

# Facilitating Context-Awareness Through Hardware Personalization Devices: The Simplicity Device

J. Papanis, S. Kapellaki, E. Koutsoloukas, N. Dellas, G.N. Prezerakos<sup>1</sup>,  
and I.S. Venieris

Intelligent Communications & Broadband Networks Laboratory,  
School of Electrical and Computer Engineering, National Technical University of Athens  
{jopapan, sofiak, lefterisk, ndellas, prezerak}@telecom.ntua.gr,  
venieris@cs.ntua.gr

**Abstract.** The paper presents the specification, development and initial performance evaluation of the Simplicity Device (SD), which is one of the key components of the SIMPLICITY Architecture. The SD is both a portable personalization device and a hardware Single-Sign-On (SSO) token. It accommodates the Simplicity User Profile (SUP), which contains user context related information and has been designed according to 3GPP and W3C standards. In cooperation with a distributed brokerage framework the SD provides the users with the means to automatically personalize terminals and services according to their context by the simple act of “plugging” the SD into any SIMPLICITY compliant user equipment. Within this paper we present the generic SD architecture which is the basis for different SD implementations and consequently we focus on a JavaCard SD implementation and its performance evaluation.

## 1 Introduction

One of the major goals of pervasive computing is to provide highly personalized context-aware services with the least possible user interference. These services usually require more complex transactions than the traditional client-server model, where the information and the required resources were conveniently stored centrally at the network side and managed by the service provider alone. Services have evolved towards a decentralized compound model; resources are distributed throughout the network but are also located in the user’s environment in the form of embedded devices with networking capabilities. All these entities must cooperate towards successful service provision. User related information needs to be retrieved and processed in order for context to be formed and exploited. Consequently service provision should adapt to user context.

We present the approach followed by the IST SIMPLICITY project (Secure, Internet-able, Mobile Platforms Leading Citizens Towards simplicity) [1]. SIMPLICITY aims at users of existing as well as emerging services who employ different terminals (including PCs, PDAs and mobile phones) over wired and wireless networks. The user should enjoy personalized context aware services with minimal configuration

---

<sup>1</sup> G. N. Prezerakos is also with the Technological Education Institute (TEI) of Piraeus, Dpt. of Electronic Computing Systems, 250 Thivon Av. & Petrou Ralli, 122 44 Athens, Greece.

overhead. SIMPLICITY tackles complexity by combining the following means: (a) A personalization device with a simple and uniform mechanism for customizing services and devices (used for user identification / authentication as well as a means to store user profiles, preferences and policies) (b) a distributed brokerage framework that encompasses the various components that constitute an end-user application.

The personalization device proposed by the SIMPLICITY, aptly named Simplicity Device, is a portable and physically robust token that the user can carry everywhere and anytime. It helps the user interact with terminal devices and services. Users can personalize terminals and services by the simple act of “plugging” their SD into the chosen terminal. Depending on the different capabilities of the SD implementation, it can store the user profiles, user preferences, user policies and terminal settings and/or be a pointer to information stored elsewhere.

Before delving into the details of the SD, the following section gives an overview of the end-to-end SIMPLICITY architecture.

## 2 The Simplicity Architecture

SIMPLICITY is a distributed system that delivers advanced personalization for users, through context awareness and policy based adaptation. Overall, SIMPLICITY introduces the novel idea of a personalization device, the Simplicity Device (SD) which is supported by a distributed brokerage framework [2].

The central entity in this framework is the Simplicity Broker, a compound software entity that consists of individual subsystems delivering Simplicity services to users. Different broker setups have been specified in SIMPLICITY, most notably the Terminal Broker (TB) which resides in user terminal equipment and is responsible to deliver personalization of the terminal environment for the user, the Network Broker (NB) which resides at the network side and mediates between the Terminal Broker and network side services, and the Service Broker (SB) which hosts interfaces, APIs and adaptors for 3rd party Simplicity compliant services. The user’s interface to Simplicity is the Simplicity Personal Assistant (SPA), an entity that manages the graphical user interfaces, takes actions on behalf of the user and orchestrates the brokerage framework through the Terminal Broker. Existing hardware and software outside Simplicity, collectively labeled “legacy entities” (LE) integrates with Simplicity through the use of special adaptor interfaces.

Simplicity brokers are modular architectural entities, consisted of loosely coupled software entities called Subsystems. Subsystems are autonomous entities that encapsulate Simplicity specific functionality; they cooperate through the exchange of messages in an asynchronous event-based fashion. Subsystems are attached in a central lightweight entity, the Mediator, which is responsible for filtering messages and delivering them to their intended recipients. A special kind of subsystem, the Adaptor subsystem is the intermediate entity between legacy entities and Simplicity. The Simplicity Broker Communication (SBC) subsystem is another special subsystem that fulfils the task of broker discovery and communication with remotely attached subsystems. Finally, the Simplicity Device Access Manager (SDAM) is a special case of an adaptor subsystem that provides interfaces for different SD realizations and abstracts the different realizations to the rest of the Simplicity system.

Figure 1 illustrates the internal broker architecture and lists the aforementioned core subsystems.

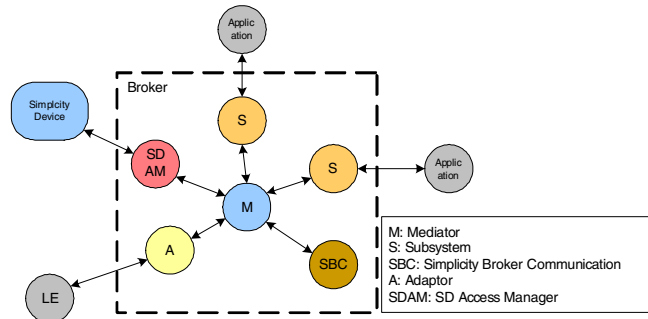


Fig. 1. Internal Broker Architecture

The focus in the following sections will be on the Simplicity Device implementation and its interaction with the Simplicity Device Access Manager.

### 3 Simplicity Device Functionality

Being the innovative part of the Simplicity vision, the Simplicity Device has a prominent role in the Simplicity Architecture. The SD is seen as something more than a secure storage token; it is a personalization enabler that inflicts on users' computing and networking environment a certain degree of mobility. A smart and personalized environment follows the user anywhere that a Simplicity compliant terminal can be found.

In order to realize this task, the Simplicity Device makes use of a structured user profile that contains personalization information about the user himself, about the available services, the network connectivity options, the device capabilities and the terminal environment preferences. The Simplicity User Profile (SUP) was modeled according to the methodology proposed by 3GPP for Generic User Profile (GUP) [3] and device related profiles were developed based on the W3C Composite Capabilities/Preferences Profile framework (CC/PP) [4]. SUP is stored on a Simplicity Device realization and the device implements store, retrieve and verify procedures while operating on the SUP, thus ensuring data integrity as well as placing access on SUP under the device's strict security control. Depending on the different capabilities of the device implementation, it can store the user profiles, user preferences, user policies and terminal settings locally and/or be a pointer to information stored at specific nodes in the network. Additional information about the relation between SUP, the GUP and CC/PP can be found at [5].

Besides the personalization features, the SD provides single-sign-on (SSO) functionality to the user. Seamless service provision requires that users entering a service provisioning framework are identified only once and consequently they can use services based on their access rights and personal preferences. Several SSO solutions have been proposed [6], [7]. The most publicized frameworks are the Microsoft's Passport [8] and the Liberty Alliance Project [9], both proposing that the user's profile is stored at a central point within the network. The innovative feature, however, that Simplicity projects, is the use of a hardware SSO token, the SD, which is both

more secure and more practical to use, since the steps required are to plug in the device and provide a PIN number to the Simplicity Personal Assistant.

Personalization and the single-sign-on function are the two key features that the Simplicity Device contributes to the Simplicity vision. The following section investigates the requirements that arise from these features and describes the architecture and implementation of a JavaCard based Simplicity Device.

#### 4 The Simplicity Device Architecture and Implementation

Summarizing the facts from the previous section, an ideal SD should meet the following requirements:

- Portability (small size, lightweight, low power consumption)
- Plug & play & unplug (bootstrap procedure, insert and removal notification)
- Computational Power (profile management, security algorithms)
- Data Storage
- Security (Authentication\Encryption)
- Ease of programming

Unfortunately some of the above requirements are conflicting. But if we had to choose one of them, the most important is portability with an emphasis on small-sized devices, which are resource-constrained. Thus the personalization device should be offered in different implementations that integrate with currently available and emerging terminals with minimum effort. Three main SD types can be defined taking also into account the fact that the functionality of the SD has to be implemented on resource constrained devices:

- Storage and processing devices offering a rich set of functions and facilities, exploiting the storage and processing capabilities of the device.
- Pure storage devices with a large storage capability allowing the storage of significant volumes of data
- Virtual devices consist of a software solution based on a network infrastructure

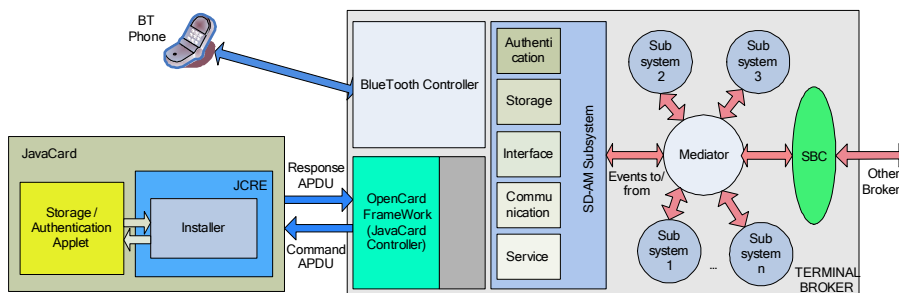


Fig. 2. Simplicity Device Architecture

Given the above options the SIMPLICITY project chose to implement a JavaCard [110] based SD and a legacy SIM card combined with Bluetooth [11] mobile phone based SD. Both of these implementations are portable, can easily connect with a wide selection of terminals (JavaCard uses the USB interface and the SIM realization uses the wide spread Bluetooth protocol) and also offer some processing power and storage space. In order to overcome the capacity limitations posed by the SD hardware, a part of the SD software entities is located in the terminal and a part in the physical SD. Figure 2 contains the main blocks of the SD and terminal architecture.

In the following section the paper focuses mainly on the JavaCard / USB SD implementation which was specified and developed by our group.

#### 4.1 SD Terminal Part

Every terminal subsystem should be able to cooperate with any SD type. For this reason a special subsystem called the SD Access Manager (SDAM) was created which:

- provides uniform interface for all classes of SD implementations towards the terminal broker
- is responsible to provide access to functionality originally assigned to the personalization device, even if this functionality is provided by some other component, due to restrictions of the specific personalization device implementation.

SDAM's API supports both types of personalization devices (JavaCard & Bluetooth phone). The terminal broker subsystems should not bear the burden of distinguishing to which type a specific SD implementation belongs to. Any functionality that is not directly supported by a specific SD implementation is provided by the SDAM. In order to achieve this, the SDAM consists of a common part and specific entities, called Controllers, for each implementation. The common part is split into five managers, which are described in the following paragraphs.

The Authentication Manager is responsible for authentication and security issues regarding the SD. The tasks include authentication of the user towards the SD (performed by entering a Personal Identifier Number (PIN) code) and authentication of the SD towards the network (in order to access the Simplicity services). Additionally specific 3rd party services and applications that may try to access and possibly modify data stored in the SD should have the proper access rights. The Storage Manager component is responsible for the management of the data stored in the physical SD. It allows accessing the contents of the SD by implementing a basic file-system, providing file-system operations like read, write, delete and update. The Interface Manager is responsible for handling the Simplicity events coming from the Mediator, forwarding them to the corresponding components and also encapsulating the responses from the components into Simplicity events that are transmitted back to the Mediator. For example a request such as "get user's billing preferences" will be interpreted into downloading a specific part from the SUP. The Interface Manager is also involved in the bootstrapping procedure of the SD. The Communication Manager component is another vital component for the proper function of the SDAM. The most important services are presence detection of a connected SD, handling of status changes

(insertion and removal), discovering of SD capabilities and finally instantiation of the respective Controller. Another responsibility is the discovery of any available online data storage that may hold SD data and then acting as a proxy towards it. Finally, the Service Manager component is offering a collection of services to other components, which focus on improving the overall speed - performance and security of the subsystem. The available services include caching profile data from the physical SD, encoding, decoding and encryption of data and a finally maintaining a permission access table.

The JavaCard Controller is a high level interface, which resides in the terminal part and is responsible for hiding the complexity of the JavaCard implementations from the SDAM. Therefore terminal software developers are not concerned with the internal organization of the JavaCard and its associated concepts such as Application Protocol Data Units (APDUs), JavaCard Applets, Application Identifiers (AID) etc. In order to achieve this we followed the OpenCard Framework (OCF) [12], which communicates via appropriate drivers with the programs (applets) running in the JavaCard SD through the USB port. By using the OCF, we avoid making vendor specific software, i.e. the terminal side becomes rather versatile and should work with smart cards from numerous vendors. When the JavaCard Controller receives a request from the SDAM, it forms a suitable APDU and passes it to the OCF that forwards it to the physical JavaCard via USB. When it receives the response from the JavaCard, it decomposes the APDU(s) and forwards the valuable information to the SDAM. The complexity of the process is alleviated by the Controller, for example when writing a file to the JavaCard, the current implementation of the Controller has to split the file in segments that fit in the APDU's payload size and also maintain and update a mapping between files (user profile, preferences) and storage positions in the memory of the card.

## 4.2 SD Physical Part

Multifunctional smart cards based on JavaCard technology can be implemented in two different ways. The first and probably most obvious is to create a separate applet for each command. This method relies a lot on the card issuers to implement their cards correctly and with a decent security scheme where necessary. A major drawback is, that the various applets know nothing of each other, so there's a strong need for the applets to implement certain interfaces, in order to perform inter-applet communication. The inter-applet communication should be considered a necessity, as the applets will share objects like PIN and SUP. Furthermore, the procedure of uploading and installing an applet onto a card is proprietary and tedious. Another strategy which is the one used in our implementation is to implement only a single applet and let that applet control everything. The applet handles creation and modification of the data and can be reached through OCF applications, which makes it easy to deliver a well-defined interface to any developer.

The physical mapping on a JavaCard foresees the use of memory locations reserved inside the context of a JavaCard applet. These memory locations are allocated by means of a memory buffer constituted by one or more byte array (depending on the memory constraints of the card). The communication between a device equipped with a smart card reader (PC, PDA, Phone, etc.) and the JavaCard is based on the APDU

protocol mapped into interface primitives. ISO 7816-4 [13] defines the APDU protocol as an application-level protocol between a smart card and an application on a suited device. The first APDU is addressed to the installer applet of the JavaCard, which is provided by the vendor by default. This command selects the suitable applet for the operation we need to perform. After selecting the applet, all following APDUs are directed to that applet and processed by its process method, until a deselect command is issued or another applet is selected.

A storage/authorization applet realizes a portable data memory with standard ISO/IEC 7816-4 commands. In order to read, write or delete data, the user has to authenticate using the PIN verification, which is stored in the card. Inside the data objects are organized in ASN.1 BER TLV coded data objects. Every data object has its own tag and length, which are used in the APDUs for read and write. The applet was developed on the GemXplore Developer environment which includes an Integrated Development Environment bundled with JavaCard 2.1 compliant smart-cards [14]. Figure 3 displays the code that is executed by the Controller during the authentication.

<pre>public boolean activateSD(String pass) throws AuthenticationException {      String[] passp = pass.split(" ");     byte[] Pinn = new byte[passp.length];     for(int i=0;i&lt;passp.length;i++)         Pinn[i] = Byte.parseByte(passp[i]);     ResponseAPDU myResponse = null;     byte bZero = (byte)0x00;     byte verify_position = (byte)0x01;     CommandAPDU request =         new CommandAPDU(5+Pinn.length);     request.setLength(0);     request.append(CLASS);</pre>	<pre>request.append(VERIFY_INS); request.append(bZero); request.append(verify_position); request.append((byte)Pinn.length); for(int i=0; i&lt;Pinn.length; i++)     request.append(Pinn[i]); card.beginMutex(); myResponse =     sdjCardProxy.send(request); if (myResponse.sw() == 0x9000)     return true; else return false; card.endMutex(); }</pre>
---	--

**Fig. 3.** Authentication method at the Controller

First, a string password is converted into bytes (JavaCard has no String) and a proper request Apdu is formed by appending APDU CLASS, INS, lengths and payload (password). Consequently, the card is being contacted, the Apdu is sent and the response is read (0x9000 is the OK code).

## 5 Performance Evaluation of the JavaCard SD

The objective of our measurements is to obtain information concerning the time needed to read and write data from and to the memory of the JavaCard. The measurements concern the reading and writing of a number of raw data with length varying from 1 to 10000 Bytes. More specifically we obtained 10 measurements of 1, 15, 50, 100, 200, 300, 500, 1000, 5000 and 10000 Bytes with a minimum of 100 samples per measurement. In order to measure the times we have used the standard method

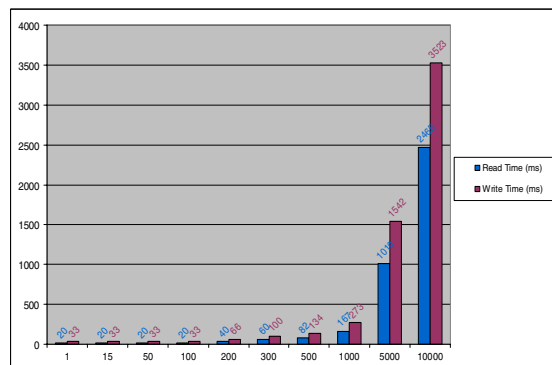
System.currentTimeMillis() supplied from the Java Core (we used is JRE 1.4.2\_01-b05 for our measurements). This method returns the current time in milliseconds, which is precise enough for our measurements.

The input Bytes were read from files with appropriate sizes (1 Byte to 10KBytes – the size of the files was actual Byte count, not the size these files occupy on disk). We measured the time needed to format the APDUs (set memory position to read or write data and apply data in case of write), send them and receive response from the card (data included also in case of read).

The JavaCard that we used for our implementation and measurements is the GemXplore Xpresso v3.2 from GemPlus, which comes together with a USB Card Reader. The PC which was connected to the card was an Intel Pentium III 1.8 MHz with 512 MBs of RAM. Finally the USB port of the PC is version 1.1. The measured mean times for varying-length bulk data read and write operations are presented in Table 1.

**Table 1.** JavaCard-based SD Measurements

Bytes	Read time (ms)	Write time (ms)
1	20	33
15	20	33
50	20	33
100	20	33
200	40	66
300	60	100
500	82	134
1000	167	273
5000	1010	1542
10000	2465	3523



**Fig. 4.** Read and Write times in ms

Figure 4 shows a graphic representation of the measurements. One can notice that the time needed for reading is always somewhat less than the time needed for writing, which is something we expected because data are read from / written to EEPROM memory and the writing EEPROM operations are slower than the reading ones.

Our specific implementation is another reason, since everyone can read from the card but in order to write the applet checks if the user is authorized.

In figures 5 and 6 we present results for different numbers of APDUs. Figure 6 shows a detail of Figure 5 from the first APDU till the eighth. One interesting observation is that for few APDUs (in our case the first 8) the measured times are multiples of the time required for one APDU. For example, if one APDU requires 20ms for reading and 33ms for writing, 8 APDUs require 167ms for reading, which is almost equal to  $8 \times 20$  (160ms) and 273ms for writing, which is almost equal to  $8 \times 33$  (264ms). For larger numbers of APDU this linear relationship no longer applies. An average SUP will be around 5 to 10 KBytes long. Our measurements suggest that writing/reading an average SUP to/from the JavaCard SD will require just a few seconds (2.5 for reading and 3.5 at most for writing). Reading/writing specific data (e.g. a username – 50 Bytes will just take 20/33 ms).

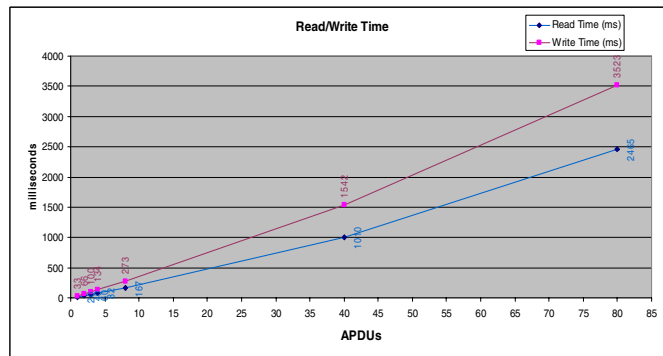


Fig. 5. Read and Write times/APDUs

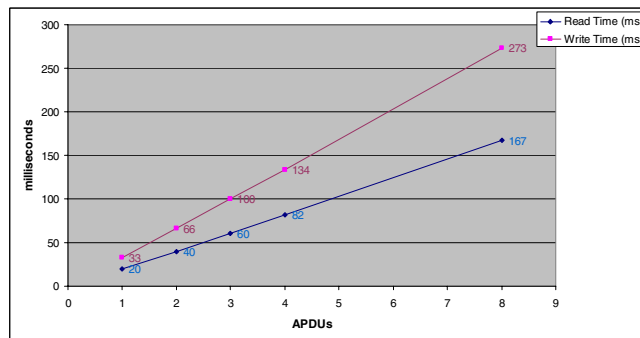


Fig. 6. Read and Write for 1 to 8 APDUs

In the future we intend to repeat these measurements on the final version of the JavaCard SD, which will provide optimized exchange of messages and maybe even a compression option for data stored in the memory of the SD. Since the data is in text format the compression will reduce the size significantly. Thus, we expect that the final version will offer significant performance enhancements.

## 6 Conclusions and Future Work

The design, implementation and initial performance evaluation of a JavaCard based SD, in the framework of the SIMPLICITY Architecture, has been extensively described within this paper. This personalization device with the usage of the SUP and the support of a distributed brokerage framework provides the users with the means to enjoy simplified, personalized and automated procedures for the usage of different terminals and services.

Ongoing work in SIMPLICITY aims at the integration of a number of applications to a fully working demonstrator. Depending on the application, a variety of terminals (laptops, PDAs, mobile phones) will be used via wireline and wireless connections combined with the different SD implementations. While the demonstrator will provide proof-of-concept of the SIMPLICITY architecture and the SD concept, on the other hand it is not a sufficient means for thoroughly evaluating the performance of the overall SIMPLICITY System as well as the SD part. For this reason, we are currently working on appropriate performance models (analytical and simulation-based) that will produce more detailed results and give a more pragmatic insight of the system's real operation. Also comparisons of the different SD implementations are included in future steps.

## Acknowledgement

The ideas that are described in this paper originate from the ongoing work in the Simplicity project, but most of them represent the personal elaboration of the authors. The authors wish to express their gratitude to the people working in the Simplicity Consortium.

## References

1. SIMPLICITY Project, <http://www.ist-simplicity.org>
2. N. Blefari Melazzi, S. Salsano, G. Bartolomeo, F. Martire, E. Fischer, C. Meyer, C. Niedermeier, R. Seidl, E. Rukzio, E. Koutsoloukas, J. Papanis, I. S. Venieris: "The Simplicity System Architecture", to appear in the Proceedings of the 14th IST Summit, 19-23 June 2005, Dresden, Germany.
3. GPP TS 23.240, "3GPP Generic User Profile (GUP) requirements; Architecture (Stage 2)"
4. W3C Recommendation 15 January 2004, "Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0"
5. E. Rukzio, G. N. Prezerakos, G. Cortese, E. Koutsoloukas, S. Kapellaki. "Context for Simplicity: A Basis for Context-aware Systems Based on the 3GPP Generic User Profile", International Conference on Computational Intelligence (ICCI 2004), pp. 466-469, 17th-19th December 2004, Istanbul, Turkey
6. J. Futagawa, "Integrating Network Services of Windows and UNIX for Single Sign-On", Proceedings of the 2004 International Conference on Cyberworlds (CW'04), pp. 323-328, November 2004, Tokyo, Japan

7. G. Zhao, D. Zheng, K. Chen, "Design of Single Sign-On", Proceedings of the E-Commerce Technology for Dynamic E-Business, IEEE International Conference on (CEC-East'04), pp. 253-256, September 2004, Beijing, China
8. Microsoft Corporation, ".NET Passport: Balanced Authentication Solutions". <http://www.microsoft.com/net/services/passport/balanced.asp>
9. Liberty Alliance Project, <http://www.projectliberty.org/>
10. Java Card technology specification available at <http://java.sun.com/products/javacard/index.jsp>
11. Bluetooth Official site, <https://www.bluetooth.org/>
12. OpenCard Framework web site, <http://www.opencard.org/>
13. ISO/IEC 7816-4:1995 Command set for microprocessor cards.
14. GemXplore Developer, [http://www.gemplus.com/products/gemxplore\\_developer/](http://www.gemplus.com/products/gemxplore_developer/)