# Command Interleaving for Environment Induced Fault Mitigation in High Consequence Embedded Software

**R A Williams**
Sandia National Laboratories, Albuquerque, USA, rawilli@sandia.gov

**Abstract:**

"At-most-once, requiring authorization" operations are special high consequence activities that must be executed as part of the device operation, but that must not be executed prematurely, repeatedly, or without authorization. Examples may include space or automotive applications that result in an irreversible operation or decision that, if executed in a premature or unauthorized fashion, could cause injury, loss of life, or significant property destruction. Embedded systems that operate in hazardous or unpredictable environments must be designed to withstand those environments, but occasionally abnormally strong physical effects occur that exceed the design envelope. Under these circumstances, an abnormal physical environment will produce enough disruption to cause unintended behavior.

The probability of an environment induced fault, or combination of faults, causing premature execution of high consequence code can be reduced by designing the hardware and software portions of an embedded system in a way that ensures a triple-fault condition is required to produce the undesired outcome. This paper introduces the strategy of "command interleaving", a design technique for embedded systems that uses formal logic to harden the high consequence portion of the code to predictable double fault conditions.

## 1. INTRODUCTION

The purpose of this paper is to introduce a simple fault-tolerant defensive design strategy that ensures that a necessary high-consequence operation can be executed reliably when authorized and when the necessary preconditions exist, but not when unauthorized or when a necessary precondition is lacking. The "command interleaving" strategy incorporates both hardware and software design principles. The resulting design will be robust to double fault conditions and is suitable for adverse operating conditions under which environment generated faults and errors are plentiful.

The command interleaving strategy is designed to protect behaviors that are within a device's operating envelope, but that must occur only once per authorization, within a narrow range of conditions. Such behaviors, characterized as "at most once, requiring authorization" (AMORA) operations, are a small subset of possible operations an embedded system might perform. They are typically high-consequent operations that, if executed prematurely or without authorization, would result in severe, irreparable harm to human life, the environment, or property.

## 2. BACKGROUND

### 2.1 Environment Induced fault

An environment induced fault is an incorrect or unexpected software or hardware state that occurs due to the physical environment in which the system is operating. Noise on a transmission line or in a transmission medium can garble a transmitted message. Complementary metal-oxide silicon (CMOS) circuits, including memory and microprocessors, are vulnerable to changes in state due to external stimuli. Radiation, for example, can induce a phenomenon known as a single-event upset (SEU) in

CMOS-7 and similar semiconductors. If a charged particle hits the gate region of a PMOS transistor or the drain region of a NMOS transistor, the particle passes through in a few picoseconds but leaves behind a region of disrupted charge that can cause a gate to turn off or on, changing from a logic 0 to a logic 1 or vice versa. [1] Similarly, frequent or rapid changes in state in nearby transmission lines or memory addresses can induce a state change in an adjacent line, latch, or memory address.

Regardless of its cause, a logical inversion at the bit level can propagate through latches, conditional logic, math logic, or any hardware, software, or firmware that relies on that bit. Although formal method analysis can sometimes capture the consequences of a single-bit fault in hardware, capturing the consequences of a single-bit inversion depends on where it occurs and, sometimes, on the state of the rest of the system. If a digital circuit is controlled or enabled by a single bit that acts as an "on" or "off" switch, and if that single bit happens to be the one that changes state in response to an environmental event, then all portions of the circuit downstream of the bit (directly or indirectly enabled by it) will start or stop. For this reason, a designer cannot simply rely on cascaded AND gates: if the output of the last gate happens to be the bit that is hit, none of the logic upstream of the final gate can prevent premature activation.

Although the SEU phenomenon has been well observed in CMOS7 technology, simply making semiconductor devices smaller to reduce the surface area that could potentially be struck by a charged particle is not a valid radiation hardening technique because SEU phenomena have been observed in sub-100 nm SRAM test. [2] Radiation hardening by process, by hardware design for critical logic blocks, by physical shielding, by physical redundancy, or by software and system mitigation, can indeed help to ensure reliability in adverse environments. However, the expense of such mitigations in terms of space and power consumption make them impractical for extremely resource-constrained applications.

### 2.2 Consequences of Environment Induced Fault in Software

An environment induced bit inversion can occur "in software" in any location occupied by executable digital instructions. These locations include, but are not necessarily limited to, memory, cache, registers, or the processor pipeline or look-ahead buffer. Depending on the design of the integrated circuit, faults can occur in a register of the microcontroller itself, or in some of the combinatorial or control logic. When it does, the line between "hardware" and "software" faults begins to blur. Yet even when the microprocessor is working perfectly, when it encounters a corrupted opcode or memory address, it attempts to execute the instruction. When it does, one of four things may occur.

The microprocessor may reset, experiencing a condition that returns it to a default state similar to what would happen in response to a momentary loss of power. It may throw a hardware exception if the bit inversion creates an invalid opcode or an illegal out-of-range memory access attempt and if the microprocessor's invalid access and opcode detection is enabled. Non-maskable hardware interrupts such as division by zero also cause execution to vector immediately to the exception handler. If the exception handler is appropriately written, it is possible to include a limited recovery or mitigation process. This behavior, in a higher-level language or RTOS that supports software exception handling, may be replicated for more sophisticated mitigation. The processor might simply halt, experiencing an internal condition that precludes further execution of instructions even though the clock is still running and other parts of the system are operational. Finally, it may execute a different valid instruction than what was intended, to a valid address, and keep running in a way that executes one or more instructions that were not intended to execute given the machine state. The impact of the latter operation varies depending on what peripheral device or memory is affected.

From a safety perspective, the worst-case scenario occurs when the software branches prematurely to code it has not yet been authorized to execute, including instructions that initiate an authorized AMORA behavior. There are several possible causes for this phenomenon.

The “jump-ahead” fault occurs when a destination address is corrupted. This address could be held in a processor register, a memory address, or the pipeline. When a SEU occurs in a destination address, unless the SEU results in an illegal address there is nothing to prevent the address from being treated as valid during a branch or a return. The processor, operating correctly, skips ahead to the valid but erroneous address and begins executing the code that resides there, with whatever internal register state, peripheral state, and memory contents were in effect when the fault occurred. Because the register and state contents are not what is expected or assumed, and might be any legal value, system behavior past this point is unpredictable. The best-case scenario when a jump-ahead fault occurs is for the processor or memory state to result in an exception, however such behavior is not guaranteed. A jump-ahead fault can therefore result in multiple cascading faults including buffer overruns, overwriting of variables, and premature execution of code.

The “wrong path” fault occurs when the SEU occurs in a flag register that is subsequently used as a branch criterion. For example, if a fault in a variable prior to a comparison causes the zero flag to be set or cleared erroneously prior to a conditional jump, the wrong path will be taken. When the false positive or false negative is related to a test of a state variable, authorization code, or similar branch that determines whether the AMORA operation is attempted, it could potentially cause the operation to occur in response to a single environment-generated fault. Similar behavior will occur if data corruption occurs in the zero flag itself within the processor.

When a sequence of machine code instructions exists in any memory location, and when there is a design mode path by which the instructions may be accessed, there is always a possibility (as anyone who has debugged with JTAG is aware) that a processor forced to a specific location will continue execution from there without changing its register contents. A well-designed processor may flush its look-ahead buffer and continue execution, however unless its exception handling behavior is triggered, the register contents will remain unchanged but be treated as valid and legitimate by the processor when executing instructions at its new address.

Modeling or simulating the range of possible behaviors in a jump-ahead fault is impractical because of the sheer number of possible register contents when the fault occurs. Furthermore, because of the extreme and unpredictable conditions that can occur in an accident environment such a lightning strike, a fire, or a plane crash, mathematical modeling tends to be extremely sensitive to tiny variations in input. Predicting what can or cannot occur when a circuit board simply burns or when impact with a sharp object severs one electrical connection or creates a physical short in another becomes exponentially more complicated as the size of the circuit, the quantity of code, the opportunities for human error or malicious tampering, and the variety of possible environmental stimuli increase.

### 2.3 AMORA Operations

To meet the definition of an AMORA operation for the purposes of this paper, a desired high consequence computer behavior must meet three criteria. First, it must be within the set of required design mode behaviors (as opposed to anomalous behavior). Second, it must be executed at most once. Finally, an AMORA operation must require explicit human initiation, authorization, or similar expression of intent. Although the operation may be automated and performed at some point after the human authorization, it must not be performed without it.

The at-most once component of an AMORA behavior presents implications for failover and retry behaviors, which are common in most high-reliability or high-availability computer systems.

Retry behaviors are highly inadvisable for AMORA operations because there are use cases in which a single authorized event can be executed more than once. Every system design in which networked devices exchange information and perform handshaking is vulnerable to a situation in which an operation succeeds at the destination, but the handshaking or acknowledgement doesn’t make it back to an earlier initiating computer. Anyone who has had a credit or debit card charged multiple times for a single purchase, or who has been hit with fines due to an electronically authorized bill payment making

repeated automated withdrawal attempts despite the first one having been successful, is aware of how severe and far-reaching the consequences of badly designed failover processes can be.

Unlike a bank account balance query that may safely be retried if a first attempt fails, an AMORA behavior should never incorporate an automatic, autonomous, or semi-autonomous retry. Examples from the banking industry would include transactions such as purchases, account debits, or transfers of funds or securities. If the transaction fails, is determined to be fraudulent, or is authorized erroneously, the default behavior in the system should not be to make repeated attempts, but to abandon the transaction. This is the "at most once" behavior described in the AMORA definition. Although AMORA behaviors are within a device's operating space and must occur when authorized, the default is for the behavior not to occur. The system, in other words, must fail in a safe configuration.

Examples of AMORA operations include the opening of an exterior airlock door on a space station, the release of a safety restraint on a roller coaster, and the destruction of an encryption key in a law enforcement drone that crashes and cannot be recovered. Each AMORA operation is an irreversible act that could, if executed prematurely, cause an unacceptable loss of human life, public trust, or similar unrecoverable assets. Most systems that incorporate AMORA operations have a hierarchy of requirements in which the at-most-once aspect of the operation takes precedence over failover behaviors. It is therefore better for an AMORA operation to not occur at all than it is for it to occur prematurely, in an unauthorized fashion, or more than once in response to a single authorization.

Because of the consequences of an additional or unauthorized AMORA operation, an embedded system that performs them should be designed to require fresh or independent authorization for each operation. Should the operation fail for any reason, if a reattempt capability is built in at all, fresh authorization should be required.

Although it is possible to build sophistication into a retry behavior, so that the system verifies that the final AMORA operation has not occurred prior to reattempting it, the process of checking for confirmation is itself susceptible to failure and can return a false negative. The same fault conditions that prevented the authorized operation from occurring could potentially interfere with the acknowledgement or verification operation. In a networked system where more than one physical data path is possible, or in adverse environments, automated retries are a bad idea.

## 3. COMMAND INTERLEAVING

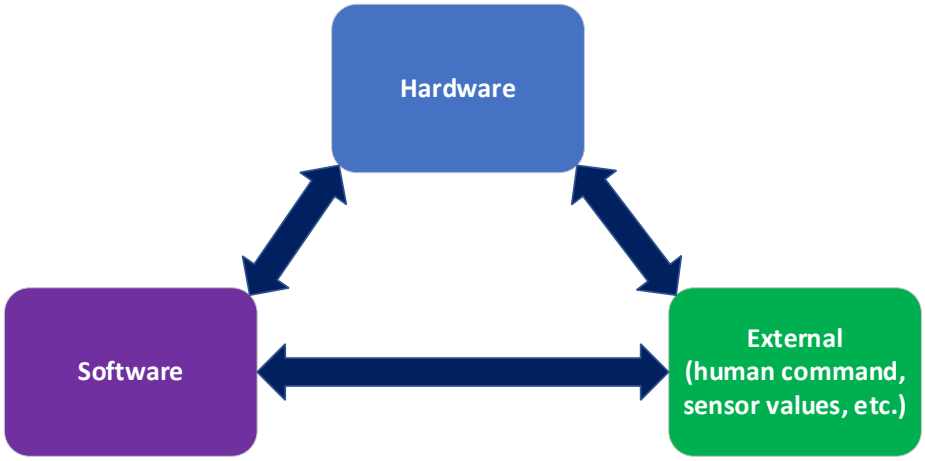
Command interleaving is a design technique intended to harden an embedded system to two environment induced faults to prevent premature or unauthorized execution of AMORA operations. It can be characterized as a form of software and system mitigation. It uses formal logic to harden the high consequence portion of the code to mitigate against predictable double fault conditions. It is built on the assumption that environment-initiated faults can and will occur, and that they may come from unpredicted sources.

The command interleaving process is designed with the assumption that code can and will jump ahead to an AMORA operation without having satisfied its necessary preconditions, and that the processor registers will coincidentally contain a state that does not preclude execution of the machine code that causes the AMORA operation. In essence, the interleaving process is a hardware and software codesign process by which the hardware aspects of the AMORA operation are divided into at least three steps, such that each step is preceded by a test of a relevant precondition.

Command interleaving relies on an extension of the McGovney Principle, which is that hardware should check software, software should check hardware, software should check software, and hardware should check hardware. Because the same environment that generates the fault can render self-checking mechanisms suspect within the same circuit, the principle is expanded to add external stimulus to the list of available sources of reality checking and capability enablement.

Figure 1 shows the relationship between hardware, software, and external stimulus. Each of the three resources can still reality-check itself, when practical, however a fault or error that occurs in more than one resource can still be caught by the other resources.

**Figure 1: External Stimuli, Software, and Hardware**




The idea is to design the embedded system so that hardware states interleave with software states along with external stimuli. The right operations must occur, in the right order, or else the system *will not be physically capable* of completing the AMORA operation. Instead of designing to resist fault or to correct fault, the system should be "born safe", and physically incapable of executing the high consequence operation without the correct sequence of events and causes.

For the external component, three different stimuli must arrive, from at least two independent sources. These stimuli should include, at minimum:

- Some means of expressing authorization, human intent, or consent for the high consequence operation to occur. This could be provided in advance, such as when enabling automatic braking in an automobile that could override the decision of a human driver
- A means for detection of the appropriate operating environment or conditions, which typically involves one or more sensors
- A "command" or proximate cause, which is the real-time trigger for the action. For operations authorized in advance, this could be initiated by a timer, a network signal, or a similar means to initiate a pre-authorized operation in real time

### 3.1 Hardware States

A command interleaved design has four possible hardware states, which can be abstracted as "unconfigurable", "configurable but not configured", "configured", and "initiating". The initiating state is the only one that functions like a music box, in which the AMORA operation is performed because all the preconditions have been satisfied. These states differ from those used in traditional embedded systems.

In traditional embedded systems there is seldom an "unconfigurable" state because the system is designed to boot into a default state of some kind and to be fully powered and configurable. In the specific case of AMORA operations, at least one hardware module responsible for the final initiation of the activity should have a state that prevents it from accepting configuration. This can be accomplished through power gating, signal gating, or any means that prevents the peripheral from accepting configuration.

Exactly which strategy is chosen to enforce an "unconfigurable" state depends on the thermal environment in which the device must operate and the speed with which the AMORA operation must occur after it is authorized. An emergency braking application, for example, cannot tolerate the full second of delay necessary to warm up the chip in an extremely cold environment. For time-critical activities, signal gating is therefore preferred.

It is important to note that it is possible for a physical fault on a power or signal line to replicate authorized gating behavior. For this reason, power and signal gating should be designed with at least two signals, one active-high and the other active-low. This requires two separate and fortuitous faults to replicate the behavior of authorized gating.

In a command interleaved design, processor and peripheral features are not enabled until immediately before they are needed. This minimizes the opportunity for a single spurious, environment generated signal to cause premature execution of any operation. For this reason, the "configurable but not configured" and "configured" hardware states are separate.

When a semiconductor device is exposed to enough charged particles to induce SEU, the distribution of inverted bits across the device can be regarded as a function of the relative sizes of the gates in terms of the surface area exposed to the particles. In the case of an interrupt status register and its corresponding enable register, the two gates will be of the same size. They will also be oriented in the same direction relative to the wave of charged particles because the physical distribution of the gates is in the same two-dimensional plane. In most cases, the gate, drain, and source regions will even be oriented in the same direction. Every similarly designed transistor in the device undergoes the same fabrication process as the others on the wafer. Accordingly, it is reasonable to assume that, given the presence of sufficient charged particles, the statistical distribution of SEU events between same-sized logic gates within a given semiconductor device will be uniform.

Simply disabling an enable bit until a signal is needed and anticipated reduces the probability of a spurious signal causing unwanted system behavior. In Equation 1 below, given an environment capable of producing a SEU in one or more bits, the probability of *both* the enable and status bit of a given register being erroneously within a given time frame is represented by $P_F$. $P_E$ represents the probability of the interrupt being erroneously enabled by the adverse environment and $P_S$ represents the probability of an erroneous signal occurring at an input latch or the state register, each of which is a separate transistor.

$$P_F = P_E \times 2(P_S) = 2{P_E}^2$$

The probability of the enable and signal both being erroneously activated by the environment can be regarded as independent, however because there are two cascaded gates associated with the signal and a SEU in either can replicate the behavior of a real active-high signal, the probability is doubled. Were the interrupt to be enabled and configured at start-up, $P_E$ would not reduce the probability of erroneous interrupt initiation.

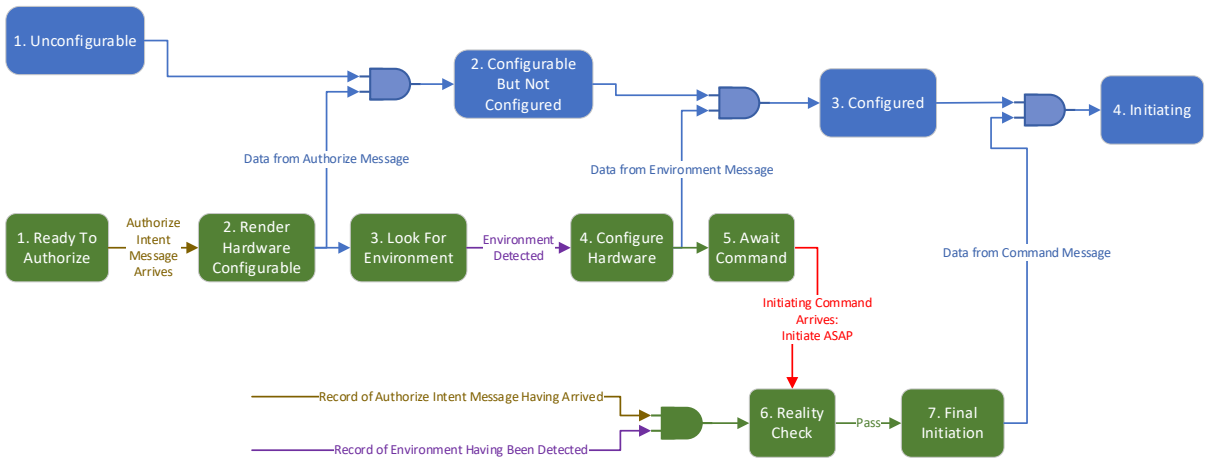
In a command interleaved system, once the hardware responsible for the AMORA operation is configured, it is physically capable of initiating the AMORA operation in response to appropriate stimulus. Yet this should never be a single-bit or single-gate operation. Instead, the means by which the AMORA hardware is commanded to perform its initiating operation should be complex enough to not be erroneously generated by any plausible environment or stimulus. Ideally the command or stimulus that provides the proximate cause for the AMORA initiation should incorporate authorization and environment detection data received from an external source, such that it is never cached or saved anywhere within the system software.

### 3.2 Software States and External Stimuli

In addition to the four hardware states described above, a command interleaved embedded system requires at least seven software states. These are “ready to authorize”, “render hardware configurable”, “look for environment”, “configure hardware”, “await command”, “reality check”, and “final initiation”. Figure 2 shows a simplified flowchart for a notional command interleaved embedded system. The four hardware states appear in blue, the seven software states appear in green, and the three commands or messages received from outside the software and hardware are depicted in brown, purple, and red.

The logic AND gates depicted in Figure 2 are not literal AND gates implemented in electronic form, but representations of logical necessity.

**Figure 2: Command Interleaved AMORA Operation Flowchart**




The “Ready to Authorize” software state occurs only when the software is ready to accept the message that expresses human-initiated intent or authorization for the AMORA event. This message can potentially represent preauthorization, such as when an automobile driver enables emergency braking or lane correction features. It can also meet “human in the loop” related requirements common in access control and physical security. But the message should be sufficiently complex to not be replicated by any plausible external environment. It should also never be stored or cached anywhere in the embedded system, particularly in non-volatile memory from which it could be prematurely or erroneously retrieved.

After receiving human-initiated authorization, the software may advance to its next state, in which it enables whatever power gating, signal gating, decryption, unlocking, or other process is necessary to render the hardware configurable. When this process is complete, the hardware advances to its “configurable but not configured” state and the software proceeds to its next state, in which it awaits evidence that the environment is appropriate for the AMORA operation.

Environment detection relies on one or more sensors that detect appropriate conditions for the execution of an AMORA operation. How many sensors there are, whether redundant sensors are present, and what conditions the sensors detect will depend on the application. In the case of a software-controlled airlock where the AMORA operation is the opening of the outer door, the system should not perform the operation unless the inner door is closed and the pressure inside the airlock is the same as the pressure outside the outer door. Such an application would require at least three sensors: one pressure sensor outside the outer door, another within the airlock, and a contact switch or sensor to detect whether the inner door is closed.

Prior to detection of the appropriate operating environment, the hardware that performs the AMORA operation is not configured and is incapable of responding even if a SEU fault were to cause the software to jump ahead to final initiation.

Once the appropriate operating environment and conditions are detected, the software configures the hardware responsible for initiating the AMORA operation. This configuration process should not be so trivial that it could be erroneously enabled by a short or any other single-bit fault. Ideally the configuration of the AMORA hardware will incorporate some portion of the authorization and/or environment detection messages, so that random memory or register contents will not produce the necessary configuration.

Once the software finishes configuring the hardware, it awaits the third and final stimulus from the outside world: the "command" or proximate cause that initiates the AMORA operation. When this command is received, the software advances to its reality checking state.

For each of the three external messages (authorization/intent, environment detection, and the initiating command), the software must retain some non-trivial form of evidence that the commands have been received and are valid. This reality checking process, which should occur in a cascaded fashion, helps mitigate against the jump-ahead fault along with wrong path faults. A stuck zero or equal bit register, for example, should not be able to cause all three of the reality checking steps to register a false success, however even if they do, a final mitigation remains.

The final initiation of the AMORA operation occurs when the software gives the initiation information to the hardware. As with the configuration of the hardware, the code or codes required by the AMORA hardware should not reside anywhere in the system. They should be passed in as part of the initiation and/or authorization messages, and that may even incorporate a portion of the environment detection.

Once the hardware advances into the "initiating" stage, the AMORA operation is inevitable if the peripheral that causes the operation is capable of functioning as designed.

### 3.3 Considerations For External Stimuli

The contents of the external stimuli messages, and the means by which they are deemed valid, are an important part of the system design process. These messages should never be trivial, and they should never consist of a single bit or state value that could be easily replicated by two independent environment generated faults.

The best strategy is to design the messages in such a way that the AMORA hardware will only respond correctly when the proper combination of intent, environment, and initiation codes are fed into an appropriately configured device. Simply comparing codes to a "known good" value usually introduces a single point of failure if the microprocessor experiences a wrong path fault due to an incorrect value in the zero or equal bit following a comparison, which could plausibly occur after a jump-ahead fault.

Under no circumstances should authorization or intent codes be stored or cached in software or in nonvolatile RAM, because then a runaway data transfer process could cause them to be unintentionally transferred and treated as good. Similarly, comparison of an intent code, key, or password against a "known good" internal value is a bad idea because it creates a single point of failure in the comparison operation.

Such communications as are used for the three external stimuli should be sufficiently complex to not be generated by any plausible external environment. This is a type of formal logic gating that creates a logical incompatibility, which in turn prevents the external stimuli from being used later in the reality checking stage of the process. It is still possible for a wrong branch path during the reality checking stage to occur as the second fault in the system, yet without the key information to pass to the hardware during final initiation the final initiation cannot occur.

Software interlocks such as firewalls can mitigate against out of order or erroneous external stimuli.

### 3.4 Additional Environment-Generated Fault Hardening Strategies For Software

There are many features that can help harden software to environment-induced faults. In systems that remain in adverse environments for a long time, such as space applications, background scrubbing and error correction codes are useful. So are redundancy-based comparison and voting strategies that perform real-time reality checks on high consequence variables such as control coefficients and state variables.

By performing some form of gating, configuration, handshaking, or enabling stimulus to whatever sensors or devices perform environment detection, the system can prevent premature or erroneous identification of the acceptable operating environment.

One way to prevent premature execution of software is to prevent it from being in executable RAM before it is needed. By dividing the executable code into more than one block, and by ensuring that the block containing the machine code instructions that perform reality checking and final initiation are held somewhere besides in executable RAM until after environment detection has occurred, the code can completely prevent initiation of the AMORA operation prior to environment detection. This strategy is very compatible with the customary division between "boot" code, which usually resides in ROM or even firmware, and "mission" code, which can potentially be changed or updated.

Every time the software is waiting for one of the three external messages, it is appropriate to insert a timeout feature. At the start of the state following the receipt of an external message, it is appropriate to verify that a timeout has not in fact occurred.

System designers should always be mindful of failure behaviors and how the system should fail. For any safety critical operation, the system should make itself as safe as possible following a perceived fault. In the event of a timeout, a garbled command, an illegal instruction or illegal memory access exception, or any sign of out of order execution, the software should have a safing behavior in which it either aborts entirely or returns to its default setting. Depending on the application this could involve an internal reset, an attempt to unpack and reload its mission code, a controlled shutdown, or notification to the operator that the AMORA operation is unavailable.

### 3.5 State Transition Analysis, Hardware

To harden a system to two faults, let us perform a cause agnostic fault evaluation not at the binary level but at the state level. This requires examination of the conditions necessary for environment induced state transition.

For hardware, this is a two-step process. One must consider the initial hardware state, of which there are three possibilities, and consider whether it is possible to advance to some subsequent state with one mechanical or logical glitch that can be generated by environment.

For the purpose of the state transition analysis, reverse transition to a previous state is not a threat because it does not cause advancement toward the AMORA operation. Transitions or changes within a hardware state due to SEU, heat initiated damage to a circuit component, or other environment initiated phenomena that affect hardware performance are labelled as self transitions and likewise could only cause advancement to the AMORA operation if they were to cause replication or duplication of hardware that could be used to initiate an AMORA operation, which is implausible. The final hardware state, "Initiating", is not a valid starting point because it represents authorized initiation of the AMORA operation.

It is possible to cause unpowered hardware to become powered if a physical short connects the power supply line of a power gated circuit or the signal gate of a signal gated circuit. For this reason, it is physically possible to advance from the unconfigurable state to the configurable state. But because the configuration of the hardware is non-trivial, and because the three external messages contain data

required to create the final initiation step, neither the messages nor the initiation of the hardware can be adequately performed simply through exposure to an adverse environment.

Table 1 shows the possible hardware state transitions from a valid initial state to some other state. Backward and internal state transitions are marked as N/A. Because only the application of power or the enablement of configurability can be generated with an externally induced short, and because transitions between hardware require information or data contained only in the external communication, there is no opportunity for a second environment induced fault to advance the hardware state.

**Table 1: Environment Initiated Hardware State Transition Analysis**

| Initial Valid State | Jump to Configurable but not Configured | Jump to Configured | Jump to Initiating | Can One More Fault Initiate AMORA Operation? |
|---|---|---|---|---|
| Unconfigurable | Possible: Short. | Impossible, non-trivial configuration complexity | Impossible, requires external message content | No, transition beyond Configurable impossible |
| Configurable But Not Configured | N/A, Self | Impossible, non-trivial configuration complexity | Impossible, requires external message content | No, transition beyond Configurable impossible |
| Configured | N/A, Backward | N/A, Self | Impossible, requires external message content | No, transition beyond Configurable impossible |

The software state transition analysis is similar, although if a single state variable code is used to determine state it can always be overwritten by an authorized write or update to a corrupted address that happens to erroneously map to the state variable. Backward and internal state transaction, as with hardware, are marked N/A since they do not represent advancement toward the AMORA operation.

Although there are seven distinct software states, from a state transition analysis with respect to the AMORA operation two pairs of states have an automatic “music box” state transition whereby normal execution of the code forces a state transition anyway. One such pairing occurs between the second and third states, “Render Hardware Configurable” and “Look for Environment”. Another occurs between the fourth and fifth states, “Configure Hardware” and “Await Command”.

The transition between “Reality Check” and “Final Initiation” is automatic if and only if all three external stimuli have been received and evidence of the same has been recorded in the system. Unauthorized transition from “Reality Check” to “Final Initiation” could occur only if the processor executing the reality check code takes a wrong path branch and processes the three external messages as having been received when they have not. Yet the hardware depends on data encoded in the first two external messages and cannot initiate the AMORA operation without the third. Therefore, for the purposes of environment-initiated state transitions, the “Reality Check” and “Final Initiation” states can be treated as one.

As with the hardware states, the final state cannot be regarded as an initial valid state for the purpose of environment-initiated state transition, since the software has already advanced to the final step by authorized means and receipt of appropriate external stimuli.

Whenever transition between one software state and the next is automatic in a correctly functioning system, the states can be treated as the same for the purposes of environment-initiated state transition.

Table 2 contains the environment-initiated software state transition analysis. The final pair of states, "Reality Check" and "Final Initiation" are included as initial valid states to demonstrate how a fault induced AMORA operation is identical to an authorized and appropriate one after the system is appropriately authorized, the environment is detected, and the initiating command arrives. The arrival of the initiating command into an appropriately configured system therefore represents a point of no return.

**Table 2: Environment Initiated Hardware State Transition Analysis**

| **Initial Valid State** | **Single Fault AMORA Possible After Jump to "Render Hardware Configurable, Look For Environment"?** | **Single Fault AMORA Possible After Jump to "Configure Hardware, Await Command"?** | **Single Fault AMORA Possible After Jump to "Reality Check, Final Initiation"?** |
|---|---|---|---|
| Ready To Authorize | No, hardware not configurable with wrong data | No, unconfigured hardware is unresponsive | No, unconfigured hardware is unresponsive |
| Render Hardware Configurable, Look For Environment | N/A, Self | No, hardware will be configured wrong and cannot initiate | No, even if reality check fails wrongly configured hardware won't initiate AMORA |
| Configure Hardware, Await Command | N/A, Backward | N/A, Self | No, even if reality check fails the hardware can't initiate AMORA without command data |
| Reality Check, Final Initiation | N/A, Backward | N/A, Backward | N/A, identical to design mode operation of an authorized AMORA operation in the correct environment wherein the initiating command has received |

### 3.6 Limitations and Caveats

The command interleaving design strategy can be used to harden an embedded system to two independent hardware and software faults that are credible but that still permit execution of software. Hardening to three or more faults is beyond the scope of the process.

Command interleaving does not prevent human initiated faults such as cyber attack or exploitation of one or more authorized or trusted channels. Preventing attacks along a bus, network, or similar communication medium is beyond the scope of the process.

## 4. CONCLUSION

The command interleaving design strategy, which incorporates both hardware and software design, is an effective means of hardening an embedded system against environment-induced faults. Instead of attempting to model all possible failure modes and conditions, designers mitigate against environment induced fault using formal logic, thereby limiting the probability space within which environment induced fault can occur.

Using the content of externally generated messages, which is not stored within the system, and ensuring that the hardware responsible for generating the AMORA operation cannot operate or initiate the final operation without correctly formatted data, ensures that if the data is complex enough it cannot be replicated by a noise environment or other data corrupting factor, the hardware cannot be made to initiate the AMORA operation prematurely or without expression of authorized human intent.

Analysis at the state level for both hardware and software shows that the logical and data dependencies between hardware, software, and external stimuli in the form are sufficient to harden the system to two environmentally generated faults, This precludes premature or unauthorized execution of AMORA operations due to two independent environment induced errors.

**Acknowledgements**

Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC (NTESS), a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (DOE/NNSA) under contract DE-NA0003525.



SAND2026-22501C

**References**

[1] E. Normand. "*Single-event effects in avionics"*, IEEE Transactions on Nuclear Science, Volume 43, Issue 2, pp. 461-474 (April 1996).
[2] M. Barker, K. Ryden, and A. Hands. *"Micro-latch and Single Event Upset phenomena observed during accelerated atmospheric neutron testing of 40-90 nm Commercial Off the Shelf Static Random Access Memories*", 22nd European Conference on Radiation and Its Effects on Components and Systems (RADECS), 03-07 October 2022, Venice, Italy.