Miniature Software for Large Pervasive Computing Applications

David B. Stewart
Embedded Research Solutions, LLC
9687F Gerwig Lane, Columbia, MD 21046
dstewart@embedded-zone.com

There is a need for new software engineering methods that apply specifically and exclusively to the implementation of real-time code for pervasive computing applications. Applications will consist of hundreds or thousands of tiny processing units. While it is expected that model-based tools and aspect-oriented techniques will be used to reason and design the large complex applications, there is currently a huge gap between the use of these tools and techniques, and the underlying implementation of the executable code on future system-on-a-chip designs. To both support component-based design in these applications, while adhering to the very rigid constraints of the target hardware and devices, significant software advances are needed in the areas of separation of concerns, components, real-world interfacing, frameworks, multi-dimensional resource optimization, hardware/software co-design, and networking of miniature devices, as applied specifically to very small embedded systems.

Introduction

There is a need for new software engineering methods that apply specifically and exclusively to the implementation of real-time code for pervasive computing applications. Applications will have hundreds to thousands of processing nodes embedded into everything, including clothing, objects, soil, liquids, and any other medium for which sensory information or intelligence is desired. The processors will communicate amongst themselves and a base station, both through very fine wires if embedded within a single item like clothing, or wireless with perhaps all the processors within an area of a few hundred square feet. Each node—including the processor, sensors, actuators, and power source—will be smaller than one or two square centimeters, and in some cases barely visible to the naked eye.

What will the software look like for these systems? We call it *miniature software*. The term captures the physical nature of the underlying hardware, and requires entirely new approaches as compared to the software engineering advances in the desktop and larger embedded system domains. Technologies such as Java, CORBA, wireless networking protocols, POSIX, middleware, and UML-based modelling all lead to software that is simply much too big for the target hardware. The techniques have been modified and streamlined for use in embedded systems. But the results, such as Real-Time Java, RT CORBA, Bluetooth wireless networking, and Embedded UML, are still too big in terms of memory, processor, and power usage, by two or more orders of magnitude!

Instead of trying to adapt desktop approaches to miniature systems, new techniques are needed that are designed specifically to address the software needs of pervasive computing applications. In particular, software must be engineered with the same precision as mechanical assemblies, where every byte of data or code used on the target system would undergo the same scrutiny as the choice of a bolt, wire, or subcomponent in the mechanical assembly. Such a level of precision for software is non-existent today.

It is expected that the solution will use and combine the knowledge learned in the areas of aspect-oriented programming (AOP), model-based design, component frameworks, and hardware-software co-design techniques. This white paper discusses a number of relevant issues, and presents a vision of the resulting software environment.

Pervasive Computing Applications

The National Institutes of Standards and Technology (NIST) defined pervasive computing as, “the strongly emerging trend toward numerous, casually accessible, often invisible computing devices, frequently mobile or embedded in the environment, and connected to an increasingly ubiquitous network structure.” [4]

There exist potential applications of pervasive computing technology in nearly every aspect of our world. Some examples include inventory tracking, environmental monitoring of toxic substances, human physiological monitoring, and machine health and diagnostics. The list of potential applications is limited only by creativity. Some applications could have such an impact as to redefine and improve what is considered standard living in our society.

There has been some research seeking related solutions. For example, DARPA is already funding projects on topics related to autonomous negotiating teams, electronic textiles, and microsensor networks. While some of these projects do address small software, such as Berkeley’s TinyOS project [3], the general trend is to address a small number of issues in order to demonstrate feasibility of a specific application or algorithm. There is not a concerted

Table 1: Comparison of Characteristics of Representative Embedded Processors

Processor	Current Development Support	Application Constraints			Resource Constraints					Example Application that uses this processor
		Peak Power (mWatt)	Cost per unit; Quantity of 1000	Package Size (cm^2))	CPU Size (bits)	Internal RAM (bytes)	Internal EPROM or Flash (bytes)	Max External RAM+ ROM	Max CPU Speed (MHz)	
Microchip PIC12C509	IDE	2.0	$1.15	0.65	8	41	1.5K	0	4	electronic infant toys
Texas Instruments MSP430F1121	IDE	0.35	$1.75	1.0	16	256	4K	0	10	wireless sensor
Motorola MC68HC912B32CFU8	IDE	225	$12.20	2.0	16	2K	32K	32K	8	automotive
Motorola ColdFire MCF5206FT33A	IDE,RTOS,CASE	772	$15.27	9.0	32	1K	0	512M	33	internet appliance
Intel StrongARM SA-1110	IDE,RTOS,CASE	400	$35.00	2.9	32	24K	0	512M	200	PDA, smart phone

effort to create integrated tools and execution environments that would lead to commercial products that the average industry programmer can use.

The Target Hardware: Tiny System Units

A *tiny system unit* (TSU) is a very small system-on-a-chip (SoC) device that includes one or more processors with extremely limited resources, that would be replicated for each node in a pervasive computing application. The primary system constraints that dictate the characteristics of this TSU are power consumption, cost-per-unit, and physical size. Pervasive computing applications need processors that can operate for years without ever needing to recharge or replace a battery. Regarding cost, microcontrollers that cost a few dollars each are commonly used in many mass produced embedded applications, such as automobiles, consumer electronics, and home security systems. These applications replicate processors by the millions, where every dollar saved is a million dollar decrease in production costs. Pervasive computing applications lead to the potential of replicating nodes by the billions, such that each penny saved on the processor could result in a $10 million decrease in production costs.

A TSU's processor will likely have internal small word sizes and data paths, and have only a few kilobytes of memory. They are designed to operate with machine cycles ranging from just a few kilohertz to 10 or 20 MHz. TSUs will be an SoC generation device that replaces microcontrollers. The Atmel *AVR 94K Series Field Programmable System Level Integrated Circuit* illustrates this new trend. The device includes an 8-bit microcontroller, a 40K gate field programmable gate array (FPGA), memory, multiple clock circuits, and a number of other system functions [1]. The TSU will adopt and extend this approach, to provide at least one processor and possibly more, an FPGA so that the I/O and computational functions can be configured, user-defined memory organization, and built-in micro-electro-mechanical sensors and actuators, all within a device that is smaller than a dime.

Although the Atmel AVR 94K uses an 8-bit microprocessor core, we expect the majority of TSUs to have 16-bit word sizes. A pure 8-bit processor with 8-bit internal data paths is inefficient, because most operations, including data computations and accesses to the address space, still need to be done as 16-bit operations. But these small systems will not need 32-bit address spaces nor internal 32-bit data paths. Such larger processor cores will on average always be larger and cost more than a 16-bit counterpart that is built using the same technology, because it needs a larger die area and consequently bigger packaging. The 32-bit processor will also use more power than an equivalent technology 16-bit processor. If 16-bits is not sufficient, it is more likely for 18 or 20-bit processors to emerge as an extension of 16-bit processors for these special cases, rather than using a 32-bit processor system. As a result, we expect a revolution in 16-bit processor technology for use as the core in TSUs.

Table 1 compares several popular embedded processors. The Texas Instruments MSP430 series is an example of a new generation of processors for embedded systems. It is a 16-bit RISC processor that uses technology developed to build 32-bit devices. But by applying the techniques to the 16-bit platform, an ultra-low power very small processor was created. Like 8-bit microcontrollers, the MSP430 has very limited memory, and is designed to operate with machine clock rates in the Kilohertz to tens of Megahertz range. The Motorola 68HC12 series is another example of the trend moving to 16-bit processors; it replaces the popular 68HC11 8-bit processors. Although the 68HC12 has a larger CPU word size, its memory, size, cost, and power usage is comparable to the 68HC11. The processor cores for a first generation of TSUs is expected to resemble the cores of the MSP430 and 68HC12.

Important to note from the information in Table 1 is the lack of development tools for the 16-bit processors. The most commonly available tool is an Integrated Development Environment (IDE), which usually comprises of a compiler, assembler, linker, downloader, a symbolic debugger, and a simulator or emulator for the target processor. With the exception of the simulators that became popular in the 1990s, today's IDE represents the same set of tools avail-

able to embedded system developers in the 1970s. There are no advanced model-based design tools, component standards, AOP languages, visual configuration management interfaces, hardware/software co-design trade-off tools, or sophisticated real-time operating system (RTOS) or frameworks available. Such "modern" embedded software technology has only been usable on the larger processors, because they result in code that has too much overhead for TSUs.

The target TSU hardware establishes a need for new tools and techniques that aid in developing software for the tiny devices in pervasive computing applications.

Miniaturizing Software for TSUs

How can software be implemented for TSUs such that it allows easy-to-use modern software engineering approaches to be used, but results in code that is at least as efficient as an elegant assembly language program that might have taken experts much too long to develop? This section looks at the issues and possible approaches.

Separation of concerns: Traditional embedded code includes multiple modules, with the entire program specified using a single language. The problem with this approach is that independent concerns are merged into modules, and some of those concerns are programmed in indirect ways. For example, a real-time software design may contain a list of five tasks to execute. But nowhere in the code do we see a table that simply lists each of those tasks and their attributes. Instead, today's application-programmer interfaces (API) require the information be encoded into system functions, such as calling the thread create function for each task, and passing attributes as arguments.

The AOP field shows great promise, because it promotes the separation of concerns. Different aspects of the software include user components, task or component management, memory management, real-time constraints, error detection and handling, as well as a few more. Some aspects are best programmed using a visual approach using a model-based design tool. Other aspects, like memory management, might benefit more from a tabular representation. Yet other aspects, like the core code aspect, might still best be programmed using a procedural language like C or Java. AOP recognizes that there is not a single language or method that is suitable for programming all concerns.

AOP research focusing specifically on miniature software is needed, because the requirements and constraints of small embedded systems are very different as compared to desktop or larger embedded systems, and thus the concerns are different. Each concern needs to be investigated independently, and additional research is needed for weaving the separated concerns into a unified program that executes on the resource-constrained target hardware. [2]

Components: Model-based design is proving to be a valuable approach for reasoning about large, complex applications. The basic element in a design is the component. Given the definition of the component, code can be simulated, components integrated, and large systems assembled from smaller subsystems. Current model-based design techniques seek to generate code from models. Alternately, code for components is created using traditional programming techniques, such as mapping each component to a C++ object.

To compliment the use of model-based design of large systems with many tiny processors, a bottom-up approach to creating the components is needed. There could exist several domain-specific standard classes of components, such as control system class, user-interface class, communication class, device driver class, and error handler class, such that each component class had characteristics suited specifically for TSUs.

Implementing this bottom-up approach would likely yield different solutions as compared to starting with desktop domain standards such as POSIX and CORBA, then trying to streamline them to build processes and components for tiny embedded systems. By understanding the engineering issues related to implementing software on TSUs, the component classes would be properly designed for their intended use. A suitable model programmer interface (MPI) can then be designed that encapsulates the crucial parameters. An MPI is similar in concept to an API, but the target user is the model-based design tool, not an application programmer. In the same way that assembly language for RISC processors targets the compiler rather than an assembly language programmer, the interface for miniature software components needs to target the tools used to build applications, and not target the programmer.

Real-world Interfacing: Significant effort has been made in device driver design to abstract the peculiarities of hardware devices from the application. For many embedded systems, however, this is not practical, because the *device drivers are the application*! This is especially true for miniature software, as most components in the system interface to sensors, actuators, on-chip peripherals or communication mechanisms in a hardware-specific manner.

To successfully create miniature software, it is necessary to have encapsulation of hardware-specific details to be an integral part of the component model [7]. A device driver thus becomes a component for a custom actuator or unique configuration of sensors and processor I/O ports. The vision is that every hardware device, such as a temperature

sensor, position resolver, or motor controller has a corresponding software component. If the hardware is present in the system, then so is the software component. But if the hardware is not in the system or it is changed, then the software component undergoes a similar change, and managed by a framework configuration tool, as described next.

Frameworks: Today's approach to improve the virtual abstraction and add complex functionality is to build a system in layers. For example, middleware is a layer of software that sits above the operating system, to provide a unified component integration and management scheme. As another example, communication protocols are built by defining layers, such as the data link, network, transport, and application layers. This layered approach is problematic in the design of systems that need TSUs, because each layer increases the need for resources, and as discussed above, resources are extremely limited. As a result, many of the techniques developed for larger systems, whether they be desktop computing or embedded systems that use 32-bit processors, are not usable with TSUs.

Lightweight implementations of some techniques have been created to be used on TSUs, but these always require sacrificing important functionality or flexibility. For example, a very small number of RTOS exist for 8-bit and 16-bit processors. Their features, however, are usually limited to context switching, a fixed-priority scheduler, and possibly semaphores or message passing for interprocess communication. Unfortunately, these limited features are not enough to implement full component-based applications, and do not include the generality that is desired to support features such as dynamic scheduling, mobile code, or networking.

The alternative to using an RTOS and middleware is to build a component framework. A framework is an implementation of a modeling environment, providing the run-time mechanisms to manage and integrate components. It can be viewed as a merger of the RTOS and middleware layers, but in such a way to provide *only* the resources needed and to directly manage components, not processes. It provides functions for component initiation, activation and real-time scheduling, management of memory, power, and on-chip devices, and mechanisms for both local and distributed inter-component communication and synchronization.

The framework needs to be a configurable entity, targeted to specific hardware architecture and application. There is no room in a framework to include every feature just in case in might be used. Instead, to ensure efficient use of resources, a framework configuration tool is envisioned, that configures the framework for a particular hardware/application combination, and dynamically targets the framework to the appropriate TSU nodes. Unlike an RTOS that is designed for the general case, each instance of the framework will contain the minimum combination of mechanisms needed by the application—no more, no less. Furthermore, it will be possible to configure a separate framework for each processor in the system. For example, one node might be a TSU for which a multi-rate cyclic executive with static scheduling is sufficient to accomplish tasks. Another node might be a base-station with complex code that requires preemption, in which case the framework could include a hardware-based dynamic scheduling strategy. The same application components, however, plug into either framework.

Multi-Dimensional Optimization: In many applications, the term "optimization" refers to the amount of execution time used by an application. With miniature software, however, there are many dimensions to the optimization problem. Each dimension is a different part of the system that could be optimized. Examples of the different dimensions include processor utilization, memory, power usage, control system performance, physical size, communication bandwidth, available features, sampling rates, and cost. There are dependencies between each of these dimensions, such that improving the performance in one dimension may decrease performance in another. For example, consider unit size vs. real-time sampling. The battery is likely the largest part by volume of the unit. To increase sampling rate means an increase in battery size because more power is needed. If there is a size constraint on the device, then the battery size becomes limited; so to obtain a certain real-time sampling, it might be necessary to alter some other part of the system, like reducing how often data is transferred other nodes to conserve power. Transferring data less frequently leads to the need for larger memory to store data locally before transmission, and this in turn increases cost.

There are many trade-offs similar to the above example that come into play when building software for TSUs. Finding the right compromise between each item is a critical issue for the software designer. Currently, there are no tools to help the designer make the best decisions, and so an ad-hoc approach is used to try and identify the right balance between features and resource usage. Finding the optimal solution is an engineering problem that is similar to the optimization problems encountered regularly in the control systems community. The mathematical foundation of these existing optimization methods needs to be leveraged to create software engineering tools that allow the system designer to quickly make the right decisions when implementing or configuring their software for a TSU.

Hardware/Software Co-Design: Resolving the resource usage problems of creating miniature software could lie in successful adoption of hardware/software co-design solutions. For example, we designed a hardware-based interpro-

cess communication mechanism that eliminates all race conditions while reducing software overhead by up to 4000 percent [5]. The key is that when the particular interprocess communication mechanism is selected for use on a processor, then a corresponding hardware component—in this case a pre-programmed DMA device—is also included.

For mass-produced devices, the hardware portion of a co-design can be built using a custom ASIC that includes a processor soft core and the needed hardware add-ons. More commonly, however, TSUs, will include FPGAs that can be configured in the field uniquely for each application [1,6]. The hardware definition language (HDL) used to program these on-chip FPGA segments is, for all intents and purposes, software, even though the functionality is more commonly linked to hardware design. Thus a miniature software design could include code written using an HDL (such as Verilog or VHDL) in addition to code written with a traditional procedural language (like C or Java).

Networking of miniature devices: Pervasive computing applications targeted by miniature software are very localized. In most cases, the shear number of nodes will force the use of wireless communication between nodes, as wiring will be impractical or impossible. The nodes will be self-powered using very small batteries or ambient power source technology, so that batteries do not need to be recharged or replaced for months or years. Emerging communication standards such as Bluetooth—even in its lowest power modes—require several orders of magnitude more power than can be allotted for these pervasive computing applications. Instead, standardized ultra-lightweight communication protocols that focus on being able to transmit only a few bits per second per node must be designed and integrated into the component framework.

With dozens to hundreds of processors, it becomes impractical to explicitly program the inter-processor communication between each component. Since these applications will likely have some form of lightweight networking capability on every node, such networking strategies must be an integral part of any solution. Trying to adapt desktop networking strategies to this environment have not worked in the past, and because of the major differences in target environments, will likely not work in the future. Instead, entirely new approaches are needed.

The protocol must be encapsulated within the framework as a standard plug-in, with details of the communication abstracted from each individual application component. Using model-based design tools, the application programmer can define the entire application as a set of components that collect and pass data amongst themselves, without concern as to whether or not components are on the same processor. That is, the programming model of computation must be independent of the communication mechanism. Furthermore, there must also be a single point for human interfacing at runtime, allowing an application programmer to have a single personal computer or base station that can program, monitor and control activity of any sensor, actuator, or processor in real-time on the network.

Summary

There is a need for new software engineering methods that apply specifically and exclusively to the implementation of code for TSUs. These methods must not have the constraints of being compatible with techniques in the desktop domain, nor be biased in selecting solutions that were successfully deployed in larger systems. It is expected that top-down approaches including model-based tools and AOP techniques will be used to reason and design the large complex applications, while a bottom-up hardware/software co-design approach is needed to create the components, frameworks, device drivers, networking, and multi-dimensional resource optimization tools. The resulting code, that we call miniature software, will then be well suited for large, pervasive computing applications.

References

[1] Atmel, *AT94K Series FPSLIC Summary*, *http://www.atmel.com/atmel/acrobat/1138s.pdf.*

[2] T.W. Carley and D.B. Stewart, “Visual aspect-oriented programming of resource constrained real-time embedded systems using the port-based object model of computation,” Presented at OOPSLA Workshop on Domain Specific Visual Language, October 2001, *http://www.isis.vanderbilt.edu/oopsla2k1/Papers/Carley.pdf*.

[3] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, “System architecture directions for network sensors,” *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, Nov. 2000.

[4] Pervasive Computing Conference, sponsored by National Institute of Standards and Technology (NIST), *http://www.nist.gov/pc2000*, Gaithersburg, MD, Jan. 2000.

[5] S. Srinivasan and D. Stewart, “High speed hardware-assisted real-time interprocess communication for embedded microcontrollers,” *Real-Time Systems Symposium*, Orlando, FL. Dec. 2000.

[6] D. B. Stewart and B.L. Jacob, “Hardware/software co-design of i/o interfacing hardware and real-time device drivers for embedded systems,” *Real-Time Applications Symposium—Work-in-Progress*, Vancouver, Canada, June 1999.

[7] D.B. Stewart, “Software components for real-time,” *Embedded Systems Programming*, v.13, n.13, Dec. 2000.