QUALITY AND RELIABILITY ENGINEERING INTERNATIONAL, VOL. 11, 403–409 (1995)

SOFTWARE SAFETY INITIATIVES

SAMUEL J. KEENE
Storage Technology Corporation, 2270 South 88th Street, Louisville, Colorado 80028, U.S.A.

SUMMARY

This paper addresses system safety aspects of product development with primary application to the U.S.A. Department of Defense (DoD) development process. Software is recognized as an increasingly important element of system reliability and safety. This is due to two reasons: first, the amount of code used in products is growing at an exponential rate over time, and secondly, the criticality of the software encoded functions is increasing dramatically. Emphasis has correspondingly increased on software and system aspects of safety. This is in addition to the more traditional elements of hardware product safety. Design guides are offered that help ensure development of both reliable and safe software.

KEY WORDS: system safety; software

INTRODUCTION

Over the past eight or so years, the U.S.A. Department of Defense (DoD) has increased its emphasis on system safety. Much of this increased concern is a result of the Space Shuttle disaster. But technical concerns in other areas have caused a focus on system safety as well: the Vincennes disaster; the apparent unmanageableness of the Strategic Defense Initiative (SDI) programme; medical calamities caused by X-ray treatment; and errant control systems of jet aircraft, to name a few. The problems cited have all been attributed to software errors.[1–5] That is, these were system problems that could have been precluded by adequate software development safeguards. U.S.A. military specifications such as 882C, 'System Safety Program Requirements' now specifically address software and systems and require software to be systematically analysed to mitigate potential system-safety problems. There are other specifications dealing with these same system safety concerns. Some of these are: IEC 65A-WG9, 'Functional Safety of Programmable Electronic Systems', 'IEEE Standard for Software Safety Plans' (1994); and Interim Defence Standard OO-SS (Part 1), 'The Procurement of Safety Crifical Software in Defence Equipment', U.K. Ministry of Defence, Glasgow, U.K. April 1991.

STRATEGIES

Accident risk assessment reports

Potential system safety problems have to be corrected by the contractor, and the residual hazards should be reported to the contracting agency of the government. The vehicle to document these hazards is either an Accident Risk Assessment Report (ARAR) or a Safety Assessment Report (SAR). Both disclose the potential safety problems, the likelihood of their occurrence, and the severity, if they occur. When the contracting agency accepts this report, it is acknowledging the problems disclosed and accepting that risk. (Residual risk will always exist).

The ARAR accomplishes three tasks:

1. It documents a systematic analysis of the safety aspects of the design.
2. It prioritizes potential problems to be mitigated.
3. It can exonerate the contractor from liability with respect to the safety limitations disclosed to the customer. In jurisprudence this is termed the 'contractor's defence'.

The contractor's defence comes from recognizing that the government is involved in the specification and design of many products they buy.[6] A landmark case, McKay vs. Rockwell International, 704 F2D 444 (Ca 9 1983), Sert. denied 464 US1043 (1984), overturned an initial judgment in favour of the widows of two U.S. Navy pilots who were killed while ejecting from their aircraft during training missions. McKay specified four requirements for the contractor defence protection to apply:

1. The U.S. must be immune from suit.
2. The government contractor must prove that the U.S. established or approved the specifications.
3. The equipment must meet specifications.
4. The government contractor must have warned the U.S. about obvious errors in the specifications or about dangers involved in the use of the equipment.

The best protection over all of these points is the delivery and approval or acceptance of a comprehensive ARAR. This legal decision has been

CCC 0748–8017/95/060403–07

Received 20 July 1995
Revised 17 September 1995

strengthened by later rulings supporting the McKay ruling. The U.S. Supreme Court upheld the issue of contractor's defence again in Boyle vs. United Technologies 108SCT. 2510 (1988).

Software safety

The recent DOD initiatives relate to system and software safety. MIL-STD-882C tasks show this. Software has traditionally been thought to be safe and failure free. But software is the 'brains' of the system and directs the system on how to react to any given situation or stimulus. The system inputs, creating the problem, may not have been anticipated in the design, or they may be out-of-bounds.

The safety question is: how benignly does the software behave given these conditions or any others that it may experience in its lifetime? The rest of this paper discusses what can be done to build and assure safe software.

Ultrasafe software practices

Programmers have taken several initiatives for developing ultrasafe software. Some of these are cited here, as background. The first three strategies apply to particularly critical missions: Space Shuttle, Federal Aviation Administration (FAA), and nuclear-reactor control. The author does not think that they routinely apply across DoD programs but they are offered to provide some insight into what can be done.

Diversity. The Space Shuttle code was developed by two independent develoopment teams using different programming languages.[7,8] Schemes were then built in to compare results or for majority voting on logical decisions before progressing in software decisions.

Diversity also comes from having an independent test team apart from the development team.[9] Design diversity is expensive. It has been cited as costing as much as $1000 per line of code to develop. This expense, of course, covers all of the associated quality and support activities to develop extremely reliable code. The Space Shuttle code (which was developed by IBM) never had a software failure during a mission. During all of its operational testing, it demonstrated a fault rate of 0.11 faults per KSLOC at the time of code release to the field.[10] A typically excellent fault rate (at time of shipment) is of the order of 1 fault per KSLOC. To get 1 fault per KSLOC requires extraordinary quality emphasis during design and test. Another such exceptional initiative is fault seeding.

Fault seeding. In fault seeding, errors or faults are inserted into the system, and the system is exercised.[11] The purpose is to see how the system handles or reacts to these embedded faults. These faults can be inserted into the code, or the messages, or the hardware. This process tends to quickly identify the limitations and susceptibilities of the deisgn, and it can rapidly identify faults.

This procedure uniquely uncovers bugs that are difficult to find any other way. These bugs stem from the recognition that all code can potentially be safety-critical code. Any code could potentially overlay some truly 'critical code', corrupting its function. Or, the benign code could hang up, taking too much system resource and causing critical queues to erupt and time-out the system activity. Fault seeding can help to identify such potential exposures.

Another safety-critical application area is in nuclear-power systems. Their requirement for deterministic application of their code is so stringent that programmers at nuclear-power stations have even designed their own custom hardware for the code to run on.

Custom hardware and provable design. Generally, commercially available modules are designed for middle-of-the-road applications. These modules are not documented to the level that one desires when providing a safety-critical function, such as nuclear-power control. These commercial modules change by an evolving design and manufacturing process. The documentation does not always keep up with these design changes or with the out-of-specification manufacturing that is sometimes done. These uncertainties cannot be tolerated here.[12] Nuclear reactors require a certainty that is hard to achieve in commercial processors.

Design problems have been experienced by the microprocessor industry from the beginning.[13,14] News of the arithmetic accuracy problems of the Intel Pentium processor raised a lot of concerns for those applications where numerical accuracy was critical. These problems typically surface at inopportune times. In a safety-critical nuclear application, more certainty and control were deemed to be required for their application. The system developer for critical applications may choose to develop a custom microprocessor with a mathematically provable design.

It should be noted that software faults are akin to hardware design faults. They occur the same way: oversights in the design, application domain exceeding the specification or design, or deleterious impact of erroneous inputs.

Practical design considerations. Any step taken to produce reliable code makes a more consistent and safe operating system. Safety goes further however. We are particularly concerned about what happens when something does fail. The failure instigator can be any component of the system as well as the attending human operator or maintainer. (It should be remembered that people can provide system inputs that go outside of the design criteria or the designer's intentions. The author has seen a cus-

tomer engineer (CE) in the field file a detent to enlarge it and then cut coils from a spring in a printer in an attempt to fix a random line-advance error that the printer was exhibiting. What designer ever considered that his design would be subjected to such post-shipment modifications?).

The system-safety perspective then is to mitigate the associated effects of failure to a tolerable level. Per MIL-STD-882C, safety is compromized whenever there is personal injury or system damage. Below are some practical guides for a reliable and safe design:

1. Exception-handling code is the most vulnerable to error. Dr. Harlan Mills, in a recent seminar, cited a case where he was called in to examine code problems that were being experienced on a JES release. He did a structured analysis of the code and found no faults in the main operational code. All of the problems discovered lay in the exception-handling code. This code is typically put last in the design, with the least emphasis, which is an exposure. Also, the variability (of inputs and operating environments) is greatest here.
2. Code change is dangerous. Jon Musa, one of the long-time leaders in software reliability, estimated that 30 per cent of the changes made to code create new faults (private conversations).
3. Interfaces should be managed by enumerating and documenting dependencies, controls, and data objects across the interface. Validate input to be as expected, within acceptable bounds. Be able to check erroneous input in real time.
4. Process management of the development and test process. When faults are discovered, examine the process from the following three aspects.

(a) How can the process be modified to discover the fault sooner in development? The software-development process is really a defect-removal process, and improving the process develops better code more quickly.
(b) How can the process be modified to preclude or mitigate the initiation of such an error? For example, could a template be devised to prompt completeness of the software statement at the time it is created?
(c) Are any similar-type bugs embedded in the code? For example, in the Space Shuttle program there was ample ground testing of the system. During one simulator experience, the astronauts were executing a piece of code and their exercise was interrupted. When they came back and re-entered the code, a safety-critical error occurred. Being on the ground, the consequence was not significant, but it would have been on a real mission. Houston then used a tool to search the entire program for code with similar potential susceptibility. This searching process took three days, but what is the value of finding and removing another potentially safety-critical fault?

5. Fault-tree analysis can help identify *a priori* the probable design exposures. This is an enumeration or brainstorming of the erroneous things that the code might be able to do. Once identified, these safety faults are traced down to lower levels of the design to identify sensitive-code areas, control points, and possible defences to protect against their occurrence. The critical faults are traced back to probable causality points. This is a top-down analysis and leads to defensive programming that is effective at checking safety exposures.
6. Failure mode, effect, and criticality analysis (FMECA) is the reverse of the fault-tree analysis. It is a bottom-up analysis. You start with an enumeration of the possible things that can go wrong with your module (a module is the lowest practical level of analysis). One could beneficially consider:

(a) Single fault-failure modes

(i) stop—termination with notification
(ii) crash—termination without notification
(iii) hang—same as crash but continues resources usage
(iv) incorrect data response—early, late, lost, spurious data
(v) exception—planned module exceptions.

A lot of the above analysis forces one to see how the system handles exceptions within a given module. Again, this amounts to managing and understanding the interfaces and system expectations and protocols.

(b) Multiple faults failure mode (distributed system)

(i) desynchronization
(ii) inconsistency
(iii) partitioning—two correctly functioning but non-communicating item partitions.

Of all the above strategies, the best one for the individual designer is to use FMECA as a tool to look at the other side of the design and to work out interface and system-level controls for handling aberrant conditions. The fault-tree analysis can be very helpful early in the design, just to alert the designer to sensitive design areas before developing the code.

FMECA provides a systematic method of analysing the design. This analysis will typically point out

interface dependencies and requirements. For example, the module will be susceptible to conditions under which it cannot recover from or contain the error without the aid of some system resource outside of itself. It will depend most probably on some system supervisory code. These interface dependencies need to be identified, documented, and agreed to by managers of interface software. FMECA is the tool to stimulate the identification of these dependencies.

For the reader who wants to go futher, there are other references this author would like to cite.[15–18] Following is a set of specific-safety design references that may stimulate some proactive design initiatives for considering safety *a priori* in the design activities.

SYSTEM SAFETY GUIDELINES

System-design guidelines for safety-critical software

1. CPUs that process entire instructions or data words are preferred to those that multiplex data or instructions (e.g. an 8-bit processor is preferred to a 4-bit processor emulating an 8-bit machine.)
2. Software design and code shall be modular. Modules shall have one entry and one exit point in accordance with DOD-STD-2167 design guidelines.
3. Software control of analogue functions shall have feedback mechanisms that provide positive indications that the functions have occurred.
4. The system and its software shall be designed for ease of maintenance by future personnel not associated with the original design team.
5. A power-up test shall be incorporated in the design that ensures that the system comes up in a safe state and that safety-critical circuits and components are tested to verify their correct operations.
6. Watchdog timers or similar devices shall be provided to ensure that the microprocessor or computer is operating properly. The timer reset shall be designed such that the software cannot enter an infinite loop and reset the timer as part of the loop sequence.
7. The system shall be designed such that a failure of the primary control computer will be detected and the system will be returned to a safe state.
8. Maintenance interlocks shall be provided to preclude hazards to personnel maintaining the system. Where interlocks must be overidden to perform tests, etc., they shall be designed such that they cannot be inadvertently overridden, or left in the overridden state once the system is restored to operational use.
9. The operational and support safety-critical software shall contain only those features and capabilities required by the system. The programs shall not contain 'undocumented features'.
10. The system shall be designed to provide a safe shutdown under power-failure conditions. Fluctuations in power shall not be capable of creating potentially hazardous states.

Specific-design guidelines for safety-critical software

1. Operational program loads shall not contain unused executable code.
2. Operational program loads shall not contain unreferenced variables.
3. Safety critical computer software components and safety-critical interfaces shall be under positive control at all times.
4. All processor memory, not used for or by the operational program, shall be initialized to a pattern that will cause the system to revert to a safe state if executed. It shall not be filled with random numbers, halt, stop, wait, or no-operation instructions. Data or code from previous overlays or loads shall not be allowed to remain. (Examples: If the processor architecture halts upon receipt of non-executable code, a watchdog timer shall be provided with an interrupt routine to revert the system to a safe state. If the processor flags non-executable code as an error, an error-handling routine shall be developed to revert the system to a safe state and to terminate processing). Information shall be provided to the operator to alert him to the failure and the reversion to a safe system state.
5. Overlays shall all occupy the same amount of memory. Where less memory is required for a particular function, the remainder shall be filled as discussed in Item No. 5. It shall not be filled with random numbers halt, stop, no-operation, wait instructions, or data or code from previous overlays.
6. The software shall be designed to detect potentially unsafe conditions and states, either within the software or the overall system, and shall be capable of preventing the potential hazard's occurrence by recovering to a safe state. When a potentially unsafe state has been detected, the software shall alert the operator to the anomaly detected, the action taken, and the safed system configuration and status.
7. The system design shall not permit detected unsafe conditions to be bypassed.
8. The software shall provide for making hardware subsystems safe under the control of software when unsafe conditions are detected.
9. The software design shall prevent unauthor-

ized or inadvertent access to or modification of the code. This includes preventing self-modification of the code.

10. Potentially catastrophic or critical hazards shall be controlled by at least two independent functions. Where a single computer program is used to control potentially catastrophic or critical hazards (MIL-STD-882B Category I or II), at least three fully independent software functions are required.
11. The software design shall ensure that the system is in a safe state during power-up, intermittent faults in the power, or in the event of power loss. The software shall provide a safe, graceful shutdown of the system due to either a fault or power-down, such that potentially unsafe states are not created.
12. The software shall be designed to perform a system level check at power-up to verify that the system is safe and functioning properly prior to application of power to safety critical functions, including hardware controlled by the software. Periodic tests shall be performed by the software to monitor the safe state of the system. Safety critical functions shall be grouped together, and the number of affected program modules shall be minimized where possible within the constraints of operational effectiveness, computer resources, and good software design practices.
13. Safety critical functions shall be grouped together. The number of affected program modules shall be minimized where possible within the constraints of operational effectiveness, computer resources, and good software design practices.
14. Conditional statements shall have all possible conditions satisfied and be under full software control (i.e. there shall be no unresolved potential input to the conditional statement).
15. Safety critical functions shall exhibit strong data typing. Safety critical functions shall not employ a logic '1' and the '0' to denote the safe and armed (potentially hazardous) states. The armed and safe states shall each be represented by at least a four-bit unique pattern. The safe state shall be a pattern that cannot, as a result of a one or two-bit error, represent the armed pattern. If a pattern, other than these two unique codes is detected, the software shall flag the error, revert to a safe state, and notify the operator.
16. The system shall provide for fail-safe recovery from inadvertent instruction jumps.
17. The system shall detect inadvertent jumps within or into safety critical computer software components, return the system to a safe state, and, if practical, perform diagnostics and fault isolation to determine the cause of the inadvertent jump.
18. The software shall be capable of discriminating between valid and invalid (e.g., spurious) internal interrupts and shall recover to a safe state if they occur.
19. Decision statements shall not rely on inputs of all ones or all zeros, particularly when this information is obtained from external sensors.
20. Halt or stop instructions shall not be used.
21. Flags shall be unique and shall have a single purpose.
22. Files shall be unique and shall have a single purpose. Scratch files shall not be used for storing or transferring safety critical information between processes.
23. Operational checks of testable safety critical system elements shall be made immediately prior to performance of a related safety critical operation.
24. Upon completion of tests where-in safety interlocks are removed, disabled, or bypassed, restoration of those interlocks shall be verified by the software prior to being able to resume normal operation. While overridden, a display shall be made on the operator's or test conductor's console of the status of the interlocks.
25. Periodic memory and database checks shall be performed. The design of the test sequence shall ensure that single point or likely multiple failures are detected and isolated. Computer and external hardware design shall preclude harmful effects from electromagnetic radiation or electrostatic interference.
26. Computer and external hardware design shall preclude harmful effects from electromagnetic radiation or electrostatic interference.
27. The software shall make provisons for logging all system errors detected.

Design guidelines for safety-critical interfaces

1. Inter-CPU communications shall successfully pass verification checks in both CPUs prior to the transfer of data. Periodic checks shall be performed to ensure the validity of data transmissions.
2. Data transfers messages shall be of a predetermined format and content. Each transfer shall contain a word or character string indicating the type of data and content of the message. As a minimum, parity checks and check sums shall be used for verification of correct data transfer. Character recognition codes (CRCs) shall be used where practical.
3. External functions requiring two or more safety critical signals from the software (e.g. command formatting and transmission) shall not receive all of the necessary signals from a single register or input/output port. In addition, these signals shall not be generated by a single CPU command.

4. The software shall be capable of discriminating between valid and invalid (e.g. spurious) external interrupts and shall recover to a safe state in the event of an erroneous external interrupt.
5. Decision statements shall not rely on inputs of all ones or all zeros, particularly when this information is obtained from external sensors.
6. Safety critical input or output functions shall not imply a logical '1' and '0' to denote the safe and armed (potentially hazardous) state. The functions shall be represented by at least four bits. The armed state shall be represented by a unique bit pattern. The safe state shall be represented by another unique pattern that cannot, as a result of a one or two bit error represent the armed pattern. If a code other than these two unique codes is detected, the software shall flag the error, revert to a safe state, and notify the operator of the erroneous input or output.
7. Limit and reasonableness chekcs shall be performed on all inputs and outputs prior to action occurring based on those values.
8. The software shall be designed to detect failures in external hardware input or output devices and revert to a safe state upon their occurrence. The design shall consider potential failure modes of the hardware involved.

Design guidelines for safety-critical operator interfaces

1. The software shall be designed such that the operator may cancel current processing by a single action so that the system reverts to a safe state. The system shall be designed such that the operator may exit potentially unsafe states with a single action. This action shall revert the system to a known safe state. (e.g. the operator shall be able to terminate command transmission processing with a single action. The action may consist of pressing two keys simultaneously, etc.)
2. Two or more unique operator actions shall be required to initiate any potentially hazardous functions or sequence of functions. The actions required shall be designed to minimize the potential for inadvertent actuation.
3. Opertor displays, legends, and other interactions shall be clear, concise, and unambiguous.
4. The software shall be capable of detecting improper operator entries or sequences of entries or operations. It shall alert the operator to the erroneous entry or operation. Alerts shall indicate the error and corrective action. The software shall also provide positive confirmation of valid data entry or actions taken (i.e. the system shall provide visual and/or aural feedback to the operator such that the operator knows that the system has accepted the action and is processing it). The system shall also provide a real-time indication that it is functioning. Processing functions requiring several seconds or longer shall provide a status indicator to the operator during processing.
5. Alerts shall be designed such that routine alerts are readily distinguished from safety-critical alerts. The operator shall not be able to clear a safety-critical alert without taking corrective action or performing subsequent actions required to complete the operation.

Guidelines for development phases of safety-critical software

Coding phase.

1. Desk audits and peer reviews shall be used to verify implementation of design requirements in the source code with particular attention paid to the implementation of identified safety critical computer software components and the guidelines provided in this document.
2. At least two people shall be thoroughly familiar with the design, code, and operation of each software module in the system.
3. Configuration control shall be established as soon as a practical software baseline can be established. All subsequent software changes must be approved by the Software Configuration Control Board prior to their implementation. A member of the Board shall have the responsibility for evaluation of all software changes for their potential safety impact. This member should be a member of the system safety engineering team. A member of the hardware Configuration Control Board shall be a member of the Software Configuration Control Board to keep members appraised of hardware changes and to ensure that software changes do not conflict with or introduce potential safety hazards due to hardware incompatibilities.
4. Conditional statements shall be analysed to ensure that the conditions are reasonable for the task and that all potential conditions are satisfied and not left to a default condition. All condition statements shall be commented with their purpose and expected outcome for given conditions.
5. Reviews of the software source code shall ensure that the code and comments agree.
6. Patches of safety critical software shall be prohibited throughout the development process. All software changes shall be coded in the source language and compiled prior to entry into test equipment.

Software testing phase.

1. Software testing shall include NO-GO path testing.
2. Software testing shall include input failure mode testing.
3. Software testing shall include boundary, out-of-bounds, and boundary crossing test conditions.
4. Software testing shall include input, values of zero, zero crossing, and approaching zero from either direction.
5. Software testing shall include minimum and maximum input data rates in worst case configurations to determine the system's capabilities and responses to these environments.
6. Safety critical computer software components in which changes have been made shall be subjected to complete regression testing.
7. Operator interface testing shall include operator errors to verify safe system response to these errors.

Life cycle logistics.

1. Changes to safety critical computer software components on deployed or fielded systems shall be issued as a complete package for the modified unit or module and shall not be patched.
2. Firmware changes shall be issued as a fully functional and tested circuit card. Design of the card and the installation procedures should minimize the potential for damage to the circuits due to mishandling, electrostatic discharge, or normal or abnormal storage environments.

REFERENCES

1. Robert Davis, 'Costly bugs', *Wall Street Journal*, 28 January 1987, pp. 1 and 17.
2. Roger Dettmar, 'Making software safer', *IEE Review*, September 1988, pp. 321–324.
3. Edward J. Joyce. 'Software bugs: a matter of life and liability', *Datamation*, 15 May 1987, pp. 88–92.
4. Phillip Ross, 'The day the software crashed', *Forbes*, 25 April 1994, pp. 142–156.
5. Evelyn Richards, 'Study: software bugs costing U.S. billions', Washington Post, 17 October 1988, p. D1.
6. Lewis Bass, 'Hazards of government contracting', *Aerospace America*, October 1984, p. 86.
7. John Ryan, 'The company that hates surprises', *Quality Progress*, September 1987, pp. 12–16.
8. Edward J. Joyce, 'Is error-free software achievable?', *Datamation*, 15 February 1989, pp. 53–56.
9. S.J. Keene, Ted Keller and John Musa, 'Developing reliable software in the shortest cycle time', *IEEE Educational Activities Video Tape*, Summer 1995, 2 1/2 hours.
10. B.G. Kolkhorst and A.J. Macina, 'Developing error-free software', *IEEE AES Magazine*, November 1988, pp. 25–31.
11. K.C. Ferrara, S.J. Keene and C. Lane, 'Software reliability from a system perspective', *Proceedings Annual Reliability and Maintainability Symposium*, 1989, pp. 332–336.
12. Martin Thomas, 'Safety: should we trust computers?', *Computer Fraud and Security Bulletin*, **11**, (1), 15–21 (1988).
13. J.V. Hill and P. Robinson, 'The development of high reliability software RRA's (Rolls Royce) experience for safety critical systems', *IEE Colloquium on 'Software Requirements for High Availability Systems'*, Digest No. 115, 1988, pp. 1/1–7.
14. Rogerio DeLemos, Amer Saeed and Tom Anderson, 'Analyzing safety requirements for process control systems', *IEEE Software*, May 1995, pp. 42–53.
15. Philip Bennett, 'Assessing the risks of a safety-critical system', *Electronics*, June 1988, pp. 51–54.
16. Mark D. Hanson and Ronald L. Watts, 'Software system safety and reliability', *Annual Proceedings of the Reliability and Maintainability Symposium*, 1988, pp. 214–216.
17. Colleen Henneberry, 'High availability applications merit fault tolerance', *Bank Systems and Equipment*, July 1988, pp. 35–37.
18. K. Geary, 'Beyond good practices: a standard for safety critical software', in *Achieving Safety and Reliability with Computer Systems*, Elsevier Appl. Sci. Publishers, 1987, pp. 232–241.

Author's biography:

Samuel J. Keene is a principle in Performance Technology, a reliability and quality consulting company and is currently manager of Systems Reliability Engineering at Storage Technology Corporation. He is the senior past president of the IEEE Reliability Society and is a Fellow of the IEEE. He has authored over 125 papers on hardware, software, and systems reliability. He may be reached at Performance Technology 3081 15th Street, Boulder, Colorado 80304, U.S.A. Telephone: (303) 673-5963; Sam_Keene@storek.com.