Sommario

[1. Quick info 2](#_Toc65764954)

[1.1 GIT 2](#_Toc65764955)

[1.2 Elenco messaggi CPU-GPU 2](#_Toc65764956)

[1.3 Struttura delle cartelle del progetto 2](#_Toc65764957)

[2 Istruzioni di compilazione 5](#_Toc65764958)

[2.1 Su Windows 5](#_Toc65764959)

[2.2 Su Linux 5](#_Toc65764960)

[3 Architettura SW 7](#_Toc65764961)

[3.1 Definizione dei termini 7](#_Toc65764962)

[3.2 Panoramica ad alto livello 8](#_Toc65764963)

[3.3 Panoramica dell’architettura SMU 9](#_Toc65764964)

[4 Menu di programmazione 11](#_Toc65764965)

[4.1 Panoramica 11](#_Toc65764966)

[4.2 Flusso di lavoro e supporto multilingua 11](#_Toc65764967)

[4.3 Architettura del file index\_template.html 11](#_Toc65764968)

[4.4 Comunicazione con SMU 12](#_Toc65764969)

[4.4.1 rhea.ajax 12](#_Toc65764970)

[4.4.2 rhea.requestGPUEvent 12](#_Toc65764971)

[5 Tutorial: aggiunta di una nuova funzionalità 14](#_Toc65764972)

[5.1 Scenario 14](#_Toc65764973)

[5.2 Primo passo: concordare i comandi con CPU 14](#_Toc65764974)

[5.3 Secondo passo: implementazione dei comandi/funzioni in CPUBridge 14](#_Toc65764975)

[5.3.1 Gestione delle funzioni “ask” in CPUBridge 16](#_Toc65764976)

[5.4 Terzo passo: implementazione comandi in SocketBridge 17](#_Toc65764977)

[5.5 Quarto passo: (opzionale) implementazione risposta ai comandi nel simulatore di CPU 17](#_Toc65764978)

[5.6 Quinto passo: (opzionale) creazione di una APP (GUI) in che sfrutti i nuovi comandi 17](#_Toc65764979)

# Quick info

## GIT

|  |  |
| --- | --- |
| Nome progetto su git Rhea | laRhea >GPU > gpu-fts-nestle-2019 |
| Clone con http | <https://rpsapp04.rheavendors.com/larhea/gpu/gpu-fts-nestle-2019.git> |
| Clone con SSH | [git@rpsapp04.rheavendors.com:larhea/gpu/gpu-fts-nestle-2019.git](mailto:git@rpsapp04.rheavendors.com:larhea/gpu/gpu-fts-nestle-2019.git) |
| Branch principale | ~~rasPISerial~~  ~~Nota: il branch “master” è stato il branch primario di sviluppo fino a luglio 2020 dopodichè è stato abbandonato in favore del branch “rasPISerial”~~  Aggiornamento 19 feb 2021  I rami “master” e “rasPISerial” sono stati merged per cui da oggi “master” è di nuovo il ramo principale di sviluppo |

## Elenco messaggi CPU-GPU

Vedi: L:\ENGINEERING\L-File Condivisi\ProtocolloComunicazioneFusion2\Nuovi\_comandi\_CPU\_GPU.xlsx

Questo documento Excel (mantenuto per ora da Gianluca Brunelli e Massimo Silvestri) contiene l’elenco dei messaggi CPU-GPU attualmente supportati.

## Struttura delle cartelle del progetto

Scaricando il progetto da git, la struttura delle cartelle è la seguente:

\

--- *current (vedi note)*

--- DOC

--- MSVC17

--- MenuProgTranslator

--- QT5

--- bin

--- src

Tralasciando di elencare tutte le sottocartelle esistenti, ecco un elenco dei path più significativi:

|  |  |
| --- | --- |
| /current | Questa cartella, sulla macchina del caffè, contiene una serie di sottocartelle nelle quali sono memorizzati i vari file di configurazione che sono stati caricati. Per file di configurazione si intende: il master cpu (file.mhx), il master GPU (file .mh6), il file .da3, la GUI, i file di linguaggio. Ogni volta che un utente carica uno di questi file (tramite l’interfaccia GPU), la GPU li salva nella relativa cartella in modo che sia possibile successivamente recuperare l’esatta configurazione della macchina, comprensiva di master CPU. La GUI attualmente visualizzata in macchina, per esempio, risiede nella cartella /current/gui.  Data la dimensione di file contenuti in questa cartella, ho ritenuto opportuno non aggiungerli direttamente al repository git. In compenso, nella cartella /bin esiste un file zip di nome current.zip che, una volta scompattato, crea la cartella current e tutte le relative sottocartelle popolandole con i contenuti necessari affinché GPU/SMU possano essere eseguiti senza errori anche sul PC.  Si consiglia quindi di scompattare current.zip direttamente dentro /bin in modo da ottenere l’alberatura completa del progetto |
| /DOC | Questa cartella non esiste sulla macchina del caffè;  sul PC locale invece, contiene una serie di documenti che possono tornare utili per la comprensione e/o sviluppo del codice. È la cartella che uso per collezionare documentazione varia. Non è pensata per essere un qualcosa di omnicomprensivo ma, piuttosto, è una sorta di repository con documentazione varia che potrebbe tornare utile anche in futuro |
| /bin | Qui viene generato l’exe della SMU/GPU dopo la compilazione e, tramite lo script shell Linux “aaa\_MH6\_CREATE\_ESAPI.sh” viene creato automaticamente il pacchetto “mh6” da caricare in macchina |
| /bin/RheaMedia20 | Questa cartella non esiste sulla macchina del caffè.  E’ la home di RheaMedia 2.0.  Qui viene generato l’exe di rheaMedia2.0 e qui esistono tutte le sottocartelle necessarie al suo funzionamento.  RheaMedia2.0 sostanzialmente esiste tutto sotto questa cartella. Copiando l’intera cartella sul desktop o su un qualunque altro percorso, ci si porta dietro l’intero rheaMedia2.0 standalone |
| /bin/varie/prog | Qui c’è il nuovo menù di programmazione. Tutto quello che lo riguarda è in questa cartella e nelle sue sottocartelle |
| /MSVC17 | Questa cartella non esiste sulla macchina del caffè.  Qui sono contenuti i file di progetto per Visual Studio. Attenzione che sotto questa cartella NON sono contenuti i file sorgenti in C (vedi /src), ma solo i file di progetto di MSVC. Il file progetto/soluzione da caricare in MSVC è /MSVC17/FusionProject.sln |
| /QT5 | Questa cartella non esiste sulla macchina del caffè.  Come sopra ma vale per il compilatore QT da usarsi sotto linux. Il file di progetto da caricare in QT è /QT5/GPUProject.pro |
| /src | Questa cartella non esiste sulla macchina del caffè.  Ogni sottocartella di /src contiene i file c/h della relativa libreria. Ad esempio, /src/rheaCommonLib contiene tutti i file sorgenti necessari a creare e compilare la libreria rheaCommonLib. |
| /lib | In questa cartella, durante una build, i compilatori mettono automaticamente le librerie C compilate a partire dai vari progetti |

# Istruzioni di compilazione

## Su Windows

|  |  |
| --- | --- |
| Ambiente di sviluppo | Microsoft Visual Studio 2017 (o successivi) |
| File della soluzione | /MSVC17/FusionProject.sln |
| Progetto principale per la generazione di SMU.exe | SMU |
| Progetto principale per la generazione di rheaMedia2.0 | rheaMedia2.0 |
| Progetto principale per la generazione dell’exe da copiare sul raspberry | rasPISerial |

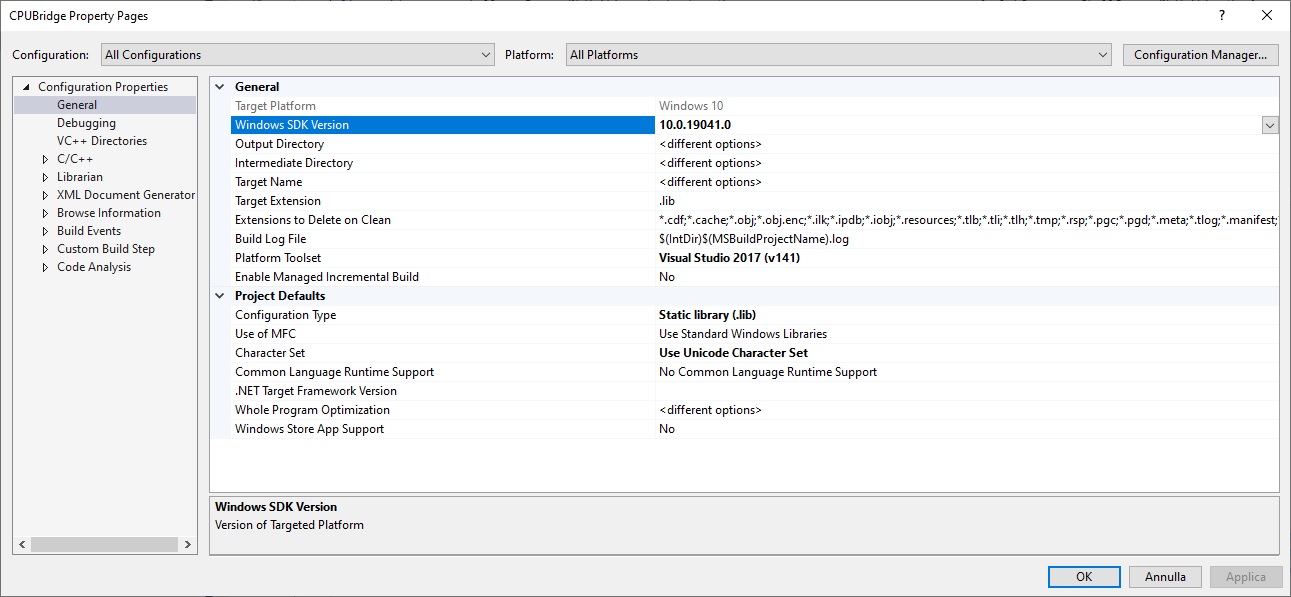
|  |
| --- |
| NB: a partire da 2021-04-02, è disponibile anche un progetto per l’ambiente MSVC 2019 Community Edition. Il file della soluzione è in /MSVC19/FusionProject.sln. Anche per MSVC19, valgono le stesse considerazioni qui sotto riportate |

Su Windows non c’è nulla di particolare da fare per compilare il progetto, è sufficiente aprire la soluzione con MSVC e poi cliccare col destro sul progetto che si vuole compilare (es: SMU) e selezionare “Set as startup project”.

A questo punto, facendo il build, il compilatore fa tutto da sé e alla fine genera un file .exe che mette nella cartella /bin. Parlando ad esempio del progetto SMU, il compilatore genera il file /bin/WINDebug\_SMU.exe.

Dal nome del file possiamo capire che la build è stata fatta su WINdows, ed è in versione Debug. Se compilassimo la versione release, il file generato sarebbe WINRelease\_SMU.exe

**NOTA**: A seconda della versione di MS Visual Studio utilizzata, probabilmente sarà necessario effettuare alcuni aggiustamenti alle proprietà dei progetti, in particolare per quanto riguarda le voci “**Windows SDK Version**” e “**Platform Toolset**”. A tale proposito, cliccare col destro sul progetto “rheaCommonLib”, poi selezionare “Proprietà” ed infine, “General Settings”. Dovrebbe comparire una schermata simile alla seguente:



Selezionare in alto a sinistra “All Configurations” e in alto a destra “All Platforms”. Alle voci “Windows SDK Version” e “Platform Toolset” selezionare quanto disponibile nel proprio compilatore.

**È necessario ripetere questa operazione per tutti i progetti della soluzione.**

Se fosse necessario, utilizzare “Visual Studio Installer” per aggiornare il proprio compilatore e scaricare il Windows SDK versione 10, eventualmente disponibile al download anche presso il sito Microsoft al seguente indirizzo: <https://developer.microsoft.com/en-US/windows/downloads/windows-10-sdk/>

## Su Linux

Al momento esiste una VM con una distribuzione Linux Ubuntu e tutte le librerie/tools necessari a compilare ed eseguire il codice. La VM purtroppo è molto grossa (160 GB) perché così l’ho ereditata. Sono sicuro che si potrebbe costruire una VM molto più contenuta in termini di spazio su HD, ma non ho mai avuto il tempo di dedicarmi a questa attività.

|  |  |
| --- | --- |
| User VM | Davide |
| Pwd VM | zenarhea |
| Cartella del progetto | ~/Desktop/gpu-fts-nestle-2019 |
| Ambiente di sviluppo | Qt Creator Community |
| File della soluzione | ~/Desktop/gpu-fts-nestle-2019/QT5/GPUProject.pro |

Su linux esistono 4 possibili build:

* Desktop-debug
* Desktop-release
* Embedded-debug
* Embedded-release

All’interno di QTCreator è possibile selezionare quale delle 4 build si desidera compilare cliccando sull’icona in basso a sinistra e selezionando la build desiderata.

Le versioni “Desktop” vengono compilate per essere eseguite direttamente nella VM la quale supporta una architettura processore x86. Il compilatore selezionato quindi è gcc e l’eseguibile generato funziona solo su linux desktop. Facendo il build della versione “desktop-debug”, viene creato il file eseguibile /bin/DESKTOP64\_DEBUG\_GPU. Facendo il build della versione Desktop release, il file generato è /bin/DESKTOP64\_RELEASE\_GPU.

Per ottenere un file eseguibile che possa girare sulle macchine del caffè, è necessario utilizzare un compilatore apposito che sia in grado di compilare per l’architettura IMX6 (che è il processore che abbiamo sulle schede GPU delle macchine TS). Tale compilatore è il gcc-imx6 ed è già installato e funzionante sulla VM. Selezionando la build “embedded-release” da dentro QTCreator, la compilazione del progetto produce il file /bin/EMBEDDED\_RELEASE\_GPU.

Questo file NON è eseguibile all’interno della VM in quanto è compilato per funzionare solo su processori IMX6. Una volta ottenuto il file /bin/EMBEDDED\_RELEASE\_GPU, è necessario eseguire lo script (doppio click va bene) /bin/aaa\_MH6\_CREATE\_ESAPI.sh

Questo script prende EMBEDDED\_RELEASE\_GPU più una serie di altri file (tra I quali per esempio buona parte della cartella del menu di programmazione) e crea un “pacchetto GPU” che è esattamente il file da mettere sulla chiavetta USB per poi caricarlo sulla macchina del caffè.

Se tutto va a buon file, lo script genera un file (sempre nella cartella bin) con un nome di questo tipo:

GPU\_TS\_v.2.4.5\_210128\_commit.mh6

dove:

* GPU\_TS\_ è un prefisso fisso che identifica il fatto che il pacco contiene una GPU per macchine Touchscreen
* v.2.4.5 è la versione della GPU. Quando si rende necessario cambiare versione, bisogna indicare quella nuova sia nel file header.h (vedi *#define GPU\_VERSION*) sia nello script aaa\_MH6\_CREATE\_ESAPI.sh , editabile con un normale text editor. Clicca col destro sullo script e seleziona “open with Other Application”, poi seleziona text editor. La riga 4 dello script è una cosa di questo tipo: *filename="GPU\_TS\_v.2.4.5\_$DATA$estensione"*. È sufficiente cambiare 2.4.5 in qualunque cosa si desideri in modo che lo script possa produrre un file il cui nome sia in linea con la versione compilata della GPU
* 210128 è la data di compilazione e viene inserita automaticamente dallo script. La stessa data di compilazione è riportata automaticamente anche nella schermata BOOT della GPU grazie all’uso delle macro \_\_DATE\_\_ e \_\_TIME\_\_ (standard c)
* commit è un placeholder da sostituirsi con l’effettivo commit quando il tutto viene pushato su git

**Nota bene**: per poter compilare la versione EMBEDDED, bisogna avviare QTCreator da riga di comando in quanto è necessario istruirlo affinché conosca il path del compilatore per IMX6. Per fare ciò, aprire un terminale e digitare /opt/Qt/Tools/QtCreator/bin/./qtcreator.sh

Dato che la cosa non è molto mnemonica, sul desktop c’è un file dal nome “z linux commands.txt”. Aprendolo con un text editor (ie: facendoci doppio click), è possibile trovare la seguente indicazione:

*Per lanciare QTCREATOR/opt/Qt/Tools/QtCreator/bin/./qtcreator.sh*

Copiando/incollando la seconda riga del file direttamente su un terminale è possibile avviare QTCreator senza doversi ricordare a memoria la stringa di lancio.

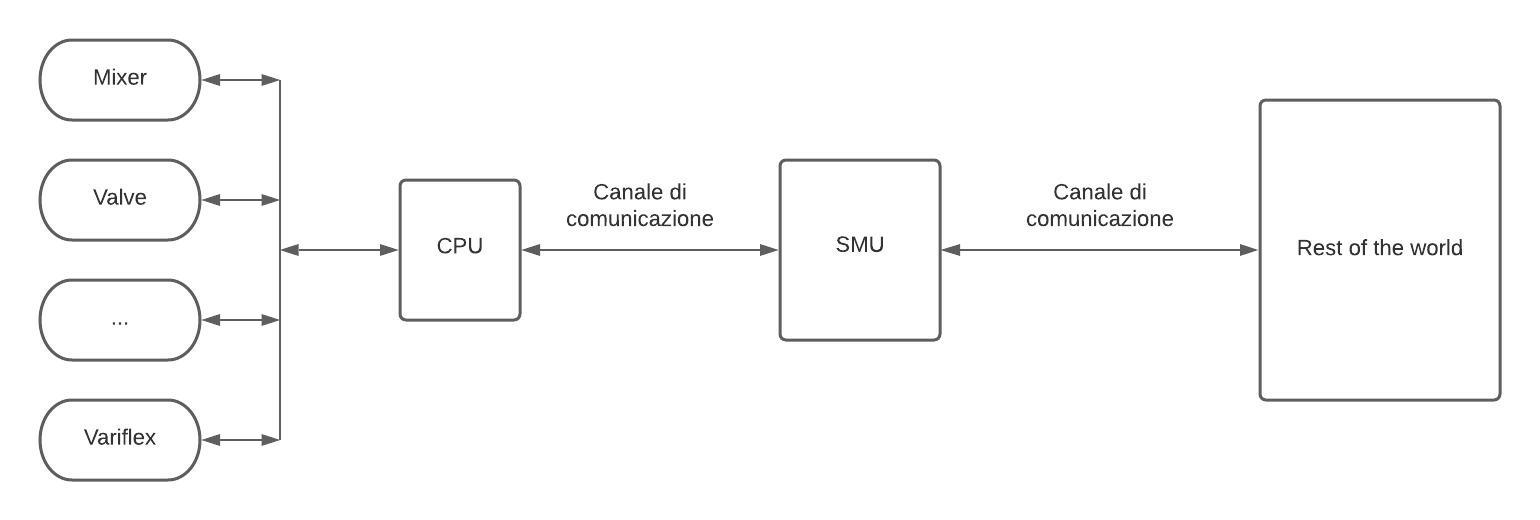
Al contrario, per poter compilare le versioni Desktop, è necessario NON avviare QTCreator da riga di comando. In questo caso è sufficiente cliccare sull’icona di Qt posta nella barra laterale sinistra.

# Architettura SW

## Definizione dei termini

|  |  |
| --- | --- |
| CPU | La CPU come la conosciamo |
| Canale di comunicazione | Un canale fisico (es: RS232) oppure logico (es: socket) sul quale è possibile fare passare un flusso di dati (bytes) |
| SMU | Smart Management Unit  E’ il processo/applicazione che ha il compito di mantenere la comunicazione con la CPU e contemporaneamente di fornire una interfaccia verso il mondo esterno tale da fornire una serie di servizi e funzionalità atti al pilotaggio della macchine del caffè |
| App esterna | Qualunque applicazione, in qualunque linguaggio d programmazione che giri o sulla macchina del caffè stessa oppure su una qualunque macchina esterna, che sia in grado di accedere al canale di comunicazione fornito dalla SMU verso il resto del mondo. Tramite una serie di comandi da inviare lungo il canale di comunicazione, la app è in grado di pilotare la macchina del caffè |
| GUI | Una “App esterna” specializzata nel proporre agli utenti il menu dei prodotti erogabili dalla macchina. Tipicamente (ma non necessariamente) realizzata in HTML e con un occhio di riguardo alla veste grafica |
| GPU | Il pacchetto SW comprendente SMU, menu di programmazione, schermata di BOOT (per il caricamento dei vari tipi di file in macchina) e schermata MAIN per la visualizzazione di una GUI in HTML da caricarsi mediante meno di BOOT |
| RheaMedia2.0 | Programma per la creazione e modifica delle GUI accettate da GPU nella schermata di BOOT |

## Panoramica ad alto livello



* CPU gestisce gli attuatori in maniera autonoma ed indipendente
* SMU e CPU condividono un CANALE DI COMUNICAZIONE sul quale è implementato un protocollo MASTER/SLAVE nel quale SMU è MASTER e CPU è SLAVE
* SMU è l’unica entità alla quale è consentito dialogare con CPU
* SMU fornisce un secondo CANALE DI COMUNICAZIONE verso il mondo esterno. Qualunque applicazione in grado di collegarsi a questo CANALE DI COMUNICAZIONE può, attraverso un protocollo di comunicazione ben definito, dialogare con SMU e richiedere determinate azioni. Ad esempio, è possibile chiedere l’esecuzione di una specifica selezione, è possibile chiedere lo stato della macchina, è possibili sapere quante selezioni sono attualmente abilitate, quale è il credito, quale è il messaggio di testo della CPU…

**Nota 1**: Il CANALE DI COMUNICAZIONE CPU-SMU è rappresentato nel codice da una classe base con metodi virtual (vedi CPUChannel.cpp nel progetto CPUBridge). Ad oggi esistono ad esempio 2 classi concrete che implementato la classe base:

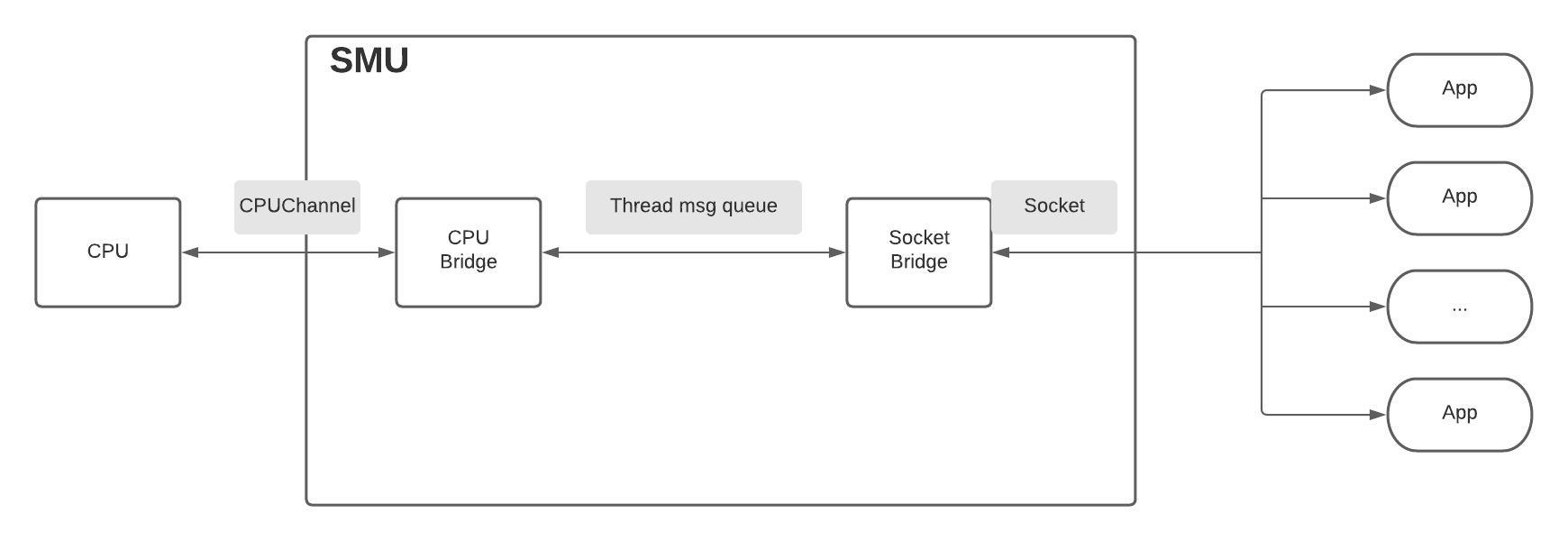
* CPUChannelCom che sfrutta una interfaccia seriale (RS232 /UART) per inviare e ricevere dati alla CPU
* CPUChannelFakeCPU che non invia i dati in nessun luogo ma semplicemente elabora le richieste proveniente da SMU e risponde di conseguenza. E’ una sorta di simulatore SW di CPU

Nella remota ipotesi che in futuro si volesse cambiare il canale fisico di comunicazione (ad esempio sostituendo la seriale con un canale Wi-Fi o infrarossi), l’unica cosa da implementare nel codice sarebbe una classe CPUChannelXXX in grado di gestire il canale fisico. Tutte le logiche di funzionamento della SMU rimarrebbero inalterate.

**Nota 2**: sul CANALE DI COMUNICAZIONE SMU-Resto del mondo, è possibile implementare più di un protocollo di trasporto. Ad oggi ne esistono 2 in effetti, uno è il protocollo Websocket implementato secondo le specifiche RFC6455 (vedi /DOC/rfc6455.txt), l’altro è un protocollo proprietario di nome RheaConsole.

Al di là del tipo di protocollo di trasporto utilizzato, il payload rappresenta invece sempre una sequenza di byte che, a sua volta, si mappano su una serie di comandi riconosciuti da SMU. Questi comandi sono esposti al resto del mondo e sono indipendenti da quelli usati tra CPU e SMU. Ad esempio, se un domani volessimo modificare il comando #B che CPU e SMU si scambiano di continuo, questa modifica NON si rifletterebbe in un cambiamento del protocollo SMU-resto del mondo. Questo significa che il “resto del mondo” può continuare a comunicare come ha sempre fatto anche se, sotto il cofano, i rapporti tra SMU e CPU potrebbero essere stati modificati.

## Panoramica dell’architettura SMU



* SMU e CPU comunicano attraverso un CPUChannel che, a livello di codice c, rappresenta un generico canale di comunicazione in grado di inviare un buffer di bytes alla CPU ed attendere una risposta da quest’ultima (vedi CPUChannel.h nel progetto CPUBridge).
* SMU e il resto del mondo comunicano attraverso una SOCKET TCP/IP aperta sulla porta 2280. Questa SOCKET accetta il protocollo WEBSOCKET ed il protocollo RHEAConsole
* Il canale SOCKET è in grado di gestire connessioni da multiple APP esterne
* Il canale SOCKET è full duplex e **NON** segue i principi dell’architettura master/slave
* Generalmente SMU riceve richieste attraverso SOCKET da qualunque APP sia collegata e risponde di conseguenza (solo all’app che ha fatto la richiesta). SMU può però anche generare delle NOTIFICHE SPONTANEE che vengono propagate a tutte le APP connesse (es: cambio di stato, cambio di messaggio testuale CPU, cambio di disponibilità delle selezioni)
* Internamente SMU è suddivisa in 2 blocchi logici principali denominati **CPUBridge** (vedi CPUBridge.h nel prj CPUBridge) e **SocketBridge** (vedi SocketBridge.h nel prj SocketBridge). CPUBridge è un thread di SMU e vive di vita propria in totale indipendenza da SocketBridge il quale, a sua volta, è un altro thread di SMU, completamente slegato da CPUBridge. I 2 thread comunicano tra di loro attraverso una message queue (thread safe). Non esistono variabili in comune tra i 2 thread, sono totalmente isolati l’uno dall’altro. Possono scambiarsi informazioni solo attraverso il canale di comunicazione rappresentato dalla message q.
* CPUBridge si occupa di dialogare con CPU e lo fa attraverso il CPUChannel. CPUBridge è l’unica entità alla quale è concesso dialogare con CPU. Nessun altro può farlo. Tipicamente, CPUBridge invia il comando **#B** alla CPU 4/5 volte al secondo per rimanere informato sullo stato di funzionamento della CPU. Le informazioni che riceve dalla CPU vengono interpretate e, se necessario, notificate a SocketBridge tramite la message Q (es: cambio di messaggio testuale da parte di CPU, cambio di disponibilità delle selezioni)
* SocketBridge si occupa di dialogare con il resto del mondo. Tipicamente apre il canale SOCKET e poi rimane in attesa di qualche connessione lungo quel canale. Nel momento in cui una o più APP si connettono tramite SOCKET, SocketBridge inizia ad “ascoltarle” in attesa di eventuali richieste e, parallelamente, inizia ad informarle di specifici ed importanti EVENTI (spesso chiamati NOTIFICHE SPONTANEE). Dall’altro lato, SocketBridge è in ascolto anche sulla message Q in attesa di eventuali notifiche da parte di CPUBridge.

# Menu di programmazione

## Panoramica

Il “nuovo menu di programmazione” (da ora in poi solo “menu di programmazione”), è una app esterna realizzato in HTML5 + javascript che parla con SMU attraverso il canale aperto da SocketBridge.

**ATTENZIONE**: *considerando che tale app deve girare in locale anche sulla macchina del caffè, è necessario prestare attenzione nell’uso di funzioni JS troppo moderne. Il browser che abbiamo a disposizione sulla macchina del caffè risale al 2010 circa, per cui non è al passo con il javascript del 2020.*

Il codice sorgente del menu di programmazione si trova nella cartella \bin\varie\prog e relative sottocartelle. Il file principale è index\_template.html il quale si appoggia a svariati script js (vedi \bin\varie\prog\js) organizzati per funzionalità.

L’app interagisce con SMU tramite l’utilizzo di API definite nei file js che si trovano in \bin\varie\prog\js\dev.

## Flusso di lavoro e supporto multilingua

In generale, le modifiche vanno apportate direttamente al file index\_template.html e, eventualemente, al file js\Task.js.

Al fine di supportare il multilingua, tutti i testi inseriti in index\_template.html sono/devono essere dei semplici placeholder, ovvero degli identificativi che vengono poi automaticamente sostituiti con il testo tradotto in lingua. A tale proposito, il file Excel **MenuProgTranslator\ translations.xlsx** contiene le traduzioni in tutte le lingue supportate.

L’eseguibile (Windows) **MenuProgTranslator\MenuProgTranslator.exe** è invece lo strumento da utilizzare per tradurre index\_template.html e generare automaticamente in vari index\_GB.html, index\_IT.html e via dicendo. Un analogo discorso vale per il file **js\Task.js** dal quale vengono automaticamente generati i file in lingua (js\Task\_GB.js, js\Task\_IT.js, …).

Tutti questi file in versione “tradotta” fanno parte del pacchetto software (generato automaticamente dallo script Linux aaa\_MH6\_CREATE\_ESAPI.sh, vedi 2.2) che viene importato direttamente in macchina tramite il solito meccanismo della chiavetta USB.

## Architettura del file index\_template.html

Questo è il cuore del menu di programmazione. Il file è suddiviso in molteplici sezioni separate logicamente con dei tag <div>. Ogni sezione presenta un commento in testa in modo da poter facilmente navigare all’interno del codice.

Il file html si avvale di un certo numero di script js tramite i quali è in grado di comunicare con la SMU utilizzando una serie di funzioni che fanno capo all’oggetto globale denominato rhea. Il flusso di esecuzione prevede obbligatoriamente il caricamento del file js/rheaBootstrap.js. Questo file implementa il meccanismo di connessione alla SMU. Una volta che la connessione è avvenuta, viene chiamata la funzione utente onRheaBootstrapFinished() la quale di fatto restituisce il controllo a index\_xx.html.

Un esempio minimale di pagina di programmazione è la seguente:

<html>

<head>

<title>**PROGRAMMING MENU**</title>

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

<link href=**"style.css"** rel=**"stylesheet"** type=**"text/css"**>

</head>

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

</body>

<script id=**"rheaBootstrapTag"** src=**"js/rheaBootstrap.js"** data-ses-clear=**"1"**></script>

<script language=**"javascript"**>

**function** onRheaBootstrapFinished**()**

**{**

**}**

</script>

Nel tag <body> si specifica di chiamare la funzione rheaBootstrap() al termine del caricamento del file html. Tale funzione, che è contenuta nello script js/rheaBootstrap.js, si occupa di connettersi alla SMU e, al termine della connessione, a sua volta chiama onRheaBootstraFinished() che deve essere definita all’interno di index\_xx.html. A partire da questo momento, la connessione è stabilita e index\_xx.html può accedere alla variabile globale rhea e a tutti i suoi metodi.

## Comunicazione con SMU

### rhea.ajax

Il metodo principale utilizzato per comunicare con la SMU è il metodo rhea.ajax:

rhea.**ajax** ("setDecounter", { "d":iDecounter, "v":decValue}).then( function(**result**)

{

//chiedo i decontatori alla SMU

pageMaintenance\_getDecountersValues();

})

.catch( function(result)

{

pleaseWait\_hide();

});

Nell’esempio soprastante, index\_xx.html invia a SMU il comando “setDecounter” indicando i parametri “d” e “v”. La funzione “rhea.ajax” è una chiamata asincrona, non bloccante, che può terminare solo in 2 modi: SMU ha risposto, oppure SMU non ha risposto entro il timeout predefinito (4 sec).

Nel primo caso, la variabile **result** conterrà il risultato (ovvero la risposta) che SMU ha prodotto. Nel secondo caso (gestito all’interno del metodo catch), result conterrà l’eventuale messaggio di errore prodotto.

### rhea.requestGPUEvent

Un secondo metodo utilizzato per comunicare con la SMU, è il metodo requestGPUEvent:

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

Questa funzione, a differenza di rhea.ajax, non prevede una risposta diretta da parte di SMU. A differenza del caso ajax infatti, il codice javascript non prevede handler per la gestione della risposta o dell’eventuale timeout.

Generalmente, alla ricezione di una richiesta di “evento”, SMU reagisce generando una “notifica spontanea” ovvero una di quelle segnalazione che, di tanto in tanto, la SMU è solita inviare per notificare importanti cambiamenti di stato.

Gli eventi spontanei ai quali index\_xx.html è in grado di reagire, sono definiti in js\dev\ rheaEvent.js e sono i seguenti:

Rhea**.prototype.**onEvent\_selectionAvailabilityUpdated **=** **function()** **{}**

Rhea**.prototype.**onEvent\_selectionPricesUpdated **=** **function()** **{}**

Rhea**.prototype.**onEvent\_creditUpdated **=** **function()** **{}**

Rhea**.prototype.**onEvent\_cpuMessage **=** **function(**msg**,** importanceLevel**)** **{}**

Rhea**.prototype.**onEvent\_selectionReqStatus **=** **function(status)** **{}**

Rhea**.prototype.**onEvent\_cpuStatus **=** **function(**statusID**,** statusStr**,** flag16**)** **{}**

Rhea**.prototype.**onEvent\_readDataAudit **=** **function(status,** kbSoFar**,** fileID**)** **{}**

All’interno di index\_xx.html è possibile (opzionalmente) intercettarli e gestirli tramite js. Ad esempio, per intercettare e gestire l’evento spontaneo “cpuMessage”:

**function** onRheaBootstrapFinished**()**

**{**

pleaseWait\_setup**();**

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

**{**

currentTask.onEvent\_cpuMessage**(**msg**,** importanceLevel**);**

**}**

E’ importante notare che questi “eventi spontanei” possono essere generati da SMU anche in assenza di una specifica requestGPUEvent(). In realtà, requestGPUEvent() è un metodo per forzare la SMU a generare l’evento desiderato in modo da poterlo gestire nel codice come se questo evento fosse arrivato spontaneamente da SMU.

Ad esempio, il menu di programmazione gestisce l’evento spontaneo “cpuMessage” che SMU genera ogni volta che la CPU cambia il suo messaggio di stato (per esempio passa da “Place your cup” a “Preparing selection”). SMU però genera questi eventi solo quando il messaggio di CPU cambia. Fintanto che il messaggio rimane (ad esempio) “Place your cup”, nessun evento spontaneo viene mai generato. Al caricamento di index\_xx.html quindi, per conoscere l’attuale messaggio di CPU, viene utilizzato rhea.requestGPUEvent (RHEA\_EVENT\_CPU\_MESSAGE) in modo da obbligare SMU a rispondere con un evento “cpuMEssage” così da permettere a index\_xx.html di ricevere l’evento e aggiornare a video la stringa attualmente visualizzata da CPU.

# Tutorial 01: aggiunta di una nuova funzionalità

## Scenario

Supponiamo di dover gestire un nuovo dispositivo, un distributore di bicchieri “on demand”. Il distributore è collegato alla CPU la quale è in grado di pilotarlo senza problemi.

Vogliamo esporre una nuova funzionalità al mondo esterno. Vogliamo quindi creare uno o più comandi che permettano alle APP di pilotare il distributore di bicchieri. In particolare, vogliamo creare un comando che permetta alle APP di avviare l’erogazione di un bicchiere, ed un secondo comando che permetta alle APP di controllare lo stato dell’operazione, in modo da sapere quando il bicchiere è effettivamente stato espulso e se l’espulsione è avvenuta correttamente o no.

|  |
| --- |
| **NB: il codice con l’implementazione di quanto discusso in questo tutorial si trova su git nel ramo “tutorial” al commit 98d012e.**  **Fare riferimento a quel codice (ampiamente commentato nelle funzioni in discussione) per una dettagliata descrizione di “come “ e “cosa” accade di preciso all’interno delle varie funzioni trattate nel tutorial.** |

## Primo passo: concordare i comandi con CPU

Concordare con CPU i comandi necessari e aggiungerli al file **Nuovi\_comandi\_CPU\_GPU.xlsx** in modo da mantenere il documento aggiornato e sempre allineato con i messaggi implementati nel codice.

|  |  |  |
| --- | --- | --- |
| **COMANDO** | **SINTASSI GPU**  **SINTASSI CPU** | **NOTE** |
| Start erogazione bicchiere | # P [len] 0x24 [ck]  # P [len] 0x24 [result] [ck] | Chiede a CPU di iniziare l’operazione di erogazione di un bicchiere.  [result] = 0x00 se CPU non è in grado di iniziare l’operazione  0x01 se CPU ha iniziato l’operazione |
|  |  |  |
| Query stato erogazione bicchiere | **# P [len] 0x25 [ck]**  **# P [len] 0x25 [stato] [ck]** | GPU chiede a CPU lo stato dell’erogazione bicchiere iniziata con il comando P 0x23  [stato] = 0x00 waiting  0x01 finished OK  0x02 finished KO |

## Secondo passo: implementazione dei comandi/funzioni in CPUBridge

Il progetto interessato in questo passo è la libreria CPUBridge, in particolare i file CPUBridge.h e CPUBridge.cpp.

Nel file CPUBridge.h sono riportati tutti i prototipi delle funzioni che è necessario implementare affiche CPUBridge sia in grado di ricevere una richiesta da SocketBridge, inviare la stessa a CPU e, infine, rispondere alla richiesta di SocketBridge sfruttando le informazioni riportate da CPU.

Il file CPUBridge.h è diviso in 3 macro sezioni:

* buildMsg\_xxxx
* notify\_xxxx
* ask\_xxxx

La sezione “buildMsg\_xxx” raccoglie tutte le funzioni necessarie a creare un comando ben formattato (rispettoso della sintassi del protocollo CPU – GPU) da inviare a CPU.

Nel caso in esempio quindi, aggiungiamo la seguente funzione sulla falsa riga di tutte le altre funzioni della stessa famiglia già presenti nel codice (leggi: fai copia incolla di funzioni analoghe già esistenti e parti da lì per creare quella nuova):

u8 buildMsg\_startErogazioneBicchiere (u8 \*out\_buffer, u8 sizeOfOutBuffer);

Questa funzione, così come tutte quelle della famiglia “build\_xxx” prende in input un [buffer] di lunghezza [sizeOfOutBuffer] byte e lo riempie con i byte che è necessario inviare alla CPU per comunicare un certo comando. La funzione ritorna il numero di byte inseriti in [out\_buffer].

Nel nostro caso, la funzione riempirà il buffer con i seguenti 5 byte:

|  |  |
| --- | --- |
| **BYTE IN DECIMALE** | **SIGNIFICATO** |
| 35 | ASCII per ‘# |
| 80 | ASCII per ‘P’ |
| 5 | lunghezza totale del messaggio in byte |
| 36 | 0x24 |
| 156 | [ck] ovvero checksum, ottenuta come somma di tutti i byte del messaggio tranne ck stessa (35 +80 +5 +36) |

L’implementazione di questa funzione è nel file CPUBridge.cpp.

La seconda funzione da implementare, fa parte della famiglia di funzioni “ask\_xxx”. Questo tipo di funzioni sono utilizzate da SocketBridge per comunicare con CPUBridge. Le funzioni “ask\_xxx” quindi inviano un messaggio lungo la “message queue” esistente tra CPUBridge e SocketBridge

void ask\_CPU\_START\_EROGAZIONE\_BICCHIERE (const sSubscriber &from, u16 handlerID);

Il parametro [from], comune a tutte le funzioni “ask\_xxx” indica il mittente del messaggio.

Il parametro [handlerID], anch’esso comune a tutte le “ask\_xxx” è un 16 bit che il mittente può valorizzare a suo piacimento e che si ritroverà invariato nella successiva risposta a tale messaggio (vedi notify\_xxx).

L’implementazione di questa funzione è nel file CPUBridge.cpp.

Il messaggio che SockeBridge invia a CPUBridge tramite le funzioni ask, è formato da un ID univoco ed un buffer dati opzionale che può contenere, ad esempio, un elenco di parametri. Nell’implementazione della nuova funziona sk\_xx quindi, è mandatorio creare un nuovo ed univoco ID che identifica il messaggio (a tale proposito, vedi il codice commentato).

Quando CPUBridge riceve un messaggio inviato tramite una funzione “ask\_xxx”, risponde a sua volta chiamando una funzione di tipo “notify\_xxx” che non fa altro che inviare indietro un messaggio anch’esso composto da un ID univoco ed un eventuale buffer di parametri. Nel nostro caso specifico quindi, andiamo a definire la seguente funzione (sempre in CPUBridge.h):

void notify\_CPU\_START\_EROGAZIONE\_BICCHIERE (const sSubscriber &to, u16 handlerID, rhea::ISimpleLogger \*logger, bool yesOrNo);

Questa funzione invia una risposta da CPUBridge a SocketBridge sempre sfruttano la “message queue” esistente tra I 2 thread. In questo caso, la risposta prevede anche l’invio di un parametro [yesOrNo] che farà capire a SocketBridge se la sua richiesta di “start erogazione bicchiere” è andata a buon fine. L’implementazione di questa funzione è nel file CPUBridge.cpp. Se andiamo a vedere cosa fa effettivamente questa funzione, possiamo notare che crea un messaggio composto da un identificatore univoco (CPUBRIDGE\_NOTIFY\_START\_EROGAZIONE\_BICCHIERE) e un buffer di byte. All’interno di questo buffer sono impacchettati I parametri del messaggio. Nel caso specifico, abbiamo un solo parametro (yesOrNo) che decidiamo di comunicare usando 1 byte che valga 0 in caso di false e 1 in caso di true.

Quando SocketBridge riceve una notifica di questo tipo, è generalmente interessata anche a conoscerne i parametri (nel caso specifico, è interessata alla bool yesOrNo). Invece che lasciare a SocketBridge il compito di decodificare il buffer del messaggio e andare ad estrarre i singoli parametri in esso memorizzate, è meglio creare una apposita funzione di “traduzione” la quale si occupi esattamente di svolgere questo compito. Sempre nel file CPUBridge.h quindi, subito sotto alla dichiarazione della funzione notify\_CPU\_START\_EROGAZIONE\_BICCHIERE andiamo a definire anche la relativa funzione di traduzione:

void translateNotify\_CPU\_START\_EROGAZIONE\_BICCHIERE (const rhea::thread::sMsg &msg, bool \*out\_yesOrNo);

Questa funzione, prende in input l’intero messaggio ricevuto da SocketBridge (msg) e ritorna nel parametro [\*out\_yesOrNo] il bool che CPUBridge ha comunicato quando ha usato la funzione notify\_CPU\_START\_EROGAZIONE\_BICCHIERE

|  |
| --- |
| *Breve riepilogo di quanto implementato fino ad ora:*   * SocketBridge usa la funzione **ask\_CPU\_START\_EROGAZIONE\_BICCHIERE** per chiedere a CPUBridge di erogare un bicchiere * CPUBridge riceve la richiesta e, a sua volta, invia a CPU la richiesta di iniziare l’erogazione del bicchiere sfruttando il messaggio # P 0x24 che è stato incapsulato dalla funzione **buildMsg\_startErogazioneBicchiere** * CPU riceve il messaggio P 0x24 e risponde (oppure no) indicando nel byte [result] 0x00 oppure 0x01 a seconda che sia in grado o meno di iniziare l’erogazione del bicchiere * CPUBridge riceve la risposta da CPU, la interpreta, legge il byte [result] e a sua volta notifica SocketBridge sull’esito dell’operazione usando la funzione **notify\_CPU\_START\_EROGAZIONE\_BICCHIERE** indicando nel parametro [yesOrNo] il valore true se CPU ha risposto con result==0x01, false altrimenti (anche in caso di eventuale non risposta da parte di CPU) * SocketBridge riceve la notifica (ovvero la risposta alla sua “ask” fatta al punto 1 di questo elenco), estrae il parametro yesOrNo usando la funzione **translateNotify\_CPU\_START\_EROGAZIONE\_BICCHIERE** e poi agisce di conseguenza (vedi paragrafi successivi) |

### Gestione delle funzioni “ask” in CPUBridge

Quando SocketBridge usa una funzione di tipo “ask\_xxx”, CPUBridge riceve una notifica e deve processare la richiesta producendo a sua volta una risposta verso SocketBridge. La gestione di questo scenario è nella funzione priv\_handleMsgFromSingleSubscriber nel file CPUBridgeServer.cpp

Nel caso particolare di questo esempio, possiamo vedere che la funzione **ask\_CPU\_START\_EROGAZIONE\_BICCHIERE** utilizza la define CPUBRIDGE\_SUBSCRIBER\_ASK\_CPU\_START\_EROGAZIONE\_BICCHIERE per identificare la richiesta.

All’interno della funzione handleMsgFromSingleSubscriber nel file CPUBridgeServer.cpp quindi, possiamo andare a gestire un “case” apposito che scatta solamente quando CPUBridge riceve questa specifica “ask”

case CPUBRIDGE\_SUBSCRIBER\_ASK\_CPU\_START\_EROGAZIONE\_BICCHIERE:

|  |
| --- |
| *Fare riferimento al codice sorgente di questo tutorial per ulteriori informazioni. La gestione di questo case in particolare è ampiamente commentata nel codice tutorial allo scopo di far comprendere passo passo cosa accade all’interno di CPUBridge quando viene ricevuta una richiesta di questo tipo.* |

Quello che deve accadere all’interno di questo “case” è che CPUBridge deve inviare a sua volta una richiesta a CPU, attenderne la risposta e, infine, notificare SocketBridge del risultato, il tutto utilizzando le funzioni “build\_xxx”, “ask\_xxxx” e “notify\_xxxx” che abbiamo appena implementato.

## Terzo passo: implementazione comandi in SocketBridge

A questo punto, il grosso del lavoro è già stato fatto. SocketBridge è in grado di chiedere a CPUBridge di avviare l’erogazione del bicchiere usando la funzione “ask\_CPU\_START\_EROGAZIONE\_BICCHIERE”; CPUBridge dal canto suo, alla ricezione della richiesta, è in grado di chiedere a CPU l’avvio dell’erogazione del bicchiere inviando un messaggio ben formattato lungo il CPUChannel. Alla ricezione della risposta da parte di CPU (o anche in caso di timeout), CPUBridge è in grado inoltre di rispondere a SocketBridge usando la funzione notify\_CPU\_START\_EROGAZIONE\_BICCHIERE.

Quello che rimane da implementare è un comando “ajax like” che possa essere invocato dalle app tramite javascript e che, in sostanza, comunichi a SocketBridge di avviare l’iter di comunicazioni interne di cui abbiamo parlato fino ad ora.

Per implementare un nuovo comando ajax like, bisogna lavorare sul progetto SocketBridge. Con riferimento alla struttura di cartelle del progetto MSVC, l’area corretta per inserire il nuovo comando all’interno del progetto è nella cartella di soluzione CmdHandler\ajax.

Qui dentro sono raccolti tutti i messaggi ajax like che le app possono utilizzare per comunicare con SocketBridge.

Come già suggerito nei paragrafi precedente, il modo migliore di agire è quello di identificare un comando simile a quello che vogliamo implementare e fare copia incolla dei relativi file h e cpp. Nel nostro caso, vogliamo implementare un comando ajax like (che chiameremo “startErogBicchiere”) che non prevede alcun parametro (infatti chiediamo solo di iniziare l’erogazione di un bicchiere, nessun parametro particolare è previsto) e dal quale ci aspettiamo una risposta semplice, magari in formato stringa, del tipo “OK” oppure “KO”.

Un comando simile esiste già ed è implementato nella classe CmdHandler\_ajaxReq\_T\_VMCDataFileTimestamp.

Andiamo nella cartella /src/ SocketBridge/CmdHandler, copia incolliamo i file CmdHandler\_eventReq\_T\_VMCDataFileTimestamp.h e CmdHandler\_eventReq\_T\_VMCDataFileTimestamp.cpp rinominandoli con il nome del nuovo comando, ovvero CmdHandler\_ajaxReqStartErogazioneBicchiere.cpp e CmdHandler\_ajaxReqStartErogazioneBicchiere.h.

All’interno del .h, andiamo a modificare il nome della classe, il nome del comando e i commenti descrittivi della funzionalità del comando stesso. La nuova classe quindi si chiamerà CmdHandler\_ajaxReqStartErogazioneBicchiere, il nuovo comando sarà “startErogBicchiere”.

Per implementare un comando ajax like, è necessario modificare 3 sole funzioni:

* getCommandName()
* passDownRequestToCPUBridge( … )
* onCPUBridgeNotification ( …)

La funzione getCommandName() è inline, direttamente nel file .h e deve ritornare la stringa che identifica univocamente il comando. Nel nostro caso quindi, si tratta di una banale return “startErogBicchiere”.

La funzione passDownRequestToCPUBridge() viene automaticamente invocata da SocketBridge ogni volta che riceve il comando “startErogBicchiere” da una delle app connesse. Dentro questa funzione ci limitiamo semplicemente ad usare la funzione ask\_xx che abbiamo implementato in precedenza in modo da avviare l’iter di comunicazioni interne che porterà la nostra richiesta fino al livello di CPU e indietro di nuovo fino a SocketBridge.

Infine, la funzione onCPUBridgeNotification() è quella che SocketBridge invoca automaticamente alla ricezione della notify da parte di CPUBridge. Questa funzione è quindi quella che si occupa di produrre la risposta da inviare indietro alla app richiedente. Tra i vari parametri ricevuti dalla funzione, c’è anche l’interno messaggio trasmesso da CPUBridge tramite la notify. La funzione notify che abbiamo implementato prevedeva il trasporto di un parametro booleano che indicasse la risposta di CPU alla richiesta di erogazione bicchiere. A questo punto, all’interno di onCPUBridgeNotification() non rimane altro che estrarre il booleano dal messaggio ricevuto e rispondere con la stringa “OK” in caso che il booleano sia true, oppure “KO” in caso sia false.

*Per ulteriori dettagli sull’implementazione vedere il codice tutorial.*

Una volta creata la nuova classe descrittiva del comando, rimane solo da aggiungere una riga di codice in modo da far sapere a SocketBridge dell’esistenza di questo nuovo comando. A tale proposito, sempre nel progetto SocketBridge, aprire il file CmdHandler\_ajaxReq.cpp e aggiungere in fondo al file la seguente:

CHECK(CmdHandler\_ajaxReqStartErogazioneBicchiere);

Questa macro non fa altro che aggiungere il nuovo comando all’elenco dei comandi noti a SocketBridge.

*NB: è necessario aggiungere anche una*

*#include "CmdHandler/CmdHandler\_ajaxReqStartErogazioneBicchiere.h"*

*in cima al file file CmdHandler\_ajaxReq.cpp*

## Quarto passo: (opzionale) implementazione risposta ai comandi nel simulatore di CPU

A questo punto, compilando la SMU e collegandola ad una vera CPU, tutto il giro dovrebbe funzionare senza problemi.

Nel caso in cui non disponessimo di una CPU vera e volessimo testare il funzionamento del nuovo comando usando il simulatore di CPU, allora è necessario implementare nella classe CPUChannelFakeCPU la sezione di codice preposta alla risposta al nuovo comando P 0x24.

A tale proposito quindi, nel progetto CPUBridge, aprire il file CPUChannelFakeCPU.cpp e cercare la funzione sendAndWaitAnswer.

All’interno di questa funzione sono presenti tutte le risposte che il nostro simulatore di CPU è in grado di produrre. Per i dettagli implementativi, cercare il

case eCPUProgrammingCommand\_startErogazioneBicchiere

all’interno di questa funzione.

Lo scopo è quello di ritornare una risposta che sia coerente con quello che una CPU vera produrrebbe. Nel nostro caso, alla richiesta P 0x24 dobbiamo rispondere (vedi 5.2) con un messaggio di questo tipo:

# P [len] 0x24 [result] [ck]

dove [result] == 0x00 se non possiamo iniziare l’erogazione di un bicchiere, 0x01 se possiamo avviare l’erogazione.

Per rendere le cose più interessanti, il codice di esempio produce una risposta 0x01 nel 70% dei casi, 0x00 nel restante 30% dei casi. In questo modo, possiamo testare il funzionamento sia nel caso “OK” che nel caso “KO”.

## Quinto passo: (opzionale) creazione di una APP (GUI) in che sfrutti i nuovi comandi

Per testare in locale la nuova funzionalità, possiamo creare una piccola app che non faccia altro che inviare il nuovo comando “startErogBicchiere” alla pressione di un bottone.

*Per i dettagli implementativi, vedi la mini GUI (il cui codice è ampiamente commentato) realizzata allo scopo e presente nella cartella tutorial2021/GUI-tutorial01.*

Per eseguire le GUI, è necessario intanto avviare la SMU. Una volta che questa è pronta e funzionante, basta fare doppio click sul file startup.html della GUI in modo da aprire un browser e vedere a video la nostra APP di test.

La GUI è derivata dalla “GUI minimale” di esempio (vedi cartella tutorial2021/GUI-minimale), per cui presenta gli stessi 3 bottoni della GUI minimale (SELEZIONE 2, SELEZIONE 13, GET CPU MODEL AND VERSION) ed in più un quarto bottone “START EROGAZIONE BICCHIERE” che non fa altro che chiamare il nuovo comando “startErogBicchiere” tramite la funzione rhea.ajax