fat12.c

Go to the documentation of this file.

```
00001 /*!     \file fs/fat12/fat12.c
00002  *      \brief FAT-12 fyle system.
00003  *      \author
00004  *              Matteo Chesi <dalamar@inwind.it>
00005  *              Rudy Manganelli <feller@libero.it>
00006  *              Andrea Righi <drizzt@inwind.it>
00007  *      \date Last update: 2003-11-09
00008  *      \note Copyright (&copy;) 2003
00009  *              Matteo Chesi <dalamar@inwind.it>
00010  *              Rudy Manganelli <feller@libero.it>
00011  *              Andrea Righi <drizzt@inwind.it>
00012  */
00013
00014 #include <const.h>
00015 #include <string.h>
00016
00017 #include <arch/mem.h>
00018
00019 #include <kernel/console.h>
00020 #include <kernel/floppy.h>
00021 #include <kernel/keyboard.h>
00022 #include <kernel/kmalloc.h>
00023
00024 #include <kernel/fat.h>
00025
00026 // FAT global variables //
00027 bootsect_t bootsector;
00028 FAT12_t fat;
00029 logical_FAT12_t lfat;
00030 word directory=0;
00031 char path[256];
00032
00033 // ---------- File system procedures ---------- //
00034 bool next_sector(word *next, word actual)
00035 {
00036         *next = lfat.data[actual];
00037         if ( (*next == 0) || (*next >= 0x0FF0) )
00038                 return(FALSE);
00039         else
00040                 return(TRUE);
00041 }
00042
00043 int how_many_cluster(word start)
00044 {
00045         word c=1, cl;
00046
00047         // The root directory (start=0) has a fixed size                        //
00048         if (!start)
00049                 return( (bootsector.RootDirectoryEntries*sizeof(FileEntry_t))/bootsector.BytesPerSector );
00050
00051         // Calculate the size of the directory (it's not the root!)             //
00052         cl=start;
00053
00054         while (next_sector(&cl, cl)) c++;
00055         return(c);
00056 }
00057
00058 void int_to_date(date_t *d, word date)
00059 {
00060         d->year=date/512;
00061         d->month=(date-(d->year*512))/32;
00062         d->day=date-(d->year*512)-(d->month*32);
00063 }
00064
00065 void int_to_time(fat_time_t *time, word t)
00066 {
00067         time->hour=t/2048;
00068         time->minute=(t-(time->hour*2048))/32;
00069         time->second=(t-(time->hour*2048)-(time->minute*32))*2;
00070 }
```

```
00071
00072 void read_attrib(attrib_t *FileAttr, byte attrib)
00073 {
00074         FileAttr->RW=       (attrib & (byte)0x1) && 0x1;
00075         FileAttr->Hidden=   (attrib & (byte)0x2) && 0x2;
00076         FileAttr->System=   (attrib & (byte)0x4) && 0x4;
00077         FileAttr->Label=    (attrib & (byte)0x8) && 0x8;
00078         FileAttr->Directory=(attrib & (byte)0x10)&& 0x10;
00079         FileAttr->Archived= (attrib & (byte)0x20)&& 0x20;
00080         FileAttr->Reserved= attrib>>6;
00081 }
00082
00083 char *show_attrib(attrib_t *FileAttr, char *stringa)
00084 {
00085         if (FileAttr->Label)
00086         {
00087                 stringa="LABEL   ";
00088                 return stringa;
00089         }
00090         if (FileAttr->Directory)
00091         {
00092                 stringa="DIR     ";
00093                 return stringa;
00094         }
00095         stringa[0]='r';
00096         if (FileAttr->RW) stringa[1]='-'; //ReadOnly
00097         else stringa[1]='+';//Read and Write
00098         stringa[2]='h';
00099         if (FileAttr->Hidden) stringa[3]='+'; //Hidden
00100         else stringa[3]='-';//Visible
00101         stringa[4]='s';
00102         if (FileAttr->System) stringa[5]='+'; //System
00103         else stringa[5]='-';//NonSystem
00104         stringa[6]='a';
00105         if (FileAttr->Archived) stringa[7]='+'; //Archived
00106         else stringa[7]='-';//NotArchived
00107         stringa[8]='\0';
00108
00109         return stringa;
00110 }
00111
00112 char *file_name(FileEntry_t *temp, char *stringa)
00113 {
00114         int k, a;
00115
00116         a=0;
00117         for(k=0; k<8; k++)
00118         {
00119                 if ((temp->Name[k]!=0)&&(temp->Name[k]!=' '))
00120                 {
00121                         stringa[k]=temp->Name[k];
00122                         a=k;
00123                 }
00124         }
00125         for(k=0;k<3;k++)
00126         {
00127                 if (temp->Extension[k]!=' ')
00128                 {
00129                         if (k==0) stringa[++a]='.';
00130                         stringa[++a]=temp->Extension[k];
00131                 }
00132         }
00133         for(a++;a<12;a++) stringa[a]=' ';
00134         stringa[12]='\0';
00135
00136         return stringa;
00137 }
00138
00139 bool show_file_entry(FileEntry_t *TempFile)
00140 {
00141         date_t FileDate;
00142         fat_time_t FileTime;
00143         attrib_t FileAttr;
00144         char Attributi[8];
00145         char Nome[13];
00146
00147         read_attrib(&FileAttr, TempFile->Attribute);
00148         if (!FileAttr.Label)
00149         {
00150                 if ( (TempFile->Name[0]) && (TempFile->Name[0]!=0xE5) )
```

```
00151                 {
00152                         if (TempFile->Name[0]==0x05) TempFile->Name[0]=0xE5;
00153                         int_to_date(&FileDate, TempFile->Date);
00154                         int_to_time(&FileTime, TempFile->Time);
00155 
00156                         kprintf("%s", file_name(TempFile, Nome));
00157                         gotoxy(15, -1);
00158                         kprintf("%s", show_attrib(&FileAttr, Attributi));
00159                         gotoxy(25, -1);
00160                         kprintf("%d/%d/%u", FileDate.day, FileDate.month, (FileDate.year+1980));
00161                         gotoxy(36, -1);
00162                         kprintf("%d:%d:%d", FileTime.hour, FileTime.minute, FileTime.second);
00163                         gotoxy(46, -1);
00164                         kprintf("S:%#x", TempFile->StartCluster);
00165                         gotoxy(60, -1);
00166                         kprintf("D:%u", TempFile->FileLength);
00167                         kprintf("\n\r");
00168 
00169                         if (TempFile->Name[0]==0xE5) TempFile->Name[0]=0x05;
00170 
00171                         return TRUE;
00172                 }
00173         }
00174         return FALSE;
00175 }
00176 
00177 void name_ext(char *filename, char *name, char *ext)
00178 {
00179         int i;
00180         unsigned char *find;
00181 
00182         // Set the string to uppercase                                  //
00183         strtoupper(filename);
00184 
00185         // Get the file name                                            //
00186         for(i=0; i<8; i++)
00187         {
00188                 if ( (filename[i]=='.') || (filename[i]=='\0') )
00189                         break;
00190                 name[i] = filename[i];
00191         }
00192         for(; i<8; i++)
00193                 name[i] = ' ';
00194         name[8] = '\0';
00195 
00196         // Get the file extension                                       //
00197         find = strstr(filename, ".");
00198         if (find++)
00199         {
00200                 for(i=0; i<3; i++)
00201                 {
00202                         if (find[i]=='\0')
00203                                 break;
00204                         ext[i] = find[i];
00205                 }
00206                 for(; i<3; i++)
00207                         ext[i] = ' ';
00208         }
00209         else
00210         {
00211                 for(i=0; i<3; i++)
00212                         ext[i] = ' ';
00213         }
00214         ext[3] = '\0';
00215 }
00216 
00217 bool compare_name_ext(unsigned char *name1, unsigned char *name2, char *ext1, char *ext2)
00218 {
00219         bool different = TRUE;
00220         if ( (*name1 != 0) && (*name1 != 0xE5) )
00221         {
00222                 if (*name1 == 0x05) *name1=0xE5;
00223 
00224                 if ( (strncmp(name1, name2, 8) == 0) && (strncmp(ext1, ext2, 3) == 0) )
00225                         different = FALSE;
00226 
00227                 if (*name1 == 0xE5) *name1=0x05;
00228         }
00229         return(!different);
00230 }
```

```
00231
00232 // --- Read procedures -------------------------------------------------//
00233 void read_file(sector_t *Buf, int start, int c)
00234 {
00235         int i, j;
00236         word a;
00237         word dir_start = bootsector.Fats*bootsector.SectorsPerFat+1;
00238         word data_start = dir_start + bootsector.RootDirectoryEntries*32/bootsector.BytesPerSector-2;
00239
00240         if(!start)
00241         {
00242                 for(j=0; j<3; j++)
00243                         if (fdc_read(dir_start, (byte *)Buf, c)) break;
00244         }
00245         else
00246         {
00247                 // Read the start sector //
00248                 a=start;
00249                 for(j=0; j<3; j++)
00250                         if (fdc_read(data_start+a, (byte *)Buf, 1)) break;
00251
00252                 // Read the other sectors //
00253                 for(i=1; i<c; i++)
00254                 {
00255                         if (next_sector(&a, a))
00256                         {
00257                                 for(j=0; j<3; j++)
00258                                         if (fdc_read(data_start+a, (byte *)(Buf+i), 1)) break;
00259                         }
00260                         else
00261                                 // End of file //
00262                                 break;
00263                 }
00264         }
00265 }
00266
00267 bool load_file(char *stringa, byte *buffer)
00268 {
00269         unsigned char Nome[9], Ext[4];
00270         SectorDir_t *Buf;
00271         int counter, count;
00272         int h, i;
00273         bool found=FALSE;
00274
00275         // Get the file name and extension //
00276         name_ext(stringa, Nome, Ext);
00277
00278         // Get the list of file into the current directory //
00279         counter = how_many_cluster(directory);
00280
00281         Buf = kmalloc(FAT_SECTOR_SIZE*counter);
00282         memset(Buf, 0, FAT_SECTOR_SIZE*counter);
00283
00284         // Read the directory //
00285         read_file((sector_t *)Buf, directory, counter);
00286
00287         for(h=0; h<counter; h++)
00288         {
00289                 for(i=0; i<FAT_SECTOR_SIZE/sizeof(FileEntry_t); i++)
00290                 {
00291                         found = compare_name_ext(Buf[h].Entry[i].Name, Nome, Buf[h].Entry[i].Extension, Ext);
00292                         if (found)
00293                         {
00294                                 // The file is empty //
00295                                 if ( !(Buf[h].Entry[i].StartCluster) ) goto founded;
00296
00297                                 // Copy the file into the buffer //
00298                                 count = how_many_cluster(Buf[h].Entry[i].StartCluster);
00299                                 read_file((sector_t *)buffer, Buf[h].Entry[i].StartCluster, count);
00300                                 goto founded;
00301                         }
00302                 }
00303         }
00304 founded:
00305         kfree(Buf);
00306
00307         if (!found) return(FALSE);
00308         return(TRUE);
00309 }
00310
```

```
00311 int get_file_size(char *file_name)
00312 {
00313 // Get the size of a file //
00314         unsigned char name[9], ext[4];
00315         SectorDir_t *Buf;
00316         int counter, count=0, h, i;
00317         bool found=FALSE;
00318
00319         // Get the file name and extension //
00320         name_ext(file_name, name, ext);
00321
00322         // Get the list of file into the current directory //
00323         counter = how_many_cluster(directory);
00324         Buf = kmalloc(FAT_SECTOR_SIZE*counter);
00325         memset(Buf, 0, FAT_SECTOR_SIZE*counter);
00326         // Read the directory //
00327         read_file((sector_t *)Buf, directory, counter);
00328
00329         // Find the file //
00330         for(h=0; h<counter; h++)
00331         {
00332                 for(i=0; i<FAT_SECTOR_SIZE/sizeof(FileEntry_t); i++)
00333                 {
00334                         found = compare_name_ext(Buf[h].Entry[i].Name, name, Buf[h].Entry[i].Extension, ext);
00335                         if (found)
00336                         {
00337                                 // The file is empty //
00338                                 if ( !(Buf[h].Entry[i].StartCluster) ) goto founded;
00339
00340                                 // Get the size in clusters //
00341                                 count = how_many_cluster(Buf[h].Entry[i].StartCluster);
00342                                 goto founded;
00343                         }
00344                 }
00345         }
00346 founded:
00347         kfree(Buf);
00348         if (!found) return(-1);
00349         return(count*FAT_SECTOR_SIZE);
00350 };
00351
00352 void ls()
00353 {
00354         int counter=0;
00355         int h, i, scroll;
00356         SectorDir_t *Buf;
00357
00358         counter = how_many_cluster(directory);
00359         Buf = kmalloc(FAT_SECTOR_SIZE*counter);
00360         memset(Buf, 0, FAT_SECTOR_SIZE*counter);
00361
00362         read_file((sector_t *)Buf, directory, counter);
00363
00364         kprintf("\nList of files into the directory:\n\r\n\r");
00365         for(h=0, scroll=0; h<counter; h++)
00366         {
00367                 for(i=0; i<(bootsector.BytesPerSector/sizeof(FileEntry_t)); i++)
00368                 {
00369                         if (scroll==20)
00370                         {
00371                                 kprintf("\n\rPress a key to continue...\n\r");
00372                                 scroll=0;
00373                                 if (kgetchar() == CTRL_C)
00374                                 {
00375                                         kprintf("\n\r");
00376                                         kfree(Buf);
00377                                         return;
00378                                 }
00379                         }
00380                         if ( show_file_entry(&(Buf[h].Entry[i])) )
00381                                 scroll++;
00382                 }
00383         }
00384         kfree(Buf);
00385         kprintf("\n\r");
00386 }
00387
00388 bool cat(char *stringa)
00389 {
00390         unsigned char Nome[9], Ext[4];
```

```
00391         SectorDir_t *Buf, *Buf2;
00392         word sector;
00393         int counter;
00394         int h, i;
00395         bool a=FALSE;
00396 
00397         name_ext(stringa, Nome, Ext);
00398 
00399         counter = how_many_cluster(directory);
00400         Buf = kmalloc(FAT_SECTOR_SIZE*counter);
00401         memset(Buf, 0, FAT_SECTOR_SIZE*counter);
00402         read_file((sector_t *)Buf, directory, counter);
00403 
00404         for(h=0; h<counter; h++)
00405         {
00406                 for(i=0; i<FAT_SECTOR_SIZE/sizeof(FileEntry_t); i++)
00407                 {
00408                         a = compare_name_ext(Buf[h].Entry[i].Name, Nome, Buf[h].Entry[i].Extension, Ext);
00409                         if (a)
00410                         {
00411                                 // The file is empty //
00412                                 if ( !(Buf[h].Entry[i].StartCluster) ) goto founded;
00413 
00414                                 // Print the file to the standard output //
00415                                 sector = Buf[h].Entry[i].StartCluster;
00416                                 Buf2 = kmalloc(sizeof(sector_t));
00417                                 memset(Buf2, 0, sizeof(sector_t));
00418                                 for (;;)
00419                                 {
00420                                         read_file((sector_t *)Buf2, sector, 1);
00421                                         for (i=0; i<FAT_SECTOR_SIZE; i++)
00422                                         {
00423                                                 kputchar( ((byte *)Buf2)[i] );
00424                                         }
00425                                         if (!(next_sector(&sector, sector))) break;
00426                                 }
00427                                 kfree(Buf2);
00428                                 goto founded;
00429                         }
00430                 }
00431         }
00432 founded:
00433         kfree(Buf);
00434         if (!a) return(FALSE);
00435         return(TRUE);
00436 }
00437 
00438 void add_dir_path(char *new_path)
00439 {
00440         strcat(path, new_path);
00441         strcat(path, "/");
00442 }
00443 
00444 void up_dir_path(void)
00445 {
00446         int i, k;
00447         for(i=0; path[i]!='\0'; i++);
00448         for(k=(i-2); path[k]!='/'; k--);
00449         path[++k] = '\0';
00450 }
00451 
00452 char *pwd()
00453 {
00454 // Return the current path //
00455         return(path);
00456 }
00457 
00458 bool cd(char *new_path)
00459 {
00460         int counter=0;
00461         int l, h, i;
00462         bool change=FALSE;
00463         word a=0;
00464         attrib_t FileAttr;
00465         char dir_name[8];
00466         SectorDir_t *Buf;
00467 
00468         // Goto the root //
00469         if (new_path[0]=='/')
00470                 if (new_path[1]=='\0')
```

```
00471                 {
00472                         path[0]='\0';
00473                         directory=0;
00474                         return(TRUE);
00475                 }
00476
00477
00478         counter = how_many_cluster(directory);
00479         Buf = kmalloc(FAT_SECTOR_SIZE*counter);
00480         memset(Buf, 0, FAT_SECTOR_SIZE*counter);
00481         read_file((sector_t *)Buf, directory, counter);
00482
00483         for(h=0; h<counter; h++)
00484         {
00485                 for(i=0; i<FAT_SECTOR_SIZE/sizeof(FileEntry_t); i++)
00486                 {
00487                         read_attrib(&FileAttr, Buf[h].Entry[i].Attribute);
00488                         if (FileAttr.Directory)
00489                         {
00490                                 if ((Buf[h].Entry[i].Name[0]!=0)&&(Buf[h].Entry[i].Name[0]!=0xE5))
00491                                 {
00492                                         if (Buf[h].Entry[i].Name[0]==5) Buf[h].Entry[i].Name[0]=0xE5;
00493                                         l=0;
00494                                         do
00495                                         {
00496                                                 dir_name[l]=Buf[h].Entry[i].Name[l];
00497                                         } while(dir_name[l++]!=' ');
00498
00499                                         dir_name[--l]='\0';
00500
00501                                         if (strcmp(new_path, dir_name)==0)
00502                                         {
00503                                                 a = Buf[h].Entry[i].StartCluster;
00504                                                 change=TRUE;
00505                                                 goto founded;
00506                                         }
00507                                         if (Buf[h].Entry[i].Name[0]==0xE5) Buf[h].Entry[i].Name[0]=0x05;
00508                                 }
00509                         }
00510                 }
00511         }
00512 founded:
00513         kfree(Buf);
00514         if (change)
00515         {
00516                 directory = a;
00517                 if (!a)
00518                         path[0]='\0';
00519                 else
00520                 {
00521                         if(new_path[0] != '.') add_dir_path(new_path);
00522                         else
00523                         {
00524                                 if (new_path[1] == '.') up_dir_path();
00525                         }
00526                 }
00527                 return TRUE;
00528         }
00529         else
00530                 return FALSE;
00531 }
00532
00533 // --- Write procedures ------------------------------------------------//
00534 bool fat12_write()
00535 {
00536 // Write the logical FAT to the disk                                    //
00537         int i, j;
00538
00539         // Converts the FAT from the logical structure into the         //
00540         // physical structure                                           //
00541         for(i=0, j=0; j<3072; j+=2)
00542         {
00543                 fat.data[i++] = (byte)lfat.data[j];
00544                 fat.data[i++] = (byte)((lfat.data[j]>>8)&(0x0F))|((lfat.data[j+1]<<4)&(0xF0));
00545                 fat.data[i++] = (byte)(lfat.data[j+1]>>4);
00546         }
00547
00548         // Copy the FAT to the disk                                     //
00549         for(j=0; j<bootsector.Fats; j++)
00550         {
```

```
00551                 for(i=0; i<3; i++)
00552                         if ( fdc_write(1, (byte *)&fat, bootsector.SectorsPerFat) )
00553                                 break;
00554                 if (i == 3)
00555                         return(FALSE);
00556         }
00557
00558         // Write successful!                                            //
00559         return(TRUE);
00560 }
00561
00562 word find_sector(int n, word actual)
00563 {
00564 // Find the sector n from actual                                       //
00565         word c, temp;
00566
00567         temp = actual;
00568         for(c=0; c<n; c++) next_sector(&temp, temp);
00569         return(temp);
00570 }
00571
00572 void write_sector_dir(SectorDir_t *sector, int num)
00573 {
00574         int i, j;
00575         word dir_start = bootsector.Fats*bootsector.SectorsPerFat+1;
00576         word data_start = dir_start + bootsector.RootDirectoryEntries*32/bootsector.BytesPerSector-2;
00577
00578         if( !directory )
00579         {
00580                 for(j=0; j<3; j++)
00581                         if ( fdc_write(dir_start+num, (byte *)sector, 1) )
00582                                 break;
00583         }
00584         else
00585         {
00586                 i = find_sector(num, directory);
00587                 for(j=0; j<3; j++)
00588                         if ( fdc_write(data_start+i, (byte *)sector, 1) )
00589                                 break;
00590         }
00591 }
00592
00593 void delete_file(word cluster)
00594 {
00595 // Delete the file from the FAT structure                              //
00596         word next;
00597
00598         while ( next_sector(&next, cluster) )
00599         {
00600                 lfat.data[cluster] = 0;
00601                 cluster = next;
00602         }
00603         lfat.data[cluster] = 0;
00604 }
00605
00606 bool rm(char *filename)
00607 {
00608 // Remove a file from the disk                                         //
00609         unsigned char name[9], ext[4];
00610         SectorDir_t *Buf;
00611         int counter, h, i;
00612         bool a=FALSE;
00613
00614         if ( (strcmp(filename, ".")==0) || (strcmp(filename, "..")==0) )
00615                 return(FALSE);
00616
00617         name_ext(filename, name, ext);
00618
00619         counter = how_many_cluster(directory);
00620         Buf = kmalloc(FAT_SECTOR_SIZE*counter);
00621         memset(Buf, 0, FAT_SECTOR_SIZE*counter);
00622         read_file((sector_t *)Buf, directory, counter);
00623
00624         for(h=0; h<counter; h++)
00625         {
00626                 for(i=0; i<FAT_SECTOR_SIZE/sizeof(FileEntry_t); i++)
00627                 {
00628                         a = compare_name_ext(Buf[h].Entry[i].Name, name, Buf[h].Entry[i].Extension, ext);
00629                         if (a)
00630                         {
```

```
00631                                   // Delete the file //
00632                                   Buf[h].Entry[i].Name[0] = 0xE5;
00633                                   write_sector_dir(&Buf[h], h);
00634                                   delete_file(Buf[h].Entry[i].StartCluster);
00635                                   fat12_write();
00636                                   goto founded;
00637                   }
00638                   }
00639           }
00640 founded:
00641           kfree(Buf);
00642           if (!a) return(FALSE);
00643           return(TRUE);
00644 }
00645
00646 // --- Init procedures -----------------------------------------------//
00647 bool init_FAT()
00648 {
00649 // Initialize the file system //
00650           int i;
00651
00652           for(i=0; i<3; i++)
00653                   if (fdc_read(FAT_BOOT_SECTOR, (byte *)&bootsector, 1))
00654                           break;
00655           if (i != 3)
00656                   return(TRUE);
00657           else
00658                   return(FALSE);
00659 }
00660
00661 bool check_FAT(void)
00662 {
00663           if ( (bootsector.BytesPerSector!=FAT_SECTOR_SIZE) || (bootsector.SectorsPerCluster!=1) ||
00664           (bootsector.Fats!=2) || (bootsector.RootDirectoryEntries!=224) ||
00665           (bootsector.LogicalSectors!=2880) || (bootsector.MediumDescriptorByte!=0xF0) )
00666              {
00667                   kprintf("\n\rNot a valid FAT12 filesystem on the disk.\n\r");
00668                   return FALSE;
00669              }
00670           return TRUE;
00671 }
00672
00673 bool Read_FAT()
00674 {
00675 // Read the FAT from the floppy (mount) //
00676           int i, j;
00677
00678           if (!( init_FAT() ))
00679           {
00680                   kprintf("\n\rFloppy I/O error. Unable to read the block!!!\n\r");
00681                   return(FALSE);
00682           }
00683
00684           // FAT initializazion OK! //
00685           for (i=0; i<3; i++)
00686                   if (fdc_read(1, (byte *)&fat, bootsector.SectorsPerFat))
00687                           break;
00688           if (i == 3)
00689           {
00690                   kprintf("\n\rFloppy I/O error. Unable to read the block!!!\n\r");
00691                   return(FALSE);
00692           }
00693
00694           // Converts the FAT into the logical structures (array of word) //
00695           for(i=0, j=0; i<4608; i+=3)
00696           {
00697                   lfat.data[j++] = (fat.data[i] + (fat.data[i+1] << 8)) & 0x0FFF;
00698                   lfat.data[j++] = (fat.data[i+1] + (fat.data[i+2] << 8)) >> 4;
00699           }
00700
00701           // Check if the FAT is correct //
00702           if (!( check_FAT() ))
00703           {
00704                   kprintf("\n\rNot a valid FAT12 filesystem!!!\n\r");
00705                   return(FALSE);
00706           }
00707
00708           // Initialize the path //
00709           path[0]='\0';
00710
```

```
00711         #ifdef FAT_DEBUG
00712         kprintf("\n\rInitializing FileSystem...\n\r");
00713         kprintf("\n\r");
00714         kprintf("Jump:\t\t\t%02x %02x %02x\n\r", bootsector.Jump[0], bootsector.Jump[1], bootsector.Jump[2]);
00715         kprintf("OS Name:\t\t\t%s\n\r", bootsector.Name);
00716         kprintf("BytesPerSector:\t\t\t%u\n\r", bootsector.BytesPerSector);
00717         kprintf("SectorsPerCluster:\t\t\t%u\n\r", bootsector.SectorsPerCluster);
00718         kprintf("ReservedSectors:\t\t\t%u\n\r", bootsector.ReservedSectors);
00719         kprintf("Fats:\t\t\t%u\n\r", bootsector.Fats);
00720         kprintf("RootDirectoryEntries:\t\t\t%u\n\r", bootsector.RootDirectoryEntries);
00721         kprintf("LogicalSectors:\t\t\t%u\n\r", bootsector.LogicalSectors);
00722         kprintf("MediumDescriptorByte:\t\t\t%X\n\r", bootsector.MediumDescriptorByte);
00723         kprintf("SectorsPerFat:\t\t\t%u\n\r", bootsector.SectorsPerFat);
00724         kprintf("SectorsPerTrack:\t\t\t%u\n\r", bootsector.SectorsPerTrack);
00725         kprintf("Heads:\t\t\t%u\n\r", bootsector.Heads);
00726         kprintf("HiddenSectors:\t\t\t%u\n\r", bootsector.HiddenSectors);
00727         kprintf("\n\r");
00728         #endif
00729 
00730         return(TRUE);
00731 }
```
