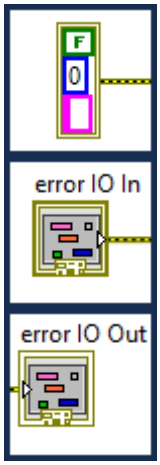
Gestire gli errori localmente è sempre la scelta migliore?

Category: LabVIEW Talk
scritto da Lorenzo Borghi | Aprile 14, 2023



Cos’è l’errore?



In LabVIEW™ l’errore è un tipo di dati definito (TypeDef) basato su un cluster, che contiene 3 elementi distinti:

Stato: Un booleano che segnala la presenza dell’errore, agendo nel caso come flag e interrompendo l’esecuzione di un VI.
Il valore *FALSE* del flag indica assenza di errore, ma può essere associato alla presenza di un warning.

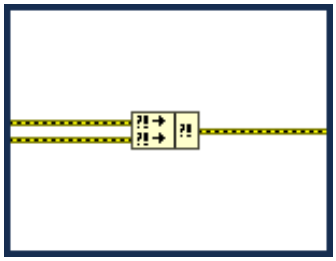
Codice: Un numero intero che identifica l’errore stesso; in alcuni casi lo stesso codice può genericamente identificare errori diversi.
Nel caso di default il codice ha valore 0, equivale a *No Error.*

Sorgente: Specifica l'origine dell'errore o del warning; nella maggior parte dei casi mostra il nome del nodo che produce l'errore. In caso non ci siano errori o warning, la stringa non conterrà alcun dato e sarà pertanto vuota.

Bundle e Unbundle

Con le funzioni di Bundle e Unbundle, ogni elemento del cluster di errore può essere sovrascritto per personalizzare l' errore o warning. Per esempio semplificare o tradurre la stringa del messaggio.

Merge Errors



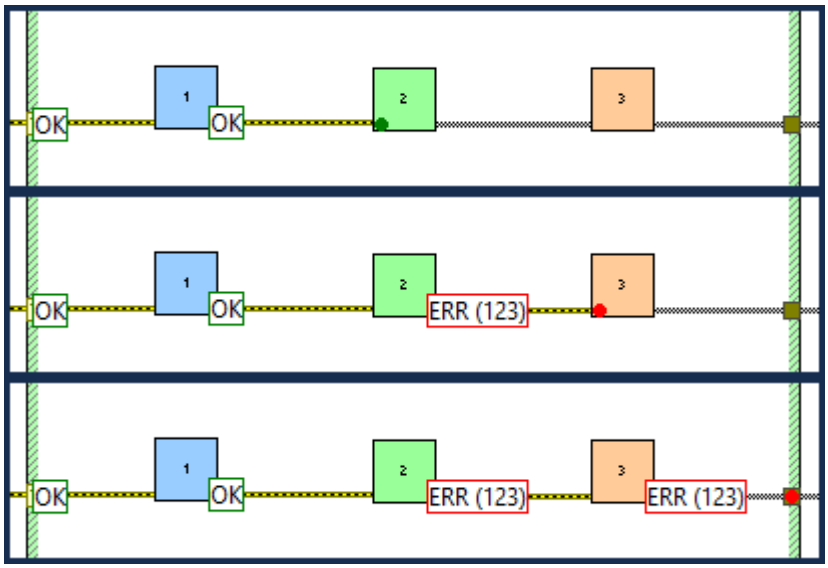
Lo scopo di collegare tra di loro i cluster di errore è quello di collimare in un unico punto tutti gli errori per poterli gestire tramite un eventuale apposito VI; per poter propagare correttamente un errore senza sovrascriverlo si può utilizzare la funzione *Merge Errors* che esegue l'append di un errore all'altro.

Error terminals per il DataFlow

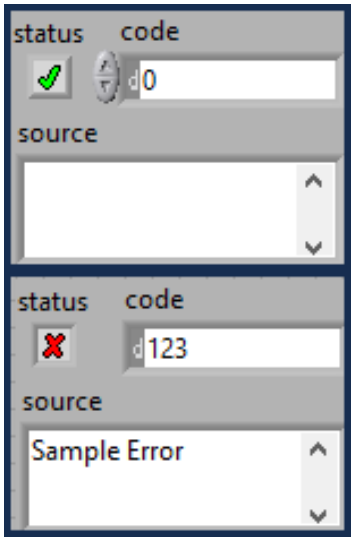
La linea guida su stile di programmazione, suggerisce nella progettazione di un subVI, l'inserimento dei terminali in basso.

Con questo procedimento di ingresso uscita, possiamo dare una sequenza ai VI in pieno accordo alle regole del DataFlow e allo stesso tempo propagare l'errore.

Come si propaga il segnale di errore?



All’interno dei VI è necessario implementare un collegamento che permetta ad un eventuale errore, di essere propagato fino all’indicatore presente sul front panel, poiché questo venga mostrato a schermo e informi all’utente che si è verificato un problema durante l’esecuzione.



Se sul terminale d’ingresso di un SubVI si presenta un errore generato da una funzione precedente, si deve propagare, quindi essere passato in uscita:

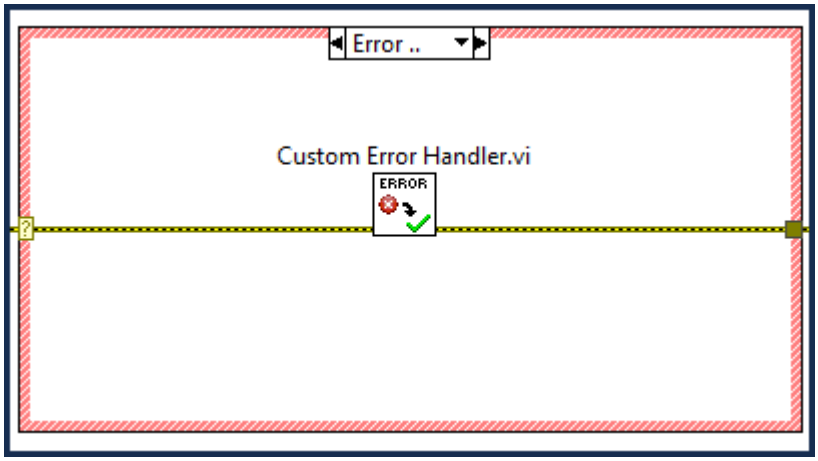
- **Senza eseguire il codice [ErrorCase]**

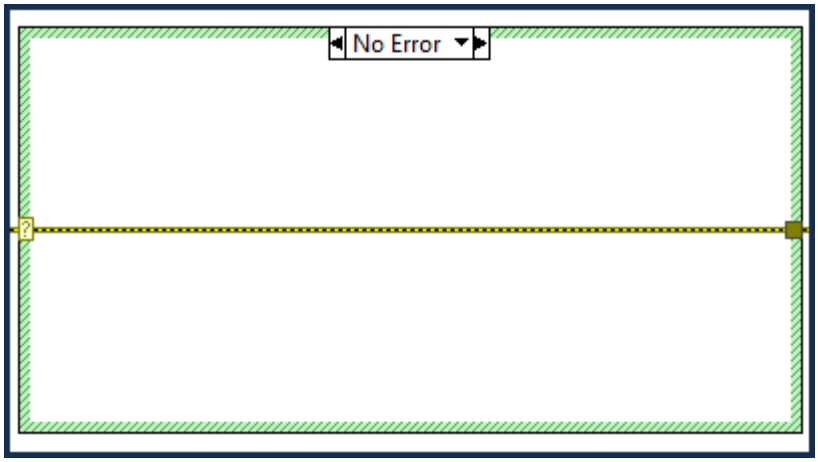
Quando si ha un codice piu o meno complesso che potrebbe generare errori indiretti.

- **Eseguendo il codice [ByPass]**

Quando si hanno funzioni senza terminali di errore, es. una funzione matematica. In questi casi, si può semplicemente collegare i terminali di Errore internamente.

Template subVI [Error]





All’interno di ogni SubVI è quindi necessario incapsulare il codice all’interno di una *case structure* collegata al controllo di errore, così si avrà modo di poter gestire in tutta libertà un eventuale errore e decidere se si vuole comunque eseguire i blocchi presenti nel sub-VI o non fare nulla e propagare l’errore cosi come è entrato.

Perché non gestire l’errore nel subVI?

Gestire l’errore se si intende solo rielaborarlo per presentarlo in uscita ad esempio tradotto ecc. Questo è una buona pratica da fare localmente dove sappiamo cosa fa il VI.

Non è invece una buona pratica inserire il codice di azione internamente, esempio mettere una funzione ErrorMessageHandler.

Anche eseguire il codice all’interno del VI in presenza di un errore in ingresso, è sbagliato.

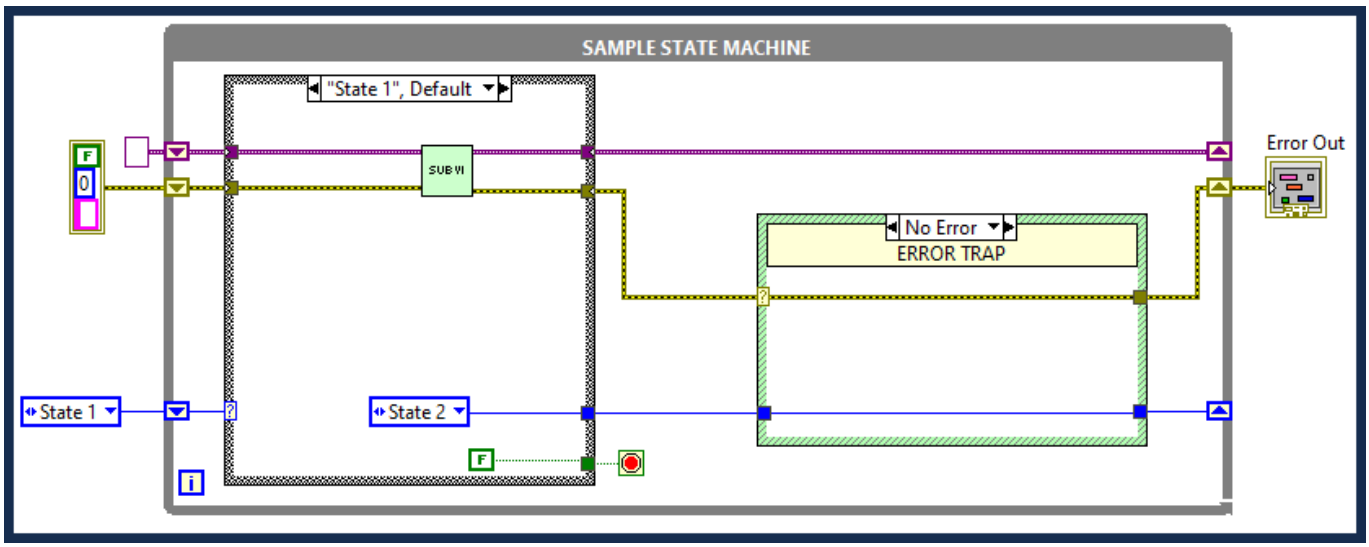
Immaginiamo tre VI Open; Read; Close

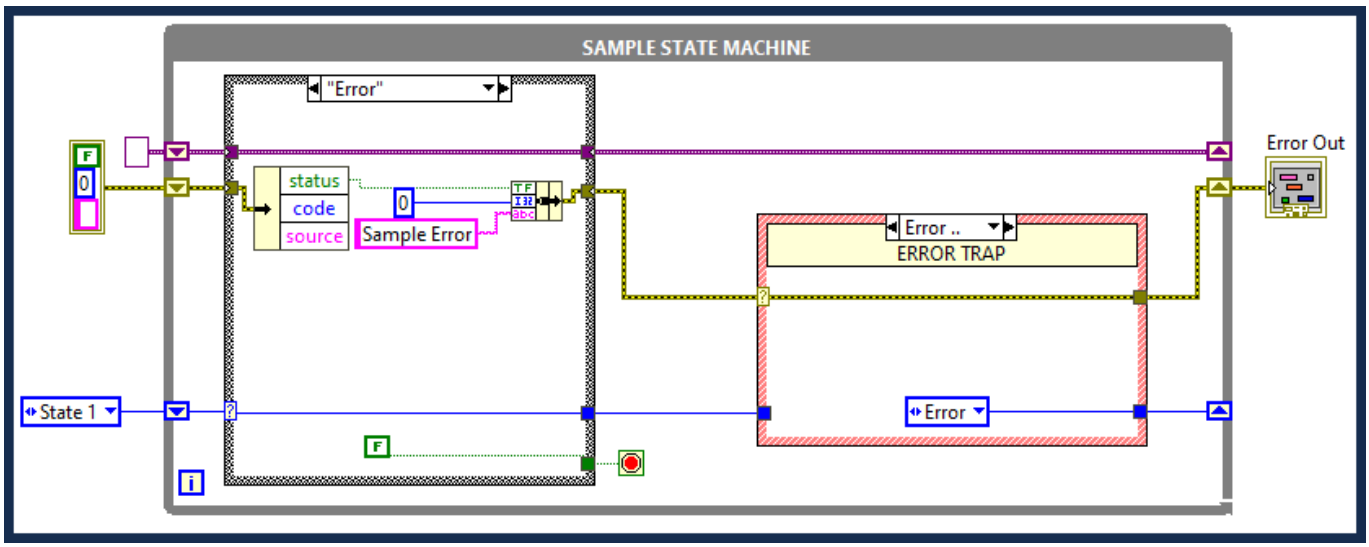
Se Open esce con un errore di path inesistente (questo il vero errore); Read produce un errore di reference(fake error) e anche Close risponde con un errore di mancata ref(fake error).

Propagando invece, in uscita al Close ottengo solo il primo errore generato da Open.

Posso ora trattarlo quindi a fine sequenza o per esempio in un pattern State Machine, catturarlo (**Error Trap**) e inviarlo ad un case Error, centralizzando in un unico posto le Azioni su errore.

Error Trap per State Machine





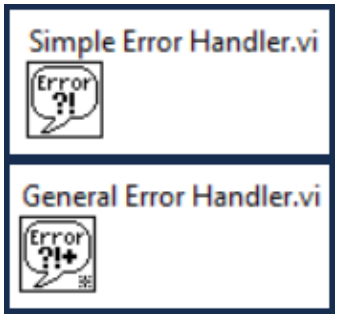
Permette di inviare l’errore propagato da uno stato, ad uno stato centrale “**Error**“.

Error Dispatch Centralizzato

Qui il programmatore, decide le azioni in base al codice o al tipo di errore ricevuto.

Azioni su Errore

Notifica



LabVIEW™ offre due strumenti per gestire gli errori in automatico: *Simple Error Handler* e *General Error Handler*.

La principale differenza è che il Simple richiama il General semplificandone l'utilizzo conservando solo le funzioni più basilari.

Entrambi i gestori automatici di errori funzionano allo stesso modo, in caso di errore mostrano una descrizione e opzionalmente un box di dialogo.

Semplificazione e localizzazione del messaggio

Proprietà dell'Automatic Error Handler

Una proprietà abilitata di default sui nuovi VI.

Inizialmente comoda perché lascia a LabVIEW la gestione dell'errore in caso non abbiamo collegato i terminali di errore.

Tuttavia andrebbe disabilitata, poiché la comodità, induce l'utente a non collegare tra loro i cluster di errore, aumentando la probabilità che si verifichi una eccezione o un errore non gestito.

Azioni di Retry

Centralizzando l'errore posso prendere le decisioni sull'azione da intraprendere.

Se tentiamo di connetterci ad un dispositivo e riceviamo un errore, possiamo verificare se la com è presente, chiudere la connessione e riprovare su un altra com .

Possiamo anche notificare "Hai collegato il cavo?, Premi OK dopo verifica" e quindi rieseguire la open connection. In questo caso guido la risoluzione ed effettuo il retry.

Azioni di Safety

Solitamente abbiamo a che fare con applicazioni che controllano Hardware dove le condizioni di sicurezza sono vitali.

Immaginiamo cosa succede se chiudo un applicazione senza arrestare un motore.

Una uscita su condizione che esegua una procedura di Safety portando il sistema in sicurezza deve essere prevista nella gestione.

Se il SubVI non è progettato per propagare correttamente l’errore, si potrebbe generare un nuovo errore che può mettere in difficolta capire correttamente in quale fase del progetto si sia verificato il malfunzionamento, rendendo complicato svolgere operazioni di individuazioni e debug.

È importante gestire correttamente la propagazione di un errore all’interno di ogni funzione di basso livello, poiché se un errore viene sostituito da un altro in cascata, si perde la memoria di quello cronologicamente eseguito prima.

L’azione di sovrascrittura generalmente crea ambiguità che non permette la corretta comprensione di quale fase del processo abbia fallito, ad esempio: se in un impianto una pressa comunica un errore di istruzione non accettata, e subito dopo si scollega la porta di connessione con il driver.

Se l’errore è gestito in modo scorretto, la disconnessione sovrascriverà l’errore di funzione, portando a conseguenze potenzialmente anche molto gravi.

Propagare l’errore è estremamente importante per mantenere il data flow senza ricorrere a strutture di tipo sequence che forzano un passaggio di azioni in modo macchinoso e non personalizzabile.

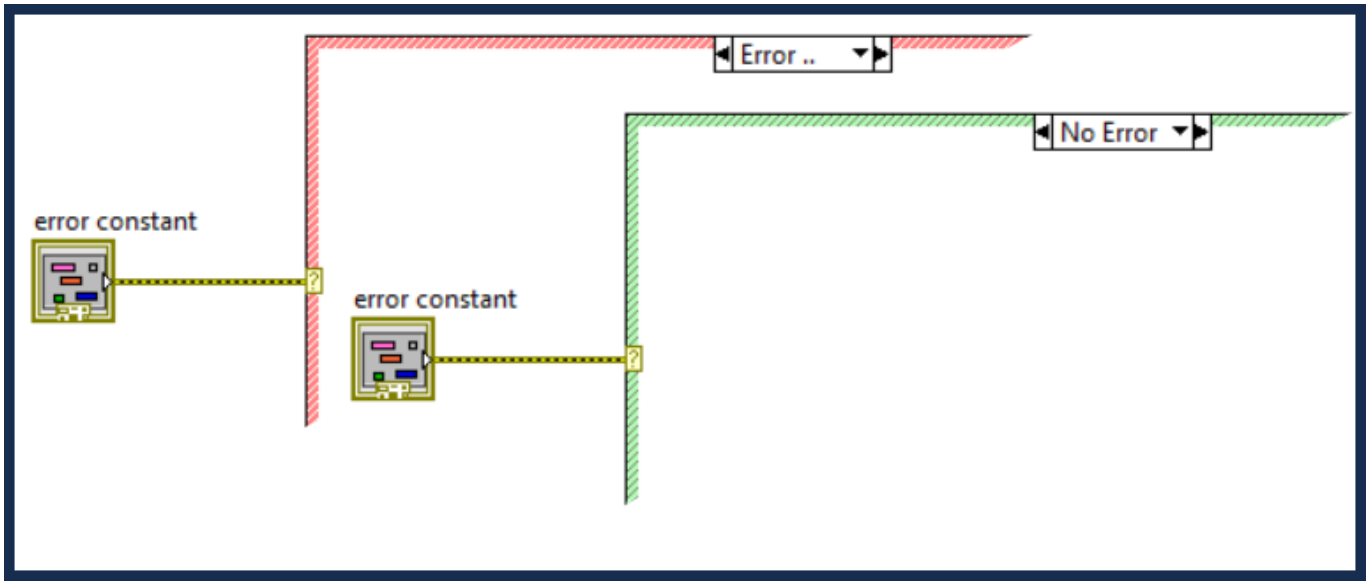
Saper gestire correttamente il flusso degli errori permette in fase di debug di poter risalire all’origine dell’errore e gestire ogni situazione nel momento stesso in cui si genera.

Evitare di risolvere localmente l’errore

Un altro errore comune in cui si cade spesso è quello di gestire l’errore localmente in ogni VI, senza affidare il compito ad un SubVI apposito; la gestione locale infatti va evitata perché può generare pesanti concatenazioni di errori.

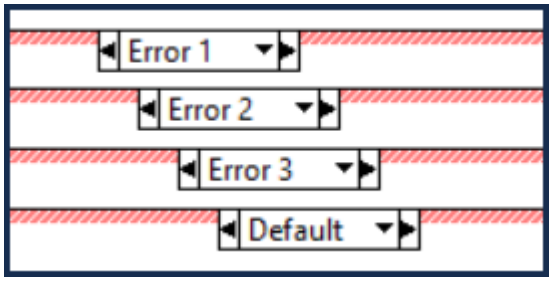
La best practice é quindi quella di centralizzare la risoluzione dell’errore tramite un unico VI generalmente posto nelle funzioni di livello più basso.

Gestire azioni sul codice errore



Per gestire efficacemente in maniera relativamente semplice l'errore, si collega il relativo cluster ad una *Case Structure*, che presenterà i bordi colorati di verde per il caso *No Error* e rosso per un qualsiasi altro caso *Error*.

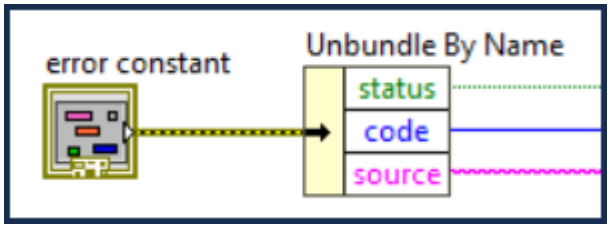
Nella struttura è possibile aggiungere infiniti casi di errore ognuno associato al relativo codice, tramite il comando *Add Case After/Before*.



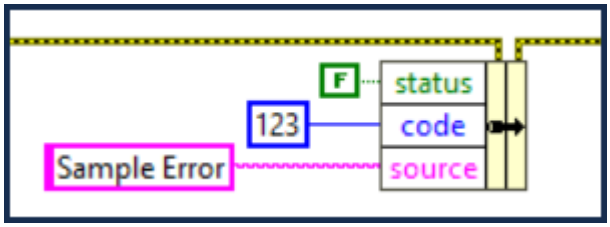
Tuttavia è necessario sottolineare che insieme a tutti i casi di errori personalizzati, bisogna inserire un caso di *Default*, altrimenti il progetto risulterà rotto e non funzionerà.

Personalizzazione e Traduzione del messaggio di errore.

Come specificato all'inizio dell'articolo il cluster di errore è personalizzabile in ogni suo elemento.



Tramite la funzione *Unbundle/Unbundle By Name* presenti nella palette *Cluster, Class & Variant* è possibile scomporre il cluster nelle sue variabili base.



Se si vuole modificare il valore dei suoi elementi occorre utilizzare le funzioni contrarie a quelle appena viste: *Bundle/Bundle By Name*.

Per capire meglio cosa si intende con personalizzazione, si può pensare di crearsi una libreria propria di codici di errori e assegnare valori arbitrari ordinati personalizzati.

Nel caso in cui all'ingresso della *Case Structure* si presenti un errore con codice **-1073807346**, che nella libreria di LabVIEW™ corrisponde all'errore di sessione o oggetto invalido, di modificare il codice con il numero 1 e sostuire la descrizione con una personalizzata più semplice.

Con lo stesso ragionamento, si può pensare di associare il codice 2 all'errore **-1073807339** che identifica il timeout di connessione con uno strumento.

Procedendo per questa strada per ogni tipo di progetto, è possibile creare liste di errori configurate su misura con codici e descrizioni specifiche per il tipo di malfunzionamento, garantendo una maggiore leggibilità da tutti gli utenti che lavoreranno o utilizzeranno il progetto.

Errori logici, prevenzione di risultati imprevisti.

Una divisione per 0 non restituisce un errore ma il risultato non può essere quantificato nell'indicatore, producendo un NaN (Not a Number). In questo caso è possibile notificare che il denominatore deve essere maggiore di 0, ma potrebbe essere preventiva ovvero coercizione del range degli input.

Se non posso mettere 0 non avrò l'errore logico.

Per garantire una corretta user experience, è buona norma accompagnare l'errore con una notifica, in questo modo vengono fornite informazioni all'utente aggiuntive sulla la forzatura del dato.

Un altro caso di errore comune si presenta quando si inserisce un percorso vuoto (path error) nei VI che prevedono un ingresso di questo tipo.

Prevenire loop infiniti

Una condizione che può diventare incontrollabile è il ciclo infinito, ovvero un loop dove la condizione di uscita non si verifica, bloccando il resto dell’esecuzione.

La regola è inserire sempre un uscita per timeout e poi decidere se notificare o meno l’errore di timeout.