TERRY COATTA, INDEPENDENT CONSULTANT

THE (NOT SO) HIDDEN COMPUTER

**The growing complexity of
purpose-built systems is
making it difficult to conceal
the computers within.**

Ubiquitous computing may not have arrived yet, but ubiquitous computers certainly have. The sustained improvements wrought by the fulfillment of Moore's law have led to the use of microprocessors in a vast array of consumer products. A typical car contains 50 to 100 processors. Your microwave has one or maybe more. They're in your TV, your phone, your refrigerator, your kids' toys, and in some cases, your toothbrush.

The increasing use of microprocessors in consumer goods is not a new trend, of course. Engineers and developers have been working on embedded systems for decades. In the past, however, the fact that a device contained a processor or two might not be immediately obvious to the outside observer. The box did some job, and the vast majority of us could remain blissfully ignorant of the technology that lay hidden away, silently performing its digital miracles.

The advent of VoIP phones, MP3 players, DVRs (digital video recorders), and a host of other digital devices, however, means that people are interacting more directly with technology on a daily basis, manipulating digital information in ways that are dramatically altering our relationship with media and information.

THE (NOT SO) HIDDEN COMPUTER

What does this mean for software developers? Certainly that more of us will be working on code that runs in some sort of device other than a traditional computer. That might not seem like a monumental change in the computing landscape, because embedded computing is a well-established discipline. So if you haven't worked on embedded systems before, there's a sizable body of books and courses that can help bring you up to speed.

There are interesting differences in the newer devices and the types of issues that come up in designing and creating the software for them. To a large extent the past of embedded computing has been about creating software that monitors and controls various aspects of the devices that it resides within. From the outside, it is mostly invisible—just another component inside the black box.

As digital technology has penetrated more fully into the mainstream, we are seeing more software that, like an embedded system, is created for a specific device, but that is visible outside the box, too. In addition to the traditional issues facing embedded systems designers, these *purpose-built systems* raise issues of interfacing with the user and integrating with other devices.

The user interface issues are perhaps the most challenging. Given that most purpose-built systems will not have a keyboard and a 17-inch LCD attached to them, one issue is how to get information in and out. Should there be one button or two? Perhaps a thumb wheel? Will it have an LCD screen? How big? Does it need a remote control? What other kinds of environmental sensors, such as temperature or motion, might be appropriate? Once you have settled on the physical characteristics of the I/O components, there is the question of how to use each one. Should the concept of double-clicking apply to the buttons on the device? Given a restricted set of controls and output capabilities, balancing function against ease of use will be an ongoing concern.

A related challenge is deciding how a given device should integrate with other products. This spans the scale from how multiple units of a given product interact with one another to how the product works with your home or office network. For example, should a music player have the ability to exchange songs with other players using Bluetooth or Wi-Fi? Obviously, this involves not only design and implementation issues, but also consideration of digital rights management, security, and privacy. For example, what are the implications of having the player act as a little server, ready to exchange songs with any other device within range without alerting its owner? Once a device has networking capability, a vast array of integration possibilities arises. What capabilities should it expose to the network? Does it communicate with the user's PC? Does it require specialized software to run on the user's PC? The list is virtually endless, so there are many decisions to be made.

Another area of interest for purpose-built systems is a trend toward using more powerful general-purpose software environments. I once worked on an embedded system to control a piece of hydraulic machinery. The computing environment was very simple. I worked in a fairly minimal assembler-like language that did not even have the ability to branch and that was geared to execute the instruction sequence over and over again with guaranteed timing. The programming environment was simple enough that I could actually prove various properties of the control logic, which was useful in making safety guarantees.

It is not uncommon to see a purpose-built system running Linux. Surveillance cameras, home networking products, DVRs, GPS units, phones, PDAs, and even toys use Linux. Given a general-purpose operating system and programming environment such as this, there's potentially a lot of work to do to make a system that runs 24/365 without requiring any administration by its user. The open source community is undertaking some of this work. Special *embedded* versions of Linux take care of some issues, such as making sure not to fill up the disk (or other storage device) with log files. Even if you are using such a system, there are still issues of robustness and error recovery for the application you're building on top of it. For example, is it reasonable simply to reboot the system when something goes wrong and hope that fixes the problem?

THE PUSH TOWARD COMPLEXITY

While it might seem sensible to avoid all this and approach a purpose-built system from the bottom up—

that is, to move as much functionality as possible into hardware and build a simple and reliable software system for it from scratch—the forces at work in the marketplace today are pushing in the other direction. Although the merits of simplicity are not to be underestimated, the need to support interaction with the user and to integrate with other systems is pushing toward more complex software.

Consider, for example, what is perhaps the canonical purpose-built device: the cellphone. Mine is a second-generation smart phone. In addition to being a phone, it has a camera, provides IP connectivity, has a Web browser, and lets me install third-party software. Of those features the only one I don't use much is the camera. These aren't frills that could easily be trimmed from the device without it losing much of its value. This level of functionality already means that it would have been virtually impossible to write all the software for the phone from scratch.

MORE

Related articles in ACM's Digital Library:

Bhatia, S., Consel, C., Pu, C. 2004. Operating systems: Remote customization of systems code for embedded devices. In *Proceedings of the 4th ACM International Conference on Embedded Software* (September).

Borriello, G., Want, R. 2000. Embedded computation meets the World Wide Web. *Communications of the ACM* 43(5).

Di Pietro, R., Mancini, L. 2003. Security and privacy issues of handheld and wearable wireless devices. *Communications of the ACM* 46(9).

Park, S. H., Won, S. H., Lee, J. B., Kim, S. W. 2003. Smart home—digitally engineered domestic life. *Personal and Ubiquitous Computing* 7(3-4).

These articles will be available online at www.acmqueue.com for an eight-week period beginning April 1. Want full access? Join ACM today at www.acm.org and sign up for the Digital Library.

Indeed, the phone is based on an operating system that provides a variety of core services.

Its operating system, however, is not as sophisticated as that on my desktop computer. The differences are particularly notable in two areas. First, the phone's operating system does not provide memory protection between processes. Second, it uses cooperative rather than preemptive multitasking. The first issue means that one bad app can take down the whole phone—not just the third-party applications running on the phone, but the *whole* phone. I've had more than a few situations in which I was in the middle of talking to someone and using one of the other applications on the phone, which then crashed and terminated my call as a result.

If I were a typical consumer, I probably wouldn't understand protected memory, but I would definitely be interested in a phone that didn't lose my calls just because of a badly behaved application. Even though providing protected memory means additional complexity, both in terms of hardware and software, it has definite value to the consumer.

Cooperative multitasking involves similar trade-offs. Without a doubt, it is easier to build an operating system based on cooperative multitasking, but the result is that when I run the MP3 player software on my phone, its response to other things (such as handling incoming calls) becomes sluggish. Given that I would like my phone to continue functioning as a phone regardless of what software I happen to be running, I consider this to be a fairly serious drawback. It is almost certainly the case that my phone doesn't have the CPU muscle necessary to play MP3s and to handle calls at the same time, but I would much rather get the calls; if my MP3 playback breaks up a little as a result, I can live with that. Again, the needs of the consumer here are pushing toward greater complexity in the software and hardware.

This last example demonstrates another interesting aspect of design in purpose-built devices. As implied by the name, these devices have some particular purpose that they were designed to fulfill. Sometimes, like the smart phone, they support a wide range of uses, but certain aspects of their feature set are primary. Although I use most of my smart phone's features, the one job it has to do really well all the time is to be a phone. If it can't do that, then I'm not going to use it regardless of how wonderfully it plays MP3s or browses the Web. While consumers will demand a broader set of functionality from devices, care needs to be taken in ensuring that these do not detract from the *purpose* of the device. A corollary is that the device should have a well-defined purpose. The market-

THE (NOT SO) HIDDEN COMPUTER

place is littered with devices that have failed to catch on because they couldn't quite decide what they should be. On the other hand, the success of the video iPod might suggest that we've reached the stage where a device can serve multiple purposes in a reasonably transparent manner. In either case, the result is the same: greater complexity in the software that drives these devices.

This pressure toward more complex systems is even being felt in areas that seem closer to traditional embedded systems. Most of the processors in your car are fairly unobtrusive, busily monitoring fuel flow or how fast the transmission is turning. As with the cellphone, however, consumer-oriented features are pushing greater degrees of integration. For example, in some cars the microprocessor in the transmission exchanges information with the microprocessor in the radio. Why? Because the volume of the radio can then be adjusted for different levels of noise that are produced as the engine varies its speed. Thus, the radio always seems to be at the same volume, despite the change in the level of ambient noise.

In fact, exchange of data among the various processors in a car has become so extensive that many cars now have a network that connects the processors. This has allowed the manufacturers to provide even more integrated functions and has at the same time significantly reduced the amount of wiring in the car. This is not without its dangers, however. An August 2003 article from Embedded.com notes that in-car networks were used to support automatic downward tilting of side mirrors when the car was placed in reverse. Processors in the mirrors received data from the processor in the transmission concerning what gear the car was in. This placed the network near the extremities of the car, so thieves were reportedly stealing cars by breaking off the mirrors, tapping into the network, and telling the car to unlock its doors and turn off its security system.

KEEPING LIFE INTERESTING FOR DEVELOPERS

Some may see this type of problem as the "wages of sin"—the sorts of failures that come from using something too complicated for the task at hand. After all, would it not have been more sensible simply to run a wire from the transmission to the mirror that signaled when the car was in reverse? As noted previously, however, the technology is not propelled solely by a desire for new features. Eliminating point-to-point wiring in the car saves weight, makes construction simpler, and improves diagnostic capabilities. The real lesson to learn here is that we have to start taking the lessons learned from building general-purpose computer systems and apply them to the software that is being developed for purpose-built systems.

It is unlikely that we are going to return to the era of simple self-contained software within our myriad devices. Intel started off 2006 with a fairly dramatic shift of direction for the company with its new Viiv platform, focusing less on pure computer horsepower and more on consumer media devices and laptops. This signals a trend for software developers as well. More of us will be creating software that runs on something other than a typical PC. We must address an abundance of design issues: human factors, power consumption, security, integration with other devices, whether to strip down a standard operating system to be robust enough or to build something more specific from the ground up, whether or not to use special hardware for certain functions; the list goes on. That's what makes life interesting for a developer. In the past, I've often assumed that I would be developing software for PCs for the rest of my career, but as purpose-built devices come of age, perhaps I won't be too surprised if I end up building a better Furby. ☺

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

TERRY COATTA is an independent consultant specializing in development methodologies. Prior to that he was the vice president of development at Silicon Chalk Inc., a small startup firm in Vancouver, British Columbia, that produced realtime collaborative software for use in higher education. He has also worked at Open Text Corporation and the Network Software Group. He has a Ph.D. in computer science from the University of British Columbia (1994), where his area of research was distributed systems. Coatta has worked with and continues to be interested in distributed component systems such as .NET, COM, EJB, and CORBA.

© 2006 ACM 1542-7730/06/0400 \$5.00