TOWSON UNIVERSITY
OFFICE OF GRADUATE STUDIES

MASS STORAGE SYSTEM FOR BARE MACHINE COMPUTING

by

William V. Thompson

A Dissertation

Presented to the faculty of

Towson University

in partial fulfillment

of the requirements for the degree

Doctor of Science

Department of Computer & Information Sciences

Towson University
Towson, Maryland 21252

May 2016

TOWSON UNIVERSITY
OFFICE OF GRADUATE STUDIES

DISSERTATION APPROVAL PAGE

This is to certify that the dissertation prepared by William V. Thompson, entitled "MASS STORAGE SYSTEM FOR BARE MACHINE COMPUTING," has been approved by the dissertation committee as satisfactorily completing the dissertation requirements for the degree Doctor of Science in Information Technology.

Ramesh K. Karne — May 18, 2016
Chair, Dissertation Committee, Dr. Ramesh K. Karne — Date

Alexander L. Wijesinha — 5/18/16
Committee member, Dr. Alexander L. Wijesinha — Date

Sungchul Hong — 5/18/16
Committee member, Dr. Sungchul Hong — Date

Yeong-Tae Song — 5/18/16
Committee member, Dr. Yeong-Tae Song — Date

Janet V. D'Agostino — 5-19-16
Dean of Graduate Studies — Date

ACKNOWLEDGEMENTS

First of all, I thank God without whom nothing is possible.

I would like to express deepest gratitude to my research committee members: Dr. Ramesh K. Karne (chair), Dr. Alexander L. Wijesinha, Dr. Yeong-Tae Song, and Dr. Sungchul Hong for supporting this research. Their thoughtful questions and comments were valued greatly. Special thanks to my wonderful advisor Dr. Ramesh K. Karne for his full support, expert guidance, understanding and encouragement throughout this research. Without his incredible patience, counsel, and sacrificing his weekends to spend long hours in the lab with me, this dissertation and research work would have been an overwhelming pursuit.

I would be remiss not to thank the Office of Graduate Studies at Towson University for granting me a full year Graduate Research Fellowship Award to complete my studies. Without this opportunity it would have been more financial struggle for me to reach thus far. Thanks also go to my fellow doctoral graduate students in the Bare Machine Computing lab, and to my employer – the Maryland Institute for Emergency Medical Services Systems – especially my supervisor, Mr. David H. Balthis, for allowing to work flexible hours during this dissertation research.

Finally, I would like to offer my heartfelt thanks to my wonderful daughters, Willette and Korlu Thompson, for their support and understanding during the time of my research, especially for the last three months. Now that this dissertation is complete, my time and attention is fully yours again!

To my best friend Augustine J. Zaizay, Jr.
Thank you for actually reading this.

ABSTRACT

This dissertation extends on-going Bare Machine Computing (BMC) research at Towson University. BMC applications run on a bare machine without dependencies on operating systems (OS), lean kernel, embedded OS, or any other centralized support. This dissertation will serve as a cornerstone for future mass storage systems that run on bare machines.

Mass storage systems based on USB flash devices are supported on many platforms and commonly used today by variety of applications. These systems vary depending upon the underlying operating system and their environments. This work leverages on the cheap flash drives with larger storage capacities. They are quite friendly to BMC as they are removable devices. Bare machine mass storage system enables computer applications to access them at run-time without the need for any operating system. This dissertation provides two avenues for mass storage systems. First, it provides a simple BMC tailored file system API for BMC programmer to store conventional file based on FAT32 specification. Second, it provides a sample database SQLite files to be stored on USBs that demonstrates the database capabilities in BMC. Both of these avenues clearly demonstrate the greater potential of BMC mass storage systems for computer applications.

This dissertation developed a novel concept where a database designed on a conventional database management system is inter-operable with the database on a bare PC system. This concept is validated with SQLite database and tested for its inter-operability. The idea is applicable to any other database management system as long as that system is transformed to run on a bare PC. The SQLite database management system

was transformed to run on a bare PC in the earlier work as part of BMC research team. This work provides a motivation to use bare machine systems as a backend and OS based systems as a front-end to the users. That is, the actual mass storage is hidden from the users to shield from OS vulnerabilities. The lean file system interfaces for conventional files and file interface to SQLite database provides a robust and scalable mass storage systems on bare machines. Bare PCs or machines can be used for big data and clouds based on this mass storage system.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1 INTRODUCTION

In today's ever more technology-driven society, virtually all businesses and individuals depend on complex computer-based systems to communicate and conduct business. Unarguably, the most important software component in modern computers is the operating system (OS). Operating systems were introduced many decades ago to facilitate computer applications sharing resources and to provide other administrative functions. Originally, operating systems were small, efficient, and easier to use as they were only supporting a small set of applications. Today, the typical OS supports many applications requiring the OS and other software to be much more complex. With the growth in OS and software complexity and size new vulnerabilities are introduced. New security holes are constantly being discovered right after existing ones are fixed.

The massive growth in computing hardware and software has created unmanageable electronic waste [5]. This is partly because often new software cannot work with legacy hardware. Software applications, operating systems, tools and gadgets become quickly obsolete in years – if not in months. The operating system size also increases dramatically. Take for example, couple years ago most businesses and individuals were using Microsoft® Window XP Professional version with SP3, the size of OS around 1.5 GB. The most recent Microsoft® OS – Windows 10 – the minimum file size is over 6 GB.

In this context, the Bare Machine Computing (BMC) research initiative at Towson University is geared towards developing secured self-contained computing environments as an alternative to the current OS/application-based model in place today. In the BMC laboratory, numerous applications have been developed to run on 32 bit x86 CPU architectures independent of traditional operating systems. Some of these application

include web servers, webmail, email servers, SIP servers, SIP client agents, VoIP softphone, and email clients. These applications demonstrate simplicity, small code size and high performance. Some of these servers can now be used as a complete system for real world applications; however, they are all in need of a bare machine mass storage system for complete operation. Thus, the premise of this research – which motivated us to research, design, develop, and implement a mass storage system for bare machine computing environment.

Another motivation stems from the following observations. Most often mass storage systems are based on a hard disk and its inherent characteristics. A USB device has a unique property of linear block addressing that can be directly mapped to real memory. That is, a USB behaves similar to main memory and it has no mechanical parts unlike a hard disk. A USB can be used as a logical extension to memory. In fact, a CPU can be designed to operate this way so that there is no semantic gap between memory and a mass storage. The above motivations led to the development of a bare machine file management system and consequently a bare machine mass storage system based on the USB technology.

In the current dissertation research, we describe the implementation of a novel bare machine USB file system designed for applications that run without the support of any OS environment/platform, lean kernel or embedded software. We also present an enhanced file API for bare PC applications. The file system enables a programmer to build and control an entire application from the top down to its USB data storage level without the need for an OS or intermediary system. This implementation can be used as a basis for extending

bare mass storage system capabilities in the future. The mass storage system can be integrated with bare PC applications such as Web servers, Webmail/email servers, SIP servers and VoIP clients.

2 RELATED WORK

2.1 Bare Machine Computing Background and Overview

The Bare Machine Computing (BMC) paradigm was started by Dr. Ramesh K. Karne at Towson University more than a decade ago, which was also referred to earlier as a dispersed operating system computing (DOSC) system and application-oriented object (AOA) architecture. The key concepts of the BMC paradigm are as follows:

a. Computing applications run in a bare machine without Operating System, centralized kernel, or any system software pre-loaded into the machine.
b. The applications can be loaded and carried in a portable device such as flash memory, and run in a bare machine anywhere.

The BMC approach is a novel approach for developing computing applications; in that unlike traditional approaches, it is application-centric. It has two primary characteristics. First, it runs on a bare machine, secondly it is a different programming paradigm, where an application programmer controls all aspects of hardware resources. It differs completely from conventional computing approaches that are environment and platform-centric. For instance, a given application suite contains its own boot, load and its applications. An end user simply carries the application suite on a removable device and can run this on any x86 based bare PC. There is little need to upgrade or patch the computing environment often; the focus is on the applications themselves. Once a computing box is made bare, the expense to protect it is minimized since it only has memory, CPU, basic user interface (input/output), and network interface(s). All persistent data are either stored on a removable device such as USB flash memory or on the network.

In the BMC paradigm, an Application Object (AO) is a self-contained, self-controllable and self-executable unit [17]; so, when an AO is developed, it can be run in any bare hardware such as desktop, laptop, and hand-held, or other electronic device.

The bare machine has no particular ownership. The BMC concept assumes a computing device to be bare and a programmer controls all computing resources. In addition, where a bare application suite is running, there are no other applications run in the system. This approach makes computing devices ownerless and readily available for any user to use without worrying about ownership. The BMC approach makes computing application simpler and more secure in that the AO controls both application and execution aspects; it avoids all the system and kernel related vulnerabilities by making the device bare. The concept can also be extended to other CPU architectures and pervasive devices. The benefits of using bare machine applications are that there is no OS overhead and no OS-related vulnerabilities and machines are bare and long lasting. The minimalist principle discussed in [38] supports the need for BMC paradigm.

2.2 BMC Applications

During the past decade, doctoral research students have developed several complex bare applications in the bare machine computing laboratory at Towson University to demonstrate the feasibility of the BMC paradigm. Long He [13] developed the first bare PC Web server and demonstrated the feasibility of building complex software that runs on a bare PC with thousands of threads and outperforms other compatible commercial Web servers. Gholam H. Khaksari [21] developed the first VoIP soft-phone [2] that runs on a bare PC and provides secure communication on an end-to-end basis. Andre Alexander [1] built a SIP server and a bare SIP user agent to demonstrate the feasibility of running high

performance SIP servers with secure communication using the SRTP protocol. George H. Ford built the first Email server that runs on a bare PC and provides compatible performance to related commercial email servers [10, 11]. Ali Emdadi [8] implemented the complex TLS protocol for a bare Web server. Roman Yasinovskyy [45] implemented the IPv6 protocol for a bare PC VoIP softphone client. Bharat Rawal [32] developed a unique split protocol concept and applied it to Web servers that run on a bare PC. He also developed mini-cluster configurations for Web servers based on the split protocol concept that offer high performance [33] and run on a bare PC. Patrick Appiah-Kubi developed a secure Webmail server using TLS [3]. Earlier research work by Uzo Okafor [28, 29] described the transformation of SQLite and OS-based applications to run on a bare PC. These previous doctoral research dissertations made possible the discovery of many novel characteristics that are unique to BMC, such as a bare mass storage system, and others that would be applicable to future computing applications that run on bare devices.

2.3 Other Related Work

There are many approaches to reduce OS overhead, use lean kernels, or build a high-performance OS such as Exokernel [9], bare-metal Linux [44], IO-Lite [30], and Palacios and Kitten [23]. While the BMC paradigm somewhat resembles these approaches, there is a significant difference in that bare machine applications run without any centralized code in the form of an OS or kernel. The bare machine computing paradigm is at the extreme end of the spectrum compared to OS based systems and other intermediate lean kernel systems.

As the capacity of flash drives continues to increase and their cost is reducing, flash drives are expected to become an important component of future mass storage systems. Thus, recently there has been considerable interest to use flash memory in mass storage devices. File systems such as Umbrella [12] demonstrates a versatile file system that uses a hard disk and flash memory. The Umbrella file system illustrates how to integrate two different types of storage devices. Also, other research has dealt with adding cache systems at a driver level to gain performance improvements [6]. As per our knowledge, no previous work has been done on bare machine USB mass storage system.

3 LEAN FILE SYSTEM INTERFACES

3.1 Overview

File systems provide a means for organizing and retrieving data needed by many computer applications. Typically, they are closely tied to the underlying operating system and mass storage technology. Bare machine file systems are, in contrast, independent of any OS or platform. Such a file system can be used with computer applications that run on a bare machine with no OS, and also in a conventional OS environment. The file system can serve as a basis to support future bare machine database management systems, big data systems, and Web and mobile applications that eliminate OS overhead and cost. Furthermore, it can be used in bare machine security applications that provide protection from attacks targeting OS vulnerabilities.

The file system developed here depends on the USB architecture [31], USB Mass Storage Specification [43], USB Enhanced Host Controller Interface Specification [16], FAT32 standard [26], and the bare machine computing paradigm. The file system is stored on a USB along with its application. The USB layout is similar to a memory layout providing a linear block addressing (LBA) scheme. That is, a USB address map is similar to a memory map. However, a USB is accessed with sector numbers that are directly mapped to memory addresses. It uses small computer system interface (SCSI) commands that are encapsulated in USB commands. Thus, a bare PC USB driver that works with this file system is needed [20]. The FAT32 standard is complex and has a variety of options that are needed for an OS based system as it is required to work with many application environments. The FAT32 options implemented in this system and the file API are designed for bare PC applications.

Bare PC applications are based on the Bare Machine Computing (BMC) or dispersed OS computing paradigm [18]. This paradigm differs from a conventional approach as there is no underlying OS to manage resources. This means that the application programmer has to deal with system programming aspects. Resident mass storage is not used in a bare PC, so applications are stored on a portable device such as a USB drive or in the cloud. The file system interface is written primarily in C/C++ (with some assembly code) and runs as an application object (AO). An AO includes its own interfaces to the hardware [19] and the necessary OS-independent device drivers.

In earlier work [25], a lean USB file system for bare PC applications was discussed and an initial version of the file system was built and tested. That work showed the feasibility of developing a bare file system. However, that file system was not easy to modify or use with existing bare PC applications. This led to the development and implementation of an enhanced USB file system with a simple file API for bare PC applications, which is discussed in this chapter. The remaining of this chapter is organized as follows: section 3.2 describes the file API for bare PC applications, section 3.3 gives details of file system internals and section 3.4 presents functional operation.

3.2 File System API

In a bare PC application, code for data and file systems reside on the same USB. In addition to the application, the USB has the boot code and loader in a separate executable, which enables the bare PC to be booted from the USB. The application suite (consisting of one or more end-user applications) is a self-contained application object (AO) [17] that encapsulates all the needed code for execution as a single entity. For example, a Webmail

server, SQLite database and the file system can all be part of one AO. Since no centralized kernel or OS runs in the machine, the AO programmer controls the execution of the application on the machine. When an AO runs, no other applications are running in the machine. After the AO runs, no trace of its execution remains.

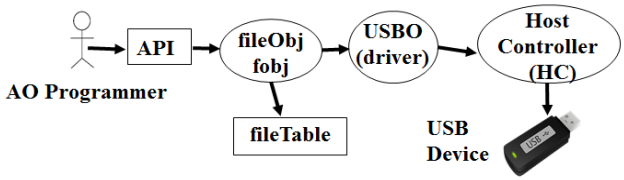


Figure 1 Bare machine USB file system

An overview of the USB file system for bare PC applications is shown in Figure 1. The simple API for the file system consists of five functions to support bare PC applications: createFile(), deleteFile(), resizeFile(), flushFile() and flushAll(). These functions provide all the necessary interfaces to create and use files in bare PC applications. The fileObj (class) uses a fileTable data structure, shown in Figure 2, to manage and control the file system. A given API call in turn interfaces with the USBO object, which is the bare PC device driver for the USB [20]. This device driver has many interfaces to communicate directly with the host controller (HC). The HC interfaces with USB device using low-level USB commands.

```
typedef struct {
    int index;
    int size;
    int startCluster;
    int numberOfCluster;
    int* cacheStartAddr;
    int* chacheEndAddr;
    int startSector;
    char* fileChar;
    //max of 64 bytes for filename
    char filename[MAX_FILENAME_LENGTH];
} fileTable;
```

Figure 2 FileTable Structure

```
int h;
h = createFile(fn, saddr, size, attr);
deleteFile(h);
resizeFile(h, size);
flushFile(h);
flushAll();
```

Figure 3 File API Functions

```
char *ptr;
char *readArray;
FileObj fobj;
h = fobj.createFile(fileName,
   &startAddress, &fileSize, attr);
ptr = (char *)startAddress;
for(i = 0; i < fileSize; i++)
    ptr[i] = 0; //write to file
for(i = 0; i < fileSize; i++)
    readArray[i] = ptr[i]; //read from
file
fobj.flushFile(h);
```

Figure 4 File API Usage

Figure 3 lists the file API functions, and Figure 4 shows an example of their usage. The parameters for the createFile() function are file name (fn), memory address pointer (saddr), file size (size) and file attributes (attr); it returns a file handle (h). The file handle is the index value of the file in the fileTable structure, which has all the control information

of a file. This approach considerably simplifies file system design as it can be used as a direct index into the fileTable without the need for searching. The deleteFile(h) function uses the file handle to delete a file. When a file is deleted, it simply makes a mark in the fileTable structure and its related structures such as the root directory and FAT table. The resizeFile() function is used to increase or decrease a previously allocated file size. Thus, an AO programmer needs to keep track of the growth of a file from within the application. The flushFile() function will update the USB mass storage device from its related data structures and memory data. An AO programmer has to call this function periodically or at the end of the program to write files to persistent storage. The flushAll() interface is used to flush all files and related structures onto the USB drive. Note that the programmer gets a file address, uses it as standard memory (similar to memory mapped files), and manages the memory to read and write to a file. There is no need for a read and write API in this file system. All standard file I/O operations are reduced to the list shown in Figure 3.

A significant difference between the bare PC file system and a conventional OS-based file system is that an AO programmer directly controls the USB device through the API. That is, a user program directly communicates with the hardware without using an OS, kernel or intermediary software. For instance, the createFile() function invokes the fileObj function, which in turn invokes the USBO function. The latter then calls the HC low-level functions.

In this approach, an API call runs as a single thread of execution without the intervention of any other tasks. Thus, writing a bare PC application is different from writing conventional programs as there is no kernel or centralized program running in the hardware

to control the application. These applications are designed to run as self-controlled, self-managed and self-executable entities. In addition, the application code does not depend on any external software or modules since it is created as a single monolithic executable.

3.3 File System Internals

Building a USB file system for bare PC applications is challenging. The system involves several components and interfaces, and it is necessary to map the USB specifications to work with the memory layout in a bare PC application and the bare machine programming paradigm. Details of file system internals are provided here to illustrate the approach.

3.3.1 USB Parameters

Each USB has its own parameters depending on the vendor, size and other attributes. Some parameters shown in Figure 5 are used for identification and laying out the USB memory map. These parameters are analogous to a schema in a database system and are located in the 0^{th} sector.

```
GetReservedSectors() 0xe - 0xf (0x0236)
GetNumOfFats() 0x10 (02)
GetNumOfSectorsPerFat() 0x24 - 0x27 (0x0ee5)
GetSectorsPerCluster() 0x0d  (08)
GetNumOfSectorsInPart() 0x20 -0x23 (0x003bafff)
GetClusterOfStartRootDir() 0x2c - 0x2f (02)
GetNumOfClustersInRootDir()  (third entry in FAT, 04)
GetFATEntryPoint()
GetDirectoryEntryPoint()
```

Figure 5 USB Parameters

3.3.2 **USB and Memory Layout**

Figure 6 displays the USB layout for a typical file system with 2GB mass storage. The boot sector contains many parameters that are accessed as shown in Figure 5. The reserved sectors parameter is used to calculate the start address of FAT1 table. The number of sectors per FAT defines the size of FAT1 and FAT2 tables, which are contiguous. The root directory entry follows the FAT2 table as shown in Figure 6.

The number of clusters in the root directory and number of sectors per cluster defines the starting point for the files stored in the USB. The root directory has 32 byte structures for each file on the USB. These 32 byte structures describe the characteristics of a FAT32 file system. The layout in Figure 6 shows two files prcycle.exe and test.exe. The first file is the entry point of a program after boot and the second one is the application. Other mass storage files created by the application are located after test.exe. The bare PC file system has to manage the FAT tables, root directory and file system data.

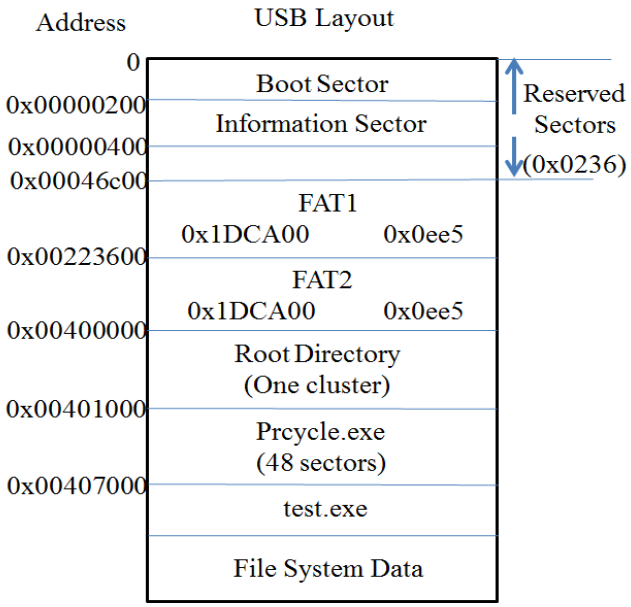


Figure 6 USB Layout

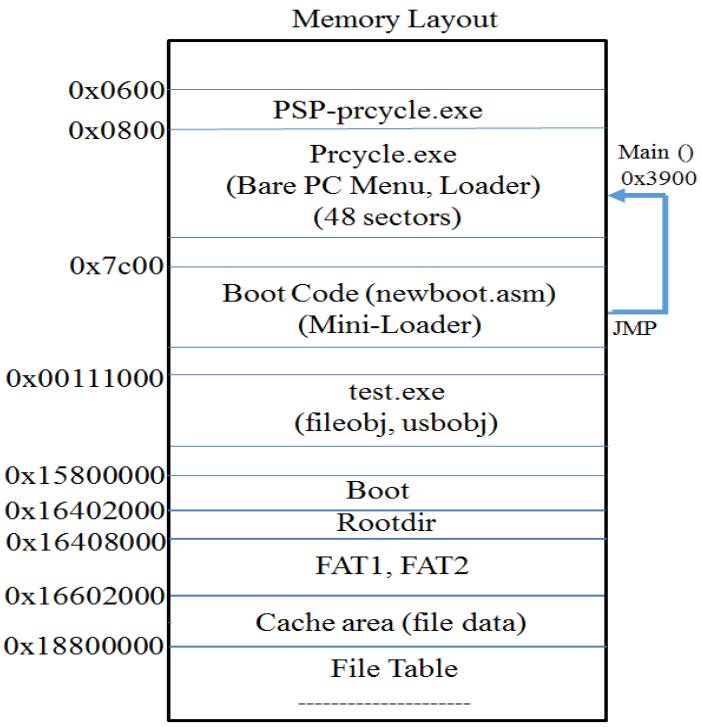


Figure 7 Memory Map

The USB layout and its entry points are used to map these sectors to physical memory. A memory map is then drawn as shown in Figure 7. During the boot process, the BIOS will load the boot sector at **0x7C00** and boot up the machine. This code will run and load prcycle.exe using a mini-loader. When prcycle.exe runs, it provides a menu to load and run the application (test.exe). The original boot, root directory and FATs as well as other existing files and data in the USB are also stored in memory to manage them as memory mapped files. The cache area stores all the user file data and provides direct access to the application program. In this system, the USB and memory maps are controlled by the application and not by any middleware.

3.3.3 File System Initialization

The Figure 8 illustrates the initialization process after the bare PC starts. During initialization, existing files from the USB are read into memory and file table attributes are populated. In addition, FAT tables and other relevant parameters are read and stored in the system. If the file data size is larger than the available memory, then partial data is read as needed and the file tables are updated appropriately. A contiguous memory allocation strategy is used to manage real memory. Because the file handle serves as a direct index to the file table, the file management system is simplified.

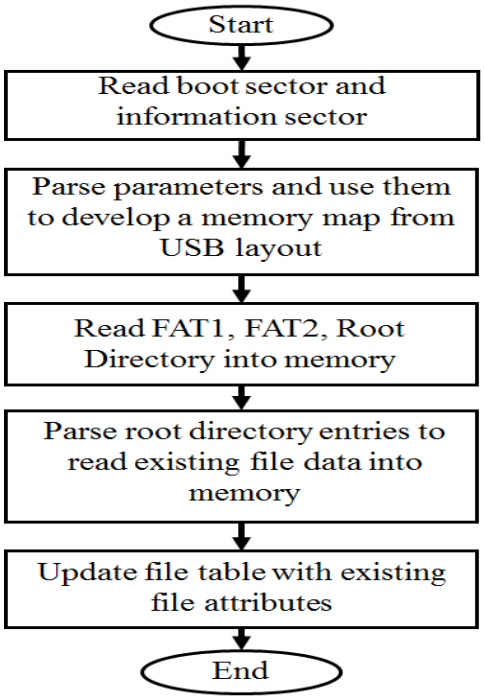


Figure 8 Initialization

3.3.4 File Table Entry (FTE)

The FTE is a 96-byte structure as shown below in Figure 9. The file name is limited to 64 bytes including name and type. 32-byte control fields are used to store the file control information needed to manage files. These attributes are derived from the root directory,

FAT tables and memory map. The file index is the first entry in the FTE and it indicates the index of the file table. The index is also used as a file handle to be returned to the user for file control.

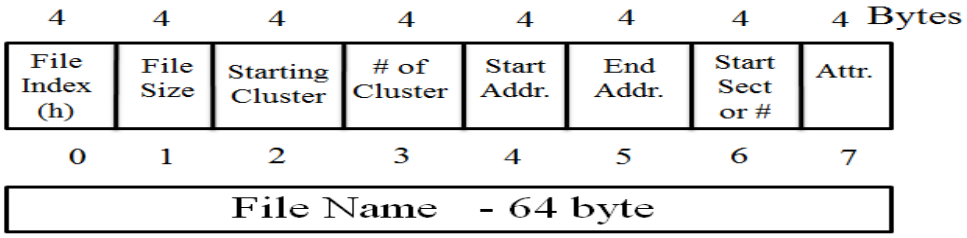


Figure 9 File Table Entry (FTE)

3.3.5 **File Operations**

The five file operations in the bare PC system use the data structures file table and device driver interfaces. The file system only covers a single directory structure. When createFile() is called, it first checks the file table for any existing file using the file name. If this file does not exist, a new file is created with the given file name and requested file size. Then an entry is made in the file table, memory is assigned, and the root directory and FAT entries are created for the file. When flushFile() is called, it updates the USB and the call returns the file handle, which is an index into the file table. Similarly, deleteFile() will delete the file from the file table and flushAll() will update the USB with all the USB data fields. The resizeFile() interface simply uses the same entry with a different memory pointer and keeps the data "as is" unless the size is reduced. When the size is reduced, the extra memory is reset. All API calls and their internals are visible to the programmer.

3.3.6 **File Name**

The file system supports both short and long file names. At present, long file names are limited to 64 characters by design since they introduce difficulties when creating the

root directory and file table entries. The FAT32 root directory structure also results in complexity that affects file system implementation.

3.3.7 System Interfaces

The USB file system runs as a separate task in the bare PC AO. The AO has one main task, one receive task and many application tasks such as server threads. The main task enables plug-and-play when the USB drive is plugged into the system. Each USB slot in the PC is managed as a separate task. Tasks and threads are synonymous in bare PC applications as threads are implemented as tasks in the system. Each event in the system is treated as a single thread of execution without interruption. Thus, each file operation runs as one thread of execution. There is no need for concurrency control and related mechanisms in a bare PC application for a single core. The files generated in the bare PC system can be read on any OS that supports FAT32 such as Windows, Linux or Mac.

3.4 Operations

The file system is written in C/C++, while the device driver code is written in C and MASM. The MASM code is 27 lines and provides two functions that read and write to control registers in the host controller. The fileObj code is 4,262 lines including comments (30% of the code), and one class definition. State transition diagrams are used to implement USB operations and their sequencing. For example, some of the state transitions occurring during the initialization process are shown in Figure 8. The fileObj in turn invokes the USB device driver calls shown in Figure 10.

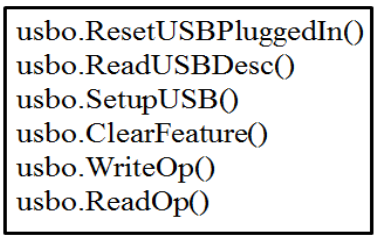
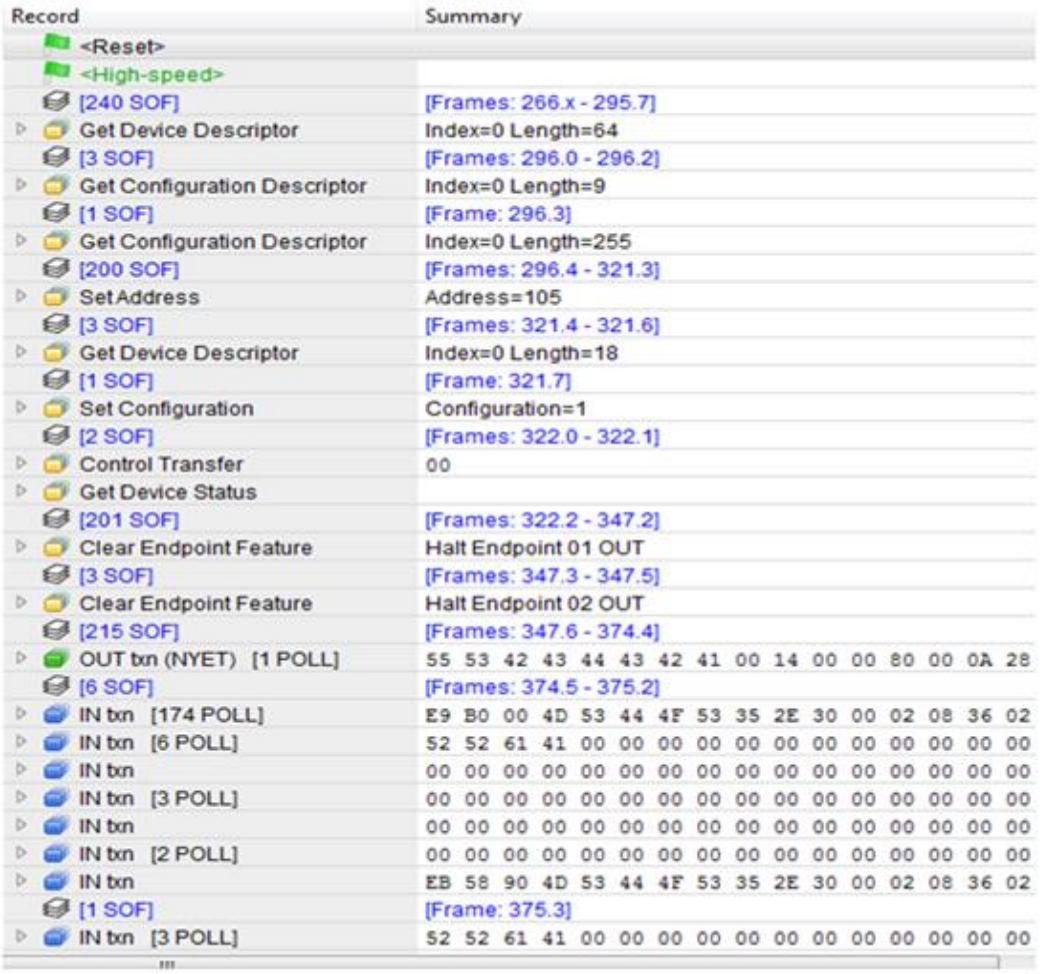


Figure 10 USB operations

File operations can be invoked anywhere in the bare PC application. The task structure that runs in the bare PC file system is similar to that used for bare Web servers [13], and runs on any Intel-based CPU that is IA32 compatible. Bare PC applications do not use a hard disk; however, the BIOS is used to boot the system. The file system, boot code and application are stored on the same USB. A bootable USB along with its application is generated by a special tool designed for bare PC applications. The USB file system was integrated with the bare PC Web server for functional testing.



Record	Summary
<Reset>	
<High-speed>	
[240 SOF]	[Frames: 266.x - 295.7]
Get Device Descriptor	Index=0 Length=64
[3 SOF]	[Frames: 296.0 - 296.2]
Get Configuration Descriptor	Index=0 Length=9
[1 SOF]	[Frame: 296.3]
Get Configuration Descriptor	Index=0 Length=255
[200 SOF]	[Frames: 296.4 - 321.3]
SetAddress	Address=105
[3 SOF]	[Frames: 321.4 - 321.6]
Get Device Descriptor	Index=0 Length=18
[1 SOF]	[Frame: 321.7]
Set Configuration	Configuration=1
[2 SOF]	[Frames: 322.0 - 322.1]
Control Transfer	00
Get Device Status	
[201 SOF]	[Frames: 322.2 - 347.2]
Clear Endpoint Feature	Halt Endpoint 01 OUT
[3 SOF]	[Frames: 347.3 - 347.5]
Clear Endpoint Feature	Halt Endpoint 02 OUT
[215 SOF]	[Frames: 347.6 - 374.4]
OUT txn (NYET) [1 POLL]	55 53 42 43 44 43 42 41 00 14 00 00 80 00 0A 28
[6 SOF]	[Frames: 374.5 - 375.2]
IN txn [174 POLL]	E9 B0 00 4D 53 44 4F 53 35 2E 30 00 02 08 36 02
IN txn [6 POLL]	52 52 61 41 00 00 00 00 00 00 00 00 00 00 00 00
IN txn	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
IN txn [3 POLL]	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
IN txn	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
IN txn [2 POLL]	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
IN txn	EB 58 90 4D 53 44 4F 53 35 2E 30 00 02 08 36 02
[1 SOF]	[Frame: 375.3]
IN txn [3 POLL]	52 52 61 41 00 00 00 00 00 00 00 00 00 00 00 00

Figure 11 Analyzer Trace

The operation of the bare PC file system is demonstrated by having two existing files (prcycle.exe and test.exe) on the USB along with the boot code. Small and large files are created by the application with file sizes varying up to 100K. To demonstrate file operations, four files were created and tested as described here in addition to the two files prcycle.exe and test.exe on the USB (after the program runs, there a total of six files on the USB). The data were read from the files and also written to them using the file API. A USB analyzer [42] was used to test and validate the file system and the driver. Figure 11 shows a sample trace from the analyzer that illustrates reset, read descriptors, set configuration

and clear. These low level USB commands are directly controlled by the programmer (they are a part of the bare PC application).

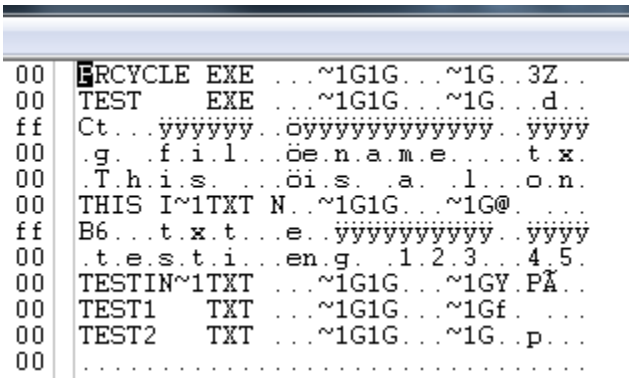


Figure 12 USB Root Directory

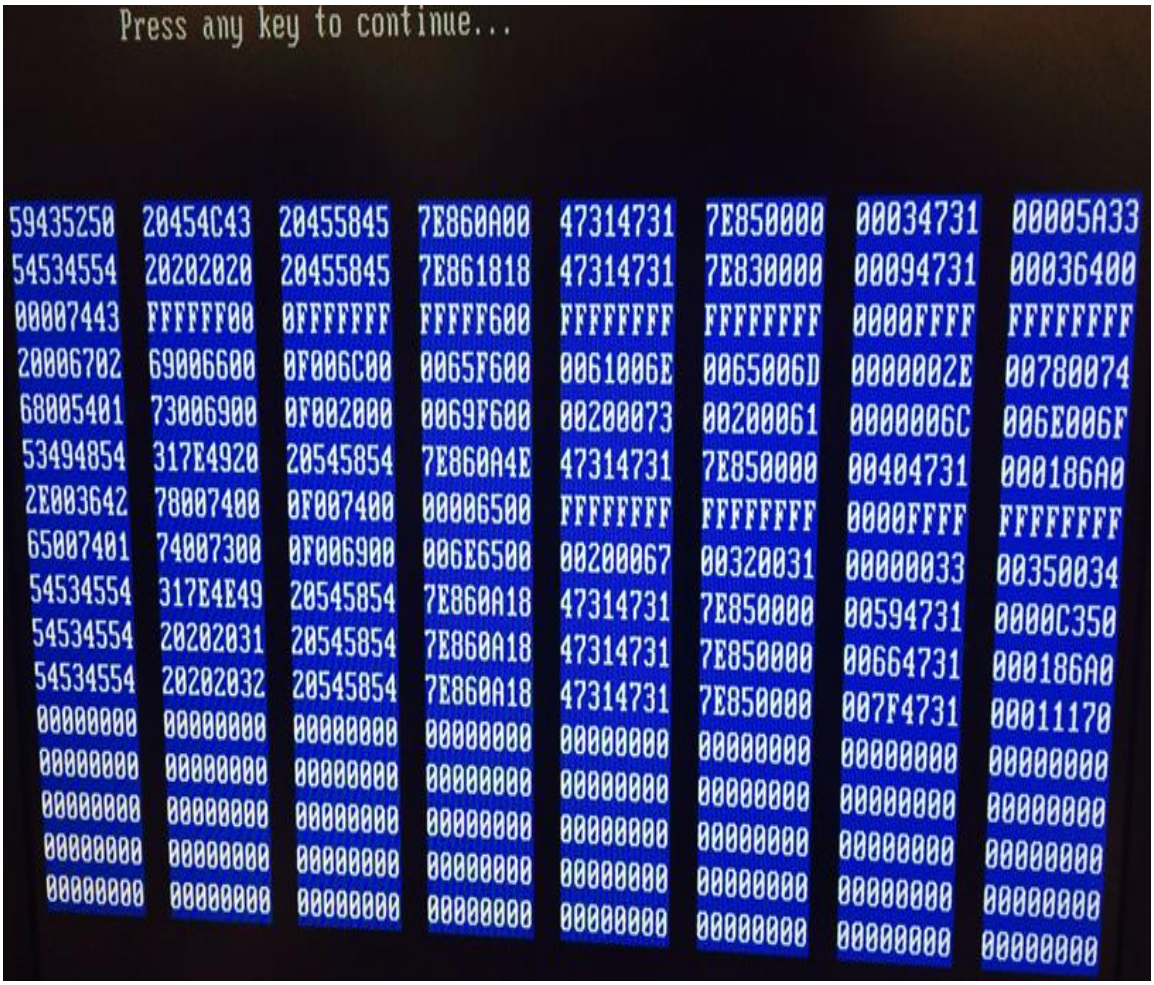


Figure 13 Bare PC Root Directory

Figure 14 shows the six files that exist on the USB displayed on the screen of a Windows PC. The four created files can be read from the Windows PC. Figure 13 shows the file system in the bare PC root directory in memory. This directory is used to update the files until they are flushed. Figure 12 shows the root directory entries on the USB after the program is complete. Figure 15 is a screen shot on the bare PC showing the four files (short and long) created successfully by the system. The bare PC screen is divided into 25 rows and 8 columns to display text using video memory. This display is used by the programmer to print functional data, and for debugging. The programmer controls writing to the display directly from the bare PC application, with no interrupts used for display operations.

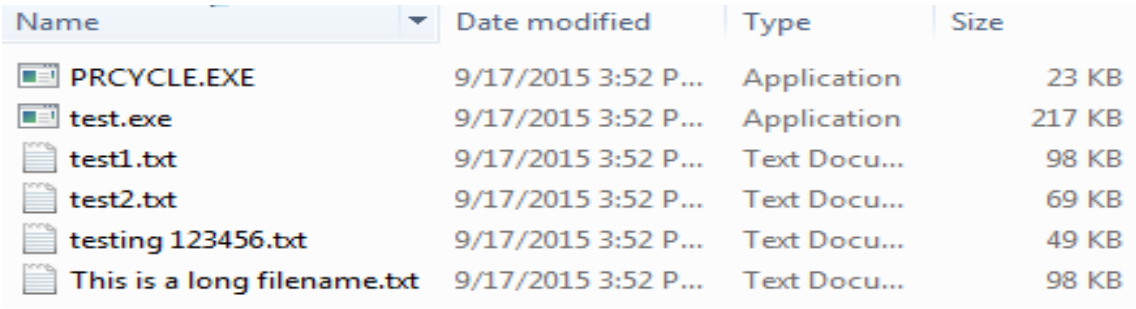


Figure 14 Windows Trace

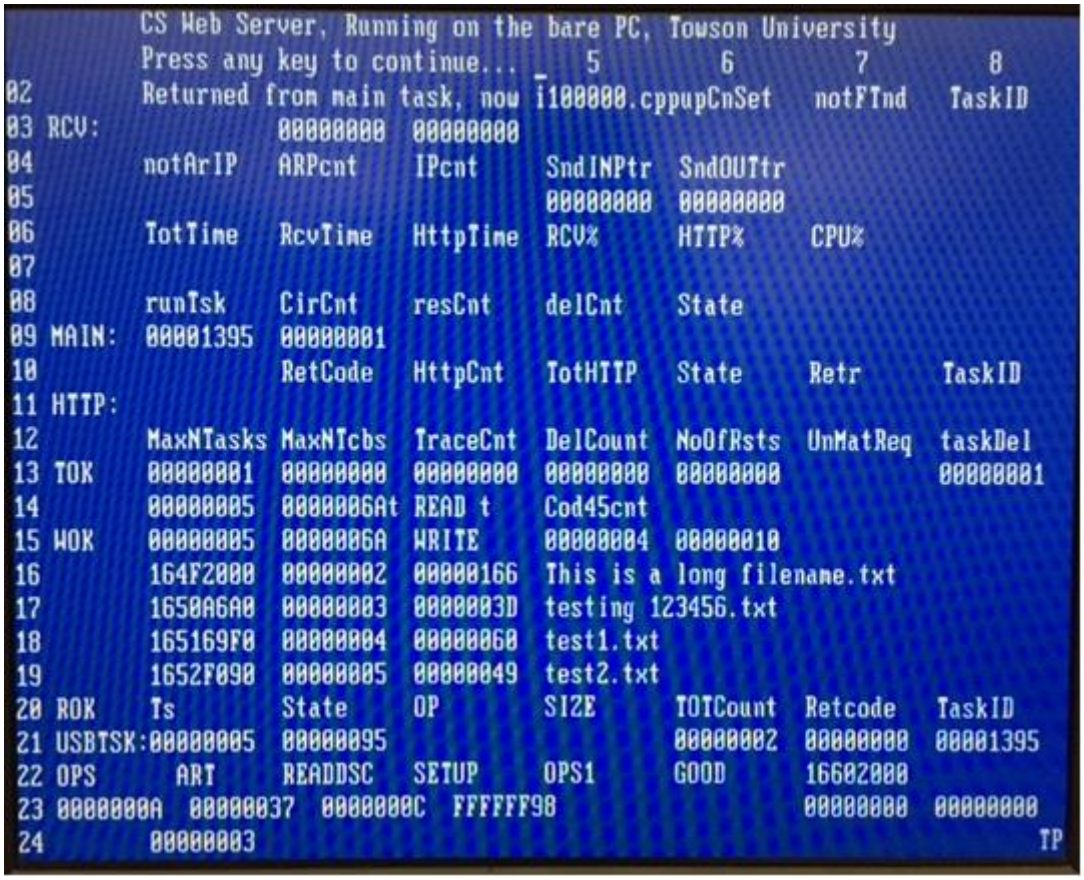


Figure 15 Bare PC Screen Shot

4 SQLITE FILE INTEGRATION

4.1 Interoperable SQLite based on the BMC Paradigm

SQLite is a self-contained, zero-configuration, stand-alone (not client/server) lean database management system [39] that is commonly used in mobile devices and often included in Web browsers. It runs on a standard operating system (OS) and has about 130K lines of code in an amalgamated version. SQLite has been transformed to run on a bare PC with no OS or kernel [28, 29] using its ":memory" option, where the database is stored in real memory with no standard file interfaces. The main objective in transforming SQLite to run on a bare PC is to make the database engine independent of any OS or its associated environments. This chapter discusses an approach to further extend the SQLite transformation with a standard FAT32 file system [26] and demonstrates the ability of the transformed SQLite database to interoperate between a standard OS and a bare PC system. It should be noted that conventional database management systems, including SQLite, use standard file systems that are provided by the host OS. A database created on one OS platform can then be ported to another using some middleware tools [34], which are themselves platform dependent.

The rest of this chapter is organized as follows: section 4.2 gives a brief overview of SQLite and the earlier transformation work, section 4.3 provides details on integrating a bare file system with SQLite and describes our implementation and interfaces, and section 4.4 demonstrates interoperability features of SQLite and includes basic performance measurements.

4.2 SQLite and Transformation Overview

The SQLite amalgamation package chosen has two source files “shell.c” and “sqlite3.c”. It has two header files and runs in Visual Studio (VS) on Windows. SQLite provides a command line interface and a file interface for user input. SQL queries can be run in single command mode or as a transaction. The database is stored using a standard SQLite file format in Windows or Linux. The transformation process as described in [29] eliminated 85 system calls and replaced them with direct bare PC hardware interfaces. These system calls can be classified into file, timer, data types, process, memory, and standard I/O. The file system calls were not replaced as the database was intended to run in main memory. The rest of the calls were replaced with equivalent bare PC calls, which are much simpler and do not require any centralized OS or kernel. These calls run in a single user mode along with its application. The SQLite transformation process to transform a Windows application to run on a bare PC provides a foundation to do the same for other applications, thus eventually constructing a bare PC application suite that can run on any desktop.

4.3 Design and Interfaces

Figure 16 shows an architectural view of a conventional operational environment and that of a bare PC configuration for running SQLite. We assume USB mass storage is used to store the SQLite database file based on the FAT32 file system. In a conventional environment such as Visual Studio (VS) on Windows, SQLite runs on top of the OS, which provides the necessary interfaces for virtual memory, file management and device drivers. A lean bare PC FAT32 file system [25] was designed before for bare PC applications. Similar lean and efficient FAT32 file systems [7] exist, but they are based on some

underlying OS and they are not directly applicable to bare PC applications. We enhanced the bare PC file system to be fully compatible with FAT32 specification so that it can interoperate with any OS platform.

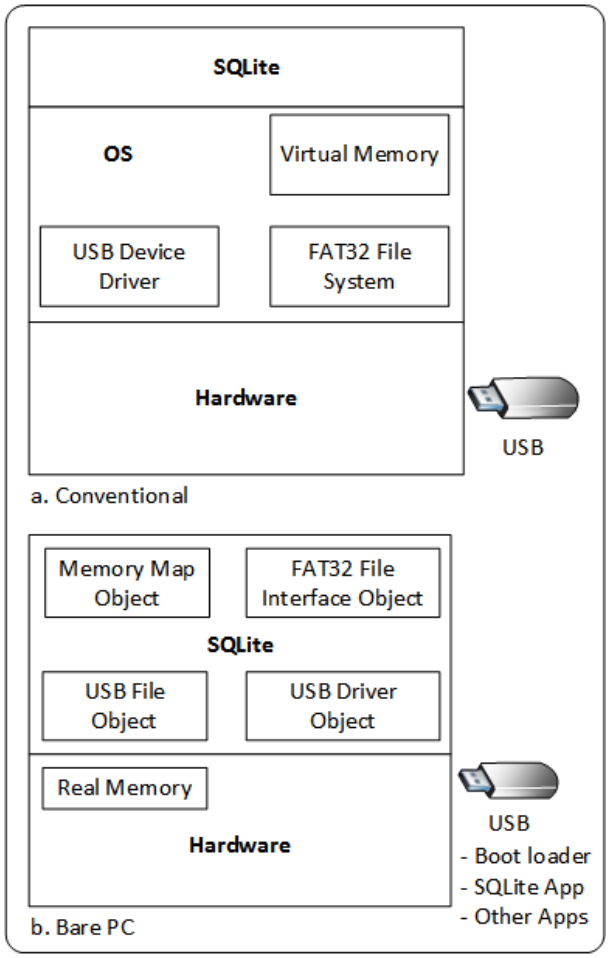


Figure 16 Software Architecture

As described in section 3.2, the enhanced file API we developed has five functions that are used to interface with the SQLite database. The createFile() function has a file name, memory address pointer, file size and file attributes. It returns a file handle. The returned file handle is the index value of the file in a file table structure shown in Figure 2, which has all the control information of a file. As illustrated in chapter 3, this approach

considerably simplifies the file system design as it can be used as a direct index into the file table without the need for searching.

In a bare PC environment, four objects MemObj, FileObj, UsbFileObj and UsbObj provide the complete functionality needed for the operation of SQLite. The MemObj provides real memory allocation and deallocation needed for SQLite (called by malloc() and free()). The FileObj provides the above file API. A USB file object (UsbFileObj) provides initialization, reset and plug-and-play features for all USB ports. Finally, a USB device driver object (UsbObj) provides driver functionality [20] specific to bare PC applications, which uses USB 2.0 standard specification [31], enhanced host controller specification [16], and USB mass storage specification [43]. These four objects are an integral part of SQLite and a database application running on a bare PC. A single executable includes all the bare PC code. SQLite runs as a separate task within the application. As SQLite is written in C, the code is wrapped in C++ to communicate with the object-oriented code in the bare PC. The USB in a Windows environment simply contains SQLite database, where as in a bare PC it contains SQLite database along with boot, load, SQLite applications and other applications if needed.

SQLite follows a special file format to store an entire database as a single file within a file system. For database file storage, SQLite doesn’t directly interface with the underlying operating system. Instead, as depicted in Figure 17 below, the host application is required to provide an object/interface that implements the SQLite Virtual File System (VFS) interface [40]. This object/interface is then responsible for translating calls made by

SQLite to the VFS interface into calls to the file system interface provided by the operating system.

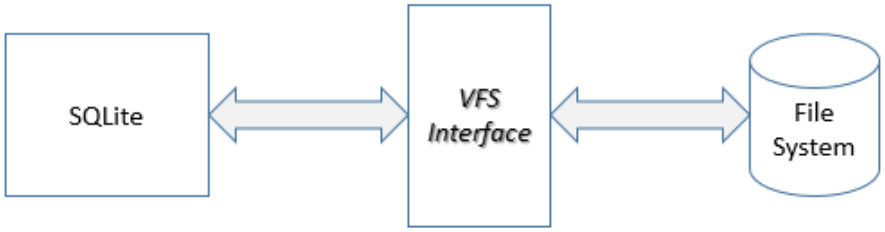


Figure 17 SQLite Virtual File System (VFS) Interface

SQLite provides a separate wrapper or virtual file system for a given OS interface. This wrapper approach in SQLite motivated us to develop a bare PC API that substitutes for a given OS and enables the development of a bare PC file storage system interface to SQLite. After studying the SQLite VFS interface in depth, the best approach was to start with an existing VFS implementation and modify it to use the primitives available instead of starting from scratch. So, the best starting point was with the OS_WIN VFS and modify it. The three structures changed in the SQLite code are "sqlite3_vfs", "sqlite3_io_methods", and "sqlite3_file". A brief overview of the changes is described below.

4.3.1 **sqlite3_vfs Object**

The sqlite3_vfs (virtual file system) structure, as shown in Figure 18, defines the interface between the SQLite core and the underlying OS [40]. In Windows OS, an instance of this structure illustrates the attributes needed for SQLite as shown in Table 2 left column. A total of 22 functions are used in this object. The equivalent bare PC functions are shown in the right column of this table. For the bare PC implementation, five functions are optional and not applicable, and five methods are provided with stubs as these are Windows

API calls. The rest of the functions are implemented for the bare PC. The most important method in this object is “bareOpen”, which is used to open/create a SQLite database file. In order to substitute our function, we created a “register_barevfs()” function as a wrapper around the SQLite’s *sqlite3_vfs_register()* and inserted it into the “sqlite3_os_init()” function, which initializes all OS parameters. Figure 19 illustrates a trace of SQLite control flow from the “open_db()” call to the “bareOpen()” function. This trace helps one to understand the database and file interactions in SQLite to implement the bare PC file system.

```
typedef struct sqlite3_vfs sqlite3_vfs;
typedef void (*sqlite3_syscall_ptr)(void);
struct sqlite3_vfs {
  int iVersion;            /* Structure version number (currently 3) */
  int szOsFile;            /* Size of subclassed sqlite3_file */
  int mxPathname;          /* Maximum file pathname length */
  sqlite3_vfs *pNext;      /* Next registered VFS */
  const char *zName;       /* Name of this virtual file system */
  void *pAppData;          /* Pointer to application-specific data */
  int (*xOpen)(sqlite3_vfs*, const char *zName, sqlite3_file*,
               int flags, int *pOutFlags);
  int (*xDelete)(sqlite3_vfs*, const char *zName, int syncDir);
  int (*xAccess)(sqlite3_vfs*, const char *zName, int flags, int *pResOut);
  int (*xFullPathname)(sqlite3_vfs*, const char *zName, int nOut, char *zOut);
  void *(*xDlOpen)(sqlite3_vfs*, const char *zFilename);
  void (*xDlError)(sqlite3_vfs*, int nByte, char *zErrMsg);
  void (*(*xDlSym)(sqlite3_vfs*,void*, const char *zSymbol))(void);
  void (*xDlClose)(sqlite3_vfs*, void*);
  int (*xRandomness)(sqlite3_vfs*, int nByte, char *zOut);
  int (*xSleep)(sqlite3_vfs*, int microseconds);
  int (*xCurrentTime)(sqlite3_vfs*, double*);
  int (*xGetLastError)(sqlite3_vfs*, int, char *);
  /*
  ** The methods above are in version 1 of the sqlite_vfs object
  ** definition.  Those that follow are added in version 2 or later
  */
  int (*xCurrentTimeInt64)(sqlite3_vfs*, sqlite3_int64*);
  /*
  ** The methods above are in versions 1 and 2 of the sqlite_vfs object.
  ** Those below are for version 3 and greater.
  */
  int (*xSetSystemCall)(sqlite3_vfs*, const char *zName, sqlite3_syscall_ptr);
  sqlite3_syscall_ptr (*xGetSystemCall)(sqlite3_vfs*, const char *zName);
  const char *(*xNextSystemCall)(sqlite3_vfs*, const char *zName);
  /*
  ** The methods above are in versions 1 through 3 of the sqlite_vfs object.
  ** New fields may be appended in figure versions.  The iVersion
  ** value will increment whenever this happens.
  */
};
```

Figure 18 SQLite Virtual File System Object

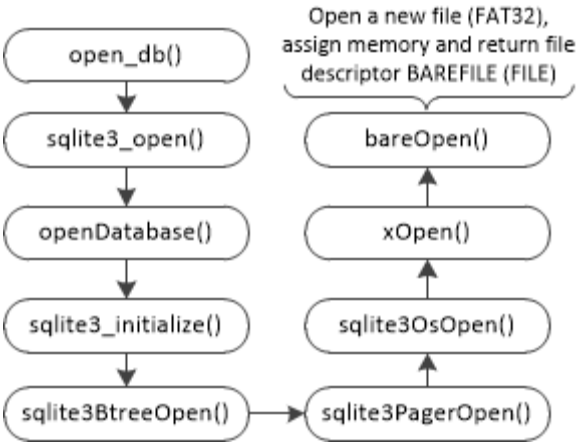


Figure 19 Trace of SQLite Control Flow

4.3.2 **sqlite3_io_method Object**

SQLite manipulates the contents of the file system using a combination of four types of file operations: create, delete, truncate and write. SQLite implements a Windows VFS called winVfs used to initialize and de-initialize the Windows operating system interface. The SQLite object named "sqlite3_io_method" consists of I/O functions as shown in the left column of Table 1. There are 17 functions in the Windows API and operations provided by the SQLite VFS for modifying the contents of the file system. Eight functions are not applicable to a bare PC environment. The other nine functions are implemented for bare PC applications. Table 1 shows these functions for the Windows API and the bare PC.

The "bareRead()" and "bareWrite()" functions read and write data from memory mapped data in real memory. In the bare PC file system, all the USB structures and data are memory mapped into main memory. The bareSync function sync the contents of the file to the persistent media. When the "bareSync()" function is called, the bare PC system flushes the database file to the USB. In addition, it also flushes the root directory, modified FAT tables and appropriate file data. A bare PC programmer uses the device driver directly

in the program to read or write to a USB flash drive. For performance reasons, it is advantageous to minimize the quantity of data read and written to and from the file system. Therefore, flush operation is only used when needed or when a transaction is complete. A bare PC user can control the frequency of updates to the mass storage. The bare PC file system is designed for optimal performance and reduces write operations as they are slow in a USB.

The “iVersion” is simply the version number of the VFS. “bareClose()” is used to close an existing file handle. In bare PC closing a file in this context means flushing the contents of the “bareFile” buffer to disk. This is a no-operation if the buffer of this particular is empty. The “bareTruncate()” function, as the name suggests, truncates a file. For the bare VFS, this is a no-operation because as of SQLite version 3.6.24, SQLite may run without working xTruncate() call, provided the user does not configure SQLite to use “journal_mode=truncate”, or use both “journal_mode=persist” and ATTACHed databases. All no-operation functions simply return SQLITE_OK which is defined as 0 – successful result. The “bareFileSize()” function write the size of the database file in bytes to **pSize* pointer, which is later used to check is the database file has been created or not, and returns the size of the file in bytes. Based on SQLite’s VFS implementation guide [40], the “xSectorSize()” and “xDeviceCharacteristics()” methods may return special values allowing SQLite to optimize file system access to some extent; however, for the bare VFS 0 (SQLITE_OK) is simply returned. That’s the case with the remaining functions that are not applicable to BMC.

Table 1 sqlite3_file I/O Methods

Windows	Bare PC
iVersion	iVersion -1
winClose	bareClose
winRead	bareRead
winWrite	bareWrite
winTruncate	bareTruncate
winSync	bareSync
winFileSize	bareFileSize
winLock	bareLock – N/A
winUnlock	bareUnlock – N/A
winCheckReservedLock	bareCheckReservedLock – N/A
winFileControl	bareFileControl – N/A
winSectorSize	bareSectorSize
winDeviceCharacteristics	bareDeviceCharacteristics
winShmMap	N/A (optional)
winShmLock	N/A
winShmBarrier	N/A
winShmUnmap	N/A

*Table 2 sqlite3_vfs *sqlite3_barevfs(void){}*

Windows	Bare PC
iVersion	iVersion -1
sizeof(winFile)	sizeof(bareFile)
MAX_PATH	MAXPATHNAME -
pNext	pNext - 0
“win32”	"bare"
pAppData	pAppData - 0
winOpen	bareOpen
winDelete	bareDelete
winAccess	bareAccess
winFullPathname	stub only
winDlOpen	stub only
winDlError	stub only
winDlSym	stub only
winDlClose	stub only
winRandomness	bareRandomness
winSleep	bareSleep
winCurrentTime	bareCurrentTime
winGetLastError	N/A (optional)
winCurrentTimeInt64	N/A
xSetSystemCall	N/A
xGetSystemCall	N/A
xNextSystemCall	N/A

4.3.3 **sqlite3_file Object**

The *sqlite3_file* object is a structure that represents an open file in the SQLite OS interface layer. We have extended this object in "bareFile" as shown below in Figure 20. The extended structure consists of the "_iobuf" structure which contains parameters that are needed to implement the bare PC file system. For example, "cacheStartAddr" is the real memory address provided by the memory object. The index value points to the entry in the file table. The "openFile" method also has an instance of a "bareFile" which is a "sqlite3_file" type. This bare PC file instance is linked with "bareio" which points to all the functions needed in a bare PC, which are the "sqlite3_io_methods" described in the previous section.

```
typedef struct _iobuf
{
        int cnt; /* characters left */
        char* ptr; /* next character position */
        int* cacheStartAddr; /* location of buffer */
        int flag; /* mode of file access */
        int fd; /* file descriptor */
        int index; /* file index */
        int size; /* file size */
} BAREFILE;

typedef struct bareFile {
        sqlite3_file base; /* Base class. Must be first. */
        BAREFILE *fd;   /* File descriptor */
} bareFile;
```

Figure 20 _iobuf and bareFile Structures

Each time a database file needs to be opened or created by SQLite, a call is made to the xOpen() function of the registered VFS object. This function takes an uninitialized sqlite3_file object as a parameter, and SQLite expects the customized xOpen() function, which in our case is *bareOpen()*, to initialize the supplied sqlite3_file object's pMethods data member to point to the sqlite_io_methods object. Upon return from the xOpen() call

execution, the sqlite3_file object can then be used SQLite in its internal calls to file operation functions listed in Table 1. Figure 21 shows the *bareOpen()* implementation.

```
/* Open a file handle. */
static int bareOpen(
  sqlite3_vfs *pVfs,      /* VFS */
  const char *zName,      /* File to open, or 0 for a temp file */
  sqlite3_file *pFile,    /* Pointer to bareFile struct to populate */
  int flags,              /* Input SQLITE_OPEN_XXX flags */
  int *pOutFlags          /* Output SQLITE_OPEN_XXX flags (or NULL) */
){
      bareFile *p = (bareFile*)pFile; /* Populate this structure */
      static const sqlite3_io_methods bareio = {
            1,                    /* iVersion */
            bareClose,            /* xClose */
            bareRead,             /* xRead */
            bareWrite,            /* xWrite */
            bareTruncate,         /* xTruncate */
            bareSync,             /* xSync */
            bareFileSize,         /* xFileSize */
            bareLock,             /* xLock */
            bareUnlock,           /* xUnlock */
            bareCheckReservedLock, /* xCheckReservedLock */
            bareFileControl,      /* xFileControl */
            bareSectorSize,       /* xSectorSize */
            bareDeviceCharacteristics    /* xDeviceCharacteristics */
      };
      .... // omitted some code.
      ... //full method implenetationn code available in Appendix

      if( flags & SQLITE_OPEN_MAIN_JOURNAL ) {
            sqlite3MemJournalOpen(pFile);
            return SQLITE_OK;
      }
      p->base.pMethods = &bareio;
      return SQLITE_OK;
}
```

Figure 21 bareOpen Method.

The implementation of the bare VFS is done in C code. The size of the new code added is approximately 1,300 lines including comments. The new SQLite runs on a bare

PC FAT32 file system. SQLite is thus made independent of any OS or kernel; it only requires the Intel x86 architecture.

4.4 Interoperability

The SQLite database that runs on a bare PC is interoperable with one running on VS/Windows. This enables one to create a database in Windows or on a bare PC and use it in either environment. The same can be done with database updates. This is also the case with other database management systems, where interoperability is achieved by porting the database management system to run on a different platform. For example, an Oracle DBMS running on Linux can be ported to run on Windows. Each such database system has OS-specific dependencies (e.g. Oracle Linux, Oracle Solaris, Oracle Windows, etc.). The bare PC SQLite database system in contrast is independent of any OS platform; it can run on any x86 based machine and create a database file (FAT32 format) that can be used by SQLite running on any platform. Also, one can easily design and implement interfaces that work with other file system formats for different CPU architectures.

As mentioned before, this approach enables OS dependencies in SQLite to be eliminated and serves as a first step to make other database management systems bare. In this dissertation, it was shown how a VS/Windows database can be used in a bare PC environment and vice versa. In future, one could use the bare PC SQLite to create and manage databases, and use the Windows OS to provide user interfaces. Database (and other) servers are naturally suited for bare PC or bare machine applications as they focus on a single monolithic executable and are easily tailored for the backend. This approach provides a novel architecture for future database management systems.

4.4.1 **Interoperability Demonstration**

This section describes experiments to demonstrate the inter-operability of SQLite. The measurements were conducted on Dell Optiplex 960 models with a 3.16 GHZ dual-core system. However, we only ran this on a single core processor to compare with a single core bare PC application. The SQLite database was run on Windows 7 with Visual Studio 2010 and also on a bare PC with no OS or hard disk. The USB flash drive contains the bare applications suite including boot and load programs. We also ran SQLite in addition to a Webserver application in a multi-threaded manner on the bare PC.

Initially, a USB is created with the bare application suite and no database file on it. This is a bootable and executable USB for the bare PC (it contains bare boot sector, loader, and application). The same USB is then used to save a SQLite database file created by VS in Windows. Figure 22 shows a screenshot on the Windows machine, where the database was created and tested. Here, one table (name: s5k) containing 5,000 inserts was edited in a file (5k.sql) formed as a single transaction. This file was read by SQLite using the ".read" command and executed. The ".tables" command shows the name of the table (s5k) in VS. One new record was inserted into the above table ("123 VS") and the "count(*) SQL statement" shows 5,001 records in the database. At this point, we use ".quit" from VS and obtained the created database "rkktest.sdb", which was saved in the USB.

This USB with the database file is used to boot up the bare PC. The bare PC, after initialization and loading, reads the database file into memory as a memory mapped file. It recognizes the existing database which came from VS/Windows environment. Figure 23 shows a bare PC screenshot showing the newly inserted record in VS ("123 VS"). Now,

we have inserted a new record “123 Bare” in the bare PC database. The new count(*) shows 5,002 records indicating that it loaded the database successfully and added a new record. Figure 24 shows the database activities performed in the bare PC; at the end it flushes the database. This database is used in VS to show the interoperability between VS and the bare PC. As shown in Figure 22, the count(*) shows 5,002 records and the “123 Bare” record. While interoperability is easily provided in OS environments, our approach shows how databases can be made operational in a bare PC without any need for an OS or related environments. The database running on the bare PC is interoperable with conventional OS based systems in addition to providing the same functionality with less complexity and more security.

```
[c:\Wm\works\SQC0A2~1\source]
CSBME03$ sqlite F:\rkktest.sdb
SQLite version 3.7.17 2013-05-20 00:56:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .read 5k.sql
Finished shell_exec in 454 milliseconds.

sqlite> .tables
s5k
sqlite> insert into s5k values('123 VS','VS-1','410-704-0010',
   ...> 'Baltimore','MD','21223','01/01/2016');
Finished shell_exec in 2F milliseconds.
sqlite> select count(*) from s5k;
5001
Finished shell_exec in 0 milliseconds.
sqlite> .quit


[c:\Wm\works\SQC0A2~1\source]
CSBME03$ sqlite F:\rkktest.sdb
SQLite version 3.7.17 2013-05-20 00:56:22
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> select count(*) from s5k;
5002
Finished shell_exec in 1F milliseconds.
sqlite> select * from s5k where sid='123 Bare';
123 Bare|Bare-1|410-705-0000|Balt|MD|21205|01/28/2016
Finished shell_exec in 0 milliseconds.
sqlite>
```

Figure 22 SQLite on Windows

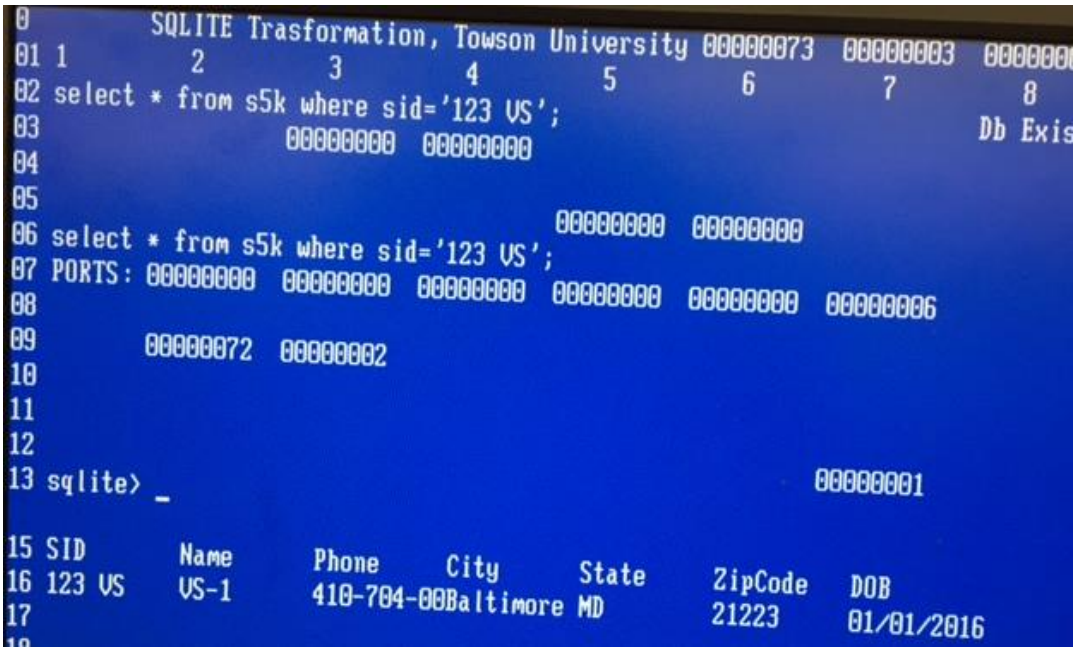


Figure 23 Bare PC reads database created on Windows (Visual Studio)

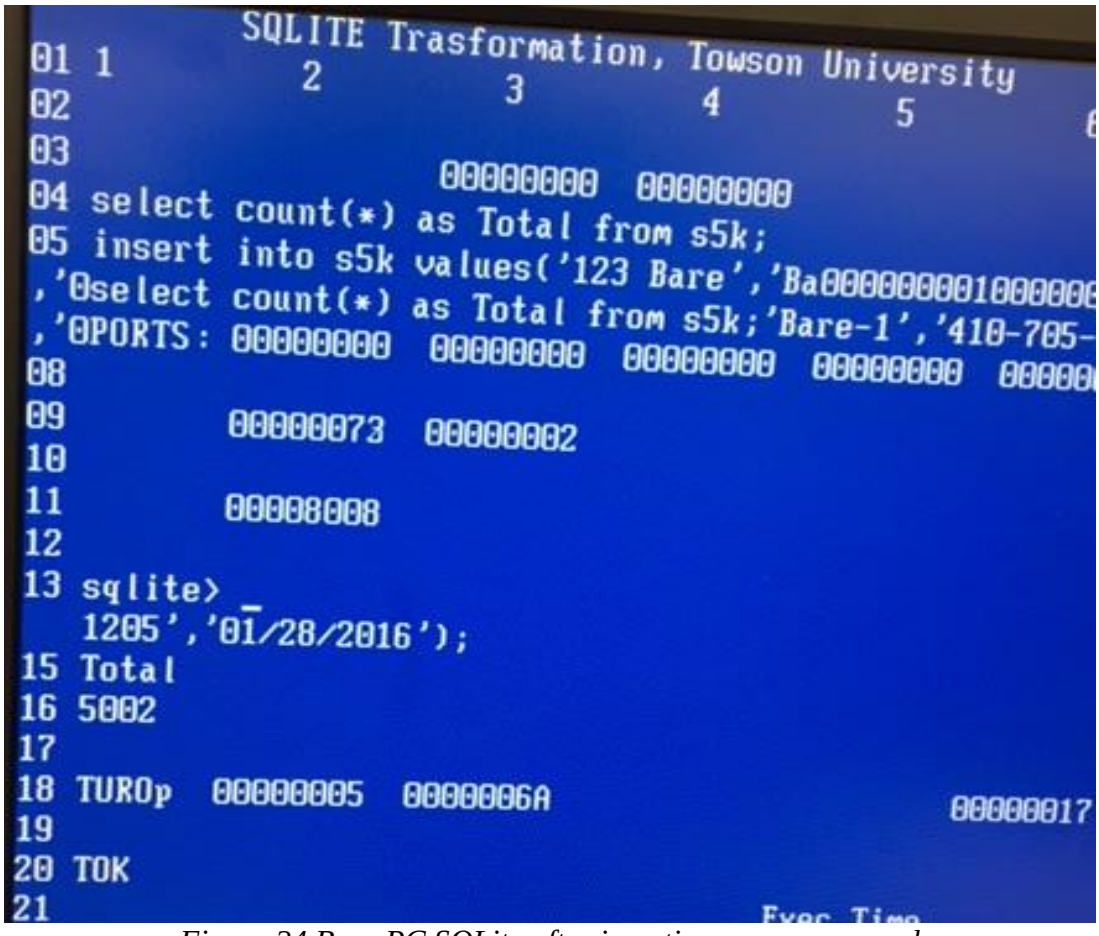


Figure 24 Bare PC SQLite after inserting one new record

5 USB MASS STORAGE SYSTEM FOR BARE PC

Mass storage systems require large storage area and they must be reliable, scalable and secure. Today's mass storage systems are usually hard disks with large capacity and usually they are prone to OS vulnerabilities [24]. Mass storage systems can be conventional files that are used in applications such as Web servers or database files that are created by a database system such as in SQLite. In either case, mass storage systems are based on file systems. There are many types of file system specifications such as FAT32 [26], NTFS [35] and extFAT [37]. This research focus on the FAT32 specification as it is simpler and easier to implement. As stated in earlier chapter, in bare machine computing software, an application contains everything needed to run on a bare PC. Thus, a device driver for USB flash drive [20] is also part of an application. The BMC application directly invokes interfaces to communicate with hardware without requiring any middleware to communicate between an application and hardware. In the previous chapter we discussed how SQLite was transformed to run on a bare PC using a FAT32 lean file system. So, there is a complete set of file system entities needed to design and build a mass storage system using BMC paradigm.

5.1 System Architecture

The bare file system and SQLite database file systems was extended to next level resulting in a mass storage system for BMC applications. These mass storage systems can be used in any BMC applications [4, 10, 14, 29, 33] previously developed. Figure 25 shows a system architecture for such applications. A USB flash drive is a complete file system by itself, which consists of boot, FAT (file allocation table), root directory and file data. The capacity of flash drives is very large and rapidly increasing. In fact, as main memories

(RAM) are getting cheaper and with larger capacity, it is conceivable to map flash drives into main memory. These memory maps provide direct, easier implementation and avoid the need for intricate memory management systems.

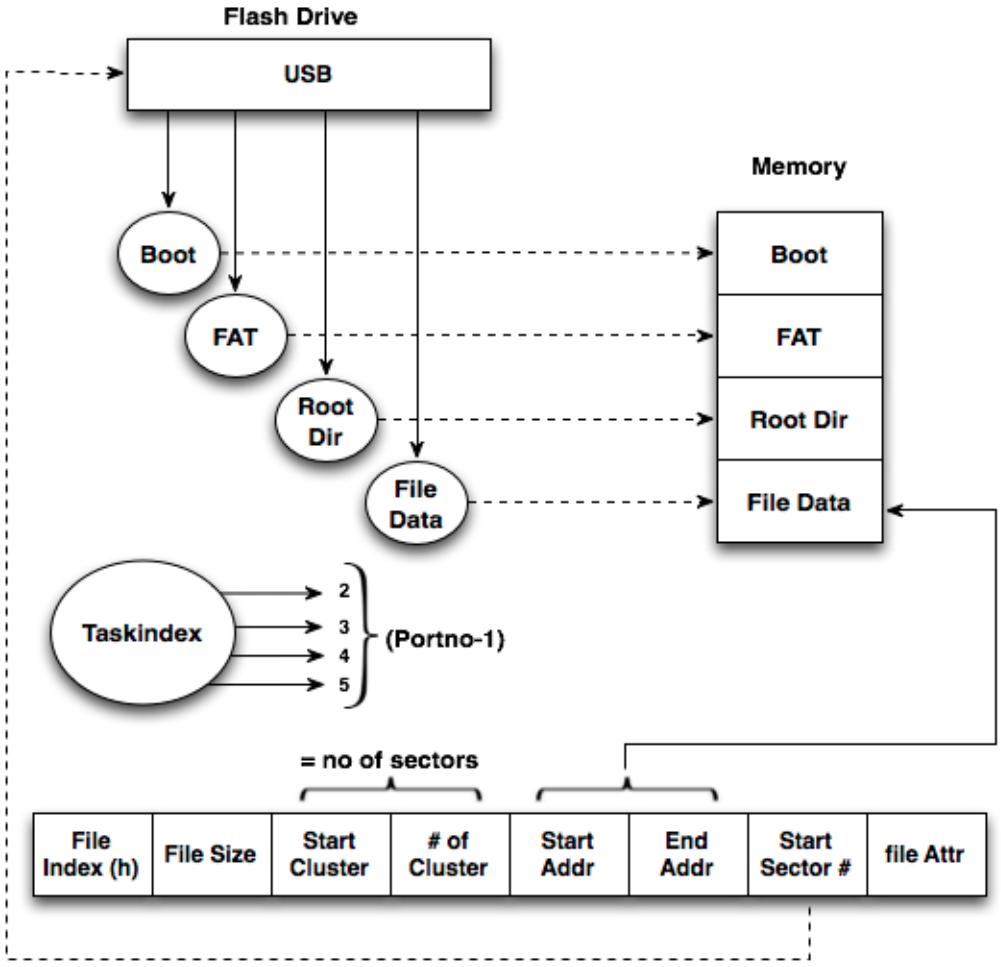


Figure 25 Mass Storage System Architecture

The file system resident on the USB flash drive can be modeled with a 32-byte data structure capturing its file attributes. This is similar to a 32-byte structure in a FAT32 root directory structure. Conventional and SQLite files can all be modeled with the same data structure. The "*file index*" field is the entry point used as an index into the root directory. The "*file size*" shows the number of bytes in the file. The "*start cluster*" and *"# of clusters*" show the starting point of the cluster on the physical media and the number of clusters

needed for the entire file. Usually, a cluster is defined as 8 sectors, but it can be configured to higher number as the file size increases. The "*start addr*" and "*end addr*" fields show the physical memory map in the system. In BMC, all memory is a physical memory thus avoiding virtual memory and paging overhead. The "*start sector number*" identifies the linear block address (LBA) needed to access the flash drive. The LBA addressing scheme on the USB provides a convenient way to address it, which is similar to addressing main memory. However, USB device needs to use SCSI (small computer system interfaces) [36] commands to access USBs. The "*file attr*" field defines the file permission attributes needed to store the file in the system. Each file in the system needs one 32-byte record, which provides all the control information needed to manage the mass storage system.

The mass storage system could be very large (in the order of terabytes) requiring large number of flash drives. This could pose a potential challenge in that the USB bus is limited to 127 devices per controller. Even though 127 seems a lot, it is important to note that individual hubs count as one device, and the root hub also counts as a device. Fortunately, such a potential issue can easily be overcome by having additional USB controllers.

A desktop usually provides a dozen USB ports, which can be extended furthermore up to 127 ports using a USB hub. These devices have unique port numbers (*portno)* varying from 1 thru 127 to address them. The system architecture allows to create a separate task for each port resulting in a "*taskindex*" 0 thru 126 (portno – 1), that manages a given port. In BMC application, one can define as many tasks as needed for ports, or a single task can also be used to manage multiple USBs. The mass storage management needed for BMC

applications can handle up to 4 GB of real memory as this is the limitation of a 32-bit address space. If more storage is required, then the resident memory can be swapped in and out of persistent storage on to the flash drives, without incurring any virtual memory or paging. A raw file structures in a temporary storage on the flash drives can be used for the swap space. We can also use a 64-bit CPU that can provide vast amount of main memory and large address space. The architecture for the mass storage system using bare PC is scalable, simple and extensible to new applications and technology. The design and implementation of this architecture is described in section 5.2 to demonstrate the feasibility of bare mass storage systems.

5.2 Mass Storage Design and Implementation

The design and implementation of BMC mass storage system poses many challenges and design issues. Integrating SQLite file system with BMC application requires C to C++ Bridge and interfaces. Multiple USBs in a desktop needs to host and manage multiple file systems in memory. The plug-play feature of USBs requires task structures that can detect the activity of flash drives and provide appropriate functionality. As the USB driver is part of the application suite, it requires timing related and device related knowledge to integrate with it. The device driver has to be managed by a separate file system task as it requires internal transactions and setup operations to perform USB operations. When SQLite is included as part of the mass storage system, it requires special handling of database functions as it requires user interface along with background operations. These design and implementation details are described below.

5.2.1 **Bridge between C/C++**

The BMC code is written as object-oriented C++ programs along with some C and Microsoft assembly code. A programing interfaces to address hardware takes a path from C++ function to a C function and then to an assembly function as needed. The C functions are used in C++ by defining them in *"extern"* block. This is a normal operation where C++ can call C code as you are going from strict type checking to a non-type checking code.

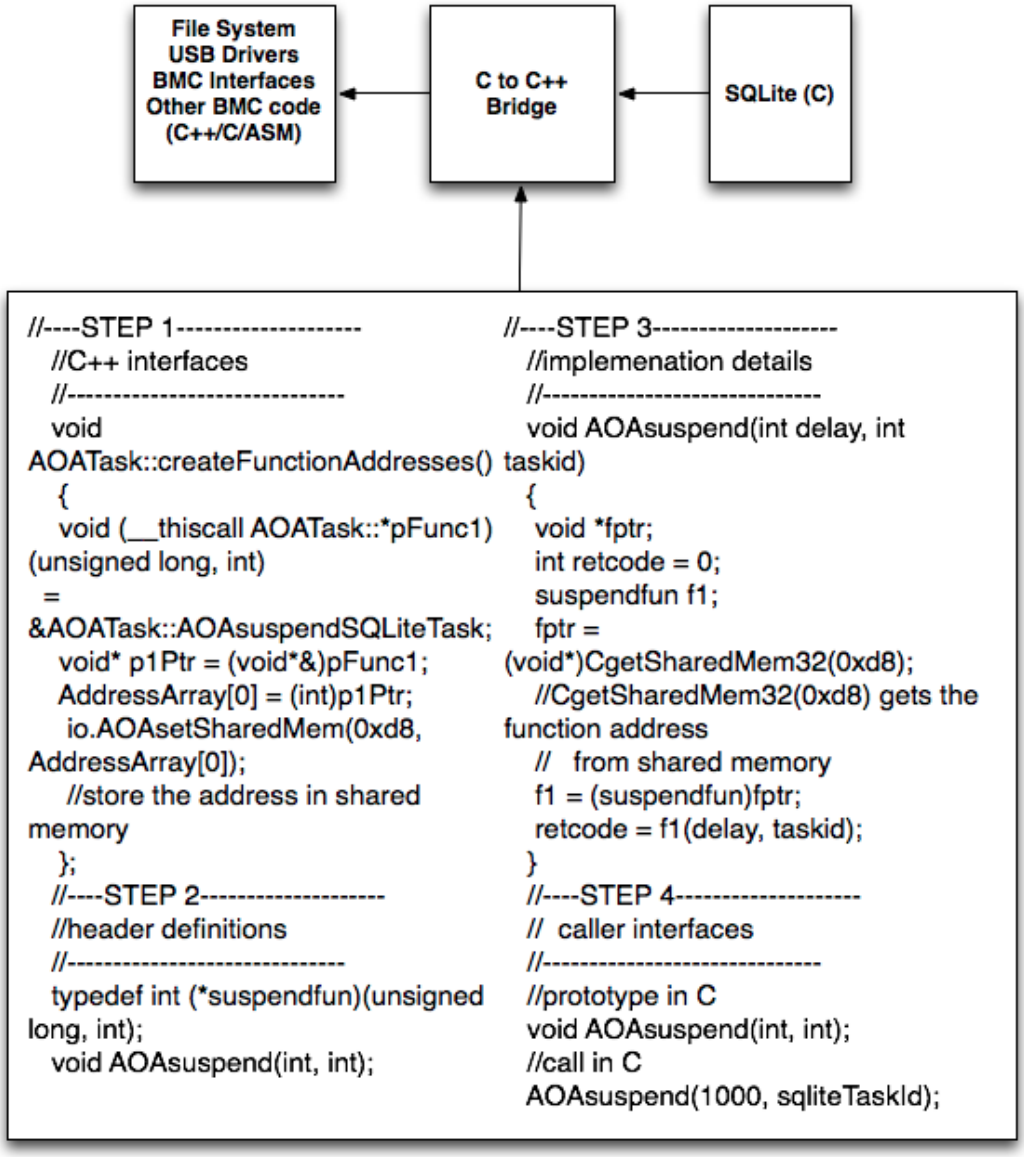


Figure 26 C to C++ Bridge

The SQLite code is written in C and it requires to communicate with BMC code for the file system, device driver and other BMC calls. Calling from C to C++ is not considered normal as it is violating the object-oriented principles and making the strict type checking of C++ language weak. The bridge as shown in Figure 26 provides a communication between C and C++ calls. There are variety of ways to implement such bridge as available on the Web [15], which requires a thorough understanding of C and C++ code and tailoring to the program needs. We modified the freely available code to suit our BMC code as shown in Figure 26. The C to C++ Bridge can be summarized in four steps. In Step 1, capture the C++ member function address and store it in a shared memory. In Step 2, define a C function header and a "*typedef*" for a dummy function. In Step 3, implement the C function, where the dummy function address is derived from the member function address in C++, which is stored in shared memory. In Step 4, simply define a C prototype, where it is needed and call the C function. Notice that the "*typedef*" function signature must be same as the C function call. We have defined many such functions in the SQLite database to call BMC C++ functions to integrate it with the BMC applications.

5.2.2 **USB Operation Flow Diagram**

The USB operation flow diagram is shown in Figure 27. Every time a USB is plugged in, it goes through a sequence of operations including: reset, read descriptors to capture device parameters, setup, and clear feature operations to enable its end points, test unit ready, and read/write. The control flow for each USB has to go through these steps before it can be used to read and write operations. Notice that some of these operations use SCSI commands, which are encapsulated in the USB commands [20]. The order of these operations are very important to make the USB operational. In addition, there are some

built-in delays needed for reset operation, which cause major issues in implementation. Determining these delay values and adjusting them as needed causes major design issues in the bare PC USB device driver.

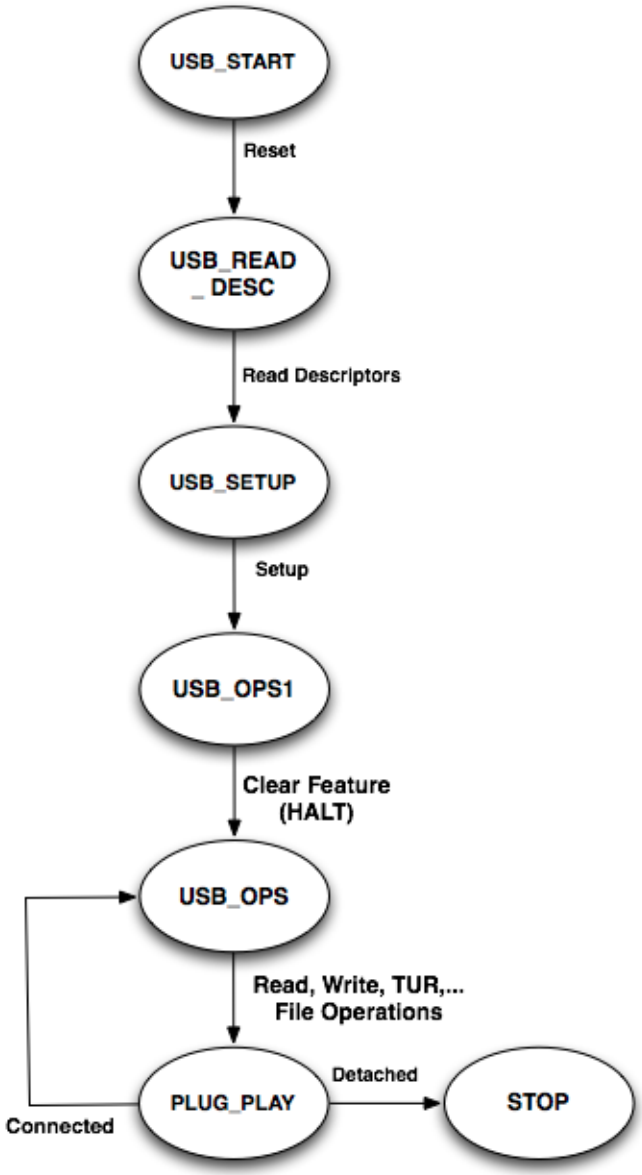


Figure 27 USB Operation Flow Diagram

The USB task diagram in Figure 28 shows the design and implementation of the adopted approach for managing the USB ports. There are two USB controllers in the Optiplex 960 desktop system. One of the controller provides four ports in the front of the

machine, which are used for testing this architecture. The second controller ports are in the back of the machine. A single task is designed to manage these four ports. These port numbers vary from 3, 4, 5, and 6. Their task indexes are one less than the port numbers (2, 3, 4, and 5). Each USB has its own file system which is resident on the flash drive. The control program is designed to check each port for its operation and functionality.

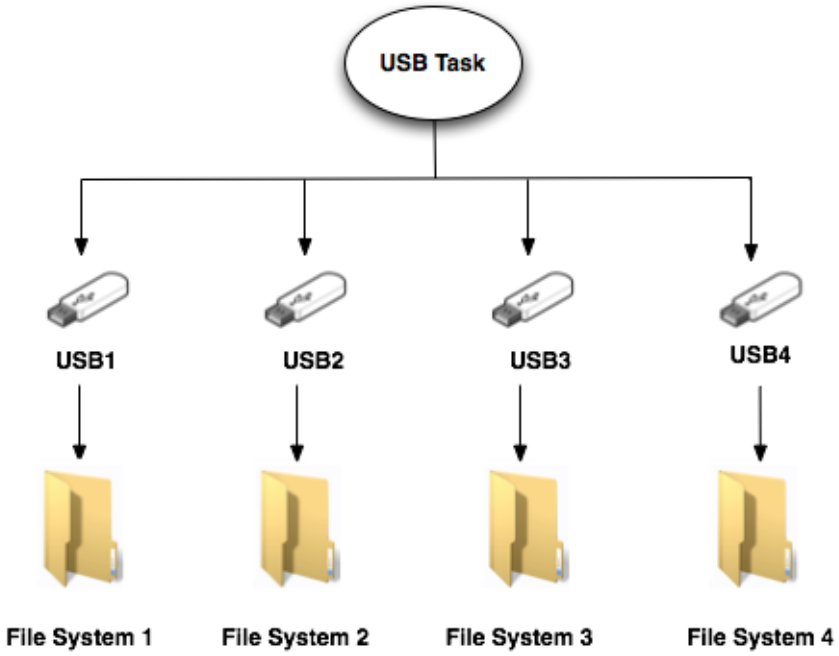


Figure 28 USB Task Diagram

We found that the USB controller behaves differently when there is only one USB in operation versus many of them plugged in. In the latter case, it requires a special reset known as mass storage reset. This is in addition to the operations as shown in Figure 27. A mass storage requires a sequence of USB operations test unit ready, read data, write data, clear feature and sense data. The single USB task will go through each device and perform read or write operations as needed by an application. This task will stay in the loop until it is terminated by the user.

5.2.3 **Task Structure**

The task structure as shown in Figure 29 illustrates the design and implementation details of a mass storage system in a BMC application. The application consists of a Web server, which requires HTTP tasks. A Webserver also requires resource files that can be used to send them to clients. It may also use a database SQLite for providing a dynamic content to clients. A USB task provides all USB interfaces to the user (it could be "***n***" tasks for "***n***" ports). A SQLite task manages all SQLite operations including user interfaces. This dissertation does not address the integration of USB file system or SQLite database files with Webserver system. However, the architecture of the mass storage system provides all the functionality needed for such integration.

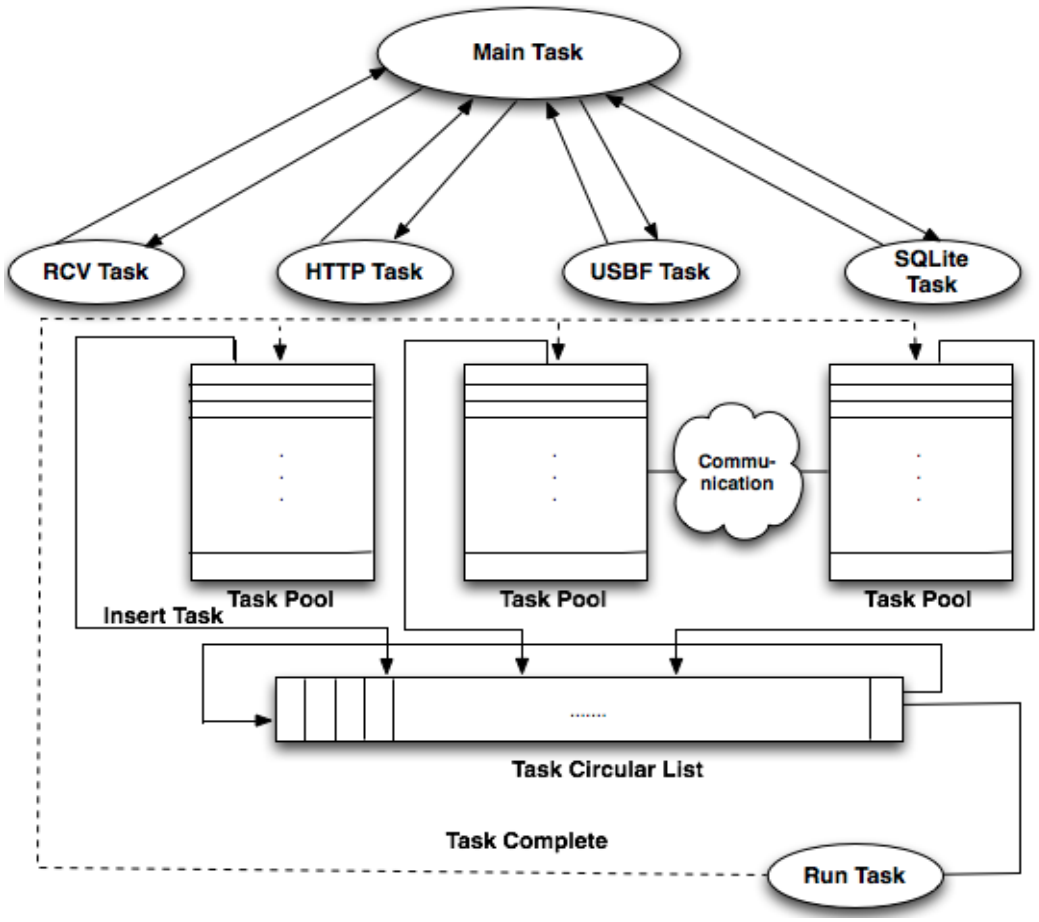


Figure 29 Task Structure

The "***Main task***" is the main task running in the bare PC all the time. When a network packet arrives, a "***RCV task***" runs to process the request. Similarly, when an HTTP data to be send to a client, then the "***HTTP task***" runs. Each task type has its own task pool created during the initialization process and kept in a stack. When a task is needed, it is popped from its appropriate task and placed in a circular list. The circular list tasks are processed in first come first serve basis. When a running task is complete, it will be pushed back on to its appropriate stack. When a task is waiting for an event, it is suspended and placed back in the circular list. Such task structure is very generic and simple in BMC and it is scalable as other types of task pools can be added in extensible manner.

5.2.4 **Class Flow Diagram**

The mass storage system consists of mainly three class objects as shown in Figure 30. The "***fileobj***" class provides all file API (application programming interface) [25, 41] that is lean and efficient in BMC. The "***USBFObj***" consists of plug-play functions of USB and interfaces to "***fileobj***." This object is managed by the USB file task. The file system API and all other interfaces can call "***USBObj***" interfaces for low level USB commands such as test unit ready, read, write, sense, reset, clear feature, etc. The "***USBObj***", which is the device driver for USB communicates with USB controllers and devices.

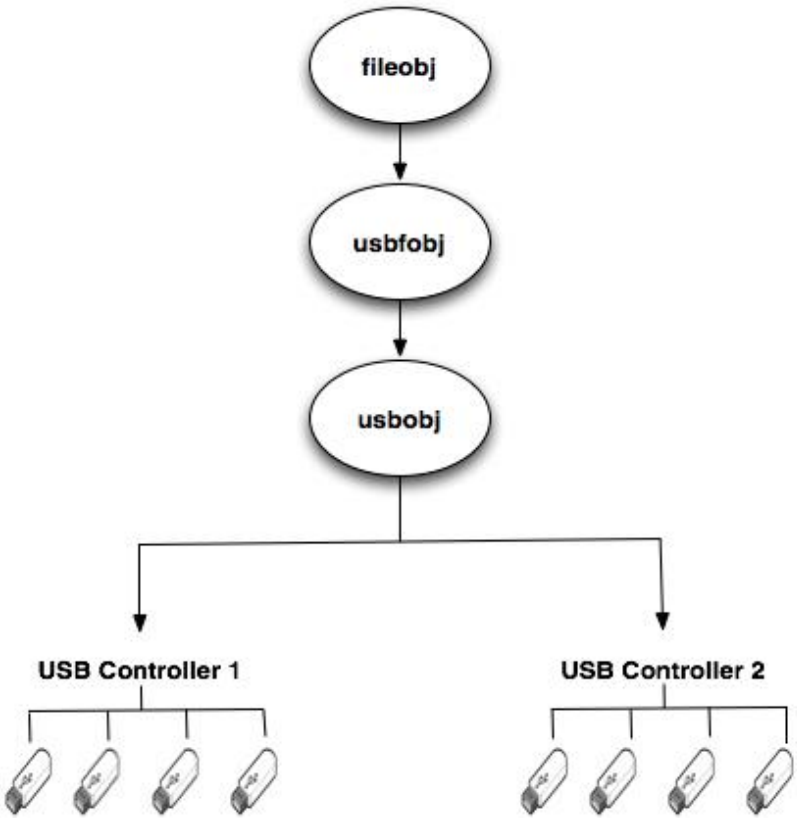


Figure 30 Class Flow Diagram

Each device has its own file system that needs to be managed by the mass storage system. In BMC, all the code needed for a given application suite is a single monolithic executable and runs itself without any need for external software or kernel. Thus, a BMC programmer has to manage all the intricacies of a given application suite. This makes the programmer systems as well as an application programmer. The application suite itself is independent of any external software and includes its application and execution environment. It only carries its needed interfaces and code and not the whole OS or kernel.

5.2.5 **Memory Map**

In BMC, the physical memory is managed by the application/system programmer. For a given physical memory, there is a need to organize a memory map at the design time. Figure 31 shows a typical memory map for the mass storage prototype. The first 1 GB

memory is used for Webserver and other BMC code including stack memory. The second 2 GB memory is used for USB File Storage including SQLite database files. Two more GBs can also be used for mass storage as needed. For four USBs, 256 MB storage is used for each consisting a total of 1 GB. 12 USBs can be mapped into a 4 GB physical memory. When large memory is needed, the mass storage can be swapped in and out of USB devices to cope with large storage.

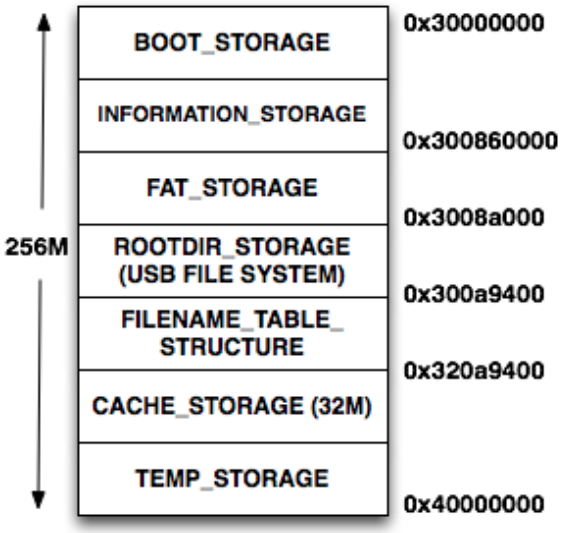


Figure 31 Memory Map for Each USB

5.2.6 **Inter-process Communication**

As SQLite and USB file tasks are two different tasks, there is a need to communicate between these two tasks to invoke file operations such as flush, read and write. As shown in Figure 29, the communication block is the inter-process communication element in the system. When SQLite is ready to flush, read, or write it issues a command to USB file task. It waits synchronously until the command is complete. We use shared memory (in real memory area; < 1M) to communicate between these two processes. A single lock is used to implement this mechanism.

5.2.7 **Implementation**

The mass storage system was implemented in C/C++ programming language with small amount of assembly code for the direct hardware interfaces. The design details, as mentioned earlier, were implemented in object oriented fashion. The FAT32 file system [25, 41] and SQLite code transformations [27, 28, 29] were previous done and simply used in the mass storage system. Similarly, the Webserver [14] mentioned here was also designed before and used in this prototype. The bare PC design is modular and extensible to add new features and design new applications. The bare PC design methodology was described in [22].

6 MEASUREMENTS AND ANALYSIS

6.1 SQLite Performance Measurement and Analysis

As one might expect, the amount of data read from the database file is minimized by caching portions of the database file in main memory. Additionally, multiple updates to the database file that are part of the same write transaction may be cached in main memory and written to the file together, allowing for more efficient I/O patterns and eliminating the redundant write operations that could take place if part of the database file is modified more than once within a single write transaction. The bare VFS implemented such strategy to improve performance.

After implementing the bare SQLite virtual file system and making it interoperable with the Windows OS platform, it was time to gather data and test database performance on both a bare PC and on a Windows PC. The analysis performed in this chapter shows some basic performance data collected to illustrate the leverage of a bare PC system. There is no OS independent performance benchmark tool available for SQLite. So, the performance analysis had to remain basic. A more detailed performance analysis of bare machine SQLite can be undertaken in the future.

The database queries in this study were done by using a single SQL statement (one at a time), or by collecting a set of SQL statements in a single transaction using BEGIN and COMMIT. Figure 32 shows the performance when the number of inserts into a single table is varied from 1,000 to 20,000 records. These inserts were placed in a file and run by using the ".read" meta-command in a single transaction. As the figure shows, SQLite on a bare PC SQLite performs much better than on VS/Windows as expected. The bare PC

performance improvements are attributed to less overhead in the hardware interfaces compared to system calls in OS. Figure 33 shows run times for inserting 10 – 100 records without transactions. Notice that the run times are very large as SQLite creates and updates a "journal" file for each SQL insert statement. In a transaction mode, it only flushes the file at the end of transaction. The bare PC system performs much faster for individual SQL statements as it updates the main database and journal in memory and does a final flush after running all the queries from a given file. The "journal" file approach provides more reliability as it provides frequent updates, but requires more time to run. The "journal" files are accessed multiple times depending on the number of SQL statements or transactions. In a bare PC system, the database is directly controlled by the user without OS involvement. In effect, bare PC SQLite is a simple user-centric application with lean functionality and greater efficiency.

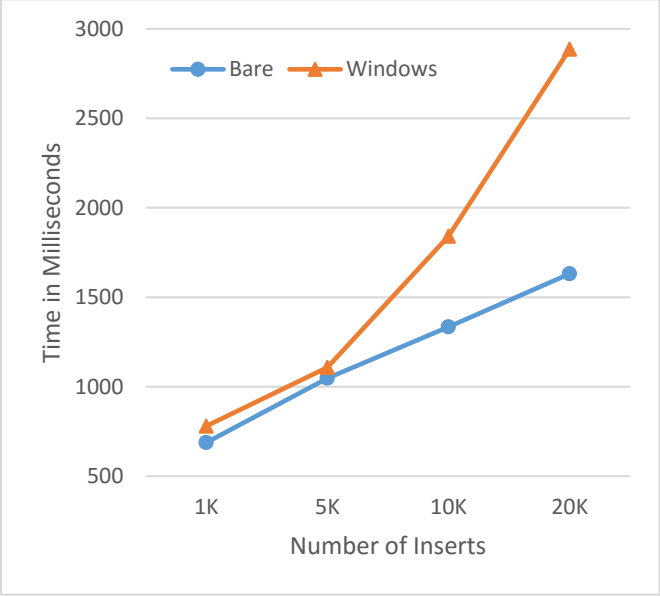


Figure 32 Inserts with transactions

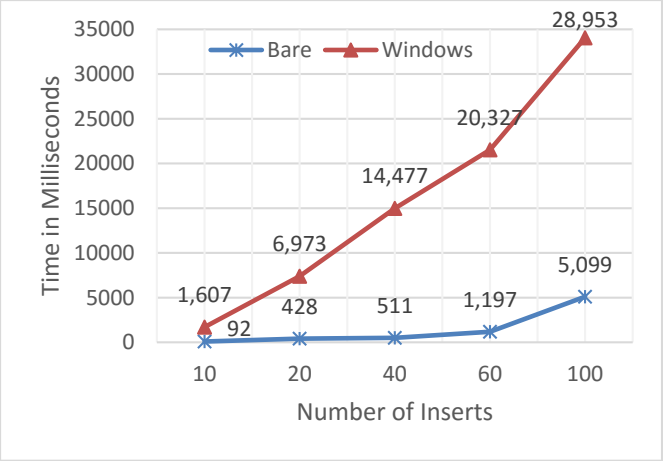


Figure 33 Inserts without transactions

6.2 File System Performance Measurement and Analysis

The mass storage system was tested on Optiplex 960 with 2 GB Verbatim USBs. Four USBs were plugged in to the first controller and file operations were performed sequentially on the port numbers 3, 4, 5, and 6. File flush, read, write, and other file operations were tested to validate the mass storage system. SQLite database files were also stored on the above four USBs using four different database files. The read and write operations for regular files and the database files are same as they use the same file system. We varied USB file sizes from 1 MB to 30 MB to measure write and read timings. Figure 34 shows write times using file system and also using raw files. A 30 MB file was written in **4.698** seconds. A 30 MB raw file (not using any file system) was written in **4.185** seconds. Thus, a raw file writes can provide **~12%** performance improvement in write operations. The BMC applications can use this leverage to use raw files in pure bare applications instead of a conventional file system. In a Windows desktop, the same file with 30 MB size took **8.2** seconds to write on the USB. It indicates that the bare PC file system can perform much better than the Windows file system. In the bare PC, a 30 MB file was read in **2.351** seconds. The same file was read as a raw file in **1.762** seconds as

shown in Figure 35. This is a **33%** improvement in raw read versus a file system read. The functional operation of mass storage and file system demonstrates a greater potential for bare PC or bare machine systems. The bare PC systems are efficient, lean and contain small footprints for executable files. The executable file size for the mass storage system is about 252 KB including Webserver and other BMC code.

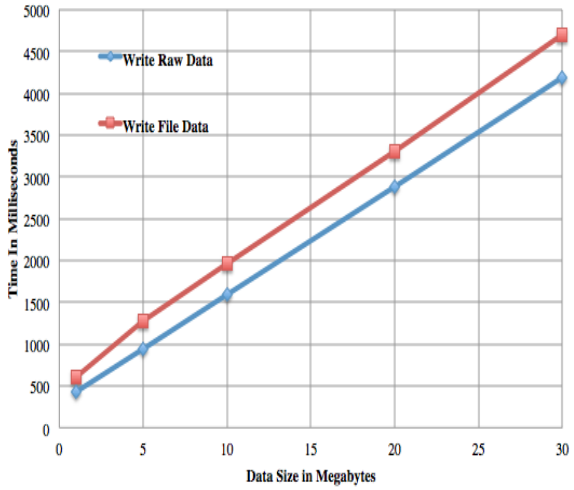


Figure 34 Write Raw Data/File

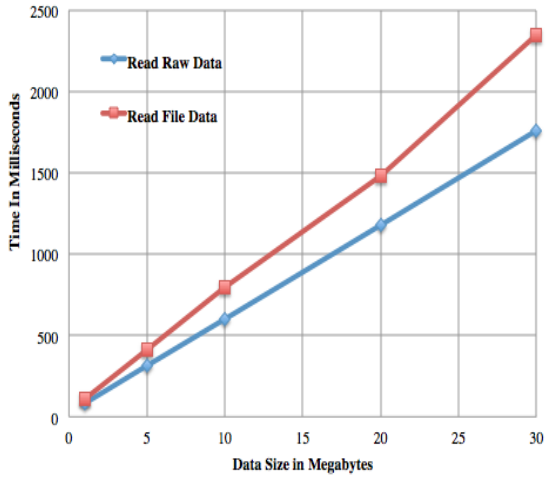


Figure 35 Read Raw Data/File

7 SIGNIFICANT CONTRIBUTIONS

The enhanced file system developed can be used to integrate with existing bare PC applications. The file system interfaces developed for SQLite can be used to extend the concept to other database management systems. The file system interfaces and the database file interfaces together will constitute a basis for mass storage systems. When a mass storage system can run on a bare PC as demonstrated in this thesis, it paves the way to make other computing devices to use bare mass storage systems thus eliminating all dependencies on execution and operating environments. Thus, one can carry a mass storage system in a removable device and use it on any bare machine.

The inter-operability between conventional and bare PC database systems provides a new avenue to build database management systems. A user interface can be implemented in a conventional system and its backend of a database can be hosted on a bare PC or a bare machine. This shields the database from OS vulnerabilities and thus makes it more secure. This thesis provided a bare file system for SQLite, which can be extended to other database management systems.

The file system offers a simple and programmer centric application programming interfaces. This allows direct hardware interfaces from an application thus avoiding semantic-gap between an application and hardware. Thus, a given application can only perform intended functions instead of providing a universal and open ended logic. As it eliminates execution and operating environments in mass storage systems, if offers pervasiveness across computing devices thus homogenizing mass storage systems.

8 CONCLUSION

Conventional file systems are OS-dependent and cannot be used with bare PC applications. So, we designed and implemented a novel FAT32-based USB file system for a bare PC. This dissertation describes the implementation of a novel bare machine USB file system designed for applications that run without the support of any OS environment/platform, lean kernel or embedded software. It presents a file API for bare PC applications. The file system enables a programmer to build and control an entire application from the top down to its USB data storage level without the need for an OS or intermediary system. The implementation can be used as a basis for extending bare PC file system capabilities in the future. The file system can be integrated with bare PC applications such as Web servers, Webmail/email servers, SIP servers and VoIP clients.

In addition, this dissertation presents a SQLite database with a file system that runs on a bare PC and demonstrated the interoperability of OS-based and bare PC database systems. The design and implementation details show how the bare PC file system interfaces to the SQLite virtual file system and highlights the simplicity of bare PC applications. The performance results suggest the feasibility of building scalable bare database management systems. The coupling between the OS based and bare machine database systems can be leveraged to design more secure database systems that do not have OS-related vulnerabilities. Future work based on this research can explore the potential benefits and tradeoffs in bare machine database systems and novel bare machine applications that use an OS-independent database and file system. Also, the concept of using multiple USB devices to store data on bare PC can be researched further to extend the concept to RAID and other secure applications.

Finally, we described a mass storage system architecture, design and its implementation. Implementing a bare machine mass storage system is not trivial because it does not use any standard system libraries and requires integrating the USB driver and FAT32 file system with the bare PC application. Large files and SQLite database files were used to demonstrate the feasibility of this architecture. Four USB flash drives were used to validate the design and measure some performance data. USB file write and read operations and their timings for large files were measured. Also, large raw file write and read timings are compared with the file operations. It is shown that raw write and reads can be used to gain higher performance in bare PC systems. The mass storage system described in this paper is lean, simple, scalable and more secure as it does not inherit any OS vulnerabilities. As the code is simple and lean, it also follows a minimalist approach to reduce security vulnerabilities. This system is also user centric and runs on any x86 based architecture, in bare mode. Further research in this area is needed to use these systems in big data applications and cloud storage.

APPENDIX

A. BMC Development Environment

Coding in the BMC laboratory is done primarily in C/C++ with the exception of small assembly language (ASM) code for boot, load and device drivers. SQLite is written in C; so is the bare virtual file system that provides file system interface for SQLite on bare PC. We use Visual Studio and Vim – Vi Improved editor for development. We use batch files to compile and link the lean file system, mass storage, SQLite, USB driver modules with the necessary bare PC code. Visual Studio C++ compiler (batch mode), MASM 6.11 assembler, and Turbo assembler compilers are used to create executable modules. There are other batch files also in place to compile and link the boot and loader programs with the resulting application(s) modules.

C/C++ Code Compilation

The SQLite C source code files are compiled using the following script in a batch file.

```
rem needs microsoft visual c++ environment.
rem run the batch file msdn.bat
erase *.obj
erase *.*~
erase *.lst
erase shell.exe

call asm.bat  rem // asm.bat is a batch file that compiles the assembly codes

..\bin\cl /c /FA /Zl /Os /Gy /GS- /F64000000 shell.c sqlite3.c aaa.c aaab.c msqlite.c
```

Figure 36 C/C++ Code Compilation

Assembly Code Compilation

All assembly code compilation is done with the batch file below using MASM 6.11 assembler.

```
rem cls
rem needs MASM 6.11 Environment
rem Run the Batch file MASM.BAT

cls
..\bin\ml /c /Cx /Fl    asmfilesb.asm
```

Figure 37 Assembly Code Compilation

Linking Objects

The following batch file was used to link all modules in the bare PC system. Notice that this linker batch file uses much more object files than needed for SQLite and the mass storage system. This is because we tested the lean file system with other bare PC applications, i.e. web server.

```
cls
rem Needs Microsoft Incremental Linker (Microsoft Visual C++ Linker)
rem To link in 32-bit we need to use the above said linker
rem Run the Batch file MSDN.BAT
rem dir shell.exe

erase ..\dosclib\*.obj
call cplib.bat

rem merge .bss to data needed in 64 bit mode, otherwise static variables have garbage values

..\bin\link /MAP /BASE:0x00000000 /NODEFAULTLIB /OPT:NOREF
/MERGE:.rdata=.data /MERGE:.bss=.data /STACK:32000000 /LIBPATH:"..\doscLIB"
/ENTRY:main test.obj aoatask.obj apptask.obj aoaprotected.obj runTask.obj
runTaskasm32.obj etherobj.obj cisupport.obj isupport.obj chkstk.obj wcirlist.obj wstack.obj
wstack1.obj wstack2.obj wttrace.obj wlist.obj ARPObj.obj IPObj.obj rand.obj tcpobj.obj
parserobj.obj fhashindex.obj udpobj.obj ftpobj.obj ftopobj.obj intexception.obj cfiles.obj
asmfiles.obj thashindex.obj wserverlist.obj mswitchasm.obj modeswitch.obj page32test.obj
usbobj.obj usbsupport.obj cusbsupport.obj fileobj.obj usbfobj.obj aaa.obj aaab.obj
asmfilesb.obj msqlite.obj shell.obj sqlite3.obj
dir  test.exe
```

Figure 38 Batch file to Link Objects

Make File

The following batch file is used to create a bootable USB in the BMC environment. We use a handy utility tool named ***rwhd.exe*** developed by Fred Ackers, a former doctoral student, to write the boot code to the USB. This tool is used in conjunction with another handy utility named ***PEfmt.exe*** to calculate the entry points in USB for prcycle.exe and test.exe and few other parameters. ***PEfmt.exe*** was developed by Hojin Chang, also a current doctoral student.

```
call asmb2.bat
call c.bat
format f: /FS:FAT32 /q
copy prcycle.exe f:
PEfmt .\webserver\test.exe test1.exe 400h 0 0
erase test.exe
copy test1.exe test.exe
copy test.exe f:
rwhd -m 6 newbootf32.bin
```

Figure 39 Make File

B. How to Boot, Load and Execute a Bare PC Application

In order to load and execute the bare PC application, i.e. SQLite, one needs to place a boot and loader program along with the bare PC application on a bootable device such as a USB flash drive. The boot program enables bare PC applications to be executed after booting is completed. One has to follow standard PC boot procedures and power up the PC with the boot device in the boot drive. During the boot process, the loader program will load the AOA Interface Menu, shown in Figure 40, into main memory (RAM) for execution. Using this interface menu, one can load and execute an application from the

same boot device. The single boot device such as flash drive has all the elements needed to run an application.

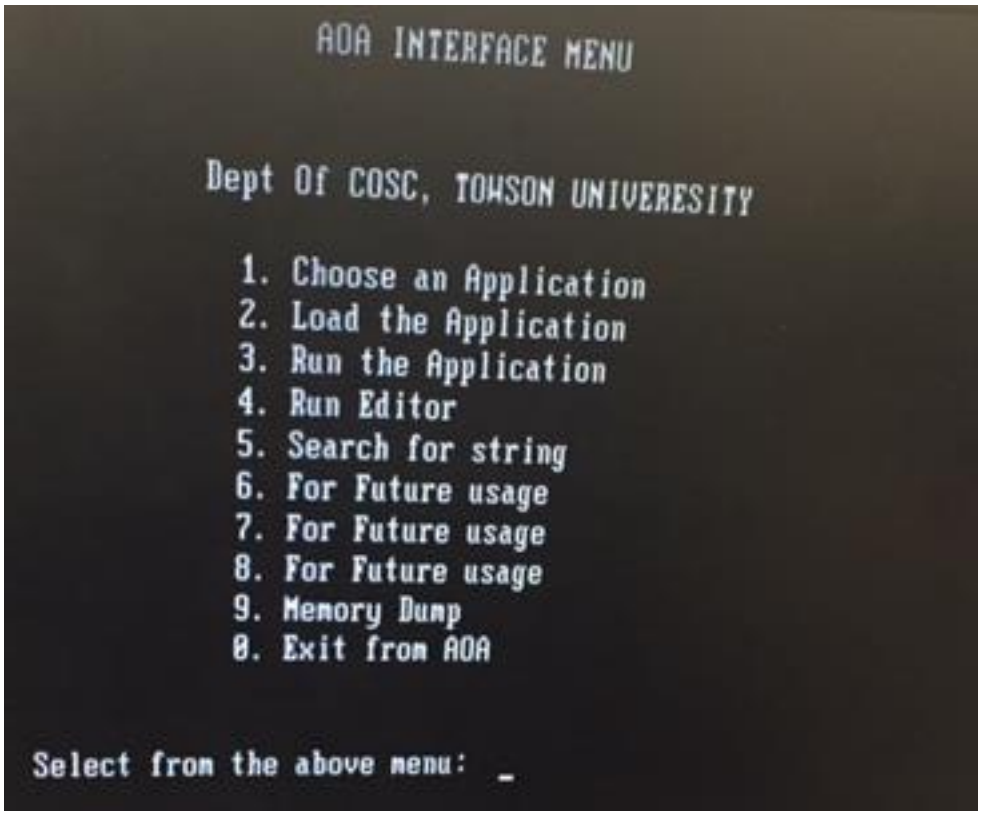


Figure 40 AOA Interface Menu

C. Mass Storage File API / SQLite Bare VFS API

The header files shown in the following pages are the lean File API and the SQLite virtual file interfaces for bare PC. These interfaces work in conjunction with other bare application APIs that have been developed over the years by numerous students and faculty in the BMC lab, including the author of this dissertation. The author of this dissertation contributed greatly to the mass storage features as well as enhancing the file system and USB driver to function and use multiple USBs for BMC.

```
//returns a file handle
int createFile(char* fname, int* startAddr, int size, char* attr);
void deleteFile(int fileHandle);
void resizeFile(int fileHandle);
void flushFile(int fileHandle);
void flushAll();
```

Figure 41 Bare Lean File API

SQLite Bare VFS Interface Header File (bareVfs.h)

The header file "***bareVfs.h***" shows the methods that are used for the SQLite bare virtual file system. Figures 42 thru 44 show these interfaces.

```
#ifndef __SQLITE_AAAB_H__
#define __SQLITE_AAAB_H__

#include "sqlite.h"

#define fopen(a, b)  openFile( a, (b[0] == 'r' ? 1:4) )
#define fflush(a)    syncFile(a)

#ifndef MAX_FILENAME_LENGTH
       #define MAX_FILENAME_LENGTH        0x40  //64 in decimal
#endif
#ifndef FILENAME_TABLE_ADDR
       #define FILENAME_TABLE_ADDR        0x18800000
#endif
#ifndef MAX_FILES
       #define MAX_FILES           1000
#endif

#ifndef SQLITEFILE_SIZE
       /* Initial file size for the DB */
       #define SQLITEFILE_SIZE     1024*10  /* file size for the DB */
#endif
#ifndef EOF
       #define EOF         (-1)
#endif
#ifndef NULL
       #define NULL         0
#endif

#ifndef BUFSIZ
       #define BUFSIZ    1024
#endif
```

Figure 42 Part 1 of bareVfs.h

```
#ifndef OPEN_MAX
#define OPEN_MAX  20    /* max #files open at once */
#endif

#define _INTERNAL_BUFSIZ    4096
typedef struct _iobuf
{
       int cnt; /* characters left */
       char* ptr; /* next character position */
       int* cacheStartAddr; /* location of buffer */
       int flag; /* mode of file access */
       int fd; /* file descriptor */
       int index; /* file index */
       int size; /* file size */
} BAREFILE;

extern BAREFILE _iob[OPEN_MAX ];

#define stdin   (&_iob[0])
#define stdout  (&_iob[1])
#define stderr  (&_iob[2])

enum _flags
{
       _READ = 01, /* file open for reading */
       _WRITE = 02, /* file open for writing */
       _UNBUF = 04, /* file is unbuffered */
       _EOF = 010, /* EOF has occurred on this file */
       _ERR = 020, /* error occurred on this file */
       _SIZECHANGE = 030 /* file size changed. Flush FAT32 & Root Directory */
};

int _fillbuf(BAREFILE*);
int _flushbuf(int, BAREFILE*);

/* macros for accessing the error and end-of-file
* status and the file descriptor.
*/
#define feof(p)     ((p)->flag & _EOF) != 0)
#define ferror(p)   ((p)->flag & _ERR) != 0)
#define fileno(p)   ((p)->index)

/*
* The getc macro normally decrements the count, advances
* the pointer,and returns the character.
* (Recall that a long #define is continued with a backslash.)
* If the count goes negative, however, getc calls the function _fillbuf to
* replenish the buffer, re-initialize the structure contents, and return a
character.
* The characters are returned unsigned, which ensures that all characters will
be positive.
*/
#define getc(p) (--(p)->cnt >= 0 \
                 ? (unsigned char) *(p)->ptr++ : _fillbuf(p))
```

Figure 43 Part 2 of bareVfs.h

```
/*
* putc  operates in much the same way as getc,
* calling a function _flushbuf when its buffer is full.
*/
#define putc(x, p) (--(p)->cnt >= 0 \
                 ? *(p)->ptr++ = (x) : _flushbuf((x) ,p))

#define fgetc(p)     getc(p)
#define getchar()    getc(stdin)
#define putchar(x)   putc((x), stdout)

BAREFILE* openFile(const char* filename, int mode);

int findFileInMemory(const char* filename, BAREFILE* file);

int readFile(int index, char* zBuf, int iAmt, int iOfst);

int writeFile(BAREFILE* fp, const void* zBuf, int iAmt, int iOfst);

int syncFile(BAREFILE* fp);

unsigned int getFileEntryPoint(int clusterNo);
int getNumberOfClusters(long fileSize);

char* fgets(char* buf, int n, BAREFILE* fp, int taskid, int* appStatus);

/* Buffer for stdin. */
char _bufin[_INTERNAL_BUFSIZ];

#endif //__SQLITE_AAAB_H__
```

Figure 44 Part 3 of bareVfs.h

```
/*
* malloc.h --- coded by William Thompson. Sept. 10, 2015
* BMC Memory Allocation for SQLite
* Towson University 2015 */
#ifndef __MY_MALLOC_H__
#define __MY_MALLOC_H__

#define MEM_BASE            0x23000000   /* Start of DB Memory Area */
#define MEM_TBASE           0x23000200   /* Start of DB Table Memory Area */
#define MEM_DBASE           0x23008000   /* Start of DB Data  Memory Area */
#define MAX_NO_TENT         2000         /* No of table entries */
#define MAX_MEM_LIMIT       0x4000000    /* Max memory size for data in bytes */
#define TE_SIZE             0x20         /* size of table entry in bytes */
#define LOCID_MAX           128
#define MAX_MEM_ALLOC_TABLE_SIZE   300       /* memory allocation table size */
#define GLOBAL_MEM_SIZE     320000      /* memory size */

int* mem_alloc_table_ptr; /* memory allocation table ptr */
static char* global_mem_addr_base; /* global memory address base  */
char* global_mem_addr_ptr; /* global memory address ptr */
static int mem_alloc_table_size; /*current mem alloc table size*/
static int global_memory_size; /*current mem size*/

static int mcb_size;

typedef struct mem_control_block mem_control_block;

struct mem_control_block
{
       int is_available;
       int size;
};

static int left_over_memory;
static char* global_mem_addr_base; /* global memory address base  */
static int* mem_alloc_table_base; /* memory allocation table base */

void AOAmalloc_init();

void AOAfree(void* ptr);
void* AOAmalloc(unsigned int size);
void* AOArealloc(void* ptr, unsigned int size);
void AOAmeminit(int tablebase, int membase);

void* AOAcalloc(int nmemb, int size);
#define  free(a)            AOAfree(a)
#define  malloc(a)          AOAmalloc(a)
#define  realloc(a, b)      AOArealloc(a, b)

void AOATrace(int marker, int value, int size, int t4);
int setSharedMemC(int addr, int locid);

#endif /* __MY_MALLOC_H__ */
```

Figure 45 BMC Memory Allocation for SQLite

REFERENCES

[1] A. Alexander, A. L. Wijesinha, and R. Karne. "A Study of Bare PC SIP Server Performance," The Fifth International Conference on Systems and Networks Communications. ICSNC 2010, August 22-27, Nice, France.

[2] A. Alexander, A. L. Wijesinha, and R. Karne, "Implementing a VoIP SIP Server and a User Agent on a Bare PC", 2nd International Conference on Future Computational Technologies and Applications (Future Computing), 2010, pp. 8-13.

[3] 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, January 2012, Maui, HI, pp. 156-160.

[4] P. Appiah-Kubi, R. K. Karne, and A. L. Wijesinha. The Design and Performance of a Bare PC Webmail Server, The 12th IEEE International Conference on High Performance Computing and Communications, AHPCC 2010, Sept 1-3, 2010, Melbourne, Australia, pp. 521-526.

[5] BBC News: PC users 'want greener machines:' http://news.bbc.co.uk/2/hi/technology/5107642.stm.

[6] 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, June 2011, p1:1-1:23

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

[8] A. Emdadi, R. K. Karne, and A. L. Wijesinha. "Implementing the TLS Protocol on a Bare PC," ICCRD2010, The 2nd International Conference on Computer Research and Development, Kuala Lumpur, Malaysia, May 2010.

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

[10] G. H. Ford, R. K. Karne, A.L. Wijesinha, and P. Appiah-Kubi. "The Design and Implementation of a Bare PC Email Server," COMPSAC"09, 33rd IEEE International Computer and Applications Conference, pp. 480-485, July 2009.

[11] G. H. Ford, Karne, R. K., Wijesinha, A. L., and Appiah-Kubi, P. "The Performance of a Bare Machine Email Server," 21st International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2009), IEEE / ACM Publications, 28-31 October 2009, Sao Paulo, SP, Brazil, pp. 143-150.

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

[13] L. He, R. K. Karne, and A. L. Wijesinha, “Design and Performance of a bare PC Web Server,” International Journal of Computer and Their Applications, Vol. 15, No. 2, June 2008, pp. 100-112, Acta Press, June 2008.

[14] L. He, R. K. Karne, A. L. Wijesinha, and A. Emdadi, “A Study of Bare PC Web Server Performance for Workloads with Dynamic and Static Content,” The 11th IEEE International Conference on High Performance Computing and Communications (HPCC-09), Seoul, Korea, June 2009, p494-499.

[15] “How to mix C and C++,” The C Programming Language. [Online]. Available at: https://isocpp.org/wiki/faq/mixing-c-and-cpp.

[16] Intel Corporation, Enhanced host controller interface specification for universal serial bus, March 2002, Rev 1, http://www.intel.com/technology/usb/download/ehci-r10.pdf [retrieved: April 8, 2016]

[17] R. K. Karne, “Application-oriented Object Architecture: A Revolutionary Approach,” 6th International Conference, HPC Asia 2002, Centre for Development of Advanced Computing, Bangalore, Karnataka, India, December 2002.

[18] R. K. Karne, K. V. Jaganathan, N. Rosa, and T. Ahmed, “DOSC: Dispersed Operating System Computing”, 20th Annual ACM Conference on Object Oriented Programming, Systems, Languages, and Applications (OOPSLA), 2005, pp. 55-61.

[19] R. K. Karne, K. V. Jaganathan, and T. Ahmed, “How to run C++ applications on a Bare PC”, 6th ACIS Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD) 2005.

[20] R. K. Karne, A. L. Wijesinha, and S. Liang, “A Bare PC Mass Storage USB Driver,” International Journal of Computers and Their Applications, March 2013.

[21] G. H. Khaksari, A. L. Wijesinha, R. K. Karne, L. He, and S. Girumala, “A Peer-to-Peer Bare PC VoIP Application,” CCNC '07, Proceedings of the IEEE Consumer and Communications and networking conference, pp. 803-807, IEEE Press, Las Vegas, Nevada, January 2007.

[22] G. H. Khaksari, R. K. Karne and A. L. Wijesinha. “A Bare Machine Application Development Methodology”, International Journal of Computers and Their Applications (IJCA), Vol. 19, No.1, March 2012, p10-25.

[23] J. Lange, et. al, “Palacios and Kitten: New High Performance Operating Systems for Scalable Virtualized and Native Supercomputing,” 24th IEEE International Parallel and Distributed Processing Symposium(IPDPS), Apr. 2010, pp. 1-12.

[24] J. Larimer, “Beyond Autorun: exploiting vulnerabilities with removable storage,” 1–66, Jan. 2011. https://media.blackhat.com/bh-dc-11/Larimer/BlackHat_DC_2011_Larimer_Vulnerabiliters_w-removeable_storage-wp.pdf.

[25] S. Liang, R. K. Karne, and A. L. Wijesinha., A Lean USB File System for Bare Machine Applications, The Proceedings of the 21st International Conference on Software Engineering and Data Engineering, ISCA, June 2012, pp.191-196.

[26] Microsoft Corp, “FAT32 File System Specification”, http://microsoft.com/whdc/system/platform/firmware/fatgn.rnspx, 2000. [retrieved: April 8, 2016]

[27] U. Okafor, R. Karne, A. Wijesinha, and P. Appiah-Kubi. Eliminating the Operating System via the Bare Machine Computing Paradigm, 5th International Conference on Future Computational Technologies and Applications, Future Computing, May 27- June 1, Valencia, Spain, 2013.

[28] U. Okafor, R. Karne, A. Wijesinha, and P. Appiah-Kubi. A Methodology to Transform an OS-Based Application to a Bare Machine Application, The 12th IEEE International Conference on Ubiquitous Computing and Communications (IUCC-2013), July 16 - 18, Melbourne, Australia, 2013.

[29] U. Okafor, R. K. Karne, A. L. Wijesinha and B. Rawal. Transforming SQLITE to Run on a Bare PC, In Proceedings of the 7th International Conference on Software Paradigm Trends, pages 311-314, Rome, Italy, July 2012.

[30] V. S. Pai, P. Druschel, and W. Zwaenepoel. “IO-Lite: A unified I/O buffering and caching system”, ACM Transactions on Computer Systems, Vol.18 (1), Feb. 2000, pp. 37-66.

[31] Perisoft Corp, Universal serial bus specification 2.0, http://www.perisoft.net/engineer/usb_20.pdf. [retrieved: April 8, 2016]

[32] B. Rawal, R. Karne, and A. L. Wijesinha. “Splitting HTTP Requests on Two Servers,” The Third International Conference on Communication Systems and Networks: COMPSNETS 2011, January 2011, Bangalore, India.

[33] B. Rawal, R. K. Karne, and A. L. Wijesinha. “Mini Web server clusters for HTTP request splitting”, IEEE Conference on High Performance, Computing and Communications (HPCC), Banff, Canada, Sept 2-4, 2011, pp. 94-100.

[34] F. F. Rezende and K. Hergula. The Heterogeneity Problem and Middleware Technology: Experiences with and Performance of Database Gateway, International Conference on Very Large Databases (VLDB ‘98), 1998, pp. 146-157.

[35] R. Russon and Y. Fledel, “NTFS Documentation,” [online]. Available at http://dubeyko.com/development/FileSystems/NTFS/ntfsdoc.pdf.

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

[37] R. Shullich, “Reverse Engineering the Microsoft Extended FAT File System (exFAT),” [online]. Available at https://www.sans.org/reading-room/whitepapers/forensics/reverse-engineering-microsoft-exfat-file-system-33274.

[38] S. Soumya, R. Guerin and K. Hosanagar, “Functionality-rich vs. Minimalist Platforms: A Two-sided Market Analysis”, ACM Computer Communication Review, vol. 41, no. 5, pp. 36-43, Sept. 2011.

[39] SQLite, http://www.sqlite.org/download.html [retrieved: April 5, 2015]

[40] The SQLite OS Interface or “VFS”, http://www.sqlite.org/vfs.html.

[41] W. V. Thompson, K. Karne, S. Liang, A. L. Wijesinha, H. Alabsi and H. Chang, “Implementing a USB File System for Bare PC Applications”, The 12th Advanced International Conference on Telecommunication, 2016, Valencia, Spain.

[42] Total Phase Inc., USB analyzers, Beagle, http://www.totalphase.com. [retrieved: April 8, 2016]

[43] Universal serial bus mass storage class, bulk only transport, revision 1.0, 1999, http://www.usb.org [retrieved: April 8, 2016]

[44] T. Venton, M. Miller, R. Kalla, and A. Blanchard, “A Linux-based tool for hardware bring up, Linux development, and manufacturing,” IBM Systems Journal, Vol. 44 (2), pp. 319-330, IBM, NY, 2005.

[45] R. Yasinovskyy, A. L. Wijesinha, R. K Karne and G. Khaksari. “Comparison of VoIP Performance on IPv6 and IPv4 Networks”, The 7th ACS/IEEE International Conference on Computer Systems and Applications (AICCSA), 2009.

CURRICULUM VITAE

NAME: William V. Thompson

[redacted]

PROGRAM OF STUDY: Information Technology

DEGREE AND DATE TO BE CONFERRED: Doctor of Science, May 2016

EDUCATION:

Doctor of Science, Information Technology
Towson University, Towson, Maryland – May 2016

Master of Science, Applied Information Technology
(concentration in Database Management Systems)
Towson University, Towson, Maryland – May 2007

Bachelor of Science, Business Administration
(specialized in Management Information Systems)
University of Baltimore, Baltimore, Maryland – May 2005

CERTIFICATES:

Post Baccalaureate Certificate, Database Management Systems

Certified Network Professional, Network+ (CompTIA)

Microsoft Certified Professional (MCP)

Certificates, Microsoft Office 2003/2007/2010 Suite Implementation

Certificate – Advanced Computer Repair and Maintenance

SKILLS AND ABILITIES:

Knowledgeable and fluent in several programming languages, including C#, C++, C, Java, R, SAS, ColdFusion, VB.Net, PHP, Perl, JavaScript, AngularJS, SQL, XML, XSD, and HTML.

AWARDS AND HONORS:

Member, Phi Theta Kappa (A scholastic honor society)

Graduated Cum Laude, University of Baltimore; 2004

Graduated Magna cum Laude, Community College of Baltimore County; 2003

ARTICLES / PULICATIONS:

Implementing a USB File System for Bare PC Applications; 2016

Interoperable SQLite based on the Bare Machine Computing Paradigm; 2016

Mass Storage System for Bare PC Applications Using USBs; 2016

PROFESSIONAL POSITIONS HELD & EXPERIENCE:

Instructor
Towson University; Towson, Maryland; 08/2013 – present

Director, Data Management and former Acting IT Director
Maryland Institute for Emergency Medical Services Systems (MIEMSS)
Baltimore, Maryland; 03/2009 – present

IT Systems Analyst
U.S. Social Security Administration
Woodlawn, Maryland; 9/2014 – 12/2014

Adjunct Lecturer/Instructor
The Community College of Baltimore County (CCBC)
Baltimore, Maryland; 08/2008 – 02/2014

Database Operations Specialist / Sr. Programmer
Maryland Institute for Emergency Medical Services Systems (MIEMSS)
Baltimore, Maryland; 03/2007 – 3/2009

Webmaster II
State of Maryland Department of Budget and Management (MDBM)
Annapolis, Maryland; 06/2006 – 03/2007

Graduate Assistant/Application Developer
State of Maryland Department of Human Resources (Through Towson University)
Essex, Maryland; 10/2005 – 06/2006

Project Specialist
International Services Office, University of Baltimore
Baltimore, Maryland; 07/2005 – 06/2006.