Device Simulator

S.Pooja Dr.K. Rajanikanth[1]
Department of Information Science & Engineering
M S Ramaiah Institute of Technology, Bangalore, India
samjipooja@gmail.com rajanikanth@msrit.edu

***Abstract*—Developing and testing applications which communicate with devices according to Universal Serial Bus (USB) protocol requires the availability of devices. Simulating the device helps in initiating the development process early and also provides an environment for testing. This paper discusses the details of simulating the device for Windows operating system according to USB protocol.**

***Keyword-- Device* emulator, simulator, Universal Serial Bus.**

I. INTRODUCTION

Universal Serial Bus (USB) is a standard specification for connecting external devices to a host controller. USB can connect computer peripherals such as mice, keyboards, digital cameras, printers, personal media players, flash drives, Network Adapters, and external hard drives.

The communication between user applications and USB device takes place as shown in the Fig 1.The user application accesses the device through the device driver. The device driver delegates the requests and responses between the application and the device. USB Driver (USBD) communicates between a specific device driver and the host controller driver for a specific operating system**.** The host controller driver in turn drives the Host Controller hardware**.** A device-controller along with device firmware communicates with the host controller.

II. NEED FOR SIMULATION

Simulation of a device can be used to replace the real device during the development of drivers for a device or while developing user applications for device. Simulations can be used to test the driver and user applications if the device hardware is not completely developed, not enough number of real devices are available for simultaneous testing of different application modules.

III. SIMULATION

Simulation of a device can be achieved by writing virtual device drivers. When a user application communicates with a device, the requests from the application can be redirected to the virtual driver by means of function pointers [4]. Virtual driver can in turn respond to the requests. Simulated responses can also be sent to a user application by monitoring the serial port. Since all the requests to a device have to go through the port, intercepting the data across the port and responding to those requests would also emulate the responses [6].

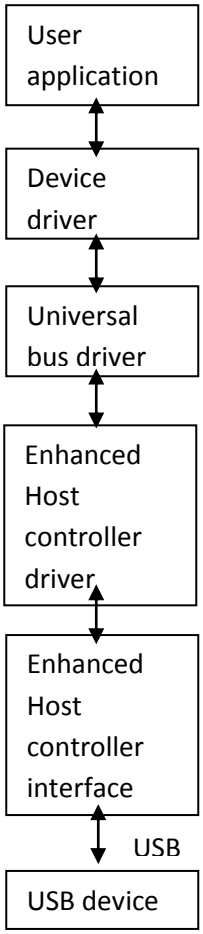

Fig 1. Universal Serial Bus Stack

Another way of achieving simulation is by using Component Object Model (COM) objects provided by Windows driver development kit (Winddk). It also provides simulated Enhanced Host Controller Interface (EHCI) and simulated root hub according to the EHCI specifications for Universal Serial Bus. The EHCI controller simulator communicates with the device simulator as if it was real hardware. The controller simulator intercepts register and direct memory access (DMA) accesses and generates simulated hardware interrupts. COM objects provided by Winddk inform the simulated controller of the connection, and the controller simulator then manipulates its registers to reflect the new connection state and simulates hardware interrupt. The EHCI miniport responds to the interrupt by detecting and reporting the new device as if real hardware had been attached. When the EHCI miniport submits transactions for execution to the simulated controller, the

1 Principal, M S Ramaiah Institute of Technology, Bangalore

controller reads them from the asynchronous and periodic schedules and executes them according to the USB 2.0 and EHCI specifications (host controller specification for the high-speed USB 2.0 functionality)[1].

IV. DEVICE ENUMERATION

USB communications require a host computer with USB support which includes a USB host controller and a root hub. The host controller translates the data to and from the device. The root hub has one or more connectors for attaching devices. The root hub and host controller together detect attached and removed devices. When the system is booted, the root hub polls its ports periodically and informs the host controller about the attached devices. Once the attached device is detected, the host sends a series of requests to the device. The device firmware should maintain different descriptors which describe the device. The device responds to the requests by sending the device descriptors.

A device can support more than one configuration, but only one configuration can be selected at time. Each configuration has one or more interfaces. Each interface corresponds to a specific device function. For example a printer device can also support scanning function. In such cases the device provides interfaces for each device function. An interface is a group of one or more endpoints. An endpoint is a uniquely addressable part of the device. It is a logical entity which is addressable by the device address and endpoint number. A logical connection is formed between the host controller and the device during the communication. Each device will have a control endpoint which is bidirectional and is used for selecting the device configuration and sending the device status and control details. Each interface will have one or more IN and OUT endpoints which are unidirectional and used to move request and responses specific to a device function. IN endpoint moves data into the host controller and OUT endpoint moves data OUT of the host controller. Once the device is detected the host sends a series of requests to the device on endpoint called control endpoint in order to identify the device. The following are the some of the details the device announces: descriptor id, length of the descriptor, the version of the USB specification supported by the device, the class of the device as categorized by USB specification, maximum packet size of the control endpoint, vendor id and product id of the device, serial number of the device, the number of configurations supported by the device, the protocol used, bus power required, number of interfaces supported in each configuration, interface id, number of endpoints supported in each interface, endpoint number, endpoint addresses, and direction and other details[2].

Simulated device object requires a monitoring system which receives the requests addressed to the device. The device also requires a decision making system which interprets the requests. A simulated device object which is associated with the simulated device has to interpret the requests sent by the host and describe itself with necessary details as described in Fig 2 and the descriptors are sent to the host using the control endpoint of the device. Once the device is identified, the host allocates the hardware device id by using the vendor id product id and the class of the device as announced by its descriptors. At this point a minimal device driver has to be written in order to test the com object. This includes testing the control endpoint, bulk IN and bulk OUT endpoints for all type of USB transactions. The Windows plug and play manager detects the category of the device and sends requests which are specific to a particular device category. The devices interpret the class requests and responses according to the format IEEE 1284 specifications for peripherals. Once the device identification is successful, the Windows plug and play manager creates two device nodes for the detected device which appear in the list of device manager. Plug and play manager goes further to search a driver and assigns to the device. After the device has been included in the device list of the Windows device manager, the functions of the device can be exposed.

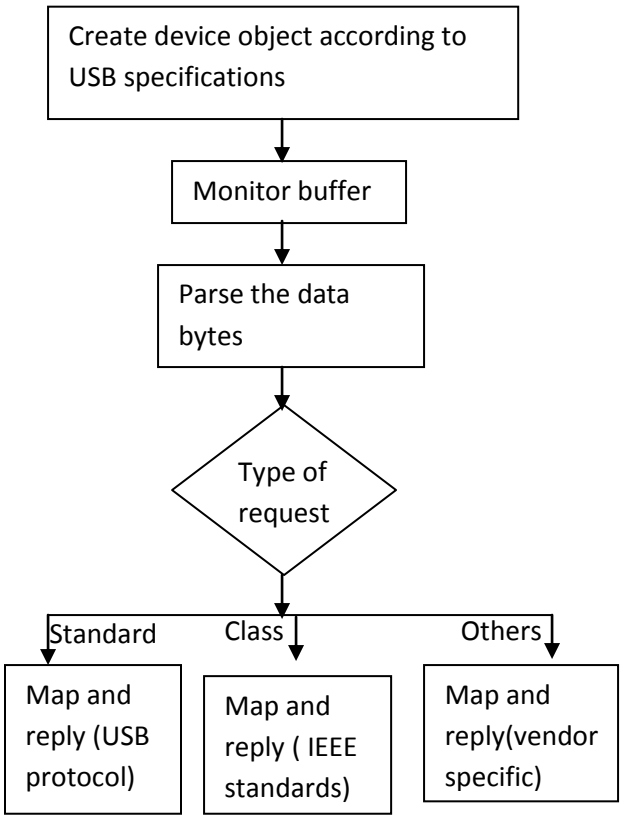

Fig. 2 Handling device requests

V. MAPPING DEVICE AND DRIVER

First device node created in the device manager is assigned the default class driver. Windows operating system provides a generic driver for all the category of devices. For example the default driver provided for printer devices is usbprint.sys which provides printing support, a mass storage device is assigned usbstor.sys The hardware id value returned by the plug and play manager is of the form USB\vidxxx&pidxxx consisting of vendor and product details.

The default class driver assigned in turn sends class specific requests to the device and creates the second device node which is assigned the vendor specific device driver. The device responds to the series of class requests for different attributes as mentioned in the IEEE standards [5]. The device also responds with the compatible ids along with the vendor id and product id. The second device node is assigned the hardware id of the form enumerator\vidxxx&pidxxx where enumerator refers to the default class driver.

The driver installation files also mention the vendor id and product id and compatible id. The plug and play manager searches for driver installation files with the matching vendor id and product id and class Globally Unique Identifier (GUID). If a match using vendor id and product id is not found, the plug and play manager searches for a compatible driver based on the compatible id announced by the device. In that case the device works with a driver of probably other model yet has almost the same functionality. The installed driver is copied to the system registry and registered in the system registry.

VI. USB TRANSACTIONS

Each USB transfer consists of one or more transactions that can carry data to or from an endpoint. A USB 2.0 transaction begins when the host sends a token packet on the bus. The token packet contains the target endpoint number and direction. An IN token packet requests a data packet from the endpoint. An OUT token packet precedes a data packet from the host. In addition to data, each data packet contains error-checking bits and a Packet ID (PID) with a data-sequencing value. Many transactions also have a handshake packet where the receiver of the data reports success or failure of the transaction At the device, transferring data typically requires either placing data to send in an endpoint’s transmit buffer or retrieving received data from an endpoint’s receive buffer, and on completing a transaction, ensuring that the endpoint is ready for another transaction The host schedules the transfers on the bus.

VII. SENDING REQUESTS TO DRIVER

The user applications communicate with the device through the device handle returned by the driver The Win32 subsystem converts the user request into a native system service call. The system service dispatcher traps into kernel mode and into the I/O Manager. The I/O Manager allocates a data structure known as an I/O Request Packet (IRP). The IRP is filled with necessary information, including a code which identifies the type of I/O request. The I/O Manager performs some validation of the arguments passed with the request. This includes verifying the file handle, checking access rights to the file object, ensuring that the device supports the requested function, and validating the user buffer addresses. If the device requests a buffered I/O (BIO) operation, the I/O Manager allocates a nonpaged pool buffer, and for a write request, copies data from user space into the system buffer. If the device requests direct I/O (DIO), the user's buffer is locked down and a list of page descriptors is built for use by the driver. The I/O Manager invokes the appropriate driver routine.

VIII. DELEGATING REQUESTS

The driver defines request codes for each type of input and output requests to be sent to the device. These requests codes are known as Input and Output Control Codes (IOCTL). The driver defines the requests codes and delegates these codes to the device. The host controller addresses these requests codes the control endpoint of the device. The driver maintains a map of requests codes and the corresponding entry points to functions of all supported I/O function request. The I/O Manager uses the function code of the I/O request to index into this table and invoke the appropriate driver function. Driver also validates the requests sent by the user application by matching with the input and output requests codes defined. Driver can inform the user application in case of unsupported requests codes. The device driver can detect device-specific limitations unknown to the I/O Manager. For example, a user might request a data transfer size in violation of the capabilities of a specific device (or driver). The simulated responses which have to be sent to a user application can be responded at this level by sending the expected responses back to the user application. If the request can be handled without device activity (e.g., reading zero bytes), the Diver routine simply completes the request and sends the IRP back to the I/O Manager, marked as complete. The requests can be responded by the driver itself by redirecting to other routines. If COM object provided by the Winddk is used to simulation, all the requests are responded by the simulated device object across the USB port and the device driver requests for the controller and handles interrupts as given in further sections. When the requests are sent to device, the IRPs are maintained in queue which is shared by the driver as well as the

I/O manager as pending. The I/O Manager is instructed to queue a call to the driver's Start I/O routine as soon as the device is free.

IX. REQUESTING CONTROLLER

The driver manages the data transfer operations. When any Driver routine requests that the I/O Manager to start a device, the I/O Manager first checks to see whether a device is already busy. The I/O Manager detects this condition by checking whether or not there is an IRP outstanding for the device (i.e., still marked as pending). If so, it queues the request to start the device. Otherwise, it checks the IRP function (read, write, etc.) and performs any setup work specific to that type of operation. If the device is part of a multifunction controller, it requests exclusive ownership of the controller hardware. If the operation requires dynamic memory access (DMA), an Adapter Control routine requests exclusive ownership of the appropriate DMA channel hardware. It returns control to the I/O Manager and awaits a device interrupt.

X. HANDLING INTERRUPTS

When an interrupt occurs, the kernel's interrupt dispatcher calls the driver's Interrupt Service Routine. The ISR would typically perform the following steps: It Check to see if the interrupt was expected. It dismisses the hardware device interrupt. If programmed I/O was in progress and the total data transfer was still incomplete, the ISR would start another byte or word of data and await another interrupt. If DMA was in progress and more data remained to be transferred, a deferred procedure call (DPC) would be scheduled to set up the DMA hardware for the next chunk of data. If an error occurred or the data transfer was complete, a DPC would be queued to perform post processing at a lower IRQL.

XI. POSTPROCESSING BY DRIVER

Once the I/O request is completed, the controller resources held are released by the driver. The final size of I/O transfer and status are updated. In case of errors the driver records the error and retries the I/O request. In case more data has to be transferred, DMA hardware is setup and driver waits for further interrupts. Once the driver's DPC marks an IRP as complete, the I/O Manager performs cleanup operations. If I/O was a buffered I/O write operation, the I/O Manager releases the nonpaged pool buffer. If I/O was a direct I/O operation, it unlocks the user's buffer pages. It queues a request to the original requesting thread for a kernel-mode asynchronous procedure call (APC). If this was a buffer I/O read operation, the APC copies the contents of the nonpaged pool buffer into the caller's original user-space buffer. It then frees the system buffer. If the original request was for a Win32 overlapped operation, the APC routine sets the associated event and/or file object into the signaled state. If the original request included a completion routine, the APC schedules a user-mode APC to execute the completion routine.

XII. TESTING

Creation of device object can be verified in the Computer Management Console. The hardware id assigned by the plug and play manager can also be checked. The class of the device according to USB specifications is also listed in the Computer Management Console. Also the compatible ids of the device can also be checked.

The states of a USB device according to the specifications can be maintained with a set of values and the corresponding responses to be replied to the requests sent from the host. Testing would involve changing the values and checking the responses sent for a particular set of values.

XII. CONCLUSIONS

Simulation of device helps in development of applications which communicate with the device by replacing the real device according to USB protocol. These ideas have been implemented and tested for a hypothetical new type of Printer device and it was found to be a very useful approach to the development of drivers for new devices. However, it does not support the testing of issues related to device power management and monitoring the traffic performance of the device when multiple devices are connected on the bus as in real case. Eve with these limitations, Simulation can be a powerful option to speed up the development work!

ACKNOWLEDGMENT

The first author would like to express her sincere gratitude to Dr. K. Rajanikanth for his motivation and guidance. Also, she would like to express her sincere thanks to Dr. M. Aswatha Kumar, Head of the Department, Information Science and Engineering, M.S. Ramaiah Institute of Technology for providing an environment conducive for research. Also thanks to all the staff of department for their continuous support.

REFERENCES

[1] *Universal Serial Bus Specification*, Revision 3.0, April 2008.

[2] usbnutshell [Online] Available: http://www.beyondlogic.org/usbnutshell.

[3] Jan Axelson, *USB Complete*, 4th ed, Lakeview Research, 2009.

[4] Art Baker, Jerry Lozano, *The Windows Device Driver Book*, 2nd ed, Pearson Education, 2005.

[5] Mark E. Russinovich, *Microsoft Windows Internals*, 4th ed, Microsoft press, 2004.

[6] William Boswell, *Inside Windows 2000 sever*, 1st ed, New Riders, 1999.