Venkat RTOS: File System

Srinivasan, Venkat

October 17, 2016

1 Software Design

1.1 Space Management

Space is managed using bit vectors in this implementation. Both free space and allocated space is managed as such. The included BitMap .c file allows us to interact with the filesystem bitvector.
A new bitvector is created when the file system is formatted. Depending on the number of disk sectors I want to manage, I can allocate the same number of bytes by calling this function:

```
BitMap* BitMap_Init(uint32_t SizeInWords)
{
   BitMap* NewBitMap = (BitMap*) OS_NewAllocation(sizeof(BitMap) * 1);

   // if memory allocation failed, return immediately
   if (NewBitMap == 0)
   {
      return 0;
   }

   NewBitMap->dataarray = (uint32_t*) OS_NewAllocation(sizeof(uint32_t) * SizeInWords);

   // if memory allocation failed, clean up the created allocation and return
   if (NewBitMap->dataarray == 0)
   {
      OS_DestroyAllocation(NewBitMap, sizeof(BitMap));
      return 0;
   }

   NewBitMap->bitsize  = SizeInWords * WORD_SIZE;
   NewBitMap->wordsize  = SizeInWords;

   return NewBitMap;
}
```

The call to this function is usually done as such:

```
BitMap* DataMap = BitMap_Init(((MAX_BLOCKS_TRACKED/(sizeof(uint32_t)*8)) + 1)); // creates vector of
    MAX_BLOCK_TRACKED bits
```

MAX_BLOCK_TRACKED is a macro defined to be the number of data sectors the file system wants to track. Once that is done, setting and clearing the bits are done by calls to:

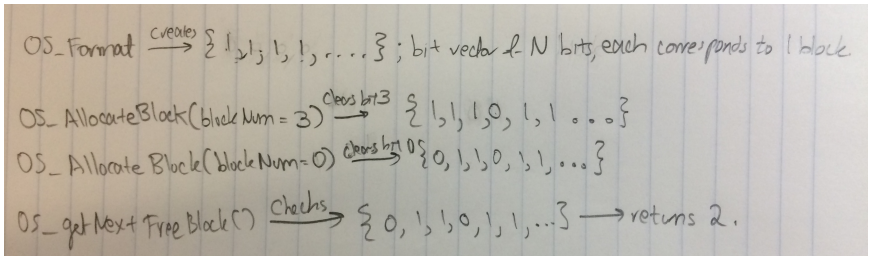
```
void BitMap_SetBit(BitMap* map, uint32_t BitNum)
{
    map->dataarray[BitNum/WORD_SIZE] |= 1 << (BitNum%WORD_SIZE); // Set the bit at the k-th position
        in A[i]
}

void BitMap_ClearBit(BitMap* map, uint32_t BitNum)
{
   map->dataarray[BitNum/WORD_SIZE] &= ~(1 << (BitNum%WORD_SIZE));
}

bool BitMap_TestBit(BitMap* map, uint32_t BitNum)
```

```
{
   return (bool) ((map->dataarray[BitNum/WORD_SIZE] & (1 << (BitNum%WORD_SIZE) )) != 0 ) ;
}
```

Calling these functions will set and clear specific bits in the large vector.
Whenever I want to allocate a block to a file, I call SetBit and set that particular bit to a 1. Whenever I want to free a block from a file, I call ClearBit and that particular bit is cleared. Any bits with 0 in them correspond to free blocks. I can find the next free block by calling:

```
bool _CheckBlockOccupancy(uint32_t BlockNum)
{
   if (BlockNum >= MAX_BLOCKS_TRACKED) return FALSE;

   BitMap* localBitMapRef = DataBitMap;

   return BitMap_TestBit(localBitMapRef, BlockNum);

}

// Returns the next free block. Does NOT mark the block as occupied.
uint32_t _GetNextFreeBlock()
{
   uint32_t BlockStart = 0;

   for (BlockStart = 0; BlockStart < MAX_BLOCKS_TRACKED; BlockStart++)
   {
      if (_CheckBlockOccupancy(BlockStart) == FALSE) return BlockStart;
   }

   OS_KernelPanic(FILESYS_OUTOFSECTORS, "_GetNextFreeInode");

   return 0; // never executes since kernel panics
}
```

Using this allows us to track free space management. This whole pattern is illustrated in this diagram:



1.2 File Directory

Both occupied space and free space is managed in the same exact way. I used a bit vector to represent each block in the disk. When the disk is initially formatted, the bit vector has 1s in every bit except for the blocks occupied by the metadata required for the filesystem.
In this implementation, the filesystem uses inodes to keep track of each file that is being created. The inode has the following structure:

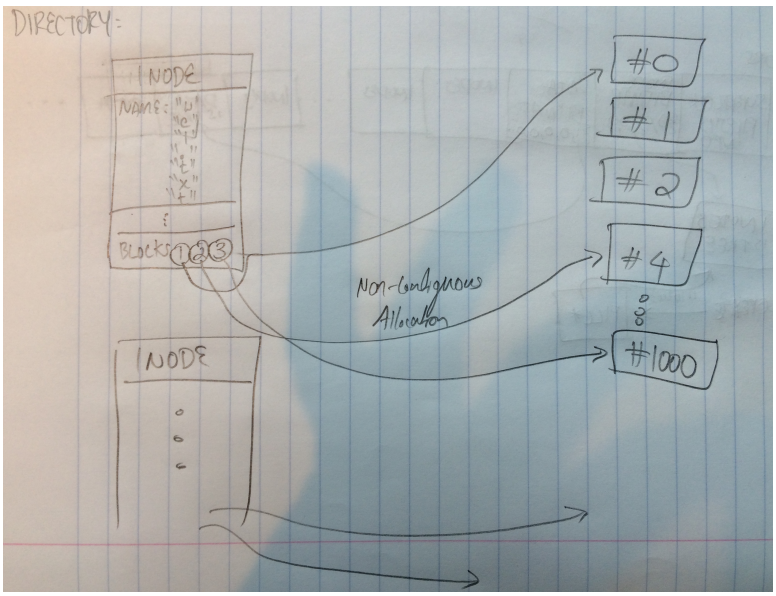
```
typedef struct nRTOS_FileNode
{                              // Total: 114 bytes per inode
   uint32_t INODE_NUM;              // 4 bytes
   uint32_t FILE_BYTES;             // 4 bytes; Increments of SECTOR_SIZE. Directly indicates how
       many blocks are used by file.
   uint32_t BYTES_USED;             // 4 bytes; The amount of bytes used by data stored within the
       file. Once it hits n*SECTOR_SIZE, need to expand file
   uint32_t LATEST_CURSOR;          // 4 bytes; The last written area of file (so that append can
       start appending from there
   char     FILE_NAME[MAX_FILE_NAME_CHARS]; // 10 bytes -- 10 characters alloId
   uint32_t BLOCKS_USED[MAX_FILE_SECTORS]; // 88 bytes -- 22 sectors alloId per file
```

```
} INODE;
```

Each inode contains all the fields listed above, bringing each individual inode to about 114 bytes in size. The inode contains information regarding its own number (for f2ile identification purposes), the amount of size allocated for the entire file (increments of sectors allocated for the file), the amount of data actually stored in the file (i.e. amount of data within each sector), the latest byte written to in the file, the name of the file (for identification purposes), and all the sector blocks allocated for this file.
When a new file is created, a single sector is allocated for the file, making the inode's FILE_BYTES field 512 bytes, and BYTES_USED field 0 bytes (since nothing has been written yet). The provided file name will be stored in the FILE_NAME array (character by character). **Due to the limitation in size of this array, only 10 characters are alloId for the file name.** Once the sector is allocated, the sector number is stored in the first index of BLOCKS_USED. As the file expands and contracts, the sectors allocated will be added and removed from this array.
The user will be delivered a FILE object when *open(filename)* is called.

```
typedef struct nRTOS_FileInfo
{
   INODE*     FileInode; // Inode associated with the file. Needs to be resident in memory.
} FILE;
```

When the user is provided this object, the user will be able to call file system calls and provide this object to interact with the specific file, as described in the section below.
The entire process can be described as shown in the picture:



1.3 File Allocation Scheme

When a new file is created, the file is given a full sector of 512 bytes, despite the fact that it does not use any data within that sector. The following code illustrates the idea:

```
   ...
   INODE* newFile = (INODE*) OS_NewAllocation(sizeof(INODE) * 1);
   ...
  // Initialize the new Inode with the proper items
  memcpy(newFile->FILE_NAME, fileName, strlen(fileName) * sizeof(char));
  newFile->FILE_BYTES = SECTOR_SIZE; // Every file starts with 512 bytes of data
  newFile->BYTES_USED = 0;
  newFile->LATEST_CURSOR = 0;
  uint32_t BlockToAssign = _GetNextFreeBlock();

  _MarkBlockAsOccupied(BlockToAssign);
  newFile->BLOCKS_USED[(newFile->FILE_BYTES/SECTOR_SIZE) - 1] = BlockToAssign; // if 512, index 0
      gets new block. If 1024, index 1 does.

  uint32_t InodeNumToAssign = _GetNextFreeInode();
```

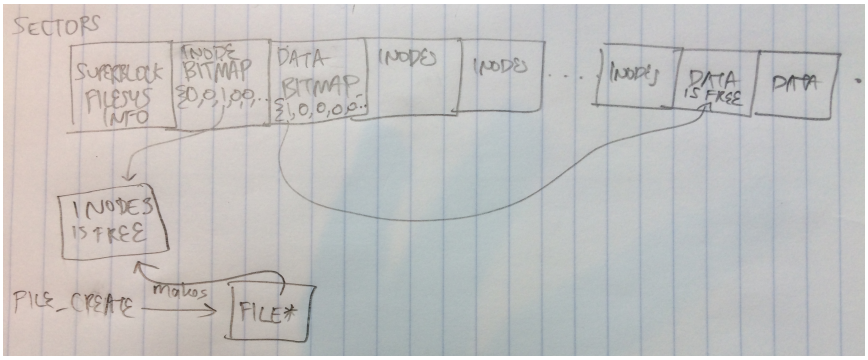
```
_MarkInodeAsOccupied(InodeNumToAssign);

newFile->INODE_NUM = InodeNumToAssign;

_FlushBitMapToDisk();
```

Once the sector is assigned, its corresponding BitMap bit is set to false and thus, the sector is marked as occupied. As the file expands or new data is written to area of the file outside its allocated bounds, new sectors are assigned as such:

```
 INODE* FileInode = fileDescriptor->FileInode;

if (FileInode == 0) return FALSE;                           // Invalid/corrupted Inode

uint32_t BlockListIndex = Offset / SECTOR_SIZE;             // If offset < 512, I need block 0. If
    < 1024, I need block 1, etc.
uint32_t BlocksAllocated = FileInode->FILE_BYTES / SECTOR_SIZE;   // Number of blocks already
    allocated

while (BlocksAllocated < (BlockListIndex + 1))                // Not enough blocks allocated yet
{
   if (BlocksAllocated >= MAX_FILE_SECTORS) return FALSE;          // Cannot create new space to
       index into this address
   // Give this file another block
   uint32_t BlockToAssign = _GetNextFreeBlock();

   _MarkBlockAsOccupied(BlockToAssign);                       // Assign the (BlockToAssign + 1)-th
       block to this space (assign 3rd block to space 2)
   FileInode->BLOCKS_USED[BlocksAllocated] = BlockToAssign;      // New block has been assigned to
       this
   FileInode->FILE_BYTES += SECTOR_SIZE;                    // Another block has been added
   BlocksAllocated = FileInode->FILE_BYTES/SECTOR_SIZE;
}

uint32_t BlockWanted = FileInode->BLOCKS_USED[BlockListIndex];
uint32_t IntraBlockIndex = Offset % SECTOR_SIZE;

memset(&tempBlock, 0, SECTOR_SIZE);

// Load the needed block from memory
_ReadFromFile((BYTE*) &tempBlock, BlockWanted);

BYTE* byteToStartWritingFrom = &tempBlock[IntraBlockIndex];

// The number of bytes that can be written within this block (in case of overlapping writes)
uint32_t ValidBytes = SECTOR_SIZE - IntraBlockIndex;

if (numBytes <= ValidBytes)
{
   memcpy(byteToStartWritingFrom, Buffer, numBytes);
   if (_WriteToFile((BYTE*) &tempBlock, BlockWanted))
   {
      uint32_t ByteWrittenTo = (((SECTOR_SIZE) * BlockListIndex) + IntraBlockIndex + numBytes);
      if (ByteWrittenTo > FileInode->BYTES_USED)
      {
         FileInode->BYTES_USED = ByteWrittenTo;
      }

      fileDescriptor->FileInode->LATEST_CURSOR = ((SECTOR_SIZE) * BlockListIndex) + IntraBlockIndex
          + numBytes;
   }
}
else
{
   // TODO: Need to load another block from SD
   // Do a recursive call by splitting the array
```

```
    // BlockSize - IntraBlockIndex is the size of the first array within this block
    // Remaining is put into offset+1
    memcpy(byteToStartWritingFrom, Buffer, ValidBytes);

    if (_WriteToFile((BYTE*) &tempBlock, BlockWanted))
    {
       uint32_t ByteWrittenTo = (((SECTOR_SIZE) * BlockListIndex) + IntraBlockIndex + numBytes);
       if (ByteWrittenTo > FileInode->BYTES_USED)
       {
          FileInode->BYTES_USED = ByteWrittenTo;
       }
    }

    // Recursive call
    // Buffer + validBytes: I're currently validBytes bytes into the buffer, so I ignore stuff
        already written
    // numBytes - validBytes: I've already written validBytes bytes out of the total numBytes byte
        count, so I subtract it
    // I're starting from square 0 at this new block due to contiguous overlap

    if (!OSFS_Write(fileDescriptor, Buffer + ValidBytes, numBytes - ValidBytes, Offset +
        (SECTOR_SIZE - Offset)))
    {
       return FALSE;
    }
 }

 return TRUE;
```

The code above basically writes to a particular location within the file. Suppose a file is only 512 bytes and the user wants to write to byte 2000. The logic featured above will repeat the loop and will create the necessary intermediate blocks in order to satisfy this write. This process is done recursively. In this way, I are able to allocate and deallocate blocks for the file. The following process can be illustrated as such:



1.4 Middle Level File Scheme

The filesystem provides the user with these abstraction interfaces:

```
bool    OSFS_Format();        // Call this whenever you want to erase the entire disk
FILE*      OSFS_Create(char* fileName); // Call this whenever a new file needs to be created
FILE*      OSFS_Open(char* fileName); // Call this Ihnever a file already created needs to be opened
/*
    Read from a particular location in file, and a particular number of bytes. If readoffset is
        larger than file, new blocks will be allocated.
*/
int32_t    OSFS_Read(FILE* fileDescriptor, BYTE* Buffer, uint32_t numBytes, uint32_t Offset);
/*
    Write to a particular location in file, and a particular number of bytes. If writeoffset is
        larger than file, new blocks will be allocated.
*/
bool    OSFS_Write(FILE* fileDescriptor, BYTE* Buffer, uint32_t numBytes, uint32_t Offset);
/*
    Append to the last written location in file
```

```
*/
bool    OSFS_Append(FILE* fileDescriptor, BYTE* buffer, uint32_t numBytes);
/*
    Close and flush changes from volatile to non-volatile memory
*/
bool    OSFS_Close(FILE* fileToClose);
/*
    Delete the file from the filesystem
*/
bool    OSFS_Delete(char* fileName);

// File Information functions
uint32_t   GetFileSize(FILE* fileToEval); // Returns the size of the file currently held
```

I designed the file system to have a very linux-like interface to the user. Whenever a user interacts with a file, there's a particular FILE object returned (upon the call to OSFS_Open()). Any further interaction with this file is not done with filenames anymore – rather, it is done by the user providing this FILE object to the function call. The Read/Write functions are very robust – they allow the user to read and write to any location within the file, **even locations that don't exist yet.** Suppose a file is 200 bytes and the user wants to write to the 2000th byte. The system will recursively allocate the necessary intermediate blocks and will then write the string provided by the user to this location. The file system also keeps track of the last written byte to the file (i.e. if someone writes to byte 500 and then to byte 100, the file system keeps track of byte 101). Any calls to OSFS_Append() allows the user to immediately append data to this location without needing to specify any particular offsets. OSFS_Close imply preserves these changes by writing the changed inode and data blocks back to non-volatile memory.

1.5 High Level Software System

There has been multiple additions to the interpreter that allows the user to create/edit/delete files. The following are the new commands:

```
char*       commandDef[]        =    {
                                    "help:          Output command information.",
                                    "creat:          Creates a new file and allows writes to it.",
                                    "del:         Deletes a file",
                                    "append:        Appends attached string to the end of the file",
                                    "printfile:     Prints content of file",
                                    "ls:            List all the files on the SD card.",
                                    "format:        Formats the entire filesystem"
                                };
char*         commandFormat[]     =    {
                                    "help",
                                    "creat <filename>",
                                    "del <filename>",
                                    "app <filename> <string>",
                                    "printfile <filename>",
                                    "format",
                                    "ls"
                                };
```

The above commands are self descriptive. Creat takes a filename and creates the file in the filesystem. Del takes a file name and deletes that corresponding file, informing the user if the delete was not successful (for example, if the file does not exist). Append adds the user provided text to the end of the file (it takes in the filename as Ill). And format basically formats the entire disk.

2 Measurements

2.1 Write Bandwidth

The following screenshots are for the bandwidth WRITE speed for the SD card. The time taken for writing is calculated by a call to OS_Time() before writing, and another call to OS_Time() after writing, and calculating the difference. Each unit is in a resolution of 20ns.

BandwidthDumps	unsigned long long[200]	0x20003A98
[0 ... 99]		
[0]	unsigned long long	65759
[1]	unsigned long long	133370
[2]	unsigned long long	133280
[3]	unsigned long long	133327
[4]	unsigned long long	131168
[5]	unsigned long long	133671
[6]	unsigned long long	131569
[7]	unsigned long long	131194
[8]	unsigned long long	133281
[9]	unsigned long long	943543
[10]	unsigned long long	131499
[11]	unsigned long long	133544
[12]	unsigned long long	131223
[13]	unsigned long long	133673
[14]	unsigned long long	133756
[15]	unsigned long long	133315
[16]	unsigned long long	133937
[17]	unsigned long long	133250
[18]	unsigned long long	133679
[19]	unsigned long long	133555
[20]	unsigned long long	131176
[21]	unsigned long long	133557
[22]	unsigned long long	131467
[23]	unsigned long long	133252
[24]	unsigned long long	133567
[25]	unsigned long long	133679
[26]	unsigned long long	131166
[27]	unsigned long long	133313
[28]	unsigned long long	134047
[29]	unsigned long long	133750
[30]	unsigned long long	131231
[31]	unsigned long long	133555
[32]	unsigned long long	131166
[33]	unsigned long long	133445

The same call to OS_ClearMsTime() before writing and OS_MsTime() after writing was made as Ill, to get the same exact bandwidth in terms of milliseconds. The following units are in terms of 1 ms (i.e. 3 means 3 ms).

BandwidthDumpsMS	unsigned int[200]	0x200040D8
[0 ... 99]		
[0]	unsigned int	1
[1]	unsigned int	2
[2]	unsigned int	2
[3]	unsigned int	2
[4]	unsigned int	2
[5]	unsigned int	2
[6]	unsigned int	2
[7]	unsigned int	2
[8]	unsigned int	2
[9]	unsigned int	18
[10]	unsigned int	2
[11]	unsigned int	2
[12]	unsigned int	2
[13]	unsigned int	2
[14]	unsigned int	2
[15]	unsigned int	2
[16]	unsigned int	2
[17]	unsigned int	2
[18]	unsigned int	2
[19]	unsigned int	2
[20]	unsigned int	2
[21]	unsigned int	2
[22]	unsigned int	2
[23]	unsigned int	2
[24]	unsigned int	2
[25]	unsigned int	2
[26]	unsigned int	2
[27]	unsigned int	2
[28]	unsigned int	2
[29]	unsigned int	2
[30]	unsigned int	2
[31]	unsigned int	2
[32]	unsigned int	2

2.2 Read Bandwidth

The following screenshots are for the bandwidth READ speed for the SD card. The time taken for writing is calculated by a call to OS_Time() before reading, and another call to OS_Time() after reading, and calculating the difference. Each unit is in a resolution of 20ns.

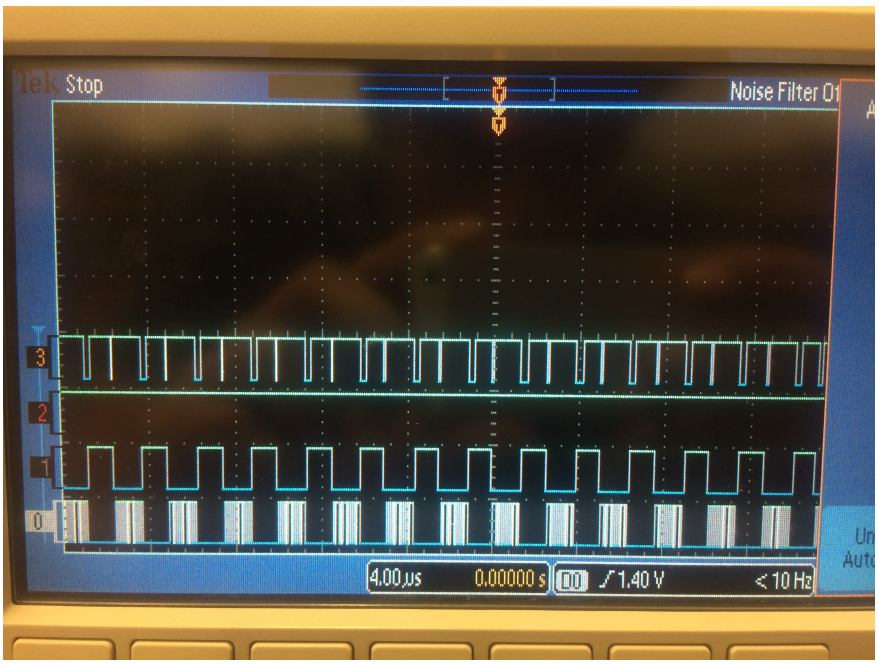
BandwidthDumps	unsigned long long[200]	0x20003A98
[0 ... 99]		
[0]	unsigned long long	89965
[1]	unsigned long long	83206
[2]	unsigned long long	83466
[3]	unsigned long long	83443
[4]	unsigned long long	83234
[5]	unsigned long long	83682
[6]	unsigned long long	83161
[7]	unsigned long long	83466
[8]	unsigned long long	83730
[9]	unsigned long long	83191
[10]	unsigned long long	83238
[11]	unsigned long long	83478
[12]	unsigned long long	83187
[13]	unsigned long long	83378
[14]	unsigned long long	83730
[15]	unsigned long long	83211
[16]	unsigned long long	83216
[17]	unsigned long long	83730
[18]	unsigned long long	83443
[19]	unsigned long long	83740
[20]	unsigned long long	83378
[21]	unsigned long long	83464
[22]	unsigned long long	83443
[23]	unsigned long long	83386
[24]	unsigned long long	83794
[25]	unsigned long long	83433
[26]	unsigned long long	83438
[27]	unsigned long long	83620
[28]	unsigned long long	83189
[29]	unsigned long long	83236
[30]	unsigned long long	83680
[31]	unsigned long long	83161
[32]	unsigned long long	83428
[33]	unsigned long long	83368

The same call to OS_ClearMsTime() before reading and OS_MsTime() after reading was made as Ill, to get the same exact bandwidth in terms of milliseconds. The following units are in terms of 1 ms (i.e. 3 means 3 ms).

BandwidthDumpsMS	unsigned int[200]	0x200040D8
[0 ... 99]		
[0]	unsigned int	1
[1]	unsigned int	1
[2]	unsigned int	1
[3]	unsigned int	1
[4]	unsigned int	1
[5]	unsigned int	1
[6]	unsigned int	1
[7]	unsigned int	1
[8]	unsigned int	1
[9]	unsigned int	1
[10]	unsigned int	1
[11]	unsigned int	1
[12]	unsigned int	1
[13]	unsigned int	1
[14]	unsigned int	1
[15]	unsigned int	1
[16]	unsigned int	1
[17]	unsigned int	1
[18]	unsigned int	1
[19]	unsigned int	1
[20]	unsigned int	1
[21]	unsigned int	1
[22]	unsigned int	1
[23]	unsigned int	1
[24]	unsigned int	1
[25]	unsigned int	1
[26]	unsigned int	1
[27]	unsigned int	1
[28]	unsigned int	1
[29]	unsigned int	1
[30]	unsigned int	1
[31]	unsigned int	1
[32]	unsigned int	1

2.3 SPI Clock Rate

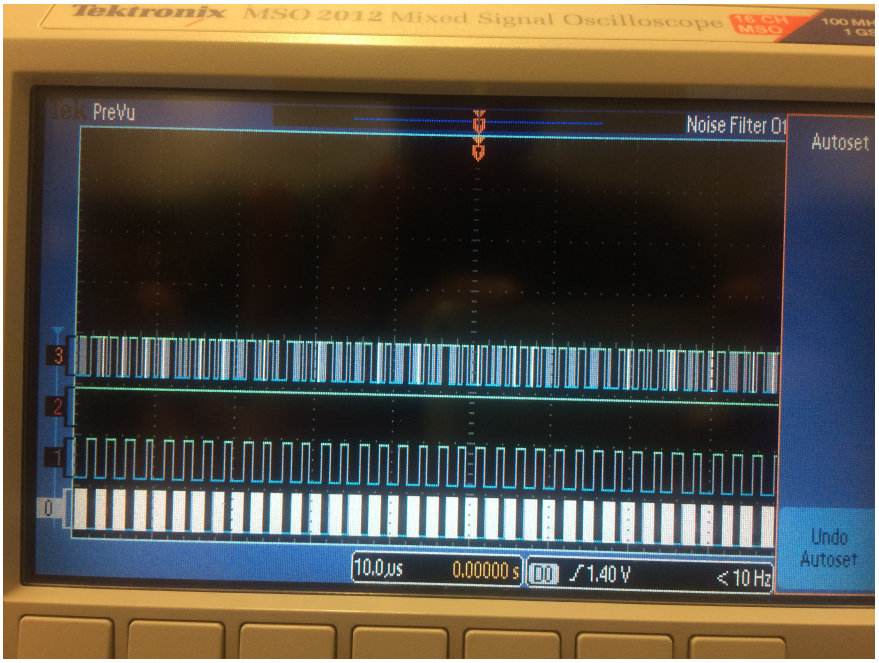
The SPI clock rate is set to be 250,000 KHz since the CPSDVSR value is set to 200 for a system clock of 50 MHz. Emprically, this can be shown by this oscilloscope trace featured below:



The SSI clock is shown in signal 1. Looking at the middle of that signal, I can see that each clock pulse has a duty cycle of about 1/3 of 4 μ seconds. Thus, each clock has a period of 2/3 of 4 μ seconds. This is about 375KHz, which is close to what I estimated. This is only during disk initialization. During disk IO, the speed goes up to 6.25MHz.

2.4 Two SPI Packets

I can demonstrate the transmission of 2 SPI packets as shown below. Each packet is a command packet, thus contains 8 bits. Together, they transmit 16 bits, which is what is shown below. The additional clock transmissions are the overhead for sending and recieving and a certain about of overlap with the transmission of the packet before and after the 2 packets featured.



3 Analysis and Discussion

3.1 External Fragmentation

No, my implementation does not have external fragmentation. None of my files require blocks to be contiguous in memory. Each file has an inode associated with it that tracks the blocks that are assigned to the file. Due

to this, a file can use the first block and the last block of the SD card and still work perfectly fine.

3.2 Internal Fragmentation

The **maximum** amount of wasted storage (assuming that only one byte is used in the last sector of each of the file) would be about 511*10 = 5110 bytes (the ACTUAL expected wastage would be much loIr).

3.3 Flash Memory

I would expect the bandwidth times to decrease – but not by a huge amount (flash non-volatile memory is sloIr than RAM, but the bulk of I/O times come from the need to go off chip to retrieve/store data).

3.4 File Count

Currently, I can store exactly 3950, since this is the number of blank inodes I write to disk when I format the disk (each file requires tne inode). I can increase the number of files I can store by increasing the number of blank inodes I write to disk when formatting.

3.5 Synchronous Access

Currently, there's no way for two files to stream debugging data into the same file. The way this could be achieved is by associating a semaphore with each FILE object and keeping track of files already open. This way, when a file is opened and a FILE object is created for that file, and another thread opens the same exact file, instead of creating a new FILE object, I would just return the same exact FILE object that was created earlier. This would allow the two threads to share the semaphore and when trying to request access to the file, they would call Wait and Signal as appropriate.
The FILE object would be changed as such:

```
typedef struct nRTOS_FileInfo
{
   INODE*     FileInode;              // Inode associated with the file. Needs to be resident in
       memory.
   Semaphore* SynchAccess;
} FILE;
```