

01 Architecture

Architecture Principles

General

- When working with clusters and SubVIs, pass only the data the SubVI needs. Do not let the SubVIs consume the entire cluster.
 - See [Passing Through](#)

Structure

- Adhere to the following:
 - [11 Project Structure](#)
 - [12 Dependency Structure](#)
 - [13 Unit Test Structure](#)
 - [14 Documentation Structure](#)
 - [15 Resources](#)
 - [16 Deployment Structure](#)

Libraries and Classes

- Wrap VIs in classes (.lvclass) or libraries (.lvlib) for name spacing and generally improved encapsulation
 - See [Organising Code](#) for details
 - Create a separate subfolder for each .lvlib or .lvclass that contains the container file and all members

Reentrancy

- If a technology supports multiple threads (more than one instance in memory at the same time), all the VIs of the corresponding driver usually need to be set to reentrant execution.

Constants

- For constants that are reused throughout the project, create a subVI that only contains a single constant on the block diagram and one indicator returning the constant value
 - See [00 Style](#) for VI naming
 - See [02 Layout & Design](#) for VI icon design

Database

- Wrap each query in a separate SubVI
 - this allows for easy reuse
 - add documentation to describe the returned values
 - optional: convert the returned 2d-array of strings into a 1d-array of typedef'ed cluster

2. A DB connection is like a shared resource that cannot be used in parallel. (also true for SQLite)
 - use one db connection instance per consumer to avoid blocking

Configuration

- **Simultaneous Creation:** When creating access to a configuration file, the file and its related content must be created at the same time.
 - **Focused SubVI Design:** A configuration access SubVI should only contain related information. This minimizes dependencies and simplifies maintenance and testing.
 - **Early Testing:** Perform testing early to identify and resolve potential issues promptly, avoiding prolonged troubleshooting later.
 - **Default Config Creation:** Implement a VI that creates a configuration file in the correct format, even if the application does not have a “save config” requirement, ensuring consistency and avoiding potential issues.
-

Runtime Behaviour

Functions/Primitives

- Do not use “First Call?”, use an FGV or AE instead
 - Working with frameworks that support dynamically starting and stopping, we often want to restart processes without having to restart the whole call chain or application.
 - Using a functional global variable that allows resetting the internal state on demand is more flexible and more readable than the “First Call?” primitive
 - See also [Emmanuel Geveaux' LinkedIn article](#)

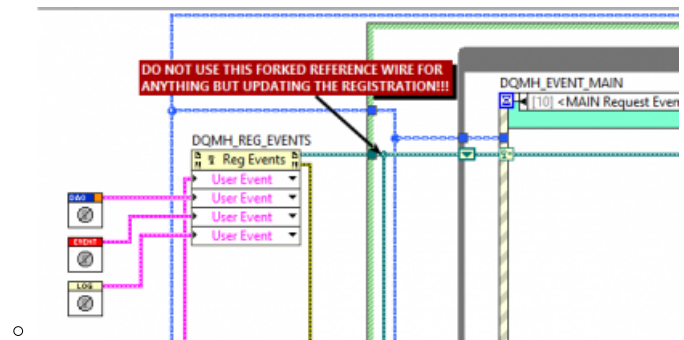
Events

- Watch [10 things \(or more\) to know about events in LabVIEW](#) (Olivier Jourdan, Wovalab)

Event Registration

- Maintain a single Event Structure per registration refnum
 - each registration creates a single event queue which *must not* be shared between event structures to avoid race conditions and non-deterministic behaviour
- Do not fork the event registration reference wire (see previous point)
 - the only exception is for updating the

registration - in that case, be very explicit with comments and explaining what to do and what not to do with the forked wire



Communication

- Transferring data from one loop to another loop (VI-local)
 - Using VI server (reference or property node) to update the value of a control or indicator is the least efficient mechanism by far.
 - Local variables come with only a slight overhead to directly writing to the terminal.
 - Global variable would be the fastest with the additional caveats of race conditions and issues when global variables are not handled correctly.
 - RT FIFO or a User Event add context and encapsulation.
 - See the [LabVIEW Craftsmen blog](#)

LabVIEW Mechanisms

Variant Attributes

- Do NOT use variant attributes for data storage unless their high performance is definitely needed.
 - Attributes hide away any information about the datatypes at edit time

Structures

Unwired Terminals

- Use “Use Default If Unwired” appropriately

- If no default value should ever be used, enable “Linked Input Tunnel”

Conditional Disable Structures

- When using CD Structures, put them into the project tree:
 - as far up as possible (ie. direct at the project root level, as opposed to the targets)
 - as far down as necessary (eg to distinguish different FPGA targets)
 - For each CD Symbol in the project, add a log entry or similar to your project so you can identify at runtime which values were defined for the symbols
 - Put these into a subVI directly after starting the application
 - When using HSE libs, put this subVI directly after initialising the HSE-Logger
-

Data Design

Comparison

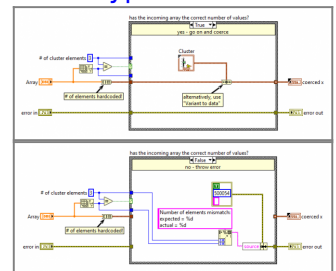
- For floating point values, do not use the “Equals?” function. Due to the resolution of the datatype, values might differ in the decimal places. One possible solution is to use “In Range?” with upper and lower limits instead.
 - See also <https://www.ni.com/docs/en-US/bundle/labview/page/floating-point->

numbers.html



Array to Cluster Conversion

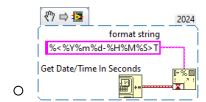
- The Array to Cluster node has the number of elements hard-coded
- As an array is dynamic, we need to check the size during runtime
- To convert the generic cluster to a typedef-ed one, use the [Coerce To Type](#) function



Timestamps

- When needing timestamp information in string format (logging, file names, etc.), always use the *Format to String* function with extended format specifiers (%<...>T) to have complete control over the output format
 - This avoids errors with special characters

like “/” or
“.”



- See [Format Codes for the Time Format String](#) in the LV help

Front Panel Interaction

These guidelines cover **reading and writing of front panel element *properties*** (e.g. visibility, color, enable/disable), not values. Reading or writing control values via references or property nodes is generally discouraged.

General principles

- Avoid updating FP elements via value property nodes whenever possible.

- Prefer the simplest possible implementation.
- It must be easy to find where a specific FP element is interacted with.
- Do not pass references “just in case”; only pass what is actually needed.
- Prefer explicit, type-safe, and local solutions over dynamic or generic abstractions.

Approaches

Choose the simplest suitable one:

1. Local Property Nodes

Place property nodes directly linked to the FP element.

- Pros:
 1. Maximum readability

and
trace
abilit
y

2. No
indire
ction,
easy
to
debu
g

3. Stron
g
visual
coupli
ng
betw
een
UI
and
logic

2. Cons:

1. Not
reusa
ble

2. Can
clutte
r the
diagr
am if
overu
sed

2. Pass minimal FP references to SubVIs

If a SubVI is
required,
pass only
the specific
references
it needs.

- Pros:
 - 1.

G

o
o
d
b
a
l
a
n
c
e
b
e
t
w
e
e
n
r
e
u
s
e
a
n
d
c
l
a
r
i
t
y
2.
E
x
p
l
i
c
i
t
d
e
p
e
n
d
e
n
c
i

e
s
3. S
t
i
l
l
e
a
s
y
t
o
t
r
a
c
e
u
s
a
g
e

2. Cons:

1. S
l
i
g
h
t
i
n
c
r
e
a
s
e
i
n
w
i
r
i
n
g
e
ff

o
r
t
2.
T
i
g
h
t
c
o
u
p
l
i
n
g
b
e
t
w
e
e
n
c
a
l
l
e
r
a
n
d
S
u
b
V
l

3.
Clust
er of
type-
speci
fic
refer
ence
s

Grou

p FP
refere
nces
into a
cluste
r with
strict
typin
g.
Use
Unbu
ndle/
Bundl
e
(and
Build
Array
wher
e
applic
able).

- P
r
o
s
:

1.

N
a
t
i
v
e
L
a
b
V
I
E
W
s
o
l
u
t
i
o
n

(
s
i
m
p
l
e
,
s
t
a
b
l
e
,
n
o
e
x
t
r
a
a
b
s
t
r
a
c
t
i
o
n
s
)
2. T
y
p
e
-
s
a
f
e
a
n
d
s
e
a

2.
- C
o
n
s
:
1.

r
c
h
a
b
l
e

S
u
b
V
l
s
m
a
y
r
e
c
e
i
v
e
r
e
f
e
r
e
n
c
e
s
t
h
e
y
d
o
n
o
t
u

s
e
2. S
l
i
g
h
t
l
y
r
e
d
u
c
e
d
c
l
a
r
i
t
y
o
f
"
w
h
o
u
s
e
s
w
h
a
t
"
(
b
u
t
s
t
i
l
l
v
i

s
i
b
l
e
o
n
B
D
)
3.
F
P
o
f
s
u
b
V
l
m
a
y
s
h
o
w
a
h
u
g
e
c
l
u
s
t
e
r

4
·
M
a
p
-
b
a

**s
e
d
a
p
p
r
o
a
c
h
(
g
e
n
e
r
i
c
r
e
f
e
r
e
n
c
e
s
+
e
n
u
m
a
c
c
e
s
s
)**

**S
t
o
r
e
r
e
f**

e
r
e
n
c
e
s
i
n
a
m
a
p
u
s
i
n
g
g
e
n
e
r
i
c
t
y
p
e
s
a
n
d
a
c
c
e
s
s
v
i
a
e
n
u
m
.

2.

C
o
n
s
:

F
r
c
j
e
c
t
H
y
g
i
e
r
e

D
e
p
e
n
d
e
n
c
i
e
s

