FLOW SENSOR PROJECT

GPIB, UART CONNECTIONS

Questa parte è stata completata nelle due settimane che hanno preceduto l'inizio della nostra collaborazione.

Il mio lavoro si è diviso in due parti:

1. controllare con l'interfaccia ***GPIB*** la pompa d'aria ***"EAD NEO PP2"*** che genera il flusso d'aria
2. controllare con l'interfaccia ***USB (UART)*** la scheda ***Arduino UNO*** che prende i dati dal sensore digitale ***FS2012*** (già calibrato)

Banalmente, la soluzione più ovvia è quella di creare due sottoprogrammi (subVI nel gergo di LabVIEW) che gestiscano i due task separatamente. Un 'mainVI', poi, richiama le funzioni dei subVI quando richieste dall'utente tramite l'interfaccia grafica (front panel nel gergo di LabVIEW).

Nel progetto condiviso al link Struffoli11/MAIP - GitHub troverete i programmi "power_supply_FSM", "serial_connection" e "mainVI" che realizzano questa logica.

In seguito, chiarirò il dove ed il come ho implementato queste funzioni e per farlo illustrerò dei (semplici) meccanismi di LabVIEW che non vi sono stati spiegati al corso.

I meccanismi da introdurre necessari sono i seguenti:

a. macchine a stati finiti (FSM)
b. controlli personalizzati
c. code
d. eventi
e. costrutto Produttore-Consumatore

Può sembrarvi tanto ma bastano pochi commenti per spiegarveli.

A) MACCHINE A STATI FINITI

PERCHÉ UNA MACCHINA A STATI FINITI?

Una **macchina a stati finiti** mi permette di rappresentare in maniera astratta lo stato di una connessione tramite un'interfaccia (GPIB o UART). In base all'interazione dell'utente definirò uno stato in cui inizializzo la connessione, uno stato in cui invio dati tramite la connessione, uno stato in cui leggo dati dalla connessione, ed infine uno stato in cui termino la connessione. Un ulteriore stato talvolta presente è quello di "attesa", che rappresenta, per l'appunto, l'attesa di un comando da parte dell'utente.

COME SI CREA UNA MACCHINA A STATI FINITI IN LABVIEW?

Servono tre elementi per fare una macchina a stati finiti:

- Un ciclo *while*
- Una coppia di strutture *case* (come un costrutto switch in C)
- Un *controllo personalizzato*

La "faccia" di una macchina a stati finiti in LabVIEW è la seguente:

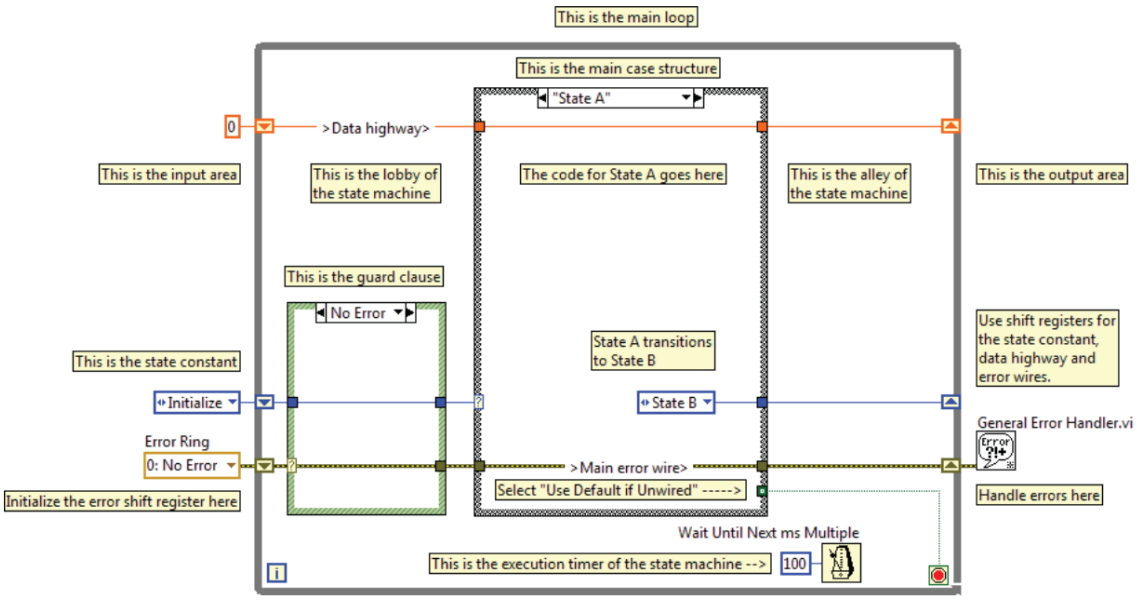


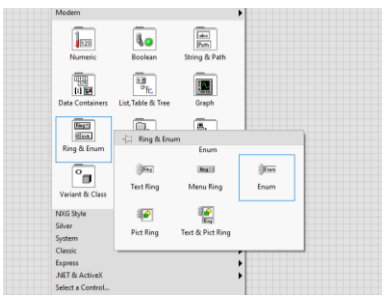
Figure 4.1: State machine terminology

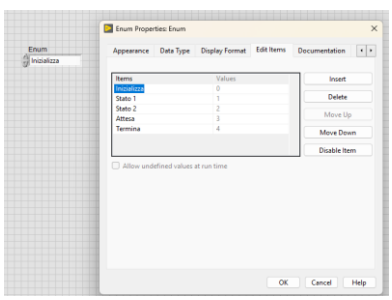
Tipicamente, a destra si inizializza la FSM dando uno stato iniziale ("Initialize"). Ove necessario, si inizializzano altri controlli.
Dentro il ciclo while troverete due costrutti case: uno (**guard clause**) assicura che la presenza di errori nelle funzioni implementate dagli stati arresti il ciclo while, altrimenti lascia che il ciclo while si ripeta infinitamente; l'altro (**main case structure**) esegue una funzione in base all'input ed allo stato attuale.
Le iterazioni del ciclo sono temporizzate (100 ms nell'esempio). Se non controllati diversamente, i "wires" che fuoriescono dalla struttura case principale assumono il valore dello stato che viene dichiarato di default (questo il significato di *"Use Default if Unwired"*). Notate che in ingresso e uscita dal ciclo while ci sono degli **"shift registers"**. Vi ricordo che i wires usati in questa modalità sono come delle variabili che nel ciclo while si aggiornano o mantengono il loro valore ad ogni iterazione. Nell'esempio, lo stato viene aggiornato ad ogni iterazione.
Come programmatore occorre trovare un meccanismo efficiente per rappresentare gli stati e gestire le transizioni da uno stato all'altro. Nell'esempio vedete un possibile approccio: **i controlli personalizzati**

B) CONTROLLI PERSONALIZZATI

COME DEFINISCO GLI STATI?

In LabVIEW possiamo definire nuove tipologie di controlli. Per farlo basta disporre un controllo sul front panel, assegnargli delle particolari proprietà (aspetto, valore massimo e minimo o valori possibili) e salvarlo. Per definire i possibili stati di una FSM creiamo un controllo che può assumere dei valori interi da 1 a N, dove N è il numero di stati possibili. Ad ogni numero si assegna un nome che identifica uno stato.
In effetti, questa è la descrizione di un "enum ring" nel gergo di LabVIEW, la particolarità è che la definizione di questo controllo mantiene i nomi associati ad ogni intero.





TRANSIZIONI TRA GLI STATI

Una soluzione è quella mostrata nell'esempio. Uno caso del costrutto case definisce anche lo stato successivo per la prossima iterazione. Gli stati, in questo modo, devono "sapere" le possibili transizioni che possono essere effettuate a partire da essi. Inoltre, non possiamo innescare una sequenza di stati volendo farlo (Attesa -> Stato 1 -> Stato 2 -> Stato 3 -> ... -> Attesa).
Come le persone in fila ad una cassa possono ignorare gli altri utenti che li precedono o li seguono, allo stesso modo gli stati possono ignorare lo stato successivo definendo un nuovo meccanismo: la **coda**.

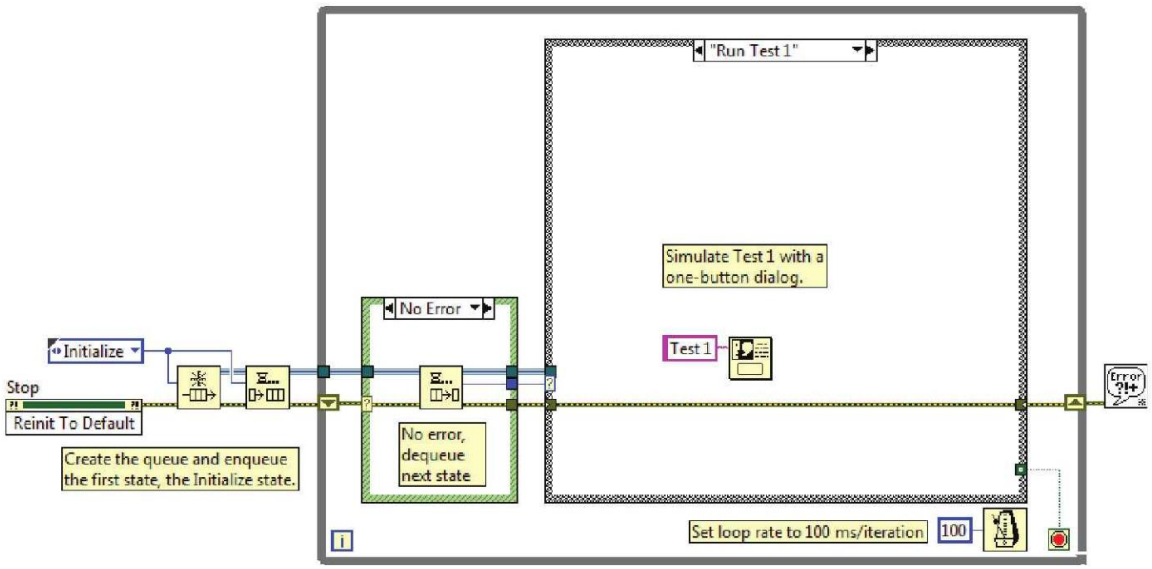


Figure 13.11: This figure shows the *Run Test 1* state; notice that there is no code in this state for determining what state to go to next—this is another benefit of using queues

Per creare ed utilizzare una coda occorrono 4 funzioni dalla subpalette *Synchronization/Queue Operation*:


Obtain Queue : usato per creare la coda


Enqueue Element : usato per mettere in coda un altro elemento


Dequeue Element : usato per prelevare un elemento dalla coda


Release Queue : usato per eliminare la coda.

Il tipo di elemento che utilizzo nella coda è di solito il controllo personalizzato che rappresenta un determinato stato.

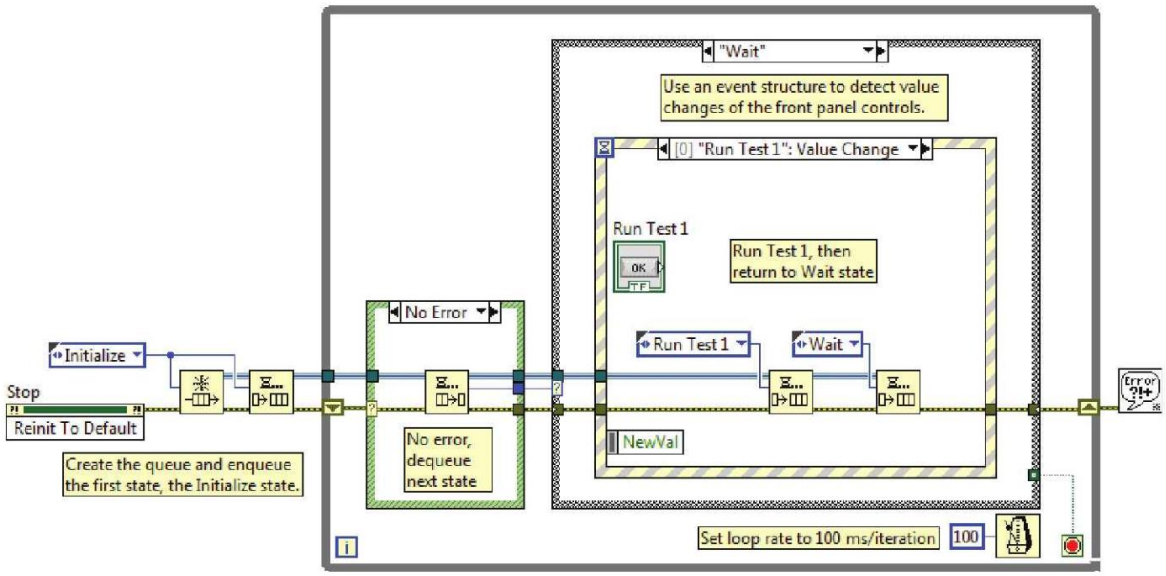


Figure 13.8: This figure shows the event case handling the value change of the RUN TEST 1 button; notice that a series of states are queued, showcasing the advantage of using a queue

D) EVENTI

Quando uno stato termina il proprio compito, tipicamente occorre attendere un nuovo input da parte dell'utente tramite i controlli sul front panel. Per rilevare ed identificare il tipo di interazione e il controllo oggetto del comando impartito dall'utente in LabVIEW usiamo la struttura "**Event Case**".

ATTENZIONE. Questa struttura blocca il programma. Quest'ultimo si sblocca solo all'avvenire di uno degli eventi definiti nella struttura degli eventi (nell'esempio il cambiamento del valore di un pulsante chiamato "Run Test 1").

Figure 27.4: The front panel of the producer-consumer `Multitest` VI

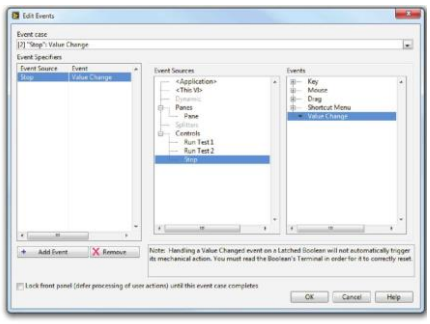
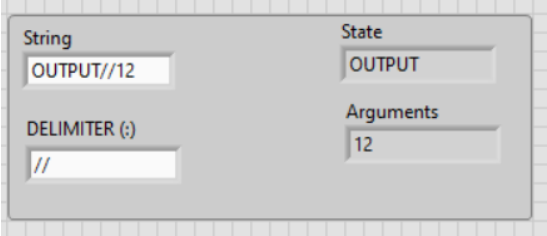
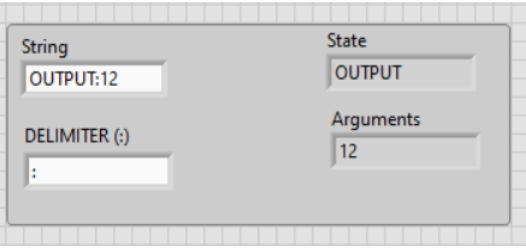


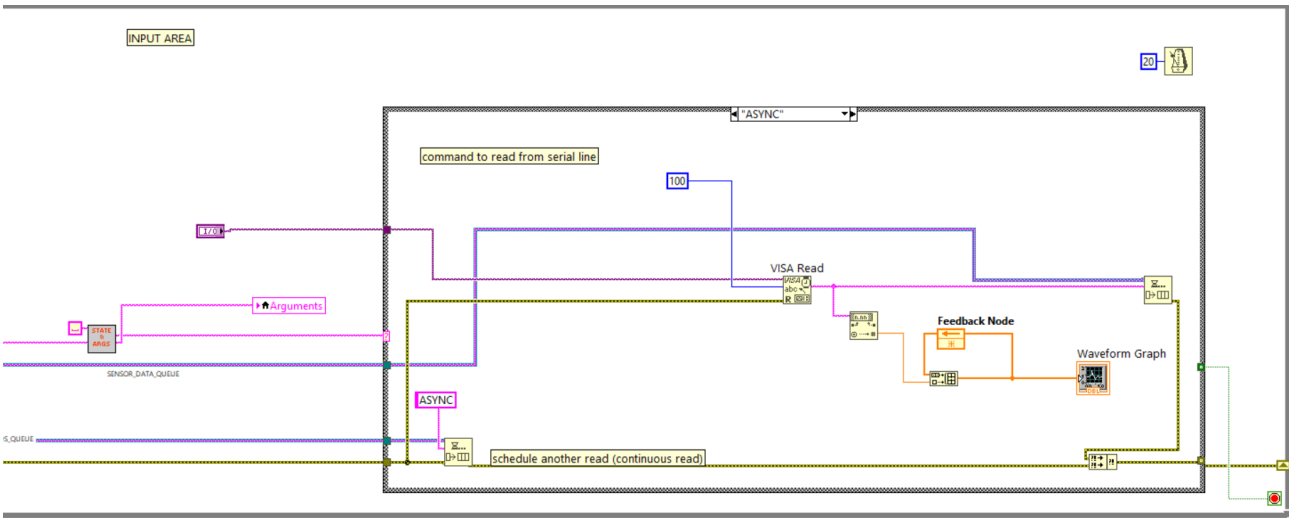
Figure 10.3: The configuration window of the STOP value change event case

Noterete nei subVI che terminano in FSM l'uso di stringhe al posto delle costanti di controlli personalizzati. Questo perché, nella visione di uno stato come implementazione di una particolare funzione del subVI, voglio poter anche "passare degli argomenti" come in C. Per questo scopo ho creato un subVI "state_and_args" che decompone una stringa utilizzando un delimitatore. Quindi se dalla coda prelevo la stringa "OUTPUT:12", nella FSM che controlla la connessione GPIB passerò nello stato di OUTPUT dove impartirò al generatore di tensione continua di settare il valore di tensione di uscita a 12V.

NOTA. Qui potrebbe essere generato un errore.







RIASSUNTO DI QUANTO VISTO FINORA

Per rappresentare gli stati della connessione GPIB, UART ho creato delle macchine a stati finiti. Per rappresentare gli stati ho creato dei controlli personalizzati. Per gestire la transizione da uno stato all'altro ho utlizzato una combinazione dei meccanismi di Event Structure e Queue (coda). Nei subVI che implementano questa logica, tuttavia ho usato nelle code delle stringhe che contengono lo stato successivo che seguirà lo stato attuale ed eventuali argomenti (parametri) che intendo utilizzare nello stato successivo.

L'ultimo tassello del puzzle è il mainVI. Qui ho utilizzato un meccanismo chiamato **Producer Consumer Pattern**

E) PRODUCER-CONSUMER PATTERN

Essenzialmente aggiungiamo un altro ciclo while (produttore), che viene eseguito parallelamente all'altro (consumatore). Il ciclo while del produttore genera le transizioni di stato nel ciclo while del consumatore

in seguito ad un determinato evento definito in una struttura ad eventi. Potrebbero esserci non uno bensì numerosi consumatori - nel mainVI vi sono quattro consumatori. Questo meccanismo permette, quindi, di mettere nella coda del consumatore degli eventi della connessione UART il successivo stato a seguito di un cambiamento di un parametro di tale connessione (ad esempio BAUD RATE); di innescare una transizione nella FSM che rappresenta lo stato della connessione GPIB a seguito di un cambiamento della corrente in uscita desiderata, ad esempio.

Oltre ad i consumatori di eventi per quanto riguarda la connessione GPIB o UART, ho aggiunto anche un meccanismo di data logging. Quindi per ogni evento nel ciclo while del produttore salvo in un file tutte le azioni che l'utente genera interagendo con il front panel. Questo mi permette, in un secondo momento, di replicare la sequenza di interazioni tra l'utente e il programma.

Per finire, un meccanismo di rampa di corrente, ai fini di future procedure di calibrazione, è stato implementato con una semplice macchina a stati finiti nel ciclo consumatore "Current Ramp".

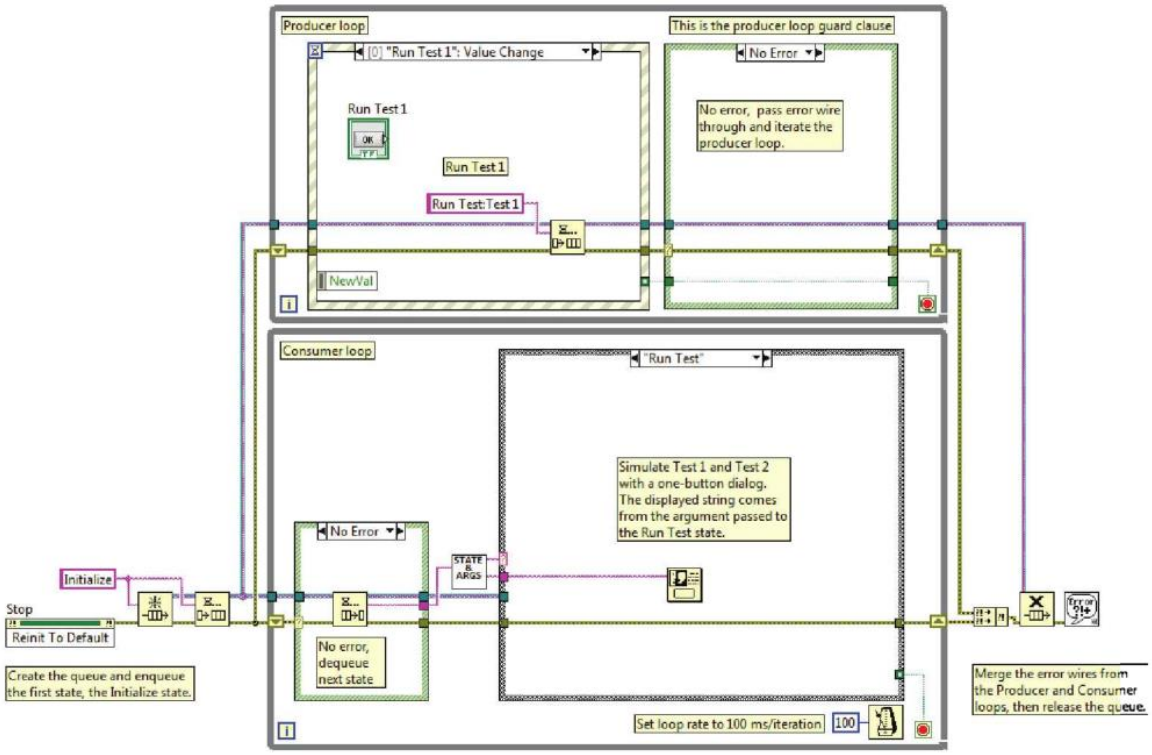
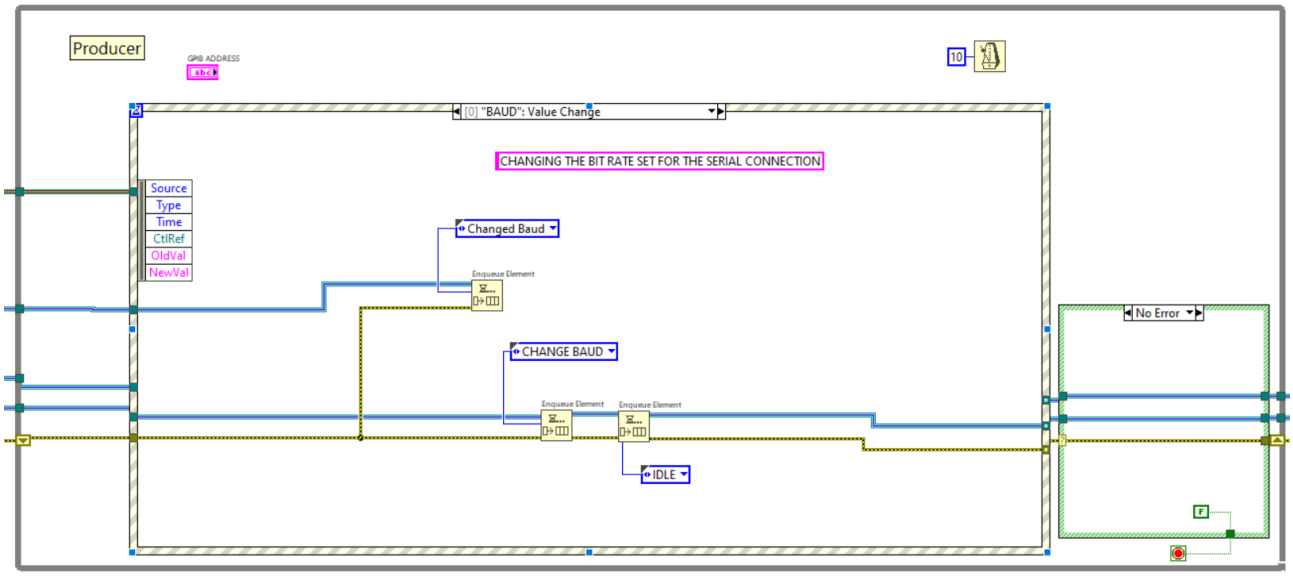
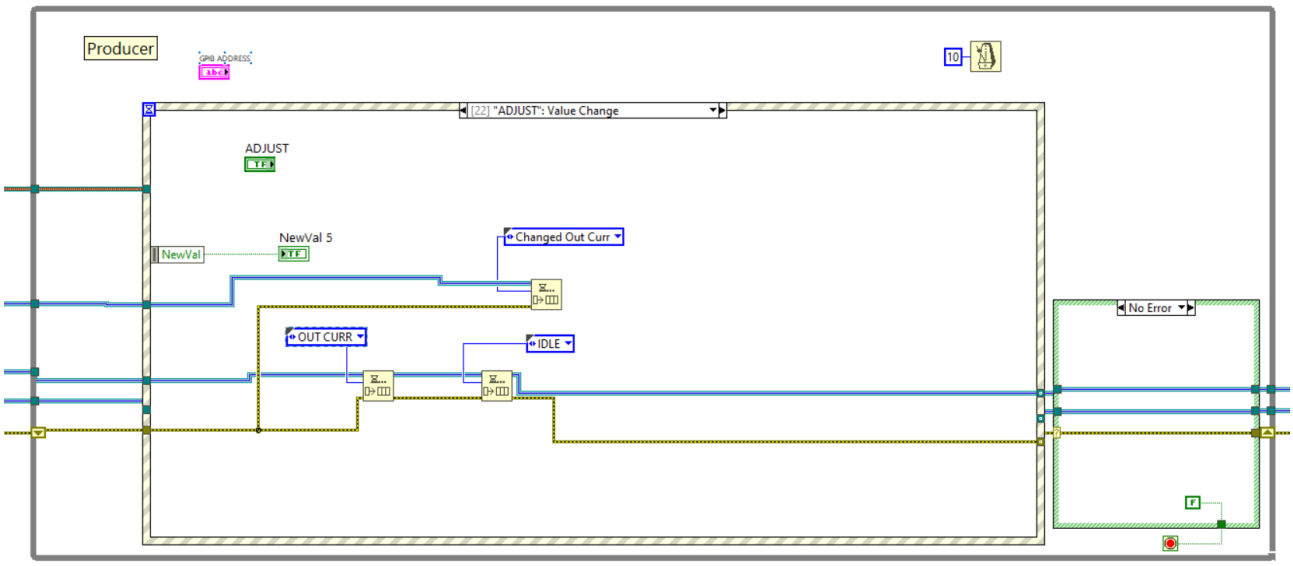


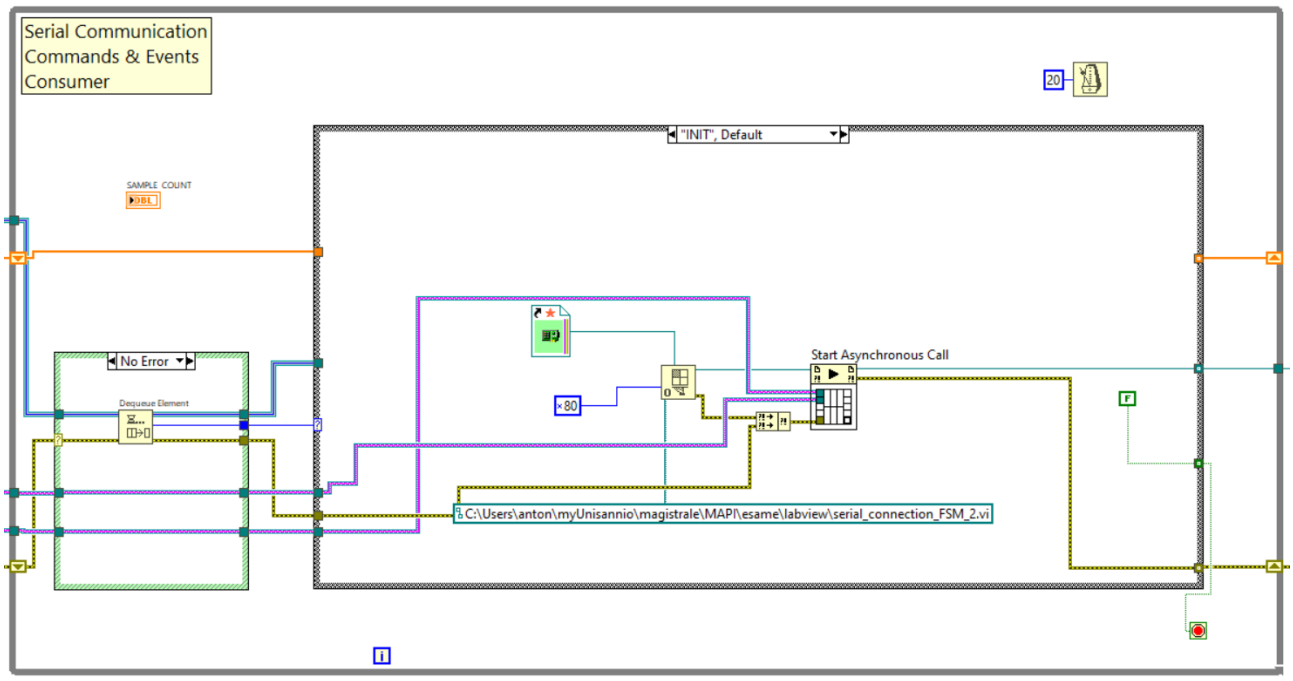
Figure 27.5: The *Run Test 1* event case and *Run Test* state of the producer–consumer `Multitest` VI



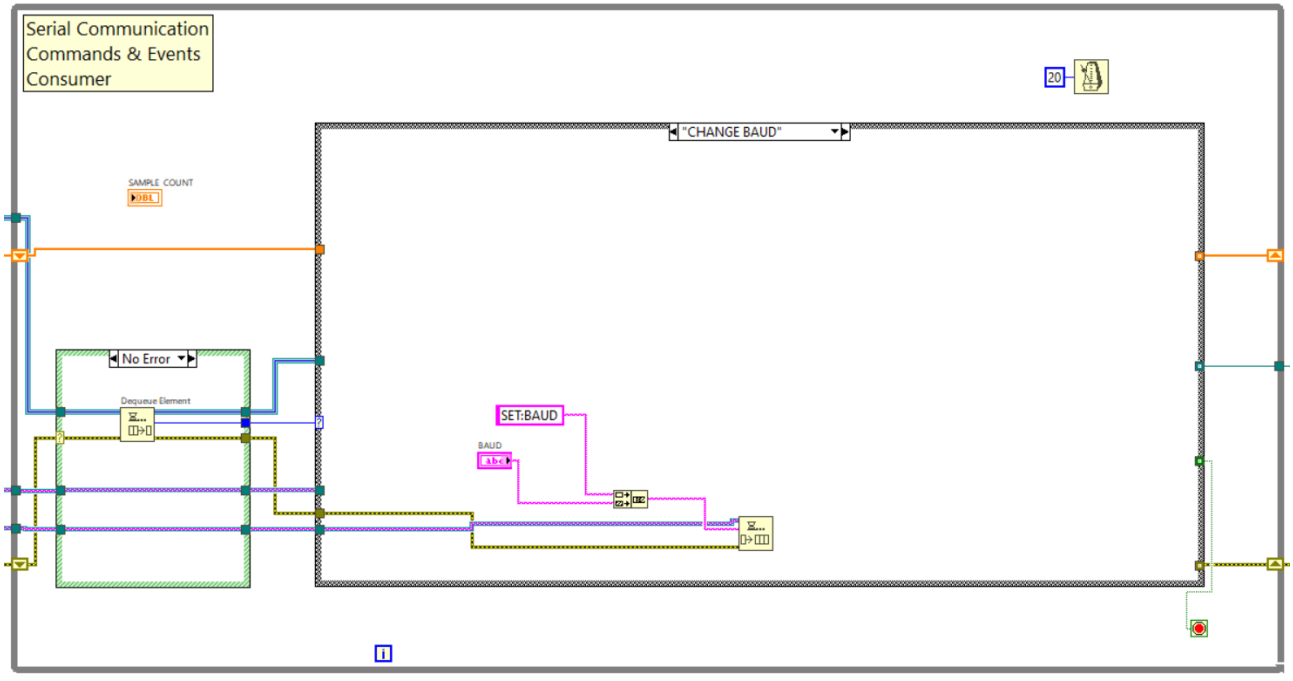


UN'ULTIMA OSSERVAZIONE

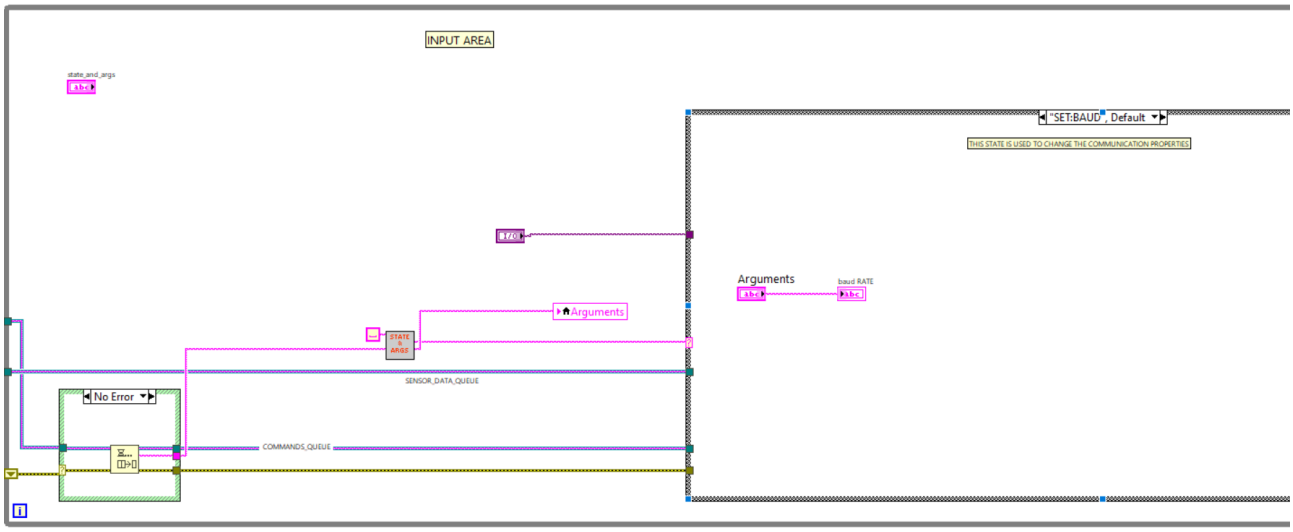
Nel block diagram del mainVI non vi è traccia dei subVI che implementano le FSM. Questo perché utilizzo **chiamate asincrone** a questi sottoprogrammi. Quindi, sebbene non interagiamo con il loro front panel, posso impartire transizioni di stato attraverso il meccanismo delle code.



A: nel mainVI (consumer loop)



B: nel mainVI (consumer loop)



C: nel subVI "serial_connection_FSM"