Developing Reliable Software Rapidly

David Kleidermacher
Green Hills Software

June 7, 2006

Background on GHS

- Leader in mission critical embedded software solutions for 24 years
- Compilers and RTOS (INTEGRITY) used for mission/safety critical systems
- First commercial IMA RTOS certified in US FAA DO-178B Level A system
- First commercial RTOS to undergo Common Criteria security evaluation at assurance level higher than EAL5 (EAL6 augmented and extended)
- Products used to run:
  - Automobiles (engine, drivetrain, infotainment)
  - Industrial Control Systems (water/chemical plants, analyzers)
  - Medical Devices (ventilators, aortic balloon pumps)
  - Avionics (flight controls, displays, weapons, aircraft engines)
  - Telecommunications (central office, optical switches)
- Yet rapid innovation
  - 6 major RTOS releases in 10 years (approx. 50 minor releases)

Introduction

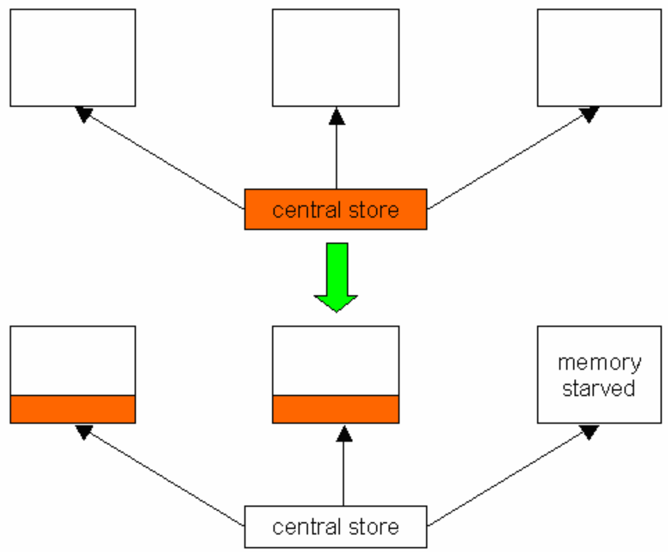
- **Rigorous software development process proven to increase reliability**
  - DO-178B
  - IEC-61508
  - ISO 9000
  - CMMI
  - etc.
- **But can stifle innovation**
  - DERs estimate 2 SLOC per person-day for DO-178B Level A process
  - A 5 KSLOC program takes 10 person-years to develop
- **Green Hills has developed and fine tuned a methodology to maximize reliability-efficiency for software development**
  - Top 20 guidance statements follow

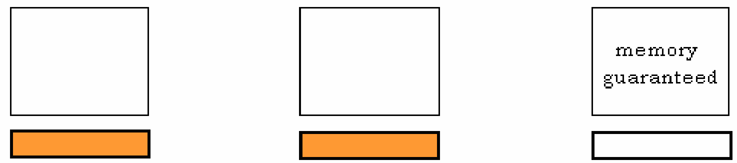
Partition Management

- **#1: Ensure that no single partition is larger than a single developer can fully comprehend.**
  - Avoid hacked features and guesswork in overly complicated, poorly understood partitions
  - Use simple, well-documented interfaces between partitions to minimize questions, confusions, and interdependencies
  - Refactor of legacy code may be initially expensive but worthwhile investment
- **#2: Ensure that every line of code has a partition manager.**
  - Keep partition manager list under CM
  - Only the PM is authorized to make or approve modifications
  - Avoid temptation to edit code when unqualified to do so

Partition Management

• **#3: If possible, use an operating system that employs true application partitioning.**





Peer Reviews

- **#4: Use asynchronous code reviews with email correspondence instead of face-to-face meetings.**
    - Partition management reduces peer review time
    - Avoid debates and grandstanding
- **#5: Use the CM system to automate enforcement of peer reviews for every modification to critical code.**
    - CM system rejects invalid userid for approver
- **#6: Require code reviews and other high integrity process controls only for critical partitions.**
    - If partitioned properly:
        - Only small portions of overall system are safety critical
        - Using a reduced assurance process for non-critical components does not affect safety/security

Build Cycle

• **#7: Use an autobuild system to quickly detect changes that break system builds.**

  • Dedicated computers for 24x7 builds of all valid configurations

  • Build failure causes automated notification (email) to build system manager and applicable partition managers (if practical)

  • Enables build errors to be detected and corrected before they affect the entire team

• **#8: Always ensure a developer has at least two development projects to work on at all times.**

  • Avoid coffee breaks while waiting for builds to complete

  • Train developers to proactively request work when there is no alternative project to work on while blocked

• **#9: Employ distributed builds to maximize computer utilization and improve developer efficiency.**

  • Take advantage of site's idle resources

Coding Standards

• **#10: Develop and deploy a coding standard that governs software development of all critical partitions.**

- Better code maintainability and testability
- Avoid dangerous constructs

• **#11: Maximize the use of automated verification of the coding standard; minimize the use of manually verified coding rules.**

- Manual human review is slow and error prone
- Ideally, the compiler enforces these during builds

• **#12: Prohibit compiler warnings.**

- Ignored warnings often the cause of subtle faults
- Ideally, use compiler option to force all warning to errors

Coding Standards

- **#13: Take advantage of the compiler's strictest language settings for safety and reliability.**
  - e.g. strict ANSI/ISO C/C++, Ravenscar Ada
  - MISRA C: *if (a = c)* vs. *if (a == c)*
- **#14: If a coding standard rule cannot be fully enforced at compile time, try to enforce it in a post-compile phase.**
  - Whole program static analyzers
    - file1: myfunc(NULL);
    - file2: void myfunc(int *p) { *p = 0; }

Symbolic Resolution

- **#15: Enforce valid resolution of code references to definitions**
  - File1:
```
void read_temp_sensor(float *ret) {
        *ret = *(float *)0xfeff0;
}
```
  - File2:
```
float poll_temperature(void) {
        extern float read_temp_sensor(void);
        return read_temp_sensor();
}
```
  - Detectable by whole program static analyzers or linker
  - Unintended resolution from libraries
    - Library uses print() internally but must be global
    - Program uses print() and gets Library definition instead of program definition
    - Use a tool to hide unexported library definitions

Complexity Control

• **#16: Use automated tools to enforce a complexity metric maximum, and ensure that this maximum is meaningful (such as a McCabe value of 20).**

- "metapartitioning"
- 1-10: simple
- 10-20: more complex, moderate risk
- 21-50: complex, high risk
- > 50: untestable
- Ideally, this is enforced by compiler at build time
- Carefully balance selection of aggressive limit with the cost to refactor legacy code that is too complex
    - Exceptions for legacy code must be approved by management

Testing/Verification

- **#17: The testing system should be running 24x7.**
  - Flaws introduced long ago are much harder to fix
  - Keeping the tests clean higher priority than development
  - New failures are likely recently introduced and thus easy to resolve
- **#18: The testing system should run on the development version as well as active shipping versions.**
  - Test fully integrated system as much as possible
  - Reduces testing time prior to a release
- **#19: The testing system should be able to effectively test a software project in less than one night.**
  - Otherwise becomes underutilized or completely irrelevant
  - Detect flaws quickly so they can be reproduced and fixed easily
  - Longer runs in the background

Testing/Verification

• **#20: It should be trivial to determine when a test run has succeeded or failed; a failed test should be trivial to reproduce.**

- Best: no output is a pass
- Voluminous output tends to get ignored
- Irreproducible failures tend to get ignored

Conclusion

- **High assurance processes are effective but inefficient**
- **GHS reliability-efficient methodologies proven-in-use for 24 years**
- **Emphasis on:**
  - **Reducing interdependencies**
  - **Apply rigor commensurate with criticality**
  - **Process automation**
  - **Rapid detection and notification of build/testing failures**