VNU Journal of Science, Natural Sciences and Technology 24 (2008) 133-146

An experience in developing embedded software using JNI

Nguyen Thi Thu Trang[1,*], Tran Canh Toan[1], Nguyen Manh Tuan[1], Cao Tuan Dung[1], Takenobu Aoshima[2]

[1]*Hanoi University of Technology, No 1 Dai Co Viet, Hai Ba Trung, Hanoi, Vietnam*
[2]*Matsushita Electric Industrial Co., Ltd, System Engineering Center, Japan*

Received 31 October 2007

Abstract. Embedded software grows more and more rapidly and complicatedly. This paper proposes a new structure and a comprehensible process to develop embedded software in JNI, a new programming framework allowing Java code running in a JVM to call or be called by native application and library written in C/C++. Therefore, programmer can develop application that benefit the simplicity and reusability features of Java and can always reuse legacy code for controlling device effectively written in C/C++ for embedded applications. Some experiences have summarized through two implemented case studies.
Keywords: Embedded software, JNI, RTSJ

1. Introduction

Embedded software has traditionally been thought of as "software on small computers" [1]. In this traditional view, the principal problem is resource limitations: small memory, small data word sizes and relatively slow clocks. Embedded software today is written using low level programming languages such as C or even Assembler to cope with the tight constraints on performance and cost typical of most embedded systems [2].

The C programming language is currently quite popular for small embedded devices. C's main advantage is its flexibility. It is quite easy to interoperate with other language and hardware. However, this advantage often easily turns into disadvantages as project complexity increases [3]. C does not provide enough abstraction to program large and complex embedded systems effectively, even on midsize projects. Moreover, it is difficult to maintain the code and describe complex algorithms using C.

C++ introduces many enhancements to C, most of which support reusability of code. However, C++ is complex and cumbersome. Putting aside its complexity, the C++ approach simply cannot guarantee the long-term support, product development and pool of qualified programmers that will be critical to the successful development of very large real-time systems and development projects that span decades, as NASA and military projects do [3].

Java platform is a better choice on all counts [4-6]. Java is more abstract than C, simpler than C++ and it is supported by a large and growing community of active software developers. The huge amount of available

* Corresponding author.
Email: trangntt-fit@mail.hut.edu.vn

packages, its efficiency, platform independence and ease of use make it the platform of choice for many developers around the world [6]. Although a very extensive API is available with classes encapsulating a large number of peripherals, it is not possible to provide an interface for every device that can be connected to a computer system. Therefore, the Java Native Interface (JNI) was introduced in Java 1.1 [7].

JNI is a programming framework that allows Java code running in the Java Virtual Machine (JVM) to call and be called by native applications and libraries written in other languages, such as C, C++ and assembly. The JNI is used to write native methods to handle situations when an application can not be written entirely in the Java. It is also used to modify an existing application, written in another programming language, to be accessible to Java applications.

This paper exposes some experiences in developing embedded software using JNI technology. We present a structure for embedded software which includes two parts: upper stream written in Java and lower stream written in C. Therefore, we can utilize Java for upper stream with some features such as object-oriented programming, ease to maintenance and high productivity while C is used to control the device effectively. JNI is used to connect these two parts of this structure. If embedded system is a real-time system, RTSJ (Real-Time Specification for Java) is used for upper stream to implement real-time aspects. In this paper, we present our process to develop embedded software using JNI through two case studies. The first is the translation of a movie player source code in C/C++ to Java and the second is implementation of a control program for a movie camera.

The remainder of this paper is organized as follows: In section 2 we give an overview of the JNI and RTSJ technologies. We show a structure for embedded system software in section 3. Section 4 introduces the process we used to develop embedded software. Two case studies are examined in section 5. Section 6 discusses related works. We conclude our results and future work in section 7.

2. Background

This section introduces JNI and RTSJ technology.

2.1. JNI

The Java™ Native Interface (JNI) [7] is a powerful feature of the Java platform. Applications that use the JNI can incorporate native code written in programming languages such as C and C++, as well as code written in the Java programming language. The JNI allows programmers to take advantage of the power of the Java platform, without having to abandon their investments in legacy code. Because the JNI is a part of the Java platform, programmers can address interoperability issues once, and expect their solution to work with all implementations of the Java platform.

As a part of the Java virtual machine implementation, the JNI is a two-way interface that allows Java applications to invoke native code and vice versa which is illustrated in Fig. 1.

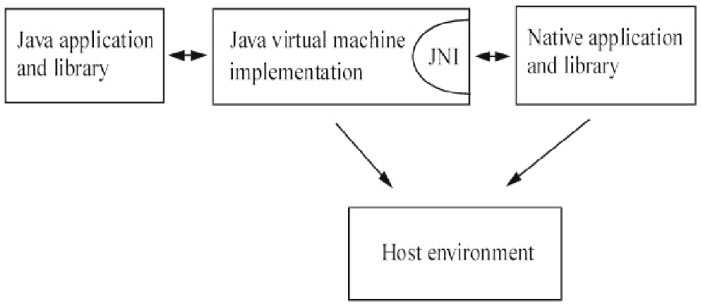

Fig. 1. Role of JNI

The JNI is designed to handle situations where we need to combine Java applications with native code. As a two-way interface, the JNI can support two types of native code: native libraries and native applications.

We can use the JNI to write native methods that allow Java applications to call functions implemented in native libraries. Java applications call native methods in the same way that they call methods implemented in the Java programming language. Behind the scenes, however, native methods are implemented in another language and reside in native libraries.

The JNI supports an invocation interface that allows embedding a Java virtual machine implementation into native applications. Native applications can link with a native library that implements the Java virtual machine, and then use the invocation interface to execute software components written in the Java programming language. For example, a web browser written in C can execute downloaded applets in an embedded Java virtual machine implementation.

2.2. RTSJ

Real-Time Specification for Java (RTSJ) [3] is one of technologies in Project Mackinac - the first commercial implementation by Sun Microsystems of JSR-1. Its purpose is to provide a real-time implementation that meets the stringent needs of real-time developers while continuing to offer all of the other advantages of the Java programming language. RTSJ supports both hard real-time and non-real-time functionality in a single system based on the Java Hotspot platform.

The RTSJ makes several modifications to the Java specification in order to make true real-time processing possible. The RTSJ represents important technological advances in the following six areas:

- Scheduling
- Memory Management
- Synchronization
- Asynchronous Event Mechanism
- Asynchronous Transfer of Control
- Physical Memory Access

The sixth is the one that real-time developers expect and demand in a real-time system. In this paper, we concentrate in two technologies: Memory Management and Physical Memory Access [8].

1) Memory management

The trends in real-time development suggest that in the future, real-time applications will, in fact, become mixed applications, with various threads running in the hard real- time (HRT), soft real-time (SRT), and non-real-time

(NRT) zones, depending on the degree of temporal control needed in each case. In the C/C++ world, memory management is done manually; program logic determines when memory is allocated (malloc) and when it is freed (free). Program logic controls the lifetime of a memory object. In Java, in contrast, memory management is automatic. Program logic still decides when objects are created in memory (new), but there is no way for the programmer to exercise direct control over the freeing of that memory. The Java garbage collector handles this task on behalf of the application.

Developers of non-real-time applications derive a great benefit from the garbage collector, which saves programming time and makes applications more reliable. However, the garbage collector is not compatible with HRT applications because it is difficult to predict when garbage collection will occur and how long it will take to complete.

The RTSJ provides the programmer with an environment in which application logic suffers zero interference from the garbage collector, by introducing a new memory model called scoped memory. In scoped memory, the lifetime of a memory object is determined by program scope, as shown here:

```
run() {
//This is the scope of this thread.
   }
```

The RTSJ allows the application developer to assign everything in the run() method, including calls to other methods, to an application-defined heap. Application-defined heaps are accessed by logic in a particular scope. So when you create a thread, you can give it one of these application- defined heaps. The garbage collector never collects objects in these heaps. Anytime a new() occurs in this scope, the object goes into the application-defined heap, not into the regular Java heap. The entire contents of the application-defined heap goes out of scope as soon as the application completes the run() method. As soon as it goes out of scope, all of the memory objects in the application-defined heap are destroyed instantaneously.

A garbage collection process takes time because the garbage collector has to traverse the entire heap, figure out which objects are still pointed to, and destroy only those objects that are not pointed to. In contrast, there is no time lost when the application-defined heap is cleared. Moreover, the garbage collector halts execution of all its associated threads while garbage collection is going on. Threads that use scoped memory do not suffer interference from the garbage collector.

Another memory model unique to the RTSJ is immortal memory. Like scoped memory, immortal memory is never subject to garbage collection; unlike scoped memory, however, memory objects created in immortal memory remain in memory for the duration of the application, even if there are no references to them.

Both scoped and immortal memories are reserved for HRT threads. The RTSJ provides for SRT threads as well. What makes SRT “soft real-time” is that threads in this zone do take advantage of automatic garbage collection? However, the SRT garbage collector must be able to interact with the application in order to avoid interfering with the predictability of the real- time applications running in the SRT zone. The initial release of Project Mackinac might support SRT.

2) Physical memory access

The PhysicalMemoryManager is available for use by the various physical memory accessor objects to create objects of the correct type that

are bound to areas of physical memory with the appropriate characteristics – or with appropriate accessor behavior. Examples of characteristics that might be specified are: DMA memory, accessors with byte swapping, etc.

The base implementation will provide a PhysicalMemoryManager and a set of PhysicalMemoryTypeFilter classes that correctly identify memory classes that are standard for the (OS, JVM, and processor) platform.

OEMs may provide PhysicalMemoryTypeFilter classes that allow additional characteristics of memory devices to be specified. Memory attributes that are configured may not be compatible with one another. For instance, copy-back cache enable may be incompatible with execute-only. In this case, the implementation of memory filters may detect conflicts and throw a MemoryTypeConflictException, but since filters are not part of the normative RTSJ, this exception is at best advisory.

3. The structure for embedded software

While developing embedded software for a device using pure Java, Java requires much more memory than C or C++ and even with JIT compilers, it runs more slowly. However, Java applications do not depend on platforms. Using Java is intended to reduce software development cost and software distribution cost.

As a result, we propose a structure for embedded software which integrates advantage of C, C++ and Java languages. This structure includes two parts: lower stream written in C and upper stream written in Java. The proposed structure is illustrated in Fig. 2.

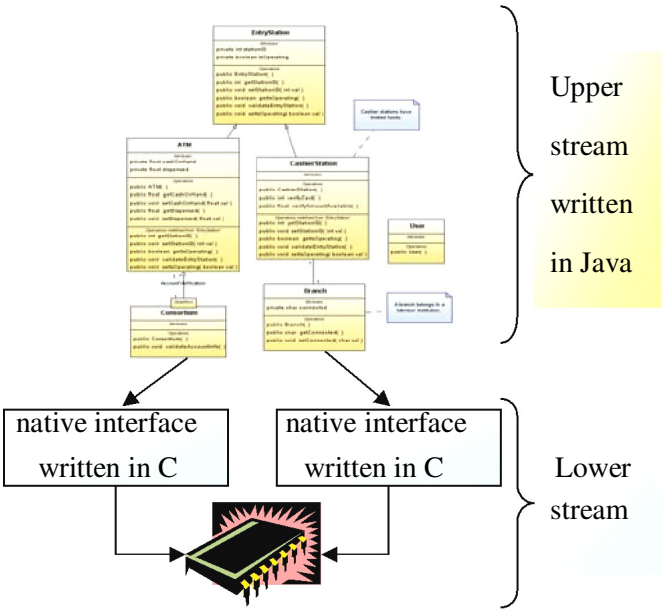

Fig. 2. The structure for embedded software.

In this structure, the lower stream has the role to interact with device or operating systems while upper stream has the role to control the main program process. The lower stream is changed when we change the operating system or device (same type) and may not have any architecture. The upper stream is written in Java so it is platform independence. The architecture of the upper stream is very important, because it affects reusability and easy maintenance.

JNI technology is used to connect these two parts of the structure. If embedded system is a real-time system, RTSJ is used for upper stream – written in Java.

This structure is better than classic embedded software structure where using only C/C++ languages. It has both advantages of C/C++ and Java. The product still keeps the efficiency (performance), in addition ease to maintain, high programmer productivity and platform portable.

Portable in this case does not mean that the embedded applications can run in any OS without changing the source code. In the structure we propose, upper stream is platform portable while lower stream is not. The lower stream written in C/C++ calls the APIs to control the device. When the platform is changed, these APIs are changed too. Therefore, we also have to change the lower stream, but not much. We only have to search the SDK of the device and write the native functions that do the same tasks, called by JNI. The most considerable point here is the ability to separate the dependent and independent part of embedded software and reject the large part of platform independence.

4. The process to develop embedded software

The process to develop embedded software we proposed in this paper follows the above structure for embedded software. First, we analyze existing C or C++ programs which have functions controlling the device. Then we redesign the upper stream in object-oriented technology and construct with native method using JNI.

Fig. 3. The process to develop embedded software.

The proposed process shown in Fig. 3 includes four steps as follows:

4.1. Step 1: Find C/C++ program that has functions controlling the device.

Most of embedded software is still written in C/C++ so this is the store of legacy software source codes which have features controlling the device. Therefore, the best first step to develop an embedded system in Java is searching the C/C++ programs which can control the device. We can search them in the device manufactory home page, or in other developer pages.

4.2. Step 2: Analyze source code and find out the Bottom APIs of this program.

After finding a satisfied control program, the next step is analyzing the source code of this program. The important task of this step is to find out the APIs in the lower layer of this program which access the device or Operating System (OS) - called Bottom APIs. These Bottom APIs are written in native code (C/C++) and we must pack them in functions to do some task for controlling the device or OS. These functions will be called by JNI from Java Program. They are the lower stream in this structure.

To find out the Bottom APIs, we must have the functions called graph by walking through the source code or using Code Analyzer. The Bottom APIs must be functions that are leaves of this graph. This means that the Bottom APIs are called by other functions and it does not call any functions. Moreover, to find which the Bottom APIs in above functions are, we have to know the main task of these functions by searching in development guide, Internet or other resources. Again, the Bottom APIs are functions that access the device or OS.

It is not necessary to use all the Bottom APIs in the lower stream. There are some Bottom APIs which both native language and Java have (such as malloc in C and new in Java). We use them in upper stream, other than call them in native code by JNI.

Another task in this step is to discover the structure of this program. This task is also important, because if we know the structure of this program, we will save our time to redesign this program in Java (in next step). We can detach some concerned modules if the program is so complex.

To grasp the structure of the program, we should wade through the program documents or developer site of this program. If the program is well documented with design specification, this step will be easier. For programs with poor documentation, we have to analyze by ourselves. This task is so difficult, because structure of C program is not easy to understand.

By our experiences, with these programs, we must know the basic structure of type of these programs. For example, web-cam controller programs have three basic modules: controller, grabber, and display. The role of controller is to control the web-cam (turn on, turn off, rotate the web-cam left, right, up, down…). The role of grabber is to get the data from web-cam sensor and pass to display module. And the role of display module is to convert data from grabber to suitable format to display.

Then we should detect the sub module of each basic module. We can use the function called graph of the program to do this task other than walking through the source code.

After completing this step, we have a list of Bottom APIs which are called by JNI from Java program. We also grasp the structure of this program, detach concerned modules.

4.3. Step 3: Redesign the upper stream by Object Oriented Analysis and Design (OOAD).

We now have the list of Bottom APIs which are called by JNI and the structure of C program. In this step, we detach the bottom APIs from source program. The remainder is redesigned in Java by OOAD.

This step defines the new program architecture. We should care about the reusability, portability, extensibility and ease for maintenance of this design. Also, we should focus on lower classes of this program which call the native code by JNI. Because in some case, native functions have to call-back methods in Java.

4.4. Step 4: Construct upper stream and glue with native method using JNI.

Based on above design, the remainder work is implementing the upper stream in Java, and packing the Bottom APIs in some functions to do specific tasks. Then we compile lower stream and generate the shared library depends on operating system.

5. Experiment in two case studies

To illustrate our suggestion, this section introduces two case studies which are implemented successfully in Laboratory of Department of Software Engineering, Faculty of Information Technology, Hanoi University of Technology (FIT-HUT). This is a collaboration research project between Department of Software Engineering, FIT-HUT and Panasonic R&D.

5.1. Overview two case studies

1. Case study 1:

In this case study, we implement a Linux Web-cam Controller program for Logitech Quick-Cam Orbit MP shown in Fig. 4. Logitech Quick Cam Chat webcam is an affordable and complete starter kit for adding live video and audio to online communications.

Fig. 4. Logitech Quick Cam Chat

This device has many advance features, such as:

- Interface: USB 2.0 (USB 1.1 compatible)
- Image Sensor: 1.3 Mega pixel
- Still Image Capture Resolution: 1.3 Mega pixels (up to 4 Mega pixels software enhanced)
- Still Image Capture Format: JPEG
- Pan/Tilt: 189 degree/ 102 degree
- Audio Capture: Built-in microphone
- Compatibility: PC, Mac.

Our implementation named QCamUvc has the following features:

Controlling the Quick Cam (turn on, turn off the device, rotate it left, right, up, down, reset to the center position,…).

- Grabbing the video data from this device.
- Display the data by using SDL library [9].
- Adjust display properties (brightness, frame rate, contract…).
- Save the video and picture files.

2. Case Study 2:

This case study illustrates reusing of source code for embedded system by converting a movie player program from C/C++ to Java and calls the bottom APIs using JNI technology. We choose XAnim [10] as the source program. XAnim (pronounced: eks-'an-im) is a program for playing a wide variety of animation, audio and video formats on UNIX X11 machines. It was written mainly for machines running UNIX (or a UNIX derivatives), but can also be compiled and run on VAX VMS machines (although without audio support). It has also been ported to the Amiga and to W95/NT.

Our implemented Linux Media Player name JNIXAnim supports two file types, one for audio, and one for video:

- WAV files: PCM format

- AVI files: with MSVC (CRAM8 and CRAM16) and MJPEG codec.

5.2. Detail implementation of two case studies

1. Case Study 1:

First, we found the Linux driver called UVC (Usb Video Controller) [11] for this Quick Cam, configured and installed it in Fedora Core 6 Linux OS. Then, we searched the Linux controller program. Unfortunately, Logitech site does not support any program so we have to search in other sites. We have found one called luvcview in the open source site http://developer.berlios.de. This program has the features as we expect so we choose it for implementation.

This program has four main parts: Controller, Grabber, Display and GUI, User Event Handle. Based on proposed approach, we analyzed source code and found the Bottom APIs of this program:

- ioctl: stands for Input/Output ConTroL. ioctl() is an API in standard C library and is enhanced in v4l2 API.
- open: a function to open the device and get the file description for this device.
- close: a function to close the device has file description pass for this function.
- SDL library: stands for Simple Direct Media Layer. SDL library is a cross-platform multimedia library designed to provide low level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D video frame buffer.

After understanding logic and order of Luvcview, we separate Bottom API from source code and the rest was redesigned by OOP using Java. We concentrate about three classes: QCam, Grabber, and Controller. There classes contain JNI method to call the native code. Fig. 5 shows the overall class diagram of QCamUvc.

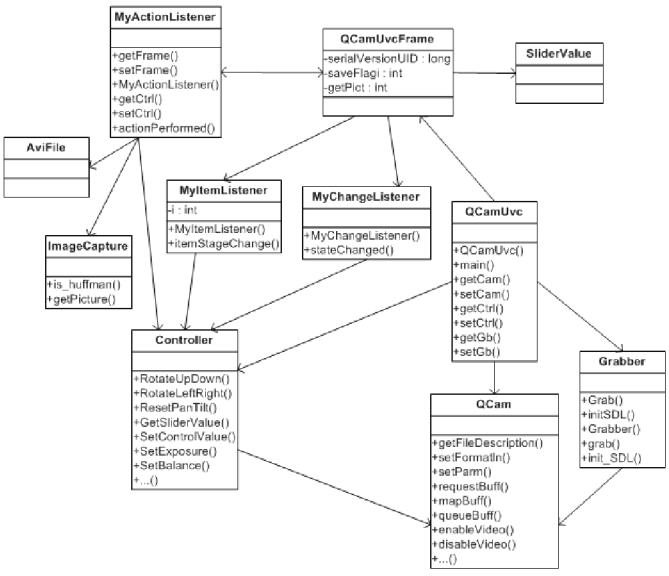

Fig. 5. Class diagram of QCamUvc.

Then we pack the Bottom APIs found in some functions to open/close the device, turn on/off video stream mode, init the display window by SDL, grab the video, decompress

MJPEG data to YUV422 and display YUV422 data by SDL.

2. Case Study 2:

In the beginning, we have suggested to choose MPlayer or Gnash for our implementation. MPlayer is a good movie player program. All features including codec, GUI and documentation are perfect but the source code is too large. There are more than 1000 C files. Gnash is a flash player and run not smoothly. We have found some other Linux media players such as Sinek, FFmpeg. After comparing them, we have chosen XAnim because its size is medium and feature is quite better than others.

In the scope of implementation, we concentrate on five modules: Media files reader, wav player, avi player, audio video synchronizer and codec.

First, we find the Bottom APIs including:

- open, close, ioctl
- write (int fildes, const void *buf, size_t nbyte): The write() function attempt to write nbyte bytes from the buffer pointed to by buf to the file associated with the open file descriptor, fildes.
- X11 library [12]: X11 is window manager for Unix OS. X11 uses a client-server model which is illustrated in Fig. 6. An X server communicates with various X client programs. X11 library brings the developer the APIs for working on X11.

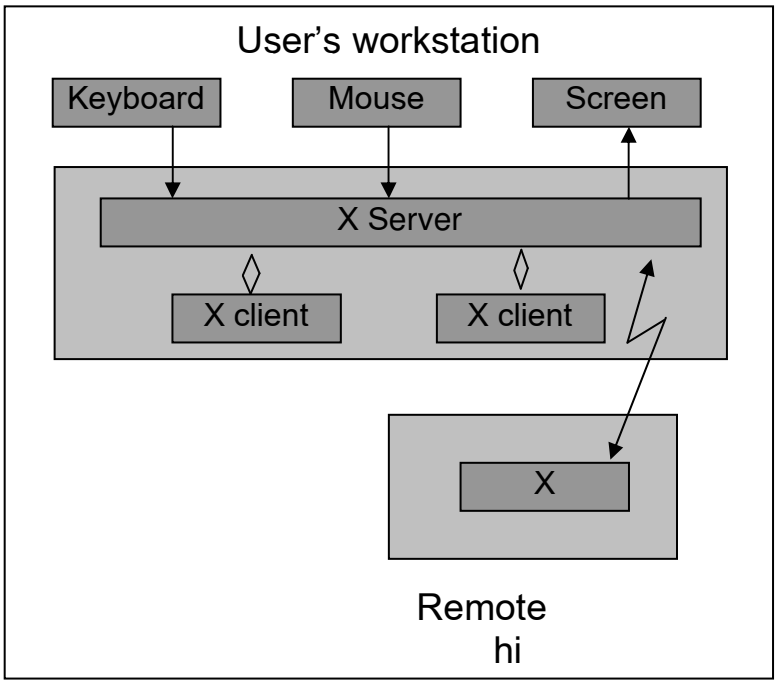

Fig. 6. Client-server model in X11.

Then five modules are detached from the XAnim program. This task took much time because XAnim's structure is complex, and the XAnim author used some techniques that difficult to monitor the main process such as function pointer, or preprocessors. It took our team two months to understand the XAnim structure and detach the modules we care.

Based on five modules we have detached, we redesign the XAnim program with desired features. JNIXAnim – our implementation of XAnim has three important classes: Controller, AVIPlayer, and WAVPlayer which declare the native method. Controller class defines the functions that init, turn on/off, and adjust volume of the sound card. WAVPlayer class defines the functions that write to sound card the audio buffer. AVIPlayer class has a WAVPlayer to play the audio in the video file. It also defines the functions to call the X11 APIs to init the display window, and display the raw video data in RGB format. Media file reader module is switched to upper stream.

There are some difficulties when we construct this case study because of XAnim's structure and APIs. We are not familiar with X11 APIs so it took much time to implement native code to use these APIs. In addition, codec module in XAnim is influenced by many user defined structures. We must restrict this influence by walking through the source code to grasp the meaning of structure element, and just retain the influenced element, to design the data flow for this module.

Another difficulty in the constructing step is the synchronization of the audio and video player as garbage collector in Java is called automatically. We can use memory management of RTSJ to solve this problem so that it does not depend on garbage collector invocation.

3) Some results:

After implementing two case studies, we have made some comparisons between our target programs and source or other programs in some features such as size of source code, used memory, etc. Table 1 and Table 2 is the comparison of source code size for two case studies.

Table 1 shows the comparison in size of source code (in Line Of Codes – LOCs) for some main modules between Luvcview – source device control program written in C and QCamUvc – target program we have developed based on our supposed structure and process. Size of source code of our implementation in case study 1 is much smaller than the source program (2658 LOCs in total).

Table 1. Comparison in size of source code between luvcview and qcamuvc

Module	Luvcview (LOCs)	QCamUvc (LOCs)
Controller	1,669	1,274
Grabber, Decode and Display	1,546	1,444
GUI and Event Handle	1,549	1,114
Other (Save Image, Video, etc)	2,915	1,189
Total	**7,679**	**5,021**

Table 2 shows the comparison in size of source code between XAnim – source movie player program written in C and JNIXAnim – JNI version for XAnim. Size of source code of JNIXAnim is about one-third smaller than the source program as JNIXAnim supports only wav and avi formats.

Table 2. Comparison in size of source code between XANIM and JNIXANIM

Module	XAnim (LOCs)	JNIXAnim (LOCs)
Read Media Files	5,163	2,437
Decoder	1,816	1,624
Display	1,453	200
Controller	1,147	240
GUI and Event Handle	3,451	670
Synchronize	1,104	120
Others	576	115
Total	**14,710**	**5,406**

Table 3 shows the result when comparing about used memory (in MB) between JMF (Java Media Framework) and JNIXAnim – our implemented JNI version. JMF is a Java Library that enables audio, video and other time-based media to be added to Java applications and applets – that means pure Java. Therefore, memory used in JMF is much more than our implementation which uses C source code in lower stream to control the device illustrated in Table 3.

Table 3. Comparison in used memory of between JMF and JNIXANIM

Time	JMF (MB)	JNIXAnim (MB)
Init	22.7	8.4
Start playing	33.4	18.8
After 30 seconds	44.3	26.7

6. Related work

Relating to works of developing embedded software, many researches have been proposed [6, 13-15].

The work of Martin Schoeberl et al [15] affirms that it is possible to develop applications in Java on resource constraint devices. Some definitions of Java for embedded and real-time systems do exist.

The paper [6] considers two frameworks including Java 2D and Java Media Framework (JMF) to control digital cameras. However, when we have tried JMF to implement case study 2, there are some errors in Linux. Furthermore, as Java 2D and JMF are written in only Java, they are not as effective and high performance as using native code to access the devices.

Another approach, Stephan Korsholm and Philippe Jean describe a new way of integrating Java with legacy code –Java Legacy Interface (JLI) [13]. The paper [13] shows in some cases such as (typically smaller) embedded platforms, without room for a full JNI implementation, or with scheduling mechanisms other than threading, JNI cannot be used for integrating Java with legacy code. JLI supports the integration of Java with legacy platforms, and JLI makes it possible to integrate types of platforms where JNI cannot be used for this integration. However, our solution for this problem here is using RTSJ for real-time aspect.

The report [14] proposes some techniques for integrating native code, typically written in C or C++, with Java code running on the PERC virtual machine. This report suggests a number of techniques one can use that can improve JNI

performance without sacrificing clean design. A common pattern in JNI code is to copy a buffer from a C function into a Java-accessible array or data structure.

7. Conclusion and future work

Different lessons can be learned two implemented case studies. JNI allows Java code that runs inside a Java Virtual Machine to interoperate with applications and libraries written in other programming languages. We can reuse effectively legacy code written in languages such as C or C++ and the result programs have both advantages of C/C++ and Java. We have proposed the structure and the process to develop embedded software.

In the structure we have proposed, upper stream is platform portable while lower stream is not. The lower stream written in C/C++ calls the APIs to control the device. When the platform is changed, these APIs are changed too so we have to change the lower stream, but not much. The proposed structure separates the dependent and independent part of embedded software and rejects the large part of platform independence. Therefore, the embedded software is partial platform portable.

In this paper, we have made some comparisons in size of source code between source programs written in C and our implemented programs written in Java and JNI based on the proposed structure and development process of embedded software. Size of source code of target implemented programs is much smaller than source program because of reusability of the upper stream written in Java.

We also compare the used memory between our implementations and other pure Java application. Our implementations obviously use less memory than others because of the efficiency of C source code to control program in lower stream of our structure.

One of the most important tasks in the proposed process is the separation of the Bottom APIs from C/C++ source code. In two case studies, we have presented some solutions to find out the Bottom APIs which are native codes called by JNI.

In the future, we expect to do further optimization of the process that is responsible for translation from native code to the Java application using some technical optimizing performance of JNI. We also continue researching embedded real-time systems with RTSJ technology.

References

[1] Edward A. Lee, "Embedded Software". In *Advances in Computers* (M. Zelkowitz editor), Vol. 56, Academic Press, London, 2002.
[2] Alberto Sangiovanni-Vincentelli, Grant Edmund Martin, "A Vision for Embedded Software". In *Proceedings of CASES 2001,* Atlanta, Georgia, November, 2001.
[3] Project Mackinac, "The Real-Time Java Platform*". A Technical White Paper, Sun Microsystems*, June 2004.
[4] Menno Lindwer, "Java in Embedded Systems". *XOOTIC Magazine*.
[5] Marius Gafen, "Java on Embedded Systems". In *Real-Time Magazine* 98- 1, 1998.
[6] Yves Vandewoude, David Urting, Kristof Pelckmans, Yolande Berbers, "A Java-Interface to Digital Cameras". In *Proceedings of Applied Informatics – AI*, 2002.
[7] Sheng Liang, "The Java Native Interface - Programmer's Guide and Specification". *The Java™ Serie*, Addison-Wesley, 1999.
[8] William S. Beebee, Jr., Martin Rinard, "An Implementation of Scoped Memory for Real-Time Java". *Lecture Notes in Computer Science*, 2001.
[9] http://www.libsdl.org, SDL Library homepage.
[10] http://xanim.polter.net, The XAnim homepage.
[11] http://linux-uvc.berlios.de, UVC driver.
[12] http://x11-basic.sourceforge.net/.
[13] Stephan Korsholm, Philippe Jean, "The Java Legacy". In *Proceedings of the 5th international*

workshop on Java technologies for real-time and embedded systems, Vienna, Austria, 2007, p187.

[14] *Best Practices for Native Code Integration with PERC*, February, 2003.

[15] Martin Schoeberl, "Restrictions of Java for Embedded Real-time Systems". In *Proceedings of the Seventh IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'04)*, May, (2004) 93.

[16] Joost Backus, "Java gets real for embedded SW development". *Real-Time Magazine* 99-3, 1999.