Technical notes on using Analog Devices DSPs, processors and development tools
Visit our Web resources http://www.analog.com/ee-notes and http://www.analog.com/processors or e-mail processor.support@analog.com or processor.tools.support@analog.com for technical support.

Implementing FAT32 File Systems on ADSP-BF533 Blackfin® Processors

Contributed by Jenny Jing

Rev 1 – February 24, 2006

Introduction

Blackfin® processors are commonly used in multimedia data processing applications such as portable media players (PMPs) and digital video recorders (DVRs). These types of products require storage for large amounts of compressed multimedia data. Though there are many storage options available, a hard drive is a good choice when making a tradeoff between price and performance. Using a hard drive also requires a file system to access data in file format. This EE-Note describes a demonstration system that implements the FAT32 file system on a hard drive, including hard drive detect and file operations including open, close, read, write, create, remove, list, directory, and search. The option for long file names is also supported.

Both the hardware and software implementations are discussed in this application note. Since the FAT32 file system code is written in C, it is totally compatible with other Blackfin processors such as the ADSP-BF534/BF536/BF537 and ADSP-BF561. Note that all references to ADSP-BF533 Blackfin processors throughout this document also apply to low-power ADSP-BF532/BF531 derivatives as well. The FAT32 file system design is modular, so it can be used in other Blackfin-based storage solutions (e.g., compact flash storage cards) with only modification to the physical driver layers.

The demo described in this note is built on the ADSP-BF533 EZ-KIT Lite® evaluation board.

Overview

Hard drives have the following parameters (Figure 1):

- Platters
- Heads
- Tracks
- Cylinders
- Sectors

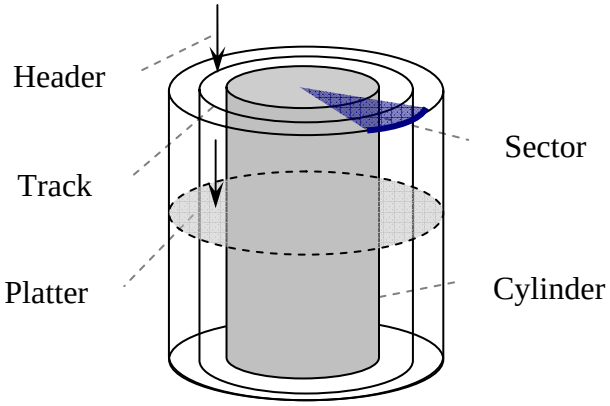


Figure 1. Hard Disk Physical Architecture

Platters have many concentric circles, which are called *tracks*. Tracks consist of *sectors*. Tracks with the same perimeters on platters form *cylinders*.

At least two headers point to one platter (one on each side, used to follow tracks). With three parameters (cylinder, header, and sector), any data on the hard drive can be addressed.

There are two ways of addressing a hard drive: physical addressing (C/H/S) and logical addressing (LBA). In this demo, logical addressing is used, employing the following conversion expressions:

$$LBA = NH \times NS \times C + NS \times H + S - 1$$

Where: $$C = \frac{LBA / NS}{NH}$$

$$H = \left(LBA / NS\right) \% NH$$

$$S = LBA \% NS + 1$$

NH: Number of headers per cylinder

NS: Number of sectors per tracks

C: Number of cylinders

Hard Disk Partition

The first sector of a hard drive (cylinder 0, header 00, sector 1) is the master boot record (MBR) sector. It includes 446 bytes of boot code, the disk partition table (DPT), and a 2-byte end flag (0x55AA). The MBR is independent of the operating system and is followed by 62 system reserved sectors, as shown in Figure 2.

DPT registers partition information, and each partition may be formatted separately with a specific file system. The FAT region (FAT1 and FAT2), DOS boot record (DBR), and data region are the basic regions included on a FAT volume. These are discussed in detail later in this document.

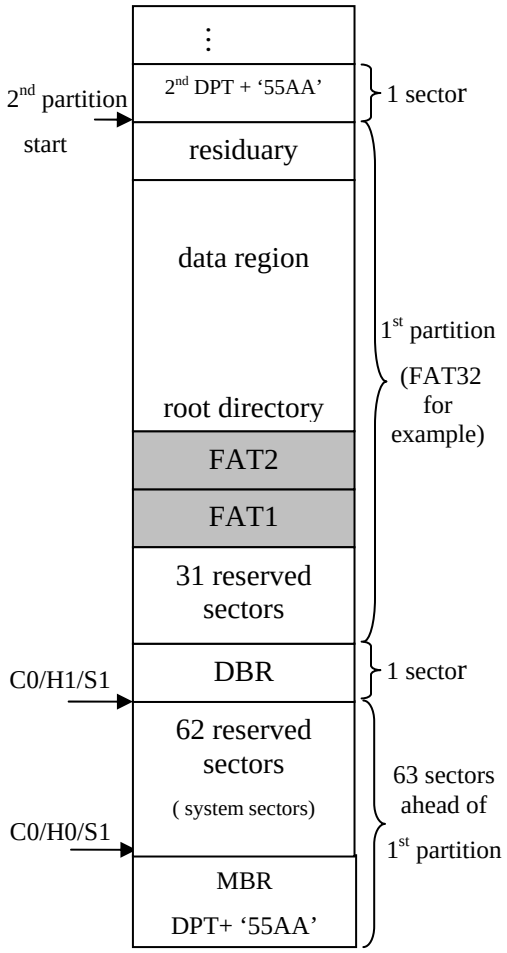


Figure 2. Hard Disk Partition Diagram

Hardware Implementation

Once the hard drive has been formatted by a host PC, it appears to the host as a standard ATA (IDE) disk drive. The hard drive can be interfaced easily to the ADSP-BF533 Blackfin processor's asynchronous memory through the external bus interface unit (EBIU). The EBIU provides glueless interfaces to external memories, as shown in Figure 3.

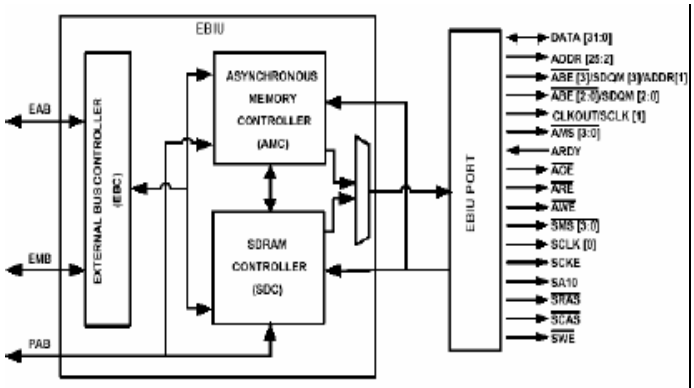


Figure 3. External Bus Interface Unit (EBIU)

The EBIU is clocked by the system clock (`SCLK`) of the ADSP-BF533 Blackfin processor via asynchronous memory bank control registers (`EBIU_AMBCTL0`, `EBIU_AMBCTL1`). These registers can be used to control wait states, `ARDY` enable/disable, and setup/hold times for each asynchronous memory bank. On the ADSP-BF533 EZ-KIT Lite, the `SCLK` runs at 118 MHz by default, which is more than sufficient for accesses to hard drives since read/write access times on hard disks are less than 120 ns (~8 MHz). To find interface/bus timing, refer to the `ATA/ATAPI-6 standard`[1].

Generally, hard disks work in two modes: PIO mode and multiword DMA mode. This EE-Note deals only with 16-bit transfers using PIO mode.

ADSP-BF533-to-Hard Disk Pin Mapping

Figure 4 shows the main pin-to-pin connections between the ADSP-BF533 Blackfin processor and the hard disk. See Appendix A for reference schematic design details.

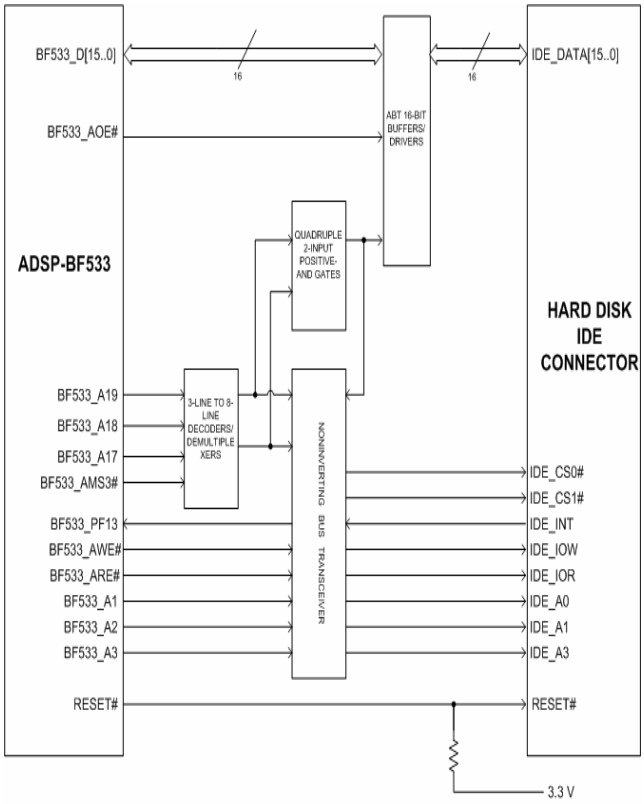


Figure 4. ADSP-BF533 - Hard Disk Interface

Hard Disk Signal Descriptions – PIO Mode

- `CS[1:0]`: Chip select signals from the ADSP-BF533 processor used to select the command block registers.
- `DA[2:0]`: 3-bit binary-coded address asserted by the ADSP-BF533 processor to access a register or data port in the hard disk.
- `DD[15:0]`: Data lines - the lower 8 bits are used for 8-bit register transfers. Data transfers are 16 bits wide.
- `/DIOR` - Device I/O read - generated by the ADSP-BF533 processor when reading data from the hard disk.
- `/DIOW` - Device I/O write - driven by the ADSP-BF533 processor when writing data to the hard disk.

- /RESET - Hardware reset - generated by the ADSP-BF533 processor when resetting the hard disk.
- INTRQ - Device interrupt - used by the hard disk to interrupt the ADSP-BF533 processor. In PIO mode, INTRQ is asserted at the beginning of each data block to be transferred. A data block is usually a single sector.

The ADSP-BF533 Blackfin processor hosts the hard drive via programmed I/O. Host address lines DA[2:0], chip selects /CS0 and /CS1, /DIOR, and /DIOW address the disk registers. Host address lines DA[9:3] generate the two chip selects /CS0 and /CS1.

Chip select /CS0 accesses the eight hard disk command block registers. Chip select /CS1 is valid during 8-bit transfers to/from the control block registers — alternate status and device control — and the drive address (see Table 1).

The hard drive selects the primary or alternate command block addresses using address bit DA7.

The command block registers are used for sending commands to the device or posting status from the device. These registers include the Cylinder High, Cylinder Low, Drive/Head, Sector Count, Sector Number, Command, Status, Features, Error, and Data Port registers. The control block registers are used for device control and to post alternate status. These registers include the device control and alternate status registers.

Hard Disk I/O Registers								
Addr	-CS1FX	-CS3FX	SA2	SA1	SA0	Read	Write	
3F6	1	0	1	1	0	Alter Status	Device Control	Control Block Registers
3F7	1	0	1	1	1	Drive Address	Not Used	Command Block Registers
1F0	0	1	0	0	0	Data Port	Data Port	
1F1	0	1	0	0	1	Error Register and Precomp		
1F2	0	1	0	1	0	Sector Count		
1F3	0	1	0	1	1	Sector Number		
1F4	0	1	1	0	0	Cylinder Low		
1F5	0	1	1	0	1	Cylinder High		
1F6	0	1	1	1	0	Drive / Head		
1F7	0	1	1	1	1	Status	Command	

Table 1. Memory-Mapped Decoding

Software Implementation

The software implementation for the hard disk driver and FAT32 file system are discussed in detail this section.

FAT32

The FAT file system was originally developed as a simple file system suitable for floppy disk drives less than 500 KBytes in size. Over time, it has been enhanced to support larger media types. Currently, there are three FAT file system types: FAT12, FAT16, and FAT32. The basic difference in these FAT file system types is the size (in bits) of the entries in the actual FAT structure on the disk. There are 12 bits in a FAT12 entry, 16 bits in a FAT16 entry, and 32 bits in a FAT32 entry. The FAT32 file system was chosen for this demo, as it is widely used in most operating systems and supports partitions of up to 32 GBytes.

A FAT32 file system volume is composed of three basic regions, as shown in Table 2:

Disk Structure	Sector Offset
Boot Sector	0
Reserved Sector	1
FAT 1	32
FAT 2	(# of Sectors per FAT) + 32
File Clusters (Data Region)	2*(# of Sectors per FAT) + 32

Table 2. FAT32 Structure

FAT32 uses 32 binary bits to record a cluster chain, which is why it is called FAT32, allowing a maximum of 2T clusters. As a result, FAT32 is theoretically able to address a 1-TeraByte partition, even if a cluster includes only one sector. In order to reduce FAT size and improve system performance, FAT32 does not support partitions in excess of 32 GBytes. Table 3 describes FAT32 partitions and cluster sizes.

FAT32 Cluster Size		
Partition Size	Sectors per Partition	Cluster Size
< 8GB	8	4KB
≥ 8GB and <16GB	16	8KB
≥ 16GB and <32GB	32	16KB
≥ 32GB	64	32KB

Table 3. FAT32 Cluster Size

FAT32 treats the root directory as a normal file, so there is no fixed offset for the root directory. Since the root directory is the first directory file after the disk is formatted, it usually occupies the first cluster of the data region.

There are two FAT data structures on the volume. This provides redundancy for the FAT data structure so that if a sector goes bad in one of the FATs, the data is not lost because it is duplicated in the other FAT.

Each FAT is a single linked list of the clusters that make up a file. These clusters map to the data region of the volume. The first two entries in the FAT are reserved so that logical cluster numbers begin at 2 (Figure 5).

Offset	Entry (next cluster)
0	0xFFFF FFF8
1	0xFFFF FFFF
2	0x0000 0005
3	0x0000 0004
4	0x0000 0007
5	0x0000 0006
6	0x0000 0008
7	0x0000 0009
9	… …

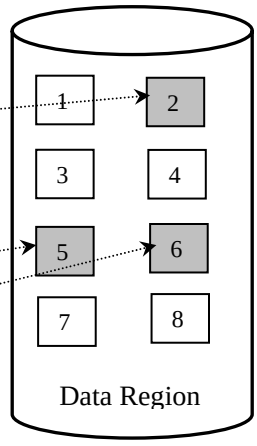


Figure 5. FAT32 File System Working Diagram

Software Architecture

The software includes five layers (Table 4).

Lib Functions
FAT Interface Functions
FAT Driver
IDE Interface Functions
IDE Driver

Table 4. Software Architecture

Drivers

Functions in the driver layer are ATAPI standard compatible. These functions are basic hard disk operation functions, which include checking I/O registers and getting the hard disk's status. The driver's source code can be found in Listing 1.

ide_base.c

```
/*** Copyright (c) 2003 Analog Devices Inc.  All rights reserved. ***/
/* File          : ide_base.C
 * Processor     : ADSP-BF533
 * IDDE          : VisualDSP++3.5
 * Description   : define IDE driver functions including:
 *            Wait_Ready();
 *            Wait_ReadyBusy();
 *            CheckforError();
 *            IdeStandby();
 *            IdeIdle();
************************************
* Function      : Wait_Ready
* Description : Wait HD getting ready
* Input      : None
* Output    : 0-ready /error ID-HD error
************************************/
BYTE  Wait_Ready(void){
   BYTE statbyte;
   BYTE flag = 0;
   while (!flag){
   // Read Status Register
   statbyte = *pStatus;
   if (statbyte & IDE_ERROR){
   //Check HD error
         statbyte=*pErrorReg;
   //read error register if error and
   //return error ID
         return  statbyte;
      }
      if (statbyte & IDE_DRDY)
  //check ready or not, set flag
      flag = 1;  }
   return 0;
}
/************************************
* Function      : Wait_ReadyBusy
* Description : query HD busy or not.
* Input          : none
* Output     : 0-busy /error ID-HD error
```

```
************************************/
BYTE Wait_ReadyBusy(void){
   BYTE statbyte;
   BYTE flag = 0;
   while (!flag){
   // Read Status Register
      statbyte = *pStatus;
      if (statbyte & IDE_ERROR){
         statbyte=*pErrorReg;
         return  statbyte;
      }
      if (((statbyte & IDE_DRDY)!=0 )&&((statbyte & IDE_BUSY)==0))
     // Ready bit is in pos 6
         flag = 1;        }
   return 0;
}
/************************************
* Function     : CheckforError
* Description : check HD error
* Input         : none
* Output    : 0-no errors /error ID -yes
************************************/
BYTE CheckforError(void){
      BYTE statbyte;
     // Read Status Register
      statbyte = *pStatus;
     if (statbyte & 0x01){
         statbyte= *pErrorReg;
         return statbyte;
      }
      else
         return 0;
}
/************************************
* Function     : IdeStandby
* Description : Set HD standby mode
* Input         : none
* Output       : none
************************************/
void IdeStandby(void){
    *pCommand = 0xe0;
}
/*****************************************************************************
* Funciton     : IdeIdle
* Description : Set HD into idle mode
* Input         : none
* Output       : none
*****************************************************************************/
void IdeIdle(void){
    *pCommand = 0x95;
}
```

Listing 1. ide_base.c

The hard disk I/O registers listed in Table 1 are re-defined in a header file '`ide_base.h`' as shown in Listing 2:

ide_base.h

```
//IDE Register Address
#define IDECS0BASEADDR 0x20340000

#define pDataPort (volatile unsigned
short *)(IDECS0BASEADDR +(0x00<<1))

#define pPreComp (volatile unsigned
char *)(IDECS0BASEADDR + (0x01<<1))

#define pSectorCount (volatile
unsigned char *)(IDECS0BASEADDR +
(0x02<<1))

#define pSectorNumber (volatile
unsigned char *)(IDECS0BASEADDR +
(0x03<<1))

#define pCylinderLow (volatile
unsigned char *)(IDECS0BASEADDR +
(0x04<<1))

#define pCylinderHigh (volatile
unsigned char *)(IDECS0BASEADDR +
(0x05<<1))

#define pDriveHead (volatile unsigned
char *)(IDECS0BASEADDR + (0x06<<1))

#define pCommand (volatile unsigned
char *)(IDECS0BASEADDR + (0x07<<1))

#define pStatus  (volatile unsigned
char *)(IDECS0BASEADDR + (0x07<<1))

#define pErrorReg (volatile unsigned
char *)(IDECS0BASEADDR + (0x01<<1))
```

Listing 2. ide_base.h

The `ide_base.h` header also defines some IDE commands and structures such as data buffers.

IDE Interface Functions

IDE interface functions read/write one sector from/to the hard disk. They also get the initial information off the hard disk via Command (Figure 6).

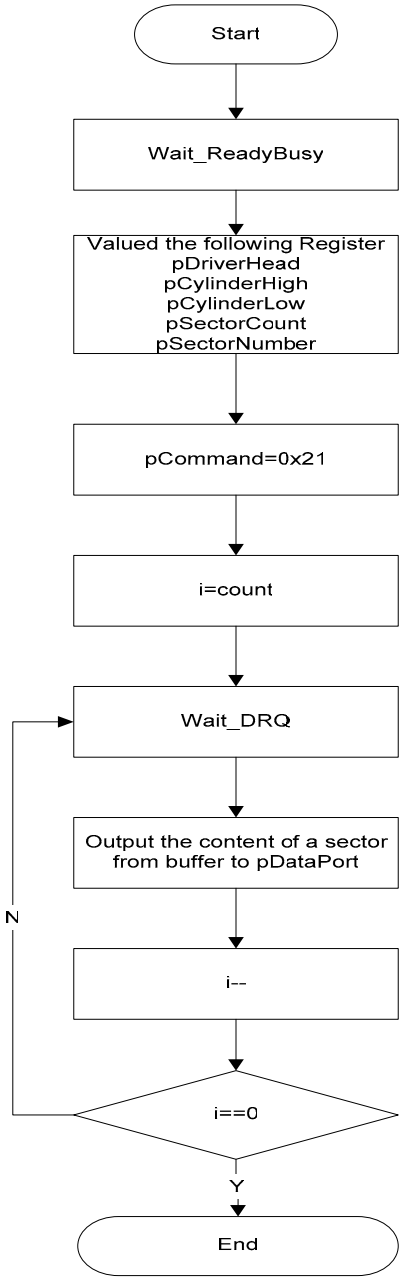


Figure 6. IDE Interface Function Flow Chart

The source code for these operations can be seen in Listing 3:

ide_access.c

```
/**********************************
***Copyright (c) 2003 Analog Devices Inc.  All rights reserved.
**********************************/
/**********************************
* Function     : fnIDE_BufferSector
* Description : read a logic sector
* Input :
*      BYTE *buffer          -reading data buffer
*      DWORD LBALocation     -logic sector address
*      BYTE count            -counts of reading sectors(up to 256), count=0
*                             means read 256 sectors
* Output: 0-normal; 1-error
**********************************/
int fnIDE_BufferSector(WORD *buf,DWORD LBALocation,BYTE count)
{
    WORD    i,temp;
    BYTE     j;
    BYTE errorcode;
    Wait_ReadyBusy();
    // update current sector loaded
    buffers.SectorCurrentlyLoaded=LBALocation;
    *pDriveHead  =((LBALocation >> 24) & 0xFF) | 0xE0;
    *pCylinderHigh     =(LBALocation >> 16) & 0xFF;
    *pCylinderLow=(LBALocation >> 8) & 0xFF;
    *pSectorNumber     =LBALocation & 0xFF;
    *pSectorCount= count;                // Read sector
    *pCommand= 0x21;

    Wait_DRQ();
    j=count;
    temp=0;
    do
    {
     Wait_DRQ();
     for(i=0; i < SECTORWORDSIZE; i++)
         {
           buf[temp+i]=*pDataPort;
         }
      j--;
      temp += SECTORWORDSIZE ;
    }while(j!=0);

    errorcode= CheckforError();
    if (errorcode!=0) return 1;

    return 0;
}
```

Listing 3. ide_access.c

FAT Drivers

The FAT driver is compatible with the FAT32 standard. It is a sector-based cluster read/write (R/W), as shown in Figure 6. The FAT driver layer determines the strategy of files R/W, which is key to file R/W efficiency.

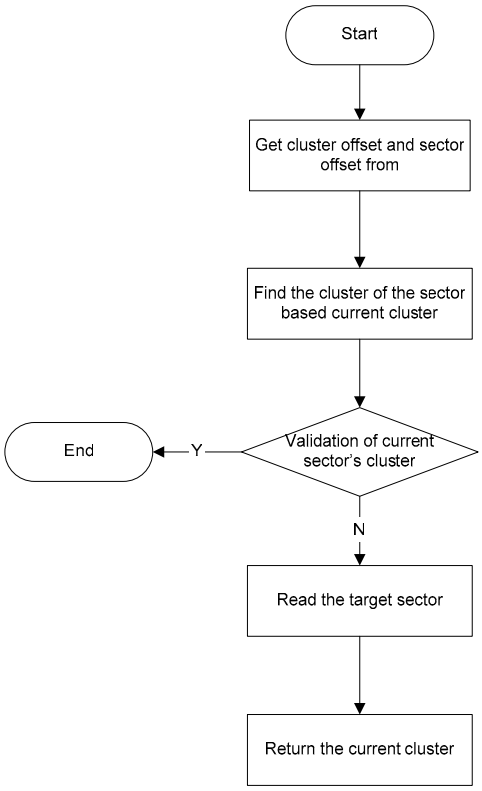


Figure 6. Driver Flow Chart

Figure 7 shows the file data — including address information in the FAT and true data — read or write process.

The data transfer rate for the hard drive can be differentiated into external transfer rate and internal transfer rate. The external transfer rate measures data transfer speed between the buffer inside the hard disk and the external device. The internal transfer rate is the transfer speed between the data buffer and platters. Because of the limitation of the hard disk's mechanism, the internal transfer rate is much lower than the external transfer rate.

As a result, frequent buffer refreshing leads to more data transfers inside the hard disk, which keeps the external bus idle and waiting for data to be ready.

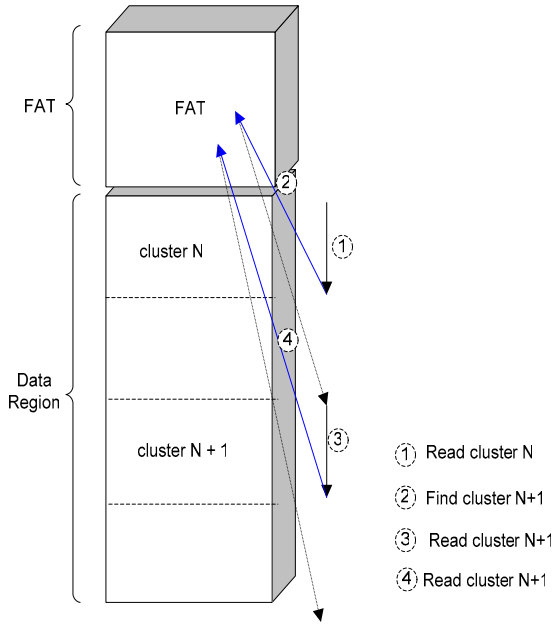


Figure 7. Hard Disk Reading File Example

One optimization is needed in this layer implementation, as shown in Figure 8:

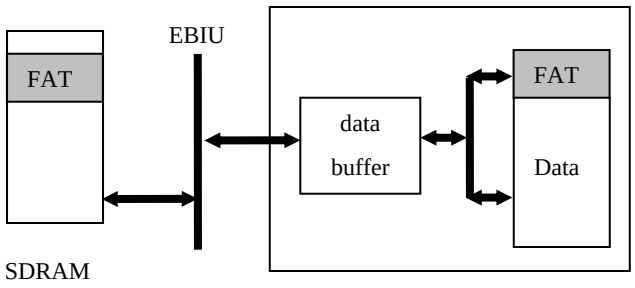


Figure 8. FAT Driver Optimization Diagram

Continuous reading is the key optimization. Place an identical data buffer outside the hard disk in SDRAM and copy the FAT32 in the hard disk to this buffer. As a result, when data is read from the hard disk, the host will read FAT32 in SDRAM to find the address of the data on the hard disk — which is much faster than reading the hard disk itself — then read the hard disk for the actual file data. With this optimization, the reading rate is upgraded to over 1000x performance.

When writing data to the hard disk, this optimized architecture does not work since FAT needs to be refreshed after each new data is written to the data region on the hard disk.

The source code associated with these operations is shown Listing 4.

FAT32_base.c

```
/*************************************
*Function     : fnFAT32_SectorReader
*Description  : read target sector (according to target sector offset in cluster)
*                case 1 - normal, target sector is in current cluster
*                case 2 - target sector is not in current cluster, find next
*                         cluster (according to offset)
*Input:
* currentcluster_t cluster - current cluster number (does not contain the
*                            sector in case 2)
* DWORD offset - offset of target sector in the cluster
*Output: cluster-current cluster number
**************************/
currentcluster_t fnFAT32_SectorReader(currentcluster_t cluster, DWORD offset)
{
 DWORD SectortoRead = 0;
 DWORD ClustertoRead = 0;
 DWORD ClusterChain = 0;
 WORD sector_per_cluster=current_fs->bpb.sector_per_cluster;
 int i;

 if(cluster.value==0xFFFFFFFF)
 return cluster;

 ClusterChain = cluster.value;
 ClustertoRead = offset / sector_per_cluster;
  SectortoRead = offset - (ClustertoRead*sector_per_cluster);

// call finFAT32_FindNextCluster() to find cluster that contains target sector

for (i=cluster.offset; i<ClustertoRead; i++)
 {
 ClusterChain = fnFAT32_FindNextCluster(ClusterChain);
 }

//register current cluster
 cluster.value=ClusterChain;
 cluster.offset=ClustertoRead;

 if (ClusterChain==0xFFFFFFFF)
 return cluster;

 //read target sector
fnIDE_BufferSector(&buffers,fnFAT32_LBAofCluster(ClusterChain)+SectortoRead,1);
 return cluster;
}
```

Listing 4. FAT32_base.c

FAT32 Interface Functions

FAT32 interface functions define some operating functions of FAT32, such as listing directories and searching, opening, and closing files. Figure 9 shows an example flow chart for the file list function.

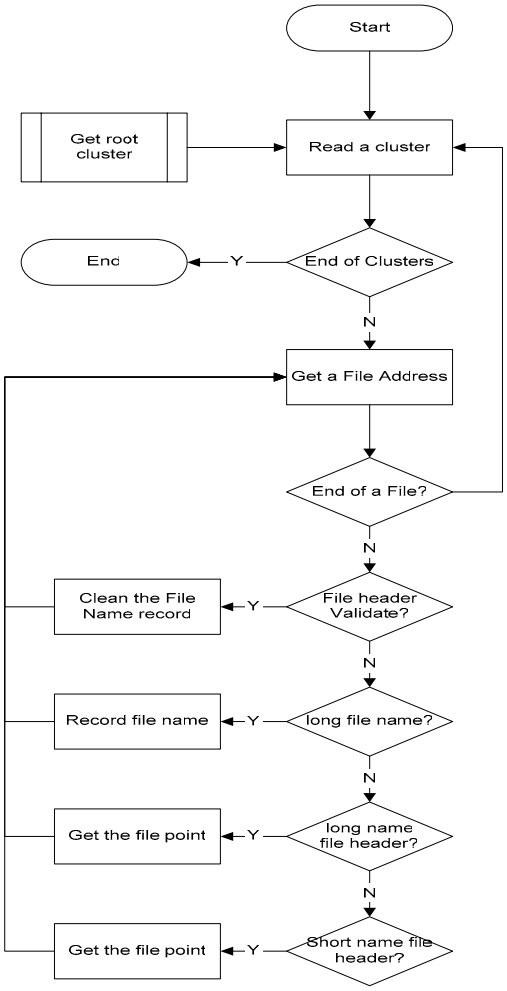


Figure 9. File List Function Flow Chart

The file search function, which is similar to the file list function, requires an extra file name that compares operation after getting the file pointer.

The file read and write functions are the two main functions of the FAT32 interface. The file read function flow chart is shown in Figure 10.

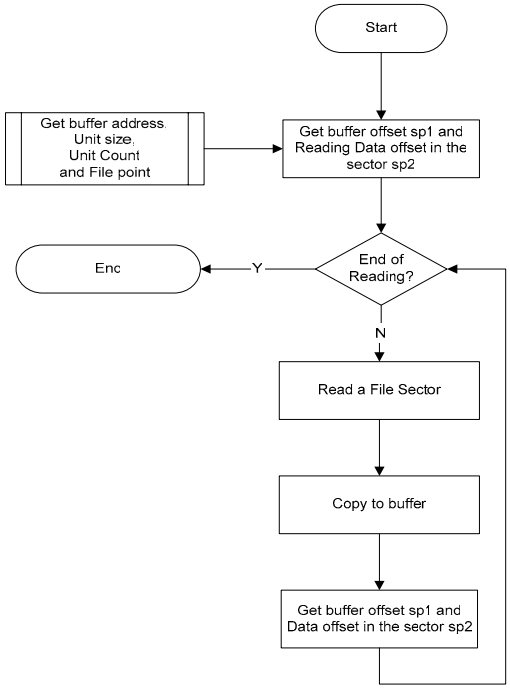


Figure 10. File Read Function Flow Chart

The file write function is similar to the file read function, but writing files requires that target sectors that are about to be written be read from hard disk first in order to protect nearby sectors. Figure 11 is a flow chart of the file write function:

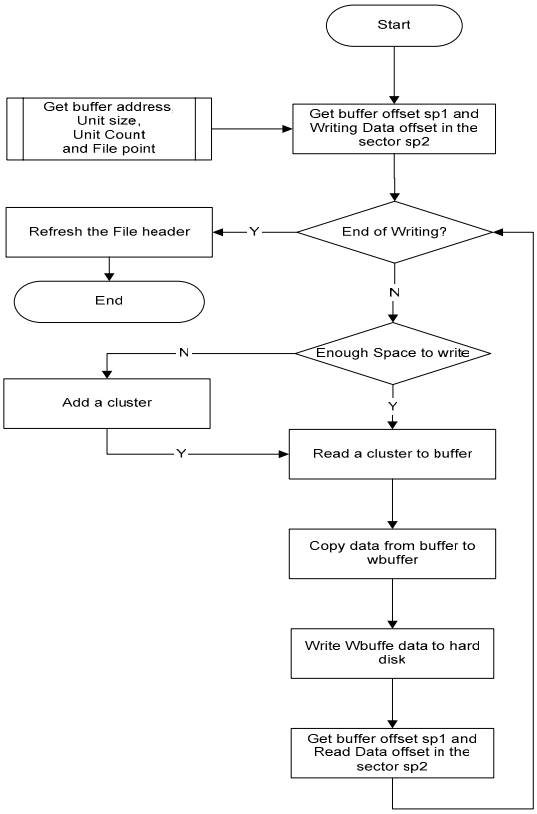


Figure 11. File Write Function Flow Chart

Library Functions

Functions in this layer mainly supply standard ANSI C compatible interfaces to the upper user application, resolve requests, and call lower layer foundational functions.

The whole project can be built into a library. You can then achieve dedicated operation by including the library and the `stdio.h` header file in the application project and calling dedicated interface functions declared in `stdio.h`.

```
//**********************************
//Function: list a file
//Input: char *path - target path
//output: FILE_t** - file list pointer
//**********************************
FILE_t** __flist(BYTE * path)

//**********************************
//Function: open a file
//Input: char *path - target path
//     BYTE *mode - allowable open mode
//                r - read
//                w - write
//                a - append
//                + - read & write
//output: FILE_t** - file list pointer
//**********************************
FILE_t* __fopen(BYTE *path, BYTE *mode)

//**********************************
//Function: read a file
//Input: char *path - target path
//       BYTE *size - data unit size
//       DWORD count - data unit number
//     FILE_t *fp - target file pointer
//output:
//unsigned long - reading data size
//**********************************
FILE_t* __fread(BYTE *path, BYTE *mode)
```

Listing 5. filelib.c

Results and Benchmarks

- Singe sector reading costs around 50K cycles.
- File reading is faster than file writing.
- When file size is over 1 cluster, the test result depends on how the file data is distributed.
- This FAT32 file system adds less than 5% overhead to the system.
- Writing larger files results in lower FAT32 overhead cost.
- When the hard disk is in DMA mode, the total cycles cost decreases rapidly.

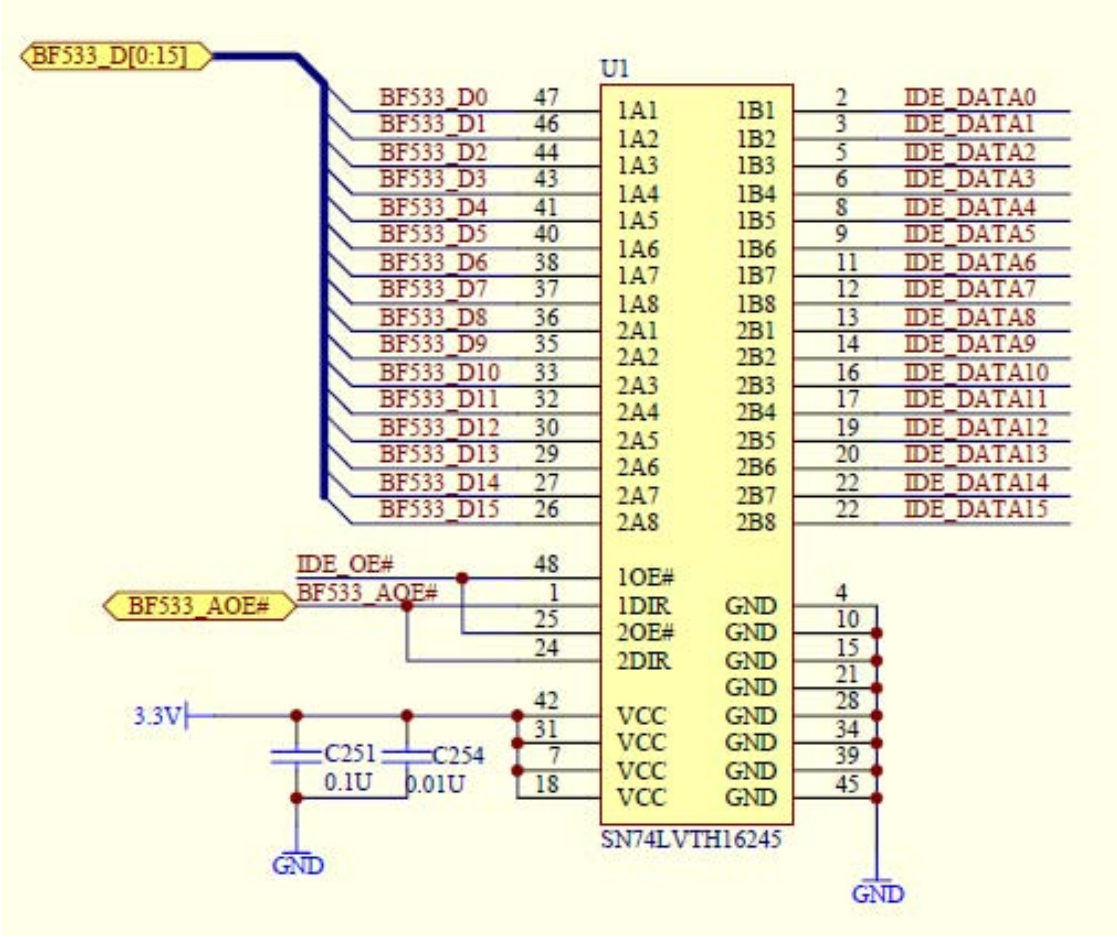
Test	Cycles	Description
flist	970195	C:/， 12 files
fsearch	6542659	2 deep directory
frename	17959442	
fopen	961171	Open 717 MB file
fseek	306	Compare 13 times
fclose	149	
fcloselist	2753	

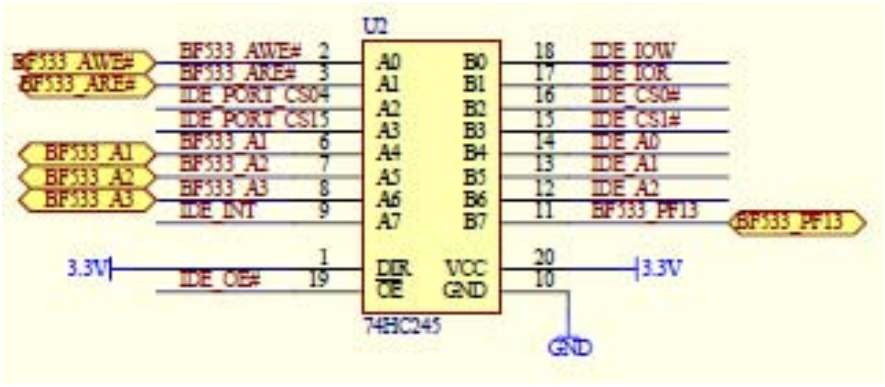
Table 5. Test Results

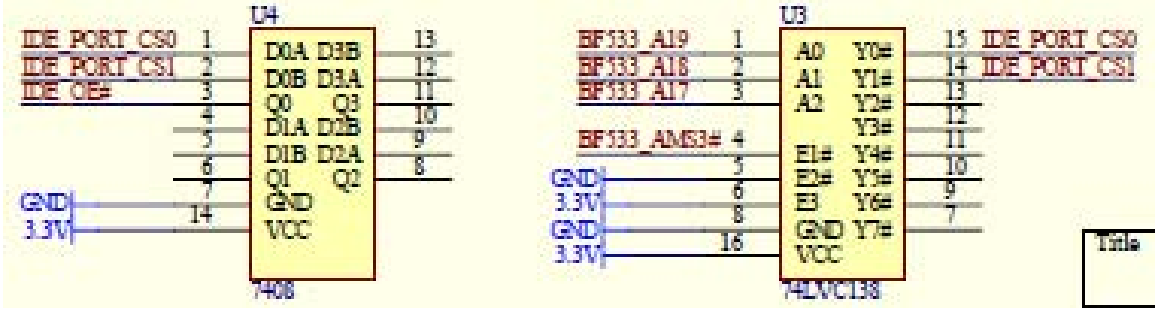
Read File Test			
Size (bytes)	Total Cycles	Cycles of FAT	Overhead
	FAT driver & interface function IDE driver & interface functions	FAT driver & interface functions	(Cycles of FAT)/(Total Cycles)
39k	~3.0×10^6	~4.5×10^4	1.4%
547k	~12.0×10^6	~5.5×10^5	4.9%
1M	~55.0×10^6	~1.0×10^6	2.1%
33M	~18.0×10^9	~50.0×10^6	0.3%
Write File Test			
1K	~23.0×10^6	~20.0×10^6	88%
16K	~48.0×10^6	~22.0×10^6	44.1%

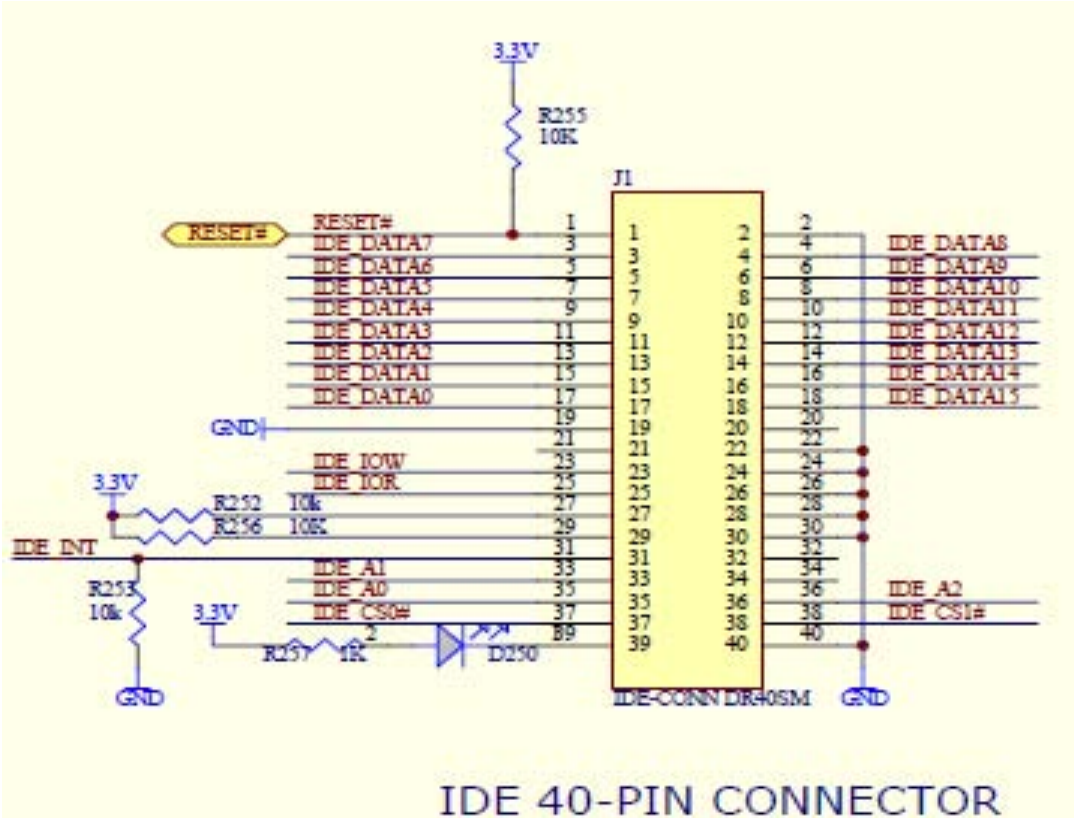
Table 6. Benchmarks

Appendix A




U2
BF533_AWE#
BF533_ARE#
BF533_A1
BF533_A2
BF533_A3
BF533_AWE#
BF533_ARE#
IDE_PORT_CS04
IDE_PORT_CS15
BF533_A1
BF533_A2
BF533_A3
IDE_INT
3.3V
IDE_OE#
A0
A1
A2
A3
A4
A5
A6
A7
DIR
OE
B0
B1
B2
B3
B4
B5
B6
B7
VCC
GND
IDE_IOW
IDE_IOR
IDE_CS0#
IDE_CS1#
IDE_A0
IDE_A1
IDE_A2
BF533_PF13
74HC245
GND


U4
IDE_PORT_CS0
IDE_PORT_CS1
IDE_OE#
GND
3.3V
D0A D3B
D0B D3A
Q0 Q3
D1A D2B
D1B D2A
Q1 Q2
GND
VCC
7408
U3
BF533_A19
BF533_A18
BF533_A17
BF533_AMS3#
GND
3.3V
GND
3.3V
A0 Y0#
A1 Y1#
A2 Y2#
Y3#
E1# Y4#
E2# Y5#
E3 Y6#
GND Y7#
VCC
IDE_PORT_CS0
IDE_PORT_CS1
74LVC138
Title


3.3V
R255
10K
J1
RESET#
IDE_DATA7
IDE_DATA6
IDE_DATA5
IDE_DATA4
IDE_DATA3
IDE_DATA2
IDE_DATA1
IDE_DATA0
GND
IDE_IOW
IDE_IOR
3.3V
R252
10k
R256
10K
IDE_INT
R253
10k
3.3V
IDE_A1
IDE_A0
IDE_CS0#
R257
1K
D250
GND
IDE_DATA8
IDE_DATA9
IDE_DATA10
IDE_DATA11
IDE_DATA12
IDE_DATA13
IDE_DATA14
IDE_DATA15
IDE_A2
IDE_CS1#
GND
IDE-CONN DR40SM
IDE 40-PIN CONNECTOR

Conclusion

This EE-Note discusses a way to implement a FAT32 file system on the ADSP-BF533 processor. The system will achieve a reading speed of ~6 MBytes. The hard disk works in PIO mode with a transfer rate up to 8.3 MBytes.

The system can be built into a library in VisualDSP++® versions 3.5 and 4.0.

References

[1] *Information Technology - AT Attachment with Packet Interface-7, Volume 2, Parallel Protocols and Physical Interconnect (ATA/ATAPI-7 V2).* Revision 4b. 21 April 2004. American National Standard.

[2] *ADSP-BF533 Blackfin Processor Hardware Reference.* Rev 3.0, September 2004. Analog Devices, Inc.

[3] *ADSP-BF535 Blackfin EZ-KIT Lite CompactFlash Interface MP3 (EE-196).* Rev 1, June 2004. Analog Devices, Inc.

[4] *Microsoft Extensible Firmware Initiative FAT32 File System Specification,* Version 1.03, December 2000. Microsoft Corporation.

Document History

Revision	Description
Rev 1 – January 24, 2006 by Jenny Jing	Initial Release