571
2024/03

[Write a reusable Software - Capital Turnover Rate of SW]

Hisao Munakata

1: What is Embedded Computing?

In this issue, we would like to discuss what "Embedded Compute" is, what the meaning of prefixing the word "Embedded" is, what are the challenges of "Embedded" that we should pursue, and how we can win.

1.1 “Embedded” does not mean built in.

The word "embedded" may recall "relatively small-scale electronic devices" controlled by microcomputers. However, we will be the first to point out that this analogy is not necessarily correct. The essence of the word "embedded" is "purpose-built," and its opposite is "general-purpose," meaning a computer that can be used for a variety of purposes, like a personal computer, depending on which applications the user installs. Figure 1 summarizes the difference between "Embedded" and "PC. According to this classification, a smartphone, for example, might not be classified as an "embedded" computer.

	General purpose computer	Embedded computer
Example	PC, Smartphone	Home appliances Industrial equipment
Purpose	General purpose	Special purpose (specific)
Application	User can install	Pre-install (fixed)
manipulation	Mouse, keyboard, touch	Dedicated physical switch
Expandability	yes	no

Fig.1 Comparison of PC (=general purpose) and embedded (=dedicated)

So what are the characteristics of "Embedded Compute" used for specific applications? The first thing you will notice is the difference in HMI (=Human Machine Interaction), which is how the device operates. Unlike PCs and smartphones, which use keyboards and touch panels to operate devices, "Embedded" devices have dedicated switches for each function. In this regard, the recent automobile, which has a large touch panel to access various functions, may no longer be in the "Embedded" category. Of course, "Embedded" devices are required not only to have HMI but also to have the following characteristics.

(1) Reliability/availability (operate continuously for a long period, reset/reboot in the middle of the operation is unacceptable. Also, we do not want to use mechanical parts such as cooling fans, which are prone to failure, if possible)

(2) Power saving (in principle, non-stop operation; thus, power consumption management becomes critical).

(3) Network connectivity (LAN, public network), non-volatile data storage support

(4) Maintainability (remote failure diagnosis, easy parts replacement, and recently, remote SW update)

(5) Privacy/security (no leakage of personally identifiable data, no theft of money, etc.)

1.2 Large Embedded Systems

Next, I would show a real-world example where it is impossible to say that "Embedded adopts microprocessor inside. Let's look at ATMs in banks and POS systems used in stores to manage and count sales. Obviously, these are "dedicated computers," but do you know which hardware and software used for controlling ATMs and POS? Most ATMs/POS use Microsoft Windows as their foundation software, primarily running PC hardware inside, and of course, the control processor is a general-purpose processor from Intel/AMD. The Windows that controls the ATM/POS is virtually the same as the one we use on our PCs, which is why some ATMs stopped operation when Windows 7 support ended in January 2020. Referring to the transition of Microsoft's Windows for embedded devices, let's examine the boundary between Renesas' "Embedded Compute" and general-purpose PC processors.

1.3 Windows Embedded Typology

Figure 2 shows the evolution of embedded versions of Windows. It indicates that there were two "Windows Embedded" product lines in the past. I say "existed" in the past tense because Microsoft has already stopped offering "Tiny embedded" products that were started with "Windows CE." Now, Microsoft only provides a highly reliable version of "Windows 11 IoT Enterprise" for ATM/POS that uses PC-compatible hardware, essentially the same as Windows for PCs. The previous "Embedded" series also included application-specific versions with some success in-car navigation systems, but Linux and Android have already replaced these. Let us examine why Microsoft withdrew from the Embedded business, why it could not succeed, and what Renesas needs to grow in the Embedded market, which is our main battlefield.

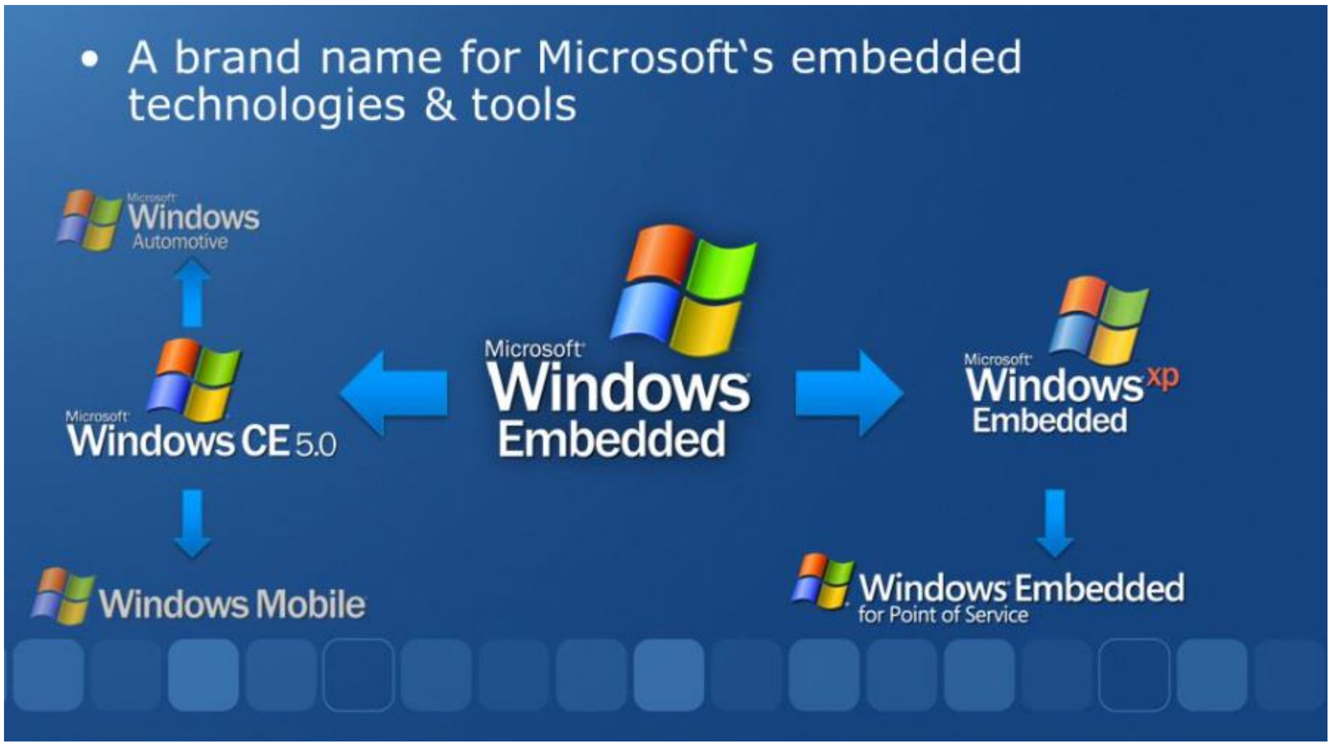
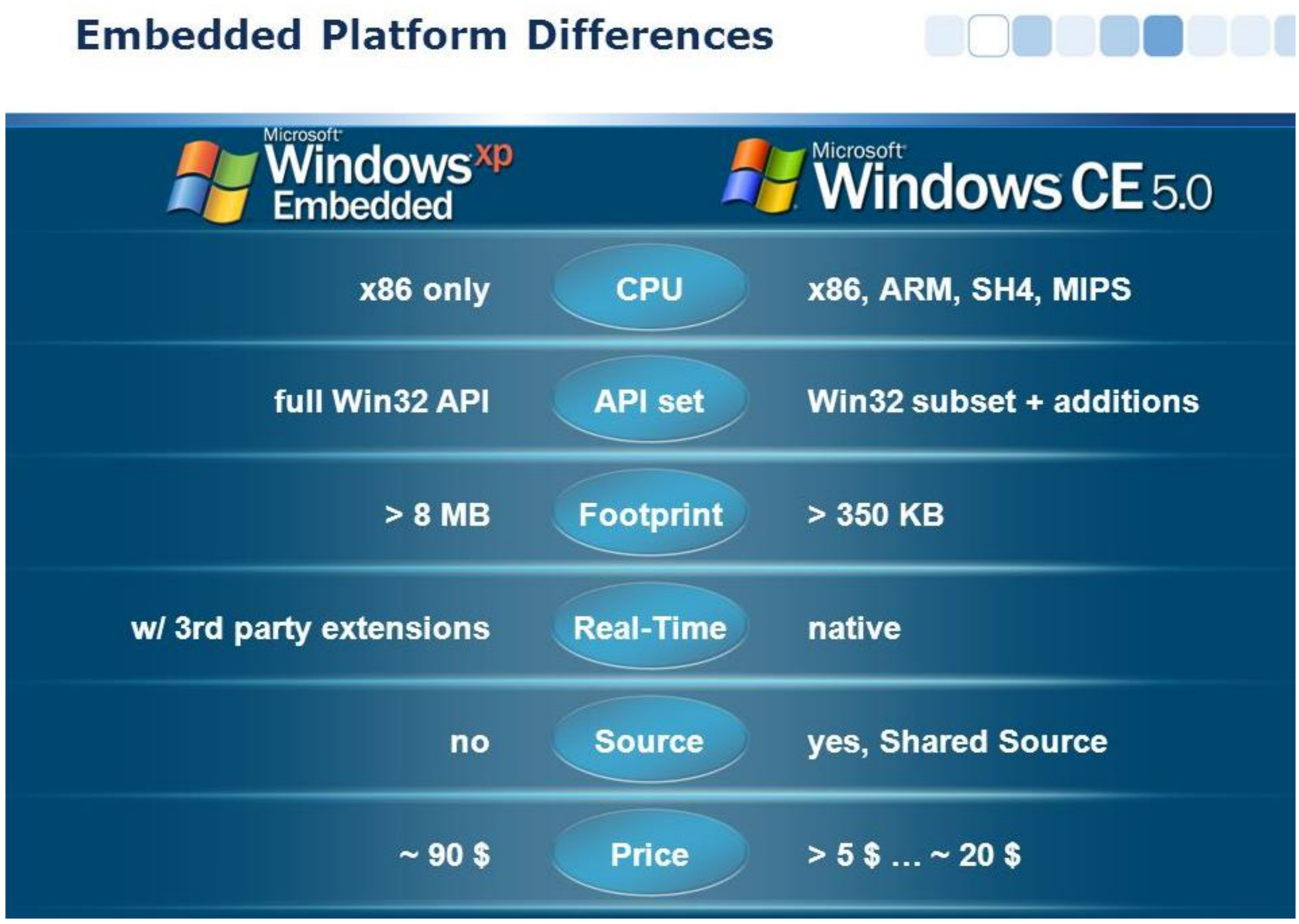

• A brand name for Microsoft's embedded technologies & tools
Microsoft Windows Automotive
Microsoft Windows CE 5.0
Windows Mobile
Microsoft Windows Embedded
Microsoft Windows xp Embedded
Windows Embedded for Point of Service

Fig.2 History of Windows Embedded product line



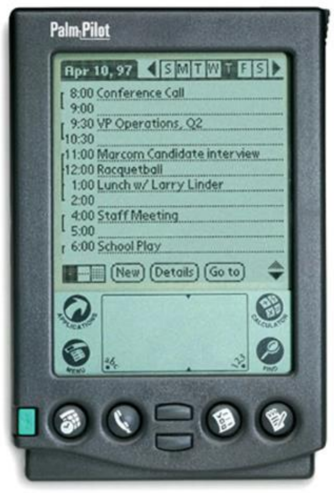
Embedded Platform Differences

Microsoft Windows XP Embedded		Microsoft Windows CE 5.0
x86 only	CPU	x86, ARM, SH4, MIPS
full Win32 API	API set	Win32 subset + additions
> 8 MB	Footprint	> 350 KB
w/ 3rd party extensions	Real-Time	native
no	Source	yes, Shared Source
~ 90 $	Price	> 5 $ ... ~ 20 $

Fig.3 The difference between Windows Xp Embedded (full featured) and WindowsCE (tiny subset)

1.4 A look back at the "NetBook" movement by the PC camp

WindowsCE (CE stands for Consumer Electronics), the starting point for Microsoft's "embedded" line, was announced in 1996 and used in "handheld PCs" such as Casio's Cassiopeia (with the Hitachi SH3 CPU), HP Jornada (SH3), NEC's Mobile Gear (NEC VR4121). The "handheld PC" was a new category of product that aimed to be a "simple PC" with more functionality than the small information terminals called PDAs (Personal Digital Assistants) that were popular at the time. Intel announced the Atom series of low-cost processors aimed at the "simple PC" proposed by Microsoft and combined them with the Windows CE series of operating systems to create a new category of consumer electronics (Windows CE), such as digital TVs and other consumer electronics. Intel is trying to enter consumer electronics such as digital TVs (Windows CE), smartphones (Windows Mobile), and automotive infotainment (Windows Automotive).



However, Intel's attempts were not successful. Around the same time, Microsoft announced a "netbook" based on the UMPC (=Ultra-Mobile PC) standard using Atom. “Netbooks" were positioned as a subset of Note PC, and Microsoft claimed that they could run full-featured browsers and business applications such as standard Excel. The rationale was that "netbooks" came with Windows XP for PCs. However, the performance gap between Atom and mainstream PC processors was so large that "netbooks" could not meet user needs, even when running business applications. As a result, all attempts by Intel, Microsoft, and other PC camps to enter the embedded space ended in failure. So why were there no such performance complaints with consumer electronics and game consoles at the time? That's because the software was designed to perform well on the microprocessors of the time (it didn't have redundant SW like WindowsCE), but why was it possible to do what Microsoft couldn't?

2. Characteristics and challenges of embedded

2.1 Optimizations that rely on implicit assumptions

Compared to a PC, "embedded" has the following limitations

1. Microprocessors have relatively low processing power and limited memory capacity.
2. Cost competition is severe because it is a dedicated machine, and it is impossible to provide performance margins or expandability.
3. Since the program is usually in ROM, it is difficult to rewrite it after shipment.

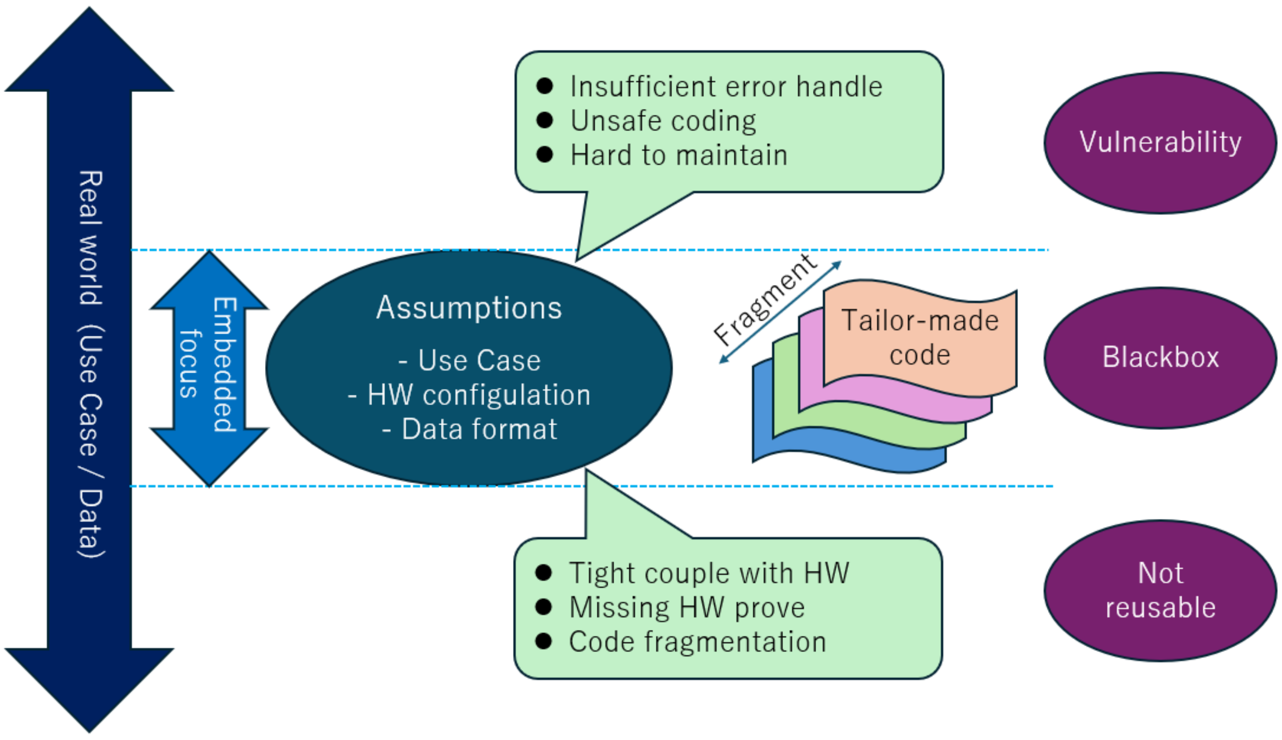
Except for ATMs, which run almost entirely PC inside, the computing resources available for "embedded" are extremely limited, so it is our destiny to develop software that can operate even in such a cramped environment. The premise of this kind of programming was that because it is an "embedded = dedicated machine," it is possible to specify (limit) use cases, and it is sufficient to develop programs that guarantee operation only for those requirements. If "data format," "event frequency," and "communication speed" are assumed to be within a specific range, the program can be simplified. If hardware requirements such as "number of input/output channels," "screen size," and "memory capacity" are fixed, it is possible to reduce functions. It is important to note that the design philosophy of Embedded is fundamentally different from that of Windows CE, which was designed on the assumption that a single program could support various applications and hardware.

2.2 Reusability

In general, embedded devices do not need to be programmed to detect (probe) how many channels of the I2C interface are available or how much memory is installed because the hardware is already determined. That is different from PCs that support Plug & Play, which automatically detects additional peripherals when they are added later. While "hardware specification" simplifies the program, it has the disadvantage that the software will not run if the hardware configuration changes, and as a result, most "embedded" software is not reusable “tailor-made” code. That was fine when the program was small, but the situation changed dramatically. In recent years, we have entered a downward spiral: "Software size increases ⇒ Developing from scratch is time-consuming and expensive, and quality does not improve ⇒ Product is not competitive ⇒ Securing development resources is a constraint, and business does not grow. To break out of this situation, reusable software development is also an urgent issue in the "embedded" field.

2.3 Vulnerability

Embedded" control software is not the only problem where "the program (i.e., the logic design) contains implicit dependencies on assumptions. In a "closed system" (i.e., a self-contained system with no external input), the problem is not apparent, but when devices are connected to a "network" or "memory cards and other storage devices are supported," several problems become apparent. One is simply a "malfunction" in which a program hangs and stops working; until about ten years ago, it was not uncommon for a program to suddenly freeze while working on a computer, rendering all previous work useless. In many cases, the program would hang as a result of inadequate "error handling" of unexpected data. More serious than these simple "malfunctions" are hacking attempts that deliberately cause the system to stop by providing unexpected input or modifying or stealing user data. The former is a "denial of service" (DoS) attack, in which a large number of connection requests are sent within a certain period to stop the system. The latter is a "buffer overflow" attack, in which data larger than the data receiving area provided by the program is sent to modify the website or run arbitrary programs. The latter is a buffer overflow attack that sends data larger than the data receiving area provided by the program to corrupt the website or run arbitrary programs. In recent years, attacks targeting such "logical vulnerabilities" (cyber-terrorism) have become a major social problem that can cause enormous economic losses. As a countermeasure, fuzz testing is widely recommended to verify that a system will not malfunction by intentionally providing non-standard (i.e., unexpected) inputs during program verification. For example, a program that is supposed to receive "video" input is tested by intentionally inputting "still image" data. In the past, people might have said, "It's the user's fault for entering non-standard data, and in such a case, it is inevitable that the system will malfunction," but such excuses are no longer valid.



2.4 Redefining Embedded

The basic philosophy of "embedded" software development, which has been regarded as the conventional wisdom of "first decide on the hardware, and then develop a minimal program that is applicable (only) to that specific use case," has inherent "reusability" and "vulnerability" problems. I also explained that these negatives have become a major problem in the recent trend of increasing program size. I want to point out that the definition of "embedded" itself is changing. Below is a list of system requirements that have become essential in the last few "embedded" years.

1 Connection to various networks

 1.1 Support for communication protocols such as LAN / Wi-Fi / BT / 5G Modem

 1.2 Support for authentication methods for various logins

 1.3 Computer virus countermeasures

 1.4 Support for SW Update

2 Data exchange with other devices (file system support)

 2.1 Support for SD cards/USB memory sticks and other recording media

 2.2 File exchange with Windows PC via network

3 Rich user interface support

 3.1 Graphical User Interface (Menu Screen Design)

 3.2 Support for smartphone-derived touch input, such as flick and swipe

The features listed here are "not supported" or "supported only to a limited extent" in traditional embedded systems. Still, these features have recently become key buying factors (KBFs) for users when selecting equipment. For this reason, Renesas' embedded solutions also emphasize state-of-the-art support for "connectivity" and "user interaction." It is important to note that the latest embedded system requirements are very similar to the concepts that Microsoft, Intel, and other PC companies tried (and failed) to achieve with the "NetBook" some 15 years ago. The modern "embedded" is more powerful, considering the 2007 "NetBook" was powered by a 1.6 GHz 32-bit Atom N270 processor.

I mentioned that the hardware performance of "embedded" in recent years surpassed that of "netbook" about 15 years ago, but the software development methodology is still the same. I want to ask you to consider the following three points.

1. Isn't fundamentally innovating the embedded software development scheme necessary?

2. There must be an opportunity for Renesas to lead this transformation and "provide new value to customers"?

3. Isn't it the cornerstone for Renesas to establish a leading position in his Embedded industry?

Microsoft tried to run Windows XP for PCs on the "NetBook" but was hopelessly defeated due to a lack of HW performance. However, we should consider that there were many suitable approaches to developing reusable software. Let's consider what indicators (benchmarks) we should use to innovate Embedded SW development.

3. Develop reusable SW

3.1 Let’s consider software as capital.

Embedded" software is typically developed from scratch for each product. software used in previous generations of products is rarely reused. As a result, it is easy to get into a situation where "only the programmer who wrote the code understands the control content," and when software modifications are needed to add features or fix bugs, no one but the programmer can respond. If it is still necessary to deal with the problem, the original program is black-boxed, and no one can touch it inside. Then, a "burn-and-erase" approach is taken, such as adding a redundant workaround that bypasses the original program, which is further transformed into a program that makes no sense. Software is generally considered an "asset". However, if the software is "stuck in a situation (see figure below) caused by a pile of code that no one can maintain," then it must be called "bad debt." many companies that develop "embedded = dedicated machines" are currently suffering from bloated "bad debts." Furthermore, we cannot ignore the fact that the software that has become such bad debt has become a "breeding ground of severe bugs. It is a well-known story that the recent large-scale system failures in Japan's financial industry were caused by bad debt software, and the situation is almost the same in the embedded world using microcomputers.

Stuck with "unmaintainable" bad assets that have become a mass of local patches and have diverged too far from Upstream

Furthermore, we cannot ignore that such software that has become bad debts has become a "hotbed of serious bugs. It is a well-known story that the recent large-scale system failures in Japan's financial industry have been caused by software that has become bad debts, and the situation is almost the same in the "embedded" world using microcomputers.

To overcome this situation, we need to

(1) Develop software from the beginning with the assumption that it will be reused in the future for other purposes or by different developers (eliminate assumptions about hardware and use cases).

(2) Before starting new development from scratch, investigate whether there are reusable assets from the past, and if so, use them to the maximum extent possible.

(3) If enhancements are made based on past assets, incorporate the changes back into the original code (to avoid code branching).

Fundamental changes in strategy are needed, such as the following. Reusable software is the seed for the next generation of products. While "assets" are the accumulation of past development results and are subject to preservation, software, the source of growth, must be considered "capital." The concept of "thinking of software as capital" is needed for embedded in the future.

1.2 Develop reusable SW

To realize "SW that can be reused as capital," it is necessary to consider (at least) the items listed below. This paper is not a technical explanation, so we will not go into the details of each item, but each topic requires proper technical understanding and continuous operation and can only be achieved after some time. However, only by practicing each of the above items one by one can the "capitalization" of "embedded" software be realized.

1. Elimination of all implicit assumptions
2. Abstraction of processing (modularization, encapsulation, objectification)
3. Unification of coding conventions
4. Preserve a complete change history (record who changed what, when, and why)
5. Maintenance of a common internal code repository (i.e., a searchable storage location)
6. Eliminate fragments (consolidate changes into common repositories (5) and do not increase SW variants)
7. Documentation (describe so that anyone can understand without knowledge of the subject matter)
8. Maintain SBOM (= SW Bill of Materials) and clarify the source of codes.

The methods used in open-source development help consider the development method of "reusable software" because all open-source software is "designed to be reused" from the beginning. Although the selection of specific tools may vary depending on the development project, the essential requirements for "SW capitalization" listed above are embodied in these tools, so you can obtain helpful information even if you aim to develop "reusable SW" within a closed scope, such as "only within your company" or "between your company and your customers."

1.3 Evaluate SW by capital turnover rate

Finally, let us consider an indicator to measure the degree to which the "development of reusable software" is realized. In financial analysis, an indicator called "capital turnover rate" is used, and its basic concept is "net sales ÷ total capital." Applying this to the purpose of this paper, "capitalization of SW," we can quantify the degree of "SW capitalization" by calculating the "total number of lines of product code ÷ number of lines of reused code." Various tools have already been developed to calculate the "similarity of source code and papers" to detect plagiarism of source code and papers. In the software industry, cost (=value) has traditionally been calculated based on "lines of code developed" or "man-months spent on development." In the software industry, cost (=value) has traditionally been calculated based on the "number of lines of code developed" or "number of man-months spent on development." In academia, the "number of papers cited" is used. Still, it would be interesting to establish a calculation method for the "SW Capital Turnover Rate" index to introduce a similar concept to embedded SW development and promote its performance to the industry.

In this paper, we discussed the need to fundamentally change the SW development scheme to become a champion in the "embedded" domain and shift from "SW development of a single product for each hardware target" to "SW development that can be reused." To achieve that, applying the latest large-scale SW development methods is necessary, referring to the good parts of the SW architecture that the PC industry, including Microsoft and Intel, tried to realize with the "NetBook." At that time, we introduced that it is beneficial the methods used in "open source" development, which are developed on the assumption that they will be reused from the beginning and can be utilized. I introduced my personal theory that SW used to be regarded as an "asset" to be preserved, but that from now on, SW can be considered as "capital to expand business," and its competitiveness can be increased by evaluating its turnover rate. Since it is difficult to become a champion in the embedded industry with hardware superiority alone, it is definitely necessary to continuously examine the advantage in the software domain.