System Software Integration: An Expansive View

Steven P. Smith
Design of Embedded Systems
EE382V
Fall, 2009

Overview

• Some Definitions
• Introduction: The Expanding Challenge
• Phases of System Software Integration
• From Requirements to Software Components Identification
• Software Selection Issues during Architectural Design
• Unit-Level Integration and Software Performance Assessment
• Subsystem and Functional-Level Software Integration
• System-Level Software Integration and Testing
• Conclusions
Definitions

• System Integration: The task of creating a properly functioning system from its constituent components
  • Hardware
  • Firmware
  • Software

• System Hardware Integration
  • Are the components wired together correctly?

• System Software Integration
  • Typically assumes hardware integration is largely complete
  • The final step before acceptance testing and deployment

The System Engineering Process

Of course, iteration occurs at all levels and among most levels…
Software Integration in Embedded Systems: “The Good Old Days”

- Software developed internally
  - Design-specific software
  - No consideration given to software reuse
  - Direct access to software design, source code and developer
- Uni-processors predominate
  - No inter-processor and limited inter-process communications
- Small, simple real-time operating systems (RTOS)
  - Easy porting and configuration
  - Comparatively simple debugging and testing
  - Single-function systems


- Software components gathered from many sources
- Heterogeneous multi-processors
- Customized, configurable processors
  - Memory management units (MMUs)
- Mix of operating systems: RTOS and Linux
- Mix of functions and operating modes
  - Browser-based configuration
- Multiple debuggers, no interoperability among tools
- Enormously challenging testing implications
Implications for Software Integration of Embedded System Trends

- System software integration issues must be addressed early and continually throughout the design!
- Tool and software component selection must be made in the context of system-level design and development considerations.
  - Debugger interoperability increasingly critical
  - Integrated Development Environments (IDE) may have long learning curves
  - Compilers each have their own idiosyncrasies
  - Disparate operating systems don’t often play well together.
- No longer just a “back-end” task

Phases of the System Software Design and Integration Effort

- Identification of required software functions
  - Begins during requirements specification
  - Architecture decisions may add or remove requirements
- Mapping of required functions to candidate components
- Analysis of trade-offs in software component selection
- Initial software component selection or specification
- Performance analysis, verification
- Subsystem integration, performance analysis, verification
- System integration, performance analysis, verification
Identifying Required Software Functions

- Embedded system design often begins with an executable specification, or a high-level language (HLL) application.
  - Or, increasingly, two, or three…
  - Natural starting place for software function identification
- Initial hardware/software partitioning during architectural design defines required software functions.
  - This is a highly iterative process as performance bottlenecks and other design criteria come into sharper focus.
- Some software functions are not performance critical, but may demand significant flexibility.
  - E.g., the Internet refrigerator and its embedded http server
- End-user or OEM/VAR customization requirements also dictate required software functionality. Java, anyone?
System Software Elements

<table>
<thead>
<tr>
<th>System Application 1</th>
<th>System Application 2</th>
<th>System Application n</th>
<th>External Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hardware Abstraction Layer (HAL)</td>
<td>Custom IPC</td>
<td>Boot Loader(s)</td>
<td>Drivers</td>
</tr>
</tbody>
</table>

Interfaces to:
- Hardware Accelerators
- Real-time clocks
- IPC control hardware (e.g., semaphores)
- Boot hardware (flash)

Provides inter-processor messaging, synchronization, and notification functions

Identifying Candidate Software Components

- Map required software functions into specific candidate components
- Buy, adapt or develop?
  - Requires consideration of all design criteria, not to mention business issues
    - Difficult to evaluate early in the project
    - But also difficult to revisit later in the effort
- Operating system or executive selections are a key step.
  - A uniform operating system in a multi-processor SoC is extremely desirable, but not always feasible.
Operating Systems Selection Criteria

- Real-time capabilities
  - “Hard” real-time: guaranteed maximum latency for entering interrupt service routines (ISRs)
  - “Soft” real-time: no guarantees, but fairly quick response to real-time events (not for pacemakers, flight control, etc.)
- General-purpose features (e.g., file system, web server)
- Operating system acquisition and unit costs
- Inter-process and inter-processor communications support
- Reliability, Quality
- Resource requirements
  - Memory footprint of program and data
  - Boot, power-on-self-test (P.O.S.T.) mechanisms

Latency in Real-Time Applications

- Interrupt
- Interrupt Service Routine Starts
- Signal Operating System
- Control Task Awakens
- Control Calculations Complete
- HW Response
- ISR
- Context Switch
- Control Calculations
- Interrupt Latency
- Preemption Latency
- Minimum RT Period
Embedded Operating System Trends

- Linux - “Hard” real-time embedded Linux versions exist, but worst-case response times may still be too long.
  - Real-Time Application Interface (RTAI.org)
  - Linux Extensions for Real Time (LXRT) – built on RTAI
- Linux “on top” of a hard RTOS or kernel
  - Linux executes only when the RTOS is otherwise idle
  - Fine for configuration and other non-critical functions
  - Highly variable performance during normal system operation; Linux may be starved indefinitely by the RTOS.
- Growing support ecosystem for embedded Linux
  - Porting, configuring still a non-trivial effort

Real-Time Linux

- Linux with Real-Time Application Interface
  - RTAI is a hard real-time kernel that runs Linux in its idle loop
  - Real-time applications run in kernel mode
- Linux with RTAI and Linux Extensions for Real-Time
  - LXRT Extends RTAI to support Linux real-time user mode applications
    - Enables use of Linux memory management
    - Pairs a kernel mode RT task with the user mode task
  - Long paths in Linux kernel getting shorter and shorter
  - Real-time extensions have now merged with the core kernel
  - Tuning the kernel using scheduling policy selection
Real-Time Middleware

- CORBA - Common Object Request Broker Architecture
  - Standard mechanism for medium to coarse grain parallelism based on objects
    - Separation of object interface from implementation
    - Services available on a computing resource can be queried
    - Standardized argument marshalling, function calls, etc.
  - Platform and language independent
  - Object Management Group (omg.org)
    - Version 2.0 released in 2003
- CORBA Real-Time
  - Adds RT scheduling services to CORBA
  - Enables (but does not explicitly provide) load balancing

Embedded Software Component Sources
Detailed Embedded Software Component Selection Issues

• Develop internally or externally?
• Acceptable cost to develop or acquire?
• Source code or black-box, object-only module?
• Well-documented?
• Standard call specifications?
• Specific to a particular operating system or linker?
• Specific to a particular hardware component?
  • E.g., device drivers
• Sufficiently small code and data footprint?

Detailed Embedded Software Component Selection Issues (continued)

• Performance critical? Reliable?
• Optimized for this system?
• Configurable?
• Debugging information and tool support?
• Module-level tests available?
• Run-time dependence upon other modules?
• Predictable workload characteristics?
• Inter-process/inter-processor communications?
• Short learning curve?
Software Component Development and Acquisition

• Hardware abstraction layer (HAL) designed and developed early in process
  • Supports unit-level hardware debug
  • Defines virtual machine for application software
  • Enables bit-accurate C models to support performance modeling and software development
• Application-level software components often developed and partially debugged on general-purpose hardware before moving to target architecture
  • Using bit-accurate C HW models underneath HAL
• IP acquisition may be slow due to business issues

The Role of Regression Testing

• Regression testing is crucial at each level of software development and integration.
  • Unit, subsystem, and system level
  • Detect new design errors, deviations quickly: don’t go backwards
  • Must be run frequently (i.e., daily)
• Goal is to maintain conformance with the gold model throughout the design
• Comparing results at each level of design not easy
  • Behavioral don’t-cares versus explicit values at lower levels
  • Increasing time accuracy at lower levels also troublesome
**Unit-Level Hardware/Software Integration**

- Unit-level power-on initialization software
- Execute and profile individual software component on its target hardware or a model of same
  - Debugging hardware, HAL, and software simultaneously
  - First meaningful opportunity to assess performance
  - Iterate until software component is “completely” debugged
- Execute and profile all software components residing on a single target processor
  - Assess multi-tasking overhead
  - Local busy-waiting on hardware resources or hardware interrupts
  - Reassess resource requirements

**Subsystem Software Integration**

- Typically addresses specific functionality in comparative isolation
- May cover a single processor and the hardware resources it manages directly
- First opportunity to test and debug HAL with application software
- Provides basis for evaluating performance estimates at the subsystem level
  - Reflects overhead such as busy-waiting and interrupt servicing not reflected in application-only or unit-level testing
  - Enables initial programming and code-tuning for real-time execution
Subsystem Decomposition Example: Media Processor

- Enables specific function-level debug and testing
- Requires cleanly separable hardware components and interfaces
System Software Integration

- Full system and application-level integration and test
- Mixture of canned tests and real-world workloads
  - Extensive regression tests absolutely necessary
- Initially based on simulation or emulation platforms
  - Provides opportunity for early integration, detection of design defects
  - Too slow for long runs, operating system execution, etc.
- Culminates with execution on real silicon
- Transition to acceptance testing
  - All regression tests pass
  - Random, real workloads behave as expected

System Level Debug Focus

- Performance measurement and tuning
- Deadlock avoidance verification
  - Still not a proof
- Real-time schedule tuning
  - Refine interrupt versus polling tradeoffs and decisions
- Error detection and recovery
- Transition to acceptance testing
  - All regression tests pass
  - Random, real workloads behave as expected
Multiprocessor and Multitasking Debug

- Requires *cooperating* debug tool instances
  - No common API means a sole-source debugger (for now)
- Single processor breakpoints
  - Other processors may halt or continue execution on breakpoints, based on user preferences
  - Precise timing usually impossible, especially with multiple clock speeds/domains
- Synchronized single-stepping for repeatable results
- Multiple processor breakpoints
  - AND, OR, XOR, IF-THEN-ELSE conditionals combine single breakpoint triggers
  - Repeatability still difficult without synchronized single-stepping

Advanced Multiprocessing Debug Issues

- Watchpoints for data-triggered execution breaks
  - May require hardware assist
  - Multiple watchpoints
- Consistent user-interface
  - Falls out of sole-source multiprocessor debugger
  - Industry needs standardized debugger API, function set.
  - Vendors currently prefer closed environments, which may be fine until a processor is selected that is not supported by the debugger vendor.
- Adapting debugger to configurable or novel processor architectures not easy
Conclusions

- Software integration must be addressed at every phase of the design process
  - Definitely NOT merely a back-end task
  - May be key driver of system architectural design, processor selection, etc.
- Already often the single most costly aspect of system design, current trends will continue to amplify the importance of system integration issues, particularly for software.
  - Software components from a growing array of sources
  - Rapidly expanding number of components
  - Multiple operation modes exacerbate the testing task