Sommario

[1. Quick info 2](#_Toc64907311)

[1.1 GIT 2](#_Toc64907312)

[1.2 Elenco messaggi CPU-GPU 2](#_Toc64907313)

[1.3 Struttura delle cartelle del progetto 2](#_Toc64907314)

[2 Istruzioni di compilazione 4](#_Toc64907315)

[2.1 Su windows 4](#_Toc64907316)

[2.2 Su Linux 4](#_Toc64907317)

[3 Architettura SW 6](#_Toc64907318)

[3.1 Definizione dei termini 6](#_Toc64907319)

[3.2 Panoramica ad alto livello 7](#_Toc64907320)

[3.3 Panoramica dell’architettura SMU 8](#_Toc64907321)

[4 Menu di programmazione 10](#_Toc64907322)

[4.1 Panoramica 10](#_Toc64907323)

[4.2 Flusso di lavoro e supporto multilingua 10](#_Toc64907324)

[4.3 Architettura del file index\_template.html 10](#_Toc64907325)

[4.4 Comunicazione con SMU 11](#_Toc64907326)

[4.4.1 rhea.ajax 11](#_Toc64907327)

[4.4.2 rhea.requestGPUEvent 11](#_Toc64907328)

# 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 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:

\

--- DOC

--- MSVC17

--- MenuProgTranslator

--- QT5

--- bin

--- src

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

|  |  |
| --- | --- |
| /DOC | Contiene una serie di documenti che possono tornare utili per la comprensione e/o sviluppo del codice. E’ la cartella che uso per collezionare documentazione varia. Fino a ieri non era nemmeno parte del progetto, esisteva solo sul mio HD per cui non è pensata per essere un qualcosa di omnicomprensivo, piuttosto è una sorta di repository con documenti che potrebbero ritornare utili anche in futuro |
| /bin | Qui viene generato l’exe della SMU/GPU dopo la compilazione e, tramite lo script shell unix “aaa\_MH6\_CREATE\_ESAPI.sh” si crea automaticamente il pacchetto “mh6” da caricare in macchina |
| /bin/RheaMedia20 | 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 sottocartella |
| /MSVC17 | 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 | 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 | 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 |

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

## 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.

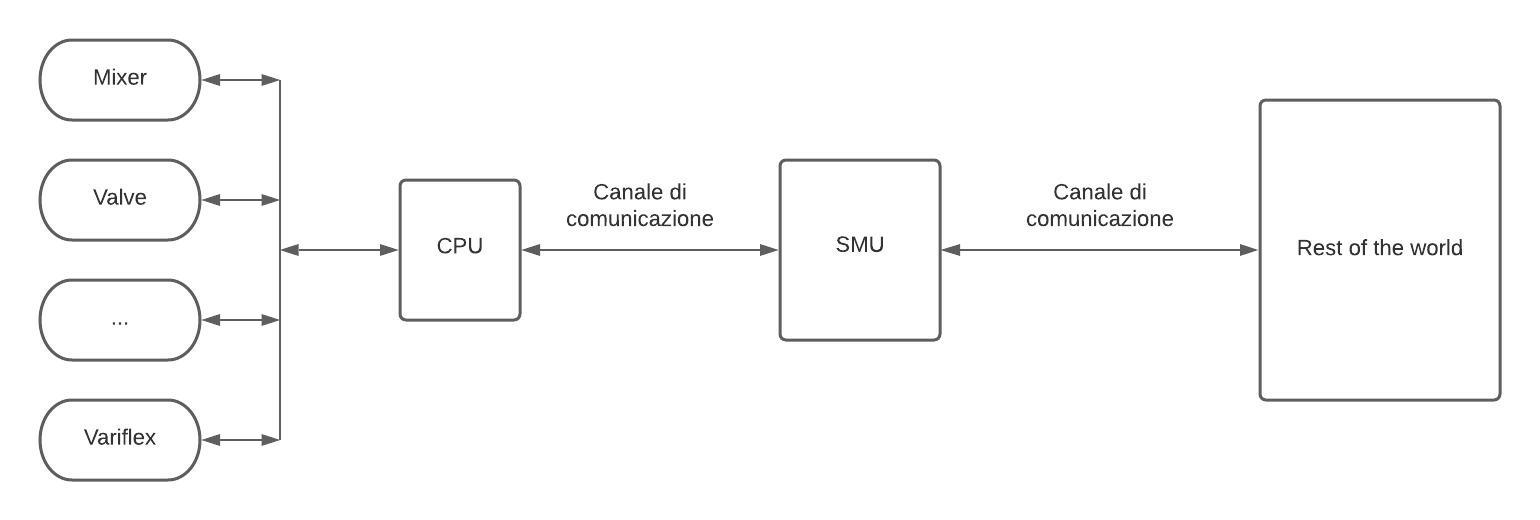
Al contrario, per poter compilare le versioni Desktop, è necessario NON avviare QTCreator da riga di comando. In questo caso è sufficiente cliccare sull’icona di Qt posta nella barra laterale sinistra.

# 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 prj CPUBridge). Ad oggi esistono ad esempio 2 classi concrete che implementato la classe base:

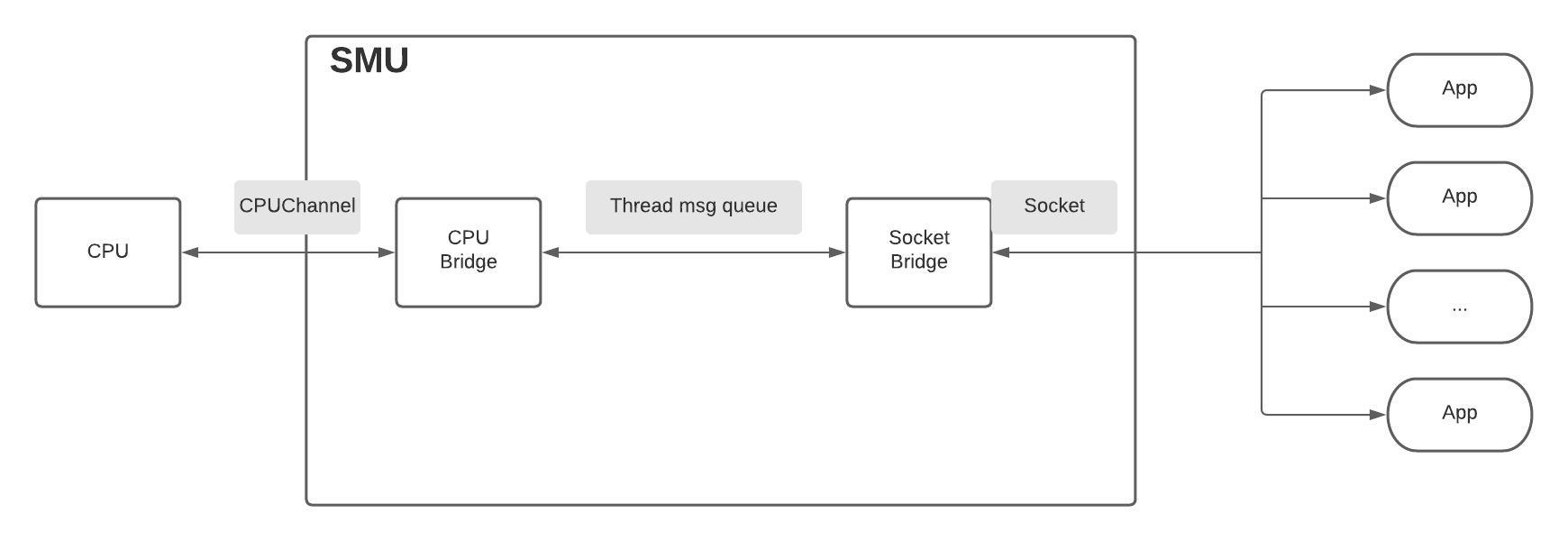
* CPUChannelCom che sfrutta una interfaccia seriale (RS232 /UART) per inviare e ricevere dati alla CPU
* CPUChannelFakeCPU che non invia i dati in nessun luogo ma semplicemente elabora le richieste proveniente da SMU e risponde di conseguenza. E’ una sorta di simulatore SW di CPU

Nella remota ipotesi che in futuro si volesse cambiare il canale fisico di comunicazione (ad esempio sostituendo la seriale con un canale wifi 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 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, è non bloccante e non prevede una risposta da SMU. Generalmente, alla ricezione di una richiesta di “evento”, SMU risponde generando una “notifica spontanea” ovvero una 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.hjtml è possibile (opzionalmente) intercettarli e gestirli tramite js. Ad esempio, per intercettare e gestire l’event 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 perè genera questi eventi solo quando il messaggio di CPU cambia. Fintanto che il messaggio rimane “Place your cup”, nessun evento spontaneo viene mai generato. Al caricamento di index\_xx.html quindi, per conosce 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.