PORTING THE ASTI BLUETOOTH™ PROTOCOL STACK TO THE WIN32® PLATFORM

Janice M. Ballesteros, Mabeth M. Borres, Lucelle C. Botardo,
Anne Margrette Q. Caccam, Bienvenido H. Galang Jr. and Billy S. Pucyutan
Advanced Science and Technology Institute
UP Technopark, Diliman, Quezon City
Email: bluetooth@asti.dost.gov.ph

ABSTRACT

The software engineering process (SEP) involves the following systems engineering activities: requirements analysis, architectural design, detailed design, test design, targeting/porting, and testing. The Department of Science and Technology-Advanced Science and Technology Institute (DOST-ASTI) was able to develop a formal model of the Bluetooth™ Host-side Protocol Stack during the first phases of development in the software engineering process. This paper describes the last few stages of the software engineering process, namely, targeting/porting and testing. It includes the necessary steps to successfully port your application/code to an Operating System (OS), specifically Windows 2000 using the Win32® Application Programming Interface (API).

***Index Terms*—Bluetooth™, Software Engineering Process, Formal Description Techniques, Protocol Stack Development, Porting, Win32® API**

1. INTRODUCTION

Bluetooth™ is an emerging wireless standard aimed to interconnect devices within the personal space. It is projected to be one of the more profitable technologies in the near future with wireless applications driving the technology to its full potential. There are now over 3,000 companies worldwide, comprising the Bluetooth™ Special Interest Group (SIG) taking advantage of this technology. The Philippines is currently lagging behind with its Southeast Asian neighbors who have two to three times more wireless applications developers [1]. A major barrier to entry for local wireless applications developers is the high cost of software, particularly the protocol stack, which costs thousands of dollars. ASTI is addressing this need by developing original systems software, the Bluetooth™ Host-side Protocol Stack to uplift the Philippine Software Industry by giving local developers access to its protocol stack and Bluetooth™ research efforts.

2. OBJECT ORIENTED REAL-TIME TECHNIQUES (OORT) IN THE SEP

"There is no longer any doubt that Object-Oriented Technology (OOT) is the development solution of the 21st century" [2]. An object-oriented approach helps system analysts and designers integrate the notion of software durability into the development process and to take into account software reuse at all levels because of its powerful concepts and mechanisms. However, they are more difficult to design and cannot cope up with the following constraints: management of physical resources, multi-tasking, data and control management, and temporal performances. Rather than invent another software engineering technique, the pragmatic approach is to combine OOT with FDT's, each of them fulfilling a clearly defined need [2].

"The overall objective of OORT is to support an iterative process that covers all aspects of system development without any paradigm shift or discontinuity" [2]. It uses an iterative approach, which is better than the V-life cycle, the most common approach used in the industry.

Figure 1 shows the V-life cycle [2]. As shown in the figure, the activities are carried out sequentially. It follows a highly top-down approach and it does not allow reuse of components that are further defined in the later activities of the development process. It can also provide difficulties in shifting from one phase to the next, and it does not support rapid prototyping. A better approach would be to use an iterative process, which supports software reuse and rapid prototyping as shown in Figure 2 [2].

OORT uses the best-adapted notations at each step of the SEP. It uses OOT in conjunction with FDTs to get the benefit of software reuse, early validation, rapid prototyping and automatic code generation [2]. The whole OORT engineering process was used to develop the ASTI Bluetooth™ Host-side Protocol Stack as discussed in [3,4,5].

3. PORTING THE SDL MODEL

A critical part of the OORT engineering process is *targeting* or *porting* because the system is validated

using a real-time environment. Real-time errors that were not detected previously through simulation can now be corrected. *Porting* is defined as the method of configuring a program/application to run in an Operating System (OS) [6]. Ideally, such a program needs only to be compiled for the OS to which it is being ported. In most cases however, porting requires a few data conversion and adaptation to new system procedures, but this doesn't have to be too tedious such that the program has to be re-developed [6]. The primary goal in porting is on an OS. The porting process avoids writing *makefile*, API or major changes in the pre-developed program; rather, it simply pushes workarounds to single points of change in low-level libraries [6].

Our purpose in porting to an OS is primarily, to validate and verify our application, which is the SDL model of the ASTI Bluetooth™ Protocol Stack.

Generally, porting an existing application to a new platform involves steps from setup to debugging, until functional enhancements on the new platform are added [7].

For this specific project, the team underwent these steps, subdividing them into more detailed tasks or procedures. These include a) integration of external task to the generated code, b) creation of *mailslots*, c) integrating threads and processes, d) defining the Graphical User Interface (GUI), e) driver development and, f) testing and debugging procedures. This is illustrated in Figure 3.

3.1. Steps in the Porting Process

3.1.1. External Task Integration

An external task is a code that can be thought of as a communication manager between any two layers of the stack and between any layer of the stack and another system to pass signals from one to another. It may represent the environment, an SDL block, and an SDL process [8].

As illustrated in Figure 3, the two important environments that the system needs to communicate with are the GUI and the Driver. The GUI is defined by the specific application used. On the other hand, it is the Driver that mediates the reception and sending of signals between the Stack System and the Bluetooth™ kits.

The development of the external tasks required knowledge in some concepts in Win32® Programming. The most critical is the creation of *mailslots,* in addition to the *threads* and *processes*.

Making the Code conformant to the Kernel

In the first attempt at porting, the main goal was to make the available executable codes, consisting of the code generated and the included ones interface with each other in the Win32® environment. In booting the kernel, it is necessary to have a start-up process to initialize the threads that you will be creating for the different processes the executable files will require [9]. This requires a compiler, a loader, and a built kernel.

The features managed by this kernel booting were basic I/O with a forever-wait capability of incoming signals to the memory manager. The Win32® Memory provides an associated address space to each process used in the system. Moreover, portions of address space can be reserved, mapped ("committed" in Win32® terminology), or unmapped ("freed").

3.1.2. The Use of Mailslots

There are different Inter-Process Communication (IPC) mechanisms that Windows® 2000 provides. Two of them are *pipes* and *mailslots*. *Pipes* also have the so-called *named pipes,* which are similar to *mailslots*. *Mailslots* were chosen over *pipes* because they are the one supported by most of the Win32® operating systems, specifically Windows® 2000. Furthermore, the team considered compatibility of components because eventually, the serial driver to be used is only operational in the Windows® 2000 platform.

Mailslots provide one-way IPC capability. A single process can create a *mailslot* and become both a *mailslot* client and server to send and receive messages [10].

The function *CreateMailslot* is used to create a mail slot. It returns a handle with which you can read all queued messages, with the *ReadFile* function. The sender of mail messages uses the *CreateFile* function to open a handle through which it can write mail messages. The sender can then use the *WriteFile* function to write a message to the mail slot. Each call to *WriteFile* function sends one message to the mailslot. After the sender is finished sending messages, it can call the *CloseHandle* function to close the communication. Unlike other kernel objects, which are released when the reference count reaches zero, a mailslot will be deleted when the last handle is

closed and the creating process terminates [11].

Inherent to the current development, there are two environments that the stack needs to communicate to. These are the Graphical User Interface (GUI) and the Bluetooth™ Serial Driver. The whole scenario of verifying the signal passing in the system is mediated by the creation of *mailslots*.

As an overview, the signal-passing loop has been identified in two parts: the Upper Module and the Lower Module, as shown in Figure 2. The communication in the Upper Module is two-way: *GUI-to-SDL* and *SDL-to-GUI*. The signals coming from the GUI are passed to the SDL via the queuing mechanism of the *mailslot*. For this part, the *MAILSLOT3* is utilized. Signals are "written" as they are received (from the GUI) on the said *mailslot* and are "read" as they are sent to the receiver (the SDL). To complete the loop for the signals from the SDL to the GUI, *MAILSLOT4* was utilized using the same idea.

On the Lower Module, the two-way communication can be found in *SDL-to-Driver* and *Driver-to-SDL*. Similar to the Upper Module, *MAILSLOT1* was used in the *SDL-to-Driver* and *MAILSLOT2* was utilized in the *Driver-to-SDL* communication.

3.1.3. Integrating Threads and Processes

In operating systems, a program in execution is called a process. An application written in the Microsoft Windows? platform can consist of more than one process and each process can support more than one *thread*. A *thread* is the basic execution unit that gets time allocation of the CPU time. The executing portion of a process is composed of one or more *threads*. The Microsoft Win32® API supports multitasking, which creates the effect of simultaneous execution of multiple processes and threads. When the last *thread* is finished, the process is terminated. The function *CreateThread* is used to create a new *thread*. One of the arguments of the *CreateThread* function is a user-defined function where execution begins. Using a thread handle can reference newly created threads.

Every *thread* gets its own private stack space. As local variables are created on the stack, there are no race conditions for access to them in *threads*. But global variables are fully sharable amongst all *threads*, and code has to be written to avoid race conditions. When coding, if the first *thread* executing the main function issues a return, the whole process terminates.

3.1.4. Defining the Graphical User Interface (GUI)

Applications typically use a GUI in order to aesthetically illustrate the data that the programmer wishes to present. An end-user is motivated to interact more with a system or application that has a GUI rather than a simple text based console, which provides a command-prompt environment for the user.

Since the system uses a GUI, the real-time component was described using SDL and the interface was implemented as an external task and integrated at code generation time.

SDL signals could also be sent to the GUI from the SDL model. Likewise, the GUI could send an SDL signal to the SDL model. The GUI could also send or receive SDL signals to or from other external tasks.

For the signals that need to be passed to the GUI, some layers need to have APIs to the GUI. These are also external tasks that are written in C/C++ and called by the system. The receiving and passing of signals from the code-generated system to the GUI is also resolved by the creation of *mailslots*.

A process was created for the driver system and the GUI. Running the executable file will call the two *exe* files from the Driver and the GUI.

3.1.5. Device Driver Development

Developing applications over Bluetooth? requires a device driver to enable communication with the Bluetooth? host controller. A device driver is a software component that provides input and output services between peripherals and the Host Operating System. The development of the device driver is discussed in [5].

3.1.6. Testing via Request-Reply Approach on a Local Setup

The last part to porting on a software engineering process is the *testing*. *Testing* includes proper compiling and iterative debugging. For this part the team performed a "One-Pass" to the system. An application to identify the *Local Information* stored on a specific Bluetooth™ device was developed for testing. The setup will ensure that the application developed shall communicate with the driver and vice versa, thus making a complete loop and an assurance to a successful pass to SEP.

To be able to track the signal request and reply behavior, the team made use of a generic serial port

sniffer, which displays the request-reply packets exchanged between the application and the driver.

3.2. Issues

The porting process for that matter also creates some issues that need to be tackled. The creation of the external tasks was implemented using the Visual C++ Integrated Development Environment (IDE). But, unlike the usual Visual C++ environment, there is no way for tracing the code at runtime. Another tool, the Object Geode Design Tracer enables this task. However, the tracing of the executable code and checking of errors at runtime can be possible in either two ways: a) placing the C/C++ function *printf* at the external tasks, or b) placing the SDL function *writeln* in the SDL Model. In this way, traces of the system pass can be kept on track and at the same time the location of runtime errors can be iterative.

4. Results

The ASTI Bluetooth™ Host-side Protocol Stack was successfully targeted/ported and tested to the Win32®® platform. It was demonstrated to communicate with the Ericsson Bluetooth™ Application Toolkit through the Serial Port driver for Windows® 2000 which the grouped has developed [5].

The first steps in targeting and porting to the Operating System is obtaining the C code and executables of the protocol stack by C code generation. C code generation is the translation of the SDL system model files and the generated architecture block files into C and C++ codes. These files are reconfigured to communicate with the Serial Port driver and GUI by *mailslots*. This procedure is characterized by a number of steps, which include graphical application tracking using the ObjectGeode SDL design tracer, configuration and use of the run-time libraries, basic Win32® programming, Bluetooth™ serial driver programming and real-time GUI programming in VC++.

In the testing procedure using the Object Geode design tracer, the graphical view provided by the tracer made the testing of the consistency of the SDL model a lot easier. The MSC's generated could be saved and reloaded to aid in the tracing of the system in every step of the development.

Another useful tool that was used in testing is the serial port sniffer. This tool can display the exchange of packets in the serial port. These packets are compared with the Bluetooth™ specification for consistency. Figure 4 shows a screenshot of the serial port sniffer. The *request* packets are the HCI commands that pass over the serial port while the *answer* packets are the HCI Events that corresponds to these commands.

The inclusion of Run-Time Libraries (RTL's) helped in simplifying the codes because most of the functions are being called mostly from the built-in codes provided in the tool or generated after compiling the system.

Win32® programming is the process that performed a great deal in the signal passing and communication of the different system modules. The use of *mailslots* was inevitable since no direct communication can be made between the SDL and the environment represented by the GUI and between the SDL and the driver to the Bluetooth™ kit. Signals were sent and read from the SDL, driver, and GUI *mailslots*. A complete cycle of requests and responses for the data signals was achieved.

Lastly, on the end-user level, the GUI played a valuable role in presenting the data in an organized manner. The GUI motivates an end-user to interact more with the system or application rather than a simple Windows® console, which provides a command-prompt environment for the user.

Figure 5 shows the final GUI with the application named by the team as the *ASTI BT Kit Identity Checker*. All the steps or processes previously discussed made it possible for the system to display the module specifications of the Bluetooth™ kit, validating that the stack has been properly ported and is actually communicating with the kit. From Figure 5, the following are shown: the *host computer name* running the Windows® 2000 OS, the selected *COM port* which is COM1 for this case and the characteristic specifications of the Bluetooth™ kit in the module report.

The application developed for the purpose of demonstrating the system pass data to and from the ASTI Bluetooth™ Stack was the Bluetooth™ (BT) Kit Identity Checker. This is similar to Ericsson's Bluetooth™ Traffic Utility (BTU), which gives the identity of the Bluetooth™ Kit used by the application. This system displays the Bluetooth™ Device's *bd (Bluetooth™ Device) address* and the *HCI Module* of the said device. Other information such as *HCI Version, HCI Revision, Manufacturer, LMP Version, and LMP Subversion* are also displayed to the end user.

5. CONCLUSION AND RECOMMENDATIONS

The SDL model of the ASTI Bluetooth™ Host-side Protocol Stack can be targeted/ ported, and tested to the Win32® Platform. There are a dozen other platforms where the protocol stack can be ported. Some of these are Linux, Unix, and other Operating Systems as well as a bare system, where there is no OS that runs on the system. The steps described in this paper are also applicable to these platforms since the architecture of these Operating Systems is similar to the architecture of the Win32® Platform.

The transport layer used in this project is the serial port. There are other implementations of this layer, which is faster such as the Universal Serial Bus (USB) that promises a data rate of up to 1.5Mbps compared to the serial ports 115 Kbps [12].

6. REFERENCES

[1] http://itmatters.com.ph/news/news_05202002a.html

[2] Leblanc, Philippe, et.al. ObjectGeode Method Guidelines. Verilog SA, April 1996.

[3] Caccam, M, Dideles, M., et.al. *Bluetooth Host-side Protocol Stack Development Using Formal Design Techniques*. 2nd Philippine National ECE Conference, November 2001.

[4] Ballesteros, J., Borres, M., et.al. Validation and Verification of the Bluetooth Host-side Protocol Stack through Code Generation and Porting. Philippine Journal for ICT. July 2002.

[5] Ballesteros, J., Borres, M., et.al. Developing a Windows® 2000 Serial Driver for Bluetooth. Philippine Journal for ICT. July 2002.

[6] http://searchstorage.techtarget.com/sDefinition/0,,sid5_gci212809,00.html

[7] http://www.support.compaq.com/PortAssist/pa-steps.html

[8] ObjectGEODE SDL C Code Generator Reference Manual for Windows NT and 98.

[9] http://pateam.esiee.fr/debian/img5.htm

[10] http://www.memorymanagement.org/articles/win32.html

[11] http://phoenix.liu.edu/~mdevi/win32/main.htm

[12] http://www.usb.org/faq/ans2.html

7. ILLUSTRATIONS

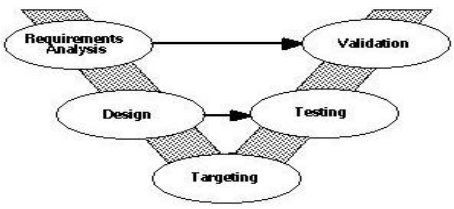

FIGURE 1 V-LIFE CYCLE [2].

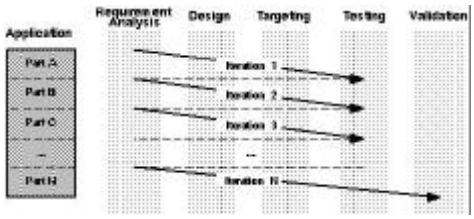

Figure 2 Iterative Approach to SEP [2].

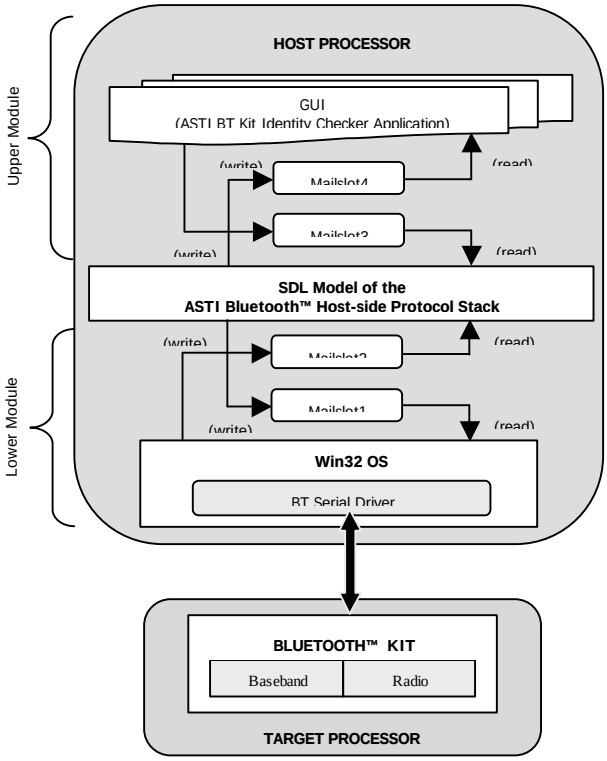

Figure 3 Communication of the SDL model with the environment.

Figure 4 Screenshot of Serial Port Sniffer.

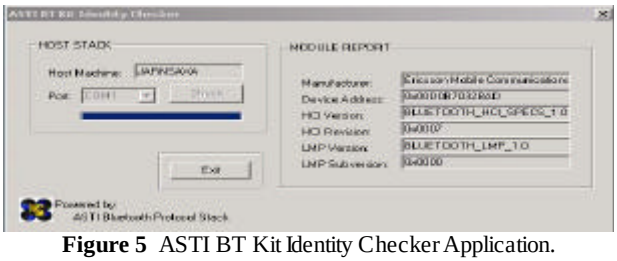

Figure 5 ASTI BT Kit Identity Checker Application.