Testing Strategies for Asymmetric Environments

Jeffrey Fortin, Product Manager, Code Testing
<table>
<thead>
<tr>
<th></th>
<th>Agenda</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Asymmetric Environments</td>
</tr>
<tr>
<td>2.</td>
<td>Testing in an Asymmetric Environment</td>
</tr>
<tr>
<td>3.</td>
<td>Summary</td>
</tr>
<tr>
<td>4.</td>
<td>Questions</td>
</tr>
</tbody>
</table>
Software Based Systems are Evolving Rapidly

Asymmetric Environments

- Hundreds of files
- Thousands of Lines of Code
- Lots of “connections”
- Lots of safety concerns

- Thousands of files
- Millions of Lines of Code
- Few “connections”
- Few safety concerns
Asymmetric Environments

Today’s challenge: Meet quality objectives while controlling cost
Today’s challenge: Meet quality objectives while controlling cost

- Code size expanding
- Same regulatory requirements
- Same or higher quality expectations
- Same budgets and schedules

A strategy is needed ...
Asymmetric Environments

Environments and DevOps

Development
- New Code
- Unit Tests
- Static Analysis
- Code Coverage criteria met
- Development Host

Integration
- Latest code base
- Updated continuously

QA
- Integration Tests
- Regression Tests
- Updated Usually Overnight

Staging
- Simulate the production environment
- Perform tests that can’t be done sooner

Production
- End user is now your tester
- Bug reports
- External monitoring
1. Asymmetric Environments

2. Testing in an Asymmetric Environment

3. Summary

4. Questions
Testing in an Asymmetric Environment

**Multiple Configurations**

<table>
<thead>
<tr>
<th>Development</th>
<th>Integration</th>
<th>QA</th>
<th>Staging</th>
<th>Production</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Windows Host</td>
<td>• Linux Build Server</td>
<td>• Docker</td>
<td>• Sim/Stim</td>
<td>• Device Management</td>
</tr>
<tr>
<td>• Microsoft Visual Studio</td>
<td>• GCC Compiler</td>
<td>• Simulator</td>
<td>• HIL</td>
<td>• Embedded “Target”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Cross Compiler</td>
<td>• Signal Generators</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Network Simulation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>• Reference Hardware</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Typical Embedded Code Example

typedef unsigned long uint32_t;
/* Memory Mapped Hardware Registers with *special* properties */
uint32_t volatile REG1;
static inline uint32_t getREG1();

/*******************************************************************************/
/* take action based on the value in REG1 and REG2 */
int avoid_collision() {
    if ((REG1 & 0xFF00) & 0xFE00) {
        return 1;
    }
    return 0;
}

Can the code logic be tested early?
typedef unsigned long uint32_t;
/* Memory Mapped Hardware Registers with *special* properties */
uint32_t volatile REG1;
static inline uint32_t getREG1();

/*************************************************************/
/* Better implementation that allows for stubbing of the special registers */
inline uint32_t getREG1() {
    /* Using *inline* should give same performance as macros or inline register access */
    return REG1;
}
int avoid_collision_api() {
    /* Notice calls through stub-able accessor functions */
    if ((getREG1() & 0xFF00) & 0xFE00) {
        return 1;
    }
    return 0;
}
Simulating Hardware States

Temperature is a 2’s compliment binary
Using a stub routine, we can easily test for any temperature setting
We can also test for “impossible” bit combinations

<table>
<thead>
<tr>
<th>Register HW_TEMP_STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 1 0 9 8 7 6 5 4 3 2 1 0</td>
</tr>
<tr>
<td>B U S Y 0 0 0 0 0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>I L L L L L L F F F F F F</td>
</tr>
<tr>
<td>I L L L L L L F F F F F F</td>
</tr>
<tr>
<td>E E E E E E E E E E E E</td>
</tr>
<tr>
<td>S H O R T Unused</td>
</tr>
<tr>
<td>External Temperature</td>
</tr>
<tr>
<td>Internal Temperature</td>
</tr>
</tbody>
</table>
### Testing in an Asymmetric Environment

#### Injecting Faults

Bits 26-30 represent fault conditions
- Testing with stubs we can stimulate the code to respond to the fault
- We can even set multiple fault conditions

---

**Register HW_TEMP_STATUS**

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>3</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>U</td>
<td>S</td>
<td>Y</td>
<td>I</td>
<td>H</td>
<td>L</td>
<td>F</td>
<td>I</td>
<td>H</td>
<td>L</td>
<td>F</td>
<td>E</td>
<td>E</td>
<td>S</td>
<td>H</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td>L</td>
<td></td>
</tr>
</tbody>
</table>

- **Unused**
- **External Temperature**
- **Internal Temperature**
Testing in an Asymmetric Environment

Testing Example

Test Driver

Module Under Test

Call to getHW_TEMP_STATUS()

Real Functions

Dummy Functions (stubs)
Using Test Automation in the Development Environment

- Development Environment is resource rich
- The automation system can cycle through hundreds of test parameters and re-run tests on any code that has changed
- Tests can be setup to run on command by anyone
Collecting Testing Metrics in the Development Environment

**Metrics**

ISO-26262 (Automotive) ASIL D

<table>
<thead>
<tr>
<th>Unit</th>
<th>Subprogram</th>
<th>Complexity</th>
<th>Statements</th>
<th>Branches</th>
<th>Pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td></td>
<td>3 / 4 (75%)</td>
<td>10 / 17 (68%)</td>
<td>2 / 6 (33%)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>16 / 22 (72%)</td>
<td>4 / 6 (66%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>11 / 11 (100%)</td>
<td>5 / 5 (100%)</td>
<td>1 / 1 (100%)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>0 / 2 (0%)</td>
<td>0 / 1 (0%)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>0 / 7 (0%)</td>
<td>0 / 11 (0%)</td>
<td>0 / 3 (0%)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0 / 3 (0%)</td>
<td>0 / 5 (0%)</td>
<td>0 / 1 (0%)</td>
<td></td>
</tr>
</tbody>
</table>

**TOTALS** 6 16 30 / 49 (61%) 19 / 45 (42%) 3 / 11 (27%)

**GRAND TOTALS** 6 15 30 / 49 (61%) 19 / 45 (42%) 3 / 11 (27%)

- Get early indication of quality risks
- Testing coverage is a key metric
- Other metrics help to make better decisions
1. Asymetric Environments
2. Testing in an Asymetric Environment
3. Summary
4. Questions
Conclusion

- Leverage the Development Environment as much as possible
- Use techniques such as testing stubs, using multiple compiler configurations and test automation
- By doing so, embedded applications can be developed more efficiently and at lower risk than by only testing in the final Production Environment.
1. Asymmetric Environments

2. Testing in an Asymmetric Environment

3. Summary

4. Questions
Questions

Testing Strategies for Asymmetric Environments
For more information about Vector and our products please visit

www.vector.com

Author:
Fortin, Jeffrey
Vector