SAE International™

SURFACE VEHICLE RECOMMENDED PRACTICE

SAE J2640 OCT2008

Issued 2006-04
Revised 2008-10

Superseding J2640 APR2006

(R) General Automotive Embedded Software Design Requirements

RATIONALE

Changes were made based on the feedback from the previously balloted document. Changes include consistent terminology usage and spelling mistakes in the following section;

3. Definitions
5.1.2 Use of Interrupts Caused by Module Input Signals (1-2)
5.1.3 Bounding Interrupt Overhead (1-3)
5.1.4 Use of Nested Interrupts (1-4)
5.2.1 Worst-Case CPU Load Measurement (2-1)
5.2.4 Use of Software Wait Loops (2-4)
5.3.1 Initialization of All Control Registers (3-1)
5.3.2 Refreshing Control Registers (3-2)
5.3.3 Clock Prescaler and Pll (3-3)
5.5.2 Unused Memory (5-2)
5.5.4.4 Data Integrity – Recovery (5-6)
5.5.4.6 Non-Volatile Memory Initialization (5-8)
Appendix A – Discussion on System Design Interactions

FOREWORD

NOTE: This document represents input from software experts in the automotive embedded software industry. Participants are from the OEMs, the automotive electronic module supplier base, related software suppliers, and consultants to the industry.

TABLE OF CONTENTS

TO PLACE A DOCUMENT ORDER: **Tel: 877-606-7323 (inside USA and Canada)**
Tel: 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: CustomerService@sae.org
SAE WEB ADDRESS: http://www.sae.org

1. SCOPE

The scope of this Recommended Practice encompasses the following objectives:

- Concentrate on general best practices for vehicular embedded software design.
- Establish programming language-independent best practices.
- Establish hardware/software interface best practices.
- Establish multi-threaded system best practices.
- Provide verification criteria to evaluate product compliance with this best practice.

2. REFERENCES

There are no referenced publications specified herein.

3. DEFINITIONS

3.1 Automotive Embedded Software

A set of instructions that controls special purpose hardware residing in a vehicle.

3.2 Corner Frequency

The corner frequency is defined as the frequency at which the output is attenuated by 3 decibels (1/2 the power) of the input signal for a single pole filter. Half the power is 1/Sqrt(2) or 0.707 gain.

3.3 Customer

The company that requested the embedded software.

3.4 Embedded Software Anomaly

Any unexpected behavior exhibited by the software regardless of whether the anomaly is induced by external module conditions (for example an electrical transient or a gamma ray that flips a bit in a control register) or internally caused by a hardware anomaly (for example, the watchdog timer fails to initialize) or internally caused by a software problem (for example, a stack overflow is not detected).

3.5 Embedded Software Robustness

A measure of the predictability of software's response to:

- Unusual and unexpected inputs and loads (CPU, interrupts, network...)
- Unanticipated events
- Hardware and software defects
- Hardware failures
- Errors introduced during system maintenance.

3.6 Hard Real-Time

A real-time system can be classified as either hard or soft. The distinction, however, is somewhat fuzzy. As illustrated in *Figure 1 - Real-Time Spectrum*, the meaning of real-time spans a spectrum. At one end of the spectrum is non-real-time, where there are no important deadlines (meaning all deadlines can be missed). The other end is hard real-time, where no deadlines can be missed. Every application falls somewhere between the two endpoints.

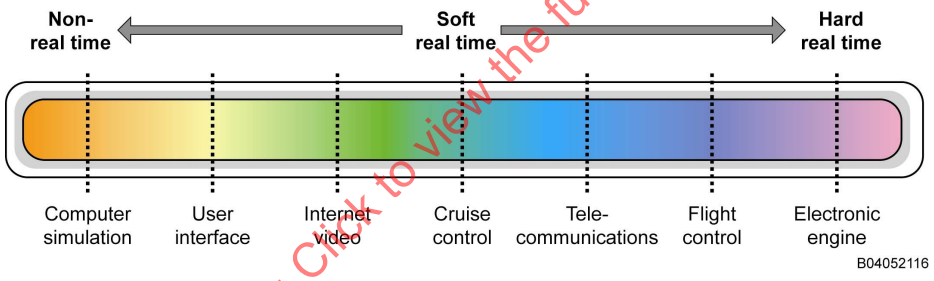

FIGURE 1 - REAL-TIME SPECTRUM

A hard real-time system is one in which one or more activities must never miss a deadline or timing constraint, otherwise the system fails. Failure includes damage to the equipment, major loss in revenues, or even injury or death.

3.7 LOS (Limited Operating Strategy)

Strategy for providing limited functionality under failure conditions.

3.8 Multi-Threading (Multi-Tasking)

Method whereby multiple tasks (or processes/threads) run on a single CPU. Only one tasks/process/threads is “running” at any one time. A “scheduler” switches between threads in order to simulate simultaneous thread execution. There are two basic types of multi-threading: *cooperative* and *preemptive*.

- Cooperative: the threads voluntarily give up control at thread defined points.
- Preemptive: control is taken from a thread at any time.

NOTE: The difference between task, processes and threads is not pertinent to this discussion.

3.9 Non-Preemptive System

A system where none of the tasks can preempt the execution of the current task and take control of the processor. A non-preemptive system doesn't use interrupts and relies entirely on polling. Any system that uses interrupts is preemptive system.

3.10 NVM (Non-Volatile Memory)

Programmatically alterable memory used to store data values when power is removed.

3.11 Overlapping Interrupts

An overlapping interrupt occurs when an interrupt is asserted prior to completion of processing of the previous assertion of the same interrupt. An overlapping interrupt can be caused by any of the following items:

- Processing of the interrupt takes longer than the minimum possible time between interrupts
- Processing of higher-priority interrupts causes the first interrupt assertion to experience latency greater than the minimum possible time between interrupts. Many advanced 32-bit CPU architectures have multiple levels of interrupts.
- Critical sections of software inhibit interrupts such that an interrupt can be asserted twice between the beginning of the critical section and the completion of processing the first assertion. This includes any latency issues caused by higher-priority interrupts.
- Poor filtering of the signal causing spurious interrupts (ringing, noise…)
- A combination of any or all the above issues.

Overlapping interrupts are a system issue and require good system design practices to ensure that they can never occur. Since interrupts are by nature asynchronous, they cannot be anticipated and worst-case analysis is usually required.

3.12 Preemptive System

A system where one or more tasks or interrupts can preempt the current task and take control of the processor. A preemptive system uses a preemptive operating system with multiple tasks or interrupts or both. A simple round-robin scheduler with interrupts is a preemptive system.

3.13 Resource Sharing

A resource is defined as any component within a computer based system that is used by the application to accomplish a task. Examples include the processor, memory, I/O, etc. Resources may be dedicated or shared. For example, the memory in which the application is running is typically dedicated, while global memory is shared. As soon as any resource is shared, the application must provide a means to manage this sharing. Resource sharing is the management of resources that are shared between different areas of the application to ensure data consistency, system integrity, and resource availability. Typically, the Operating System or Kernel has resource sharing management services, but it is left to the developer to determine when to use these services. In embedded systems, typical shared resources that must be managed include Non-Volatile RAM or EEPROM, communications channels (SAE J1850, CAN, LIN), global variables used by interrupts or multiple tasks, and I/O. Often overlooked resources are data items shared between tasks that take multiple memory cycles to access (bit-fields, large memory blocks…).

3.14 Sleep Mode

A special module-operating mode that uses a low quiescent current but is still able to respond to a limited set of inputs. Usually this only occurs when the ignition is off.

3.15 Soft Real-Time

A soft real-time system is one that has timing requirements, but occasionally missing them has negligible effects, as application requirements as a whole continue to be met.

3.16 Stop-Band Frequency

The frequency above which the attenuation of the signal could exceed a specified value. Typically, this is specified to guarantee that the signal does not produce a response. For example, TTLs would use an attenuation of 0.9 – for a 5 volt signal, this attenuation would guarantee the filter response in the Stop-Band frequency to be 0-0.5V.

3.17 Supplier

The company that provided the embedded software to the Customer.

3.18 Watchdog

A watchdog is a device that is used to protect a system from lockup causing the system to stop responding. Software errors are one of the major contributors to lockup, but other issues can also cause it. Normally the system must periodically send information to the watchdog (pet the dog) to keep it from rebooting/restarting the microcontroller or system.

4. OVERVIEW

4.1 Elements of a Requirement

Each requirement is composed of the following elements (some of which are optional):

- Requirement Title (m-n) – where m-n is the traceable requirement number.
  - "m" is the requirement category…
    - 1 is Interrupts
    - 2 is Timing Consistency
    - 3 is Data Consistency
    - 4 is Watchdog
    - 5 is Memory
- REQUIREMENT – Describes the requirement.
- CATEGORY – Type of requirements
- JUSTIFICATION – Supports the reasons why the requirement is needed.
- EXCEPTIONS – Lists any acceptable exceptions to the requirement.
- REVIEW QUESTIONS – These are typical questions that will be asked at the design review.

4.2 Requirement Heuristics

Embedded software is not robust when the software design and implementation have not addressed all conditions that might introduce an anomaly. The requirements in this document will reduce the incidence of these anomalies.

5. FUNDAMENTAL REQUIREMENTS

5.1 Interrupt Requirements

5.1.1 Use of Interrupts (1-1)

5.1.1.1 Requirement

Use of interrupts must be limited to the absolute minimal set needed to perform the required task(s).

5.1.1.2 Justification

Interrupts introduce preemptive resource sharing issues. The history of module development has demonstrated that preemptive resource sharing anomalies are a leading cause of recalls, product introduction delays, and slow product update turnarounds.

5.1.1.3 Review Question(s)

- Are any interrupts used by the software?
- What is the design of the interrupt structure?
- If interrupts are disabled or masked, can events be lost? If yes, explain what events are lost.
- Is interrupt overload behavior completely defined and fully understood?
- What is the expected subsystem behavior if the interrupt rate is exceeded?
- What is the worst-case CPU load due to all interrupts running at their worst possible frequency including the maximum latency and Interrupt working time (interrupt timing analysis)?
- How many milliseconds of the basic scheduler loop time can be consumed by these worst-case interrupts?
- For all external interrupts prove the design compensates for expected sources of input signal degradation. Sources of degradation include: time/usage, operational temperature range, mechanical vibration environment, EMC, harness routing, connectors, daughter cards, in-module jumpers, water, dust, salt, fretting corrosion, etc.

5.1.1.4 Recommendations and Suggestions

- Additional hardware must be seriously considered to reduce the number of interrupts.
- Polling should be used for event detection whenever possible.
- Interrupt Service Routines should be as small and fast as possible.
- Using Polling versus Interrupts is a hardware/software/complexity/cost issue and must be considered when designing a system and selecting a microcontroller.

5.1.2 Use of Interrupts Caused by Module Input Signals (1-2)

5.1.2.1 Requirement

Interrupts triggered by module input signals shall be used only when they are essential to satisfy hard real-time requirements, unless otherwise required by the Customer. All use of interrupts triggered by module input signals must be fully documented by the supplier and approved by the Customer.

5.1.2.2 Justification

Experience has shown that microcontroller interrupts can potentially consume 100% of the microcontroller processing bandwidth, producing software control anomalies, and cause unstable behavior. When the microcontroller interrupts come from a source external to the module, wire harness crosstalk, EMC, chassis vibration… can produce extraneous interrupts exacerbating the problem.

5.1.2.3 Review Question(s)

- Demonstrate how the microcontroller is protected from excessive interrupts caused by noise. Proof include: worst-case voltage analysis, worst-case filter corner frequency, and worst-case stop-band frequency.

5.1.2.4 Recommendations and Suggestions

If external interrupts are needed, then the microcontroller must be protected from excessive (noise) interrupts.

External interrupts can be used to wake the module from the Sleep mode. However, the input should be configured as a non-interrupt digital input once the module is out of Sleep mode. (i.e., Door Ajar switch is used to wake the module up, then it is configured to a digital input and the signal is debounced).

Failure Mode and Effects Analysis (FMEA) may indicate that a LOS is needed to accommodate overlapping interrupts (refer to 3.11 Definitions for the definition). LOS may be as simple as masking the interrupt for a limited time.

5.1.3 Bounding Interrupt Overhead (1-3)

5.1.3.1 Requirement

All source(s) of interrupts shall be bounded and accounted for during worst-case software analysis.

Interrupts occurring at the worst-case frequency must not exceed the CPU load design margin. These types of interrupts may include: Clock timers, Input captures, Output compares, CAN controller, SAE J1850 controller, UARTs, SPI communication, PCB mounted Hall-effect sensors,...

5.1.3.2 Justification

Experience with some module software implementations that use interrupts has demonstrated unstable and incorrect module behavior.

5.1.3.3 Review Question(s)

- List each interrupt. Include nominal frequency and worst-case frequency
- Describe the interrupt handling behavior for each internal module interrupt?
- For all interrupts running at their worst-case frequency, what is the CPU load?

5.1.4 Use of Nested Interrupts (1-4)

5.1.4.1 Requirement

Hierarchically prioritized pre-emptive (nested) interrupts shall be used only when they are essential to satisfy hard real-time requirements (refer to *EXCEPTIONS* below), unless otherwise required by the Customer. All use of nested interrupts must be fully documented by the supplier and approved by the Customer.

An ISR must not be interrupted and serviced by the same ISR.

5.1.4.2 Justification

Use of nested interrupts: increases complexity, reduces robustness, introduces more resource sharing issues, and it is more difficult to prove determinism.

5.1.4.3 Exception

The use of nested interrupts will be allowed only when;

1. The execution time of one interrupt is longer than the duration of an event handled by a different interrupt.

2. The data handled by the second IRS is unrecoverable,

For example, the CAN receive ISR execution time is longer that an RF (Radio Frequency) signal event duration.

5.1.4.4 Review Question(s)

- How are nested interrupts avoided?
- Prove that the system won't run out of resources (stack space...)
- Prove that the level of nesting is bounded and that the CPU load is bounded.

5.2 Timing Consistency Requirements

5.2.1 Worst-Case CPU Load Measurement (2-1)

5.2.1.1 Requirement

The system must continuously measure and report (via a diagnostic service) the worst-case CPU load over a Customer approved period.

5.2.1.2 Justification

This requirement is just one part of a set of requirements designed to guarantee execution deadline margin. Helps to identify when excessive noise (out of design) affects CPU load. Also, this measurement verifies the worst-case CPU load analysis.

5.2.1.3 Exception

None.

5.2.1.4 Review Question(s)

- Is the system event driven or priority based?
- Does the system have access to a diagnostic link? Was the worst-case analysis performed assuming that the link was active?
- What actions does the system take when idle time drops too low?

5.2.1.5 Recommendations and Suggestions

- The developer can create an idle task or instrument the operating system scheduler/idle loop.
- Either a network test tool or emulator (or even toggling a microcontroller output) can be used to read the actual worst-case measurement.
- Do not disable the measurement code for production release. Modifying the code after it was tested could have undesired side effects.
- If system is event driven or priority based, investigate deadline monotonic analysis.

5.2.2 Worst-Case CPU Load Margin (2-2)

5.2.2.1 Requirement

Supplier must prove that the software meets all Customer defined CPU load margins using worst-case (as defined by the Customer) analysis and empirically measuring the margin as required in requirement 2-1.

5.2.2.2 Justification

Prevents CPU overload from adversely impacting temporal determinism (necessary but not sufficient), accommodates feature creep, and ensures that most deadlines will be met, even under worst case loading.

Also allows each Customer to establish independent target levels for CPU load margins based on their own business needs should an Customer choose to do so.

5.2.2.3 Exceptions

- System initialization and shutdown.
- Special modes of operation such as: programming, diagnostics, manufacturing end-of-line...
- Other Customer approved exceptions.

5.2.2.4 Review Question(s)

- Are the timing requirements for all tasks and interrupts clearly defined in the requirements and design documentation?
- Have the worst-case load requirements been clearly documented by the Customer?
- Provide the worst-case CPU load margin analysis (analytical and empirical).

5.2.2.5 Recommendations and Suggestions

Load margins may be calculated in many ways based upon actual implementation. For a simple round robin loop or time sliced executive, this is a measurement of idle time divided by execution period.

For priority based scheduling (RTOS), use a methodology such as Rate Monotonic Analysis (RMA) or Deadline Monotonic Analysis (DMA). DMA is ideal where the deadline of a task execution is shorter than the task period. To prove the margin analytically, a dummy task representing the margin requirement must be assumed to occur at the Customer approved period (refer to requirement 2-1).

For example: a customer may require *Pre-production CPU Load Margin* >= 20% at sixteen months before production or first prototype before production and *Production CPU Load Margin* >= 10% at product launch

5.2.3 Software Measuring Timing (2-3)

5.2.3.1 Requirement

Counting instructions for timing purposes must only be used under these conditions:

1. Used for low-level hardware interfacing.
2. Used for very short times that are not effectively measured using normal software timers.
3. The reason must be clearly documented and code must be thoroughly commented. List all assumptions.
4. Must be clearly tagged in the code so that a file search will produce all instances of these low-level timing assumptions.
5. Reason must be presented in the design review. Include timing calculation and basis for calculation, microcontroller type, and clock speed...

5.2.3.2 Justification

The practice of timing by instruction counting makes the subsystem sensitive to hardware and software changes.

5.2.3.3 Exception

None.

5.2.3.4 Review Question(s)

- Describe all instances where you have used counting instructions for timing purposes.

5.2.4 Use of Software Wait Loops (2-4)

5.2.4.1 Requirement

Each software loop waiting for an event must have an alternate escape mechanism other than the watchdog timer reset.

5.2.4.2 Justification

Allows improved exception handling, promotes good design practice and control independence.

Using the watchdog as the only escape mechanism may cause the microcontroller to continually reset. When the microcontroller is continually reset, it can not report on what error is occurring.

5.2.4.3 Exception

Scheduler (main) loop or idle task.

5.2.4.4 Review Question(s)

- Identify each software loop waiting for an event (hardware, internal, external) and its escape mechanism.
- Provide the failure mode analysis on the appropriate course of action to each of these alternate loop escapes.

5.3 Microcontroller Consistency Requirements

Microcontrollers have many control registers that must be initialized upon reset. Industry experience has shown that many software developers do not realize that these values may change over time due to hardware and software anomalies. This forces the rigorous software developer to periodically refresh the control registers. The requirements in this section address microcontroller control registers and memory usages:

- Control register refresh
- Clock pre-scalar

5.3.1 Initialization of All Control Registers (3-1)

5.3.1.1 Requirement

Upon microcontroller reset, all control registers must be set to some known value - even if it is just the default value.

5.3.1.2 Justification

Never trust the microcontroller defaults. The microcontroller potentially resets from any number of different operating states/voltages. The microcontroller is only tested/verified for a limited number of operating states/voltages.

5.3.1.3 Exception

- Control registers that cannot affect the normal product operation do not need to be initialized.
- Control register that have been disabled by other control registers need not be initialized.

5.3.1.4 Review Question(s)

- List all control registers, their default values and what value they are set to after reset.
- List all control registers that are not initialized and your justification why.
- How familiarly are you with the selected microcontroller? Do you understand all the interactions that can occur between the uninitialized registers and normal product operation?

5.3.1.5 Recommendations and Suggestions

- Strongly recommend to initialize all peripheral in the silicon that are not connected to a microcontroller pin. Un-initialized ports will "float" (voltage will vary unpredictably) and may cause inadvertent interaction with A/D, Interrupts, clocks, PLL…

5.3.2 Refreshing Control Registers (3-2)

5.3.2.1 Requirement

All microcontroller control registers that can affect normal product operation shall be periodically refreshed. The rate(s) at which control registers are to be periodically refreshed shall be fully documented by the supplier and approved by the Customer. For a "sleep" module, upon wakeup all microcontroller control registers must be re-initialized.

5.3.2.2 Justification

Improves robustness by protecting control registers from EMI, degradation of data storage silicon, corruption via inadvertent modification, etc.

5.3.2.3 Exception

- A specific control register does not have to be refreshed if it could de-stabilize the microcontroller or cause undesired side effects.
- Control registers that can not affect the normal product operation do not need to be initialized.
- Control register that have been disabled by other control registers need not be initialized.

5.3.2.4 Review Question(s)

- Show the microcontroller-programming model and explain how each register is configured and how often the register is refreshed.
- How familiarly are you with the selected microcontroller? Do you understand all the interactions that can occur between the uninitialized registers and normal product operation?

5.3.2.5 Recommendations and Suggestions

- Strongly recommend to initialize all peripheral in the silicon that are not connected to a microcontroller pin. Un-initialized ports will "float" (voltage will vary unpredictably) and may cause inadvertent interaction with A/D, Interrupts, clocks, PLL…

5.3.3 Clock Prescaler and PLL (3-3)

5.3.3.1 Requirement

The microcontroller system clock pre-scalar and/or PLL shall not be dynamically changed.

Use of a second, backup/failsafe clock shall be restricted to a Limited Operation Strategy that clearly indicates (via diagnostic information) that there is a problem with the primary clock.

5.3.3.2 Justification

Simplifies testing. Prevents design errors. Lessons learned - previous use has caused a misinterpretation of input signal durations and incorrect functional operation. Also, backup clocks are not as accurate in timing or as stable and will result in customer perceivable differences in operation.

5.3.3.3 Exception

- The clock pre-scalar/PLL may be changed immediately before entering Sleep mode and immediately after exiting Sleep mode. However, the software design must account for the perceived difference of an event duration.
- The pre-scalar/PLL may also be changed as part of a Limited Operation Strategy in the event of a failure.

5.3.3.4 Review Question(s)

- Is the microcontroller system clock pre-scalar changed at any time other than immediately after reset or coming out of Sleep mode?
- If the pre-scalar is changed, present your design that addresses the difference in input signal duration.
- How does the system use a backup/failsafe clock (if at all)?
- Does the microcontroller immediately switch to high power mode on a wake-up event, if not, justify?

5.3.3.5 Recommendations and Suggestions

- In Sleep mode, the microcontroller can respond to external stimulus. However, these exception must be justified during a design review with the Customer.

5.4 Watchdog Requirements

5.4.1 Use of a Watchdog Timer (4-1)

5.4.1.1 Requirement

A watchdog mechanism must be used.

5.4.1.2 Justification

Allows recovery from microcontroller lock-ups due to software and/or hardware errors.

5.4.1.3 Review Question(s)

- What is the watchdog timeout period?
- Are there any conditions when the microcontroller is forced to reset via the watchdog? Explain.

5.4.2 Servicing the Watchdog Timer (4-2)

5.4.2.1 Requirement

The watchdog mechanism shall be serviced within one software location only, and not within an interrupt service routine. The Customer shall approve the watchdog duration and servicing location.

5.4.2.2 Justification

Ensures the watchdog isn't serviced within an unintended infinite loop. Also ensures that the watchdog timeout doesn't occur under minimum to maximum loading conditions as defined by the Customer. Ensures that the watchdog times out due to unexpected event (field failure, silicon failure, software error...).

5.4.2.3 Exception

- Startup, shutdown and Sleep modes.
- Special modes of operation such as: programming, diagnostics, manufacturing end-of-line...
- Other Customer approved exceptions.

5.4.2.4 Review Question(s)

- Has the Customer clearly defined the minimum and maximum loading conditions necessary for calculating the watchdog timeout period?
- Where is the watchdog timer serviced?

5.5 Memory Requirements

5.5.1 Run-Time Data Integrity Checking (5-1)

5.5.1.1 Requirement

Identify any dynamically calculated variables or inputs that could cause serious problems in the software (e.g. divide by zero, out of range index, invalid pointer, overflow, underflow...) These variables must protect against invalid values by using: redundant data, boundary checking, ...

5.5.1.2 Justification

Improves software robustness.

5.5.1.3 Exception

None.

5.5.1.4 Review Question(s)

- List all critical variables.
- What technique is used to detect that these variables are out-of-range/invalid?
- What are the observable effects of these out-of-bounds conditions?

5.5.2 Unused Memory (5-2)

5.5.2.1 Requirement

Execution of unused executable memory location(s) shall result in a microcontroller hardware reset. The worst-case period for reset must be reported to Customer.

5.5.2.2 Justification

Increases robustness. Since the program counter is outside of the program address range, a hardware reset is the safest recovery method because a hardware reset reinitializes all the registers to a predetermined state. If a hardware reset is not performed, then you are trusting that the initialization performed in the application will completely initialize all necessary hardware registers.

5.5.2.3 Review Question(s)

- Does the application initialize RAM on reset? Explain.
- What is done with unused executable memory (RAM/ROM/FLASH/ NVM)?

5.5.2.4 Recommendations and Suggestions

Memory space should be filled with the smallest size instructions that will cause the microcontroller to reset. One method is to fill unused memory with NOP instructions except the last instruction, which is a jump/branch to itself. This will cause the watchdog to fire. Do not transfer control to the reset vector because it doesn't reset the hardware (see Justification above).

5.5.3 Dynamic Memory Allocation (5-3)

5.5.3.1 Requirement

Any use of dynamic memory allocation (except that automatically performed by the compiler) must be approved by the Customer.

5.5.3.2 Justification

RAM is a precious resource on small microcontrollers. This requirement improves robustness by eliminating: memory fragmentation, memory leaks, out-of-memory condition, stack/heap collision, garbage collection and heap management.

5.5.3.3 Exception

Customer approved.

5.5.3.4 Review Question(s)

- Is dynamic memory allocation used? Why?
- Do you use the malloc() function?

5.5.4 Non-Volatile Memory Requirements

5.5.4.1 Requirement

The good news about NVM is that memory values are preserved between resets - the bad news is that memory values are preserved between resets. A microcontroller reset may temporarily "fix" the software problems (corrupted RAM, obscure infinite loops) in modules that do not have NVM.

NVM is special. Poorly managed NVM can wreak havoc even between resets. For example, care must be exercised to ensure that all NVM writing will finish even if a reset occurs. The following requirements address problems with NVM management:

- Data integrity
- Waiting on updates
- Life expectancy
- NVM initialization
- Guaranteed write cycle
- Memory access.

Non-Volatile Memory includes: Keep Alive RAM (KRAM), Static RAM (SRAM), EEPROM, EEPROM emulated in FLASH...

Keep Alive RAM is defined as RAM that has its own battery to keep the contents alive when power is removed from the ECU. SRAM or FLASH is not KRAM.

5.5.4.2 Data Integrity of Programmatically Modified Data Stored in Non-Volatile Memory (5-4)

5.5.4.2.1 Requirement

1. All programmatically modified data that is stored in non-volatile memory must also use a data integrity verification method (e.g., checksum, redundant data...).

2. The verification method must be executed at initialization and periodically afterwards.

3. If the verification method reports a failure, a defined containment strategy must be followed.

5.5.4.2.2 Justification

Non-volatile memory can become corrupted leading to faulty feature operation. This is especially true with modules that are never reset i.e., modules with Sleep mode that monitors inputs.

5.5.4.2.3 Review Question(s)

- Why is non-volatile memory used?
- How is non-volatile memory integrity verified?
- What is the corrupted non-volatile memory containment strategy?
- When/how often is the verification method executed?
- What happens when an unrecoverable error occurs in non-volatile memory? What default values are used?

5.5.4.2.4 Recommendations and Suggestions

Example Containment Strategy For Critical Data: you can use three different copies of the data and if they don't agree, update the errant copy with the value of two agreeing copies. If all copies disagree, then re-initialize.

5.5.4.3 Data Integrity - Update Period (5-5)

5.5.4.3.1 Requirement

Changes to values in non-volatile memory shall be updated within a maximum allowed time period specified by the Customer.

5.5.4.3.2 Justification

Prevent loss of updated information i.e., diagnostic information, memory seat and RKE rolling code data by forcing non-volatile memory to be updated as soon as possible instead of delaying until an operator induced event (i.e., ignition switch transitions to OFF). In other words, don't store all changes in "shadow RAM" for a long time waiting for an opportunity to update non-volatile memory.

5.5.4.3.3 Exception

Customer approved exception

5.5.4.3.4 Review Question(s)

- How long does it take to update non-volatile memory once a new value is ready for storage?

5.5.4.4 Data Integrity – Recovery (5-6)

5.5.4.4.1 Requirement

Regulatory or other performance critical configuration data (flags, other...) that are stored in non-volatile memory must use an encoding technique that allows recovery from a minimum of one-bit failures.

5.5.4.4.2 Justification

Regulatory and performance critical configurations must be immune to a single bit non-volatile memory corruption error.

5.5.4.4.3 Exception

None.

5.5.4.4.4 Review Question(s)

- List all of the Regulatory configuration data.
- List all performance critical configuration data. How did you determine what is "performance critical?"
- How does the encoding technique allow you to detect multi-bit errors?
- Explain the recovery process for errors in this data.
- What is the effect of multiple NVM data cell updates are required and

5.5.4.4.5 Recommendations and Suggestions

- ECC does not count as a redundant copy. All programmatically modifiable data values must have some sort of redundancy for recovery from failures. When a data value is incompletely updated due to a power loss/reset, the data value is corrupted. How does the software recover unless there is an alternate copy?

5.5.4.5 Life Expectancy (5-7)

5.5.4.5.1 Requirement

The number of writes/erase cycles to any non-volatile memory location over the life cycle of the product shall not exceed the maximum allowable number of worst-case write/erase cycles specified by the non-volatile memory manufacturer.

5.5.4.5.2 Justification

Prevents the degradation of feature performance over time. Reduces potential for data corruption. Warning - some NVM designs can result in the eventual complete failure of NVM after one cell fails.

5.5.4.5.3 Exception

Customer approved

5.5.4.5.4 Review Question(s)

- What is the maximum allowable number of worst-case write/erase cycles for a memory location as defined by the NVM manufacturer?
- Provide the data used to determine the expected number of write/erase cycles for each parameter over the life of the product.
- What is the failure mode of NVM once one cell fails?

5.5.4.6 Non-Volatile Memory Initialization (5-8)

5.5.4.6.1 Requirement

If NVM is used, the module supplier must initialize it to defined values before the module leaves the module manufacturing facility.

5.5.4.6.2 Justification

Design robustness. Don't rely on defaults from the NVM manufacturer.

5.5.4.6.3 Exception

Customer approved.

5.5.4.6.4 Review Question(s)

- At what point is the NVM initialized?
- What is your agreement with the NVM supplier regarding NVM initialization?
- What is the failure mode of the ECU when NVM is not initialized or initialized improperly?
- How do you guarantee that the NVM is initialized properly when the ECU is shipped to the Customer?

5.5.4.7 Avoidance of Non-Volatile Memory Corruption (5-9)

5.5.4.7.1 Requirement

Corruption of NVM due to power failure shall not require reinitialization of NVM from ROM values. This may imply that multiple NVM write/erase cycles be completed before power loss to the memory cells.

5.5.4.7.2 Justification

This avoids data corruption. Interrupting a write/erase cycle can damage the NVM.

5.5.4.7.3 Exception

Customer approved. It may be adequate to reinitialize NVM for ROM values in some instances.

5.5.4.7.4 Recommendations and Suggestions

- Add enough reserve capacitance to guarantee that a single NVM cell is updated before power loss. For example: When NVM uses triple redundancy and the first copy of NVM has been updated - if a power failure can corrupt the second cell then no two cells contain the same data necessary for recovery.
- Add enough reserve capacitance to guarantee that all NVM cells are updated before power loss

5.5.4.7.5 Review Question(s)

- Explain how software and hardware ensures that the NVM write/erase cycle is not interrupted in the event of a power failure.
- Explain how the NVM recovery procedures also address corruption due to power failure.

5.5.4.8 Guaranteed Non-Volatile Memory Access (5-10)

5.5.4.8.1 Requirement

If there is a potential for concurrent NVM access, then provision for NVM contention resolution shall be made.

5.5.4.8.2 Justification

Avoids data corruption due to concurrent access to/from NVM.

5.5.4.8.3 Review Question(s)

- Explain how NVM access contention is resolved.
- Is off-chip NVM used? If so, how do you resolve potential contention over the communication bus?

5.5.4.8.4 Recommendations and Suggestions

It is strongly recommended that an NVM manager be implemented to control access (read and write) to NVM.

5.6 Microcontroller Selection Requirements

Selecting the microcontroller is very important to meeting all the functional, operational, and software requirements. Problems arise when the microcontroller is undersized and printed circuit board changes are needed to upgrade the microcontroller.

5.6.1 Use of a Custom Microcontroller (6-1)

5.6.1.1 Requirement

Use of custom or early generation microcontrollers shall be avoided. A custom micro is defined as any micro where the module supplier purchases 50% or more of the total sales of that specific micro.

5.6.1.2 Justification

Custom microcontrollers increase risk when compared to standard microcontrollers due to various reasons. Validation, tool support and upgrade path are limited when compared with standard microcontrollers. Microcontroller manufacturers provide greater support (manufacturing and engineering) for standard devices and tend to extend/improve all standard devices while custom micros receive no attention.

5.6.1.3 Exception

Custom microcontrollers may only be used with customer approval – contingent upon a risk mitigation plan that addresses: costs, benefits, validation issues, tool support issues, critical path for micro delivery versus development timing, etc.

5.6.1.4 Review Question(s)

- Is this microcontroller an off-the-shelf product?
- Is this an early generation microcontroller?
- How does the selection of this microcontroller affect your tool set?

5.6.2 Determining Memory Size (6-2)

5.6.2.1 Requirement

The customer shall specify the required amounts of unused RAM and unused ROM/Flash at specific program milestones. If the code size exceeds any of the targets, then the customer must be immediately notified to help determine a risk mitigation plan.

5.6.2.2 Justification

Accommodates feature creep and improves memory margin management. Improved risk management. Can help avoid late board layout changes.

For example:

- Initial Design Estimate :: 50%
- At Design Validation :: 20%
- At Production Validation :: 10%

5.6.2.3 Exception

Customer approval is required for any deviation.

- A microcontroller upgrade path (having larger RAM, Flash and EEPROM) that uses the same footprint could also be considered.
- The customer could also specify the actual part.

5.6.2.4 Review Question(s)

- What is the expected memory size at production?
- What size RAM and ROM/Flash did you select for your micro?
- Is there a upgrade path for the selected micro that uses the same footprint?
  - Is it pin compatible?
  - Is it register compatible?
  - Is the errata different?

6. HARDWARE INTERFACE REQUIREMENTS (NEVER TRUST A RAW INPUT SIGNAL)

To be able to design software that interfaces to the hardware correctly, the software engineer must completely understand the characteristics of the hardware device and the input circuit design.

6.1 Discussion

Like the old military adage the battle plan never survives first contact with the enemy, software never survives first contact with hardware. This does not imply that software and hardware are at war with each other – it's more of a comment on the difficulty of getting hardware to work with software. And one of the most difficult areas to make robust is where software and hardware merge – Input Processing. In order to make input processes robust to all noise factors that can occur, you must first understand basic control theory, noise sources, hardware filters and debounce algorithms/software filtering.

6.1.1 Control Theory

Nyquist and Shannon were pioneers in the area of Information Theory. Specifically, Nyquist's Sampling theory requires the "sampling rate must be at least twice the highest frequency present in the sample in order to reconstruct the original signal." There are several implications that one should understand about this theorem:

- The signal must be sampled using a regular, periodic frequency.
- To detect a non-periodic event with duration N, the sample period must be at least N/2, but N/4 or faster would work better in a noisy environment.
- "Highest frequency present" can be derived from the Fourier series if one is truly interested in recreating the original signal. This implies that square waves, saw-tooth and other non-sinusoidal waves require very high sampling rates.
- The mathematics behind information theory requires regular, periodic sampling. This becomes very important with the performance of digital filtering and control systems (Z-transforms).
- Signal or noise frequencies that are higher than the sampling frequency can introduce aliasing. Refer to *Figure 2 - Aliasing Example: two different sinusoids that fit the same set of samples* for an example of an aliasing problem.

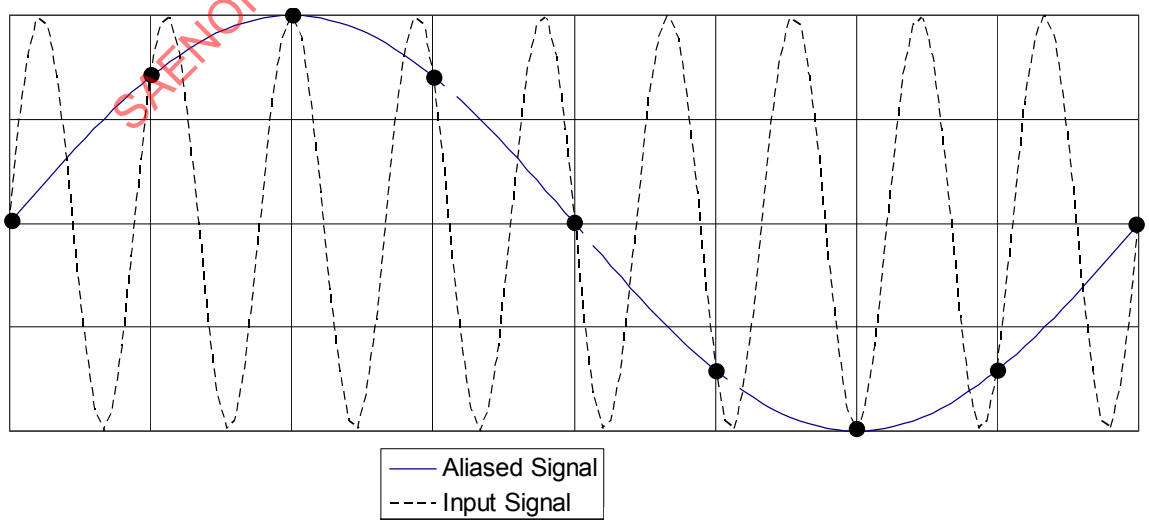

FIGURE 2 - ALIASING EXAMPLE: TWO DIFFERENT SINUSOIDS THAT FIT THE SAME SET OF SAMPLES

6.1.2 Noise Sources

The automotive electrical environment is extremely noisy. Most students and novices treat the electrical power as a DC source while in reality, it is an extremely noisy AC signal with a DC offset.

There are many sources of noise in the automobile, but regular periodic noise sources can cause the biggest problems. Here is a partial list of noise sources and their approximate frequencies.

TABLE 1 - NOISE SOURCES AND TYPICAL FREQUENCIES

Noise Source	Typical Frequency
Alternator regulation	90 – 120 Hz
Motor commutation	Variable
Body Vibration causing crosstalk or connector noise (poor connections cause periodic intermittents: harness rubbing, connector pushouts…)	10 – 30 Hz 50 Hz for the steering column
Crosstalk	Variable
Horn crosstalk (can be especially problematic in the steering column area when horn switch directly drives the horn)	100+ Hz
Current inrush	Non-periodic
Source impedance	Non-periodic
Switch and relay bounce	Millisecond duration, high frequency
External RF (AM/FM radio, military radar, TV, CB radio).	AM: 500–2000KHz, FM: 88–108MHz, Military Radar: 420MHz, TV: 54-72MHz and 76-88MHz and 174-216MHz and 470-698MHz, CB: 27MHz. Refer to NTIA frequency chart at http://www.ntia.doc.gov.
Fuel Injectors	Dependent on engine RPM and number of cylinders
High voltage ignition noise	Dependent on engine RPM and number of cylinders
Hybrid electric motors	Variable
Fuel pumps	PWM control, brush noise. The RPM pump design.
Automatic Transmission pressure solenoids	300Hz
LIN networks	10.4KHz (SAE J2602)
Climate Control Blowers	PWM control up to 20KHz Brush noise
PWM controlled lighting	80 – 140Hz

6.1.3 Debounce Algorithms

These do not compensate for just the theoretical bounce duration of a mechanical switch. They are used to filter out:

- Contact bounce duration
- Electrical noise
- switch contact timing differences for switches with multiple contacts
- make-before-break versus break-before-make
- settling time
- effects of aging
- effects of temperature
- effects of variation in load...

6.2 Digital Input Interfacing Requirements

6.2.1 Regular, Periodic Sampling (7-1)

6.2.1.1 Requirement

For each polled input, sample the input using a steady state frequency. This applies to: Single Digital Inputs, Multiple Digital Inputs, Continuous Analog Inputs, and Discrete Analog Inputs.

The Customer may specify a maximum sampling period.

6.2.1.2 Justification

Mechanical switches have a worst-case settling time. The debounce duration should be based on this settling time - typically several times this duration. The sampling period is based on the debounce duration - typically multiple samples. Care must be taken to avoid a sampling period that causes aliasing of low-frequency noise sources or that adversely affect overall response times. Refer to *Figure 3 - Switch Debounce and Sample Duration* and *Figure 4 - Sources of Phase Delays.*

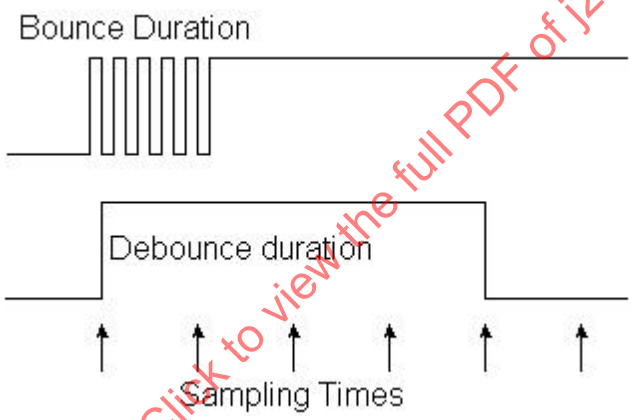

FIGURE 3 - SWITCH DEBOUNCE AND SAMPLE DURATION

6.2.1.3 Exception

Customer Specified.

6.2.1.4 Review Question(s)

- Explain general digital input operation/debounce strategy.
- For all inputs (except interrupts), prove the design compensates for expected sources of input signal degradation. Sources of degradation include: time/usage, operational temperature range, mechanical vibration environment, EMC, harness routing, connectors, daughter cards, in-module jumpers, water, dust, salt, etc.

6.2.2 Low Pass Input Filter (7-2)

6.2.2.1 Requirement

All inputs must have a low-pass filter (software or hardware).

NOTE: When you rely totally on a digital software filter to remove noise, you must compensate for potential aliasing by increasing your software sample rate. Ideally, the hardware filter should provide anti-aliasing for your selected software sample rate.

The customer may specify filter characteristics.

6.2.2.2 Justification

There is a lot of noise in the automotive electrical environment. Most non-pulse train signals do not have high frequency components that are needed in software. Using a low-pass filter removes all high frequency noise (and reduces aliasing) but it also introduces phase-lags, which may adversely affect response to changes in system inputs. So you need to balance between rejecting noise (and eliminating aliasing) versus response time. Again, aliasing doesn't have to be totally eliminated in hardware: it can be reduced with higher sampling rates in software.

The drawing below represents the relationship between the raw digital input signal, the effects of the low-pass filter and the final software debounced state. This illustrates the transition from an ideal signal to the real world noise environment. The low-pass filter introduces a time delay between the raw signal and the input to the microcontroller. The filter settling time must be considered in the software design if the digital circuit is polled. E.g., The power to the pull up resistor must be energized for six RC time constants to guarantee that it is stable before the microprocessor to trust it.

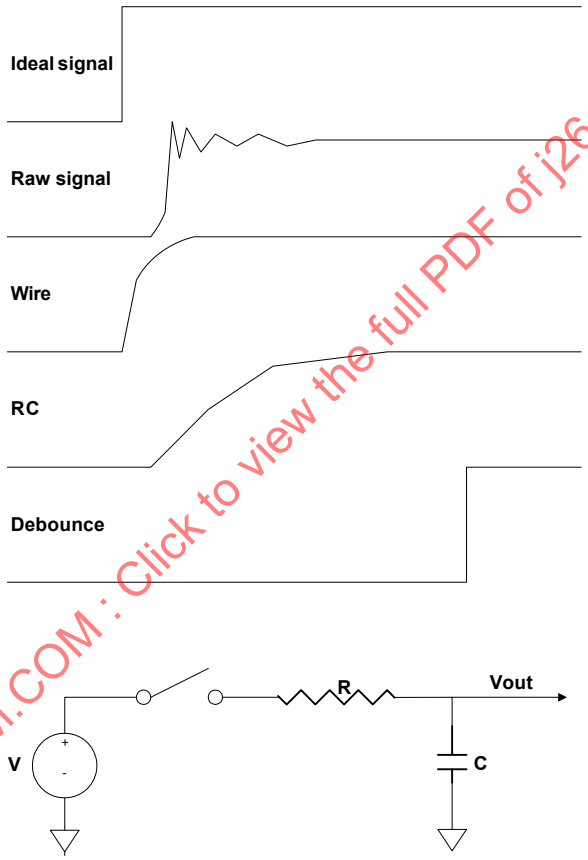

FIGURE 4 - SOURCES OF PHASE DELAYS

6.2.2.3 Review Question(s)

- List the corner frequency of each digital input into your microprocessor. Provide proof of this frequency using schematics and worst-case analysis.
- List the sample rate or each input.
- List frequency above which aliasing can occur.

6.2.3 Digital Input Debounce (7-3)

6.2.3.1 Requirement

All digital inputs must be debounced with multiple samples. Signals that travel across the vehicle harness need longer debounce times/more samples than locally generated signals (e.g., hall-effect sensors mounted on the same board as the microcontroller).

The customer may specify debounce requirements.

NOTE: Voltage fluctuations can adversely affect debouncing.

6.2.3.2 Justification

Mechanical switches/relays have a worst-case settling time. The debounce duration should be based on this settling time - typically several times this duration. The sampling period is based on the debounce duration - typically multiple samples. Care must be taken to avoid a sampling period that causes aliasing of low-frequency noise sources or that adversely affect overall response times. For more information, see general debounce discussion above.

6.2.3.3 Exception

- Interrupts - refer to the requirements in Section 5.1.

6.2.3.4 Review Question(s)

- Explain general digital input operation/debounce strategy.
- For all inputs (except interrupts), prove the design compensates for expected sources of input signal degradation. Sources of degradation include: time/usage, operational temperature range, mechanical vibration environment, EMC, harness routing, connectors, daughter cards, in-module jumpers, water, dust, salt, etc.

6.3 Analog Input Interfacing Requirements

There are no requirements here, just some discussions on some of the issues with using A/D inputs.

Since Battery Voltage has a lot of noise components associated with it, the noise sources can affect the stability of an analog signal. Hardware characteristics of the microprocessor can also adversely affect the resulting analog value. These discussion points will hopefully allow you to evaluate the robustness of a design.

6.3.1 Filter Matching - Just To Get One Valid Sample

6.3.1.1 Requirement

None for this release.

6.3.1.2 Discussion

Different filters cause different responses - specifically phase delays. When an A/D signal is referenced to battery voltage, two different input filters are used: one for the measured signal and one for the reference signal (battery voltage).