IEEE-copyrighted material Proc. 1997 Pacific Rim Int. Symp. on Fault-Tolerant Systems (PRFTS), Dec 15-16, 1997, Taipei.

A Fault-Tolerant Embedded Microcontroller Testbed

David A. Rennels, Douglas W. Caldwell, Riki Hwang, and Malena Mesarina
University of California at Los Angeles

Abstract

This paper presents a design approach for implementing a fault-tolerant embedded computing node based on the use of low-cost commodity microcontrollers. A combination of software and relatively simple external logic is used to implement fault-tolerance in a redundant set of microcontrollers. A node can be protected with different amounts of redundancy (duplex, triplex, hybrid) depending upon the needs of its host subsystem, and is intended to be interconnected with other nodes into a modular distributed network. The structure of the node, and fault detection and recovery algorithms are described, along with a description of an experimental testbed that is being implemented.[1]

1. Introduction

Microcontrollers are highly integrated computer systems on a single chip, containing processor, RAM, ROM, and I/O including A/D converters and bus controllers [1]. These very inexpensive commodity devices can form the basis for modular distributed networks, but they lack fault-tolerance features required for highly reliable systems. Thus fault-tolerance must be added using redundant microcontrollers and additional supporting circuitry. A goal of this project is to develop a network of embedded fault-tolerant microcontrollers while maintaining, to as great an extent possible, the low-cost leverage of these devices and their existing I/O interfaces which designers are accustomed to using.

The basic approach, as one would suspect, is to have redundant microcontrollers run identical applications programs with comparison or voting for error detection and correction. Interesting design problems occur because the microcontroller is a highly integrated device that provides many existing functions that cannot be changed, yet they must be covered by fault tolerance. Some of these design problems include:

- Interactive consistency -- The microcontrollers may sample slightly different values of some input signals (e.g. analog inputs). Thus special message exchanges are required for agreement [2].
- Bounding Error Latency -- With very little internal checking in microcontrollers, periodic internal testing must be interleaved with normal operation.
- Circuit Isolation -- Protection must be supplied against shorts and babbling modules to prevent single points of failure.

This design is intended to be usable in space applications, where a high rate of transient errors is expected, and latchup conditions may occur that require a microcontroller module to be immediately powered down and then restarted to prevent permanent failure. This requires power switching and the ability of the circuit isolation to prevent unpowered chips from being back-powered from the common logic signals.

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 its implementation as a testbed for validating its functionality and fault-tolerance properties.

2. The Fault-tolerant Microcontroller Node

We envision the final implementation of a fault-tolerant microcontroller node to be a redundant set of identical densely packaged microcontroller modules that are stacked in such a way as to provide the same footprint and interconnections to the host subsystem in which it is embedded -- regardless of the amount of redundancy used, as shown in Figure 1. The node is connected to other similar nodes in different subsystems via redundant serial buses.

The node may contain two microcontroller modules for transient error recovery and fail-safe shut-down upon permanent faults, three modules to provide uninterrupted operation during a fault or error, or four modules for applications requiring extended life. Thus we chose not to use

1. This research is sponsored by the Office of Naval Research under grant #N00014-96-1-0837

the massively redundant intercommunication schemes between modules that guarantee Byzantine resilience, but instead rely upon a multi-level recovery approach that provides restarts with diagnosis in the unlikely event that the normal recovery algorithms become confounded [3].

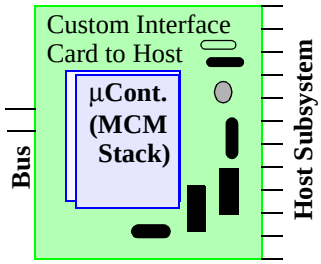

Figure 1: An Embedded Node

2.1 Operation of a Microcontroller Node

A fault-tolerant node consists of two or more redundant microcontroller modules. Each microcontroller module (hereafter described as "modules") can be assigned one of three operating states: (1) Master, (2) Slave, and (3) Off-Line. The Master and Slave modules, designated "active" modules, execute identical applications programs scheduled by a time-driven operating system on the basis of a common real-time interrupt (RTI) [4]. The applications programs execute at approximately the same time -- bounded by the frequency skew of their clocks within an RTI interval and slight differences in operating system functions in Master and Slaves. Off-line modules do not participate in the processing, but wait for commands via the inter-node serial buses or from the Master module (or from an RS232 debug port in the testbed) to modify memory, perform tests, and reconfigure the module to go back on line.

Although applications programs are the same, the operating systems in Master or Slave modules carry out different functions for comparison, voting, and error recovery. The general strategy taken is:

- **Inputs** -- All active modules independently sample all inputs. In order to achieve the same value for the input in all the modules the Master sends the value it received to the Slaves (via a special internal bus called the Master Channel). The Slaves determine if the value is within tolerance limits of what they sampled and respond with an agree or disagree symbol over a special Status Channel that all modules can see. If at least one Slave agrees, all the modules use the Master's value. Since I/O events have bounded latencies, modules that do not respond can be detected by time-outs. Both disagreements and time-outs trigger recovery actions to be discussed later.

- **Outputs** -- When an output is to be produced, the Master generates an output value and sends it to the Slaves. The Slaves' operating systems do an exact comparison of this value with the outputs they would have generated and respond with agree or disagree status symbols. At least one Slave must agree with the Master to enable an output. There are several variations of output algorithms that can be accommodated. In one case the Master sets an output, the Slaves read the actual output lines, compare and send back status. The Master is allowed to follow up with an enable signal to the I/O device only if there has been an agreement. (Between outputs, the output lines are placed in an invalid, uncoded state to assure that in the unlikely event of a spurious enable signal, an output will be rejected.) Alternatively, the Master sends an output value for comparison via the Master Channel, and if there is agreement the Master and Slaves all send outputs that are voted at the I/O device. As in the case of inputs described above, time-outs and disagreements trigger recovery actions.

- **Protection of I/O** -- Typically microcontrollers provide no hardware error detection on ports. Therefore, along with the normal uncoded input and output ports, additional input and output ports may be reserved for parity check bits, with error detection being implemented in software. Input and output ports are protected against catastrophic shorting and babbling faults by special circuit isolation. For certain critical functions two ports may be assigned for redundant communications paths.

2.2 Structure of the Fault-Tolerant Node

Figure 2 shows a block diagram of a fault-tolerant node including four microcontroller modules -- a Master, two Slaves and an Off-Line module. (Note that the operating states of the modules can be dynamically changed, so different modules may be Master, Slave, and Off-Line at different times due to fault-induced reconfigurations.)

2.2.1 Normal I/O

The interconnects at the top of the diagram, designated normal I/O are the digital and analog inputs and digital outputs to external sensors and actuators. The inputs are bussed to all modules. The output connections are bidirectional (programmable from within the microcontroller modules). Only the Master module is configured to drive the output lines, and Slaves are configured to receive the output lines as inputs in order to do comparisons. If the

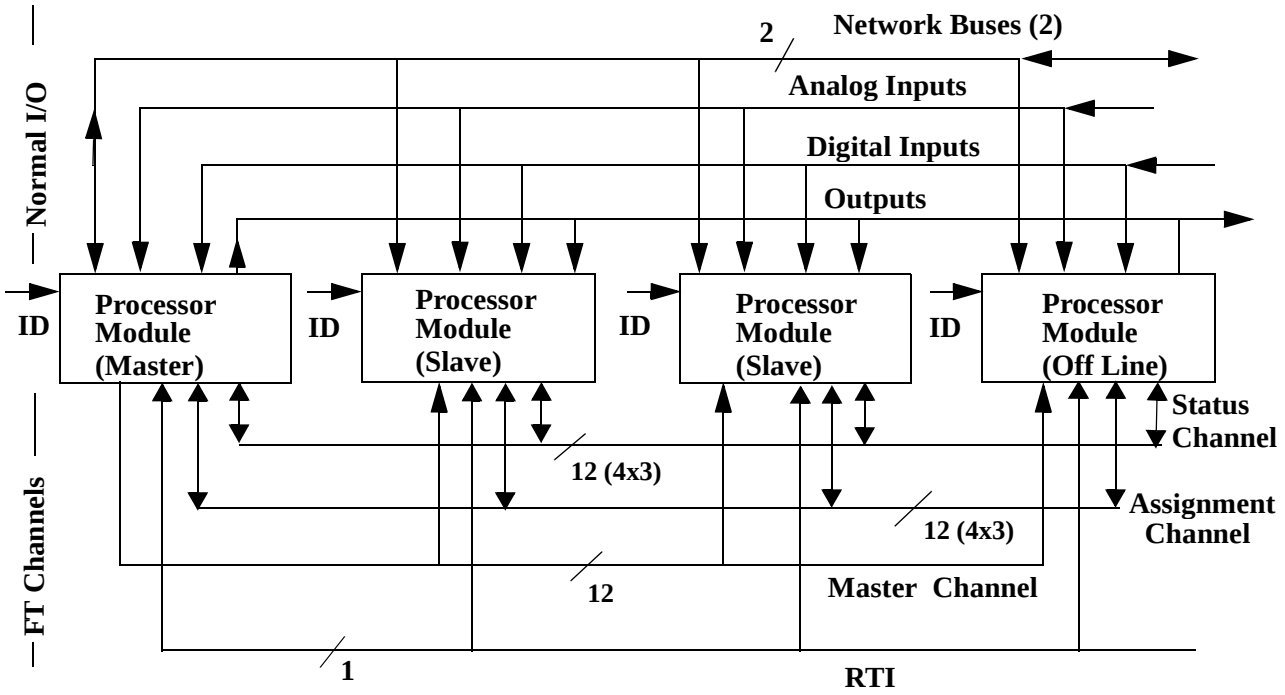

Figure 2: Redundant Microcontroller Modules and Their Interconnects

Master module fails, another module can be assigned as Master and can be reconfigured to drive the outputs. A separate (redundant) hardware enable circuit is used in each module that independently prevents a faulty module from generating outputs. At least two modules must "vote" for a module to be Master (via the Assignment Channel described below) before this hardware allows it to generate outputs.

Each module also has a set of ID pins that define the number of the node within the system and the number of the module within the node.

2.2.2 Fault-Tolerance Channels

The interconnects at the bottom of the diagram, labeled FT-Channels are used for implementing fault-tolerance features. They consist of the Master Channel, Status Channel, and Assignment Channel.

The Master Channel

The Master Channel is a bus used by the Master: i) to send data to **all** the Slaves for comparison and voting, ii) to command program rollbacks, and iii) to send commands to **specific** modules to command them on-line or off-line or to load and access data in their memories. A summary of Master Channel functions is shown in Table 1.

addressed to all modules	addressed to a specific module (or I/O device)	
	Send High Address	Jump to address
send values for comparison	Send Low Address	read status register
command program rollback	Write next High Data	Power down
	Write next Low Data	Power up / restart
	Read next High Data	Go off-line
	Read next Low Data	Go on-line

Table 1: Master Channel Functions

It is connected to programmable bidirectional ports so that the Master can be configured as the bus driver and the other modules can be configured to receive. If the Master is replaced, its outputs are disabled, and a new module assigned as Master is reconfigured to drive the Master Channel. The redundant hardware enable circuit in each module (mentioned previously) checks to see that at least

two modules have voted for it as Master via. the assignment channel before enabling its output capability.

The Master Channel represents a potential single point of failure since it is needed for the entire node to function. It is protected with circuit isolation to prevent catastrophic failures due to shorts. The redundant hardware enable circuit is used to prevent another form of catastrophic Master Channel failure due to “babbling” modules. Finally, an errant Master in control of the Master Channel could issue faulty commands and bring down the whole redundant set of modules. Protection is provided against this using the Status channel described below. At least one other module must signal agreement with the Master, via the status channel before hardware in any module will allow a command to be accepted.

The Status Channel

Since normal I/O is constrained so that only the Master can generate outputs and messages to the other modules, it is necessary to provide a mechanism by which (1) the modules can synchronize their actions, and (2) the Slaves can communicate whether they agree or disagree with the actions of the Master. Each module can broadcast a symbol, that serves as a status message to the other modules, over a set of three wires. The collection of wires conveying the status messages from all the modules are designated the Status Channel as shown in Figure 3..

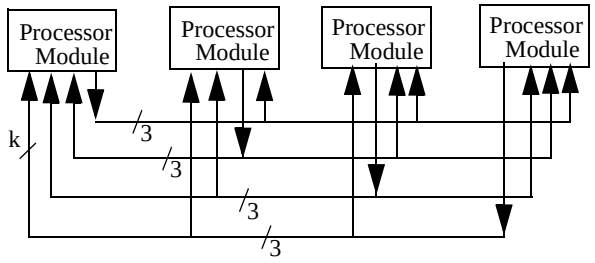

Figure 3: The Status and Assignment Channels

The status messages (symbols) are shown in Table 2:

Master Status Messages	null	check data	retry	time-out	internal error
Slave Status Messages	null	agree	disagree	time-out	internal error

Table 2: Status Messages

The null symbol is used to produce a handshake protocol between the Master and Slaves and to assure that a module has not died and left its status set to “agree”. An “internal error” symbol is used by a module that detects an internal error to signal that condition to the other modules. The other symbols are used during the comparison procedure and will be discussed in Section 3 below.

The Assignment Channel.

The Assignment Channel consists of four three-wire buses -- one from each module to the other three microcontroller modules (connected similarly to the Status Channel in Figure 3). Each module always sends one of the following messages as 3-bit symbols to the others over its dedicated lines as shown in Table 3:

Message
Vote for Module 0 as Master
Vote for Module 1 as Master
Vote for Module 2 as Master
Vote for Module 3 as Master
Module Off Line
Module attempting system start-up
Module Power down / reset

Table 3: Assignment Channel Messages

Each module votes the messages on the Assignment Channel to determine whether it is the Master module (two or more votes - from itself and at least one Slave module.) This is done redundantly. The microcontroller software performs the vote and only enables its output drivers and operates as Master if it “wins”. In addition each module has redundant hardware voting logic that must independently agree on the vote before outputs are enabled. This is crucial to avoid the problem of blabbing modules.

2.2.3 Short Protection and Electrical Isolation

Inputs and outputs are connected together on the microcontroller modules, so it is necessary to provide isolation to prevent a shorted I/O line in one module from preventing the other modules from using it. Isolation is also needed to prevent an unpowered module from being back-powered from the logic signals of the other modules through the input protection diodes. This is especially important for space applications where a module may have to be totally powered down to clear a latch-up condition.

Inputs are protected by series resistors and outputs are protected with series diodes with passive pull down resistors on the common output lines. This approach dates all the way back to the JPL-STAR Computer [5]. The resistors can be implemented directly in an MCM substrate, but the diodes require additional circuit packages.

3. Synchronization and Checking

Error checking occurs in the following fashion. The Master sets an output value or broadcasts an input value over the Master Channel, and the Slaves compare their local values with the Masters value and respond with agree or disagree status messages. The time skew between the Master and Slaves is bounded, so that the Master and Slaves can set an internal time-out counter when they initiate an I/O operation and expect the comparison to complete before a time-out occurs. The Master and Slaves only reset their time-out counters at the completion of a comparison (when the Master returns its status from "check output" or "repeat" to "null" as described below).

3.1 Error-Free Output and Input Operations

For an ***output,*** the Master changes its outputs and then sends a **check data** status message via the status bus. Within some delta of time, the Slaves sample the Master's output values and respond with an agree status.

For an ***input,*** the modules all sample the same input; the Master sends the value it received over the **Master Channel** and raises its **check data** status, and the Slaves respond with agree or disagree. In the case of analog signals, the Slaves respond with agree status if the Master's value matches the value they obtained within a threshold associated with the signal. The Slaves use the value sent by the Master in their computations to assure that the computations will be the same in every module.

It is possible for Slaves to disagree if an input signal is changing, since the modules read the input at slightly different times. If at least one Slave agrees, the Slaves accept the value sent by the Master. If no Slave agrees the Master asserts retry – causing the read procedure above to be repeated. The slew and sampling rate of input signals are set so that the in lieu of faults, the second read is guaranteed to obtain identical values.

The Synchronization and Checking function is shown diagrammatically as a function of time using the following symbols:

- \- null status
- c check status (output by Master)
- a agree status (output by Slave)
- d disagree status (output by Slave)
- r retry status (output by Master)
- t time-out status (output by Master or Slave)
- x module ready for checking, time-out counter set (null status output)
- ^ time-out counter deactivated (null status output)

Consider the error-free case where the Slaves come to an I/O point slightly ahead of the Master due to the slight differences in their clocks.

```
Slave  ------------x--------aaaa^----
Slave  -----------------x---aaaa^----
Master --------------------xccc^-----
```

In the example above, the Slaves arrive at a point of expecting an output from the Master (at times marked by x) they set a time-out counter and wait (spin) for the Master to send its output and raise its check output status (c). Note that this could be when their application software generates an output or after they do an input and expect a value for confirmation from the Master. When the Master provides the value to compare, it raises its check data status, the Slaves do the compare and send agree status messages (a). The Master sees the agree statuses from the Slaves and returns its compare status to null, causing the Slaves to return their status to Null. Both Master and Slaves then clear their time-out counters and resume computations.

When the Master resets its compare (or retry) status to null, this signals that the comparison is complete. Then the Master and Slaves sample all symbols on the Status Channel. This collective status from all the modules is used for error recovery. In the error-free case, there are "agree" signals from all the Slaves, so no further action is taken. If there is a disagree or null signal from a Slave, an error is indicated and a recovery action is invoked.

In the case above the Master was last to arrive at the data checking point in its program. Two similar cases are not shown where the Master is first or is in the middle.

The sampling of status messages at the end of a compare operation must be done very carefully to minimize the probability of Byzantine errors. If a Slave is changing its status just as the Master returns to null, different modules could "see" different values of status and perform inconsistent recovery actions. For example, a late Slave might signal "agree" just as the Master drops its compare level. A module that sampled the status quickly would see a "null" from that Slave and a slower-sampling module would see the belated "agree".

Because of this problem each Slave is constrained to changing its status within a time period δ of sampling a compare output status from the Master, and it must hold (not change) its status a time period ψ after it sees the Master's status return to null. The modules all sample the status channel in the period between δ and ψ. In order to simplify recovery algorithms, a Master is designed to hang up if it does not receive one agree status on an output compare or during a retry of an input operation. It raises the check output status but does not return to null. This simplifies recovery algorithms as described below.

3.2 Synchronization and Checking with Errors

The fault-tolerance mechanisms must deal with cases where one or more of the Master and Slave modules are in error. We present selected examples below to illustrate the process. Here we assume that there are at least two modules that are functional when the error occurs.

3.2.1 Well-Behaved Initial Error Modes (that can be handled by the Master)

Case 1: One Slave Disagrees on Output

The simplest error occurs when one Slave agrees and one Slave disagrees after the Master outputs data for comparison.

```
Slave  ----------------xddddd^-------
Slave  ------------------xaaa^-------
Master --------------xccccccc^--------
```

Seeing this condition, the Master commands the disagreeing Slave off-line. In space applications the disagreeing module must execute a short diagnostic after being commanded off-line in order to ascertain if a latchup has occurred. If the module cannot compute a correct diagnostic result, it is immediately unpowered, and its Vcc and ground are shorted together for a predetermined period of time before it is re-powered and configured into a standby state.

For most embedded applications an errant module will be brought back on line by "roll-forward" techniques because the recovery state is small. A state vector will be copied to the off-line module, via the Master Channel, and it will be commanded to rejoin the others at a subsequent RTI. If recovery is unsuccessful, the module is recorded as having a permanent fault and abandoned.

If only two working modules remain, then a program rollback is the only option to achieve transient fault recovery. We are currently working on techniques that will allow a redundant set of modules to revert to program rollback for transient recovery when only two are working properly.

Case 2: One Slave Misses Master's Time-out Deadline

Here one Slave doesn't respond at all, so the Master times out and drops its compare status. This case is treated the same as if the non-responding module responded with a disagree status

```
Slave  ----------------------------
Slave  ----------------xaaaaaaaa----
Master -------xccccccccccccccct----
```

Case 3: One Slave Disagrees on Input

This is the case of a disagreement of one Slave during the comparison with the Master's value after an input. It is assumed that the inputs have a high slew rate and are gated in such a way that a second read will obtain a stable value if there are no errors.

```
Slave  ---------xddddd--------dddd^---
Slave  -----------xaaa--------aaaa^---
Master ----xccccccccccrrr----cccc^----
```

After the disagreement the Master changes its status to retry (r) causing Master and Slaves to re-sample the inputs and retry the comparison. If the retry results in agreement, computations continue. The case above shows that the retry results in disagreement, so the Master must command the errant Slave off-line and possibly try to bring it back into step later.

3.2.2 Master Fault Cases

When the Master is correct, at least one Slave agrees with the Master, and error recovery actions are handled by the Master. The situation becomes more complex when the Master is in error, because the Master is the module that controls the comparison and synchronization process. In examining the error conditions that occur, we assume that the Slaves are still operating correctly, so the Slaves will come to some next I/O operation and set time-out windows waiting for the Master as shown below.

Since in these cases the Master generates a wrong output, it does not see a single agreement status from a Slave, so it hangs up -- leaving its status at compare output. Therefore the Slaves time out at their next I/O operation. When a Slave times out, it checks to see if the other Slave also times out. If they both time out, the Slaves must initiate a recovery action. This can be done by commanding one of the Slaves to assume the role of Master (by the Slaves changing their vote in the Assignment Channel).

```
Slave  -------x----ddddddddddddtttt--
Slave  ----------x-ddddddddddddddddt--
Master ----------xccccccccccccccccccc
```

4. Recovery

Recovery procedures are implemented as a hierarchy. The **first** level correspond to actions taken when an error is detected during a comparison operation. These actions are

summarized in Table 4.

Master Status	Slave Status	Slave Status	Diagnosis	Recovery Actions
c	a	a	no errors	
c	a	d	Slave in error	Master commands Slave off-line, may attempt to roll forward and restart Slave at next convenient time
c	d	a		
c	a	t	Slave in error	same as above
c	t	a		
c	t	t	Master in error	Slaves command new Master via Assignment Channel,
-	t	t		New Master retries failed input/output, attempts rollback to restart decommissioned Master as a Slave at next convenient time

Table 4: First Level Recovery Actions

This type of system relies on fairly complex recovery actions and it is not immune from Byzantine faults, so there is a small possibility that the recovery actions above will fail -- especially under transient error conditions. Therefore we implement a second, higher level, recovery mechanism that restarts the system if the first level fails. It is outlined below.

4.1 Second Level Restart/Recovery Procedures

The level-two recovery procedure is invoked: i) when the entire system is powered up, or ii) when a single module is turned on after powerdown, or iii) the modules collectively decide to do a system restart due to failure of recovery algorithms or massive multiple error conditions. The latter condition is identified when either a period of time elapses where no I/O activities occur with a corresponding agree status, or no two modules can agree upon a Master. Sensing this desperate condition, modules command themselves to the power-up restart state. (These high level recovery algorithms are also used to support the manual debugging process as the testbed system is built and integrated.) Figure 5 shows the high-level recovery algorithm.

In the power-up/restart state, the module sets its Assignment Channel lines to signal powerdown/restart (000). Note that if power is turned off, an Assignment Channel output of 000 will also result due to the diode isolation with passive pull-downs used on its outputs. Next, the module runs a diagnostic that checks basic instruction execution and also checks to see if program memory is intact using checksums. If the diagnostic passes, the module checks a Manual control input (used in the testbed to allow external control of the module), and if it is set, it goes into Manual Mode waiting for commands from either the Asynchronous serial port, the Master Channel, or the inter-node bus, and it sets its Assignment Channel bits to indicate to the other modules that it is in an off-line state.

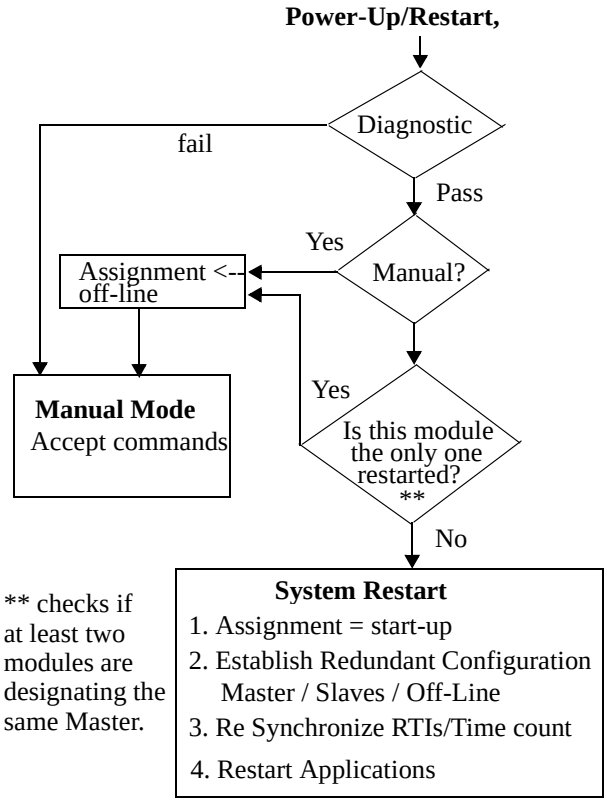

Figure 5: The Second-Level Recovery Algorithm

If manual operation is not specified, the module must decide what to do in order to rejoin the system. Here the module must determine whether the rest of the system is working, in which case it simply goes into off-line mode and waits from commands from the Master. If, however, the whole node is being restarted, the module must participate in a process of restarting the system -- agreeing on a new Master and Slave configuration and participating in a coordinated system restart. Therefore the module checks again to see if, over a period of time, no two modules select the same node as Master. As a result of this test the module will either enter the Manual Mode or attempt to join others in a System Restart as described below:

Manual Mode -- In the manual mode, the microcontroller module simply polls the internode bus, serial port and Master Channel interfaces and it accepts essentially the same commands from each source, i.e., set memory address high, set memory address low, read next byte, store next byte, jump.

System Restart -- The restart mode consists of three steps: 1) Configure, 2) Sync-up, and iii) Start-up.

1. Configure -- The module waits until either:

- all four modules have set their Assignment Channel Symbol (ACS) to start-up, or
- at least two modules have ACS = start-up and a time-out occurs. The time-out is set long enough that all ACS changes can be expected.

Then the modules assign a Master as the highest numbered module that has successfully completed the diagnostic (is not in manual mode) and has set its ACS to start-up. Modules with lower numbered IDs and are in start-up status take on the role as Slaves. This is done by setting the assignment channel to specify the new Master -- thus enabling its outputs.

2) Sync/Start -- The Master inquires of the Slaves the last value they had as a time count by reading out a known memory location. It then determines a "best guess" of the current time (zero if all are widely different, its own last time count plus an estimated downtime, if it is approximately the same as a Slave, or a Slave value plus an estimated downtime if two Slaves are nearly identical.).

The Master then enables its RTI, and at the next RTI it broadcasts the time count to the other modules and commands them to: i) readout a start-up command sequence from ROM, ii) to enable their RTI, and iii) to assume normal operation.

5. Testbed Implementation

The fault-tolerance algorithms of the redundant microcontroller nodes are complex, so it will be necessary to experiment with representative application programs and recovery algorithms to fully understand their behavior. Due to the very inexpensive nature of the hardware and support software for microcontrollers, we have chosen to build and experiment rather than to perform simulations. By inducing faults into the running system and observing its response, we will obtain a good understanding of the levels of coverage that can be attained [6]. The structure of the testbed is similar to Figure 2. A PC will be used to generate inputs and sample outputs and the data on the fault-tolerance channels. Additional signals are connected to each of the redundant microcontroller modules to allow the PC to load memory, control program execution, and to allow each module to signal the PC as to its state of program execution.

6. Conclusions

We have described the initial steps toward a generic approach to implementing cost-effective fault-tolerance augmentations of commercially available microcontrollers in demanding applications such as spacecraft control systems. The described testbed environment is 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. Frison, S. G., and J. H. Wensley, "Interactive Consistency and Its Impact on TMR Systems," Dig. Int. Symp. Fault Tolerant Computing, FTCS-12, June 1982, pp. 228-233.

3. Harper, R.. E., J. H. Lala, and J. J. Deyst, "Fault-Tolerant Parallel Processor Overview," Dig. Int. Symp. Fault Tolerant Computing, FTCS-18, June 1988, pp. 252-257.

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

5. Avizienis, A. A., et. al., "The STAR (Self-Testing and Repairing) Computer: An Investigation of the Theory and Practice of Fault-Tolerant Computer Design," Dig. Int. Symp. Fault Tolerant Computing, FTCS-1, June 1970, pp. 92-96.

6. Arlat, J., Y. Crouzet and J.C. Laprie, "Fault Injection for Dependability Validation of Fault-Tolerant Computing Systems," Dig. Int. Symp. Fault Tolerant Computing, FTCS-19, June 1989, pp. 348-355.