Technology Reports

Application Functions for Autumn/Winter 2008 Models (2) Next-generation Java Applications

*It is now eight years since the launch of the i-appli service. To further strengthen i-appli's position as an application platform for the creation of diverse services and solutions, we have developed a new Java®*1 platform called "Star."*

Communication Device Development Department

Eriko Oseki
Mao Asai
Akiko Tobe
Niro Tsuchiya

1. Introduction

The i-appli service was started in 2001 when a profile*2 called DoCoMo Java (DoJa)*3 was installed in the mova 503i series. Since then, the DoJa profile has evolved to the point where a wide range of functions and services are now available in i-appli, including games, mobile electronic payments and corporate solutions.

However, with improvements in the performance of mobile terminals and diversification of mobile services, the demands being placed on DoJa are also becoming more sophisticated and diverse. This has resulted in a functional demand to undertake a comprehensive review of DoJa's basic design, and has necessitated a restructuring of its functional configuration.

To further strengthen DoJa's position as an application platform, we developed the new "Star" platform after considering its compatibility with existing content. Specifically, we reviewed the application life cycle and programming model, and we developed a new i-appli base class*4. To augment the existing i-appli functions we also developed WidgetView (i-Widget™*5), Java-Flash®*6 connectivity, library addition functions, My Menu registration functions, i-appli online functions, and i-appli call functions.

This article describes the main additional functions. Details of the i-appli online and i-appli call functions can be found in Ref. [1].

2. Overview of the Star Platform

2.1 System Configuration

The configuration of the Star platform module is shown in **Figure 1**.

This platform consists of a full application execution environment, a mini application execution environment and Java Application Managers (JAMs) performing tasks such as controlling the start-up of i-appli software in both execution environments. Different types of applications run in these two types of execution environment.

2.2 Advantages of Execution Environments

The full application execution environment is backwards-compatible with services provided by conventional i-appli software, and has been designed to facilitate the addition of new functions for a long time into the future. This environment consists of the standard Java class library Connected Limited Device Configuration (CLDC)*7 and a Star profile for implementing new services including DoJa functions. The Star profile has been designed to take over the services of the DoJa profile,

*1 **Java®**: A registered trademark of Sun Microsystems, Inc. in the United States.
*2 **Profile**: A group of APIs (see *9) and class libraries.
*3 **DoJa**: A suite of Java program components that provide functions used by the i-appli.
*4 **Base class**: A basic class from which other class definitions are derived. In object-oriented programming, a class is a template that defines data structures and methods.
*5 **i-Widget™**: A trademark of NTT DOCOMO.

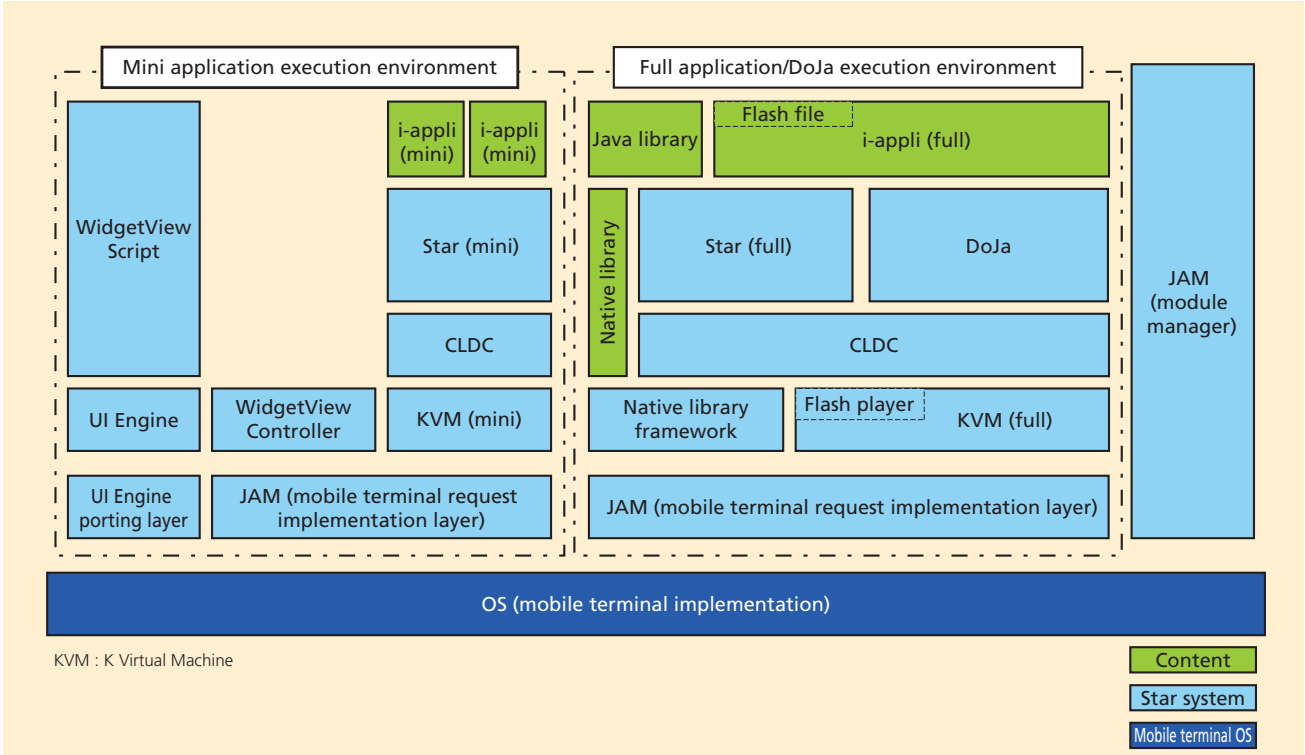

Figure 1 Star platform module configuration

but due to the refactoring[*8] of the Application Program Interface (API)[*9], there is no binary[*10] compatibility with the DoJa API.

Meanwhile, the purpose of the mini application execution environment (called WidgetView) is to run multiple applications simultaneously. Unlike the full application environment, WidgetView has a more limited range of functions and uses a Virtual Machine (VM)[*11] to process multiple applications simultaneously.

The two execution environments are configured as independent systems, and it is possible to switch between the two execution environments depending on the method[*12] calls made by the applications. In this way, we aimed to implement the creation and linked operation of applications so as to exploit the respective advantages of both execution environments.

2.3 Review of Application Life Cycle Model

Figure 2 shows the runtime state transition diagrams of the StarApplication class (the common application base class of the Star profile) and the IApplication class (the application base class of the DoJa profile). The various states of the StarApplication class are described in **Table 1**. Methods are indicated by the arrows in Fig. 2, where method names beginning with a forward slash represent callback methods invoked by the corresponding state transition, while the other method names represent state transitions that occur when a method is called from i-appli.

The StarApplication class has three advantages over the IApplication class.

The first is that it provides the substates "Semi-Active" and "Full-Active" when applications are running. Since the StarApplication class must support multiple applications running in paral-

*6 **Flash®**: A type of content developed by Macromedia and Adobe Systems which allows audio and vector graphics to be combined in animations. Flash is a trademark or registered trademark of Adobe Systems Inc. in the United States and other countries.
*7 **CLDC**: A Java configuration defined for miniature terminals such as mobile terminals and PDAs.
*8 **Refactoring**: Updating the source code in order to make improvements such as reducing redundancy without changing the program behavior.
*9 **API**: A set of rules that define programming procedures used to access commands and functions when software is developed for a particular platform (operating system or middleware).
*10 **Binary**: A computer program in a form that can be run directly on a computer.
*11 **VM**: Software that allows Java programs to run on different platforms.
*12 **Method**: In object-oriented programming, a method is an operation performed on an object's data.

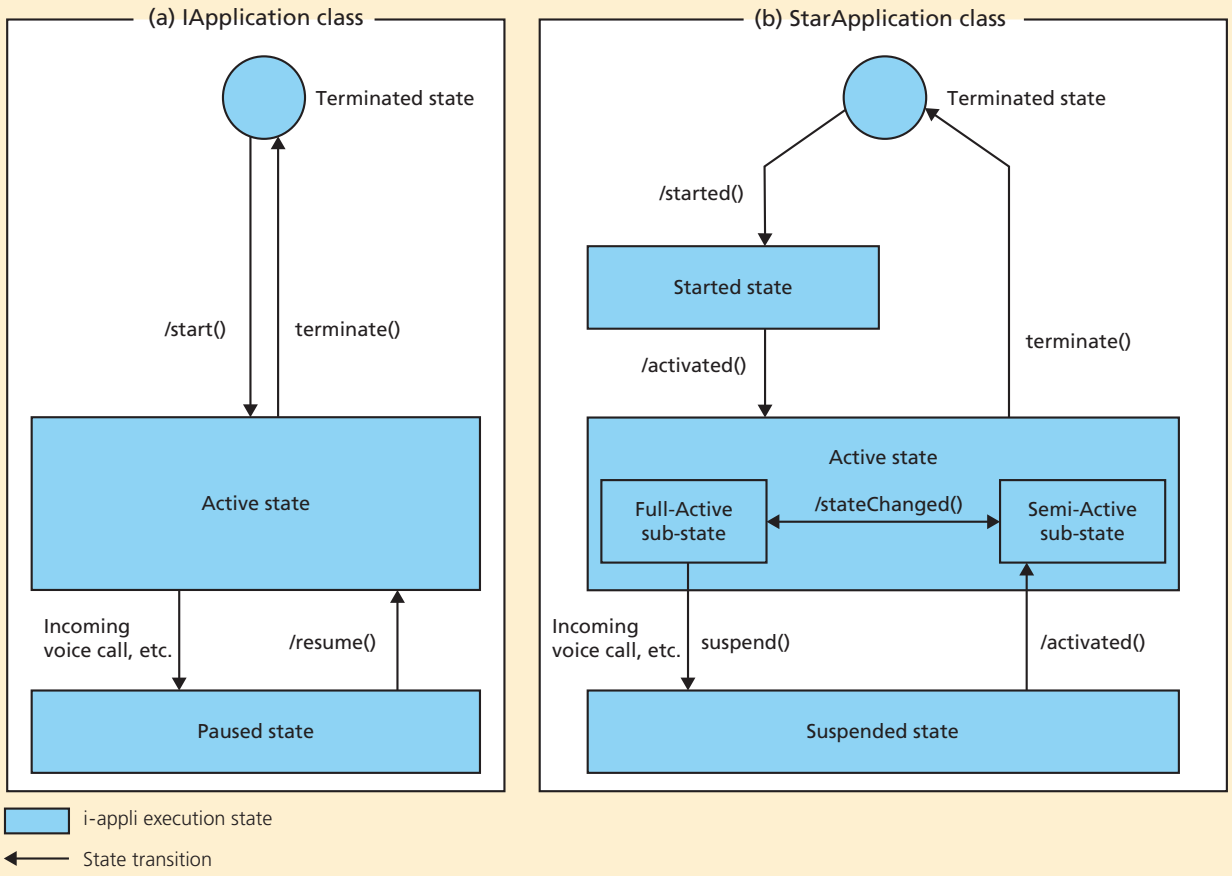

Figure 2 State transitions of application base classes

Table 1 Description of each state in the StarApplication state transition diagram

Started state		• State immediately after program has started running • A callback to StarApplication.started() is made at this state transition • Automatically switches to the Active state after completion
Active state		• Active state • A callback to StarApplication. activated() is made at this state transition
	Full-Active	• Sub-state of the Active state • State where the mobile terminal's resources are made preferentially available
	Semi-Active	• Sub-state of the Active state • State where resources cannot be used exclusively due to factors such as multiple execution
Suspended state		• Paused state • Entered when there is an incoming phone call or when an application's pause method is called
Terminated state		• End state

lel, it offers two levels of support — a "Semi-Active" state where resources are used cooperatively, and a "Full-Active" state where resources can be reserved with priority — and allows applications to find out which of these states they are running in.

The second is that it provides a "Started" state which is entered as soon as the program has started running. This makes it possible to program separate initialization processes that only need to be performed once when a program is run.

The third is that an application is able to issue a transition request to place itself in a "Suspended" state. This ability to pro-actively put software on stand-by when waiting for user input

and the like allows application developers to write software that pays more attention to power savings.

3. WidgetView Functions

1) Functional Overview

To allow multiple mini applications to be run simultaneously, we developed a mini application execution environment called WidgetView. A mini application is a subset of a full application with a smaller program size, a smaller usable memory size, and fewer available functions (**Table 2**). WidgetView is run by pressing a special key in the standby screen. This allows the user to run multiple mini applications (e.g., daily life type applications such as a news application, a weather report application, a stock price application and a map application) at the same time instead of having to run each application individually. Since each mini application has its own drawing region in WidgetView, the user can simultaneously obtain information provided by multiple mini applications.

2) System Configuration

The WidgetView system configuration is shown in **Figure 3**. WidgetView consists of three parts: a MiniVM which simultaneously processes multiple mini applications, a User Interface (UI) Engine which generates a single screen display by applying effects and the like to various elements such as background images and the frames drawn by mini applications, and a WidgetView Controller which connects the MiniVM and UI Engine together. Each mini application draws in an off-screen buffer[*13] (Fig. 3 (1)-(3)), which is then swapped with a front buffer[*14] (Fig. 3 (4)). The UI Engine acquires the mini application's off-screen buffer (Fig. 3 (5)-(8)), and combines it with elements such as a background image to produce a single screen which is displayed to the user (Fig. 3 (9)).

3) WidgetView Display States

WidgetView has three display states (**Figure 4**).

In the multi-view display state, multiple mini applications are run simultaneously and displayed in miniature on a single screen. However, in this state the user is not able to interact with the mini applications. WidgetView is displayed in this state when the user presses a special key in the standby screen.

In the individual display state, only one mini application is running (the other active mini applications are suspended), and only this mini application is displayed on the screen. In this state, the user is able to interact with the mini application.

In the launcher[*15] display state, a list of mini application icons is displayed, and the user is able to start up a mini application by selecting the corresponding icon. In this state, the active mini applications are all suspended.

4) Mini Applications

In a mini application, the available functions are limited depending on the display state. Specifically, the functions that can be used are more restricted in the WidgetView multi-view display state than in the individual display state. For example, the functions for starting up a browser and for making calls are only available in the individual display state.

Table 2 Differences between full applications and mini applications

	Feature	Supported functions
Full application	• Total of application size and data storage size can be up to 2 MB • Only one application can run at a time • All Star APIs available for use	• Functions compatible with mobile FeliCa[*1] • Positional information acquisition functions (GPS function) • HTTP(S) communication functions • Push initiation functions • TCP/UDP communication functions • My Menu registration / deletion functions • Library download functions • Java-Flash linkage functions • Mini application linkage and initiation etc.
Mini application	• Application size limited to 50 kB, data storage size limited to 200 kB • Up to eight applications can run simultaneously in a single screen • Star API is partially available • The memory size that can be used when a mini application is running is about 1/10 the memory size that can be used when a full application is running	• Functions compatible with mobile FeliCa • Positional information acquisition functions (GPS function) • HTTP(S) communication functions • Full application linkage and initiation

*1 FeliCa : A registered trademark of Sony Corp.

*13 **Off-screen buffer**: A memory device or memory region which is not directly shown on the display but is instead used for the temporary storage of data so as to compensate for differences in processing speed and/or data transfer speed when exchanging data between different devices or programs.

*14 **Front buffer**: A memory region that stores the data shown on the display. When a program has finished writing data to the back buffer, this data is display the data shown on the display. When a program has finished writing data to the back buffer, this data is displayed by swapping the front and back buffers.

*15 **Launcher**: A function that displays a list of pre-registered files and programs, and allows them to be started up easily.

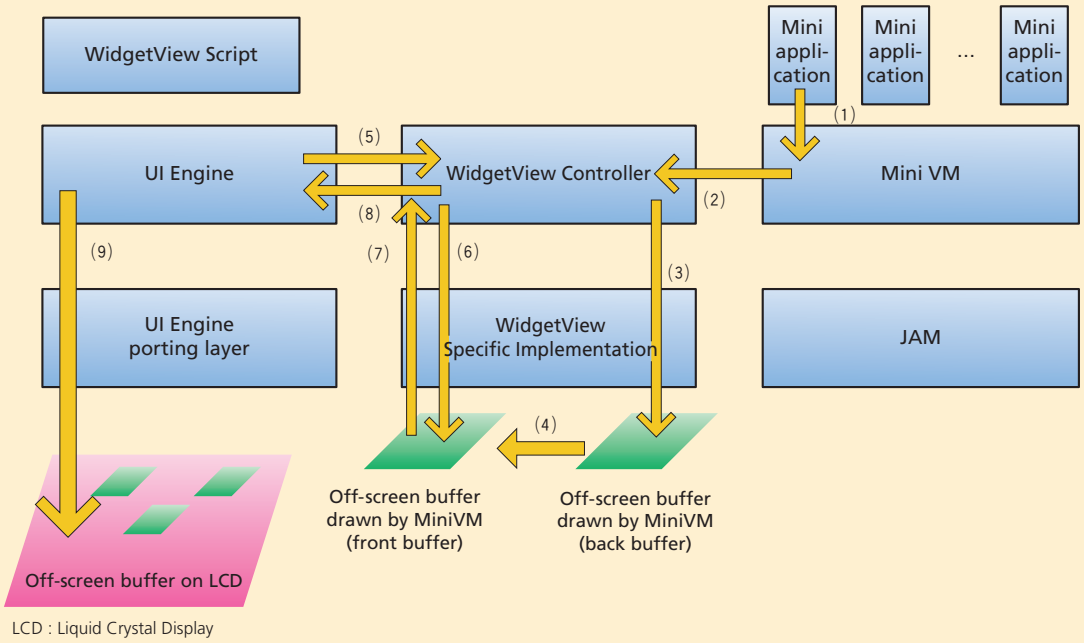

Figure 3 WidgetView system configuration

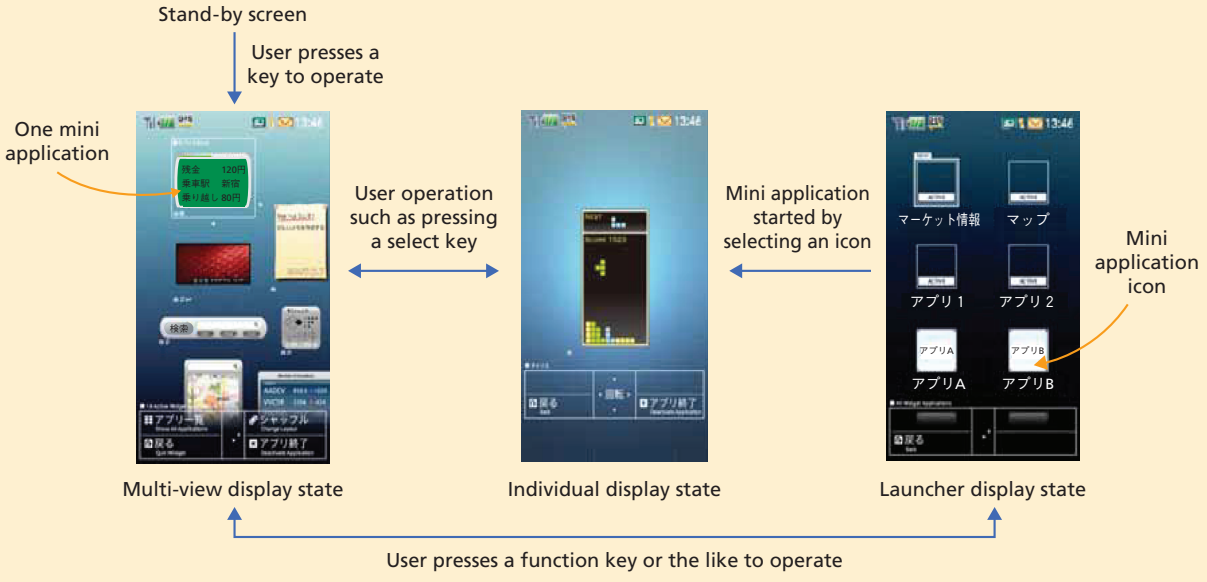

Figure 4 WidgetView display states

4. Java-Flash Linkage Function

1) Functional Overview

We have developed a mechanism that allows Flash files[*16] to be used in i-appli. Specifically, we have developed a mechanism that can play back Flash files in i-appli and allows commands from Flash files to be processed by i-appli. This makes it possible for i-appli to be used as a host application for Flash games, for example, and allows Flash files to be used as an i-appli user interface.

This model is not compatible with in-line playback[*17], so Flash files are played back in the full i-appli drawing region.

2) System Configuration

As shown in **Figure 5**, the system consists of a Flash player that runs Flash files, and a Java VM that runs the i-appli software. The Java VM also operates as a host application[*18] for the Flash player.

3) Flash Files Usage Method

We set up an API for using Flash files in i-appli. Through this API, i-appli is able to control Flash players and to accept and process commands sent back from the Flash players.

The sequence of interaction between i-appli and a Flash player is shown in **Figure 6**. The Flash player is generated by a method call from the i-appli (Fig. 6 (1)). This Flash player is then assigned a Flash file (Fig. 6 (2)), and playback is started by a method call (Fig. 6 (3)). ActionScript can be used to transmit processing requests from the

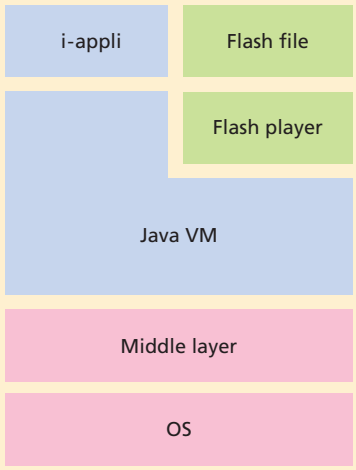

Figure 5 System configuration

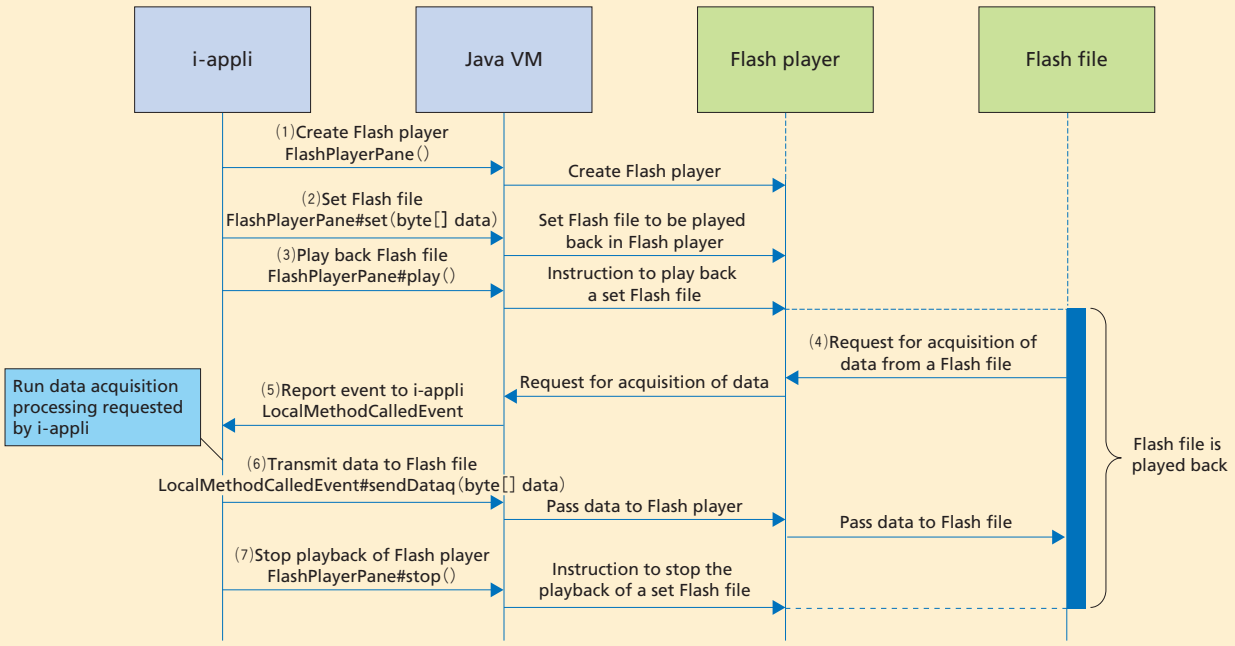

Figure 6 Usage method from i-appli

*16 **Flash file**: A type of content developed by Macromedia Inc. (now Adobe Systems Inc.) which combines audio content with animated vector graphics (file type ".swf").

*17 **In-line playback**: A playback method where content is played back by embedding it in part of another application's drawing region (e.g., a page of HTML).

*18 **Host application**: An application that provides services and/or processes to other applications.

Flash file to the i-appli software (Fig. 6 (4)). Requests from the Flash file are received by i-appli as event notifications (Fig. 6 (5)), and after performing the required processing, i-appli sends the results bask to the Flash file (Fig. 6 (6)). It is also possible for i-appli to stop the playback of the Flash file by means of a method call (Fig. 6 (7)).

Support is also provided for interactive playback whereby Flash files are played back based on user commands, and key events are reported to the Flash file. Function key events are reported to i-appli, allowing i-appli to control the playback of Flash files based on the user's actions.

5. Library Addition Functions

1) Functional Overview

We have developed a mechanism whereby library-formatted native functions and their Java API can be downloaded to and run on mobile terminals. The library addition functions can be broadly divided into class files and native functions[*19] which are referenced from i-appli and are downloaded to the mobile terminal as files (libraries) that are separate from i-appli, and an execution environment where libraries are loaded and executed from i-appli.

Libraries may be written in Java (Java library) or in other languages such as C or C++ (native library). There are two ways in which libraries can be used — one where only Java libraries are used, and one where native libraries are used from a Java library. Native libraries can only be used with i-appli DX.

Using a Java library makes it possible to develop processing code libraries separately from the GUI code. Using a native library makes it possible to:

- Add native functions that are unavailable in the pre-installed Java class library (e.g., adding new code types for a code recognition function together with a recognition engine).
- Implement performance-critical code in a native language.
- Re-use existing native code libraries in i-appli.

2) System Configuration

As shown in **Figure 7**, the system is configured from a JAM for downloading i-appli software and libraries, a Java VM to run the i-appli software and libraries, and a native library framework to load and run native libraries from the Java library.

3) Download Method

Following on from the conventional i-appli download mechanism, the i-appli software and corresponding libraries are downloaded in turn. Specifically, the JAM acquires the library's Application Description File (ADF)[*20] and Security Description File (SDF)[*21] according to the ADF/SDF statements in the acquired i-appli, then acquires the library according to the library ADF/SDF declarations, and finally acquires the i-appli software.

4) Library Usage Method

Java libraries can be loaded and used by i-appli at any time through the standard API of the CLDC. The Java libraries are also able to load native libraries via a new API provided for this purpose. When a native method defined by a class included in the Java library is called, it is possible to execute the native function corresponding to this native method via the native library

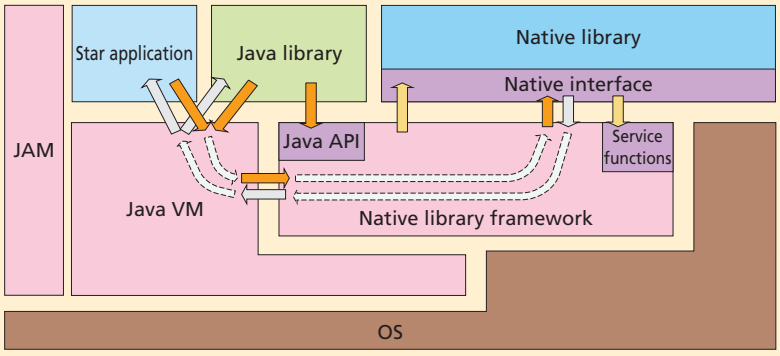

Figure 7 System configuration diagram

*19 **Native function**: A general term for software functions other than i-appli that are installed on the mobile terminal.

*20 **ADF**: A file that contains definitions and information for Star applications and DoJa applications.

*21 **SDF**: A file that contains security-related definitions and information used by trusted applications. A trusted application is one that is allowed to use special functions.

framework (**Figure 8**). The native library framework implements functions necessary for the execution of a native library, such as functions for acquiring and releasing heap momory used by the native library, functions for calling callback functions to perform native library suspension processing when i-appli is suspended or terminated, and functions for accessing logs in order to debug native libraries. By using these functions from the native library, it is possible to terminate the native library processing at suspension opportunities particular to mobile terminals such as when receiving a call, thereby allowing the terminal to be rapidly switched over to handling calls.

6. My Menu Registration Function

1) Functional Overview

We have specified an interface from i-appli to treasure Casket of i-mode service, high Reliability platform for CUStomer (CiRCUS)[*22], and we have developed a function that can transmit the parameters necessary for My Menu registration from i-appli. So far, it has only been possible to perform My Menu registration from an i-mode browser, which meant it was necessary to start up the browser from i-appli in order to perform My Menu registrations. The introduction of this function means that My Menu registrations can now be performed completely by i-appli. For example, users can now directly register in My Menu an i-appli game they are trying out. This added convenience can be expected to promote sales of content.

2) System Configuration

The My Menu registration function consists of a Java VM which hands over parameters passed from i-appli to the mobile terminal, and a mobile terminal module which links these parameters with parameters such as the password input from the mobile terminal, transmits then to CiRCUS, and processes and displays the response from CiRCUS.

3) My Menu Registration Usage Method

We provided a new API for performing My Menu registrations from i-appli. The processing sequence is shown in **Figure 9**. For security reasons, the password input by the user and the response from CiRCUS are not handled directly by i-appli, so some of the functions are allocated to the mobile terminal instead.

This API is used to select the registration/deletion parameters and whether to perform a single or multiple registration. When performing these actions, the Java VM puts the parameters into a

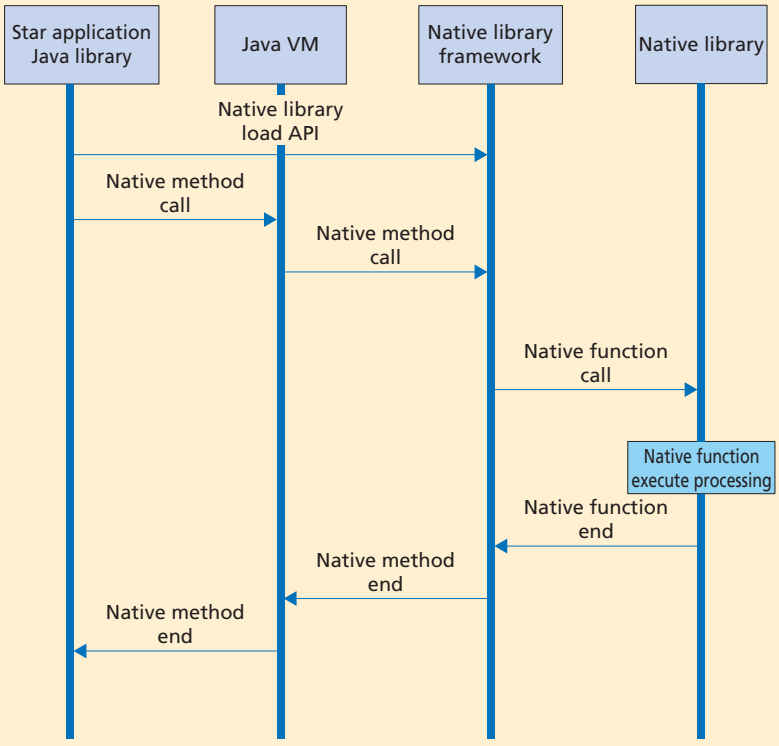

Figure 8 Native library call sequence

*22 **CiRCUS**: A device that serves as an interface between the NTT DOCOMO core network and the Internet, provides i-mode mail, i-mode menu, ordinary Internet access, and other functions.

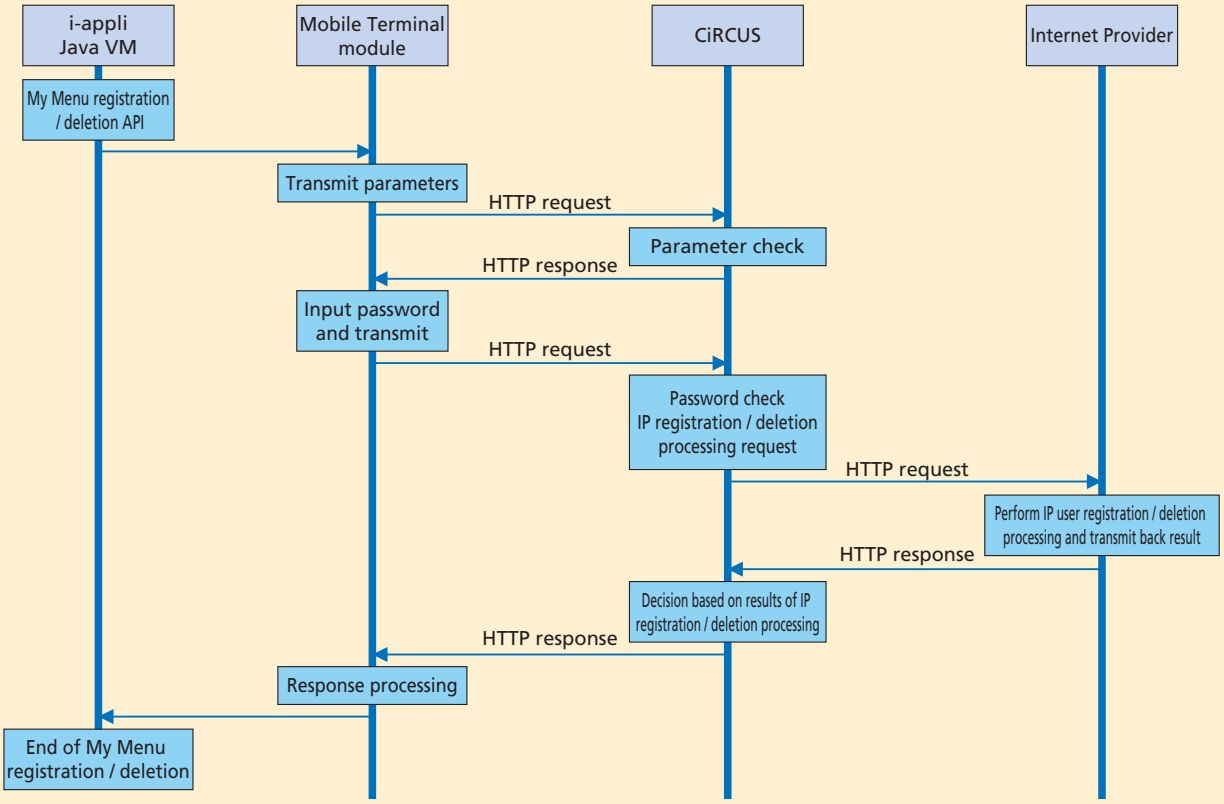

Figure 9 My Menu registration processing sequence

suitable format and hands them over to the mobile terminal module. At this time, the Java VM is suspended. The mobile terminal module transmits the parameters to CiRCUS, and performs actions such as prompting the user to input a password or displaying a confirmation screen depending on the response from CiRCUS. According to the user's actions, the mobile terminal module links together the parameters passed by the API with the password provided by the user and the response from CiRCUS, and transmits them back to CiRCUS. After that, the registration/deletion processing results are processed and reported back to i-appli via the Java VM to inform the user of the success or failure of the operation.

To protect users, only i-appli DX is allowed to use this function because there are cases where registration payments may be issued in order to access non-free sites. Access from ordinary applications is prohibited.

7. Conclusion

In this article, we have described an overview of the newly developed Star platform, and we have described the WidgetView, Java-Flash linkage, library addition and My Menu registration functions.

In the future, we aim to make the Star platform play the role of an application platform, and to pursue further enhancements, performance improvements and added convenience.

REFERENCES

[1] N. Mizuguchi et. al: "i-appli online and i-appli call System for Real-time Communication with Mobile Terminals," NTT DOCOMO Technical Journal, Vol. 10, No. 4, pp. 60-67, Mar. 2009.