send request (as just mentioned) to which the SMU will answer, but the SMU also can send “events” on its own, without a specific request, to notify the connected applications that something has happened.

For example, as soon as the SMU detects a problem let’s say with mixer 1 (because the CPU told to the SMU that there’s a problem), the SMU will in turn send a “event” to everybody connected through the socket informing that “selection availability has changed” and reporting which selections are not available due to the mixer 1 not working.

If the user inserts some credits, then the SMU will send an event to every connected applications informing that “credit has changed” and reporting the actual available credit.

# SMU

## Key feature

* Platform independent code, written in c++. As of today, it’s already working on linux desktop, linux embedded and windows without changing a line of code, you just need to recompile with the appropriate machine dependent tool (ie: MSVC for windows, Qt/gcc for linux).
* High code modularity achieved by extensive use of library of source code that are statically linked if/when needed. The same libraries that are used to create the SMU, are also used for the GPU and for the upcoming new RheaMedia and will/can be used for every future application that will deal with the VMC.
* Open to external applications through communication via socket. The socket itself, can accept different protocols. As of today, you can communicate using 2 different protocols:
  + the RFC 6455 websocket protocol (<http://www.rfc-editor.org/info/rfc6455>)
  + the “console protocol” which is a rhea proprietary protocol under development which will be used mostly for debugging purpose

If in the future there should be the need to add further protocols, the code is already structured in a way that it’s going to be easy to perform a task like this. You just need to create a new class derived from the base rhea::IProtocol class and plug it into the source code. The SMU will use the new protocol without the need to change anything in the core functionality (see IProtocol.h in the attachments).

* No graphic or textual user interface. Only ad hoc applications can direct interact with the SMU. As of today, the Fusion beta 1 GPU/GUI is an example of such an application. Another application, called “Console” is in development. The console application will be used mostly for debugging purpose and will look like a classical terminal window, with a prompt and a list of textual command that you can use to query the SMU to check its state or to issue commands like “start selection” or “reset CPU”. The new RheaMedia will be another example of an application that deals with the SMU; it will let you configure the GUI and then test it on your (windows) PC in order to preview on screen the actual GUI by simulating a real scenario of usage.
* The SMU is capable of handling multiple connected applications at the same time. You can have for example one GPU (or more) and one or more console, all connected to the same SMU and all receiving “events” from the SMU itself, and all capable of issuing query or command in real time; these applications can run on the same PC as the SMU or on different PC, everywhere in the world. The only requisite is that the application can reach the TCP/IP port of the SMU (i.e.: the machine running the SMU and the machine running an application have to be somehow networked together).

## Architectural details

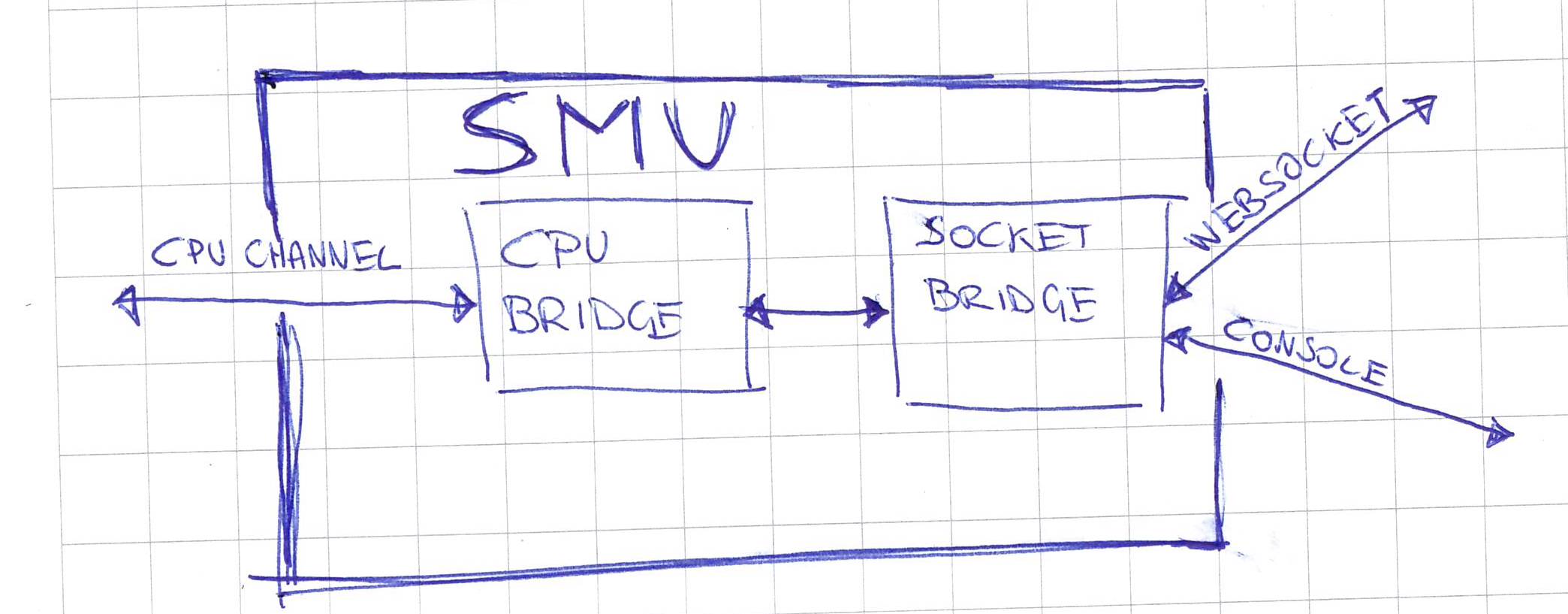


Figure 2

As you can see in figure 2, the SMU is composed of 2 threads called CPUBridge and SocketBridge.

CPUBridge is the core of the SMU. Its tasks are to talk to the CPU using a “CPUChannel” and to answer to query and commands coming from the SocketBridge.

Talking to the CPU means sending messages along the CPUChannel in order to stay informed about the state of the CPU and its attached devices. Should something important happens, then the CPUBridge will notify SocketBridge. For example, if the CPU is reporting some changes in the selections availability, then CPUBridge will send a “Selection availability change” notification to SocketBridge. If the CPU is reporting that the user has inserted some coins, then CPUBridge will send a notification to SocketBridge stating that “Credit has changed”.

Answering to SocketBridge means that SocketBridge is allowed to send query/commands to ask for specific configuration parameters, or to start a selection, or to ask about the availability of the selections; CPUBridge will answer to this requests reporting actual data gathered from the CPU or from some internal configuration file that only CPUBridge can access.

CPUChannel is an abstraction. It’s a piece of code that is used to send a command to the CPU and wait for an answer (see attached CPUChannel.h).

As of today, there are already 2 different CPUChannel implementation, one that deals with the serial port (i.e.: send and receive buffer or data through the serial port), and one used to simulate the presence of CPU even when the CPU is not physically there (this is used for example with the new RheaMedia. The SMU communicate using this special channel that in turn is simulating a fake CPU that will always report a specific good health state and that is able to fake the complete procedure of beverage delivering. This let the new RheaMedia be able to show you on your PC how the GUI will work during the beverage delivering even if you are not physically delivering anything and you are not even attached to a CPU).

The SocketBridge is a separate thread that communicate with CPUBridge by using a specific thread safe message queue. SocketBridge is the thread that opens the TCP/IP socket and handle incoming connection from applications from the rest of the world.

Applications send commands and query to the SocketBridge via socket using any of the allowed protocol (as of today, websocket or console protocol); SocketBridge handle all the requests abstracting from the specific protocol used by the application and eventually send them down to CPUBridge via thread message queue; CPUBridge handle all the request coming from SocketBridge and answer to SocketBridge again via thread message queue; SocketBridge receive an answer from CPUBridge and send it back to the right application encoding the answer using the right protocol needed by the specific application (i.e.: websocket protocol for browser, console protocol for console and so on).

Should the CPUBridge issue a notification on its own (for example when the user inserts some credits or the availability of the selections has changed), then CPUBridge send a notification to SocketBridge that, in turn, will send the same notification to every connected application, encoding the message with the appropriate protocol needed by every single application.

## Source code structure

As of today (2019-09-06), the source code has already evolved a lot WRT to the source code used when the Fusion Beta v1 was released. The following information are therefore referred to the current structure of the code that will lead to the release of Fusion Beta v2 and new Rheamedia, but that it’s still under development and subject to change.

The code is available through git here: https://gitlab.com/rheagroup/gpu-fts-nestle-2019.git

### A note about the code of the beta v1

Beta v1 has been a first step into moving from the previous software architecture to the one illustrated so far in this document. As we have to carefully move step by step, the beta v1 was still a hybrid approach in which the old GPU and the new SMU were fused together. Many new concepts were already in place (i.e.: the fact that the GUI is working by communication via websocket, the subdivision of the source code in small and well distinct library of source code, the platform independence of the core c++ code…) but not everything is as described in this document. The foundations have been laid in anticipation of the beta v2 in which the architecture reported in this document finally be 100% in place. If you want to take a look at the code from the beta v1, the GIT repository is here: <https://gitlab.com/giallanon/gpu-fts-nestle-betav1.git>

### Code for beta v2

As of today, the code consists of 3 projects that produce libraries, and one project that produce an executable.

A part from the standard c/c++ library that comes with the compiler, no external library or framework have been used, there’s no need to install any third-party software on the target machine.

### Library rheaCommonLib

This library consists of a set of utilities that will come handy for every projects now and in the future and it’s also the one responsible for the “platform independent” aspect of the project. Everything that is not already platform independent by its nature, it should be made that way by introducing a sort of abstraction layer/interface in this library.

It’s subdivided into 2 big parts, one that deals with the local OS in order to achieve the platform independence, and one the is composed of many classes and templates that can/will be used a basic programming tools.

#### The platform independence

Everything under the OS folder is something that deals with OS specific implementation. The OS folder, in turn, has 2 subfolder called linux and win.

The first problem to solve it’s about data size of basic data types like int or long. Depending on the compiler, on the host OS and on the processor architecture, you can never be sure for example of how big is a “int”. Is it 2bytes? Is it 4 bytes? Is it 8 bytes? The same goes for “short”, “word”, “long”, “long long” and so on.

So, to fix this problem and to give uniformity to the whole source code, the first thing that rheaCommonLib does it to provide you with some typedef that are platform/compiler independent and that will univocally identify any of the basic c/c++ data types. Here is the file (see rheaDataTypes.h):

//typedef dei dati di base

typedef int8\_t i8; //8 bit signed

typedef uint8\_t u8; //8 bit unsigned

typedef int16\_t i16; //16 bit signed

typedef uint16\_t u16; //16 bit unsigned

typedef int32\_t i32; //....

typedef uint32\_t u32;

typedef int64\_t i64;

typedef uint64\_t u64;

typedef float f32; //32 bit floating point

typedef uintptr\_t uiPtr; //un "intero" la cui dimensione in byte dipende dalla piattaforma, ma che è sempre in grado di ospitare un puntatore

#define u16MAX 0xFFFF

#define u32MAX 0xFFFFFFFF

#define u64MAX 0xFFFFFFFFFFFFFFFF

What this header does, it to typedef various datatypes using names that are short, easy to understand and remember:

* u8 is an unsigned 8 bit integer
* u16 is an unsigned 16 bit integer
* u32 is an unsigned 32 bit integer
* u64 is an unsigned 64 bit integer
* i8 is a signed 8 bit integer
* i16 is a signed 16 bit integer
* i32 is a signed 32 bit integer
* i64 is a signed 64 bit integer
* f32 is a 32 bit floating point

Do you need a byte? Use u8 or i8.

Do you need a “word”? Use u16 or i16.

You’re guaranteed that u8 will always be 8 bit long and the same goes for every other types defined here.

This is the very base of the code portability, every function in this and other libraries is using this data types, this is mandatory to achieve platform independence.

Having solved the data types problem, there are still some function that we need and that are not natively platform independent:

* thread creation and handling
* thread communication mechanism (semaphore, events)
* socket creation and handling
* serial port creation and handling
* high precision timer
* current date and time query

rheaCommonLib already provide all these in a platform independent way (see OS/OS.h).

The rest of the commonLib function will automatically call this specific OS implementation in order achieve an abstraction layer that let you, for example, create a thread with a syntax that will not change based on the compiler or the host OS.

In general, you should not have to directly call the function that starts with OS\_ prefix, since this are already used by more high level function and classes found in the second part of the rheaCommonLib.

#### Tools and general purpose class/template

Everything under the main folder and under the rhea namespace is single class/template that implements some sort of utility that you can use in the developing of your code.

Let’s take some examples:

* rheaFIFO.h => it’s a template for a FIFO of a generic datatype. Need a FIFO of int? use FIFO<int>, need a FIFO for your struct? Use FIFO<myStruct>
* rheaLIFO.h => same as FIFO but in a LIFO way
* rheaRandom.h => a class to generate random number given a seed. It does not depend on specific compiler implementations, so you can be sure that given the same seed, you’ll have the same output on different compiler and or architecture
* rheaString.h => many functions and sub-namespace that operate on c string (i.e.: const char \*s). There are function for parsing strings, converting string into int, float, hex, date timeformat and so on
* rheaThread.h => function that deals with the creation and handling of thread. Also provided thread safe message queues that threads can use to communicate between each other
* rheaAllocator.h => basic interface to implement a memory allocator. As of today, only one allocator is available and it’s based on the \_aligned\_malloc\_ provided by compiler/OS. In the future, in order to improve memory allocation and fragmentation, several specific allocator could be implemented (for example it could be useful to have an allocator that is not thread safe and that you will use only inside a specific thread. This way, the allocator could be a little faster since it does not have to deal with thread safety stuff and you, as the programmer, will guarantee to use this specific instance of allocator only on a specific thread). Another allocator that will come handy is a “scrap allocator”, used for fast and small memory allocation generally needed as temporary data-buffer while processing data. The scrap allocator could use a small pre-allocated memory allocated at the start and deal with so it won’t fragment the main memory handled by malloc. For more information, look on the internet with keywords like “memory arena”, “memory pools”, “memory allocator”.
* rheaBit.h => several function that takes a generic void \*buffer and treat it as a sequence of bit letting you, for example, set/get then-th bit of a sequence.
* ProtocolSocketServer.h => a class that open a socket and listen for incoming connection, maintains a list of connected clients and deals with many different protocols. Clients that wants to connect to this server can use any of the IProtocol already implemented in the library (see ProtocolWebsocket.h and ProtocolConsole.h)

### Library CPUBridge

Depends on: rheaCommonLib

This library implements the CPUBridge as seen in section SMU.

CPUBridge it’s a thread that exposes a write-only message queue that other thread can use to “subscribe” to the CPUBridge itself. Once a thread is subscribed, it will receive an ad-hoc message queue handle that CPUBridge will use to notify the thread of anything that happens and that the thread can use to ask something to the CPUBridge.

Any communication between CPUBridge and the thread will happens using this thread safe message queue (using rhea::thread::pushMsg() and rhea::thread::popMsg() provided by rheaCommonLib).

CPUBridge also use a CPUChannel (provided by the main application) which is a class that implements a communication channel between the thread and the CPU. As of today, CPUBridge communicate with the physical CPU using an instance of CPUChannelCom which is an implementation of CPUChannel capable of sending and receiving data via serial com port. If in the future we need to communicate with the CPU using something different from the serial port, we only have to implement a new class derived from CPUChannel and pass it to CPUBridge. CPUBridge and its logic won’t be changed in any way.

### Library SocketBridge

Depends on: rheaCommonLib, CPUBridge

This library implements the SocketBridge thread as seen in section SMU.

It also implements many CmdHandler classes that have to deal with all the available command and query that applications can send through the socket.

For example, let’s say an application is asking for “selection availability”. When SocketBridge receive the request, it will instantiate a new class of type CmdHandler\_eventReqSelAvailability to handle the command.

If an application is asking to “start a selection”, SocketBridge will automatically instantiate a class of type CmdHandler\_eventReqStartSel to handle the command.

The instancing of specific class related to specific command is automatized using the program paradigm known as “factory” which means that anytime you add a new command, you don’t have to go through a gigantic switch and add a new “case”. The instancing is somewhat automatic and you do not have to touch the SocketBridge source code at all.

### Application GPUServer

Depends on: rheaCommonLib, CPUBridge, SocketBridge

This is the SMU.

At startup it will create a the CPUBridge thread. When the thread is ready, it will create a SocketBridge thread that, in turn, will register itself with the CPUBrdige thread. After this, the main thread will sleep until the CPUBrdige thread dies.

The code itself consists of only one cpp file (main.cpp) which does nothing but call already existing function inherited from the libraries.

This is a perfect example of how applications will be created. They will heavily rely on the libraries and their code will almost only consist of specific code that deals with the specific functionality of the application itself.

# Fusion Beta V1 GUI and Javascript interface

The GUI is written in HTML5 and pure javascript.

Regarding HTML5, we have some limitations on what we can use since the browser we use on the target machine is a bit old and not up to date with the current HTML5 standards.

Many of the common framework/library (like jquery and bootstrap) are either not working properly or too heavy to be used by our machine.

Some feature like video playback and gif animation are either not working or working poorly, depending on the size and the weight of the video/gif itself.

So, in theory we can use anything HTML5 has to offer, but in practically we have to carefully choose and test everything before saying that we can use it.

Note that this is NOT a software or architectural problem, it only depends on the computation power provided by the target machine and on the browser version installed/supported by the target machine.

If, for example, we run the SMU + GUI on a modern laptop, we could have video in 4K and animated gif everywhere.

The GUI itself does not have many requirements, you can do almost whatever you want in terms of graphic presentation. When/if you need some runtime information, like the availability of selections, or the credits inserted, or the selections prices, or the CPU status, you can ask for them in a “ajax like” way by using the method provided by the rhea javascript library.

If the SMU needs to inform you that something has happened, you will receive an event that you can trap and handle in your javascript code.

Everything is already in place and working; whoever will develop a new GUI, he won’t have to deal with “how to send” and “how to receive” information, the mechanism and the function are already in place.

## How to

When creating a new html page, the only thing you need to do is include a script called rheaBootstrap.js and call the function rheaBootstrap() on the onLoad event of the body element. The script will automatically handle the websocket connection and instantiate a global variable called rhea which is the main object that the html page will use in order to communicate with the SMU.

Example:

<html>

<head>

<title>GUI</title>

<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">

<meta http-equiv="X-UA-Compatible" content="IE=edge"/>

<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">

</head>

<body onLoad="**rheaBootstrap**()">

…

…

…

<script src="js/rheaBootstrap.js"></script>

<script language="javascript">

function **onRheaBootstrapFinished**()

{

rhea.onEvent\_**selectionAvailabilityUpdated** = function () {

onSelectionAvailabilitUpdated();

}

rhea.**onEvent\_cpuMessage** = function(msg, importanceLevel)

{

var dCPU = rheaGetElemByID ("divCPUMessage");

var dNav = rheaGetElemByID ("divNavigator");

if (importanceLevel == 0)

{

rheaHideElem(dCPU);

rheaSetDisplayMode(dNav, "inline-block");

}

else

{

rheaSetElemHTML(dCPU, msg);

rheaShowElem(dCPU);

rheaHideElem(dNav);

}

}

rhea.**onEvent\_selectionPricesUpdated** = function ()

{

}

rhea.**onEvent\_creditUpdated** = function ()

{

}

**As you can see, the <body> tags has the onLoad event set to call the function rheaBoostrap() which is inside the js/rheaBootstraps.js script.**

**As soon as the rheaBootstrap() has finished, it will call the function onRheaBootstrapFinished() that you have to provide.**

**Once in this function, everything is already being setup for you, and you can use the object “rhea” to issue command or query.**

**As said in previous chapter, the SMU can (and will) sometimes send notifications when something meaningful happens.**

**Your code can (it does not “have to”, but it can, which means that you can choose to ignore any notification) handle this notification implementing the rhea.onEvent\_xxx method.**

**In the code above, you can see that we have the onEvent\_selectionAvailabilityUpdate() and many other implemented.**

**This functions will be automatically called as soon as the SMU send the specific notification.**

**For a list of all the events that can be trapped, see rheaEvent.js.**

**On the other side, if you need to ask something to the SMU, you can do it by using one of the following:**

rhea.requestGPUEvent (RHEA\_EVENT\_SELECTION\_AVAILABILITY\_UPDATED);

rhea.requestGPUEvent (RHEA\_EVENT\_SELECTION\_PRICES\_UPDATED);

rhea.requestGPUEvent (RHEA\_EVENT\_CREDIT\_UPDATED);

rhea.requestGPUEvent (RHEA\_EVENT\_CPU\_MESSAGE);

rhea.requestGPUEvent (RHEA\_EVENT\_SELECTION\_REQ\_STATUS);

**Each of this command will ask the SMU to send back an event that will trigger the handler seen before.**

**For example, if you call**

rhea.requestGPUEvent (RHEA\_EVENT\_SELECTION\_AVAILABILITY\_UPDATED);

**then the function you’ve associated with the rhea.onEvent\_selectionAvailabilityUpdated() will be called.**

**If you need to deal with selections, you’ve the following function that will return immediately since selections information are already available as soon as you enter in the onReahBootraspeFinished() function (information are stored in session variable and automatically updated as needed):**

**rhea.selection\_start (iSelNumber)**

**rhea.selection\_stop()**

**rhea.selection\_getCount()**

**rhea.selection\_getBySelNumber(iSelNumber)**

**If you have to deal with the Main Menu Icon (MMI), you’ve the following (that works like the selection\_xxx):**

**rhea.MMI\_getCount()**

**MMI\_getDisplayName(iIcon)**

**MMI\_getPrice(iIcon)**

**MMI\_getImgForPageMenu(iIcon)**

**MMI\_getImgForPageConfirm(iIcon)**

**MMI\_getLinkedSelectionNumber(iIcon)**

**MMI\_canUseSmallCup(iIcon)**

**MMI\_canUseMediumCup(iIcon)**

**MMI\_canUseLargeCup(iIcon)**

**MMI\_hasDoubleShot(iIcon)**

**MMI\_hasDblGrinder(iIcon)**

**and many more.**

**The js library also provided you with some common utilities and functionality to deal with html elements.**

**Just like library like jquery does, this library try to abstract from the pure js syntax when you have to deal with html elements because of compatibility issue that can arise moving from one browser to another.**

**In the file js/rheaUtils.js you can find many useful function:**

* **rheaLoadScript (url) => it returns a promise so you do not have to wait while the script is loading, you will be notified as soon as the function finished**
* **rheaGetElemByID(), rheaDoesElemExistsByID(), rheaSetDivHTMLByName(), rheaGetElemWidth(), rheaGetElemHeight(), rheaSetElemTextColor(), rheaHideElem(), rheaShowElem() and many others that deals with html elements**

**The library also provided a way for you to memorize session variable in a browser independent way. Session variable are useful while moving from page to page since you can memorize a value in a session variable on page 1, and retrieve the same value on page 2.**

**The API for session variable are the following:**

* **rhea.Session\_clearValue (itemName)**
* **rhea.Session\_getValue (itemName)**
* **rhea.Session\_getOrDefault (itemName, defaultValue)**
* **rhea.Session\_setValue (itemName, itemValue)**
* **rhea.Session\_clearObject (itemName, plainJSObject)**
* **rhea.Session\_setObject (itemName, plainJSObject)**
* **rhea.Session\_getObject (itemName)**