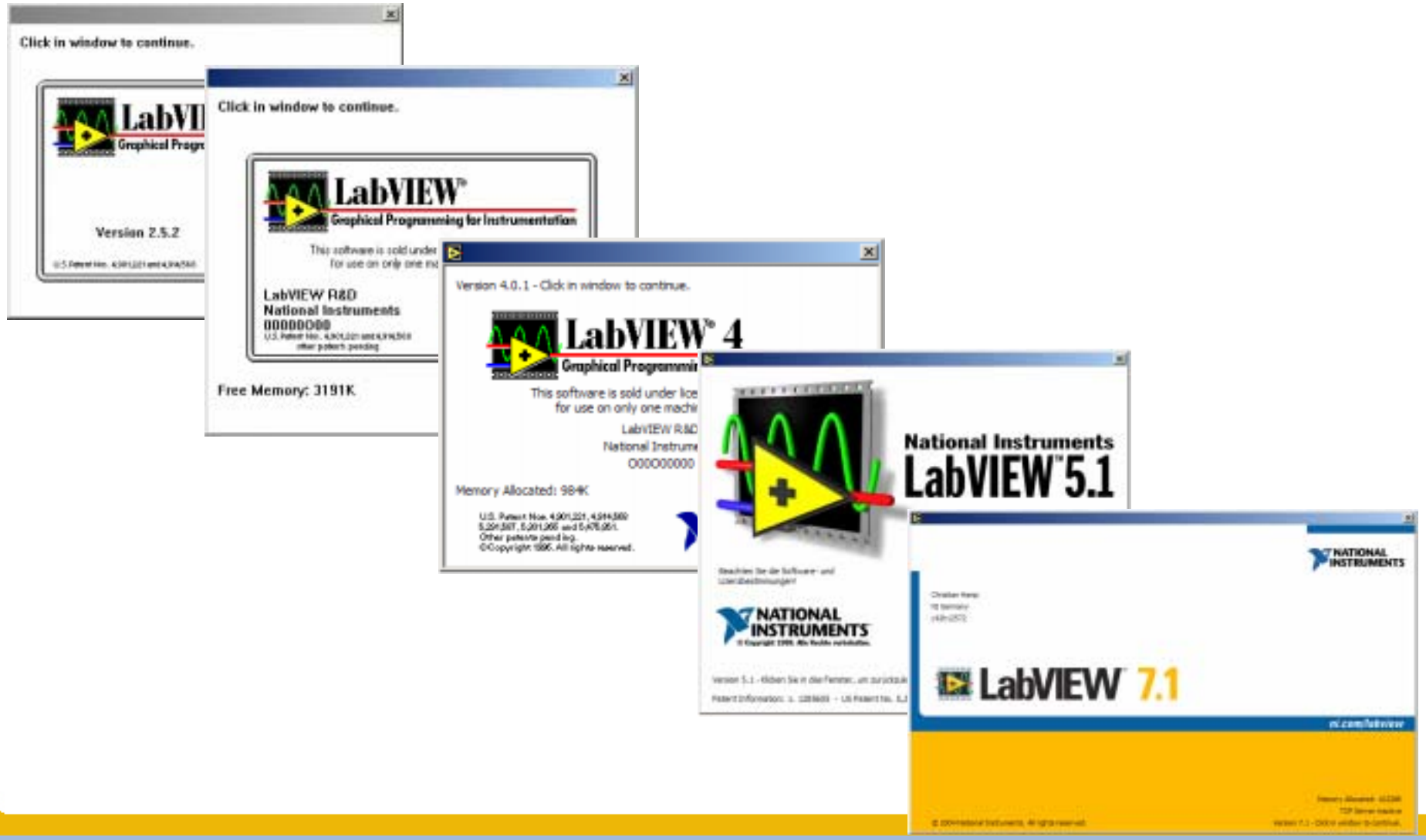
LabVIEW Advanced

Christian Hamp

Project Consultant for Automated Test

Started with LabVIEW 3

Wie lang sind Sie schon dabei?



ni.com

NATIONAL INSTRUMENTS

Agenda

- Neues in LabVIEW 7.1
- Softwareengineering in LabVIEW
  - LabVIEW Programming Guidelines
  - Programmarchitekturen und Designpatterns
  - LabVIEW Performance Issues
- Advanced Functions
  - Event-Programmierung in LabVIEW
  - Multithreading
  - Externen Code aus LabVIEW aufrufen – CINs, DLLs, .Net & Co

Neues in LabVIEW 7.1

- **Bluetooth VIs for wireless communication**
- Polynomial Computation VIs
- Faster BLAS/LAPACK-based math
- **Navigation Window**
- Exporting Controls as Images
- **Display Buffer Allocation tool**
- Support for hyperthreading
- **Modular Instrument Express VIs**
- Radio Button
- Express VI for Appending Signals
- Execution of external Xmath
- NI-DAQmx support for LabVIEW
- **LabVIEW Execution Trace Toolkit identifies sources of jitter in RT applications***
- Desktop PC support for LabVIEW Real-Time Applications*
- Expanded Data Acquisition for LabVIEW PDA applications*
- FPGA Support for new hardware and NI Compact Vision System*
- Re-use existing VHDL code in LabVIEW FPGA applications*
- Faster execution for FPGA apps*

Software Engineering in LabVIEW™

Mythos 1

“LabVIEW erfordert keine Programmierkenntnisse”

Realität ist:

LabVIEW macht das Programmieren einfacher!

Mythos 2

“Mit LabVIEW lassen sich gute Programme leichter entwickeln”

Realität ist:

Mit LabVIEW lassen sich gute und schlechte Programme leichter entwickeln

Mythos 3

“LabVIEW ist schuld, wenn ich schlechte Programme entwickle”

Realität ist:

„A fool with a tool is still a fool“

Was macht ein gutes Programm aus?

- Erfüllt die Erwartungen (Anwender-Sicht)
  - Funktioniert
  - Schnell genug
  - Ressourcen schonend
- Wartbar (Entwickler-Sicht)
  - Module einzeln wartbar
  - Änderungen an Modulen betreffen nicht sofort die gesamte Applikation

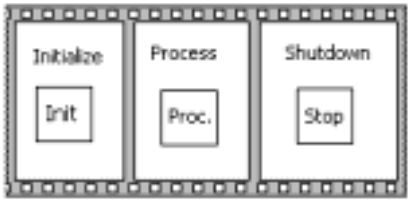
LabVIEW Programming Guidelines

SubVIs benutzen!

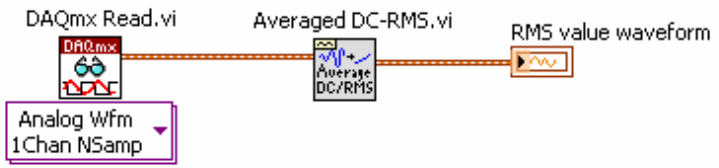
- Zusammenfassen von Funktionalitäten in VIs statt in Sequenzstrukturen
- SubVIs verbessern den Code mit minimalen Performance-Einbussen
- Zeichen dafür das ein Code mehr Sub-VIs braucht:
  - es ist mehrfach der gleiche Code vorhanden
  - Ein Diagramm ist größer als der Bildschirm
  - Sequence structures

Ablaufreihenfolge

- Auf Applikationsebene: Initialisieren, Arbeiten, Beenden
- Untere Ebene: Datei öffnen, Lesen, Schliessen
- Mit Sequenzen



- Mit Drähten



- Objektorientiert
  - z.B. GOOP
  - Referenzen statt Werte
  - Kapselung der Daten

Nicht versuchen, den Datenfluss zu vermeiden

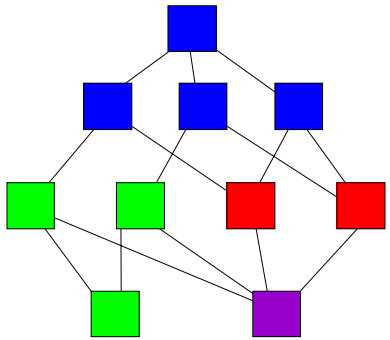
- Keine Lokale, Globalen Variablen statt langer Drähte!
  - Umgehen den Datenfluss und damit das Prinzip in LV
  - Kompliziert zu debuggen (Fehler sind u. U. erst in der EXE sichtbar...)
  - Erfordern häufig weiteren Code (Semaphoren)
- Datenfluss bringt:
  - Effizienten, parallelen Code ohne Gefahr von “Race Conditions”
  - Lesbare Programme mit sichtbaren Abhängigkeiten & Ablaufrichtung
  - Definiertes Verwenden von Variablen
  - Automatische Speicherverwaltung

Zitat der LabVIEW-Entwickler:

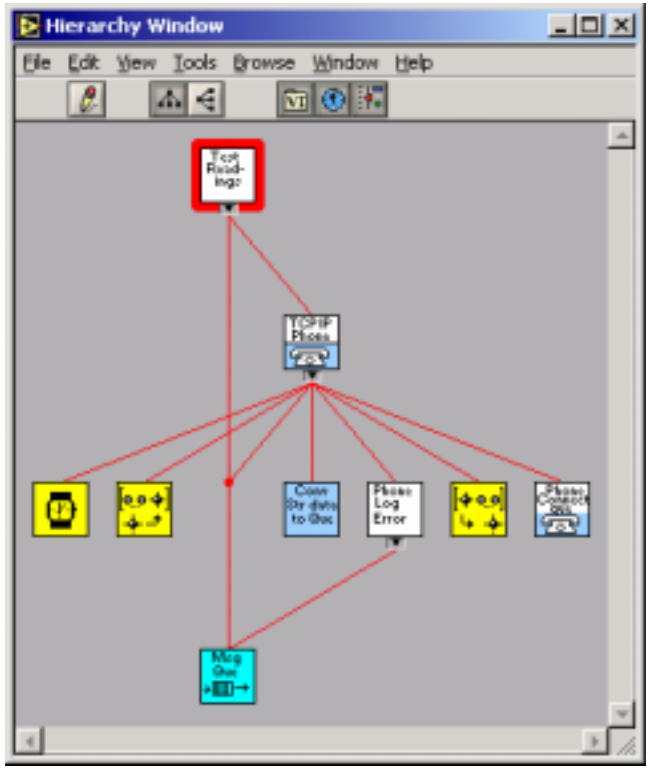
“Use of globals usually indicates failure of design”

Die VI Hierarchie-Ansicht

- Abhängigkeiten von VIs finden
- übersichtlicher, wenn Farben für verschiedene Ebenen verwendet werden (Treiber, Top-Level)
- Ebenen unabhängig halten
  - User Interface
  - Funktionskern
  - Treiber und IO
- Viele Abhängigkeiten über mehrere Ebenen deuten auf Probleme im Design hin

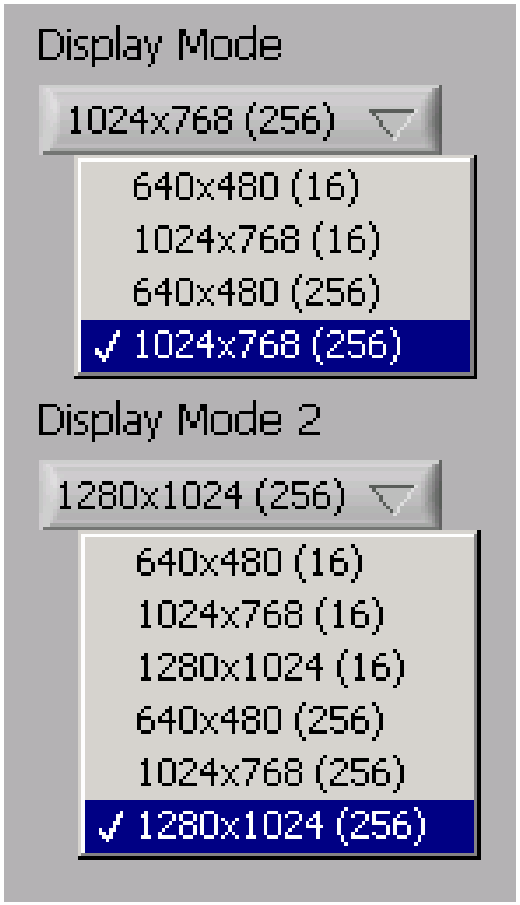


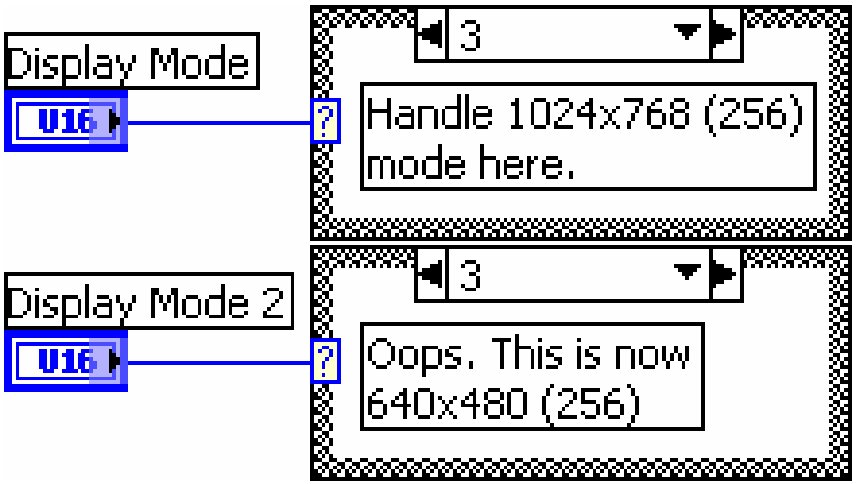
Diamond ideal



Schnittstelle und Implementierung trennen

Zum Beispiel: Angezeigten Text loslösen von Parametern im Code





Kleine Änderungen am Frontpanel erfordern u. U. große Änderungen am Programmcode

Dokumentation

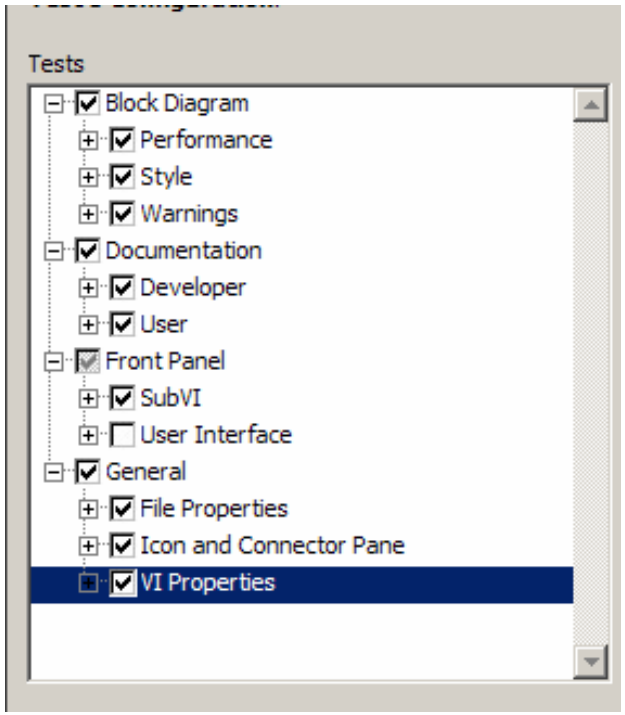
- Anwender
  - Bedienoberfläche
  - Programmierschnittstelle (API)
  - Funktionsweise, Theorie der Algorithmen
- Entwickler
  - Was ist zu beachten, wenn Funktionen weiterverwendet werden
  - Welche Codemodule sind von einander abhängig
    - Lesen und Schreiben von Dateien

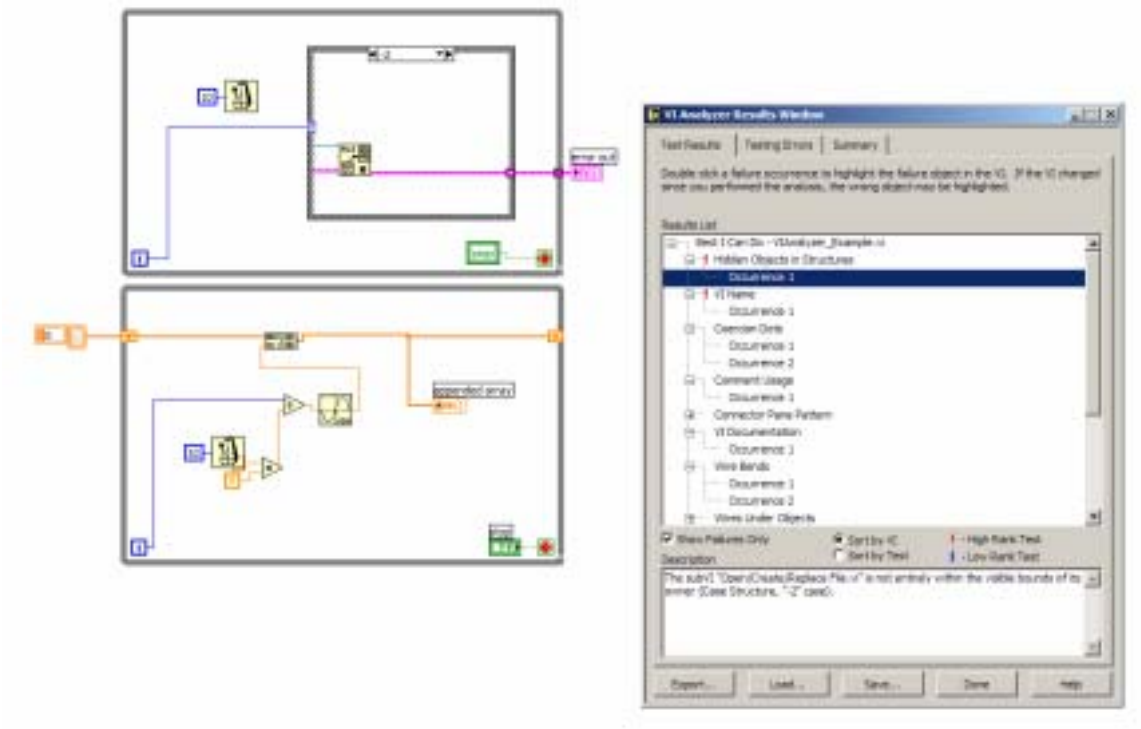
Testen

- Neben der reinen Funktion muss getestet werden...
  - Über- / Unterschreiten der Grenzen von Controls
  - Was passiert im Fehlerfall
  - Unsinnge Benutzereingaben
  - Multithreading-Aspekte
    - Sind Variablen Thread-Safe verwendet?
    - Sind DLL und CIN Aufrufe Thread-Safe?

VI Analyzer

- Analysieren von VIs nach Designkriterien
- Verifizieren von Designgrundlagen





Programmarchitekturen und Designpatterns

Programmarchitekturen und Design Patterns

Was sind Design-Patterns?

- Wiederkehrende Strukturen in Programmen
- Grundlagen für Programme
- Typische, standardisierte Lösungsansätze (z.B. Fehlerbehandlung)

"…simple and elegant solutions to specific problems in … software design. [They] capture solutions that have developed and evolved over time…"

– Design Patterns, Gamma, Helm, Johnson, Vlissides

Vorteile von Design Patterns

- Leichtere Einarbeitung in Code anderer Entwickler
- einheitliches Erscheinungsbild des Programmcodes
- Schnellere und effizientere Entwicklung durch fertige Komponenten
- Stabilere Programme durch getestete Komponenten und Designvorlagen

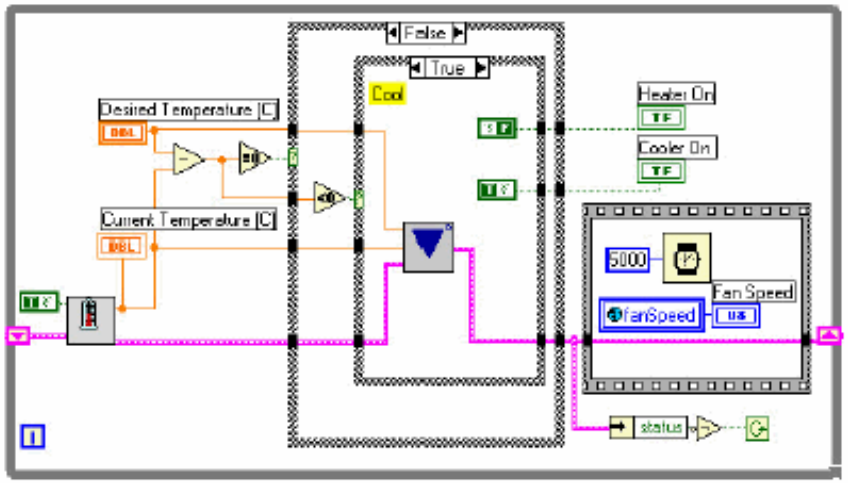
- Ein Design Pattern muss nicht notwendigerweise ein Stück Code sein –
  Es kann auch eine prinzipielle Vorgehensweise sein.
  Zum Beispiel den Fehlerein- und Ausgang unten am VI anzuschließen.

Wer im Publikum nutzt bereits Design Patterns?

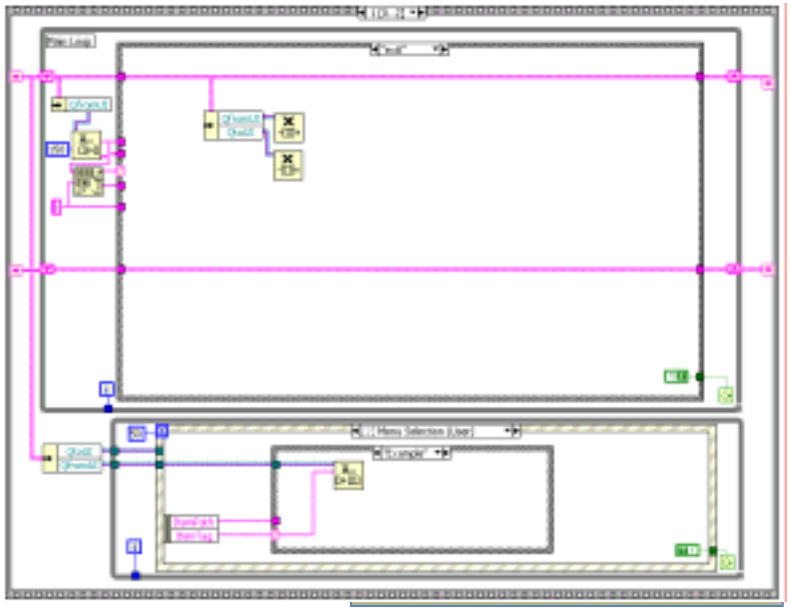
- Erzeuger-Verbraucher Konzept mit Queues
- Zustandsautomat
- Queued Message Handler
- User Event Loop
- Programm-Template

Programmarchitekturen und Designpatterns

- Monolithische Programme
  - GUI und Funktionalität sind „vermischt“
  - gemeinsame Daten z.B. globale Variablen oder Shift-Register
  - gut für kleine Programme, Beispiele etc.

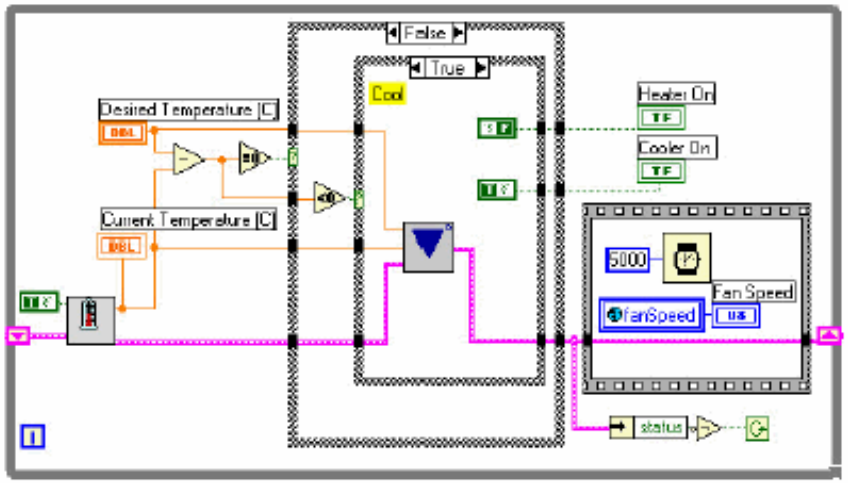


- Modulare Programme
  - GUI und Programmkern sind getrennt
  - Programmkern kann verteilt sein
  - Kommunikation der Komponenten über definierte Schnittstellen

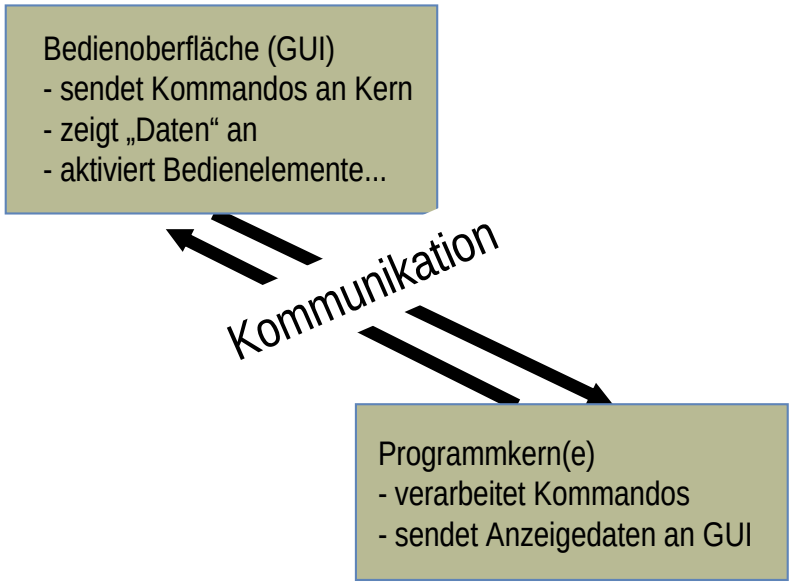


Programmarchitekturen und Designpatterns

- Monolithische Programme
  - GUI und Funktionalität sind „vermischt“
  - gemeinsame Daten z.B. globale Variablen oder Shift-Register
  - gut für kleine Programme, Beispiele etc.



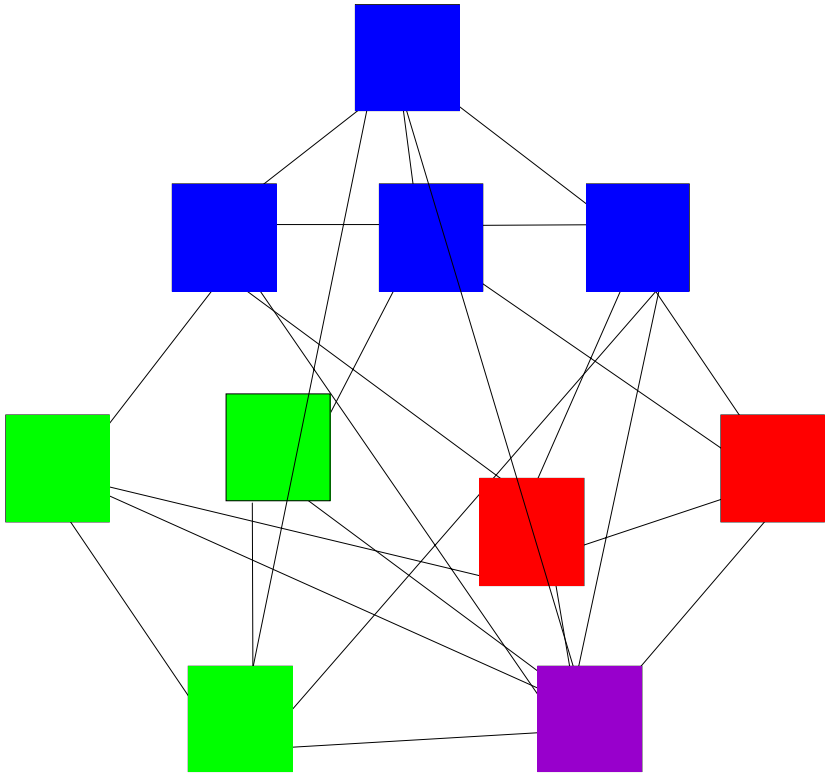
- Modulare Programme
  - GUI und Programmkern sind getrennt
  - Programmkern kann verteilt sein
  - Kommunikation der Komponenten über definierte Schnittstellen



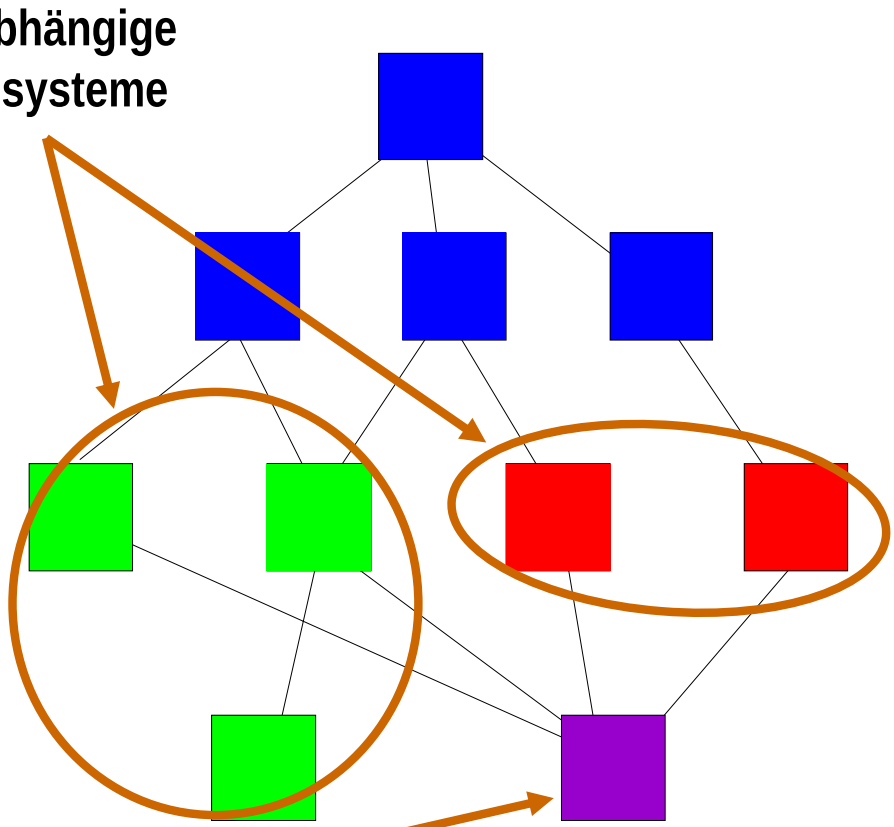
Vorteile modularer Programme

- gute Skalierbarkeit
  - Module sind unabhänging von einander erweiterbar
  - Teilfunktionen können leicht ausgelagert werden (z.B. LabVIEW-RT)
- Wartbarkeit
  - Einzelne Komponenten können isoliert betrachtet werden
  - Implementation unabhängig von der Schnittstelle (Nachrichten)
  - Änderung des GUIs verursacht keine Änderungen an der Funktionalität
- Wiederverwendung möglich
  - Zustandsautomaten für Teilfunktionen

Die VI Hierarchie



Übertriebene Kopplung von Modulen

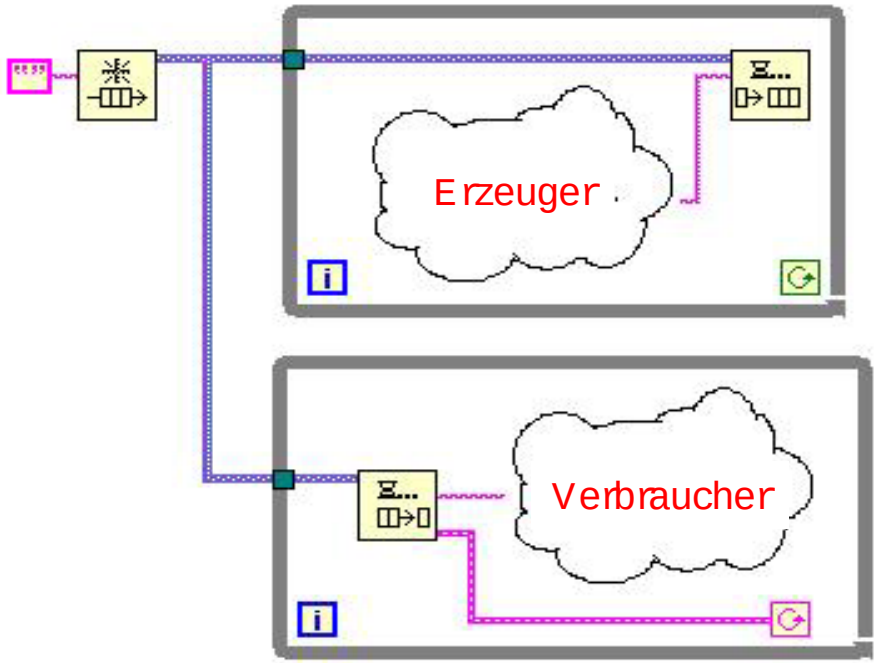


Funktionsbibliotheken

Gute Kopplung (Diamantstruktur)

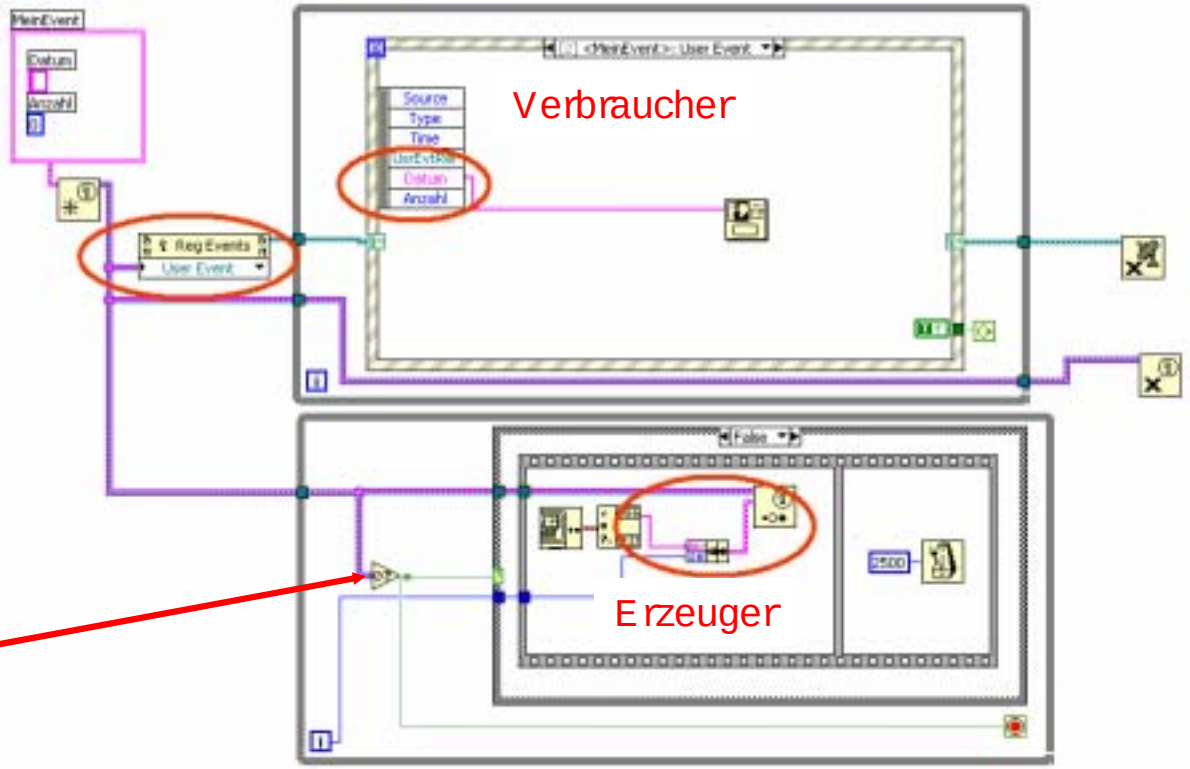
Erzeuger-Verbraucher-Konzept (Daten)

- Kommunikation über Queues oder Notifier



Erzeuger-Verbraucher-Konzept (Events)

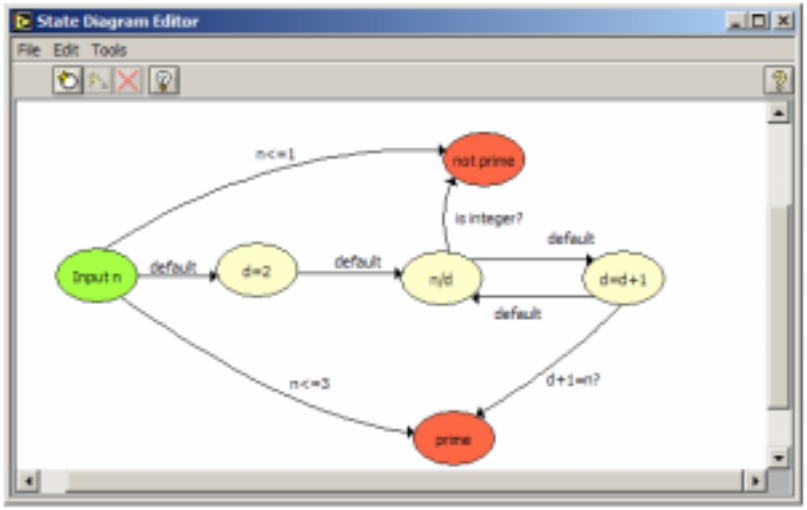
- Kommunikation über Events
- Definition von User-Events
- Neu in LabVIEW 7.1

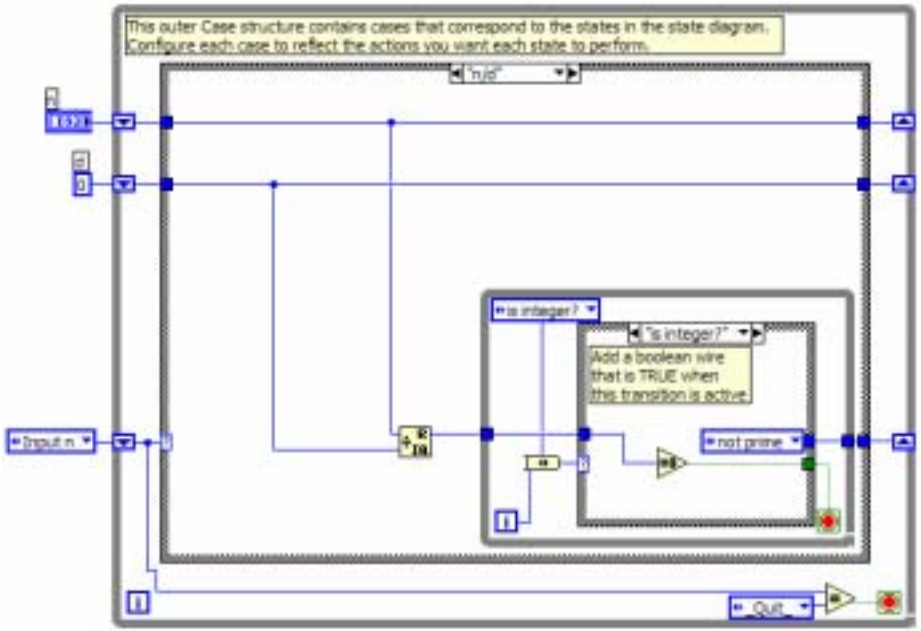


Tipp: zum Beenden beider Schleifen:
Prüfen auf ungültige Queue-Referenz

Pattern 1: Zustandsautomat (State-Machine)

- manuelles Erstellen
- State Diagram Toolkit für LabVIEW für aufwändige Automaten
  - grafisches Entwerfen der Zustände und Übergänge
  - automatisches Erzeugen des LabVIEW Codes

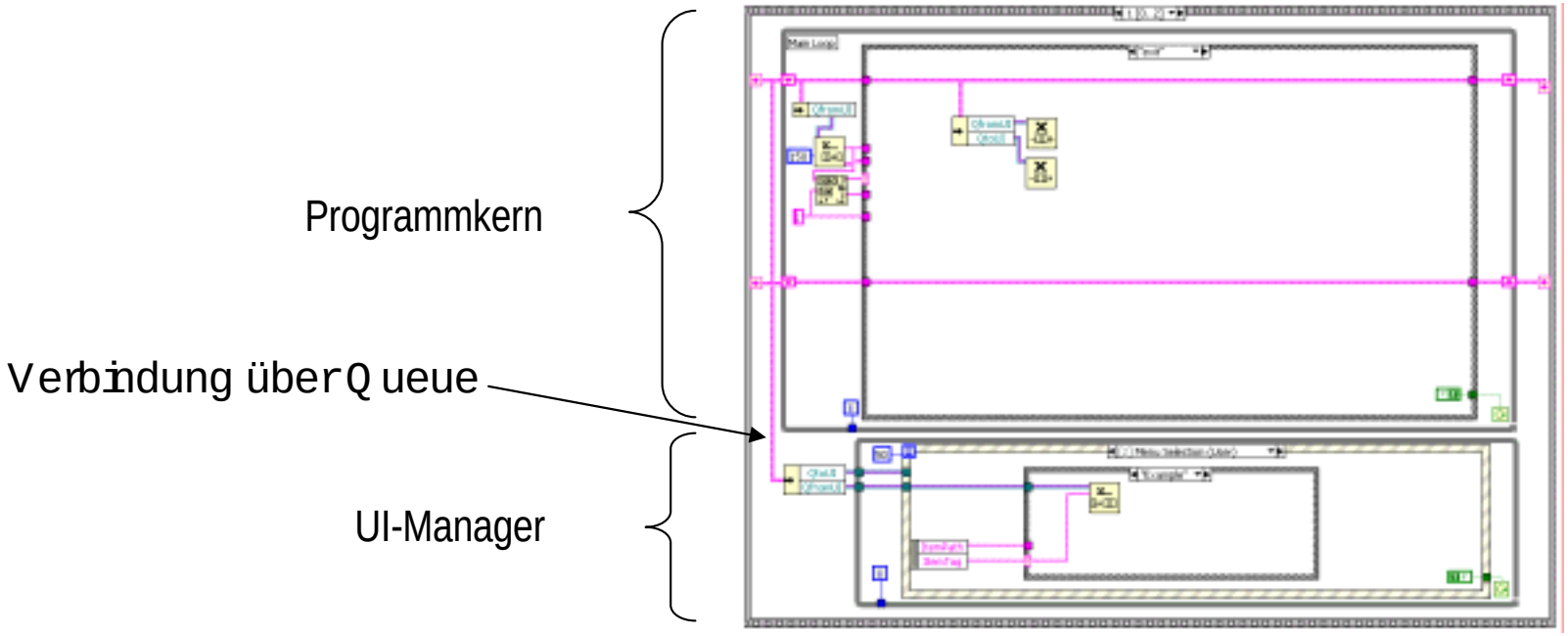




NATIONAL INSTRUMENTS

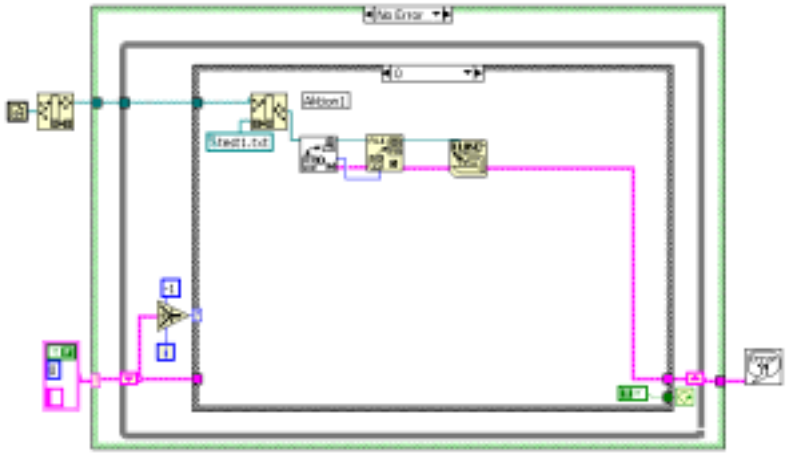
Pattern 2: Programm-Grundgerüst

- Queued Message Handler und UI Event Loop
  - User Event Loop erfasst Benutzer-Events
  - Zustandsautomat nimmt Kommandos entgegen und arbeitet sie ab
  - Austausch definierter Messages (Typedefinitionen)

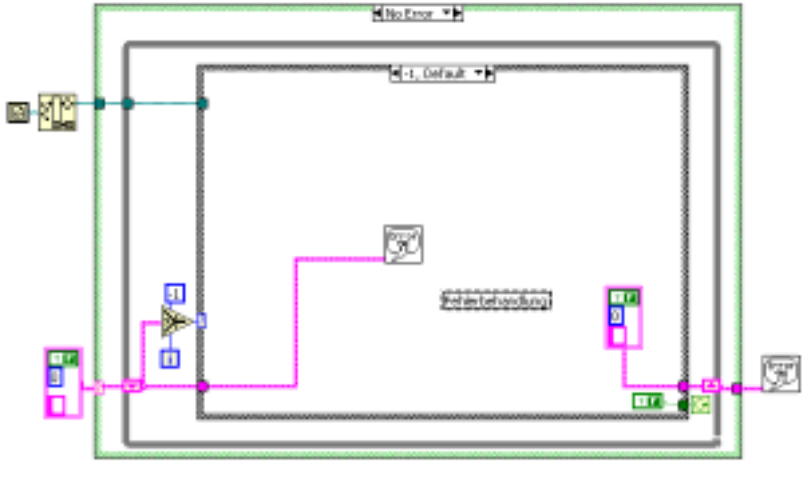


Pattern 3: Sequenz mit Fehlerbehandlung

- LabVIEW-Sequenz kann nicht vorzeitig (z.B. mit BREAK) beendet werden
- While-Loop und Case-Struktur als Ersatz
- Abbruch durch Fehler etc. möglich



hier könnte ein Fehler auftreten



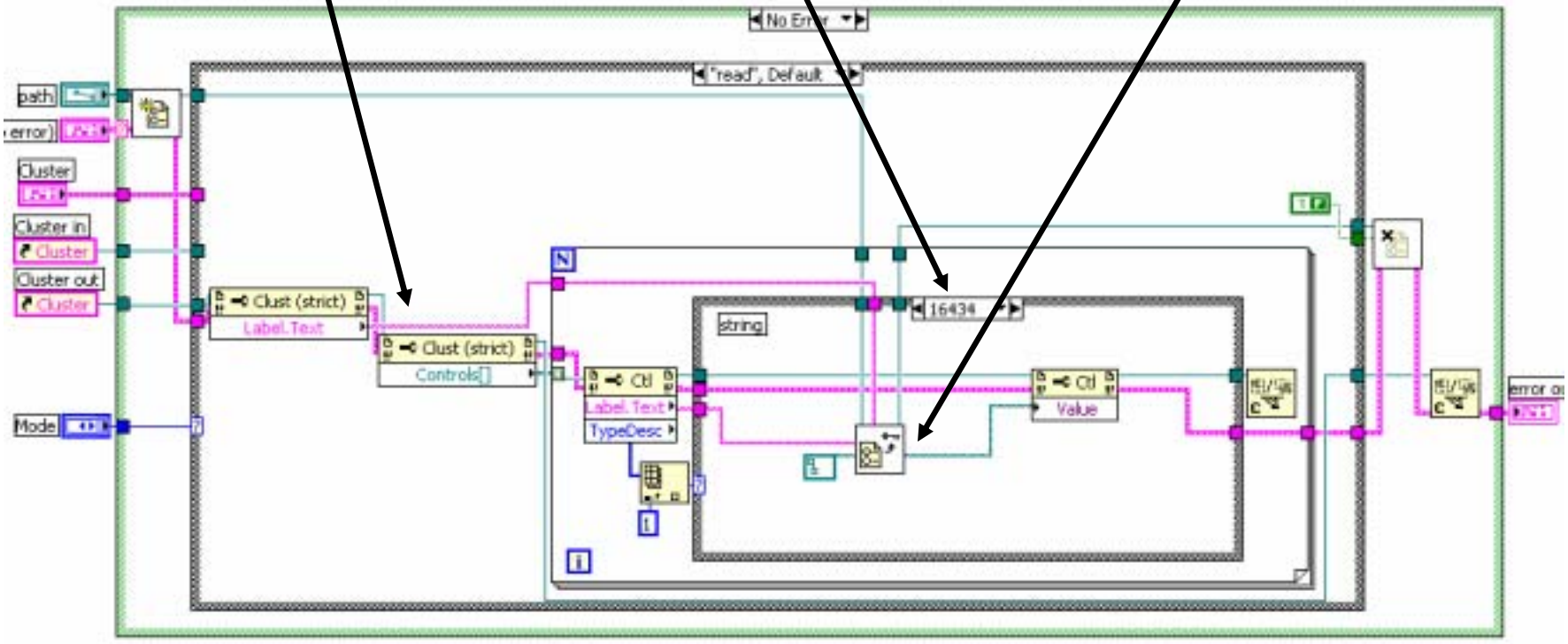
Case für die Fehlerbehandlung

Pattern 4: Konfigurationsdaten-Verwaltung

Zugriff auf die Cluster-Inhalte
Über Referenz (universell!)

Unterscheidung
der Datentypen

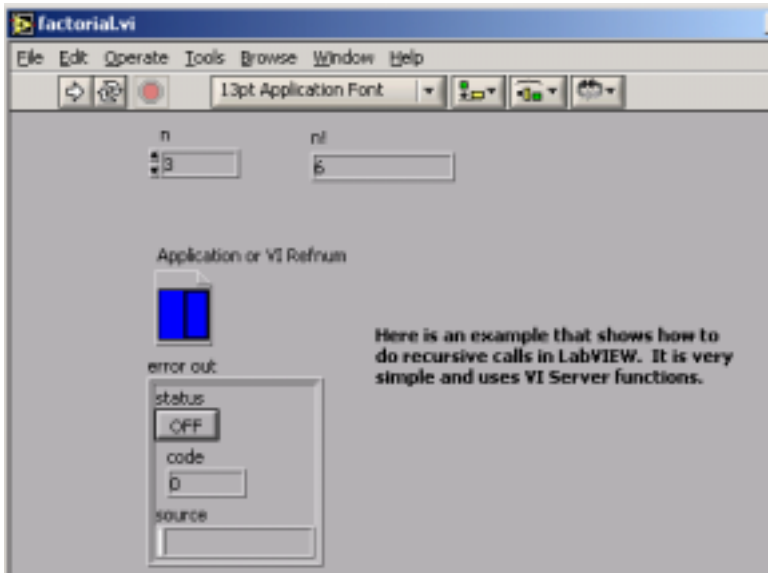
Lesen aus *.ini-Datei

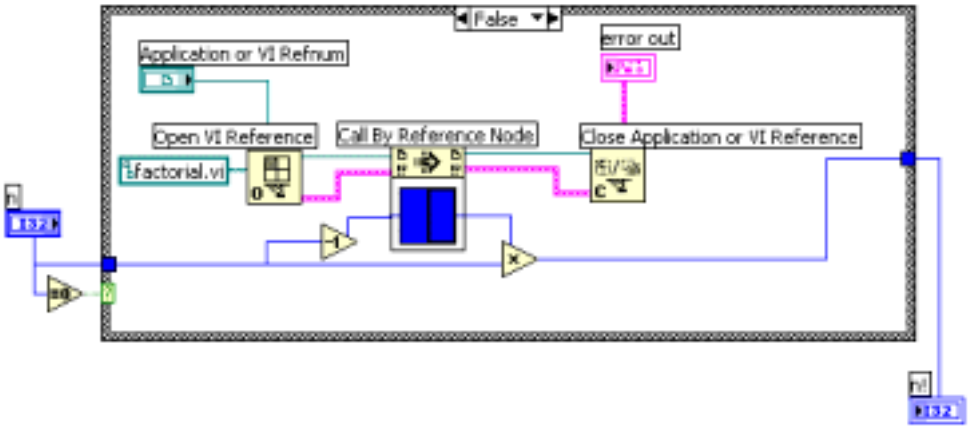


NATIONAL INSTRUMENTS

Pattern 5a: Rekursion mit VI-Aufrufen

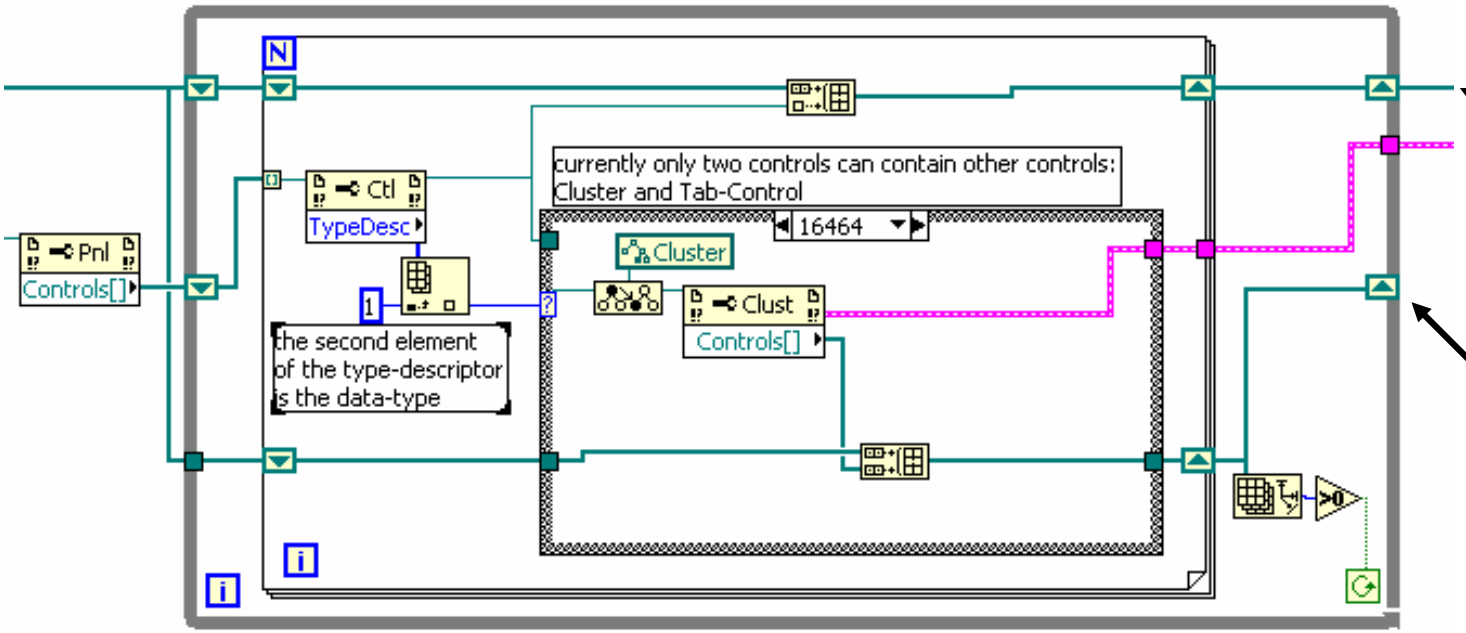
- Berechnen der Fakultät einer Zahl
- Nutzt VI-Server
- kein Debugging da reentrant-execution





Pattern 5b: Rekursion mit Schleifen

- Suchen in Dateibäumen
- Zugriff auf Frontpanelelemente über Referenzen
- mathematische Algorithmen
- alle rekursiven Algorithemen lassen sich mit Schleifen realisieren



Referenzen auf alle Controls des Frontpanels

„Zwischenspeicher“

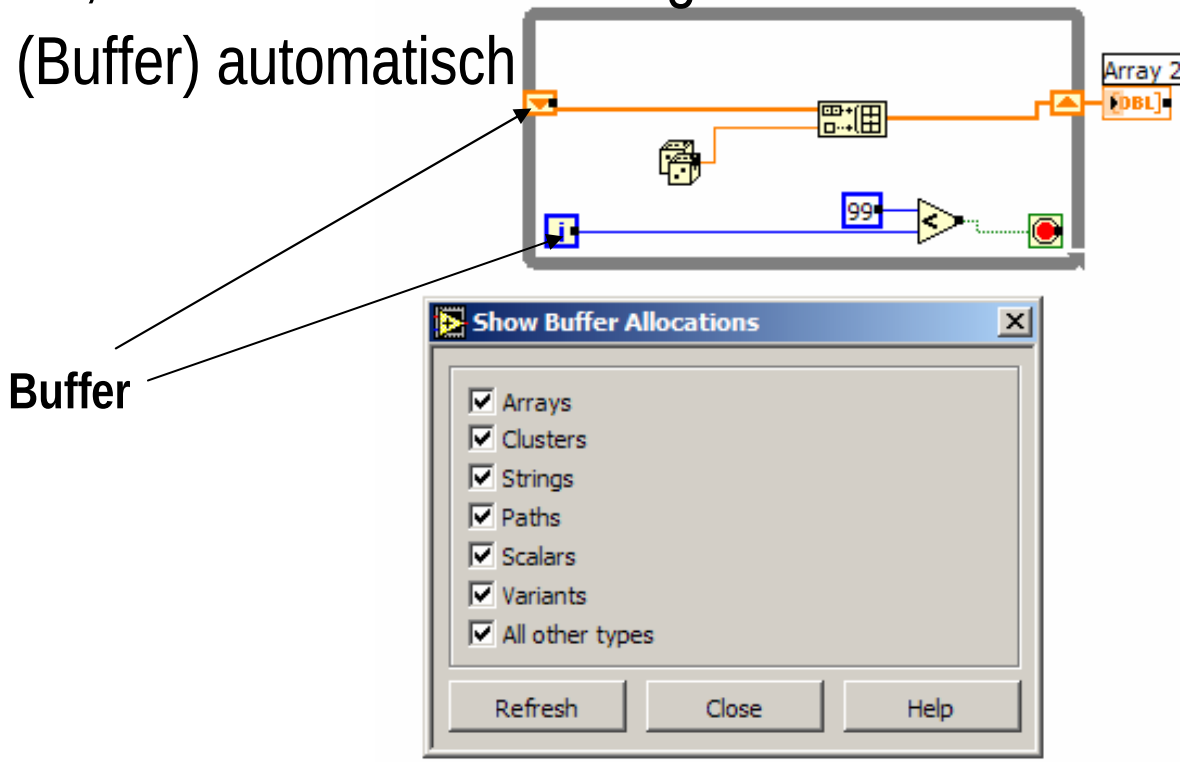
LabVIEW Performance Issues

Speicherverwaltung

- LabVIEW bietet effizientes, aber nicht unfehlbares Speichermanagement
- LabVIEW benutzt Speicher für
  - Frontpanel
  - Blockdiagram
  - Code (kompiliertes Diagramm)
  - Daten (Bedienelemente, Konstanten, Arrays, etc.)
- LabVIEW nutzt Speicher in Form von „Buffer“
  - an Eingangen und Ausgängen von Blöcken und Strukturen
  - Überprüfen mit:
    - Tools->Advanced->Show Buffer Allocations
    - Fil->VI Properties->Memory Usage

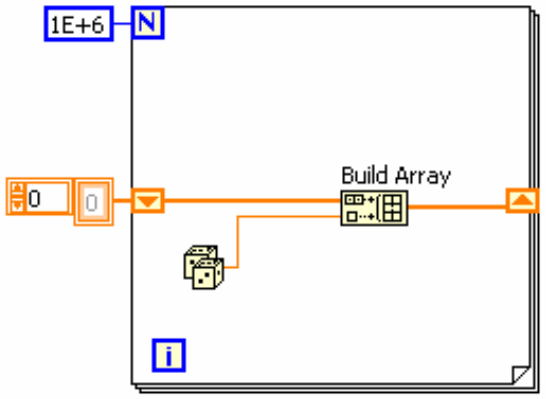
Speicheranforderung (Buffer Allocation)

- LabVIEW bekommt Speicher vom OS zugeteilt
- Benötigt ein VI mehr Speicher muss dieser vom OS neu zugeteilt werden
- Speicheranforderung kostet Zeit, besonders beim Auslagern auf HDD
- LabVIEW verwaltet Speicher (Buffer) automatisch
- LabVIEW versucht Buffer zu recylen
- Buffer hat...
  - jeder Ausgang
  - Coercion Dot
  - User Interface

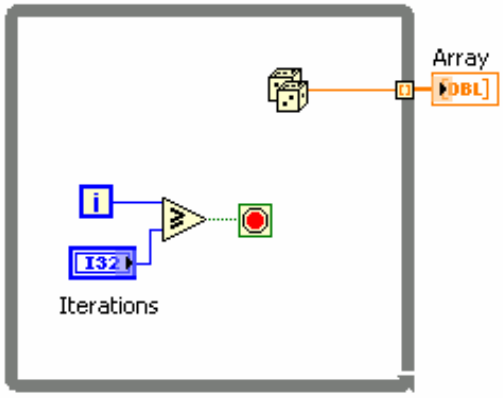


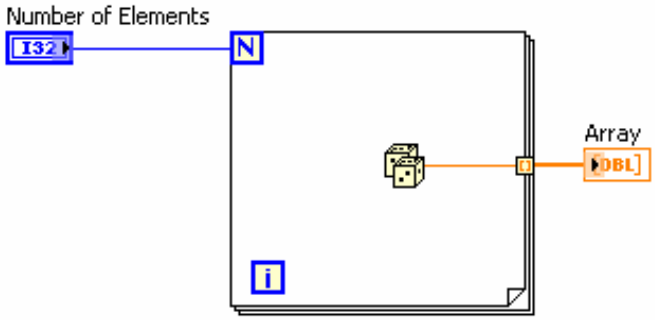
Buffer Recycling

- LabVIEW versucht Speicher weiterzuverwenden
- Einige Funktionen fordern oft Speicher neu an
  - Build Array
  - Concatenate Strings



Häufiges Anfordern von Speicher

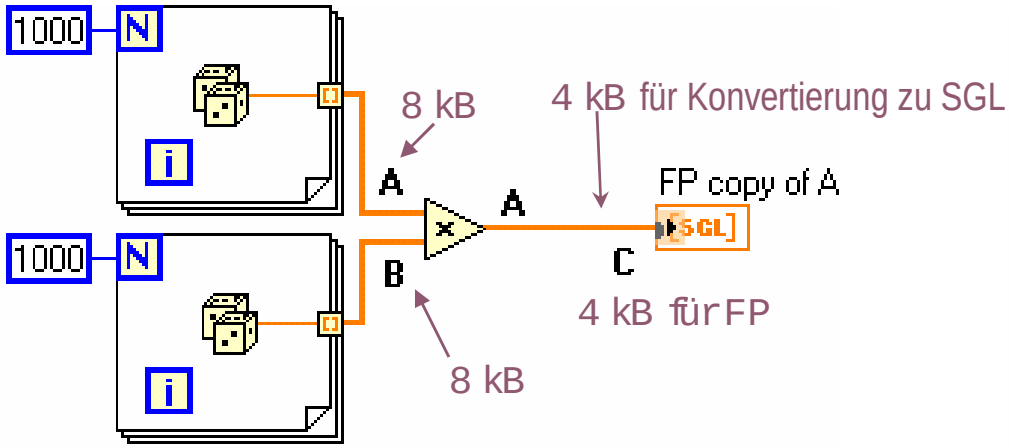




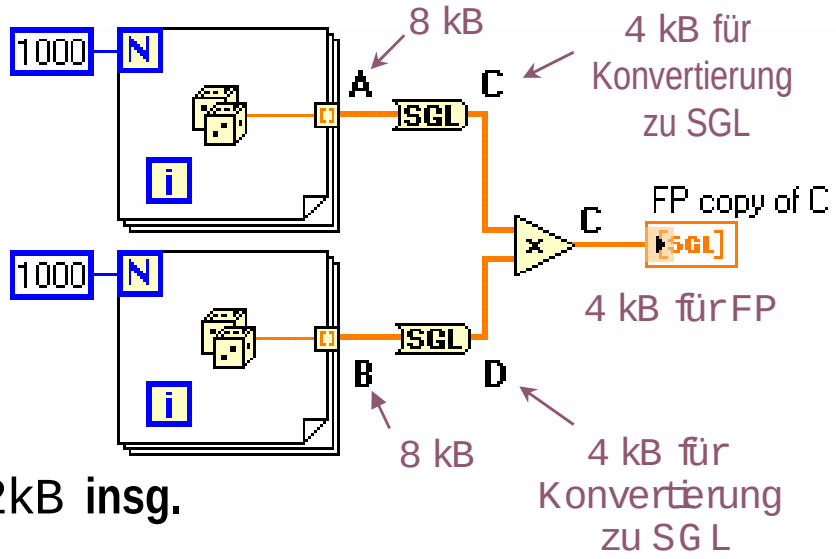
Seltenes Anfordern von Speicher

Vergleich von Array-Operationen

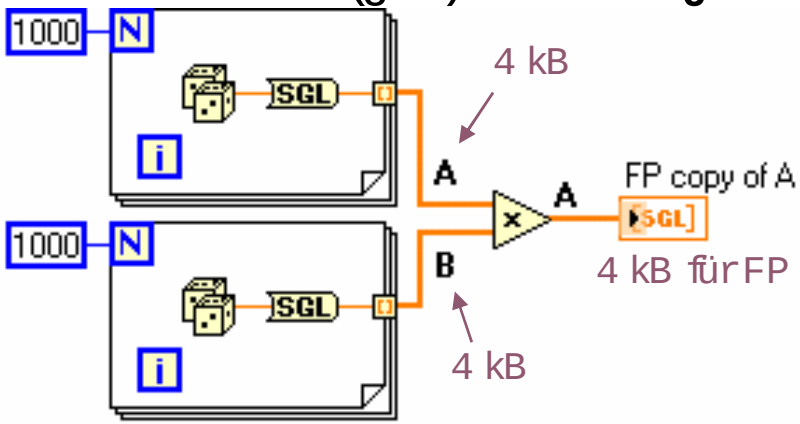
Methode 1 (schlecht): 24 kB insg.



Methode 2 (schlecht): 28kB insg.



Methode 3 (gut): 12kB insg.

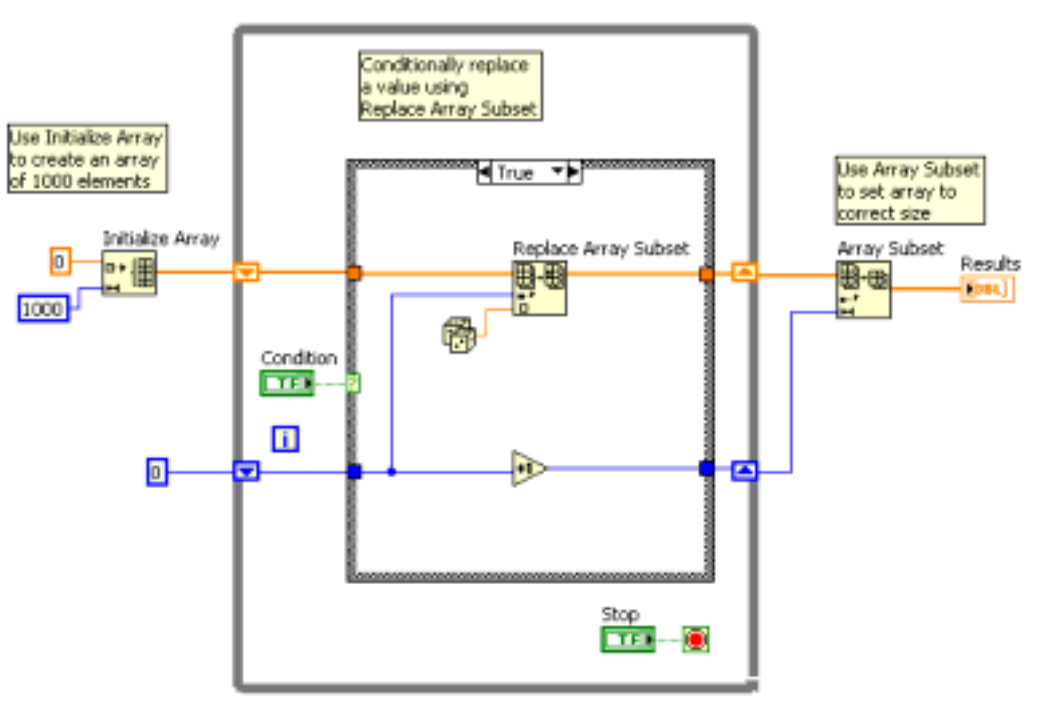


NATIONAL INSTRUMENTS

Tipp: Performance-Steigerung bei Arrays

Arrays Vor-Anfordern

- selten Speicher anfordern
- mehr auf einmal anfordern
- gut möglich, wenn sich die Obergrenze abschätzen lasst



Tipp: Globale and Lokale Variablen

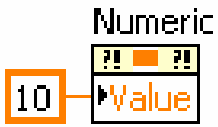
- Jeder Lese-Zugriff auf Variablen erzeugt einen neuen Buffer
- besonders zu beachten bei Strings, Arrays und komplexen Strukturen
- Datenfluss ist immer besser als Variablen was Ausführungszeit und Speicherverbrauch betrifft

Performancesteigerung auf dem GUI

- Zahl an Bedienelementen klein halten
- Bei Grafen und Charts das Autoscaling sparsam einsetzen
- Nur „neue“ Daten auf Grafen und Charts schreiben
- keine Elemente übereinander legen
- Auf Charts mehrere Punkte auf einmal schreiben
- Asynchronous Display als Standardeinstellung beibehalten

Property Nodes und Control References

- Werden in UI Thread abgearbeitet
- Bei Zugriff wird vom Thread in dem das VI läuft in den UI Thread und danach zurück gewechselt
- nicht zum Ändern des Wertes nutzen!
- Wenn schon auf ein Property Nodes oder Control RefNums zugegriffen wird, um Eigenschaften zu ändern, kann auch der Wert geschrieben werden – Threadwechsel passiert sowieso



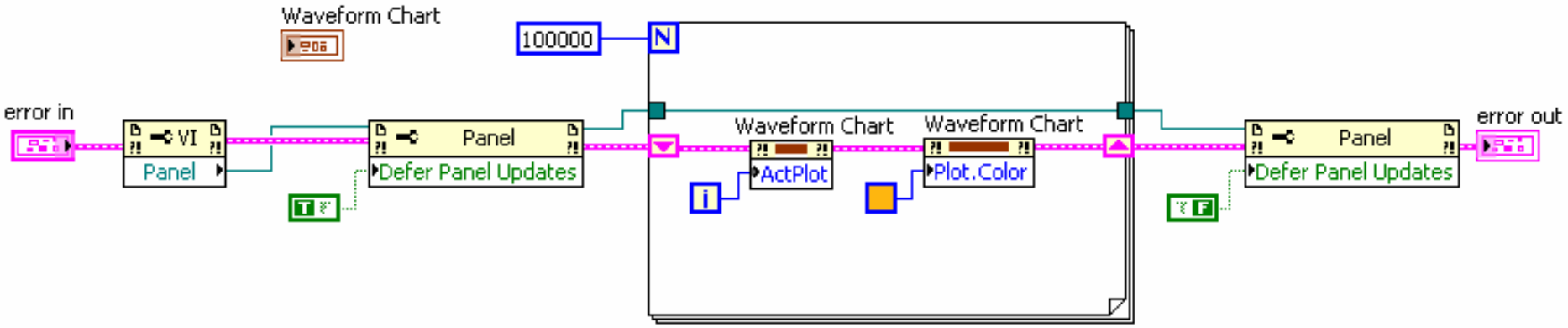
Good

Better

Best

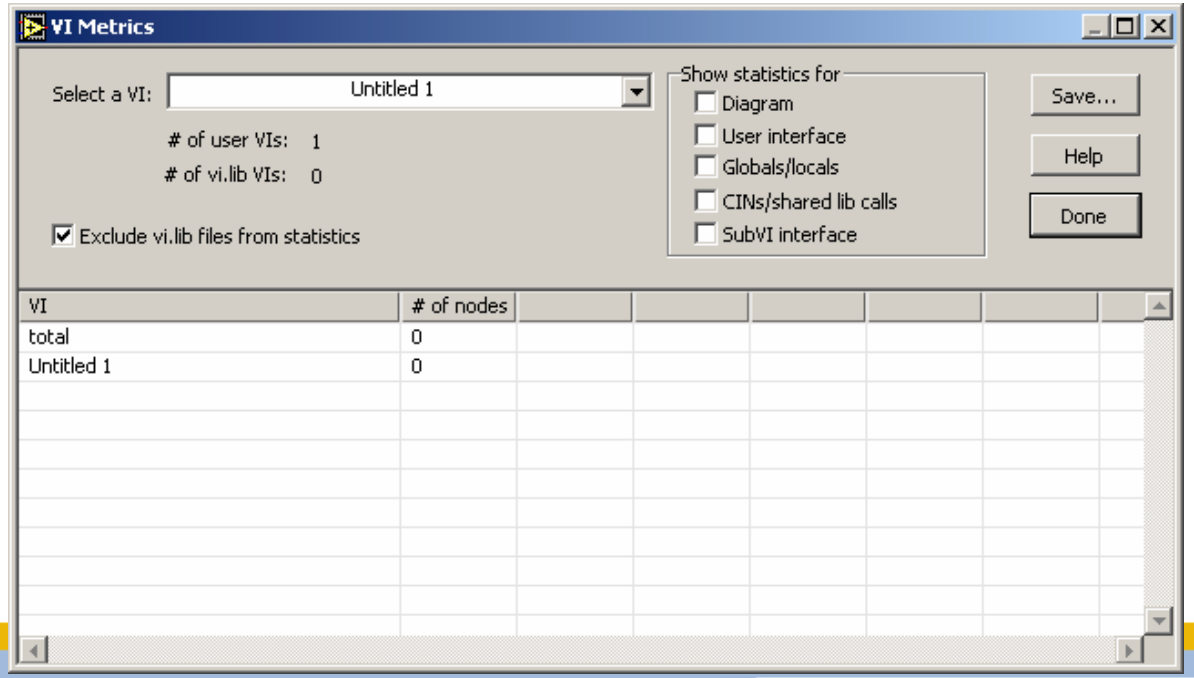
Tipp: Performance-Steigerung auf dem GUI

- Defer Panel Updates benutzen!
  - nur einmaliges Update des GUIs



Tipp: Das VI Metrics-Werkzeug

- Zeigt ungünstige Verwendung von Variablen
- Zeigt implizite Typumwandlungen (coercion dots) an
- Hilft beim Erkennen von zu tiefen Schachtelung und übertriebenen Einsatz von Strukturen
- Lässt zu große Blockdiagramme erkennen



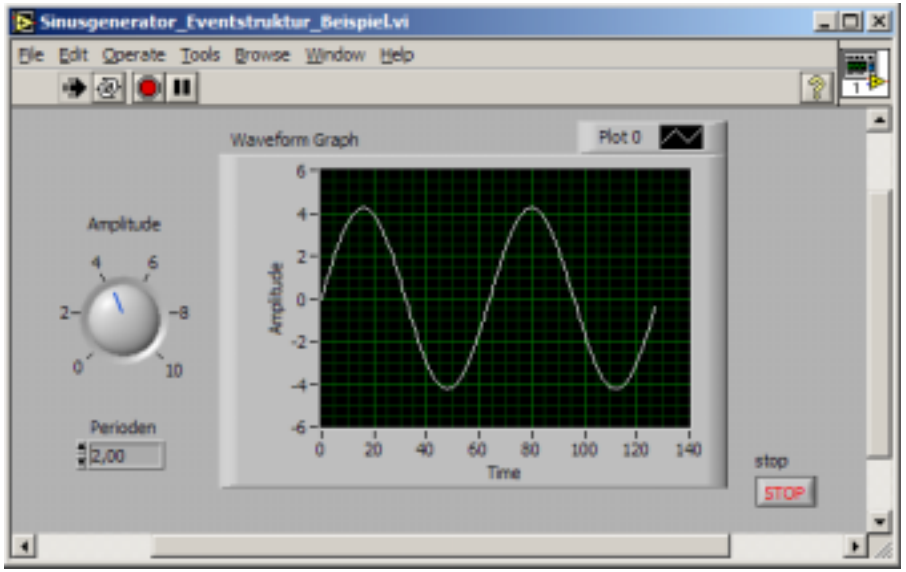
Event-Programmierung in LabVIEW

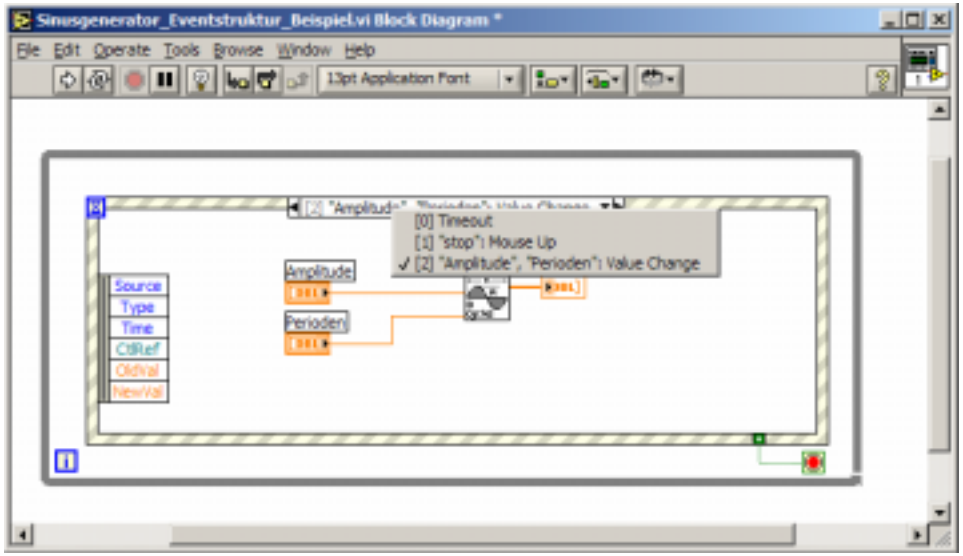
Event-Programmierung

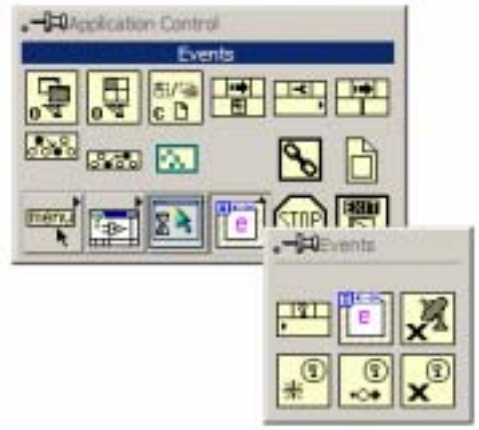
- Warum Event Programmierung?
  - Benutzer sind „Eventgesteuert“ – z.B. Mausklick
  - übersichtlichere Programmstruktur da zu jedem Event ein dedizierter Programmteil gehört
  - verringert die Systemlast durch Vermeiden von aktivem Warten (Polling)
- Welche Eventprogrammierung ist in LabVIEW möglich
  - Event-Struktur
    - Erfassen von GUI Events
    - User-Events definieren und auslösen
  - ActiveX-Events
    - Events von ActiveX Servern
  - Occurence

Event-Struktur

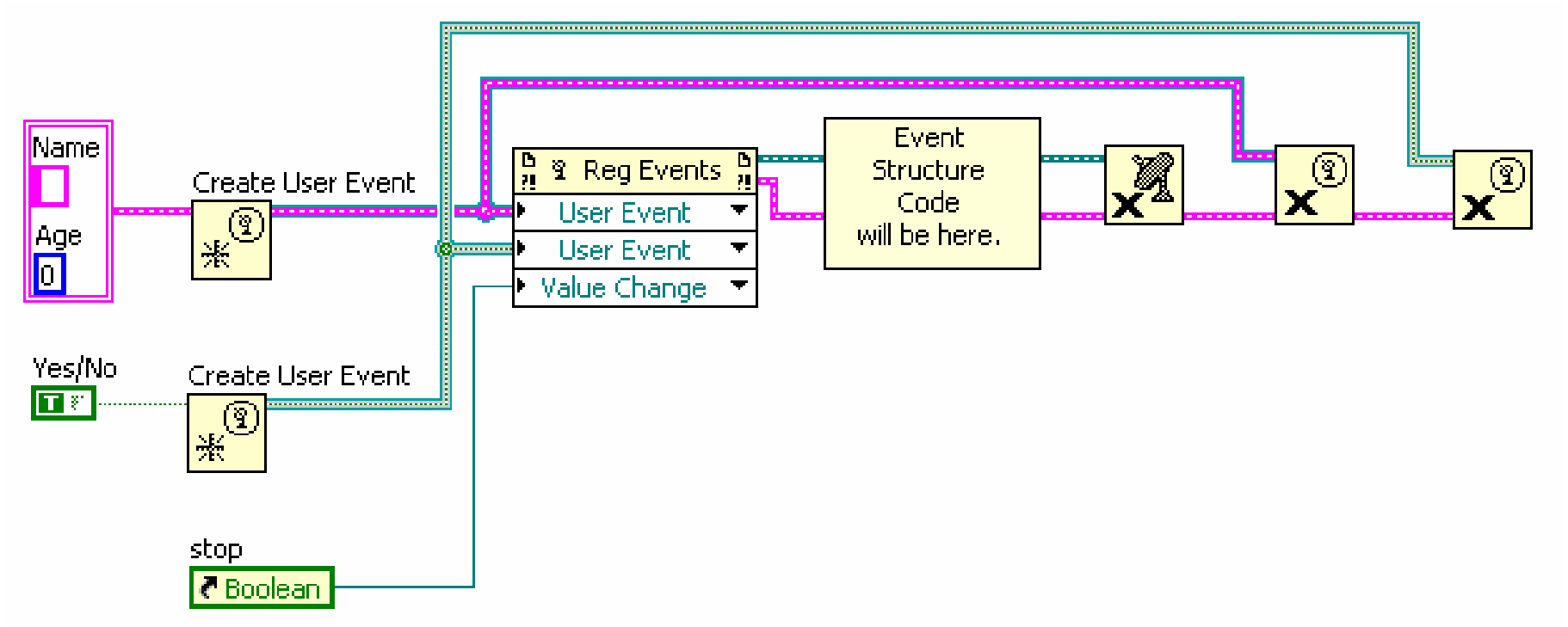
- Erfassen und verarbeiten von GUI Events
- Ein Rahmen pro Event
- Zusammenfassen von Events, z.B. „ValueChanged“







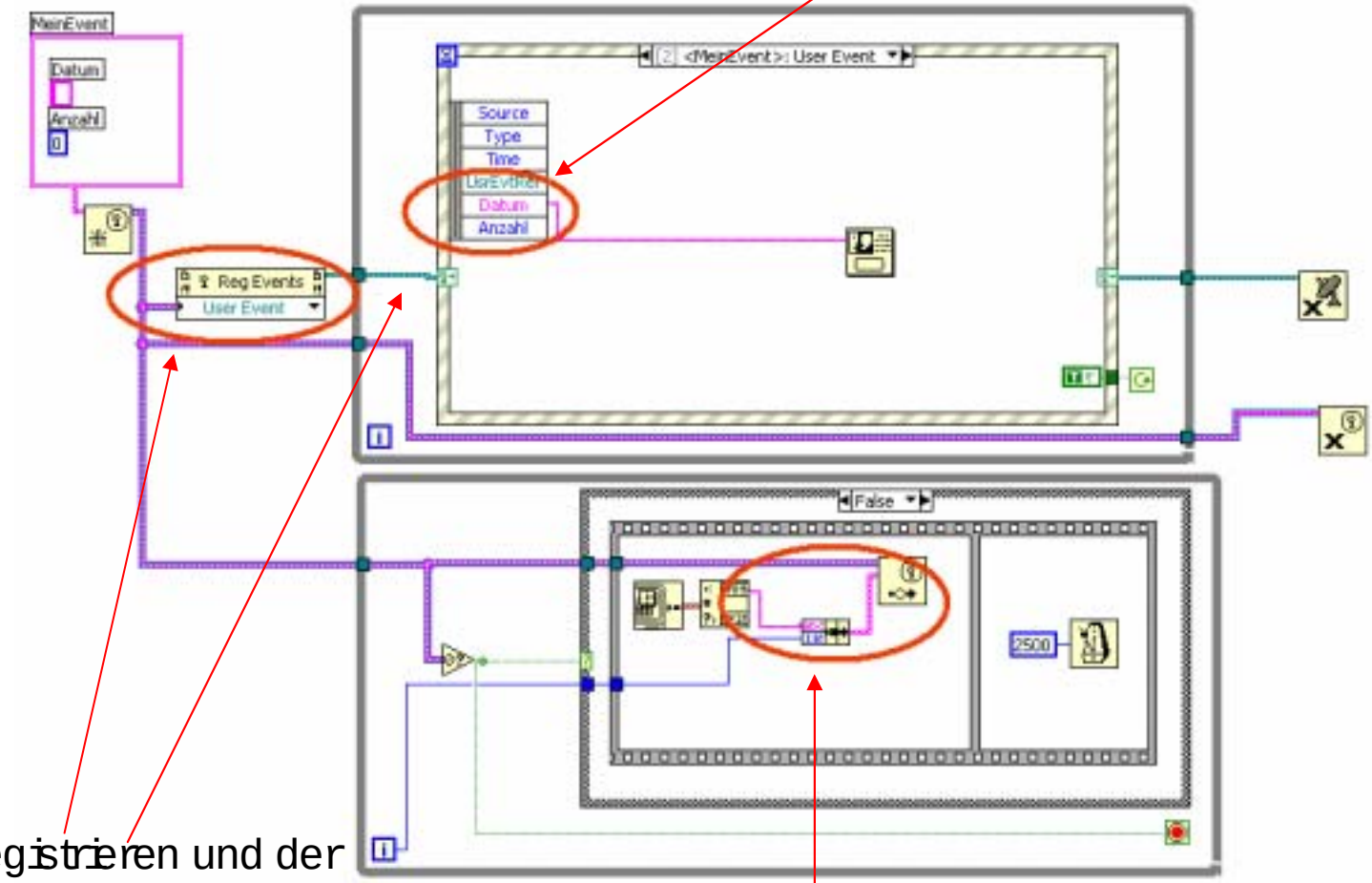
Programmieren mit User Events



Event-Struktur in der Anwendung

Eventdaten definieren

Ausgabe der Eventdaten im Event-Case

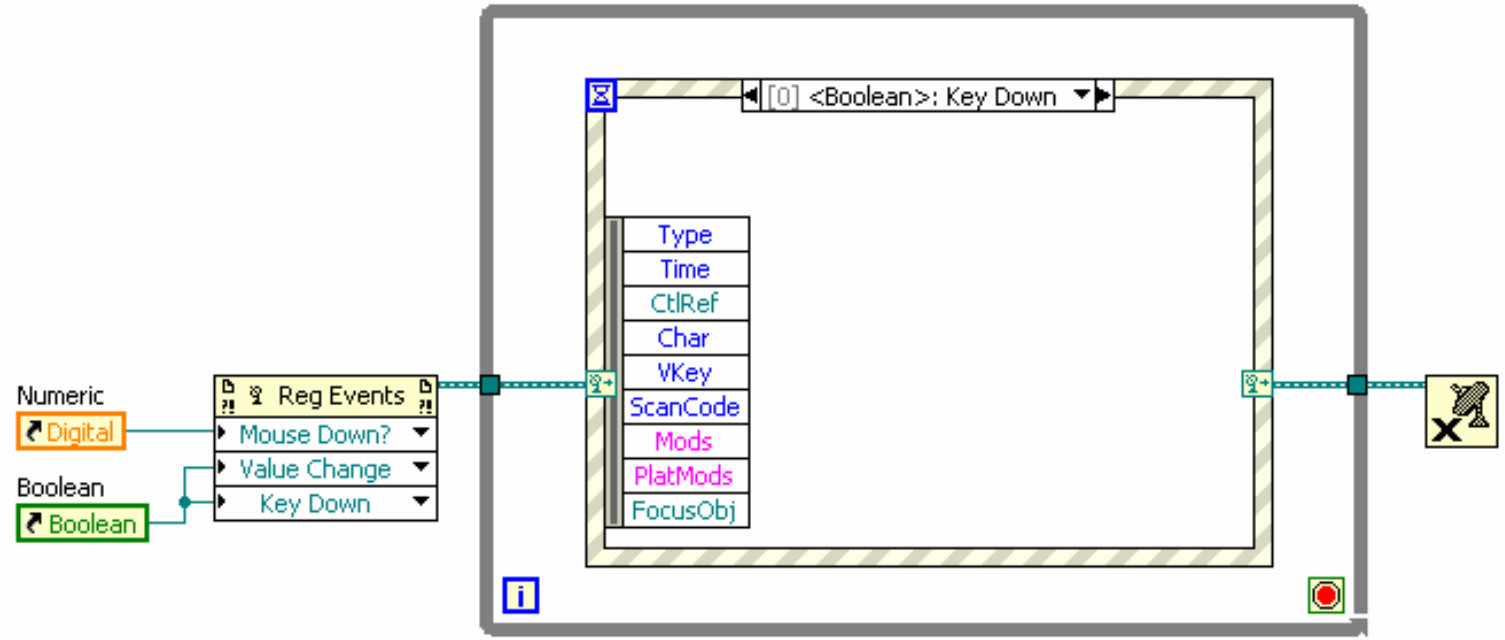


Event registrieren und der Struktur bekanntmachen

Events auslösen

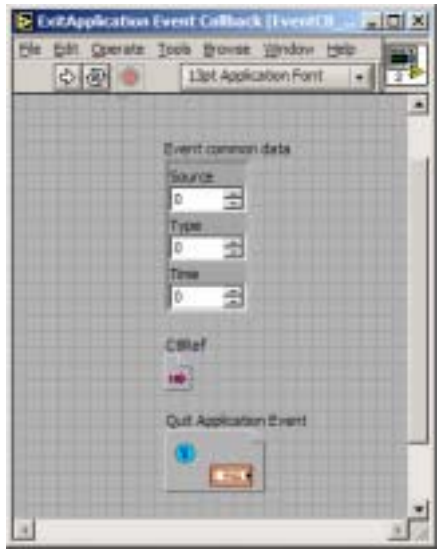
Programmieren mit dynamischen Events

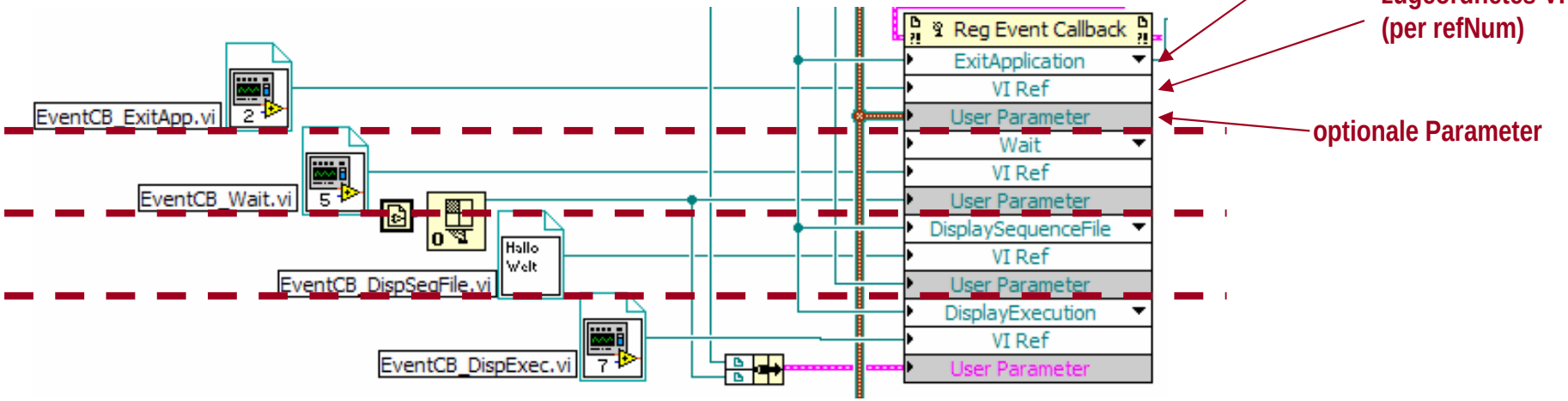
Event registrieren → Event an Struktur anmelden → Warten auf Events & Abarbeiten → Event löschen



ActiveX-Events

- Reagieren auf Ereignisse in ActiveX-Servern
  - Änderung von Werten in Excel
  - Testsequenz beendet in TestStand
- Einem ActiveX-Event wird ein VI zugeordnet
- Optionale Parameter möglich





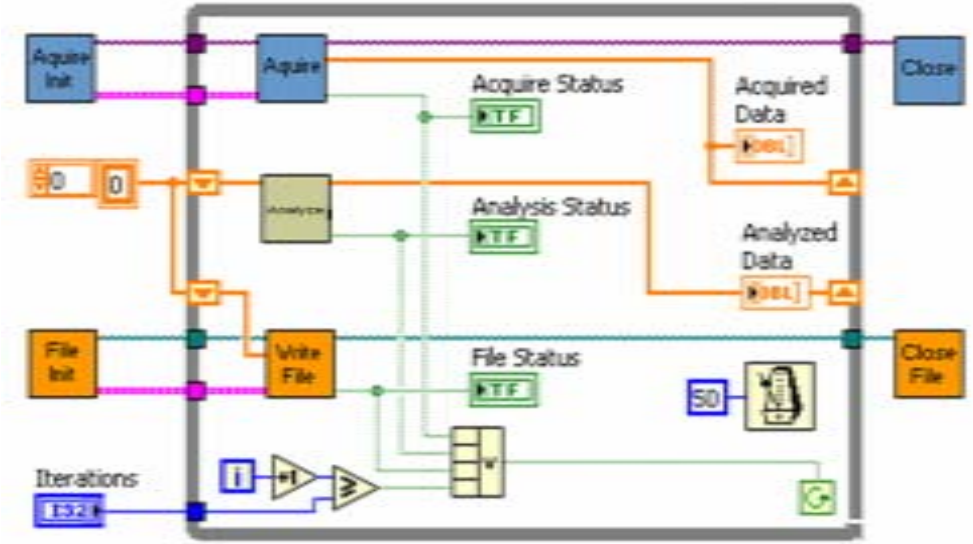
Multithreading in LabVIEW

Multithreading

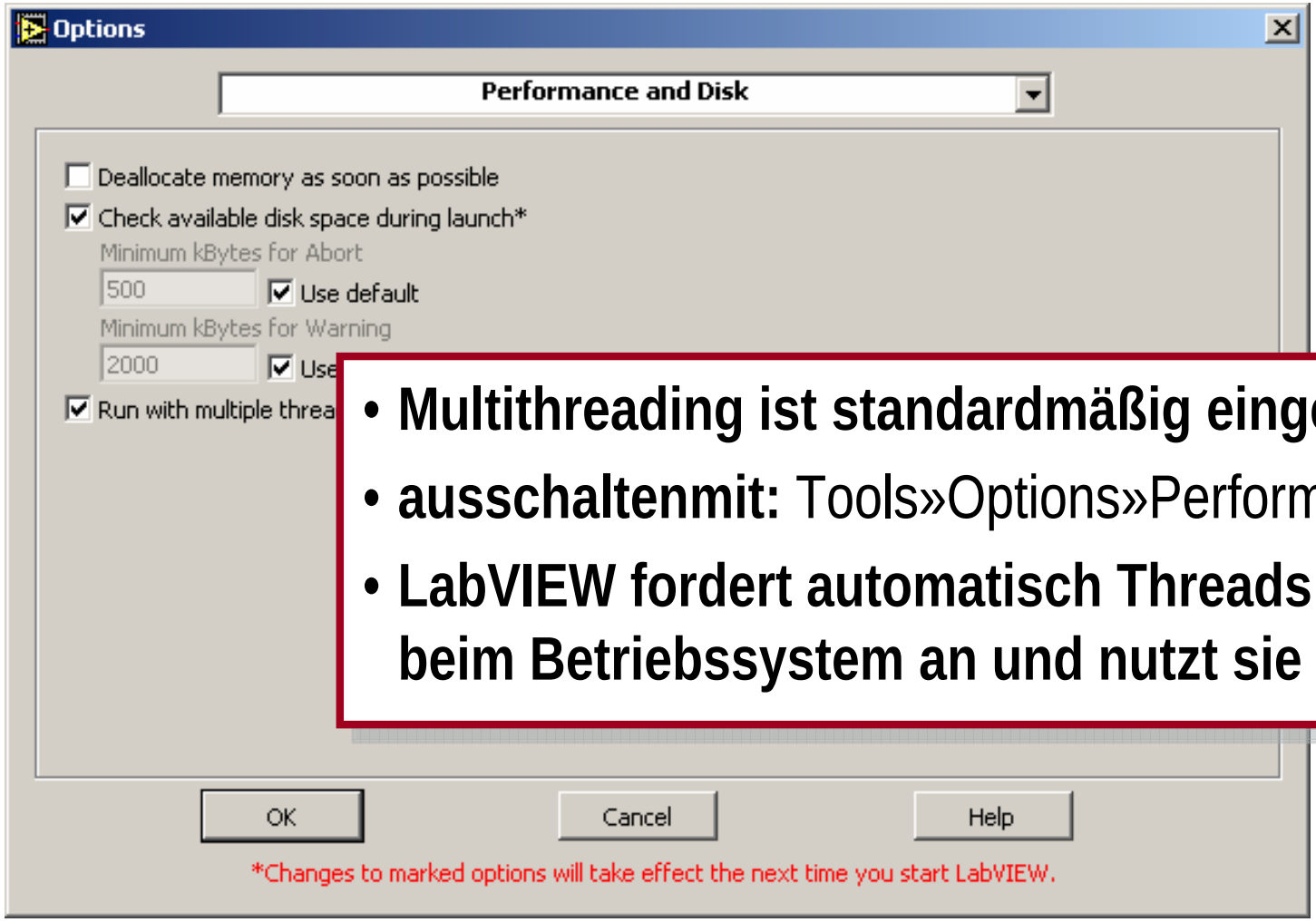
- Besseres nutzen der Prozessorzeit
  - Threads können laufen, wenn andere warten müssen, z.B. auf Hardware
- Zuverlässigere Antwortzeiten
  - durch feste Zeitzuteilung
- Stabilere Systeme
- Benutzerfreundlicher
  - kein „einfrieren“ des Bildschirms mehr bei z.B. Drucken
- Größter nutzen auf Mehrprozessorsystemen
  - rechenintensive Aufgaben können auf einem separaten Prozessor laufen

Multithreading in LabVIEW nutzen

- Ganz einfach...es geht automatisch!
  - es ist kein extra Code erforderlich
- Aber aufgepasst...es geht automatisch!
  - greifen mehere Threads auf z.B. eine globale Variable zu kann es zu Überschneidungen und Race-Conditions kommen.



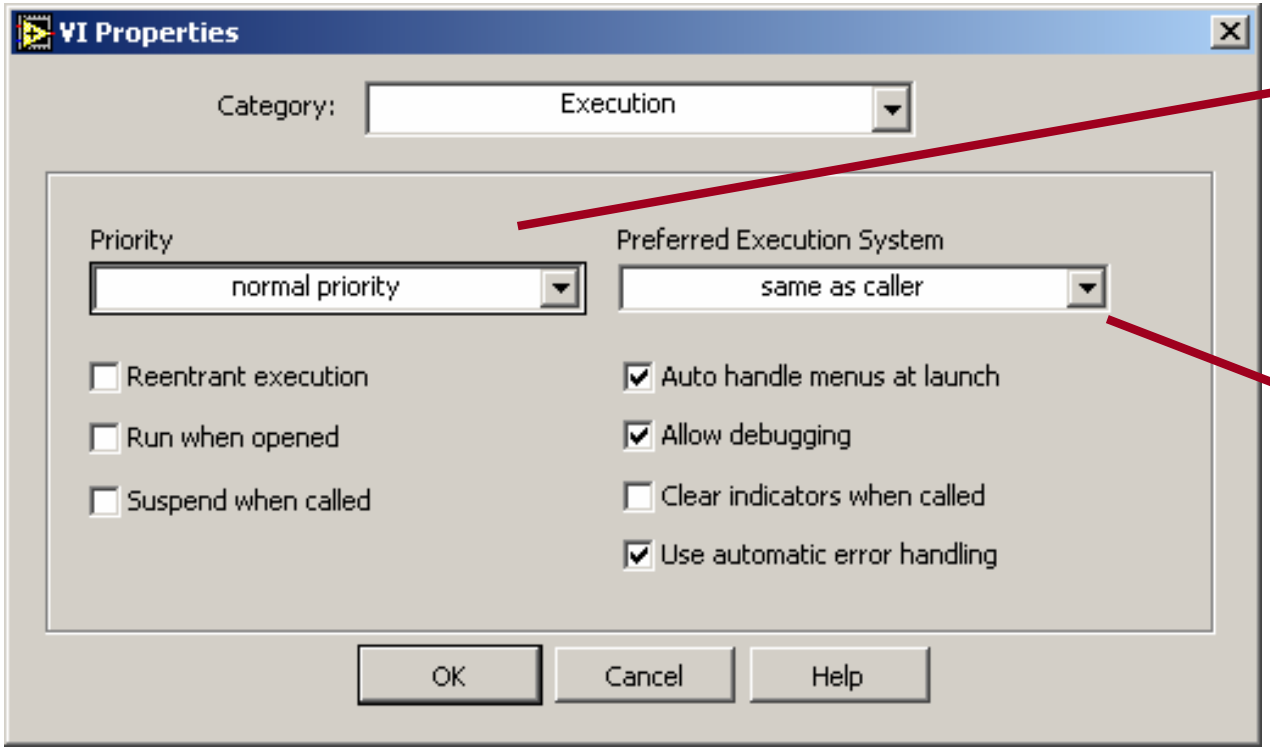
Multithreading in LabVIEW einschalten

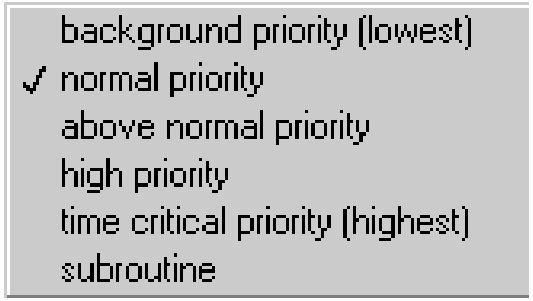


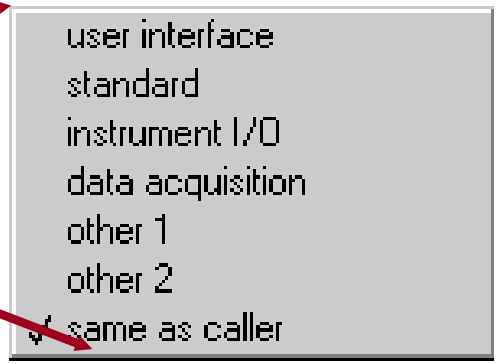
- **Multithreading ist standardmäßig eingeschaltet**
- **ausschaltenmit:** Tools»Options»Performance and Disk
- **LabVIEW fordert automatisch Threads ausreichend beim Betriebssystem an und nutzt sie**

Threadprioritäten für VIs konfigurieren

File->VI Properties->Execution







Execution Systems, Prioritäten und Threads

- Wie nutzt LabVIEW das Multithreading
  - 5 Exectuion Systems
  - 5 verschiedene Prioritäten
  - 2 Threads und eine Abarbeitungsliste pro Priorität
  - Subroutine Priority
    - für effizientes Abarbeiten komplexer Berachnungen etc.
    - teilt keine Ausführungszeit mit anderen VIs
    - kein Zugriff auf asynchrone Funktionen wie Wait, GPIB, VISA, oder Dialog Box,...
  - User Interface
    - benutzt eigenes Execution System (Normal Priority) mit einem Thread

Reentrant Execution

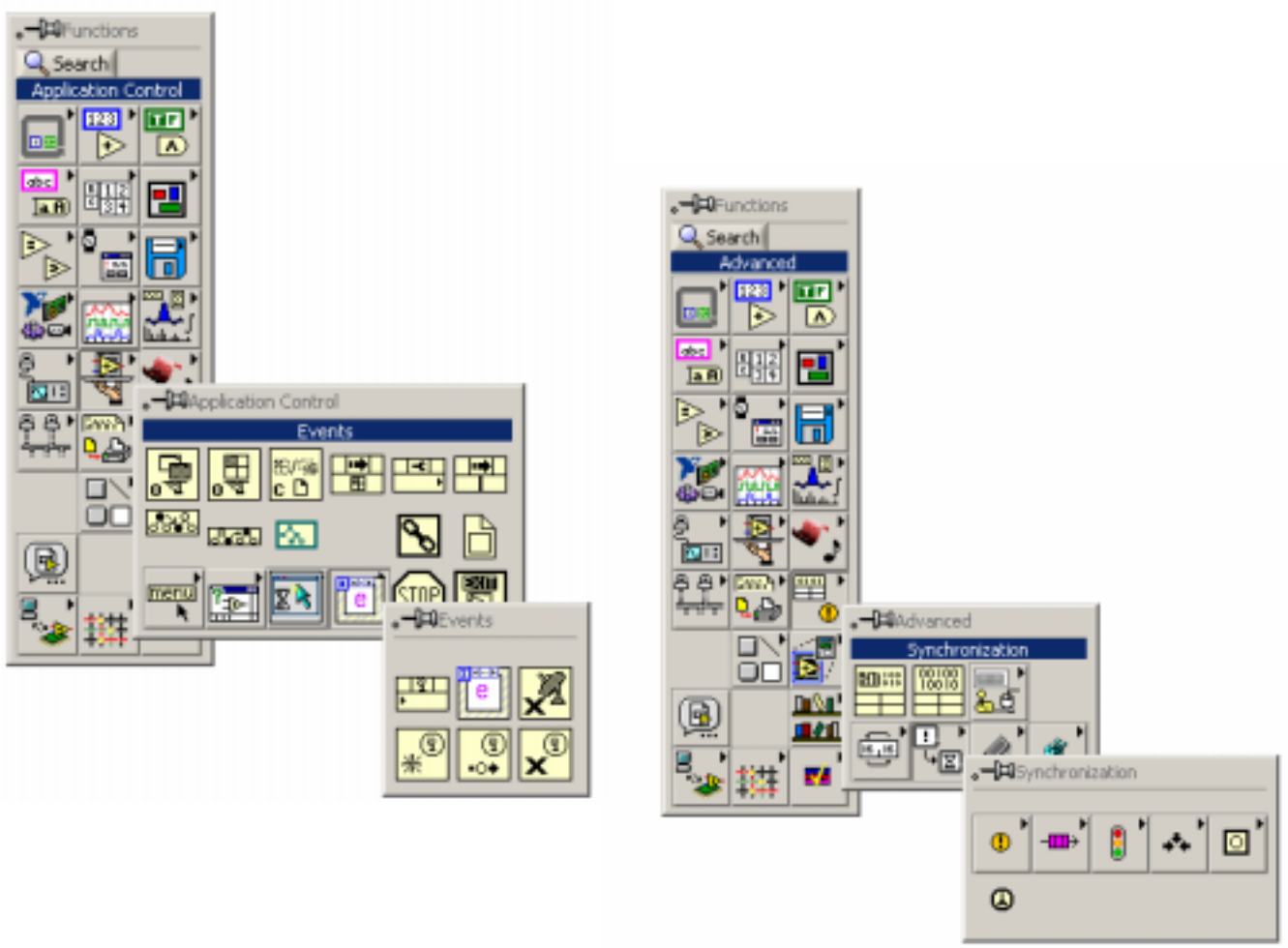
- Ermöglicht das mehrfache gleichzeitige Zugreifen auf ein VI
- Führt dazu, dass ein reentrant VI seinen eigenen Datenbereich und Ausführungsinformationen führt
- Reentarnt Execution erlaubt allerdings nicht:
  - Run when opened
  - Suspend when called
  - Auto Handling of Menus at Launch
  - Allow Debugging

Kommunikation zwischen Threads

- Wozu?
  - Synchronisierung
  - Datenaustausch
  - Wartende Threads aktivieren
  - Verhindern von gleichzeitigem Zugriff auf Variablen, Hardware, Dateien etc. (Race Conditions)
  - Verhindern von gegenseitigem Ausschluss (Deadlocks)

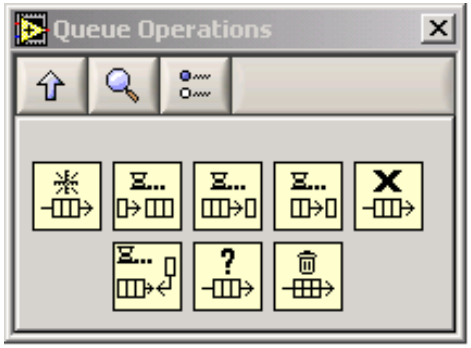
Thread-Kommunikationsmechanismen

- Queues
- Notifier
- Semaphoren
- Rendezvous
- Events

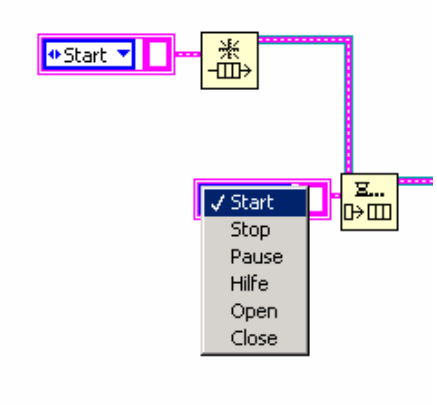


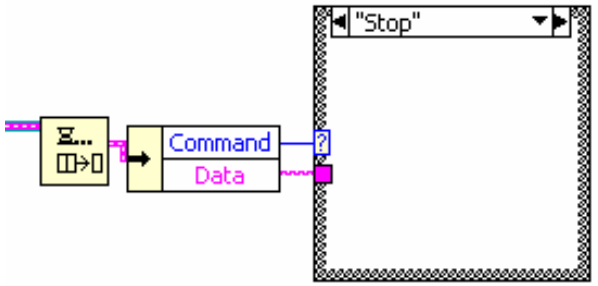
Queues

- FIFO-Prinzip
- einer oder mehere Erzeuger und ein Verbraucher
- alle Datentypen, auch Cluster, Typedefs, etc.



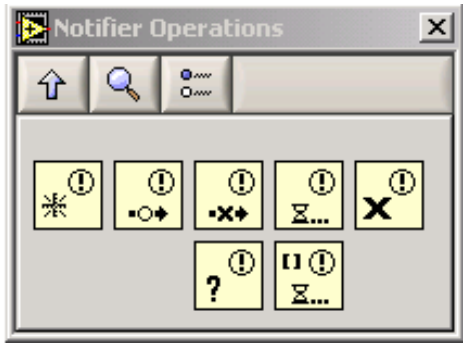
Idee für Zustandsautomaten und Kommandos mit Enums





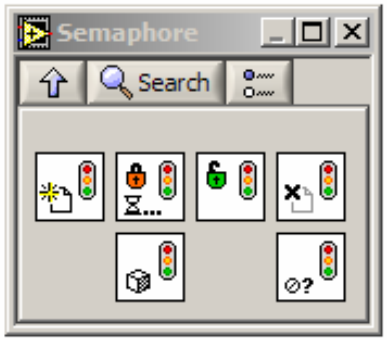
Notifier

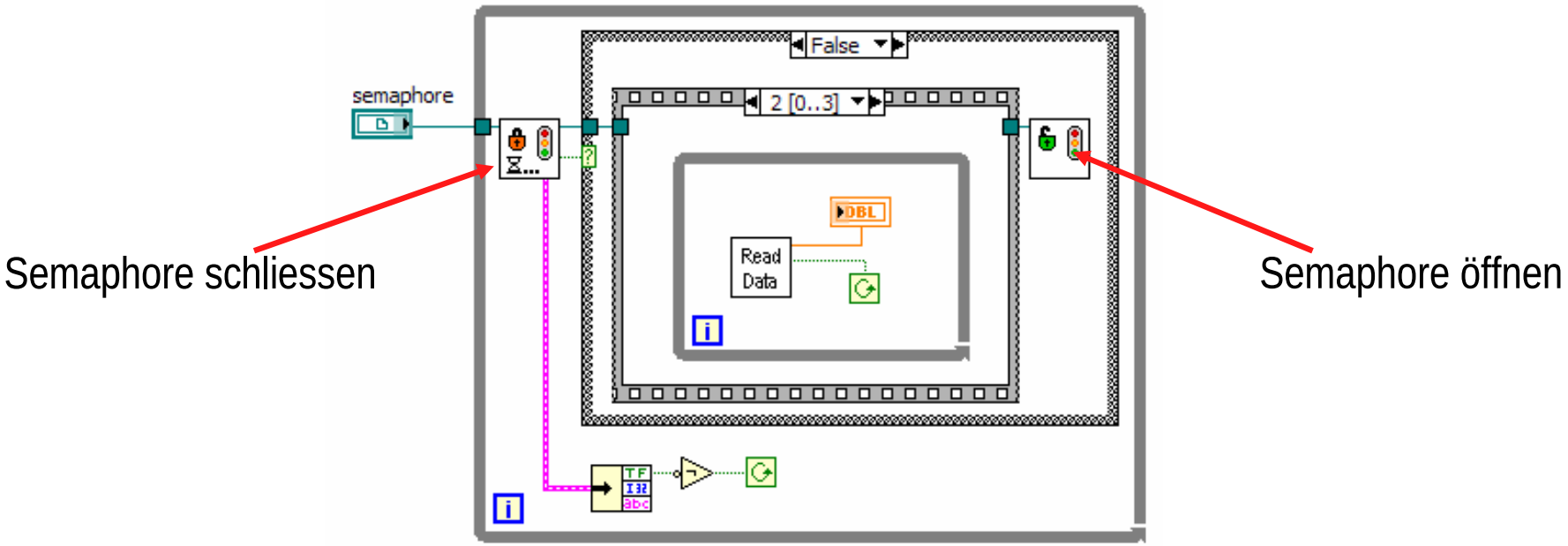
- nicht gepuffert
- von einem Sender an einen oder mehrere Empfänger
- von mehreren Sendern an einen oder mehrere Empfänger
- alle Datentypen, auch Cluster



Semaphoren

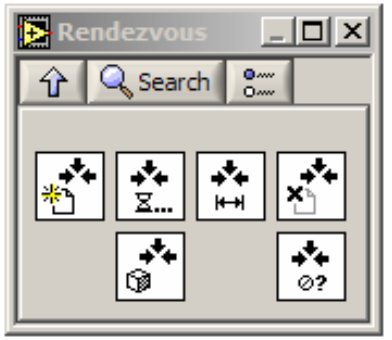
- Wichtig in Multithreading-Applikationen
- Mit Semaphoren kann verhindert werden, dass bestimmte Programmteile gleichzeitig laufen

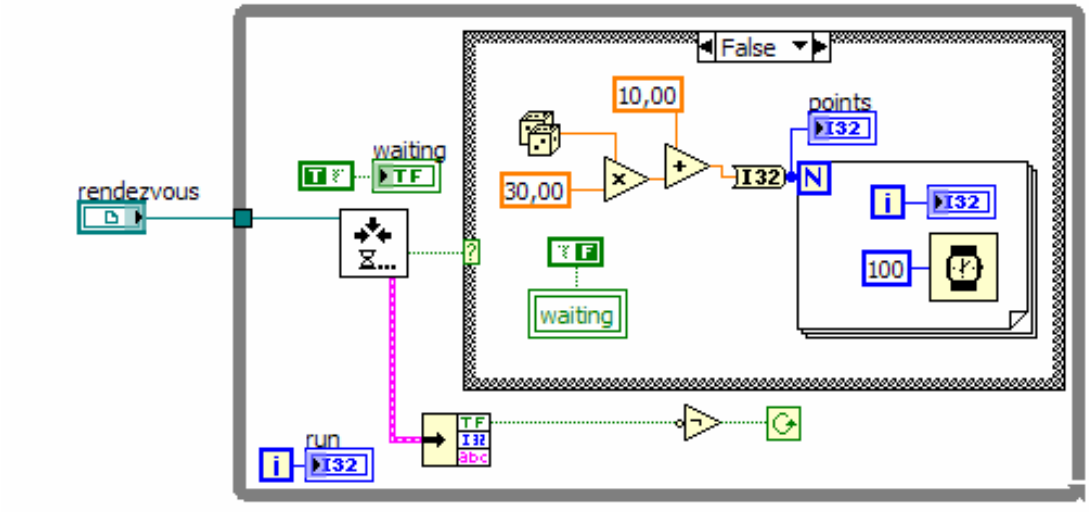




Rendesvous

- Wichtig in Multithreading-Applikationen
- Rendesvous können benutzt werden um, parallel laufende Programmfäden zu synchronisieren





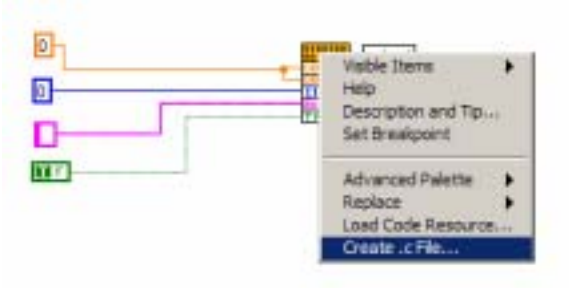
Externen Code aus LabVIEW aufrufen

Externen Code aufrufen

- Welche Möglichkeiten gibt es?
  - Code Interface Nodes (Win, MacOS, Linux)
  - DLLs (Windows) bzw. Shared Libraries (Linux, MacOS)
  - ActiveX (Windows)
  - .Net (Windows)

Code Interface Nodes

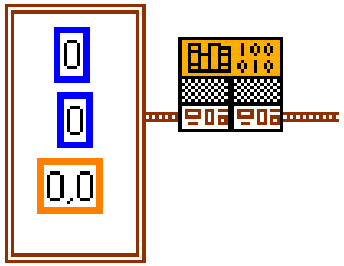
- Erstellen von eigenen Primitiven die kompilierten Code aufnehmen
- Entstand in den Anfangstagen von LabVIEW, bevor es DLLs gab
  - + keine separaten Dateien wie bei DLL
  - + Zugriff auf alle LabVIEW Manager-Funktionen
  - - funktioniert nur mit speziellen Compilern
  - - properitäres Format, nur bei NI dokumentiert
  - – limitierter Funktionsumfang: CINLoad, CINInit, CINAbort, CINSave, CINDispose, CINUnload, and CINProperties




```
/* CIN source file */

#include "extcode.h"

MgErr CINRun(float64 *arg1, float64 *arg2, int32 *arg3, LStrHandle arg4,
    LVBoolean *arg5);

MgErr CINRun(float64 *arg1, float64 *arg2, int32 *arg3, LStrHandle arg4,
    LVBoolean *arg5)
    {

    /* Insert code here */

    return noErr;
    }
```

DLLs und Shared Libraries

- Prinzip existiert auf allen Betriebssystemen
  - + Mit vielen Compilern möglich
  - + gut dokumentiert
  - + beliebig viele Funktionen in einer DLL
  - - Zu jedem DLL-Aufruf gehört eine extere Datei
  - - Nicht alle LV-Manager können verwendet werden

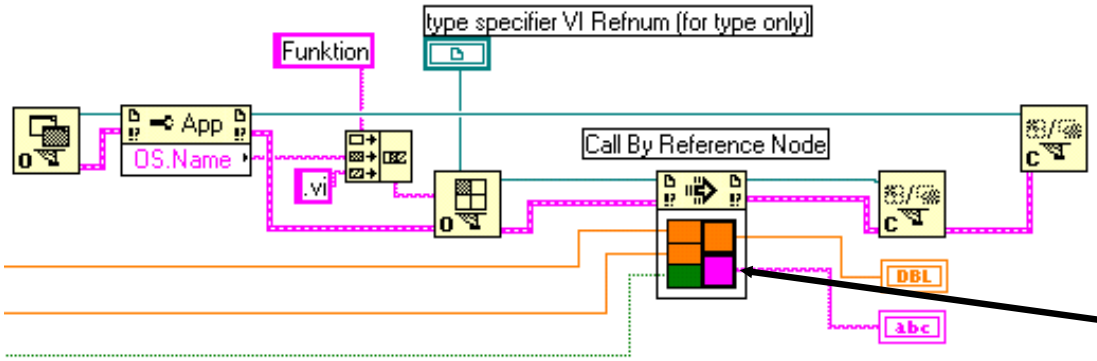


Debuggen eines DLL-Aufrufs

- DLL-Aufrufe sind u.U. kritisch, vor allem wenn Pointer verwendet werden
- Bei Fehlern beendet Windows oft den ganzen Prozess, also das komplette LabVIEW.
- Ist ein VI mit DLL-Aufruf nicht Lauffähig oder stürzt ab, ist folgendes zu
  - Ist die DLL unter dem angegebenen Pfad zu finden
  - Die Fehlermeldung ***function not found in library***, deutet auf falsch geschrieben Funktionsname oder Gross-/Kleinschreibung hin
  - Alle Eingange zum DLL-Knoten müssen verdrahtet sein
  - Typen der Parameter im DLL-Aufruf müssen mit denen der Funktion übereinstimmen
  - Aufrufkonvention (Calling Convention) muss zur Funktion passen
    - C (cdecl)
    - Default (_stdcall, wird oft von der Windows API verwendet)
- Detaillierte Information gibt es in der LabVIEW-Hilfe
  - “Using External Code in LabVIEW manual”

Plattformunabhängigkeit mit DLLs

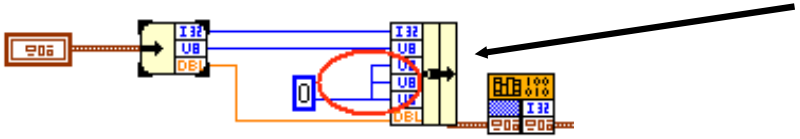
- DLLs müssen für jedes Betriebssystem übersetzt und in den VIs ausgetauscht werden.
- LabVIEW läd DLLs immer statisch, es muss also schon beim Programmieren die richtige DLL zugeordnet werden
- Plattformunabhängigkeit kann erreicht werden, wenn DLL-Aufrufe in VIs verpackt werden, die wiederum dynamisch geladen werden (Application Control)



Hier werden die Wrapper-VIs geladen, die dann wiederum die DLLs aufrufen

Cluster in Parametern

- Oft werden Datenstrukturen in Funktionen (in DLLs) erwartet
- Werden Cluster verwendet ist das Byte-Alignment zu berücksichtigen und eventuell durch Füll-Bytes anzupassen
  - Byte-Alignment ist die minimale Paketgröße für Parameter im Cluster
  - LabVIEW verwendet 1: Ein U8-Wert benötigt ein Byte
  - MS Visual C++ verwendet 4: Ein U8 Wert benötigt 4 Byte



Hier sorgen 3 Bytes dafür, dass die U8 Variable auch in einem 4-Byte Paket übergeben wird

Was sind LabVIEW Manager?

- Low-Level Funktionen für die Verwendung in DLLs und CINs
  - Memory Manager
    - Ändern der Arraygröße in CINs/DLLs Wichtig! Nur mit diesen Funktionen erlaubt
  - File Manager
  - Support Manager
  - Datentypen

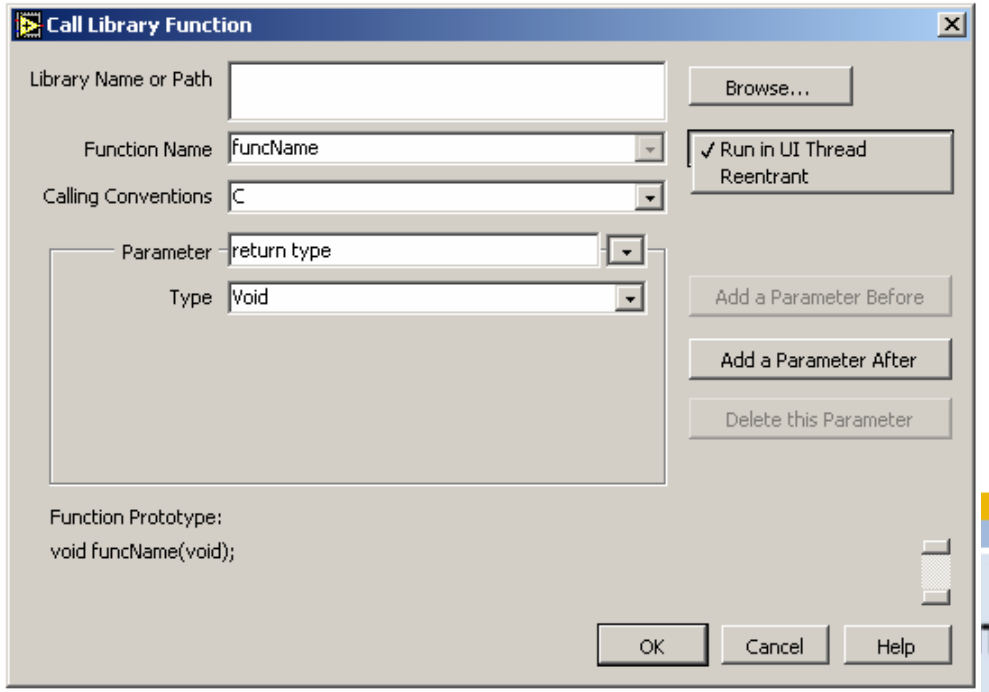
- Mehr Info in der LabVIEW Hilfe->Bookshelf->
  Using External Code in LabVIEW

Achtung bei Multithreading!

- Beim Verwenden von CINs und DLLs in multithreaded Applikationen müssen die externen Funktionen Thread-Safe sein!
  - Zugriff auf gemeinsame Ressourcen (globale Variablen, Hardware, Dateien) muss in DLL/CIN vermieden werden
- LabVIEW kann Thread-Safe in DLLs nicht automatisch erkennen
- Bei CINs kann es LabVIEW mit einem #define bekannt gemacht werden

Thread-UnSafe
(run in UI Thread)

Thread-Safe
(Reentrant)



Wann ist Multithreading zu vermeiden?

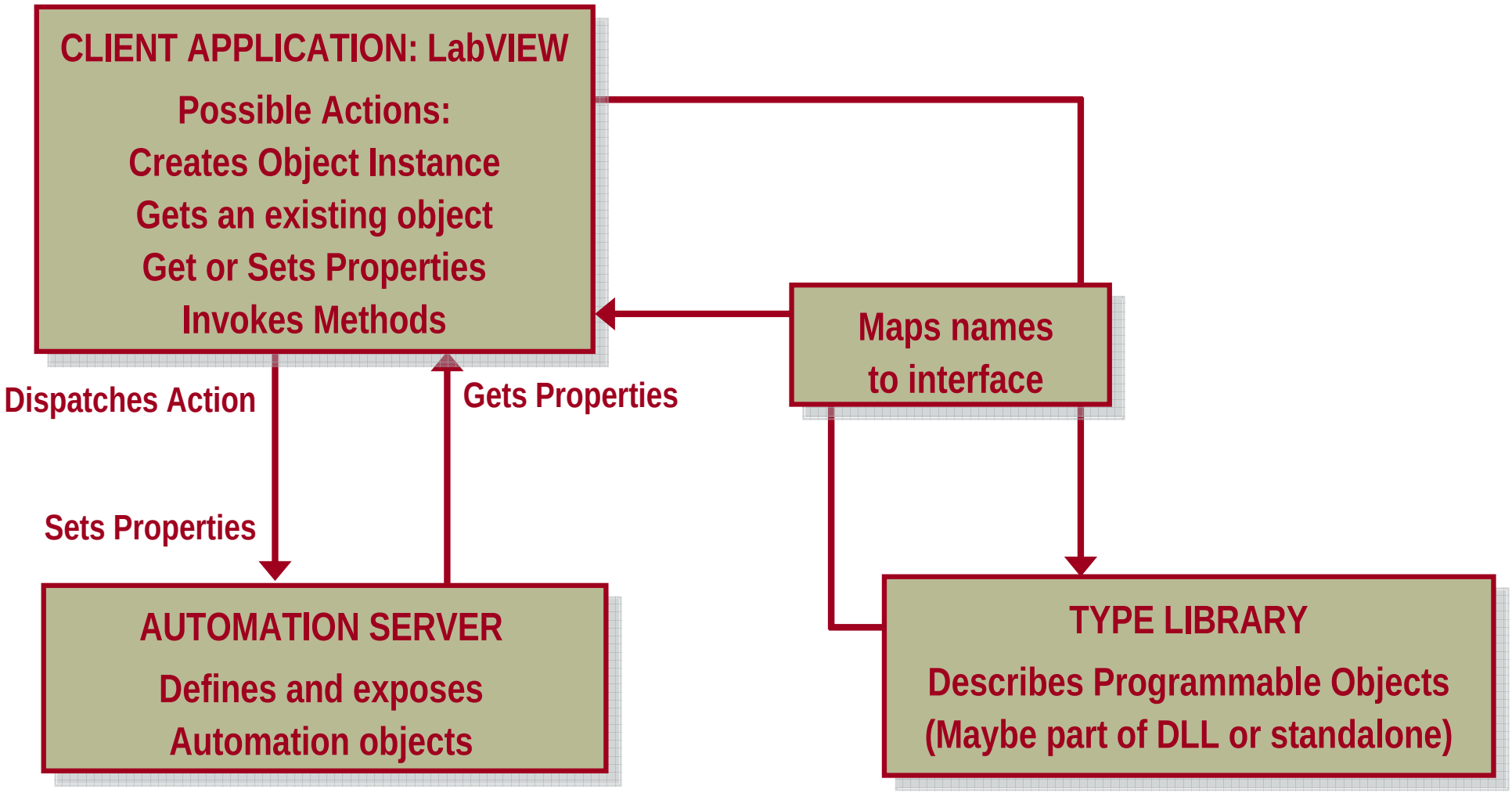
- Abläufe sind prinzipbedingt sequentiell
- Thread-Wechsel kosten zu viel Performance
- wenig Speicher oder langsamer Prozessor
- Viele thread-unsafe CINs und DLLs
- Viele Zugriffe auf das User Interface (bzw. UI Thread)
- wenn es den Aufwand nicht lohnt...
  - Priority-Invertierung: Niedrige Threads können wichtige verdrängen
  - Thread Starvation: Zu viele Wichtige Threads verdrängen alle unwichtigen
  - Deadlocks: mehrere Threads warten auf Ressourcen die andere Threads

    bereits blockieren und ebenfalls warten...

ActiveX

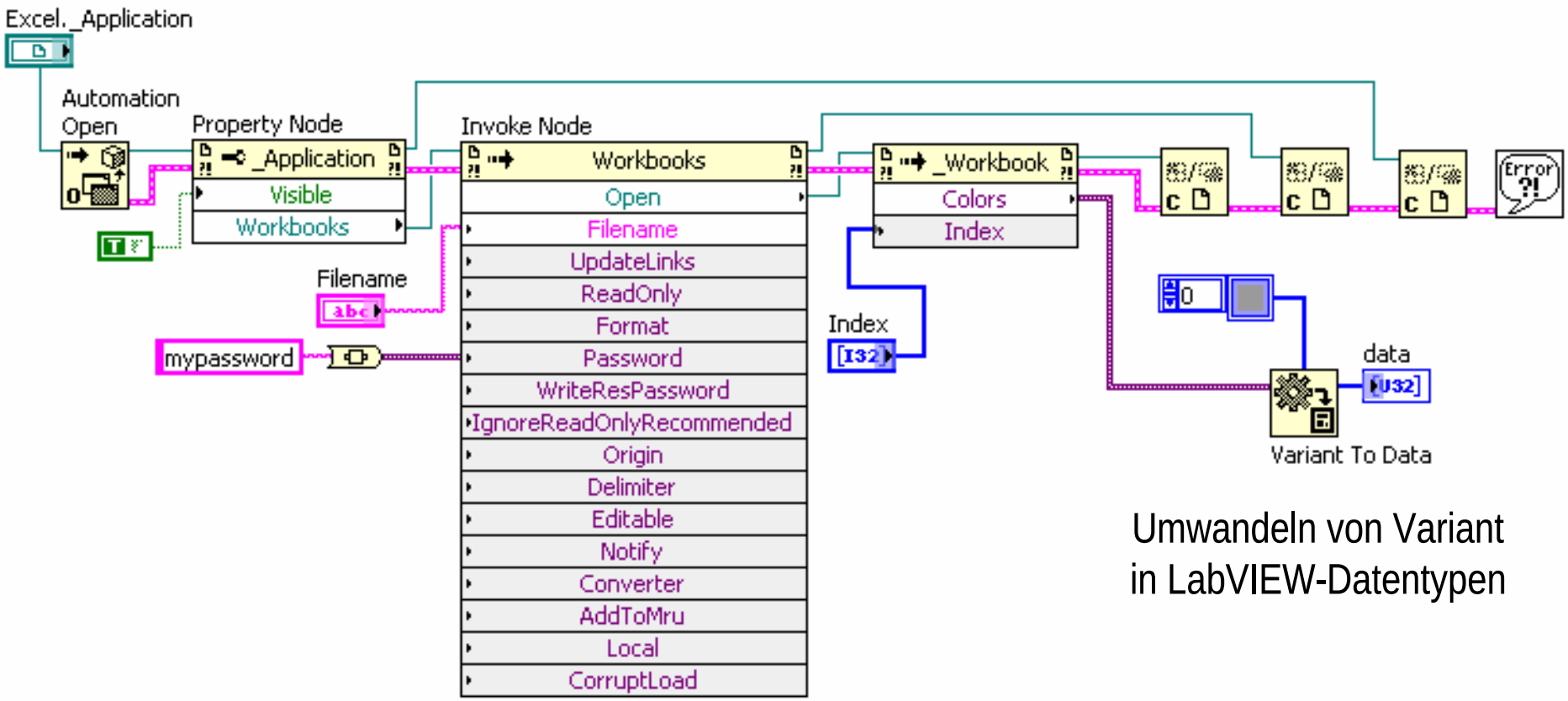
- ActiveX umfasst
  - Automation Server — Applikation die Funktionen bereitstellet
  - Automation Client — nutzt exportierte Funktionen von Automation Servern
  - ActiveX Controls — Bedienelemente zum einbinden in Frontpanel
- Ein Automation Objekt besitzt
  - Methoden
  - Eigenschaften
- ActiveX-Komponenten müssen am System registriert werden. Entweder passiert dies automatisch bei der Installation oder manuell mit *regsrv32.exe* und dem Namen der Komponente

LabVIEW als ActiveX Client



PS: LabVIEW kann auch ActiveX Server sein

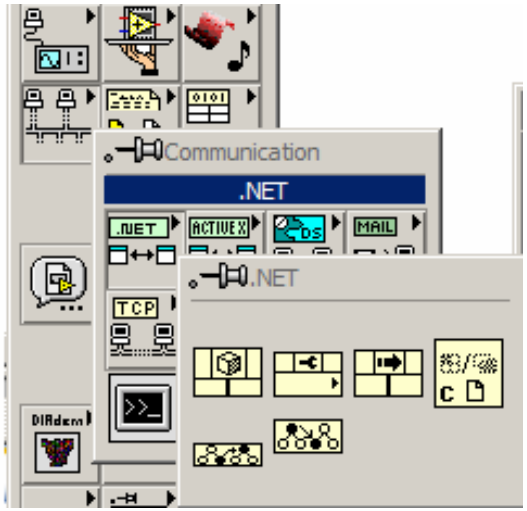
ActiveX Beispiel

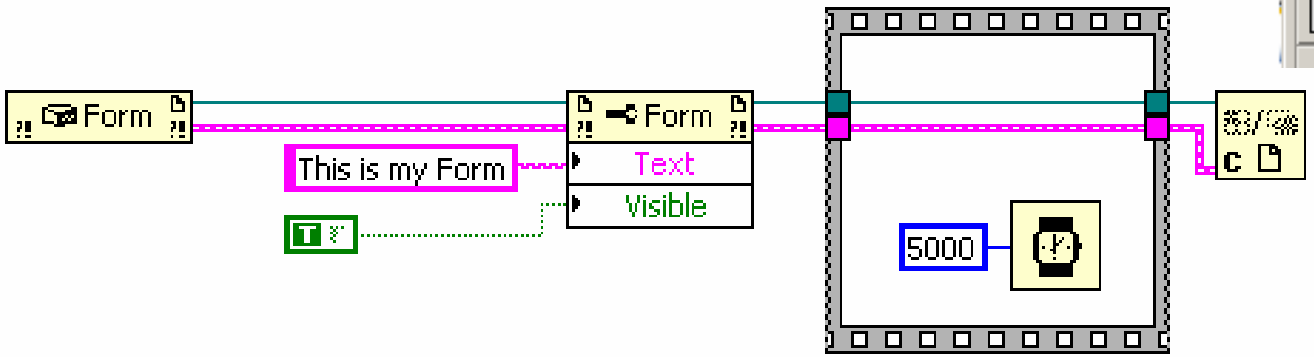


Umwandeln von Variant
in LabVIEW-Datentypen

.Net

- Aufrufen von Asseblys in VB.Net, C#, etc
- Aufrufen von API-Funktionen von Windows (.Net Framework)
- Ähnlich wie ActiveX
  - Open Assembly
  - Property Node
  - Invoke Node
  - Close Assembly





Weiterführende Infos und Links und Literaturtipps

- Software Engineering in
  - LabVIEW Hilfe: Development Guidelines
  - LabVIEW Technical Resource – http://ltrpub.com
  - www.openg.org
  - zone.ni.com
  - Buxton, Naur and Randell – *Software Engineering Concepts and Techniques*, 1968 NATO Conference
  - D. Parnas – *On the Criteria to be Used in Decomposing Systems into Modules,* CACM 1972
  - E. W. Dijkstra – *A Discipline of Programming*, 1976
- Design Patterns
  - NI Training: *LabVIEW Intermediate I: Successful Development Practices* (3-day course)
  - *A Software Engineering Approach to LabVIEW* (Watts, Conway) ISBN 0-13-009365-3
- Performance
  - NI Application Note 168: LabVIEW™ Performance and Memory Management
- Multithreading
  - Moderne Betriebssysteme von Andrew Tannenbaum
  - LabVIEW Intermediate 2 Kurs