Issues and Guidelines of Software Design for Safety Critical Applications

Jeffrey M. Fornoff
US Army RDECOM-ARDEC
Picatinny Arsenal, NJ 07806-5000
(973) 724-3014

Providing America Advanced Armaments for Peace and War
Outline

• A Brief Overview - What is Software?
• Design and Engineering Issues, Best Practices
• Using Commercial Off-The-shelf Software (COTS)
• Testing, Testing, Testing
• Guideline Documents in Progress
Hardware & Software
Hardware & Software

• Simply Stated, Software:
  • Is just a series of binary information (instructions & data) stored in (hardware) memory
  • Does not work in a vacuum – hardware executes the instructions and modifies data
  • Cannot make hardware do anything it was not designed to do

• Hardware can be programmed many ways
  • Binary information clocked into memory by hand via switches
  • Assembly language (based on [micro]processor-specific architecture)
  • High-level language, providing abstract logic constructs and operations (independent of [micro]processor architecture)
  • Object-oriented language, providing Graphical User Interface (GUI)
Common Problems

• Common Problems with Software
  
  • Software performs as intended under normal or anticipated operating conditions, but may not behave properly when stressed
  
  • Possible causes
    
    • Inadequate design specifications
    • Inadequate, or poor testing procedures
    • Incomplete or poor implementation of design criteria
  
  • Some Solutions
    
    • Software design should be detailed enough to be unambiguous to programmers
    • Testing criteria should include bench checking and module unit testing
    • Use of proper software language and development tools help structure code
Common Problems

- Common Problems with Software (continued)

  - For given conditions, software experiences random or unexpected anomalies or spurious behavior

  - Possible causes
    - Input/output data received/generated at wrong time
      - External causes – interaction with other software/hardware components
      - Internal causes – poor interrupt handling
    - Operating System cannot guarantee timely processing of events

- Some Solutions
  - Provide timing specifications and specify design criteria to be followed
  - Design and implement tests to check for conformance to design specs
  - Use of proper operating system that is designed for real-time performance
Common Problems

• Common Problems with Software (continued)

  • Program execution skips over code or jumps to other modules unexpectedly

    • Possible causes

      • No logical path to reach unexecuted code (commonly called “dead code”)
      • Logic errors or invalid data cause improper branching to occur
      • Improper addressing used when branching to other code segments/modules

  • Some Solutions

    • Perform desk checking and/or use analysis tools to check code coverage
    • Verify logical program flow conforms to design specifications
    • Use appropriate programming language to prevent improper addressing
Common Problems

• Common Problems with Software (continued)

  • Program data values get modified or become corrupted for no apparent reason

  • Possible causes
    • Data definitions do not match between modules, globally or as parameters
    • Logic errors cause improper indexing into data areas and/or program areas
    • Operations on data values are of the wrong data type

• Some Solutions
  • Perform desk checking and/or use analysis tools to check data definitions
  • Verify addressing methods to data values
  • Use appropriate programming language to prevent improper addressing
Choose Carefully

• The Choice of Hardware Platform
  • Hardware must be capable of supporting physical and logical functions based on program requirements
  • Must have adequate performance, capacity, and reliability

• The Choice of Operating System
  • Must support given choice of hardware processor
  • Must support all communication with all required peripheral devices
  • Must support real-time, or near real-time processing as required
  • Should be certified for performance, reliability, security and safety
  • Source code should be made available (for testing) if not certified
  • Should be mature software with proven track record with similar applications
Choose Carefully

• Two major programming paradigms for flow control
  • Assembly and High Level Languages are programmatic – control flows from one statement to the next – easy to understand and analyze
  • Object Oriented Languages are user or interrupt driven – control is governed by what the user “clicks” next – much harder to analyze

• The Choice of Implementation Language(s)
  • Assembly language
    • When speed and timing are critical requirements
    • When I/O requirements cannot be met via the operating system
  • Drawbacks
    • Hardware dependent
    • Takes much more time to design, develop and test
Compilation Process

Source Code File(s)
(Human Readable)

Assembler/Compiler

Object Code File(s)
(non-executable/non-readable)

Linker

Executable Code File
(machine readable/executable)
Choose Carefully

• High level language (Ada, Fortran, PL/1, Pascal, Basic, Cobol)
  • When speed and timing are not critical requirements
  • General features – major advantages
    • Hardware and Operating System independent
    • Takes less time to design, develop and test
  • Some language dependent features
    • Provides a level of automatic program and data segmentation
    • Provides mechanisms for error and exception handling
    • Provides static and dynamic data type checking
  • Additional Analysis Required
    • Programming occurs far away from the hardware – the role of the compiler becomes more important as far as system analysis is concerned
    • Optimization and memory allocation schemes may be inappropriate for design requirements
Choose Carefully

• Object Oriented Languages (Visual Basic, Visual C++, Java)
  • Built-in Graphical User Interface (GUI)
  • General features – major advantages / disadvantages
    • Hardware independent – however may be Operating System dependent
    • Takes less time to design, develop, but much longer to test (and design tests)
  • Some language dependent features
    • Provides a level of automatic program and data segmentation
    • Provides mechanisms for error and exception handling
  • Additional Analysis Required
    • Programming occurs far away from the hardware – the role of the program environment becomes more important as far as analysis is concerned
    • Optimization and memory allocation schemes may be inappropriate for design requirements

• C-based languages are considered middle-level (not high-level) languages
Major Requirements

• Software engineering practices required to implement safe code
  • Modular programming - a module performs only a single simple task
    • Ideally, modules should only be one page long and easy to understand
    • Recursive modules strongly discouraged
  • Strong data types - data definitions are required in every module
    • Global data definitions are centrally defined - use of “include” statements
    • Local data definitions are enforced by design (if not by the language)
    • Static and dynamic parameter checking by design (if not by language)
  • Dynamic error checking - must be part of the design
    • Design requirements should specify the type of corrective action in case of program or data faults
    • If no design requirements exist, the program should be designed to gracefully exit
Flow Control

Simple State Machine Diagram

Providing America Advanced Armaments for Peace and War
Safety Critical Software

• Where safety critical software is concerned
  • If an operating system is needed, write your own
    • Source code can be fully tested as part of an integrated system
  • Do not utilize COTS
    • Unless the product provides source code that can be tested
  • Use a high-level language that enforces good programming practices
    • ADA, Pl/1, Pascal, FORTRAN 90 are some examples
  • Avoid Assembler, C, C++, Visual C++, Java (and other such variants)
    • Language definitions are contrary to good programming practices
Testing

• For Safety Critical Software:
  • Source code review should be performed by multiple people
    • Preferably by independent parties
  • Module level testing is performed
    • Test plans need to incorporate specific tests for safety critical code
  • Module integration testing
    • To insure software interfacing does not change intended behavior
  • System level integration testing
    • To insure hardware/software interfacing conforms to design specifications
  • Fault Insertion, bounds testing, stress testing, and regression testing are to be done at all testing levels and fully described in test plans
Pending Documents

• North Atlantic Treaty Organization (NATO)
  • Committee is drafting an AOP for safety critical software in fuzes

• Fuze Engineering Standardization Working Group (FESWG)
  • Drafting a set of guidelines for designing safety critical software

• Society of Automotive Engineers (SAE)
  • G-11 committee is working on possible standards for safety critical software

• Army Fuze Safety Review Board (AFSRB)
  • Considering drafting a position on issues of safety critical software