PROGETTAZIONE SOFTWARE PER IL CONTROLLO MACCHINA

Chris Washington

Questo articolo illustra una possibile architettura di progettazione software per il controllo macchina, basata sulle macchine a stati

L'architettura che descriveremo qui può essere utilizzata per realizzare applicazioni di controllo su controllori LabVIEW Real-Time, inclusi CompactRIO, Compact FieldPoint e PXI. La stessa architettura di base può funzionare anche su altre piattaforme, come controllori basati su Windows. LabVIEW Real-time è un linguaggio di programmazione completo, che offre agli sviluppatori numerose opzioni su come costruire un controllore e permette loro di creare sistemi molto flessibili e complessi.
I controllori LabVIEW Real-Time sono utilizzati in applicazioni che spaziano dal controllo delle barre in centrali nucleari, al collaudo Hardware-in-the-Loop per le ECU dei motori, dal controllo adattativo per la trivellazione di pozzi petroliferi, al monitoraggio delle vibrazioni ad alta velocità per la manutenzione predittiva. Tuttavia, la grande flessibilità di LabVIEW Real-Time potrebbe anche scoraggiare l'inizio di una nuova applicazione, specialmente per i programmatori che non hanno familiarità con LabVIEW o con la programmazione di sistemi real-time. Questo articolo non ha lo scopo di fornire un'architettura valida per ogni applicazione e non vuole sostituirsi ai corsi di formazione su LabVIEW Real-Time. E' stato scritto per offrire un'infrastruttura operativa ai progettisti di applicazioni di controllo industriale e specialmente a chi ha familiarità con l'uso dei PLC. L'articolo è stato concepito per spiegare come costruire facilmente applicazioni LabVIEW Real-Time in grado di offrire anche la flessibilità necessaria a gestire applicazioni non tradizionali, come I/O bufferizzati ad alta velocità, data logging e visione artificiale.

TERMINOLOGIA

Nonostante vi siano molti modi per costruire un'applicazione di controllo in LabVIEW Real-time, è necessario iniziare dalla comprensione di alcuni concetti fondamentali relativi alla programmazione real-time e alle applicazioni di controllo.

Responsività: Un'applicazione di controllo deve essere in grado di intervenire nel caso di eventi come un cambiamento di I/O, un input dal dispositivo HMI o un cambiamento di stato interno. Il tempo richiesto per eseguire un'azione dopo un evento è noto come responsività e applicazioni di controllo differenti avranno tolleranze differenti per la responsività, che possono variare da microsecondi a minuti. La maggior parte delle applicazioni industriali ha requisiti di responsività compresi fra il millisecondo e qualche secondo. Un criterio di progettazione importante per un'applicazione di controllo è la responsività richiesta, perché essa determinerà le velocità dei loop di controllo e influirà sulle decisioni a livello di I/O, processore e software.

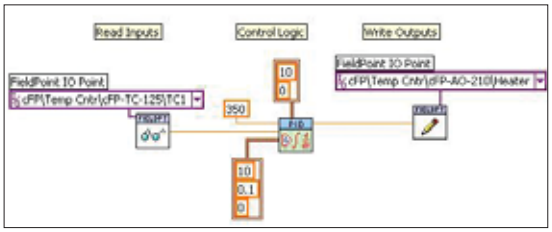
Determinismo e jitter: Il determinismo consiste nella garanzia di ripetibilità dei tempi di un loop di controllo.
Il jitter rappresenta una misura del determinismo, cioè l'errore che si ha nel mancato rispetto dei tempi-ciclo. Per esempio, se un loop è impostato per essere eseguito ogni 50 ms ma qualche volta viene eseguito a 50,5 ms, il jitter è di 0,5ms. Determinismo e affidabilità più elevati sono i vantaggi primari di un sistema di controllo real-time: un buon determinismo è critico per applicazioni di controllo stabili.
Un basso determinismo porta a un cattivo controllo analogico e può rendere un sistema non responsivo.

Priorità: La maggior parte dei controllori usa un singolo processore per gestire tutti i task di controllo, monitoraggio e comunicazione. Essendoci così una singola risorsa (il processore) con più richieste in parallelo, il programmatore deve trovare il modo per gestire quelle più importanti.
Impostando i loop critici di controllo in alta priorità, ottenete un controllore che presenta ancora buon determinismo e responsività. Per esempio, in un'applicazione con un loop di controllo della temperatura e funzionalità di logging embedded, il loop di controllo può essere impostato a priorità più elevata rispetto all'operazione di logging per ottenere un controllo deterministico della temperatura. Ciò fa sì che i task di priorità inferiore (logging dei dati, web server, HMI, ecc.) non influiscano negativamente sui controlli analogici o sulla logica digitale.

RICHIAMI SULL'ARCHITETTURA DI BASE DEI CONTROLLORI

La maggior parte dei programmatori che non hanno familiarità con LabVIEW iniziano con un'applicazione molto semplice ed espandono quindi il loro programma per aggiungere maggiori funzioni. Un vantaggio di

LabVIEW è che è molto facile costruire ed eseguire un semplice programma. Per esempio, vediamo la logica di un semplice loop PID eseguito su un sistema Compact FieldPoint.



Purtroppo, costruire sistemi più complessi in LabVIEW richiede un'architettura più sofisticata che permetta un migliore riutilizzo del codice, scalabilità e gestione del flusso di esecuzione. Questo paragrafo descrive ed insegna a costruire un'architettura di base per le applicazioni di controllo ed esegue lo stesso loop PID utilizzando tale architettura.

Il controllore più semplice ha tre sequenze o routine fondamentali:

1. Routine di inizializzazione
2. Routine di controllo
3. Routine di chiusura

Tali routine riguardano lo stato del controllore. Dato che i controllori real-time di National Instruments vi permettono di definire le capacità complete del vostro sistema, dovrete costruire le sequenze in LabVIEW.

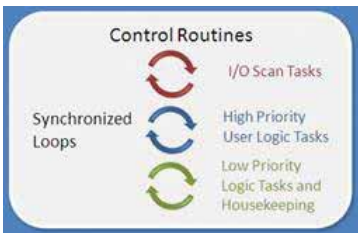
ROUTINE DI INIZIALIZZAZIONE

Prima di eseguire il loop di controllo principale, il programma deve eseguire una routine di inizializzazione che prepara il controllore per l'esecuzione. Non è qui che trova posto la logica di gestione della macchina (come la logica di startup o di inizializzazione della macchina); tale logica dovrebbe essere inserita nel loop di controllo principale. La routine di inizializzazione dovrà quindi:

1. Impostare allo stato di default tutte le variabili interne, incluse le variabili d'uscita nella tabella di memoria degli I/O.
2. Creare tutte le code necessarie, le FIFO RT, i VI Refnum e scaricare ogni bit file negli FPGA.
3. Eseguire eventuale logica aggiuntiva definita dall'utente per rendere operativo il controllore, come la preparazione dei file di log.

ROUTINE DI CONTROLLO

La routine di controllo è il cuore del controllore, dove viene eseguita tutta la logica utente in forma di loop multipli paralleli. Anche se potete schematizzarla come un loop che esegue l'I/O Scan Task, i task ad alta priorità e quindi i task a bassa priorità, in realtà si tratta di loop multipli sincronizzati.



Il primo loop scrive e legge gli I/O fisici in una tabella di memoria e viene eseguito per primo. Quindi, i task successivi eseguono la logica utente dalla priorità più elevata a quella più bassa, terminando con i task di comunicazione e operazioni ausiliarie che funzioneranno alla minima priorità. Per servire questi ultimi task efficacemente, dovete verificare di avere programmato il vostro controllore garantendo un tempo sufficiente di inattività del processore.

ROUTINE DI CHIUSURA

Quando si comanda al controllore di interrompere l'esecuzione o il controllore rileva una condizione di errore, esso interrompe l'esecuzione del loop di controllo principale ed esegue una routine di chiusura. La routine ferma il controllore e lo mette in sicurezza: serve solo per lo shutdown del controllore e non è la sede per le routine di chiusura della macchina, che invece vanno inserite nel loop di controllo principale. La routine di chiusura deve:

1. Impostare tutte le uscite a uno stato di sicurezza
2. Interrompere tutti i loop paralleli in esecuzione
3. Eseguire qualsiasi logica aggiuntiva come quella di notifica all'operatore di un guasto al controllore o informazioni sullo stato del logging

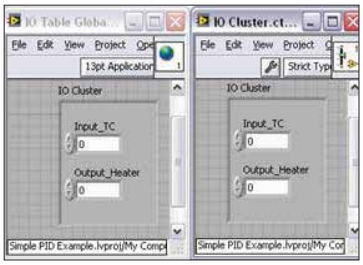
DETTAGLI DELLA ROUTINE DI CONTROLLO

La maggior parte dei programmatori LabVIEW ha familiarità con l'accesso diretto agli I/O, dove le subroutine inviano e ricevono direttamente input e output dall'hardware. Questo metodo è ideale per l'acquisizione di forme d'onda, l'elaborazione dei segnali e piccole applicazioni single-point. Tuttavia, le applicazioni di controllo normalmente utilizzano letture e scritture single-point e possono diventare molto grandi, con stati multipli che devono tutti accedere agli I/O. L'accesso agli I/O introduce un overhead nel sistema e può essere relativamente lento; inoltre, gestire accessi multipli agli I/O attraverso tutti i livelli di un programma rende molto difficile cambiare gli I/O e implementare funzionalità come la simulazione o il "forcing". Per evitare questi problemi, la routine di controllo usa un'architettura a scansione di I/O, dove l'accesso all'hardware fisico avviene una sola volta per ogni iterazione del loop. I valori di input e output sono immagazzinati in memoria in un costrutto noto come tabella di memoria degli I/O e il codice di controllo accede allo spazio di memoria anziché all'hardware diretto. Questa architettura offre numerosi benefici:

- Astrazione degli I/O, per consentire il riutilizzo di subVI e funzioni (no hard coding degli I/O)
- Basso overhead
- Funzionamento deterministico
- Facile transizione fra piattaforme hardware
- Supporto per la simulazione
- Supporto per interlocks e "forcing"
- Eliminazione del rischio legato al cambio degli I/O durante l'esecuzione della logica

TABELLA DI MEMORIA DEGLI I/O

La tabella di memoria degli I/O serve a passare i dati fra il loop di scansione degli I/O ed altri loop o task paralleli di controllo o misura. Vi sono numerosi metodi per la comunicazione fra i loop, ma uno dei più semplici è quello di usare una variabile globale. Per rendere semplice l'aggiornamento del programma, utilizzeremo un cluster basato su un controllo custom strict type def. Con un controllo strict type def, qualsiasi cambiamento apportato al controllo si propaga automaticamente all'intero programma. Ciò vi permette di aggiungere o rinominare facilmente gli I/O senza la necessità di ricodificare il programma.



SCANSIONE DEGLI I/O

Il loop di controllo principale sarà il loop con la massima priorità, quindi sarà eseguito per primo ad ogni ciclo e presenterà due fasi, che si attiveranno ripetutamente alla velocità desiderata del loop di controllo:

1. Scrittura degli output - La tabella di memoria degli I/O viene letta ed i moduli di output sono aggiornati.
2. Lettura degli input - I moduli di input vengono letti ed i valori vengono scritti nella tabella di memoria degli I/O.

Nota: Benché non sia intuitivo, scrivere prima sugli output aiuta ad assicurare il funzionamento deterministico e permette di ottenere un programma più pulito per la gestione degli errori e la transizione a uno stato di shutdown.

TASK LOGICI DELL'UTENTE

La logica utente è la logica specifica alla macchina che definisce l'applicazione di controllo. In molti casi, si basa su una macchina a stati per gestire il controllo di macchine complesse con stati multipli. Un paragrafo successivo spiegherà come utilizzare le macchine a stati per progettare la logica utente.Per essere eseguita nell'architettura di controllo, la logica utente deve:

- Essere eseguita in meno tempo del loop temporizzato
- Accedere agli I/O attraverso la tabella di memoria degli I/O anziché attraverso letture o scritture dirette degli I/O
- Non usare "while loop" se non per mantenere informazioni di stato negli shift register
- Non usare "for loop" se non negli algoritmi
- Non usare "wait" (attese), ma funzioni di temporizzazione o "Tick Count" per la logica di timing
- Non eseguire operazioni non deterministiche, di logging o su forme d'onda

La logica utente può:

- Includere operazioni single point come PID o analisi punto-punto
- Usare una macchina a stati per strutturare il codice

Ogni task logico verrà strutturato utilizzando un Timed Loop. La priorità del Timed Loop deve essere inferiore a quella del loop di scansione degli I/O e la temporizzazione dovrebbe essere un multiplo intero della velocità del loop di scansione degli I/O. Il task logico conterà tre fasi:

1. Copiare la tabella di memoria degli I/O nel cluster locale - La tabella di memoria degli I/O viene letta e un cluster locale (strict type def) viene scritto. Un cluster locale permette di offrire scalabilità per task multipli e di utilizzare il modulo LabVIEW StateChart.
2. Eseguire la logica - Gli I/O vengono letti e scritti nei cluster locali per ingressi e uscite
3. Copiare il cluster locale nella tabella di memoria degli I/O - Il cluster locale degli output (strict type def) viene letto e scritto nella tabella di memoria degli I/O. Anche in questo caso, un cluster locale permette di offrire scalabilità per task multipli e di utilizzare il modulo LabVIEW StateChart.

ESEMPIO DI ARCHITETTURA DI BASE DEI CONTROLLORI IN LABVIEW

Per mostrare questa architettura di controllo, costruiremo un'applicazione di controllo PID elementare. Questa semplice applicazione controlla una camera climatica in modo tale da mantenervi una temperatura di 350 gradi. Essa prevede un ingresso analogico da termocoppia, un'uscita analogica (0-10V) collegata all'ingresso di un elemento riscaldante e un algoritmo PID per il controllo. Questa applicazione è particolarmente semplice ed è utilizzata qui solo per spiegare i componenti dell'architettura senza aggiungere la complessità di un controllo intricato.

TABELLA DI MEMORIA DEGLI I/O

Per creare la tabella di memoria degli I/O:

1. Create un controllo custom strict type def chiamato *IO Cluster.ctl*. Nel controllo custom aggiungete un cluster e create due controlli, I*nput_TC* e *Output_Heater*.
2. Create una variabile globale chiamata *IO Table Global.vi* contenente *IO Cluster.ctl*.

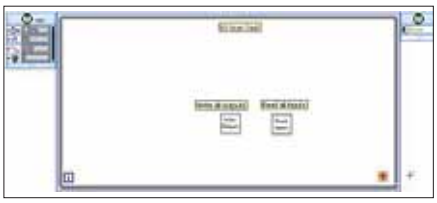
Suggerimento: L'uso di un controllo strict type def custom permette il semplice aggiornamento del codice tramite il LabVIEW project.

SCANSIONE DEGLI I/O

1. Aprite un top level vi.
2. Inserite un Time Loop sul diagram-

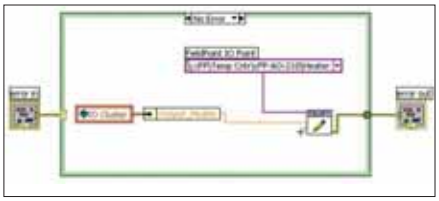
ma e configurate il rate in funzione del tempo-ciclo richiesto dalla vostra applicazione.

3. Sullo schema, create due subVI:



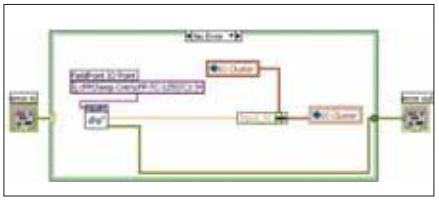
Write Outputs e *Read Inputs*.

4. Aprite *Write Outputs*. Sullo schema a blocchi, inserite la variabile globale *IO Table Global.vi* e la logica per scrivere gli output fisici. Questo esempio è programmato per FieldPoint, quindi utilizza un *FP Write VI*. La stessa architettura potrebbe essere facilmente utilizzata per programmare CompactRIO,



PXI o DAQ.

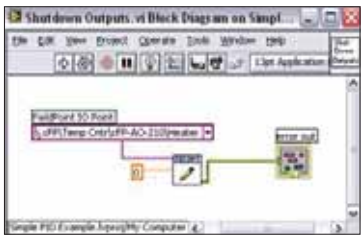
5. *Aprite Read Inputs*. Sullo schema a blocchi, inserite la variabile globale *IO Table Global.vi* e la logica per gli



input fisici.

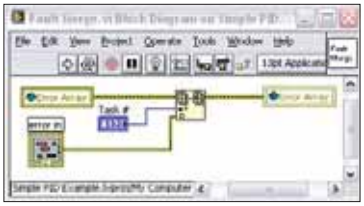
6. Salvate e chiudete i due subVI. Sul *top level* vi disponete una struttura case intorno ai subVI *Write Outputs* e *Read Inputs*. Nel caso false create un nuovo subVI chiamato *Shutdown Outputs*, dove entrerete nel caso in cui il controllore debba essere fermato.

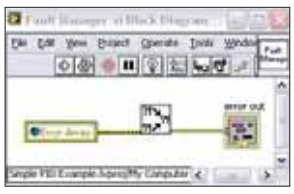
7. Aprite *Shutdown Outputs.vi* e aggiungete la logica necessaria per scrivere valori di sicurezza sulle uscite alla chiusura e salvate il subVI. Invece di scrivere i valori d'uscita dalla tabella di memoria degli I/O, scriverete i valori d'uscita di sicurezza. In questo caso, il riscaldatore dovrebbe spegnersi se si esce dal programma, quindi il valore *Output_Heater* viene sostituito



con una costante zero.

8. A livello di *top level vi* dovete ora aggiungere della logica per decidere quando eseguire la chiusura. Questa chiusura ferma il controllore. Nella maggior parte dei casi, ciò dovrebbe avvenire solo al verificarsi di un errore o se il controllore deve essere riprogrammato. Per questo esempio eseguiremo la chiusura in presenza di un errore. Abbiamo creato un cluster di errore globale che ci permette di raccogliere eventuali errori da più task paralleli. Abbiamo creato due subVI per gestire e gli errori: *Fault Manager.vi*

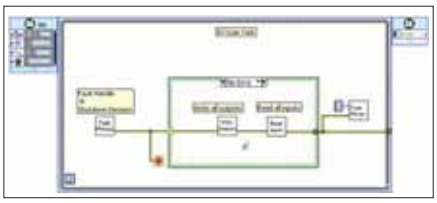




e il *Fault Merge.vi*.

Nota: Il comportamento in fase di chiusura è definito dal programmatore. Potete decidere di ignorare certi errori o di riprovare a eseguire le operazioni prima della chiusura.

9. Nel *top level* vi utilizzate il *Fault Manager.vi* e il *Fault Merge.vi* per rilevare gli errori e controllare la struttura case



per effettuare lo shutdown in caso di errore. Notate che per ogni task dobbiamo assegnare un Task #: utilizzeremo '0' per la routine di inizializzazione e '1' per il Task di scansione degli I/O.

ROUTINE DI INIZIALIZZAZIONE E CHIUSURA

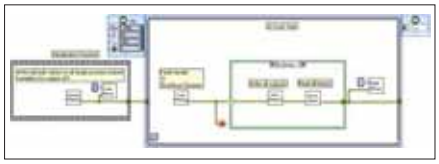
1. Ora dobbiamo aggiungere la routine di inizializzazione e una routine di chiusura. La routine di inizializzazione deve configurare il controllore in modo che sia pronto per eseguire qualsiasi logica e la routine di chiusura deve eseguire qualsiasi azione legata alla fase di shutdown.

2. La routine di inizializzazione imposterà i valori di default per gli output ed eseguirà l'error trapping (gestione dell'errore). Ricordate che il loop di controllo principale inizia scrivendo le uscite sull'hardware, quindi dobbiamo caricare i valori di startup per le uscite nella tabella di memoria degli I/O.

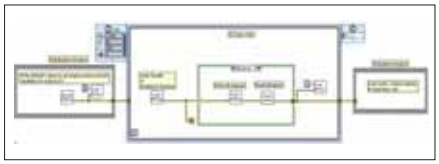
3. Create un subVI chiamato *Default Outputs.vi* e scrivete del codice per impostare la tabella di memoria

degli I/O con i valori di default. Vogliamo che il comportamento di default dell'uscita del riscaldatore sia off, quindi creeremo una costante '0' e la collegheremo alla variabile condivisa *Output_Heater*.

4. Aggiungete l'error trapping con il *Fault Merge.vi* ed usate il Task #0.



5. Sul *top level* vi aggiungete una struttura a sequenza (sequence structure) a sinistra dell'I/O Scan Task. Collegate il cluster di errore da *Default Outputs.vi* al bordo del Timed Loop per assicurare l'ordine di esecuzione.
6. Per la routine di chiusura aggiungete una *sequence structure* a destra



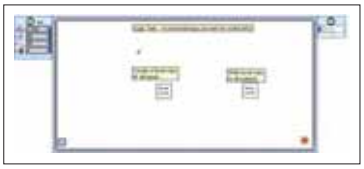
dell'I/O Scan Task e collegate il cluster di errore al bordo della struttura. Aggiungete l'eventuale logica di chiusura all'interno della struttura. In questo esempio non abbiamo routine di chiusura.

7. Ora avete una routine completa di inizializzazione, scansione degli I/O e chiusura. A questo punto dovete aggiungere i vostri task logici.

USER LOGIC TASK

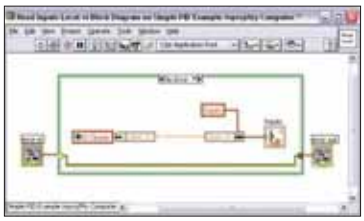
Ogni task logico verrà strutturato utilizzando un Timed Loop con una priorità inferiore a quella del loop di scansione degli I/O e un timing pari a un multiplo intero della velocità del loop di scansione degli I/O. Per scalare la vostra applicazione e per consentirvi di eseguire i loop a velocità diverse rispetto al task di scansione degli I/O, dovete creare una versione locale degli I/O. A tale scopo, copiate la tabella di memoria degli I/O in due cluster locali (input e output) che sono strict type def.

1. In primo luogo viene creato un controllo strict type def custom chiamato Input Cluster.ctl. Nel controllo custom aggiungiamo un cluster e creiamo un controllo chiamato Input_TC.
2. Viene quindi creato un controllo strict type def custom chiamato Output Cluster.ctl. Nel controllo custom aggiungiamo un cluster e creiamo un controllo chiamato Output_Heater. Suggerimento: Se utilizzate lo



StateChart Module, esso creerà automaticamente uno strict type def Input Cluster.ctl e Output Cluster.ctl.

3. Sullo schema create due subVI, uno



chiamato Read Inputs Local e uno chiamato Write Outputs Local.

4. Aprite *Read Inputs Local*. Sullo schema a blocchi, inserite la *I/O*

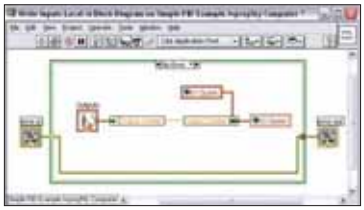
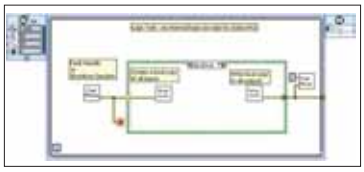


Table Global e la logica per leggere gli ingressi e scriverli nel cluster.

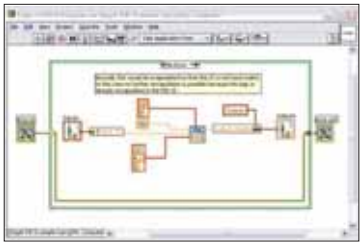
5. Aprite *Write Outputs Local*. Sullo schema a blocchi, inserite la I/O *Table Global* e la logica per leggere



gli ingressi e scriverli nel cluster.

6. Salvate e chiudete i due subVI. Sul *top level* vi disponete una struttura case e la gestione degli errori. Notate che per *Fault Merge.vi* questo è il Task #2.
7. Ora create un subVI chiamato *Logic.vi*. In questo vi cablerete in ingresso e in uscita i cluster di errore e metterete la vostra logica. Ricordate che la logica deve:
   - Essere eseguita in meno tempo di quello richiesto dal Timed Loop
   - Non accedere direttamente all'hardware per letture e scritture degli I/O
   - Non usare *while loop* se non per mantenere informazioni di stato negli shift register
   - Non usare *for loop* se non negli algoritmi
   - Non usare *wait* (attese), ma funzioni di temporizzazione o "Tick Count" per la logica di timing
   - Non eseguire operazioni non deterministiche, di logging o su forme d'onda

8. Inserite un vi PID sullo schema a blocchi e collegate le costanti per impostare l'intervallo d'uscita a [10, 0], i guadagni del PID a [10, 0,1, 0] e il setpoint a 350. I guadagni ed il setpoint del PID avrebbero potuto anche essere cablati ad elementi della tabella di memoria degli I/O e pubblicati su un terminale operatore per modificarli successivamente. Collegate il valore

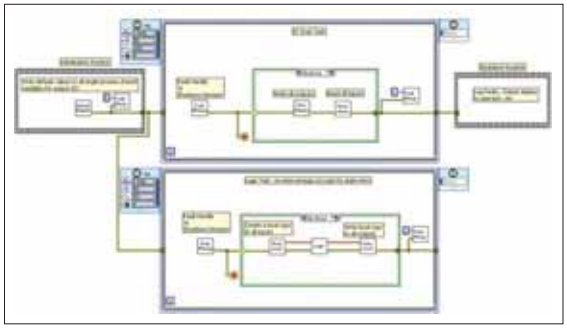


Input_TC al terminale *Process Variable* e collegate il valore *Output_Heater* al terminale *Output*. Aggiungete gli appropriati componenti per la gestione degli errori.

Suggerimento: Si possono utilizzare StateChart per applicazioni di controllo sequenziale, controllo batch o controllo macchina.

9. Sul *top level* vi collegate il cluster di errore dalla routine di inizializzazione sul bordo del Timed Loop, quindi salvate ed eseguite l'applicazione.

Ora possiamo eseguire il programma di controllo della temperatura. Esso ha procedure di avviamento e chiusura, un'architettura a scansione, subVI



riutilizzabili e la gestione degli errori. Questa è l'architettura fondamentale per il controllo di macchine utilizzata nel presente articolo.

Esamineremo ora un esempio più complesso che evidenzia i benefici di questo tipo di architettura.

MACCHINE A STATI

La maggior parte delle applicazioni di controllo reali è molto più complessa della semplice applicazione PID costruita nell'esempio precedente.

Benché si possa utilizzare il PID per controllare un setpoint di temperatura, esso sarà solo un piccolo componente nell'applicazione.

Un esempio un po' più realistico può essere quello di un reattore chimico. In questa applicazione immaginaria, il controllore deve:

1. Attendere dall'operatore un comando di partenza con la pressione di un pulsante

2. Misurare due flussi chimici in un serbatoio sulla base dell'uscita di un totalizzatore di flusso (due processi paralleli, uno per ciascun flusso chimico)

3. Dopo il riempimento del serbatoio, avviare un miscelatore ed aumentare la temperatura

4. Quando la temperatura ha raggiunto 200 gradi, il sistema disattiva i miscelatori e tiene costante la temperatura per 10 secondi

5. Pompare il contenuto in un serbatoio di contenimento

6. Tornare allo stato di attesa

Notate che, per semplicità, in questa applicazione le portate dei flussi chimici sono stato codificate a 850, la temperatura a 200 e il tempo a 10 secondi. In un'applicazione reale questi valori potrebbero essere caricati da una procedura o immessi da un operatore.

RICHIAMI SULLE MACCHINE A STATI

Il modo migliore per rappresentare questo tipo di sistema di controllo è quello di utilizzare una macchina a stati. Il modello di progettazione basato su macchina a stati è uno dei modelli di progettazione più utili e diffusi per LabVIEW; potete implementare qualsiasi algoritmo che possa essere esplicitamente descritto da un diagramma a stati o schema di flusso. Una macchina a stati normalmente illustra un algoritmo decisionale di media complessità, come una routine diagnostica o un controllore di processo.

Una macchina a stati, che si può definire con maggiore precisione *macchina a stati finiti*, consiste di un insieme di stati e di una funzione di transizione che mappa il passaggio allo stato successivo. Una macchina a stati completa dovrebbe essere progettata per eseguire azioni d'ingresso, in attività dello stato, o di uscita.

Poiché utilizziamo la macchina a stati all'interno della nostra più ampia architettura di controllo macchina, essa non può usare istruzioni di attesa né loop, se non per mantenere uno stato o eseguire algoritmi, come un *for loop* usato per la manipolazione di array.

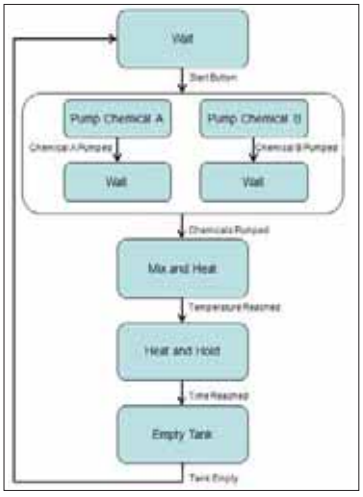
Usate la macchina a stati nelle applicazioni dove esistono stati distinti. Ogni stato può portare ad uno o più stati o terminare il flusso del processo. Una macchina a stati si basa sugli input dell'utente o su calcoli effettuati all'interno dello stato per determinare in quale stato andare successivamente. Molte applicazioni richiedono uno stato di inizializzazione, seguito da uno stato di default, dove è possibile eseguire molte azioni differenti.

Le azioni eseguite possono dipendere da input precedenti e correnti, nonché da altri stati.

Per eseguire le azioni di azzeramento viene utilizzato comunemente uno stato di chiusura.

ESEMPIO DI MACCHINA A STATI IN LABVIEW

Per costruire questa applicazione, il primo passo è quello di mappare la logica e i punti di I/O. Poiché questa applicazione implica una sequenza di

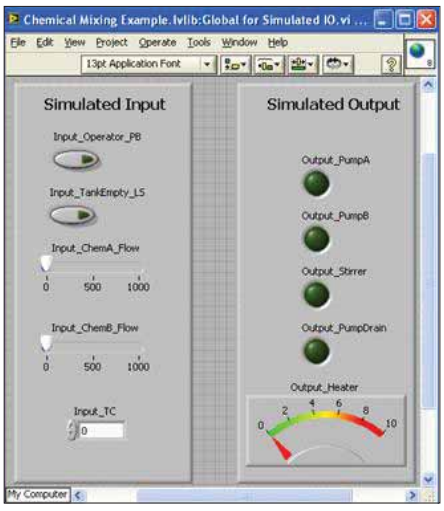


I/O Signals	I/O Name
Operator push button	Input_Operator_PB
Pump A	Output_PumpA
Pump B	Output_PumpB
Chemical A Flow	Input_ChemA_Flow
Chemical B Flow	Input_ChemB_Flow
Stirrer	Output_Stirrer
Heater	Output_Heater
Thermocouple	Input_TC
Drain Pump	Ouput_PumpDrain
Tank Empty Level Sensor	Input_TankEmpty_LS

passi, uno schema di flusso è un ottimo tool per pianificare l'applicazione stessa. Qui sotto sono riportati uno schema di flusso per l'applicazione e un elenco dei segnali di I/O.
Ogni stato in una macchina a stati esegue un'azione unica e richiama altri stati. Le transizioni fra gli stati dipendono dal verificarsi o meno di qualche condizione o sequenza.
Tradurre il diagramma di transizione di stato in uno schema a blocchi LabVIEW richiede i seguenti componenti:

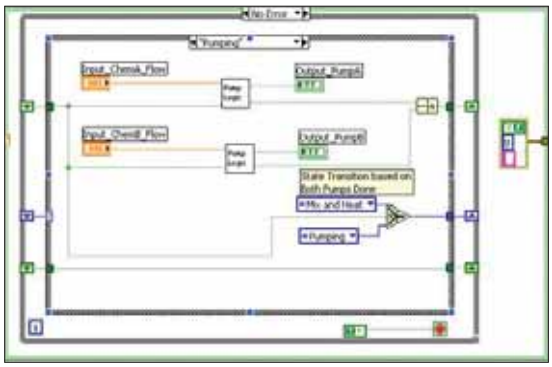
- Case structure - Contiene un caso per ogni stato e il codice per eseguire ciascuno stato
- Shift register (registro a scorrimento) - Contiene le informazioni di transizione fra gli stati
- State functionality code - Implementa la funzione dello stato
- Transition code - Determina lo stato successivo nella sequenza

Per questo esempio utilizzeremo I/O simulati invece di I/O fisici, in modo da potere testare la nostra logica. A tale scopo, utilizzeremo una variabile globale invece di VI di lettura/scrittura. Questo è un modo comodo per testare



la nostra logica con controlli e indicatori interattivi prima della distribuzione sull'hardware. La capacità di simulare facilmente I/O è un vantaggio di questa architettura.

Poiché uno stato nella nostra macchina presenta processi paralleli, costruiremo una seconda macchina a stati



per rappresentare la logica parallela e richiamarli in quello stato.
Usciremo dallo stato solo quando entrambi i processi paralleli saranno terminati.

LABVIEW STATECHART MODULE

L'esempio precedente è stato costruito utilizzando le strutture fondamentali di LabVIEW: cicli, sequenze e *strict type def enum*. Con LabVIEW 8.5 è nato un nuovo modulo, il LabVIEW StateChart Module. Si tratta di un nuovo editor di LabVIEW che vi permette di costruire rapidamente una logica completa per macchine a stati, combinando opportunamente diagrammi di stato multipli e VI di LabVIEW in applicazioni complesse. Inoltre, può essere eseguito sia su Windows sia su target real-time ed FPGA. Lo StateChart Module è particolarmente utile se dovete gestire più transizioni possibili da uno stato o se volete che eventi specifici attivino una transizione da stati multipli. Per esempio, sulla base di un comando fornito dall'operatore potreste volere abortire la produzione da qualsiasi stato. Lo StateChart Module vi permette di programmare facilmente il sistema in modo che esegua una transizione da qualsiasi stato e di dare priorità alle transizioni in modo che il comando di interruzione venga eseguito anche se altre transizioni vengono attivate nello stesso tempo. Tutte le funzionalità dello StateChart Module possono essere programmate in LabVIEW, ma il modulo StateChart migliora nettamente il tempo di sviluppo ed è molto più facile da leggere e correggere rispetto alle macchina a stati tradizionali scritte in LabVIEW. Si raccomanda quindi fortemente a chiunque abbia necessità di costruire applicazioni di controllo macchina.

MODIFICA DELL'ESEMPIO

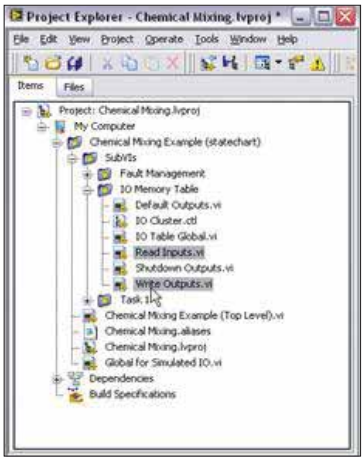
Il modo più semplice per iniziare questo progetto è quello di modificare l'esempio esistente. In questo paragrafo procederemo con la modifica del precedente esempio della miscelazione chimica, dove è stato utilizzato il diagramma di stato per costruire l'applicazione. Ci sono quattro passi principali:

1. Modificate I/O Scan Task per leggere e scrivere sugli I/O fisici della vostra applicazione.
2. Modificate la routine di inizializzazione per scrivere i valori di default dei vostri output fisici.
3. Modificate il Task 1 per mappare gli I/O. Cambierete lo StateChart, il *Read Inputs Local.vi* e il *Write Inputs Local.vi*.
4. Modificate/riscrivete lo StateChart per adattarlo alla vostra applicazione.

MODIFICA DI I/O SCAN TASK

Aprite *Chemical Mixing.lvproj*. Poiché la vostra applicazione utilizzerà I/O differenti, dovrete modificare I/O Scan Task per leggere e scrivere sui vostri I/O. Ricordate che gli I/O possono essere I/O fisici, I/O in rete o I/O simulati.

1. Aprite il controllo strict type custom *IO Cluster.ctl*. Esso definisce tutti gli



I/O del vostro sistema.

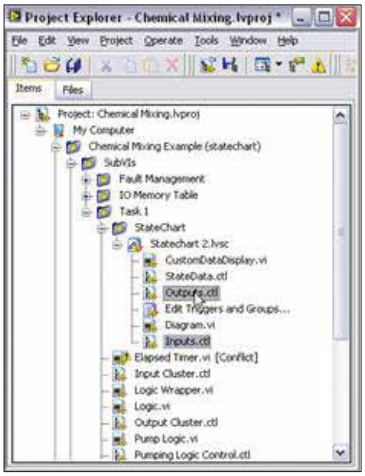
2. Cambiate i controlli nel cluster con quelli richiesti per la vostra applicazione. Salvate *IO Cluster.ctl*. LabVIEW aggiornerà automaticamente la tabella di *IO Global.vi* con queste modifiche.
3. Aprite *Write Outputs.vi* e *Read Inputs.vi*.
4. Modificate i VI in modo che scrivano e leggano i vostri I/O fisici invece degli I/O simulati.
5. Aprite *Shutdown Outputs.vi*. Modificate il VI in modo che scriva sui vostri I/O fisici invece degli I/O simulati.

MODIFICA DELLA ROUTINE DI INIZIALIZZAZIONE

Poiché la vostra applicazione utilizzerà I/O differenti, dovrete modificare la routine di inizializzazione per impostare i valori di default dei vostri output.

1. Aprite *Default Outputs.vi* e modificatelo per impostare i valori di default dei vostri output.

MODIFICA DEL TASK 1 PER MAPPARE GLI I/O



Ora avete un I/O Scan Task completo e routine di inizializzazione e chiusura. Ora dovete modificare la logica. Poiché ogni task logico crea una copia locale degli I/O per la sua esecuzione, dovete rimappare gli I/O

1. Sotto il folder StateChart aprite *Outputs.ctl e Inputs.ctl*. Modificateli per adattarli agli I/O della vostra applicazione.
2. Aggiornate *Write Outputs Local.vi* e *Read Outputs Local.vi* per leggere e scrivere i vostri I/O.

Modifica/riscrittura dello StateChart

Ora dovete inserire la vostra logica. Potete farlo modificando/riscrivendo lo StateChart in modo che esegua la logica specifica della vostra applicazione.

Potete scaricare gli esempi utilizzati in questo articolo al seguente link: ni.com/italian (info code: it3nch)

Nota sull'autore

Chris Washington è Product Manager di LabVIEW in National Instruments

SCELTI PER TE

Abbiamo scelto per te alcune risorse utili per approfondire le tua conoscenza di LabVIEW.

Forum
ILVG.it www.ilvg.it
LabVIEW www.ni.com/labviewzone
LAVA - LabVIEW Advanced Virtual Architects http://forums.lavag.org/home.html

Community
DevZone www.zone.ni.com
Community on ni.com http://community.ni.com/
Mindstorm NXT http://www.ni.com/academic/mindstorms/community.htm

Contenuti
LabVIEW on ni.com www.ni.com/labview
LabVIEW on ni.com/italy www.ni.com/labview/i
LabVIEW Jobs http://www.labviewjobs.com/
The VI Road Show http://viroadshow.blogspot.com/