Minimalist Recovery Techniques for Single Event Effects in Spaceborne Microcontrollers

D. W. Caldwell*, D. A. Rennels*
University of California, Los Angeles, CA 90024
doug@cs.ucla.edu, rennels@cs.ucla.edu

Abstract

This paper presents a fault-tolerant design approach to allow the use of non-hardened, commodity microcontrollers as embedded computing nodes in spacecraft, where a high rate of transient errors and occasional latch ups are expected due to the space radiation environment. In order to preserve their primary advantage of high functional density, low-cost approaches were explored that leverage features of existing commercial microcontrollers. A built-in, high-speed serial port is used for voting among redundant devices and a novel wire-OR output voting scheme exploits the bidirectional controls of I/O pins. A fault-tolerant node testbed was implemented, and fault-insertion tests were conducted to test the effectiveness of the fault-tolerance techniques.

1. Introduction

Microcontrollers are highly integrated computer systems on a single low-cost commodity chip containing a processor, program memory, RAM, discrete I/O, A/D converters, serial ports, and other support functions [1,2]. They offer great potential for reducing the cost and increasing the performance of modern spacecraft, but they have not been widely used in space because of their relatively low radiation tolerance. They are susceptible to single event upsets (SEU) which are transient bit-flips, and less frequent but more destructive single event latchups (SEL) which stimulate parasitic circuits within a CMOS device causing a local short that can only be cleared by power-cycling [3,4].

The fault-intolerance approach (radiation-hardening) is costly because rad-hard chips are difficult and time-consuming to develop. Functional density must be sacrificed to the design rules of rad-hard and SEE-immune layout. Because the fabrication technologies for EPROM, program memory, and A/D converters are generally incompatible with each other in rad-hard processes, such functions are

* Also with the Jet Propulsion Laboratory, Pasadena, CA

often moved off-chip. Fault-tolerance techniques potentially allow a much wider choice of higher performance devices, supported by a wider range of development tools, which evolve as device families and thus incorporate the latest software development paradigms.

Microcontrollers have little or no built-in fault-tolerance features and, since they are very highly integrated, it becomes an interesting design challenge to protect the features already there (e.g., serial buses, programmable bi-directional I/O pins) while using as few on-chip resources as practical in implementing fault-tolerance. A key constraint is to minimize the amount of external support logic so that the main advantage of microcontrollers' high functional density can be maintained, otherwise it makes more sense to use a microprocessor and design custom I/O.

The architecture presented is intended to handle transient errors and some permanent faults. When viewed from a spacecraft engineering perspective, the subsystem-embedded microcontroller often has a lower permanent failure rate than its host subsystem. If some single points of failure represent a small part of the overall failure rate of the microcontroller, but covering them would significantly add to its complexity, then they may be accepted without significantly compromising the reliability of the host subsystem. Similarly, although we would like to provide uninterrupted computations, an occasional restart of a subsystem is acceptable for most of our applications. Since our fault-tolerance requirements are less stringent than many other applications and our resources are much more constrained, we trade reliability for simplicity [5].

To test the approaches described herein, an inertial measurement unit (IMU) was constructed as an example of a typical spacecraft application and integrated with the microcontroller fault-tolerance testbed.

The techniques described herein are necessarily somewhat *ad hoc* because of the limitations of off-the-shelf devices. We apply well-known fault-tolerance techniques in a different context: How much tolerance can be provided against errors induced by space radiation using redundant microcontrollers with a minimum amount of additional hardware? Most previous work is based on microprocessor designs where the basic computers can be customized by adding a significant amount of circuitry for fault-tolerance, e.g., ECC in memory and custom hardware voters [6]. We achieved substantial improvements in coverage using only an external power switch, isolation resistors, and a couple of low-density programmable chips. We attempted to choose fault-tolerance techniques that apply to a wide range of microcontrollers and to take maximum advantages of their common features, e.g., serial buses for intercommunication of messages for voting and programmable bi-directional I/O pins that can be used to provide a level of I/O protection.

We will discuss very simple techniques which use small amounts of microcontroller resources to yield substantial improvements in fault-tolerance. If one considers that these may be embedded as controllers in a dozen or more subsystems on a small power-constrained spacecraft, the resource limitations become clear.

2. Physical Architecture

The block diagram of a spacecraft subsystem (e.g., an IMU) containing a fault-tolerant set of microcontrollers is shown in Figure 1. Redundant microcontrollers are voted to create an error containment region. At the top of the figure, the subsystem interface is shown as just a source of power and communications. The I/O of multiple microcontrollers are combined and protected by I/O isolation and connected with the sensors and actuators. An External Conflict Resolver circuit may reset or power-cycle the individual microcontrollers to support recovery.

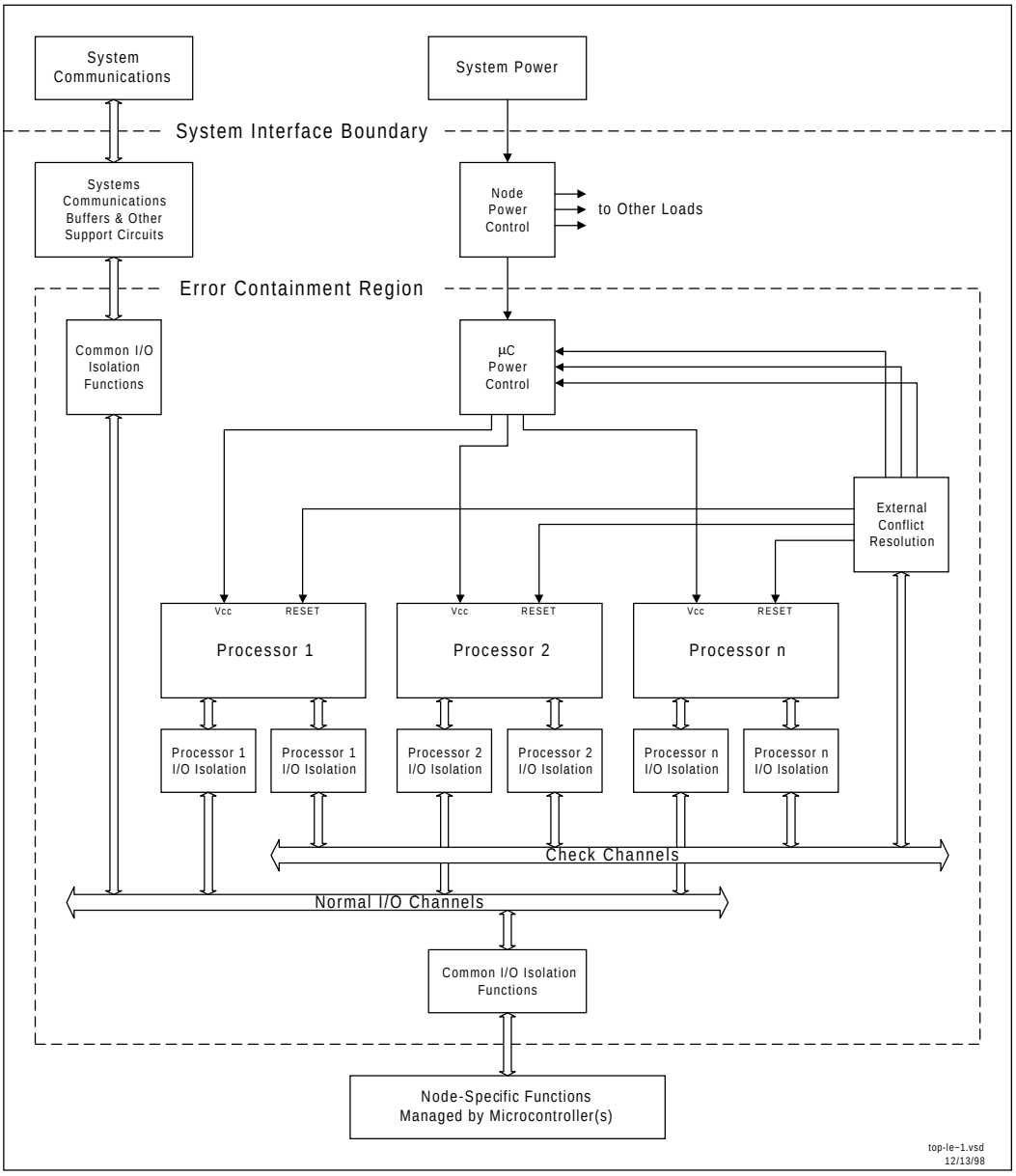

Figure 1. Physical Architecture.

During normal operation, one microcontroller is the Master of the system, while the others provide redundant computation and voting opinions as Checkers.

The Master and Checkers are loosely synchronized and execute identical application programs, periodically calling support functions which implement the fault-tolerance features. If a microcontroller disagrees with its peers, it can be commanded offline and brought back as a Checker if it can be successfully restarted. Devices are not statically assigned, so the operating mode of each device is fluid [7].

Most of the I/O pins of the multiple microcontrollers are bussed together – corresponding pins are connected to a common circuit node through isolation elements (resistors); only the signals governing the external conflict resolution are unique to each processor. This approach simplifies interconnection and makes the architecture easily extensible. Some of each microcontroller's I/O pins are consumed implementing the Check I/O necessary for fault-tolerance; the remainder are available to the application as Normal I/O.

The Check I/O pins provide three functions supporting fault-tolerance: 1) a *Master Channel* (2-pin I^2C serial bus) for data communications between processors, 2) an *Operating Mode Channel* (6 pins) to allow each processor to broadcast one of the three operating modes to the other two, and 3) four *External Resolver Control Signals* (4 pins) from each microcontroller to request recovery actions.

The *Master Channel* is the primary data path for communications between the Master and others. It is used by the software fault-tolerance functions to exchange I/O values vote and develop consistent data. It is also used to exchange other internal state data (e.g. for roll-forward) , and control commands.

Each microcontroller generates a two-bit quasi-static *Operating Mode* signal indicating to its peers its level of participation in the system. The three modes are Off-line (00), Checker (01), and Master (10).

The external conflict resolution block (or simply Resolver) serves as a hard core recovery unit. When normal communications and recovery techniques using the Master Channel fail, this element provides an independent means for establishing a valid configuration. The Resolver can either reset or power-cycle one or more microcontrollers. Table 1 and Table 2 enumerate the control signals sent from each microcontroller to the Resolver.

Table 1. Resolver Action Requests.

	Requested of State or Action
00	Idle Offline. Unpowered or not voting. No action requested.
11	Idle Online. Expected to vote but no action requested.
01	Vote to Initiate Resetting.
10	Vote to Initiate Power-Cycling.

Table 2. Resolver Device Selection.

DevSel	**SCP Usage**	**TMR Usage**
00	Self/All	Self/All
01	N/A	Right
10	N/A	Left
11	All (Both)	Both Peers

The Action Request bits indicate the state of the microcontroller and whether an action request is being made. When an action is requested, the Device Select bits specify the microcontroller(s) to be acted upon. An action request by a

microcontroller directed at itself is immediately taken, but action requests directed at other microcontrollers must be concurred by a peer for the requested action to be taken. The "reset self" combination of Action Request and Device Select is not required since this action can be taken by any device without the aid of the Resolver. Thus, this input combination is redefined to mean "power-cycle all."

Finally, the Microcontroller Power Control block of Figure 1 allows the devices to be power-cycled by the Resolver and also provides some SEL mitigation functions. More details about the hardware may be found in [7,8,11].

3. Application Characteristics

Most microcontroller applications are structured around some real-time task which is executed frequently and periodically [9]. A "real-time frame" is initiated by a real-time interrupt (RTI) and will have many sub-tasks. In the prototypical application investigated, the R-T frame is 62.5 ms long, being initiated by a 16 Hz interrupt. A low-level I/O task samples three rate gyros at 10.8 kHz, and 220 samples are aggregated for each gyro in every frame. After data are gathered, some output results are generated; these may be outputs which will drive physical devices, telemetry data returned to a higher-level (system) application, or state data which will be used as the initial conditions of the next R-T frame.

A number of fault-tolerance processes are required to support this application processing. Before outputs are generated or propagated to the next R-T frame, they must be checked for validity to prevent error propagation. Data may also be checked at various user-defined intermediate points to limit error detection latency. The asynchronous detection of Single-Event Latchup (by external hardware) will result in a device being automatically power-cycled. The loss of any processor, such as due to a peer-commanded reset following a vote should not interrupt processing but is unavoidable in some pathological cases which result in a system restart. Finally, time at the end of the frame is reserved for communicating state data to a previously reset or power-cycled processor to rejoin the voting ensemble as an online member.

4. Fault-Tolerance Functionality Overview

Figure 2 expresses system behavior in a statechart containing four state machines. One of these, SYSTEM_STATE, represents system state as a whole; each of the *n* microcontrollers operates according to a very similar state machine (PROCESSOR_n_STATE). For each of the *n* processors, a state machine determines the operating mode of that processor. The aggregate of these operating modes results in a system state which is the mastership of the system.

Initially OFF, the system transitions to a RESET state as soon as any processor is powered. Processors automatically transition from their individual PROCESSOR_RESET states to running code. Initialization codes are executed which perform self-tests, configure I/O and start the processor fault-tolerance (PFT) functions. The first PFT function is to select one of the processors as a Master. When at least two processors are online and have agreed on a Master, the

user application starts running. At user-specified locations in the application code, calls are made to PFT functions which implement the fault-detection and recovery functions.

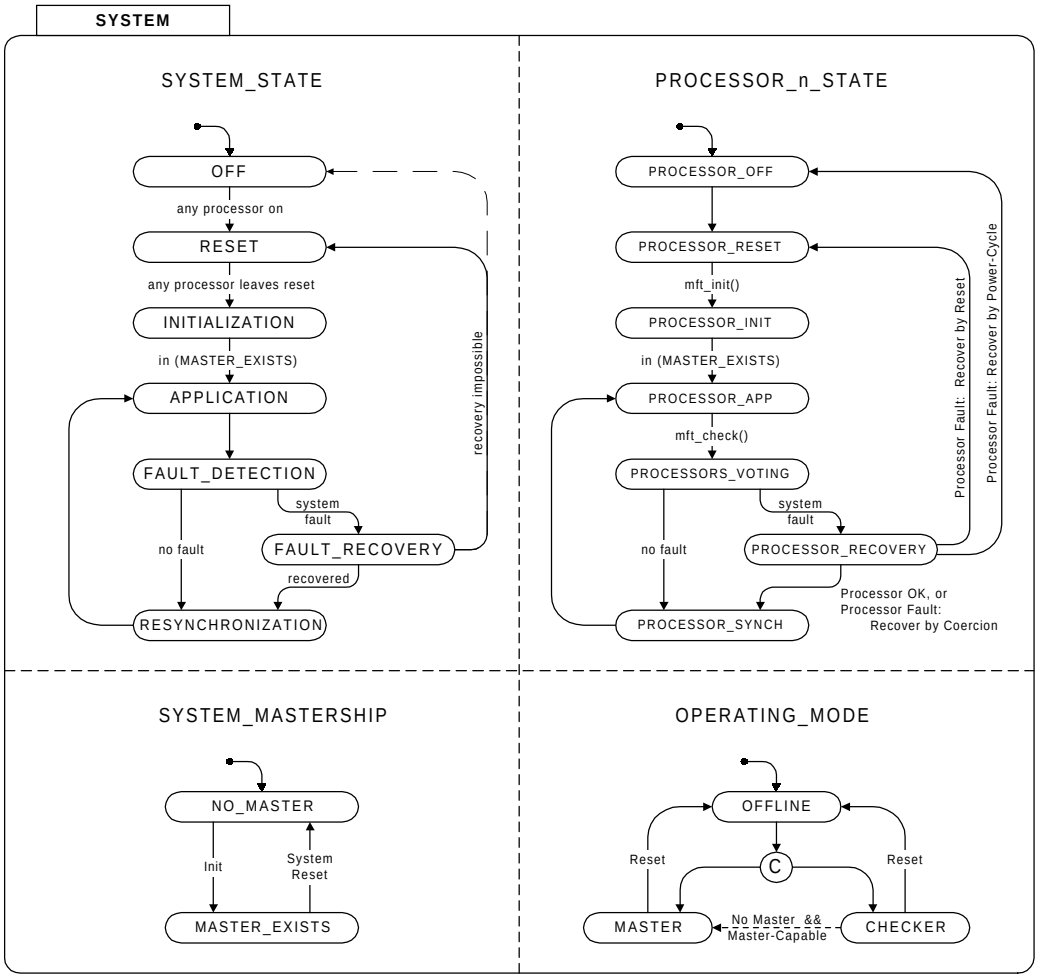

Figure 2. System-Level Statechart.

Fault-detection and isolation are primarily covered by software-implemented voting algorithms. In voting, the Master transmits a packet of data to each Checker. Since both Master and Checkers run the same code, they remain roughly synchronized. Thus, the Checker will reach its voting point at the same time as the Master and start expecting a check-packet. Each Checker computes its own check-packet, compares the received packet with its own, and reports equality (OK) or inequality. If no error is detected, the processors are resynchronized and the application continues.

If an error is identified, correct data are transmitted to the disagreeing member. If a processor does not receive an expected message (identified by a timeout), it can generate an action request to restart the expected sender. If a Checker is restarted, processing continues and the Checker is resynchronized at the next rejoin point specified by the application. If a Master is restarted, the Checkers

must detect the corresponding change in the Operating Mode Channel, and have the highest numbered Checker take over as Master to prevent processing disruption. One reason that the Master may go Offline is that current-sensing hardware in the external power control can detect an SEL in the Master at any time and will then immediately power-cycle the Master to clear the latchup. Because the Master is the sole generator of some outputs (e.g., high-speed clocks), an output glitch will occur if the application waits until the next cooperative voting point to deal with the error. Thus, the system must identify this case, select a new Master, and the new Master must reconfigure its outputs to fill the void. If all the processors are restarted, they must revert to a system restart point in the application software.

5. I/O Isolation and Fault-Masking

Circuit isolation is essential in this design: (i) to prevent catastrophic shorts, and (ii) to make it possible to remove power from a module for latchup circumvention. Isolation is provided with resistors in series with the bussed I/O pins of the microcontroller as shown in Figure 3. Pull-down resistors are also employed so that the common output becomes the logical OR of the individual signals.

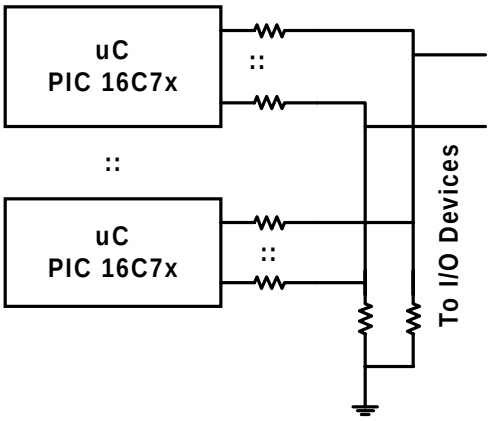

Figure 3: Circuit Isolation

Power must quickly be removed when a latchup is detected, but current from external logic signals leaking through input protection diodes can parasitically power the chip enough to sustain the latchup. In this design when power is removed, Vcc is shorted to ground, and the series resistors divide logic voltages to an acceptably low level.

In addition to the conventional ways of protecting outputs (voting outputs at the actuators or strobing data into radiation-resistant latches in the actuators immediately after a successful vote), a novel low-cost masking approach was developed to protect outputs against single-event upsets. Since the isolation circuits between microcontrollers produce a logical OR of their individual output

signals, if any microcontroller produces a "one" on a bussed output line, the result will be a logic "one". A bi-directional microcontroller pin is implemented with two flip flops one which controls the data direction (input or output) and one which contains output data (one or zero). In order to output a "one," both flip-flops must be set appropriately. In order to make a "zero" farther away from a "one" than a single bit-flip, the output value can be configured as zero and the pin set as an *input* with an external pull-down resistor. In this way, it takes two bit-flips to generate an erroneous output.

Latent faults in port registers can lead to double-bit errors, and output errors can be generated if a machine fails in such a way as to output a "one" before its error is detected by its neighbors. Periodic scrubbing (including reading in and comparing output values) can be used to detect errors of this type. This, and other pathological cases associated with high speed inputs cannot be discussed here due to space limitations, but they are discussed further in [11].

6. Detection, Reconfiguration and Recovery

Fault masking only applies to output ports. In general, errors are identified and corrected through software-implemented detection, reconfiguration and recovery. The detection process begins with software voting. Following diagnosis, the system is reconfigured if necessary by taking the errant device offline (isolating it from the rest of the system) and possibly selecting a new Master. The system recovers by forcing errant devices to restart.

6.1. Normal Operation

During normal operation, processors are loosely synchronized so the identical application code running on each voting processor reaches a checkpoint function at approximately the same time. The Master manages the voting process by first passing data to the Checkers and getting responses from them.

Voting by Exact Match – The Master transmits a block of data (or a shorter syndrome) to a Checker which has computed its own check-packet and expects one from the Master. The Checker compares the received block (or syndrome) with its own block and reports equality (OK) or inequality in an encoded status message.

Inexact Inputs – For data which are derived directly from noisy sources, such as A/D converters, an exact match is generally impossible. Here, the Master passes its value to the Checkers and, if it is within an acceptable tolerance of the value they sampled, they signal agreement and use the Master's value. Other algorithms may also be implemented. Greater accuracy can be obtained for analog values by selecting the middle value as the best or by computing an average (after discarding outliers). Details of implementing voting using the I^2C bus are described in [11].

"**Meta-Measurements" –** In microcontroller applications, it is common to perform simple processing on a possibly large number of samples. For example, the reported angular rate of a gyro may be the normalized average of many rate measurements. In the testbed application, three gyros are sampled at 10.8 kHz, and 220 sample values from each gyro are aggregated to form a reported angular rate. This rate is reported to the host system 16 times a second. In this case, it is completely impractical to vote each individual sample – but it is also unnecessary; an aggregated value is essentially a single measurement from an abstract input device. Since the values are simply summed and averaged, the resulting averages can be dealt with by inexact voting as described above. This is discussed in more detail in [11].

Placement of Checkpoints – The simplest approach of voting after every input may be prohibitively expensive and unnecessary. However, data which will be output to ports, intermediate results which can lead to control-flow divergence, and state variables which are "output" by one iteration of processing (e.g., a periodic calculation) and then used as input to the next iteration must all be voted. It is probably wise to simply "vote early, vote often" with only as much moderation as dictated by the computational and communication resources available.

6.2. Control Errors in Checkers

Data-only upsets are fairly benign but an upset can cause a processor control error, e.g., causing it to jump erroneously or get locked up looping on completely invalid data. A significant effect on a Checker is for it to not reach its next voting checkpoint thereby precluding the Master from exchanging data with it. Timeouts are used by the Master to diagnose this condition.

During voting, communications timeout durations are set to accommodate the maximum expected clock skew between processors and any additional delay which results from application interrupt processing (e.g., handling real-time tasks) and from differences between Master and Checker fault-tolerance functions. If a Checker fails to communicate within the allocated time, either the Checker has failed or neither Checker was expecting the Master, in which case the latter is probably in error.

In the former case, the Master can communicate with the functioning Checker and the data to be checked can be at least tested for validity (as a self-checking pair). Additionally, the Master will request that the functioning Checker participate in voting the failed Checker to the Offline state using the External Resolver.

6.3. Control Errors in the Master

Just as a Checker may get lost, so may the Master. The process for dealing with a Master is necessarily quite different because the cooperative process of

asking for help cannot be effected without the control of the Master; the Checkers are on their own.

Just as the Master sets communication timeouts, so too do the Checkers. If the Master fails to contact a Checker and the Checker believes itself to be healthy, it must conclude that the Master is in error. Without waiting for some confirmation of this error, it signals to the Resolver its desire to reset the Master. If the other Checker observes the same phenomenon, it will also have voted for an external reset and the Resolver will reset the Master.

When the Master is reset or power-cycled, its Operating Mode bits will go tri-state and be pulled down to (0, 0). This transition indicates a Master-less system and must be identified by the Checkers; if any outputs are generated solely by the Master (as already described), loss of the Master must immediately be followed by the selection of a new Master and its outputs properly configured. An interrupt may be generated externally by the all-zeroes case on the Master Operating Mode signals. Microcontrollers which can generate interrupts on input port pin changes do not need such external hardware; the PIC16C73 can flag a change on any of the high-order four bits of port B with an interrupt.

When the system loses its Master, the change is detected immediately if an interrupt is used (as is necessary if there are Master-only outputs) or when the next data checking operation occurs. In both of these cases and when the system is first started, a Master must be selected.

When the processor is restarted, the initialization process first checks whether a Master is present. No action is required if one is already present; the processor remains Offline until the Master commands it to join the ensemble. If there is no Master and the device is Master-capable, it will attempt to claim membership. A potential race condition exists between Master-capable devices so an additional step reverts to Checkers those peers with lower priority than that of the highest priority Master-claimant. Once a Master has been selected, it begins transmitting its system state to Offline members and follows the data with “join” commands. If a peer is ready, it will accept the incoming state data and become a Checker when it sees the join command. An example timeline of this process is shown in Table 3. It also includes the case where a restarted checker is brought back on-line.

Note that when a Master goes offline, the system may not operate non-stop; the Mastership selection process takes a few microseconds and glitches may be observed on Master-only high-speed outputs as the responsibility for their generation changes from the old Master to the new one. Any deleterious effects of this outage must be dealt with at the application level.

Table 3. Master Selection Process Timeline.

Conditions or Event	Operating Mode		
	P0	**P1**	**P2**
Initial condition after system reset.	o	o	o
Two Master-capable processors reach Master selection and claim Mastership.	M	M	o
Second processor observes a higher priority Master (P0) and relinquishes its claim.	M	o	o
Third processor (P2) initializes; sees Master; takes no action.	M	o	o
Master (P0) transmits state data to Offline peers.	M	o	o
Master sends "join" command to peers; they do so.	M	C	C
--- Arbitrary time passes. System operates nominally. ---			
P1 is taken offline; system is unaffected.	M	o	C
--- P1 reset occurs; one real-time frame passes ---			
Master sends state data to P1, requests that it rejoin as Checker.	M	C	C
--- Arbitrary time passes. System operates nominally. ---			
P0, the Master, taken offline; system has no Master.	o	C	C
P1, a Master-capable Checker, observes the Master-less system, claims Mastership, and reconfigures its outputs as Master.	o	M	C
--- P0 reset occurs; one real-time frame passes ---			
Master sends state data to P0, requests that it rejoin as Checker.	C	M	C

6.4. Corrupted Master Channel

The most severe fault is one which renders the Master Channel inoperative. A "babbling" device may cause this but the simplest mechanism is a Checker which simply sets one of its Master Channel I/O lines to an active state thereby creating a conflict for the channel.

Although the Master Channel could be designed to preclude (with high probability) that Checkers cannot take it down, the complexity required to both preclude bad behavior but allow all devices to be Master-capable will violate any notion of minimality. Thus, it must be assumed that there is a non-negligible probability that either the Master or a Checker can make the Master Channel inoperative.

If any processor sees that the Master Channel is inoperative, it cannot know which peer is at fault so after a self-check it requests a reset of both of its peers. In reality, this condition is indistinguishable from a failed Master as seen by the Checkers. If the two correctly-functioning processors identify this condition and are the first ones to command the Resolver, the failed processor will be reset by the Resolver.

However, if the processor which caused the error has also managed to command the Resolver to reset its peers, then when the first correctly-functioning processor requests that the Resolver reset both its peers, the other correctly-functioning processor will wrongly be reset (since the Resolver would have received two votes to reset it). The one remaining good processor will then observe a device going Offline and then expect communication over the Master Channel, either transmitting as the Master or receiving as a Checker. Since this

operation will fail, the good processor requests that the Resolver restart all three processors. Since this design is only intended to mitigate transient faults, the full restart would be expected to clear whatever fault initiated the entire chain.

Clearly in this worst case the system will be taken Offline for a period of time but in doing so it will not permit bad data from being propagated beyond the error-containment boundary.

6.5. Resolver Operation

The implemented Resolver is a synchronous state machine, with the Action Requests and Device Select as its control inputs. If the selected device is "Self" or "All", the action takes place immediately. If another device is selected, two such votes are required to take action against a device. This approach allows a microcontroller which believes itself to be in error to be reset or power-cycled and it allows a single microcontroller to believe that the entire ensemble is beyond help and to restart everyone. But it does not allow any lone device to affect others except when it is included by selecting "all", thus preventing a large class of wayward behavior.

The control inputs are used to control the Resolver state machine shown in Figure 4. The [GO_xxx] conditions are derived from the control signals from each processor, but the formulae are beyond the scope of this paper. A processor may be turned off (power-cycled) either as the result of an Action Request (GO_CYCLE) or asynchronously as the result of an overcurrent (IFAULT) being detected. The substates of PROCESSOR_OFF shown with broken outlines may or may not be implemented distinctly. The ONLINE state contains the substates of MASTER and CHECKER.

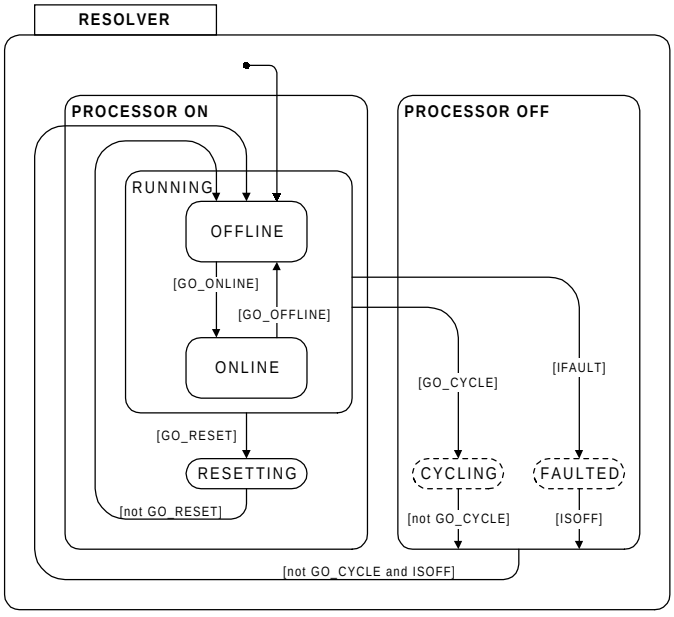

Figure 4. Statechart for Single-Processor Restart.

Figure 4 is principally comprised of two smaller state machines, one for power on/off state and one for reset state. The External Resolution - Power Cycling (ERP) state machine has only two states, Processor On and Processor Off, represented by a single register bit which controls the processor's local power switch (a totem-pole FET pair). The External Resolution - Reset (ERR) state machine similarly has only two states, Resetting and Running, with its single output bit tied directly to the microcontroller's reset pin. As is evident in Figure 4, the state of the ERR machine is irrelevant if the microcontroller is off. The three copies (one for each microcontroller) of Figure 4 are implemented in two PALs, one for three ERP machines and one for three ERR machines. The resetting state machine (ERR) outputs a reset pulse which satisfies the timing requirements of the target processor. Timing details are described in [11].

The power-cycling state machine (ERP) is similar to the ERR machine except that it awaits positive confirmation of the desired effect. By watching for the ISOFF flag which indicates that the current flowing into the microcontroller has dropped below a hardware-set threshold, a closed-loop control is implemented which obviates the need for precise control of the state machine's clock frequency.

7. The Experimental Testbed

In order to produce an outcome valuable to the spacecraft avionics community and to provide a testbed for evaluating the effectiveness of the techniques, a prototypical application was built that is representative of a typical spacecraft subsystem, specifically a 3-axis inertial measurement unit (IMU). This application is sufficiently complex to provide insights into real problems while sufficiently simple that its implementation was not overly distracting. The supporting testbed accommodates three microcontrollers to implement fault-tolerance and thus provide examples of different processor (duplex and triplex) configurations. The application example forces many I/O requirements to be addressed, including bi-level and analog voting, pulse train generation, event timing, and serial communications. The testbed and IMU application use the Microchip PIC16C76 and '77 [2]. Its functionality, while relatively limited, is sufficient to implement the chosen applications but these same limitations force a frugal approach to fault-tolerance – it would be very easy to use all the I/O pins just implementing fault-tolerance.

A block diagram of the testbed environment is shown in Figure 5.

A circuit board was constructed for the three redundant microcontrollers with a daughter board for the External Resolver. PIC in-circuit emulators were substituted for the three microcontrollers to provide ease of controlling and monitoring the processors while maintaining fidelity as to their behavior. A laptop PC was used to inject faults and monitor the state of the External Resolver. A photo of the testbed is shown in Figure 6.

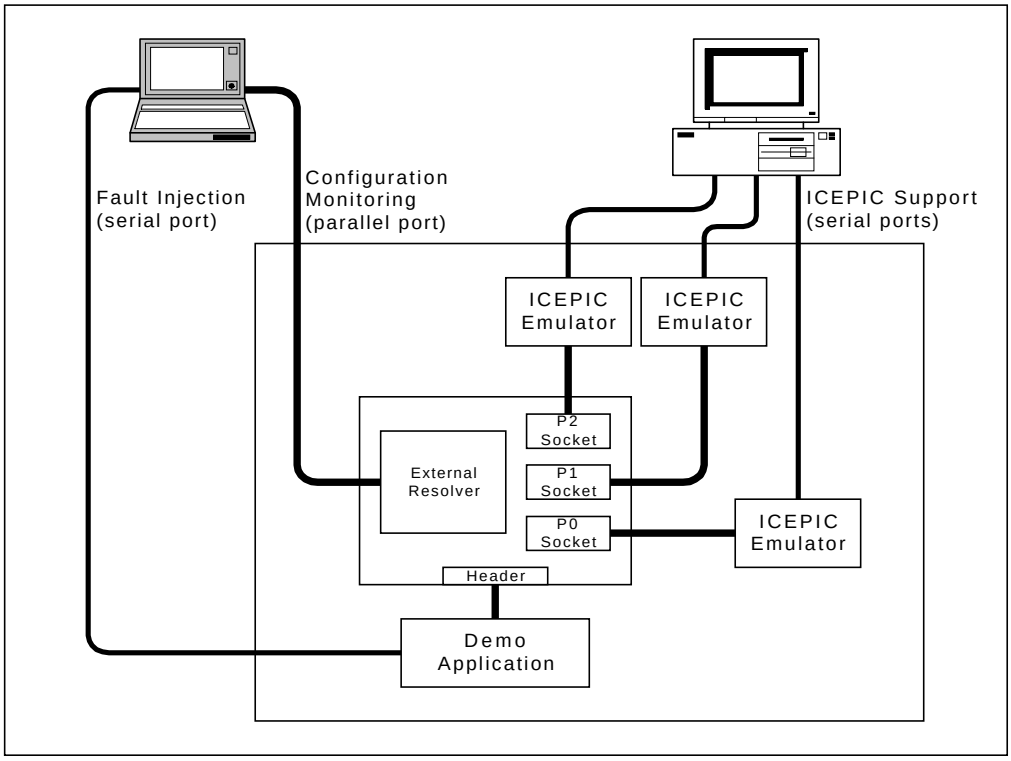

Figure 5: TMR Testbed Hardware Configuration

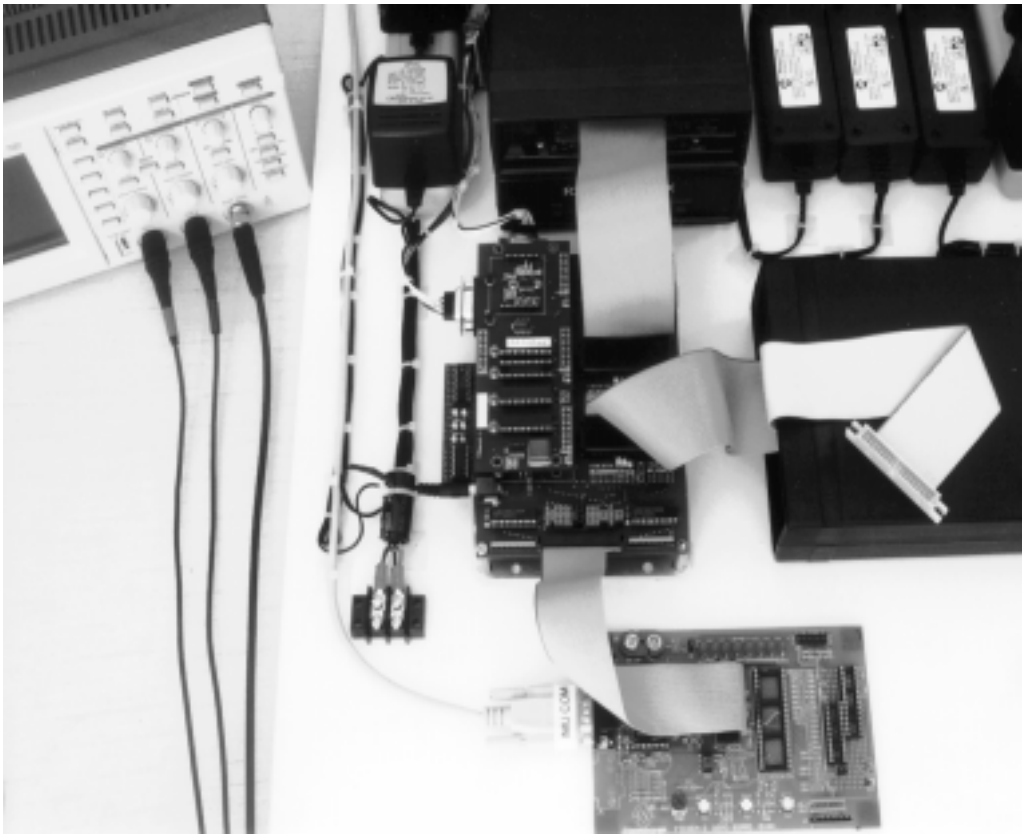

Figure 6. The Testbed

8. Fault Injection

Simulated SEUs were injected into the system and its response monitored to determine if the recovery mechanisms worked and to get insights into the coverage of the system. To inject faults, each processor of the fault-tolerant system contains an interrupt service routine (ISR), which receives messages from the asynchronous serial (RS-232) port. An external support program randomly generates and sends messages consisting of two bytes specifying the processor (1-3), byte (0-255), and bit (0-7) to upset. All such generated "strikes" are upsets consisting of flipping the state of one bit. One, two or all three processors can be the targets of this random process. During the time that faults are injected into the system, statistics are gathered to characterize the system's responses.

8.1. Fault Response and Monitoring

Each injected fault can be viewed as an experiment that can have many possible outcomes of decreasing desirability:

1. No effect.
2. Latent effect. A fault creates a situation which will cause an error only after some other fault occurs.
3. Computing continues correctly without significant interruption. The system correctly identifies a failed device and the operational devices reset the errant one; the two good devices continue after a brief delay.
4. Computing continues correctly after an automatic retry. Recovery is effected by a system reset but the user data is unaffected; the system recovers with no data loss but a slightly higher temporal impact than in case 3.
5. System output is flagged as invalid. Computations are correct and somehow corrupted before the user sees their results, but the incorrect results are easily identifiable (e.g., using a checksum). System state is not lost.
6. Computing continues correctly after a rollback. Recovery following a system reset requires the use of previously-checkpointed user state data; the data since the last checkpoint is lost (e.g., the data taken during one real-time frame).
7. Computing continues following system re-initialization. Volatile user data is lost.
8. System outputs incorrect results. Data errors which are not easily identifiable by the user are produced.
9. Computing ceases. The system fails. It stops computing or becomes unstable (and thus useless).

The monitoring program (which is part of the fault-injection program) uses externally-visible information to identify behavior in the above categories and

accumulate statistics to characterize the probability of each response. From these statistics, the effectiveness of the techniques can be determined.

The externally-visible indications of behavior which the monitor observes are: the system's Operating Mode configuration; the occurrences of hard resets; the integrity of the serial channel which communicates computational results to the monitor; and the computational results themselves.

Operating Mode Configuration Monitoring – The system Operating Mode configuration is the easiest indicator to monitor. The monitoring program samples the Operating Mode at about 430 Hz, keeping a histogram of how long the system dwelled in each mode and the number of times a mode was entered.

Reset Event Counting – The generation of a reset action by the Resolver is the best indication that a non-recoverable error took place. Because these actions are too short-lived (10 μs) to be easily monitored externally, the spare PAL socket on the Resolver daughter board was populated with a device programmed to capture the resets and hold them until the external monitor could sample and rearm it. The monitor tallies the number of times that one, two or three resets are observed during one sample time (at the 430 Hz rate).

Behavioral Model Monitoring – To trap errors which were not caught by communications software, a behavioral model of the test application was incorporated into the monitoring program. This model implements algorithms similar to the IMU application and propagates them for each data packet received. It then compares the received data with the model to determine if the application is behaving properly (i.e., according to the model). Observable system resets, and therefore interruptions in operation, are identified when the application output reverts to its initialized state. Time, rate and position errors are counted as data errors when the application output lies exceeds an allowed tolerance of the model.

8.2. Test Runs, Results and Analysis

Exploring the significant dimensions of a complex system requires more than a single demonstration run. Approximately 80 multi-hour test runs were performed on the system, starting from the time the software became relatively stable. As would be expected, the majority of these runs resulted in discovery of a number of "interesting behaviors" which required subsequent software improvements. Many of the insights gained from this process are documented in [11] "Problems Encountered." Table 4 summarizes about 15 runs made near the end of the experimentation phase. In these tests, both TMR configurations and pairs of microcontrollers operating as self-checking pairs (SCP) were exercised.

Table 4. Summary of Test Runs.

				Errors		Coverage		Effectiveness	
Run	**Cfg**	**Configuration Notes**	**Strikes**	**Resets**	**Data**	**Reset**	**Data**	**Reset**	**Data**
A	SCP	no retry, 8-bit checksum	33650	2013	1	94.0%	99.994%	88.0%	99.988%
B	SCP	with retry, 8-bit checksum	59191	2540	3	95.7%	99.993%	91.4%	99.986%
C	SCP	same as B, hits only on P1	28462	1105	0	96.1%	99.996%	96.1%	99.993%
D	TMR	with retry, 8-bit checksum	30552	538	6	98.2%	99.977%	94.7%	99.931%
E	SCP	with retry, 16-bit checksum	47669	1814	0	96.2%	99.998%	92.4%	99.996%
F	TMR	with retry, 16-bit checksum	44846	762	13	98.3%	99.969%	94.9%	99.906%
G	TMR	cfg F, w/o P2 participation	16199	622	4	96.2%	99.969%	92.3%	99.907%
H	SCP	cfg E, with user checkpoint	49691	1003	3	98.0%	99.992%	96.0%	99.984%
I	SCP	cfg H, with 2x strike-rate	77673	1260	2	98.4%	99.996%	96.8%	99.992%

An estimate of the non-coverage with respect to system resets or data errors is the number of resets or data errors recorded divided by the number of faults (strikes) injected into the system. Since some of the test runs resulted in small numbers of data errors (e.g., zero), the values shown for data coverage (and effectiveness) are the mean likelihood estimates. "Effectiveness" is coverage relative to a single-device system; since the number of injected faults are spread over two or three target processors (for SCP and TMR), the number of faults injected into a single device is one half or one third as large. Thus, the effectiveness is coverage computed by dividing the number of faults (strikes) by the number of processors participating in the run.

In run "A" all internal resets (2013) result in observed Reset Errors. With subsequent runs, "retry" and "user checkpointing" were used to reduce the number of externally visible resets.

The simplest approach is to simply "retry" a voting operation after a system reset. Since many errors are due to control errors, a very low overhead improvement is to allow a resetting system to immediately retry the voting process. If state data are unaffected, the vote will succeed and the operation may continue. Obviously, if the state data were affected, the system must reinitialize. The improvement due to this approach is about a 30% reduction in system errors (case A to case B).

Instead of reinitialization, the state data may be checkpointed (by making a copy) and following a retry failure, the saved data may be used (a "rollback" recovery). This approach has higher coverage at the expense of significantly higher memory resource requirements (and consequently could not be tested in the TMR case). The improvement over retry only is an additional 50% reduction in system errors (case E to case H).

Specific details of the tests can be found in [11].

It is interesting that SCP was uniformly more robust than TMR. This is undoubtedly due to the larger effective cross section presented by TMR due its complexity. It may be that the very simple application program was dominated by the voting software. If so, more complex applications (on more sophisticated processors) might not show this disparity. However, it is apparent that there is a trade-off which must be considered lest the apparent benefit of TMR be offset by its substantially higher complexity.

9. Summary

This paper has described steps toward a generic approach to implementing cost-effective fault-tolerance augmentations of commercial microcontrollers in spacecraft control systems, focusing on the transient error recovery needed in a space radiation environment. The work explores how much fault-tolerance can be implemented in a minimal design that preserves the high functional density advantages of microcontrollers – without taking the costly step of implementing microprocessor designs with extensive supporting interface circuitry. While there has been extensive research and development of systems which provide extremely high levels of fault-tolerance, including Byzantine resilience, they are expensive custom designs which exceed the limited resources available for spaceborne applications [5].

The described techniques are simple ones that allow multiple microcontrollers to be connected in multi-processor fault-tolerant configurations. A hardware testbed and software prototype allowed experimentation with variants on a set of core architectural concepts. For the given experimental fault set, coverage with respect to propagation of bad information due to simulated single-event upsets was demonstrated to be higher than 99.99%. The highly integrated nature of microcontroller chips make it impossible to access internal variables (e.g., the processor-memory bus) for experimental fault insertion. Thus these experimental test results based on software-induced disturbances must be viewed as preliminary. Validation in a high-energy radiation test facility is needed to obtain more accurate coverage estimates and to obtain an increased level of confidence.

10. Acknowledgments

This work was sponsored by the Office of Naval Research, under grant #N00014-96-1-0837 at the University of California, Los Angeles.

11. References

[1] "8XC196Kx, 8XC196Jx, 87C196CA Microcontroller Family User's Manual." Intel Corporation, June 1995.

[2] "PIC16/17 Microcontroller Data Book." Microchip Technology, Inc. 1995/1996.

[3] A. Holmes-Siedle, L. Adams. "Handbook of Radiation Effects." Oxford Science Publications, Oxford, 1993.

[4] G. C. Messenger, M. S. Ash. "The Effects of Radiation on Electronic Systems." Second Edition. Van Nostrand Reinhold, New York, 1992.

[5] R. E. Harper, J. H. Lala, J. J. Deyst. Fault Tolerant Parallel Processors Overview. FTCS-18, pp. 252-257. 1988.

[6] T. Takano, et. al., "Fault-Tolerance Experiments of the "Hiten" Onboard Space Computer ," FTCS21, pp. 26-33, 1991.

[7] D. W. Caldwell, D. A. Rennels. “A Minimimalist Hardware Architecture for Using Commercial Microcontrollers in Space.” 16th Digital Avionics Systems Conference, Irvine, CA. 28-30 Oct 1997.

[8] D. A. Rennels, D. W. Caldwell, R. Hwang, M. Mesarina. “A Fault-Tolerant Embedded Microcontroller Testbed.” 1997 Pacific Rim Fault-Tolerance Conference, Taipei, Taiwan. 15-16 Dec 1997.

[9] H. Kopetz, et al. Distributed Fault-Tolerant Real-Time Systems: The MARS Approach. IEEE Micro, February 1989.

[10] S. G. Frison, J. H. Wensley. Interactive Consistency and Its Impact on TMR Systems in Dig. Int. Symp. Fault Tolerant Computing, FTCS-12, June 1982, pp. 228-233.

[11] Douglas Caldwell, "Minimalist Fault Masking, Detection and Recovery Techniques for Mitigating Single Event Effects in Non-Radiation-Hardened Microcontrollers", Ph.D. Dissertation, UCLA Computer Science Department, Los Angeles, CA, June 1998.