Important developments are taking place in the areas of fault tolerance and high availability. Fault tolerant and high availability systems require thoughtful design of both the hardware and the software, with the goal of providing error detection, error correction, fault tolerance, and ease of repair. CompactPCI’s hot swap specification has defined the hardware requirements for fault tolerant, high availability systems, and now significant progress is being made in defining the software requirements. In this article Curt lists the operating system features that are important when building fault tolerant, high availability systems, and discusses why each is important.

Fault tolerance is a term used to describe how a system performs when failures occur. These failures might be caused by a wide range of conditions, including:

- a hardware malfunction
- accidental or malicious software corruption
- errors in the application software
- attempts to access illegal addresses
- attempts to execute illegal instructions

Why worry about fault tolerance?
Industry averages for the number of errors in production-level code range from 15 to 50 errors per 1000 lines.
Table 1 shows that the more complex the software becomes, the greater the number of errors per line of code.

Based on these statistics, it seems inevitable that large programs will contain many programming errors. Given that fact, it is vital to find ways to limit the adverse effect that these errors will have on the reliability and the availability of embedded systems.

<table>
<thead>
<tr>
<th>Project Size (in Lines of Code)</th>
<th>Error Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; 2000 lines</td>
<td>0-25 errors per thousand lines of code</td>
</tr>
<tr>
<td>2000-16000 lines</td>
<td>0-40 errors per thousand lines of code</td>
</tr>
<tr>
<td>16000-64000 lines</td>
<td>.5-50 errors per thousand lines of code</td>
</tr>
<tr>
<td>64000-512000 lines</td>
<td>2-70 errors per thousand lines of code</td>
</tr>
<tr>
<td>&gt; 512000 lines</td>
<td>4-100 errors per thousand lines of code</td>
</tr>
</tbody>
</table>

Source: “Program Quality and Programmer Productivity” (Jones 1977)

The software components used in embedded systems
Figure 1 shows the four software components that make up an embedded system. These include:

- the real-time operating system
- the I/O drivers
- the procured code
- the application code

The real-time operating system
The foundation of an embedded system is its real-time operating system, or RTOS. The RTOS is a layer of code that defines a generic interface between the application program and each of the system’s hardware resources, making application software less hardware specific, and easier to write.

The I/O drivers
The I/O drivers configure and control the I/O hardware, such as the Ethernet port, the serial ports, the hard drives, and the graphical display. These drivers might, or might not, be provided as part of the RTOS.

The procured code
Most embedded systems require some specialized software, such communication protocol stacks, or graphics software. This software is typically purchased in the form of libraries and/or toolkits, and represents “black box technology” which can be simply incorporated into the embedded system.

The application code
The application code is the software
that you write – the software that makes your embedded system behave in a manner that is useful in your particular application.

**Providing high reliability and high availability**
There are two approaches to providing high reliability and high availability:

- Try to create “provably correct” software, which can be shown to have no programming errors that will cause a system failure.
- Build into the system special features that are designed to detect and handle programming errors, before they lead to a system failure.

Trying to create provably correct software is unrealistic, due to development time restrictions and the level of complexity that is typical of today’s embedded systems software. A more realistic approach is to use an RTOS with comprehensive facilities for handling errors. This is called exception handling.

**RTOS architectures**
The RTOS is the foundation upon which application software is built. Therefore, the magnitude of any adverse effects of application software errors will depend upon the fault tolerance of the RTOS.

The fault tolerance of an operating system depends on its architecture. RTOS architectures fall into two basic categories:

- **thread-based architectures** (shown in Figure 2)
- **process-based architectures** (shown in Figure 3)

**The thread-based architecture**
A thread-based architecture allows any thread of execution (any task) unconstrained access to all system resources. There are two primary benefits to using a thread-based architecture:

- It minimizes the RTOS overhead
- It allows tasks to access any hardware or software resource in the system

For simple systems (systems with limited processing power and with small software programs, that are less likely to contain errors) a thread-based architecture might make sense. However, in complex systems (where the probability of software errors is much higher) thread-based architectures are not able to provide any real degree of fault tolerance.

**The process-based architecture**
A process-based architecture keeps track of the memory (and other system resources) that are “owned” by each process. If some process in the system attempts to access memory that is owned by some other process (without first getting permission), the RTOS intervenes trapping, isolating, containing, and cleaning up after the offending process. This provides a much higher level of protection against accidental (or malicious) system corruption.

The main drawback of a process-based architecture is that the amount of memory needed to keep track of every-
thing can be large, and performance measures (such as context switching, interrupt latency, and memory allocation times) are typically slower than in a thread-based architecture.

Fortunately, today’s inexpensive memory chips and fast processor chips have made the size and the performance differences between the thread-based and the process-based architectures insignificant. In general, for any system with...

- a processor clock speed above 25 MHz
- at least 128 Kbytes of RAM and ROM

...the performance and the size differences between a thread-based architecture and a process-based architecture are negligible.

In fact, for some applications the performance of a process-based architecture can actually exceed the performance of a thread-based architecture. For example, the performance and the memory requirements of an operating system with a network interface running TCP/IP actually favors a process-based architecture. This is because process-based architectures typically include facilities to support complex I/O systems more efficiently than a thread-based architecture.

I don’t want to give the wrong impression – thread-based architectures can be appropriate for many small embedded applications. However, more complex applications require a fault tolerant and high availability environment.

**A method for blocking the replication of viruses**

The OS-9 RTOS from Microware has a unique way of handling software viruses. The standard OS-9 executable module consists of:

- a header
- a data/executable area
- a module CRC

Thus there is a module CRC to validate the integrity of each and every executable module in the system – whether it resides on the disk or in memory. If a virus corrupts some executable module, the RTOS will detect the CRC failure, and will not allow the module to be loaded or run.

**Maintaining data integrity**

One vital factor in providing fault tolerance is maintaining the integrity of all the data in the application memory, and in the system memory. In this case the degree of fault tolerance is heavily dependent on the RTOS architecture.

A memory management unit (MMU) is integrated into many processor chips to provide data integrity with a minimum of overhead. While a thread-based architecture is not able to take full advantage of this MMU hardware, a process-based architecture is able to do so.

Note: Some thread-based RTOS vendors allow the use of MMU hardware by providing a library of routines called a system security module (SSM). These routines are called by the RTOS when-ever a task requests permission to access memory, or other address-mapped system resources. These library routines allow thread-based application programmers the option of implementing a process-based philosophy explicitly, within the source code of each task. However, the success of this approach depends entirely upon the discipline of the application programmers. If just one programmer on the team forgets to make the appropriate calls, the entire system might collapse, like a house of cards.

What if the target hardware doesn’t include an MMU? In that case the routines in the SSM can be written to emulate all of the functions of a hardware MMU in software. While this will be considerably slower, it provides some measure of protection, even when no MMU hardware is available in the system.

**Maintaining executable integrity**

Another vital factor in providing fault tolerance is maintaining the integrity of the executable images in system memory. One example where the executable images in a system might be compromised is a software virus. Software viruses are executable code that append copies of themselves to other executable images, and then modify those executable images to run the “virus code” at some point – either immediately or after some particular condition is met.

Most RTOSs cannot detect and block the proliferation of software viruses. This can be very dangerous when a virus infects a networked embedded system. (See the sidebar entitled A method for blocking the replication of viruses.)

**Defining formal boundaries between application code and the system code**

We’ve talked about fault tolerance in terms of protection of application programs from one another. As shown in Figure 4, some RTOSs also implement two execution states:

- a system state
- a user state

All code executes in one or the other of these two states. The RTOS and all of the system software executes in the system state. Application software executes in the user state. This approach prevents corruption of the “trustworthy” system software by faulty (or malicious) application software.

**Error detection and recovery facilities**

Another vital RTOS feature that is needed to provide fault tolerance is a

---

**Figure 4**
highly capable and customizable error detection and recovery facility. Some RTOSs embed this capability into the kernel, providing no way for the user to customize what happens when an exception occurs – unless the kernel source code is purchased.

Other RTOSs allow the user to provide exception handling modules that get called by the kernel whenever exceptions occur. This allows the application software to log exceptions, and to respond to them, instead of simply accepting the action that the kernel’s exception handler takes. A good example of a versatile exception handling mechanism is shown in the sidebar entitled A method for allowing user-defined exception handling.

**User identification and authorization**

Some RTOSs implement a multi-user environment. This means that each process has an associated group and/or user identifier (ID). While this doesn’t deal specifically with fault tolerance, a multi-user environment can be used to define restrictions on the access that each process has to system resources, and thus provide a more secure environment.

**High availability**

Fault tolerance provides higher system availability by allowing the system to detect (and recover from) errors that might otherwise disrupt system operation.

One very important advancement in the area of high availability is the CompactPCI hot swap specification. The goal of this specification is to allow users to maintain (or upgrade) a high availability system while the system is online, and in continuous use.

The hot swap model for CompactPCI high availability systems defines four levels of hot swap capability:

- Non-hot swap
- Basic hot swap
- Full hot swap
- High availability hot swap

**Non-hot swap**

Non-hot swap capability provides no support for the replacement of boards while the system is online.

**Basic hot swap**

Basic hot swap capability requires the operator to manually initiate the software disconnection before powering down the board and removing it from the backplane. The operator must also manually initiate the software reconnection, after inserting a replacement board and powering it up.

Note: Most process-based architectures can be made to support basic hot swap capability. However, the specific RTOS design (and the feature set of the RTOS) determines whether full hot swap, or high availability hot swap, is achievable.

**Full hot swap**

The unique characteristic of full hot swap capability is that the operating system automatically handles the software connection process by loading the corresponding device driver when a board is inserted. Upon board extraction, the operating system:

- notifies all processes attempting to

---

**A method for allowing user-defined exception handling**

Figure A, Figure B, and Figure C show how the OS-9 RTOS provides for exception handling.

Figure A shows the exception handling architecture. The OS-9 exception handler is executed whenever an exception condition occurs. It compares the exception type to a list of all the exceptions handled by custom exception handlers in the system, and then decides between two alternatives:

- If the current exception matches one of the exception types in the list, the custom exception handler is given full control. (See Figure B) This exception handler might log the failure, initiate a recovery, or relaunch the process that caused the exception.

- If the current exception matches none of the exception types in the list, it will be dealt with in the default way: the process will be terminated and all of its memory and I/O resources will be recovered by the RTOS. (See Figure C)
the hot swapped board is missing from the backplane. In order to do this, the system must have redundant hardware resources. When the RTOS is notified that a particular board is scheduled for extraction, it switches over all processes in the system that are currently using the hardware resources of that board to redundant hardware resources.

**Summary**

Table 2 summarizes the RTOS capabilities needed to provide for fault tolerance in an embedded system, along with a brief description and some questions to ask when evaluating RTOSs for your application. Table 3 summarizes the RTOS capabilities that are needed to support full hot swap and high availability hot swap.Ω

If you have questions about this article, or if you would like to know more about Microware’s products, you can contact Curt at:

**Microwave Systems Corporation**

1500 NW 118th Street
Des Moines, IA 50325-7077
Tel: 515-223-8000
Fax: 515-224-1352
Email: curts@microware.com.

---

### Table 2 - RTOS features needed to support fault tolerance

<table>
<thead>
<tr>
<th>Category</th>
<th>Description/Questions to ask</th>
</tr>
</thead>
<tbody>
<tr>
<td>RTOS architecture</td>
<td>Is the RTOS thread-based or process-based?</td>
</tr>
<tr>
<td>Data integrity</td>
<td>Does the RTOS provide memory management unit (MMU) support? Is an MMU required or optional? Is there a system security module available for use if the target hardware has no MMU?</td>
</tr>
<tr>
<td>Executable integrity</td>
<td>Does the RTOS provide facilities to ensure the integrity of all executables? Are there any safeguards to prevent the proliferation of software viruses? Does the RTOS support both a system state and a user state of execution?</td>
</tr>
<tr>
<td>System/User state boundary</td>
<td>Does the RTOS have built-in exception handling facilities? What is the default RTOS exception handling process? Can custom exception handlers be integrated with the RTOS?</td>
</tr>
<tr>
<td>Error detection and correction</td>
<td>Does the RTOS implement a multi-user environment? How many group/user IDs are supported?</td>
</tr>
<tr>
<td>Multi-user environment</td>
<td></td>
</tr>
</tbody>
</table>

### Table 3 - RTOS features needed to support high availability

<table>
<thead>
<tr>
<th>Hot swap capability</th>
<th>Description/Questions to ask</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-hot swap</td>
<td>--</td>
</tr>
<tr>
<td>Basic hot swap</td>
<td>Does the RTOS have a modular run-time, or is it a single unified body of code? Can the run-time image be dynamically added to without downloading the entire image again? Is the RTOS position independent? (If “no” you can’t load/unload modules.)</td>
</tr>
<tr>
<td>Full hot swap</td>
<td>Can the RTOS dynamically add modules while the system is online and in use? Can the RTOS dynamically remove modules while the system is online and in use?</td>
</tr>
<tr>
<td>High availability</td>
<td>Can the RTOS dynamically replace software modules/components while the system is online and in use? Can the RTOS dynamically download replacement modules, while the system is using the module that is being replaced? Can two instances of the same module reside in the RTOS environment simultaneously?</td>
</tr>
</tbody>
</table>

---

Curt Schwaderer is the director of network technologies at Microware Systems Corporation, and is responsible for world-wide network software solutions. Curt has an MS in computer engineering from Iowa State University, with emphasis on networked embedded systems and network security. Prior to joining Microware, he designed and implemented secure and non-secure digital switching systems for E-Systems/Raytheon. Curt also has a patent pending in the communications industry.

**References**

Fault tolerant LAN interfaces

The best context for illustrating fault tolerant or high availability issues is in the area of telco applications. This area is of interest because the requirements for fault tolerance are the most demanding. The fundamental requirement simply stated is continuous availability of the service being provided. LANs are used to interconnect the computing platforms in a distributed computing model.

For services to be continuously available, the LAN must be fault tolerant. Fault tolerance is delivered by providing:

- Fault detection
- Failover to a backup interface

If fault detection and failover happen fast enough, connections are preserved during a LAN fault and service is not disrupted. If this happens slowly, connections are lost and service is temporarily disrupted for part of the end-user community.

The goal in providing fault tolerant LAN solutions is to provide sophisticated fault detection that includes any type of LAN failure. This can be implemented at the driver level by addressing the three areas:

- Link failure detection
- Watchdog timeout
- Application participation

The link failure detection implementation is relatively straightforward. If a cable is disconnected or cut, if a connected NIC, hub, or switch is powered down, an interrupt is generated, the device driver recognizes a link failure event, and executes the failover sequence.

The watchdog timeout feature deals with more complex failure scenarios where the link is still active but connectivity is no longer present. This can happen when there is a more complex network topology and an intermediate component fails. The watchdog timeout also covers the case when there is a partial failure such as when a node transmits packets with errors, or a receiver has problems that cause errors on reception. The watchdog timeout is implemented using a heartbeat. The heartbeat is produced by traffic generated at assigned intervals – every two seconds for example. If the watchdog doesn’t see a (good) packet every two seconds, a timeout occurs and the failover sequence is executed.

Application participation can also be valuable. The application is aware of other nodes in the environment and can look for heartbeat signals from specific nodes. When the application detects a missing heartbeat from a critical node, the driver can be instructed (via APIs developed for this purpose) to execute a failover sequence. Cooperation between the hardware, driver, and application yields the optimum failover scenario: comprehensive fault detection and fast deterministic failover.

This combination can deliver bullet-proof LAN connectivity with failover mechanisms so fast that end users of real-time voice systems (such as Voice over IP) will be unaware that a failure occurred.

In addition to the features and mechanisms described above, implementing meaningful CompacctPCI hot swap solutions requires yet another feature: fault isolation.

After a LAN failure is detected and a failover sequence executed, the problem must be corrected. Managers must be notified that a problem occurred. Technicians must be supplied with information that helps them correct the problem.

To do this, fault isolation capability needs to be added to the driver. APIs must then export the results of fault isolation procedures to monitoring and management applications. The implementation of fault isolation after a failover sequence on a LAN interface is straightforward. When a failure occurs, conduct an internal loopback test to verify the onboard circuitry, verify that rear transition cards are still inserted. If the internal loopback test succeeds and the rear transition card is present, the fault is an external fault. Otherwise the fault is an internal fault.

Internal faults tell technicians to (hot) swap out the failed adapter. External faults tell technicians that troubleshooting must be done away from the adapter. Additional information about the failure can also be useful such as link failure, watchdog timeout, CRC errors, and collisions.

To supply fault tolerant LAN interfaces to the embedded and telephony markets using RTOS platforms, ZNYX Corporation built the fault tolerant features into a software module called RAINlink (RAIN stands for Redundant Array of Independent Netports – analogous to RAID). RAINlink services are embedded into a layered architecture used to build the ZNYX Portable Driver Kit (PDK).

The layering of the PDK is important to the fault tolerant implementation. RAINlink services are invisible to the RTOS protocol stack. In the diagram below the PAL layer presents a virtual port to the RTOS TCP/IP stack. RAINlink delivers fault tolerant features transparently below the PAL – in the RAL layer. The HAL layer sits below the RAL and contains all knowledge of the Ethernet MAC. The SAL contains all knowledge of system-dependent functions such as memory allocation routines.

Using the PDK architecture, ZNYX has built fault tolerant LAN interfaces for VxWorks, pSOS+, and LynxOS real-time operating systems. Most of the RAINlink implementation is common across the RTOSs. Differences are localized to the PAL and SAL functions. The PDK also includes software switches to migrate RAINlink drivers across Pentium, Sparc, and PowerPC processors.

The PDK architecture has another possible application. The PDK approach could be generically integrated into high availability RTOSs as a new driver model. Third party developers would develop a HAL for their specific hardware. RAINlink features would then be generically available across all third party interfaces for that particular RTOS.

The most feature-rich solution for Carrier Class CompactPCI platforms is a combination of RAINlink on ZNYX’s ZX470 Carrier Class Ethernet adapter. The ZX470 features full PICMG hot swap support, interrupt driven link failure detection, internal and external fault LEDs, and rear panel I/O. The rear panel cabling approach addresses fundamental telco preferences. Technicians want to be able to (hot) swap adapters without disturbing cabling (disturbing cabling means they might actually cause a problem while trying to solve one).

Delivering fault tolerant LAN solutions for RTOS-based platforms requires building features into several layers of the ISO model. To deliver a complete, bullet-proof LAN interface requires specialized hardware, specialized drivers, and cooperation between drivers and management applications.

For more information, contact ZNYX Corporation at 510-249-0800 or www.znyx.com.