A MINIMALIST HARDWARE ARCHITECTURE FOR USING COMMERCIAL MICROCONTROLLERS IN SPACE

D. W. Caldwell[*], D. A. Rennels[+]

[*+]**Jet Propulsion Laboratory**
[*]**Avionic Systems Engineering Section**
[*+]**Avionic Equipment Section**
4800 Oak Grove Drive, Pasadena, CA 91109
doug.caldwell@jpl.nasa.gov

[+]**University of California, Los Angeles**
School of Engineering and Applied Science
Department of Computer Science
Los Angeles, CA 90024
rennels@cs.ucla.edu

ABSTRACT

Microcontrollers provide very dense functionality for embedded applications ranging from telephones to automobiles. The acceptance of these devices for space applications has been hindered by their manufacture which often uses multiple semiconductor fabrication techniques and thereby compromises radiation tolerance. If such concerns could be mitigated, microcontrollers would provide a substantial increase in performance for builders of spacecraft electronics.

This paper presents hardware considerations for using commercial microcontrollers in space applications. The motivations for starting with commercial devices and the concerns associated with their use are presented. The advantages of software versus hardware voting schemes to mitigate single-event effects are discussed. Interprocess communications approaches and schemes for improving I/O robustness are presented.

INTRODUCTION

Microcontrollers are highly integrated computer systems on a chip: a processor and various support functions such as program memory, scratchpad RAM, discrete I/O, A/D converters, serial communications, counter/timers, and watchdog timers (WDT). While some microcontrollers have been used in space, their application has been more like that of small computers augmented with external memory and I/O devices; very inexpensive commodity devices have not been widely used because of their lower radiation tolerance.

Our research explores how to use commercial devices cost-effectively to support distributed, modular spacecraft architectures. Rather than develop a single representative system, we seek to create a "toolbox" of techniques which can be broadly applied to many microcontrollers and which provide a designer with approaches that can be tailored for a particular application. Hardware minimization is balanced with reliability gains; neither is considered the sole objective. We seek solutions which have obviously lower implementation and deployment costs compared with the state-of-the-art, and which have real-estate (mass), power and reliability characteristics similar to those of current design practice.

This research considers primarily the space single-event effects environment, where a high transient error rate is expected and where latchup conditions may occur that require a microcontroller module to be immediately powered down to prevent permanent failure. We are attempting a minimalist approach where fault-tolerance must be applied at minimal cost whilst providing fault recovery under difficult conditions. This paper presents a work in-progress; we have spent the last nine months on the design of a fault-tolerant microcontroller node, and we will soon start the implementation of designs to validate functionality and fault-tolerance.

MOTIVATION

Most discussions of the desirability of using commercial components focus on the low cost of such devices, but this is only a small part of the picture since parts typically contribute a small fraction to the cost of a fielded space system.

The primary motivation for using commercial microcontrollers is their high functional density and low power. Rad-hard microcontrollers must generally sacrifice functional density since the fabrication technologies required to implement the different on-chip functions such as program memory implemented with EPROM cells and A/D converters are generally incompatible with rad-hard processes. Also, the plethora of device types which allows the designer to choose an appropriate set of I/O functions for a given application could not be replaced with rad-hard devices owing to their high non-recurring development costs.

Additionally, commercial devices are supported by a wide range of development tools, which evolve with device families and thus incorporate the latest software development paradigms.

Systems Considerations

<u>Distributed vs. Centralized I/O Signal Processing</u>. Historically, many simple sensors and actuators are serviced by one set of front-end electronics; in the degenerate case, a central computer supports I/O on its local bus and signals are distributed using the system cable harness. This approach incurs significant (usually unquantified) cost penalties for system engineering, integration and test due to the cost of managing the dedicated, point-to-point interfaces; each interface must be defined, tracked and tested. Although uniqueness is an intrinsic property of low-level devices, allowing this uniqueness to propagate throughout the system substantially increases system complexity.

Distributed I/O functions and associated front-end or reactive processing can be managed more efficiently, at higher levels of abstraction, and with more standardization; providing, for instance, engineering-units conversion and packetization of data for serial communication using a standard protocol. Local processing can reduce transmitted data volume, and can provide dramatically lower latency than can be accommodated by a central computer with multi-tasking software.

In contrast to this approach, there is a disturbing tendency toward placing in a central computer virtually every function which can be implemented in

software. This strategy arises from the belief that it simplifies software management and reduces computer cost and power. In fact, it probably has the reverse effect. By forcing software elements with disparate needs to run in the same environment, development costs are increased. For example, both Mars Pathfinder and Deep Space One perform power management in a central general-purpose computer, relying on system software to manage low-voltage situations on the main power bus. In both cases, there is justifiable concern about the speed with which the software can react to critical situations. To combat software costs, higher performance computers are required to provide larger margins so as to allow processing cycles for the inevitable cross-coupling interactions which will inevitably be discovered.

<u>Implementing Distributed Functionality</u>. Local data handling requires some form of data processor. Small nodes may implement the requisite functionality as a relatively simple state machine in a field-programmable gate array (FPGA) or as software on a microcontroller.

The acceptance of FPGAs has dramatically changed the spacecraft avionics development process, allowing the creation of complex application-specific logic which can be readily changed, thereby reducing parts count and development time. These attributes are also true for microcontrollers. Although a function implemented by an FPGA will generally be much faster than if it is implemented in a microcontroller, far more complexity can be captured in the software of a microcontroller. Functions such as engineering units conversion and sophisticated fault-protection algorithms are simply impractical in FPGAs. Microcontrollers fill the gap between general-purpose computers and FPGAs; they are more like small computers than are FPGAs but their applicability is more like FPGAs than general-purpose computers.

<u>Partitioned Architectures</u>. As on-board software grows increasingly complex, there will be a continuing discrepancy between the capabilities and costs of commercial and rad-hard general-purpose computers. Simultaneously, demand for higher performance will increase. One approach to this problem is to partition the spacecraft information system. Generally, high-level functions such as on-board autonomy and data reduction are where the greatest amount of software

growth is occurring since the demands for these are increasing more rapidly than the basic housekeeping and attitude control tasks. The former functions (autonomy and data reduction) have very weak real-time constraints compared with spacecraft health functions. This property can be exploited by basing the computers which perform weakly real-time operations on commercial high-performance computers, protected with checkpointing and rollback, and accepting the occasional service interruptions caused by transient upsets.

In such an architecture, the spacecraft software functions would be partitioned using a combination of hardware and software techniques; operations critical to spacecraft health and safety would be supported with very high real-time reliability, allowing less-critical, less-real time computations to be performed with less robustness. Performing reactive I/O processing with distributed microcontrollers in one such technique.

Computational Efficiency. It is also worth considering that not all computation is created equal. A general-purpose 32-bit computer like the MFC[1] delivers about 22 MIPS (at 20 MHz) and consumes 7.5W -- an efficiency of about 3 MIPS/W. The 16-bit 80C196CA [1] delivers about two MIPS[2] (at 20 MHz) while consuming 375 mW, yielding 5 MIPS/W -- roughly equivalent to the MFC when considering the types of code which are likely to be implemented on the 196. A rather startling example is the 8-bit PIC16C73A [2] which delivers 5 MIPS (at 20 MHz) at about 60 mW (typical) for a whopping 80 MIPS/W! For a large class of applications, this processor is far more efficient than a central computer. This comparison is tabulated in Table 1.

---

[1] The Mars Pathfinder Flight Computer built by Lockheed-Martin Federal Systems.

[2] The MFC RAD6000 and the PIC microcontroller are both RISC machines, thus the reported values are relatively accurate. The 8xC196 family has a CISC instruction set which is much more difficult to analyze, with instruction execution varying between 3 and 33 state times. The given figure is an estimate based on assuming that the compiler will emit predominately the simpler instructions, as is predicted by the RISC paradigm. The 2 MIPS figure corresponds to an average of 5 state times at 20 MHz.

One might argue that this is not a fair comparison because a spacecraft computer like the MFC has far more capability than a microcontroller. However, both have all the attributes of a "computer" and from the perspective of typical computations on 8-bit sensor or actuator data their processing is largely comparable.

Table 1. Computational Efficiency Comparison.

Manufacturer	Computer	Word Size	Clock Frequency	Throughput	Power, typical	Efficiency, typical
		[bits]	[MHz]	[MIPS]	[W]	[MIPS/W]
LMFS	MFC	32	20	22	7.50	3
Intel	87C196CA	16	20	2.0	0.38	5
Dallas	DS87C520	8	33	8.3	0.15	55
Microchip	PIC16C73A	8	20	5	0.06	83

CHALLENGES

Four decades of fault-tolerance research for space applications provides the designer with a variety of ways to make a microcontroller robust in the space environment [3]. Balancing cost and reliability is non-trivial.

An obvious choice is redundant microcontrollers running identical real-time applications software with lock-step comparison or voting for error detection and correction. While it might appear that lock-step execution and hardware voting is the most straightforward approach, the highly integrated nature of microcontrollers makes this approach questionable. Variations between A/D converters will result in different values being read. Even with lockstep devices, digital edges have finite transition times and each microcontroller has its own input thresholds so edges will be seen at different times. There will be a tendency to diverge even with perfect operation. Also, with more than two processors, it would be valuable to be able bring a wayward processor back on line without stopping the others -- clearly not possible with lock-step operation.

Thus, in our implementation approach, processors run with their own local oscillators, with a combination of

software voting and simple external hardware to combine signals.

Interesting design problems occur because the highly integrated microcontroller provides many functions that cannot be modified, yet must be covered by fault-tolerance. Some of these design problems include:

- Interactive Consistency. Consistent computations for voting or comparison must be guaranteed even though individual microcontrollers may read different values for the "same" input. [4]
- Error Latency. Error latency should be bounded, but there is no access to the internals of the microcontroller and there is essentially no internal checking -- not even memory parity. Thus, periodic internal testing must be interleaved with normal operation.
- Circuit Isolation. In order to use the same external interconnects for varying numbers of redundant microcontrollers, the inputs and outputs are connected (bussed) to all of them. Thus, protection must be supplied against shorts, babbling and back-driving unpowered devices.
- Fast I/O. Real microcontroller applications do more than poll analog inputs and generate an occasional output; they often read high-speed timers or generate output waveforms (e.g., PWM) on time scales inappropriate for software consistency determination. Thus, techniques are needed which allow these signals to be generated correctly without frequent software interactions.

In this research, systems containing from one to four processors are being investigated using two testbeds. One approach is aimed at general-purpose application and high fault coverage. The other, described herein, is tailored toward spacecraft applications; the focus is single-event effects rather than random parts failures, trying to minimize the resources consumed by fault-tolerance features.

Space limitations preclude a detailed description of the software fault-tolerance implementation (to be presented in [5]). The remainder of this paper describes the physical architecture necessary to support such software algorithms.
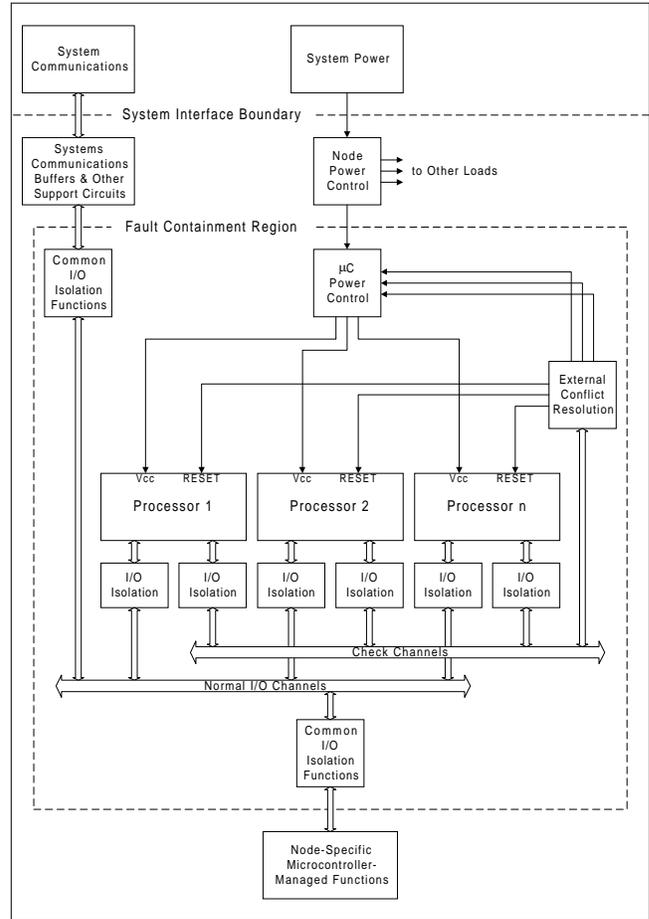

Figure 1. Physical Architecture.

PHYSICAL ARCHITECTURE

Figure 1 is the abstract block diagram of a spacecraft functional element (e.g., an IMU) containing a fault-tolerant group of microcontrollers. At the top of the figure, the system to which the functional element is attached is shown as nothing more than a source of power and communications. The I/O of multiple processors are combined and protected by I/O isolation; external conflict resolution may reset or power-cycle the devices.

The processors (microcontrollers) provide fault-tolerant attributes to the system. The implemented software architecture is slightly asymmetrical in that a *Master* device coordinates fault-tolerance among one or more *Checkers*. The Master might be considered to be the microcontroller of the system while Checkers determine the validity of its computations.

If a microcontroller disagrees with its peers, it can be commanded off-line and brought back on-line if it can be successfully restarted. Devices are not statically assigned so the operating mode of each device (Master, Checker or Off-Line) is fluid.

Normal I/O

The primary use of microcontroller I/O is to interact with the sensors and actuators which define the functionality of the element. This analog and digital I/O, which is provided to accomplish desired node functionality even without fault-tolerance, is called *Normal I/O*. An important part of this research is to describe how Normal I/O is protected against faults and how processing is protected from faulted I/O. As shown in Figure 1, Normal I/O is isolated from the rest of the functional element and from the microcontrollers. I/O isolation of each processor allows individual devices to be turned off in support of SEL mitigation. Common I/O isolation protects circuits outside the fault containment region. A novel method for protecting I/O ports against upsets is described shortly.

Check I/O

Some of the microcontroller's pins are used as *Check I/O* to support fault-tolerant aspects of the node. These provide three functions: the Master Channel, the Status Channel and the Assignment Channel.

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 or other internal state data. It may be parallel or serial, bussed or point-to-point, broadcast or directed. Although the Master Channel may be implemented in various ways, use of a simple serial channel minimizes I/O pins. Sophisticated serial I/O like I^{2}C may be used, as may be UARTs or even software-implemented serial channels. The consumption of I^{2}C or UART hardware resources must be balanced with their simplicity and speed. In all cases, communication latency may be minimized by computing a syndrome (e.g., checksum) over a large block of data and checking only the syndrome, not the data itself.

The *Status Channel* is used for simple signaling which is independent of the Master Channel. Status may be used to signal the validation of data passed on the Master Channel. In the minimalist approach, two I/O pins per microcontroller are reserved for the Status Channel to invoke external conflict resolution when normal communications and recovery techniques using the Master Channel fail. The only mechanisms available for this purpose are individual (or multiple) processor resets and individual (or multiple) power-cycling of the devices. The Microcontroller Power Control block of Figure 1 allows the devices to be power cycled and also provides SEL mitigation functions.

The quasi-static *Assignment Channel* indicates which device will act as Master, which will be active Checkers, and which will be off line. It must satisfy the following requirements:
- A unique master can and will be selected given at least two operational processors;
- The selection scheme is not susceptible to race conditions;
- A master-capable unit rejoining the system will not affect ongoing processing (e.g., by demanding mastership).

The assignment strategy selected is first-come, first-claim with conflict resolution using hardwired node IDs. The I/O requirements may be minimized if some configurations are disallowed. For instance, in a three processor system, operation with any two may be allowed but not with only one. In this case, one unit need not assert mastership since it will always be paired with one of the other two. Thus, the mastership selection algorithm need only select between two master-capable units and resolve conflicts between those two. In this example, only two I/O pins are required to implement the Assignment Channel.

I/O ISOLATION AND VOTING

To allow devices to be turned off in support of SEL mitigation, they must be prevented from being parasitically powered through their input protection circuits. Complete isolation might be effected using an active device like the MAX367 Signal-Line Circuit Protectors [6] but a current-limiting resistor on each I/O pin suffices.

In addition to being isolated, output ports must be voted. Although the Master will ascertain from the

Checkers whether its computed results are correct, it cannot be the sole generator of output because its output port flip-flops are susceptible to upset. Instead, both the Master and its Checkers must output their versions of truth and these must be voted externally. A straightforward approach would use digital voting circuits, but these do not allow ports to be bidirectional and do consume significant real-estate. The minimalist implements an analog voter using the current-limiting resistors required to prevent back-driving unpowered devices, where the majority vote drives the output above or below the midpoint voltage.

This approach is overly simplistic since voltage levels generated under fault conditions are not compatible with standard logic families. A single output fault in a three-processor system will generate a voltage which is either 1/3 or 2/3 of V_{CC}; a single fault in a four-processor system will generate either 1/4 or 3/4 V_{CC}. Although 2/3 or 3/4 of a standard 5V supply (even at 4.5V) is a legitimate TTL logic high, no logic family considers 1/3 or even 1/4 of V_{CC} to be a legal low.

Self-Checking Ports

The problem may be addressed by exploiting the microcontroller's I/O port structure. For most devices, I/O pins are bidirectional and it takes two conditions to output a high level on an I/O pin; the output flip-flop must be a "1" and the I/O pin must be configured as an output. There are three other state pairs which do not result in an active high level being output. One of these states corresponds to neither of the requisite conditions for "high" being met: the output flip-flop contains a "0" and the I/O pin is configured as an input. This point lies Hamming distance two away from the active-one state; if the correct output is zero, two flip-flops of one device must be incorrect to generate an active high. If an external resistor is used to pull the output down when the pin is in the {output=0, direction=input} state, then the only legal high state lies two faults away from a legitimate low state. While pullups are generally preferred in logic design to pulldowns, this stems from the ability of N-channel devices to sink more current than P-channels and thus speeds the high-to-low transition -- a requirement not needed for the relatively slow microcontroller pins used to control the real world. The pulldown resistor also serves to drive the output to a safe state during initialization.

Table 2 shows the four possible combinations; diagonal entries correspond to states with Hamming distance two. As noted in italics, the solution is imperfect; a single bit error can result in an active low output with probability 0.25. Similarly, the input protection circuits of an unpowered device will sink current. Either of these cases will result in a signal which is clearly "high" but not necessarily compatible with a given logic family.

These effects can be handled in one of three ways:
- Use low-threshold input devices (e.g., the TTL-compatible inputs of the PIC microcontrollers or the HCT family, both with $V_{IH} = 2.0$ V);
- Use diode isolation to prevent current from flowing into the microcontroller (although this precludes bidirectional port operation);
- Use transmission gates (analog switches) which are turned off when the microcontroller is unpowered.

The diode network option forms a diode-OR function and could be replaced by an OR gate (with input pullups). Essentially, a number of self-checking registers (the dual bits of the output port) try to generate active ones and any internally-checked port which succeeds sets the output value for the ensemble.

Coverage Limitations: Computational Errors

The technique relies on making it very difficult for a port SEU to result in an active high fault; if any active high exists, we assume that it was correct. But this only protects the I/O port bits; it does not prevent a bad computation by the Master or a Checker to drive the output to a bad state. If the correct output value is a zero but a single processor decides to output a low-impedance one, the one processor will override any number of high-impedance zeros. This is a common-mode failure and must be addressed by the software-implemented fault-tolerance functions which use the Master Channel. Single-bit errors which occur in the short window after software voting and before port output are similarly uncovered.

One additional side-effect of this I/O voting approach is that an output will be high from the time that the first device outputs a high until the last device releases it. The pulse widening is equal to the maximum clock skew between any two processors.

Table 2. I/O Pin States for SEU-Tolerance with Active One.

Output Value	Tri-State Enable Bit (PIC Microcontroller Interpretation)	
	1 (Configure as Input)	0 (Configure as Output)
0	**Valid State: 0.** **Output = hi-Z**. External pull-down yields a "zero."	**Invalid State.** **Output = low-Z, logic-zero voltage.** If fault is incorrect I/O configuration state, low voltage output matches desired state (helps the external pull-down). *If fault is incorrect output bit, a majority is generating low-Z, logic-one voltage; a push-pull conflict exists. "Voted" node state depends on logic thresholds (high and low) of receiver.*
1	**Invalid State.** **Output = hi-Z.** Fault is either incorrect I/O configuration or incorrect output bit; output is tri-state. Correct state of the voted node will be forced by other microcontrollers and the external pull-down.	**Valid State: 1.** **Output = low-Z, logic-one voltage.** Output overrides external pull-down. (But an output-bit fault in another microcontroller results in a push-pull conflict.)

Coverage Limitations: Double-Bit Errors

The previous techniques provide an output structure which can tolerate power recycling for SEL mitigation and also for single bit-errors in the output registers. However, this latter characteristic provides only some tolerance to SEU in the registers themselves; the approach is not tolerant to double bit-errors in the port registers (half of which result in incorrect output state).

Latent faults in port registers can lead to double-bit errors, a concern since static outputs are vulnerable 100% of operating time. Periodic scrubbing can be used to detect errors but the error rate can be reduced only so far; scrubbing too frequently increases the probability of introducing a common-mode "double-bit error" (as noted previously) as a result of a single-bit error during the scrubbing computation.

Scrubbing requires additional information to determine what the correct output should be. The port state may be stored redundantly in RAM or may be determined by reading the port bits themselves. The latter approach requires that both the output bit and the configuration bit be independently readable, a feature which is also valuable for validation

experiments to determine the probability of I/O port flip-flop upset. The PIC microcontrollers [2] have this capability, as do the Intel 87C196Kx, Jx and CA [1]; many older parts do not.

If the expected error rate after scrubbing is still too high (e.g., for safety-critical functions), hardware interlocks may be used wherein a critical function must be enabled by an independent control signal. Interlocks are particularly valuable for a two processor (self-checking pair) configuration since a single-bit error will always result in ambiguity if only two devices are being voted. An I/O pin on each microcontroller used as an interlock in such a system will result in the equivalent of four microcontrollers participating in the state vote.

Scrubbing may be avoided entirely in selected cases. For some outputs, there is either no need to deal with the problem or no practical way of dealing with it. As an example, it is impractical to apply any form of software fault-tolerance to, say, a 2400 Hz PWM signal or a 9600 baud serial channel. Fortunately, these frequently-modified signals are essentially self-scrubbing; the short time before the state is driven to a new state makes double bit-errors very unlikely.

EXPERIMENTS

In order to force an outcome which will be valuable to the spacecraft avionics community and to provide a testbed for evaluating the effectiveness of the techniques, some experimental designs will be built which are representative of space system elements. These application examples are sufficiently complex to provide insights into real problems while sufficiently simple that their implementation should not distract from the investigation. The selected applications will use different numbers of controllers to implement their reliability goals and thus provide examples of different processor configurations including a triple-modular redundant configuration, a self-checking pair configuration which is designed for block redundancy, and a distributed computing system. The examples also span I/O requirements from simple bi-level and analog voting to pulse train generation, event timing, and serial communications. Example problems to be considered include an inertial measurement unit, a propulsion/pyro switching unit, and a distributed sun sensor.

The prototypical example applications will use the Microchip PIC16C73A [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. Because this microcontroller family does not provide access to its internal address and data busses (unlike other families), any temptation to use too much I/O and then reconstruct it externally is removed. Finally, one of the authors knows of avionics practitioners interested in this chip so results of this research should be immediately valuable to them.

CONCLUSIONS

We have described the initial steps toward a generic approach to implementing cost-effective fault-tolerance augmentations of commercial microcontrollers in demanding applications such as spacecraft control systems. The described experimental designs, to be implemented in late 1997, are expected to provide us with the insights to determine the effectiveness of these techniques.

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] V. P. Nelson, B. D. Carroll. "Tutorial: Fault-Tolerant Computing." IEEE Computer Society Press, 1987.

[4] 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.

[5] 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.

[6] "1996 New Releases Data Book, Volume V." Maxim Integrated Products, 1996.