Lesson 09: SD Card Interface

---

1. Introduction

A Secure Data (SD) card is a data storage device that can be used as massive storage in an embedded system. We will introduce a method to access data on a SD card using the Altera University Program (UP) SD Card IP core. We will use the **Media Computer** as a sample system that utilizes this interface. Please refer to the Qsys tool and the Media Computer documents for more information.

2. SD Card IP Core

The Altera University Program (UP) *SD Card* core is a hardware circuit that enables the use of an SD card on the Altera DE-series boards [1]. The core has been designed for use in a Qsys-implemented system. The **Media Computer** provided on the course's syllabus page should already contain a *SD Card core* [2-3]. Please refer to the Qsys tool to determine the base address for this core.

An SD card supports two operations modes: SD mode and SPI mode (serial peripheral interface). The SD mode is a proprietary format and uses four lines for data transfer. SPI is an open standard for serial interfaces and is widely used in embedded applications**. The SD Card IP core configures the card at system's initialization/reset to communicate using the SPI mode.**

Register Map:

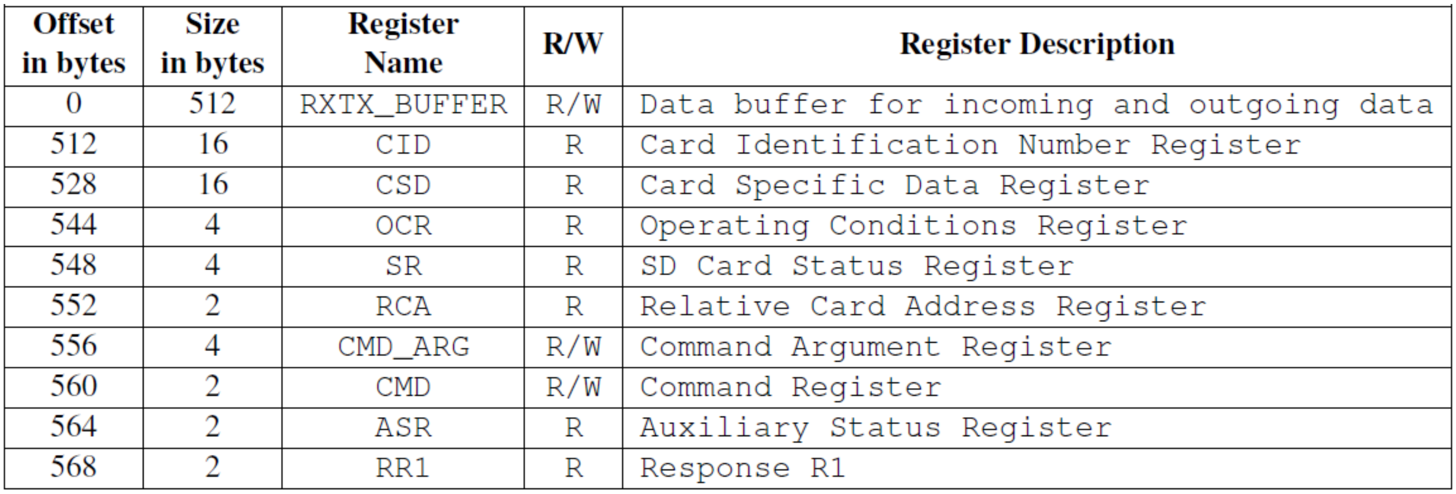
The memory-mapped registers allow a program running on the Nios II processor to read the status of the SD Card as well as send commands to it. The commands include reading, writing and erasing a block of data. When a command to read a block of data is issued, the core reads a **512-byte block of data** into a local memory buffer. Once the data is stored in the buffer, the buffer can be read and written to using memory reads/writes from a program.

The address offsets of the memory-mapped registers and the data buffer, relative to the starting address specified by the designer in the Qsys tool, are listed in Table 1.

Registers listed in Table 1 are accessible by reading and/or writing data to the corresponding memory locations. Registers *CID*, *CSD*, *OCR*, *SR*, and *RCA* are described in the *SD Card Physical Layer Specification* document [4]. The meaning of bits in these registers is described there. Although these registers contain useful information, users will primarily interface with an SD card using registers and buffers from the Altera IP Core which include the ***CMD_ARG***, ***CMD,*** and ***ASR*** registers, as well as the ***RXTX_BUFFER*** buffer.

The *SD Card* core abstracts the low-level SD card communication protocol using memory-mapped registers. It can transfer data to and from an SD card requiring only that users wait for each transaction to be completed. To facilitate this level of abstraction, the core uses three registers and a memory buffer.

Table 1. *SD Card* Core Register Map [1].



Offset in bytes	Size in bytes	Register Name	R/W	Register Description
0	512	RXTX_BUFFER	R/W	Data buffer for incoming and outgoing data
512	16	CID	R	Card Identification Number Register
528	16	CSD	R	Card Specific Data Register
544	4	OCR	R	Operating Conditions Register
548	4	SR	R	SD Card Status Register
552	2	RCA	R	Relative Card Address Register
556	4	CMD_ARG	R/W	Command Argument Register
560	2	CMD	R/W	Command Register
564	2	ASR	R	Auxiliary Status Register
568	2	RR1	R	Response R1

The *Auxiliary Status Register (ASR)* holds the status information for the core. The meaning of each bit is as follows:

- *bit 0* indicates if the last command sent to the core was valid.
- *bit 1* indicates if an SD Card is present in the SD card socket.
- *bit 2* indicates if the most recently sent command is still in progress.
- *bit 3* indicates if the current state of the SD card Status Register is valid.
- *bit 4* indicates if the last command completed due to a timeout.
- *bit 5* indicates if the most recently received data contains errors.

Once the card is initialized by the core, it can be accessed by issuing various commands via the *Command Argument register (CMD_ARG)* and the *Command (CMD)* registers. While the *SD Card* core supports a wide array of SD card functions (see Appendix A), the most frequently used commands are ***READ_BLOCK*** and ***WRITE_BLOCK***.

Reading a Sector:

When a *READ_BLOCK* command is issued, the data from an SD card is read in 512 byte blocks known as **sectors**. Once the block/sector is read, the ***RXTX_BUFFER*** can be accessed to read the data from that sector.

To execute the *READ_BLOCK* command, write the starting address of the block into the *Command Argument register (CMD_ARG)*. Then, write the *READ_BLOCK* command ID (**0x11**) to the *Command register (CMD)*. This sequence of events causes the SD Card core to read 1 sector (512 bytes) from the SD Card. When the command completes execution, the requested data will be accessible via the ***RXTX_BUFFER***.

Example: In this example, we first wait for the SD card to be connected to the SD card socket. Once a card is detected, we proceed to read the sector 480 (481th sector) on the SD card. The 481th th sector begins on byte 246,272 and ends on byte 246,783. Note that when the command to read data from the SD card has been sent, the program waits in a loop. This is because the operation may take some time and the data will not be available immediately. It is necessary to wait until the ***ASR* register** indicates that the read operation has been completed.

```
//Base Addresses from DE2-70 Media Computer with SD
#define SD_CARD_BASE_ADR     0x10003400
#define JTAG_UART_BASE_ADR    0x10001000
#define READ_BlOCK            0x11
#define SECTOR                480

/* function prototype */
void put_JTAG_UART_string(char * );

int main(void)
{
      int *command_argument_register = (int *)(SD_CARD_BASE_ADR + 556);
      short int *command_register    = (short int *)(SD_CARD_BASE_ADR + 560);
      short int *aux_status_register = (short int *)(SD_CARD_BASE_ADR + 564);
      volatile short int status;

      /* Wait for the SD Card to be connected to the SD Card Port. */




      /* print a message*/
      put_JTAG_UART_string ("\nCard connected.\0");

      /* Read a sector on the card */
                                                      // sector’s starting address
                                                      // read command to CMD register

      /* Wait until the operation completes. */




      /* print a message*/
      put_JTAG_UART_string ("\nRead sector operation completed.\0");

      return 0;
}
/* function to print a text string to the Terminal via JTAG UART */
void put_JTAG_UART_string(char * text_ptr)
{
       volatile int * JTAG_UART_ptr = (int *) JTAG_UART_BASE_ADR; // jtag_uart base address
       while ( *(text_ptr) )
       {
              if (*(JTAG_UART_ptr + 1) & 0xFFFF0000)   // if WSPACE > 0,
              {
                     *(JTAG_UART_ptr) = *(text_ptr);
                     ++text_ptr;
              }
       }
}
```

Writing to a Sector:

Executing *WRITE_BLOCK* is performed in the same manner as executing *READ_BLOCK* command. **However, before the *WRITE_BLOCK* is executed, the *RXTX_BUFFER* must be filled with 512 bytes of data to be written on the SD card**. Once the buffer contains the desired data, write the destination address to the *CMD_ARG register* (a multiple of 512 bytes as for the read command), and then write *WRITE_BLOCK* command ID (**0x18**) to the *CMD register*.

IMPORTANT: An SD card is a flash memory device, and as such writing to it takes longer than reading data from it. Also, each 512 block of data on an SD card can only be written a limited number of times (depending on the SD card used, this number varies between 1000 and 100000 times), thus users should take care to write to the SD card only when necessary.

Example: Write a C program that writes data to the 514th sector (sector 513) of the SD card. Remember to fill the *RXTX_BUFFER* before writing operation.

```
//Base Addresses from DE2-70 Media Computer with SD
#define SD_CARD_BASE_ADR      0x10003400
#define JTAG_UART_BASE_ADR    0x10001000
#define READ_BLOCK            0x11
#define WRITE_BLOCK           0x18
#define SECTOR                513

/* function to print a text string to the Terminal via JTAG UART */
void put_JTAG_UART_string(char * text_ptr)
{
      // code shown in the previous example
}
int main(void)
{
      int *command_argument_register = (int *)(SD_CARD_BASE_ADR + 556);
      short int *command_register = (short int *)(SD_CARD_BASE_ADR + 560);
      short int *aux_status_register = (short int *)(SD_CARD_BASE_ADR + 564);
      volatile short int status,i;

      /* Wait for the SD Card to be connected to the SD Card Port. */




      /* print a message*/
      put_JTAG_UART_string ("\nCard connected.\0");

      /* fill up buffer before writing to SD card */
      char* buffer = (char *) SD_CARD_BASE_ADR;


```

```
     /* Write to a sector on the card */
                                                    //sector’s starting address
                                            // write command to CMD register

     /* Wait until the operation completes. */




     /* print a message*/
     put_JTAG_UART_string ("\nWrite sector operation completed.\0");

     return 0;
}
```

3. Using Standard *stdio.h* Library

As we have seen in the previous examples, you will need to write your own function to display outputs in the terminal. You will also have to write your own function to accept inputs from users via the terminal. This process can be cumbersome.

The C compiler from the *Altera Monitor Program* supports standard C libraries. We will find it very useful to utilize the *stdio.h* library to support input and output via the *JTAG UART* core. The two useful functions are the *printf()* and *scanf()* functions. Some examples using these two functions are shown below. For more information, look at the format, parameters and other examples of these functions in a C programming book or an online references [5-6].

printf() examples [5]:

```
/* printf example */
#include <stdio.h>

int main()
{
   printf ("Characters: %c %c \n", 'a', 65);
   printf ("Decimals: %d \n", 1977);
   printf ("Preceding with blanks: %10d \n", 1977);
   printf ("Preceding with zeros: %010d \n", 1977);
   printf ("Some different radixes: %d %#x \n", 100, 100);
   printf ("floats: %4.2f %4.2f \n", 3.1416);
   printf ("%s \n", "A string");
   return 0;
}
```

Results:

```
Characters: a A
Decimal: 1977
Preceding with blanks:       1977
Preceding with zeros: 0000001977
Some different radixes: 100 0x64
float: 3.14
A string
```

`scanf()` examples [5]:

```c
/* scanf example */
#include <stdio.h>

int main ()
{
  char str [80];
  int i;

  printf ("Enter your family name: ");
  scanf ("%s",str);
  printf ("Enter your age: ");
  scanf ("%d",&i);
  printf ("Mr. %s, %d years old.\n",str,i);
  printf ("Enter a hexadecimal number: ");
  scanf ("%x",&i);
  printf ("You have entered %#x (%d).\n",i,i);

  return 0;
}
```

Results:

```
Enter your family name: John
Enter your age: 29
Mr. John, 29 years old.
Enter a hexadecimal number: ff
You have entered 0xff (255).
```

4. FAT16 File System

FAT, which was developed by Microsoft, is the most widely used file system for SD cards. The FAT16 can support storage sizes up to 4 GB. Files stored in the FAT16 file system can be read and written by almost all computers and microcontrollers. Before the SD card can be used with a FAT16 file system, you will need to format it.

Formatting an SD card with FAT16:

In the window explorer, right click on the SD card drive (i.e. E:\) and select *Format*. From the pop-up window, select *FAT (Default)* as the *File System*. You can enter *Volume label* for the SD card (optional). Select *Quick Format* if you don't want to erase old data on the card.

FAT16 File System Structure

The basic layout of the SD card that has been formatted with FAT16 file system is shown below. More information can be found in the references [7-9]. The layout without a *Master Boot Record (MBR)* is shown below.

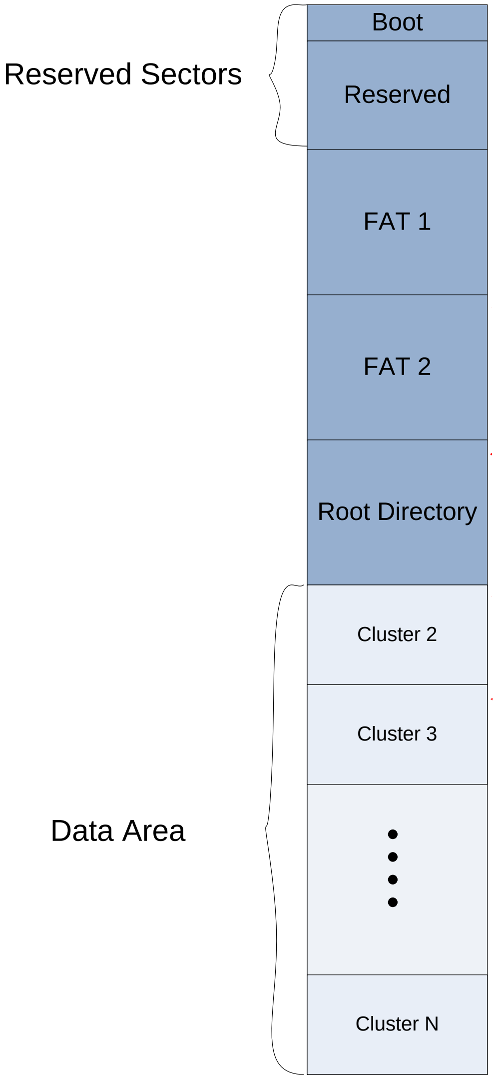


Fig. 1. *SD Card* FAT16 file system layout without an *MBR* sector.

The *Master Boot Record (MBR)*

While some of the newer SD cards **do not contain a *MBR* section as shown in Fig. 1**, most cards do have this section at the beginning (sector 0) of the cards as shown in Fig. 2 below. The layout of an SD card with an *MBR* section is shown below.

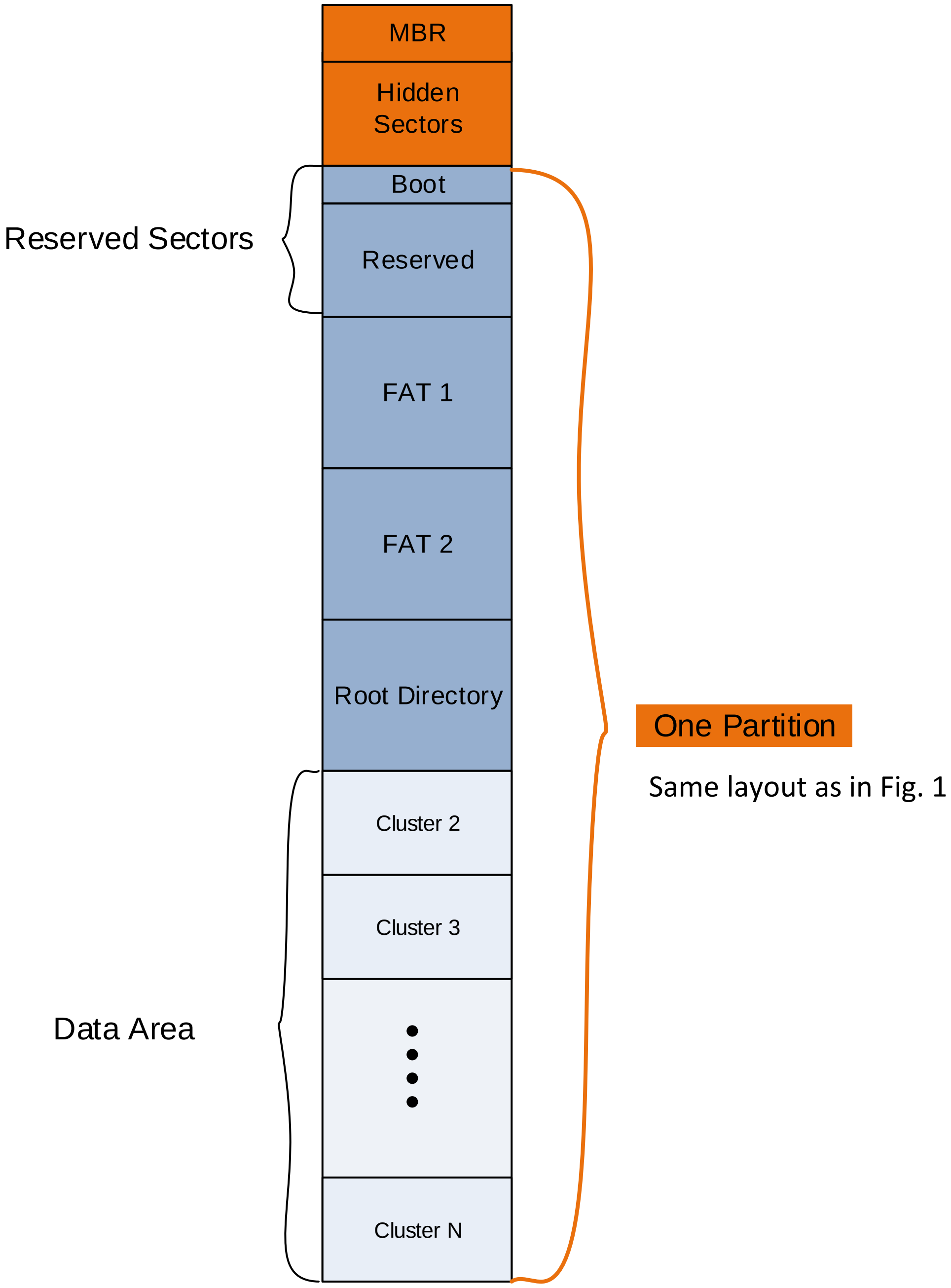


Fig. 2. *SD Card* FAT16 file system layout with an *MBR* sector.

The *MBR* section is located at the first sector (sector 0) of the SD card. The *MBR* contains information to locate data partition(s) within the card. The basic layout of the *MBR* is shown in Table 1 below.

Each partition entry in the table above contains specific information about that partition. The basic layout for each partition entry is shown in Table 2. Note that the offsets are calculated from the start of partition entry in the *MBR* table above.

For our class, we can expect to have **one partition in each SD card**.

Table 1. Layout of a *Master Boot Record (MBR).*

Offset (hex)	Description	Size
000	Executable code	446 Bytes
1BE	Partition 1 entry	16 Bytes
1CE	Partition 2 entry	16 Bytes
1DE	Partition 3 entry	16 Bytes
1DE	Partition 4 entry	16 Bytes
1FE	Executable marker (0x55 and 0xAA)	2 Bytes

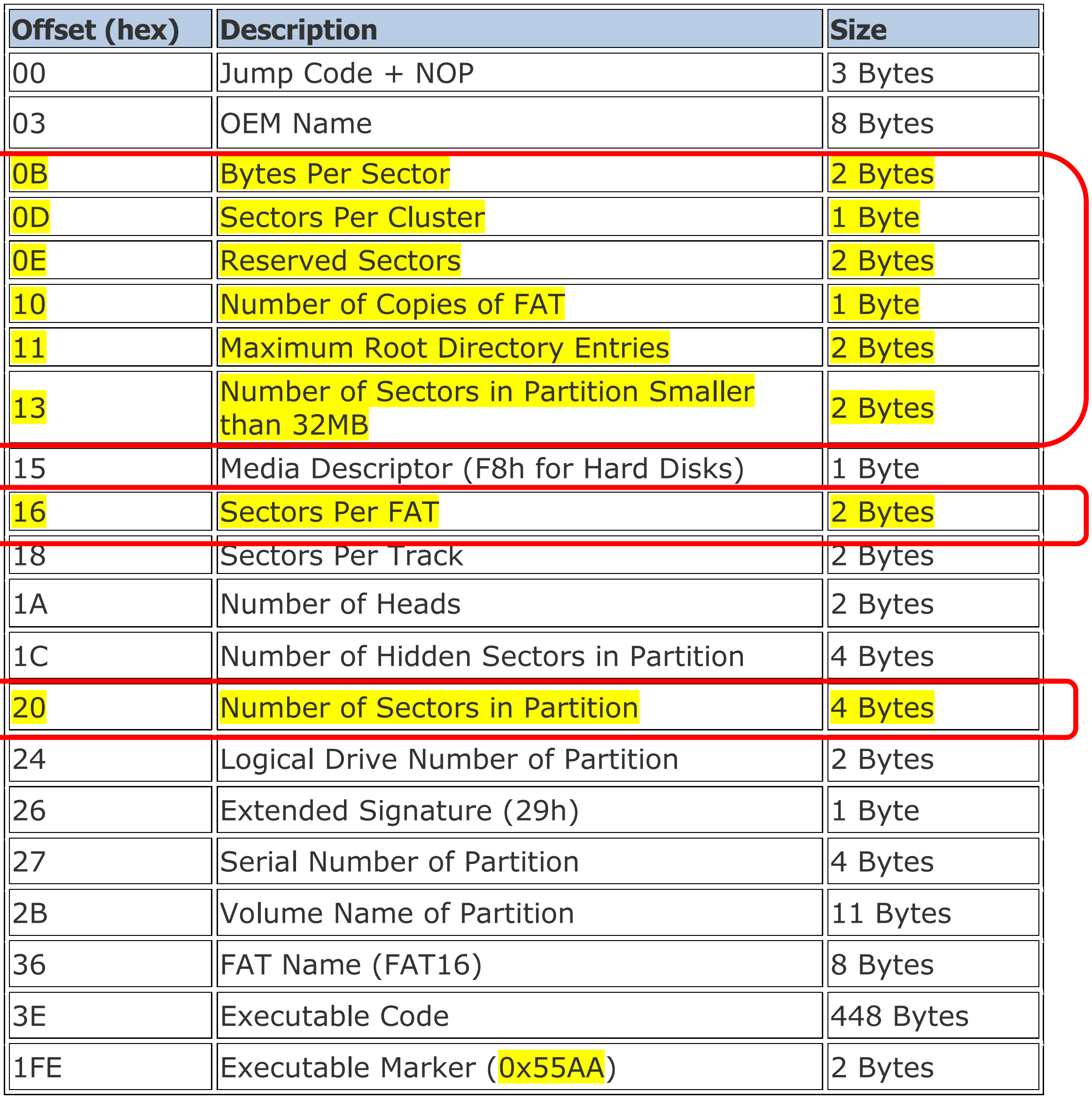
Table 2. Layout of one partition entry in the *Master Boot Record (MBR)*.

Offset (hex)	Description	Size	Comment
00	State	1 Byte	0x80 (active), 0x00 (inactive)
01	Start Head	1 Byte	
02	Start Cylinder/Sector	2 Bytes	Start Sector and Cylinder
04	Partition Type	1 Byte	0x01 = FAT12 0x04 = FAT16 (<32MB) 0x05 = Ex MSDOS 0x06 = FAT16 (>32 MB) 0x0B = FAT32
05	End Head	1 Byte	
06	End Cylinder/Sector	2 Bytes	End Sector and Cylinder
08	Start Sector	4 Bytes	Start Sector of Partition 1
0C	Partition Length	4 Bytes	Number of Sectors in the Partition

The *Boot Record*

The *Boot Record* is located at the first sector of the partition (or *sector 0* if an SD card does not have an *MBR* section.) It contains important information about the card format, structure, etc. The basic layout of the Boot Record is shown in Table 3 below [7]. Some of the most commonly used fields are highlighted below.
Note that the offset numbers are determined from the starting address of the *Boot Record* sector.

Table 3. *Boot Record* layout of a FAT16 file system [7].



Offset (hex)	**Description**	**Size**
00	Jump Code + NOP	3 Bytes
03	OEM Name	8 Bytes
0B	Bytes Per Sector	2 Bytes
0D	Sectors Per Cluster	1 Byte
0E	Reserved Sectors	2 Bytes
10	Number of Copies of FAT	1 Byte
11	Maximum Root Directory Entries	2 Bytes
13	Number of Sectors in Partition Smaller than 32MB	2 Bytes
15	Media Descriptor (F8h for Hard Disks)	1 Byte
16	Sectors Per FAT	2 Bytes
18	Sectors Per Track	2 Bytes
1A	Number of Heads	2 Bytes
1C	Number of Hidden Sectors in Partition	4 Bytes
20	Number of Sectors in Partition	4 Bytes
24	Logical Drive Number of Partition	2 Bytes
26	Extended Signature (29h)	1 Byte
27	Serial Number of Partition	4 Bytes
2B	Volume Name of Partition	11 Bytes
36	FAT Name (FAT16)	8 Bytes
3E	Executable Code	448 Bytes
1FE	Executable Marker (0x55AA)	2 Bytes

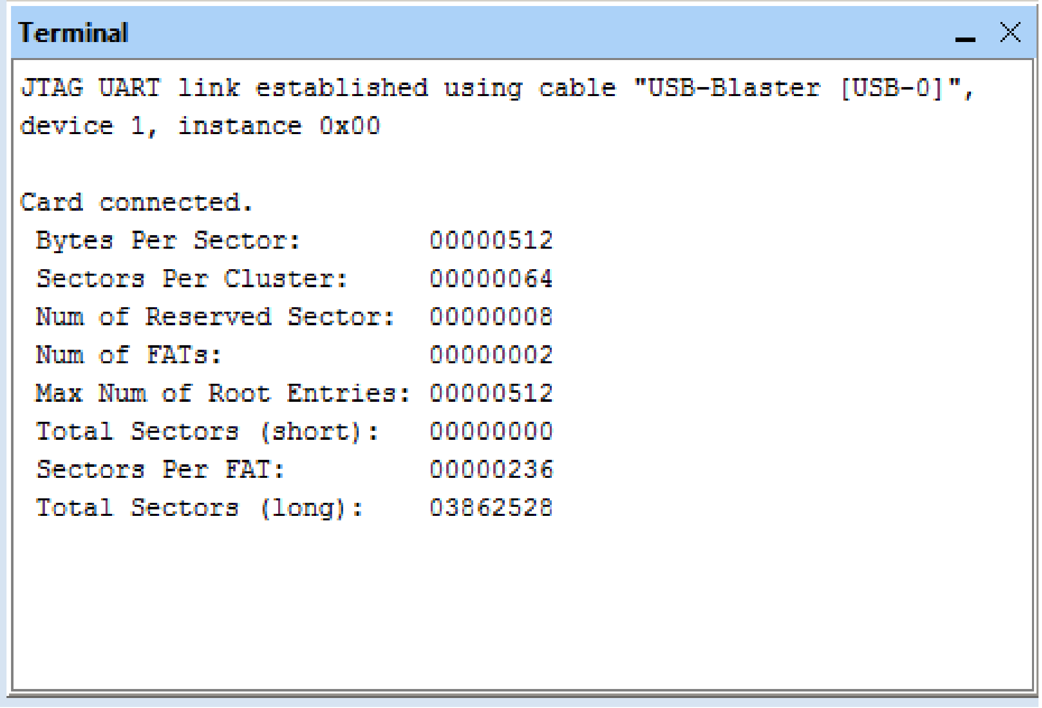
Example: Write a C program to extract the eight data fields highlighted above from the Boot Record sector.

```
#include <stdio.h>
//Base Addresses from DE2-70 Media Computer with SD
#define SD_CARD_BASE_ADR   0x10003400
#define JTAG_UART_BASE_ADR 0x10001000
#define READ_BlOCK         0x11       //read command
#define WRITE_BLOCK        0x18       //write command
```

```
typedef struct FAT16BootSector {
      unsigned short  BytesPerSector;
      unsigned char   SectorsPerCluster;
      unsigned short  NumReservedSectors;
      unsigned char   NumFATs;
      unsigned short  MaxNumRootEntries;
      unsigned short  TotalSectorsShort;
      unsigned short  SectorsPerFAT;
      unsigned int    TotalSectorsLong;
}FAT16BootSector;

/* function prototype */
void read_BootSector ();
void print_BootSector ();

/* global variable */
int *command_argument_register = (int *)    (SD_CARD_BASE_ADR + 556);
short int *command_register = (short int *)(SD_CARD_BASE_ADR + 560);
short int *aux_status_register = (short int *)(SD_CARD_BASE_ADR + 564);
volatile short int status;
FAT16BootSector Entry;     // structure to store the Boot Record information

/* main function */
int main(void)
{
      /* Wait for the SD Card to be connected to the SD Card Port. */
      status = (short int) *(aux_status_register);
      while ((status & 0x02) == 0)
            status = (short int) *(aux_status_register);

      /* print a message*/
      printf ("\nCard connected.\0");

      /* call function to read Boot sector*/
      read_BootSector();

      /* call function to print Boot sector*/
      print_BootSector();

      return 0;
}
```

```
void print_BootSector ()
{
      printf ("\n Bytes Per Sector:        %08d",                      );
      printf ("\n Sectors Per Cluster:     %08d",                         );
      printf ("\n Num of Reserved Sector:  %08d",                          );
      printf ("\n Num of FATs:             %08d",             );
      printf ("\n Max Num of Root Entries: %08d",                         );
      printf ("\n Total Sectors (short):   %08d",                         );
      printf ("\n Sectors Per FAT:         %08d",                     );
      printf ("\n Total Sectors (long):    %08d",                        );
}
/* function to read the Boot Record at sector 0 (without MBR section) */
void read_BootSector ()
{
      char *byte; short *dbyte; int * word;

      /* Read a sector 0 on the card */
      *(command_argument_register) = (0 << 9);    // starting address
      *(command_register)   = READ_BLOCK;    // write command to CMD register

      /* Wait until the operation is completed. */
      status = ((short int) *(aux_status_register) );
      while ((status & 0x04)!=0) status = ((short int) *(aux_status_register) );

      /* Populate Boot Entry structure */
                                                        // Bytes per Sector


                                                       // Sector per Cluster


                                                       // No of Reserved Sectors


                                                       // No of FATs


                                                       // Max Num of Root Entries


                                                       // Total Sectors (short)


                                                       // Sectors per FAT


                                                       // Total Sectors (long)


}
```

A screenshot of the execution of the example above is shown below.



```
JTAG UART link established using cable "USB-Blaster [USB-0]",
device 1, instance 0x00

Card connected.
 Bytes Per Sector:         00000512
 Sectors Per Cluster:      00000064
 Num of Reserved Sector:   00000008
 Num of FATs:              00000002
 Max Num of Root Entries:  00000512
 Total Sectors (short):    00000000
 Sectors Per FAT:          00000236
 Total Sectors (long):     03862528
```

Fig. 3. Results of a program to read the *Boot Record* information.

The File Allocation Tables (FAT1 and FAT2)

When a file is saved on an SD card with FAT16 file system, it is divided into one or more data clusters (size of a cluster can be defined when an SD card is formatted). **In the example shown in Fig. 3, the cluster size is 64 sectors = (64 × 512) = 32768 bytes.** From an application program's point of view, a file is a linear contiguous storage space in which data are accessed sequentially. But that may not be the case at the physical level. A large file may be divided and stored in many clusters that may or may not be continuous in storage space. **They are connected in a linked-list like manner**. A conceptual view of file stored in a FAT16 file system is shown in Fig. 4. In this illustration, *myfile.txt* file is allocated to **clusters 4, 5, 8, and 9**. Each cluster has an entry in the ***File Allocation Table*** (**FAT**) that indicates to next cluster in the file.

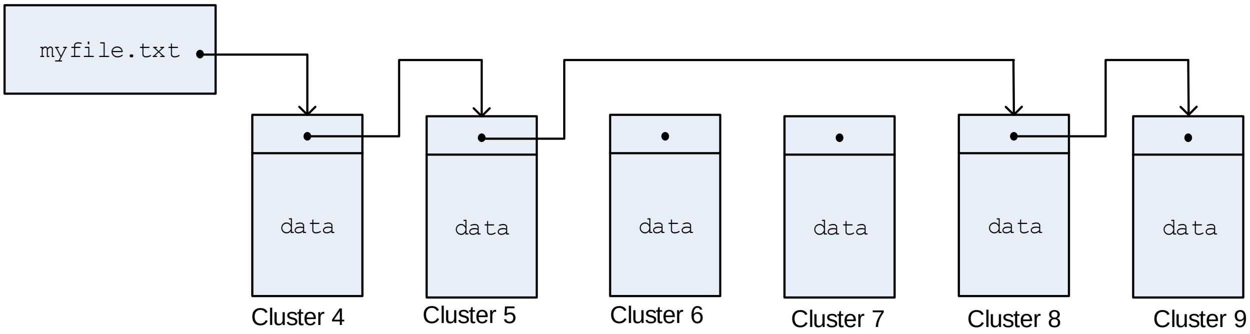


Fig. 4. A conceptual view of myfile.txt file stored with FAT16 file system [10].

There are normally two identical FAT tables to prevent data corruption in the FAT tables. The starting sector of ***FAT1*** table is located at the first sector after the ***Reserved Sectors*** (see Fig. 2 and Fig. 3).

Question: What is starting sector of *FAT1* based on the *Boot Record* shown in Fig. 3?

The starting sector of ***FAT2*** is located at the first sector after the ***FAT1*** table (see Fig. 2 and Fig. 3).

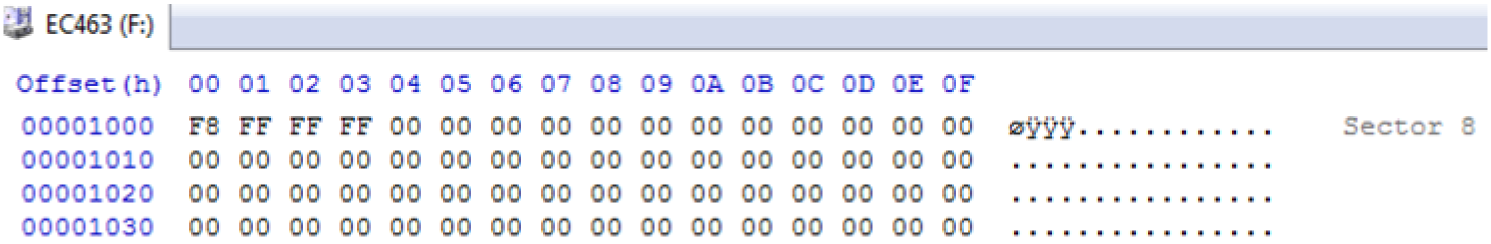
Question: What is the sector number of the starting sector of FAT2?

Each cluster is identified by a 2-byte cluster entry in the FAT table. The first two clusters are not used (and always 0xF8FF and 0xFFFF). Data area starts with cluster 2.

Important facts:

- A 0x0000 in the FAT entry indicates that the cluster does not contain data.
- A 0xFFFF in in the FAT entry indicates that this is the last entry in the linked list (no cluster in after this one).
- Any other numbers in in the FAT entry indicates the next cluster in the linked list.
- Clusters 0 and 1 are not used (always 0xF8FF and 0xFFFF).

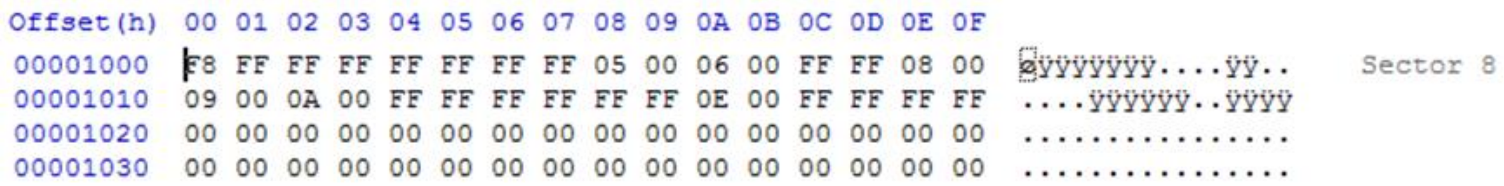
A portion of the memory layout of first sector in FAT1 is shown below. Note that all entries (except the first two clusters) contain 0s. This means that the SD card does not contain any files (empty SD card).



```
Offset(h)  00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F

00001000   F8 FF FF FF 00 00 00 00 00 00 00 00 00 00 00 00  øÿÿÿ............   Sector 8
00001010   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00001020   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00001030   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Fig. 5. A portion of *FAT1* layout of an empty SD card.

A portion of the memory layout of first sector in FAT1 of another SD card is shown below. Note that this SD card contains data files as indicated by the entries in the FAT table.



```
Offset(h) 00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
00001000  F8 FF FF FF FF FF FF FF 05 00 06 00 FF FF 08 00  øÿÿÿÿÿÿÿ....ÿÿ..    Sector 8
00001010  09 00 0A 00 FF FF FF FF FF FF 0E 00 FF FF FF FF  ....ÿÿÿÿÿÿ..ÿÿÿÿ
00001020  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00001030  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

Fig. 6. A portion of *FAT1* layout of an SD card that contains data files.

Question: What is the next cluster in the cluster chain that has cluster 2?

Question: What is the next cluster in the cluster chain that starts with cluster 7?

Question: List all clusters in the chain that starts with cluster 7.

Question: What would a FAT1 table look like if the SD card contains only *myfile.txt* file as shown in Fig. 4.

Recall that *myfile.txt* file is allocated to **clusters 4, 5, 8, and 9.** Blank cells contain 0x00.

Offset	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**A**	**B**	**C**	**D**	**E**	**F**
1000	F8	FF	FF	FF												
1010																
1020																

The Root Directory

The *Root Directory* contains the file structure of the SD card. The *Root Directory* is located at the first sector after the last FAT table.

Question: What is the starting sector of the *Root Directory*?

Each file or folder is described with a **32-byte** entry in the *Root Directory*. The layout of each entry is shown in the table below. Note that the offset numbers are determined based on the starting address of the *Root Directory*.

Table 4. Layout of an entry in the *Root Directory* [7].

Offset (hex)	**Description**	**Size**	**Comment**
00	DOS Filename	8 Bytes	ASCII code
08	DOS File Extension	3 Bytes	ASCII code
0B	File Attributes	1 Byte	
0C	NT Case Info	1 Byte	
0D	Create Time (ms)	1 Byte	10ms Units
0E	Create Time (Hrs/Mins/Secs)	2 Bytes	Hrs: bits 15:11 Mins: bits 10:5 Secs: bits 4:0
10	Create Date (Yr/Mo/Da)	2 Bytes	Yr: bits 15:9 (offset = 1980) Mo: bits 8:5 Da: bits 4:0
12	Last Access Date	2 Bytes	Same format as offset 0x10
14	File / Folder Start Cluster (High)	2 Bytes	Only used in FAT32 Systems
16	Last Modified Time	2 Bytes	Same format as offset 0x0E
18	Last Modified Date	2 Bytes	Same format as offset 0x10
1A	File / Folder Start Cluster (Low)	2 Bytes	
1C	File Size (Bytes)	4 Bytes	Folders will have a File Size of 0x0000

Question: How many sectors does *Root Directory* contain? Use information from the Boot Record shown in Fig. 3.

The Data Area

The ***Data Area*** is located right after the ***Root Directory***. Files are stored in clusters as discussed earlier. Cluster size can be configured when the SD card is formatted. For example, the SD card shown in Fig. 3 contains clusters that are 64-sector long. Again, *cluster 2* is the first valid cluster right after the *Root Directory* (cluster 0 and 1 are not available).

Question: What is the starting sector of the *cluster 2* (first cluster in the *Data Area)*? Use information from the Boot Record shown in Fig. 3.

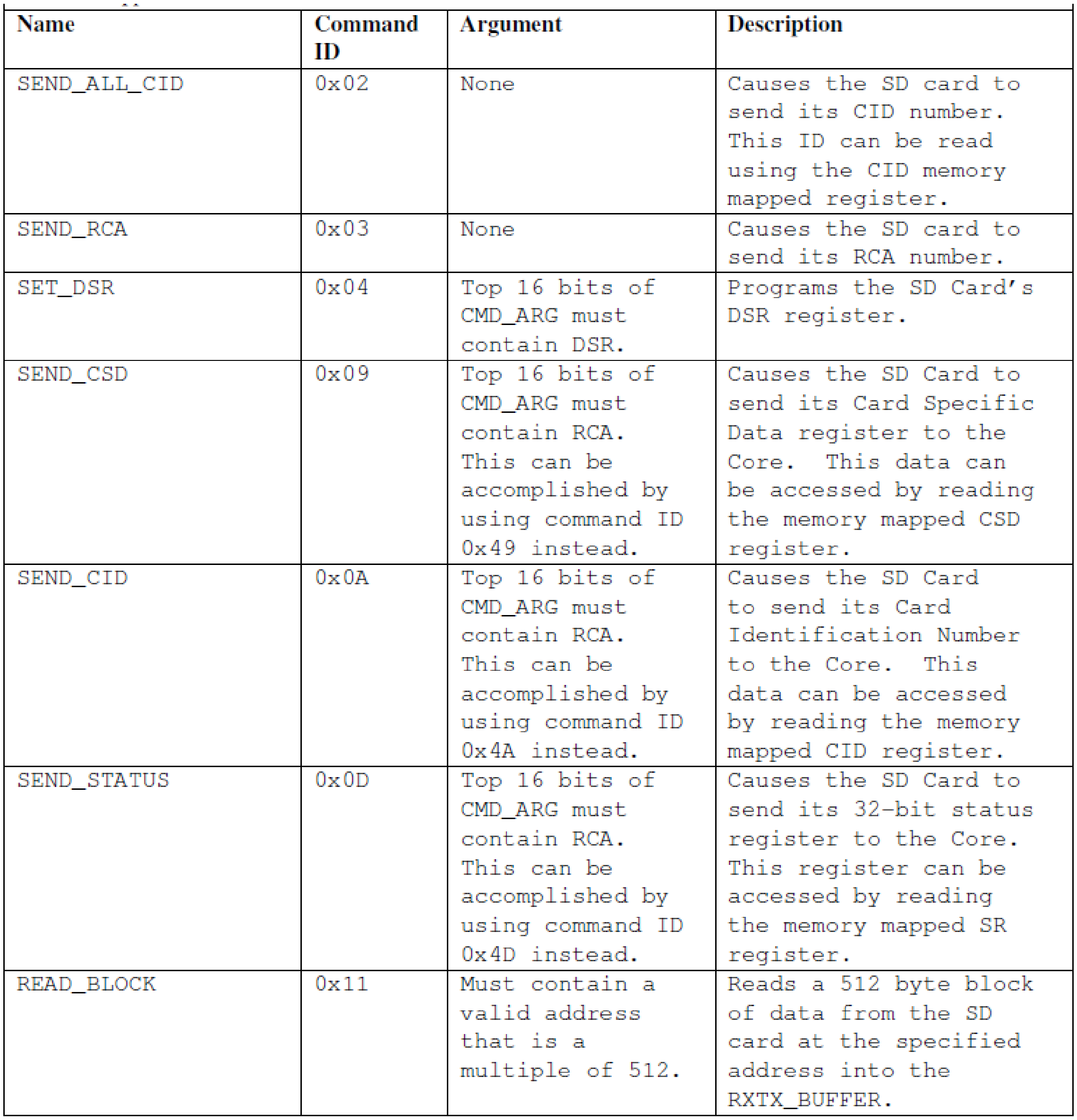
Question: What is the starting sector of the *cluster 3* (second cluster in the *Data Area)*? Use information from the Boot Record shown in Fig. 3.

5. References

[1] Altera, "Altera University Program Secure Data Card IP Core", *for Quartus II v.13.0*, May 2013.
[2] Altera, "Media Computer System for the Altera DE2-70 Board," *for Quartus II v.13.0*, May 2013.
[3] Altera, "Media Computer System for the Altera DE2 Board," *for Quartus II v.13.0*, May 2013.
[4] SD Group and SD Card Association, "SD Specifications Part 1: Physical Layer Simplified Specification," ver. 1.10, April 2006.
[5] http://www.cplusplus.com/reference/cstdio/printf/
[6] http://www.cplusplus.com/reference/cstdio/scanf/
[7] http://home.teleport.com/~brainy/fat16.htm
[8] http://en.wikipedia.org/wiki/File_Allocation_Table
[9] http://pjgcreations.blogspot.com/2011/03/fat16-file-system-with-sd-cards.html
[10] Chu, *Embedded SoPC Design With Nios II Processor and VHDL Examples*, John Wiley and Sons Inc., 2011.

APPENDIX A – SD Card Commands

Supported SD Card Commands [1].



Name	Command ID	Argument	Description
SEND_ALL_CID	0x02	None	Causes the SD card to send its CID number. This ID can be read using the CID memory mapped register.
SEND_RCA	0x03	None	Causes the SD card to send its RCA number.
SET_DSR	0x04	Top 16 bits of CMD_ARG must contain DSR.	Programs the SD Card's DSR register.
SEND_CSD	0x09	Top 16 bits of CMD_ARG must contain RCA. This can be accomplished by using command ID 0x49 instead.	Causes the SD Card to send its Card Specific Data register to the Core. This data can be accessed by reading the memory mapped CSD register.
SEND_CID	0x0A	Top 16 bits of CMD_ARG must contain RCA. This can be accomplished by using command ID 0x4A instead.	Causes the SD Card to send its Card Identification Number to the Core. This data can be accessed by reading the memory mapped CID register.
SEND_STATUS	0x0D	Top 16 bits of CMD_ARG must contain RCA. This can be accomplished by using command ID 0x4D instead.	Causes the SD Card to send its 32-bit status register to the Core. This register can be accessed by reading the memory mapped SR register.
READ_BLOCK	0x11	Must contain a valid address that is a multiple of 512.	Reads a 512 byte block of data from the SD card at the specified address into the RXTX_BUFFER.

Name	**Command ID**	**Argument**	**Description**
WRITE_BLOCK	0x18	Must contain a valid address that is a multiple of 512.	Write a 512 byte block of data from the RXTX_BUFFER to the SD card at the specified address.
SET_WRITE_PROTECT	0x1C	Must contain a valid address that is a multiple of 512.	Sets a flag that designates the block to be write-protected.
CLR_WRITE_PROTECT	0x1D	Must contain a valid address that is a multiple of 512.	Clears a flag that designates the block to be write-protected.
ERASE_BLOCK_START	0x1E	Must contain a valid address that is a multiple of 512.	Specifies the block address where earsing should begin.
ERASE_BLOCK_END	0x1F	Must contain a valid address that is a multiple of 512.	Specifies the last block to be erased.
ERASE	0x26	None	Erases the previously selected array of blocks on the SD card.
APP_CMD	0x38	Top 16 bits should contain RCA. This can be accomplished by using command code 0x78 instead.	Allows the next instruction to be executed to be an Application Specific Instruction, as defined by the SD Card Physical Layer Specification 1.10 document.