JetZ 

Contents:

---

1. Introduction

This document gives a short overview on what the **JetZ, the Java embedded Technology -- Zurich** project is all about. The first result of this project, the IBM JavaCard, is highlighted in some more detail. This way, the basic foundation, i.e., cutting-edge Java VM technology, and very concrete implementation benefits of our approach are presented.

---

2. The JetZ research project

As already mentioned above, JetZ stands for *Java embedded Technology -- Zurich*.
This very much captures, what the JetZ project is about: We are developing Java technolgy for resource-constrained embedded devices. This entails very efficient Java Virtual Machines, low-level device drivers suitable for Java on embedded platforms, and the required system libraries on the respective hardware. The JetZ project follows good engineering practices in that the developed system is highly portable, scalable, and efficient. These claims are substantiated below.

Portability

The software we have developed is portable in the sense, that it is easy to adapt to different hardware platforms. This has been achieved by writing all device-independent parts in a very general C dialect understood by almost every C compiler. Only the most device-dependent parts of any particular JetZ engine are written in Assembler.
This goal has been constantly adhered to during development by ensuring that the same code base can be built at any time for 8-bit processors, and 32-bit processors (big endian and little endian alike). The net result is a runtime environment that is *guaranteed* to behave identical on the different platforms. Code once tested on a PC under Windows 95/98/NT/2000 or Linux works identically on the JetZ VM on the 8-bit SmartCard CPU.

Scalability

Contrary to most embedded Java development approaches, JetZ did **not** begin by trimming down a regular, JDK-type Virtual Machine. Instead, we chose to approach the implementation solely based on the specifications of the Java language and runtime published by Sun Microsystems. This has two major benefits: One is that no overhead from old Virtual Machine technology found its way into the JetZ system. Only the parts of Java that are essentially required on a particular platform are present. If no threading is necessary, not even the slightest data structure or code only necessary to support threading is present in a JetZ built. The same holds true for example for Garbage Collection, or the range of supported data types.
The second advantage gained by not basing our implementation on JDK-type code is that this technology is free of any copyright claims by outside parties, i.e., it constitutes a *clean-room implementation* in the legal sense of the word. Practically, this means that we do not have to pay license fees to Sun Microsystems for any system using this software.

Performance

Of course, everybody tends to claim that the own system is the most highly performing one available. In this section, we will only give a basic rationale for this property in JetZ systems. In the section on the actual JavaCard implementation using a JetZ-type runtime system, we can give hard numbers on the performance of a typical reference application.
There is no number one reason for the outstanding performance of JetZ VMs. It is the result of the combination of the different properties of the JetZ system:

- *Scalability*: Only the required components are available in one particular configuration. No additional overhead incurred by dormant code or data structures has to be carried around.
- *Preprocessing*: All code to be executed on the VM is preprocessed on a converter in our development environment. In this conversion step, information not essential at run time are stripped, and all possible static analyses are made, thus reducing the need for an on-chip bytecode verification. The resultant code is a factor of 2 to 5 smaller than the original Java classfiles, but still contains all runtime structures to enable the virtual machine to actually interpret the code with all the security benefits known from this approach.
- *Specialized bytecode*: Due to the preprocessing of standard Java classfiles, specially optimized bytecodes are executed in the Virtual Machine.

The net result of this approach is that the functionality of a Java virtual machine is effectively split in two intimately connected, but nevertheless separate parts: One is run during development, the other at execution time on the embedded device. Integrity and authenticity of downloaded code is checked using standard code signing techniques. The required cryptographic methods are bundled with all JetZ software, if they are not already present on the respective hardware. The converter for any given platform JetZ supports is always provided together with the Virtual Machine. It is a standalone program that may also be run immediately prior to upload of new, standard Java classfiles into an embedded device.

Advantages

The main advantages of the architecture outlined above for a user of the JetZ technology are listed below in the order in which they were important for us during development. Each topic concludes with a link to the respective part substantiating the claim in the concrete JavaCard implementation.

- *Performance*: Code executed on a JetZ VM runs with a speed comparable to that of special-purpose code for any particular device. This is mainly due to the fact that JetZ is mainly

used for programming control-flow problems. All time- and resource-consuming operations like communications or cryptography are implemented in efficient, platform-dependent system library modules. [JavaCard example]

- *Size*: Awareness of the hardware costs and power consumption properties of RAM, EEPROM/Flash, and ROM as found in typical embedded devices led to a VM that requires in its smallest configuration 3600 Bytes for its code in ROM and 256 Bytes of RAM for its execution. [JavaCard example]
- *Security*: Integrity of the runtime system, i.e., the sandboxing known from Java is sustained. [JavaCard example]
- *Hardware constraints*: The special properties of embedded hardware, e.g., the different memory-access characteristics of RAM vs. EEPROM has been taken into account in the design of the runtime and system libraries of JetZ. In this respect, a JetZ VM significantly eases the job of application developers. [JavaCard example]
- *Ease of development*: By having identically performing runtime environments on PCs, developers can concentrate on the problem, and do not have to worry about slightly changing semantics of execution as known on today's JDK-type Java implementations. [JavaCard example]

---

3. The first JetZ result: The IBM JavaCard

SmartCard basics

A SmartCard is a credit-card sized device bearing a microprocessor under a usually golden-colored contact plate. It is defined in all respects, including dimension, electrical, and physical properties by a standard issued by the International Organization for Standardization (ISO). The standard **ISO 7816** in particular defines and limits the extensions of the silicon die embedded into the plastic card body to be at most 25 square millimeter. It also defines the microprocessor powering, clocking, and the communications protocol. Various hardware manufacturers produce chips conforming to these constraints. The distinguishing factor is always the type of microprocessor (8-, 16-, or 32 bit), the amount of ROM, RAM, EEPROM or Flash memory, and the presence of various coprocessors. Examples for the latter are DES or RSA computational engines, or hardware communications support. All of these dimensions contribute to the ultimate difference in hardware, the price of the SmartCard. Given the immense volumes of current SmartCard sales, and the even bigger contemplated numbers, even minute differences in price can make a huge difference. Therefore, the least requirements the software running on the SmartCard has, the more inexpensive, and the more competitive the whole package can be.
A relatively recent development are SmartCards using a contactless, i.e., an RF interface. They are primarily used in building access or transportation applications. The communications protocol is executed 'over the air' using an antenna embedded into the plastic body of the SmartCard. ISO is also standardizing this type of SmartCard under ISO 14443.

Special issues

Compared with regular computers, the following are the main differences imposed by the SmartCard hardware and software environments.

- **Persistent Memory**: As compared to RAM, EEPROM has a write-access time of typically 7 milliseconds and therefore is some factor of 10000 slower than RAM. Also, due to the technology employed, a program can only rely on a limited number of writes (typ. 100000-500000) to succeed.

- **Consistency**: As a SmartCard does not contain a power source of its own, it is totally dependent on external power supply and clocking. Therefore, care for the consistency of the internal, persistent data structures must be taken when programming a system. Inadvertent, or malicious power interruptions are the main cause for breaches of a SmartCards integrity.
- **Standards**: A SmartCard must fully comply to the spirit of the ISO 7816 or 14443 standards to be interoperable with the plethora of SmartCard terminal equipment already deployed worldwide.
- **Programming**: There is much more diversity in SmartCard hardware and operating system support than is currently present for example in the PC area, thus making SmartCard application programming a tedious, error-prone, and expensive undertaking.
- **Production turn-around**: Due to the production process of SmartCards, there is an extremely long turn-around between ROM code readiness and card availability. Typically, this is between 5 and 8 weeks. Reducing this turn-around time benefits significantly the use of SmartCards in today's fast moving world.

JavaCard

The term JavaCard is copyrighted by Sun Microsystems, and defines a SmartCard as outlined above bearing a Java-type virtual machine for execution of bytecodes in an interpreted way similar to the one laid down in Sun's *Java language and runtime specification*. The specification currently has the level 2.1, and is being developed jointly by Sun Microsystems, and the JavaCard Forum, a collection of the world's major SmartCard manufacturers. In short, a JavaCard is for all means and purposes a SmartCard; it can be used in all cases where currently SmartCards are used. From the outside, it is not different than other SmartCards. The main difference is, that programming the SmartCard can now be done using Java as the language, a Virtual Machine as the execution platform, and common Java development tools.
In the following, a rough breakdown of the software present on the JavaCard is given.

- Device drivers: The lowest level of functionality. It provides access to the typically two or three main devices with which the SmartCard processor interacts:
  - Communications: Each SmartCard hardware has some I/O facilities providing access to the physical contacts to the outside world. In the case of contact-based SmartCards, one of the two main communications protocols, T=0 or T=1 must be provided to interact with SmartCard terminals. In the case of contactless SmartCards, the upcoming ISO 14443 standard T=CL protocol would need to be supported.
  - Memory access: Each SmartCard hardware has different types of transient and persistent storage facilities. The first is usually static RAM, the latter EEPROM, increasingly also Flash RAM. The device drivers must cater for efficient use of the various possible page sizes. This improves general timing behavior and the lifetime of the memory.
  - Cryptography: This is an optional element, but usually the one that mainly defines the practical usefulness of a SmartCard. If no DES coprocessor is provided, the complete algorithm is implemented in this low-level driver making use of the SmartCard main processor. If it is, as in the case of public key support, some driver code has to be written to efficiently execute encryption, decryption, signature, or hash functions.
- Virtual Machine: It interprets bytecode instructions as defined in the JavaCard specification. The latter differs from the standard, JDK-type Virtual Machine specification in that not all data types are supported (no floating and 32 bit operations), and that no multithreading support is required.
- System libraries: A special set of Java APIs has been defined which regulates access to the device driver's capabilities. The JavaCard API also casts into a language-specific structure the server-like behavior of the JavaCard as a SmartCard. In particular, it is specified that only one application (one *applet*) can be active at any given time. All incoming commands from the

SmartCard terminal are directed to this applet's `process()` method. Special `SELECT` commands are used to switch between applets.
Special library functions are provided to efficiently load and link new code into a JavaCard. These are part of the OpenPlatform specification outlined below.

- ROM Applets: While the system libraries provide JavaCard specific interfacing to the actual hardware, and therefore contain some amount of native code, this does not hold true for true Java applets. Examples for the latter are standard SmartCard applications like loyalty, application registration, e-cash, or signature applets. There are two big advantages of providing as much functionality in Java as possible. The first is that the applet development can be carried out on any JetZ-type Virtual Machine on any platform. The second one is that the code is guaranteed to run unchanged on all devices featuring a JetZ-type Virtual Machine.

OpenPlatform

VISA has defined the OpenPlatform API and command specifications as a general guideline for securely managing a multiapplication card in general, and loading new functionality onto it in particular. Special care has been taken to devise a system that is highly resilient against common as well as novel attacks to a multiapplication SmartCard. The concept of a *CardDomain* has been introduced to capture the data structures representing the issuer of a SmartCard. It is used to control access to the card as a whole. The second, optional concept of *SecurityDomain* represents the possibly different applet issuers that are permitted to upload new functionality onto the same card.

IBM JavaCard capabilities

In this section, the raw numbers and features contained in the IBM JavaCard implementation based on the JetZ technology is given.

Hardware

Infineon 66 series (formerly known as Siemens SLE66) bearing an Intel 8051-core microprocessor, a big integer arithmetic coprocessor for public key computations, and a hardware-based true random number generator. It features 1200 Bytes of RAM, 16 kBytes of EEPROM, and 32 kBytes of ROM.

Device Drivers

We have implemented three types of device drivers. The first one provides cryptographic functionality using the main processor for DES computations, and the cryptographic coprocessor for public key cryptography. The IBM JavaCard therefore can compute DES, Triple DES, RSA with key lengths varying between 512 and 1024 bits both using private key operations and CRT (Chinese remainder theorem) operations. It can also run DSA, SHA1, and generate on-card both RSA and DSA private keys. The latter capabilities are (so far) unique in JavaCards.
The second type of device driver handles the complete communications protocol as defined in ISO7816. We have implemented the two major protocols, T=0 and T=1. This dual protocol capability is rare in regular SmartCards, and (so far) unique in JavaCards. It ensures an optimum interoperability with nearly all SmartCard terminals in the world, which are more or less evenly split between T=0 and T=1.
The third major device driver category is the EEPROM and RAM memory management. Efficient copying, and EEPROM access routines make up a significant part of good performance of a SmartCard. This part is specially designed to most effectively cater to the higher-layer services, e.g., Java garbage collection, as outlined below.

Virtual Machine

The JavaCard virtual machine implements 127 different bytecodes. It provides all the runtime security checks known from regular Java, e.g., stack overflow, array boundary checks, or type-safe assignment of objects. It provides an efficient execption handling mechanism and offers the complete set of object oriented features known from regular JDK-type Java.

System libraries

The IBM JavaCard implements the complete set of APIs as required in the JavaCard standard. As optional components, a highly efficient RAM management, cryptographic capability access, and transaction support have been implemented. All optional elements have been created out of a concern for optimum performance and code size.
The concept of *Transient and Persistent Environment* for efficient RAM usage has been submitted to Sun and the JavaCard Forum. It is a major departure from the standard Java memory management model, as it introduces the difference between persistent (EEPROM) and transient (RAM) memory at the application programming level. It has been motivated by the general JetZ concerns of resource constraints as present in embedded devices. It has found the blessing of both Sun and the SmartCard industry. Application developers familiar with the problems of embedded devices have found it most useful, too. Similar statements hold true for our small cryptographic API called *CryptoZ*, and our proposal for support of long-lived transactions.

OpenPlatform

The full set of capabilities defined in VISA OpenPlatform, specification level 1.0+ has been implemented. This includes secure applet upload using single and triple DES session keys over the OP CardDomain. Also implemented is the SecurityDomain concept. Besides the low-level linking routines used during applet upload, all code of the IBM OpenPlatform implementation are written in Java.
Development of the complete OpenPlatform functionality took place on a Linux machine with a 32 bit version of the JetZ engine for reasons of convenience. After running the converter for the 8051 platform of the JavaCard hardware, the system ran there immediately OK.

Applets

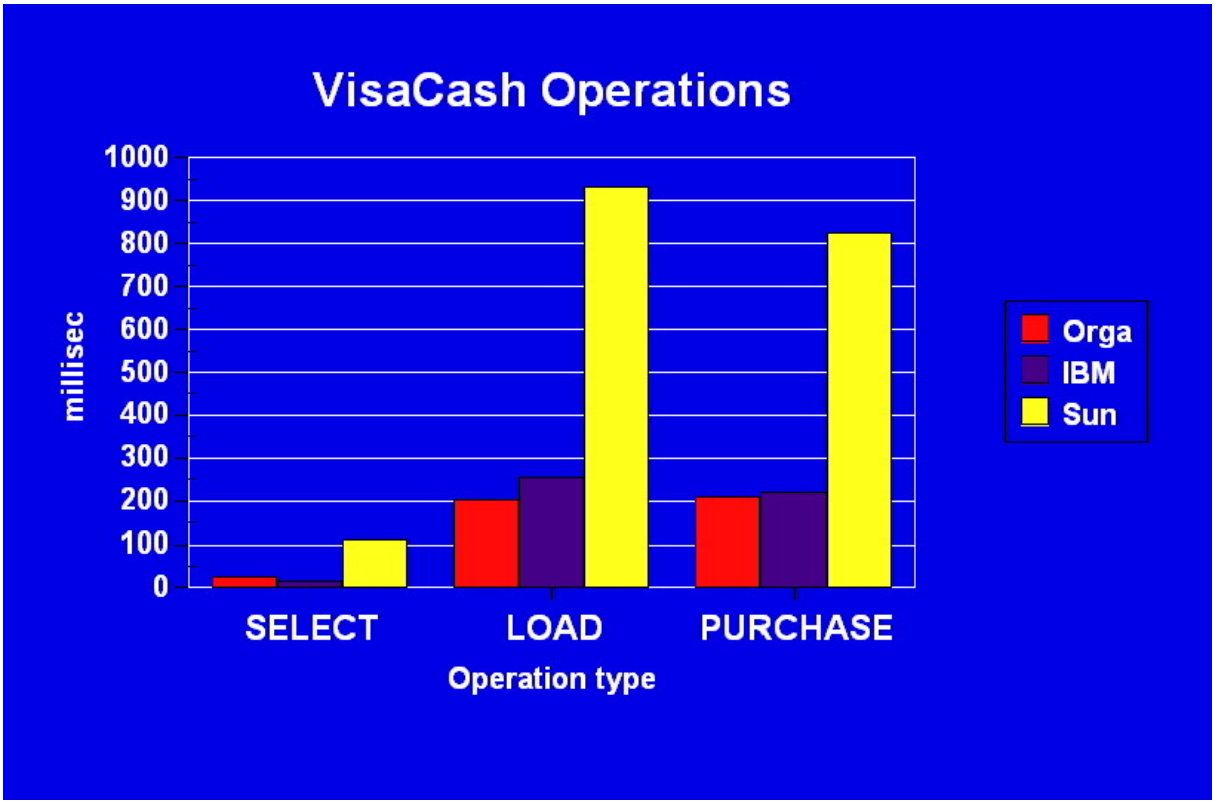
In order to fully comply with the VISA requirements for an OpenPlatform card, the standard applications *EMV PSE (Payment Systems Environment)* and *EasyEntry* are present as applets in the ROM mask.
Also available in ROM is an applet providing convenient access to the cryptographic capabilities of the IBM JavaCard. Its abstraction is that of a vault, or cryptographic object contained, thus facilitating its use as a cryptographic service provider for PC-DLLs offering PKCS#11 services.

VisaCash performance

In an effort to make the capabilities of all JavaCards comparable at the level where performance and size matter in the end, i.e., the application level, we have implemented an applet fully compliant to the VISA specifications for VisaCash. It is an electronic cash protocol based on DES cryptography which is used in existing SmartCard installations. The applet is written entirely in Java, and has been functionally certified by VISA. It received an excellent rating in terms of security and correctness. Below, we have listed the raw performance of the main VisaCash operations as they can be measured from outside the card at a card clock rate of 3.75 MHz:

VisaCash operation	Performance (in milliseconds)
SELECT	16.74
INIT LOAD	66.96
LOAD	139.59
INIT UPDATE	23.06
UPDATE	108.62
INIT PURCHASE	19.06
PURCHASE	177.91
READ BINARY	8.84

In a more direct comparison between the performance of a standard VisaCash card (native code implementation) from Orga, and a VisaCash applet running on the JavaCard Virtual Machine developed by Sun Microsystems, the performance advantage of our JetZ technology becomes clearly visible.



Summary

This section shall be concluded by reminding the reader again of the fact that the hardware on which this plethora of functionality is provided, bears just 32 KBytes of ROM into which all the code fits, and 1200 Bytes of RAM for the machine stack, the Java runtime stack, the communications and cryptographic buffers.
To close the circle to the promise of JetZ, we have made available another configuration of the IBM JavaCard, running in 20 kBytes of ROM and 256 Bytes of RAM. Missing from all the features outlined above are only the VISA applets, the garbage collector, and support for public key cryptography. None of the JetZ code in C has been changed to assembler.

Usage and Productization

The IBM JavaCard is currently in use by many different organizations. This entails educational institutions, as well as systems integrators, and end users. Due to confidentiality agreements, only a few can be currently named. In particular, actual SmartCard rollouts are kept under locks until the actual event. To always keep you up to date on this, we are maintaining a Website referring to the press releases announcing the respective rollouts.
At the time of writing, the most interesting project where this card is used, involved the US General Services Administration (GSA). It is so far the sole JavaCard product rollout in the US. The cards are issued by CitiBank, NY. Integration services have been provided by IBM, 3G-International, and GTE. The card is used to log on to computers using a biometric authentication procedure and strong cryptography. It also serves as a loyalty card for the existing American Airlines frequent flyer program. Further applications may be loaded on demand using the secure OpenPlatform applet upload process. A sample press release is accessible at
http://www.fcw.com/pubs/fcw/1999/0426/fcw-newsgsa-4-26-99.html

4. Future paths

After having proven the viability and applicability of the JetZ approach to the smallest computer for which a Java runtime was desired, the SmartCard, the project now focuses on new categories of embedded devices. For one, these are SmartCard terminals. This will solve many of the interoperability problems currently seen in these devices. From an engineering perspective, this will be a straightforward application of our current JavaCard technology, since these devices typically carry the same class of processor as the actual SmartCards do.
A significantly more interesting area will be the one of mobile phones. Here, the new dimension of power consumption will be playing a serious role in the deployment of Java virtual machine technology. The VM may not require more resources than the core functionality of the device, i.e., providing mobile connectivity over the air. IBM already participates actively in the respective standardization bodies in this area. We are looking forward to backing these efforts by a reference implementation if so desired by a major mobile phone provider.
A third area of applicability of JetZ is foreseen for the PDA form factor. Here, resource constraints are not as pressing as on mobile phones, but given the imminent merge of the core functions mobile phone, PDA, and personal security/identification token, this is an area where JetZ technology can be effectively employed.
In any case, it shall be possible to keep any code currently developed for the IBM JavaCard and run it on any other device the JetZ VM becomes available. This is an important competitive advantage if early product availability is desired. For example, a PDA bearing a JetZ-type Virtual Machine with the JavaCard APIs could be used to sign electronic checks without requiring --as currently-- the IBM JavaCard. In another scenario, a mobile phone may be used as a reloadable VisaCash token: The code is available, it is certified, and just needs to be loaded in a secure manner to the device.

5. Conclusion

This document outlined the general ideas of the JetZ research project executed at the IBM Zurich Research Laboratory. It showed the first practical result of this work, the IBM JavaCard, a small-scale Java Virtual Machine running on an 8-bit SmartCard processor with extremely constrained resources, e.g. with a mere 256 Bytes of RAM. The result of this work is available as a product: You can get it for example from GemPlus, one company to whom the SmartCard technology part of JetZ has already been licensed. It is called *GemXpresso 210PK*.

Michael Baentsch (mib@zurich.ibm.com)