## Training Outline

<table>
<thead>
<tr>
<th>Lesson 1</th>
<th>Lesson 2</th>
<th>Lesson 3</th>
<th>Lesson 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Engineering Process</td>
<td>System Architecture</td>
<td>Development Best Practices</td>
<td>Validation</td>
</tr>
</tbody>
</table>

## Lesson 1: Software Engineering Process

### TOPICS

1. Software Development Process Models
2. Configuration Management and Source Control
3. Requirements Gathering and Documentation
Software Engineering Process

Software engineering process – A structured process that defines specific phases and tasks to accomplish during the development of software

Example phases
• Requirements Gathering
• Design
• Development
• Validation
• Deployment

Do I Need It?

• Scope of application:
  – Time/Size
  – Multiple Developers
• Time that it takes to learn the method or overhead associated with the process vs. overall scope of the project
• One size does not fit all

I. Software Engineering Process Models

Software engineering process model – A model that describes the steps to follow when developing software

Sample software engineering process models
• Waterfall Model
• V-Model
• Spiral Model
• Agile Model
Waterfall Model

**Advantages**
- Heavily documented
- Good for regulated industries

**Drawbacks**
- No overlap between stages
- Returning to an earlier phase can involve costly rework
- Do not see actual results for a long time

---

Software Engineering V-Model

---

Spiral Model

---
Agile Process Model

- Break application into small tasks
- Requirements Gathering
- Design
- Development
- Validation
- Deployment
- Demonstrate Iteration to Stakeholders

Each cycle represents completion of one small task.

Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Agile Manifesto

- **Individuals and interactions** over processes and tools
  - Pair programming
  - Scrum model (daily cooperation and status reports, “Scrum master”)
  - Co-location
  - Trust in developers

- **Working software** over comprehensive documentation
  - Use of prototypes rather than design

- **Customer collaboration** over contract negotiation
  - “Product Owner” represents customers
  - Frequent demonstration of working software

- **Responding to change** over following a plan
  - Iterative development
  - Feature prioritization
Basic Agile Process In Work

1. Sprint Planning Meeting (Plan 2-4 weeks)
   A. Review feature backlog
   B. Define teams
   C. Assign resources to each requirement
2. “Sprint”
   A. Daily Meeting (“Scrum” or “Daily Standup”)
   B. Scrum of Scrums (representative from each team)
   C. Develop Features (sometimes in pair programming)
   D. Repeat until Sprint is done (must finish on time)
3. Sprint Review Meeting
   A. Present work,
   B. Return unfinished features to backlog

Summary of Process Models

- Waterfall
  - Advantages: documentation, audit trail, accountability
  - Disadvantages: static requirements, high overhead
- V-model
  - Advantages: Tie between design and test
  - Disadvantages: needs well-written requirements, relationships sometimes contrived
- Spiral model
  - Advantages: Risk-based approach to development, iterative development
  - Disadvantages: Integration issues
- Agile model
  - Advantages: “Light-weight” process, iterative development
  - Disadvantages: Co-location generally required, works better for maintenance

Discussion

Is there already a process in place?
What process(es) make sense?
CASE Tools

- Computer Aided Software Engineering (CASE) is the practice of using software tools to ease software development burdens.
- Types of CASE Tools
  - Configuration management
  - Project management
  - Requirements management
  - Modeling
  - Code generation
  - Automation of development tasks
  - Code analysis
  - Testing

II. Configuration Management

- Typically performed with the assistance of software configuration management tools
- Should be used throughout software engineering process

Tools for Configuration Management

- Source Code Control
- LabVIEW Tools
  - Graphical Differencing
  - VI Merge
  - Compare VI Hierarchies
- Automated Build Tools
- Deployment Tools (RTAD)
Source Code Control

- Software that tracks changes to files
  - Stores all versions of files and their change records
    - Files are stored in a code repository
    - Changes are made within a local workspace
  - Provides multiple developers access to files
- Source code control options
  - Use within LabVIEW Professional Development System
  - Use directly through a source code control tool

Integration with the LabVIEW Project

- LabVIEW Project makes accessing SCC in LabVIEW simpler
  - Right-click one or more files to check in or out
  - Right-click and select Show Differences to view edits interactively (Perforce and VSS)
- File icon shows current status
  - Checked in
  - Checked out

Common Source Code Control Functions

- Check out
- Commit / Submit
- Add
- Update to revision
- Lock
- Branch
- Merge
Branching Code

Branch—Split from the main development line to create a new version of the code

Merging Code

Merge—Integrate the development split into the main development line

Team-Based Development

Server stores all revisions of files
- Project files
- VIs
- LLBs
- Documents

Each developer can check copies of files in and out as needed
Team-Based Development Policies

- Maintain consistent file hierarchy
- Branch code to create new versions
  - Or to create a testing sandbox
- Parallel development on the same code module should be avoided when possible
- Only check out code that you are actively modifying
- Test code before you check it back in
- Document what was changed in the code
- When revision is complete, merge your code with the main line of development

Source Code Control – Supported Packages

- Perforce
- Subversion
- Microsoft Visual SourceSafe
- IBM Rational ClearCase
- MKS Source Integrity

Features of Perforce and Subversion

**Perforce**
- Proprietary License (Free for up to two users)
- Graphical UI
- Changelists
-Branchspecs
- Hybrid of merge and lock concurrency model

**Subversion**
- Open-source, free application
- Command line interface
- Free third-party client applications are available (for example, TortoiseSVN)
- Branching and tagging are cheap operations

Requires a third-party plug-in for LabVIEW Integration
TortoiseSVN Client for Subversion

- A client for Subversion source code control
- Implemented as a Microsoft Windows shell extension
  - All commands are available directly from Windows Explorer
  - View the status of your files directly from the Windows Explorer

III. Gathering and Managing Requirements

Requirements – Statements that define the needs and objectives of the software project

- Provide a mechanism to understand what the application should do

Defining Requirements

The process of defining requirements includes the following items, not necessarily in this order:

- Identification
- Constraints
- Analysis
- Representation
- Communication
- Validation
- Management
Identification

Use the following steps to identify the software requirements:

• Determine software purpose
• Elicit requirements from stakeholders
• Determine system requirements
• Determine user interface requirements

Determine Software Purpose

• What is the context of the project?
• What is the purpose of the software you want to create?
• Why create the software?
• What conditions must the software satisfy?

Elicit Requirements From Stakeholders

• Interviews
• Meetings/brainstorm sessions
• Questionnaires
• Observe the environment where the software will be used
Determine System Requirements

- System requirements clearly describe the technical environment
  - Operating system
  - CPU power
  - System memory
  - Peripherals
  - Drivers and APIs

Constraints

Limits within which the system must operate
- Typical constraints
  - Cost
  - Hardware interface characteristics
  - Reliability
  - Safety

Determine User Interface Requirements

- Define how users interact with the software
- Analyze the tasks the user performs
Analysis

Compile requirements that are determined to be adequate by the stakeholders
• Identify issues with the requirements
• Classify requirements
• Determine risks

Representation

• Create a written form of the requirements
  - Use a specification such as IEEE-830
• Create a prototype that demonstrates the implementation of the requirements

Representation (continued)

Keywords help define the degree of necessity of the requirements

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shall</td>
<td>Essential requirement</td>
</tr>
<tr>
<td>Should</td>
<td>Conditional requirement</td>
</tr>
<tr>
<td>Can</td>
<td>Optional requirement</td>
</tr>
</tbody>
</table>
**Communication**

- Allow all stakeholders to review the requirements
  - Often leads to a modification of the requirements
- Result of communication must be agreement

**Validation**

Verifies that requirements are robust
- Understood by all, including the developers
- Do not illustrate implementation
- Testable
- Traceable

**Management**

Activities that manage and track the requirements
- Evaluate change requests throughout the development cycle
  - Requirements often change or are added as the project progresses
- Verify and track requirements
Defining Requirements

Specific
Measurable
Attainable
Realizable
Traceable

= S.M.A.R.T.

Examples of Requirements

Bad
1. “easy to use”
2. “secure”

Good (Testable & Traceable)
1. An operator, given X amount of training, is able to do A, B, and C
2. The system must be password protected. The password must have X alphanumeric characters and change every X days.

Common Requirements of Embedded Systems

- Functional
  - IO points
  - Configuration methods
  - User commands
  - External Interfaces (network, etc)
- Environmental
  - Temperature
  - Vibration
  - Ingress Protection
- Performance
  - Determinism
  - Latency/delay
  - Throughput
  - Storage Volume
- Reliability
  - Up-time
  - Maintenance windows
  - MTBF
  - Fault responses/failure modes
- Security
  - Physical
  - Authentication
  - Encryption
- Non-Functional
  - Regulatory standards
  - Deployment and update procedures
  - Documentation and coding standards

Requirements template included in course resources
Requirements Tracking Introduction
(We Will Discuss More During Validation Section)

• A project can have hundreds or thousands of requirements
• Each requirement must be implemented and tested
  − Trace implementation to a specific section or sections of code
  − Trace testing to a defined testing procedure and/or test results

Requirements Tracking (continued)

• Log and maintain links between the requirements document, source code, and test systems
  − End result is a traceability matrix or other report
  − Structured requirements tracking is often a non-functional requirement of a project
• Requirements tracking can be performed in one of two ways
  − Manually, using a spreadsheet or other data entry tool
  − Automatically, using requirements management tools

Requirements Management Tools

Requirements Management Tool – A tool that aids in requirements gathering and tracking

• Requirements management tools should be considered when:
  − The number of requirements becomes large enough that manual tracking becomes tedious
  − Multiple documents are used to outline different sets of requirements
  − Project requirements are being met in more than one application
• NI Requirements Gateway is an example of a requirements traceability tool
NI Requirements Gateway

Links development and verification documents with formal requirements

---

Quiz - Question 1

What are the 5 parts to a SMART requirement?

---

Quiz - Answer 1

The five parts are specific, measurable, attainable, realizable, and traceable.
Question 2

What are the key features of each process model?
• Waterfall
• V-Model
• Spiral
• Agile

Quiz- Answer 2

What are the key features of each process model?
• Waterfall – Documentation and accountability
• V-Model – Tie between req/design and test
• Spiral – Risk-based development, iterative development
• Agile – “Light-weight” management, iterative development

Lesson 2: System Architecture
I. Software Design-Architectures

Design – A method of problem-solving and planning for a software solution

i.e. taking requirements → design

What is an Architecture?

Architecture – Set of processes, modules, data transfer mechanisms, and synchronization mechanisms in a system

- The term gets overused
- Synonymous with design
- All main pieces to the system, at a high level

How is an Architecture Represented?

- Prototypes/Skeleton
- Documentation
- Diagrams
II. How is an Architecture Used?

- To communicate with stakeholders
  - Builds confidence in investment
  - Helps to identify gaps in requirements
- To guide development
  - Answers “Where are we going?”
  - Defines processes, communication paths, modules, and algorithms
  - Provides a framework to fill in

How is an Architecture Used? (cont)

- To guide test
  - Identifies areas to focus on for integration testing
  - Identifies potential trouble spots for white-box testing
- To guide debugging
  - Shows which processes and modules have access to specific data items
- To guide maintenance
  - Provides a primer for future developers

III. Creating an Architecture

Customize to the Project needs, but in general we need:
A. Prototypes/Skeleton
B. Documentation
C. Diagrams
Prototype

Prototype – A early sample or model built to test a concept.

Advantages of Prototyping
- Risk mitigation
- Evaluation of requirements
- Rapid development (minimal software engineering process)
- Communication to stakeholders
- Evaluation of design alternatives

️ Creating a Prototype is not the same thing as developing.

Skeleton

Top-Level VI with the general structure of the program filled out

Programming Stubs

Part of the Skeleton

Stubs provide a way to communicate what functionality will be needed

Stubs should be executable
B. Documentation

Design documents take the requirements and explain how we will implement each requirement.

The design checklist is a quick and easy way to ensure you have considered most common problems.

C. The Architecture Diagram

- Best used when the application is complex and communication between modules is difficult to visualize
  - Aids in defining software components
  - Aids in visualizing how components interact with each other
  - Descriptions of components can be used to document code later
  - Communicating to stakeholders
- For RIO systems - diagram or model sometimes sufficient
Architecture Diagram Examples

Types of Architecture Diagrams
- Communication
- State
- Class
- Module Hierarchy
- Use Case
- Activity
- Physical
- UML Diagrams

Many types of Diagrams, but for this class we will focus on Communication, State, and Physical with emphasis on Communication.
Architecture Diagram - Communication

Focuses on displaying the communication paths between processes

Maps very well to LabVIEW

Vector-Based Drawing Programs

• Aid in creating our Diagrams
• Some offer specific support for drawing flowcharts and sharing data
  – yEd Graph Editor
  – DrawAnywhere
  – OmniGraffle
  – SmartDraw
  – Visio

Demonstration: yEd Graph Editor
Modeling Languages

Modeling language – Any artificial language that can be used to express knowledge or systems in a structure that is defined by a consistent set of rules.

Examples of modeling languages:
• Flowchart
• Unified Modeling Language

Modeling Language Pros:
• Consistent Diagram enforced by the language
• Enables interesting features like code generation

Modeling Language Cons:
• Have to learn another language
• Strictness implies longer design time

Unified Modeling Language (UML)

• Standard set of graphical representations used to define large systems
• Not specific to any individual programming environment
• Builds upon object-oriented programming concepts
• Documentation is tied directly to objects
• Combines best practices from other data modeling concepts
UML Tools

UML Tool – A software application that supports some or all of the notation and semantics associated with the Unified Modeling Language

- Includes applications which are not exclusively for UML
- Support some functions of UML
  - Diagramming
  - Round-trip engineering
    - Code generation
    - Reverse engineering
  - Model Transformation

Examples of UML Tools

- Examples of LabVIEW UML tools
  - Symbio UML Modeler
  - G# toolkit uses StarUML
- Other UML tools

IV. Components of an Architecture

A. Processes
B. Communication
C. Modules
D. Synchronization
E. State Machines
A. What is a Process?

- Process—(for purposes of this discussion) an independent, asynchronously executing segment of code
- Basically: A loop
- Not the same as a thread or a task in LabVIEW
- May execute on multiple cores, processors, or machines.
- May be created statically or dynamically

Common Examples of Processes

- Data Logger
- Control Model
- IO Engine
- Alarm Engine
- Cross-System Data Relay
- Sequencer
- User Interface

How Do You Identify a Process?

- Requires some practice
- Technique
  - Identify and group high level tasks
  - Determine which tasks can share processes
  - Determine which actions are needed to accomplish each task
  - The actions needed for each task may become states, messages, and/or module commands
How Do You Identify a Process? (cont)

- It might be a process if:
  - It involves sequential steps (probably, esp if its async responding to other processes)
  - It requires a specific priority (like time-critical)
  - It requires a specific level of determinism
  - It happens at the same time as other tasks

How Do You Identify a Process? (cont)

More processes =
- More to develop
- More complex communication
- More complex relationships to debug

Diagramming a Process

Communication Diagram

- Process A
- Process B
B. What is a Communication?

Communication—The act of transferring data from one process to another process

- Includes moving data intra-program, between programs, or between machines
- We will go deep into communication in a later lesson

Diagramming Communication

C. What is a module?

Module—(for purposes of this discussion) a functional grouping of code that may be called by one or more processes

- May be tied to data storage/transfer (FGVs)
- May be tied to synchronization (Mutex)
- Discussed in more detail later in the course
How do you Identify a Module?

1. Start with the analysis done for processes
2. Study the actions needed to accomplish each task
3. Make sure actions are as elemental as possible without implementing them
4. Group the actions into logical groups
5. Groups become modules, actions become commands/Vis

Tip: Modules and data items are usually nouns, commands are usually verbs

How do you Identify a Module? (cont)

Modules should
- High cohesion: sense of purpose
- Low coupling: Reliance on the calling process or other modules
  - Good for unit testing
  - Good for debugging
- Encapsulate similar operations

Examples of Common Modules

- Memory Maps
- Data Transfer Mechanisms
- Configuration Mechanisms
- File Managers
- Protocols
- Instrument Drivers

Modules are good places to get code reuse
Discussion

What is your current method of creating modules in LabVIEW?

Modules and Processes

• A module is:
  – The stuff that goes into a process
  – Grouping of functions
  – Class, Library, or Single SubVI
• A process is:
  – The loop. Module is the code within it.
• A process can utilize multiple modules, and a given module can be utilized by multiple processes.

Diagramming Modules and Processes

Normally modules are not represented in Communication Diagrams
Diagramming Modules and Processes

VI Hierarchy

D. What is synchronization?

Synchronization—(for purposes of this discussion) knowing when to execute a particular process

• Processes rarely execute all the time
• Often tied closely with data (messages)
• Not always tied to HW clock
• Processes may use more than one type of synchronization
  – Ex: State Machine can have one per state

Synchronization (Implementations)

• Event Driven
  – Queue
  – Notifier
  – Occurrence
  – User Event
  – RT FIFO
  – Hardware
  – Rendezvous
  – Timed Loop
• Periodic Synchronization
  – Timed Loop
– Timing Functions
  • Scheduled
  – Timed Loop
  • Mutex
  – Semaphore
  – DVR
  – Single-Element Queue
  – SubVI
E. What is a State Machines

State machine—A type of process that comprises of states associated to transitions

How do we diagram a state machine?

State Machine Basic Example

How do we represent in a Communication Diagram?

Diagramming State Machines as a Process

Option A?

Option B?
Diagramming State Machines as a Process

Why is this wrong?
• State diagrams should not be mixed with our communication diagram
• It implies we have three processes
• Process Initialize synchronizes with process Run
• Process Run synchronizes with process Exit

Diagramming State Machines as a Process

Some Design Tools allow for linking diagrams

yEd Example:

The Power behind Diagrams and LabVIEW

Graphical Programming Language :=
Communication Diagram is very similar to Block Diagram
**Quiz- Question 1**

True or False:
A process can utilize multiple modules, but a given module can only be utilized by a single processes.

**Quiz- Answer 1**

False.
A given module can be utilized by multiple processes.

**Quiz- Question 2**

Identify the processes in the following system:
Acquire a stream of data from the FPGA
• Extract a tone from the data
• Publish the tone frequency to an HMI
• Run a PID calculation at 10 Hz that uses the tone as a process variable, a set point from the user interface, and sends and output to the FPGA
• Log the tone to disk every second
• Enable or disable the control upon user request
Quiz- Answer 2

One potential answer:
• FPGA Acquisition and analysis
  − Both synchronized to the FPGA Acquisition
  − Consider modularity
• Logging
  − Periodic Synchronization
• PID and FPGA Output
  − Periodic Synchronization
• Publish data to HMI
• Receive data from HMI (possibly two processes)
Note: The last two are not necessarily processes within the application (SVE, for example)

Lesson 3: Development Best Practices

A. Communication

I. Overview
II. FPGA ↔ FPGA
III. FPGA ↔ RT (Bus Communication)
IV. RT ↔ RT
V. RT ↔ Host (Network Communication)
I. Overview

• Processes must exchange data
• Choice of the correct communication mechanisms is the most important part of an architecture
  – Clearly defined mechanisms aid in verification and maintenance
  – Aim for low coupling between processes
• Document data paths and their associated mechanisms

Considerations

• Type of transfer
• Performance
  – Latency
  – Throughput
• Ease of implementation
• Scope/Scalability

Communication Overview
**Communication Overview**

- **Windows System**
  - Windows VI
  - Enterprise
  - LabVIEW for Windows

- **LabVIEW Real-Time System**
  - Normal Priority Loop
  - Data Storage
  - LabVIEW Real-Time

- **Reconfigurable FPGA**
  - Inter-process Communication
  - High Critical Loop
  - FPGA Interface

- **RT Target**

**Type of Communication**

1. **Inter-process Communication on the FPGA**
   - All will introduce the concept of a shared resource and arbitration on the FPGA
   - Options for communication:
     - Variables: local or global
     - Memory Items: block memory or LUT
     - FIFOs: target or VI scoped

2. **FPGA to RT Target**

3. **Inter-process Communication on RT Target**

4. **RT Target to Windows**

**1. Inter-process Communication on FPGA**

- All will introduce the concept of a shared resource and arbitration on the FPGA
- Options for communication:
  - Variables: local or global
  - Memory Items: block memory or LUT
  - FIFOs: target or VI scoped
Shared Resources on FPGA

Shared Resource – Any resource that can be accessed by multiple locations on the block diagram from the FPGA VI

• Digital outputs on most targets
• Analog inputs on most targets
• Analog output on most targets
• Digital inputs on some targets
• Memories
• FIFOs
• Non-reentrant subVIs
• Local and global variables

Shared Resources on FPGA (cont.)

Before a task can begin using a shared resource, it must wait until the resource is free.

Arbitration on FPGA

Arbitration- the need for extra logic for accessing a shared resource on an FPGA from multiple resources at the same time

• Takes more time than can execute in a SCTL
• Will occur almost all times when have a shared resource accessed from multiple locations on FPGA
• Not desired

Question: what are the shared resources in this block diagram?
Arbitration on FPGA (cont)

Arbitration Options

• Always Arbitrate
• Arbitrate if Multiple Requestors Only
• Never Arbitrate

Avoiding Arbitration

• Minimize Shared Resources
• Use double-sided resources in time-critical code
  – Memory blocks
  – FIFOs
• Avoid multiple readers/writers to double-sided resources
• Set arbitration settings when appropriate
  – Initialization
  – State machines
A. Variables

- Local or global, allow you to access or store data in the flip-flops of the FPGA
- Good for tags; store only the latest data you write to it
  - Good choice if you don’t need every value you acquire
  - Do not need extra code to discard unused values

B. Memory Items

- Block memory, LUT, or DRAM
- Only most recent value is stored
- Can use all available memory on the FPGA
- Use of memory items consumes relatively few logic resources

B. Memory Items- When to Use

Block Memory
- If larger amounts of memory are needed
- If do not have enough logic resources available to use look-up tables

LUT
- Accessing memory in a SCTL and need to read data from memory in the same cycle as the one in which you give the address
- Memory needed < minimum amount of embedded block memory on the FPGA
- Memory needed > free embedded block memory on the FPGA
C. FIFOs

• Target scoped or VI scoped
• Implementation options: flip flops, block memory, or LUT
• Buffer data
• Good for updates and streams

C. FIFOs- When to Use

• Flip-Flops – Use gates on the FPGA to provide the fastest performance. Recommended only for very small FIFOs <100 bytes.
• Look-up Table – Store data in look-up tables available on the FPGA (2 per slice). Recommended only for small FIFOs < 300 bytes.
• Block Memory – Store data in block memory to preserve FPGA gates and LUTs for your VI.

Rules of Thumb

• Messaged based communication, updates, or streams
  – FIFOs
  – Implement own handshaking or memory buffers
• Tags
  – Memory
  – Functional Globals, Local Variables, Global Variables
• Circular Buffers
  – Memory
Exercise 4.1

- Determine Data Sharing FPGA methods for your Project
- Indicate in your diagram

Type of Communication

1. Inter-process Communication on the FPGA
2. FPGA to RT Target
3. Inter-process Communication on RT Target
4. RT Target to Windows

2. FPGA to RT Target

- FPGA and RT Target are inherently asynchronous; amount of synchronization desired will depend on application
- Options for communication:
  A. Scan Engine (IOVs)
  B. Front Panel Controls/Indicators
  C. DMA FIFOs
A. Scan Engine

The NI Scan Engine
- Acquires data from c-Series modules
- Communicates data between RT and FPGA
  - Over PCI bus
  - Over EtherCAT network
- Provides a fault engine
- Publishes I/O values to the network
- Assists in diagnostics and debugging

Scan Engine Architecture

RSI Architecture
Scan Engine:
FPGA Synchronization

• Use the Wait on Rising Edge method of the Scan Clock to start acquisition cycle
  ∆t between rising edge and IO var write must be less than scan period if module acquires every scan

B. Front Panel Controls/Indicators

• Basically tags
• Ideal when
  – Tight synchronization is not required
  – Small amounts of data
• Latest value only
• Large controls/indicators use significant resources on FPGA
  – each register requires a resource
  – Necessitates synchronization mechanisms (can’t transfer half an array)

Handshaking

• Simple simplex handshaking with front panel terminals allows you to send messages or updates.
Handshaking

- Simple simplex handshaking with front panel terminals allows you to send messages or updates.

C. DMA FIFOs

- DMA – A single FIFO transfers data to or from Host VIs by directly accessing memory.
- Consists of two parts – FPGA and RT
- Streams large amounts of data between computer memory and the FPGA
- Done mostly without the CPU
- Provides better performance than using the CPU to read and write data to the indicators and controls

Using DMA FIFOs

- **Blocking** – Host DMA FIFO Read waits indefinitely to read a fixed number of elements from the FIFO.
- **Polling** – Host DMA FIFO Read reads all of the available elements in the FIFO at a user-defined rate.
- **Polling with a Fixed Number of Elements** – Host DMA FIFO Read reads a fixed number of elements at a user-defined rate.
Ensuring Lossless DMA Transfer

- There are two buffers in a DMA Transfer: FPGA and RT
- Buffer overflow usually occurs on the RT side
- DMA size defines size of FPGA buffer, RT buffer will then be a multiple of that

Ensuring Lossless DMA Transfer

- Reduce the rate at which you write data to the FIFO
- Increase number of elements to read on host
- Increase FPGA/host buffer sizes
- Reduce load on CPU
  - Speed of CPU and competing tasks impact transfer rate from DMA to application memory
Interrupts

- Events that can be triggered in the FPGA and acknowledged on the real-time side
- Can use timeouts
- Have numeric identifiers

Rules of Thumb: FPGA to RT Target

- Streams
  - DMA FIFOs
- Messages
  - Front Panel Controls with handshaking and buffers
  - DMA FIFOs
- Tags
  - Front Panel Controls/Indicators
- Updates
  - Front Panel Controls with handshaking
  - Front Panel Controls with interrupts

Exercise 4.2

- Determine communication paths between FPGA and RT
- Indicate in your diagram
Type of Communication

1. Inter-process Communication on the FPGA
2. FPGA to RT Target
3. Inter-process Communication on RT Target
4. RT Target to Windows

3. Inter-process Communication on RT Target

Need to communicate between processes of varying priorities
Options:
A. Single-process Shared Variables with RT FIFO enabled
B. RT FIFO functions
C. Standard Queues
D. Current Value Table using functional global variables (CVT Reference Library)
E. Circular Buffer using functional global variables

A. Single-Process Shared Variables with the RT FIFO Enabled Method

- Enable RT FIFO for deterministic writes and reads
  - Use single element RT FIFO to transfer latest values
  - Use multi-element RT FIFO to transfer buffered values
B. RT FIFO Functions

- Like a queue, but ensures deterministic behavior
  - Imposes size restriction, preallocates memory
  - Must read and write data of defined size
- Accessible by reader and writer at the same time
- Can be lossy; writing element when FIFO is full will overwrite oldest element

Comparison of Two Methods

RT FIFOs
- More control over features; ability to decide when created, shut down, etc
- More configuration options: blocking, polling, etc.
- Use of more complex data types

Shared Variable with RT FIFO
- Communicate directly from high priority piece of code to other network target
- Simpler to use

Note: the two are the same under the hood

C. Standard Queues

- Same queue functions from LabVIEW (non-RT)
- May introduce jitter
- Can transfer any data type
Comparison of Two Methods

RT FIFOs
- Deterministic in time critical code (do not use blocking calls)
- Fixed size
- Execute regardless of errors at input
- Not every data type

Standard Queues
- Use blocking calls when reading/writing to shared resources; can introduce jitter
- Grow as more elements added to them
- Will not execute on error
- Any data type

D. Using FGV – Current Value Table
- Centralizes operations with data shared by many processes
- Allows many application components to share a common data repository
- Allows direct access to latest values of data
- Similar to a collection of single process shared variables

E. Using FGV – Circular Buffer
- Useful way to buffer data between two operations such as data acquisition and analysis
- Useful in applications where operations using the same data set are happening at different intervals
- FIFOs use a circular buffer
Exercise 4.3

• Determine your Interprocess RT communication
• Indicate in your Diagram

Type of Communication

1. Inter-process Communication on the FPGA
2. FPGA to RT Target
3. Inter-process Communication on RT Target
4. RT Target to Windows

RT Target to Windows (Network)

A. Network Published Shared Variables
B. CVT Client Communication
C. Network Streams
D. TCP/UDP
E. AMC (Asynchronous message communication)
F. STM (Simple TCP/IP messaging)
G. Web Services
A. Network Published Shared Variables

- Most likely used for tags
- Easy to use
- Disable network buffer for latest value use case
  - Or use Network Streams

A. Using Shared Variables Effectively

Initialize Shared Variables
Serialize Shared Variable Execution
Avoid Reading Stale Shared Variable Data

A. Using Shared Variables Effectively

Avoid Unnecessary Buffering
- Prevents data loss caused by temporary networking delays
- Doesn’t guarantee lossless data transfer
- Consumes more CPU and memory
- **On by default** with Network-Published Shared Variables
A. Location for Network-Published Shared Variables

Network-published shared variables can be located on the host PC or the target. Both locations have advantages:

<table>
<thead>
<tr>
<th>Target Advantages</th>
<th>Host PC Advantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reliability due to the stability of the RT target</td>
<td>Less memory and CPU usage on RT target</td>
</tr>
<tr>
<td></td>
<td>LabVIEW DSC features available</td>
</tr>
<tr>
<td></td>
<td>Other computers reading the data do not use resources on target</td>
</tr>
<tr>
<td></td>
<td>If more than 100+ distinct items</td>
</tr>
</tbody>
</table>

B. CVT Client Communication

- Share data between CVT on targets and CVT on host computers
- Extend the intertask communication advantages of CVTs to network communication
- CCC API
  - Allows client to bind its local CVT values to server CVT values
  - Transfers values between client and server CVT

C. Network Streams

Use cases:
- No data can be lost during transfer between RT target and host computer
- Transfer data from RT target to host computer for logging data to file
- Transfer data from RT target to host computer for data processing and analysis that requires more memory than the RT target has available
- Can also be used for updates
C. Network Streams - Disadvantages

- Not an open API - cannot be used with C
- Bit slower than what you can do with lower level TCP

D. Standard Protocols - TCP/UDP

- Communicate with hardware and software that does not support LabVIEW
- Only in string format
- Must manage buffers and connections

E. STM Reference Library

- Built on TCP
- Easily packages and parses data
- Hides the transport layer (TCP/IP, UDP, etc.) implementation details
- Minimizes network traffic by sending data only when it is needed
- Lends itself to communication with environments other than LabVIEW (C, C++, etc)
- Available at ni.com
F. AMC Reference Library

• Uses UDP
• Provides an easy to use messaging architecture for local and distributed LabVIEW applications.
• Can send messages within one process, different processes or different targets over a network
• Intended to be used for asynchronous messages which can occur at undefined intervals.
• Available at ni.com

G. Web Services

• Does not require any NI software on host computers
• Host computers view RT target data using a web browser
• Good for one-target-to-many-hosts use case

• Exchange data with VIs over a network
• Any HTTP-capable Web client can invoke VIs and exchange data using a URL and standard HTTP methods
## Rules of Thumb - Network Communication

<table>
<thead>
<tr>
<th>Use case</th>
<th>Examples</th>
<th>Protocols</th>
</tr>
</thead>
<tbody>
<tr>
<td>Latest value</td>
<td>• Host displays the most recent I/O values of RT target</td>
<td>• Network-published Shared Variables</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• CCC</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• UDP</td>
</tr>
<tr>
<td>Buffered values</td>
<td>• Transfer data to host for file logging</td>
<td>• Network Streams</td>
</tr>
<tr>
<td></td>
<td>• Data must not be lost during transfer</td>
<td>• TCP/IP</td>
</tr>
<tr>
<td>Message Based Communication</td>
<td>• Sending value changes from host to RT target</td>
<td>• Network Streams</td>
</tr>
<tr>
<td></td>
<td>• Synchronization</td>
<td>• STM</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• AMC</td>
</tr>
</tbody>
</table>

## Network Communication Comparison

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Common Use</th>
<th>Speed</th>
<th>Deterministic Read/Write</th>
<th>Deterministic Data Transfer</th>
<th>Advantages</th>
<th>Caveats</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network-Published Shared Variable</td>
<td>Latest value, windows</td>
<td>Fast</td>
<td>With RT FIFO enabled</td>
<td>No</td>
<td>Ease of programming</td>
<td>LV only</td>
</tr>
<tr>
<td>Network Streams</td>
<td>Data streaming</td>
<td>Faster</td>
<td>No</td>
<td>No</td>
<td>Built-in functions</td>
<td>LV only</td>
</tr>
<tr>
<td>TCP</td>
<td>Data streaming</td>
<td>Fastest</td>
<td>No</td>
<td>No</td>
<td>High transfer rates, standard protocol</td>
<td>String data</td>
</tr>
<tr>
<td>UDP</td>
<td>Broadcast latest values</td>
<td>Fastest</td>
<td>No</td>
<td>No</td>
<td>High transfer rates, standard protocol</td>
<td>String data, Lossy</td>
</tr>
</tbody>
</table>

## Choosing Transfer Types

<table>
<thead>
<tr>
<th>Examples</th>
<th>Message</th>
<th>Update</th>
<th>Stream</th>
<th>Variable (Tag)</th>
<th>Window (Circ. Buf)</th>
<th>Fundamental Features</th>
<th>Optional Features</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Examples</td>
<td>Exec-Action</td>
<td>Error</td>
<td>Waveform</td>
<td>Setpoint</td>
<td></td>
<td>History Graph</td>
<td>Group Mgmt</td>
<td>Low-Latency</td>
</tr>
<tr>
<td>Fundamental Features</td>
<td>Buffering</td>
<td>Block</td>
<td>Block</td>
<td>Nonhistorical</td>
<td></td>
<td>Multi-Read</td>
<td>Dynamic Lookup</td>
<td>Low-Latency</td>
</tr>
<tr>
<td>Optional Features</td>
<td>Ack</td>
<td>Broadcast</td>
<td>Multilayer Buffering</td>
<td>Dynamic Lookup</td>
<td></td>
<td>Group Mgmt</td>
<td>Group Mgmt</td>
<td>Low-Latency</td>
</tr>
<tr>
<td>Performance</td>
<td>Low-Latency</td>
<td>Low-Latency</td>
<td>High-Throughput</td>
<td>Low-Count</td>
<td></td>
<td>Latching</td>
<td>Latching</td>
<td>High-Count</td>
</tr>
</tbody>
</table>
### Choosing Transfer Types – Cross-Process

<table>
<thead>
<tr>
<th>Message</th>
<th>Update</th>
<th>Stream</th>
<th>Variable (Tag)</th>
<th>Window (Circ. Buffer)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Windows</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Queue (AMC)</td>
<td>• SE Queue</td>
<td>• Queue</td>
<td>• Local/Global Variable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Notifier</td>
<td></td>
<td>• SE Queue</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• FGV (CVT)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Shared Variable</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• DVR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Chart</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• FGV</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Queue (Get Queue Status)</td>
<td></td>
</tr>
<tr>
<td><strong>RT</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• Same as Windows</td>
<td>• Same as Windows</td>
<td>• Same as Windows</td>
<td>• FGV</td>
<td></td>
</tr>
<tr>
<td>• RT FIFO</td>
<td>• SE RT FIFO</td>
<td>• RT FIFO</td>
<td>• Queue (Get Queue Status)</td>
<td></td>
</tr>
<tr>
<td>• SV w RT FIFO</td>
<td></td>
<td>• SV w SE RT FIFO</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>FPGA</strong></td>
<td>• FIFO (2009)</td>
<td>• FIFO</td>
<td>• Local/Global Variable</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• FGV</td>
<td></td>
</tr>
</tbody>
</table>

### Choosing Data Transfer – Cross-System

<table>
<thead>
<tr>
<th>Message</th>
<th>Update</th>
<th>Stream</th>
<th>Variable (Tag)</th>
<th>Window (Circ. Buffer)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Network</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>• TCP (STM)</td>
<td>• UDP</td>
<td>• TCP</td>
<td>• Shared Variable (Buffered)</td>
<td></td>
</tr>
<tr>
<td>• Shared Variable</td>
<td>• Shared Variable</td>
<td>• Shared Variable</td>
<td>• CCCC</td>
<td></td>
</tr>
<tr>
<td>(Buffered, Blocking, Flushed)</td>
<td>(Blocking)</td>
<td>(Buffered, Blocking)</td>
<td>• Industrial Protocols</td>
<td></td>
</tr>
<tr>
<td>• Network Streams</td>
<td></td>
<td>• Network Streams</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(flushed)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Bus</strong></td>
<td>• DMA</td>
<td>• DMA</td>
<td>• Registers</td>
<td></td>
</tr>
<tr>
<td>• Register</td>
<td></td>
<td></td>
<td>• Registers</td>
<td></td>
</tr>
<tr>
<td>Handshaking</td>
<td></td>
<td></td>
<td>• Custom</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Quiz Question 1

Q: What might be subject to arbitration on an FPGA?
Quiz Answer 1

Q: What might be subject to arbitration on an FPGA?

A: Anything that is a shared resource: hardware I/O, FIFOs, non-entrant subVI’s, variables, etc.

Quiz Question 2

Q: What options do you have for FPGA inter-process communication that do not buffer data?

Quiz Answer 2

Q: What options do you have for FPGA inter-process communication that do not buffer data?

A: Variables, memory items.
Quiz Question 3

Q: What would you use to stream data between processes on an FPGA?

Quiz Answer 3

Q: What would you use to stream data between processes on an FPGA?

A: FIFOs are able to buffer data.

Quiz Question 4

What happens if you were to wire a timeout to your DMA FIFO read?
Quiz- Answer 4

Q: What happens if you were to wire a timeout to your DMA FIFO read?

A: Most likely, max your CPU.

Follow up question: what is the proper way to read from a DMA FIFO?

B. Modularity

I. Overview
II. Module Goals and Suggestions
III. OOP

I. Overview:
What is a module?

Module—(for purposes of this discussion) a functional grouping of code that may be called by one or more processes

• May be tied to data storage/transfer (FGVs)
• May be tied to synchronization (Mutex)
• Discussed in more detail later in the course
How do you Identify a Module?

1. Start with the analysis done for processes
2. Study the actions needed to accomplish each task
3. Make sure actions are as elemental as possible without implementing them
4. Group the actions into logical groups
5. Groups become modules, actions become commands/Vis

Tip: Modules and data items are usually nouns, commands are usually verbs

How do you Identify a Module? (cont)

Modules should
• High cohesion: sense of purpose
• Low coupling: Reliance on the calling process or other modules
  − Good for unit testing
  − Good for debugging
• Encapsulate similar operations

Examples of Common Modules

• Memory Maps
• Data Transfer Mechanisms
• Configuration Mechanisms
• File Managers
• Protocols
• Instrument Drivers

Modules are good places to get code reuse
How to Implement Modules in LabVIEW?

- SubVIs
- Grouped VIs
  - Virtual Folder
  - LVLIB
  - Packed Project Library*
- Action Engine (FGV)
- LVClass

II. Module Goals and Suggestions

A. Coupling
B. Cohesion
C. Encapsulation
D. Scalability
E. Testability

Coupling

Coupling measures the number of dependencies a system of modules contains.

The goal of any hierarchical architecture is to ensure that the implementation uses low or loose coupling:

- If one module does not function correctly, coupled modules may not function correctly
- Higher coupling makes debugging difficult
- Low coupling allows more changes in the code without breaking other modules
Coupling Suggestions

- Minimize shared resources
- Access shared resources at the highest level possible
- Use loose coupling communication mechanisms
  - Ex. Queues vs Variables
- Handle Errors Locally
- Use defensive programming practices
  - Self-verifying inputs, shared resources, and communication
  - Asserts

Cohesion

Cohesion measures the number of goals a specific module fulfills.

The goal of any hierarchical architecture is to ensure that the implementation has one goal per module—high cohesion.

- When a module tries to accomplish multiple goals, the block diagram becomes harder to read
- High cohesion decreases overall development time

Cohesion Suggestions

- Design before implementing
- Call other modules for any task outside of the module’s scope
- Create more modules when necessary

Pitfall: Use caution when creating communication paths to improve cohesion.
Encapsulation

Restrict access to the data inside the module

Encapsulation—Consolidation of data and methods into a module with restricted access to data

Encapsulation Suggestions

- Classes provide the best encapsulation
- Globals or Functional Global Variables in marked Private in a Library are a close second
- Avoid non-private variables for storing module data

Scalability

Scalability represents the ability of a module to adapt to changing requirements without a re-design.
Scalability Suggestions

- Always type define clusters and enumerated types!
- Avoid repeating code
- Use Conditional Disable Structures rather than code branches for
  - System Variants
  - Debug Hooks
- Consider flexible data types
  - Enums instead of booleans
  - Clusters instead of scalars
  - Strings or Variants add scalability but with complexity and error potential
- Plug-in-architectures and/or Classes greatly enhance scalability but add complexity

Testability

Scalability represents the ability of a module to be validated independent of the rest of the system

Considering test during design and development greatly expands the amount of unit testing that can be done
- Unit Testing is cheaper than integration or system testing
- Unit Testing is the easiest type of testing to automate

Testability Suggestions

- Low coupling modules are inherently better for testing
- Good encapsulation reduces the amount of integration testing needed
- Minimize communication paths within the module
- Provide simple setup and teardown hooks to put the module in a given state
III. LVOOP

Benefits
• Excellent scalability through inheritance
• Best method of encapsulation

Drawbacks
• Increases development effort
• Often increases system complexity
• Somewhat unproven on RT

Common Use Cases
• Hardware Abstraction Layers
• Highly reusable frameworks
• User-customizable software

C. Memory Management

I. Fixed-Size Memory Allocation
II. Dynamic Memory Allocation
   A. MemoryLeaks
   B. Over-Allocation
   C. Memory Fragmentation
III. Working with Fixed-Size Data
IV. Suggestions

Why care about memory management?
• Hardware Selection
• Determinism
• Performance
• Reliability
Question

Is this bad?

Answer

Yes. It is too big to fit on some RT targets.

But... it will not download and is therefore easy to detect and fix

Question

How about this? Is it bad?
Answer

Yes. It uses memory unnecessarily

But…

Answer (cont)

Fixed-size memory allocation will either fail on the first iteration or not at all.

Fixed-Size Memory Allocation

Two kinds of fixed-size memory
• Memory stored in the data space of the VI
• Fixed-sized data allocated at run-time

Fixed-sized memory allocation
• Fixed sized memory is allocated the first time it is used
• In a loop, LabVIEW re-uses the memory after the first iteration
• The memory manager is only engaged when initially allocating the memory
• Fixed sized memory is freed when the code goes idle
Fixed-Size Memory Suggestions

If code will not fit
• Check for memory copies
• Save data off to disk (watch for disk wear and fragmentation)
• Eliminate unnecessary drivers
• Choose a larger hardware target

Watch out for large fixed-size memory allocation in rarely called sections of code
• Example: Error handling code
• Use stress testing and test code coverage

Why is memory management important for RT?

Hardware Selection
Determinism
Performance
Reliability

Dynamic allocation
Dynamic Memory Allocation

If your program can run in the available memory then fixed-size memory allocation (i.e. memory copies, etc) is not particularly important
• Memory copies may still exacerbate the effects of dynamic memory allocation

Dynamic Memory Allocation

For the purposes of this presentation: Dynamic memory allocation is code that allocates memory most or all of the time it is called.

Initialization is generally considered a “freebie”
• Only runs once, so it will either be a problem or not

Impacts of Dynamic Memory Allocation

Determinism
Performance
Reliability
Impacts of Dynamic Memory Allocation

Determinism
• The memory manager is a shared resource
• Even with the mutex, execution time varies depending on
  – Amount of memory allocated
  – Current state of memory

Performance

Reliability
• If the memory manager cannot find a large enough contiguous segment to fill a request, it terminates the program
  – Leaks
  – Over-allocation
  – Memory fragmentation
Memory Leaks

A memory leak is an allocation of a resource which is never released. Common sources
• Calls to libraries with leaks
• Unclosed file handles
• Unclosed VI server references
• Unclosed driver handles
• Unclosed Shared Variable API references
• Unclosed TCP connection IDs or Listener IDs
Often results in leaking more than memory
• May cause subsequent calls to a resource to fail
Can take a long time to use the available memory

Over-allocation

Over-allocation occurs when the application tries to store too much data in RAM.
Typically the result of a queue or buffer without a fixed size
• If the reader is delayed, the buffer expands
Make sure to stress test system

Memory Fragmentation

LabVIEW Real-Time does not use virtual memory
When programs contain a mixture of dynamic allocations, memory can become fragmented.
Memory Fragmentation - Example

100 bytes of memory

Largest Contiguous = 100
Available memory = 100

Memory Fragmentation - Example

20 byte

Largest Contiguous = 80
Available Memory = 80

Memory Fragmentation - Example

20 byte 10 byte

Largest Contiguous = 70
Available Memory = 70
Memory Fragmentation - Example

Largest Contiguous = 70
Available Memory = 90

Memory Fragmentation - Example

Largest Contiguous = 70
Available Memory = 70

Memory Fragmentation - Example

Largest Contiguous = 40
Available Memory = 40
Memory Fragmentation - Example

Largest Contiguous = 20
Available Memory = 20

Memory Fragmentation - Example

Largest Contiguous = 30
Available Memory = 70

Memory Fragmentation - Example

Largest Contiguous = 30
Available Memory = 60
Memory Fragmentation - Example

Largest Contiguous = 30
Available Memory = 60

Memory Fragmentation (cont)

Memory fragmentation is very difficult to avoid
Rebooting will defragment memory
Larger allocations fragment memory faster and are more likely to fail

Working with Fixed-Sized Data

RT programs should attempt to minimize dynamic memory allocation (see impacts section)
The following are common sources of dynamic memory allocation
• Queues without fixed size
• Variable sized arrays (and Waveforms)
• Variable sized strings
• Variants
Working with Fixed-Sized Data: Queues

Queues have a dynamic number of elements by default. Even with a fixed number of elements, queues with variable sized data are still variable sized (also true of similar mechanisms such as Network Streams or Shared Variables). Pre-filling and flushing a variable sized queue in init may reduce allocations.

Working with Fixed-Sized Data: Arrays

Pre-allocate arrays whenever possible. More important for large arrays. Use RT FIFOs rather than queues to pass arrays as they allow you to specify the element size.

Working with Fixed-Sized Data: Strings

Strings are arrays of characters (U8)
- Usefull for pre-allocating strings
- Usefull for transmitting through RT FIFOs
Use Replace String Subset rather than concatenate string when possible
Most strings are relatively small in size
- Less important to fix in size than large arrays, etc
- Go after "low hanging fruit" first
Memory Management: Suggestions

Eliminate “Low-hanging fruit”
• Use fixed size for queues
• Pre-allocate memory when size is known
• Avoid memory copies on dynamic data
Watch out for large fixed-size memory allocations in rarely called code
Try to limit large dynamic memory allocations
• Small string manipulation, etc. will not cause significant fragmentation

Memory Management: Suggestions

Include memory monitoring code that reboots the target safely when memory is low
• Close files to prevent corruption
• Terminate communication to avoid conflicts
• Check out Fail-Safe Control Reference Design
Evaluate uptime needs vs flexibility and development time
• Systems which can handle regular reboots can use dynamic memory freely as long as they monitor

D. LabVIEW RT Execution Systems

Threads
Processing
LabVIEW
Execution Systems
Examples
Motivation and Goal

Motivation
Knowledge of LabVIEW execution systems may assist with design decisions and troubleshooting - especially when using the LabVIEW Real-Time Module and the Real-Time Execution Trace Toolkit.

Goal
Conceptually understand LabVIEW RT execution systems from the ground up.

Presentation Outline

Threads
- Program
- Process
- Thread
- Priority

Processing
- OS
- Multithreading
- Time Slices
- Multithreading
- Scheduler
- RTOS

LabVIEW
- VI Resource File
- Node
- Clump
- Compilation
- QElement

Execution System
- RunQ
- LabVIEW
- Exec
- Multithreaded LV
- LV Exec System
- Exec System Priority

Threads
Program

Instructions and data stored in a file

Why start here?
• We have to start somewhere
• Capabilities
• Features
• "My program crashed"

Process

A program that's running or ready to run

Context
• Address space
• Instructions
• CPU registers
• Etc.

Thread

An independent instruction stream within a process

Feature of the program

LabVIEW became multithreaded in version 5.0
Priority

Dictate which operations the designers feel are most important

Windows 7 Priorities

<table>
<thead>
<tr>
<th>Process Priority (Priority Class)</th>
<th>Thread Priority</th>
<th>Base Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>NORMAL_PRIORITY_CLASS</td>
<td>THREAD_PRIORITY_D dolphin</td>
<td>1</td>
</tr>
<tr>
<td>THREAD_PRIORITY_IDLE</td>
<td>THREAD_PRIORITY_LOWEST</td>
<td>6</td>
</tr>
<tr>
<td>THREAD_PRIORITY_BELOW_NORMAL</td>
<td>THREAD_PRIORITY_NORMAL</td>
<td>5</td>
</tr>
<tr>
<td>THREAD_PRIORITY_NORMAL</td>
<td>THREAD_PRIORITY_NORMAL</td>
<td>6</td>
</tr>
<tr>
<td>THREAD_PRIORITY_ABOVE_NORMAL</td>
<td>THREAD_PRIORITY_NORMAL</td>
<td>7</td>
</tr>
<tr>
<td>THREAD_PRIORITY_HIGHEST</td>
<td>THREAD_PRIORITY_NORMAL</td>
<td>8</td>
</tr>
<tr>
<td>THREAD_PRIORITY_TIME_CRITICAL</td>
<td>THREAD_PRIORITY_NORMAL</td>
<td>15</td>
</tr>
</tbody>
</table>

Operating System

Software that supports a computer's basic functions

Including
- Process management
- Interrupts
- Memory management
- File system
- Device drivers
- Networking
- Security
- I/O
Multitasking

The OS’s ability to divide processor time among processes that need it

Types
- Cooperative
- Preemptive

Time slice

The period of time a process is allowed to run

Example
- Quad core Intel Core i7 3.4 GHz
- 13.6K MIPS
- 15 ms per time slice
- 0.204 Mf per process time slice

Multithreading

The OS’s ability to support multi-threaded processes

Extends multitasking into processes

Benefits
- Separate multiple tasks
- Facilitate priority
- Perform (slow) operations concurrently
- Improve performance
- Perform the same task at the same time in more than one context
Scheduler

The OS component responsible for deciding whether the current process should continue running and, if not, which process should run next.

- Fair, efficient, high throughput, etc.

**Types**
- Round robin
  - Time slice
- Priority
  - Number

Real Time Operating System

A specially designed operating system

- Precise timing
- High degree of control of priority

High reliability

LabVIEW
Virtual Instrument Resource File (.vi)

A .vi contains source code, UI resources and executable code

Always present
- Data
- Compiled code

Loaded when used
- FP
- BD

Node

Objects with inputs and/or outputs that perform operations when a VI runs

Types of nodes
- Functions
- SubVIs
- Express VIs
- Structures
- Formula and Expression Nodes
- Property and Invoke Nodes
- Call by Reference Nodes
- Call Library Function Nodes

Clump

Group of nodes selected by the compiler with a fixed execution order

Characteristics
- No parallelism within a clump
- Clumps can be scheduled independently
- Parallelism between clumps
- Elegant extension of dataflow programming
Compilation

Process of generating machine instructions from G code

Steps
- Type propagation
- Dataflow intermediate representation (DFIR)
  - Decompositions and optimizations
  - Back-End Transforms
    - Clumper, Inplacer, Allocator
    - Code Generator → Intermediate Language (IL)
    - Low-Level Virtual Machine (LLVM)

The code generated presumes the existence of a LabVIEW execution system

Compilation

DFIR representation of a simple diagram

QElement

A record in the VI's data space corresponding to a clump

PC
- Address of the next instruction to execute

DS
- Points within a VI's data space
Execution System

RunQ

A collection of QElements representing code that’s ready* to run

RunQ QElements are independent
  • May depend on previously run code
  • Execution order does not change outcome of the program

curElem

Points to the currently running QElement

curElem is a variable inside the execution system
Exec()

The routine that runs the LabVIEW execution system

curElem = Dequeue (runQ)
Load curElem DS
Resume at curElem PC
If (return = NULL)
    Then discard curElem
Else /* returned new PC */
    Store new PC in curElem
    Enqueue (runQ, curElem)

Multithreaded LabVIEW

The program is split into a single UI thread and multiple execution threads

Exec ( )
• Multitasks your G code cooperatively

Execs
• Multitasked preemptively by the OS
LabVIEW does this automatically!

LabVIEW Execution Systems

Provide the programmer partitions for VIs to run independently from other VIs

Six Execution Systems
• Standard
• Instrument I/O
• Data Acquisition
• Other 1
• Other 2
• User Interface
Execution System Priority

Determines the execution thread of the VI

Six Available
• Background (lowest)
• Normal
• Above Normal
• High
• Time Critical
• Subroutine

RT Execution System Priority

Default thread configuration

By default, your VI runs in the Standard execution system at Normal priority in one of four threads

RT Execution System Priority (Partial List)

<table>
<thead>
<tr>
<th>ETS</th>
<th>VxWorks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Priority</td>
<td>Meaning</td>
</tr>
<tr>
<td>15, 0x9F</td>
<td>OS Max</td>
</tr>
<tr>
<td>15, 3</td>
<td>LV RT I/O Scan Engine</td>
</tr>
<tr>
<td>15</td>
<td>LV I/O Time Critical</td>
</tr>
<tr>
<td>10</td>
<td>Thread Loops</td>
</tr>
<tr>
<td>7</td>
<td>LV ≥ 8.5 Network Rx</td>
</tr>
<tr>
<td>2</td>
<td>LV I/O High</td>
</tr>
<tr>
<td>1</td>
<td>LV I/E Above Normal, Network Tx</td>
</tr>
<tr>
<td>0</td>
<td>LV I/E Normal, FTP Server, LogosXT, disk cache</td>
</tr>
<tr>
<td>-1</td>
<td>LV I/E Below Normal</td>
</tr>
<tr>
<td>-2</td>
<td>LV I/E Background</td>
</tr>
<tr>
<td>-16</td>
<td>OS Null thread</td>
</tr>
</tbody>
</table>
Summarizing LabVIEW RT Execution Systems

- During compilation, LabVIEW gathers independent block diagram nodes into clumps and inserts YieldIfNeeded statements into the graph (the overall compilation does a lot more than that)
- Clumps become QElements in the VI's data space
- QElements are assigned to an execution system based on the VI's Execution System and Priority properties
- By default, LabVIEW RT creates four threads per core, per execution system, per priority automatically, with no additional programming
- By default, VIs run in the Standard Normal threads
- LabVIEW cooperatively multitasks within the execution system using the statements inserted by the clumping algorithm during compilation
- The operating system preemptively multitasks between execution systems
- The G code determines when VIs are run
- The OS scheduler determines which thread will execute next and when its turn is over
- RTOS schedulers follow the programmer's priority strictly
- Things are not always as simple as they seem

Examples

Example - Background

Contents of subVI SubVI Execution Properties
Example A – Parallel VIs of Different Priority

Top level VI “A” configured to Normal priority.

The resulting trace from Top Level VI “A”.

Are there discrepancies in the trace from Top Level VI “A”?
Example A – Parallel VIs of Different Priority

Not unless you make assumptions about when the subVIs are loaded.

Example B – Parallel VIs of Different Priority

Top Level VI "B" configured to Normal priority.

Example B – Parallel VIs of Different Priority

The resulting trace from Top Level VI "B".
Example B – Parallel VIs of Different Priority

Zoomed in on the first 300 us, notice when the subVIs are loaded.

Example C – Parallel Loops with VIs of Different Priority

Top Level VI "C" configured to Normal priority.

Example C – Parallel Loops with VIs of Different Priority

The resulting trace from Top Level VI "C".
Example C – Parallel Loops with VIs of Different Priority

Zoomed into the trace from Top Level VI “C”.

Example D – Parallel Loops with VIs of Different Priorities

The resulting trace from Top Level VI “D” with initial sleep interval.

Example D – Parallel Loops with VIs of Different Priorities

Zoomed into the trace from Top Level VI “D” with initial sleep interval.
Maximizing Reliability

I. Reliability Overview
II. Failure Analysis
III. Mitigating Issues
   A. Error Handling
   B. Fail-Safe Design
   C. Redundancy
   D. Debugging and Testing
   E. System Monitoring

Reliability

• Broad category for multiple goals
  − Maximize System Availability
  − Maximize System Predictability
  − Reduce Incorrect Actions/Results (errors)
  − Fail Safely

Reliability and Availability

• Reliability - the probability of success
  \[
  \text{Reliability} = \frac{\text{# of Successes}}{\text{# of Attempts}}
  \]

• Availability - the amount of time the system is available for use
  \[
  \text{Availability} = \frac{\text{Uptime}}{\text{Uptime} + \text{Downtime}}
  \]
Example Requirements- Welding Robot

• System must be available for 16 hours at a time to accommodate 1st and 2nd shift manufacturing
• Scheduled maintenance can be performed during the 3rd shift
• Expected MTBF of 8,000 hours (1 failure per 2 years of use)

Failure Analysis- Potential Points of Failure

• Mechanical hardware
• Electronic hardware
• Software Failure
• User

Using COTS hardware mitigates failure

Electronic HW Failure

• Power
• Network Infrastructure
• Hard Drive
• Input/Output Devices
**Software Failure**

- Application-level bugs
- Interactions with other applications
- Drivers
- Operating System

**Understand the Role of the User**

- By placing more of the burden on the machine, you improve the reliability by requiring less of the operator

**Reduce Complexity**

\[
R_{1+2} = R_1 R_2 \\
(0.9)(0.9) = 0.81
\]
Failure Modes and Effects Analysis (FMEA)

• Proactive approach to problem solving (What it...?)
• Piece by piece study of potential failures and their implications for the system
• Risk Priority Number (RPN) helps prioritize problems
  – RPN = Severity x Problem x Detection

Mitigating Issues

A. Error Handling
B. Fail-Safe Design
C. Redundancy
D. Debugging and Testing
E. System Monitoring

A. Error Handling

Mission critical applications require more complete error handling than ordinary LabVIEW code
Comprehensive error handling usually involves tradeoffs
• Performance
• Development Time
Types of Error Handling

• Specific Error Handling
• Error Classification and Description
• Error Communication
• Central Error Handling
• Error Reporting
• Error Logging
• Throwing Errors

Specific Error Handling

Code called in specific locations to respond to specific error codes with an action.
Example actions
• Ignore
• Retry
• Correct
Specific Error Handling
Analyzing a Section of Code

• What could go wrong?
• Can I do anything about it (including logging/reporting)?
• Does responding to it require higher level code? (specific/central handling)?
• What will be the affect on subsequent code?
• What should be the affect on subsequent code?

Specific Error Handling
Guidelines

Avoid allowing errors to affect subsequent code
• Not all code behaves the same in the event of an error
• Better to specifically decide which code to execute using case structures

Example:
• As part of shutting down your application, you need to turn off three systems in order by writing false to three variables.

Specific Error Handling
Example
Specific Error Handling

Available Tools

Error palette
• General Error Handler
• Clear Errors
Structured Error Handler
• Specific Error Handler Express VI

Error Classification and Description

Organizes errors into classifications
Examples
• Warning
• Critical Error
• User Error
• Communications Error

Records additional information about error
Examples
• Timestamp
• Number of Occurrences
• Variable values
Error Classification and Description

Available Tools

SEH
• Classify/Declassify
• Notification provides timestamp and # of occurrences

Scan Engine Faults
• Provides priority (classification), timestamp and # of occurrences

Error Communication

Transfers an errors from their origin to a central location
Desirable features:
• Priority
• Filtering (count occurrences)
• Deterministic transfer for RT applications
Error Communication

Available Tools

Structured Error Handler
• Notification system provides priorities and filtering

Scan Engine Faults
• Provides priorities and filtering

Central Error Handling

Responds to classes of errors rather than specific codes
Takes asynchronous or system wide actions

Example Actions
• Logs errors to file
• Display prompts or sends messages to HMI
• Initiate system shutdown/reboot
• Apply Safe-state to outputs

Central Error Handling

State | Specific Handler | Code
---|---|---
State | Specific Handler | Code

State | Specific Handler | Code
---|---|---
State | Specific Handler | Code

Priority Queue

Central Handler
Central Error Handling
Available Tools

SEH
• Central Error Handler Template
Scan Engine Faults
• Fault Monitor.vi from examples is a good starting point

Error Reporting

Display a message on a user interface
Desirable features:
• Understandable messages
• Localization
• User response
• Adaption to error classification

Error Reporting
Available Tools

Error Handling Palette
• Simple Error Handler
• General Error Handler
Error Logging

Create a record of errors for future reference

Desirable features:

• Human readable
• High-performance
• Limited space or lifespan

Error Logging

Available Tools

Real-Time Error Log (RTEL)

Throwing Errors

Generate an error based on something that happened during execution

Often uses custom error codes/messages

• Usually defined using the error code editor/files
• Sometimes hard coded
  − Bad for localization, maintenance, and performance
  − Good for deployment
  − Use the <err> tag to avoid confusing with source.
Throwing Errors

Available Tools

• Error Code from Error Cluster
  – Has a significant performance impact
  – Avoid use on cRIO or similar targets
• Bundle by name

Which Errors To Handle?

• Only handle errors you can do something about
  – Ignore errors in error handling and shutdown
• Focus on mission-critical sections of code
• Document any assumptions or unhandled errors
• Use code reviews to achieve consensus on where to focus effort
• Use Fault-Insertion to identify error locations and codes
• Keep future changes in mind

Error Handling - Summary

Make an informed decision about the level of error handling in your application
• Identify system criticality
• Identify critical modules
• Identify specific error codes
Consider the different types of error handling and make use of the available tools for each
Search “SEH” on ni.com to find the Structured Error Handler
B. Fail-Safe Design on cRIO

cRIO is well suited for fail-safe designs
• I/O is routed through the FPGA, which has fewer potential failure modes than a software system
• FPGA and RT systems operate independently and can monitor each other
Uncontrollable failures are still possible
• Output Module Failure
• FPGA Hardware Failure
• Use redundancy or external safeguards when necessary

FPGA Safe States

FPGA state machine with one or more safe states
• Safe states control all outputs
• Leaving a safe state requires active confirmation from all system components
Normal operation state
• Normal operation of the machine
• Must be maintained each iteration by active confirmation

FPGA Safe States (cont.)

Primary safe state
• Default behavior when system boots
• Must not rely on RT system or any inputs
Other safe states
• Respond to specific failures
• May use select inputs only if they are actively confirmed
• Represents a partially working system or a system in a controlled shutdown
Fail-Safe Control Reference Design

Provides a framework for control systems which must behave predictably in the event of a hardware or software failure

Features:
- FPGA Safe State Machine
- Bi-directional RT<>FPGA watchdog

C. Redundancy

- Operate in parallel with equivalent functionality
- Greatly improve reliability and availability

Redundancy Reference Design
E. System Monitoring

- Good complement to error handling strategy and fail-safe design
- Monitoring Process
  - Memory
  - CPU Usage
  - System Temp
  - Disk Space
- Watchdogs
- Heartbeats

System Monitoring - Network Heartbeats

- Uses UDP to ensure presence of network connection
- Listener can perform some action if connection is lost
- Uni-directional
- Available on ni.com

Implementing Documentation

Types of Documentation
Designing Documentation
Developing User Documentation
Describing VIs, Controls, and Indicators
Creating Help Files
Types of Documentation

Developer
• Code Documentation (inline or printed)
• API Handbook
• Configuration Management Reports
• Traceability Matricies
• Design Docs

User
• Users Manual
• Wiring Guide
• Etc

A. Designing Documentation

LabVIEW includes features that simplify the process of creating documentation for the VIs you design
• History window—Use the History window to record changes to a VI as you make them
• Print dialog box—Use the Print dialog box to create printouts of the front panel, block diagram, connector pane, and descriptions of a VI

B. Developing User Documentation

• Systematically organize documentation: divide the help information into three categories—Concepts, procedures, and reference material
• Example—LabVIEW documentation
Developer Documentation

Documenting a library of VIs for other developers to use:
• Assume the audience has a working knowledge of LabVIEW
• Select File > Print to print VI documentation in a format almost identical to the format used in the VI and function reference information in the LabVIEW Help
• Use the Print dialog box to save the documentation to a file and to create documentation for multiple VIs simultaneously

Documenting a Design Pattern

• Embed state diagram into block diagram
• Comment design pattern; include reason for choice

C. Describing VIs, Controls, and Indicators

• Create VI Descriptions
  – Add VI descriptions for each VI
• Document Front Panels
• Create Control and Indicator Descriptions
  – Include a description for every control and indicator
    • Functionality
    • Data type
    • Range
    • Default Values
    • Behavior
D. Creating Help Files

• Use the Print dialog box to help create source material for your help file
• Use a help compiler to create a compiled help file (.chm) document
• Add help files to the Help menu of LabVIEW by placing them in the <help> directory
• Link to help files directly from a VI using the VI Properties » Documentation dialog box

Summary – Quiz

1. Which of the following items generates an HTML file documenting the icon and connector pane of a VI?
   a. LabVIEW Help
   b. Icon Editor
   c. VI Properties
   d. Print dialog box

2. Which type of information should a VI description NOT include?
   a. Overview of the VI
   b. Name of the developer
   c. Instructions for using the VI
   d. Descriptions of inputs and outputs

Summary – Quiz Answers

1. Which of the following items generates an HTML file documenting the icon and connector pane of a VI?
   a. LabVIEW Help
   b. Icon Editor
   c. VI Properties
   d. Print dialog box

2. Which type of information should a VI description NOT include?
   a. Overview of the VI
   b. Name of the developer
   c. Instructions for using the VI
   d. Descriptions of inputs and outputs
Refactoring Inherited Code

Inherited VIs may be poorly designed, making it difficult to add features later in the life of the VI.

Refactoring:
- Process of redesigning software to make it more readable and maintainable so that the cost of change does not increase over time.
- Changes the internal structure of a VI to make it more readable and maintainable, without changing its observable behavior.

Refactoring vs. Performance Optimization

- Changes that optimize the performance of a VI are not the same as refactoring.
- Refactoring specifically changes the internal structure of a VI to make it easier to read, understand, and maintain.

Refactoring a System

1. Start with a code review.
2. Create design documentation (communication diagram).
   (If you cannot create a communication diagram, start over).
3. Tackle major code review or design issues.
4. Refactor Individual VIs.
When to Give Up

- There is value in a VI that functions, even if the block diagram is not readable
- Good candidates for complete rewrites:
  - VIs that do not function
  - VIs that satisfy only a small portion of your needs
  - VIs that show significant design issues

Refactoring VIs

- When you refactor to improve the block diagram, make small cosmetic changes before tackling larger issues
- For example, it is easier to find duplicated code if the block diagram is well organized and the terminals are well labeled

Typical Refactoring Issues

The following issues can make it difficult to work with an inherited VI:
- Too disorganized
- Uses incorrect object names and poor icons
- Uses unnecessary logic
- Has duplicated logic
- Does not use data flow programming
- Has complicated algorithms
- Is too big
Typical Refactoring Issues - Disorganized

The block diagram is too disorganized

- Move objects within the block diagram
- Create subVIs for sections of the VI that are disorganized
- Place comments to improve readability

Typical Refactoring Issues

The block diagram uses incorrect object names and poor icons

Typical Refactoring Issues

The block diagram uses unnecessary logic
Typical Refactoring Issues

The block diagram uses duplicate logic

• Refactor the VI by creating a subVI for the duplicated logic

Typical Refactoring Issues

The block diagram does not use dataflow programming

• Replace Sequence structures with state machines if appropriate
• Delete local variables and wire directly to controls or indicators if possible

Typical Refactoring Issues

The block diagram has complicated algorithms

Simplify:
Typical Refactoring Issues

The block diagram is too big (larger than the screen size)
• Refactor the VI to make it smaller
• Create subVIs for sections of code within the block diagram

Debugging

LabVIEW debugging tools (probe, breakpoint, etc) work on RT, not on FPGA.
• Use controls/indicators on FPGA or emulate
You can remotely debug RT executables if you enable debugging when building
Debugging on RT generally does not work at 100% CPU (executes at ~Normal priority)

Lesson 3: Validation

TOPICS

I. Introduction
II. Types of Validation
III. Debugging and Testing Tools
IV. Customizing Requirements Gateway
I. Introduction

Validation – Confirmation that software satisfies user needs and fulfills its intended use

- Develop a level of confidence as a function of risk of failure
- Testing alone cannot verify that software is complete and correct
- Deliverables
  - Test results report
  - Code coverage report
  - Code review reports
  - Static Code Analysis reports

Why is Validation Important?

- Risk of failure
- Government regulations
  - FDA, FAA, DoD, DoE
- Quality initiatives
  - 6 Sigma, CMMI, IEEE, ISO

Last Minute Changes are Expensive

<table>
<thead>
<tr>
<th>Software Engineering Process Phase</th>
<th>Cost Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>1</td>
</tr>
<tr>
<td>Design</td>
<td>3-6x</td>
</tr>
<tr>
<td>Development</td>
<td>10x</td>
</tr>
<tr>
<td>Development – testing</td>
<td>15-40x</td>
</tr>
<tr>
<td>Validation</td>
<td>30-70x</td>
</tr>
<tr>
<td>Post-release</td>
<td>40-1000x</td>
</tr>
</tbody>
</table>
I. Types of Validation

A. Code Reviews

B. Static Code Analysis

C. Unit Testing

D. Integration Testing

E. System Testing

A. Code Reviews

During the code review, discuss quality goals and consider the following questions:
• What happens if a specific VI or function returns an error?
• Are there any race conditions?
• Is the block diagram implemented well?
• Are the algorithms efficient in terms of speed and/or memory usage?
• Is the block diagram easy to maintain?
• Does the developer adhere to established guidelines?

B. Static Code Analysis

– Any tool or method that has pre-established criteria by which it can compare source code to see if it meets standards for style, organization, and technique

• Can be performed without compiling or executing the code
• Used to find potential code problems before execution
• Examples of static code analysis tools
  – VI Metrics
  – VI Analyzer
C. Unit Testing

Verifying that a given set of inputs returns a known output

- Testing on a single VI
- May require simple setup and teardown to configure shared resources
- May also include wrappers for benchmarking

Unit Testing

Methods

- By hand
  - Least investment
  - Difficult to regression test
  - No documentation
- With Unit Test Framework
  - Generates great documentation
  - Ties to traceability
  - Good for regulated applications
  - Maintenance can be difficult
- With a custom tool
  - Most investment
  - Allows tool to match your needs

Black Box Testing

<table>
<thead>
<tr>
<th>INPUTS</th>
<th>Code Component</th>
<th>EXPECTED RESULTS</th>
<th>ACTUAL RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>200</td>
<td>-1</td>
</tr>
<tr>
<td>0</td>
<td>10</td>
<td>1999</td>
<td></td>
</tr>
<tr>
<td>+Infinity</td>
<td>+Infinity</td>
<td>2000</td>
<td></td>
</tr>
</tbody>
</table>

Pros:

- The tester does not need to know how it has been implemented
- Tester can still verify that the actual results = expected results
- Asses the behavior of the code component

Cons:

- The tester does not know if all statements have been exercised inside of the code component
White Box Testing

Pros:
• Designed to test paths inside of the code component
• Useful for integration testing to test the structure of the code

Cons:
• Not as useful for determining if requirements are met
• Test Designer must be knowledgeable of the source code

Boundary Conditions

• Requirements state a maximum value
• Sample test parameters:
  - maximum value + 1
  - maximum value - 1
  - maximum value

Hand Checking

If you know the value that a VI should return when run with a certain set of inputs, use those values as a test
• Sin(253.4569) = Difficult to know the answer
• Sin(3.14159) = Answer is known and exercises the VI the same way as 253.4569 because the values are of the same data type
Error Tests

- Test the error handling code—create test plans that force errors
- Verify that the proper errors are reported and recovered
- Verify that the error handling code handles errors gracefully

Unit Testing for RT

Run tests directly on the target
- Requires hardware
- Extends test time
- Good for testing
  - Communication
  - Time critical code
Run a subset of tests on Windows
- Does not work for RT specific functions
- Timing may be different
- Good for testing
  - Logic and algorithms
Unit Test Framework works for either

Unit Testing for FPGA

Compiling to unit test is infeasible
Use Emulation
- Allows you to define I/O simulation without modifying code
- Tools can allow cycle-accurate simulation
Run directly on host
- Allows you to use graphs
- Allows you to use UTF
D. Integration Testing

Integration testing consists of testing larger segments of code
• In RTs systems integration testing = testing a loop
• Generally requires custom code to simulate other parts of the system
  ~ Needs to run synchronously with code under test, so setup and teardown are insufficient
• Can still be executed through a test framework for documentation and traceability

Regression Testing

• Re-executing a subset of tests after an addition or change
• Set of tests should be chosen intelligently
• Much easier with automated testing

E. System Testing

Testing a finished or nearly finished system
• Acceptance (Functional) Test
• UI Testing
• Performance Testing
• Stress Testing
• Pilot Testing (Alpha/Beta)
System Testing

Stress testing is usually the most important for RT systems
• Test with extreme conditions
  − Temperature
  − Data volume/channel count
  − Network congestion
• Long-run test
  − Run with diagnostics
  − Use diagnostics to identify long-term trends (memory)

Debugging and Testing Tools

A. Console Out
B. Syslog
C. CPU and Memory Monitoring VIs
D. Distributed System Manager
E. VI Analyzer
F. Real-Time Execution Trace
G. Unit Test Framework

Console Out

Good for:
• Debugging headless systems
• Debugging crashes, boot sequence, OS calls, or non-LV code
• Diagnostics and defensive programming
Use RT Debug String to send messages to console
Enabling Console Out

Enable console out switch on cRIO
Connect a crossover serial cable
Use a terminal program (9600 baud)
  • No terminal in Windows 7 (Need to download)

Use web interface
  • Cannot debug boot sequence
  • Cannot enter commands (read only)

Syslog

Good for
  • Error/fault reporting
  • Debugging headless systems
  • Diagnostics and defensive programming

Industry standard
UDP Based
Reference Library on ni.com

Using Syslog for Error Reporting
Get CPU Loads and Get Memory Usage VIs

Good for

- Monitoring running applications
  - Fragmentation
  - Memory Leaks
  - Overloaded CPU
- Benchmarking systems
  - CPU is a better comparison than execution speed at the system level

Using Syslog to Create a Performance Profiler

Distributed System Manager

Good for

- Diagnostics on running applications
  - Memory and CPU
  - Scan engine monitoring, configuration, and forcing
- Benchmarking systems
Real-Time Execution Trace Tool

Good for
• Debugging memory allocations, shared resources
• Getting rough benchmarks on timing
• Getting a picture of the system execution
• Identifying the effect of non-application processes on the system

VI Analyzer

Good for
• Static code analysis
  – Testing for common mistakes
  – Complying to style and coding guidelines
• Regulated applications
Define a configuration that makes sense for your project/organization and save it
Create custom tests to check for past mistakes
Unit Test Framework

- Create test frameworks to functionally validate VIs
- Develop and execute tests from the LabVIEW Project Window
- Can test I/O as well as status of global variables and reference data
- Runs on Windows or Real-Time

Unit Test Framework Functionality

- Create multiple tests for a VI
- Each test can have multiple test cases
- Configure suites of tests for project hierarchy or run tests programmatically
- Filter tests based on priority
- Generate HTML, XML (ATML), or ASCII reports
- Identify and locate untested code using the code coverage metric

Code Coverage Example

1st Test Vector
- Block diagram, 2 Case diagrams executed. \( \frac{2 + 1}{6} = 50\% \) Code Coverage
Code Coverage Example

2nd Test Vector (aggregates covered code from 1st pass):
• Block diagram, 4 Case diagrams executed. \( \frac{4 + 1}{6} = 83.33\% \) Code Coverage

Automating the Unit Test Framework

• Run Tests from Project
• Run Tests from File
• Open Results Window
• Create Report

Debugging and Testing Tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Debug</th>
<th>Benchmark</th>
<th>Test</th>
<th>Monitor/Diagnostics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Console</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Sylog</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Distributed System Manager</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>CPU and Memory Usage VIs</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>Execution Trace</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>VI Analyzer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit Test Framework</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Summary – Quiz

Which type of test involves testing at each step of integration to verify that previously tested features still work?

a. Regression
b. Stress/load
c. Performance
d. Configuration

Summary – Quiz Answer

Which type of test involves testing at each step of integration to verify that previously tested features still work?

a. Regression
b. Stress/load
c. Performance
d. Configuration

Summary – Quiz

Which tool is best suited to interactively testing for coding and style issues?

a. Profile Performance and Memory Window
b. VI Metrics
c. VI Analyzer
d. None of the above
Summary – Quiz

Which tool is best suited to interactively testing for coding and style issues?

a. Profile Performance and Memory Window
b. VI Metrics
c. VI Analyzer
d. None of the above

Summary – Quiz

Identify whether each statement describes black box testing, white box testing or both:

a. The tester does not need to know how the code component was implemented — Black box testing
b. The tester can verify that the actual results match the expected results — Both
c. Designed to test paths inside of the code component — White box testing
d. Useful for integration testing to test the structure of the code — White box testing
Requirements Gateway Training Here