Submitted to UCC Bulletin on March 3, 2001

The Problem of Embedded Software in UCITA and Drafts of Revised Article 2 (Part 2)

Philip Koopman, Ph.D.
Associate Professor
Electrical & Computer Engineering Department
Carnegie Mellon University
and

Cem Kaner, J.D., Ph.D.
Professor
Department of Computer Sciences
Florida Institute of Technology

This is the second part of a two-part article. The first part, available at www.badsoftware.com/embedd1.htm, focused primarily on UCITA; this one focuses more on Article 2. All references to UCITA are to the amended, commented draft dated July 28-August 4 2000, at www.law.upenn.edu/bll/ulc/ucita/ucita1200.htm. References to the latest scope proposal for Article 2 are to the "February 2001 Draft 2-103 Scope" at www.law.upenn.edu/bll/ulc/ucc2/scope01.htm.

1. Introduction

Having failed to develop a workable distinction between embedded and non-embedded software, the UCC Article 2 drafting committee now proposes to throw the problem at judges and tell them to make the distinction on a case-by-case basis. The draft provides four factors to help the judge decide when software is embedded. Unfortunately, as we will show below, these factors will not be of much help.

Why is this such a hard problem?

The Article 2B / UCITA drafting committee worked long and hard on this issue. Its meetings were attended by lawyers who had computer law sophistication. *Why did they ultimately fail to make a meaningful distinction between embedded and non-embedded software?* (See Part 1 of this paper at <www.badsoftware.com/embedd1.htm> for a detailed discussion of the UCITA problems.)

The Article 2 drafting committee has made substantial additional efforts to tackle this issue, especially over the past year, but they haven't had any greater success. *Why not?*

In our view as computer scientists, the reason nobody has succeeded is because they are working on a problem that is fundamentally impossible to solve. There is no principled distinction between embedded and non-embedded software, not even if you give "embedded software" a new name, like "smart goods" or "integrated software."

2. Turing Equivalence and Article 2

The reason that there is no principled distinction between embedded and non-embedded software is that the decision to make a program “embedded” (whatever “embedded” means) is merely an implementation choice, subject to the usual cost/benefit tradeoffs that influence engineering design decisions. Change the costs and benefits (for example by changing the regulatory structure associated with software) and the engineer can replace one solution with an equivalent alternative. From a Computer Science viewpoint, two alternative programs and the machines they run on can easily be equivalent even if one appears to be “embedded” and the other does not.

The concept of Turing Equivalence is based on one of the early discoveries in Computer Science (CS), the Church/Turing thesis (A. Church, "An Unsolvable Problem of Elementary Number Theory", 58 *Am. J. Mathematics* 346, 1936; Alan M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem", Series 2, 42 *Proc. London Mathematical Soc.* 230, 1936.) This is a basic result, studied by all CS majors, typically in their sophomore or junior year. For a detailed presentation, see Jack Copeland, "The Church-Turing Thesis" at <www.alanturing.net/pages/Reference%20Articles/The%20Turing-Church%20Thesis.html>. For a gentler introduction, see Steven Harnad, "Lecture Notes" at <www.cogsci.soton.ac.uk/~harnad/Hypermail/Foundations.Cognitive.Science2000/0055.html>.

The Turing Equivalence principle holds that any computer can perform the same computations as any other computer, given that it has enough memory and is given enough time. The general argument to support this principle describes a very simple computer called a "Turing Machine", and then demonstrates that it can compute the same functions as more complicated machines. Thus, all computers are said to be "Turing Equivalent".

One implication of Turing Equivalence is that a program written for one computer can be made to run on any other reasonable computer. This can be accomplished by recompiling the program, by simulating the instruction set of the first machine (such as PC emulation software running on a Macintosh), or by writing the program for a so-called virtual machine (such as the Java virtual machine) that runs on any computer. The important result of this is that there is no reason why a program written for a desktop computer can't be used in an embedded system, and no reason why a program written for use in an embedded system can't run on a desktop computer. And, indeed, we now see things such as the Windows CE operating system being used for both purposes, and desktop PC designs used in embedded products to simplify development efforts. (*See* Rebecca Buckman, "Microsoft Renews Push to Develop Chips that Run Embedded Systems, *Wall Street Journal (Interactive Edition*), Feb. 6, 2001.)

Another implication of Turing Equivalence is that any particular computer function can be implemented in a wide variety of ways, and each of these different programs is equivalent (except for perhaps the amount of memory used and the length of time to complete the computation). As a simple example, if you want to multiply the numbers 3 and 27, you can do it in hardware by using a multiply instruction, by using a program with a set of bit-shifting instructions that simulates the way people do multiplication with pencil and paper, by using only addition (adding the number 27 three times), by using a table of precomputed products for single digits similar to the times tables we all learned in grade school, or even by combinations of these techniques. The multiplication program could be written in any of dozens of programming languages, and compiled to one specific computer, to a virtual computer running on any computer, or simply interpreted directly from the program at run time without ever being compiled to a machine-executable language. All these techniques can and have been widely used for decades in software for both embedded computers and desktop computers. The choice of implementation technique is based on economics, ease of development, and sometimes other factors such as regulatory and legal

constraints. Additionally, if one has sufficient computing power, it is always possible to replace software with hardware, and special-purpose hardware with software.

Thus, attempts to legislate distinctions between embedded and desktop computing applications based on the way software and/or hardware are implemented must fail because they attempt to deny the implications of the Turing Equivalence principle. While such distinctions might be useful in describing the way systems are implemented *in the absence of artificial influences*, they are merely the results of engineering tradeoffs. They do not indicate differences in terms of theory of computation. In fact, if there were an incentive to change implementations (such as protection from legal liability or expanded intellectual property rights), it would be a very simple matter to change one implementation to another that receives the more favorable treatment. Thus, any attempt to distinguish which software should receive favorable treatment that is based on implementation approach is doomed to failure from the outset.

3. The New Article 2 Factors

Proposed Article 2 Section 103(b) provides:

> (b) If a transaction includes goods and a copy of a computer program, the following rules apply:
>
> (1) This article applies to the goods.
>
> (2) If appropriate, this article, including provisions that by their terms are applicable to "goods," also applies to the product consisting of the goods and the copy considered as a whole. Factors that may be considered in deciding whether it is appropriate to apply this article to that product include whether acquiring the copy is incidental to acquiring the goods, the manner in which the copy is associated with the goods, the nature of and circumstances surrounding the transaction, and whether the copy is subject to a license.

Let us consider these factors in turn.

3.1 Acquisition is Incidental to Acquiring the Goods

How should the judge decide "whether acquiring the copy is incidental to acquiring the goods"?

The handling of a modern car is heavily influenced by the software that controls it. If you buy the car because you love how it feels when you drive it, you may be buying the car largely because of its software.

If you pay substantial amounts of money to get an internet-enabled kitchen, is the software incidental to the appliances? What about the Electrolux "Screenfridge" < http://www.electrolux.com/screenfridge/>? This beauty, reportedly due to hit the market this year, can recognize food that is past its expiration date, can suggest recipes, order food over the internet, *etc*. In these cases, the software is not incidental; it is central to the purchase.

A classic example of software being the central driving force behind purchases is the evolution of Japanese rice cooker appliances in the past decade. Old-style rice cookers had no software, and produced varying results depending on the ingredients used. Then fuzzy logic rice cookers were designed that used sophisticated control software to adjust cooking cycles to variations in water and rice. Japanese consumers turned out in droves to replace their old, perfectly usable, rice cookers with ones having this new software technology. This stimulated a flurry of "fuzzy logic" labels on products of all kinds, turning the type of control software used into the most important selling point on many consumer products. More recently, neuro-fuzzy logic rice cookers have appeared that use neural net software to dynamically adapt

fuzzy logic to produce even better rice. And many people have tossed out perfectly usable fuzzy logic cookers to upgrade to cookers with neuro-fuzzy software. Thus, the introduction of new control techniques (software) has been a major distinguishing feature in the rice cooker market for years.

Software can often replace functional hardware at a lower cost of goods. More capabilities become economically feasible as chips get cheaper. The use of software to replace hardware is accelerating with flash memory or equivalent solutions that look to the customer like non-erasable, read-only memory, but afford the manufacturer or a third party the ability to upload new software as needed to improve operation or correct product defects. Your personal computer has flash memory for the “BIOS” software. Your car probably has flash memory in its engine controller, as do DVD players and even some sewing machines. More capabilities become feasible as sophisticated operating systems (such as Windows) and the associated software design tools become available as inexpensive embedded software platforms. As software plays an increasing role in your kitchen appliances, your home heating system, your car, your television, and your bed, the capabilities of the software will play a significant role in your choice and perception of the product. Furthermore, a significant fraction of the engineering effort spent on developing new products is often spent on creating software, and this trend is accelerating (would manufacturers spend that much effort on a mere incidental aspect of a product?). As we understand the meaning of the word "incidental", over time, software will become less and less "incidental" to the sale of many traditional consumer goods. In fact, increasingly as in the case of rice cookers, the market for goods is fundamentally driven by innovations in software.

3.2 The Manner in which the Copy is Associated with the Goods

How should the judge interpret "the manner in which the copy is associated with the goods"?

The Article 2 draft comments suggest that a consideration is "whether its [the software's] range of functions is pre-set and cannot be modified during operation." Most off-the-shelf software products have a pre-set range of functions. It is difficult to modify most programs, especially off-the-shelf programs, while they are running. This doesn't make these programs embedded or suggest that they are like embedded programs. Some programs have options--functions that allow you to select how other functions will work--but these can be provided just as well to a person driving a car as to a person driving a word processor.

An obsolete view of embedded software is that it comes on Read Only Memory chips that can never be modified, just replaced. Some software does come this way, but increasingly embedded software is delivered and stored in a way that allows for updating of the software as necessary. The software might come on flash memory. Or it might come on disk and be uploaded to a device by the customer. Or it might come on ROM cartridges (as many computer games do) that can easily be swapped out by the customer. The choice among delivery/storage media is a classic example of engineering cost tradeoff. That choice should not determine whether we call something "embedded" or not.

3.3 The Nature of and Circumstances Surrounding the Transaction

How should the judge evaluate "the nature of and circumstances surrounding the transaction"?

The comments ask (a) whether the transaction occurs in the consumer market, (b) whether an alternative program is available that performs a similar function, and (c) whether it is available from a source other than the seller of the goods.

The consumer/non-consumer distinction has absolutely nothing to do with whether software is embedded. Large commercial aircraft, tanks, and nuclear reactors have a great deal of embedded software, are subject to Article 2, but are not consumer goods.

It is easy to provide customers with alternative versions of a program to control a device. This is a marketing choice, not an engineering choice. Customers can buy fuel efficient versus high performance versions of cars, and a key difference often lies in the software. Cars could just as well come with antilock brake and stability control options that use different algorithms depending on the experience and driving style of the driver.

The issue of availability from a third party also involves a marketing choice, not an engineering choice. If the law provides a strong enough incentive to manufacturers to allow others to develop and sell software that is compatible with their devices, those manufacturers will publish appropriate specifications and third parties will be able to create the software. Third party component suppliers already write much of the software in cars, so whether they sell it directly or via car manufacturers is simply an issue of distribution and marketing agreements, not technology. Independent third parties already sell aftermarket software to control many cars' fuel injectors. (*See*, for example, Turbo City Performance Headquarters, *Hey There Corvette Crossfire Owners!* <www.turbocity.com/CorvetteCrossfileECMUpgrade.htm>.)

Suppose that Customer A buys a stock Corvette. Customer B buys the same car but then buys fuel injector control software from Turbo City and installs it in her car. Customer B's fuel injector software has limited functionality and exists to control a device. To us, it looks somewhat like embedded software (whatever that is). But the transaction is independent of the car sale, and the software is licensed. So under proposed Article 2, this is probably not embedded software. But if the Turbo City software is not embedded, then why should the original software be considered embedded? The one is a direct replacement for the other, they serve the same functions, they control the same device, and they are for the same customer. If one is not embedded (and therefore escapes the scope of Article 2), the other must not be embedded either. But suppose Customer C buys a Cadillac and suppose that its fuel injector software is very similar to the Corvette's (perhaps even running on the very same computing hardware) but there is no third party replacement software on the market yet. Would this mean that Cadillac fuel injector software is embedded even though Corvette fuel injector software is not? What if Customer C's fuel injector software is then upgraded by the dealer to a newer software version that comes with a software license (and that is identical to preloaded software being shipped with newer vehicles)? What if an independent garage does the upgrade? Why should any of these circumstances have any relevance to whether software that runs a car's fuel injection is in or out of scope for Article 2?

The "nature and circumstances surrounding the transaction" factor will create enormous confusion because it sweeps into relevance facts that we think would otherwise be irrelevant in deciding whether software is embedded or not.

3.4 Whether the Copy is Subject to a License

Finally, the judge must determine whether the copy of the software was provided subject to a license.

This factor, at least, is easy. Either there is a valid licensing contract or there is not.

4. Evaluating Distinctions Between Embedded and Non-Embedded

There is no engineering basis for a principled distinction between embedded and non-embedded software. Software is software, no matter how it is stored. However, this issue has been argued so many times, in so many meetings of the Article 2B and Article 2 committees that we doubt that we are seeing the last attempt at a distinction in the February 2001 proposal.

This section suggests a few other approaches to evaluating any proposed distinction between embedded software (feel free to substitute the equivalent phrase du jour) and non-embedded software.

We have published two other detailed analyses of proposed distinctions. The first was in November 2000, when the Article 2 drafting committee floated a distinction between integrated (goods and software) and non-integrated products. (Phil Koopman & Cem Kaner, *Why the Proposed Article 2 Revisions Fall Short For Embedded Systems* at <www.badsoftware.com/embedd0.pdf>.) The second was our analysis of the UCITA distinctions in Part 1 of this paper (<www.badsoftware.com/embedd1.htm>.)

In Part 1, we suggested that you ask three questions of any proposal to distinguish embedded from non-embedded software:

(a) Does this distinction go to the heart of the difference between embedded and non-embedded software, or does it merely reflect a difference in how these types of software are (as far as you or the drafters know) commonly implemented today?

(b) What would it take for a manufacturer to redesign its product in a way that brings the embedded software under UCITA (or out of Article 2)? Are there examples already on the market that most people would consider to be embedded products that would fall under UCITA without such modifications?

(c) Suppose that a manufacturer made the least-cost design changes that bring its embedded software under UCITA (or out of Article 2). What are the expected impacts of the changes? For example, do we expect the resulting product to be safer? Easier to set up and use? Less likely to need repairs?

We respectfully suggest that the four-factor approach of the February 2001 draft would not fare well under these questions. The factors appear not to go to the heart of any differences between embedded and non-embedded software. The engineering and marketing changes needed to change an "embedded" product covered by Article 2 to one outside the Article 2 scope would be minimal and there is no reason to expect those changes would provide anything more than a legal benefit to the company, without a corresponding benefit to society.

Gail Hillebrand, of Consumers Union (letter to Lance Liebman, Director of the American Law Institute, November 30, 2000) also suggested several factors for evaluating a proposed distinction between embedded and non-embedded software (her letter called them integrated versus non-integrated products). We are rephrasing her questions as assertions:

(1) Software that operates features of consumer goods like cars, stereos, home appliances, and home medical devices should always be covered by Article 2.

(2) It should not be easy to "engineer around" the scope rule by changing the location or manner of delivery of programs contributing to the features of goods.

(3) Merely changing the storage medium of the software, such as by using flash memory, should not make a difference to the determination of whether software is embedded.

(4) Competing products in the marketplace should not be treated differently under the definition of embedded software. For example, would consumers of one brand of camera should not find their transactions fully under Article 2 while consumers of another brand of camera with similar features find their purchase partially excluded from Article 2 because of how the software is delivered to or used in the product.

(5) The current technological trends dealing with embedded software should not have the effect of excluding more and more goods from Article 2. Article 2's scope should not automatically and predictably be shrinking as more of the functionality of goods is implemented in software rather than in equivalent hardware.

(6) The application of the test for an embedded product, and thus for the scope of Article 2, should not depend on factual questions which will have to be litigated on a product-by-product basis.

(7) The characterization of a traditional consumer product as information rather than goods should not cause a change in the warranty or intellectual property rights of the parties.

We respectfully suggest that the four-factor approach of the February 2001 draft would not fare well under these questions either.

5. Conclusions

As technology advances, more functions of everyday products will be implemented in software. In our view, functional products that are sufficiently finished that they could be sold in a mass market should be considered as goods, whether they consist partially of software or not. To say otherwise is to say that eventually most things we use in everyday life will no longer be goods. We don't really believe anyone is trying to intentionally remove most everyday goods from coverage under UCC 2. But the problem is, nobody has been able to propose a way to distinguish different types of software that preserves the important principle of everyday goods staying entirely within the scope of UCC 2 as one would expect.

While several approaches to distinguishing between embedded and non-embedded software based on implementation approach have been proposed, none have withstood technical scrutiny. Nor are any such proposals likely to have a sound engineering basis, because fundamental principles of Computer Science hold that if there is an advantage to any particular implementation approach, it is possible to modify any software to use that approach, whether "embedded" or not. The current proposed wording for Article 2 has the (in our view undesirable) effect of encouraging manufacturers to skew their designs in order to take products out of Article 2 (courts would probably reason by analogy to the more vendor-friendly UCITA). Furthermore, the current proposed wording would result in an ever-increasing number of products being driven out of scope from Article 2 with the normal passage of time and the routine evolution of technology.

States that have adopted UCITA have made a different policy choice, but in states that have not adopted UCITA, we see no reason for the drafters of the Uniform Commercial Code to compromise longstanding protections for customers and competitors, nor the doctrines of exhaustion, fair use, and first sale. If something looks like a first sale, and the state has not adopted UCITA, the UCC should let it stay a first sale.

Mere technical implementation decisions should not be the primary basis for a powerful public policy change that will remove the functionality of whole classes of goods from UCC protection. The existing UCC should not be weakened by removing an ever-increasing number of everyday products from its scope. Copyright and patent laws already protect the industry against piracy--state laws need not provide these essentially federal protections, especially if they are unable to do so without undermining the applicability of the UCC to everyday goods.