# A Bare PC Mass Storage USB Driver

Ramesh K. Karne, Songjie Liang, Alexander L. Wijesinha, and Patrick Appiah-Kubi

Computer and Information Sciences, Towson University, Towson, MD 21252, USA.

**Abstract**

Today's device drivers are dependent on a given operating system, kernel, or an embedded system platform that provides a higher level of abstraction for its use. We present a USB mass storage device driver that does not depend on any operating system, or kernel, or embedded application. The device driver runs on a bare PC and directly interfaces with an application written in C/C++. The application programmer controls and manages all driver facilities. We describe a step-by-step approach to design the driver and provide code snippets of its key elements that can be used to build a device driver that runs on a bare PC, or a similar computing device. The bare PC API can be used to build a file management system that is independent of standard file system specifications. A bare device driver inherently provides simplicity, total control, flexibility, portability, platform independence, and better performance compared to a standard device driver. The performance measurements for read and write operations show a performance gain over a Linux system with significant improvements in stress tests. The experiments suggest that large transfers are optimized for groups consisting of a particular number of sectors. Bare device drivers can be used for building bare applications or as a foundation for building other types of USB drivers that can run directly on a variety of devices with no system support.

***Key Words: Bare Machine Computing, USB, Bare PC Driver, Mass Storage USB, Device Drivers***

## 1 Introduction

Mass storage USB device drivers can be used with many applications. However, such drivers are typically bundled with some type of operating system (OS) or kernel. While operating systems have

useful features, they come with additional overhead and are difficult to secure. Bare PC applications run on the hardware without the support of any form of operating system (OS) or kernel. These applications perform better than, and are easier to secure than OS-based systems. A bare PC mass storage USB device driver is suitable for any desktop or laptop PC with an Intel x86 CPU. It can thus be used with a variety of applications. This paper describes the development of a bare PC USB mass storage driver from the ground up, and also presents performance results using the driver. The novel approach for developing the bare mass storage driver presented here can be used to construct any bare USB driver. Moreover, a variety of bare PC applications including Web and email servers (and clients), VoIP soft-phones, and server mini-clusters have been previously developed. Since these applications do not use any local mass storage, they could also benefit from the bare PC mass storage USB driver.

Developing device drivers that can run on a bare PC poses numerous challenges since device drivers are usually written based on a given OS and platform. The approach used to develop a bare mass storage USB driver that is described in this paper together with the details and results provided would help researchers and developers to understand bare USB driver internals, build such drivers, and/or investigate the characteristics of bare drivers. In particular, it demonstrates the feasibility of writing bare device drivers that are independent of any OS, and that can run directly on bare hardware.

In order to write a USB mass storage driver, a systems programmer has to master many skills. The know-how to construct a device driver is typically scattered in different USB standards documents, online sources and other references. In some cases, a programmer's reference for a driver may not be available, and in other cases it may be necessary to resolve unclear technical issues by consulting with the architects who developed the standards. Often, there is a big semantic-gap between an architectural reference and its implementation.

Given this situation, the development of a bare USB mass storage device driver that is independent of any OS or environment is even more difficult. One possible approach is to take an existing Linux driver and transform it to run on a bare PC. However, this is a prohibitive task since such drivers are platform-dependent and closely coupled with the host OS; moreover, they use environment variables and several system-related header files. Some drivers that are supposed to be "bare metal" are actually dependent on some underlying lean version of Linux, or DOS interrupts such as Interrupt 21H. While USB drivers that are built for some controller applications [8] provide details on a USB protocol implementation, they do not include the internal details needed to implement a bare PC driver.

This paper describes the development of a bare PC USB mass storage driver from the ground up, and also presents performance results using the driver. The rest of the paper is organized as follows. Section 2 discusses related work, Section 3 provides the necessary details for building a bare PC driver and its implementation, Section 4 gives performance measurements, and Section 5 contains the conclusion.

## 2 Related Work

We implemented a USB driver on a bare PC with no kernel or OS running on the machine. Bare PC applications use the Bare Machine Computing (BMC) or dispersed OS computing paradigm [11], wherein self-supporting applications run on a bare PC. That is, there is no operating system (OS) or centralized kernel running in the machine. Instead, the application is written in C/C++ and runs as an application object (AO) by using its own interfaces to the hardware [12], and device drivers. While the BMC paradigm resembles approaches that reduce OS overhead and/or use lean kernels such as Exokernel [5], IO-Lite [15], Palacios and Kitten [13], there are significant differences such as the absence in BMC of centralized code that manages system resources. We have built complex BMC applications such as Web servers [7, 18] and Web-based email servers [1].

There has been considerable interest recently to use flash memory for mass storage. For example, the Umbrella file system [6] is a heterogeneous file system that uses a hard disk and a flash drive. This file system also illustrates how to integrate two different types of storage devices. Some researchers are also interested in adding cache systems at a driver level to gain performance improvements [3]. The design and implementation of a FAT32 file system [14] for high performance clusters is described in [4]. In future, flash drives are likely to become more prevalent in mass storage systems due to the decrease in the cost of flash drives and the increase in their capacity. While information relevant to building a USB driver is provided in [2] and in some online sources, there is no previous work addressing the problem of building a driver without using any OS or kernel or system libraries. Although citation [8] demonstrates how to write low-level driver code, it does not provide the details necessary to build a bare PC mass storage device driver. As per our knowledge, there is currently no bare PC file system that uses a bare PC USB mass storage driver. The bare PC mass storage driver that is the focus of this paper will serve as a foundation for building bare PC file systems (or bare file systems in general) in the future.

## 3 Building a Bare PC USB Driver

### 3.1 Overview

The general architectural overview of a USB driver in a typical OS environment is shown in Fig. 1. A given application interacts with the OS when requesting file I/O via interfaces such as open(), read() and write(). The OS in turn calls a device driver that communicates to the underlying host controller through its API to process the above user interfaces. The host controller issues low-level USB commands to a USB device to perform mass storage commands. Some of the mass storage commands involve SCSI (small computer system interface) commands that are encapsulated within a USB command message block. A USB device also has a unique device address used for communicating with the driver. In

addition, a USB device consists of several descriptors that indicate the characteristics and attributes of a given USB.

This paper focuses on designing a USB device driver that provides direct communication between an application and a USB device through the host controller. Such an approach requires understanding the given USB descriptors, detecting plug-in and plug-out of a USB device, resetting the device, and performing mass storage operations. An application programmer is in direct control of the above interfaces to a USB device through the device driver. Thus, a bare PC device driver is quite different from a conventional OS-based driver, and it does not depend on any other middleware or other environments to perform mass storage operations.

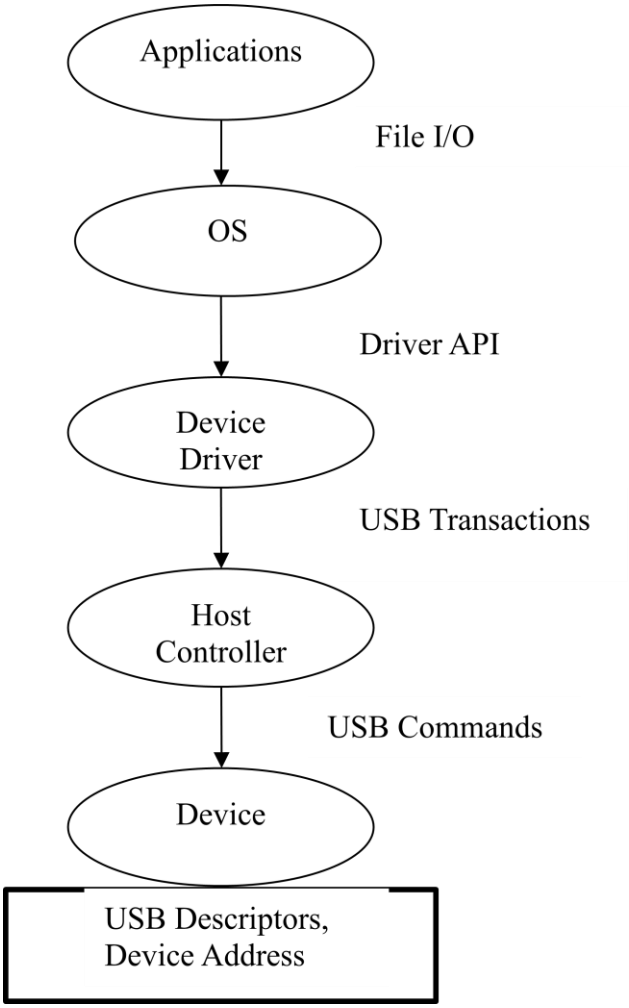


Figure 1. Architectural Overview

Fig. 2 shows the overall process flow to construct a bare PC driver for a mass storage USB device. The following sub-sections describe the construction process and its key elements, and include the relevant design and internal details.

### 3.2 Base Address (BA)

(1) For a given Intel x86 based PC, the Device ID is **0x24cd** and the Vendor ID is **0x8086** [10]. Using these parameters and PCI BIOS interrupt **0x1a** [16], we can obtain the bus number in the BH register, the device number in the BL register (upper 5 bits), the function number in the BL register

(bottom 3 bits), and the return code in AH. The sample assembly code is shown in Fig. 3. In our system, the device number is **29** and function number is **7** with a bus number of **0**.

(2) Using the bus number, device number and function number, we issue another BIOS interrupt to get the vendor ID and device ID, using read configuration DWORD [16]. This step validates the device's vendor and device ID.

(3) Using the bus number, device number and function number, we also issue the write configuration WORD command [16] to turn on bus enable and memory space enable bits in the PCI command register [10].

(4) Using the bus number, device number and function number, we next issue a read configuration DWORD command to read the configuration memory base address register [10] with PCI interrupt **0x1a** [16]. The ECX register gets the base address after this operation.

(5) The base address (BA) is stored in real memory and can be used later by the driver code. The base address obtained in our system is **0xffa00800**. Notice that this is a memory mapped I/O address for the EHCI USB Controller [10].

(6) All of the above operations have to be executed in real mode of the Intel x86 processor, since protected mode does not support low-level BIOS interrupts.

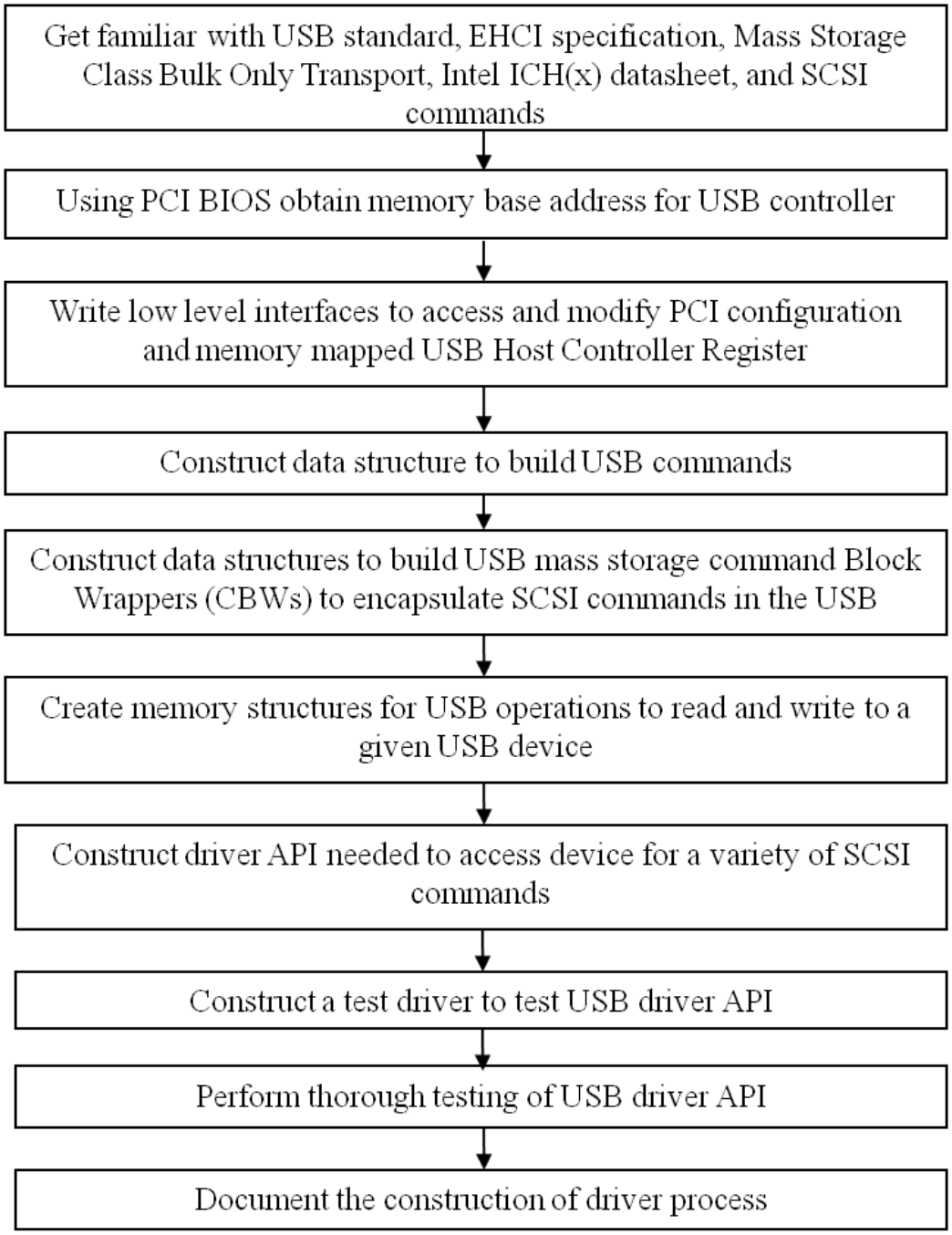


Figure 2. Process Flow

**Reading device no, function no and bus no (find PCI device)**

```
mov ah, 0b1h
mov al, 02h
mov cx, 24cdh
mov dx, 8086h
mov si, 0
int 1ah
```

**Reading vendor id and device id (read config. dword)**

```
mov ah, 0b1h
mov al, 0ah
mov ebx, edx
mov di, 0h
int 1ah
```

**Write to command register (write config. word)**

```
mov ah, 0b1h
mov al, 0ch
mov ebx, edx
mov di, 04h
mov cx, 06h
int 1ah
```

**Read from 10h base address register, ecx has the base address (read config. dword)**

```
mov ah, 0b1h
mov al, 0ah
mov ebx, edx
mov di, 10h
int 1ah
```

Figure 3. Base Address Code

### 3.3 USB Initialization (init())

The BA obtained above plays a key role in the operation of an EHCI [9] controller. Two basic interfaces, readControllerReg32() and writeControllerReg32(), were written to read and write EHCI registers as shown in Fig. 3. These interfaces are used to initialize the USB controller. There are two types of EHCI registers in the controller [9]. The capability registers describe a variety of characteristics of the USB device as illustrated in [9]. The CAPLENGTH register (which is the 1st byte) value indicates the size of capability registers. For example, in our system it is **0x20**, which can be used as an offset to

address operational register space. The CAPLENGTH register can be read using readControllerReg32(0). The HCSPARAMS register shows the structural parameters of the USB host controller (HC). This register can be read with offset of 4. Some of the key parameters in this register include: N_PORTS (number of ports in the PC), N_PCC (number of ports for companion control) and debug port number. The extended capability parameters can be read with an offset of 8. Some of the key features in this register include: EHCI extension capabilities, frame list flag, 32 or 64 bit addressing capabilities and asynchronous schedule capabilities. These parameters can be read and stored in memory as per the needs of a given application.

The operational registers can be read in a similar manner to reading the capability registers. However, they need an offset of **0x20** (CAPLENGTH) in addition to a register offset. This offset is also referred to as Operational Base in Fig.4. For example, PERIODICLISTBASE register [9] has an offset of **0x14,** and thus an offset of **0x34** should be used to access this register. Similarly, other operational registers can be read or written using the read and write API as shown in Fig. 4.

Fig.4 also shows some other initializations for the USB controller. First, initialize the periodic base register with its memory address, reset the frame index register, set the configuration flag, set the command register with 8 frames per millisecond, and set the run bit. A detailed description of these registers is given in [9]. The mass storage device requires an asynchronous schedule as illustrated in [9].

### 3.4 USB Device Initialization and Operation

The USB 2.0 specification document [17] presents comprehensive information on building hardware and software related to any USB device. Additionally, knowledge of USB commands that are relevant to the mass storage domain bare application is also needed. The mass storage driver partial API is illustrated using Fig. 5 throughout this paper. An application programmer only needs this API to

construct complex file management systems, whereas the device driver developer has to use USB commands at the byte or bit level to implement these calls. The crucial commands for the mass storage application were taken from [17]. Most of the commands that are essential to mass storage control and operation of a USB device were designed and implemented.

```
//Read Register
int UsbObj::readControllerReg32(int offset)
{
   ptr = (int*) (BaseAddress  + offset);
   return *ptr;
 };

//Write Register
void UsbObj::writeControllerReg32(int offset, int value)
{
   ptr = (int*) (BaseAddress + offset);
   *ptr = value;
};

//Periodic base
retcode = usbo.writeControllerReg32(usbo.OperationalBase+20, PT_BASE_ADDRregister);

//Frame Index Register
retcode = usbo.writeControllerReg32(usbo.OperationalBase+12, 0x00);

//Config. Flag
retcode = usbo.writeControllerReg32(usbo.OperationalBase+64, 0x00000001);

//RUN command register  8 micro frames per milliseconds, run bit on
 retcode = usbo.writeControllerReg32(usbo.OperationalBase+0, 0x00080001);
```

Figure 4. Init Method

When a USB is plugged in, it must be detected by the bare PC application and an appropriate initialization must be done before it can perform any operations. The init() process described in section 3.3 above only addresses the host controller or EHCI controller in the PC. The init() method has to be run once to collect capability parameters and initialize the operational registers as described before. A given PC has "**n**" ports (N_PORTS), and a USB can be plugged in or removed at any time by a user.

However, every time it is plugged in or removed, it must be detected by the driver (CheckUSBPluggedIn()), and appropriate actions must be taken to enable the device to provide the necessary functionality.

We describe the chronological order of events and the actions that need to be implemented by the bare PC USB mass storage driver using the state transition diagram (STD) in Fig.6.

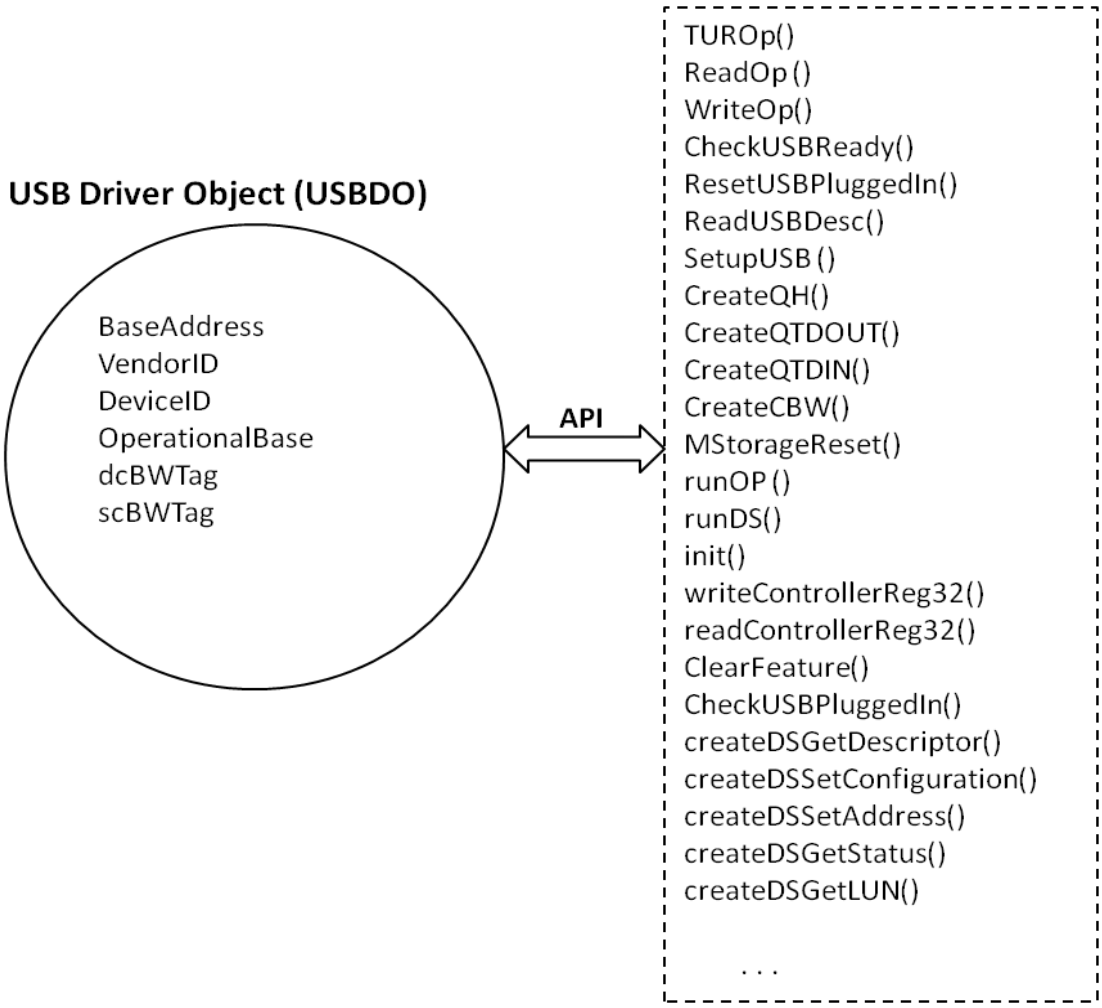


Figure 5. USB Driver API

**3.4.1 ResetUSBPluggedIn():** After detecting the port that is plugged-in, a port reset is issued using the API resetUSBPluggedIn(portno) as shown in Fig. 5. The implementation of this interface is shown in Fig. 7. The reset operation is the most critical operation for a USB port before it becomes available for

operation. The reset process does the following actions: it reads the port status, checks if it is enabled, sets a reset bit in the status register, waits for 10 milliseconds, unsets the reset bit, waits for another 1 millisecond, and continues this process until the port is enabled. The port status registers [9] act as status as well as control registers. This can be read or written by the programmer. When a bit is set (to 1 or 0) and written to the register, it will reset that bit. The delay in this function is crucial to the success of its operation. Bit 8 is the Port Reset bit in the register. Bits 2, 1, 0 (Port Enable, Connect Status Change, Current Connect Status) will be all 1s (x7) when the port is properly enabled. Errors and exceptions that could result in this operation are not discussed in this paper. After completion of a reset, the process control will go to the READDESC state.

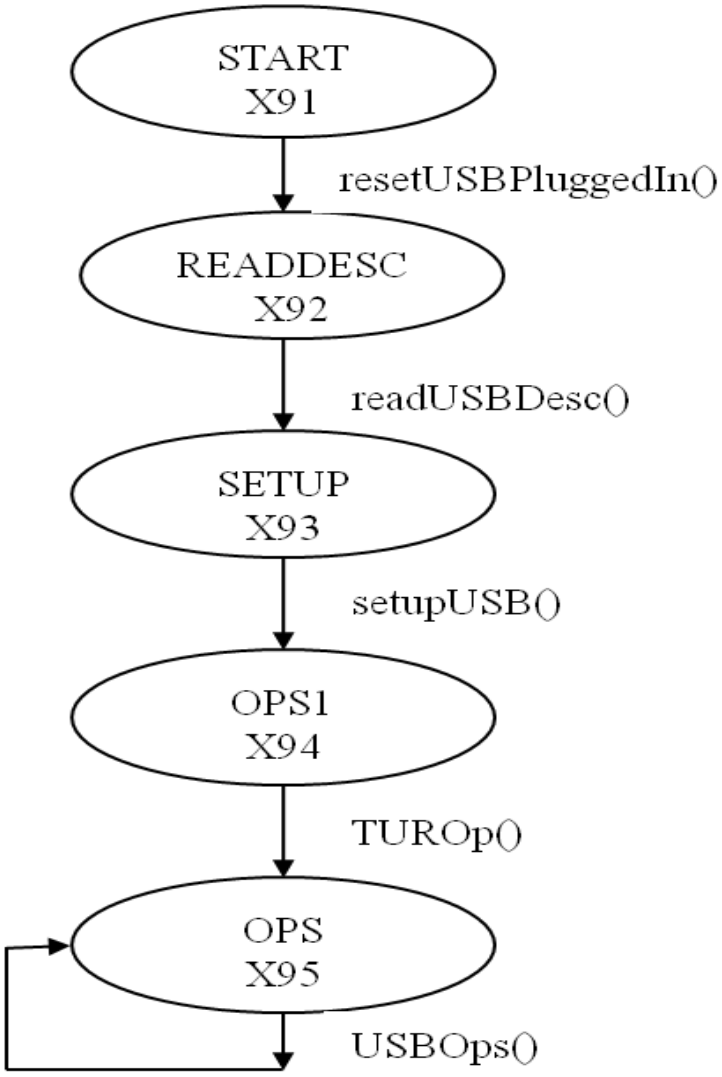


Figure 6. State Transition Diagram

**3.4.2 ReadUSBDesc()**: There are several descriptors in USB 2.0 [17]; some detailed explanations regarding them are given in [2]. Device, Configuration, Interface and Endpoint are the most important descriptors that have to be read from the USB device. These descriptors provide device characteristics such as packet size, vendor id, product id, number of configurations, length of descriptors, interface numbers, protocol code (bulk) and end point addresses. We need some of these parameters to communicate with the device. Every mass storage device has a control endpoint (address 0), one IN endpoint (address 1) and OUT endpoint (address 2), and also a direction flag. Descriptors can be read at any point in time during USB operations. We read the essential descriptors from the device, parse the parameters and store them in memory for further use.

```
void UsbObj::resetDevicePluggedIn(int portno)
{
  while(1)
  {
    v2 = readControllerReg32(OperationalBase+68+(portno-1)*4);//Port Status
    if (v2 == 0x00001007)
      break;  //when a HS port is enabled, then quit
      retcode = writeControllerReg32(OperationalBase+68+(portno-1)*4,
                                     0x00000100); //reset port
      delay(10); //10 milliseconds
      retcode = writeControllerReg32(OperationalBase+68+(portno-1)*4,
                                     0x00000000); //remove it from reset state
      delay(1); //1 milliseconds
      v2 = readControllerReg32(OperationalBase+68+(portno-1)*4);//Port Status
  }; //end of while
}; //end of function
```

Figure 7. ResetDevicePluggedIn()

We now give the details of one interface in the bare PC mass storage device driver API that is used to read a device descriptor. We have to create a data structure in memory and create a transaction to run a given interface. The readUSBdesc() API in Fig. 5 only shows a generic API for descriptors, and it does not show all the different data structures needed to build USB commands. Different data structures are

needed to implement different USB commands as they differ in many attributes and connections in a given transaction.

There are two building blocks involved in these data structures. Queue Head (QH) [9] describes the details of a QH and Queue Transfer Element Descriptor (qTD) [9] shows the details of a qTD structure. In addition, USB commands as described in USB 2.0 [17] are needed to build the complete data structure that can be run by the host controller. Some of the USB commands involved include: CLEAR_FEATURE, GET_CONFIGURATION, GET_DESCRIPTOR, GET_INTERFACE, GET_STATUS, SET_ADDRESS, and SET_CONFIGURATION [17].

Fig. 8 shows the data structure for reading a device descriptor. The real memory addresses are shown inside each data structure element. QH consists of 48 bytes and qTD consists of 32 bytes in this data structure. The actual command and data fields used in the bare system are shown in the figure. After creating the data structure, the transaction must be run by setting a run bit in the command register. Also, the QH address (e.g. 0x13530000) is stored in the ASYNCLISTADDR register. Some details of this process are shown in Fig. 9. Notice that after execution, the data read during IN operation is stored in memory address 0x13631200. There is only one QH, which points to itself, indicating that it will terminate after its execution. QTD Setup points to data, where a command is encoded as **06** indicating GET_DESCRIPTOR. The details of this command can be found in [17]. QTD IN points to data, where the device descriptor read will be stored after a successful operation. The device descriptor consists of 18 bytes (as indicated in **size** field **0x12**), and is encoded as in [2, 17]. QTD OUT points to data, where a status value of zero is stored and sent to the device as an acknowledgement of the successful command. For reading other descriptors, similar data structures are constructed and executed. After all descriptors are read, the process control goes to SETUP state.

**3.4.3 Setup():** The setup() function consists of several steps that require more USB APIs as in Fig. 5. The necessary operations are set address, set configuration, get LUN and get status functions as shown in Fig. 9. For each of these interfaces, a data structure is created (similar to that in Fig. 8) and USB commands are coded as in [17]. Each command is run individually using the runDS() API. The get descriptor and get status commands can be used to verify the port address and the configuration of the device. Each device is given a unique port address (1-128) that can be used to address the device after the set address command is successful.

We also need to do error checking during the setup to ensure that each command runs properly. In a bare PC environment, we check the state of execution of each entity in the transaction using real memory. Real memory can be used to display a given data structure and its encodings, and it is also used to validate the data and errors in the driver program. We also use a Beagle Analyzer [20] trace to validate each command and its operation. However, this analyzer does not display all elements shown in Fig. 8. A bare PC programmer has total control of the creation and operation of USB commands via the direct hardware interfaces (or API) used in the C/C++ application program as in Fig. 5. After reading the essential descriptors, the process control goes to the OPS1 state.

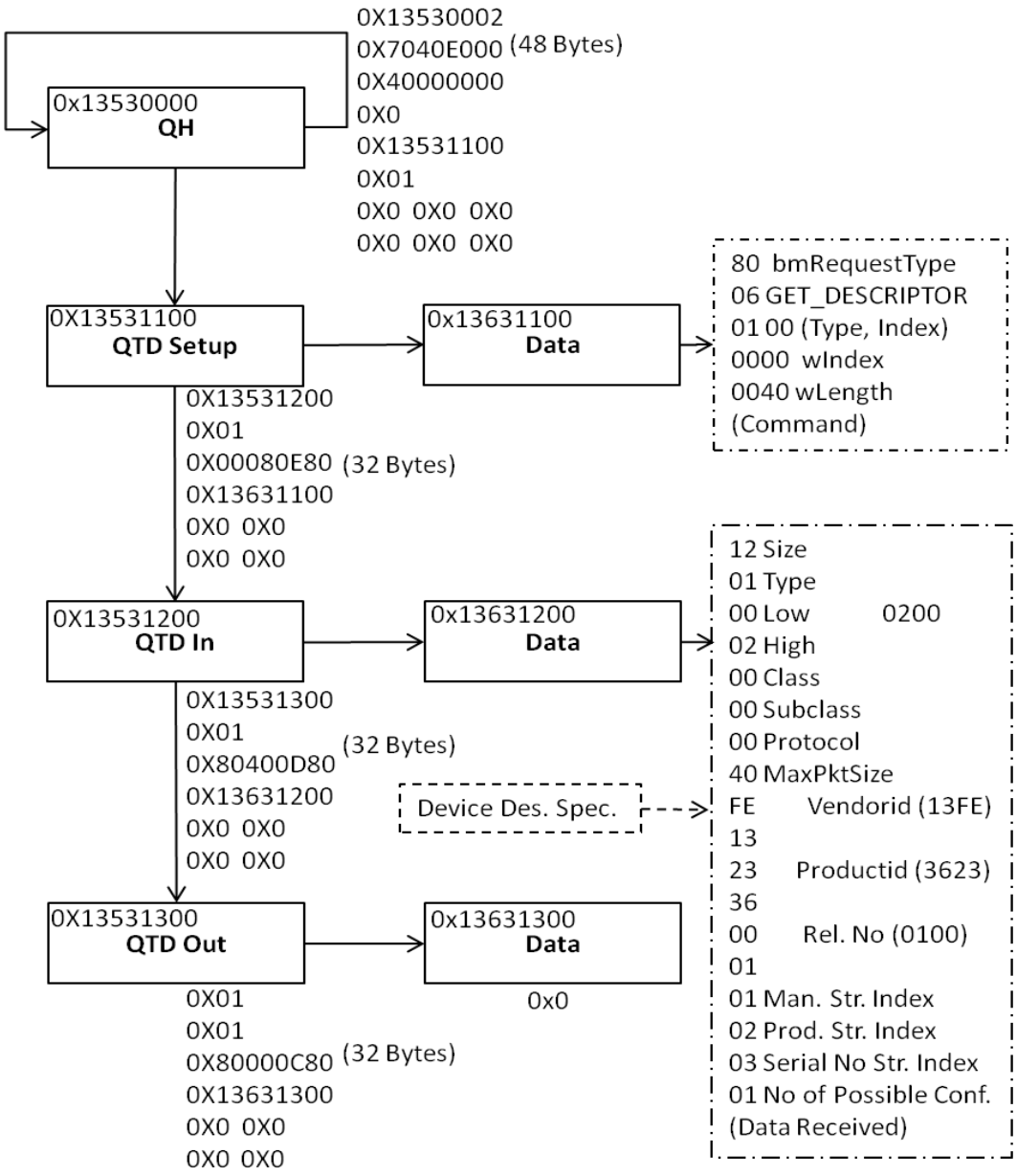


Figure 8. Read USB Descriptor Data Structure

```
//readUSBDesc()
ptr1 = usbo.createDSGetDescriptor (0x00, 0x0001, 0x0040, &dptr, 0, 0);
 //dev addr, type, length, dataptr, langid, index returns last
//   qtd address + 8 to check for ACTIVE status bit
 //when running the transaction
//Run transaction
retcode = usbo.runDS (QH_ISTART_ADDR, ptr1);   //run the QH
//runDS() requires the following setup
//   se t ASYNCLISTADDR
retcode = writeControllerReg32(OperationalBase+24, qh);
//Enable RUN bit in the Command Register
retcode = writeControllerReg32(OperationalBase+0, 0x00080021);

//setup()
ptr1 = usbo.createDSSetAddress (devaddress);
retcode = usbo.runDS (QH_ISTART_ADDR, ptr1);
 ptr1 = usbo.createDSSetConfiguration (devaddress, 0x0001);
//dev addr, config value
retcode = usbo.runDS (QH_ISTART_ADDR, ptr1);
 ptr1 = usbo.createDSGetLUN (devaddress, &dptr);
retcode = usbo.runDS (QH_ISTART_ADDR, ptr1);
ptr1 = usbo.createDSGetStatus (devaddress, 0x0002, &dptr);
 //dev addr, length, dataptr
retcode = usbo.runDS (QH_ISTART_ADDR, ptr1);
```

Figure 9. ReadUSBDesc() and Setup() Code

### 3.5 SCSI Command API

The USB device is ready for operation once the above steps are completed successfully. SCSI [19] commands are used in the mass storage USB similar to those in a hard disk or CD ROM. There are many types of SCSI commands (and versions). We have implemented the essential commands that are needed for a mass storage device. The SCSI commands are encapsulated inside a USB command as a block referred to as a command block wrapper (CBW) [21]. The status received from the device is referred to as a command status wrapper (CSW) [21]. The data structures required for SCSI commands are slightly different from those for control commands in Fig. 8.

This design includes standard USB controller interfaces or an API that can be used to build a variety of SCSI commands. The controller interfaces need to perform a ReadOp() is shown in Fig. 10. Step (1)

creates a QH for a read operation. It requires the QH address, the out endpoint and some type parameters. Step (2) creates QTD OUT that is linked with the above QH. The QTD OUT command requires the QH address, the data toggle bit, the LBA (logical block address) and the number of sectors to read. Step (3) creates a second QH that will link with the first QH created in step (1). The first QH is used for the OUT command and the second is used for the IN command. Step (4) creates a QTD IN that is used to read data from the device. This requires the QH address, the data toggle, the number of sectors and the data address where data will be stored.

The transaction diagram for a read operation is shown in Fig. 11. After all elements are connected, Step (5) executes the data structure starting from the first QH (address qh1). Certain fields in the QH and the QTD play significant roles in performing a given read or write operation. Some of these fields are expanded in Fig. 11 to illustrate the significance of their values. The Head of Reclamation Flag (H), Data Toggle Control (DTC) and Endpoint Speed (EPS) as described in [9] form a four bit hex digit field. This hex digit is different for the first QH and the subsequent QHs. For the first QH, this digit should be "**0xE**" indicating that "**H**" bit is '**1**', DTC is '**1**' and EPS is '**10**'. All first QHs should have an "**H**" bit of 1 as they indicate the head of all other QHs. All subsequent QHs should have the "H" bit set to "**0**". In our system, since the mass storage device is a high-speed device, the EPS field is "**10**" [9]. The DTC bit in the QH indicates where the host controller should get the initial data toggle bit on overlay transition. In our system, we set this bit to one indicating that the data toggle (DT) bit comes from the first QTD. It is essential that the first QTD reflect the current DT bit needed for its command. The Interrupt on Completion (IOC) flag is set in the last QTD (QTD IN, Fig. 11) so that the driver program can check this interrupt bit, which is status register bit (0) [9]. The IOC bit in QTD OUT as shown in Fig. 11 should be '**0**' so it does not cause any interrupt after its completion. WriteOp() is similar to ReadOp() except it has more QTD OUT Commands to send data.

```
STEP 1: qh  = usbo.CreateQH  (1, qh1, outend, devaddr, 1);
//cbwtype = 1 (read), type = 1 for OUT QH

      //QTD OUT format: cbwtype, qhstart, dtout, nosectors, qtdno, lba, dpt, lunbyte, lastone
STEP 2: qh  = usbo.CreateQTDOUT (1, qh1+0x200, dtout[taskindex],  nosectors, 1, lbara, 0, 0, 1);
//OUT

STEP 3: qh  = usbo.CreateQH  (1, qh1, inend, devaddr, 2);
//cbwtype = 1 (read), type = 2 for IN qh

      //QTD IN format: cbwtype, qhstart, dtin, nosectors, qtdno, dstart, lastone
STEP 4: qh  = usbo.CreateQTDIN (1, qh1+0x0800, dtin[taskindex], nosectors,  1, dptr, 1);
//IN

STEP 5: retcode  = usbo.runOP  (qh1);
```

Figure 10. ReadOp()

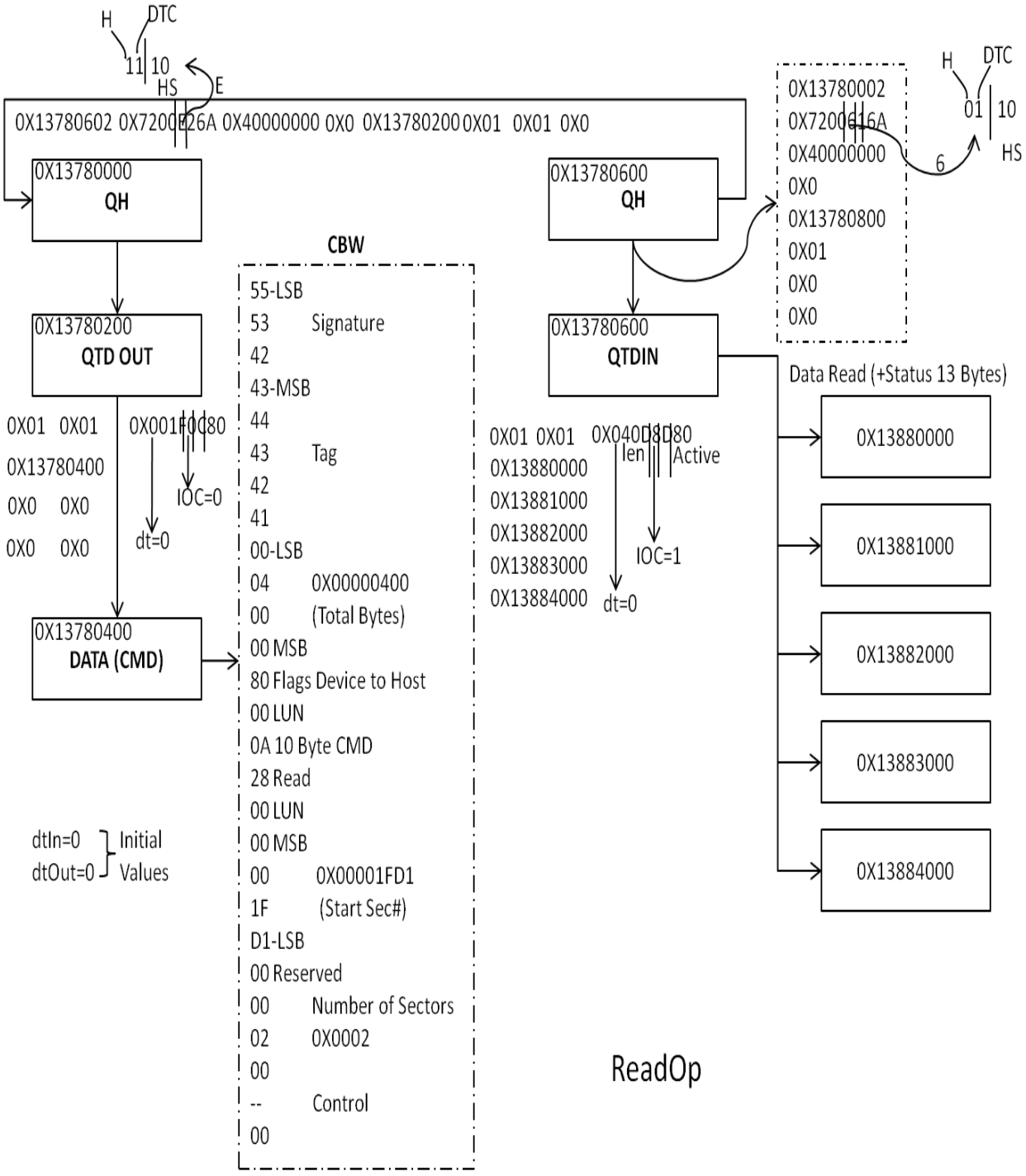


Figure 11. ReadOp Structure

### 3.6 Large Read or Write Operations

A typical Read or Write QTD is limited to 20,480 bytes or 40 blocks. Large read operations can be achieved by linking multiple QTD INs together. The total of number of bytes must be specified in the

QTD OUT command for read. The total of number of bytes must be specified in the first QTD OUT command for write.

### 3.7 Data Toggle Bit

Data toggle operation and usage is quite ambiguous in all USB documentation. Data toggle is used between a host and a device to send or receive the proper sequence of data [2]. There is a data toggle control (DTC) bit in a QH, which controls the usage of data toggle bit. In a QH, when the DTC bit is 1, it uses the first QTD data toggle (DT) bit as the start of data toggle bit. Any subsequent QTD's DT bit will not be used by the host controller. The HC automatically toggles the data bit during the execution of each transaction. For each mass storage operation, it is better to start with toggle bit '**0**' so that the sequence of data received or sent will not be out of synchronization. At the end of a given mass storage operation, the data toggle bit is updated by the host controller and the latest toggle state can be obtained from bit 31 of the **7th** word (4 byte word) of QH in its overlay area. QH and the executed QTD are overlaid as in [9]. This toggle bit state must be fed to the next operation; otherwise the device will stall resulting in an incorrect mass storage data transfer operation. A ClearFeature() for an IN endpoint, and a ClearFeature() for an OUT endpoint can be used to reset data toggle bits in the system (by using the HALT command option [17] ).

### 3.8 Command Block Wrapper (CBW)

A command block wrapper (CBW) is a 32-byte data field that is encapsulated in a USB OUT command. The format of CBW is given in [21]. A CBW includes several parameters that can be used to synchronize operations. The signature field placed in CBW will be returned by the device to validate an operation. The tag field can be used to identify a given operation uniquely. The number of bytes included in this block indicates the total number of bytes to be transferred for this operation. A sample CBW for the read operation is shown in Fig. 11.

### 3.9 Command Status Wrapper (CSW)

A command status wrapper (CSW) is a 13-byte status field that is returned by a device at end of its operation. The CSW must be read with the last QTD IN command in the transaction. The number of bytes of status (13 bytes) must be added to the total number of bytes in the first QTD OUT command's CBW. The data residue field can be checked to ensure completion of data transfer. The tag and signature fields can be validated to verify appropriate command completion. The status field also contains one byte of error code.

### 3.10 USB Plug-in Detection

Port status and controller register [9] indicate status and control information for a given port. Bit 0 in this register shows current connect status and bit 1 indicates connect status change. Using these two bits we can determine the plug-in status of a given USB port. For each port, the device driver should maintain its status and connectivity. When a device is plugged-in or removed, its status must be updated. When a device is plugged-in, it will go through state transitions shown in Fig. 6 before it becomes operational.

## 4 Performance Measurements

The performance of bare PC USB mass storage driver was compared with that of a Linux application program that uses raw I/O calls: open(), read() and write(). We also used the USB Beagle analyzer [20] to validate USB operations. The measurements were conducted on Dell Optiplex GX260 desktop with 2.4 GHZ CPU. The USB device used was Verbatim with 2GB capacity. Some implementation details of the driver are provided in the next subsection.

### 4.1 Implementation

The bare PC mass storage device driver was implemented based on the details given in Section 3. The driver code was written in C except for a few Microsoft ASM assembly language instructions. The number of lines of C code is 5765 (3830 executable lines, 60 member functions) and the number of lines of Microsoft MASM assembly code is 116 (6 function calls). The test program and the driver code run on any bare PC with an x86 CPU. The program and driver comprise a single executable, which resides on a USB. The API described in the paper is used to perform USB operations. The entire code does not use any system calls and has no dependence on an OS, kernel (lean or otherwise) or other software. The driver API can be directly accessed from a C or C++ application program without the need for any system libraries. An application programmer controls the sole operation of the USB device which runs on a bare PC. The next subsection presents the results from testing the performance of this driver implementation.

### 4.2 Results

Fig. 12 shows read timings when the number of sectors is varied from 32 to 4096 on a Linux (Fedora 11) platform. It shows the Linux time and also the Beagle analyzer measured time in milliseconds. The measured time reported by the analyzer is always less than the time measured in the application. Linux uses interrupt I/O for mass storage operations. It is likely that the measured time in the application and the driver time (measured with the analyzer) overlap each other.

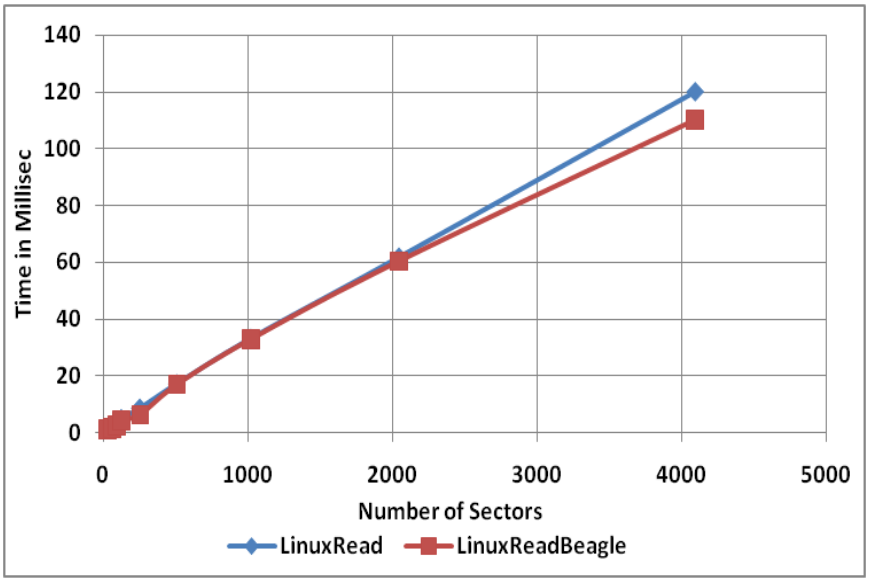


Figure 12. Linux Read Operation

Thus, the time measured in the application does not include driver initialization and setup for both read and write operations. In order to measure and compare drivers, we need to put timing probes in the driver itself in Linux. This was not done as it is a very tedious and time consuming task.

Fig. 13 shows read timings when the number of sectors is varied from 32 to 4096 on a bare PC with the bare PC driver. The Beagle time and application time are now very close to each other as there is no overlap in the measurements. This also indicates that the driver program only took a small amount of time and that it invoked the host controller directly without any delay. Read time comparisons when the number of sectors is varied from 32 to 4096 are shown in Fig. 14. As expected, the bare PC driver performs better than the Linux application. However, the time measured in Linux application is less than actual time taken in the system as mentioned before due to interrupt I/O and the overlapping of application program and driver times. The bare PC read shows a 7.6% read access time improvement over the Linux application when reading 4096 sectors.

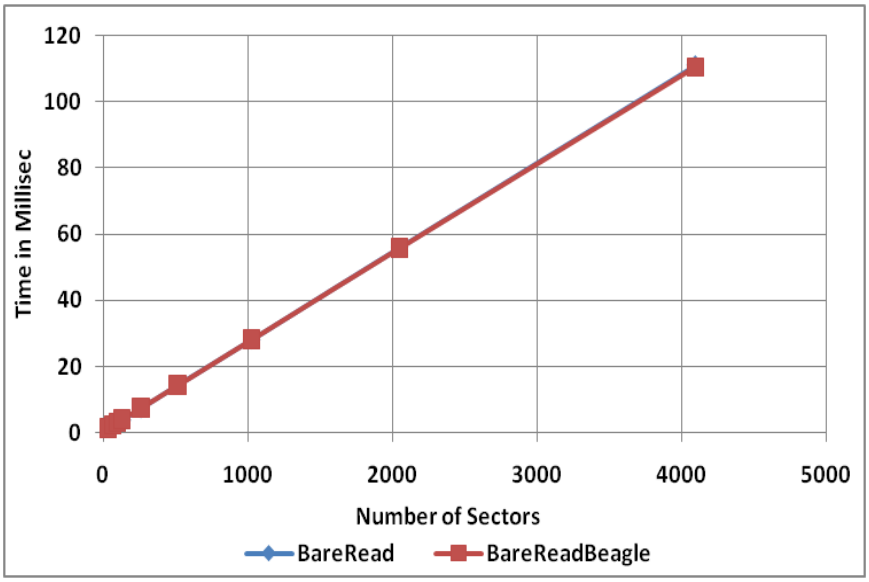


Figure 13. Bare Read Operation

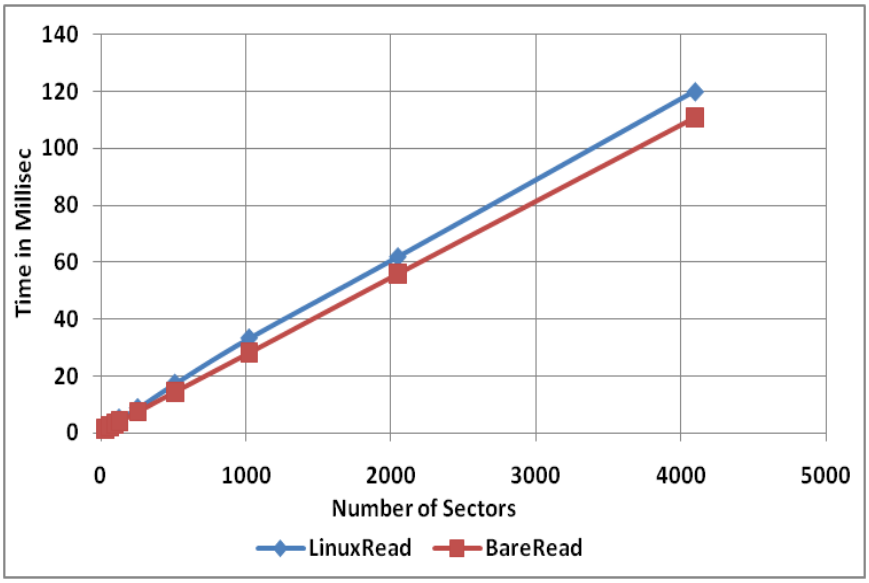


Figure 14. Read Comparison

In order to conduct a stress test on the USB drivers, we varied the number of sectors from 5000 to 2,000,000 as shown in Fig. 15. There is a significant performance difference between the Linux and bare PC applications. The latter demonstrates a performance gain of 18% to 21% over a broad range of read operations.

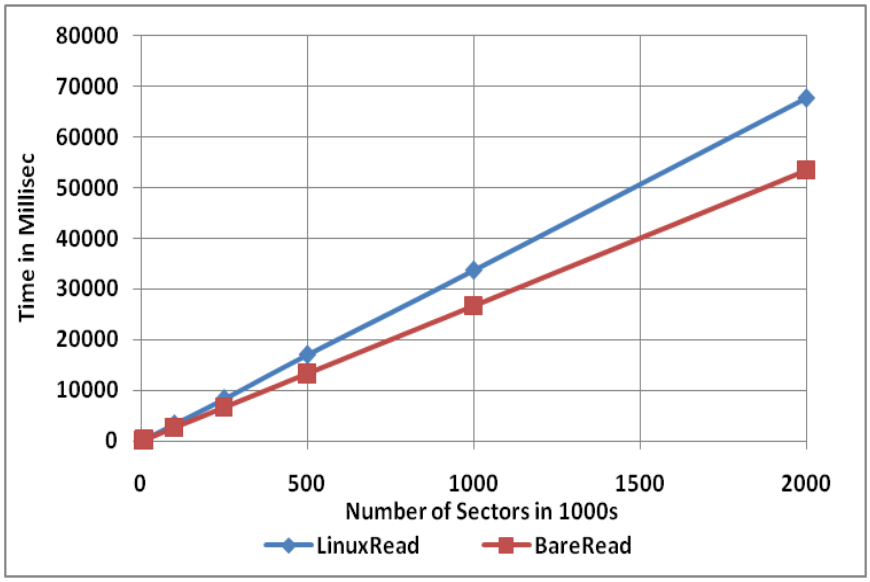


Figure 15. Read Comparison for Large Sectors

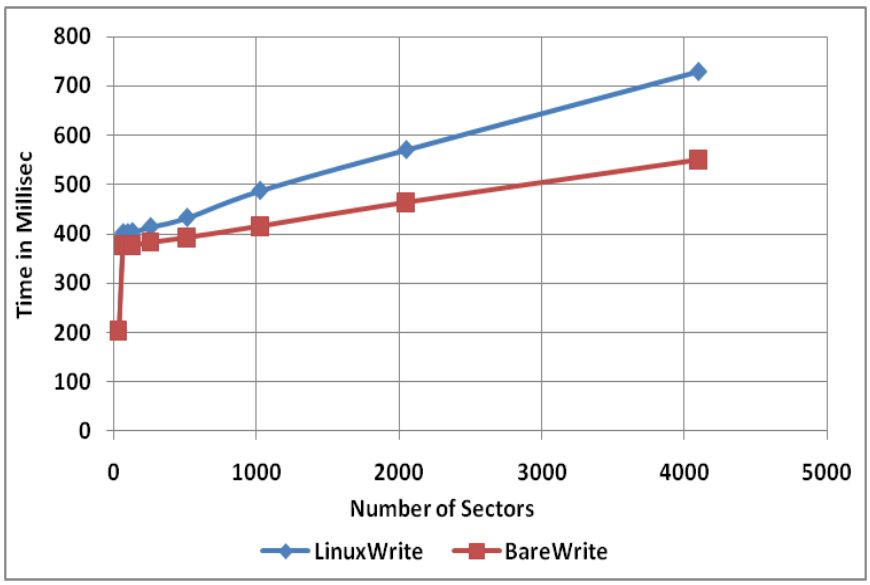


Figure 16. Write Comparison

Fig. 16 shows write timings when the number of sectors is varied from 32 to 4096. The bare PC application performs better than the Linux application when the number of sectors to write increases. The bare PC performance gain when writing 4096 sectors to the device is about 24.5%.

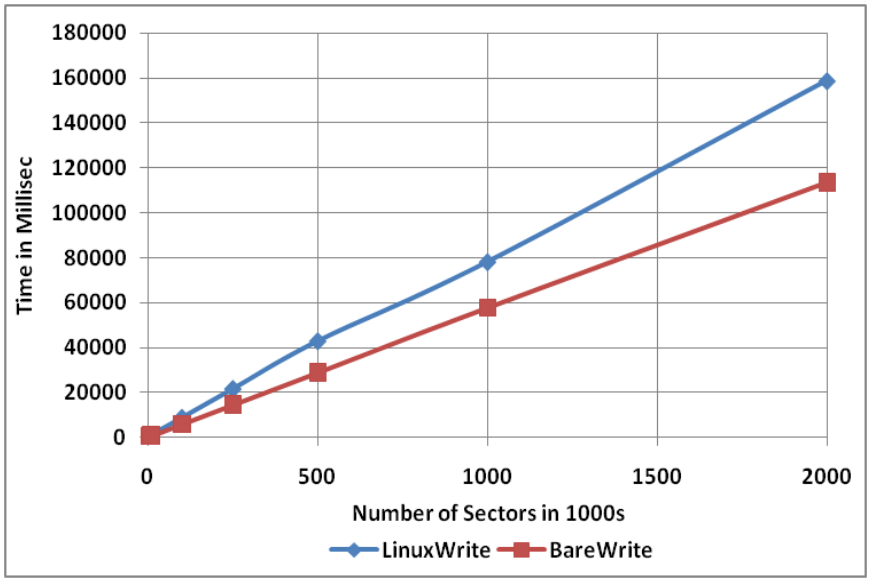


Figure 17. Write Comparison for Large Sectors

Stress test measurements for the write operation are shown in Fig. 17. The number of sectors to write is varied from 5000 to 2,000,000. The bare PC application outperforms the Linux system by an average of 30.6% with respect to processing time. Fig. 18 shows read timings for 100,000 sectors. When the number of sectors to read is very large, USB drivers divide them into small groups. In the bare PC application, we can form groups of any desired sizes as the driver is designed and controlled by the programmer. Linux uses groups of 128 sectors for a single read operation. On the bare PC, the number of sectors in a group was varied from 64 to 25,000 for a single read operation. It can be seen that a group of about 10,000 sectors appears to provide the minimal read time. Therefore, a group of 10,000 sectors was used to perform large read operations. The sharp peak in timing for a group of 64 sectors indicates that it requires a large number of USB commands for fetching a large number of sectors. Each USB read operation also requires the creation of a data structure for the transaction that must be linked to the USB schedule. This results in severe performance degradation.

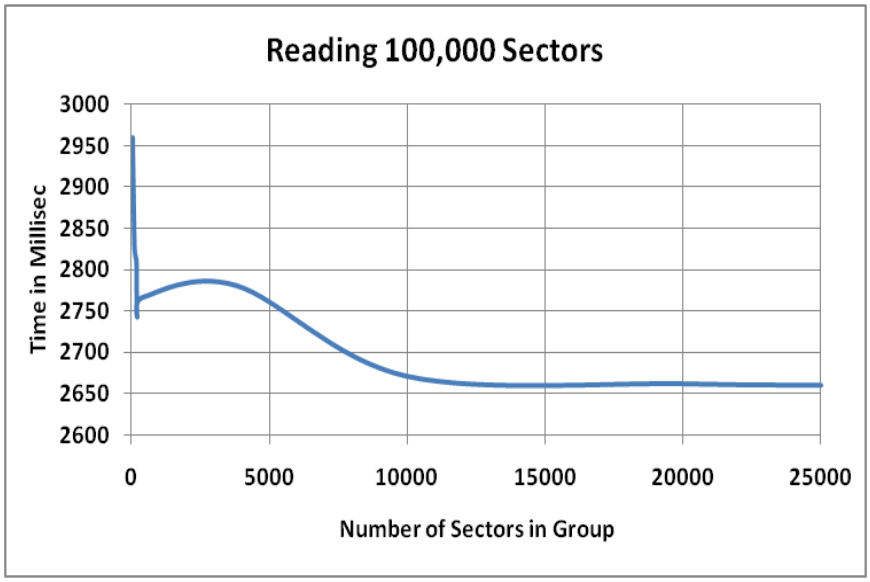


Figure 18. Read Timings for 100,000 Sectors

Fig. 19 shows write timings for 100,000 sectors. Grouping to perform write operations is done in a manner similar to the read operation. Here, the number of sectors in a group was varied from 64 to 50,000 for a single write operation. Again, a group of about 10,000 sectors appears to provide minimal time and so a group of 10,000 sectors was used to perform large write operations.

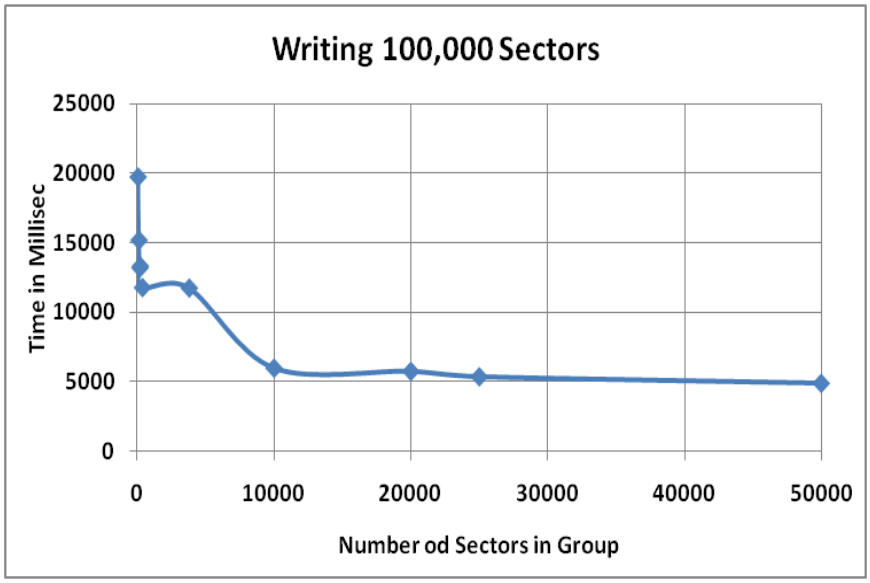


Figure 19. Write Timings for 100,000 Sectors

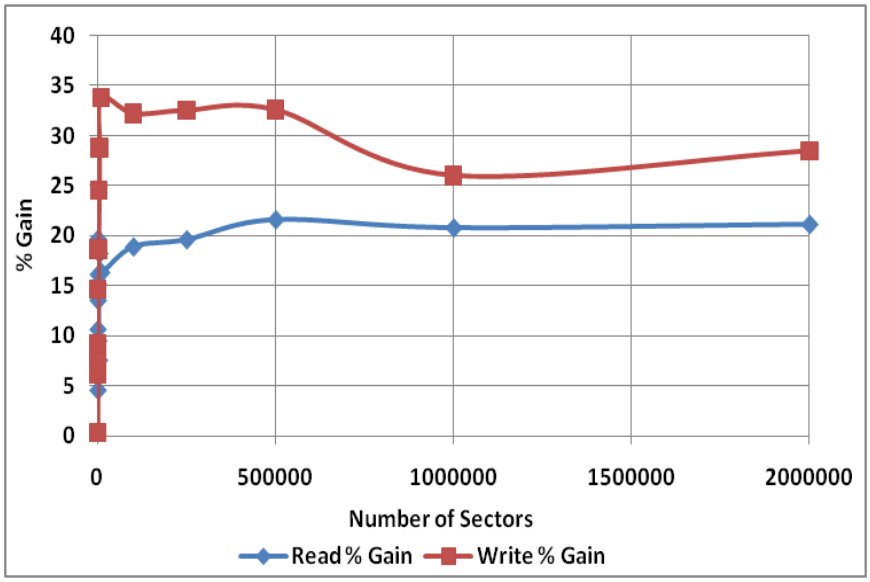


Figure 20. Read/Write % Gain

The percentage gains in performance for read and write operations for the bare PC application compared to the Linux application are shown in Fig. 20. The read timing performance gain increases for large operations and stabilizes at about a 21% gain with an average gain of about 15%. The write timing performance gain varies based on the number of sectors to write to the device. The maximum gain for write operations is at 10,000 sectors. The average gain is about 19% for write operations.

### 4.3 Analysis

Several observations can be made by using the preceding performance data for bare PC driver and its application.

(1) A group size of 10,000 sectors seems to be best for read and write operations

(2) A bare driver performs well for large data transfers from mass storage to memory

(3) The read/write group sizes can be dynamically varied to achieve higher gains in performance

(4) The bare driver takes minimal CPU processing time (since its time closely matches with the Beagle analyzer time).

## 5 Conclusion

This paper described an approach for building a bare PC mass storage USB device driver that is independent of any OS, kernel, environment, or embedded application. A step-by-step procedure for constructing a USB mass storage device that runs on any Intel x86 based CPU was presented. A variety of standards, documents and references were given that contain more detailed information relevant to implementing a bare driver. A standard API and code snippets from the actual implementation were included in order to illustrate the internal workings of the driver. Performance measurements for read and write operations indicate a performance gain for the bare PC driver over Linux.

The novelties of this approach reflect its broader impacts. As USB interfaces are used in home appliances, electronics, controllers, games and toys, this research demonstrates that the necessary software in these and other scenarios can be made bare i.e., independent of any environment. The bare USB drivers are small in code size, simple to design and implement, controllable by an application programmer, flexible to upgrade and enhance, and independent of any platform. Since the USB provides standard functions and controls, it is possible to make a generic API that is common to many pervasive devices. Ideally, USB vendors will provide this standard API within the device itself so that it will become possible for an application programmer to directly access the device from a bare application. Devices with such an API will enable application programs to bypass the OS and directly access I/O devices making it easier to design applications with less overhead and better security.

## References


[1] P. Appiah-Kubi, R.K. Karne and A.L. Wijesinha, "A Bare PC TLS Webmail Server," Proc. Of IEEE Workshop on Computing, Networking and Communications, ICNC-CNC, Maui, HI, pp. 156-160, January 2012.

[2] J. Axelson, "USB Mass Storage, Designing and Programming Devices and Embedded Hosts," Lakeview Research, 2006.

[3] Y. H. Chang, P. Y. Hsu, Y. F. Lu, and T. W. Kuo, "A Driver-Layer Caching Policy for Removable Storage Devices," ACM Transactions on Storage, Vol. 7, No. 1, Article 1, p1:1-1:23, June 2011.

[4] M. Choi, H. Park, and J. Jeon, "Design and Implementation of a FAT File System for Reduced Cluster Switching Overhead," 2008 International Conference on Multimedia and Ubiquitous Engineering, 2008.

[5] D. R. Engler and M.F. Kaashoek, "Exterminate all operating system abstractions," Fifth Workshop on Hot Topics in operating Systems, USENIX, Orcas Island, WA, p. 78, May 1995.

[6] J. A. Garrison and A. L. N. Reddy, "Umbrella File System: Storage Management across Heterogeneous Devices," Vol. 5, No. 1, Article 3, p3:1-3:24, March 2009.

[7] L. He, R. K. Karne, and A. L. Wijesinha, "The Design and Performance of a Bare PC Web Server," International Journal of Computers and Their Applications, IJCA, Vol. 15, No. 2, pp. 100-112, June 2008.

[8] M. Hunter, "Simplified EHCI Data Structures for the High-End ColdFire FamilyUSB Modules," Freescale Semiconductor Application Note, Document Number AN3520, Rev. 0, AN3520.pdf, September 2007.

[9] Intel Corporation, "Enhanced Host Controller Interface Specification for Universal Serial Bus," Rev 1, http://www.intel.com/technology/usb/download/ehci-r10.pdf, March 2002

[10] Intel Corporation, "Intel 82801DB I/O Controller HUB 4 (ICH4) Datasheet," http://download.intel.com/design/chipsets/specupdt/29074525.pdf, May 2002.

[11] R. K. Karne, K. V. Jaganathan, and T. Ahmed, "DOSC: Dispersed Operating System Computing," OOPSLA '05, 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications, Onward Track, ACM, San Diego, CA, pp. 55-61, October 2005.

[12] R. K. Karne, K. V. Jaganathan, and T. Ahmed, "How to run C++ Applications on a bare PC," SNPD 2005, Proceedings of SNPD 2005, 6th ACIS International Conference, IEEE, pp. 50-55, May 2005.

[13] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Z. Cui, L. Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, R. Brightwell, "Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing," Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010), April, 2010.

[14] Microsoft Corp, "FAT32 File System Specification," http://microsoft.com/whdc/system/platform/firmware/fatgn.rnspx, 2000.

[15] V. S. Pai, P. Druschel, and Zwaenepoel. "IO-Lite: A Unified I/O Buffering and Caching System," ACM Transactions on Computer Systems, Vol.18 (1), ACM, pp. 37-66, Feburary 2000.

[16] PCI Special Interest Group, "PCI Bios Specification," Revision 2.1, http://www.o3one.org/hwdocs/bios_doc/pci_bios_21.pdf, August 1994.

[17] Perisoft Corp, "Universal Serial Bus Specification 2.0," http://www.perisoft.net/engineer/usb_20.pdf.

[18] B. Rawal, R. K. Karne, and A. L. Wijesinha. “Mini Web Server Clusters for HTTP Request Splitting,” IEEE International Conference on High Performance, Computing and Communications, Banff, Canada, pp. 94-100, 2011.

[19] SCSI 2.0 Specifications, http://ldkelley.com/SCSI2/index.html.

[20] Total Phase Inc., “USB Analyzers,” Beagle, http://www.totalphase.com.

[21] USB Implementers Forum, “Universal Serial Bus Mass Storage Class, Bulk Only Transport,” Revision 1.0, http://www.usb.org/developers/devclass_docs/usbmassbulk_10.pdf, September 1999.