QNX SOFTWARE SYSTEMS

Choosing an RTOS for Remote-care Medical Devices

Somu Vadali, Product Manager (Middleware)
Justin Moon, Product Manager (Medical)
QNX Software Systems
svadali@qnx.com, jmoon@qnx.com

Abstract

Three trends are driving a dramatic increase in the number and diversity of remote-care medical devices entering the market: aging populations in industrialized countries, pressures on private and public healthcare providers and insurers to reduce health care costs, and a new focus on primary and secondary care. For example, in the U.S. alone, sales of remote-care devices are expected to double to $1.3 billion in 2011.

This paper is about choosing an embedded OS for remote-care medical devices — or indeed any system where reliability, recovery and safety are critical differentiators. It provides a high-level "shopping list" of requirements that device manufacturers can use to evaluate OSs they are considering for their remote-care medical devices.

An Expanding Market

It is no secret that the populations of industrialized countries are aging. By 2020, the over-60s in France, for instance, are expected to represent more than 27% of the population — an increase of some three million seniors from 2010. Germany expects a similar increase, and in the U.S. the number of over-60s will increase from 58 million to 77 million, or 22% of the population. In Japan, at almost 43 million, they will represent a staggering 34% of the population[1].

While these aging populations continue to expect and demand the best care that medical science and technology can provide, the cost of this care continues to rise. There is no guarantee that governments (taxpayers) and private insurers (premium payers) will be willing — or even able —

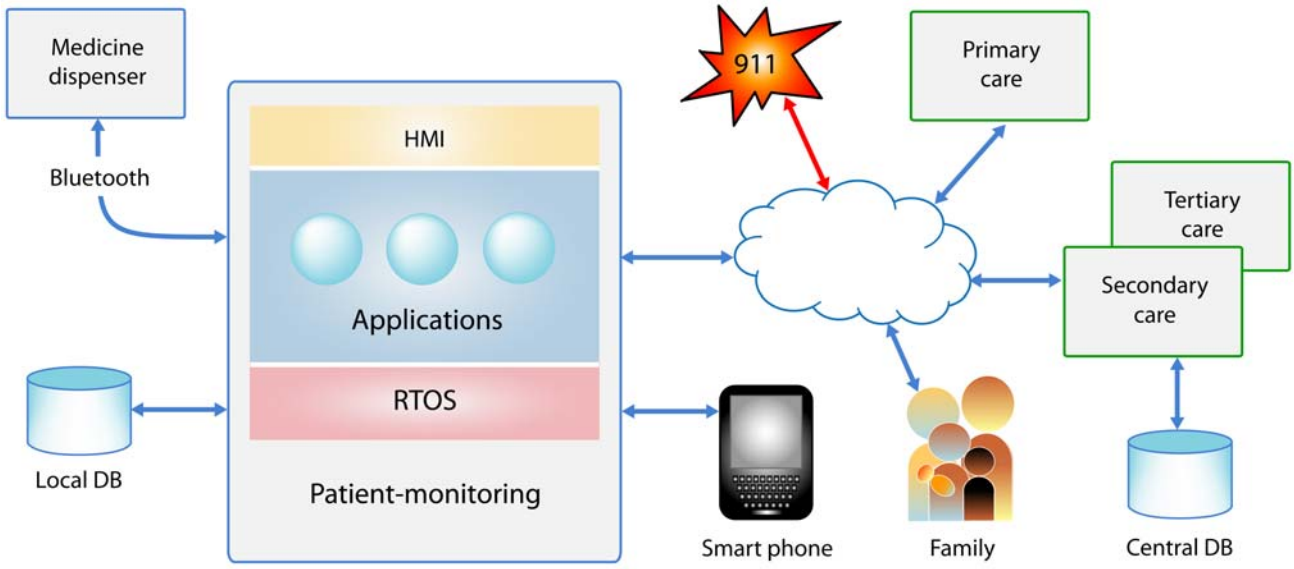

Figure 1: A simple remote-care patient-monitoring network.

[1] Population data (rounded to the nearest million) is from the United Nations Population Division, *World Population Prospects: The 2008 Revision Population Database*, 2008.

to continue paying for the same sort of tertiary care that has come to characterize healthcare in industrialized countries. As their demographics change and financial pressures increase, jurisdictions everywhere are rethinking their approach to healthcare.

Moving patient care out of the hospital

Organizations and jurisdictions from the World Health Organization on down have identified healthcare systems' disproportionate focus on hospitals and specialists, not only as a key contributor to the rising cost of healthcare, but also as an impediment to improvements to healthcare[2]. And, in fact, healthcare delivery in industrialized countries has already begun moving beyond the familiar hospital-centric model.

For example, the Family Health Team program was introduced by the Canadian province of Ontario in 2004 precisely to maintain the quality of healthcare while addressing its current and future burden on provincial coffers. The same province launched the Ontario Telemedicine Network (OTN) for similar reasons, but also to extend and improve healthcare delivery to remote areas[3].

These and other similar initiatives focusing on primary and secondary care rather than hospital visits are made possible thanks, of course, to innovative thinking, but also to enabling technologies such as digital imaging, video communications, wireless technologies, and the advent of dependable, portable and inexpensive medical devices that can be used outside hospital settings, often by the patients themselves[4].

Already, devices such blood glucose meters and lancing devices for diabetics, and medical alert and medication dispensing systems for the elderly are on the market and gaining acceptance. Their benefits are clear. Small, relatively inexpensive devices that allow diabetics to monitor their glucose themselves instead of visiting a clinic, or that permit the elderly to stay in their homes instead of moving into "a home" both improve their quality of life and reduce the cost of caring for them.

Demand for these remote-care devices is driving market growth. Writing in *Bloomberg Businessweek* in 2010, Olga Kharif noted that U.S. sales for wireless consumer healthcare devices reached US $600 million in 2010, and may reach US $1.3 billion in 2011[5]. In the same article, Kharif lists just some of the heavyweights moving into the market: Qualcomm, AT&T, Microsoft, General Electric … While US $1.3 billion represents only a sliver of a worldwide medical device market worth over US $200 billion annually, its projected growth is nothing to be sneezed at.

The OS — A Key Differentiator

To gain and sustain market share, remote-care device manufacturers will certainly have to demonstrate to users (patients and healthcare providers), payers (governments, insurance companies, users), and regulatory agencies (FDA in the U.S., MDD in Europe, etc.), not just that their devices are viable, cost-effective alternatives to current practices, but that they are better and less expensive than their competitors' devices, and that they are safe.

A critical element and key differentiator in any electronic medical device is its operating system. Everything above the silicon depends on the OS: if the OS fails, everything fails. And medical devices are not like desktops. Even for FDA Class I and II devices, random failures and reboots are not acceptable. Users have been conditioned to expect these from their desktops, but they are unlikely to tolerate them when their health or the health of their patients or loved ones is concerned.

Clearly, medical device manufacturers are aware of how much depends on the OS. In the world of embedded systems, the hardware is the first decision point. Manufacturers overwhelmingly choose their boards first, then choose the OS, tools, etc. The exceptions are mobile phone manufacturers, and medical device manufacturers.

---

[2] World Health Organization. *The world health report 2008: primary healthcare now more than ever*. Geneva: 2008. p. 11-12.

[3] Ontario Telemedicine Network. www.otn.ca.

[4] See for example, Low Cheng Ooi, Chairman of the Medical Board, Changi General Hospital, Singapore quoted in "How will technology change the future of healthcare?" (*FutureGov* 1 Sept. 2010): "The convergence of broadband penetration into homes and the emergence of more sophisticated portable medical devices is creating an opportunity for harnessing innovative technology to push the point of healthcare delivery to the home."

[5] Olga Kharif. "Qualcomm, AT&T Move in on 'M-Health'". *Bloomberg Businessweek*. 23 Aug. 2010.

VDC Research reports that, in 2010, of manufacturers surveyed in, for example, the automotive/rail/transportation industry, just 9.3% chose the OS first; in telecommunications and networking 20.8% projects started with the OS. In contrast, 53.3% mobile phone projects started with the OS, as did 36.4% of medical device projects[6].

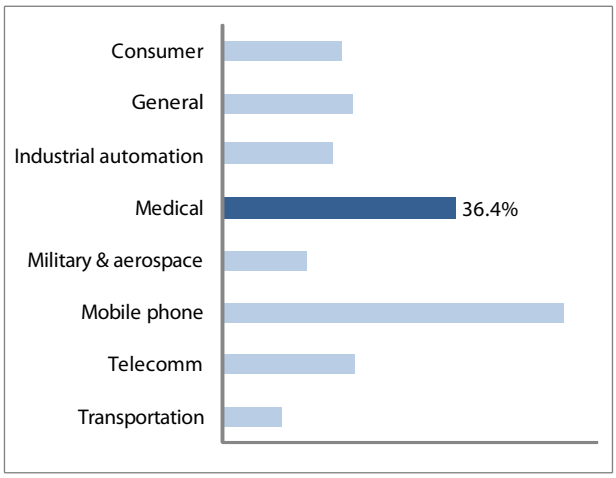

Figure 2: Percent of projects selecting the OS first, by industry (adapted from Balacco et al*).*

Requirements Overview

At the highest level, the requirements for selecting an OS for remote-care medical devices can be separated into business requirements, compliance requirements, and technical requirements.

Business requirements

The business requirements driving OS selection for remote-care medical devices are little different from the requirements for other types of devices, and are familiar to anyone in the business: cost, quality, time-to-market, portability, support, vendor history, ecosystem and vendor track record and long-term viability.

Compliance and pre-market approval

In the medical device industry, compliance and pre-market approval are critical to success. Before a device can be released on the market, the manufacturer must demonstrate that the device complies with the relevant legislation in jurisdictions where it will be sold, which often requires pre-market notification: in the U.S., FDA 510(k), for example, the Medical Devices Directive (MDD), and myriad national standards in Europe and elsewhere.

Additional legislation, such as the U.S. Health Insurance Portability and Accountability Act (HIPAA) and the Health Information Technology for Economic and Clinical Health Act (HITECH Act), governs medical data security and privacy.

These compliance requirements add considerably to both the cost and the time needed to bring a device to market, but they cannot — and should not — be bypassed. Though agencies such as the FDA evaluate devices as a whole and not their discrete parts, it is to a manufacturer's advantage to build its devices with an OS that has a history of use in systems that comply with FDA or other regulatory bodies' requirements. Using such an OS does not guarantee smooth sailing through to certification, but it does greatly reduce the unknowns and allow efforts to focus on the device-specific design and development.

Technical requirements

The technical requirements for an OS for remote-care medical devices can be grouped into three broad categories:

- Dependability — responds correctly to events in a timely manner, for as long as required (sometimes loosely called "performance").
- Connectivity — communicates with diverse devices and systems, either directly or through networks.
- Data integrity and security — data is safely stored, and protected from unauthorized scrutiny.

Platform independence

To these requirements we can add platform independence. While it is unlikely that any one OS will run on every hardware platform available, an OS that will run on different hardware architectures and multiple boards offers distinct advantages. It allows a manufacturer to develop modular systems that can be re-used for different product lines and different versions of the same product.

For example, a manufacturer might market both professional and a home remote-care patient monitoring systems (see Figure 1 above). The home device, used by patients

[6] Steve Balacco, *et al. 2010 Survey Year, Track 2: Embedded System Engineering Survey Data, Vol. 3: Vertical Markets*. VDC Research. 2010. p. 19-20.

themselves, might include only a subset of the capabilities offered with the professional systems, and, therefore, might not need the computing power required of the professional system, and it might be considerably cheaper. Re-using the same OS and modules built for the professional system in home systems running on lower-cost boards would not only simplify development of multiple product lines, but it would also reduce their cost.

About Dependability

Dependability is a combination of two characteristics:

- **Availability** — how often the system responds to requests in a timely manner
- **Reliability** — how often these responses are correct

In other words, a dependable OS is an OS that responds when it is required in the time required, and responds correctly. The question, then, is: "What should an embedded OS in a medical device look like?"

RTOS Versus GPOS

If dependability is indeed essential to a system, this system's OS should be a realtime operating system (RTOS) rather than a general purpose operating system (GPOS).

GPOS — general purpose operating systems

The essential problem with GPOSs is that they offer "best-effort performance". They may be slow or fast, wasteful or efficient, but whatever they are doing, and however brilliantly they do it, GPOSs can offer no hard guarantees that they will *always* perform as required. GPOSs are designed to do many things and to do them well, often extremely well, but they are not designed to offer the strict guarantees of availability and reliability required of a medical device.

Add to this the economic disadvantages of deploying a GPOS in devices produced on a scale where even a $1 reduction in per-unit hardware costs can save the manufacturer a small fortune. These devices cannot afford the cost (not to mention the heat dissipation) of multi-gigahertz processors. Using a GPOS for such devices not only risks increasing the cost of the device, but does so while offering less.

RTOS — realtime operating systems

In contrast to GPOSs, RTOSs are engineered to guarantee availability and reliability. A designer building an embedded system with an RTOS can be confident that the OS will always be available when it is expected to be available, and that it will always perform tasks as expected. This fundamental characteristic of RTOSs not only ensures the devices they run can meet the most stringent technical and legislative requirements, but it can also save the device manufacturer money[7].

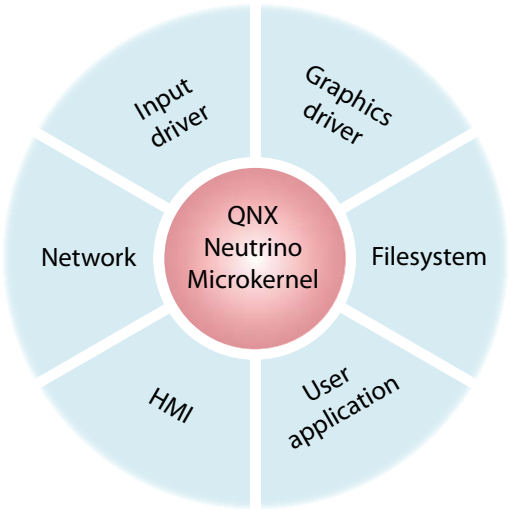

Figure 3: In a microkernel RTOS, system services run as standard, user-space processes. A failure in one user-space is isolated to that space; the microkernel and other user-spaces are protected.

RTOS Architectures

Simply stated, an RTOS is required for any embedded system used anywhere except, perhaps, low-end consumer disposables. Not all RTOSs are the same, however, and the wise will examine closely each RTOS under consideration, starting with their architectures, for architecture has a profound effect on a system's reliability and ability to recover from faults. The three most common RTOS architectures are *realtime executive*, *monolithic*, and *microkernel*.

[7] For a more detailed discussion, see Paul Leroux, "Exactly When Do You Need an RTOS?" QNX Software Systems, 2009.

Realtime executive architecture

The realtime executive model is now 50 years old, yet still forms the basis of many RTOSs. In this model, all software components — kernel, networking stacks, filesystems, drivers, and applications — run together in a single memory address space.

While efficient, this architecture has two immediate drawbacks. First, a single pointer error in any module, no matter how trivial, can corrupt memory used by the kernel or any other module, leading to unpredictable behavior or system-wide failure. Second, the system can crash without leaving diagnostic information that could help pinpoint the location of the bug.

Implementing a realtime executive architecture in a remote-care patient monitoring system makes sense only if the system is not doing anything terribly important. Even a device as simple as an in-home medication dispenser cannot afford a crash. First, a crash would confuse the person the device is supposed to be helping. Second, if the crash causes data loss or corruption, the dispenser might skip or double a medication, which could be dangerous or fatal for the patient.

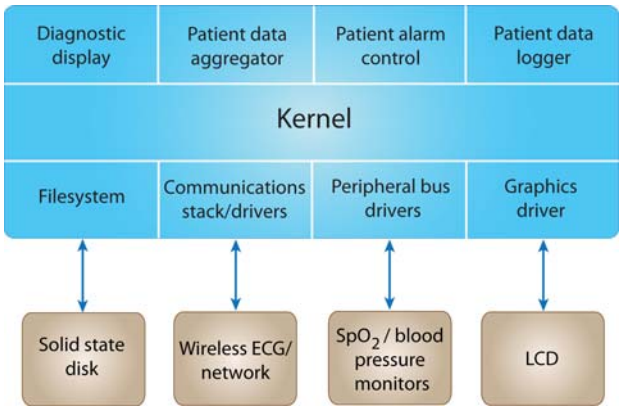

Figure 4: In a realtime executive, any software module can cause system-wide failure.

Monolithic architecture

Some RTOSs attempt to address the problem of a memory error provoking a system-wide corruption by using a monolithic architecture in which user applications run as memory-protected processes.

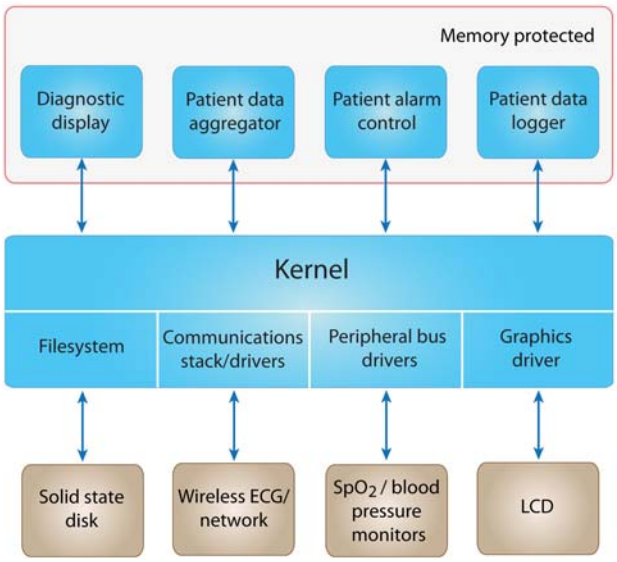

Figure 5: In a monolithic OS, the kernel is protected from errant user code, but can still be corrupted by faults in any driver, filesystem, or networking stack.

This architecture does protect the kernel from errant user code. However, kernel components still share the same address space as filesystems, protocol stacks, and drivers. Consequently, a single programming error in any of those services can cause the entire system to crash. As with systems built on OSs with realtime executive architectures, systems built with monolithic architectures may have difficulties meeting the dependability requirements of all but the most trivial medical devices.

Microkernel architecture

In a microkernel RTOS, applications, device drivers, filesystems, and networking stacks all reside outside the kernel in separate address spaces, and are thus isolated from both the kernel and each other. This approach offers superior fault containment: a fault in one component will not bring down the entire system. With realtime executive and monolithic operating systems, recovery would require a device reboot — a process that can take seconds to minutes, undermining the system's ability to meet its availability criteria.

Key RTOS Characteristics

The operating system's architecture is of course only one of many design characteristics that must be evaluated when choosing an RTOS. Other important characteristics include the RTOS's ability to:

- meet realtime commitments by pre-empting lower priority kernel calls
- prevent unpredictable behavior and system failure due to priority inversions
- guarantee availability by CPU resource scheduling to prevent critical processes starvation
- facilitate migration to multicore systems, and ensure efficiency and correct behavior on these systems

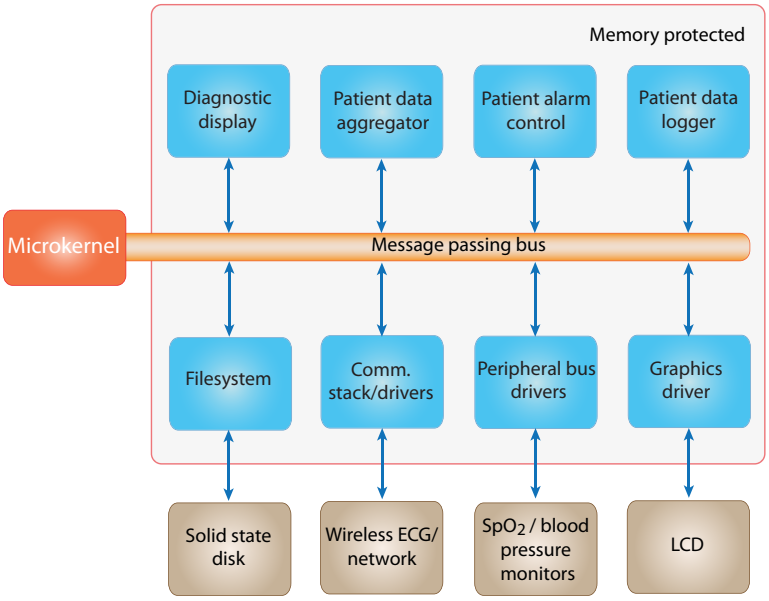

Figure 6: In a microkernel OS, memory faults in drivers, protocol stacks, and other services cannot corrupt other processes or the kernel. The OS can automatically restart any failed component, without a system reboot.

Meet realtime commitments

A pre-emptible kernel is essential to any system that relies on tasks completing on time; that is, any system that requires better than low-end consumer-grade dependability. For instance, an alarm triggered when a patient falls should be able to pre-empt processes drawing a diagnostic display, as should processes required by the communications stack in order to send the alarm out. It doesn't really matter how long it takes the system to display a reminder to eat lunch if the person being reminded is lying on the floor with a broken hip. The alarm and the communications stack need to get in and summon help.

In most GPOSs, the OS kernel is not pre-emptible; that is, a high-priority user thread can never pre-empt a kernel call, but must instead wait for the entire call to complete — even if the call was invoked by the lowest-priority process in the system. To make matters worse, all priority information is usually lost when a driver or other system service, usually performed in a kernel call, executes on behalf of a client thread. Such behavior can cause unpredictable delays and prevents critical activities from completing on time.

In an RTOS, on the other hand, kernel operations are pre-emptible. As in a GPOS, there are time windows during which preemption may not occur; though in a well-designed RTOS, these windows are extremely brief, often in the order of hundreds of nanoseconds. Moreover, the RTOS imposes an upper bound on how long preemption is held off and interrupts disabled; this upper bound allows developers to ascertain worst-case latencies.

To realize this goal of consistent predictability and timely completion of critical activities, the RTOS kernel must be as simple and elegant as possible. The best way to achieve this simplicity is to design a kernel that includes only services with a short execution path. By excluding work-intensive operations (such as process loading) from the kernel and assigning them to external processes or threads, the RTOS designer can help ensure that there is an upper bound on the longest non-pre-emptible code path through the kernel.

Protect against priority inversions

An RTOS with a microkernel architecture provides a qualitative advantage in reliability over other RTOS designs, but it cannot protect a system from all possible errors. One of the more common — and notorious — errors is *priority inversion*. The problem that, infamously, plagued the Mars Pathfinder project in July 1997,[8] priority inversion is a

[8] Michael Barr, "Introduction to Priority Inversion". *Embedded Systems Programming,* Volume 15: Number 4, April 2002.

condition where a higher-priority task is prevented from completing its work by a lower-priority task.

For example, the higher-priority task (alarm control) must wait for the lower-priority task (data logger) to complete before it can continue. A third task (data aggregator) has a lower priority than the alarm control, but a higher priority that the data logger. The data aggregator preempts the data logger, effectively preempting the alarm control, even though it has the highest priority of the three jobs. Blocked by the data logger, the alarm control can no longer meet its real time commitments.

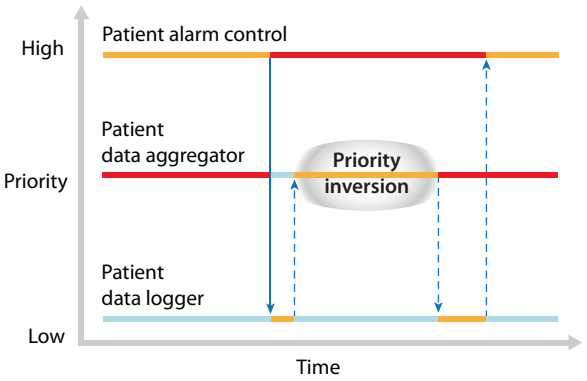

Figure 7: The patient data logger blocks the patient alarm control, even though the alarm control has a higher priority — a classic priority inversion problem.

Priority inheritance

Priority inheritance is a technique for preventing priority inversions by assigning the priority of a blocked higher-priority task to the lower-priority thread doing the blocking until the blocking task completes. In the example above, the data logger would inherit the alarm control's priority, and hence could not be preempted by the data aggregator. It would complete and revert to its original priority, and the alarm control would unblock and continue, unaffected by the data aggregator.

If there is no mechanism to ensure that the higher-priority task completes its work on time, the data logger might be preempted indefinitely. This unbounded priority inversion would prevent the alarm control from meeting its deadlines. With critical deadlines being missed, the consequences of priority inversion can range from unusual system behavior to outright failure.

Priority inheritance enables complex systems to meet their guaranteed realtime commitments, dramatically increasing system reliability. When selecting an RTOS for a medical device, gaining an understanding of the mechanisms it uses to prevent priority inversion, and confirming the effectiveness of its priority inheritance implementation will pay huge dividends, both during product development and certification, and for the duration of its market life.

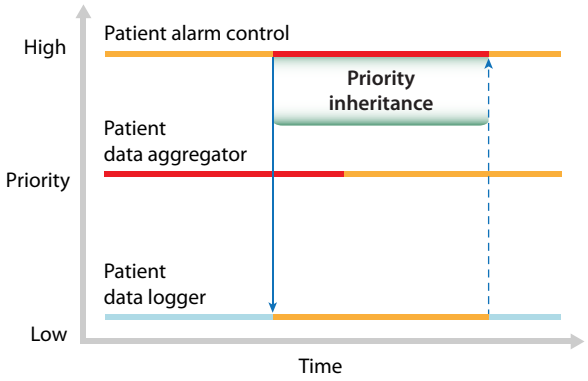

Figure 8: The patient data logger inherits its priority from the patient alarm control, which boosts its priority so that it cannot be pre-empted by the patient data aggregator.

Guarantee availability

For many systems, guaranteeing resource availability is critical. If, for instance, a key subsystem is starved of CPU cycles, the services provided by that subsystem become unavailable to other subsystems and, ultimately, to users, with possible dire consequences. For example, a heart monitor linked to a central monitoring system over a network that loses connectivity may cause the central system to incorrectly assume an alarm condition and dispatch help, or — far worse — the patient may be in distress with no one alerted and no help forthcoming.

Process starvation can have a variety of causes, from denial-of-service attacks (DoS), to the addition of new software functionality. Too often, just as the addition of a single car can cause traffic on a highway to grind to a halt, the addition of a new feature to a software system can result in applications being starved of CPU time and failing to respond and execute as required.

Historically, the solution to this problem was to either retrofit hardware or to recode (or redesign) software — both

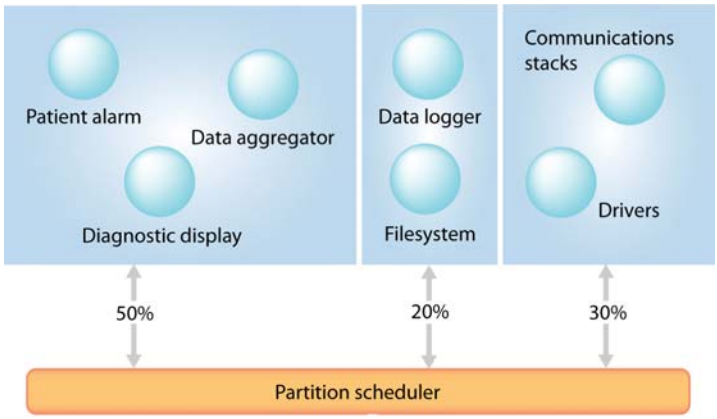

Figure 9: Fixed partitioning guarantees that processes will get the resources specified by the system designer, but lacks flexibility.

undesirable alternatives. While it would be technically possible to push redesigned software to connected medical devices, not only would the software redesign be costly, but it would likely invalidate the device's certification. Hardware retrofits would amount to a product recall, with all the attendant damages to the manufacturer's reputation and revenue.

Medical device manufactures must ensure that their systems implement mechanisms to guarantee resource availability, preventing resource starvation due to error or increased application loads, and restarting failed processes to maintain system integrity.

Partitions

Partitioning addresses the problem of resource starvation by enforcing CPU budgets, either through hardware or software. It prevents processes or threads from monopolizing CPU cycles needed by other processes or threads. Two types of partitioning are possible: fixed partitioning and *adaptive* partitioning.

With an RTOS implementing fixed partitioning, the system designer can divide tasks into groups, or partitions, and allocate a percentage of CPU time to each partition. No task in any given partition can consume more than that partition's statically defined percentage of CPU time. For instance, if a partition is allocated 30% of the CPU, the processes in that partition can't consume more than 30% of CPU time. This allocated limit allows processes in other partitions to maintain their availability, and can thus ensure that all key processes are always available.

Fixed partitioning has a significant shortcoming, however. The scheduling algorithm is fixed, so a process can never use more CPU cycles than the allocated limit of its partition, even if the cycles allocated to other partitions are unused. Fixed partitioning protects against resource starvation, but it also squanders CPU cycles and reduces the system's ability to handle peak demands. To work around this limitation, systems designers must use more-expensive processors, tolerate a slower system, or restrict the amount of functionality that a system can support. Thus, while fixed partitioning is a far better alternative than system failure or hardware retrofits, it is also far from ideal.

Adaptive partitioning

Adaptive partitioning is a relatively new partitioning model. Like fixed partitioning, adaptive partitioning lets the system designer reserve CPU cycles for a process or group of processes to create a system whose parts are all protected against resource starvation.

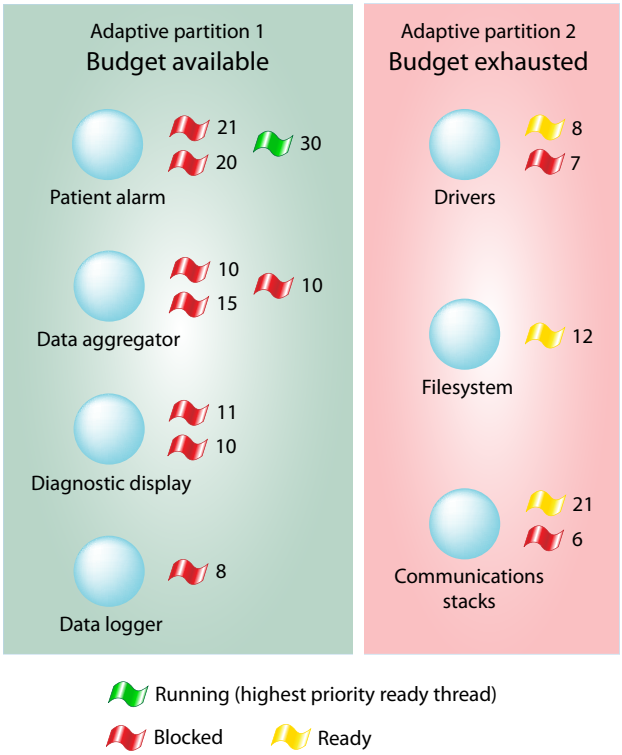

Figure 10: Adaptive partitioning is a set of rules that protect specified threads and groups of threads, and is an excellent solution for dynamic embedded systems.

QNX Adaptive Partitioning

QNX adaptive partitioning can be overlaid on top of an existing system without code redesign or modifications.

In the QNX Neutrino RTOS, a system designer can simply launch existing POSIX-based applications in partitions, and the RTOS scheduler ensures that each partition receives its allocated budget. Inside each partition, each task continues to be scheduled according to the rules of priority-based preemptive scheduling; there is no need to change application scheduling behaviors.

As a final step, the designer can dynamically reconfigure the partitions to fine-tune the system for optimal performance.

QNX High Availability Framework

The QNX high availability framework enables developers to construct custom failure-recovery scenarios, and design systems to recover quickly and transparently. It provides tools for building systems that isolate and even repair faults before they domino through a system.

The QNX high availability framework includes a high availability manager API library, which offers a simple, thread-safe mechanism for communicating with the high availability manager; and a client recovery library, which provides a drop-in enhancement solution for many standard libc I/O operations.

Unlike static partitioning, adaptive partitioning uses a dynamic scheduling algorithm; it dynamically reassigns CPU cycles from partitions that are not using them to partitions that can benefit from extra processing time. Partition budgets are enforced only when the CPU is running to capacity. Adaptive partitioning thus lets systems run at 100% of capacity. All available cycles are used if they are needed, but when processes in more than one partition compete for cycles, the partitioning enforces resource budgets and prevents resource starvation. Designers can count on resource guarantees, while not having to work around what is, in effect, the reduced capacity of their processers imposed by static partitioning.

Self-healing systems

A microkernel RTOS architecture offers excellent safeguards against process failures cascading through the system. Devices requiring high availability guarantees may also implement hardware-oriented high-availability solutions, and a software high availability manager.

This manager is a process that monitors the system and performs multi-stage recoveries whenever system services or process fail or no longer respond. A high availability manager should:

- automatically restart failed processes — without a system rebooting
- automatically recover inter-process communications following process failures
- perform customized failure recovery actions, where applications identify failure conditions and perform specified activities to mitigate consequences and speed recovery
- be self-monitoring and resilient to internal failures; if, for whatever reason, the high availability manager is stopped abnormally, it must immediately and completely reconstruct its own state by handing over to a mirror process

Multicore support

Not all remote-care medical devices are good candidates for multicore processing; their computing requirements do not justify the greater cost and complexity of multicore. That said, multicore processing is becoming essential to meet the processing demands of portable medical imaging systems.

As more sophisticated imaging and measurement technologies become available, more and more medical devices of all kinds will need multicore processing just to handle the data they receive. Remote-care devices, even devices for use in the home by the patients themselves, are no exception. The aging populations of industrialized countries suggest a growth in the need for in-home and personal devices that support sophisticated monitoring activities coupled with simple interfaces and even voice-promoting.

Thus, even if a manufacturer's current devices do not require multicore processing, future devices almost certainly will. It would be wise to review, for each RTOS under

consideration, not only how well these RTOSs run on multicore systems, but also what they do to support migration from single-core to multicore. For instance, do they implement processor affinity, so that applications originally designed for single-core systems can be moved to multicore without redesign? And, if so, how dependable is this new implementation, and how much time and work will be required to achieve certification? In short, an RTOS should not only support multicore implementations, but it should facilitate eventual migration from single-core to multicore platforms, and reduce the risks associated with this migration.

Connectivity

Support for networking protocols should be high on an RTOS capabilities shopping list.

Connectivity is already a key capability of remote-care medical devices, and its importance will only increase as patients, healthcare professionals, hospitals, emergency services, and insurers reap its benefits.

Connectivity for remote-care medical devices means the ability to connect to and interact with other local devices, both medical and non-medical, and it means connectivity to the network, and through the network to everything from secure data storage to emergency services.

For example, a blood pressure monitor could be connected via Bluetooth to a voice-prompt medicine dispenser that informs the monitor when the patient is instructed to take medication. Both devices could be connected to the patient's electronic file at a centralized location, as well as a nursing station if the patient is in a hospital or other care facility.

Network connectivity — either directly or through a smart phone — would give the patient complete freedom of movement, while maintaining communications with all essential components in the patient-care infrastructure: family, physicians, hospital, and even emergency services. The patient could work, play, shop, travel and even vacation overseas confident that monitoring continues uninterrupted.

Depending on the type of device — more specifically, the nature of data the device transmits and receives — it may be useful or necessary to build connectivity to standards such as IEC 61784, which defines how to use a 61158 "unreliable" medium (ethernet, for instance) for the reliable transmission of safety-related information.

User Interface Design

Discussions of user interface design, the good, the bad, the ugly, and the downright dangerous could fill a few miles of shelf space, and is far beyond the scope of this short paper. Even limited to a very specific implementation, such as a blood pressure monitor for people with mild cognitive impairment, the discussion might well merit a doctoral dissertation. Suffice to say here that the RTOS for a portable medical device should offer:

- support for whatever HMI design the device requires, including the concurrent use of multiple technologies, such as Open GL ES and Adobe Flash
- a system architecture that facilitates the re-use of the same underlying system with a variety of user interfaces, for instance, for devices with similar functionality but requiring different interfaces, say home and professional

Data Integrity and Security

The confidentiality of patients' medical records is a primary concern of physicians, hospitals and other healthcare institutions, the patients themselves, insurers, and governments. The stiff penalties for breaches of privacy and security stipulated by the U.S. 2009 HITECH Act is just one example of how seriously governments treat the question. Similarly, Canadian provinces (which are responsible to healthcare delivery in Canada) have strict Health Privacy Acts; the U.K. has the Data Protection Act, and so on.

Confidentiality is not sufficient, however. Data must also be accurate, and available when required. In short, data integrity and security means that:

- nothing bad happens to the data — it is not corrupted, erased or lost
- only authorized users (persons and systems) have access to the data — default is no access, and permission to read, write and copy must be explicitly granted

Data integrity and security depend on a host of factors; key among these are kernel design, filesystem support, and security encryption and authorization protocols.

Kernel design

Kernel design is critical to data integrity and security simply because if the kernel is compromised, everything is vulnerable. Thus, the more dependable the kernel, the better the data.

"RTOS Architectures" above presents the high-level argument for an RTOS microkernel architecture. An observation about microkernels made by Eugen Bacic in a 2006 paper is, nonetheless, worth quoting here:

> Microkernel architecture makes for a more secure and safe system since the actual *security aware* component — the microkernel — is small and easily understood with a focus on what has been historically defined as *security relevant*.[9]

In other words, a microkernel architecture greatly improves system security, because it isolates processes in protected user space, and what remains in the kernel is relevant to security, and is relatively small — small enough to be easily understood and, hence, debugged, optimized and hardened.

Filesystem support

Filesystems come in a wide range of flavors, each more or less appropriate for different systems and implementations. A detailed discussion of filesystems is beyond the scope of this paper. However, it is worth noting here that when evaluating an RTOS's filesystem support, it is important to understand how the medical device or devices will use filesystems, and, with this information in mind, to ask more than if the RTOS supports this or that media (NAND, NOR RAM), and this or that filesystem types (POSIX, Ext2, FAT, NTFS, etc.). An evaluation of an RTOS's filesystem support should also include questions about:

- Performance—does the RTOS filesystem support include garbage collection, file defragmentation, etc., and are bad sectors marked and bypassed?
- Error recovery — what protocols does the RTOS follow to recover from a fault, such as an unexpected power loss?
- Restarts — can the filesystem be restarted or even upgraded without stopping the system?
- Accessibility — can the filesystem be accessed through standard POSIX / C API calls, such as *open()*, *read()*, *write()*, *close()*, etc.?

QNX Safe Kernel

The QNX Neutrino RTOS Safe Kernel has been certified by Sira to conform to IEC 61508 at Safety Integrity Level 3 (SIL 3).

It incorporates all characteristics required of an IEC 61508 safe kernel:

- Design safe state — a well-defined state to which the kernel reverts when it encounters a situation it cannot handle
- Isolation — between application processes, and between applications processes and the kernel itself
- Scheduling predictability — guaranteed processor resources according to thread priorities, assurance against "lazy" resource allocations, and scheduling analysis through techniques such as deadline and rate monotonic scheduling

QNX Secure Kernel

The QNX Neutrino RTOS Secure Kernel is certified to meet the stringent requirements of the Common Criteria ISO/IEC 15408 Evaluation Assurance Level (EAL) 4+.

The Target of Evaluation (TOE) includes not only the mature QNX Neutrino OS kernel, but also its multi-core (symmetric and bound multiprocessing) and secure partitioning technology.

Encryption and authorization

To ensure data security, and hence confidentiality, a system must support encryption and authorization protocols for the required levels of security. Treatises on the subject could fill a library — with a good number of shelf-miles most probably

[9] Eugen Bacic, "Security as a Core Competency of the QNX Neutrino Microkernel". Cinnabar Networks (Bell Security Solutions) and QNX Software Systems, 2006.

classified — and there is little that needs to be added here. Briefly, whatever the level of data security a medical device or line of devices will implement, the RTOS will have to support protocols such as IPSec (Internet Protocol Security) , WPA/WPA2 (WiFi Protected Access), IEEE 802.1X (Extensible Authentication Protocol over IEEE 802), and RADIUS (Remote Authentication Dial In User Service), and local data encryption to DES (Data Encryption Standard) requirements.

Compliance

Ensuring that their products comply with standards and legislation in the jurisdictions where they will be marketed and operated is a critical part of any medical device manufacturer's product plan. Compliance is a necessary condition for getting a device to market, a *sine qua non* condition that must be met before investments can be transformed into revenue.

Like other businesses, medical device manufacturers are caught in a perpetual race against time to beat the competition to market. Timing is not as calendar-driven as for consumer electronics. A new blood analysis unit does not have to make the Christmas shopping season, for instance; but timing is still critical. The longer a device takes to develop, the more the development is likely to cost, and, the greater the window for the competition to secure a market share. This is particularly true for new technologies and existing technologies entering new markets. Often, the first one in sets the standard; all others are forced to play catch-up. One need only to be reminded that acetylsalicylic acid (ASA) is commonly known as Aspirin, the name Friedrich Bayer gave the drug when he began marketing it in 1899.

Anything that speeds a device through the certification process (or processes, as there are as many certifications as there are jurisdictions) not only saves time and money, but also increases the likelihood that that device will secure a significant share of its market.

The use of device components, such as the RTOS kernel, that have already received safety certification, such as the IEC 61508 Safety Integrity Level 3 (SIL 3), should come complete with all the relevant documents, including documentation of the development and verification processes. An IEC 61508 SIL 3 certified kernel reduces the number of unknowns, and hence may help reduce the time and effort required to certify a medical device to, for example, IEC 62304. Similarly, working with a supplier who has a history of successful certifications can reduce the uncertainties and speed product certifications[10].

Tools

In the its *Embedded System Engineering Survey Data* report for 2010, VDC Research notes that 55.3% of respondents in the medical embedded systems sector reported that their projects were behind schedule, with almost 40% more than three months behind[11]. The reasons cited for these delays are many and various, ranging from project complexity to poor management.

Whatever the reason or reasons a project falls behind, tools should never be to blame. Especially with the increasing demand for multicore processing in embedded systems, it is essential to look closely at the tools available for developing, debugging, testing and implementing systems running on the preferred OS. If these tools are inadequate or difficult to use, it may be worth reconsidering the choice. After all, no matter how well a system runs once it is implemented, the longer it takes to complete and implement, the more it will cost, and the less successful will be the project.

Tools and compliance

A further benefit of a using a good tool set is that it can help provide concrete evidence of functionality and behaviors in a given system. If a tool set can, for example, offer code coverage, system profiling and memory analysis[12], the artifacts it delivers can be included as evidence when building the case for regulatory compliance.

---

[10] See Chris Hobbs, "Using an IEC 61508-Certified Kernel for Safety-Critical Systems." QNX Software Systems, 2010.

[11] Steve Balacco, *et al. 2010 Survey Year, Track 2: Embedded System Engineering Survey Data, Vol. 3: Vertical Markets*. VDC Research. 2010. p. 27.

[12] See, for example, Elena Laskavaia, *et al.*, "Memory Errors in Embedded Systems: Analysis, Prevention, and Risk Reduction". QNX Software Systems, 2010.

Conclusion

Aging populations and tightening health budgets in industrialized countries are driving a shift in healthcare delivery strategies from hospitals to remote-care and home care. These changes and new technologies are driving the markets for remote-care medical devices, which are growing at a pace few anticipated only a decade ago.

Medical devices must meet stringent requirements for dependability, and they must pass the most exacting compliance standards in order to be certified. Device manufacturers can reduce their costs (in time and money) and greatly improve their products' chances of success by paying careful attention to the characteristics of the operation system they select. Devices that cannot be allowed to fail and reboot require a microkernel RTOS with viable strategies for availability, resource allocation, connectivity, data integrity and security, and, in some cases, eventual migration to multicore systems. An RTOS supplier with a track record of successful product safety and security certifications will help reduce the costs of obtaining FDA, MDD and other certifications.

References

Bacic, Eugen. "Security as a Core Competency of the QNX Neutrino Microkernel". Cinnabar Networks (Bell Security Solutions) and QNX Software Systems. 2006. www.qnx.com

Balacco, Steve, *et al*. *2010 Survey Year, Track 2: Embedded System Engineering Survey Data, Vol. 3: Vertical Markets*. Natick: Massachusetts: VDC Research. 2010.

Canadian Medical Association. "Canada's Physicians lead the way in charting a new course for healthcare" (press release). 3 Aug. 2010. www.newswire.ca/en/releases/archive/August2010/03/c7852.html

Hicks, Robin. "How will technology change the future of healthcare?" FutureGov. 1 Sept. 2009. www.futuregov.asia/articles/2009/sep/01/how-will-technology-change-future-healthcare/

Hobbs, Chris. "Using an IEC 61508-Certified Kernel for Safety-Critical Systems." QNX Software Systems. 2010. www.qnx.com

Kharif, Olga. "Qualcomm, AT&T Move in on 'M-Health'". *Bloomberg Businessweek*. 23 Aug. 2010. http://www.businessweek.com/technology/content/aug2010/tc20100823_511801.htm

Laskavaia, Elena, *et al*. "Memory Errors in Embedded Systems: Analysis, Prevention, and Risk Reduction". QNX Software Systems. 2010. www.qnx.com

Leroux, Paul. "Exactly When Do You Need an RTOS?" QNX Software Systems. 2009. www.qnx.com

——. "Using Resource Partitioning to Build Secure, Survivable Embedded Systems". QNX Software Systems. 2009. www.qnx.com

Nagarajan, Shiv, *et al*. "Processor Affinity or Bound Multiprocessing? Easing the Migration to Embedded Multicore Processing". QNX Software Systems. 2009. www.qnx.com

Rosser, Walter W., *et al*. "Patient-Centered Medical Homes in Ontario. *New England Journal of Medicine*. 362:e7. January 2010. www.nejm.org/doi/full/10.1056/NEJMp0911519

United Nations Population Division. *World Population Prospects: The 2008 Revision Population Database*. 2008. http://esa.un.org/unpp/

World Health Organization. *The world health report 2008: primary healthcare now more than ever*. Geneva: 2008.

About QNX Software Systems

QNX Software Systems is the leading global provider of innovative embedded technologies, including middleware, development tools, and operating systems. The component-based architectures of the QNX® Neutrino® RTOS, QNX Momentics® Tool Suite, and QNX Aviage® middleware family together provide the industry's most reliable and scalable framework for building high-performance embedded systems. Global leaders such as Cisco, Daimler, General Electric, Lockheed Martin, and Siemens depend on QNX technology for vehicle telematics and infotainment systems, industrial robotics, network routers, medical instruments, security and defense systems, and other mission- or life-critical applications. The company is headquartered in Ottawa, Canada, and distributes products in over 100 countries worldwide.

www.qnx.com