Volume 07 Issue 01 Published, February 19, 2003 ISSN 1535-766X

Intel® Technology Journal

Managed Runtime Technologies

Runtime Abstractions in the Wireless and Handheld Space

A compiled version of all papers from this issue of the Intel Technology Journal can be found at:
http://developer.intel.com/technology/itj/index.htm

Runtime Abstractions in the Wireless and Handheld Space

Lynn Comp, Wireless Computing and Communications Group, Intel Corporation
Tim Dobbing, Wireless Computing and Communications Group, Intel Corporation

Index words: Wireless Handheld Devices, Mobile Information Devices, Java, J2ME, .NET, Managed Runtime Environments

ABSTRACT

The wireless handheld industry faces a number of challenges that managed runtime environments are uniquely positioned to solve. Unlike the Information Technology (IT) and personal computing industries in recent history, the handheld and wireless markets have experienced significant fragmentation in the associated application development environments. Developer support is referred to in the tens and hundreds of thousands of individual developers, rather than the millions referred to in the IT and personal computing industries. Another challenge that managed runtime environments help solve is providing the ability to connect to back-end infrastructure, whether in the carrier network or the enterprise. This connectivity is critical to device functionality in wireless. While runtime abstractions are intended to simplify the lives of the developers by providing the added protection and capabilities of Java* and .NET*, the abstraction can also present a challenge for users desiring optimal device performance, and component suppliers wishing to quickly enable advanced functionality on their latest products. This paper examines the tradeoffs of implementing an abstracted runtime environment in the wireless and handheld spaces, focusing on two of the most common managed runtime architectures, Java and .NET.

A BRIEF HISTORY OF THE CELLULAR INDUSTRY

The cellular industry (traditional wireless) is a relatively new industry having evolved over just the last two to three decades. In that time, the technology that came out of AT&T research labs, was first used by the military, then moved to the business community, and finally made available to the consumer at large. An important point regarding the cellular telephone, in regards to its relatively rapid adoption when compared to other ground-breaking technologies, is that the user experience with a cellular phone is nearly identical to the user experience with a business or home phone: the cellular phone looks and dials just like a landline phone, and it has, by and large, been limited to voice communication. Adding functionality to a traditional cellphone is accomplished by replacing the cellphone altogether – there is no provision for adding software or applications to this type of device. To support dynamic software additions/updates, the cellphone needed to integrate voice communication services with the system management and data communications services of the traditional computing industry while maintaining the integrity of both.

The portable version of the data computing industry, handheld computing, is also relatively new; it attempts to mimic must-have computing functionality in small, more portable devices capable of achieving better battery life. Entering the market in 1993 [1] the Apple Newton* was one of the first mainstream handheld computers, or as they were called at the time, Personal Digital Assistants (PDAs). Apple's hope was that the loyal developer community of Apple enthusiasts would also support the handheld version. Unfortunately, the Newton was unwieldy in functionality, size, and cost; it was too large for a shirt pocket or a pocketbook, too expensive to be an impulse purchase and unable to perform a few key basic tasks well. Due to these factors the Newton achieved a faithful, yet small following of users.

The Palm Pilot*, introduced in 1996 [2] was an invention that benefited a great deal from observing the market introduction and acceptance of the Newton. The tools to program the device were very inexpensive; the synchronization with the desktop PC was reliable and fast; the device was small enough to be carried in pockets or pocketbooks; and the price of $300 was just slightly higher than an impulse purchase for the more technically

* Other brands and names are the property of their respective owners.

savvy consumer. The Palm Pilot ultimately attracted as many as 60,000 developers to its platform and unique API set. In the UK, a company called Psion made headway with a keyboard-based device with similar functionality to the Palm and Newton. This success resulted in a third set of unique APIs for a software vendor to write to, test, and support. By the time Microsoft entered the market in 1997 with the WindowsCE* platform, the independent software vendors had to write, test, and support no fewer than four different platforms to support the portable market at large. Given the low-cost nature of the handheld market in general, it was tough to support all the platforms and still have a positive return on development investment as an Independent Software Vendor (ISV). The highest volume market, the cellular telephone, allowed no extensibility once the platform was on the market, and the portable but extensible platforms were fragmented to the point where an ISV faced a potentially negative return on investment on his or her development efforts.

THE GREAT UNIFIER

When Sun Microsystems introduced Java* in 1995 with the tagline "write once, run anywhere," it was unclear exactly what Java was intended to do. Editors and analysts who studied and researched the computing and software industries wrestled with how to categorize Java: Was it intended to replace the operating system, the CPU instruction set, or create network computers[1], or was it meant to provide an alternative to Microsoft desktop operating systems? Whatever category that Java was placed in at that time, it was too large and unwieldy, too proprietary and too slow for small, memory-constrained devices (although that didn't stop early visionaries such as the Nortel Orbiter [3] phone development team from attempting to create an advanced cellular phone running Java). Java gained its early strength and acceptance in the enterprise infrastructure, simplifying and unifying multiple types of Information Technology (IT) and Internet systems through a higher layer of abstraction.

Java in Wireless

A number of developments spurred the acceptance of Java into wireless client devices ("client" refers to a device that must attach to a network or other infrastructure to accomplish specific mission-critical tasks). The first key development was Sun's introduction of a small-Java solution and virtual machine for memory-constrained devices in 1999 [4]. The inherent protection from memory overruns provided in the Java architecture itself already made Java attractive, since wireless operators are highly sensitive to handsets suspending operations without warning while a customer is operating the handset, or being capable of proliferating network-damaging viruses. Non-functional handsets increase operator costs, and a healthy on-line network represents the only revenue source for the wireless operator.

A second reason for the wireless operators having a favorable response to Java is the inherent memory efficiency of bytecodes versus native code [3, 6]. Operators have extremely high investment costs in preparing to provide wireless services. The first cost to an operator is paying national governments extremely high license fees for rights to a limited amount of wireless spectrum in a specific geographical area. The operators are limited to servicing wireless data traffic within that allocated spectrum. The more customers that an operator can support within the limited amount of spectrum purchased without customers experiencing dropped calls or being unable to connect to the network, the faster the operator is able to recoup cost and earn positive cash flow.

The final factor in Java's favor is that operators are familiar with consortium-based standardization processes through years of participation in the International Telecommunication Union (ITU) and are more comfortable with standardization than accepting whatever an external vendor develops in binary form, with no ability to influence or adapt it. The establishment of the Java Community Process further enabled the acceptance of Java by wireless operators. Not only was Java a relatively open standard, it attracted a great number of software developers in the enterprise space due to its ability to provide a cohesive layer over whatever system existed underneath it. A tenth of the three million developers supporting Java in 2002 represents quadruple the number of developers supporting the most popular handheld device of the 90s, the Palm Pilot*.

Java in the Wireless Client Device–Devices Now vs. Devices of the Future

Java technology for the wireless client device is in a state of transition. Currently, Java wireless client devices are based on the Java 2 Platform, Micro Edition (J2ME) and specifically on the Connected Limited Device Configuration (CLDC) version 1.0 and the Mobile Information Device Profile (MIDP) version 1.0. The specifications for CLDC 1.0 and MIDP 1.0, JSR-030 and JSR-037 respectively, were released in 2000 at a time

* Other brands and names are the property of their respective owners.

[1] Network computers are thin clients in the network architecture where the "network is the computer." In this model server computers provide the intelligence.

when memory capacity and processor power were limited, and as such, compromises were made in terms of the level of support for the Java Language Specification and the Java Virtual Machine Specification. In addition, new libraries (e.g., user interface, networking, and persistence libraries) were developed to support these resource-constrained devices, reducing the consistency between standard and wireless Java.

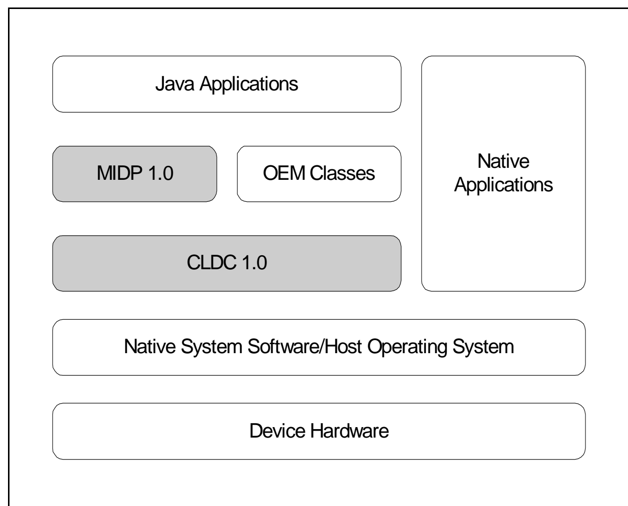

Figure 1: CLDC 1.0/MIDP 1.0 architecture

The high-level architecture of these CLDC 1.0/MIDP 1.0 devices is shown in Figure 1 above. One of the goals of the J2ME architecture was to provide a highly portable, secure, small footprint application development environment for resource-constrained connected devices [8], and it has been largely successful.

The CLDC 1.0 provides the platform that is intended to serve as the lowest common denominator for all such devices (e.g., mobile phones, pager, and point-of-sale terminals) while the MIDP 1.0 addresses the needs of a specific vertical market (e.g., mobile phones).

CLDC 1.0 addresses the following areas: support for the Java language and virtual machine features, core libraries (e.g., java.lang, java.io, java.util), input/output, networking, security, and internationalization. MIDP 1.0 addresses the following areas: application models, user interfaces, persistent storage, networking, and timers.

In the case of CLDC 1.0, a number of sacrifices were made in terms of the support for the Java Language Specification and the Java Virtual Machine Specification. Specifically, there is no floating-point support, there is no support for finalization, and there is limited exception handling support. In addition, there is no support for the Java Native Interface[2] (JNI), no support for user-defined class loaders, no reflection, no support for thread groups or daemon threads, and no support for weak references. The lack of support for reflection also means that there is no support for language or virtual machine features that rely on reflection (e.g., RMI, object serialization, JVM debugging, and profiling). From a programming perspective and also in terms of being able to port Java 2 Standard Edition (J2SE) applications to the J2ME platform, these are serious issues that have led to fragmentation of the J2ME architecture.

In terms of library support, the CLDC adopted subsets of most of the corresponding J2SE libraries. The exception to this was the specification of a new networking library, the Generic Connection Framework.

While CLDC 1.0 addressed the needs of the horizontal market, MIDP 1.0 addressed the needs of a specific vertical market (e.g., mobile phones). Specifically it covered the following areas: user interface support, networking support (i.e., HTTP), persistent storage, application models, and timers. In doing so, MIDP 1.0 diverged significantly from the J2SE. New user interface and persistent storage libraries were specified, and the networking libraries were based on extensions of the CLDC 1.0 Generic Connection Framework.

In terms of achieving the goals that the J2ME expert groups set for themselves in specifying the J2ME architecture, the results have been mixed. Certainly the goal of a small footprint has been achieved. However, there has been a cost in terms of application portability and security. The approach consistently taken by the expert groups has been one of specifying the lowest common denominator in terms of functionality for both CLDC 1.0 and MIDP 1.0, which has led to a large degree of fragmentation of the architecture. Specifically, proprietary Original Equipment Manufacturer (OEM) classes have been used by individual phone and PDA OEMs to make up for the lack of functionality in the CLDC and MIDP specifications. A number of companies, including Sprint PCS, Motorola, and Nokia, have added their own unique device-specific functionality (e.g., sound, additional network protocols, additional user interface functionality) in the OEM classes, and in doing so sacrificed application portability for enhanced functionality.

Portability has not only been sacrificed between J2ME CLDC/MIDP1.0 devices but also between the J2ME (i.e.,

[2] Java methods are able to invoke native methods through proprietary means.

wireless) and J2SE* (i.e., desktop) clients. In addition to the inconsistencies between user interfaces and networking libraries, the CLDC/MIDP 1.0 client device does not align with the J2SE security model.

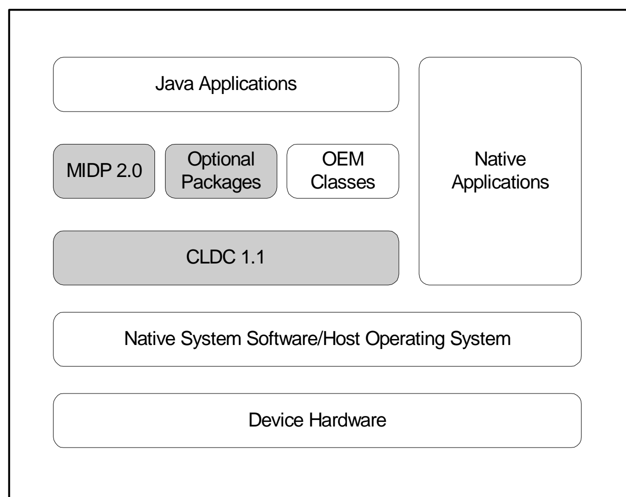

Figure 2: CLDC 1.1/MIDP 2.0 architecture

MIDP 2.0 and CLDC 1.1 (JSR-118 and JSR-139, respectively) address many of the concerns of MIDP 1.0 and CLDC 1.0. CLDC 1.1 has added back support for floating point as well as support for weak references. MIDP 2.0 provides significant improvements over MIDP 1.0. It provides enhanced networking support, enhanced user interface support, support for gaming, support for sound, and a security model that better aligns with the J2SE model of security.

The architecture model for a CLDC 1.1/MIDP 2.0 device is shown in Figure 2 above. Aside from the inclusion of CLDC 1.1 and MIDP 2.0, the major enhancement is the support for optional packages. Optional packages are intended to supplement the functionality provided by CLDC 1.1 and MIDP 2.0 and eliminate the need for the use of OEM classes. Examples of optional packages include JSR-120 (Wireless Messaging API), JSR-135 (Mobile Media API), JSR-82 (Bluetooth), JSR-80 (USB), JSR-177 (Security and Trust APIs), JSR-172 (Web Services) and more. The Java Specification that ties all of these together is JSR-185–Java Technology for the Wireless Industry. This Java Specification Review (JSR), due out in 2003, will specify the mandatory JSRs that must be supported by wireless client devices along with a list of optional JSRs that may be supported.

Java is predicted to be the dominant technology for wireless client devices through to 2007 [9]. It is expected that, on average, 50% or more of the applications running on wireless client devices will be Java applications. A breakdown of the predicted amount of data traffic generated by Java applications (as a percentage of all technologies) by type of application is shown in the following table.

	2002	2003	2004	2005	2006	2007
Messaging	2%	6%	13%	26%	44%	67%
m-Commerce	7%	16%	29%	50%	65%	77%
Content	<1%	12%	26%	38%	51%	63%
Information	<1%	<1%	2%	4%	8%	15%
Location-based Services	3%	8%	16%	29%	45%	64%
Industry Applications	6%	10%	18%	36%	56%	85%
Intranet	2%	5%	14%	30%	50%	73%
Total	3%	9%	18%	31%	46%	62%

Table 1: Data traffic generated by Java applications

In terms of numbers, it is predicted that there will be 691.6 million Java-enabled phones in the marketplace by 2007 out of a total 727.3 million handsets (95% of the market). This is a significant number and could increase as alignment improves between the J2ME platform and the J2SE platform. This improvement in alignment is due to the potential for improved application portability across a wider range of devices, including the desktop, mobile phones, and PDAs.

OTHER MANAGED RUNTIME ENVIRONMENTS

In 2000, Microsoft introduced .NET*, an architecture intended to accomplish many of the same tasks as Java*. According to Microsoft, .NET intends to provide more coherence and less fragmentation across device types for application developers, providing a true "write once, run anywhere" experience on Microsoft-based platforms. What makes .NET attractive is the fact that it does allow a developer to write an application once, in the most commonly used Microsoft Visual Studio tools, and deploy it across a variety of Microsoft-based platforms. Similar to the deployment of Java, the larger, enterprise-focused .NET was available prior to the smaller footprint Compact Framework (CF) .NET, which began to roll out gradually in the 2002 releases of Microsoft Windows CE*.

* Other brands and names are the property of their respective owners.

* Other brands and names are the property of their respective owners.

.NET has a number of differences when compared to other runtime environments including Java. First, .NET supports a number of different languages including C, C++, C#, Visual Basic, and JavaScript. Using Microsoft's Visual Studio .NET, programs written in these languages are compiled into a common intermediate language representation that executes within the Common Language Runtime Environment (CLR). Some of the benefits to this approach are that existing developer skills (such as programming language experience) and existing software collateral (such as applications) may be reused with minimal effort. In other words, it's not necessary to learn C# nor to completely rewrite applications (in C#) in order to support .NET architectures.

Microsoft also claims that their driver model is more flexible in that it avoids the need for proprietary extensions and improves application portability provided the same operating system is present across all devices. Drivers in the application expose underlying hardware differences, and the device only makes use of the underlying features if it can. Applications do not need to be recoded for each device [9].

In 2001 and 2002, Microsoft signed a number of agreements with a variety of wireless network operators such as Deutsche-Telecom, indicating that the wireless industry views .NET as a possible alternative to Java. However, the main question with .NET is not if, but when, the average user will be able to utilize .NET applications and services on wireless handheld devices.

Technical Tradeoffs in a Difficult Software Environment

Even when using the smaller footprint Java or CF .NET in the wireless and handheld spaces, successfully running these managed runtime environments is a tightrope act between good enough performance and the portability of the device. Unlike in the desktop and server space, it is an ineffective use of memory to compile every bytecode in an application to native code and optimize the code during subsequent runtime iterations to maximize performance. The majority of phones on the market currently have 8MB of ROM/RAM, feature phones have up to 40MB, and only the most feature-rich segment of the market reaches 64MB. Stored bytecodes use memory far more efficiently than compiled code. Relying solely upon a Just in Time (JIT) compiler that assumes it has unlimited free space available to it, negates one of the reasons the operators selected Java for an applications framework in the first place.

What might seem to be an obvious solution to the memory and performance constraints, adding an execution unit dedicated to executing only bytecodes, turns out to have its own shortcomings. Bytecode translators are actually less efficient overall, because they cannot use compilation to improve execution efficiency for a CPU pipeline architecture or a given software program code flow. Bytecode translators also cannot use the general-purpose register set available to native execution, and because the protocol stacks for a phone are written in compiled C, the processor must attempt to execute both the native processor instruction set as well as the bytecodes. Even after adding the bytecode translation units to RISC processors, there are still a number of bytecode instructions that must be emulated vs. executed directly. This is due to the higher level of abstraction inherent in Java: a single bytecode operation often cannot map directly to a single RISC CPU instruction since the operations required by some bytecode operations are, in reality, a combination of RISC CPU instructions. To have a direct 1:1 mapping between bytecode operations and CPU instructions requires significant enough changes and additions that the benefits of adopting a RISC architecture because of lower power consumption and reduced complexity could be completely lost.

What enabled Java to become small enough to run on a phone is also a tradeoff between integration and functionality with standard systems running Java in carrier or enterprise infrastructures. In order to downsize Java to fit into an unfriendly climate like the wireless handset, Sun made a number of tradeoffs, essentially cutting Java to the bare essence. So Java developers from the J2SE or J2EE world find themselves without any of the standardized graphics (AWT/Swing) and without the security libraries in the J2ME CLDC/MIDP 1.0 world. It is only slightly better in the J2ME CDC world: the graphics libraries are added, but they are not the most recent version adopted into J2SE (e.g., AWT vs. Swing).

Currently the CF.NET situation has improved coherence, but is limited to the high-end segment of the phone market. While Java can be fit into all but the least expensive of the handsets, CF.NET awaits the inevitable increase in capability and lowering of cost for memory and processing components (sometimes referred to as "Moore's Law") to catch up to the memory and performance requirements necessary to adequately support it.

THE INTEL APPROACH TO MANAGED RUNTIME ENVIRONMENTS

The Intel approach to the challenges presented in the wireless runtime world is to achieve the most optimal balance between speed, memory use, and power efficiency and to focus on decreasing the fragmentation of functionality in the wireless space. While Intel believes

Java* and CF.NET* are key technologies in wireless for many of the reasons mentioned above, the power of these managed runtime environments is in using them as a unifying framework to simplify application development experience. "Write once, run anywhere" should not be applied without accounting for the challenges specific to running in a handheld wireless device (e.g., intermittent, low-bandwidth connections).

The first underlying principle in Intel's wireless managed runtime approach is to focus on advanced, low-power, high-performance and scalable processors such as the Intel® XScale™ microarchitecture core. Rather than adding bytecode translation, which has its own problems, the focus of Intel's efforts is on providing a processor capable of scaling a range of devices running Java, CF.NET, and native code efficiently. For the simplest phones running basic Java, components utilizing the Intel XScale microarchitecture running directly out of execution-capable flash memory is expected to be an appropriate balance of performance and functionality for a phone that is essentially free with the purchase of a network contract. By running directly from the flash memory, execution speed is increased and memory is saved. In this situation, the interpretation process does not copy into RAM in order to run. The program executes directly and immediately from its original place in the device memory.

The mid-range of functionality adds a self-limiting Just in Time (JIT) compiler. Other descriptions include "dynamic, adaptive" or "small footprint" JIT. The primary difference between the small footprint JIT and the larger JIT compilers seen in the PC world is the fact that the system integrator is able to determine how much memory the JIT may use within that specific piece of equipment. The smaller footprint JIT compilers limit themselves to a specific area in memory for "scratch pad" space during the process that compiles Java bytecodes into native code. The power of a JIT compiler is that by and large, compilation is more efficient at executing code over the lifetime of the program execution as a whole. The goal is to determine on an iterative basis how to best optimize the code that executes the most often: where the branches are taken, what branches are not taken, which variables to store and which are transient, and how code blocks can be rearranged to execute most efficiently on a given CPU architecture. The difference is that the JIT compiler process happens on the fly, and in the case of the wireless handheld devices, it happens within a limited space in memory. In this scenario, the easiest way to maximize the available memory to the JIT compiler is to execute the JIT from flash memory, leaving the available RAM space for the results of the compilation process. Eliminating the need to copy the compiler into shadow RAM saves both time and memory space for the operation of the JIT compiler.

Another related process to the JIT compiler that increases performance is to pre-compile the libraries required by the Java profile in use to native code. The goal is to selectively compile the most performance-critical libraries used by applications popular in the wireless space to balance the speed of native code against the memory compactness of Java bytecodes. In wireless, the most popular Java applications currently are entertainment, which are game and graphics focused. Hence, the libraries supporting the user interface and the multimedia operations (optional in some profiles but highly recommended by Intel for an improved user experience) should be the first order of business for system implementers. The primary difference between a feature phone and the higher end smartphones and PDAs is the memory dedicated to improved JIT compiler performance as well as the ability to pre-compile all the existing libraries to native code vs. selecting a few key areas to focus on for improving performance.

FRAGMENTATION AND RE-UNIFICATION

The specification of CLDC 1.0 and MIDP 1.0 were constrained by the limited amount of processor power and memory. However, recent advances in hardware technology have all but eliminated these constraints and it is not uncommon to find devices with processor speeds of 200-400MHz and total memory budgets of 40MB. This is a far cry from the 328KB of memory allotted for CLDC 1.0/MIDP 1.0.

The removal of the constraints that drove the specification of CLDC 1.0 and MIDP 1.0 means that these devices may easily support the Connected Device Configuration (CDC). CDC is the big brother of CLDC, and together with the Foundation Profile (FP), provides the same platform functionality as the J2SE version 1.3.

The J2ME architecture of a CDC-based device is shown in Figure 3 below. Similar to CLDC, CDC provides the common platform services while the FP and Personal Profile (PP) address the needs of a particular vertical market.

---

* Other brands and names are the property of their respective owners.

Intel® XScale™ is a trademark of Intel Corporation or its subsidiaries in the United States and other countries.

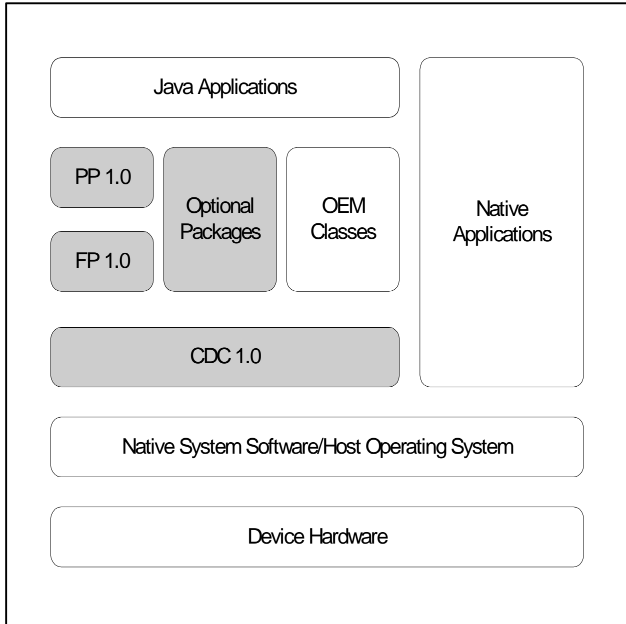

Figure 3: CDC/FP/PP architecture

The PP is similar in scope to MIDP in that it provides user interface libraries, networking libraries, and the application model. Unlike MIDP, the PP is compatible with the J2SE* (e.g., the PP supports AWT).

Many of the optional packages that have been developed to date are reusable across both CLDC and CDC platforms (e.g., JSR-120 [Wireless Messaging API], JSR-135 [Mobile Media API], etc.).

By moving towards a CDC-based J2ME architecture, many of the fragmentation and portability issues associated with CLDC/MIDP can be addressed and a closer alignment with the J2SE achieved. This will facilitate the development of common client devices that may be run on both the desktop and wireless client devices.

Figure 4, below, shows how the transition from today's CLDC/MIDP-based J2ME architecture to tomorrow's CDC/FP/PP-based J2ME architecture might be accomplished. On the left-hand side is the situation today: a CDC-based software stack and a CLDC-based software stack. Moving to the right, CDC is adopted as the configuration platform for MIDP with the benefit that MIDP and MIDP applications now enjoy full Java Language and Java Virtual Machine support. The next phase in the migration supports a transition period between MIDP and PP. It is during this time that MIDP applications are ported to the PP. The final stage on the right is the result of the migration: a single application development and runtime environment based on CDC/FP/PP. The CLDC/MIDP stack is no longer required nor should it be used.

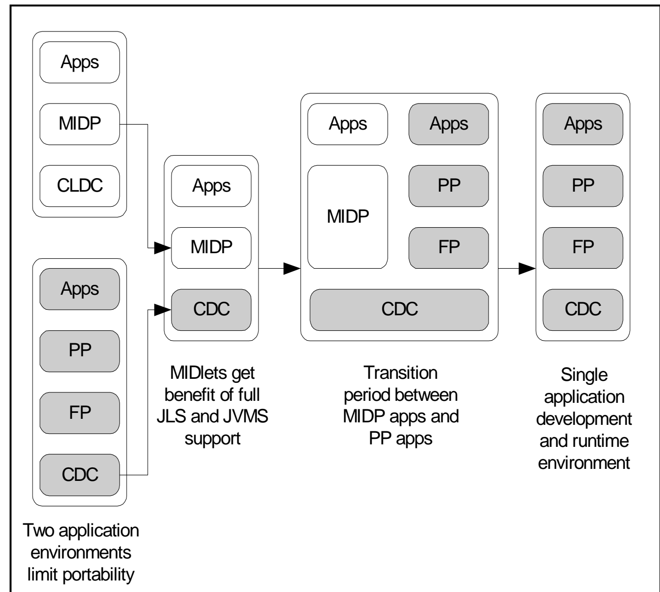

Figure 4: CLDC/MIDP evolution strategy

The proposed CDC-based platform that forms the basis of the Intel PCA Java architecture is shown in Figure 5 below. Along with CDC, FP, and the PP, it also shows the optional packages, organized by application type, that are required to support the predicted growth of Java* applications for wireless client devices[3].

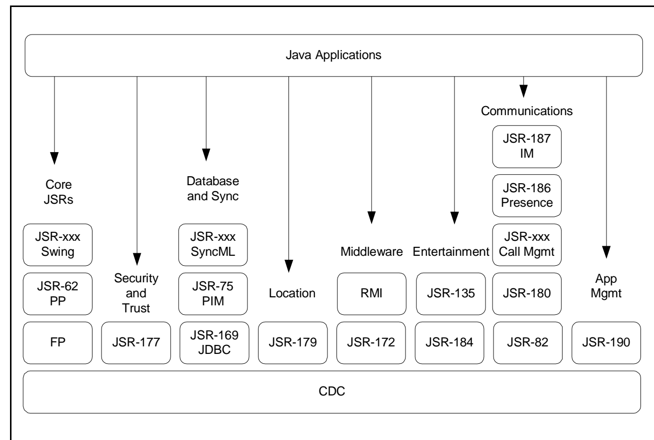

Figure 5: Proposed CDC/FP/PP platform

* Other brands and names are the property of their respective owners.

* Other brands and names are the property of their respective owners.

[3] Those JSRs that do not have numbers associated with them do not yet exist.

CONCLUSION

The handheld and wireless markets have their own unique challenges and opportunities. An emphasis on the ability to connect to back-end infrastructure, the need for highly compressed and efficient data transmission and highly constrained screen and battery all shift the dynamics of the traditional development environment.

Runtime abstraction solves many problems presented to the average application developer, and it presents many challenges to the hardware implementation teams in adapting to the unique execution environment. The Intel philosophy on balancing application developers' desires for simplicity and coherence against the hardware implementers' desire for low memory and high performance is to appropriately select and tune key components in the system. A high-performance yet low-power general-purpose processor combined with execution-capable flash memory and selectively tuned native components provides the basis for a wide range of wireless client devices.

JSR-185 (Java* Technology for the Wireless Industry) is slated to be completed in 2003, and it will address some of the fragmentation concerns with respect to J2ME. It will provide an architectural overview of the essential client components of an end-to-end wireless solution including recommended combinations of J2ME components (i.e., configurations, profiles, and optional packages). The availability of this architectural specification will also be used to trigger compatibility requirements that, in turn, will be reflected in the associated Technology Compatibility Kit (TCK) that is used to determine conformance.

At the same time, .NET* will also be evolving and spreading throughout the wireless ecosystem as Microsoft-based platforms are deployed more widely. While .NET-based platforms do not face as many integration challenges due to the fact that the number of variables decrease, the rollout of .NET devices is just beginning to ramp.

Regardless of the managed runtime environment an application developer selects, Intel technologies create a balanced and flexible platform upon which the hardware implementer and application developer have freedom to innovate and differentiate their solutions and products. As the platform costs decrease and the available performance and memory increase, the user experience improves. It is expected that the wireless client device will become a significant force in the computing industry at large.

REFERENCES

[1] http://www.wired.com/news/mac/0,2125,54580,00.html, Apple's Newton Just Won't Drop, Leander Kahney, Aug. 29, 2002 PT.

[2] http://www.fortune.com/fortune/fsb/specials/innovators/dubinsky.html , "HOW WE GOT STARTED," Donna Dubinsky.

[3] http://www.microjava.com/articles/perspective/shosteck?content_id=2179, "The Battle For BREW, J2ME And Related Technologies," The Shosteck Group, 10/29/2001.

[4] http://java.sun.com/features/2000/06/time-line.html, "The Java Platform: Five Years in Review."

[5] http://more.abcnews.go.com/sections/business/dailynews/silicon_insights_seybold_010716.html#1, "Spectral Efficiency, Which technologies work best?," Andy Seybold, July 2002.

[6] http://www.byte.com/documents/s=693/byt19990811s0006/index.htm, August 1999.

[7] http://www.pctel.com/cellular_problem.php, other references available upon request.

[8] Roger Riggs et. al., "Programming Wireless Devices with the Java 2 Platform," Micro Edition.

[9] ARC Group, "Wireless Java 2002 Handset and Application Revenue Streams."

AUTHORS' BIOGRAPHIES

Lynn Comp is a strategic marketing engineer in the Wireless Computing and Communications Group at Intel Corporation and has been active in the portable and wireless computing market for the last six years, setting strategy for WCCG in runtime environments and software platforms. Prior to her involvement in the wireless and portables' market, Lynn was an applications engineer on data communications silicon supporting customers developing routers, cellular basestations, and WAN/LAN bridges. Lynn has a B.S.E.E. degree from Virginia Tech and an MBA degree in Technology Management from the University of Phoenix. Her e-mail is lynn.a.comp@intel.com

Tim Dobbing has more than 15 years of software architecture, design, and implementation experience in the telecommunications industry in the areas of network management, high availability, traffic management, and call processing for a variety of technologies including CDMA, ATM and ISDN. In his current position as a J2ME Architect at Intel Corporation, his focus is on the specification of a J2ME reference architecture for Intel's Personal Internet Client Architecture (PCA). Other Java

experience includes the architecture, design, and implementation of a Web-based Management solution for a CDMA Base Station Subsystem using J2SE and PersonalJava. Tim received a B.E. and an M.E. degree in Electrical Engineering in 1984 and 1990, respectively from Carleton University in Ottawa, Canada, and he is a member of the Association of Professional Engineers, Geophysicists, and Geologists of Alberta (APEGGA). His e-mail is timothy.c.dobbing@intel.com

Copyright © Intel Corporation 2003. This publication was downloaded from http://developer.intel.com/.

Legal notices at http://www.intel.com/sites/corporate/tradmarx.htm.

For further information visit:

developer.intel.com/technology/itj/index.htm

Copyright © 2002, Intel Corporation. All rights reserved.
Intel is a trademark or registered trademark of Intel Corporation or its subsidiaries in the United States and other countries.
For a complete listing of trademark information vistit: www.intel.com/sites/corporate/tradmarx.htm