

INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

---

# Využití CompactRIO systémů pro radiace aplikace

Učební texty ke kurzu

---

Přednášející:

Ing. Mgr. Márk Jónás (ANV s.r.o., Bratislava)

Ing. Zuzana Petráková (ANV s.r.o., Bratislava)

Mgr. Silvia Mókosová (ANV s.r.o., Bratislava)

Ing. Gregor Izrael, PhD. (ANV s.r.o., Bratislava)

Datum:

26.-27. 9. 2012

Centrum pro rozvoj výzkumu pokročilých řídicích a senzorických technologií CZ.1.07/2.3.00/09.0031

TENTO STUDIJNÍ MATERIÁL JE SPOLUFINANCOVÁN EVROPSKÝM SOCIÁLNÍM  
FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY



## OBSAH

Obsah .....	1
1. Návrhové techniky a vzory .....	3
2. Používanie premenných .....	10
3. Návrh používateľského rozhrania s dotykovou obrazovkou .....	24
4. Pridanie počítačového videnia a rozpoznávania obrazu do CompactRIO systému pre meracie a riadiace aplikácie.....	30
5. Riadenie pohonov pomocou CompactRIO systémov .....	46
6. Nasadenie a replikácia systémov.....	71



## 1. NÁVRHOVÉ TECHNIKY A VZORY

Prvým krokom pri vývoji LabVIEW projektu je prieskum architektúr ktoré existujú v LabVIEW. Architektúry sú podstatné pre tvorbu úspešného návrhu softvéru. Väčšina bežných architektúr je zoskupená do návrhových vzorov.

Čím viac je návrhový vzor akceptovaný, tým ľahšie rozoznateľné že bol použitý. Toto rozpoznanie pomôže vám a iným vývojárom interpretovať a modifikovať VI ktoré sú založené na návrhových vzoroch.

Pre LabVIEW VI existuje veľký počet návrhových vzorov. Väčšina aplikácií používa aspoň jeden návrhový vzor. V tomto kurze sa venujeme návrhovému vzoru Stavový stroj (State machine).

### **Sekvenčné programovanie**

Veľký počet VI ktoré napíšete v LabVIEW vykonáva úlohy sekvenčne. Spôsob naprogramovania sekvenčných úloh môže byť veľmi rozdielny.

V LabVIEW môžete vykonať sekvenčné úlohy pomocou vytvorenia subVI pre každú úlohu, a následného prepojenia subVI pomocou error clustrov v poradí akom sa majú úlohy vykonať.

Môžete použiť Sequence štruktúru na určenie poradia vykonávania operácií v blokovom diagrame. Sequence štruktúra obsahuje jeden alebo viac subdiagramov, alebo rámcov, ktoré sa vykonajú sekvenčne; rámec nezačne vykonávanie kódu skôr ako sa ukončí predošlý rámec.

K využitiu výhod vnútorného paralelizmu v LabVIEW sa vyhnite používaniu Sequence štruktúry v neopodstatnených prípadoch. Sequence štruktúry garantujú poradie vykonávania, ale zamedzia paralelnému behu operácií.

Druhým negatívnym javom použitia Sequence štruktúry je fakt, že vykonávanie kódu nemôžete zastaviť pokým sa celá štruktúra nevykonala.

Najlepšou možnosťou ako naprogramovať toto VI je zahrnúť funkciu One Button Dialog do Case štruktúry, a napojiť error cluster na terminál podmienky.

Používajte Sequence štruktúry obozretne, nakoľko nepodporujú kontrolu chýb, a ich beh sa neukončí v prípade výskytu chyby. Radšej zvolte správny dátový tok na riadenie poradia vykonávania kódu ako Sequence štruktúru.

## Programovanie pomocou stavov

Hoci Sequence štruktúra a sekvenčne prepojené subVI splnia svoj účel, často potrebujete zložitejšiu architektúru na vývoj algoritmov.

- Čo v prípade ak musíte zmeniť poradie vykonávania sekvencie?
- Čo v prípade ak musíte vykonať časť sekvencie viackrát než iné časti?
- Čo v prípade ak niektoré položky v sekvencii sa majú vykonať iba ak sú splnené určité podmienky?
- Čo ak potrebujete ukončiť program okamžite, namiesto čakania do konca sekvencie?

Aj keď vaša aplikácia nemusí spĺňať ani jednu z vyššie uvedených požiadaviek, je možné, že program bude musieť byť modifikovaný v budúcnosti. Z týchto dôvodov je architektúra programovania pomocou stavov dobrou voľbou, aj keď by bolo sekvenčné programovanie dostatočné.

## Stavové stroje

Návrhový vzor stavový automat je štandardný a veľmi užitočný návrhový vzor pre LabVIEW. Tento návrhový vzor môžete použiť na implementáciu ľubovoľného algoritmu ktorý sa dá explicitne opísať pomocou stavového diagramu alebo diagramu toku údajov. Stavový automat vo všeobecnosti implementuje stredne komplexné rozhodovacie algoritmy, ako napríklad diagnostické rutiny alebo monitoring procesov.

*Stavový stroj*, presnejšie stavový stroj s konečným počtom stavov, sa skladá zo skupiny stavov a z prechodových funkcií. Stavový stroj s konečným počtom stavov má viacero variácií. Dva štandardné stavové stroje sú Mealyho stavový stroj a Moorov stavový stroj. Mealyho stavový stroj vykoná nejaký úkon pri každom prechode. Moorov stavový stroj vykoná špecifický úkon pre každý stav v diagrame stavov. Šablóna návrhového vzoru stavový automat v LabVIEW implementuje ľubovoľný algoritmus opísaný pomocou Moorovho stroja.

## Použitie návrhového vzoru Stavový stroj

Stavové stroje použite v aplikáciách v ktorých existujú rozlíšiteľné stavy. Každý stav môže viesť k jednému alebo k viacerým následným stavom, resp. k

ukončení procesného toku. Stavový stroj sa spolieha na vstup od používateľa, resp. vnútro stavovú kalkuláciu na určenie nasledujúceho stavu. Viacero aplikácií požaduje stav inicializácie, nasledovaný štandardným stavom, kde sa môže vykonať viacero rôznych akcií. Vykonané akcie závisia od predošlých a aktuálnych vstupov a stavov. Stav ukončenia obyčajne vykoná dealokačné, uvoľňovacie akcie.

Stavový automaty sú bežne používané na tvorbu používateľských rozhraní. V prípade používateľského rozhrania, rôzne akcie používateľa posúvajú používateľské rozhranie do rôznych segmentov spracovania. Každý segment spracovania je stav v stavovom stroji. Každý segment môže viesť k inému segmentu kvôli ďalšiemu spracovaniu alebo čakať na ďalšiu akciu používateľa. Pri takejto implementácii používateľského rozhrania, stavový stroj neustále monitoruje možný výskyt akcie vykonanej používateľom (polling).

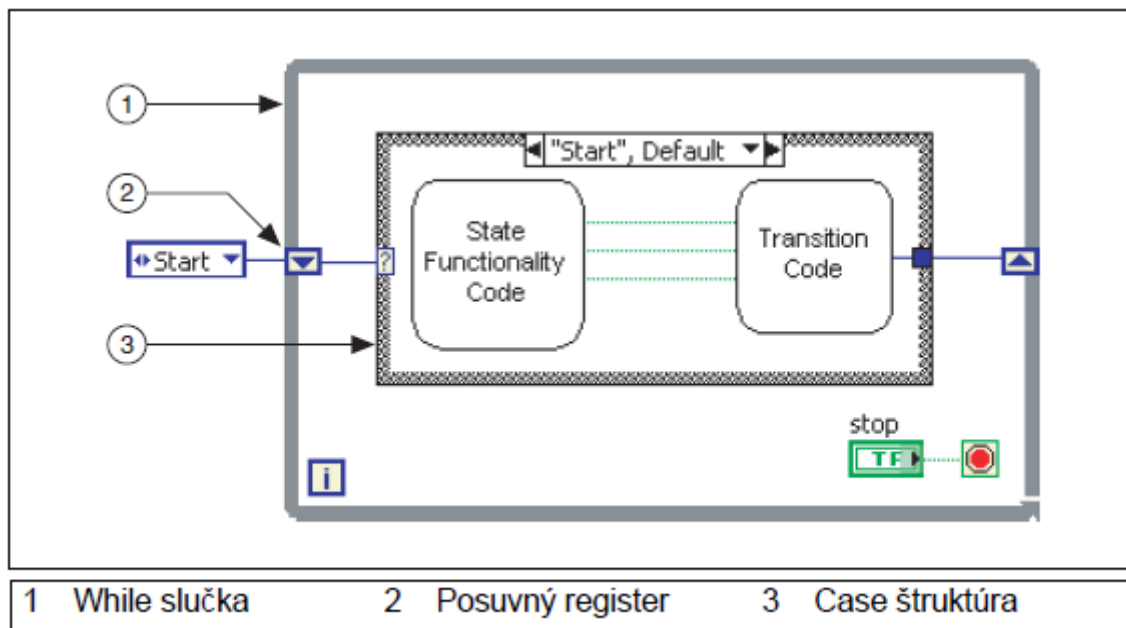
Procesné testovanie je ďalšou bežnou aplikáciu návrhového vzoru stavový automat. Pri procesnom testovaní, stav reprezentuje časť procesu. V závislosti od výsledkov testov v jednotlivých stavoch, ďalšie stavy môžu byť volané. Celý tok sa môže vykonávať kontinuálne, výsledkom je hĺbková analýza testovaného procesu.

Výhoda použitia stavového stroja je tá, že návrhový vzor je ľahko implementovateľný ako náhle je vytvorený stavový diagram.

### **Infraštruktúra stavového stroja**

Pri konverzii zo stavového diagramu do LabVIEW blokového diagramu sú potrebné nasledovné komponenty:

- **While slučka**—neustále vykonáva jednotlivé stavy
- **Case štruktúra**—obsahuje prípad pre každý stav. V prípade sa nachádza kód ktorý sa vykoná pre daný stav.
- **Posuvný register**—obsahuje informáciu o prechode medzi stavmi
- **Funkcionálny kód stavu**—implementuje činnosť stavu
- **Kód prechodu**—určí nasledujúci stav



Základná infraštruktúra stavového stroja v LabVIEW

Figure 1.1: Základná infraštruktúra stavového stroja v LabVIEW

Tok stavového diagramu je implementovaný pomocou While slučky. Individuálne stavy sú reprezentované jednotlivými prípadmi v Case štruktúre. Posuvný register vo While slučke udržiava informáciu o aktuálnom stave a je vstupom do terminálu podmienky Case štruktúry.

### Riadenie stavových strojov

Najlepšou metódou pre riadenie inicializácie a presunu medzi stavmi stavového stroja je použitie dátového typu enum. Dátový typ enum je štandardne používaný pre vstup do terminálu podmienky Case štruktúry. Avšak, ak používateľ zmení položku v enum dátovom type, spojenia ktoré sú použité pri kópiách tohto enumu budú nefunkčné. To tvorí prekážku pri implementácii stavového stroja pomocou enum. Jedným z riešení tohto problému je vytvoriť typovú definíciu enum kontrolky. Tým pádom sa zabezpečí, aby sa všetky kópie automaticky zaktualizovali ak pridáte alebo odoberiete stav.

### Presun medzi stavmi stavového stroja

Existuje viacero spôsobov na riadenie Case štruktúry v stavovom stroji. Vyberte si metódu ktorá je najvhodnejšia pre danú funkcionality a zložitosť stavového stroja. Najbežnejšie používanou ľahko použiteľnou metódou na implementáciu presunov medzi stavmi v stavovom stroji je implementácia pomocou jednej Case štruktúry (single Case structure transition code), ktorá sa môže použiť na

presun medzi ľubovoľným počtom stavov. Táto architektúra poskytuje najviac škálovateľnú, čitateľnú a udržateľnú architektúru stavového stroja. Ostatné metódy môžu byť užitočné v špecifických situáciách, a je dôležité aby ste sa s nimi oboznámili.

### Predvolený presun

V prípade predvoleného presunu nie je potrebný žiadny rozhodovací algoritmus, nakoľko nasledovný stav je pevne daný.

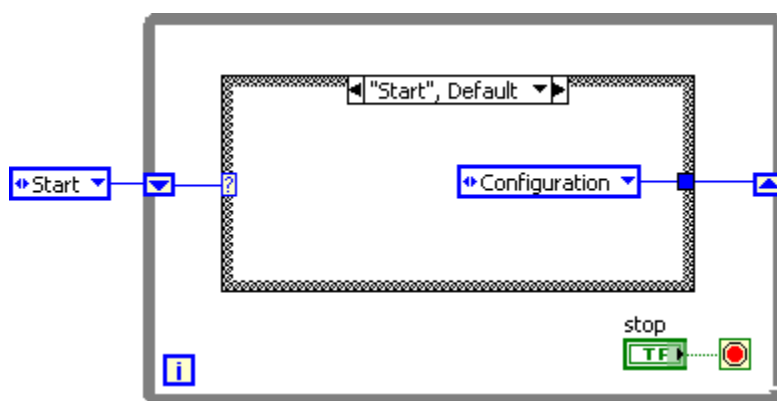


Figure 1.2: Predvolený presun

### Presun medzi dvoma stavmi

Nasledovná metóda zahŕňa rozhodovanie o presune medzi dvoma stavmi. Existuje viacero štandardných vzorov na uskutočnenie tejto úlohy.

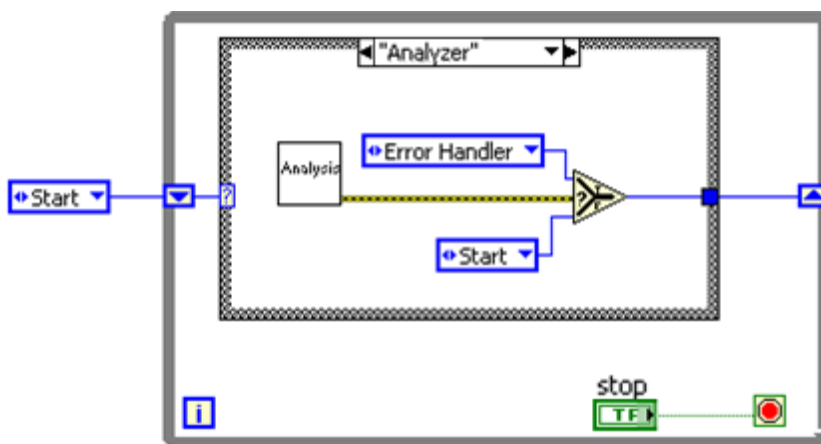


Figure 1.3: Použitie funkcie Select

Táto metóda funguje dobre, ak viete, že z daného stavu môže nastať presun iba do ďalších dvoch stavov. Zároveň, použitie tejto metódy limituje rozšíriteľnosť aplikácie. Ak potrebujete modifikovať stav rozšírením možných presunov na viac ako dva, toto riešenie by bolo nedostačujúce a vyžadovalo by značnú modifikáciu v kóde.

### Presun medzi dvoma a viacerými stavmi

Škálovateľnejšiu architektúru vytvoríte použitím niektorej metódy z nasledovných možností.

**Case štruktúra**—použite Case štruktúru namiesto Select funkcie na určenie nasledovného stavu.

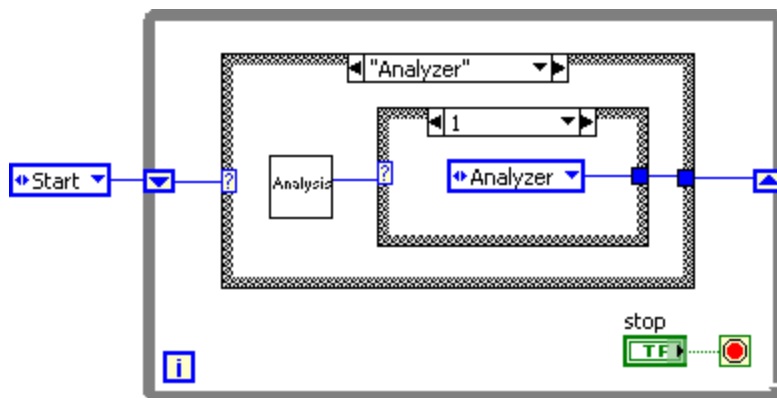


Figure 1.4: Použitie Case štruktúry pri určení nasledujúceho stavu

Jednou z výhod použitia Case štruktúry je dobrá čitateľnosť kódu. Takýto kód je jednoduché čítať a pochopiť, nakoľko každý prípad v Case štruktúre korešponduje niektorej položke v enume. Zároveň, Case štruktúra je rozšíriteľná. Nárastom aplikácie, máte možnosť pridať

dodatočné opcie na presun medzi stavmi pomocou pridania nových prípadov do Case štruktúry. Nevýhodou použitia Case štruktúry je to, že iba časť kódu je viditeľná naraz. Je to vlastnosť Case štruktúry, vďaka ktorej nie je viditeľná celá funkcionálna kódu prenosového kódu na jeden pohľad.

- **Pole presunov**—ak potrebujete, aby kód bol viditeľnejší ako to umožňuje Case štruktúra, môžete vytvoriť pole presunov všetkých možností.

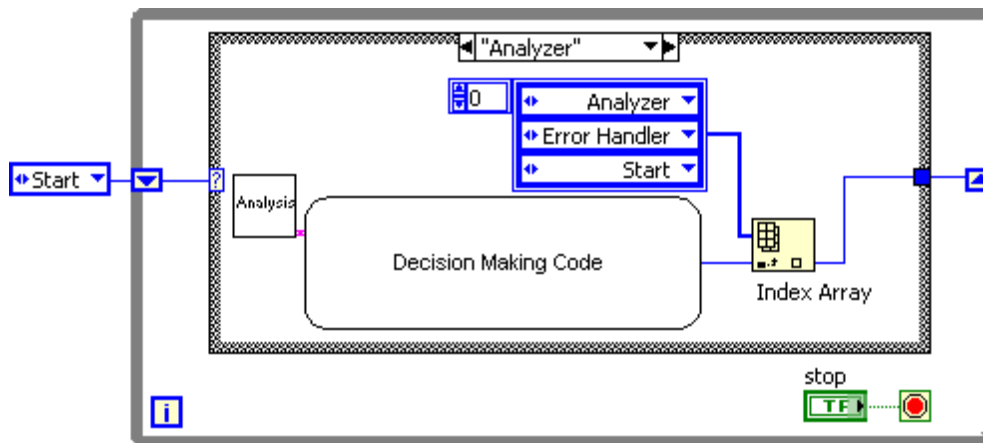


Figure 1.5: Použitie pola presunov pri určení nasledujúceho stavu

V tomto príklade, výstupom z rozhodovacieho algoritmu je index ktorý určí ďalší stav. Napríklad, ak ďalším stavom má byť stav Error Handler, výstupom z rozhodovacieho algoritmu bude hodnota 1, ktorá sa použije pri indexácii pomocou funkcie Index Array. Tento návrhový vzor robí kód rozšíriteľným a ľahko čitateľným. Jediná nevýhoda použitia tohto vzoru je, že musíte byť pozorný pri vývoji rozhodovacieho algoritmu, nakoľko pole je indexované od nuly.

## 2. POUŽÍVANIE PREMENNÝCH

V tejto lekcii sa naučíte ako použiť premenné na prenos dát medzi viacerými slučkami a VI. Taktiež sa dozviete o programovacích problémoch ktoré použitie premenných prináša, ako aj o možnostiach riešenia týchto výziev.

### Paralelizmus

V tomto kurze sa paralelizmus vzťahuje na vykonávanie viacerých úloh súčasne. Zamyslite sa nad nasledovným príkladom, kde zobrazujete dva sínusové priebehy s rôznymi frekvenciami. Využitím paralelizmu, umiestnite jednej priebeh do jednej a druhý do druhej slučky.

Výzvou pri programovaní paralelných úloh je prenos dát medzi viacerými slučkami bez vytvorenia dátovej závislosti. Napríklad, ak preniesiete dáte pomocou spojenia, zrušíte paralelizmus slučiek. Pri príklade dvoch sínusových priebehov je užitočné, ak máte spoločný stop prepínač na ukončenie behu slučiek

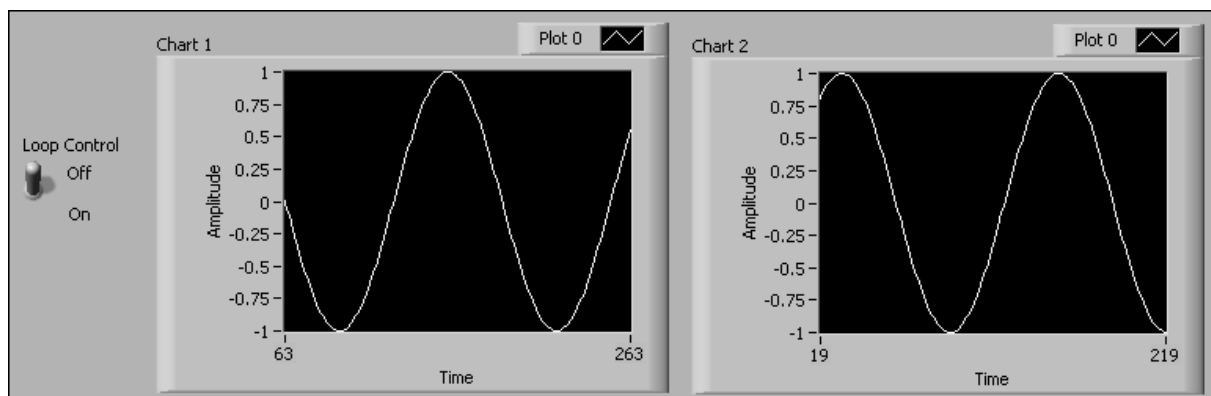


Figure 2.1: Čelný panel paralelných slučiek

Všimnite si, čo sa stane ak sa pokúsite zdieľať dáta medzi paralelnými slučkami pomocou spojenia s použitím dvoch rôznych metód.

### Metóda 1 (nesprávna)

Umiestnite **Loop Control** terminál mimo slučiek a pripojte ho k obom podmieňovacím terminálom. **Loop Control** terminál poskytuje vstupné dáta do oboch slučiek, tým pádom je vyčítaný iba raz, a to predtým ako dôjde k vykonaniu While slučiek. Ak hodnota False je vstupom do slučiek, While slučky pobežia do nekonečna. Vypnutím prepínača neukončíte beh VI, nakoľko prepínač (terminál) nie je vyčítaný pri iteráciách slučky.

## Metóda 2 (nesprávna)

Presuňte terminál **Loop Control** do slučky Loop 1, ako je to zobrazené na nasledovnom blokovom diagrame. Terminál je vyčítaný pri každej iterácii slučky Loop 1. Slučka Loop 1 sa ukončí riadne, avšak slučka Loop 2 sa nezačne vykonávať pokiaľ nemá dostupné dáta na všetkých vstupoch. Slučka Loop 1 nepustí dáta na výstup pokiaľ sa beh slučky neukončí, tým pádom slučka Loop 2 musí čakať, hodnota terminálu **Loop Control** sa preniesie až po ukončení slučky Loop 1. Z toho dôvodu sa slučky nebudú vykonávať súčasne. Zároveň, slučka Loop 2 vykoná iba jednu iteráciu, nakoľko vstupom do podmieňovacieho terminálu je hodnota True z prepínača **Loop Control** zo slučky Loop 1.

## Metóda 3 (riešenie)

Ak by ste vedeli čítať a zapisovať ukončovaciu podmienku do súboru, zrušili by ste závislosť slučiek na toku dát, nakoľko každá slučka môže nezávisle pristupovať k súboru. Avšak čítanie a zapisovanie do súborov môže byť časovo náročné. Iným spôsobom na uskutočnenie tejto úlohy je čítať a zapisovať do spoločného pamäťového miesta. V ďalších častiach tejto lekcie sa budeme zaoberať s metódami ktoré riešia tento problém.

## Premenné

V LabVIEW je poradie vykonávania prvkov v blokovom diagrame určené pomocou dátového toku namiesto poradia príkazov. Tým pádom, môžete vytvoriť blokový diagram ktorý má súbežné operácie. Napríklad, môžete súčasne vykonávať dve paralelné For slučky a zobrazovať výsledok na čelnom paneli, tak ako je to zobrazené na nasledujúcom blokovom diagrame. Avšak, ak použijete spojenia na prenos dát medzi paralelnými blokovými diagramami, tie už nebudú vykonávané paralelne. Paralelné blokové diagramy môžu byť dve paralelné slučky v tom istom blokovom diagrame bez závislosti na dátovom toku, alebo dve osobitné VI ktoré sú volané súčasne.

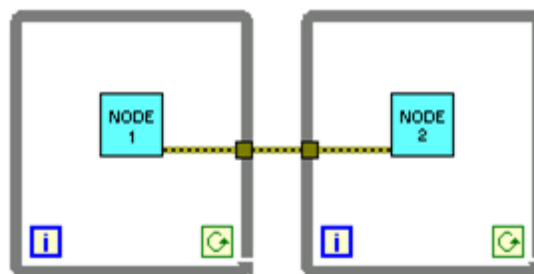


Figure 2.2: Dátová závislosť kvôli prepojeniu

Spojenie vytvorí dátovú závislosť, nakoľko druhá slučka sa nezačne vykonávať pokiaľ prvá slučka neskončí a dáta sa neprenesú cez tunel.

Odstráňte spojenie ak chcete, aby slučky bežali súčasne. Na prenos dát medzi subVI, použite inú techniku, ako napríklad premenné.

V LabVIEW, *premenné* sú prvky blokového diagramu ktoré umožnia prístup a uloženie dát do pamäťového miesta. Aktuálne pamäťové miesto sa mení v závislosti od typu premennej. Lokálne premenné uložia dáta do kontroliek a indikátorov čelného panela. Globálne premenné a single-process zdieľané premenné uložia dáta do špeciálnych repozitárov, ku ktorým môžete prísť z viacerých VI. Funkcionálne globálne premenné uložia dáta do posuvných registrov While slučiek. Nezávisle na mieste uloženia dát, všetky typy premenných umožnia obísť štandardný dátový tok pomocou prenosu dát z jedného miesta na druhé bez prepojenia týchto miest. Z tohto dôvodu sú premenné užitočné v paralelných architektúrach, zároveň ale majú aj určité nevýhody, ako napríklad výskyt chýb súbehu (race condition).

### **Používanie premenných v rámci jedného VI**

Lokálne premenné prenášajú dáta v rámci jedného VI.


V LabVIEW, čítate resp. zapisujete dáta do objektov čelného panelu pomocou terminálu na blokovom diagrame. Avšak, objekt na čelnom panelu disponuje iba jedným terminálom na blokovom diagrame. Vaša aplikácia však môže požadovať prístup k dátam z terminálu z viacerých miest.

Lokálne a globálne premenné prenášajú informácie medzi bodmi v aplikácii ktoré neviete prepojiť klasickým spôsobom. Použite lokálne premenné na prístup k objektom čelného panela z viac než jedného miesta v jednom VI. Použite globálne premenné na prístup a prenos dát medzi viacerými VI.

Použite uzol spätnej väzby na uloženie dát z predošlého behu VI alebo slučky.

### **Tvorba lokálnych premenných**

Lokálnu premennú vytvoríte kliknutím s pravým tlačidlom myši na existujúci terminál blokového diagramu a voľbou **Create»Local Variable** z kontextovej ponuky. Ikona lokálnej premennej sa zobrazí na blokovom diagrame.

 Môžete vybrať lokálnu premennú aj z palety **Functions** a vložiť ju do

blokového diagramu. Uzol lokálnej premennej ešte nie je asociovaná so žiadnou kontrolkou alebo indikátorom.

Asociáciu lokálnej premennej ku kontrolke alebo indikátoru vytvoríte pravým kliknutím na premennú a voľbou **Select Item** z kontextovej ponuky. Rozšírená kontextová ponuka zobrazí všetky objekty čelného panela ktoré majú vlastné návestia.

LabVIEW používa vlastné návestia na asociáciu lokálnych premenných s objektmi čelného panela, preto označte kontrolky a indikátory s opisnými návestiami.

### Čítanie a zapisovanie do premenných

Potom, čo ste si vytvorili premennú, môžete do nej zapisovať dáta resp. čítať z nej dáta. Štandardne, nová premenná je nastavená na zápis dát. Takáto premenná funguje ako indikátor. Keď zapíšete nové dáta do lokálnej alebo globálnej premennej, asociovaná kontrolka alebo indikátor na čelnom paneli sa zaktualizuje s novými dátami.

Takisto môžete nakonfigurovať premennú tak, aby bola zdrojom dát. Konfiguráciu fungovania premennej ako kontrolka nastavíte s pravým klikom na premennú a voľbou **Change To Read** z kontextovej ponuky. Keď sa tento uzol vykoná, VI načíta dáta z asociovanej kontrolky alebo indikátora.

Zmenu premennej na príjem dát z blokového diagramu vykonáte pomocou pravého kliknutia na premennú a voľbou **Change To Write** z kontextovej ponuky.

Premenné nastavené na príjem alebo poskytovanie na blokovom diagrame dát rozlíšite rovnakým spôsobom ako kontrolky a indikátory. Premenná na poskytovanie dát má hrubé ohraničenie, podobné kontrolke. Premenná na príjem dát má tenké ohraničenie, podobné indikátoru.

### Príklad na lokálnu premennú

V sekcii *Paralelizmus* tejto lekcie ste videli príklad na použitie paralelných slučiek. Čelný panel obsahoval jeden prepínač, ktorý ukončil generáciu dát v oboch slučkách. Dáta pre každý graf sú generované osobite v individuálnych While slučkách. Vďaka tomu sme mohli použiť rozdielne časovanie pre každú

slučku. Terminál Loop Control ukončil obe While slučky. V tomto príklade, kvôli ukončeniu behu naraz, obe slučky musia zdieľať prepínač.

Kvôli aktualizácii grafu podľa očakávaní, obe While slučky sa musia vykonávať paralelne. Prepojenie slučiek pomocou spojenia na prenos ukončovacej podmienky by zabránilo paralelnému behu slučiek.

Slučka Loop 2 číta z lokálnej premennej ktorá je asociovaná s prepínačom. Keď prepnete prepínač do stavu False na čelnom paneli, terminál prepínača v slučke Loop 1 zapíše False do podmieňovacieho terminálu slučky Loop 1. Slučka Loop 2 číta lokálnu premennú **Loop Control** zapíše False do podmieňovacieho terminálu slučky Loop 2. Tým pádom, paralelné slučky sa ukončia súčasne keď prepnete prepínač na čelnom paneli.

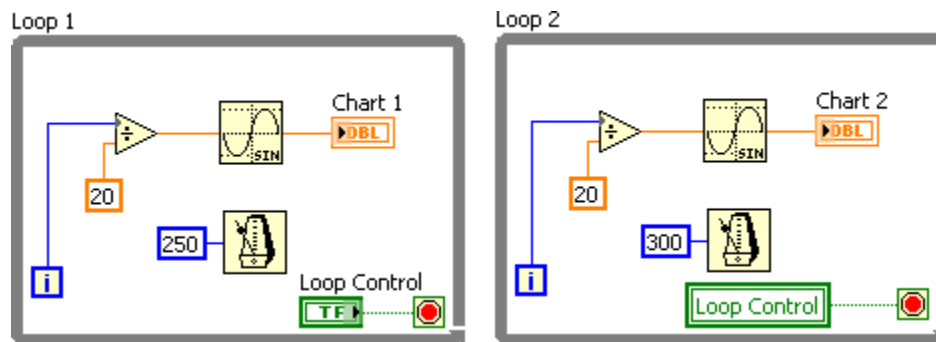


Figure 2.3: Lokálna premenná použitá na ukončenie paralelných slučiek

Pomocou lokálnych premenných môžete zapisovať a vyčítať dáta z kontroliek a indikátorov. Zapisovanie do lokálnej premennej je podobné ako prenos dát do iného terminálu. Avšak, do lokálnej premennej môžete zapisovať aj v prípade kontrolky, a vyčítať údaje aj v prípade indikátora. Tým pádom, pomocou lokálnej premennej môžete použiť objekt čelného panela ako vstup dát ale aj ako výstup dát.

Napríklad, ak používateľské rozhranie požaduje od používateľov prihlásenie, môžete vymazať políčka **Prihlasovacie meno** a **Heslo** pri každom novom prihlásení. Použite lokálnu premennú na vyčítanie hodnôt s kontroliek **Prihlasovacie meno** a **Heslo**, a po následnom odhlásení zapíšte prázdny string do týchto kontroliek.

### Používanie premenných na prenos dát medzi VI

Premenné môžete použiť na sprístupnenie a prenos dát medzi viacerými VI ktoré bežia súčasne. Lokálna premenná zdieľa dáta v rámci VI. Globálna premenná zdieľa dáta medzi viacerými VI. Napríklad, predstavte si, že máte dva VI ktoré bežia súčasne. Každé VI obsahuje While slučku a zapisuje dáta do waveform chartu. Prvé VI obsahuje boolean kontrolku na ukončenie oboch VI. Môžete použiť globálnu premennú s jedinou boolean kontrolkou na ukončenie slučiek. Ak by obe slučky boli na tom istom blokovom diagrame v rámci jedného VI, mohli by ste použiť lokálnu premennú na ukončenie slučiek.

Spôsobom, akým používate globálnu premennú, môžete použiť aj single-process zdieranú premennú. Zdieľaná premenná je podobná lokálnej premennej alebo globálnej premennej, pričom ale umožňuje zdieľanie dát cez sieť. Zdieľaná premenná (shared variable) môže byť single-process alebo network-published. Hoci network-published zdieľané premenné sú mimo rozsahu tohto kurzu, ak použijete single-process zdieľané premenné, v budúcnosti ich môžete zmeniť na network-published.

Použite globálnu premennú na zdieľanie dát medzi VI v rámci jedného počítača, najmä ak nepoužívate projektový súbor. Použite single-process zdieľanú premennú ak v budúcnosti budete potrebovať zdieľať dáta medzi VI na rôznych počítačoch.

### **Tvorba globálnych premenných**

Globálne premenné použite na sprístupnenie a prenos dát medzi viacerými VI ktoré bežia súčasne. Globálne premenné sú vstavané LabVIEW objekty. Keď vytvoríte globálnu premennú, LabVIEW automaticky vytvorí špeciálne globálne VI, ktoré disponujú iba čelným panelom bez blokového diagramu. Pridajte kontrolky a indikátory na čelný panel globálneho VI. Týmto definujete dátové typy ktoré bude globálna premenná obsahovať. Vskutku, tento čelný panel je kontajner z ktorého sprístupňuje dáta viacerým VI.

Napríklad, predstavte si, že máte 2 VI ktoré bežia súčasne. Každé VI obsahuje While slučku a zapisuje dáta do waveform chartu. Prvé VI obsahuje boolean kontrolku na ukončenie oboch VI. Musíte použiť globálnu premennú s jedinou boolean kontrolkou na ukončenie slučiek. Ak by obe slučky boli na tom istom blokovom diagrame v rámci jedného VI, mohli by ste použiť lokálnu premennú na ukončenie slučiek.



Zvoľte globálnu premennú z palety **Functions** a umiestnite ju na blokový diagram.

Dvojklikom na globálnu premennú zobrazíte čelný panel globálneho VI. Umiestnite kontrolky a indikátory na čelný panel, tak ako by ste to robili v prípade štandardného čelného panelu.

LabVIEW používa vlastné návestia na identifikáciu globálnych premenných, preto označte kontrolky a indikátory s opisnými návestiami.

Môžete vytvoriť viacero globálnych premenných, každé VI bude mať vlastný čelný panel. Môžete aj zoskupiť podobné premenné, vytvorením jednej globálnej premennej s viacerými objektmi na čelnom paneli.

Globálne VI s viacerými objektmi je účinnejšie riešenie, nakoľko môžete zoskupiť súvisiace premenné. Blokový diagram VI môže obsahovať viacero uzlov globálnej premennej. Tieto uzly sú asociované s kontrolkami a indikátormi z čelného panela globálnej premennej. Globálne VI vložíte do VI tak ako by ste vložili subVI do VI. Vždy, keď vložíte novú globálnu premennú na blokový diagram, LabVIEW vytvorí nové globálne VI asociované iba s aktuálne vloženým uzlom globálnej premennej resp. s kópiami tejto premennej.

Po ukončení vkladania objektov do čelného panela VI globálnej premennej, uložte ju a vráťte sa k blokovému diagramu pôvodného VI. Následne musíte vybrať objekt z VI globálnej premennej ku ktorému chcete pristupovať. Kliknite na uzol globálnej premennej a zvoľte objekt čelného panela z kontextovej ponuky. Kontextová ponuka zobrazí všetky objekty čelného panela VI globálnej premennej ktoré majú vlastné návestia. Môžete aj pravým tlačidlom myši kliknúť na uzol globálnej premennej a zvoliť prvok z **Select Item** kontextovej ponuky.

Môžete použiť aj nástroj Operating tool alebo Labeling tool a kliknúť na uzol globálnej premennej a následne zvoliť objekt čelného panelu z kontextovej ponuky.

Ak chcete použiť túto globálnu premennú aj v iných VI, zvoľte položku **Select a VI** na palette **Functions**. Štandardne je globálna premenná asociovaná s prvým objektom čelného panelu ktorý má vlastné návestie. Asociáciu globálnej premennej na iný objekt čelného panela vykonáte ak kliknete pravým tlačidlo

myši na uzol globálnej premennej v blokovom diagrame a zvolíte objekt čelného panela z kontextovej ponuky **Select Item**.

### Tvorba single-process zdieľaných premenných

K použitiu zdieľaných premenných musíte mať projektový súbor. Single-process zdieľanú premennú vytvoríte pomocou pravého kliknutia na **My Computer** v okne **Project Explorer** a voľbou **New»Variable**. Zobrazí sa **Shared Variable Properties** dialógové okno.

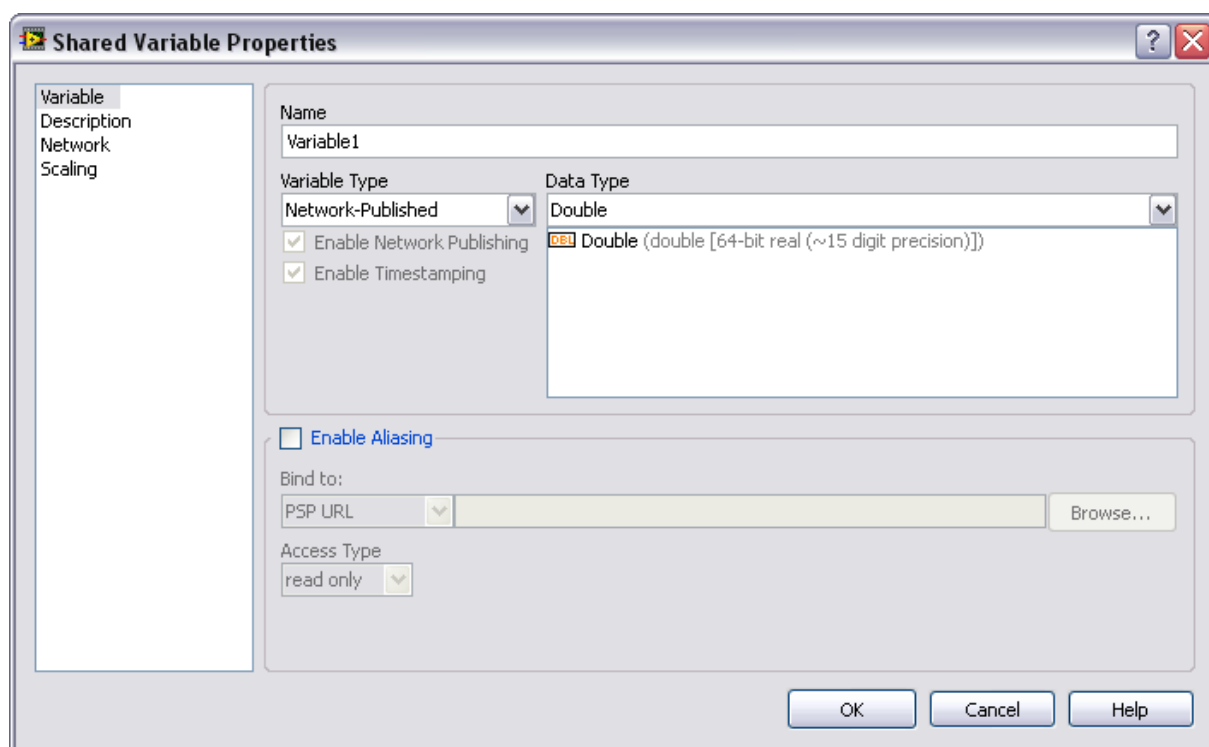


Figure 2.4: Shared Variable Properties dialógové okno

V časti **Variable Type**, zvolíte **Single Process**. Zadaťte názov a typ premennej. Po vytvorení zdieľanej premennej, premenná sa automaticky zobrazí v novej knižnici v projektovom súbore. Uložte knižnicu. Do tejto knižnici môžete pridávať dodatočné zdieľané premenné, podľa potreby. Premennú môžete jednoducho uchopiť a pretiahnuť z okna **Project Explorer** priamo do blokového diagramu. Použite kontextovú ponuku na prepínanie medzi zápisom a vyčítaním. Použite error cluster premenných na zabezpečenie toku dát, toku vykonávania.

### Používanie premenných s nadhľadom

Lokálne aj globálne premenné sú pokročilé koncepty LabVIEW. Nie sú súčasťou modelu vykonávania kódu podľa toku údajov, na ktorom je LabVIEW postavený. Blokové diagramy sa môžu stať ťažko čitateľné ak používate lokálne alebo globálne premenné, aj preto by ste ich mali používať opatrne. Nesprávne použitie lokálnych a globálnych premenných, ako napríklad nahradenie konektorového panelu premennými, alebo ich použitie kvôli prístupu k dátam v jednotlivých rámcoch sequence štruktúry, môže viesť neočakávanému správaniu VI. Nesprávne použitie, ako napríklad z dôvodu obísť nutnosť vytvárať dlhé spojenia cez celý blokový diagram, alebo ich používanie namiesto prenosu údajov pomocou toku dát, spomalí vykonávanie kódu.

### **Inicializácia premenných**

Inicializáciu lokálnej alebo globálnej premennej vykonávame kvôli tomu, aby premenná obsahovala známu hodnotu predtým ako sa použije. V opačnom prípade, premenné môžu obsahovať dáta ktoré zapríčinia chybné správanie VI. Ak premenná ako vstupnú, inicializačnú hodnotu očakáva z výsledku určitého výpočtu, uistite sa, že LabVIEW zapíše hodnotu do premennej predtým ako použije túto premennú v nejakej operácii v inej lokácii. Paralelný zápis do premennej z viacerých miest môže spôsobiť chyby súbehu (race conditions).

Aby ste sa uistili, že inicializácia sa vykoná prvá v poradí, môžete kód vložiť do prvého rámca sequence štruktúry, alebo do subVI ktoré je prepojené ďalšími časťami tak, aby sa subVI vykonalo ako prvé.

Ak nevykonáte inicializáciu premennej predtým ako VI použije premennú po prvý krát, VI bude obsahovať štandardnú hodnotu priradenú k danému objektu čelného panela.

### **Funkcionálne globálne premenné**

Na uloženie dát môžete použiť neinicializované posuvné registre v For alebo While slučkách, pokiaľ je dané VI v pamäti. Posuvný register si udržiava poslednú zapísanú hodnotu. Vložte While slučku do subVI a použite posuvné registre na uloženie dát. Táto technika je podobná použitiu globálnej premennej. Táto metóda sa často nazýva funkcionálnou globálnou premennou. Výhodou tejto metódy oproti globálnej premennej je fakt, že môžete kontrolovať prístup k dátam v posuvnom registri. Všeobecná forma

funkcionálnej globálnej premennej obsahuje neinicializovaný posuvný register s For alebo While slučkou, ktorá sa vykoná iba raz.

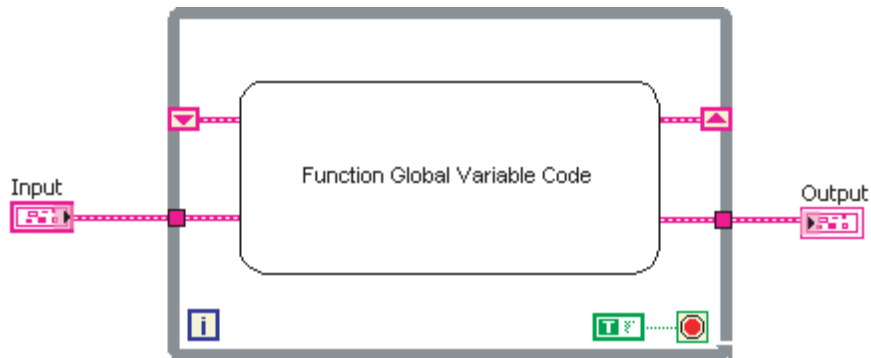


Figure 2.5: Základný tvar funkcionálnej globálnej premennej

Funkcionálna globálna premenná obyčajne má vstupný parameter **akcia**, ktorý špecifikuje ktorú úlohu VI vykoná. VI používa neinicializovaný posuvný register vo While slučke na zachovanie hodnôt operácie.

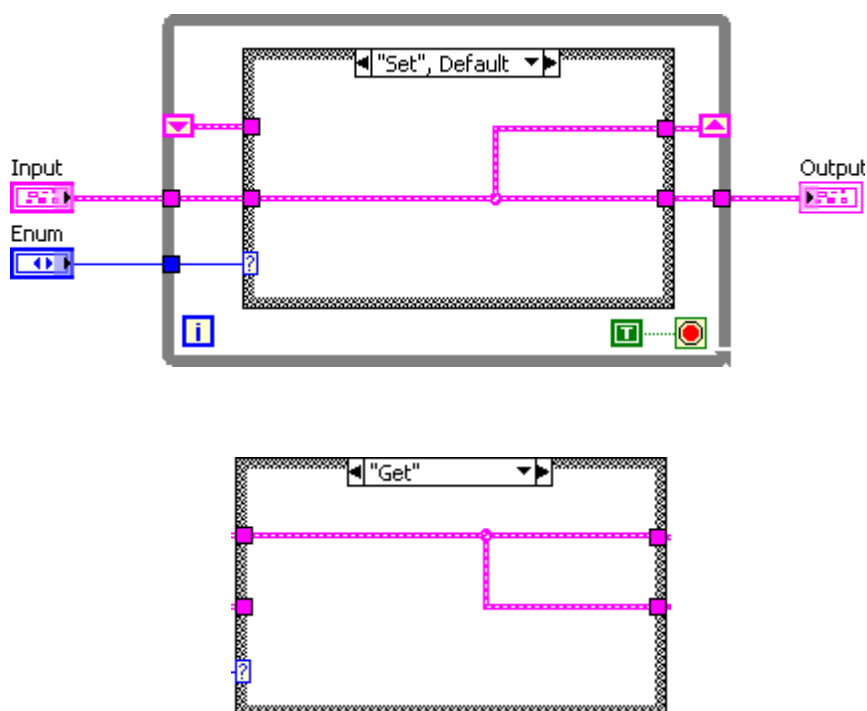


Figure 2.6: Funkcionálna globálna premenná s funkcionalitou na nastavenie a získanie údajov

V tomto príklade, dáta prichádzajú do VI a posuvný register ich uloží ak je enum kontrolka nastavená na Set. Dáta sú vyčítané z posuvného registra, ak je enum kontrolka nastavená na Get.

Funkcionálne globálne premenné môžete použiť na implementáciu jednoduchých globálnych premenných, avšak ako to bolo zobrazené v predošlom príklade, sú obzvlášť užitočné keď implementujete komplexnejšie dátové štruktúry, ako napríklad zásobník. Funkcionálne globálne premenné môžete použiť aj na ochranu prístupu k globálnym zdrojom, ako napr. súbory, prístroje, DAQ zariadenia, ktoré sa nedajú reprezentovať globálnou premennou.

Funkcionálna globálna premenná je subVI, ktorá nie je reentrantná. To znamená, ak je subVI volané z viacerých miest, je stále použitá tá istá kópia subVI v pamäti. Tým pádom, iba jedno volanie subVI môže nastať naraz.

### **Použitie funkcionálnych globálnych premenných na časovanie**

Jedna naozaj užitočná aplikácia funkcionálnych globálnych premenných je uskutočnenie časovania VI. Početné množstvo VI ktoré vykonávajú meracie a automatizačné úlohy, požaduje určitú formu časovania. Často sa stáva, že prístroj alebo zariadenie potrebuje čas na inicializáciu. Tým pádom sa musí zahrnúť explicitné časovanie do VI tak, aby aplikácia poskytla potrebný čas na inicializáciu systému. Môžete vytvoriť funkcionálnu globálnu premennú ktorá meria čas trvania medzi jednotlivými volaniami VI.

V prípade Elapsed Time sa vykoná odpočítanie času uloženého v posuvnom registri od aktuálneho času získaného pomocou funkcie Get Date/Time In Seconds. Prípad Reset Time inicializuje funkcionálnu globálnu premennú s aktuálnym časom.

Elapsed Time Express VI implementuje totožnú funkcionálnosť ako táto funkcionálna globálna premenná. Výhodou používania funkcionálnej globálnej premennej je jednoduchá možnosť úprav jej implementácie, ako napríklad pridanie možnosti pre pauzu.

### **Chyby súbehu (Race conditions)**

K chybám súbehu dochádza keď časovanie udalostí alebo úloh neúmyselne ovplyvní výstup alebo hodnotu dát. Chyby súbehu sú bežným problémom v programoch ktoré vykonávajú viacero úloh súbežne a zdieľajú dáta medzi úlohami.

Na jednoprocessorových počítačoch, úkony v multi-taskingovom programe sa vykonávajú v skutočnosti sekvenčne. LabVIEW a operačný systém rýchlo prepína medzi úlohami, tým pádom úlohy sa javia ako súčasne vykonávané. V tomto prípade chyba súbehu nastane ak prepnutie z jednej úlohy do druhej nastane v istom čase.

Chyby súbehu sú ťažko identifikovateľné a laditeľné, nakoľko výstup závisí od poradia v akom operačný systém vykoná plánované úlohy a od časovania externých udalostí. Spôsob interakcie úloh a operačného systému, ako aj ľubovoľné časovanie externých udalostí, robí toto poradie v zásade náhodným. Kód s chybou súbehu môže vracaať tú istú hodnotu tisíc krát v testovacej prevádzke, a predsa v reálnej prevádzke môže vrátiť neočakávaný výstup.

Najlepším spôsobom na zabránenie chýb súbehu je použitie následovných techník:

- Riadenie a obmedzenie zdieľaných zdrojov.
- Identifikácia a ochrana kritických sekcií v kóde.
- Špecifikácia poradia vykonávania.

### **Riadenie a obmedzenie zdieľaných zdrojov.**

Chyby súbehu sú bežné ak obe úlohy čítajú a zapisujú do zdieľaného zdroja, ako je to v predošlom prípade. Zdroj je ľubovoľná entita, ktorá je spoločná pre viaceré procesy. Pri chybách súbehu sú najbežnejšími zdieľanými zdrojmi úložiská dát, ako napríklad premenné. Príkladom na ďalšie zdroje sú súbory a referencie na hardvérové zdroje.

Umožnenie zmeny zdroja z viacerých miest často vnáša možnosť výskytu chýb súbehu. Tým pádom, ideálnym spôsobom na zabránenie chýb súbehu je minimalizácia zdieľaných premenných a počtu zapisujúcich uzlov do zvyšných zdieľaných premenných. Vo všeobecnosti platí, že mať viacero uzlov na čítanie alebo monitoring zdieľaných premenných nie je nebezpečné. Avšak, použite iba jeden zapisujúci uzol alebo kontrolér na jednu zdieľanú premennú. Väčšina chýb súbehu nastane ak zdroj má viacero zapisujúcich uzlov.

V predošlom prípade môžete znížiť závislosť na zdieľaných zdrojoch ak každá slučka si bude udržiavať počet lokálne. Následne, až po kliknutí na tlačidlo **Stop** zdieľajte počty. K tomu je potrebný iba jeden uzol na čítanie a jeden na zápis do

zdieľanej premennej, tým pádom sa eliminuje možnosť chyby súbehu. Ak všetky zdieľané zdroje majú iba jeden zapisujúci uzol alebo kontrolér, a VI má správne poradie inštrukcií, tak chyby stehu nenastanú.

### **Ochrana kritických sekcií**

Kritická sekcia kódu je tá časť kódu ktorá sa musí správať konzistentne za všetkých okolností. Keď používate multi-tasking programy, úloha môže prerušiť beh inej úlohy. Toto sa deje takmer vo všetkých súčasných operačných systémoch. Za bežných okolností, prepínanie medzi úlohami nemá žiadny následok. Avšak keď úloha ktorá prerušila vykonávanie inej úlohy zmení zdieľanú premennú, ktorú prerušená úloha považuje za konštantnú, dôjde k chybe súbehu.

Ak jedna slučka preruší druhú slučku počas vykonávania kódu v kritickej sekcii, môže dôjsť k chybe súbehu. Jedným zo spôsobov na elimináciu chýb súbehu je identifikácia a ochrana kritickej sekcie kódu. Existuje viacero techník na ochranu kritického kódu. Dve najefektívnejšie sú funkcionálne globálne premenné a semaforey.

### **Funkcionálne globálne premenné**

Jednou z možností ako chrániť kritický kód je vložiť ho do subVI. Volať nereentrantné subVI môžete iba z jedného miesta súčasne. Tým pádom, vloženie kritického kódu do nereentrantného subVI zabezpečí ochranu kódu pred prerušením inými volajúcimi procesmi. Používanie architektúry funkcionálnej globálnej premennej na ochranu kritickej sekcie kódu je obzvlášť efektívne, nakoľko pomocou posuvných registrov môžete nahradiť menej chránené metódy dátových úložísk, ako napr. globálne, alebo single-process zdieľané premenné. Funkcionálne globálne premenné taktiež podnecujú tvorbu multi-funkcionálnych subVI ktoré obslúžia všetky úlohy asociované s daným zdrojom.

Po identifikácii všetkých sekcií kritického kódu, zoskupte tieto sekcie podľa zdrojov ku ktorým pristupujú, a následne vytvorte funkcionálne globálne premenné pre každý zdroj. Z kritických sekcií vykonávajúcich rôzne operácie môžete spraviť príkazy pre funkcionálnu globálnu premennú, a tým pádom môžete zoskupiť totožné operácie do jediného príkazu, čím zvýšite znovupoužiteľnosť kódu.

## **Semaforey**

Semaforey sú synchronizačné mechanizmy, špecificky navrhnuté na ochranu zdrojov a kritických sekcií kódu. Môžete zabrániť tomu, aby sa kritické sekcie kódu sa navzájom prerušovali. Uzavrite kritický kód medzi Acquire Semaphore a Release Semaphore VI. Štandardne, semafor neumožní aby ho získala viac než jedna úloha. Tým pádom, po tom, čo jedna z úloh vstúpi do kritickej sekcie, ostatné úlohy nemôžu vstúpiť do kritickej sekcie pokým sa prvá úloha neukončí. Pri správnom použití, táto metóda eliminuje chyby súbehu.

### **Určenie poradia vykonávania**

Kód, v ktorom tok dát nie je správne použitý na riadenie poradia vykonávania, môže spôsobiť chyby súbehu. Keď závislosť na dátach nie je zavedená, LabVIEW môže naplánovať úlohy v ľubovoľnom poradí, čo môže mať za následok chyby súbehu, ak úlohy sú vzájomne závislé.

### 3. NÁVRH POUŽÍVATELSKÉHO ROZOHRAŇÍ S DOTYKOVOU OBRAZOVKOU

#### **Building User Interfaces and HMIs Using LabVIEW**

This chapter examines one software design architecture for building an operator interface with a scalable navigation engine for cycling through different HMI pages.

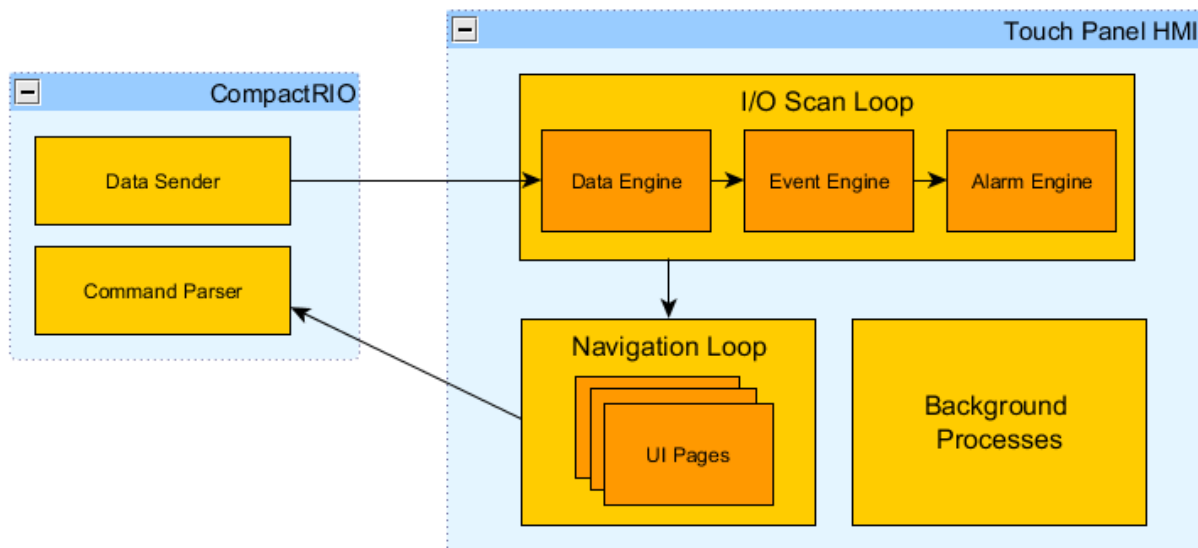
You can use this architecture to build an HMI based on LabVIEW for any HMI hardware targets including the NI TPC-2512 touch panel computer running the Windows XP Embedded OS or the NI TPC-2106 running the Windows CE OS and the NI PPC-2115 panel PC running the Windows XP OS. LabVIEW is a full programming language that provides one solution for a variety of development tasks ranging from HMI/SCADA systems to reliable and deterministic control applications. The LabVIEW Touch Panel Module offers a graphical programming interface that you can use to develop an HMI in a Windows development environment and then deploy it to an NI touch panel computer (TPC) or any HMIs running Windows CE. This chapter offers a framework for engineers developing HMIs based on both Windows Vista/XP and Windows CE.

#### **Basic HMI Architecture Background**

An HMI can be as simple or as complex as the functionality you require. The software architecture defines the functionality of an HMI as well as its ability to expand and adapt to future technologies. A basic HMI has three main routines:

1. Initialization and shutdown (housekeeping) routines
2. I/O scan loop
3. Navigation (user interface) loop

Before executing the I/O scan loop and the navigation loop, the HMI needs to perform an initialization routine. This initialization routine sets all controls, indicators, internal variables, and variables communicating with the hardware (controller) to default states. You can add more logic to prepare the HMI for operations such as logging files. Before stopping the system, the shutdown task closes any references and performs additional tasks such as logging error files. Initialization and shutdown tasks are not diagramed since they are not processes (they execute only once).



**Figure 3.1: Basic Software Architecture of a Touch Panel HMI Communicating With a CompactRIO Controller**

## I/O Scan Loop

The I/O scan loop typically consists of three components: a Data Engine, an Event Engine, and an Alarm Engine.

## Data Engine

The Data Engine exchanges tag values with the controller via an Ethernet-based communication protocol or Modbus. It receives this data across the network and makes it available throughout the HMI application.

## Event Engine

The I/O scan loop might also handle alarms and events. An Event Engine compares a subset of tag values to a set of predefined conditions (value equal to X, value in range or out of range, and so on) and logs an event when a tag value matches one of its event conditions.

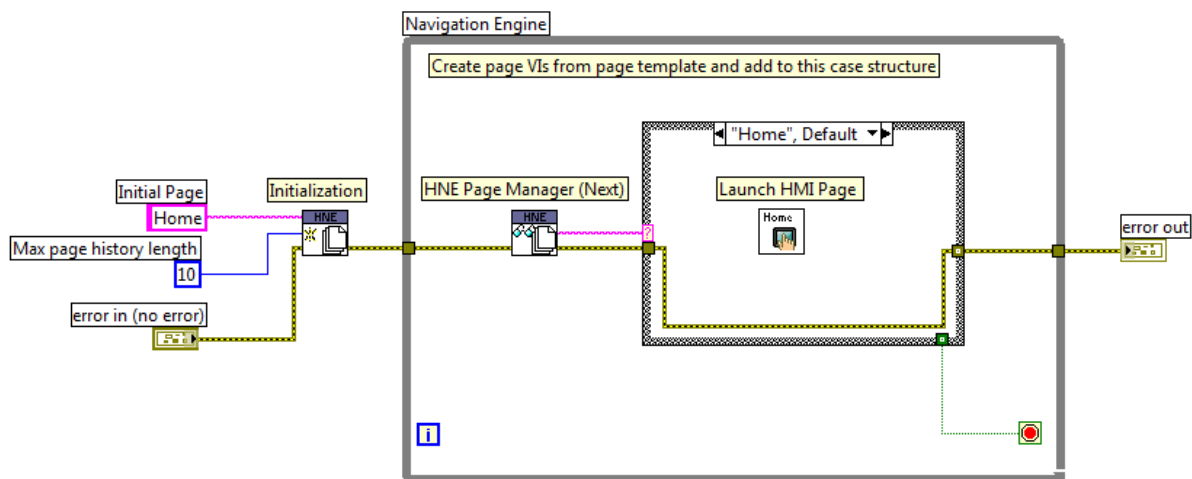
## Alarm Engine

Some events are simply logged while other events require operator intervention and are configured as alarms. The alarm event data is sent to the Alarm Displays Engine, which manages how the alarm is presented to the operator. When the tag value leaves the alarm state, the Event Engine sends an alarm canceling the event to the Alarm Displays Engine.

## Navigation Loop

With a Windows CE-based touch panel, you do not have individual windows like you do in a Windows XP or Windows 7 application. Instead, the top-level VI occupies the entire screen. In addition, the tab control is not supported in Windows CE. Therefore, to switch between different panels (UIs) in the application, you need to add support in your UI such as buttons to switch to other panels.

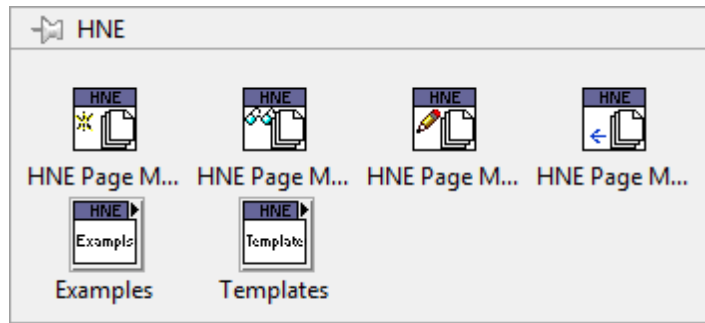
The navigation loop handles the management and organization of different UIs in your application. You can implement a navigation loop as a simple state machine built with a While Loop and a Case structure. Each case encloses an HMI page VI that, when called, is displayed on the HMI screen. Figure 3.2 is an example of a navigation loop.



**Figure 3.2: Example Navigation Loop Block Diagram**

This example uses the HMI Navigation Engine (HNE) reference library, which was created for HMI page management and navigation. The HNE is based on VIs included with the LabVIEW Touch Panel Module, but it features an additional history cache so the user can define a button to jump backward toward the previously viewed screen. The HNE also includes basic templates and examples for getting started. Refer to the NI Developer Zone document HMI Navigation Engine (HNE) Reference Library to download the HNE library.

The HNE installs a page manager API palette named HNE to the User Libraries palette in LabVIEW.



**Figure 3.3. HNE (HMI Navigation Engine) Palette**

**HNE Page Manager (Init)**—This VI initializes the HNE by setting the navigation history depth and setting the name of the first page to be displayed.

**HNE Page Manager (Next)** —This VI returns the name of the next page and passes it to the Case structure in the HNE.

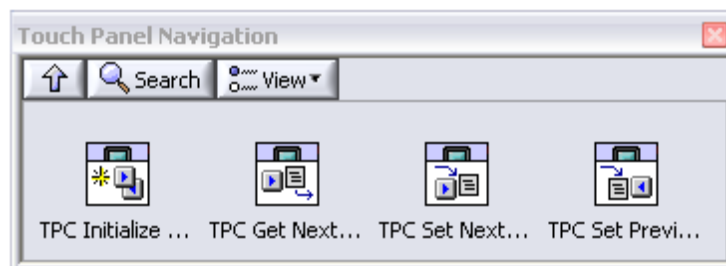
**HNE Page Manager (Set)**—This VI sets the name of the next page to be displayed. It is used within HMI pages to support navigation buttons.

**HNE Page Manager (Back)**—This VI returns the name of the previous page from the page history. It is used within HMI pages to support the operation of a “back” navigation button.

**Examples subpalette**—This is a subpalette that contains the example VI for the HNE API.

**Templates subpalette**—This is a subpalette that contains two template VIs for the HNE API. The first is a template VI for the navigation loop called **HMI\_NavigationEngine VI** and the second is a template for an HMI page called **HMI\_Page VI**.

The navigation engine uses VIs from the Touch Panel Navigation palette.



**Figure 3.4. The Navigation Palette**

TPC Initialize Navigation—This VI initializes the navigation engine by setting the navigation history depth and the name of the first page to be displayed.

TPC Get Next Page—This VI returns the name of the next page and passes it to the Case structure in the HMI navigation engine.

TPC Set Next Page—This VI sets the name of the next page to be displayed. Use it within HMI pages to support navigation buttons.

TPC Set Previous Navigation Page—This VI returns the name of the previous page from the page history. Use it within HMI pages to support the operation of a “back” navigation button.

The page state and history are stored in a functional global variable that each of these VIs access.

## UI Pages

As stated above, the navigation loop contains all of the HMI pages for an application. Each HMI page is a LabVIEW VI created to monitor and configure a specific process or subprocess in the machine. The most common elements on a page front panel are navigation buttons, action buttons, numeric indicators, graphs, images, and Boolean controls and indicators. Figure 3.5 shows an example page containing a typical set of front panel elements.

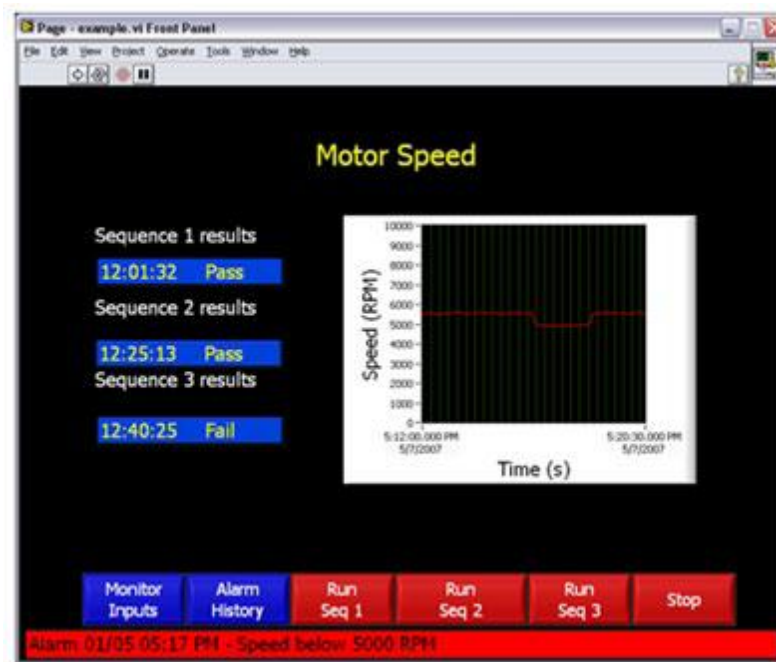
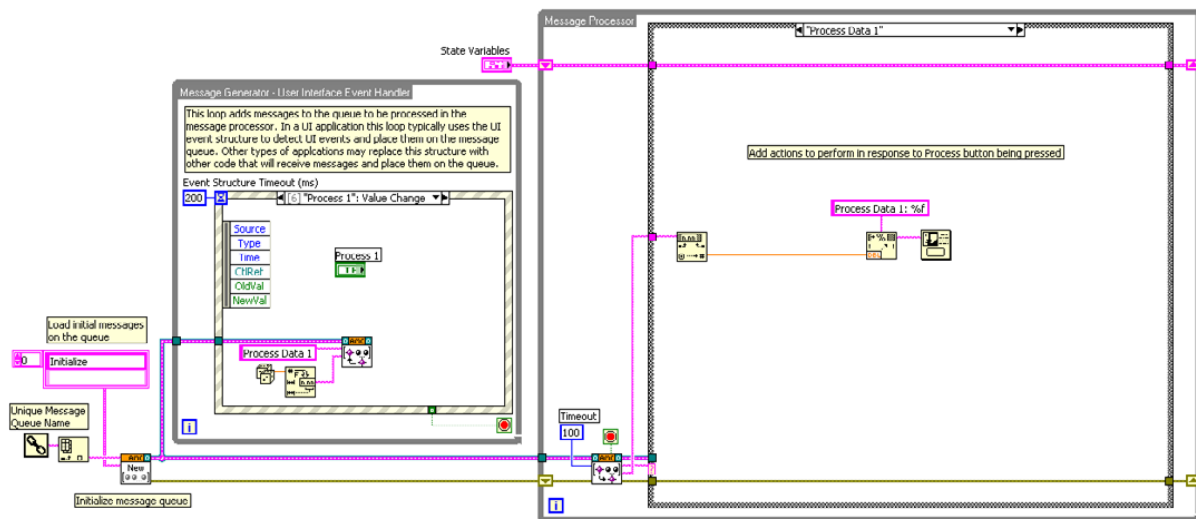


Figure 3.5. Example of a Typical UI Page in LabVIEW

The page block diagram uses the event-based producer consumer design pattern to implement a responsive, event driven UI. The example in Figure 3.6 uses the Asynchronous Message Communication (AMC) reference library for interprocess communication. You can download the AMC reference library, which uses queues for interprocess communication, from the NI Developer Zone document [Asynchronous Message Communication \(AMC\) Reference Library](#). The AMC library also has an API based on UDP that you can use to send messages across the network. You can implement the Figure 3.6 example using queue functions for communicating between the Event Handler process and the Message process.



**Figure 3.6. Example Page Block Diagram Using AMC Design Pattern**

For more information on creating UI pages, see the NI Developer Zone document [Creating HMI Pages for the LabVIEW Touch Panel Module](#).

## 4. PRIDANIE POČÍTAČOVÉHO VIDENIA A ROZPOZNÁVANIA OBRAZU DO COMPACTRIO SYSTÉMU PRE MERACIE A RIADIACE APLIKÁCIE

### Machine Vision/Inspection

Machine vision is a combination of image acquisition from one or more industrial cameras and the processing of the images acquired. These images are usually processed using a library of image processing functions that range from simply detecting the edge of an object to reading various types of text or complex codes.

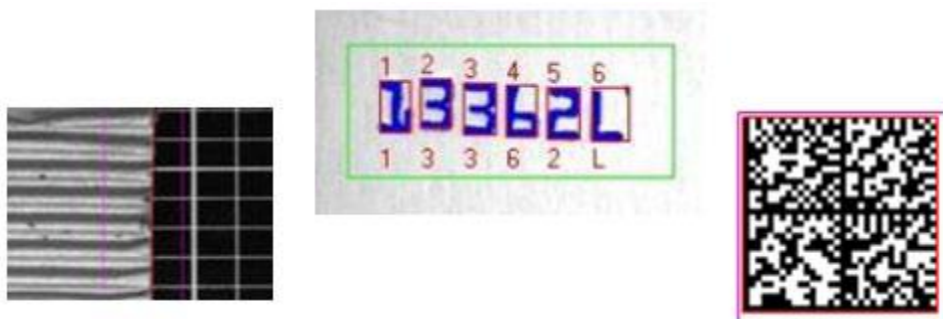


Figure 4.1. Edge detection, optical character recognition, and 2D code reading are common machine vision tasks.

Often more than one of these measurements is made on one vision system from one or more images. You can use this for many applications including verifying that the contents of a container match the text on the front of the bottle or ensuring that a code has printed in the right place on a sticker.

The information from these processed images is fed into the control system for data logging, defect detection, motion guidance, process control, and so on.

For information on the algorithms in NI vision tools, see the Vision Concepts Manual.

### Machine Vision System Architecture

Typical machine vision systems consist of an industrial camera that connects to a real-time vision system, usually via a standardized camera bus such as IEEE 1394, Gigabit Ethernet, or Camera Link. The real-time system processes the images and has I/O for communicating to the control system.

A few companies have combined the camera with the vision system, creating something called a smart camera. Smart cameras are industrial cameras that have onboard image processing and typically include some basic I/O.

NI offers both types of embedded machine vision systems. The NI Compact Vision System (Figure 4.2) is a real-time embedded vision system that features direct connectivity to up to three IEEE 1394 cameras as well as 29 general use I/O channels for synchronization and triggering. This system features Ethernet connectivity to your industrial network, allowing communication back to CompactRIO hardware.

NI Smart Cameras (Figure 4.2) are industrial image sensors combined with programmable processors to create rugged, all-in-one solutions for machine vision applications. These cameras have a VGA (640x480 pixels) or SXGA (1280x1024 pixels) resolution as well as an option to include a digital signal processor (DSP) for added performance for specific algorithms. These cameras feature dual Gigabit Ethernet ports, digital inputs and outputs, and a built-in lighting controller.



**Figure 4.2. NI Compact Vision System and NI Smart Camera**

NI also offers plug-in image acquisition devices called frame grabbers that provide connectivity between industrial cameras and PCI, PCI Express, PXI, and

PXI Express slots. These devices are commonly used for scientific and automated test applications, but you also can use them to prototype a vision application on a PC before you purchase any industrial vision systems or smart cameras.

All NI image acquisition hardware uses the same driver software called NI Vision Acquisition software. With this software, you can design and prototype your application on the hardware platform of your choice and then deploy it to the industrial platform that best suits your application requirements with minimal code changes.

## Lighting and Optics

This guide does not cover lighting and optics in depth, but they are crucial to the success of your application. These documents cover the majority of the basic and some more advanced machine vision lighting concepts:

- A Practical Guide to Machine Vision Lighting—Part I
- A Practical Guide to Machine Vision Lighting—Part II
- A Practical Guide to Machine Vision Lighting—Part III

The lens used in a machine vision application changes the field of view. The field of view is the area under inspection that is imaged by the camera. You must ensure that the field of view of your system includes the object you want to inspect. To calculate the horizontal and vertical field of view (FOV) of your imaging system, use Equation 4.1 and the specifications for the image sensor of your camera.

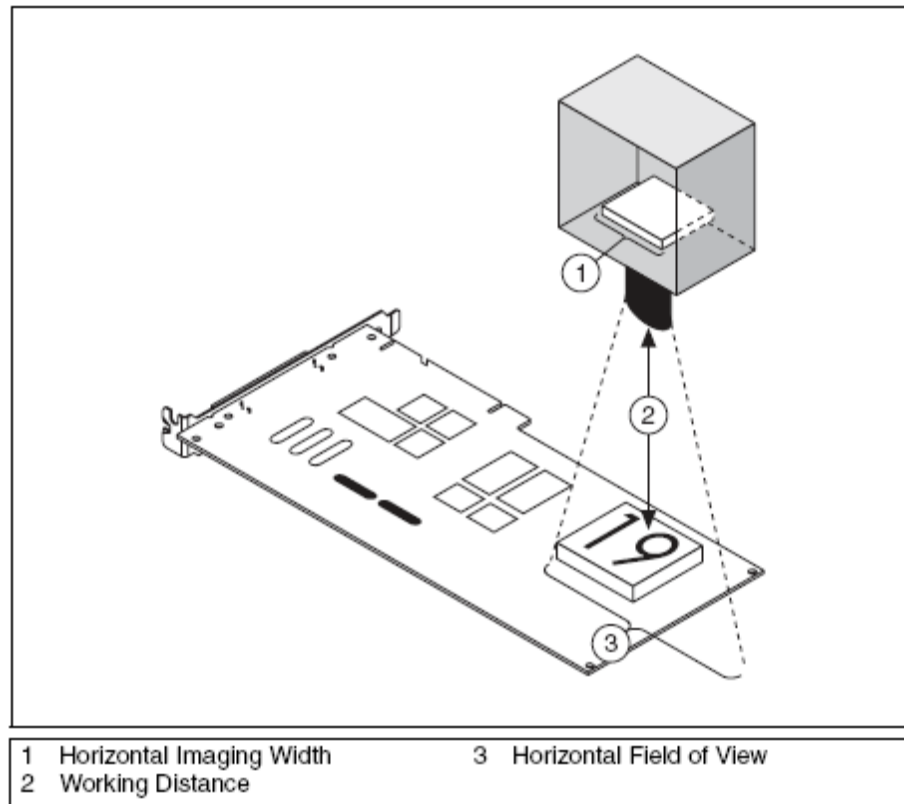
$$FOV = \frac{\text{Pixel Pitch} \times \text{Active Pixels} \times \text{Working Distance}}{\text{Focal Length}}$$

### Eq 4.1. Field of View Calculation

Where

- FOV is the field of view in either the horizontal or vertical direction
- Pixel pitch measures the distance between the centers of adjacent pixels in either the horizontal or vertical direction
- Active pixels is the number of pixels in either the horizontal or vertical direction

- Working distance is the distance from the front element (external glass) of the lens to the object under inspection
- Focal length measures how strongly a lens converges (focuses) or diverges (diffuses) light



**Figure 4.3. The lens selection determines the FOV.**

For example, if the working distance of your imaging setup is 100 mm, and the focal length of the lens is 8 mm, then the FOV in the horizontal direction of an NI Smart Camera using the VGA sensor in full Scan Mode is

$$FOV_{horizontal} = \frac{0.0074 \text{ mm} \times 640 \times 100 \text{ mm}}{8 \text{ mm}} = 59.2 \text{ mm}$$

Similarly, the FOV in the vertical direction is

$$FOV_{vertical} = \frac{0.0074 \text{ mm} \times 480 \times 100 \text{ mm}}{8 \text{ mm}} = 44.4 \text{ mm}$$

Based on the result, you can see that you may need to adjust the various parameters in the FOV equation until you achieve the right combination of components that match your inspection needs. This may include increasing

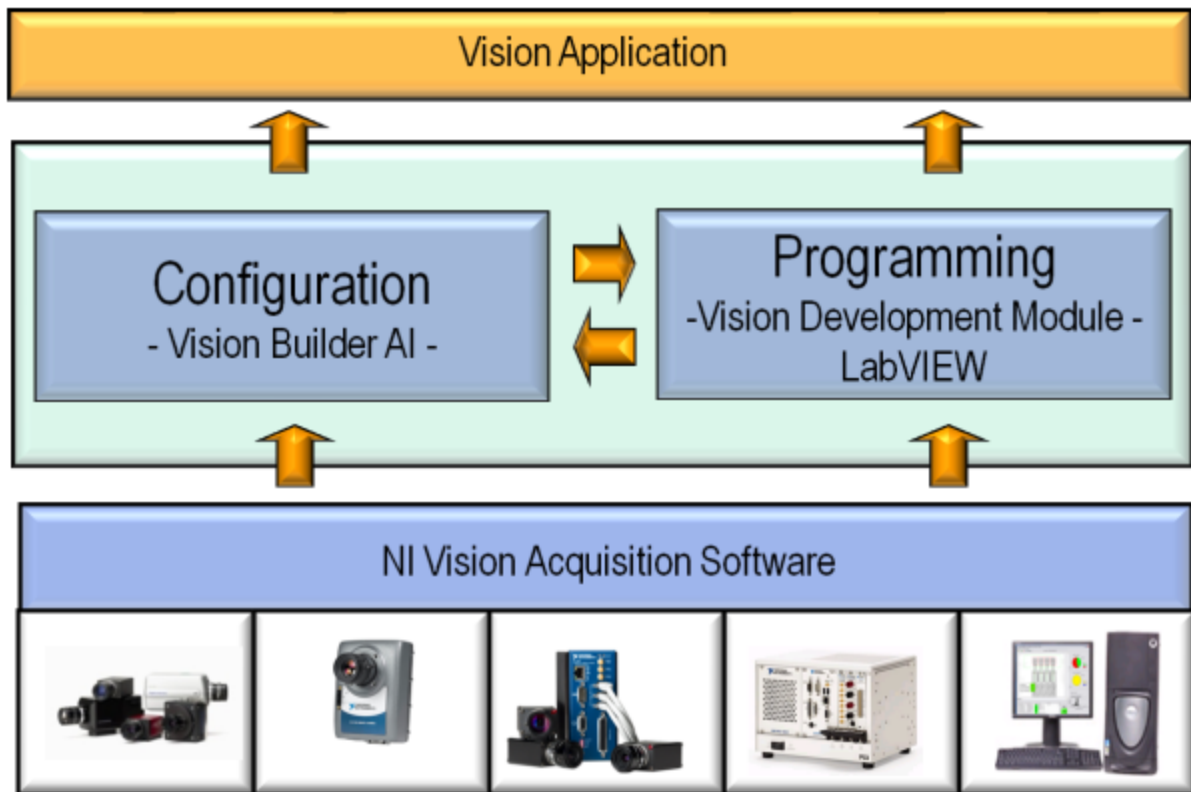
your working distance, choosing a lens with a shorter focal length, or changing to a high-resolution camera.

## **Software Options**

Once you have chosen the hardware platform for your machine vision project, you need to select the software platform you want to use. NI offers two application development environments (ADEs) for machine vision. Both NI Compact Vision Systems and NI Smart Cameras are LabVIEW Real-Time targets, so you can develop your machine vision application using the LabVIEW Real-Time Module and the NI Vision Development Module.

The Vision Development Module is a library of machine vision functions that range from basic filtering to pattern matching and optical character recognition. This library also includes the NI Vision Assistant and the Vision Assistant Express VI. The Vision Assistant is a rapid prototyping tool for machine vision applications. With this tool, you can use click-and-drag, configurable menus to set up most of your application. With the Vision Assistant Express VI, you can use this same prototyping tool directly within LabVIEW Real-Time.

Another software platform option is NI Vision Builder for Automated Inspection (AI). Vision Builder AI is a configurable machine vision ADE based on a state diagram model, so looping and decision making are extremely simple. Vision Builder AI features many of the high-level tools found in the Vision Development Module. Both hardware targets work with Vision Builder AI as well, giving you the flexibility to choose the software you are most comfortable with and the hardware that best suits your application.



**Figure 4.4. NI offers both configuration software and a full programming environment for machine vision application development.**

### **Machine Vision/Control System Interface**

Most smart camera or embedded vision system applications provide real-time inline processing and give outputs that you can use as another input into the control system. The control system usually has control over when this image acquisition and processing starts via sending a trigger to the vision system. The trigger can also come from hardware sensors such as proximity sensors or quadrature encoders.

The image is processed into a set of usable results such as the position of a box on a conveyor or the value and quality of a 2D code printed on an automotive part. These results are reported back to the control system and/or sent across the industrial network for logging. You can choose from several methods to report these results, from a simple digital I/O to shared variables or direct TCP/IP communication, as discussed previously in this document.

### **Machine Vision Using LabVIEW Real-Time**

The following example demonstrates the development of a machine vision application for an NI Smart Camera using LabVIEW Real-Time and the Vision Development Module.

### Step 1. Add an NI Smart Camera to the LabVIEW Project

You can add the NI Smart Camera to the same LabVIEW project as the CompactRIO system. If you wish to prototype without the smart camera connected, you can also simulate a camera.

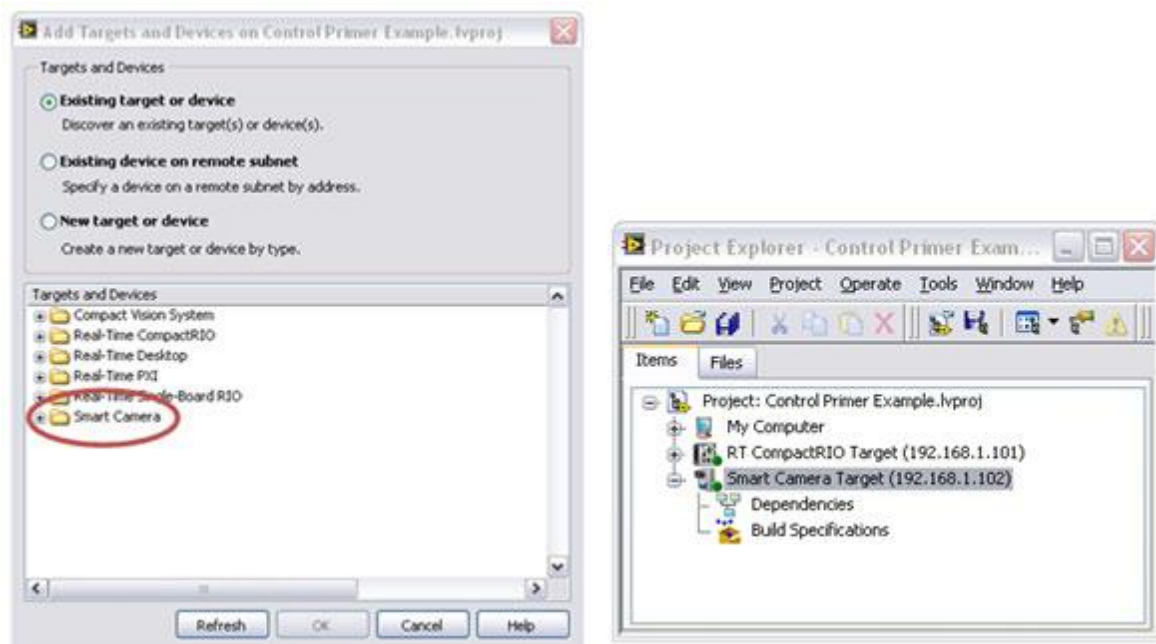
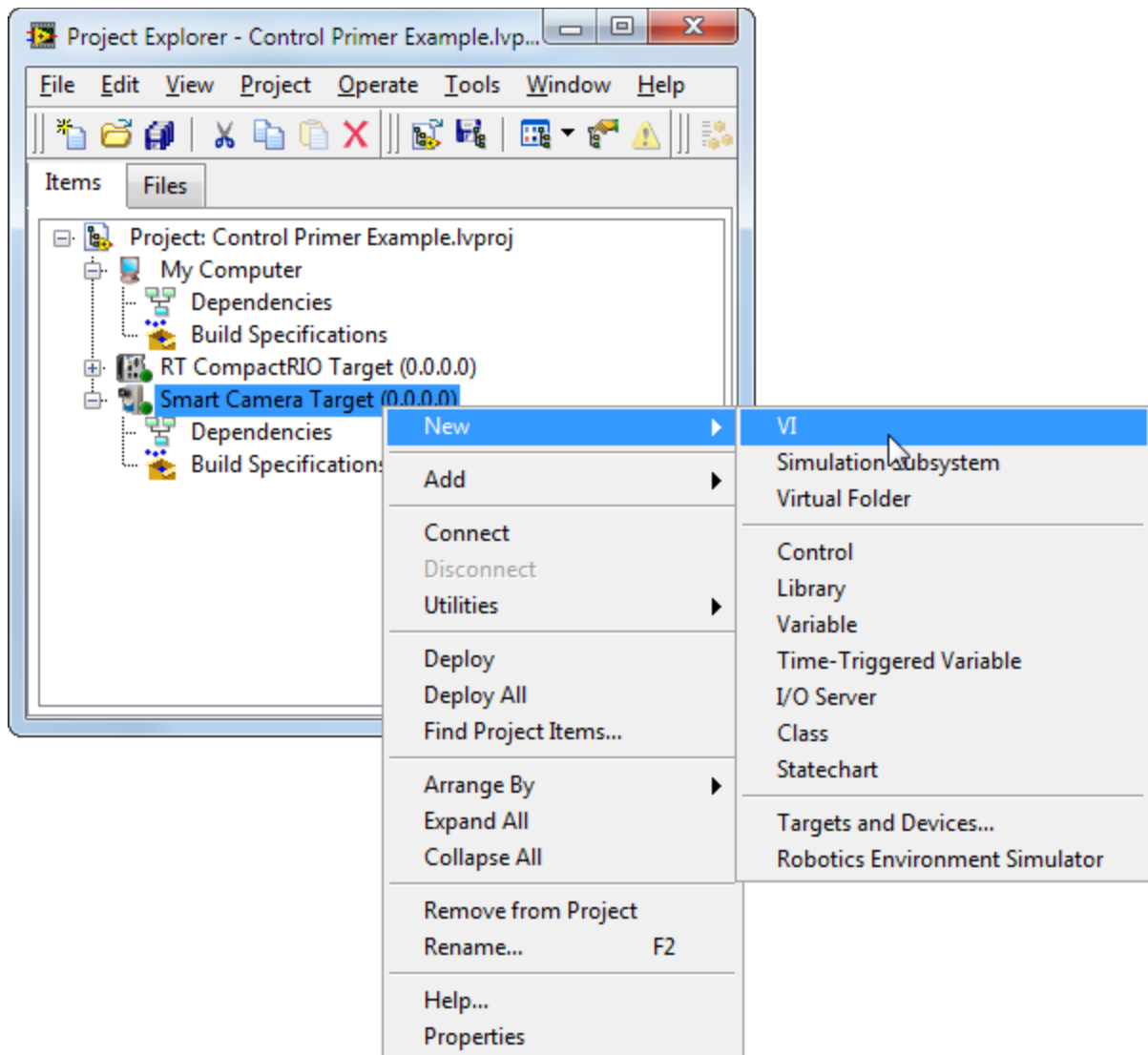


Figure 4.5. You can add NI Smart Camera systems to the same LabVIEW project as CompactRIO systems.

### Step 2. Use LabVIEW to Program the NI Smart Camera

Creating an application for the smart camera is almost identical to creating an application for a CompactRIO real-time controller. The main difference is using the NI Vision Acquisition driver to acquire your images and algorithms in the Vision Development Module in order to process them.

You can create a new VI and target it to the smart camera just as you have created VIs for CompactRIO.



**Figure 4.6. Adding a VI on Your NI Smart Camera to Your Project in LabVIEW Real-Time**

Upon deployment, this VI resides in smart camera storage and runs on the smart camera during run time.

To simplify the process of acquiring and processing images, NI includes Express VIs in the Vision palette. Use these Express VIs in this example to acquire images from the smart camera (or vision system) as well as process the images. To access these Express VIs, right-click on the block diagram and choose **Vision»Vision Express**.



Figure 4.7. The Vision Express Palette

The first step is to set up an acquisition from your camera. Or, if your camera is not available or not fixtured correctly yet, you also can set up a simulated acquisition by opening images stored on your hard drive.

Start by dropping the Vision Acquisition Express VI onto the block diagram. This menu-driven interface is designed so that you can quickly acquire your first image. If you have any recognized image acquisition hardware connected to your computer, it shows up as an option. If not, you have the option on the first menu to open images from disk.

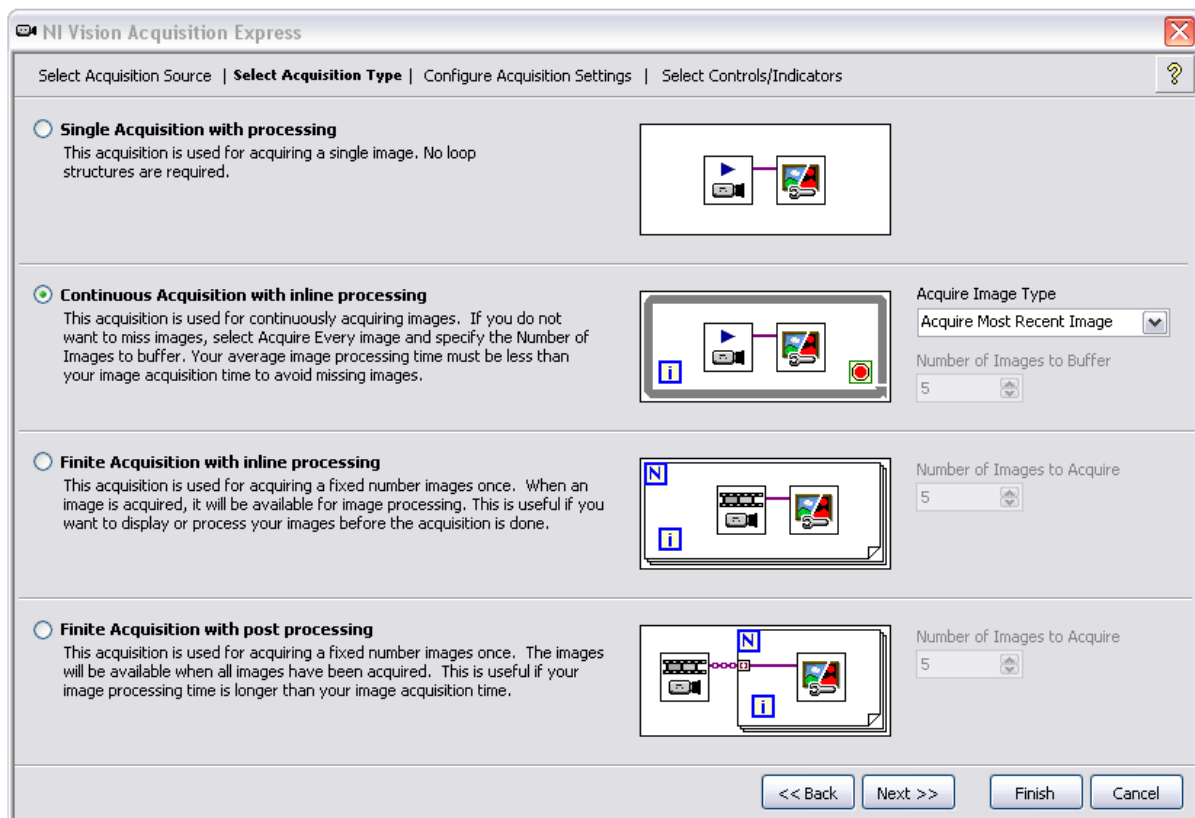


Figure 4.8. The Vision Acquisition Express VI guides you through creating a vision application in LabVIEW.

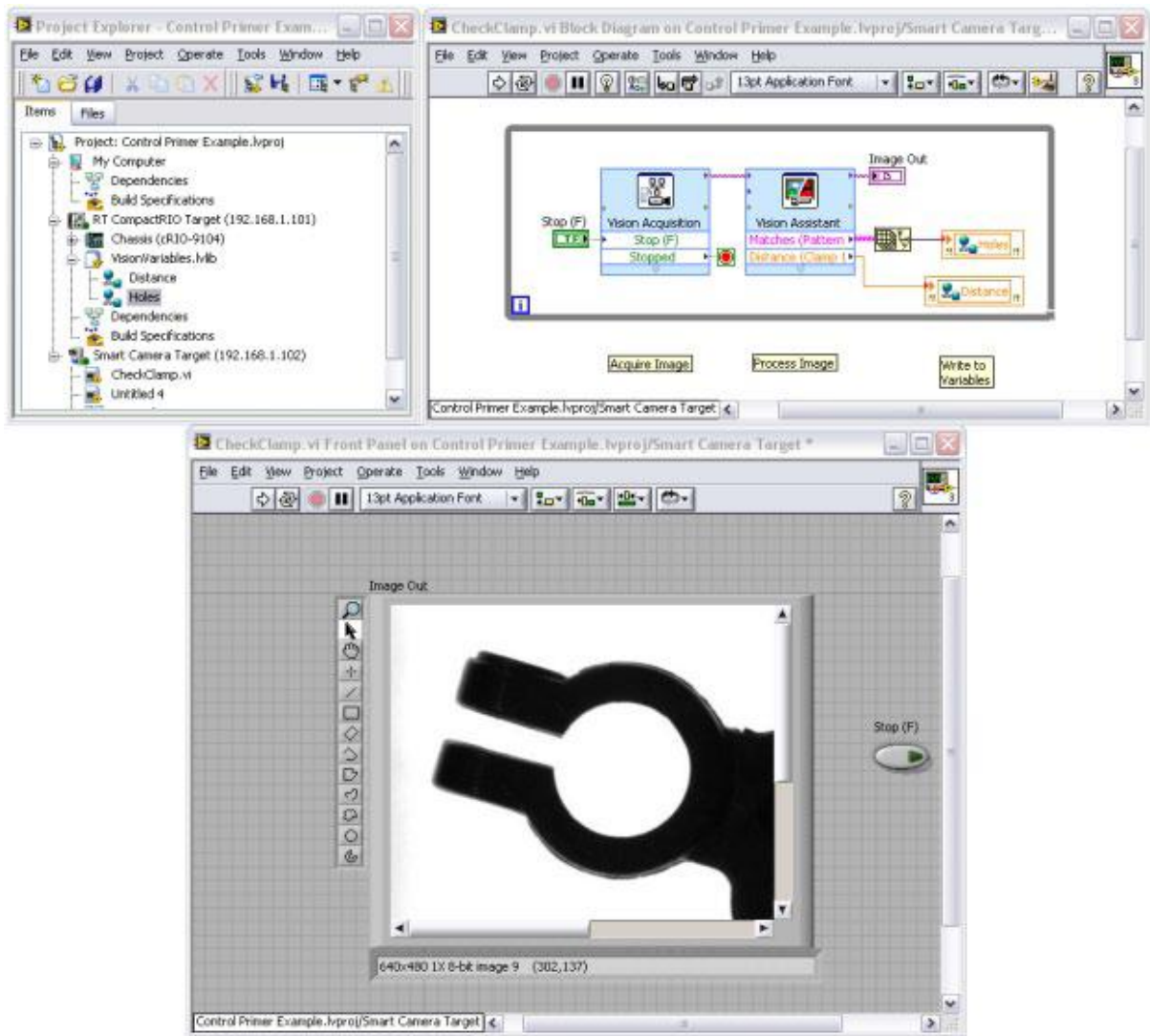
Next, choose which type of acquisition you are implementing. For this example, select Continuous Acquisition with inline processing so you can sit in a loop and process images. Next, test your input source and verify that it looks right. If it does, then click the finish button at the bottom.

Once this Express VI generates the LabVIEW code behind the scenes, the block diagram appears again. Now drop the Vision Assistant Express VI just to the right of the Vision Assistant Express VI.

With the Vision Assistant, you can prototype your vision processing quickly. You can deploy this same tool to real time systems, although traditional LabVIEW VIs are usually implemented to provide greater efficiency in the real-time systems. For an overview of the tools in this assistant, view the NI Vision Assistant Tutorial.

### **Step 3. Communicate With the CompactRIO System**

Once you have set up the machine vision you plan to conduct with the Vision Assistant Express VI, the last thing to do is communicate the data with the CompactRIO system. You can use network-published shared variables to pass data between the two systems. Network communication between LabVIEW systems is covered in depth in an earlier section in this document. In this example, you are examining a battery clamp, and you want to return the condition of the holes (if they are drilled correctly) and the current gap shown in the clamp.



**Figure 4.9. A Complete Inspection in LabVIEW**

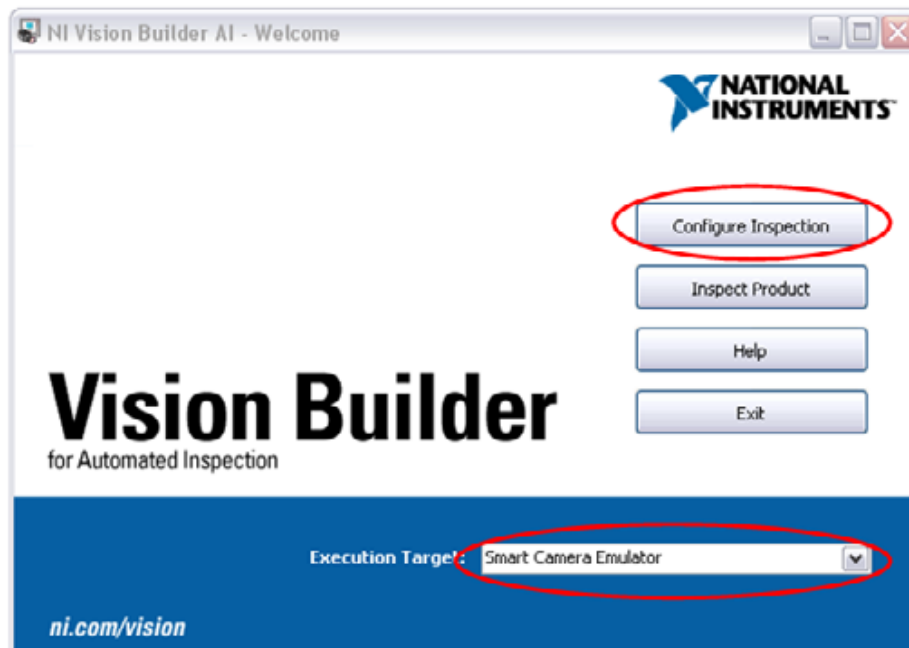
As you can see in Figure 4.9, the results of the inspection are passed as current values to the CompactRIO system via shared variables hosted on the CompactRIO system. You can also pass the data as commands to the CompactRIO system.

### **Machine Vision Using Vision Builder AI**

As mentioned previously, Vision Builder AI is a configurable environment for machine vision. You implement all of the image acquisition, image processing, and data handling through configurable menus.

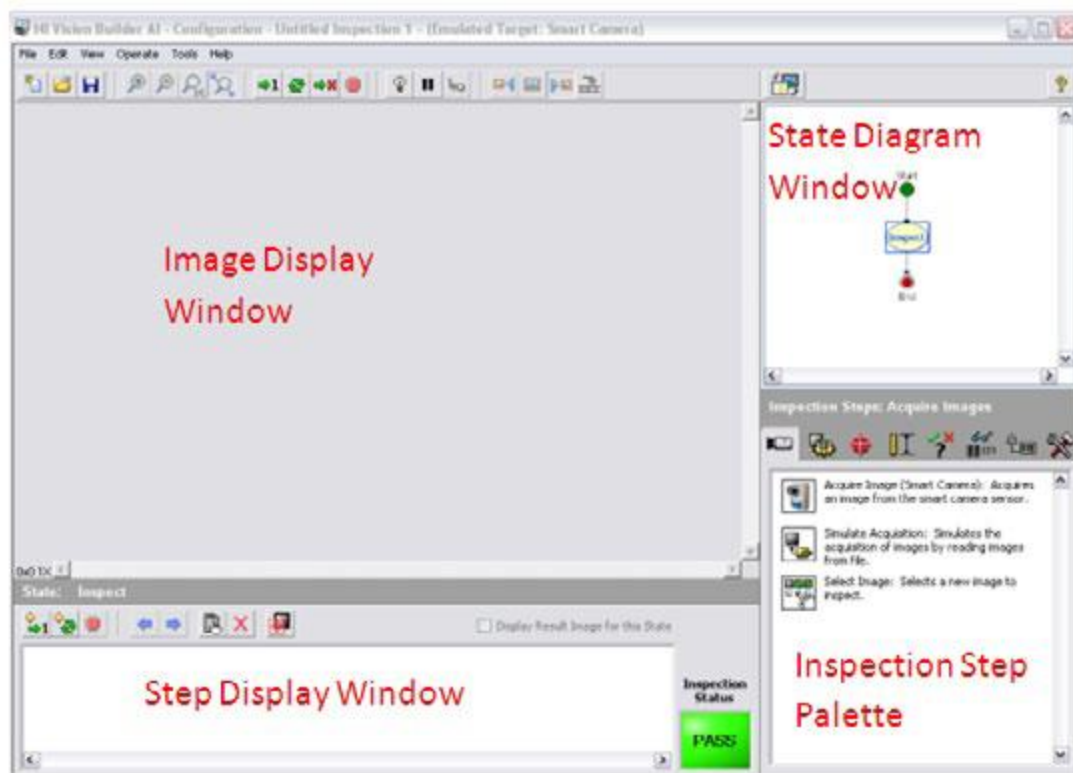
#### **Step 1. Configure an NI Smart Camera With Vision Builder AI**

When you open the environment, you see a splash screen where you can choose your execution target. If you do not have a smart camera connected, you can simulate a smart camera. Choose this option for this example.



**Figure 4.10.** Choose your execution target from the Vision Builder AI splash screen.

With this emulator, you can set up lighting and trigger options, arrange I/O to communicate to the CompactRIO hardware, and complete many of the other actions required to configure the smart camera without having one available for your development system. Once you have selected your execution target, click on **Configure Inspection**. This takes you into the development environment, which features four main windows. Use the largest window, the **Image Display Window**, to see the image you have acquired as well as any overlays you have placed on the image. Use the window to the upper right, the **State Diagram Window**, to show the states of your inspection (with your current state highlighted). The bottom right window, the **Inspection Step palette**, displays all the inspection steps for your application. Lastly, the thin bar at the bottom is the **Step Display Window**, where you see all of the steps that are in the current state.



**Figure 4.11. The Vision Builder AI Development Environment**

This example implements the same inspection as the LabVIEW example that inspected battery clamps and returned the number of holes and the gap of the clamp back to CompactRIO via the two shared variables hosted on the CompactRIO hardware.

## **Step 2. Configure the Inspection**

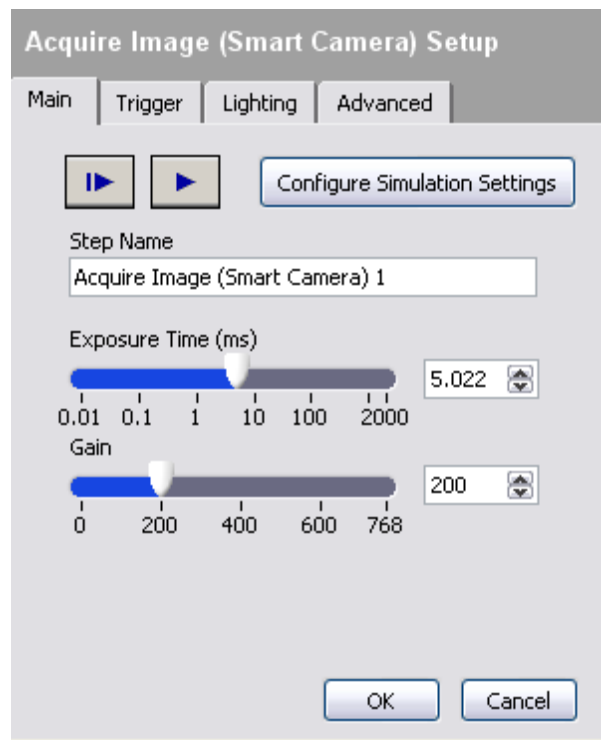
The first step of the inspection, just like in LabVIEW, is to acquire the image. Implement this with the Acquire Image (Smart Camera) step. Select this step and configure the emulation by clicking on the Configure Simulation Settings button. For example, set the image to grab the images from the following path:

**C:\Program Files\National Instruments\Vision Builder AI 3.6\DemoImg\Battery\BAT0000.PNG**

This library of images should install with every copy of Vision Builder AI (with differing version numbers). After selecting the file above, also make sure to check the box for Cycle through folder images.

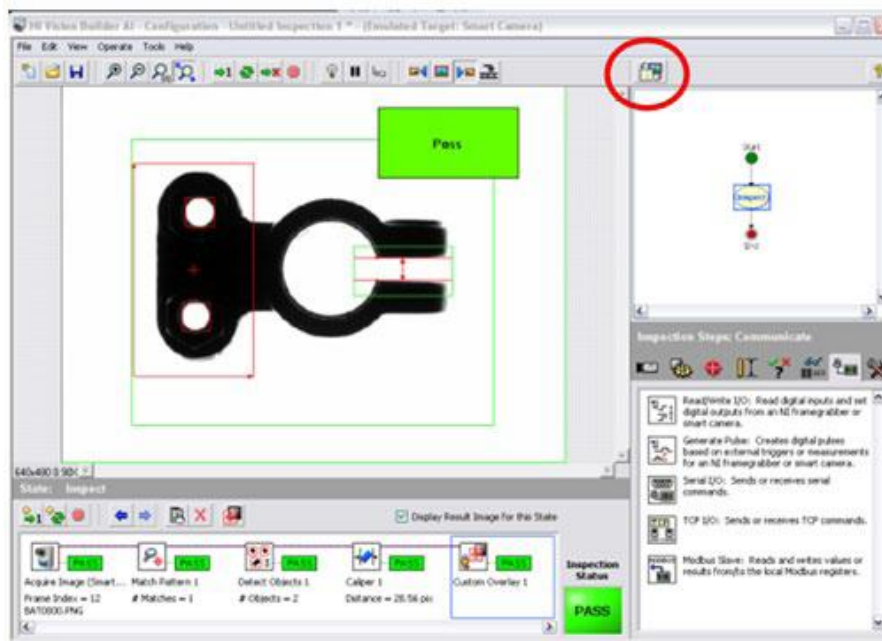
You also see that with this window, you can configure exposure times, gains, and other settings for the camera. These settings do not make any difference in

the emulator, but in the real world, they go hand-in-hand with lighting and optics choices.



**Figure 4.12. The Image Acquisition Setup Step**

From here, set up pattern matching, edge detection, object detection, code reading, or any other algorithms you need. For example, implement a pattern match to set the overall rotation of the object, a detect objects to see if both of the holes were in the clamp, and a caliper tool to detect the distance between the clamp prongs. This generates a few pass/fail results that you can use to set the overall inspection status, which you can display on the overlay to the user.



**Figure 4.13. A Completed Inspection in Vision Builder AI**

Now create two new states by clicking the toggle main window view button (circled in red). Create a pass state and a fail state. In both states, report the values back to CompactRIO but, in the fail state, also reject the part using some digital I/O.

### **Step 3. Communicate With the CompactRIO System**

Vision Builder AI provides access to many of the I/O types discussed previously, including network-published shared variables, RS232, Modbus, Modbus TCP, and raw TCP/IP. In this example, use the variable manager to access the shared variables hosted on CompactRIO. Access the variable manager by navigating to **Tools»Variable Manager**.

Here you can discover the variables on the CompactRIO system by navigating to the Network Variables tab. From there, select and add the variables and bind them to variables within the inspection.

Step Name  
Publish Variables

Variables

Name	Scope	Current Value	Operation	New Value
Distance	Inspection	0	Set to Caliper 1 - Distance (Pixel)	28.55705
Number of Holes	Inspection	0	Set to Match Pattern 1 - # Matches	1

Operation

☐ Do not Set  
☐ Set to Constant 0  
☒ Set to Measurement Caliper 1 - Distance (Pixel)  
☐ Increment  
☐ Decrement

Comment

Edit Variables OK Cancel

**Figure 4.14. Setting Up Variable Communication in Vision Builder AI**

Now these results are sent back to the CompactRIO system, and the inspection waits on the next camera trigger.

As you can see, both methods (programmable and configurable) offer the user a way to acquire images, process them, and then use the pertinent information to report results back to a CompactRIO control system or to directly control I/O from a real-time vision system.

## 5. RIADENIE POHONOV POMOCOU COMPACTRIO SYSTÉMOV

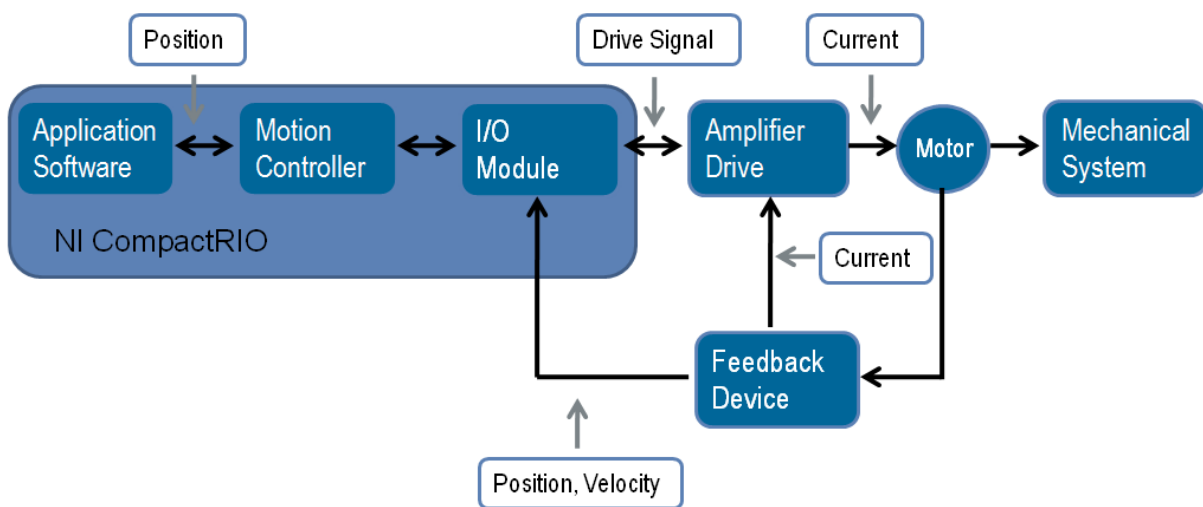
### Motion Control

This section examines precision motion control. Motor control is the on-off control or simple velocity control of rotary equipment such as fans and pumps. You can implement motor control with standard digital output modules to an appropriate motor starter or with an analog output to a Variable Frequency Drive (VFD). With specialized sensors, actuators, and fast control loops, you can perform precise position or velocity motion control, often on multiple axes. This section discusses the more sophisticated task of performing high-precision motion control on CompactRIO hardware.

A full motion control implementation is a complex system with several nested control loops—some running at high speeds—and precise mechanical components. A reliable, high-performance motion control system consists of the following devices:

1. Motion controller—This controller is the processing element that runs software algorithms and closed control loops to generate the motion profile commands based on the move constraints from the user-defined application software and I/O feedback.
2. Communication module—This I/O module interfaces with the drive and feedback devices. It converts the command signals from the motion controller to digital or analog values that the drive can interpret.
3. Drive/amplifier—The drive/amplifier consists of the power electronics that convert the analog or digital command signal from the communication module into the electrical power needed to spin the motor. In many cases, it also has a processing element that closes high-speed current control loops.
4. Motor—The motor converts the electrical energy from the drive/amplifier into mechanical energy. The motor torque constant,  $k_t$ , and the motor efficiency define the ratio between motor current and mechanical torque output.

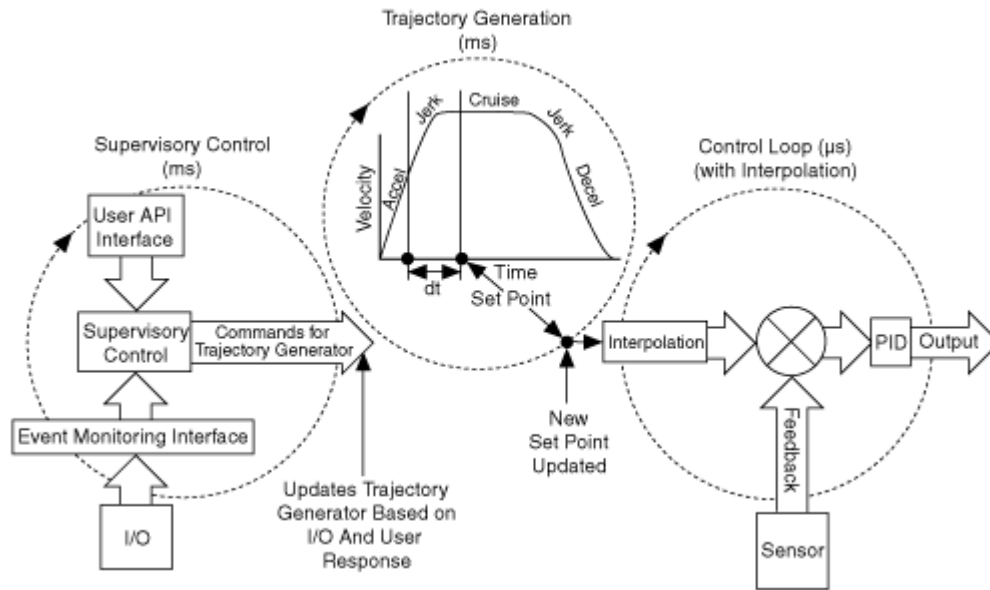
5. Mechanical transmission—The transmission consists of the components connected to the motor that direct the rotary motion of the motor to do work. This typically involves mechanical devices such as gearboxes or pulleys and lead screws that convert the rotary motion at the motor shaft to a linear movement at the payload with a certain transmission gear ratio. Common examples are belts, conveyors, and stages.
6. Feedback devices—These are sensors such as encoders and limit switches that provide instantaneous position and velocity information to the drive/amplifier and the motion controller



**Figure 5.1. Simplified Motion System Diagram for CompactRIO**

### The Motion Controller

The motion controller, the heart of the motion system, contains the motion control software that offers you the flexibility to create complicated multiaxis motion control applications. The motion controller consists of three cascaded control loops.



**Figure 5.2. Functional Architecture of NI Motion Controllers**

1. Supervisory control—This top control loop executes command sequencing and passes commands to the trajectory generation loops. This loop performs the following:
  - System initialization, which includes homing to a zero position
  - Event handling, which includes triggering outputs based on position or sensor feedback and updating profiles based on user-defined events
  - Fault detection, which includes stopping moves on a limit switch encounter, safe system reaction to emergency stop or drive faults, and other watchdog actions
2. Trajectory generator—This loop receives commands from the supervisory control loop and generates path planning based on the profile specified by the user. It provides new location setpoints to the control loop in a deterministic fashion. As a rule of thumb, this loop should execute with a 5 ms or faster loop rate.
3. Control loop—This is a fast control loop that executes every 50  $\mu$ s. It uses position and velocity sensor feedback and the setpoint from the trajectory generator to create the commands to the drive. Because this loop runs faster than the trajectory generator, it also generates

intermediate setpoints based on time with a routine called spline interpolation. For stepper systems, the control loop is replaced with a step generation component.

### **LabVIEW NI SoftMotion and NI 951x Drive Interface Modules**

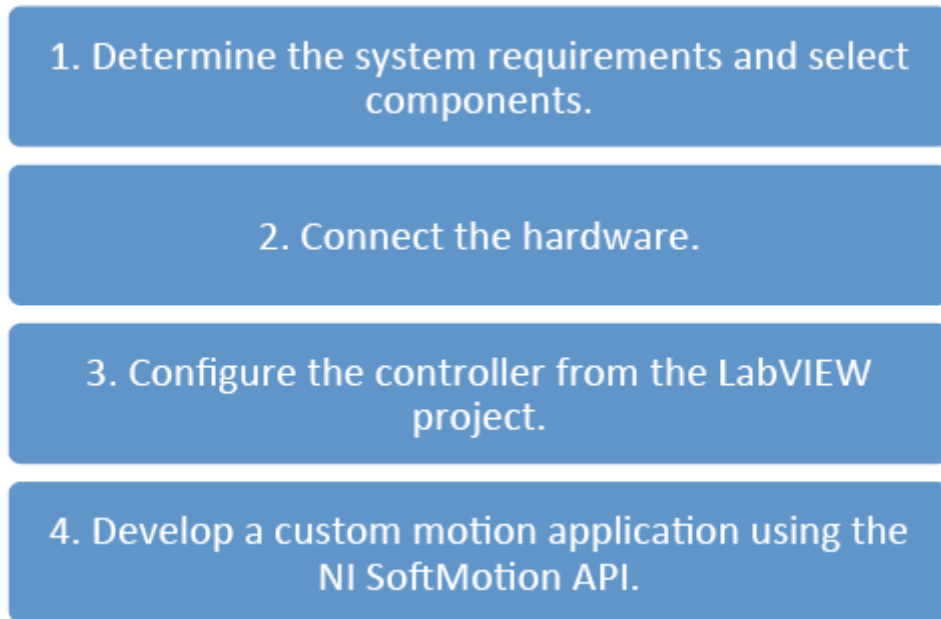
You can build all of the motion components from scratch using the LabVIEW Real-Time and LabVIEW FPGA modules, but NI also provides a software module and C Series I/O modules featuring prebuilt and tested components. The LabVIEW NI SoftMotion Module offers a high-level API for motion control programming and features the underlying motion architecture described previously as a driver service. The supervisory control loop and the trajectory generator run on the real-time processor on the CompactRIO controller. NI also offers C Series drive interface modules (NI 951x) that run the control loop and provide I/O to connect with drives and motion sensors. All of the loops and I/O modules are configured from the LabVIEW project.

For applications that need greater customization, such as a custom trajectory generator or a different control algorithm, LabVIEW NI SoftMotion offers software tools to open the code and customize these components with LabVIEW Real-Time and LabVIEW FPGA. For applications that require functionality or features that are not available with the NI 951x C Series modules, LabVIEW NI SoftMotion provides axis interface nodes so you can use other I/O modules or communication interfaces to third-party drives. In addition, LabVIEW NI SoftMotion enables virtual prototyping for motion applications and machine design by connecting to the SolidWorks Premium 3D CAD design application. With NI SoftMotion for SolidWorks, you can simulate your designs created in SolidWorks using the actual motion profiles developed with LabVIEW NI SoftMotion function blocks before incurring the cost of physical prototypes.

For more information, visit [ni.com/virtualprototyping](http://ni.com/virtualprototyping).

### **Getting Started With Motion on CompactRIO**

You can build a motion control application with CompactRIO in four basic steps.



**Figure 5.3. Four Steps for Building a Motion Control System**

### **Determine the System Requirements and Select Components**

Start by selecting the right mechanical components and motors for your system. One of the most common applications involves moving an object from one position to another. A typical way to change the rotary motion of a rotary motor to a useful linear motion is by connecting the motor to a stage and using a leadscrew mechanism to move the payload. Stages are mechanical devices that provide linear or rotary motion useful in positioning and moving objects. They come in a variety of types and sizes, so you can use them in many different applications. To find the correct stage for your application, you need to be familiar with some of the common terminology used when describing stages. Some of the key items to consider when selecting a stage include the following:

- **Transmission gear ration**—Determines the linear travel distance of the stage per rotary revolution of the motor.
- **Accuracy**—How closely the length of a commanded move compares to a standard length.
- **Resolution**: The smallest length of travel that a system is capable of implementing—can be as small as a few nanometers.

- **Travel distance**—The maximum length the system is capable of moving in one direction.
- **Repeatability**—The repeated motion to a commanded position under the same conditions. Often this is specified in unidirectional repeatability, which is the ability to return to the same point from one direction, and bidirectional repeatability, which specifies the ability to return from either direction.
- **Maximum load**—The maximum weight the stage is physically designed to carry, given accuracy and repeatability.

## **Stage Selection**

You can choose from a variety of stages to meet your application needs. You can narrow down these stages to two main types—linear and rotary. Linear stages move in a straight line and are often stacked on each other to provide travel in multiple directions. A three-axis system with an x, y, and z component is a common setup used to position an object anywhere in a 3D space. A rotary stage is a stage that rotates about an axis (usually at the center). Linear stages are used to position an object in space, but rotary stages are used to orient objects in space and adjust the roll, pitch, and yaw of an object. Many applications, such as high-precision alignment, require both position and orientation to perform accurate alignment. The resolution for a rotary stage is often measured in degrees or arc minutes (1 degree equals 60 arc minutes). Special types of stages include the goniometer, which looks like a linear stage that moves in an arc rather than a straight line, or a hexapod, which is a parallel mechanism that gives you movement in six axes to control—x, y, z, roll, pitch, and yaw. With a hexapod, you can define a virtual point in space about which the stage can rotate. While that is a benefit, the disadvantage is that hexapods are parallel and the kinematics involved are much more complex than those for simple stacked stages.

## **Backlash**

Another stage attribute to consider when choosing a stage for your precision motion system is backlash. Backlash is the lag created when one gear in a system changes direction and moves a small distance before making contact with a companion gear. It can lead to significant inaccuracies especially in

systems using many gears in the drive train. When moving on a nanometer scale, even a small amount of backlash can introduce major inaccuracy in the system. Effective mechanical design minimizes backlash but may not be able to eliminate it completely. You can compensate for this problem by implementing dual-loop feedback in software. The NI 9516 C Series servo drive interface module supports dual-encoder feedback and accepts feedback from two different sources for a single axis. To understand the circumstances for which you need to use this feature, consider a stage. If you monitor the stage position directly (as opposed to the position of the motor driving the stage), you can tell if a move you have commanded has reached the target location. However, because the motion controller is providing input signals to the motor and not to the stage, which is the primary source of feedback, the difference in the expected output relative to the input can cause the system to become unstable. To make the fine adjustments necessary to help the system stay stable, you can monitor the feedback directly from the encoder on the motor as a secondary feedback source. Using this method, you can monitor the real position of your stage and account for the inaccuracies in the drive train.

## **Motor Selection**

A motor provides the stage movement. Some high-precision stages and mechanical components feature a built-in motor to minimize backlash and increase repeatability, but most stages and components use a mechanical coupling to connect to a standard rotary motor. To make connectivity easier, the National Electrical Manufacturers Association (NEMA) has standardized motor dimensions. For fractional horsepower motors, the frame sizes have a two-digit designation such as NEMA 17 or NEMA 23. For these motors, the frame size designates a particular shaft height, shaft diameter, and mounting hole pattern. Frame designations are not based on torque and speed, so you have a range of torque and speed combinations in one frame size.

The proper motor must be paired with the mechanical system to provide the performance required. You can choose from the following four main motor technologies:

1. Stepper motor—A stepper motor is less expensive than a servo motor of a similar size and is typically easier to use. These devices are called stepper motors because they move in discrete steps. Controlling a

stepper motor requires a stepper drive, which receives step and direction signals from the controller. You can run stepper motors, which are effective for low-cost applications, in an open-loop configuration (no encoder feedback). In general, a stepper motor has high torque at low speeds and good holding torque but low torque at high speeds and a lower maximum speed. Movement at low speeds can also be choppy, but most stepper drives feature a microstepping capability to minimize this problem.

2. Brushed servo motor—This is a simple motor on which electrical contacts pass power to the armature through a mechanical rotary switch called a commutator. These motors provide a 2-wire connection and are controlled by varying the current to the motor, often through PWM control. The motor drive converts a control signal, normally a  $\pm 10$  V analog command signal, to a current output to the motor and may require tuning. These motors are fairly easy to control and provide good torque across their range. However, they do require periodic brush maintenance, and, compared to brushless servo motors, they have a limited speed range and offer less efficiency due to the mechanical limitations of the brushes.
3. Brushless servo motor—These motors use a permanent magnet rotor, three phases of driving coils, and Hall effect sensors to determine the position of the rotor. A specialized drive converts the  $\pm 10$  V analog signal from the controller into three-phase power for the motor. The drive contains intelligence to perform electronic commutation and requires tuning. These efficient motors deliver high torque and speed and require less maintenance. However, they are more complex to set up and tune, and the motor and drive are more expensive.
4. Piezo motor—These motors use piezoelectric material to create ultrasonic vibrations or steps and produce a linear or rotary motion through a caterpillar-like movement. Able to create extremely precise motion, piezo motors are commonly used in nanopositioning applications such as laser alignment. For high accuracy, they are often integrated into a stage or actuator, but you can also use piezo rotary motors.

Stage Drive Technology	Speed	Maximum Load	Travel Distance	Repeatability	Relative Complexity	Relative Cost
Stepper	Medium/Low	Medium/Low	High	Medium/Low	Low	Low
Brushed Servo	High	High	High	Medium	Medium	Low
Brushless Servo	Very High	High	High	High	High/Medium	High
Piezo	Medium	Low	Low	Very High	High	High

**Table 5.1. Comparison of Motor Technology Options**

Because the motor and drive are so closely coupled, you should use a motor and drive combination from one vendor. Though this is not a requirement, it makes tuning and selection easier.

### **Connect the Hardware**

Once you have selected the appropriate mechanics, motor, and drive for your application, you need to connect your CompactRIO system to the hardware. With the Axis Interface Node and LabVIEW FPGA, you can use any C Series module to connect to the CompactRIO system; however, for simplicity, NI recommends using the NI 951x C Series drive interface modules.

### **NI 951x Drive Interface Modules**

NI 951x modules provide servo or stepper drive interface signals for a single axis and can connect to hundreds of stepper and servo drives. In addition to supplying the appropriate I/O for motion control, the modules feature a processor to run the spline interpolation engine with a patented NI step generation algorithm or the servo control loop.

- The NI 9512 is a single-axis stepper or position command drive interface module with incremental encoder feedback.
- The NI 9514 is a single-axis servo drive interface module with incremental encoder feedback.
- The NI 9516 is a single-axis servo drive with dual-encoder feedback.

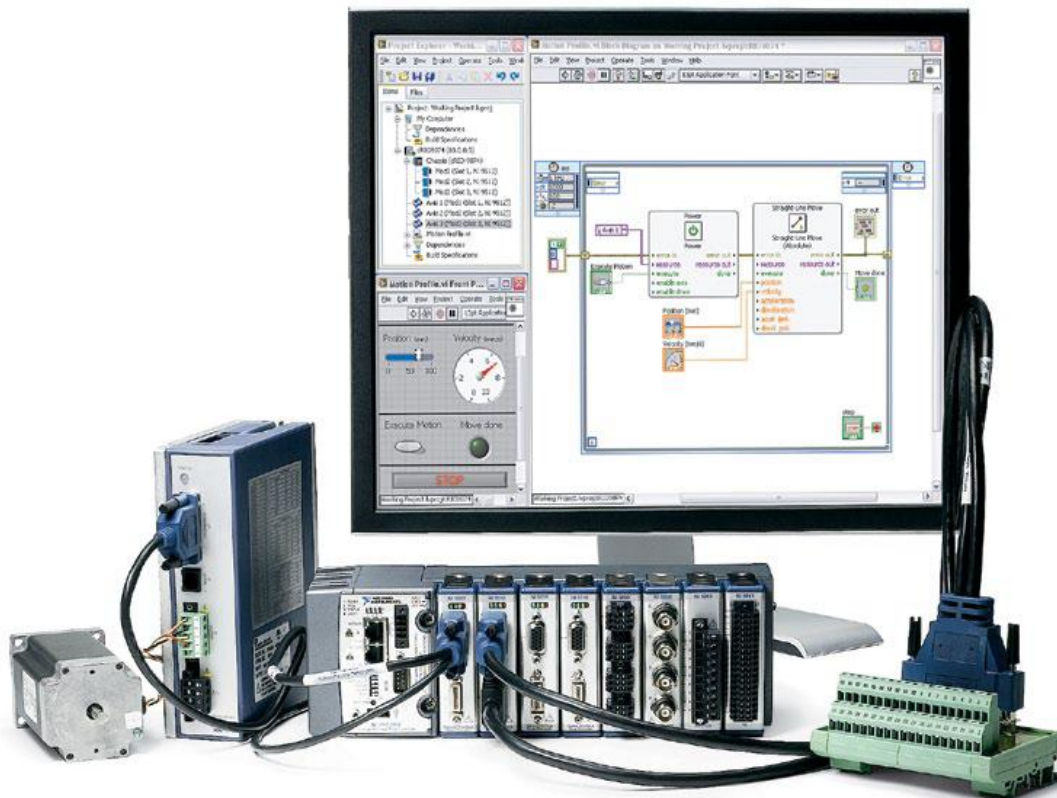
When run in Scan Mode, these modules must be used in any of the first four slots in a CompactRIO chassis. With LabVIEW FPGA, you can use them in any slot.

NI 951x modules were designed to simplify wiring by providing the flexibility to hook up all the I/O for motion control to a single module. To further ease connectivity, the digital I/O is software configurable to connect to either sinking or sourcing devices. The modules offer the following:

- Analog or digital command signals to the drive
- Drive enable signal; software configurable as sinking or sourcing (24 V)
- DI signals for home, limits, and general digital I/O; all software configurable as sinking or sourcing (24 V)
- Encoder inputs and 5 V supply; configurable for single ended or differential
- DI/DO for high-speed position capture/compare functions (5 V)
- LEDs to provide quick debugging for encoder states, limit status, and axis faults

To further simplify wiring, NI offers several options for connecting NI 951x drive interface modules to external stepper drives or servo amplifiers including the following:

- NI 9512-to-P7000 Stepper Drives Connectivity Bundle—Connects the NI 9512 to the P70530 or P70360 stepper drives from NI.
- NI 951x Cable and Terminal Block Bundle—Connects an NI 951x module with 37-pin spring or screw terminal blocks.
- D-SUB and MDR solder cup connectors—Simplifies custom cable creation.
- D-SUB to pigtails cable and MDR to pigtails cable—Simplifies custom cable creation.



**Figure 5.4. Choose from several options to simplify connecting NI 951x modules directly to drives.**

### **Configure the Controller From the LabVIEW Project**

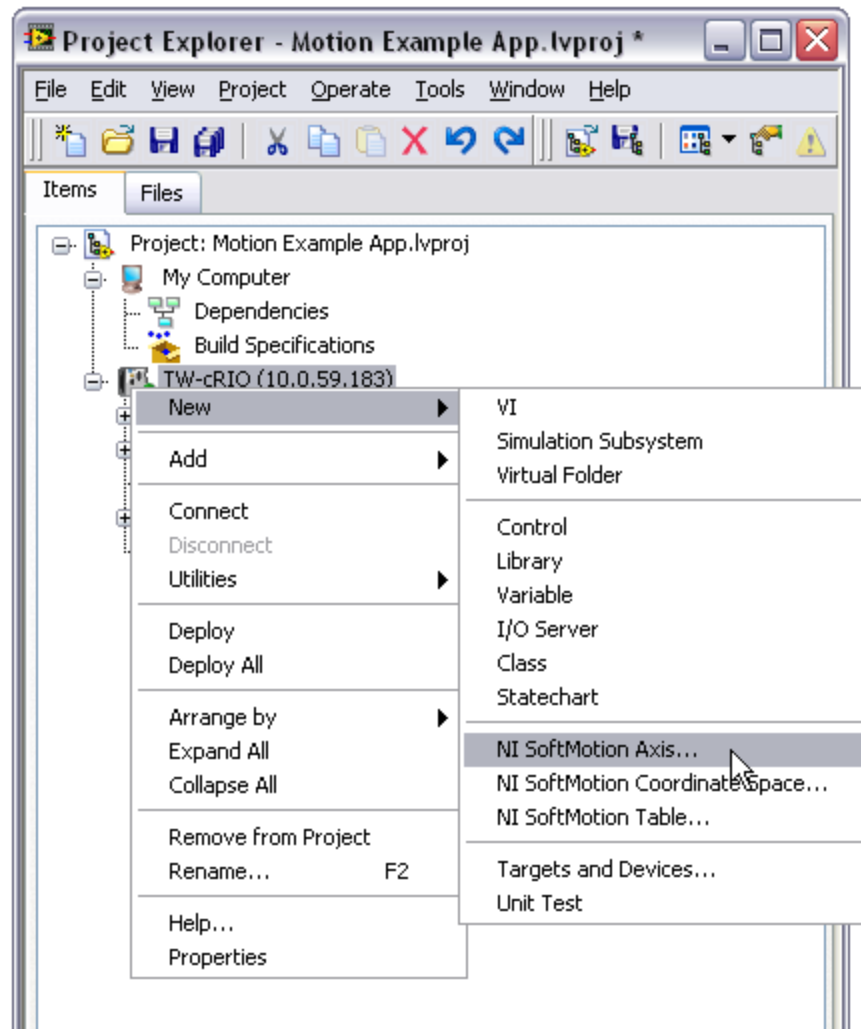
To use LabVIEW NI SoftMotion in LabVIEW, you must first create axes, coordinates, and tables in the LabVIEW project. When you create these items, you associate a logical channel with the physical motion resources and instantiate background control loops to run on the controller. You use the logical channels you create when developing motion applications using the NI SoftMotion LabVIEW API.

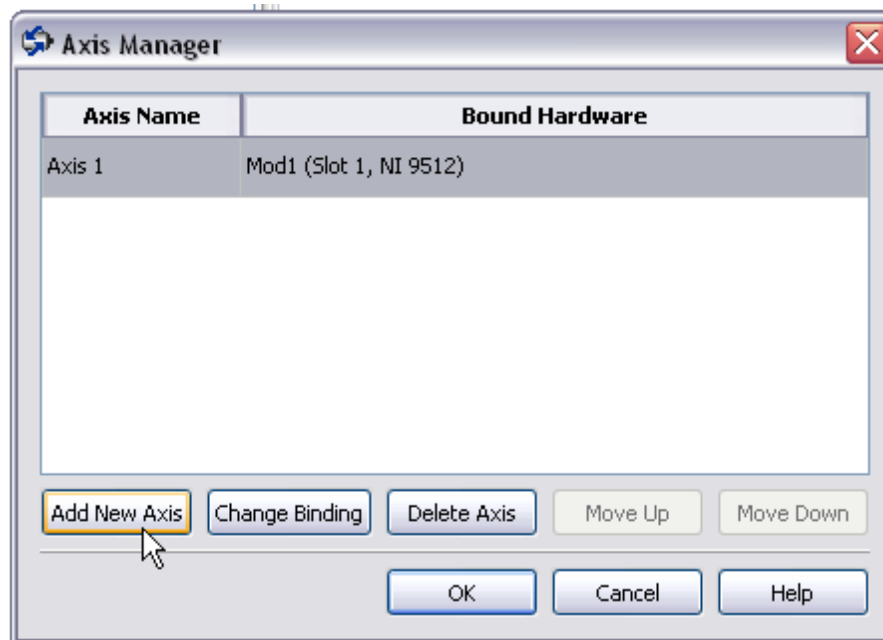
- An axis is a logical channel associated with an individual motor. Each motor you are controlling must be in an axis.
- A coordinate is a grouping of one or more axes. By planning for multiple axes in a coordinate, you can create multiaxis coordinated motion. For instance, if you have an XY stage and you want to create an oval, it is difficult to command the two motors individually. But if you group them as an axis, the LabVIEW NI SoftMotion trajectory generator automatically develops the commanded points to each axis so the resulting motion is an oval.

- A table is used to specify more complex move profiles such as contour moves and camming. You can import the move profile from an external tab delimited text file.

### Adding Axes, Coordinates, and Tables to the Project

You can add axes, coordinates, and tables by right-clicking on the CompactRIO controller in the LabVIEW project and selecting New.





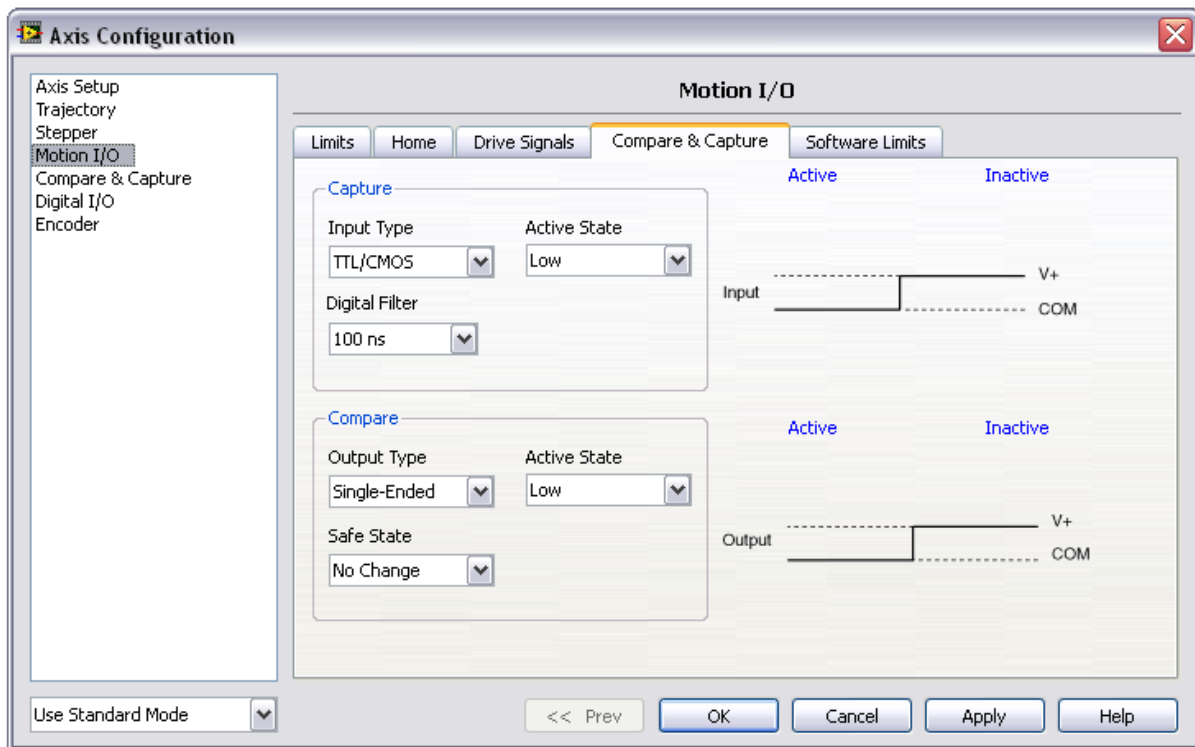
**Figure 5.5. Adding an Axis to a CompactRIO System**

An axis consists of a trajectory generator, PID control loop or stepper output, and supervisory control. You can associate a LabVIEW NI SoftMotion axis with simulated hardware or with actual hardware. Servo axes require an encoder feedback resource. Open-loop stepper axes do not require feedback for operation.

Once you have created axes, you can generate coordinates and add axes to the coordinates. When using coordinate resources in LabVIEW, you receive target positions and other coordinate information in a 1D array with axis information arriving in the order that axes are added using this dialog box.

### **Configuring the Axes**

Once you add an axis to the project, you need to configure the axis by right-clicking on it and selecting properties.



**Figure 5.6.** You can use the Axis Configuration feature to configure all of the motion I/O parameters.

To configure a stepper drive connected to the NI P7000 series stepper drive, follow these steps:

1. Right-click the axis in the LabVIEW Project Explorer window and select Properties from the shortcut menu to open the Axis Configuration dialog box.
2. On the Axis Setup page, confirm that Loop Mode is set to Open-Loop. Axes configured in open-loop mode produce step outputs but do not require feedback from the motor to verify position.
3. Also on the Axis Setup page, confirm that the Axis Enabled and Enable Drive on Transition to Active Mode checkboxes contain checkmarks. These selections configure the axes to automatically activate when the Scan Engine switches to active mode.
4. If the modules do not have physical limit and home input connections, you must disable these input signals for proper system operation. To disable limits and home, go to the Motion I/O page and remove the checkmarks from the Enable checkboxes in the Forward Limit, Reverse Limit, and Home sections.

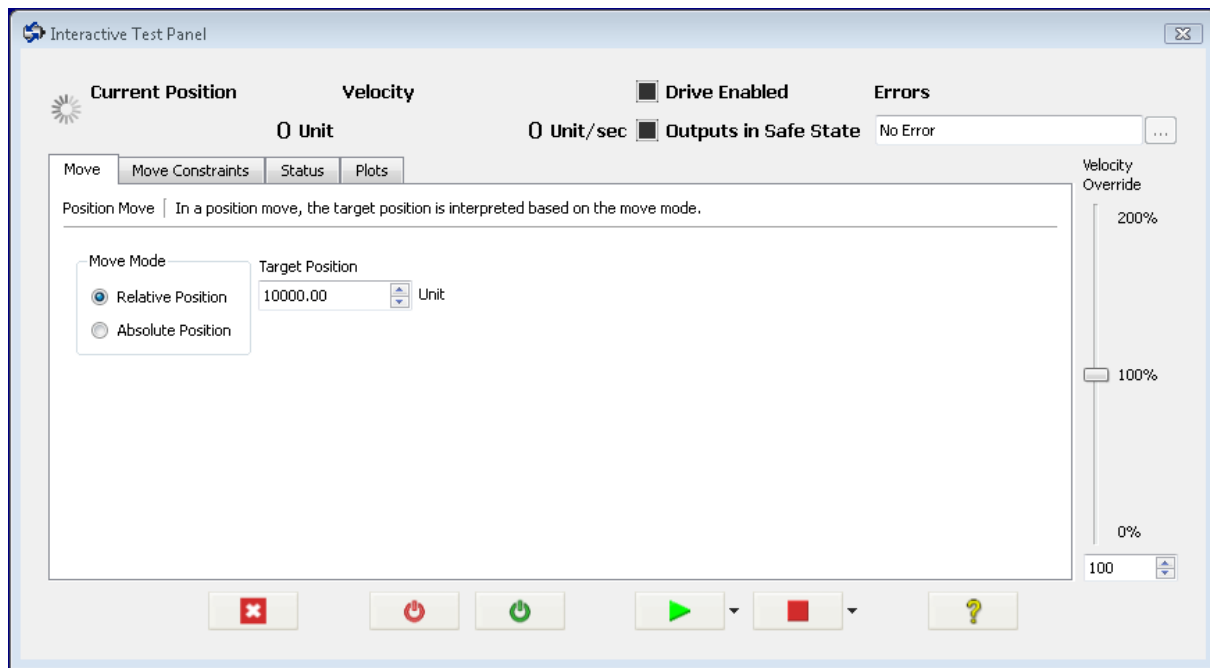
5. Configure any additional I/O settings according to your system requirements.
6. Click OK to close the Axis Configuration dialog box.
7. Right-click the controller item in the LabVIEW Project Explorer window and select Deploy All from the shortcut menu to deploy.

**Note:** Make sure all hardware connections are made and power is turned on before deploying the project. Deployment switches the Scan Engine to active mode and enables your axes and drive, if connected, so that you can start a move immediately.

### **Test the Motion System**

To make sure your motion is configured and connected correctly, you can use the Interactive Test Panel to test and debug your motion system. With the Interactive Test Panel, you can perform a simple straight-line move and monitor move and I/O status information, change move constraints, obtain information about errors and faults in the system, and view the position or velocity plots of the move. If you have a feedback device connected to your system, you can also obtain feedback position and position error information.

To start the Interactive Window, right-click the axis in the LabVIEW Project Explorer window and select Interactive Test Panel from the shortcut menu. Set the desired position, move mode, and move constraints using the tabs.



**Figure 5.7.** With the Interactive Test Panel, you can verify a motion configuration before writing code.

Click the Start button on the bottom of the dialog box to start the move with the configured options. Use the Status and Plots tabs to monitor the move while it is in progress.

### **Develop Custom Motion Applications Using the LabVIEW NI SoftMotion API**

LabVIEW NI SoftMotion provides a function block API to build deterministic motion control applications. These function blocks, which are inspired by the PLCopen motion function blocks, use the same terminology and execution paradigm as IEC 61131-3 function blocks. While function block programming is similar to LabVIEW data flow, you need to know about some execution differences before building applications using the function blocks.

The function blocks themselves do not run any motion algorithms. Instead, they are an API that sends commands to the motion manager that runs as a driver service on the CompactRIO controller. The motion manager runs at the scan rate, but you can run the function block API at any rate you want and even call the blocks in nondeterministic code.

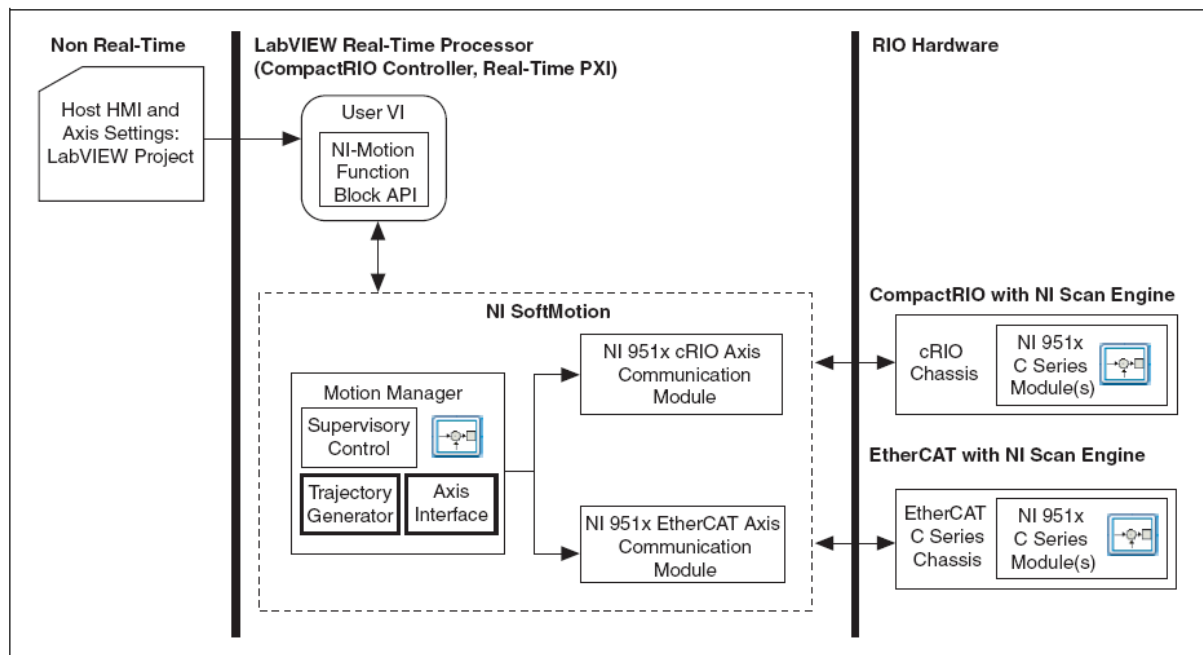


Figure 5.8. Block Diagram of LabVIEW NI SoftMotion Components on a CompactRIO System

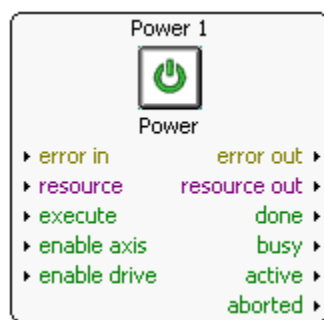


Figure 5.9. LabVIEW NI SoftMotion Function Block

Motion function blocks are an API to the motion manager running on the CompactRIO system. They perform two actions:

1. **Send a command to the motion manager**—The function block sends the command to the manager when the “execute” input transitions from low to high (rising edge). The default value is false so if a true constant is wired into the block, the first iteration counts as a rising edge.
2. **Poll the motion manager**—Every iteration, the function block polls the manager to see if the command was executed successfully. The results of the poll are returned through the “done,” “busy,” “active,” “aborted,” and “error out” outputs.

The error out, done, aborted, busy, and active outputs behave according to the following guidelines:

Output exclusivity	Busy, done, error out, and aborted are mutually exclusive and set at any given time. Only one of these outputs can be TRUE at any time on one function block. If the execute input is TRUE, one of these outputs must be TRUE.
Output status	The done and aborted outputs are reset with the falling edge of execute (state latches while execute is high). However, the falling edge of execute does not stop or influence the execution of the motion manager once it has received the command.
Behavior of done output	The done output is set TRUE when the commanded action successfully completes. The default value is FALSE. Once a move has completed, it is reset to FALSE when the execute is false.
Behavior of aborted output	Aborted is set when a commanded operation is interrupted by another command. The reset behavior of aborted is like that of done.
Behavior of busy output	Every function block has a busy output that indicates that the function block operation is not complete. Busy is set at the rising edge of execute and resets when either done, aborted, or error out is set. It is recommended that the application controlling the execution of this function block not be terminated for at least as long as busy is TRUE because the software may be left in an undefined state.
Output active	The active output is set to TRUE at the moment the function block takes control of the specified resource, all properties have been committed, and the function block has executed.

## Using LabVIEW NI SoftMotion Function Blocks

The following tips may help you when programming with LabVIEW NI SoftMotion function blocks in LabVIEW:

#### Function blocks

- Are nonblocking and always execute in a defined period of time.
- Are triggered to pass commands to the motion manager based on a rising edge of the “execute” input.
- Provide feedback if the motion manager successfully finishes the commanded task through a “done” output.
- Are instanced (have a unique memory space) and reentrant, but each instance can be called from only one location in the program.
- Must be executed in a VI that is part of a LabVIEW project.
- Have a double-click dialog that you can use to configure default values, automatically bind data to variables, and configure the data source as terminal, variable, or default.
- Have methods exposed on the right-click menu. For instance, the “Stop Move” function block can be either decelerate, immediate, or disable drive.
- Must always be run in a loop. Depending on your application requirements, you can use either a While Loop timed using a Wait Until Next ms Multiple Function or you can use a Timed Loop.

You should use the function block status inputs and outputs (execute, done, and so on)—not standard LabVIEW programming methods—to determine the order of function block execution. For example, do not place function blocks inside a Case structure unless the Case structure is controlled by the status outputs. Consider the Figure 5.4 block diagram:

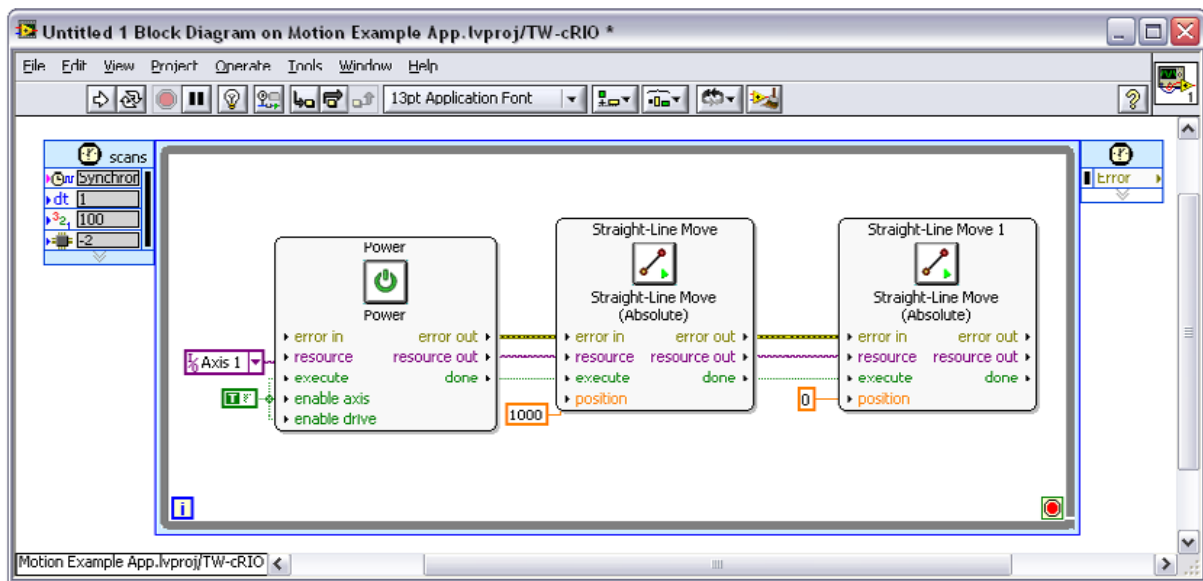


**Figure 5.10. Incorrect Example of Programming Motion Using Motion Function Blocks**

If you are not familiar with function blocks, you likely expect this code to move Axis 1 to position 1000 and then back to 0. However, these function blocks are not actually performing the motion—they are simply sending commands to the motion manager when they see a rising edge on their “execute” inputs.

Instead this code enables the drive and axis, but no motion takes place.

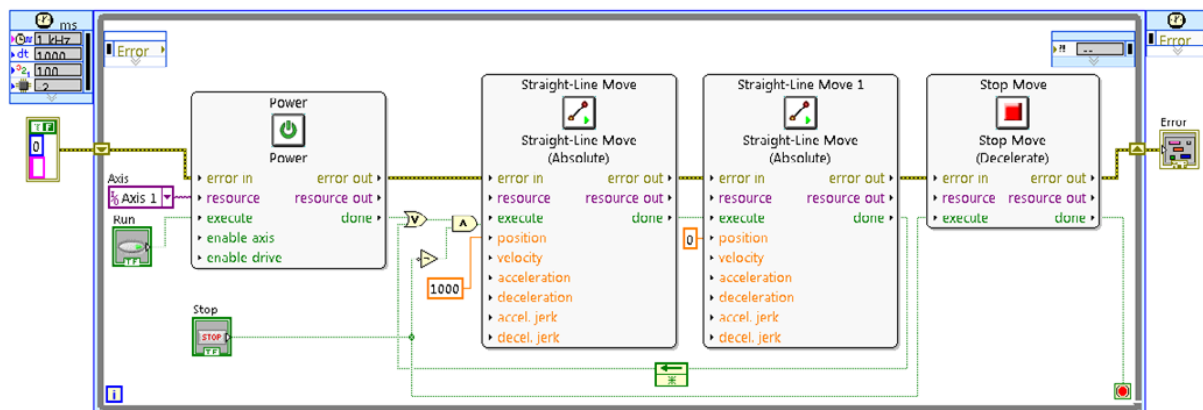
- The Power function sees a rising edge on the execute input (the default is false, so if you wire a true, the first iteration executes).
- When it receives a rising edge, it sends a command to the motion manager to enable the axis and drive.
- This nonblocking function does not wait for the manager to finish the command before continuing the LabVIEW code, and the “done” output is false on the first iteration.
- All remaining blocks in the code chain do not see a rising edge on their execute inputs and do not send any commands to the manager.
- No motion takes place.



**Figure 5.11. Example of a Working Program Using Motion Function Blocks**

In this example, the drive is enabled, moves to position 1000, and moves to position 0, but it does not perform repeated moves. This is because each function block receives a rising edge on the execute input only once.

To make the axis cycle repeatedly between 1000 and 0, you need to write code like that shown in Figure 5.12.

















**Figure 5.12. Correct Example of Programming Repeated Motion Using Motion Function Blocks**

This code causes repeated motion between position 1000 and 0.

- The loop with the Power function block sends a command to the motion manager.

- It then polls the manager to check the status of the command. Once it gets confirmation that the command was finished, it writes a true to the done bit and exits the first loop.
- In the Timed Loop, the first straight-line move function block sees a true on the execute terminal and commands the manager to move to position 1000.
- The done output is false until the move has completed, and the second Straight-Line Move function block does nothing because it has a false in the execute input.
- At each iteration of the loop, the first straight-line move polls the manager to see if the move completed. When the first move is complete, it outputs a true on the done terminal.
- The second straight-line move sees a low-to-high transition on the execute bit and sends its command to the manager.
- When this block receives confirmation that it completed, it transitions its done terminal from false to true.
- Through the shift register, this causes the first function block to see a low-to-high transition on the execute terminal and the moves repeat.
- If a stop command is ever executed, the clean-up code with the stop command stops the motion. Because the motion manager is a separate process, without this stop command to the manager the move continues until completion even though the LabVIEW code is no longer executing.

Palette Object	Palette Symbol	Description
Line		Performs a straight-line move using an axis or coordinate resource. A straight-line move connects two points using one or more axes. The behavior of the move changes based on the Straight-Line Move Mode.
Arc		Performs a circular, spherical, or helical arc move. An arc move produces motion in a circular shape using a radius you specify. The type of arc to perform changes based on the Arc Move Mode.
Contour		Performs a contour move using an axis or coordinate resource. A contour move is a move expressed as a series of positions that the software uses to extrapolate a smooth curve. These positions are stored in a table. Each point in the move is interpreted as an absolute position using the starting point of the move as a temporary "zero" position. The type of contour move changes based on the Contour Mode.
Reference		Performs a reference move, such as locating a home or limit position, on an axis resource. Reference moves are used to initialize the motion system and establish a repeatable reference position. The behavior of the move changes based on the Reference Move Mode.
Capture		Records encoder position based on an external input, such as the state of a sensor. You can use the captured position to execute a move relative to a captured position, or simply record the encoder position when the capture event occurs.
Compare		Synchronizes the motor with external activities and specified encoder positions. When the specified position is reached, a user-configurable pulse is executed. The behavior of the position compare operation changes based on the Compare Mode.
Gearing		Configures the specified axis for gearing operations. Gearing synchronizes the movement of a slave axis to the movement of a master device, which can be an encoder or the trajectory of another axis. The movement of the slave axes may be at a higher or lower gear ratio than the master. For example, every turn of the master axis may cause a slave axis to turn twice. The type of gearing operation to perform changes based on the Gearing Mode.
Camming		Configures the specified axis for camming operations. These ratios are handled automatically by LabVIEW NI SoftMotion, allowing precise switching of the gear ratios. Camming is used in applications where the slave axis follows a nonlinear profile from a master device. The type of camming operation changes based on the Camming Mode.
Read		Reads status and data information from axes, coordinates, feedback, and other resources. Use the read methods to obtain information from different resources.
Write		Writes data information to axes, coordinates, or feedback resources. Use the write methods to write information to different resources.
Reset Position		Resets the position on the specified axis or coordinate.
Stop		Stops the current motion on an axis or coordinate. The behavior of the move changes based on the Stop Mode.
Power		Enables and disables axes and/or drives on the specified axes or coordinate resources.
Clear Faults		Clears LabVIEW NI SoftMotion faults.

**Table 5.2. Motion Function Block Overview**

## LabVIEW Example Code



LabVIEW example code is provided for this section.

A state machine is a common programming architecture for motion applications. However, while a state machine increases the application

flexibility, some additional caveats are introduced when using motion function blocks in a state machine.

In sequential programming, you transition between motion commands based on the function block error, aborted, and done outputs. (The busy output is a logical combination of these other states, so for simplified programming, you can also monitor the busy output.) In state machine programming, you may have parallel states or events that affect the motion and cause states to exit before a move is complete.

For instance, you may want to build logic where a stop command can be sent to the system. If a stop state is running in parallel to the motion state, the motion manager stops the move and the motion state returns a True on the aborted output.

You may also structure your code to exit without waiting for the abort command to be returned through the function block, for example, when you are writing an ESTOP state into your program. In most application designs, the ESTOP immediately quits all other states and transitions into a safe emergency stop state. In this case, because the function block did not exit cleanly, you need to reset it before execution. A function block is reset when a false is written to the execute input.

To address this problem, you need to ensure that the function block had a false written to the execute input. You can choose from several methods to complete this in LabVIEW, including adding logic to maintain state data for each motion state to ensure that the function block is always cleared before execution. Or you can simply always write a false to the execute input on the first iteration of a state. This delays your motion execution by one cycle.

Consider an example featuring a simple state machine with four states: idle, initialize, move, and stop. In each state, the busy output is used to determine if the function block has completed.

You can transition the stop move from any state. This means the stop state can leave the move or initialize state during execution and cause the function block to require a reset before you can execute it again. To accomplish this, the state machine is set so each function block receives a false on the execute input on

the first iteration and a true every subsequent iteration. This automatically cleans up a function block that was aborted.

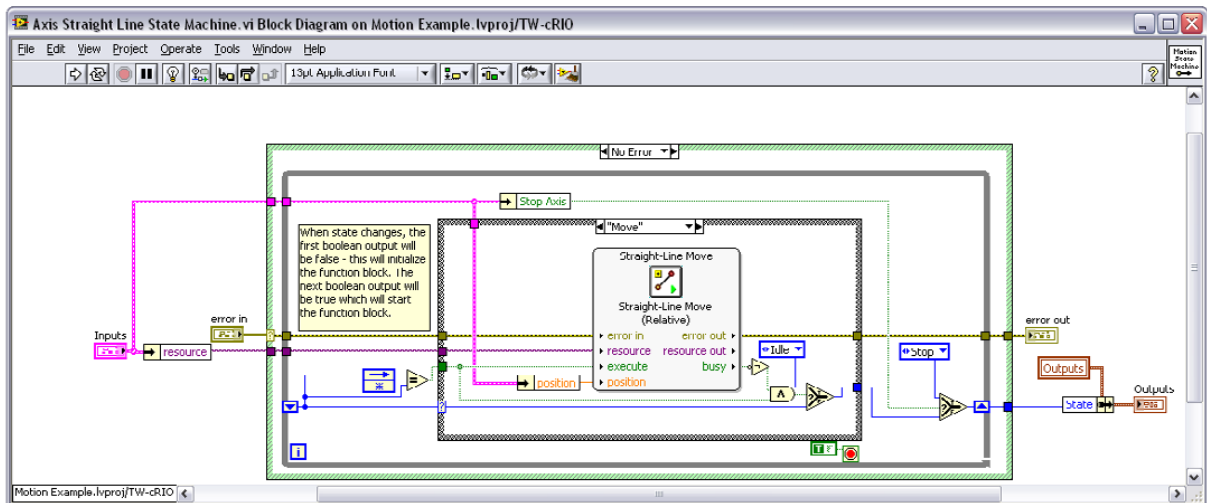


Figure 5.13. A State Machine With Motion Function Blocks

You can place this state machine in your standard CompactRIO controller architecture with initialization and shutdown states.

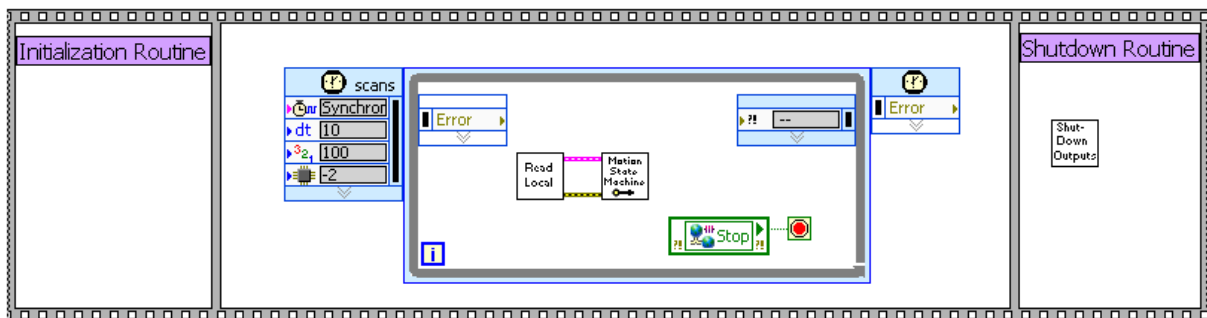


Figure 5.14. You can drop a state machine with motion commands into the standard controller architecture.

## 6. NASADENIE A REPLIKÁCIA SYSTÉMOV

### **Application Deployment**

All LabVIEW development for real-time targets and touch panel targets is done on a Windows PC. To run the code embedded on the targets, you need to deploy the applications. Real-time controllers and touch panels, much like a PC, have both volatile memory (RAM) and nonvolatile memory (hard drive). When you deploy your code, you have the option to deploy to either the volatile memory or nonvolatile memory on a target.

#### **Deploy to Volatile Memory**

If you deploy the application to the volatile memory on a target, the application does not remain on the target after you cycle power. This is useful while you are developing your application and testing your code.

#### **Deploy to Nonvolatile Memory**

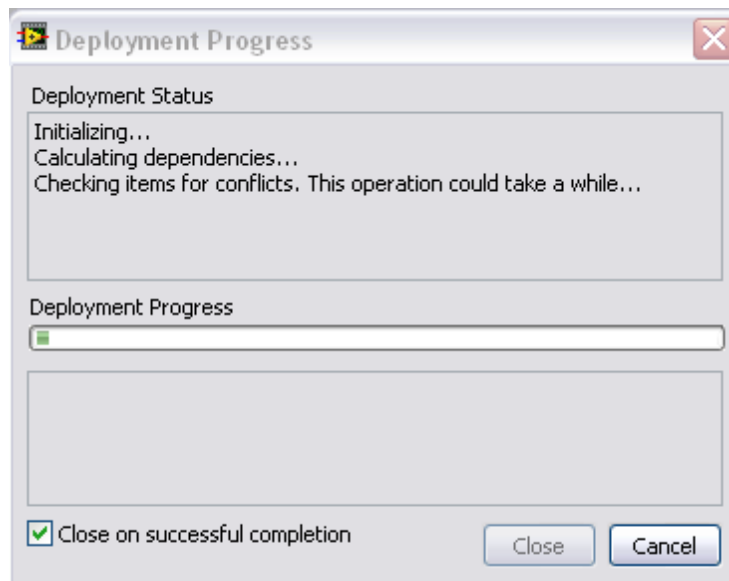
If you deploy the application to the nonvolatile memory on a target, the application remains after you cycle the power on the target. You also can set applications stored on nonvolatile memory to start up automatically when the target boots. This is useful when you have finished code development and validation and want to create a stand-alone embedded system.

#### **Deploying Applications to CompactRIO**

**Deploy a LabVIEW VI to Volatile Memory** When you deploy an application to the volatile memory of a CompactRIO controller, LabVIEW collects all of the necessary files and downloads them over Ethernet to the CompactRIO controller. To deploy an application you need to

- Target the CompactRIO controller in LabVIEW
- Open a VI under the controller
- Click the Run button

LabVIEW verifies that the VI and all subVIs are saved, deploys the code to the nonvolatile memory on the CompactRIO controller, and starts embedded execution of the code.



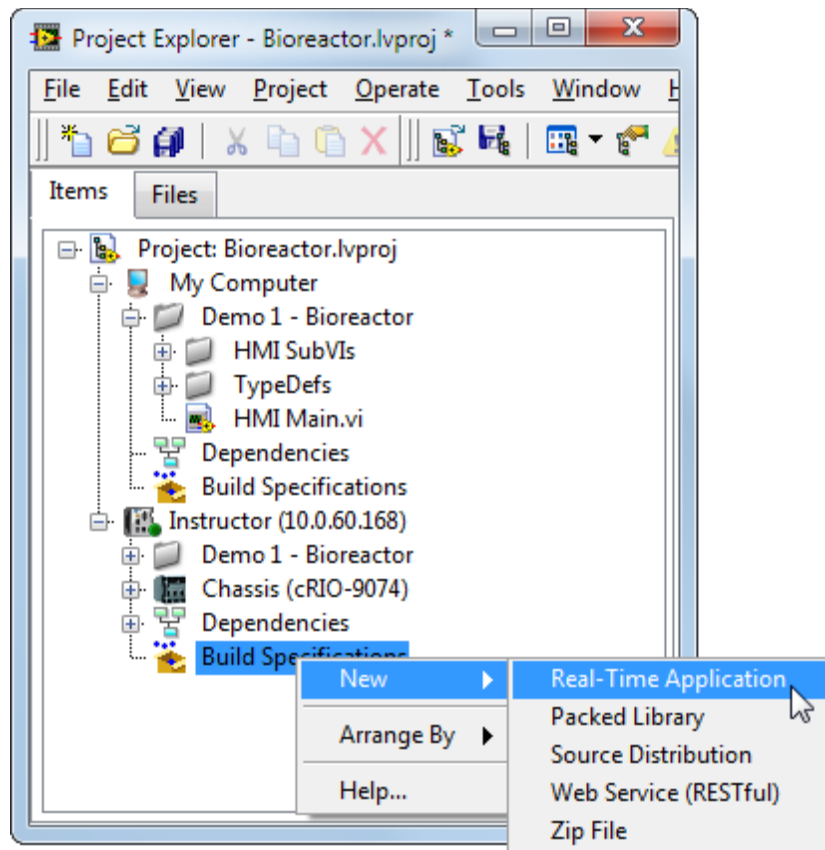
**Figure 6.1. LabVIEW Deploying an Application to the Nonvolatile Memory of the Controller**

### **Deploy a LabVIEW VI to Nonvolatile Memory**

Once you have finished developing and debugging your application, you likely want to deploy your code to the nonvolatile memory on the controller so that it persists through power cycles and configure the system so the application runs on startup. To deploy an application to the nonvolatile memory, you first need to build the VI into an executable.

### **Building an executable from a VI**

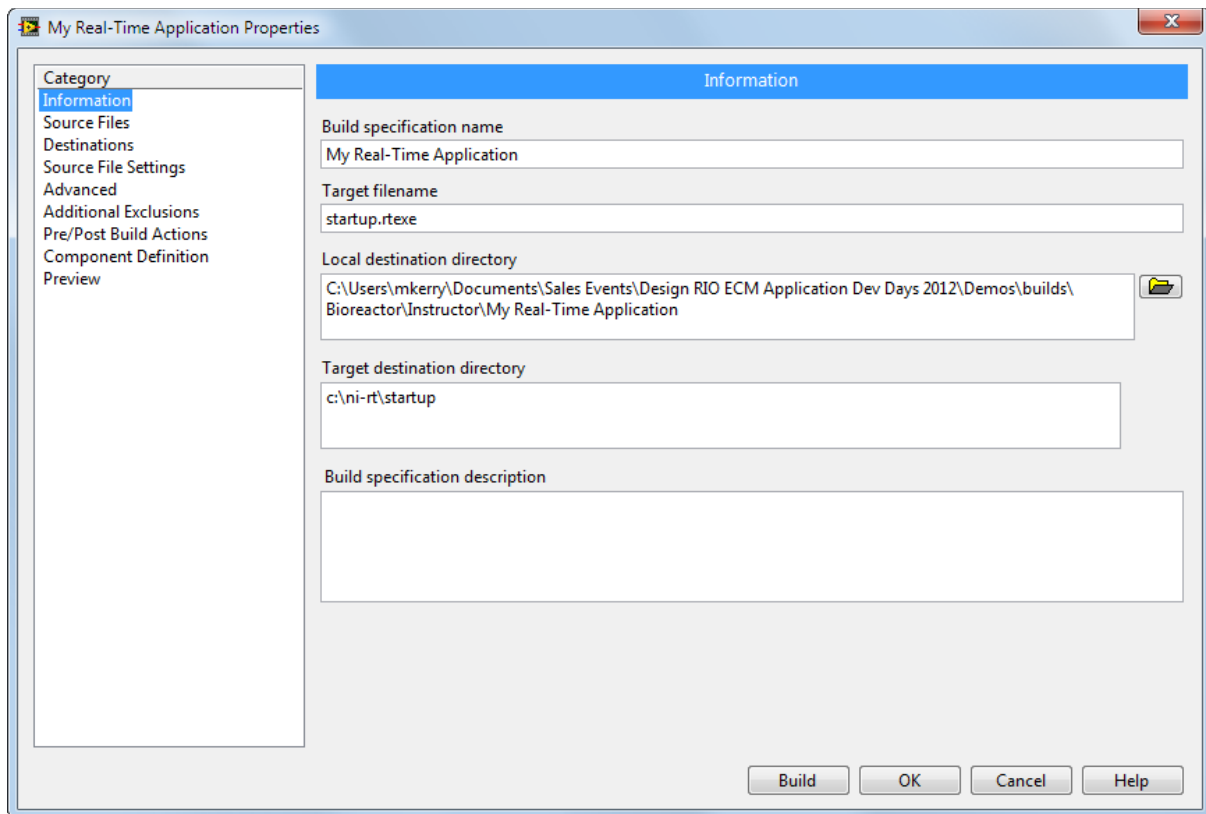
With the LabVIEW project, you can build an executable real-time application from a VI by creating a build specification under the real-time target in the LabVIEW Project Explorer. When you right-click on Build Specifications, you are presented with the option of creating a Real-Time Application along with a Source Distribution, Zip File, and so on.



**Figure 6.2. Create a new real-time application build specification.**

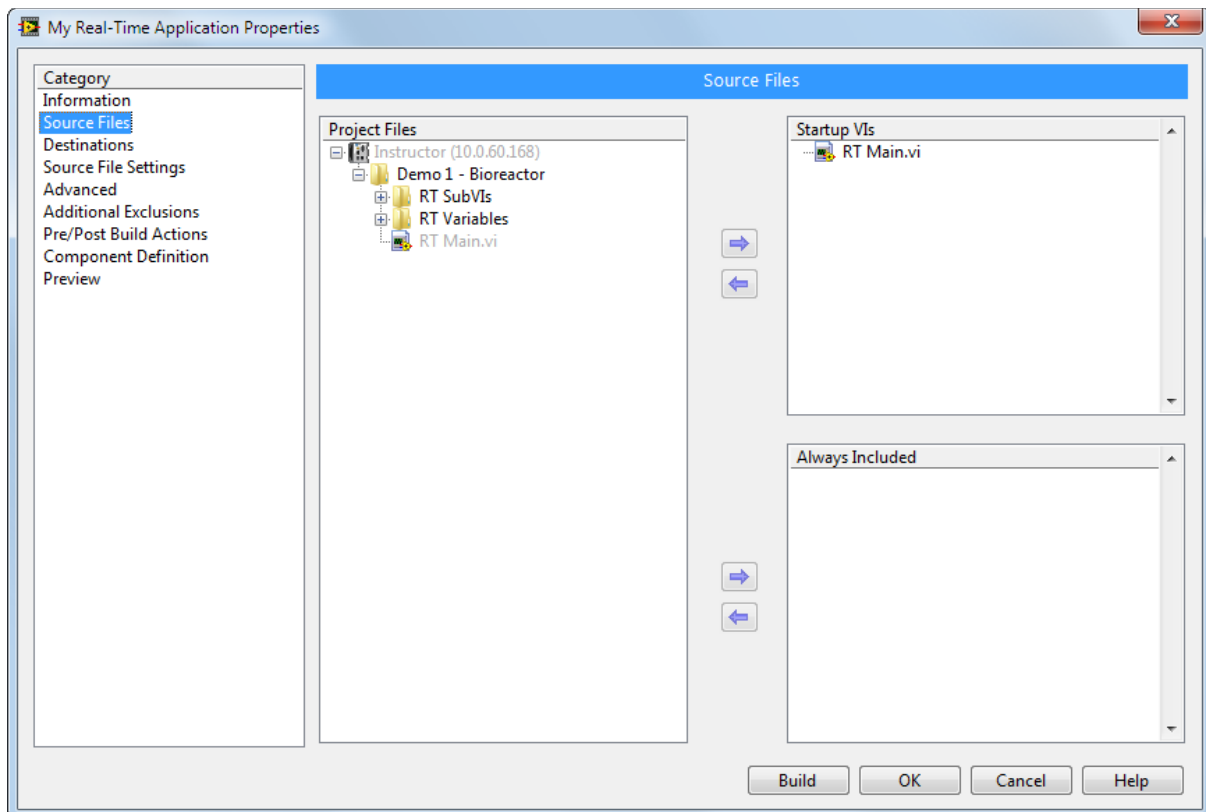
After selecting Real-Time Application, you see a dialog box featuring two main categories that are most commonly used when building a real-time application: Information and Source Files. The Destinations, Source File Settings, Advanced, and Additional Exclusions categories are rarely used when building real-time applications.

The Information category contains the build specification name, executable filename, and destination directory for both the real-time target and host PC. You can change the build specification name and local destination directory to match your nomenclature and file organization. You normally do not need to change the target filename or target destination directory.



**Figure 6.3. Information Category in the Real-Time Application Properties**

The Source Files category is used to set the startup VIs and include additional VIs or support files. You need to select the top-level VI from your Project Files and set it as a Startup VI. For most applications, a single VI is chosen to be a Startup VI. You do not need to include Ivlb or set subVIs as Startup VIs or Always Included unless they are called dynamically in your application.



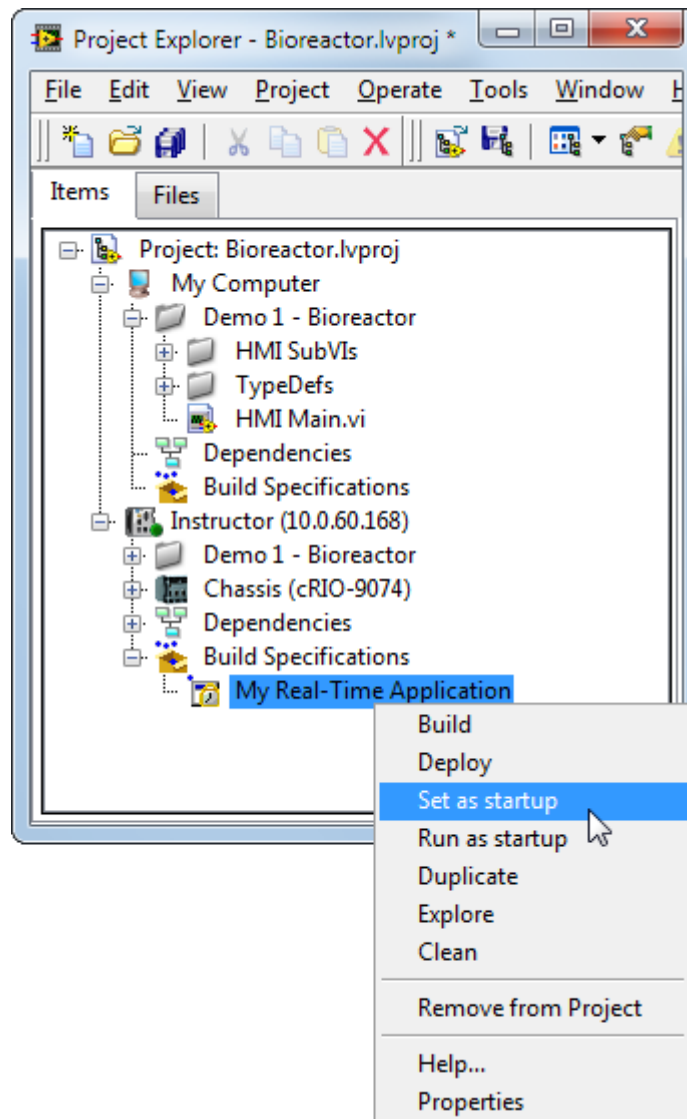
**Figure 6.4. Source Files Category in the Real-Time Application Properties (In this example, the cRIOEmbeddedDataLogger (Host).vi was selected to be a Startup VI.)**

After all of the options have been entered on the required category tabs, you can click OK to save the build specification or you can directly build the application by clicking the Build button. You can also right-click on a saved build specification and select Build to build the application.

When you build the application, an executable is created and saved on the hard drive of your development machine in the local destination directory.

### **Setting an executable real-time application to run on startup**

After an application has been built, you can set the executable to automatically start up as soon as the controller boots. To set an executable application to start up, you should right-click the Real-Time Application option (under Build Specifications) and select Set as startup. When you deploy the executable to the real-time controller, the controller is also configured to run the application automatically when you power on or reboot the real-time target. You can select Unset as Startup to disable automatic startup.

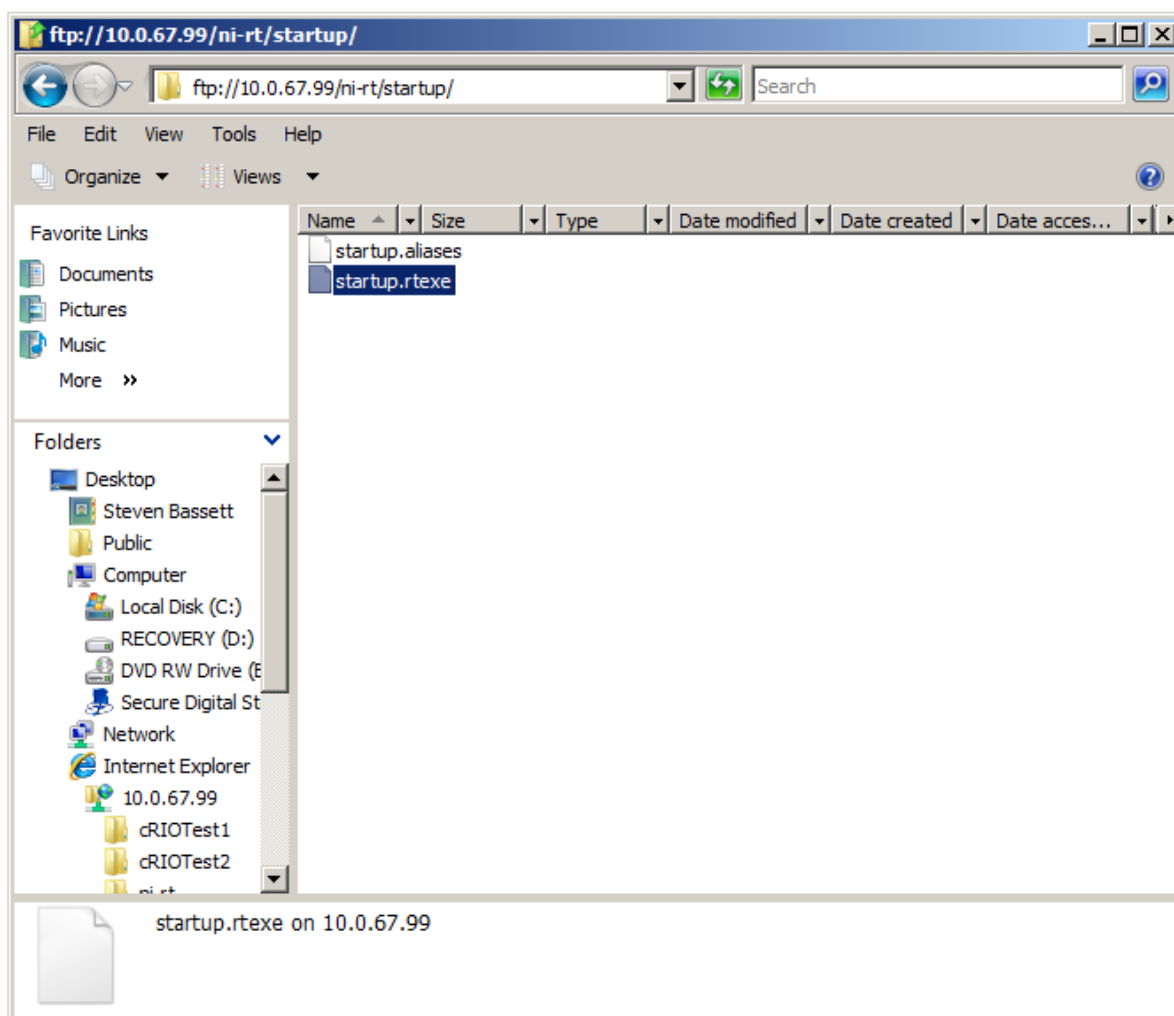


**Figure 6.5. Configuring a Build Specification to Run When an Application Boots**

## **Deploy executable real-time applications to the nonvolatile memory on a CompactRIO system**

After configuring and building your executable, you now need to copy the executable and supporting files to the nonvolatile memory on the CompactRIO controller and configure the controller so the executable runs on startup. To copy the files and configure the controller, right-click on the Real-Time Application option and select **Deploy**. Behind the scenes, LabVIEW copies the executable files onto the controller's nonvolatile memory and modifies the `ni-rt.ini` file to set the executable to run on startup. If you rebuild an application or change application properties (such as configuring it not to run on startup), you must redeploy the real-time application for the changes to take effect on the real-time target.

At some point, you may want to remove an executable you stored on your real-time target. The easiest way to do this is to use FTP to access the real-time target and delete the executable file that was deployed to the target. If you used the default settings, the file is located in the NI-RT\Startup folder with the name supplied in the target filename box from the Information category and the extension .rtexe.

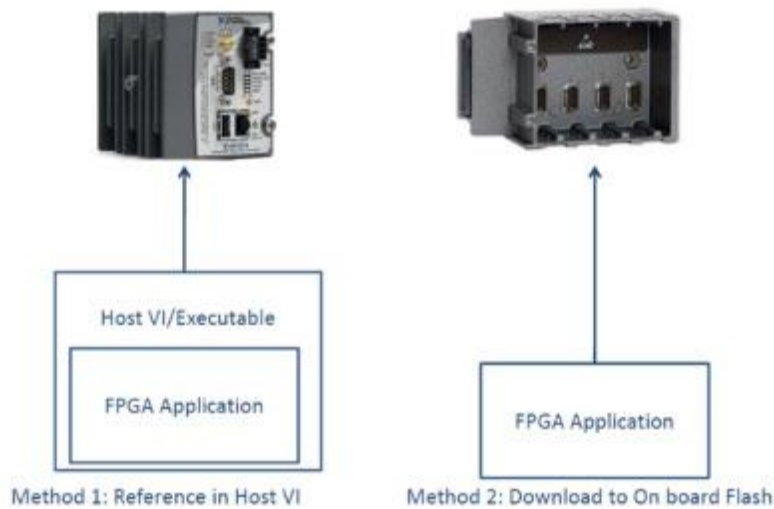


**Figure 6.6. Deleting the startup.rtexe From a CompactRIO Controller**

## Deploying LabVIEW FPGA Applications

Once the development phase of the FPGA application is complete, you need to deploy the generated bitfile, also referred to as a personality, to the system. The way you deploy the file depends on whether the FPGA is independent or dependent on a host VI. If the FPGA VI does require communication or data logging, then the development of a host VI is required. However, if the FPGA runs independently of any other target, then you should store the personality

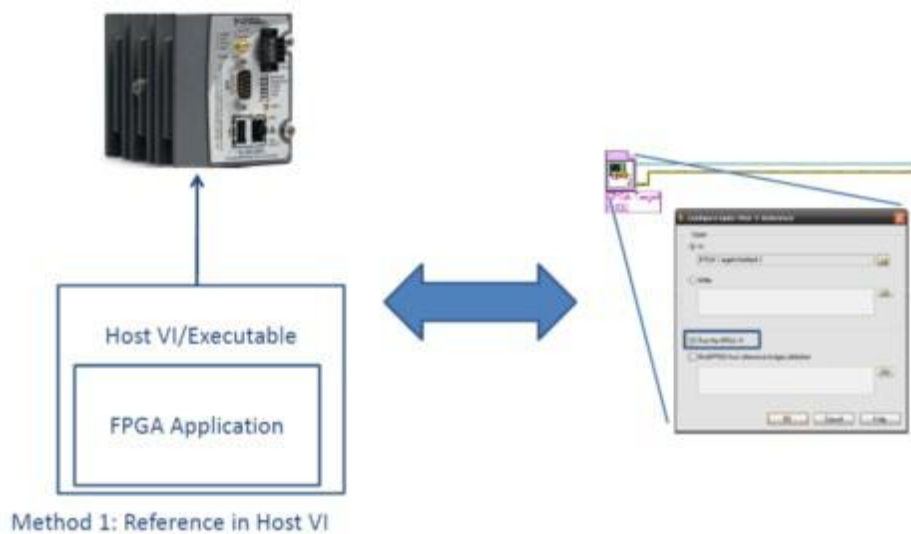
in nonvolatile flash memory on the FPGA target. The relationship between these two deployment methodologies and where the FPGA personality is then stored for a CompactRIO controller is displayed in Figure 11.7.



**Figure 6.7. Two Stand-Alone Deployment Options for LabVIEW FPGA**

### Method 1: Reference in Host VI

The most prevalent method for deploying an FPGA personality is to embed it in the host application as shown in Figure 6.8. This inclusion in the application occurs when the Open FPGA VI Reference function is used in the host implementation. When the host application is then compiled into an executable, the FPGA application is embedded inside this file. Therefore, when the host application is deployed and run, it downloads the bitfile and opens a reference to the FPGA when the Open FPGA VI Reference function is called.



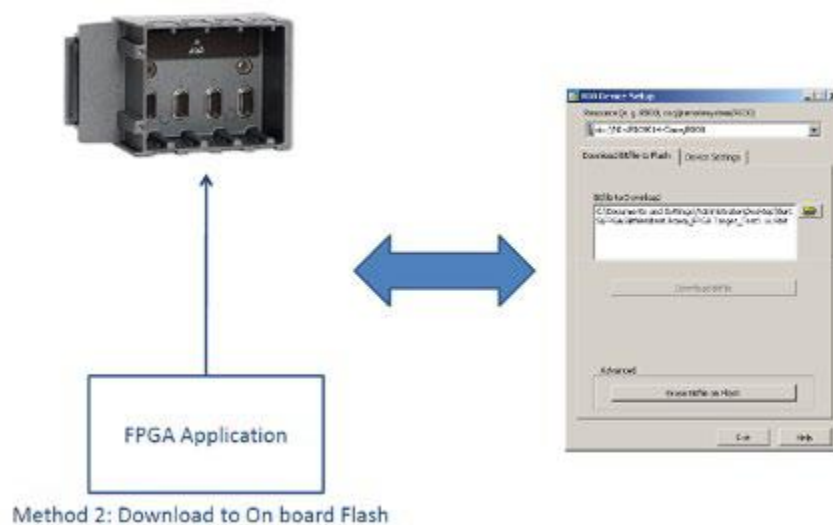
**Figure 6.8. If you use an Open FPGA VI Reference in a host VI, then you embed the FPGA's bitfile in the host executable.**

A benefit of this method is that the executable includes both targets' applications in one file. In addition, since the host file is an executable, you can simply transfer it to the host using any FTP program.

The host application is required to initialize before the FPGA application is loaded. As a result, there is a delay from device power up to the configuration of the FPGA. In addition, on power up, the state of the input and output lines of your target is unknown since the FPGA has not yet been configured. Therefore, if the FPGA is completely independent of the host application, its personality should be stored in the onboard FPGA flash memory.

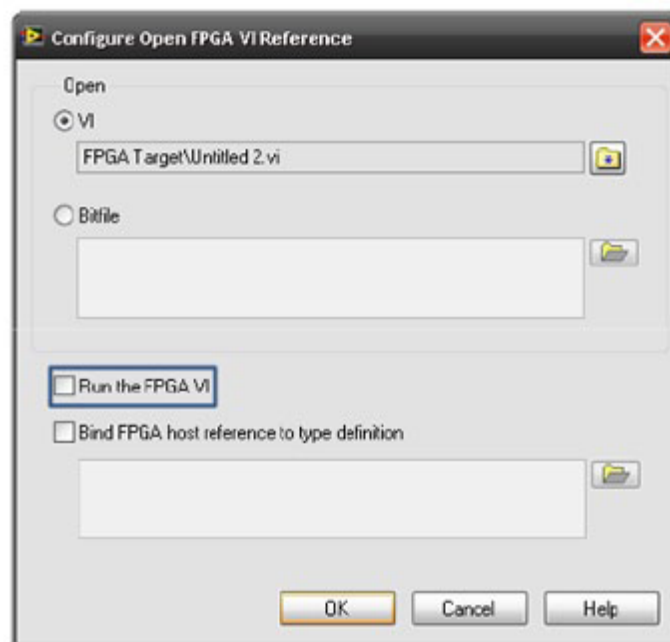
### **Method 2: Storing the Application in Nonvolatile Flash Memory on the FPGA Target**

You also can download the bitfile to flash memory on the FPGA target device using the RIO Device Setup that is included with the NI-RIO driver as shown in Figure 6.9. Subsequently, whenever the target is rebooted, it immediately loads the personality onto the FPGA from the flash memory independent of what the host application is executing. To learn more about this download process, reference KnowledgeBase 47D9Q22M. In addition, downloading the personality to the flash memory ensures that when the device is in power up, it drives all of the input and output lines to a known state since it is loaded immediately on the FPGA.



**Figure 6.9. You can download the FPGA bitfile to the FPGA's onboard flash memory using the RIO Device Setup.**

If communication does occur with the host, you must modify the host application so that the Open FPGA Reference function does not overwrite the personality that is automatically loaded on the FPGA. To disable this download, uncheck the Run the FPGA VI option in the Open FPGA VI Reference function configuration, as shown in Figure 6.10. In addition, since the FPGA personality is loaded immediately on boot up from the flash memory, the connecting host does not immediately have control over the current state of the FPGA. If you need host control of the FPGA, you should embed the personality in the host application rather than store it in flash memory.



**Figure 6.10.** You need to disable the Run the FPGA VI option in the Open FPGA VI Reference configuration window if the FPGA bitfile is loaded from the onboard flash memory so that the host application does not overwrite the bitfile on the FPGA.

The recommended personality deployment architecture depends on the needs of the specific application. However, in the majority of implementations, you should embed the personality in the host application since the FPGA often is dependent on a host application. For more information on managing FPGA deployments, see the NI Developer Zone document Managing FPGA Deployments.

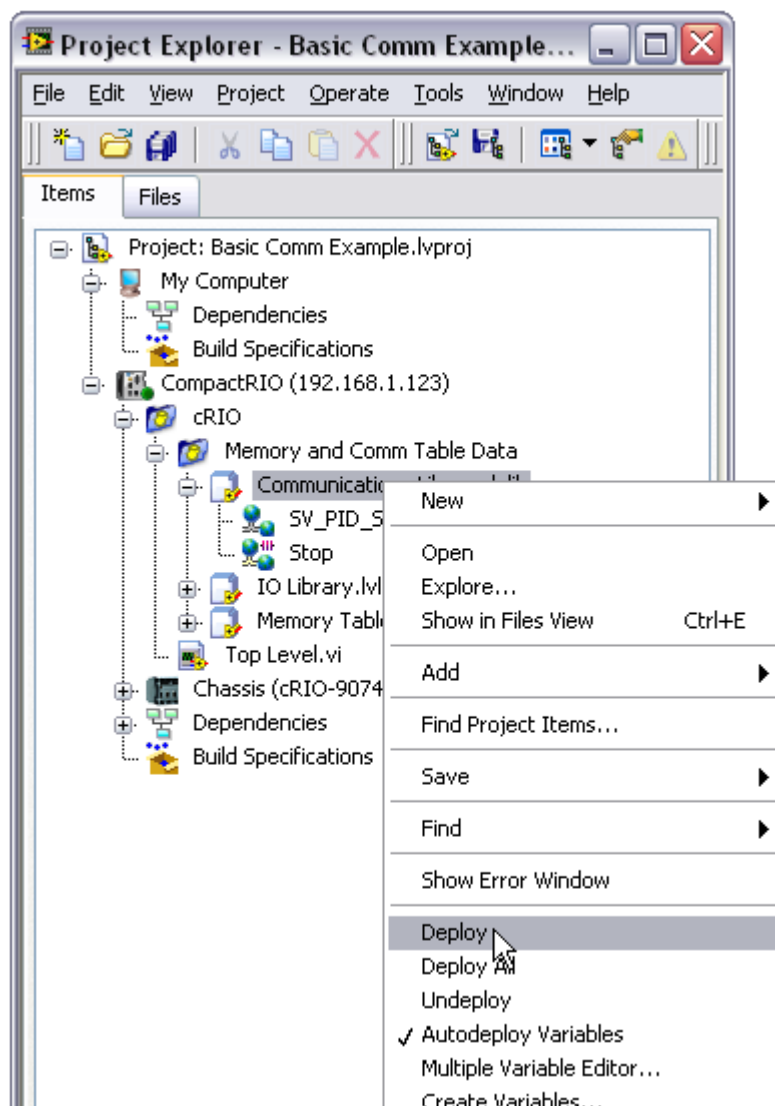
### **Deploying Applications That Use Network-Published Shared Variables**

The term *network shared variable* refers to a software item on the network that can communicate between programs, applications, remote computers,

and hardware. Find more information on network shared variables in Chapter 4: Best Practices for Network Communication.

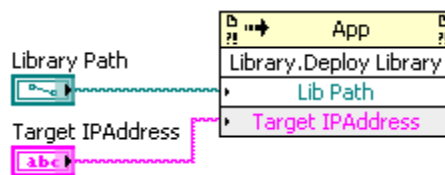
You can choose from two methods to explicitly deploy a shared variable library to a target device.

1. You can target the CompactRIO system in the LabVIEW project, place the library below the device, and deploy the library. This writes information to the nonvolatile memory on the CompactRIO controller and causes the Shared Variable Engine to create new data items on the network.



**Figure 6.11. Deploy libraries to real-time targets by selecting Deploy from the right-click menu.**

2. You can programmatically deploy the library from a LabVIEW application running on Windows using the Application Invoke Node.
  - On the block diagram, right-click to bring up the programming palette, go to **Programming»Application Control**, and place the Invoke Node on the block diagram.
  - Using the hand tool, click on Method and select **Library»Deploy Library**.



**Figure 6.12. You can programmatically deploy libraries to real-time targets using the Application Invoke Node on a PC.**

- Use the Path input of the Deploy Library Invoke Node to point to the library(s) containing your shared variables. Also specify the IP address of the real-time target using the Target IP Address input.

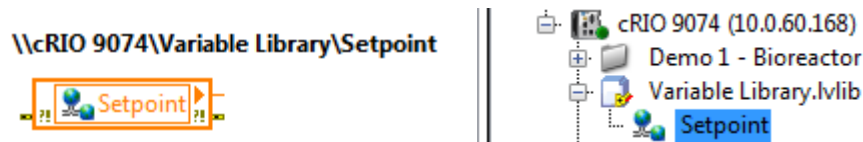
### Undeploy a Network Shared Variable Library

Once you deploy a library to a Shared Variable Engine, those settings persist until you manually undeploy them. To undeploy a library

1. Launch the NI Distributed System Manager (from **LabVIEW»Tools** or from the Start Menu).
2. Add the real-time system to My Systems (**Actions»Add System to My Systems**).
3. Right-click on the library you wish to undeploy and select Remove Process.

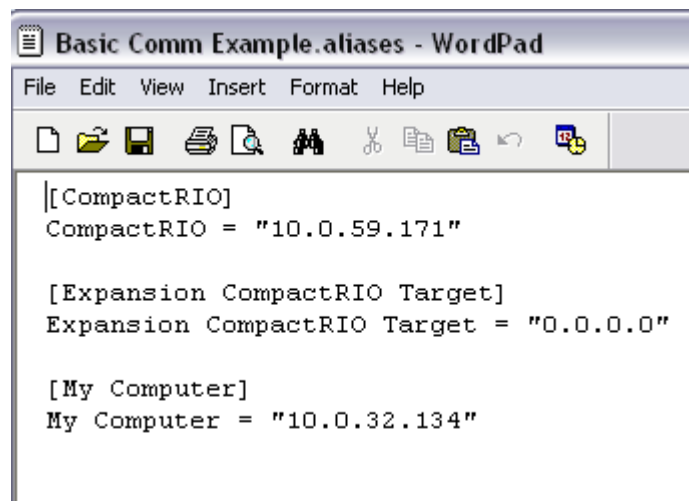
### Deploy Applications That Are Shared Variable Clients

Running an executable that is only a shared variable client (not a host) does not require any special deployment steps to deploy libraries. However, the controller does need a way to translate the name of the system that is hosting the variable into the IP address of the system that is hosting the variable.



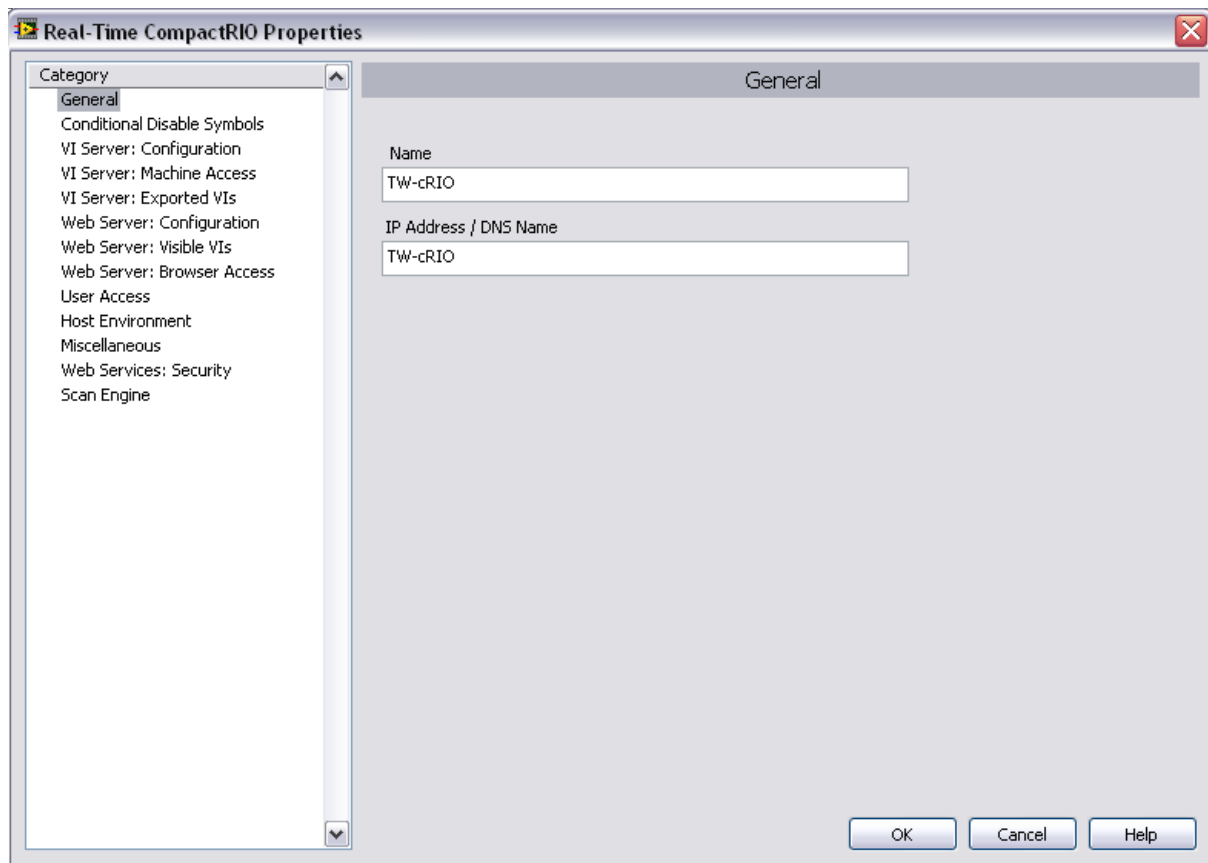
**Figure 6.13. Network Variable Node and Its Network Path**

To provide scalability, this information is not hard-coded into the executable. Instead, this information is stored in a file on the target called an alias file. An alias file is a human-readable file that lists the logical name of a target (CompactRIO) and the IP address for the target (10.0.62.67). When the executable runs, it reads the alias file and replaces the logical name with the IP address. If you later change the IP addresses of deployed systems, you need to edit only the alias file to relink the two devices. For real-time and Windows XP Embedded targets, the build specification for each system deployment automatically downloads the alias file. For Windows CE targets, you need to configure the build specification to download the alias file.



**Figure 6.14. The alias file is a human-readable file that lists the target name and IP address.**

If you are deploying systems with dynamic IP addresses using DHCP, you can use the DNS name instead of the IP address. In the LabVIEW project, you can type the DNS name instead of the IP address in the properties page of the target.

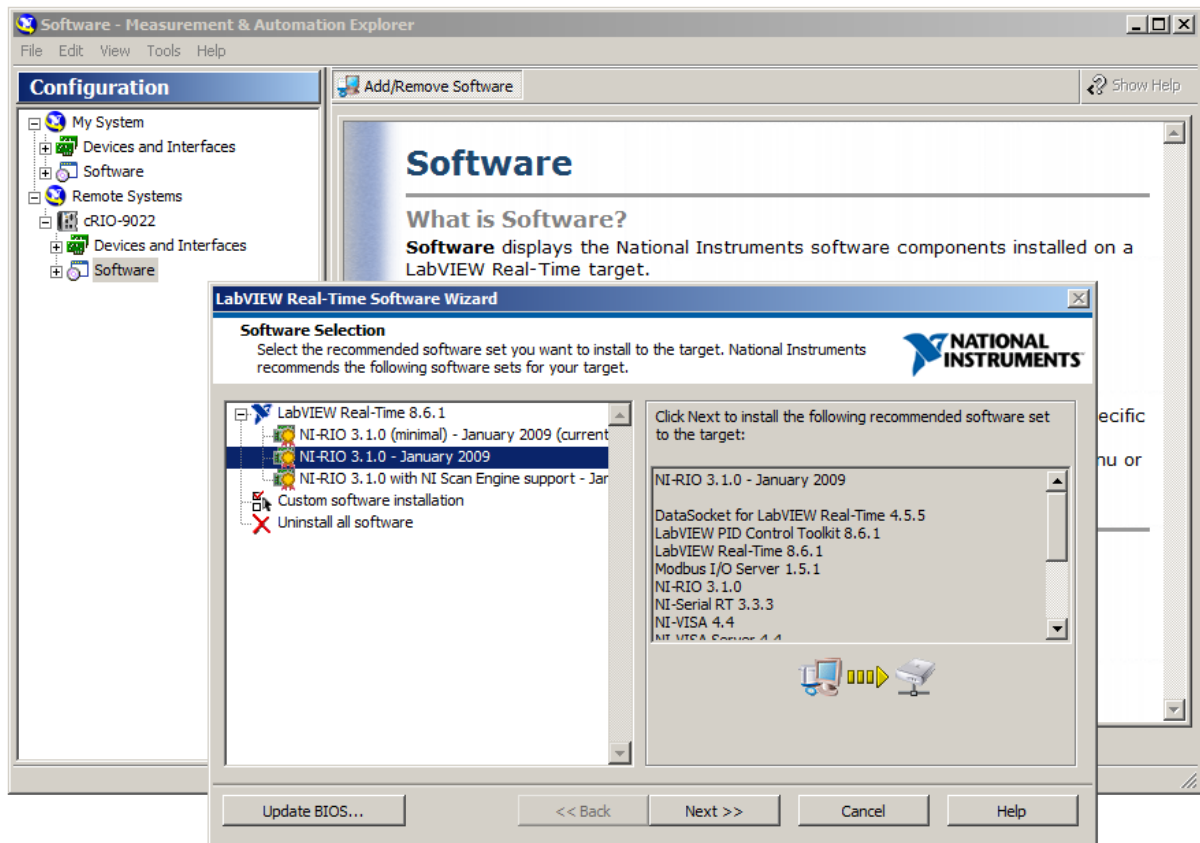


**Figure 6.15.** For systems using DHCP, you can enter the DNS name instead of the IP address.

One good approach if you need scalability is to develop using a generic target machine (you can develop for remote machines that do not exist) with a name indicating its purpose in the application. Then as part of the installer, you can run an executable that either prompts the user for the IP addresses for the remote machine and My Computer or pulls them from another source such as a database. Then the executable can modify the aliases file to reflect these changes.

### **Recommended Software Stacks for CompactRIO**

NI also provides several sets of commonly used drivers called recommended software sets. You can install recommended software sets on CompactRIO controllers from MAX.



**Figure 6.16. Recommended Software Sets Being Installed on a CompactRIO Controller**

Recommended software sets guarantee that an application has the same set of underlying drivers for every real-time system that has the same software set. In general, there is a minimal and full software set.

## System Replication

After you deploy a LabVIEW Real-Time application to a CompactRIO controller, you may want to deploy that image to other identical real-time targets. Being able to replicate the image of a real-time target makes deploying targets and systems easier and more efficient. Whether the user is making periodic backups of a system, deploying from a developed system to many new ones, updating an image on a target, or giving someone else the tools to duplicate a working system, replicating an image makes all of these applications possible. The Real-Time Application Deployment utility makes the system replication simple and intuitive.

NI offers a variety of tools for the replication of LabVIEW Real-Time targets. You can use the tools to replicate one real-time target into multiple copies, circumventing the use of MAX and an FTP client in favor of a simple utility or

the ability to customize your own using LabVIEW. The imaging process includes the following steps:

1. Deploy the built application to a real-time target from the LabVIEW project.
2. Create a disk image from this controller (the image packages every file on the real-time target hard drive in a zip file and saves it to the host machine).
3. Deploy that image to one or more targets.



**Figure 6.17. With NI imaging tools, you can deploy images to multiple real-time targets.**

With this imaging process, you can obtain an exact copy of a real-time system and easily deploy it to multiple targets. This reduces the risk of deployment error and the need for using the LabVIEW development environment for deployment. When using this process, you must deploy images to the same controller model used to create the image. For example, an image that was created for a cRIO-9022 controller cannot be used on a cRIO-9024 controller.

You can choose from two methods for imaging CompactRIO systems. You can use a prebuilt imaging utility or you can design your own custom utility using built-in functions in LabVIEW Real-Time. The next section provides an overview of the Real-Time Application Deployment Utility in addition to several APIs for developing your own custom utility.

### **The Real-Time Application Deployment Utility (RTAD)**

RTAD is a turnkey application for deploying and replicating LabVIEW Real-Time applications from one LabVIEW Real Time target to another. This type of utility is commonly used in the production and software/firmware installation of systems such as machine control systems, medical devices, automated testers,

and so on. OEMs may use this type of utility as part of their factory installation processes when assembling their products.

You can download this utility from the NI Developer Zone document, Automated LabVIEW Real-Time Deployment (RTAD) Reference Application. After installation, you can start the RTAD directly from the Tools menu in the LabVIEW environment.

When replicating applications from one target to another, the application image is retrieved from one real-time target and copied to another. The application image is the contents (all of the files and directories) of the hard drive of a real time target that define the behavior of the real-time target as well as any bitfiles set to deploy to the FPGA flash memory. This utility only transfers images between identical controllers and backplanes. **If you receive an image from a specific controller, you can deploy that image only to controllers with the same model number.**

The RTAD utility shows these two entities, Real-Time Targets and Application Images, in two tables on the main UI of the utility.

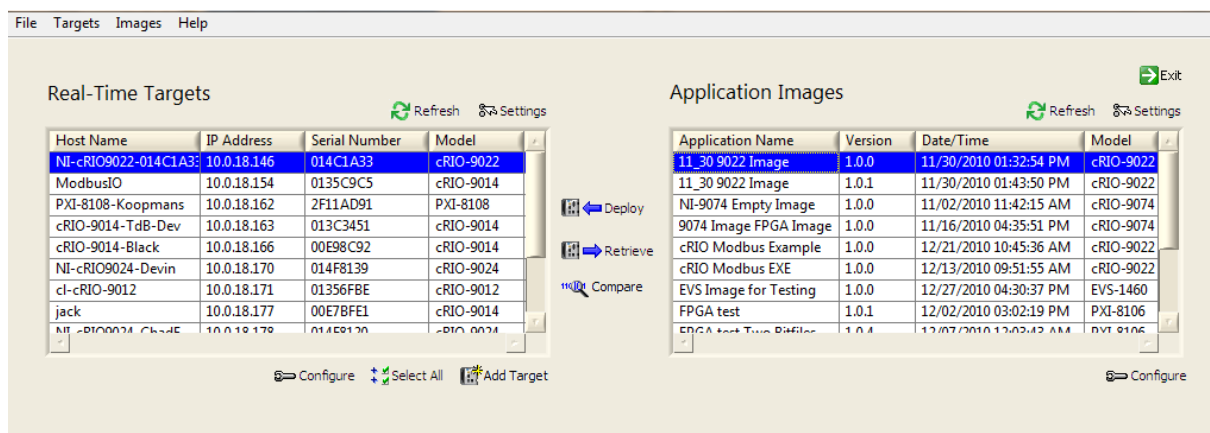


Figure 6.18. With NI imaging tools, you can deploy images to multiple real-time targets.

The Real-Time Targets table shows all of the targets on the local subnet as well as any network targets manually added to the list. You can use these targets for both image retrieval and image deployment. The Application Images table shows all of the images that are stored on the local hard drive and can be deployed to a target system.

## Retrieving Application Images

To copy the application image from a real-time target to the local hard drive, select the appropriate target in the table and click the Retrieve button in the center of the UI. You can retrieve an application image from a target in three ways. The application could either be a brand new application or a new version of a previous image either currently on the controller or previously saved to disk. If the current application is a new version of an older application, it is best to inherit the old application properties. If you are creating an image of the application for the first time, select New Application Image.

After making this selection, you are presented with the dialog box in Figure 6.19 to specify the local file for the application image and specify some additional information that is stored with the application image. If this is not a new image, properties from the old version of the image are automatically populated.

**Application Image Properties**

Name  
cRIO Modbus Test

Old Version  
New Version  
1.0.0

Description  
This is a cRIO Modbus application.

Configure Bitfile(s) for FPGA Flash Deployment

Application Image Destination  
C:\AppImages\cRIO Modbus EXE 1\_0\_0.lvappimg  
Browse

Retrieve image from ()

Cancel

**Figure 6.19. Configuring Your Application Image Properties**

In addition to retrieving and deploying an image on the real-time hard drive, the utility can deploy bitfiles to FPGA flash memory. Saving a bitfile in flash memory has some advantages to deploying the bitfile from the RT EXE. For example, the host application is required to initialize before the FPGA application is loaded and, as a result, there is a delay from device power up to the configuration of the FPGA. In addition, on power up, the state of the input and output lines of your target is unknown since the FPGA has not yet been configured. Therefore, if the FPGA is completely independent of the host application, its personality should be stored in the onboard FPGA flash memory.

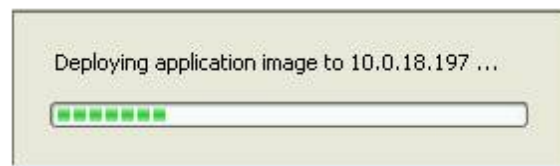
Using the RTAD utility, you can now save bitfiles with an image during retrieval and then later deploy them to flash memory when you deploy the image. Click

Configure Bitfile(s) for FPGA Flash Deployment to edit your FPGA flash deployment settings.

### Deploying Application Images

To deploy an application image to one or more targets, select the image in the right table and select the desired real time targets in the left table. You can select multiple real-time targets using a <Ctrl-Click> or <Shift-Click> or by clicking the Select All button. After completing your selection, click on Deploy at the center of the UI. A dialog confirms your real-time target selection and allows for additional target configuration options.

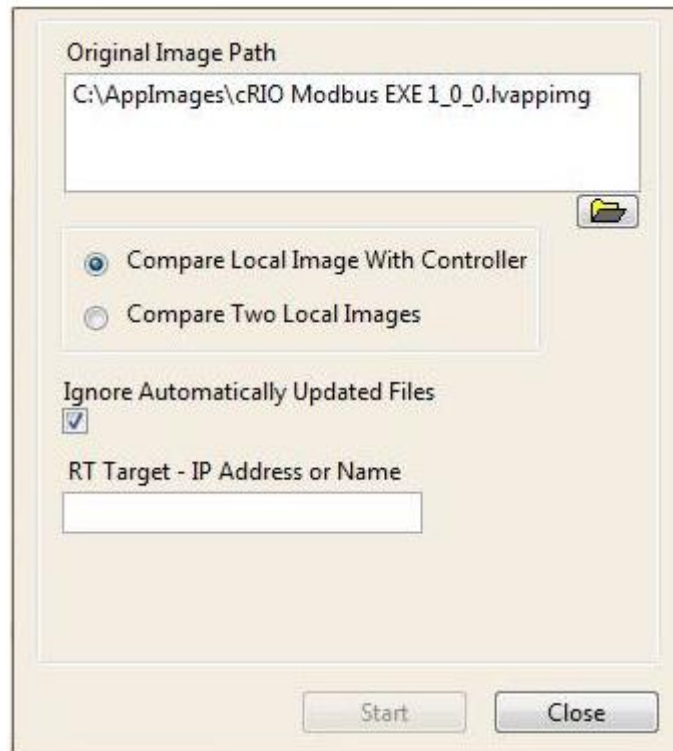
After verifying the selected targets and setting any desired network configuration options for your real-time targets, click on Deploy Application Image to Listed Targets to begin the deployment process. The application image is 195 deployed to the selected targets in sequential order, one at a time. During this process, the Figure 6.20 progress dialog is shown. This process takes several minutes per real-time target.



**Figure 6.20. Application Image Deployment Process**

### Comparing Application Images

In addition to retrieving and deploying application images, the utility offers an image comparison feature. The comparison tool compares each file in two images to notify you if any files have been modified, added, or removed. You can choose to compare a local image with an image on a controller, or two local images.



**Figure 6.21. Compare a Local Image to Network Controller Dialog**

Click the Start button to begin the comparison process. Once the comparison begins, each file on both images is compared for changes. Files that are found in one image and not the other are also identified. This process may take several minutes to complete, and a progress dialog is displayed until completion. If the tool completes with no differences found, the Figure 6.22 dialog appears.



**Figure 6.22. Identical Images Notification**

However, if any differences are identified, they are listed in the Figure 6.23 table. You can then log the results to a TSV file.

Image differences are listed below...

List of Modified Files	Files Found Only In Original Image (local)	Files Found Only In Comparison Image (local or remote)
c:\ni-rt\config\criocfg.bin	c:\lvppimage.info	c:\ni-rt\config\nimcdmdata.xml
c:\ni-rt\config\masterRegistry.xml	c:\ni-rt\system\mxsCheckpoints\20101119_225018.cpt\config3.mxs	c:\ni-rt\system\dmECAT3rdPartyComm.out
c:\ni-rt\config\scanConfig.xml		c:\ni-rt\system\dmECATComm.out
c:\ni-rt\config\variables.xml		c:\ni-rt\system\dmRIOComm.out
c:\ni-rt\system\config.cdf		c:\ni-rt\system\dmsimcom.out
c:\ni-rt\system\mxsjar.ini		c:\ni-rt\system\nimcca.out
c:\ni-rt\system\mxsjar.mx5		c:\ni-rt\system\nimcdm.out
c:\ni-rt\system\mxsSchema.log		c:\ni-rt\system\nimcdmtg.out
c:\ni-rt\system\nimcdm\ab\IEWAdapter.out		

Log Results OK

Figure 6.23. Image Comparison Results Table

## Deploying CompactRIO Application Updates Using a USB Memory Stick

If your CompactRIO systems are not available on the network, you may want to deploy an image using a USB stick. The process of updating or deploying a new application image from a USB memory device to a CompactRIO controller is based on a simple file copy operation that replaces the files on the hard drive of the CompactRIO controller with files stored on the USB memory device. This file copy operation is added as a VI to the main LabVIEW Real-Time application deployed to the controller.

A LabVIEW Real-Time application once loaded and running on the controller is stored completely in the active memory (RAM) of the controller. Because of this, you can delete and replace the application files stored on the controller hard drive while the application is running in memory. The new application becomes active by performing a reboot operation on the controller and loading the new application during the boot process.

To update the deployed code from the USB memory device in the future, you must add code to the main application that handles the deployment process. The Deploy Image.vi shown in Figure 6.24 handles the update process.

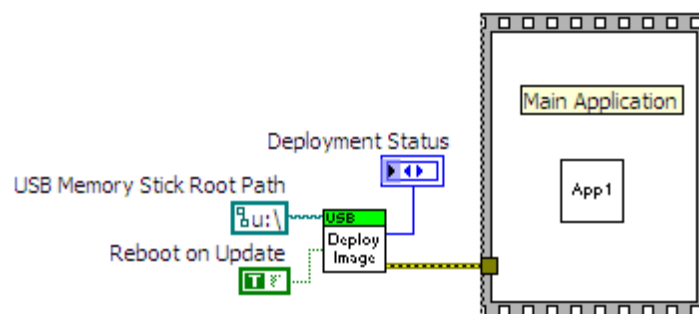


Figure 6.24. Include the USB Deploy Image VI in your deployed real-time application to enable updates from a USB memory stick.

Once you have added this code, you can build the main application into an executable and deploy it to a CompactRIO controller. The deployed application executable, together with all of the other files on the CompactRIO controller hard drive, becomes the application image. For more information on this utility including downloadable files, see the NI Developer Zone document *Reference Design for Deploying CompactRIO Application Updates Using a USB Memory Device*.

### APIs for Developing Custom Imaging Utilities

You can choose from several imaging APIs—all with nearly identical functionality—depending on the version of LabVIEW you are using. The RTAD utility is based on the RT Utilities VIs.

System Configuration API—Recommended for LabVIEW 2011 or later

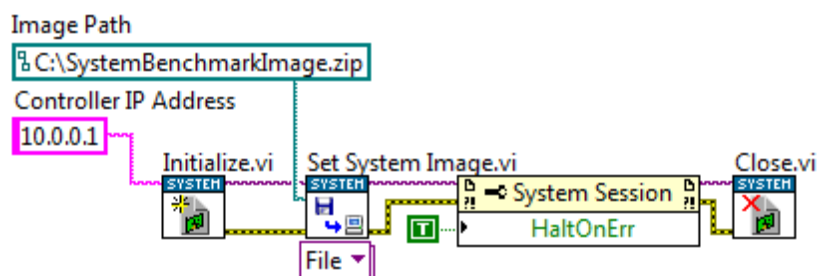
RT Utilities API—Recommended for LabVIEW 2009 and 2010

NI System Replication Tools—Recommended for LabVIEW 8.6 or earlier

Another useful API discussed in this section is the cRIO Information Library (CRI), which you can use with the three APIs listed above to return information about the current hardware configuration of a CompactRIO system.

### System Configuration API

The System Configuration API exposes MAX features programmatically. For example, you can use this API to programmatically install software sets, run self-tests, and apply network settings in addition to image retrieval and deployment. This API is located in the Functions palette under **Measurement I/O»System Configuration»Real Time Software**.



**Figure 6.25. Programmatically Setting a System Image Using the System Configuration API  
RT Utilities API**

The RT Utilities API also includes VIs for image retrieval and deployment.

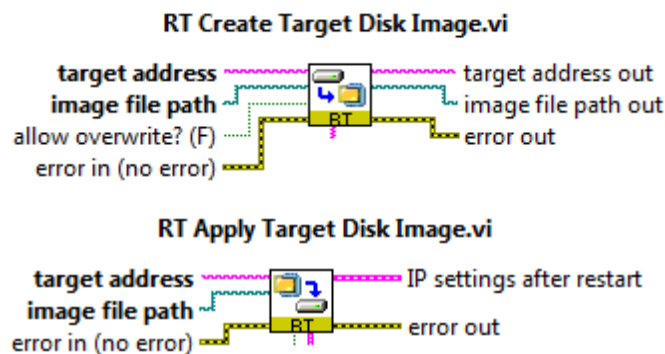


Figure 6.26. You also can use the RT Utilities VIs in LabVIEW 2009 or later to programmatically deploy and retrieve a system image

## LabVIEW Real-Time system replication tools

For LabVIEW 8.6 and earlier, neither the RT Utilities API nor the System Configuration API is available. For these applications, NI recommends using LabVIEW Real-Time system replication tools. For more information on these tools, including the downloadable installation files, see the NI Developer Zone document Real-Time Target System Replication.

## cRIO Information Library (CRI)

You can use the CRI with the three APIs discussed above to detect the current configuration of a CompactRIO system. The cRIO Information component provides VIs to retrieve information about a local or remote CompactRIO controller, backplane, and modules including the type and serial number of each of these system components.

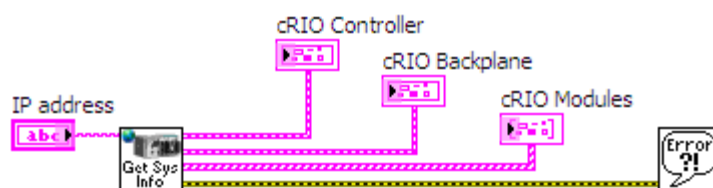


Figure 6.27. CRI Get Remote cRIO System Info.vi

You can download this library from the NI Developer Zone document Reference Library for Reading CompactRIO System Configuration Information.

## IP Protection

Intellectual property (IP) in this context refers to any unique software or application algorithm(s) that you or your company has independently

developed. This can be a specific control algorithm or a full-scale deployed application. IP normally takes a lot of time to develop and gives companies a way to differentiate from the competition. Therefore, protecting this software IP is important. LabVIEW development tools and CompactRIO provide you the ability to protect and lock your IP. In general, you can implement two levels of IP protection:

### **Lock algorithms or code to prevent IP from being copied or modified**

If you have created algorithms for a specific functionality, such as performing advanced control functions, implementing custom filtering, and so on, you may want to distribute the algorithm as a subVI but prevent someone from viewing or modifying that actual algorithm. This may be to achieve IP protection or to reduce a support burden by preventing other parties from modifying and breaking your algorithms.

### **Lock code to specific hardware to prevent IP from being replicated**

Use this method if you want to ensure that a competitor cannot replicate your system by running your code on another CompactRIO system or want your customers to come back to you for service and support.

### **Locking Algorithms or Code to Prevent Copying or Modification**

Protect Deployed Code LabVIEW is designed to protect all deployed code, and all code running as a startup application on a CompactRIO controller is by default locked and cannot be opened. Unlike other off-the-shelf controllers or some PLCs for which the raw source code is stored on the controller and protected only by a password, CompactRIO systems do not require the raw source code to be stored on the controller.

Code running on the real-time processor is compiled into an executable and cannot be “decompiled” back to LabVIEW code. Likewise, code running on the FPGA has been compiled into a bitfile and cannot be decompiled back to LabVIEW code. To aid in future debugging and maintenance, you can store the LabVIEW project on the controller or call raw VIs from running code, but by default any code deployed to a real-time controller is protected to prevent copying or modifying the algorithms.

### **Protect Individual VIs**

Sometimes you want to provide the raw LabVIEW code to enable end customers to perform customization or maintenance but still want to protect specific algorithms. LabVIEW offers a few ways to provide usable subVIs while protecting the IP in those VIs.

### Method 1: Password protecting your LabVIEW code

Password protecting a VI adds functionality that requires users to enter a password if they want to edit or view the block diagram of a particular VI. Because of this, you can give a VI to someone else and protect your source code. Password protecting a LabVIEW VI prohibits others from editing the VI or viewing its block diagram without the password. However, if the password is lost, you cannot unlock a VI. Therefore, you should strongly consider keeping a backup of your files stored without passwords in another secure location.

To password protect a VI, go to **File»VI Properties**. Choose Protection for the category. This gives you three options: unlocked (the default state of a VI), locked (no password), and password-protected. When you click on password-protected, a window appears for you to enter your password. The password takes effect the next time you launch LabVIEW.

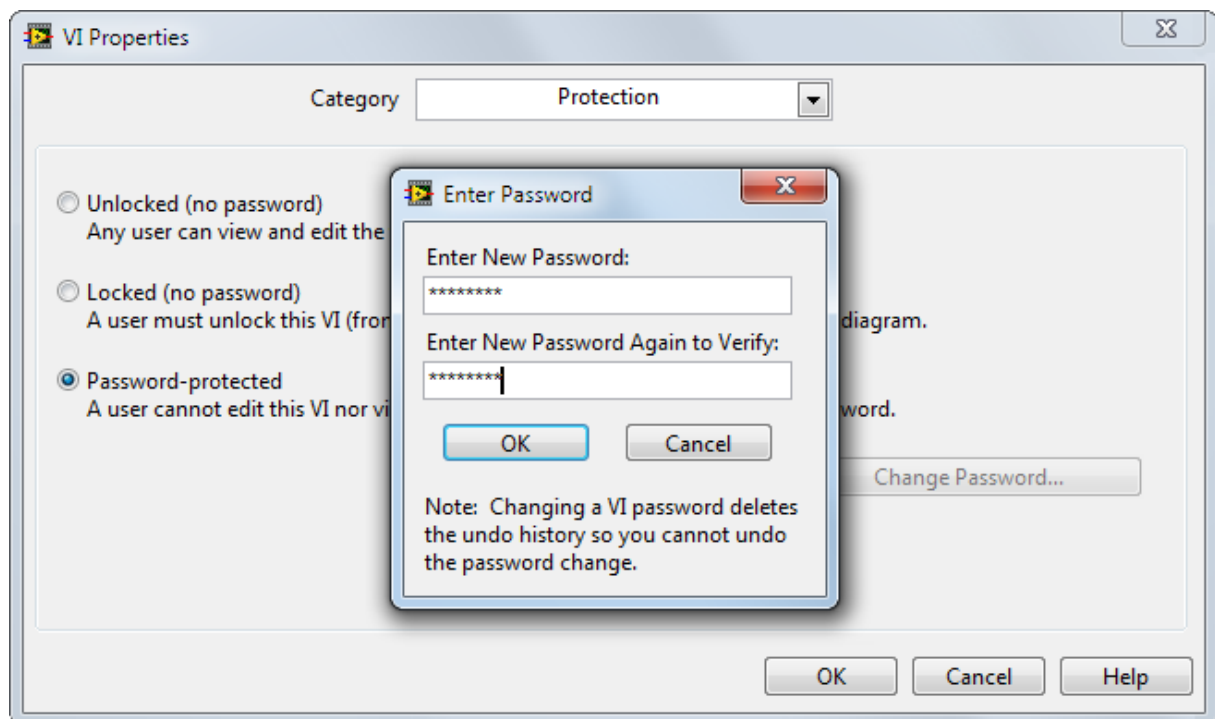


Figure 6.28. Password Protecting LabVIEW Code

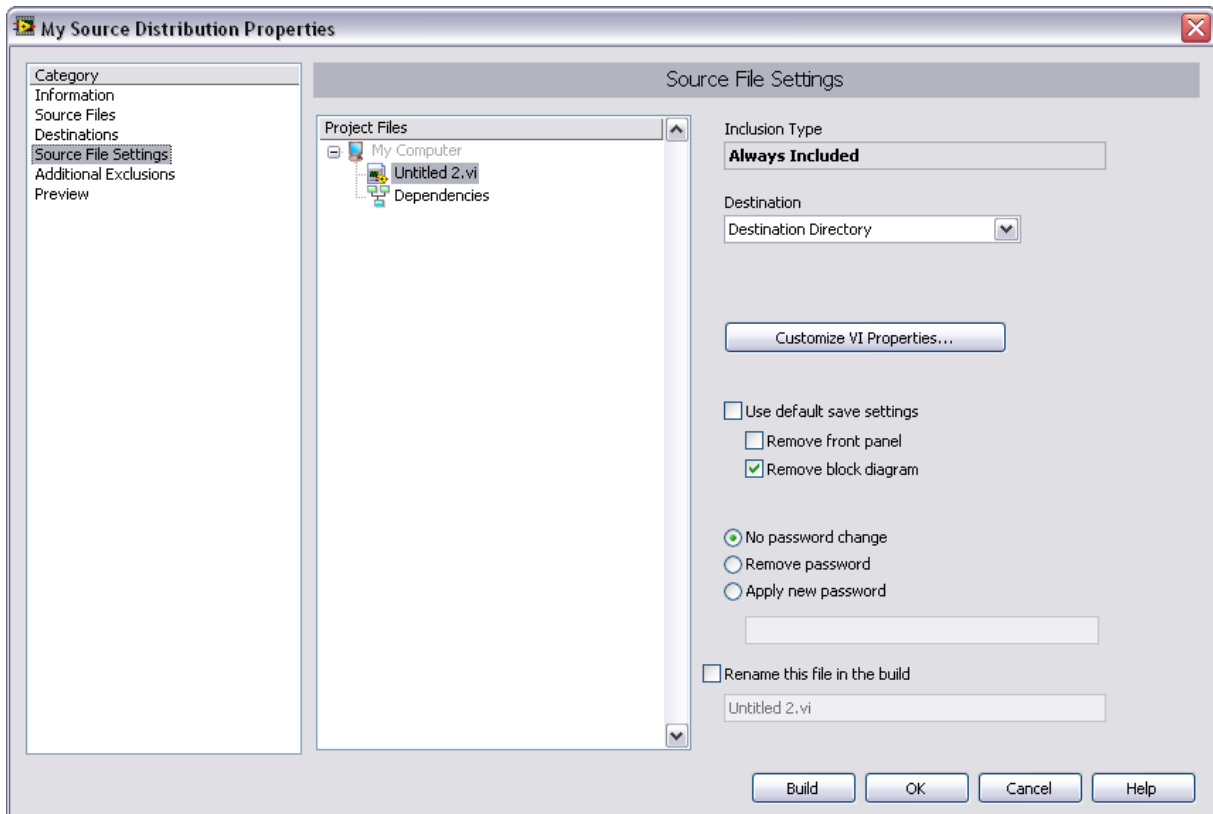
The LabVIEW password mechanism is quite difficult to defeat, but no password algorithm is 100 percent secure from attack. If you need total assurance that someone cannot gain access to your source code, you should consider removing the block diagram.

### **Method 2: Removing the block diagram**

To guarantee that a VI cannot be modified or opened, you can remove the block diagram completely. Much like an executable, the code you distributed no longer contains the original editable code. Do not forget to make a backup of your files if you use this technique because the block diagram cannot be recreated. Removing the block diagram is an option you can select when creating a source distribution. A source distribution is a collection of files that you can package and send to other developers to use in LabVIEW. You can configure settings for specified VIs to add passwords, remove block diagrams, or apply other settings.

Complete the following steps to build a source distribution.

1. In the LabVIEW project, right-click **Build Specifications** and select **New»Source Distribution** from the shortcut menu to display the Source Distribution Properties dialog box. Add your VI(s) to the distribution.
2. On the Source File Settings page of the Source Distribution Properties dialog box, remove the checkmark from the Use default save settings checkbox and place a checkmark in the Remove block diagram checkbox to ensure that LabVIEW removes the block diagram.
3. Build the source distribution to create a copy of the VI without its block diagram.



**Figure 6.29.** You can programmatically deploy libraries to real-time targets using the Application Invoke Node on a PC.

**Note:** If you save VIs without block diagrams, do not overwrite the original versions of the VIs. Save the VIs in different directories or use different names.

### Lock Code to Hardware to Prevent IP Replication

Some OEMs and machine builders also want to protect their IP by locking the deployed code to a specific system. To make system replication easy, by default the deployed code on a CompactRIO controller is not locked to hardware and can be moved and executed on another controller. For designers who want to prevent customers or competitors from replicating their systems, one effective way to protect application code with CompactRIO is by locking your code to specific pieces of hardware in your system. This ensures that customers cannot take the code off a system they have purchased from you and run the application on a different set of CompactRIO hardware. You can lock the application code to a variety of hardware components in a CompactRIO system including the following:

- The MAC address of a real-time controller
- The serial number of a real-time controller

- The serial number of the CompactRIO backplane
- The serial number of individual modules
- Third-party serial dongle

You can use the following steps as guidelines to programmatically lock any application to any of the above mentioned hardware parameters and thus prevent users from replicating application code:

1. Obtain the hardware information for the device. Refer to the following procedures for more information on programmatically obtaining this information.
2. Compare the values obtained to a predetermined set of values that the application code is designed for using the Equal? function from the Comparison palette.
3. Wire the results of the comparison to the selector input of a Case structure.
4. Place the application code in the true case and leave the false case blank.
5. Performing these steps ensures that the application is not replicated or usable on any other piece of CompactRIO hardware.

## **License Key**

Adding licensing to a LabVIEW Real-Time application can protect a deployed application from being copied and run on another similar or identical set of hardware without obtaining a license from the vendor or distributor of the application. Most modern-day applications running on desktop computers are protected by a license key that is necessary to either install the application or run it in its normal operational mode. Many vendors use license keys to determine if an application runs in demo mode or is fully functional. License keys may also be used to differentiate between versions of an application or to enable/disable specific features.

You can add this behavior to a LabVIEW Real-Time application using the reference design and example code developed by NI Systems Engineering. You can download this code from the NI Developer Zone document Reference Design for Adding Licensing to LabVIEW Real-Time Applications. The main modification is how you create a unique system ID for a specific hardware

target. In the case of CompactRIO, you use the controller, backplane, and module serial numbers. For other targets, you may use the serial number or Ethernet MAC address of a given target to create a unique system ID.

### **Choosing a License Model**

The license model defines how the license key is generated based on particular characteristics of the target hardware. One simple example of a license model is to base the license key on the serial number of the controller. In this case, the application runs if the controller has the serial number matching the license key provided.

If the controller serial number does not match the license key, the application does not run. If the hardware target includes additional components such as a CompactRIO backplane and CompactRIO modules, you may choose to lock the application not only to the controller but also to these additional system components. In this case, you can link the license key to the serial numbers of all of these hardware target components. All components with the correct serial numbers must be in place to run the application.

The unique characteristic of the hardware target (for example, a serial number) is called the system ID for the purpose of this guide.

One example of a more complex license model is to base the license key on the serial numbers of multiple system components but require only some of these components to be present when running the licensed application. This allows the application user to replace some of the system components, in case a repair is required, without needing to acquire a new application license key. The number of components linked to the license key that must be present to run the application is defined by the developer as part of the license model.

### **Application Licensing Process**

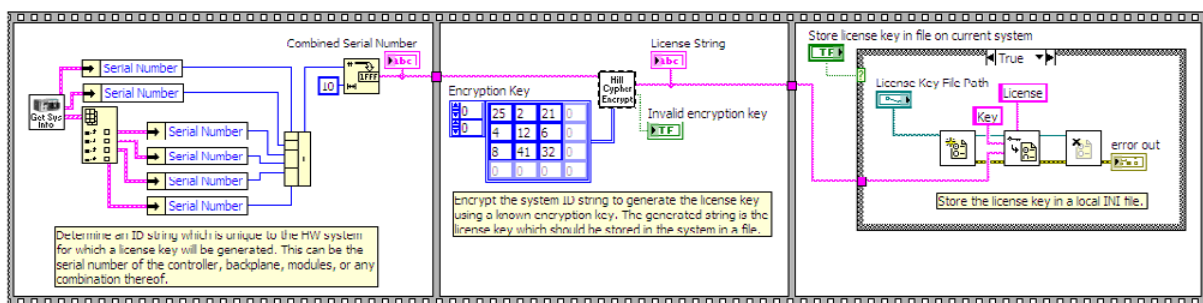
Adding licensing (creating and using a license key) to a LabVIEW Real-Time application consists of the following steps:

1. Create a unique system ID for each deployed hardware target
2. Create a license key based on the system ID
3. Store the license key on the hardware target

4. Verify the license key during application startup system for each deployed hardware target

### Create a unique system ID for each deployed hardware target

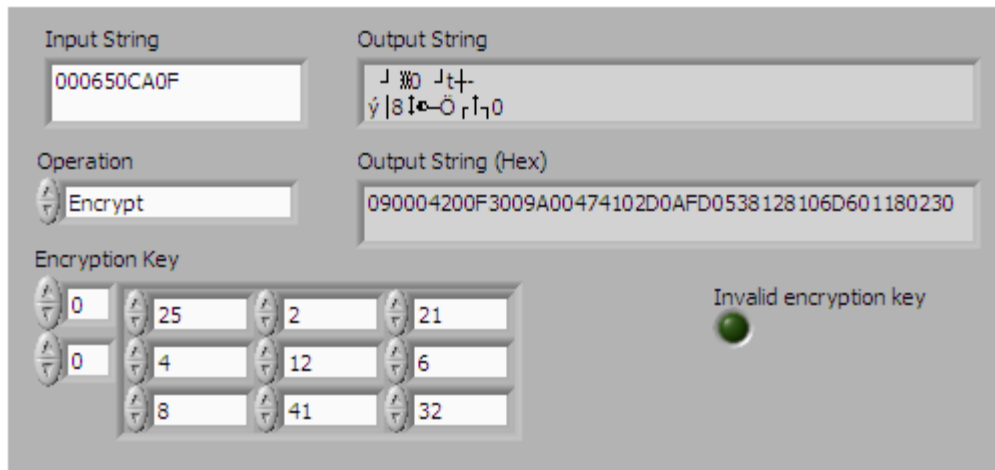
To create a license key for a specific hardware target, you must first create a unique system ID. The system ID is a number or string that uniquely identifies a specific hardware target based on the physical hardware itself. The licensed application is locked to the hardware target using the license key, which is linked to the system ID. The system ID can be a number such as the serial number of the target or a value that is a combination of each of the system component's serial numbers. Another source for a system ID can be the Media Access Control (MAC) address stored in every Ethernet interface chipset. The example uses the Reference Library for Reading CompactRIO System Configuration Information to retrieve these different pieces of information from a CompactRIO system.



**Figure 6.30. Block Diagram Showing the Process of Creating the License Key**

Figure 11.31 shows one possible example of generating a system ID for a 4-slot CompactRIO system using these VIs. The serial numbers for all six system components are added together. While this is not a truly unique value for a given CompactRIO system, it is unlikely that after replacing one or more system components, the sum of all the serial numbers will be the same as when the license key was generated.



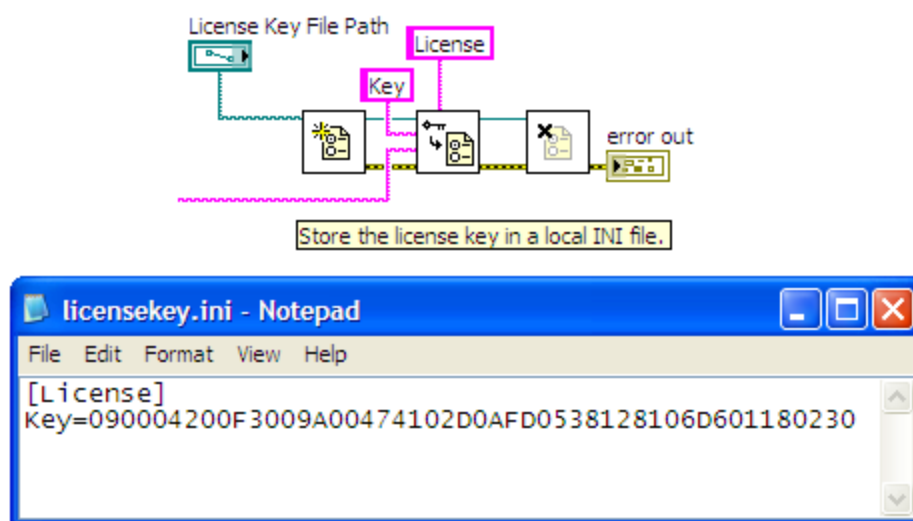


**Figure 6.32. Encryption of the System ID Into the License Key**

The encryption VI provides two versions of the license key. The default algorithm returns a stream of bytes that can have any value between 0 and 255. Therefore these bytes may not represent printable characters and may not be easily stored in a text file or passed along using other mechanisms such as email. To simplify the process of storing and transferring the license key, the VI provides a hexadecimal version of the encrypted string made up of the ASCII representation of the hexadecimal value of each of the byte values in the original string. This string is stored in a file as the license key.

### Store the license key on the hardware target

The reference example stores the license key in a simple INI file on the CompactRIO controller hard drive.



**Figure 6.33. Storing the License Key in an INI File on the CompactRIO Controller**

If the license key and file are generated away from the actual CompactRIO system, then you must copy the license file to the CompactRIO system when you deploy the application to the controller.

### Verify the license key during application startup

When the deployed application is starting up, it needs to verify the license key and, based on the result of the verification process, adjust its behavior according to the license model. If the license key is verified correctly, the application runs normally, but if the key is not verified, it may not run at all or run in an evaluation mode.

The reference example provides a basic verification VI that is added before the actual application.

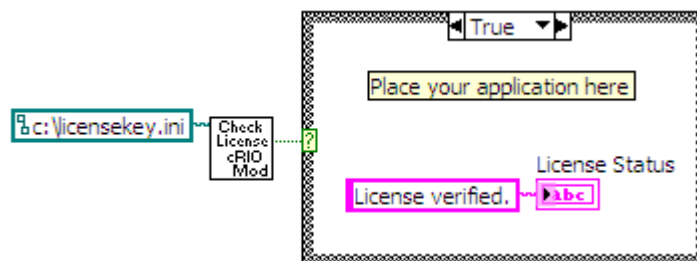
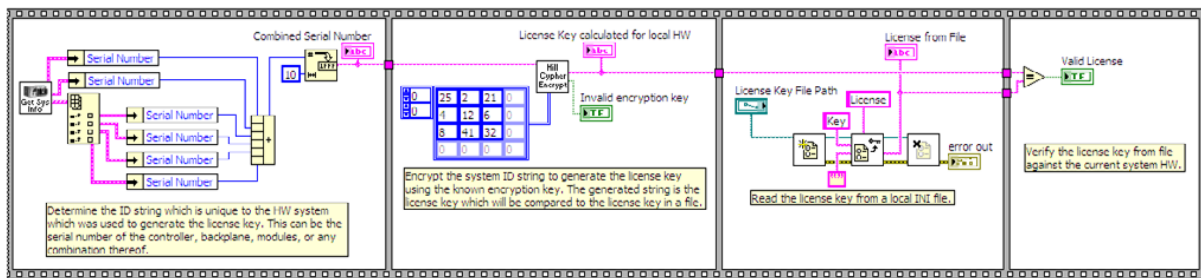


Figure 6.34. Adding the License Key Verification to an Application

You can choose from two different methods to verify a license key. The first and preferred method is to recreate the license key on the target system (described in this section). The second method, which consists of decrypting the license key, is described in the NI Developer Zone white paper titled Reference Design for Adding Licensing to LabVIEW Real-Time Applications under the section titled “Enabling Features based on the License Key”.

The more basic and more secure method to verify the license key is to run the same algorithm you use to create the license key on the target system and then compare the new license key with the license key stored in the license file. If the two match, then the license key is verified and the application may run.



**Figure 6.35. Block Diagram to Verify the License Key Stored in the File on the System**

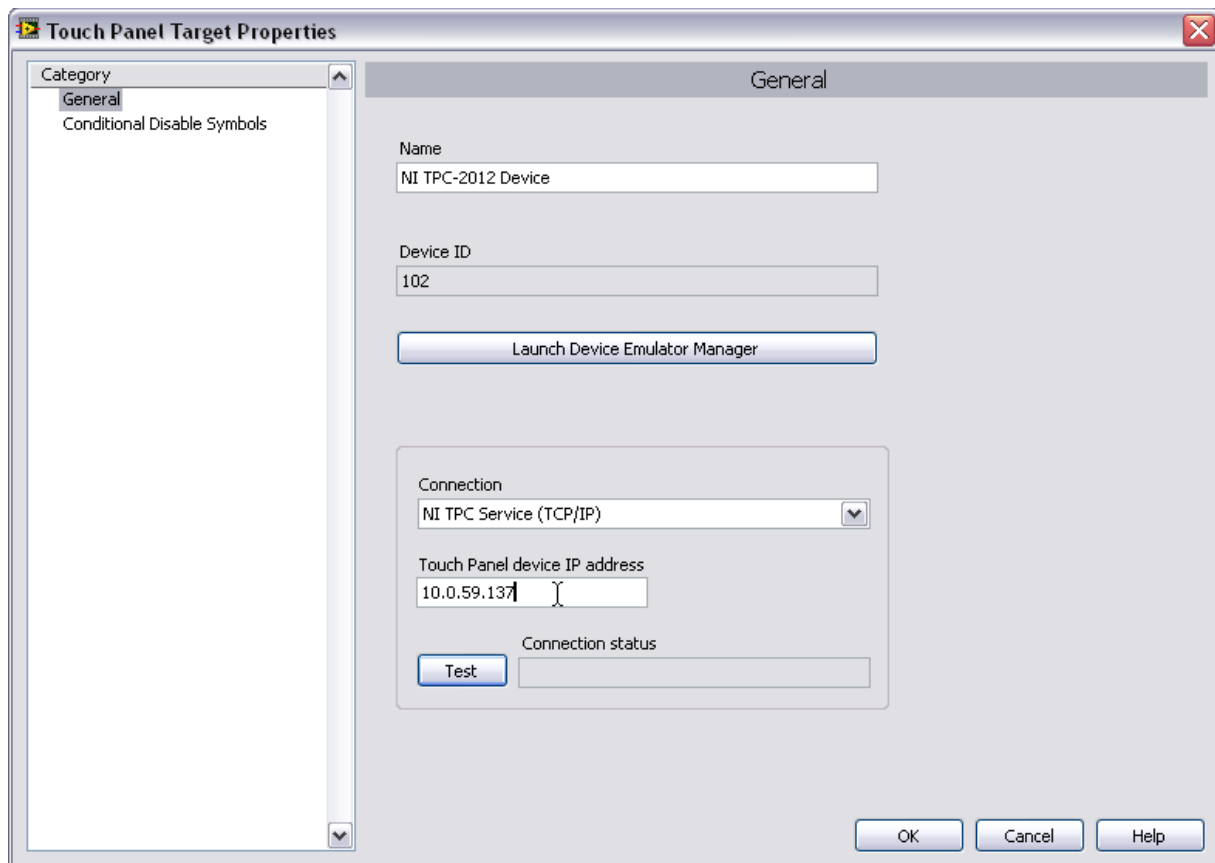
Figure 6.35 shows that this process is almost identical to the process of generating the license key. Instead of writing the license file, however, the license file is read and compared to the newly generated license key.

This method of verifying the license key works well if you do not need any partial information from the license key such as information about enabling or disabling individual features or the individual serial numbers of system components. For licensing models that require more detailed information about the license key, the key itself must be decrypted. For more information on this type of licensing, see the NI Developer Zone document *Reference Design for Adding Licensing to LabVIEW Real-Time Applications*.

## Deploying Applications to a Touch Panel

### Configure the Connection to the Touch Panel

Although you can manually copy built applications to a touch panel device, you should use Ethernet and allow the LabVIEW project to automatically download the application. NI touch panels are all shipped with a utility called the NI TPC Service that allows the LabVIEW project to directly download code over Ethernet. To configure the connection, right-click on the touch panel target in the LabVIEW project and select Properties. In the General category, choose the connection as NI TPC Service and enter the IP address of the touch panel. Test the connection to make sure the service is running.



**Figure 6.36. Connect to a touch panel through Ethernet using the NI TPC Service.**

You can find the IP address of the touch panel by going to the command prompt on the TPC and typing `ipconfig`. To get to the command prompt, go to the Start menu and select Run... In the popup window, enter `cmd`.

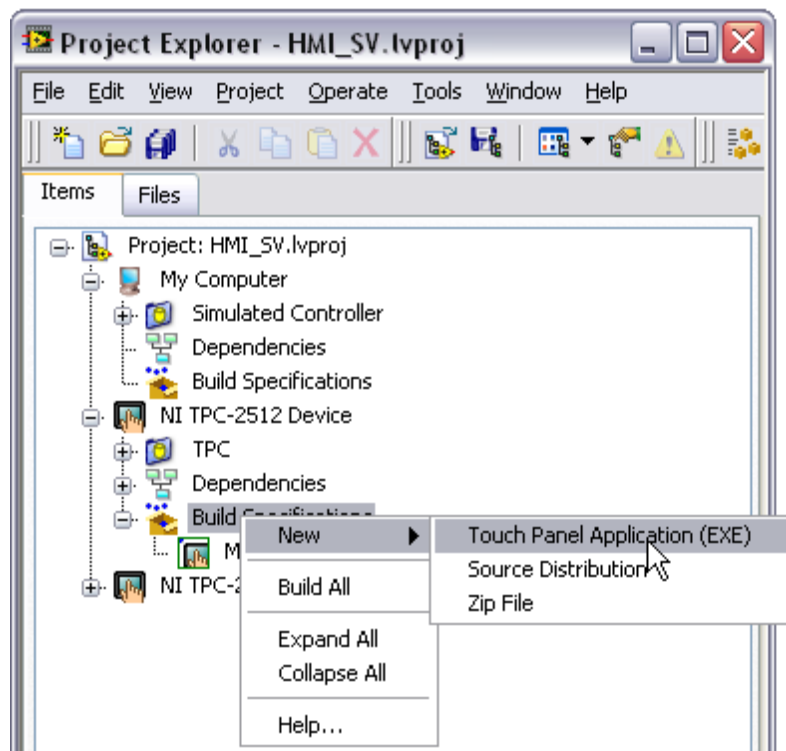
### **Deploy a LabVIEW VI to Volatile or Nonvolatile Memory**

The steps to deploy an application to a Windows XP Embedded touch panel and to a Windows CE touch panel are nearly identical. The only difference is on an XP Embedded touch panel, you can deploy an application to only the nonvolatile memory, and, on a Windows CE touch panel, you can deploy to volatile or nonvolatile memory, depending on the destination directory you select. To run a deployed VI in either volatile or nonvolatile memory on a touch panel, you must first create an executable.

### **Building an executable from a VI for an XP Embedded touch panel**

The LabVIEW project provides the ability to build an executable touch panel application from a VI. To do this, you create a build specification under the touch panel target in the LabVIEW Project Explorer. By right-clicking on Build

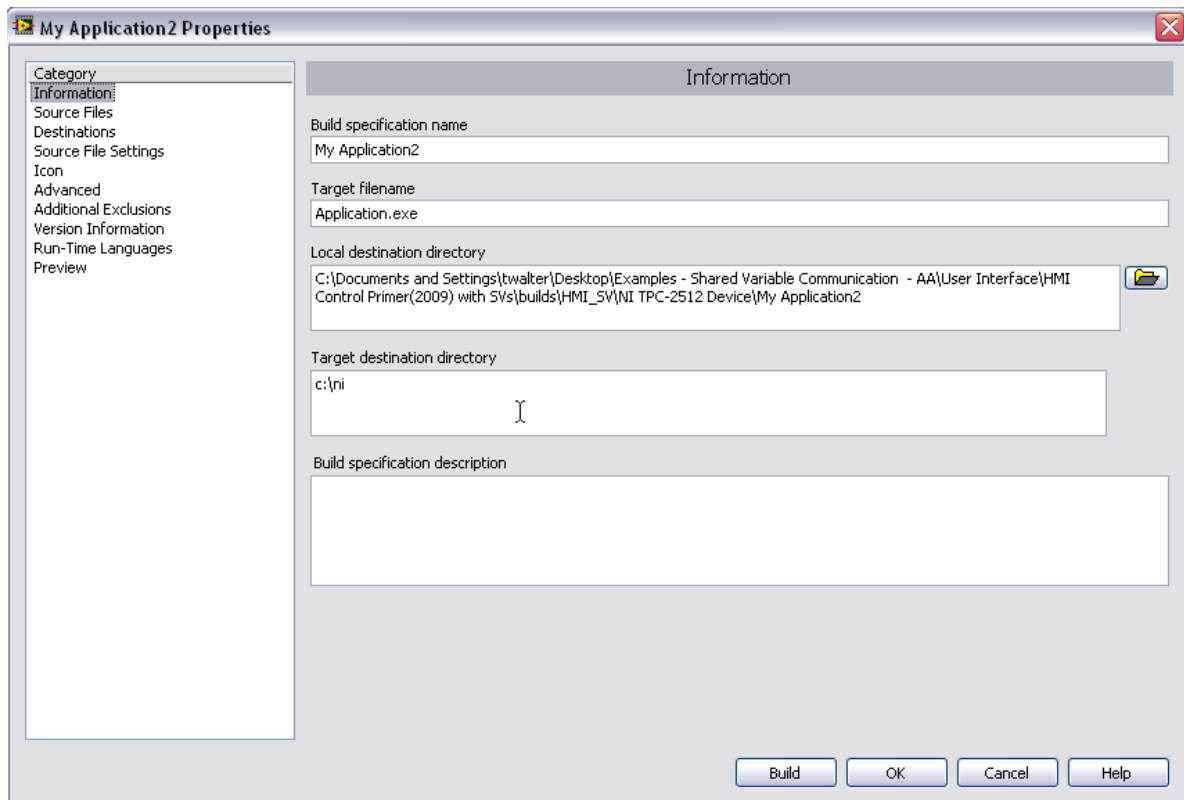
Specifications, you can select the option of creating a Touch Panel Application, Source Distribution, Zip File, and so on.



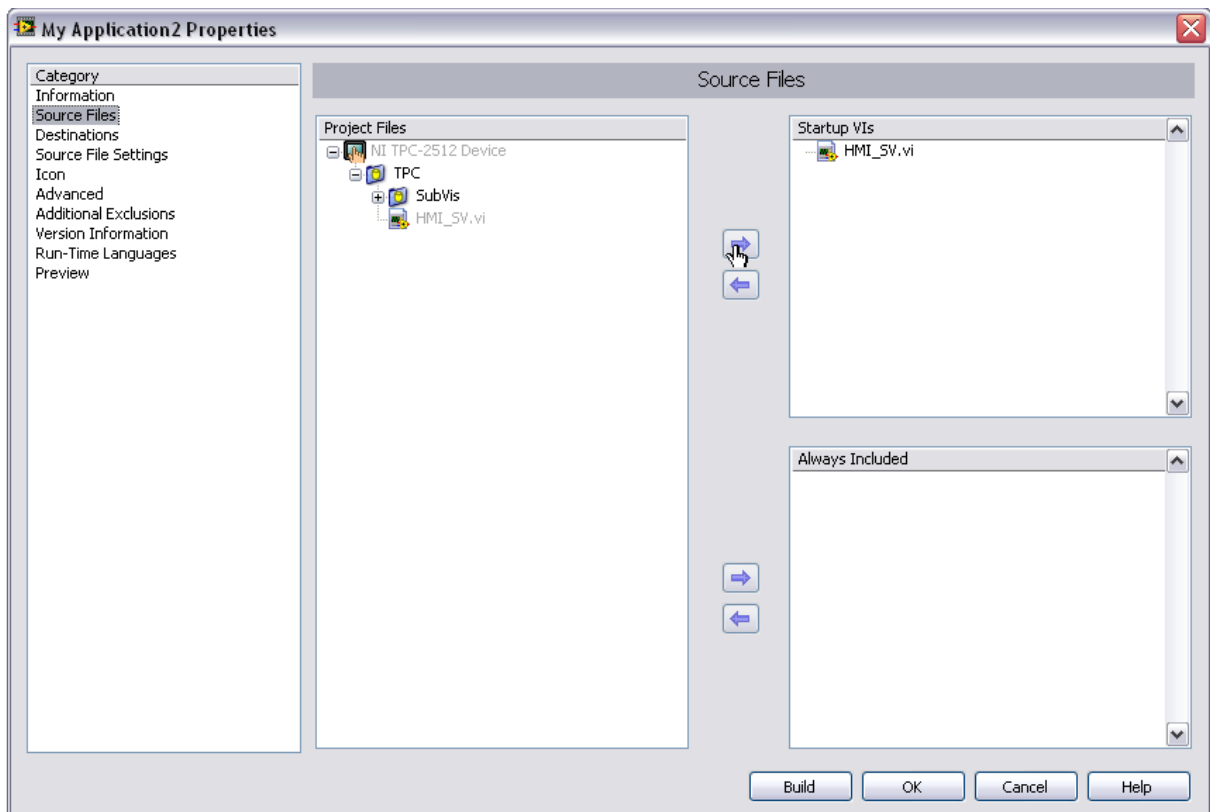
**Figure 6.37. Create a touch panel application using the LabVIEW project.**

After selecting the Touch Panel Application, you are presented with a dialog box. The two most commonly used categories when building a touch panel application are Information and Source Files. The other categories are rarely changed when building touch panel applications.

The Information category contains the build specification name, executable filename, and destination directory for both the touch panel target and host PC. You can change the build specification name and local destination directory to match your nomenclature and file organization. You normally do not need to change the target filename or target destination directory.



**Figure 6.38. The Information Category in the Touch Panel Application Properties**



**Figure 6.39. Source Files Category in the Touch Panel Application Properties (In this example, the HMI\_SV.vi was selected to be a Startup VI.)**

You use the Source Files category to set the startup VIs and obtain additional VIs or support files. You need to select the top-level VI from your Project Files

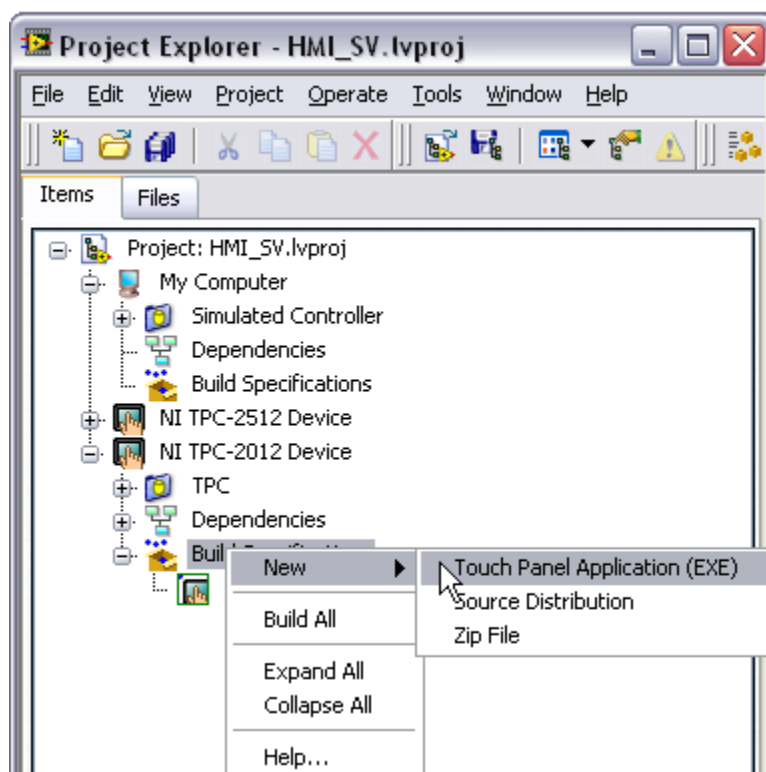
and set it as a Startup VI. For most applications, a single VI is chosen to be a Startup VI. You do not need to include Ivlb or set subVIs as Startup VIs or Always Included unless they are called dynamically in your application.

After you have entered all of the information on the required category tabs, you can click OK to save the build specification or you can directly build the application by clicking the Build button. You can also right-click on a saved build specification and select Build to build the application.

When you build the application, an executable is created and saved on the hard drive of your development machine in the local destination directory.

### **Building an executable from a VI for a Windows CE touch panel**

The LabVIEW project provides the ability to build an executable touch panel application from a VI. To build this application, you create a build specification under the touch panel target in the LabVIEW Project Explorer. By right clicking on Build Specifications, you can select the option of creating a Touch Panel Application, Source Distribution, Zip File, and so on.



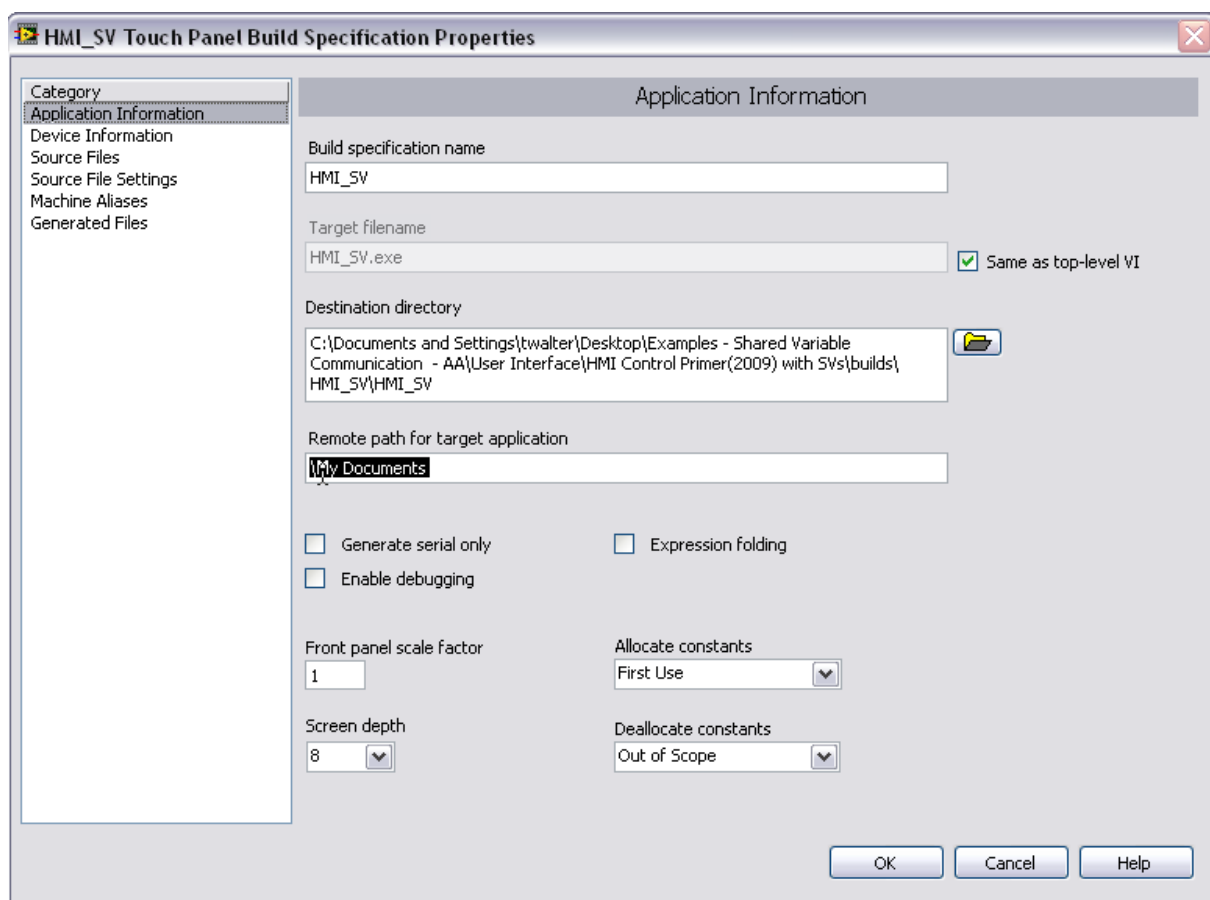
**Figure 6.40. Creating a Touch Panel Application Using the LabVIEW Project**

After selecting Touch Panel Application, you see a dialog box with the three main categories that are most commonly used when building a touch panel

application for a Windows CE target: Application Information, Source Files, and Machine Aliases. The other categories are rarely changed when building Windows CE touch panel applications.

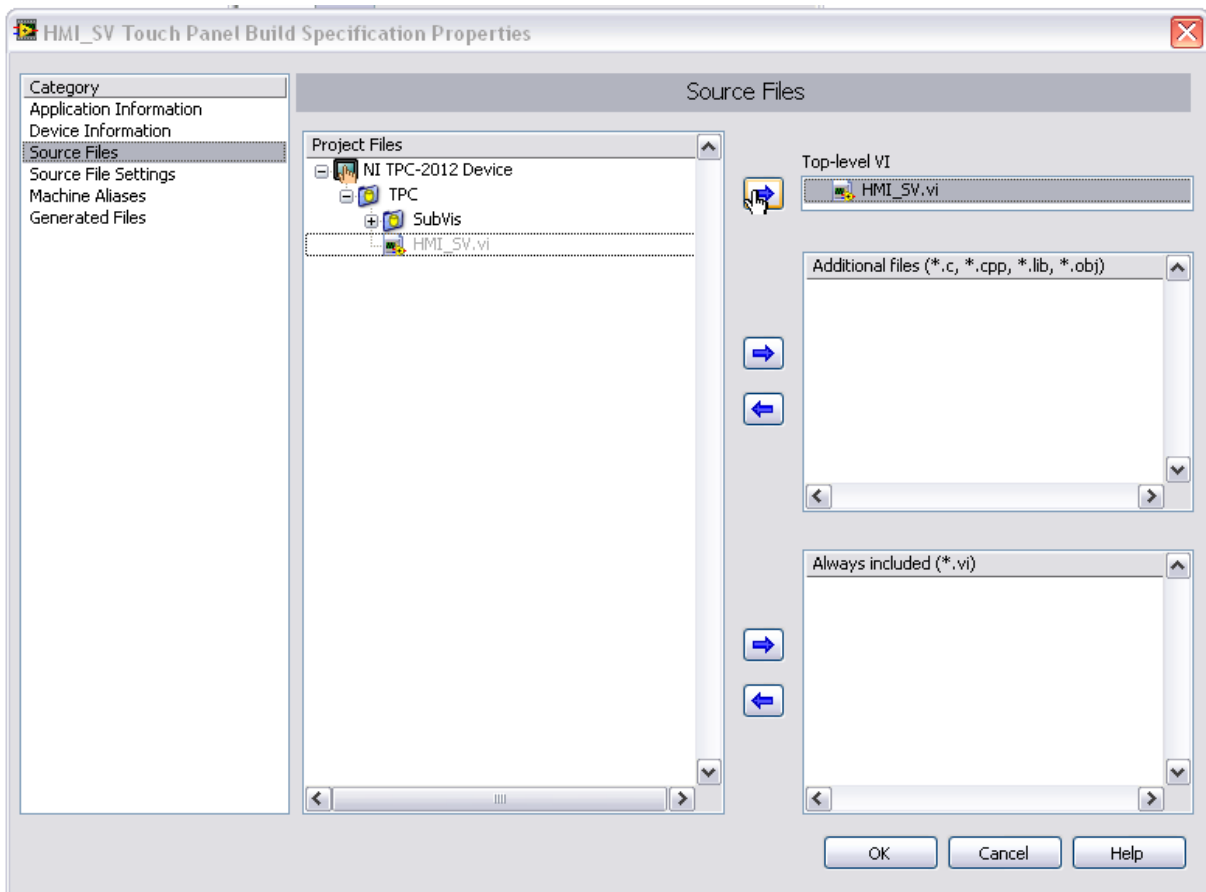
The Application Information category contains the build specification name, executable filename, and destination directory for both the touch panel target and host PC. You can change the build specification name and local destination directory to match your nomenclature and file organization. You normally do not need to change the target filename. The target destination determines if the deployed executable runs in volatile or nonvolatile memory. On a Windows CE device

- \My Documents folder is volatile memory. If you deploy the executable to this memory location, it does not persist through power cycles.
- \HardDisk is nonvolatile memory. If you want your application to remain on the Windows CE device after a power cycle, you should set your remote path for target application to a directory on the \HardDisk such as \HardDisk\Documents and Settings.



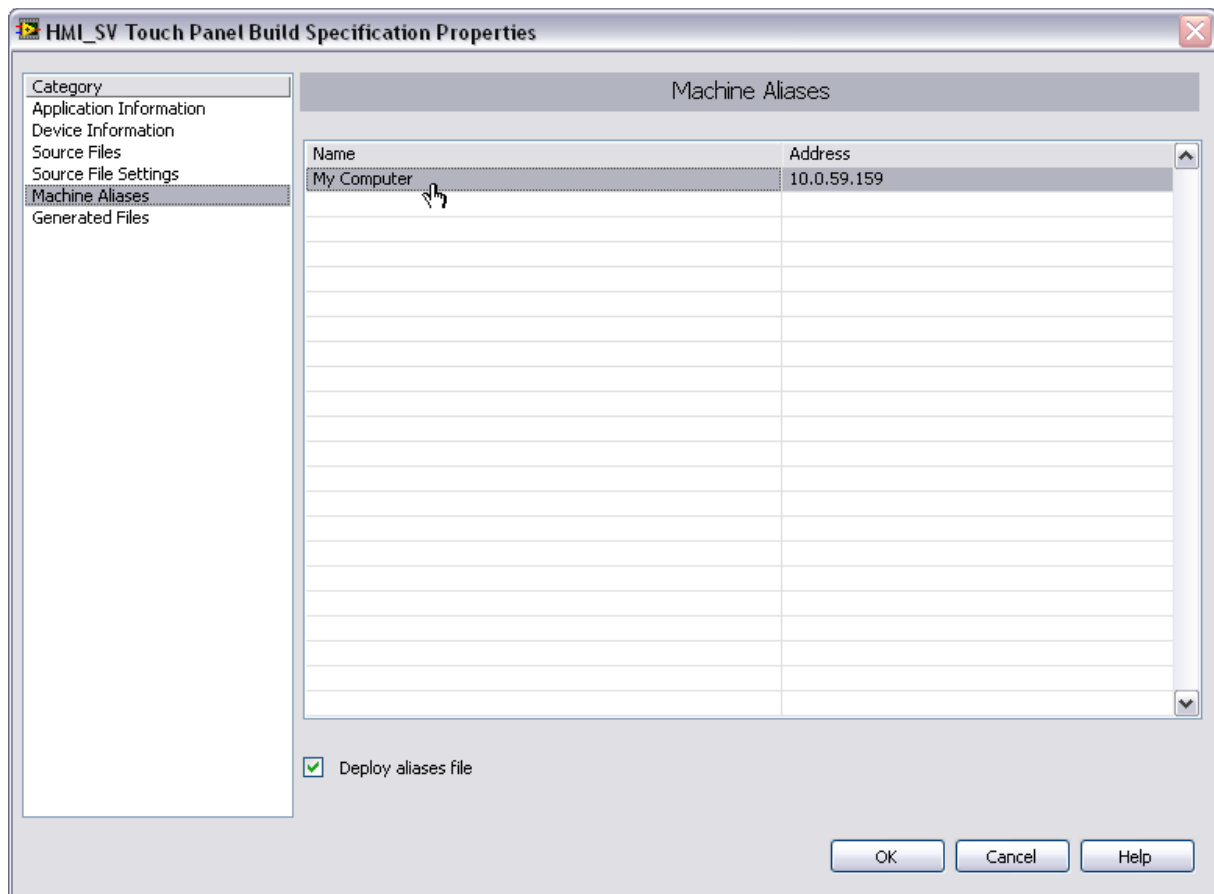
**Figure 6.41. The Information Category in the Touch Panel Application Properties**

Use the Source Files category to set the startup VI and obtain additional VIs or support files. You need to select the top-level VI from your Project File. The top-level VI is the startup VI. For Windows CE touch panel applications, you can select only a single VI to be the top-level VI. You do not need to include lvlib or subVIs as Always Included.



**Figure 6.42. Source Files Category in the Touch Panel Application Properties (In this example, the HMI\_SV.vi was selected to be the top-level VI.)**

The Machine Aliases category is used to deploy an alias file. This is required if you are using network-published shared variables for communication to any devices. Be sure to check the Deploy alias file checkbox. The alias list should include your network-published shared variable servers and their IP addresses (normally CompactRIO or Windows PCs). You can find more information on alias files and application deployment using network-published shared variables in the section titled Deploying Applications That Use Network-Published Shared Variables.



After you have entered all of your information on the required category tabs, you can click OK to save the build specification or you can directly build the application by clicking the Build button. You can also right-click on a saved build specification and select Build to build the application.

When you build the application, an executable is created and saved on the hard drive of your development machine in the local destination directory.

## Deploy an Executable Touch Panel Application to a Windows CE or XP Embedded Target

After configuring and building your executable, you now need to copy the executable and supporting files to the memory on the touch panel. To copy the files, right-click on the Touch Panel Application and select Deploy. Behind the scenes, LabVIEW copies the executable files to the memory on the touch panel. If you rebuild an application, you must redeploy the touch panel application for the changes to take effect on the touch panel target.

## The Run Button

If you click the Run button on a VI targeted to a touch panel target, LabVIEW guides you through creating a build specification (if one does not exist) and deploys the code to the touch panel target.

### Setting an executable touch panel application to Run on Startup

After you have deployed an application to the touch panel, you can set the executable so it automatically starts up as soon as the touch panel boots. Because you are running on a Windows system, you do this using standard Windows tools. In Windows XP Embedded, you should copy the executable and paste a shortcut into the Startup directory on the Start Menu. On Windows CE, you need to go to the STARTUP directory on the hard disk and modify the startup. ini file to list the path to the file (\HardDisk\Documents and Settings\HMI\_SV.exe). You can alternatively use the Misc tab in the Configuration Utility (**Start»Programs»Utilities»Configuration Utilities**) to configure a program to start up on boot. This utility modifies the startup.ini file for you.

### Porting to Other Platforms

This guide has focused on architectures for building embedded control systems using CompactRIO systems. The same basic techniques and structures also work on other NI control platforms including PXI and NI Single-Board RIO. Because of this, you can reuse your algorithms and your architecture for other projects that require different hardware or easily move your application between platforms. However, CompactRIO has several features to ease learning and speed development that are not available on all targets. This section covers the topics you need to consider when moving between platforms and shows you how to port an application to NI Single-Board RIO.

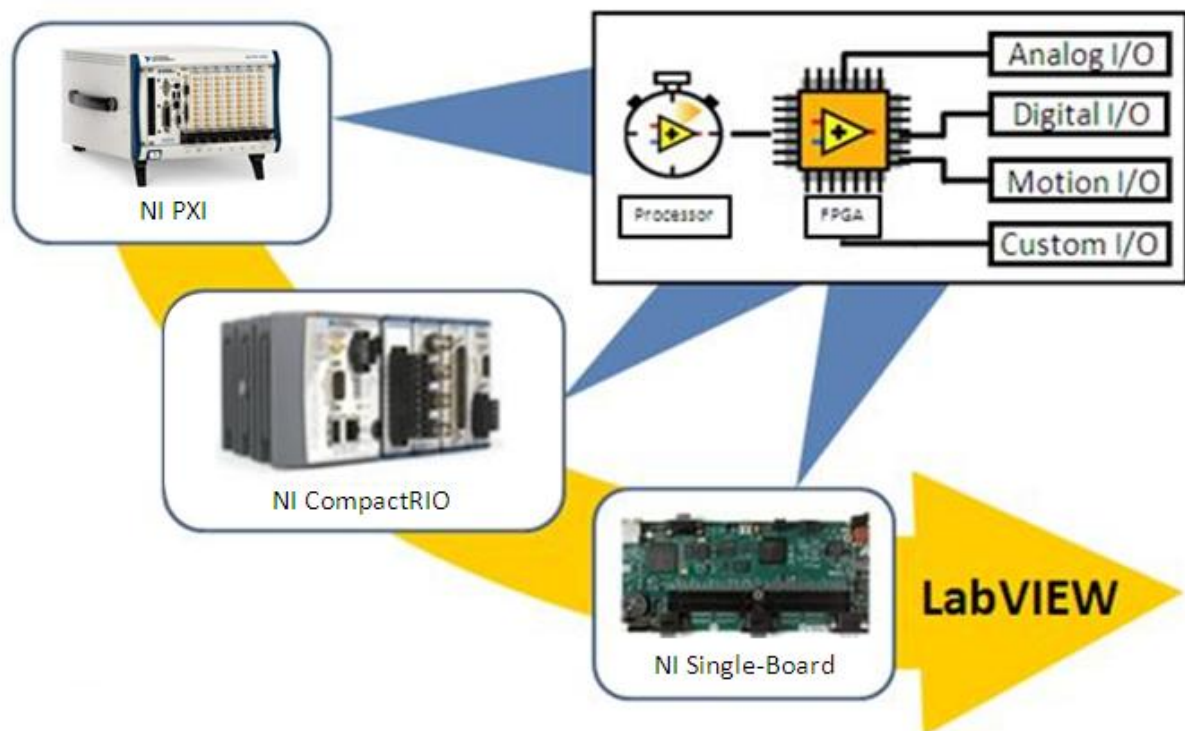


Figure 6.44. With LabVIEW, you can use the same architecture for applications ranging from CompactRIO to high-performance PXI to board-level NI Single-Board RIO.

### LabVIEW Code Portability

LabVIEW is a cross-platform programming language capable of compiling for multiple processor architectures and OSs. In most cases, algorithms written in LabVIEW are portable among all LabVIEW targets. In fact, you can even take LabVIEW code and compile it for any arbitrary 32-bit processor to port your LabVIEW code to custom hardware. When porting code between platforms, the most commonly needed changes are related to the physical I/O changes of the hardware.

When porting code between CompactRIO targets, all I/O is directly compatible because C Series modules are supported on all CompactRIO targets. If you need to port an application to NI Single-Board RIO, all C Series modules are supported, but, depending on your application, you may need to adjust the software I/O interface.

### NI Single-Board RIO

NI Single-Board RIO is a board-only version of CompactRIO designed for applications requiring a bare board form factor. While it is physically a different design, NI Single-Board RIO uses the processor and FPGA, and most models

accept up to three C Series modules. NI Single-Board RIO differs from CompactRIO because it includes I/O built directly into the board. NI offers three families of NI-Single-Board RIO products:

### **Digital I/O With RIO Mezzanine Card Connector**

The smallest option for NI Single-Board RIO combines the highest performance real-time processor with a Xilinx Spartan-6 FPGA and built-in peripherals such as USB, RS232, CAN, and Ethernet. In addition to the peripherals, the system includes 96 FPGA digital I/O lines that are accessed through the RIO Mezzanine Card (RMC) connector, which is a high-density, high-bandwidth connector that allows for direct access to the FPGA and processor. With this type of NI Single-Board RIO, you can create a customized daughtercard designed specifically for your application that accesses the digital I/O lines and processor I/O including CAN and USB. This NI Single-Board RIO family currently does not support C Series connectivity.

### **Digital I/O Only or Digital and Analog I/O With Direct C Series Connectivity**

NI also offers NI Single-Board RIO devices with both built-in digital and analog I/O on a single board. All I/O is connected directly to the FPGA, providing low-level customization of timing and I/O signal processing. These devices feature 110 3.3 V bidirectional digital I/O lines and up to 32 analog inputs, 4 analog outputs, and 32 24 V digital input and output lines, depending on the model used. They can directly connect up to three C Series I/O and communication modules for further I/O expansion and flexibility.

### **LabVIEW FPGA programming**

Not all NI Single-Board RIO products currently support Scan Mode. Specifically, the NI Single-Board RIO products with a 1 million gate FPGA (sbRIO-9601, sbRIO-9611, sbRIO-9631, and sbRIO-9641) are not supported, in addition to the NI Single-Board RIO products with the RMC connector (sbRIO-9605 and sbRIO-9606). Instead of using Scan Mode to read I/O, you need to write a LabVIEW program to read the I/O from the FPGA and insert it into an I/O memory table. This section examines an effective FPGA architecture for single-point I/O communication similar to Scan Mode and shows how to convert an application using Scan Mode.

### **Built-in I/O and I/O modules**

Depending on your application I/O requirements, you may be able to create your entire application to use only the NI Single-Board RIO onboard I/O, or you may need to add modules. When possible, design your application to use the I/O modules available onboard NI Single-Board RIO. The I/O available on NI Single-Board RIO with direct C Series connectivity and the module equivalents are listed below:

- 110 general purpose, 3.3 V (5 V tolerant, TTL compatible) digital I/O (no module equivalent)
- 32 single-ended/16 differential channels, 16-bit analog input, 250 kS/s aggregate (NI 9205)
- 4-channel, 16-bit analog output; 100 kS/s simultaneous (NI 9263)
- 32-channel, 24 V sinking digital input (NI 9425)
- 32-channel, 24 V sourcing digital output (NI 9476)214

This NI Single-Board RIO family accepts up to three additional C Series modules. Applications that need more than three additional I/O modules are not good candidates for NI Single-Board RIO, and you should consider CompactRIO integrated systems as a deployment target.

### **FPGA size**

The largest FPGA available on NI Single-Board RIO is the Spartan-6 LX45. CompactRIO targets offer versions using both the Virtex-5 FPGAs and the Spartan-6 FPGAs as large as the LX150. To test if code fits on hardware you do not own, you can add a target to your LabVIEW project and, as you develop your FPGA application, you can periodically benchmark the application by compiling the FPGA code for a simulated RIO target. This gives you a good understanding of how much of your FPGA application will fit on the Spartan-6 LX45.

### **Port CompactRIO Applications to NI Single-Board RIO or R Series Devices**

Follow these four main steps to port a CompactRIO application to NI Single-Board RIO or PXI/PCI R Series FPGA I/O devices.

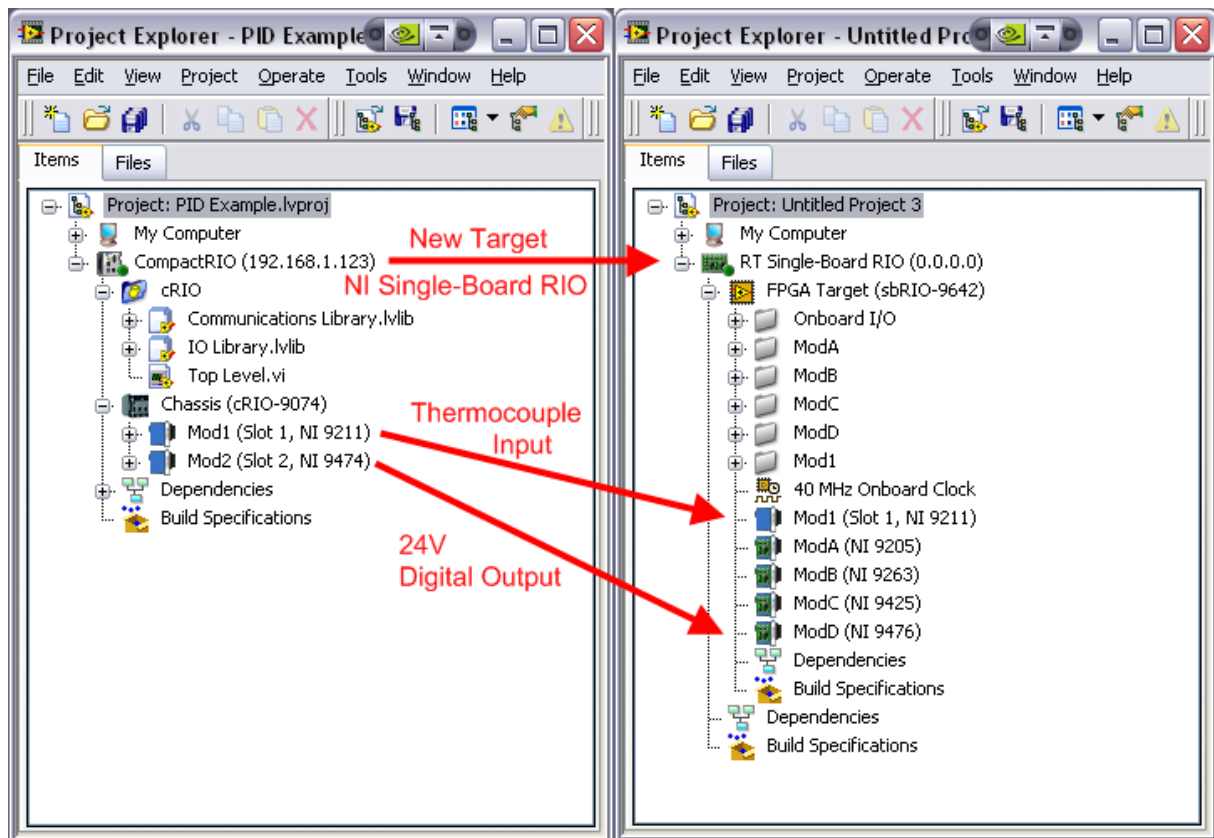
1. Build an NI Single-Board RIO or R Series project with equivalent I/O channels.

2. If using the NI RIO Scan Interface, build a LabVIEW FPGA-based scan API if porting to unsupported NI Single-Board RIO devices or to PXI/PCI R Series FPGA I/O Devices
  - Build LabVIEW FPGA I/O scan (analog in, analog out, digital I/O, specialty digital I/O).
  - Convert I/O variable aliases to single-process shared variables with real-time FIFO enabled.
  - Build a real-time I/O scan with scaling and a shared variable-based current value table.
3. Compile LabVIEW FPGA VI for new target.
4. Test and validate updated real-time and FPGA code.

The first step in porting an application from CompactRIO to NI Single-Board RIO or an R Series FPGA device is finding the equivalent I/O types on your target platform. For I/O that cannot be ported to the onboard I/O built into NI Single-Board RIO or R Series targets, you can add C Series modules. All C Series modules for CompactRIO are compatible with both NI Single-Board RIO and R Series. You must use the NI 9151 R Series expansion chassis to add C Series I/O to an R Series DAQ device.

Step two is necessary only if the application being ported was originally written using the NI RIO Scan Interface, and if porting to unsupported NI Single-Board RIO or PXI/PCI R Series FPGA I/O Devices. If you need to replace the NI RIO Scan Interface portion of an application with an I/O method supported on all RIO targets, an example is included below to guide you through the process.

If the application you are migrating to NI Single-Board RIO or PXI/PCI R Series did not use the RIO Scan Interface, the porting process is nearly complete. Skip step 2 and add your real-time and FPGA source code to your new NI Single-Board RIO project, recompile the FPGA VI, and you are now ready to run and verify application functionality. Because CompactRIO and NI Single-Board RIO are both based on the RIO architecture and reusable modular C Series I/O modules, porting applications between these two targets is simple.



**Figure 6.45. The first step in porting an application from CompactRIO to an alternate target is finding replacement I/O on the future target.**

### **Example of Porting a RIO Scan Interface-Based Application to use LabVIEW FPGA**

If you used the RIO Scan Interface in your original application, you might need to create a simplified FPGA version of the Scan Engine. Use these three steps to replace the RIO Scan Interface with a similar FPGA-based scan engine and current value table:

1. Build a LabVIEW FPGA I/O scan engine
2. Replace scan engine I/O variables with single-process shared variables
3. Write FPGA data to current value table in LabVIEW Real-Time

First, create a LabVIEW FPGA VI that samples and updates all analog input and output channels at the rate specified in your scan engine configuration. You can use IP blocks to recreate specialty digital functionality such as counters, PWM, and quadrature encoder. Next, create an FPGA Scan Loop which synchronizes the I/O updates by updating all outputs and reading the current value of all inputs in sequence.

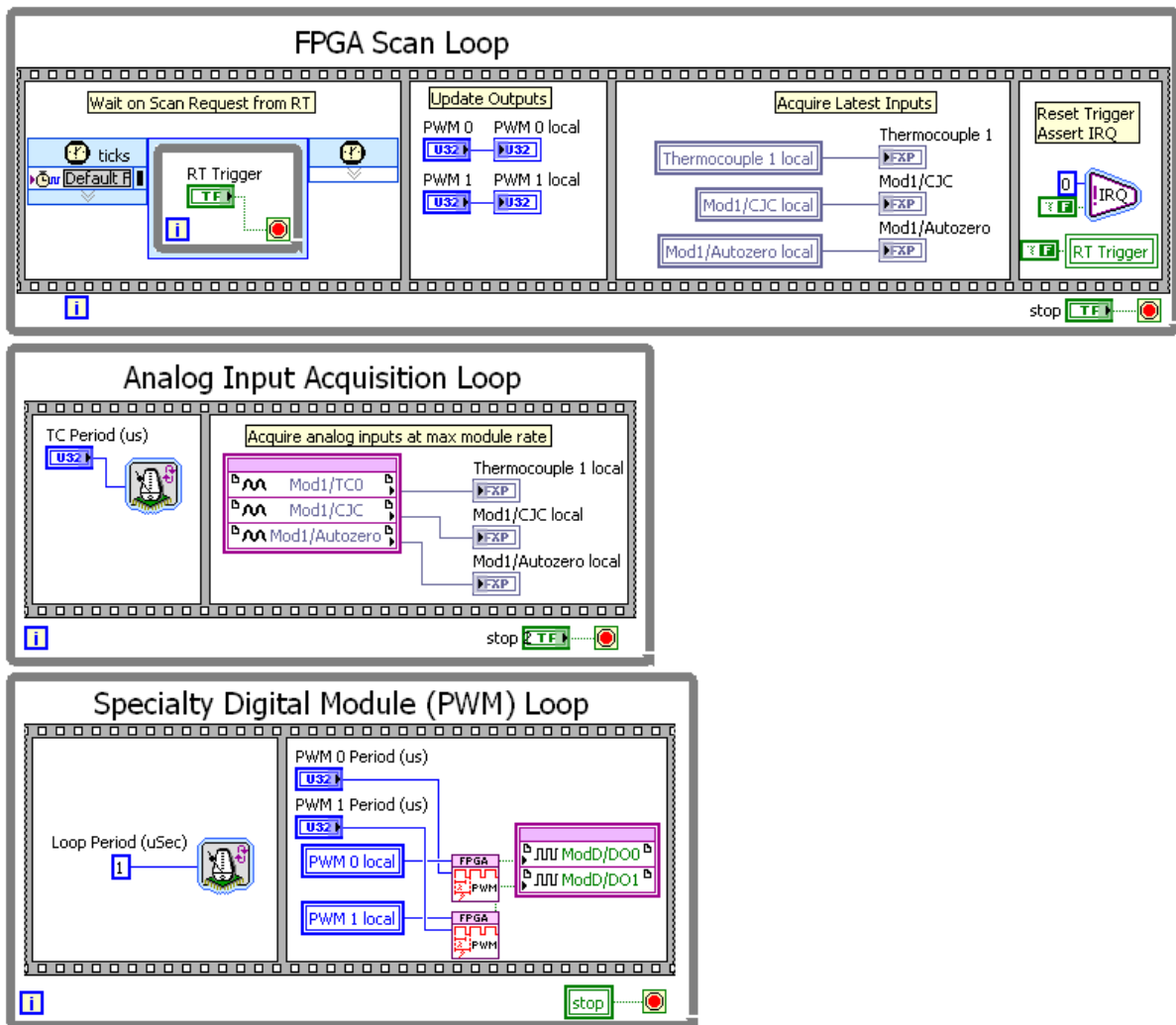
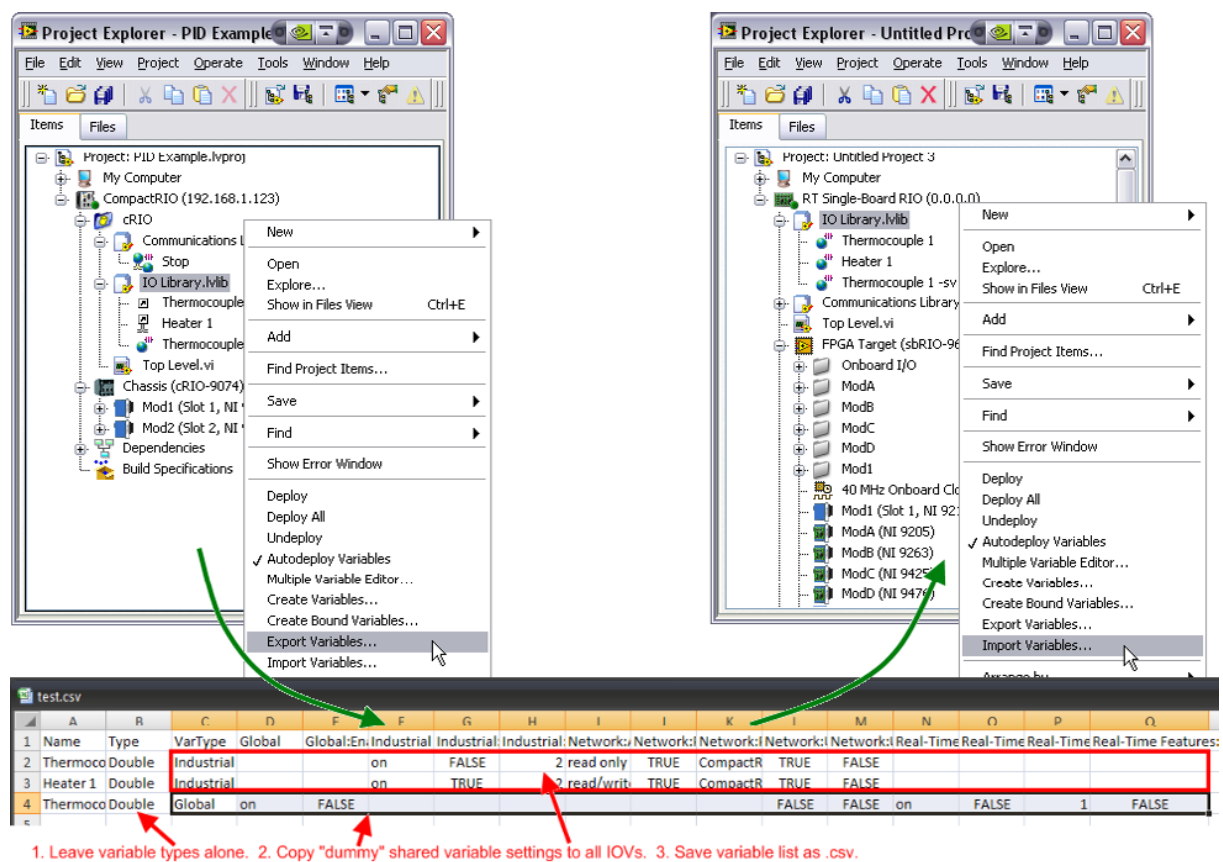


Figure 6.46. Develop a simple FPGA application to act as an FPGA scan engine.

After you have implemented a simple scan engine in the FPGA, you need to port the real-time portion of the application to communicate with the custom FPGA scan engine rather than the scan engine I/O variables. To accomplish this, you need to first convert all I/O variable aliases to single-process shared variables with the real-time FIFO enabled. The main difference between the two variables is while I/O variables are automatically updated by a driver to reflect the state of the input or output channel, single-process shared variables are not updated by a driver. You can change the type by going to the properties page for each I/O variable alias and changing it to single-process.

**Tip:** If you have numerous variables to convert, you can easily convert a library of I/O variable aliases to shared variables by exporting to a text editor and changing the properties. To make sure you get the properties correct, you should first create one “dummy” single-process shared variable with the single-

element real-time FIFO enabled in the library and then export the library to a spreadsheet editor. While in the spreadsheet editor, delete the columns exclusive to I/O variables and copy the data exclusive to the shared variables to the I/O variable rows. Then import the modified library into your new project. The I/O variable aliases are imported as single-process shared variables. Because LabVIEW references shared variables and I/O variable aliases by the name of the library and the name of the variable, all instances of I/O variable aliases in your VI are automatically updated. Finally, delete the dummy shared variable that you created before the migration process.

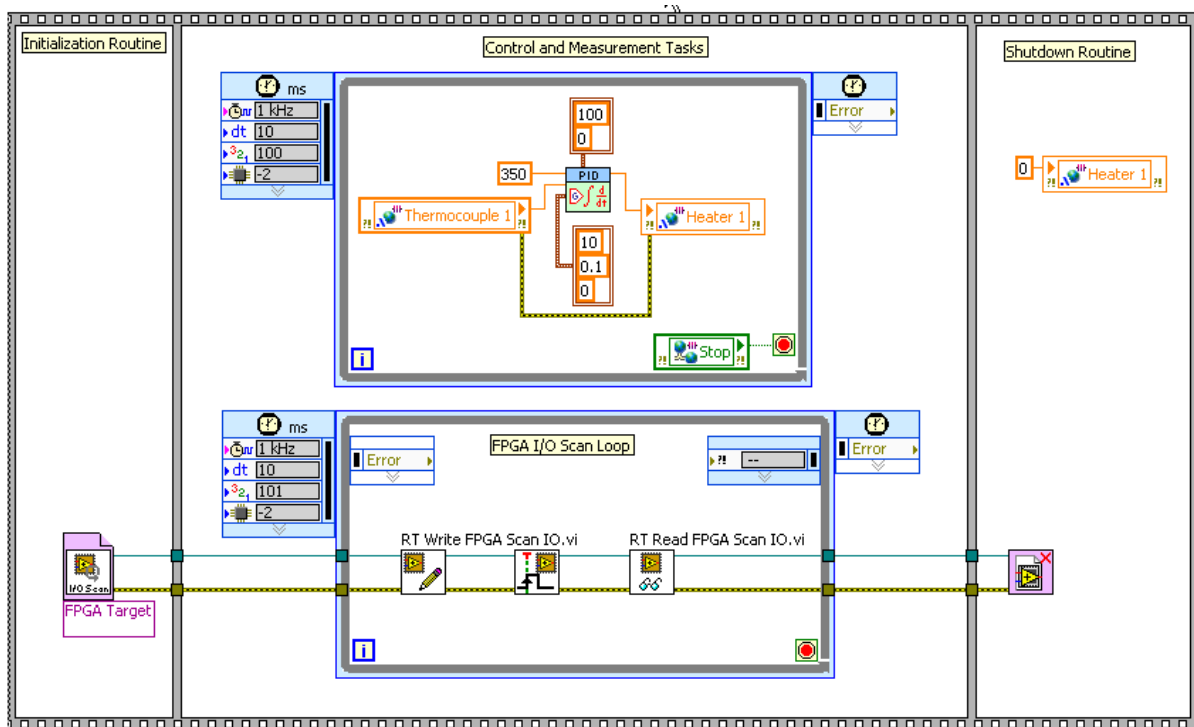


**Figure 6.47. You can easily convert an I/O variable (IOV) Alias Library to shared variables by exporting the variables to a spreadsheet, modifying the parameters, and importing them into your new target.**

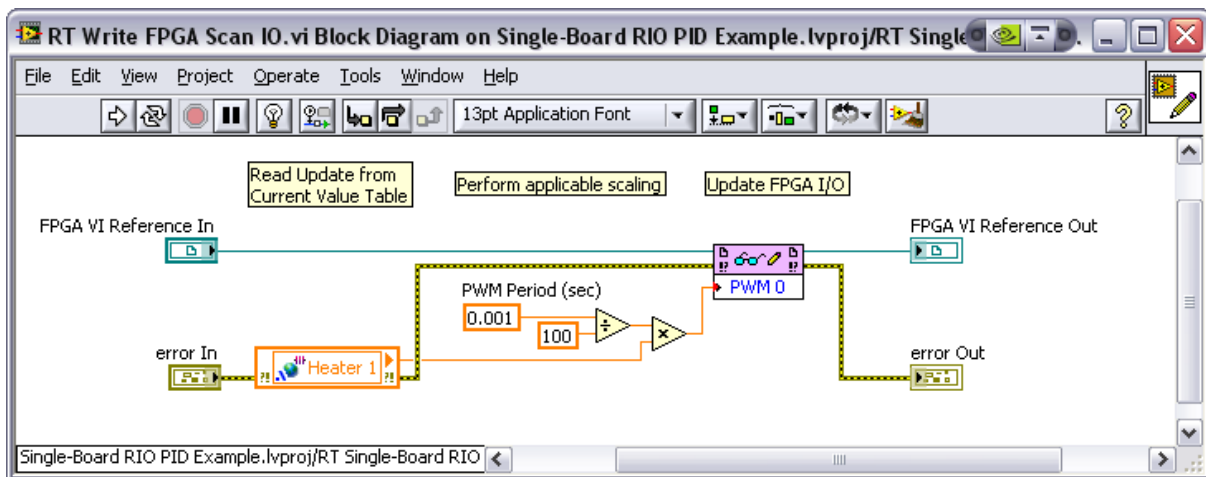
The final step for implementing an FPGA scan engine is adding a real-time process to read data from the FPGA and constantly update the current value table. The FPGA I/O you are adding to the shared variables is deterministic, so you can use a Timed Loop for implementing this process.

To read data from the FPGA scan engine, create a Timed Loop task set to the desired scan rate in your top-level RT VI. This Timed Loop is the deterministic I/O loop, so you should set it to the highest priority. To match the control loop speed of your previous Scan Mode application, set the period of this loop to match the period previously set for Scan Mode. Any other task loops in your application that were previously synchronized to the Scan Mode also need to change their timing sources to the 1 kHz clock and set to the same rate as the I/O scan loop.

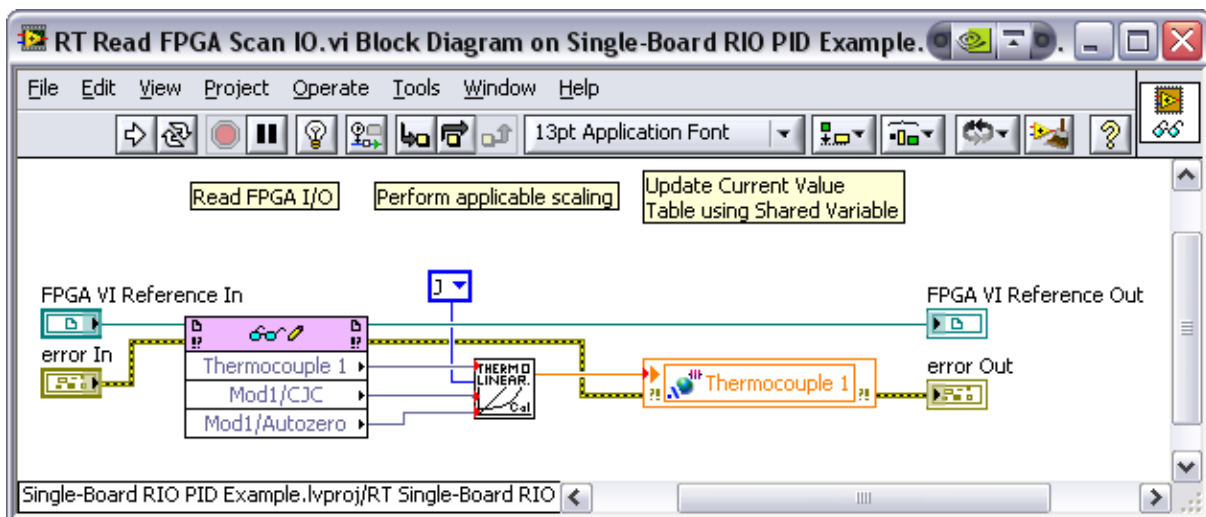
The I/O scan loop pushes new data to the FPGA and then pulls updated input values. The specific write and read VIs are also responsible for the scaling and calibration of analog and specialty digital I/O.



**Figure 6.48.** The FPGA I/O scan loop mimics the RIO Scan Interface feature by deterministically communicating the most recent input and output values to and from the FPGA I/O and inserting the data into a current value table.



**Figure 6.49.** The RT Write FPGA Scan IO VI pulls current data using a real-time FIFO single-process shared variable, scales values with appropriate conversion for the FPGA Scan VI, and pushes values to the FPGA VI.



**Figure 6.50.** The RT Read FPGA Scan IO VI pulls all updates from the FPGA Scan IO VI, performs applicable conversions and scaling, and publishes data to a current value table using a real-time FIFO single-process shared variable.

After building the host interface portion of a custom FPGA I/O scan to replace Scan Mode, you are ready to test and validate your ported application on the new target. Ensure the FPGA VI is compiled and the real-time and FPGA targets in the project are configured correctly with a valid IP address and RIO resource name. After the FPGA VI is compiled, connect to the real-time target and run the application.

Because the RIO architecture is common across NI Single-Board RIO, CompactRIO, and R Series FPGA I/O devices, LabVIEW code written on each of these targets is easily portable to the others. As demonstrated in this section,

with proper planning, you can migrate applications between all targets with no code changes at all. When you use specialized features of one platform, such as the RIO Scan Interface, the porting process is more involved, but, in that case, only the I/O portions of the code require change for migration. In both situations, all of the LabVIEW processing and control algorithms are completely portable and reusable across RIO hardware platforms.





Centrum pro rozvoj výzkumu pokročilých řídicích a senzorických technologií  
CZ.1.07/2.3.00/09.0031

Ústav automatizace a měřicí techniky  
VUT v Brně  
Kolejní 2906/4  
612 00 Brno  
Česká Republika

<http://www.crr.vutbr.cz>

[info@crr.vutbr.cz](mailto:info@crr.vutbr.cz)