Appendix: Source Code

Please note that the source code, as an organized project, is available at http://www.eng.utah.edu/~tew/pen/

(boot.s)

```
; PEN (Personal Electronic Notebook)
; CS4710 Fall 2006
; Neal Tew
;-----------------------------------

; bootstrap code..

  IMPORT |Load$$SDRAM$$Base|
  IMPORT |Image$$SDRAM$$Base|
  IMPORT |Image$$SDRAM$$Limit|
  IMPORT |Image$$SDRAM$$ZI$$Base|
  IMPORT |Image$$SDRAM$$ZI$$Limit|

  IMPORT __use_no_semihosting_swi ;this makes the linker throw an error if C libs try to use host
SWI's
  IMPORT threadmanager_init
  IMPORT _init_alloc

  EXPORT codestart
  EXPORT codeend
  EXPORT __rt_heap_extend

  AREA |.text|, CODE
  ENTRY
;-------------------------------
program_starts_here
  ands r0,pc,#0x80000000   ;we are running in ram already?
  bne ram_boot

rom_boot ;use GPIOs to decide how to boot (run PEN or u-Boot?)

  b uBoot_skip

  ldr r3,=0x40e00000
  mov r0,#0
  str r0,[r3,#0x60]   ;GAFR1_U - no alt function
  str r0,[r3,#0x10]   ;GPDR1 - make GPIOs inputs
  ldr r0,[r3,#0x04]   ;GPLR1
  ands r0,r0,#0x000c0000
  cmpne r0,#0x000c0000

  moveq pc,#0x850     ;GPIO50=51 (default state), go to u-boot

uBoot_skip

  ;change system clock right away, so we don't need to worry about SDRAM refresh

  ldr r3,=0x41300000
  ldr r0,=0x0161
  str r0,[r3]  ;CCCR=010_11_00001: memfreq=100, runfreq=mem*4, turbo=run*1
  mov r0,#2
  mcr p14,0,r0,c6,c0,0;CCLKCFG=FCS (start freq change sequence)

  ;wait 200us for SDCLK to stabilize

  ;setup SDRAM stuff

  ;copy all code to ram

  adr r0,program_starts_here ;src
  ldr r1,=|Image$$SDRAM$$Base| ;dst
  ldr r2,=|Image$$SDRAM$$Limit|;dst_end
  mov r3,#4       ;srcinc
  mov lr,r1       ;copy self to SDRAM and jump to ram
  b copy2

ram_boot
  msr cpsr_c,#0xdf  ;disable interrupts

  bl initmem    ;setup MMU, etc

  adr r5,copylist
  bl copymem    ;clear ZINIT area

  ldr r0,=0x40D00004  ;(ICMR)
  mov r1,#0
  str r1,[r0]   ;disable all PXA IRQs

  mov r0,#0xa4000000  ;r0=SDRAM top
  msr cpsr_c,#0x17
  sub sp,r0,#0x10000  ;abort stack
  msr cpsr_c,#0x1b
  sub sp,r0,#0x10000  ;undefined stack
  msr cpsr_c,#0x12
  sub sp,r0,#0x20000  ;IRQ stack
  msr cpsr_c,#0x13
  sub sp,r0,#0x30000  ;supervisor (swi) stack
  msr cpsr_c,#0x1f
  sub sp,r0,#0x40000  ;system mode stack

  ldr r0,=0xA0200000  ;heap start at 2MB
  ldr r1,=0xA0800000  ;heap end
  bl _init_alloc    ;initialize heap

  ldr r12,=threadmanager_init
  blx r12       ;start up thread manager and launch PENmain as the main thread
;-------------------------------
copymem  ldmia r5!,{r0-r3}
copy2  cmp r1,r2
  ldrcc r4,[r0],r3
  strcc r4,[r1],#4
  bcc copy2
  bx lr
copylist
  dcd zero, |Image$$SDRAM$$ZI$$Base|, |Image$$SDRAM$$ZI$$Limit|
zero dcd 0

codestart dcd |Image$$SDRAM$$Base|
codeend dcd |Image$$SDRAM$$Limit|
;-------------------------------
initmem         ;initialize MMU, cache, etc
  mov r12,lr

  mcr p15,0,r0,c9,c1,1;unlock instruction cache
  mcr p15,0,r0,c9,c2,1;unlock data cache
  mcr p15,0,r0,c7,c7  ;invalidate BTB, I+D caches
  mrc p15,0,r0,c2,c0
  mov r0,r0     ;wait for it
  sub pc,pc,#4

  mov r0,#0x78
  mcr p15,0,r0,c1,c0,0;disable MMU
  mov r0,#0
  mcr p15,0,r0,c1,c0,1;aux control
  ldr r2,=0xa3ffc000  ;set TLB base to top 16K of ram
  mcr p15,0,r2,c2,c0
  ldr r3,=0x55555555  ;domain control - use TLB permission flags for everything
  mcr p15,0,r3,c3,c0
  mcr p15,0,r0,c8,c7  ;invalidate TLB
        ;prepare descriptor table:
  ldr r0,=0x00000c02  ;00000000-01000000: flash
  ldr r1,=0x01000c02
  bl fillTLB
  ldr r0,=0x01000002  ;01000000-40000000: empty
  ldr r1,=0x40000002
  bl fillTLB
  ldr r0,=0x40000c02  ;40000000-4c000000: registers
  ldr r1,=0x4c000c02
  bl fillTLB
  ldr r0,=0x4c000002  ;4c000000-a0000000: empty
  ldr r1,=0xa0000002
  bl fillTLB
  ldr r0,=0xa0000c0e  ;a0000000-a0800000: SDRAM (cached) code,data,heap
  ldr r1,=0xa0800c0e
  bl fillTLB
  ldr r0,=0xa0800c02  ;a0800000-a3f00000: (uncached) DMA-safe mem
  ldr r1,=0xa3f00c02
  bl fillTLB
  ldr r0,=0xa3f00c0e  ;a3f00000-a4000000: SDRAM (cached) stack, vectors, etc
  ldr r1,=0xa4000c0e
  bl fillTLB
  ldr r0,=0xa4000002  ;a4000000-fff00000: empty
  ldr r1,=0xfff00002
  bl fillTLB
  ldr r0,=0xa3f00c0e  ;fff00000-00000000: top of ram (cached)
  ldr r1,=0xa4000c0e
  bl fillTLB

  bl install_interrupt_handlers;take care of this while MMU is still offline

  mcr p15,0,r0,c7,c10 ;drain write buffer
  ;ldr r0,=2_0000100001111011  ;enable MMU, alignment faults, BTB
  ldr r0,=2_0011100001111111 ;enable MMU, alignment faults, BTB, I+D caches, FFFF0000 vector
  mcr p15,0,r0,c1,c0,0

  bx r12
fillTLB        ;fill descriptor table from r0 to r1
  str r0,[r2],#4
  add r0,r0,#0x00100000 ;1MB increments
  cmp r0,r1
  bne fillTLB
  bx lr
;-----------------------
__rt_heap_extend  ;return error if malloc wants to grow the heap
  mov r0,#0
  bx lr
;-----------------------
install_interrupt_handlers ;move vector table to "0xFFFF0000"

  adr r0,FFFF_vectors
  adr r1,FFFF_end
  ldr r2,=0xa3ff0000
iih0 ldr r3,[r0],#4
  str r3,[r2],#4
  cmp r0,r1
  bne iih0
  bx lr

  LTORG
  IMPORT irq_handler
  IMPORT swi_handler
  IMPORT undefined_handler
  IMPORT data_abort
  IMPORT prefetch_handler
FFFF_vectors
  ldr pc,=0    ;RESET
  ldr pc,=undefined_handler
  ldr pc,=swi_handler
  ldr pc,=prefetch_handler
  ldr pc,=data_abort
  ldr pc,=0
  ldr pc,=irq_handler
  ldr pc,=0    ;FIQ
  LTORG
```

```
FFFF_end
;-----------------------
  END
```

(console.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

// tty console

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "pen.h"
#include "fat.h"

#define BUFFSIZE 64

static int idx=0;
static char buff[BUFFSIZE];

void console_thread(void);
void testpattern(int n);

void console_init() {
  newthread((void*)console_thread,0);
}

//apparently armlib's atoi() is broken, so we write our own
int _atoi(char *s) {
  int c,i,hex=0;
  i=0;
  while(1) {
    c=*s++;
    if(c>='a') c=c-'a'+'A';
    if(hex) {
      if(c>='A' && c<='F')
        c=c-'A'+10;
      else if(c>='0' && c<='9')
        c=c-'0';
      else break;
      i=(i<<4)+c;
    } else {
      if(c=='X') {
        hex=1;
        continue;
      }
      if(c<'0' || c>'9')
        break;
      i=i*10+(c-'0');
    }
  }
  return i;
}

extern u32 codestart, codeend;
void flashOS() {
  int i;
  u8 *src, *dst;

  src=(u8*)codestart;
  dst=(u8*)0x40000;

  printf("Writing %i bytes ",codeend-codestart);
  for(i=0;i<codeend-codestart;i+=0x20000) {
    flash_writeblock(src,dst);
    src+=0x20000;
    dst+=0x20000;
  }
  printf("\n");
}

char *fstypes[]={"UNKNOWN","FAT12","FAT16","FAT32"};
void dir() {
  char filename[256];
  u32 filesize;
  FILE_TYPE f;
  u32 dirtotal=0;
  u32 fstotal;

  f=FAT_FindFirstFileLFN(filename);
  while(f!=FT_NONE) {
    filesize=FAT_GetFileSize();
    dirtotal+=filesize;
    printf("%c%s\t\t%i\n",(f==FT_DIR)?'/':' ',filename,filesize);
    f=FAT_FindNextFileLFN(filename);
  };
  fstotal=FAT_GetFileSystemTotalSize();
  printf("%s volume, %i / %i
(%i%%)\n",fstypes[FAT_GetFileSystemType()],dirtotal,fstotal,100*dirtotal/fstotal);
}

void threadtest(int i) {
  msleep(5*1000);
  suspend();
  while(1) {
    wait(SIG_ANY);
  }
}

//process console input.  call me periodically...
void console_thread() {
  peninfo p;
  int c,i;
  int h,m,s;

  while(1) {
    wait(SIG_UART);
    c=ff_uart_getc();
    if(c<0)      //no input
      continue;
    if(c==8) {   //backspace
      if(idx>0) {
        idx--;
        printf("\x08 \x08");
      }
    } else if(c!='\r') {
      if(idx<BUFFSIZE-1) {
        buff[idx++]=c;
        ff_uart_putc(c);
      }
    } else {
      buff[idx]=0;
      printf("\n");
      i=_atoi(buff+2);
      switch(buff[0]) {
        //case 0:  //CR
        //break;
        case 'G':
          gif_read(&buff[2],frame);
          break;
        case 'S':
          gif_save(&buff[2], frame, palette);
          break;
        case 'i':
          printf("\nBuilt " __DATE__ " " __TIME__"\n");
          //printf("memory free:
          i=RTC-1000*600;
          s=i/1000%60;
          m=i/(1000*60)%60;
          h=i/(1000*60*60);
          printf("Time since reboot: %02i:%02i:%02i.%03i\n",h,m,s,i%1000);
          flashinfo();
          pageinfo();
          break;
        case 'l':
          dir();
          break;
        case 'a':
          settings.antialias=!settings.antialias;
          printf("Antialiasing is %s.\n",settings.antialias? "ON":"OFF");
          save_settings();
          break;
  /*    case 'B':
          set_background(i);
          redraw_background();
          break;*/
        case 't':
          testpattern(i);
          break;
        case 'c':
          wacom_calibrate();
          break;
        case 'b':
          backlight(i);
          break;
        case 'P':
          printf("Push any key to stop.\n");
          do {
            wait(SIG_PEN|SIG_UART);
            if(wacom_read(&p))
              printf("(%i,%i) %02X\n",p.x,p.y,p.flags);
          } while(ff_uart_getc()<0);
          break;
        case 'r':
          clean_all_pages();
          flashall();
          reboot(i);
          break;
        case 'R':
          resume(i);
          break;
        case 's':
          for(i=0;i<MAXTHREADS;i++) {
            if(threadstate[i].regs[0]) //thread is alive?
              printf("%x: %x %x time=%x
id=%i\n",(u32)&threadstate[i],threadstate[i].prev,threadstate[i].next,threadstate[i].ticks,thread
state[i].id);
            else
              printf("%x: --\n",(u32)&threadstate[i]);
          }
          break;
        case 'T':
          i=newthread((void*)threadtest,(void*)i);
          printf("threadID=%i\n",i);
          break;
        case 'k':
          killthread(i);
          break;
        case 'O':
          flashOS();
          break;
        case 'F':
          flash_format();
          if(!FAT_InitFiles())
            printf("FAT_InitFiles() failed.\n");
          break;
        default:
          printf(
            "O         write OS to flash\n"
            "G file    read GIF file to screen\n"
            "S file    save GIF file\n"
            "F         format FAT image\n"
            "l         directory listing\n"
            "s         thread stats\n"
            "T         spawn new test thread\n"
            "R         resume thread\n"
            "k NNN     kill thread\n"
            "a         antialias toggle\n"
            "t NNN     test pattern (0..?)\n"
            "b NNN     set backlight (0..255)\n"
            //"B NNN     set background\n"
            "c         calibrate pen\n"
            "P         pen test\n"
            "i         miscellaneous info\n"
            "r NNN     reboot (jump to addr)\n" //u-boot reset is 0x850
          );
      }
      printf(">");
      idx=0;
    }
  }
}

void testpattern(int n) {
  int i,x,y;
  u8 *p;

  memset(frame,WHITE,FRAMESIZE);
  switch(n) {
    case 0:
      for(i=0;i<32;i++) {
        palette[i]=i;
        palette[i+32]=i<<6;
        palette[i+64]=i<<11;
      }
      p=frame;
      for(y=0;y<32*3;y++) {
        for(x=0;x<8*1024;x++) {
          *p++=y;
        }
      }
      break;
    case 1:
      for(y=0;y<1024;y++) {
        point(y*768/1024,y);
        point(767-(y*768/1024),y);
      }
      for(y=32;y<1024;y+=32)
        for(x=0;x<768;x++)
          point(x,y);
      for(x=24;x<768;x+=24)
        for(y=0;y<1024;y++)
          point(x,y);
      for(x=0;x<768;x++) {
        if(x&16)
          point(x,0);
        else
          point(x,1023);
      }
      for(y=0;y<1024;y++) {
        if(y&16)
          point(0,y);
        else
          point(767,y);
      }
      break;
    case 2:
      for(i=0;i<16;i++) {
        palette[i]=1<<i;
      }
      p=frame;
      for(y=0;y<16;y++) {
        for(x=0;x<48*1024;x++) {
          *p++=y;
        }
      }
      break;
    case 3:
      palette[WHITE]=0xffff;
      break;
    case 4:
      palette[WHITE]=0;
      break;
    case 5:
      for(x=0;x<768;x++) {
        for(y=0;y<1024;y+=2) {
          point(x,y);
        }
      }
      break;
    case 6:
      i=0;
      for(x=0;x<768;x++) {
        i=!i;
        for(y=0;y<1024;y++) {
          i=!i;
          if(i)
            point(x,y);
        }
      }
      break;
  }
}
```

(disc_io.c)

```
/*

  disc_io.c

  uniformed io-interface to work with Chishm's FAT library

  Written by MightyMax

  Modified by Chishm:
  2005-11-06
    * Added WAIT_CR modifications for NDS

  Modified by www.neoflash.com:
  2006-02-03
    * Added SUPPORT_* defines, comment out any of the SUPPORT_* defines in disc_io.h to remove
support
      for the given interface and stop code being linked to the binary

      * Added support for MK2 MMC interface

    * Added disc_Cache* functions

  Modified by Chishm:
  2006-02-05
    * Added Supercard SD support
*/

#include "disc_io.h"

//#ifdef NDS
//#include <nds.h>
//#endif


// Include known io-interfaces:
```

```
#ifdef SUPPORT_GSTIX
 #include "io_gstix.h"
#endif

#ifdef SUPPORT_MPCF
 #include "io_mpcf.h"
#endif

#ifdef SUPPORT_M3CF
 #include "io_m3cf.h"
#endif

#ifdef SUPPORT_SCCF
 #include "io_sccf.h"
#endif

#ifdef SUPPORT_SCSD
 #include "io_scsd.h"
#endif

#ifdef SUPPORT_FCSR
 #include "io_fcsr.h"
#endif

#ifdef SUPPORT_NMMC
 #include "io_nmmc.h"
#endif

// Keep a pointer to the active interface
LPIO_INTERFACE active_interface = 0;


/*

  Disc Cache functions
  2006-02-03:
    Added by www.neoflash.com

*/

#ifdef DISC_CACHE

#include <string.h>

#define CACHE_FREE 0xFFFFFFFF

static u8 cacheBuffer[ DISC_CACHE_COUNT * 512 ];

static struct {
  u32 sector;
  u32 dirty;
  u32 count;
} cache[ DISC_CACHE_COUNT ];

static u32 disc_CacheFind(u32 sector) {
  u32 i;

  for( i = 0; i < DISC_CACHE_COUNT; i++ )  {
    if( cache[ i ].sector == sector )
      return i;
  }

  return CACHE_FREE;
}

static u32 disc_CacheFindFree(void) {

  u32 i = 0, j;
  u32 count = -1;

  for( j = 0; j < DISC_CACHE_COUNT; j++ )  {
    if( cache[ j ].sector == CACHE_FREE ) {
      i = j;
      break;
    }

    if( cache[ j ].count < count ) {
      count = cache[ j ].count;
      i = j;
    }
  }

  if( cache[ i ].sector != CACHE_FREE && cache[i].dirty != 0 ) {

    active_interface->fn_WriteSectors( cache[ i ].sector, 1, &cacheBuffer[ i * 512 ] );
    /* todo: handle write error here */

    cache[ i ].sector = CACHE_FREE;
    cache[ i ].dirty = 0;
    cache[ i ].count = 0;
  }

  return i;
}

void disc_CacheInit(void)  {

  u32 i;

  for( i = 0; i < DISC_CACHE_COUNT; i++ )  {
    cache[ i ].sector = CACHE_FREE;
    cache[ i ].dirty = 0;
    cache[ i ].count = 0;
  }

}

bool disc_CacheFlush(void) {

  u32 i;

  if( !active_interface )  return false;

  for( i = 0; i < DISC_CACHE_COUNT; i++ )  {
    if( cache[ i ].sector != CACHE_FREE && cache[ i ].dirty != 0 ) {
      if( active_interface->fn_WriteSectors( cache[ i ].sector, 1, &cacheBuffer[ i * 512 ] ) ==
false )
        return false;

      cache[ i ].dirty = 0;
    }
  }
  return true;
}

bool disc_CacheReadSector( void *buffer, u32 sector) {
  u32 i = disc_CacheFind( sector );
  if( i == CACHE_FREE ) {
    i = disc_CacheFindFree();
    cache[ i ].sector = sector;
    if( active_interface->fn_ReadSectors( sector, 1, &cacheBuffer[ i * 512 ] ) == false )
      return false;
  }
  memcpy( buffer, &cacheBuffer[ i * 512 ], 512 );
  cache[ i ].count++;
  return true;
}

bool disc_CacheWriteSector( void *buffer, u32 sector ) {
  u32 i = disc_CacheFind( sector );
  if( i == CACHE_FREE ) {
    i = disc_CacheFindFree();
    cache [ i ].sector = sector;
  }
  memcpy( &cacheBuffer[ i * 512 ], buffer, 512 );
  cache[ i ].dirty=1;
  cache[ i ].count++;
  return true;
}

#endif

/*

  Hardware level disc funtions

*/

bool disc_setGbaSlotInterface (void)
{
  // If running on an NDS, make sure the correct CPU can access
  // the GBA cart. First implemented by SaTa.

#ifdef SUPPORT_M3CF
  // check if we have a M3 perfect CF plugged in
  active_interface = M3CF_GetInterface() ;
  if (active_interface->fn_StartUp())
  {
    // set M3 CF as default IO
    return true ;
  } ;
#endif

#ifdef SUPPORT_MPCF
  // check if we have a GBA Movie Player plugged in
  active_interface = MPCF_GetInterface() ;
  if (active_interface->fn_StartUp())
  {
    // set GBAMP as default IO
    return true ;
  } ;
#endif

#ifdef SUPPORT_SCCF
  // check if we have a SuperCard CF plugged in
  active_interface = SCCF_GetInterface() ;
  if (active_interface->fn_StartUp())
  {
    // set SC CF as default IO
    return true ;
  } ;
#endif

#ifdef SUPPORT_SCSD
  // check if we have a SuperCard SD plugged in
  active_interface = SCSD_GetInterface() ;
  if (active_interface->fn_StartUp())
  {
    // set SC SD as default IO
    return true ;
  } ;
#endif

#ifdef SUPPORT_FCSR
  // check if we have a GBA Flash Cart plugged in
  active_interface = FCSR_GetInterface() ;
  if (active_interface->fn_StartUp())
  {
    // set FC as default IO
    return true ;
  } ;
#endif
```

```
#ifdef SUPPORT_GSTIX
  // gumstix flash
  active_interface = GSTIX_GetInterface() ;
  if (active_interface->fn_StartUp())
  {
    // set GSTIX as default IO
    return true ;
  } ;
#endif

  return false;
}

bool disc_Init(void)
{
#ifdef DISC_CACHE
  disc_CacheInit();
#endif


  //if (active_interface != 0) {
  //return true;
  //} //force interface re-init (to allow for interface swapping, formatting, etc)

  if (disc_setGbaSlotInterface()) {
    return true;
  }

  // could not find a working IO Interface
  active_interface = 0 ;
  return false ;
}

bool disc_IsInserted(void)
{
  if (active_interface) return active_interface->fn_IsInserted() ;
  return false ;
}

bool disc_ReadSectors(u32 sector, u8 numSecs, void* buffer)
{
#ifdef DISC_CACHE
  u8 *p=(u8*)buffer;
  u32 i;
  u32 inumSecs=numSecs;
  if(numSecs==0)
    inumSecs=256;
  for( i = 0; i<inumSecs; i++) {
    if( disc_CacheReadSector( &p[i*512], sector + i ) == false )
      return false;
  }
  return true;
#else
  if (active_interface) return active_interface->fn_ReadSectors(sector,numSecs,buffer) ;
  return false ;
#endif
}

bool disc_WriteSectors(u32 sector, u8 numSecs, void* buffer)
{
#ifdef DISC_CACHE
  u8 *p=(u8*)buffer;
  u32 i;
  u32 inumSecs=numSecs;
  if(numSecs==0)
    inumSecs=256;
  for( i = 0; i<inumSecs; i++) {
    if( disc_CacheWriteSector( &p[i*512], sector + i ) == false )
      return false;
  }
  return true;
#else
  if (active_interface) return active_interface->fn_WriteSectors(sector,numSecs,buffer) ;
  return false ;
#endif
}

bool disc_ClearStatus(void)
{
  if (active_interface) return active_interface->fn_ClearStatus() ;
  return false ;
}

bool disc_Shutdown(void)
{
#ifdef DISC_CACHE
  disc_CacheFlush();
#endif
  if (active_interface) active_interface->fn_Shutdown() ;
  active_interface = 0 ;
  return true ;
}

u32  disc_HostType (void)
{
  if (active_interface) {
    return active_interface->ul_ioType;
  } else {
    return 0;
  }
}
```

(disc_io.h)

```
#ifndef DISC_IO_H
#define DISC_IO_H

//----------------------------------------------------------------------
// Customisable features

// Use DMA to read the card, remove this line to use normal reads/writes
// #define _CF_USE_DMA

// Allow buffers not alligned to 16 bits when reading files.
// Note that this will slow down access speed, so only use if you have to.
// It is also incompatible with DMA
#define _CF_ALLOW_UNALIGNED

// Device support options, added by www.neoflash.com

//#define SUPPORT_NMMC   // comment out this line to remove Neoflash MK2 MMC Card support
//#define SUPPORT_MPCF   // comment out this line to remove GBA Movie Player support
//#define SUPPORT_M3CF   // comment out this line to remove M3 Perfect CF support
//#define SUPPORT_SCCF   // comment out this line to remove Supercard CF support
//#define SUPPORT_SCSD   // comment out this line to remove Supercard SD support
//#define SUPPORT_FCSR   // comment out this line to remove GBA Flash Cart support
#define SUPPORT_GSTIX
//----------------------------------------------------------------------

#if defined _CF_USE_DMA && defined _CF_ALLOW_UNALIGNED
 #error You cant use both DMA and unaligned memory
#endif

// When compiling for NDS, make sure NDS is defined
#ifndef NDS
 #define ARM9
 #define NDS
#endif

#include <asm/types.h>
/*
```

```
#ifdef NDS
 #include <nds/jtypes.h>
#else
 #include "gba_types.h"
#endif
*/

// Disable NDS specific hardware and features if running on a GBA
#ifndef NDS
 #undef SUPPORT_NMMC
 #undef DISC_CACHE
#endif

/*

  Interface for host program

*/

#define BYTE_PER_READ 512

/*-----------------------------------------------------------------
disc_Init
Detects the inserted hardware and initialises it if necessary
bool return OUT:  true if a suitable device was found
-----------------------------------------------------------------*/
extern bool disc_Init(void) ;

/*-----------------------------------------------------------------
disc_IsInserted
Is a usable disc inserted?
bool return OUT:  true if a disc is inserted
-----------------------------------------------------------------*/
extern bool disc_IsInserted(void) ;

/*-----------------------------------------------------------------
disc_ReadSectors
Read 512 byte sector numbered "sector" into "buffer"
u32 sector IN: address of first 512 byte sector on disc to read
u8 numSecs IN: number of 512 byte sectors to read,
 1 to 256 sectors can be read, 0 = 256
void* buffer OUT: pointer to 512 byte buffer to store data in
bool return OUT: true if successful
-----------------------------------------------------------------*/
extern bool disc_ReadSectors(u32 sector, u8 numSecs, void* buffer) ;
#define disc_ReadSector(sector,buffer)disc_ReadSectors(sector,1,buffer)

/*-----------------------------------------------------------------
disc_WriteSectors
Write 512 byte sector numbered "sector" from "buffer"
u32 sector IN: address of 512 byte sector on disc to write
u8 numSecs IN: number of 512 byte sectors to write  ,
 1 to 256 sectors can be read, 0 = 256
void* buffer IN: pointer to 512 byte buffer to read data from
bool return OUT: true if successful
-----------------------------------------------------------------*/
extern bool disc_WriteSectors(u32 sector, u8 numSecs, void* buffer) ;
#define disc_WriteSector(sector,buffer) disc_WriteSectors(sector,1,buffer)

/*-----------------------------------------------------------------
disc_ClearStatus
Tries to make the disc go back to idle mode
bool return OUT:  true if the disc is idle
-----------------------------------------------------------------*/
extern bool disc_ClearStatus(void) ;

/*-----------------------------------------------------------------
disc_Shutdown
unload the disc interface
bool return OUT: true if successful
-----------------------------------------------------------------*/
```

```
extern bool disc_Shutdown(void) ;

/*-----------------------------------------------------------------
disc_HostType
Returns a unique u32 number identifying the host type
u32 return OUT: 0 if no host initialised, else the identifier of
  the host
-----------------------------------------------------------------*/
extern u32 disc_HostType(void);

/*-----------------------------------------------------------------
disc_CacheFlush
Flushes any cache writes to disc
bool return OUT: true if successful, false if an error occurs
Added by www.neoflash.com
-----------------------------------------------------------------*/
#ifdef DISC_CACHE
extern bool disc_CacheFlush(void);
#else
__inline bool disc_CacheFlush(void) {
  return true;
}
#endif // DISC_CACHE


/*

  Interface for IO libs

*/

#define FEATURE_MEDIUM_CANREAD    0x00000001
#define FEATURE_MEDIUM_CANWRITE   0x00000002
#define FEATURE_SLOT_GBA     0x00000010
#define FEATURE_SLOT_NDS     0x00000020

typedef bool (* FN_MEDIUM_STARTUP)(void) ;
typedef bool (* FN_MEDIUM_ISINSERTED)(void) ;
typedef bool (* FN_MEDIUM_READSECTORS)(u32 sector, u8 numSecs, void* buffer) ;
typedef bool (* FN_MEDIUM_WRITESECTORS)(u32 sector, u8 numSecs, void* buffer) ;
typedef bool (* FN_MEDIUM_CLEARSTATUS)(void) ;
typedef bool (* FN_MEDIUM_SHUTDOWN)(void) ;


typedef struct {
  unsigned long     ul_ioType ;
  unsigned long     ul_Features ;
  FN_MEDIUM_STARTUP   fn_StartUp ;
  FN_MEDIUM_ISINSERTED fn_IsInserted ;
  FN_MEDIUM_READSECTORS  fn_ReadSectors ;
  FN_MEDIUM_WRITESECTORS fn_WriteSectors ;
  FN_MEDIUM_CLEARSTATUS  fn_ClearStatus ;
  FN_MEDIUM_SHUTDOWN     fn_Shutdown ;
} IO_INTERFACE, *LPIO_INTERFACE ;

extern LPIO_INTERFACE active_interface;

#endif // define DISC_IO_H
```

(fat.c)

```
/*
  gba_nds_fat.c
  By chishm (Michael Chisholm)

  Routines for reading a compact flash card
  using the GBA Movie Player or M3.

  Some FAT routines are based on those in fat.c, which
  is part of avrlib by Pascal Stang.
  This software is completely free. No warranty is provided.
  If you use it, please give me credit and email me about your
  project at chishm@hotmail.com

  See gba_nds_fat.txt for help and license details.
*/
//-----------------------------------------------------------------
// Includes

#include <stdio.h>
#include "fat.h"
#include "disc_io.h"
#include <string.h>
#include "pen.h"

//#ifdef NDS
// #include <nds/ipc.h>  // Time on the NDS
//#endif
//------------------------------------------------------------------
// Data  types
#ifndef  NULL
 #define NULL 0
#endif

//------------------------------------------------------------------
// NDS memory access control register
#ifdef NDS
 #ifndef WAIT_CR
  #define WAIT_CR (*(vu16*)0x04000204)
 #endif
#endif

//-----------------------------------------------------------------
// Appropriate placement of CF functions and data
#ifdef NDS
 #define _VARS_IN_RAM
#else
 #define _VARS_IN_RAM __attribute__ ((section (".sbss")))
#endif


//-------------------------------------------------------------------
// FAT constants
#define CLUSTER_EOF_16 0xFFFF
#define  CLUSTER_EOF 0x0FFFFFFF
#define CLUSTER_FREE 0x0000
#define CLUSTER_FIRST  0x0002

#define FILE_LAST 0x00
#define FILE_FREE 0xE5

#define ATTRIB_ARCH 0x20
#define ATTRIB_DIR  0x10
#define ATTRIB_LFN  0x0F
#define ATTRIB_VOL  0x08
#define ATTRIB_HID  0x02
#define ATTRIB_SYS  0x04
#define ATTRIB_RO 0x01

#define FAT16_ROOT_DIR_CLUSTER 0x00


//-------------------------------------------------------------------
// long file name constants
#define LFN_END 0x40
#define LFN_DEL 0x80

//-------------------------------------------------------------------
// Data Structures

// Take care of packing for GCC - it doesn't obey pragma pack()
// properly for ARM targets.
#ifdef __GNUC__
 #define __PACKED __attribute__ ((__packed__))
#else
 #define __PACKED
// #pragma pack(1)
#endif

// Boot Sector - must be packed
__packed typedef struct
{
  __PACKED u8 jmpBoot[3];
  __PACKED u8 OEMName[8];
  // BIOS Parameter Block
  __PACKED u16  bytesPerSector;
  __PACKED u8 sectorsPerCluster;
  __PACKED u16  reservedSectors;
  __PACKED u8 numFATs;
  __PACKED u16  rootEntries;
  __PACKED u16  numSectorsSmall;
  __PACKED u8 mediaDesc;
  __PACKED u16  sectorsPerFAT;
  __PACKED u16  sectorsPerTrk;
  __PACKED u16  numHeads;
  __PACKED u32  numHiddenSectors;
  __PACKED u32  numSectors;
  __packed union  // Different types of extended BIOS Parameter Block for FAT16 and FAT32
  {
    __packed struct
    {
      // Ext BIOS Parameter Block for FAT16
      __PACKED u8 driveNumber;
      __PACKED u8 reserved1;
      __PACKED u8 extBootSig;
      __PACKED u32  volumeID;
      __PACKED u8 volumeLabel[11];
      __PACKED u8 fileSysType[8];
      // Bootcode
      __PACKED u8 bootCode[448];
    } fat16;
    __packed struct
    {
      // FAT32 extended block
      __PACKED u32  sectorsPerFAT32;
      __PACKED u16  extFlags;
      __PACKED u16  fsVer;
      __PACKED u32  rootClus;
      __PACKED u16  fsInfo;
      __PACKED u16  bkBootSec;
      __PACKED u8 reserved[12];
      // Ext BIOS Parameter Block for FAT16
      __PACKED u8 driveNumber;
      __PACKED u8 reserved1;
      __PACKED u8 extBootSig;
      __PACKED u32  volumeID;
      __PACKED u8 volumeLabel[11];
      __PACKED u8 fileSysType[8];
      // Bootcode
      __PACKED u8 bootCode[420];
    } fat32;
  } extBlock;

  __PACKED u16  bootSig;

} BOOT_SEC;

// Directory entry - must be packed
typedef __packed struct
{
  __PACKED u8 name[8];
  __PACKED u8 ext[3];
	__PACKED u8	attrib;
	__PACKED u8	reserved;
	__PACKED u8	cTime_ms;
	__PACKED u16	cTime;
	__PACKED u16	cDate;
	__PACKED u16	aDate;
	__PACKED u16	startClusterHigh;
	__PACKED u16	mTime;
	__PACKED u16	mDate;
	__PACKED u16	startCluster;
	__PACKED u32	fileSize;
}	DIR_ENT;

// Long file name directory entry - must be packed
typedef __packed struct
{
	__PACKED u8 ordinal;	// Position within LFN
	__PACKED u16 char0;
	__PACKED u16 char1;
	__PACKED u16 char2;
	__PACKED u16 char3;
	__PACKED u16 char4;
	__PACKED u8 flag;	// Should be equal to ATTRIB_LFN
	__PACKED u8 reserved1;	// Always 0x00
	__PACKED u8 checkSum;	// Checksum of short file name (alias)
	__PACKED u16 char5;
	__PACKED u16 char6;
	__PACKED u16 char7;
	__PACKED u16 char8;
	__PACKED u16 char9;
	__PACKED u16 char10;
	__PACKED u16 reserved2;	// Always 0x0000
	__PACKED u16 char11;
	__PACKED u16 char12;
}	DIR_ENT_LFN;

// End of packed structs
#ifdef __PACKED
 #undef __PACKED
#endif
#ifndef __GNUC__
// #pragma pack()
#endif

//-----------------------------------------------------------------
// Global Variables

// _VARS_IN_RAM variables are stored in the largest section of WRAM
// available: IWRAM on NDS ARM7, EWRAM on NDS ARM9 and GBA

// Files
_VARS_IN_RAM FAT_FILE openFiles[MAX_FILES_OPEN];

// Long File names
_VARS_IN_RAM char lfnName[MAX_FILENAME_LENGTH];
bool lfnExists;

// Locations on card
int filesysRootDir;
int filesysRootDirClus;
int filesysFAT;
int filesysSecPerFAT;
int filesysNumSec;
int filesysData;
int filesysBytePerSec;
int filesysSecPerClus;
int filesysBytePerClus;

FS_TYPE filesysType = FS_UNKNOWN;
u32 filesysTotalSize;
```

```
// Info about FAT
u32 fatLastCluster;
u32 fatFirstFree; //cluster

// fatBuffer used to reduce wear on the CF card from multiple writes
_VARS_IN_RAM char fatBuffer[BYTE_PER_READ];
u32 fatBufferCurSector;

// Current working directory
u32 curWorkDirCluster;

// Position of the directory entry last retreived with FAT_GetDirEntry
u32 wrkDirCluster;
int wrkDirSector;
int wrkDirOffset;

// Global sector buffer to save on stack space
_VARS_IN_RAM unsigned char globalBuffer[BYTE_PER_READ];

//-----------------------------------------------------------------
// Functions contained in this file - predeclarations
char ucase (char character);
u16 getRTCtoFileTime (void);
u16 getRTCtoFileDate (void);

bool FAT_AddDirEntry (const char* path, DIR_ENT newDirEntry);
bool FAT_ClearLinks (u32 cluster);
DIR_ENT FAT_DirEntFromPath (const char* path);
u32 FAT_FirstFreeCluster(void);
DIR_ENT FAT_GetDirEntry ( u32 dirCluster, int entry, int origin);
u32 FAT_LinkFreeCluster(u32 cluster);
u32 FAT_NextCluster(u32 cluster);
bool FAT_WriteFatEntry (u32 cluster, u32 value);
bool FAT_GetFilename (DIR_ENT dirEntry, char* alias);

bool FAT_InitFiles (void);
bool FAT_FreeFiles (void);
int FAT_remove (const char* path);
bool FAT_chdir (const char* path);
FILE_TYPE FAT_FindFirstFile (char* filename);
FILE_TYPE FAT_FindNextFile (char* filename);
FILE_TYPE FAT_FileExists (const char* filename);
bool FAT_GetAlias (char* alias);
bool FAT_GetLongFilename (char* filename);
u32 FAT_GetFileSize (void);
u32 FAT_GetFileCluster (void);

FAT_FILE* FAT_fopen(const char* path, const char* mode);
bool FAT_fclose (FAT_FILE* file);
bool FAT_feof(FAT_FILE* file);
int FAT_fseek(FAT_FILE* file, s32 offset, int origin);
u32 FAT_ftell (FAT_FILE* file);
u32 FAT_fread (void* buffer, u32 size, u32 count, FAT_FILE* file);
u32 FAT_fwrite (const void* buffer, u32 size, u32 count, FAT_FILE* file);
char FAT_fgetc (FAT_FILE* file);
char FAT_fputc (char c, FAT_FILE* file);

// semaphore stuff to make functions thread-safe
int FS_lock;
void FAT_lock() {
  lock(&FS_lock);
}
void FAT_unlock() {
  unlock(&FS_lock);
}

/*-----------------------------------------------------------------
ucase
Returns the uppercase version of the given char
char IN: a character
char return OUT: uppercase version of character
```

```
-----------------------------------------------------------------*/
char ucase (char character)
{
  if ((character > 0x60) && (character < 0x7B))
    character = character - 0x20;
  return (character);
}


/*-----------------------------------------------------------------
getRTCtoFileTime and getRTCtoFileDate
Returns the time / date in Dir Entry styled format
u16 return OUT: time / date in Dir Entry styled format
-----------------------------------------------------------------*/
u16 getRTCtoFileTime (void)
{
#ifdef xxxNDS
  return (
    ( ( (IPC->rtc_hours > 11 ? IPC->rtc_hours - 40 : IPC->rtc_hours) & 0x1F) << 11) |
    ( (IPC->rtc_minutes & 0x3F) << 5) |
    ( (IPC->rtc_seconds >> 1) & 0x1F) );
#else
  return 0;
#endif
}

u16 getRTCtoFileDate (void)
{
#ifdef xxxNDS
  return (
    ( ((IPC->rtc_year + 20) & 0x7F) <<9) |
    ( (IPC->rtc_month & 0xF) << 5) |
    (IPC->rtc_day & 0x1F) );
#else
  return 0;
#endif
}


/*-----------------------------------------------------------------
Disc level FAT routines
-----------------------------------------------------------------*/
#define FAT_ClustToSect(m) \
  (((m-2) * filesysSecPerClus) + filesysData)

/*-----------------------------------------------------------------
FAT_NextCluster
Internal function - gets the cluster linked from input cluster
-----------------------------------------------------------------*/
u32 FAT_NextCluster(u32 cluster)
{
  u32 nextCluster = CLUSTER_FREE;
  u32 sector;
  int offset;

  switch (filesysType)
  {
    case FS_UNKNOWN:
      nextCluster = CLUSTER_FREE;
      break;

    case FS_FAT12:
      sector = filesysFAT + (((cluster * 3) / 2) / BYTE_PER_READ);
      offset = ((cluster * 3) / 2) % BYTE_PER_READ;

      // If FAT buffer contains wrong sector
      if (sector != fatBufferCurSector)
      {
        // Load correct sector to buffer
        fatBufferCurSector = sector;
        disc_ReadSector(fatBufferCurSector, fatBuffer);
      }

      nextCluster = ((u8*)fatBuffer)[offset];
      offset++;

      if (offset >= BYTE_PER_READ) {
        offset = 0;
        fatBufferCurSector++;
        disc_ReadSector(fatBufferCurSector, fatBuffer);
      }

      nextCluster |= (((u8*)fatBuffer)[offset]) << 8;

      if (cluster & 0x01) {
        nextCluster = nextCluster >> 4;
      } else   {
        nextCluster &= 0x0FFF;
      }

      if (nextCluster >= 0x0FF7)
      {
        nextCluster = CLUSTER_EOF;
      }

      break;

    case FS_FAT16:
      sector = filesysFAT + ((cluster << 1) / BYTE_PER_READ);
      offset = cluster % (BYTE_PER_READ >> 1);

      // If FAT buffer contains wrong sector
      if (sector != fatBufferCurSector)
      {
        // Load correct sector to buffer
        fatBufferCurSector = sector;
        disc_ReadSector(fatBufferCurSector, fatBuffer);
      }

      // read the nextCluster value
      nextCluster = ((u16*)fatBuffer)[offset];

      if (nextCluster >= 0xFFF7)
      {
        nextCluster = CLUSTER_EOF;
      }
      break;

    case FS_FAT32:
      sector = filesysFAT + ((cluster << 2) / BYTE_PER_READ);
      offset = cluster % (BYTE_PER_READ >> 2);

      // If FAT buffer contains wrong sector
      if (sector != fatBufferCurSector)
      {
        // Load correct sector to buffer
        fatBufferCurSector = sector;
        disc_ReadSector(fatBufferCurSector, fatBuffer);
      }

      // read the nextCluster value
      nextCluster = (((u32*)fatBuffer)[offset]) & 0x0FFFFFFF;

      if (nextCluster >= 0x0FFFFFF7)
      {
        nextCluster = CLUSTER_EOF;
      }
      break;

    default:
      nextCluster = CLUSTER_FREE;
      break;
    }

    return nextCluster;
}

#ifdef CAN_WRITE_TO_DISC
/*-----------------------------------------------------------------
FAT_WriteFatEntry
Internal function - writes FAT information about a cluster
-----------------------------------------------------------------*/
bool FAT_WriteFatEntry (u32 cluster, u32 value)
{
  u32 sector;
  int offset;

  if ((cluster < 0x0002) || (cluster > fatLastCluster))
  {
    return false;
  }

  switch (filesysType)
  {
    case FS_UNKNOWN:
      return false;
      break;

    case FS_FAT12:
      sector = filesysFAT + (((cluster * 3) / 2) / BYTE_PER_READ);
      offset = ((cluster * 3) / 2) % BYTE_PER_READ;

      // If FAT buffer contains wrong sector
      if (sector != fatBufferCurSector)
      {
        // Load correct sector to buffer
        fatBufferCurSector = sector;
        disc_ReadSector(fatBufferCurSector, fatBuffer);
      }

      if (cluster & 0x01) {

        ((u8*)fatBuffer)[offset] = (((u8*)fatBuffer)[offset] & 0x0F) | ((value & 0x0F) << 4);

        offset++;
        if (offset >= BYTE_PER_READ) {
          offset = 0;
          // write the buffer back to disc
          disc_WriteSector(fatBufferCurSector, fatBuffer);
          // read the next sector
          fatBufferCurSector++;
          disc_ReadSector(fatBufferCurSector, fatBuffer);
        }

        ((u8*)fatBuffer)[offset] =  (value & 0x0FF0) >> 4;

      } else {

        ((u8*)fatBuffer)[offset] = value & 0xFF;

        offset++;
        if (offset >= BYTE_PER_READ) {
          offset = 0;
          // write the buffer back to disc
          disc_WriteSector(fatBufferCurSector, fatBuffer);
          // read the next sector
          fatBufferCurSector++;
          disc_ReadSector(fatBufferCurSector, fatBuffer);
        }

        ((u8*)fatBuffer)[offset] = (((u8*)fatBuffer)[offset] & 0xF0) | ((value >> 8) & 0x0F);
      }
      break;

    case FS_FAT16:
      sector = filesysFAT + ((cluster << 1) / BYTE_PER_READ);
      offset = cluster % (BYTE_PER_READ >> 1);

      // If FAT buffer contains wrong sector
      if (sector != fatBufferCurSector)
      {
        // Load correct sector to buffer
        fatBufferCurSector = sector;
        disc_ReadSector(fatBufferCurSector, fatBuffer);
      }

      // write the value to the FAT buffer
      ((u16*)fatBuffer)[offset] = (value & 0xFFFF);

      break;

    case FS_FAT32:
      sector = filesysFAT + ((cluster << 2) / BYTE_PER_READ);
      offset = cluster % (BYTE_PER_READ >> 2);

      // If FAT buffer contains wrong sector
      if (sector != fatBufferCurSector)
      {
        // Load correct sector to buffer
        fatBufferCurSector = sector;
        disc_ReadSector(fatBufferCurSector, fatBuffer);
      }

      // write the value to the FAT buffer
      (((u32*)fatBuffer)[offset]) =  value;

      break;

    default:
      return false;
      break;
  }

  // write the buffer back to disc
  disc_WriteSector(fatBufferCurSector, fatBuffer);

  return true;
}
#endif

#ifdef CAN_WRITE_TO_DISC
/*-----------------------------------------------------------------
FAT_ReadWriteFatEntryBuffered
Internal function - writes FAT information about a cluster to a
 buffer that should then be flushed to disc using
 FAT_WriteFatEntryFlushBuffer()
 Call FAT_WriteFatEntry first so as not to ruin the disc.
 Also returns the entry being replaced
-----------------------------------------------------------------*/
u32 FAT_ReadWriteFatEntryBuffered (u32 cluster, u32 value)
{
  u32 sector;
  int offset;
  u32 oldValue;

  if ((cluster < 0x0002) || (cluster > fatLastCluster))
    return CLUSTER_FREE;


  switch (filesysType)
  {
    case FS_UNKNOWN:
      oldValue = CLUSTER_FREE;
```

```
        break;

      case FS_FAT12:
        sector = filesysFAT + (((cluster * 3) / 2) / BYTE_PER_READ);
        offset = ((cluster * 3) / 2) % BYTE_PER_READ;

        // If FAT buffer contains wrong sector
        if (sector != fatBufferCurSector)
        {
          // write the old buffer to disc
          if ((fatBufferCurSector >= filesysFAT) && (fatBufferCurSector < (filesysFAT +
filesysSecPerFAT)))
            disc_WriteSector(fatBufferCurSector, fatBuffer);
          // Load correct sector to buffer
          fatBufferCurSector = sector;
          disc_ReadSector(fatBufferCurSector, fatBuffer);
        }

        if (cluster & 0x01) {

          oldValue = (((u8*)fatBuffer)[offset] & 0xF0) >> 4;
          ((u8*)fatBuffer)[offset] = (((u8*)fatBuffer)[offset] & 0x0F) | ((value & 0x0F) << 4);

          offset++;
          if (offset >= BYTE_PER_READ) {
            offset = 0;
            // write the buffer back to disc
            disc_WriteSector(fatBufferCurSector, fatBuffer);
            // read the next sector
            fatBufferCurSector++;
            disc_ReadSector(fatBufferCurSector, fatBuffer);
          }

          oldValue |= ((((u8*)fatBuffer)[offset]) << 4) & 0x0FF0;
          ((u8*)fatBuffer)[offset] =  (value & 0x0FF0) >> 4;

        } else {

          oldValue = ((u8*)fatBuffer)[offset] & 0xFF;
          ((u8*)fatBuffer)[offset] = value & 0xFF;

          offset++;
          if (offset >= BYTE_PER_READ) {
            offset = 0;
            // write the buffer back to disc
            disc_WriteSector(fatBufferCurSector, fatBuffer);
            // read the next sector
            fatBufferCurSector++;
            disc_ReadSector(fatBufferCurSector, fatBuffer);
          }

          oldValue |= (((u8*)fatBuffer)[offset] & 0x0F) << 8;
          ((u8*)fatBuffer)[offset] = (((u8*)fatBuffer)[offset] & 0xF0) | ((value >> 8) & 0x0F);
        }

        if (oldValue >= 0x0FF7)
        {
          oldValue = CLUSTER_EOF;
        }

        break;

      case FS_FAT16:
        sector = filesysFAT + ((cluster << 1) / BYTE_PER_READ);
        offset = cluster % (BYTE_PER_READ >> 1);

        // If FAT buffer contains wrong sector
        if (sector != fatBufferCurSector)
        {
          // write the old buffer to disc
          if ((fatBufferCurSector >= filesysFAT) && (fatBufferCurSector < (filesysFAT + 
 filesysSecPerFAT)))
           disc_WriteSector(fatBufferCurSector, fatBuffer);
          // Load correct sector to buffer
          fatBufferCurSector = sector;
          disc_ReadSector(fatBufferCurSector, fatBuffer);
        }

        // write the value to the FAT buffer
        oldValue = ((u16*)fatBuffer)[offset];
        ((u16*)fatBuffer)[offset] = value;

        if (oldValue >= 0xFFF7)
        {
          oldValue = CLUSTER_EOF;
        }

        break;

      case FS_FAT32:
        sector = filesysFAT + ((cluster << 2) / BYTE_PER_READ);
        offset = cluster % (BYTE_PER_READ >> 2);

        // If FAT buffer contains wrong sector
        if (sector != fatBufferCurSector)
        {
          // write the old buffer to disc
          if ((fatBufferCurSector >= filesysFAT) && (fatBufferCurSector < (filesysFAT + 
 filesysSecPerFAT)))
           disc_WriteSector(fatBufferCurSector, fatBuffer);
          // Load correct sector to buffer
          fatBufferCurSector = sector;
          disc_ReadSector(fatBufferCurSector, fatBuffer);
        }

        // write the value to the FAT buffer
        oldValue = ((u32*)fatBuffer)[offset];
        ((u32*)fatBuffer)[offset] =  value;

        if (oldValue >= 0x0FFFFFF7)
        {
          oldValue = CLUSTER_EOF;
        }

        break;

      default:
        oldValue = CLUSTER_FREE;
        break;
    }

    return oldValue;
 }
 #endif

 #ifdef CAN_WRITE_TO_DISC
 /*-----------------------------------------------------------------
 FAT_WriteFatEntryFlushBuffer
 Flush the FAT buffer back to the disc
 -----------------------------------------------------------------*/
 bool FAT_WriteFatEntryFlushBuffer (void)
 {
    // write the buffer disc
    if ((fatBufferCurSector >= filesysFAT) && (fatBufferCurSector < (filesysFAT + 
 filesysSecPerFAT)))
    {
      disc_WriteSector(fatBufferCurSector, fatBuffer);
      return true;
    } else {
      return false;
    }
```

```
}
#endif

#ifdef CAN_WRITE_TO_DISC
/*-----------------------------------------------------------------
FAT_FirstFreeCluster
Internal function - gets the first available free cluster
-----------------------------------------------------------------*/
u32 FAT_FirstFreeCluster(void)
{
  u32 start;

  // Start at first valid cluster
  if (fatFirstFree < CLUSTER_FIRST)
    fatFirstFree = CLUSTER_FIRST;

  start=fatFirstFree;
//***
  while (FAT_NextCluster(fatFirstFree) != CLUSTER_FREE)
  {
    fatFirstFree++;
    if (fatFirstFree > fatLastCluster) {
      fatFirstFree = CLUSTER_FIRST;//wraparound..
    }
    if(fatFirstFree==start) {
      return CLUSTER_EOF;
    }
  }
  return fatFirstFree;
}
#endif

#ifdef CAN_WRITE_TO_DISC
/*-----------------------------------------------------------------
FAT_LinkFreeCluster
Internal function - gets the first available free cluster, sets it
to end of file, links the input cluster to it then returns the
cluster number
-----------------------------------------------------------------*/
u32 FAT_LinkFreeCluster(u32 cluster)
{
  u32 firstFree;
  u32 curLink;

  if (cluster > fatLastCluster)
  {
    return CLUSTER_FREE;
  }

  // Check if the cluster already has a link, and return it if so
  curLink = FAT_NextCluster (cluster);
  if ((curLink >= CLUSTER_FIRST) && (curLink < fatLastCluster))
  {
    return curLink; // Return the current link - don't allocate a new one
  }

  // Get a free cluster
  firstFree = FAT_FirstFreeCluster();

  // If couldn't get a free cluster then return
  if (firstFree == CLUSTER_EOF)
  {
    return CLUSTER_FREE;
  }
//***
  if ((cluster >= CLUSTER_FIRST) && (cluster <= fatLastCluster))
  {
    // Update the linked from FAT entry
    FAT_WriteFatEntry (cluster, firstFree);
  }
  // Create the linked to FAT entry
  FAT_WriteFatEntry (firstFree, CLUSTER_EOF);

  return firstFree;
}
#endif


#ifdef CAN_WRITE_TO_DISC
/*-----------------------------------------------------------------
FAT_ClearLinks
Internal function - frees any cluster used by a file
-----------------------------------------------------------------*/
bool FAT_ClearLinks (u32 cluster)
{
  u32 nextCluster;

  if ((cluster < 0x0002) || (cluster > fatLastCluster))
    return false;

  fatFirstFree=cluster;

  // remember next cluster before erasing the link
  nextCluster = FAT_NextCluster (cluster);

  // Erase the link
  FAT_WriteFatEntry (cluster, CLUSTER_FREE);

  // Move onto next cluster
  cluster = nextCluster;

  while ((cluster != CLUSTER_EOF) && (cluster != CLUSTER_FREE))
  {
    cluster = FAT_ReadWriteFatEntryBuffered (cluster, CLUSTER_FREE);
  }

  // Flush fat write buffer
  FAT_WriteFatEntryFlushBuffer ();

  return true;
}
#endif


/*-----------------------------------------------------------------
FAT_InitFiles
Reads the FAT information from the CF card.
You need to call this before reading any files.
bool return OUT: true if successful.
-----------------------------------------------------------------*/
bool FAT_InitFiles (void)
{
  int i;
  int bootSector;
  BOOT_SEC* bootSec;

  FAT_unlock(); //semaphore reset

  if (!disc_Init())
  {
    return (false);
  }

  // Read first sector of CF card
  disc_ReadSector (0, globalBuffer);
  // Check if there is a FAT string, which indicates this is a boot sector
  if ((globalBuffer[0x36] == 'F') && (globalBuffer[0x37] == 'A') && (globalBuffer[0x38] == 'T'))
  {
    bootSector = 0;
  }
  // Check for FAT32
  else if ((globalBuffer[0x52] == 'F') && (globalBuffer[0x53] == 'A') && (globalBuffer[0x54] ==
'T'))
  {
    bootSector = 0;
  }
  else // This is an MBR
  {
    // Find first valid partition from MBR
    // First check for an active partition
    for (i=0x1BE; (i < 0x1FE) && (globalBuffer[i] != 0x80); i+= 0x10);
    // If it didn't find an active partition, search for any valid partition
    if (i == 0x1FE)
      for (i=0x1BE; (i < 0x1FE) && (globalBuffer[i+0x04] == 0x00); i+= 0x10);

    // Go to first valid partition
    if ( i != 0x1FE)  // Make sure it found a partition
    {
      bootSector = globalBuffer[0x8 + i] + (globalBuffer[0x9 + i] << 8) + (globalBuffer[0xA + i]
<< 16) + ((globalBuffer[0xB + i] << 24) & 0x0F);
    } else {
      bootSector = 0; // No partition found, assume this is a MBR free disk
    }
  }

  // Read in boot sector
  bootSec = (BOOT_SEC*) globalBuffer;
  disc_ReadSector (bootSector,  (void*)bootSec);

  // Store required information about the file system
  if (bootSec->sectorsPerFAT != 0)
  {
    filesysSecPerFAT = bootSec->sectorsPerFAT;
  }
  else
  {
    filesysSecPerFAT = bootSec->extBlock.fat32.sectorsPerFAT32;
  }

  if (bootSec->numSectorsSmall != 0)
  {
    filesysNumSec = bootSec->numSectorsSmall;
  }
  else
  {
    filesysNumSec = bootSec->numSectors;
  }

  filesysBytePerSec = BYTE_PER_READ;  // Sector size is redefined to be 512 bytes
  filesysSecPerClus = bootSec->sectorsPerCluster * bootSec->bytesPerSector / BYTE_PER_READ;
  filesysBytePerClus = filesysBytePerSec * filesysSecPerClus;
  filesysFAT = bootSector + bootSec->reservedSectors;

  filesysRootDir = filesysFAT + (bootSec->numFATs * filesysSecPerFAT);
  filesysData = filesysRootDir + ((bootSec->rootEntries * sizeof(DIR_ENT)) / filesysBytePerSec);

  filesysTotalSize = (filesysNumSec - filesysData) * filesysBytePerSec;

  // Store info about FAT
  fatLastCluster = (filesysNumSec - filesysData) / bootSec->sectorsPerCluster;
  fatFirstFree = CLUSTER_FIRST;
  fatBufferCurSector = 0;
  disc_ReadSector(fatBufferCurSector, fatBuffer);

  if (fatLastCluster < 4085)
  {
    filesysType = FS_FAT12;  // FAT12 volume - unsupported
  }
  else if (fatLastCluster < 65525)
  {
    filesysType = FS_FAT16;  // FAT16 volume
  }
  else
  {
    filesysType = FS_FAT32;  // FAT32 volume
  }

  if (filesysType != FS_FAT32)
  {
    filesysRootDirClus = FAT16_ROOT_DIR_CLUSTER;
  }
  else // Set up for the FAT32 way
  {
    filesysRootDirClus = bootSec->extBlock.fat32.rootClus;
    // Check if FAT mirroring is enabled
    if (!(bootSec->extBlock.fat32.extFlags & 0x80))
    {
      // Use the active FAT
      filesysFAT = filesysFAT + ( filesysSecPerFAT * (bootSec->extBlock.fat32.extFlags & 0x0F));
    }
  }

  // Set current directory to the root
  curWorkDirCluster = filesysRootDirClus;
  wrkDirCluster = filesysRootDirClus;
  wrkDirSector = 0;
  wrkDirOffset = 0;

  // Set all files to free
  for (i=0; i < MAX_FILES_OPEN; i++)
  {
    openFiles[i].inUse = false;
  }

  // No long filenames so far
  lfnExists = false;
  for (i = 0; i < MAX_FILENAME_LENGTH; i++)
  {
    lfnName[i] = '\0';
  }

  return (true);
}

/*-----------------------------------------------------------------
FAT_FreeFiles
Closes all open files then resets the CF card.
Call this before exiting back to the GBAMP
bool return OUT: true if successful.
-----------------------------------------------------------------*/
bool FAT_FreeFiles (void)
{
  int i;

  // Close all open files
  for (i=0; i < MAX_FILES_OPEN; i++)
  {
    if (openFiles[i].inUse == true)
    {
      FAT_fclose(&openFiles[i]);
    }
  }

  // Flush any sectors in disc cache
  disc_CacheFlush();

  // Clear card status
  disc_ClearStatus();

  // Return status of card
  return disc_IsInserted();
}
```

```
/*-----------------------------------------------------------------
FAT_GetDirEntry
Return the file info structure of the next valid file entry
u32 dirCluster: IN cluster of subdirectory table
int entry: IN the desired file entry
int origin IN: relative position of the entry
DIR_ENT return OUT: desired dirEntry. First char will be FILE_FREE if
  the entry does not exist.
-----------------------------------------------------------------*/
DIR_ENT FAT_GetDirEntry ( u32 dirCluster, int entry, int origin)
{
  DIR_ENT dir;
  DIR_ENT_LFN lfn;
  int firstSector = 0;
  bool notFound = false;
  bool found = false;
  int maxSectors;
  int lfnPos, aliasPos;
  u8 lfnChkSum, chkSum;

  dir.name[0] = FILE_FREE; // default to no file found
  dir.attrib = 0x00;

  // Check if fat has been initialised
  if (filesysBytePerSec == 0)
  {
    return (dir);
  }

  switch (origin)
  {
  case SEEK_SET:
    wrkDirCluster = dirCluster;
    wrkDirSector = 0;
    wrkDirOffset = -1;
    break;
  case SEEK_CUR:  // Don't change anything
    break;
  case SEEK_END:  // Find entry signifying end of directory
    // Subtraction will never reach 0, so it keeps going
    // until reaches end of directory
    wrkDirCluster = dirCluster;
    wrkDirSector = 0;
    wrkDirOffset = -1;
    entry = -1;
    break;
  default:
    return dir;
  }

  lfnChkSum = 0;
  maxSectors = (wrkDirCluster == FAT16_ROOT_DIR_CLUSTER ? (filesysData - filesysRootDir) :
filesysSecPerClus);

  // Scan Dir for correct entry
  firstSector = (wrkDirCluster == FAT16_ROOT_DIR_CLUSTER ? filesysRootDir :
FAT_ClustToSect(wrkDirCluster));
  disc_ReadSector (firstSector + wrkDirSector, globalBuffer);
  found = false;
  notFound = false;
  do {
    wrkDirOffset++;
    if (wrkDirOffset == BYTE_PER_READ / sizeof (DIR_ENT))
    {
      wrkDirOffset = 0;
      wrkDirSector++;
      if ((wrkDirSector == filesysSecPerClus) && (wrkDirCluster != FAT16_ROOT_DIR_CLUSTER))
      {
        wrkDirSector = 0;
        wrkDirCluster = FAT_NextCluster(wrkDirCluster);
        if (wrkDirCluster == CLUSTER_EOF)
        {
          notFound = true;
        }
        firstSector = FAT_ClustToSect(wrkDirCluster);
      }
      else if ((wrkDirCluster == FAT16_ROOT_DIR_CLUSTER) && (wrkDirSector == (filesysData - 
filesysRootDir)))
      {
        notFound = true;  // Got to end of root dir
      }
      disc_ReadSector (firstSector + wrkDirSector, globalBuffer);
    }
    dir = ((DIR_ENT*) globalBuffer)[wrkDirOffset];
    if ((dir.name[0] != FILE_FREE) && (dir.name[0] > 0x20) && ((dir.attrib & ATTRIB_VOL) != 
ATTRIB_VOL))
    {
      entry--;
      if (lfnExists)
      {
        // Calculate file checksum
        chkSum = 0;
        for (aliasPos=0; aliasPos < 11; aliasPos++)
        {
          // NOTE: The operation is an unsigned char rotate right
          chkSum = ((chkSum & 1) ? 0x80 : 0) + (chkSum >> 1) + (aliasPos < 8 ? 
dir.name[aliasPos] : dir.ext[aliasPos - 8]);
        }
        if (chkSum != lfnChkSum)
        {
          lfnExists = false;
          lfnName[0] = '\0';
        }
      }
      if (entry == 0)
      {
        if (!lfnExists)
        {
          FAT_GetFilename (dir, lfnName);
        }
        found = true;
      }
    }
    else if (dir.name[0] == FILE_LAST)
    {
      if (origin == SEEK_END)
      {
        found = true;
      }
      else
      {
        notFound = true;
      }
    }
    else if (dir.attrib == ATTRIB_LFN)
    {
      lfn = ((DIR_ENT_LFN*) globalBuffer)[wrkDirOffset];
      if (lfn.ordinal & LFN_DEL)
      {
        lfnExists = false;
      }
      else if (lfn.ordinal & LFN_END) // Last part of LFN, make sure it isn't deleted (Thanks 
MoonLight)
      {
        lfnExists = true;
        lfnName[(lfn.ordinal & ~LFN_END) * 13] = '\0'; // Set end of lfn to null character
        lfnChkSum = lfn.checkSum;
      }
      if (lfnChkSum != lfn.checkSum)
      {
        lfnExists = false;
      }
      if (lfnExists)
      {
        lfnPos = ((lfn.ordinal & ~LFN_END) - 1) * 13;
        lfnName[lfnPos + 0] = lfn.char0 & 0xFF;
        lfnName[lfnPos + 1] = lfn.char1 & 0xFF;
        lfnName[lfnPos + 2] = lfn.char2 & 0xFF;
        lfnName[lfnPos + 3] = lfn.char3 & 0xFF;
        lfnName[lfnPos + 4] = lfn.char4 & 0xFF;
        lfnName[lfnPos + 5] = lfn.char5 & 0xFF;
        lfnName[lfnPos + 6] = lfn.char6 & 0xFF;
        lfnName[lfnPos + 7] = lfn.char7 & 0xFF;
        lfnName[lfnPos + 8] = lfn.char8 & 0xFF;
        lfnName[lfnPos + 9] = lfn.char9 & 0xFF;
        lfnName[lfnPos + 10] = lfn.char10 & 0xFF;
        lfnName[lfnPos + 11] = lfn.char11 & 0xFF;
        lfnName[lfnPos + 12] = lfn.char12 & 0xFF;
      }
    }
  } while (!found && !notFound);

  // If no file is found, return FILE_FREE
  if (notFound)
  {
    dir.name[0] = FILE_FREE;
  }

  return (dir);
}


/*-----------------------------------------------------------------
FAT_GetLongFilename
Get the long name of the last file or directory retrived with
  GetDirEntry. Also works for FindFirstFile and FindNextFile.
  If a long name doesn't exist, it returns the short name
  instead.
char* filename: OUT will be filled with the filename, should be at
  least 256 bytes long
bool return OUT: return true if successful
-----------------------------------------------------------------*/
bool FAT_GetLongFilename (char* filename)
{
  if (filename == NULL)
    return false;

  strncpy (filename, lfnName, MAX_FILENAME_LENGTH - 1);
  filename[MAX_FILENAME_LENGTH - 1] = '\0';

  return true;
}


/*-----------------------------------------------------------------
FAT_GetFilename
Get the alias (short name) of the file or directory stored in
  dirEntry
DIR_ENT dirEntry: IN a valid directory table entry
char* alias OUT: will be filled with the alias (short filename),
  should be at least 13 bytes long
bool return OUT: return true if successful
-----------------------------------------------------------------*/
bool FAT_GetFilename (DIR_ENT dirEntry, char* alias)
{
  int i=0;
  int j=0;

  alias[0] = '\0';
  if (dirEntry.name[0] != FILE_FREE)
  {
    if (dirEntry.name[0] == '.')
    {
      alias[0] = '.';
      if (dirEntry.name[1] == '.')
      {
        alias[1] = '.';
        alias[2] = '\0';
      }
      else
      {
        alias[1] = '\0';
      }
    }
    else
    {
      // Copy the filename from the dirEntry to the string
      for (i = 0; (i < 8) && (dirEntry.name[i] != ' '); i++)
      {
        alias[i] = dirEntry.name[i];
      }
      // Copy the extension from the dirEntry to the string
      if (dirEntry.ext[0] != ' ')
      {
        alias[i++] = '.';
        for ( j = 0; (j < 3) && (dirEntry.ext[j] != ' '); j++)
        {
          alias[i++] = dirEntry.ext[j];
        }
      }
      alias[i] = '\0';
    }
  }

  return (alias[0] != '\0');
}

/*-----------------------------------------------------------------
FAT_GetAlias
Get the alias (short name) of the last file or directory entry read
  using GetDirEntry. Works for FindFirstFile and FindNextFile
char* alias OUT: will be filled with the alias (short filename),
  should be at least 13 bytes long
bool return OUT: return true if successful
-----------------------------------------------------------------*/
bool FAT_GetAlias (char* alias)
{
  bool ret;
  if (alias == NULL)
  {
    return false;
  }

  FAT_lock();

  // Read in the last accessed directory entry
  disc_ReadSector ((wrkDirCluster == FAT16_ROOT_DIR_CLUSTER ? filesysRootDir :
FAT_ClustToSect(wrkDirCluster)) + wrkDirSector, globalBuffer);

  ret=FAT_GetFilename (((DIR_ENT*)globalBuffer)[wrkDirOffset], alias);
  FAT_unlock();
  return ret;
}

/*-----------------------------------------------------------------
FAT_GetFileSize
Get the file size of the last file found or openned.
This idea is based on a modification by MoonLight
u32 return OUT: the file size
-----------------------------------------------------------------*/
u32 FAT_GetFileSize (void)
{
  // Read in the last accessed directory entry
  disc_ReadSector ((wrkDirCluster == FAT16_ROOT_DIR_CLUSTER ? filesysRootDir :
FAT_ClustToSect(wrkDirCluster)) + wrkDirSector, globalBuffer);

  return   ((DIR_ENT*)globalBuffer)[wrkDirOffset].fileSize;
}

/*-----------------------------------------------------------------
FAT_GetFileCluster
Get the first cluster of the last file found or openned.
u32 return OUT: the file start cluster
-----------------------------------------------------------------*/
u32 FAT_GetFileCluster (void)
{
  // Read in the last accessed directory entry
  disc_ReadSector ((wrkDirCluster == FAT16_ROOT_DIR_CLUSTER ? filesysRootDir :
FAT_ClustToSect(wrkDirCluster)) + wrkDirSector, globalBuffer);

  return   (((DIR_ENT*)globalBuffer)[wrkDirOffset].startCluster) |
(((DIR_ENT*)globalBuffer)[wrkDirOffset].startClusterHigh << 16);
}

#ifdef FILE_TIME_SUPPORT
time_t FAT_FileTimeToCTime (u16 fileTime, u16 fileDate)
{
  struct tm timeInfo;

  timeInfo.tm_year = (fileDate >> 9) + 80;   // years since midnight January 1970
  timeInfo.tm_mon = ((fileDate >> 5) & 0xf) - 1; // Months since january
  timeInfo.tm_mday = fileDate & 0x1f;        // Day of the month

  timeInfo.tm_hour = fileTime >> 11;         // hours past midnight
  timeInfo.tm_min = (fileTime >> 5) & 0x3f;    // minutes past the hour
  timeInfo.tm_sec = (fileTime & 0x1f) * 2;   // seconds past the minute

  return mktime(&timeInfo);
}

/*-----------------------------------------------------------------
FAT_GetFileCreationTime
Get the creation time of the last file found or openned.
time_t return OUT: the file's creation time
-----------------------------------------------------------------*/
time_t FAT_GetFileCreationTime (void)
{
  // Read in the last accessed directory entry
  disc_ReadSector ((wrkDirCluster == FAT16_ROOT_DIR_CLUSTER ? filesysRootDir :
FAT_ClustToSect(wrkDirCluster)) + wrkDirSector, globalBuffer);

  return   FAT_FileTimeToCTime(((DIR_ENT*)globalBuffer)[wrkDirOffset].cTime,
((DIR_ENT*)globalBuffer)[wrkDirOffset].cDate);
}

/*-----------------------------------------------------------------
FAT_GetFileLastWriteTime
Get the creation time of the last file found or openned.
time_t return OUT: the file's creation time
-----------------------------------------------------------------*/
time_t FAT_GetFileLastWriteTime (void)
{
  // Read in the last accessed directory entry
  disc_ReadSector ((wrkDirCluster == FAT16_ROOT_DIR_CLUSTER ? filesysRootDir :
FAT_ClustToSect(wrkDirCluster)) + wrkDirSector, globalBuffer);

  return   FAT_FileTimeToCTime(((DIR_ENT*)globalBuffer)[wrkDirOffset].mTime,
((DIR_ENT*)globalBuffer)[wrkDirOffset].mDate);
}
#endif

/*-----------------------------------------------------------------
FAT_DirEntFromPath
Finds the directory entry for a file or directory from a path
```

```
Path separator is a forward slash /
const char* path: IN null terminated string of path.
DIR_ENT return OUT: dirEntry of found file. First char will be FILE_FREE
  if the file was not found
-----------------------------------------------------------------*/
DIR_ENT FAT_DirEntFromPath (const char* path)
{
  int pathPos;
  char name[MAX_FILENAME_LENGTH];
  char alias[13];
  int namePos;
  bool found, notFound;
  DIR_ENT dirEntry;
  u32 dirCluster;
  bool flagLFN, dotSeen;

  // Start at beginning of path
  pathPos = 0;

  if (path[pathPos] == '/')
  {
    dirCluster = filesysRootDirClus;  // Start at root directory
  }
  else
  {
    dirCluster = curWorkDirCluster; // Start at current working dir
  }

  // Eat any slash /
  while ((path[pathPos] == '/') && (path[pathPos] != '\0'))
  {
    pathPos++;
  }

  // Search until can't continue
  found = false;
  notFound = false;
  while (!notFound && !found)
  {
    flagLFN = false;
    // Copy name from path
    namePos = 0;
    if ((path[pathPos] == '.') && ((path[pathPos + 1] == '\0') || (path[pathPos + 1] == '/'))) {
      // Dot entry
      name[namePos++] = '.';
      pathPos++;
    } else if ((path[pathPos] == '.') && (path[pathPos + 1] == '.') && ((path[pathPos + 2] ==
'\0') || (path[pathPos + 2] == '/'))){
      // Double dot entry
      name[namePos++] = '.';
      pathPos++;
      name[namePos++] = '.';
      pathPos++;
    } else {
      // Copy name from path
      if (path[pathPos] == '.') {
        flagLFN = true;
      }
      dotSeen = false;
      while ((namePos < MAX_FILENAME_LENGTH - 1) && (path[pathPos] != '\0') && (path[pathPos] !=
'/'))
      {
        name[namePos] = ucase(path[pathPos]);
        if ((name[namePos] <= ' ') || ((name[namePos] >= ':') && (name[namePos] <= '?'))) //
Invalid character
        {
          flagLFN = true;
        }
        if (name[namePos] == '.') {
          if (!dotSeen) {
            dotSeen = true;
          } else {
            flagLFN = true;
          }
        }
        namePos++;
        pathPos++;
      }
      // Check if a long filename was specified
      if (namePos > 12)
      {
        flagLFN = true;
      }
    }

    // Add end of string char
    name[namePos] = '\0';

    // Move through path to correct place
    while ((path[pathPos] != '/') && (path[pathPos] != '\0'))
      pathPos++;
    // Eat any slash /
    while ((path[pathPos] == '/') && (path[pathPos] != '\0'))
    {
      pathPos++;
    }

    // Search current Dir for correct entry
    dirEntry = FAT_GetDirEntry (dirCluster, 1, SEEK_SET);
    while ( !found && !notFound)
    {
      // Match filename
      found = true;
      for (namePos = 0; (namePos < MAX_FILENAME_LENGTH) && found && (name[namePos] != '\0') &&
(lfnName[namePos] != '\0'); namePos++)
      {
        if (name[namePos] != ucase(lfnName[namePos]))
        {
          found = false;
        }
      }
      if ((name[namePos] == '\0') != (lfnName[namePos] == '\0'))
      {
        found = false;
      }

      // Check against alias as well.
      if (!found)
      {
        FAT_GetFilename(dirEntry, alias);
        found = true;
        for (namePos = 0; (namePos < 13) && found && (name[namePos] != '\0') && (alias[namePos]
!= '\0'); namePos++)
        {
          if (name[namePos] != ucase(alias[namePos]))
          {
            found = false;
          }
        }
        if ((name[namePos] == '\0') != (alias[namePos] == '\0'))
        {
          found = false;
        }
      }

      if (dirEntry.name[0] == FILE_FREE)
        // Couldn't find specified file
      {
        found = false;
        notFound = true;
      }
      if (!found && !notFound)
```

```
        {
          dirEntry = FAT_GetDirEntry (dirCluster, 1, SEEK_CUR);
        }
      }

      if (found && ((dirEntry.attrib & ATTRIB_DIR) == ATTRIB_DIR) && (path[pathPos] != '\0'))
        // It has found a directory from within the path that needs to be followed
      {
        found = false;
        dirCluster = dirEntry.startCluster | (dirEntry.startClusterHigh << 16);
      }
    }

    if (notFound)
    {
      dirEntry.name[0] = FILE_FREE;
      dirEntry.attrib = 0x00;
    }

    return (dirEntry);
  }


  #ifdef CAN_WRITE_TO_DISC
  /*-----------------------------------------------------------------
  FAT_AddDirEntry
  Creates a new dir entry for a file
  Path separator is a forward slash /
  const char* path: IN null terminated string of path to file.
  DIR_ENT newDirEntry IN: The directory entry to use.
  int file IN: The file being added (optional, use -1 if not used)
  bool return OUT: true if successful
  -----------------------------------------------------------------*/
  bool FAT_AddDirEntry (const char* path, DIR_ENT newDirEntry)
  {
    char filename[MAX_FILENAME_LENGTH];
    int filePos, pathPos, aliasPos;
    char tempChar;
    bool flagLFN, dotSeen;
    char fileAlias[13] = {0};
    int tailNum;

    unsigned char chkSum = 0;

    u32 oldWorkDirCluster;

    DIR_ENT* dirEntries = (DIR_ENT*)globalBuffer;
    u32 dirCluster;
    int secOffset;
    int entryOffset;
    int maxSectors;
    u32 firstSector;

    DIR_ENT temp;
    DIR_ENT_LFN lfnEntry;
    int lfnPos = 0;

    int dirEntryLength = 0;
    int dirEntryRemain = 0;
    u32 tempDirCluster;
    int tempSecOffset;
    int tempEntryOffset;
    bool dirEndFlag = false;

    // Store current working directory
    oldWorkDirCluster = curWorkDirCluster;

    // Find filename within path and change to correct directory
    if (path[0] == '/')
    {
      curWorkDirCluster = filesysRootDirClus;
    }

    pathPos = 0;
    filePos = 0;
    flagLFN = false;

    while (path[pathPos + filePos] != '\0')
    {
      if (path[pathPos + filePos] == '/')
      {
        filename[filePos] = '\0';
        if (FAT_chdir(filename) == false)
        {
          curWorkDirCluster = oldWorkDirCluster;
          return false; // Couldn't change directory
        }
        pathPos += filePos + 1;
        filePos = 0;
      }
      filename[filePos] = path[pathPos + filePos];
      filePos++;
    }

    // Skip over last slashes
    while (path[pathPos] == '/')
      pathPos++;

    // Check if the filename has a leading "."
    // If so, it is an LFN
    if (path[pathPos] == '.') {
      flagLFN = true;
    }

    // Copy name from path
    filePos = 0;
    dotSeen = false;

    while ((filePos < MAX_FILENAME_LENGTH - 1) && (path[pathPos] != '\0'))
    {
      filename[filePos] = path[pathPos];
      if ((filename[filePos] <= ' ') || ((filename[filePos] >= ':') && (filename[filePos] <=
  '?'))) // Invalid character
      {
        flagLFN = true;
      }
      if (filename[filePos] == '.') {
        if (!dotSeen) {
          dotSeen = true;
        } else {
          flagLFN = true;
        }
      }
      filePos++;
      pathPos++;
      if ((filePos > 8) && !dotSeen) {
        flagLFN = true;
      }
    }

    if (filePos == 0) // No filename
    {
      return false;
    }

    // Check if a long filename was specified
    if (filePos > 12)
    {
      flagLFN = true;
    }

    // Check if extension is > 3 characters long
    if (!flagLFN && (strrchr (filename, '.') != NULL) && (strlen(strrchr(filename, '.')) > 4)) {
      flagLFN = true;
    }

    lfnPos = (filePos - 1) / 13;

    // Add end of string char
    filename[filePos++] = '\0';
    // Clear remaining chars
    while (filePos < MAX_FILENAME_LENGTH)
      filename[filePos++] = 0x01;  // Set for LFN compatibility


    if (flagLFN)
    {
      // Generate short filename - always a 2 digit number for tail
      // Get first 5 chars of alias from LFN
      aliasPos = 0;
      filePos = 0;
      if (filename[filePos] == '.') {
        filePos++;
      }
      for ( ; (aliasPos < 5) && (filename[filePos] != '\0') && (filename[filePos] != '.') ;
  filePos++)
      {
        tempChar = ucase(filename[filePos]);
        if (((tempChar > ' ' && tempChar < ':') || tempChar > '?') && tempChar != '.')
          fileAlias[aliasPos++] = tempChar;
      }
      // Pad Alias with underscores
      while (aliasPos < 5)
        fileAlias[aliasPos++] = '_';

      fileAlias[5] = '~';
      fileAlias[8] = '.';
      fileAlias[9] = ' ';
      fileAlias[10] = ' ';
      fileAlias[11] = ' ';
      if (strchr (filename, '.') != NULL) {
        while(filename[filePos] != '\0')
        {
          filePos++;
          if (filename[filePos] == '.')
          {
            pathPos = filePos;
          }
        }
        filePos = pathPos + 1; //pathPos is used as a temporary variable
        // Copy first 3 characters of extension
        for (aliasPos = 9; (aliasPos < 12) && (filename[filePos] != '\0'); filePos++)
        {
          tempChar = ucase(filename[filePos]);
          if ((tempChar > ' ' && tempChar < ':') || tempChar > '?')
            fileAlias[aliasPos++] = tempChar;
        }
      } else {
        aliasPos = 9;
      }

      // Pad Alias extension with spaces
      while (aliasPos < 12)
        fileAlias[aliasPos++] = ' ';

      fileAlias[12] = '\0';


      // Get a valid tail number
      tailNum = 0;
      do {
        tailNum++;
        fileAlias[6] = 0x30 + ((tailNum / 10) % 10); // 10's digit
        fileAlias[7] = 0x30 + (tailNum % 10); // 1's digit
        temp=FAT_DirEntFromPath(fileAlias);
      } while ((temp.name[0] != FILE_FREE) && (tailNum < 100));

      if (tailNum < 100)  // Found an alias not being used
      {
        // Calculate file checksum
        chkSum = 0;
        for (aliasPos=0; aliasPos < 12; aliasPos++)
        {
          // Skip '.'
          if (fileAlias[aliasPos] == '.')
             aliasPos++;
          // NOTE: The operation is an unsigned char rotate right
          chkSum = ((chkSum & 1) ? 0x80 : 0) + (chkSum >> 1) + fileAlias[aliasPos];
        }
      }
      else // Couldn't find a valid alias
      {
        return false;
      }

      dirEntryLength = lfnPos + 2;
    }
    else // Its not a long file name
    {
      // Just copy alias straight from filename
      for (aliasPos = 0; aliasPos < 13; aliasPos++)
      {
        tempChar = ucase(filename[aliasPos]);
        if ((tempChar > ' ' && tempChar < ':') || tempChar > '?')
          fileAlias[aliasPos] = tempChar;
      }
      fileAlias[12] = '\0';

      lfnPos = -1;

      dirEntryLength = 1;
    }

    // Change dirEntry name to match alias
    for (aliasPos = 0; ((fileAlias[aliasPos] != '.') && (fileAlias[aliasPos] != '\0') && (aliasPos
  < 8)); aliasPos++)
    {
      newDirEntry.name[aliasPos] = fileAlias[aliasPos];
    }
    while (aliasPos < 8)
    {
      newDirEntry.name[aliasPos++] = ' ';
    }
    aliasPos = 0;
    while ((fileAlias[aliasPos] != '.') && (fileAlias[aliasPos] != '\0'))
      aliasPos++;
    filePos = 0;
    while (( filePos < 3 ) && (fileAlias[aliasPos] != '\0'))
    {
      tempChar = fileAlias[aliasPos++];
      if ((tempChar > ' ' && tempChar < ':' && tempChar!='.') || tempChar > '?')
        newDirEntry.ext[filePos++] = tempChar;
    }
    while (filePos < 3)
    {
      newDirEntry.ext[filePos++] = ' ';
    }

  //***
    // Scan Dir for free entry
    dirCluster = curWorkDirCluster;
    secOffset = 0;
    maxSectors = (dirCluster == FAT16_ROOT_DIR_CLUSTER ? (filesysData - filesysRootDir) :
  filesysSecPerClus);
```

```
  firstSector = (dirCluster == FAT16_ROOT_DIR_CLUSTER ? filesysRootDir :
FAT_ClustToSect(dirCluster));
  disc_ReadSector (firstSector + secOffset, (void*)dirEntries);

  dirEntryRemain = dirEntryLength;
  entryOffset = -1;

  // Search for a large enough space to fit in new directory entry
  do {
    entryOffset++;

    if (entryOffset == BYTE_PER_READ / sizeof (DIR_ENT)) {
      entryOffset = 0;
      secOffset++;
      if ((secOffset == filesysSecPerClus) && (dirCluster != FAT16_ROOT_DIR_CLUSTER))
      {
        secOffset = 0;
        if (FAT_NextCluster(dirCluster) == CLUSTER_EOF)
        {
          dirCluster = FAT_LinkFreeCluster(dirCluster);
          dirEntries[0].name[0] = FILE_LAST;
          dirEndFlag = true;
        }
        else
        {
          dirCluster = FAT_NextCluster(dirCluster);
        }
        firstSector = FAT_ClustToSect(dirCluster);
      }
      else if ((dirCluster == FAT16_ROOT_DIR_CLUSTER) && (secOffset == (filesysData -
filesysRootDir)))
      {
        return false; // Got to end of root dir - can't fit in more files
      }
      if(!dirEndFlag) //don't read if new cluster was just created
        disc_ReadSector (firstSector + secOffset, (void*)dirEntries);
    }

    if(dirEntryRemain==dirEntryLength) {
      //temp** points to first free entry
      tempDirCluster = dirCluster;
      tempSecOffset = secOffset;
      tempEntryOffset = entryOffset;
    }

    if (dirEntries[entryOffset].name[0] == FILE_FREE) {
      dirEntryRemain--;
    } else if(dirEntries[entryOffset].name[0] == FILE_LAST) {
      dirEntryRemain=0;
    } else {
      dirEntryRemain=dirEntryLength;
    }
  } while(dirEntryRemain > 0);

  // Modifying the last directory is a special case - have to erase following entries
  if (dirEntries[entryOffset].name[0] == FILE_LAST)
  {
    dirEndFlag = true;
  }

  // Recall first free entry
  dirCluster = tempDirCluster;
  secOffset = tempSecOffset;
  entryOffset = tempEntryOffset;
  dirEntryRemain = dirEntryLength;

  // Re-read in first sector that will be written to
  if (dirEndFlag && (entryOffset == 0)) {
    memset ((void*)dirEntries, FILE_LAST, BYTE_PER_READ);
  } else {
    disc_ReadSector (firstSector + secOffset, (void*)dirEntries);
  }

  // Add new directory entry
  entryOffset--;
  while (dirEntryRemain > 0)
  {
    // Move to next entry
    entryOffset++;
    if (entryOffset == BYTE_PER_READ / sizeof (DIR_ENT))
    {
      // Write out the current sector if we need to
      entryOffset = 0;
      if (dirEntryRemain < dirEntryLength) // Don't write out sector on first pass
      {
        disc_WriteSector (firstSector + secOffset, (void*)dirEntries);
      }
      secOffset++;
      if ((secOffset == filesysSecPerClus) && (dirCluster != FAT16_ROOT_DIR_CLUSTER))
      {
        secOffset = 0;
        if (FAT_NextCluster(dirCluster) == CLUSTER_EOF)
        {
          dirCluster = FAT_LinkFreeCluster(dirCluster);
          dirEntries[0].name[0] = FILE_LAST;
        }
        else
        {
          dirCluster = FAT_NextCluster(dirCluster);
        }
        firstSector = FAT_ClustToSect(dirCluster);
      }
      else if ((dirCluster == FAT16_ROOT_DIR_CLUSTER) && (secOffset == (filesysData -
filesysRootDir)))
      {
        return false; // Got to end of root dir - can't fit in more files
      }
      if (dirEndFlag)
      {
        memset ((void*)dirEntries, FILE_LAST, BYTE_PER_READ);
      } else {
        disc_ReadSector (firstSector + secOffset, (void*)dirEntries);
      }
    }

    // Generate LFN entries
    if (lfnPos >= 0)
    {
      lfnEntry.ordinal = (lfnPos + 1) | (dirEntryRemain == dirEntryLength ? LFN_END : 0);
      lfnEntry.char0 = filename [lfnPos * 13 + 0];
      lfnEntry.char1 = (filename [lfnPos * 13 + 1] == 0x01 ? 0xFFFF : filename [lfnPos * 13 +
1]);
      lfnEntry.char2 = (filename [lfnPos * 13 + 2] == 0x01 ? 0xFFFF : filename [lfnPos * 13 +
2]);
      lfnEntry.char3 = (filename [lfnPos * 13 + 3] == 0x01 ? 0xFFFF : filename [lfnPos * 13 +
3]);
      lfnEntry.char4 = (filename [lfnPos * 13 + 4] == 0x01 ? 0xFFFF : filename [lfnPos * 13 +
4]);
      lfnEntry.char5 = (filename [lfnPos * 13 + 5] == 0x01 ? 0xFFFF : filename [lfnPos * 13 +
5]);
      lfnEntry.char6 = (filename [lfnPos * 13 + 6] == 0x01 ? 0xFFFF : filename [lfnPos * 13 +
6]);
      lfnEntry.char7 = (filename [lfnPos * 13 + 7] == 0x01 ? 0xFFFF : filename [lfnPos * 13 +
7]);
      lfnEntry.char8 = (filename [lfnPos * 13 + 8] == 0x01 ? 0xFFFF : filename [lfnPos * 13 +
8]);
      lfnEntry.char9 = (filename [lfnPos * 13 + 9] == 0x01 ? 0xFFFF : filename [lfnPos * 13 +
9]);
      lfnEntry.char10 = (filename [lfnPos * 13 + 10] == 0x01 ? 0xFFFF : filename [lfnPos * 13 +
10]);
      lfnEntry.char11 = (filename [lfnPos * 13 + 11] == 0x01 ? 0xFFFF : filename [lfnPos * 13 +
11]);
```

```
      lfnEntry.char12 = (filename [lfnPos * 13 + 12] == 0x01 ? 0xFFFF : filename [lfnPos * 13 +
12]);
      lfnEntry.checkSum = chkSum;
      lfnEntry.flag = ATTRIB_LFN;
      lfnEntry.reserved1 = 0;
      lfnEntry.reserved2 = 0;

      *((DIR_ENT_LFN*)&dirEntries[entryOffset]) = lfnEntry;
      lfnPos --;
      lfnEntry.ordinal = 0;
    } // end writing long filename entries
    else
    {
      dirEntries[entryOffset] = newDirEntry;
      if (dirEndFlag && (entryOffset < (BYTE_PER_READ / sizeof (DIR_ENT))) )
        dirEntries[entryOffset+1].name[0] = FILE_LAST;
    }

    dirEntryRemain--;
  }

  // Write directory back to disk
  disc_WriteSector (firstSector + secOffset, (void*)dirEntries);

  // Change back to Working DIR
  curWorkDirCluster = oldWorkDirCluster;

  return true;
}
#endif

/*-----------------------------------------------------------------
FAT_FindNextFile
Gets the name of the next directory entry
  (can be a file or subdirectory)
char* filename: OUT filename, must be at least 13 chars long
FILE_TYPE return: OUT returns FT_NONE if failed,
  FT_FILE if it found a file and FT_DIR if it found a directory
-----------------------------------------------------------------*/
FILE_TYPE _FAT_FindNextFile(char* filename)
{
  // Get the next directory entry
  DIR_ENT file;

  file = FAT_GetDirEntry (curWorkDirCluster, 1, SEEK_CUR);

  if (file.name[0] == FILE_FREE)
  {
    return FT_NONE; // Did not find a file
  }

  // Get the filename
  if (filename != NULL)
    FAT_GetFilename (file, filename);

  if ((file.attrib & ATTRIB_DIR) != 0)
  {
    return FT_DIR;  // Found a directory
  }
  else
  {
    return FT_FILE; // Found a file
  }
}
FILE_TYPE FAT_FindNextFile(char* filename) {
  FILE_TYPE ret;
  FAT_lock();
  ret=_FAT_FindNextFile(filename);
  FAT_unlock();
  return ret;
}
```

```
/*-----------------------------------------------------------------
FAT_FindFirstFile
Gets the name of the first directory entry and resets the count
  (can be a file or subdirectory)
char* filename: OUT filename, must be at least 13 chars long
FILE_TYPE return: OUT returns FT_NONE if failed,
  FT_FILE if it found a file and FT_DIR if it found a directory
-----------------------------------------------------------------*/
FILE_TYPE _FAT_FindFirstFile(char* filename)
{
  // Get the first directory entry
  DIR_ENT file;

  file = FAT_GetDirEntry (curWorkDirCluster, 1, SEEK_SET);

  if (file.name[0] == FILE_FREE)
  {
    return FT_NONE; // Did not find a file
  }

  // Get the filename
  if (filename != NULL)
    FAT_GetFilename (file, filename);

  if ((file.attrib & ATTRIB_DIR) != 0)
  {
    return FT_DIR;  // Found a directory
  }
  else
  {
    return FT_FILE; // Found a file
  }
}
FILE_TYPE FAT_FindFirstFile(char* filename) {
  FILE_TYPE ret;
  FAT_lock();
  ret=_FAT_FindFirstFile(filename);
  FAT_unlock();
  return ret;
}

/*-----------------------------------------------------------------
FAT_FindFirstFileLFN
Gets the long file name of the first directory entry and resets
  the count (can be a file or subdirectory)
char* lfn: OUT long file name, must be at least 256 chars long
FILE_TYPE return: OUT returns FT_NONE if failed,
  FT_FILE if it found a file and FT_DIR if it found a directory
-----------------------------------------------------------------*/
FILE_TYPE FAT_FindFirstFileLFN(char* lfn)
{
  FILE_TYPE type;
  type = FAT_FindFirstFile(NULL);
  FAT_GetLongFilename (lfn);
  return type;
}

/*-----------------------------------------------------------------
FAT_FindNextFileLFN
Gets the long file name of the next directory entry
  (can be a file or subdirectory)
char* lfn: OUT long file name, must be at least 256 chars long
FILE_TYPE return: OUT returns FT_NONE if failed,
  FT_FILE if it found a file and FT_DIR if it found a directory
-----------------------------------------------------------------*/
FILE_TYPE FAT_FindNextFileLFN(char* lfn)
{
  FILE_TYPE type;
  type = FAT_FindNextFile(NULL);
  FAT_GetLongFilename (lfn);
  return type;
}


/*-----------------------------------------------------------------
FAT_FileExists
Returns the type of file 
char* filename: IN filename of the file to look for
FILE_TYPE return: OUT returns FT_NONE if there is now file with 
  that name, FT_FILE if it is a file and FT_DIR if it is a directory
-----------------------------------------------------------------*/
FILE_TYPE FAT_FileExists(const char* filename)
{
    DIR_ENT dirEntry;
    // Get the dirEntry for the path specified
    dirEntry = FAT_DirEntFromPath (filename);

    if (dirEntry.name[0] == FILE_FREE)
    {
        return FT_NONE;
    }
    else if (dirEntry.attrib & ATTRIB_DIR)
    {
        return FT_DIR;
    }
    else
    {
         return FT_FILE;
    }
}

/*-----------------------------------------------------------------
FAT_GetFileSystemType
FS_TYPE return: OUT returns the current file system type
-----------------------------------------------------------------*/
FS_TYPE FAT_GetFileSystemType (void)
{
  return filesysType;
}

/*-----------------------------------------------------------------
FAT_GetFileSystemTotalSize
u32 return: OUT returns the total disk space (used + free)
-----------------------------------------------------------------*/
u32 FAT_GetFileSystemTotalSize (void)
{
  return filesysTotalSize;
}



/*-----------------------------------------------------------------
FAT_chdir
Changes the current working directory
const char* path: IN null terminated string of directory separated by 
  forward slashes, / is root
bool return: OUT returns true if successful
-----------------------------------------------------------------*/
bool FAT_chdir (const char* path)
{
//if locks are ever added, make sure FAT_remove is fixed up too (calls chdir while locked)
  DIR_ENT dir;
  if (path[0] == '/' && path[1] == '\0')
  {
    curWorkDirCluster = filesysRootDirClus;
    return true;
  }
  if (path[0] == '\0')// Return true if changing relative to nothing
  {
    return true;
  }
  dir = FAT_DirEntFromPath (path);

  if (((dir.attrib & ATTRIB_DIR) == ATTRIB_DIR) && (dir.name[0] != FILE_FREE))
  {
    // Change directory
    curWorkDirCluster = dir.startCluster | (dir.startClusterHigh << 16);

    // Move to correct cluster for root directory
    if (curWorkDirCluster == FAT16_ROOT_DIR_CLUSTER)
    {
      curWorkDirCluster = filesysRootDirClus;
    }

    // Reset file position in directory
    wrkDirCluster = curWorkDirCluster;
    wrkDirSector = 0;
    wrkDirOffset = -1;
    return true;
  }
  else
  {
    // Couldn't change directory - wrong path specified
    return false;
  }
}

/*-----------------------------------------------------------------
FAT_fopen(filename, mode)
Opens a file
const char* path: IN null terminated string of filename and path
  separated by forward slashes, / is root
const char* mode: IN mode to open file in
  Supported modes: "r", "r+", "w", "w+", "a", "a+", don't use
  "b" or "t" in any mode, as all files are openned in binary mode
FAT_FILE* return: OUT handle to open file, returns NULL if the file
  couldn't be openned
-----------------------------------------------------------------*/
int FAT_lasterr;

FAT_FILE* _FAT_fopen(const char* path, const char* mode)
{
  int fileNum;
  FAT_FILE* file;
  DIR_ENT dirEntry;
#ifdef CAN_WRITE_TO_DISC
  u32 startCluster;
  int clusCount;
#endif

  char* pchTemp;
  // Check that a valid mode was specified
  pchTemp = strpbrk ( mode, "rRwWaA" );
  if (pchTemp == NULL)
  {
    FAT_lasterr=BADMODE;
    return NULL;
  }
  if (strpbrk ( pchTemp+1, "rRwWaA" ) != NULL)
  {
    FAT_lasterr=BADMODE;
    return NULL;
  }

  // Get the dirEntry for the path specified
  dirEntry = FAT_DirEntFromPath (path);

  // Check that it is not a directory
  if (dirEntry.attrib & ATTRIB_DIR)
  {
    FAT_lasterr=CANT_OPEN_DIR;
    return NULL;
  }

#ifdef CAN_WRITE_TO_DISC
  // Check that it is not a read only file being openned in a writing mode
  if ( (strpbrk(mode, "wWaA+") != NULL) && (dirEntry.attrib & ATTRIB_RO))
  {
    FAT_lasterr=READONLY;
    return NULL;
  }
#else
  if ( (strpbrk(mode, "wWaA+") != NULL))
  {
    FAT_lasterr=READONLY;
    return NULL;
  }
#endif

  // Find a free file buffer
  for (fileNum = 0; (fileNum < MAX_FILES_OPEN) && (openFiles[fileNum].inUse == true); fileNum++);

  if (fileNum == MAX_FILES_OPEN) // No free files
  {
    FAT_lasterr=MAXOPEN;
    return NULL;
  }

  file = &openFiles[fileNum];
  // Remember where directory entry was
  file->dirEntSector = (wrkDirCluster == FAT16_ROOT_DIR_CLUSTER ? filesysRootDir :
FAT_ClustToSect(wrkDirCluster)) + wrkDirSector;
  file->dirEntOffset = wrkDirOffset;

  if ( strpbrk(mode, "rR") != NULL )  //(ucase(mode[0]) == 'R')
  {
    if (dirEntry.name[0] == FILE_FREE)  // File must exist
    {
      FAT_lasterr=DOESNT_EXIST;
      return NULL;
    }

    file->read = true;
#ifdef CAN_WRITE_TO_DISC
    file->write = ( strchr(mode, '+') != NULL ); //(mode[1] == '+');
#else
    file->write = false;
#endif
    file->append = false;

    // Store information about position within the file, for use
    // by FAT_fread, FAT_fseek, etc.
    file->firstCluster = dirEntry.startCluster | (dirEntry.startClusterHigh << 16);

#ifdef CAN_WRITE_TO_DISC
    // Check if file is openned for random. If it is, and currently has no cluster, one must be
    // assigned to it.
    if (file->write && file->firstCluster == CLUSTER_FREE)
    {
      file->firstCluster = FAT_LinkFreeCluster (CLUSTER_FREE);
      if (file->firstCluster == CLUSTER_FREE) // Couldn't get a free cluster
      {
        FAT_lasterr=NO_FREE_CLUSTER;
        return NULL;
      }

      // Store cluster position into the directory entry
      dirEntry.startCluster = (file->firstCluster & 0xFFFF);
      dirEntry.startClusterHigh = ((file->firstCluster >> 16) & 0xFFFF);
      disc_ReadSector (file->dirEntSector, globalBuffer);
      ((DIR_ENT*) globalBuffer)[file->dirEntOffset] = dirEntry;
      disc_WriteSector (file->dirEntSector, globalBuffer);
    }
#endif

    file->length = dirEntry.fileSize;
    file->curPos = 0;
    file->curClus = dirEntry.startCluster | (dirEntry.startClusterHigh << 16);
    file->curSect = 0;
    file->curByte = 0;

    // Not appending
    file->appByte = 0;
    file->appClus = 0;
    file->appSect = 0;

    disc_ReadSector( FAT_ClustToSect( file->curClus), file->readBuffer);
    file->inUse = true; // We're using this file now

    return file;
  } // mode "r"

#ifdef CAN_WRITE_TO_DISC
  if ( strpbrk(mode, "wW") != NULL ) // (ucase(mode[0]) == 'W')
  {
    if (dirEntry.name[0] == FILE_FREE)  // Create file if it doesn't exist
    {
      dirEntry.attrib = ATTRIB_ARCH;
      dirEntry.reserved = 0;

      // Time and date set to system time and date
      dirEntry.cTime_ms = 0;
      dirEntry.cTime = getRTCtoFileTime();
      dirEntry.cDate = getRTCtoFileDate();
      dirEntry.aDate = getRTCtoFileDate();
      dirEntry.mTime = getRTCtoFileTime();
      dirEntry.mDate = getRTCtoFileDate();
    }
    else // Already a file entry
    {
      // Free any clusters used
      FAT_ClearLinks (dirEntry.startCluster | (dirEntry.startClusterHigh << 16));
    }

    // Get a cluster to use
    startCluster = FAT_LinkFreeCluster (CLUSTER_FREE);
    if (startCluster == CLUSTER_FREE) // Couldn't get a free cluster
    {
      FAT_lasterr=NO_FREE_CLUSTER;
      return NULL;
    }

    // Store cluster position into the directory entry
    dirEntry.startCluster = (startCluster & 0xFFFF);
    dirEntry.startClusterHigh = ((startCluster >> 16) & 0xFFFF);

    // The file has no data in it - its over written so should be empty
    dirEntry.fileSize = 0;

    if (dirEntry.name[0] == FILE_FREE)  // No file
    {
      // Have to create a new entry
      if(!FAT_AddDirEntry (path, dirEntry))
      {
        FAT_lasterr=ADDDIR_FAILED;
        return NULL;
      }
      // Get the newly created dirEntry
      dirEntry = FAT_DirEntFromPath (path);

      // Remember where directory entry was
      file->dirEntSector = (wrkDirCluster == FAT16_ROOT_DIR_CLUSTER ? filesysRootDir :
FAT_ClustToSect(wrkDirCluster)) + wrkDirSector;
```

```
      file->dirEntOffset = wrkDirOffset;
    }
    else // Already a file
    {
      // Just modify the old entry
      disc_ReadSector (file->dirEntSector, globalBuffer);
      ((DIR_ENT*) globalBuffer)[file->dirEntOffset] = dirEntry;
      disc_WriteSector (file->dirEntSector, globalBuffer);
    }


    // Now that file is created, open it
    file->read = ( strchr(mode, '+') != NULL ); //(mode[1] == '+');
    file->write = true;
    file->append = false;

    // Store information about position within the file, for use
    // by FAT_fread, FAT_fseek, etc.
    file->firstCluster = startCluster;
    file->length = 0; // Should always have 0 bytes if openning in "w" mode
    file->curPos = 0;
    file->curClus = startCluster;
    file->curSect = 0;
    file->curByte = 0;

    // Not appending
    file->appByte = 0;
    file->appClus = 0;
    file->appSect = 0;

    // Empty file, so empty read buffer
    memset (file->readBuffer, 0, BYTE_PER_READ);
    file->inUse = true; // We're using this file now

    return file;
  }

  if ( strpbrk(mode, "aA") != NULL ) // (ucase(mode[0]) == 'A')
  {
    if (dirEntry.name[0] == FILE_FREE)  // Create file if it doesn't exist
    {
      dirEntry.attrib = ATTRIB_ARCH;
      dirEntry.reserved = 0;

      // Time and date set to system time and date
      dirEntry.cTime_ms = 0;
      dirEntry.cTime = getRTCtoFileTime();
      dirEntry.cDate = getRTCtoFileDate();
      dirEntry.aDate = getRTCtoFileDate();
      dirEntry.mTime = getRTCtoFileTime();
      dirEntry.mDate = getRTCtoFileDate();

      // The file has no data in it
      dirEntry.fileSize = 0;

      // Get a cluster to use
      startCluster = FAT_LinkFreeCluster (CLUSTER_FREE);
      if (startCluster == CLUSTER_FREE) // Couldn't get a free cluster
      {
        FAT_lasterr=NO_FREE_CLUSTER;
        return NULL;
      }
      dirEntry.startCluster = (startCluster & 0xFFFF);
      dirEntry.startClusterHigh = ((startCluster >> 16) & 0xFFFF);

      if(!FAT_AddDirEntry (path, dirEntry)) {
        FAT_lasterr=ADDDIR_FAILED;
        return NULL;
      }

      // Get the newly created dirEntry
      dirEntry = FAT_DirEntFromPath (path);

      // Store append cluster
      file->appClus = startCluster;

      // Remember where directory entry was
      file->dirEntSector = (wrkDirCluster == FAT16_ROOT_DIR_CLUSTER ? filesysRootDir :
FAT_ClustToSect(wrkDirCluster)) + wrkDirSector;
      file->dirEntOffset = wrkDirOffset;
    }
    else // File already exists - reuse the old directory entry
    {
      startCluster = dirEntry.startCluster | (dirEntry.startClusterHigh << 16);
      // If it currently has no cluster, one must be assigned to it.
      if (startCluster == CLUSTER_FREE)
      {
        file->firstCluster = FAT_LinkFreeCluster (CLUSTER_FREE);
        if (file->firstCluster == CLUSTER_FREE)  // Couldn't get a free cluster
        {
          FAT_lasterr=NO_FREE_CLUSTER;
          return NULL;
        }

        // Store cluster position into the directory entry
        dirEntry.startCluster = (file->firstCluster & 0xFFFF);
        dirEntry.startClusterHigh = ((file->firstCluster >> 16) & 0xFFFF);
        disc_ReadSector (file->dirEntSector, globalBuffer);
        ((DIR_ENT*) globalBuffer)[file->dirEntOffset] = dirEntry;
        disc_WriteSector (file->dirEntSector, globalBuffer);

        // Store append cluster
        file->appClus = startCluster;

      } else {

        // Follow cluster list until last one is found
        clusCount = dirEntry.fileSize / filesysBytePerClus;
        file->appClus = startCluster;
        while ((clusCount--) && (FAT_NextCluster (file->appClus) != CLUSTER_FREE) &&
(FAT_NextCluster (file->appClus) != CLUSTER_EOF))
        {
          file->appClus = FAT_NextCluster (file->appClus);
        }
        if (clusCount >= 0) // Check if ran out of clusters
        {
          // Set flag to allocate new cluster when needed
          file->appSect = filesysSecPerClus;
          file->appByte = 0;
        }
      }
    }

    // Now that file is created, open it
    file->read = ( strchr(mode, '+') != NULL );
    file->write = false;
    file->append = true;

    // Calculate the sector and byte of the current position,
    // and store them
    file->appSect = (dirEntry.fileSize % filesysBytePerClus) / BYTE_PER_READ;
    file->appByte = dirEntry.fileSize % BYTE_PER_READ;

    // Store information about position within the file, for use
    // by FAT_fread, FAT_fseek, etc.
    file->firstCluster = startCluster;
    file->length = dirEntry.fileSize;
    file->curPos = dirEntry.fileSize;
    file->curClus = file->appClus;
    file->curSect = file->appSect;
    file->curByte = file->appByte;
```

```
        // Read into buffer
        disc_ReadSector( FAT_ClustToSect(file->curClus) + file->curSect, file->readBuffer);
        file->inUse = true; // We're using this file now
        return file;
    }
#endif

    // Can only reach here if a bad mode was specified
    FAT_lasterr=BADMODE;
    return NULL;
}
//wrapper...
FAT_FILE* FAT_fopen(const char* path, const char* mode) {
    FAT_FILE *f;
    FAT_lock();
    f=_FAT_fopen(path,mode);
    FAT_unlock();
    return f;
}

/*-----------------------------------------------------------------
FAT_fclose(file)
Closes a file
FAT_FILE* file: IN handle of the file to close
bool return OUT: true if successful, false if not
-----------------------------------------------------------------*/
bool FAT_fclose (FAT_FILE* file)
{
    // Clear memory used by file information
    if ((file != NULL) && (file->inUse == true))
    {
#ifdef CAN_WRITE_TO_DISC
        if (file->write || file->append)
        {
            FAT_lock();

            // Write new length, time and date back to directory entry
            disc_ReadSector (file->dirEntSector, globalBuffer);

            ((DIR_ENT*)globalBuffer)[file->dirEntOffset].fileSize = file->length;
            ((DIR_ENT*)globalBuffer)[file->dirEntOffset].mTime = getRTCtoFileTime();
            ((DIR_ENT*)globalBuffer)[file->dirEntOffset].mDate = getRTCtoFileDate();
            ((DIR_ENT*)globalBuffer)[file->dirEntOffset].aDate = getRTCtoFileDate();

            disc_WriteSector (file->dirEntSector, globalBuffer);

            // Flush any sectors in disc cache
            disc_CacheFlush();

            FAT_unlock();
        }
#endif
        file->inUse = false;
        return true;
    }
    else
    {
        return false;
    }
}

/*-----------------------------------------------------------------
FAT_ftell(file)
Returns the current position in a file
FAT_FILE* file: IN handle of an open file
u32 OUT: Current position
-----------------------------------------------------------------*/
u32 FAT_ftell (FAT_FILE* file)
{
    // Return the position as specified in the FAT_FILE structure
    if ((file != NULL) && (file->inUse == true))
```

```
  {
    return file->curPos;
  }
  else
  {
    // Return -1 if no file was given
    return -1;
  }
}

/*-----------------------------------------------------------------
FAT_fseek(file, offset, origin)
Seeks to specified byte position in file
FAT_FILE* file: IN handle of an open file
s32 offset IN: position to seek to, relative to origin
int origin IN: origin to seek from
int OUT: Returns 0 if successful, -1 if not
-----------------------------------------------------------------*/
int FAT_fseek(FAT_FILE* file, s32 offset, int origin)
{
  u32 cluster, nextCluster;
  int clusCount;
  u32 position;
  u32 curPos;

  if ((file == NULL) || (file->inUse == false))  // invalid file
  {
    return -1;
  }

  // Can't seek in append only mode
  if (!file->read && !file->write)
  {
    return -1;
  }

  curPos = file->curPos;

  switch (origin)
  {
  case SEEK_SET:
    if (offset >= 0)
    {
      position = offset;
    } else {
      // Tried to seek before start of file
      position = 0;
    }
    break;
  case SEEK_CUR:
    if (offset >= 0)
    {
      position = curPos + offset;
    }
    else if ( (u32)(offset * -1) >= curPos )
    {
      // Tried to seek before start of file
      position = 0;
    }
    else
    {
      // Using u32 to maintain 32 bits of accuracy
      position = curPos - (u32)(offset * -1);
    }
    break;
  case SEEK_END:
    if (offset >= 0)
    {
      // Seeking to end of file
      position = file->length; // Fixed thanks to MoonLight
    }
    else if ( (u32)(offset * -1) >= file->length )
    {
      // Tried to seek before start of file
      position = 0;
    }
    else
    {
      // Using u32 to maintain 32 bits of accuracy
      position = file->length - (u32)(offset * -1);
    }
    break;
  default:
    return -1;
  }

  if (position > file->length)
  {
    // Tried to go past end of file
    position = file->length;
  }

  // Save position
  file->curPos = position;


  // Calculate where the correct cluster is
  if (position > curPos)
  {
    clusCount = (position - curPos + (file->curSect * filesysBytePerSec) + file->curByte) /
filesysBytePerClus; // Fixed thanks to AgentQ
    cluster = file->curClus;
  } else {
    clusCount = position / filesysBytePerClus;
    cluster = file->firstCluster;
  }

  // Calculate the sector and byte of the current position,
  // and store them
  file->curSect = (position % filesysBytePerClus) / BYTE_PER_READ;
  file->curByte = position % BYTE_PER_READ;

  FAT_lock();

  // Follow cluster list until desired one is found
  if (clusCount > 0)  // Only look at next cluster if need to
  {
    nextCluster = FAT_NextCluster (cluster);
  } else {
    nextCluster = cluster;
  }
  while ((clusCount--) && (nextCluster != CLUSTER_FREE) && (nextCluster != CLUSTER_EOF))
  {
    cluster = nextCluster;
    nextCluster = FAT_NextCluster (cluster);
  }
  // Check if ran out of clusters, and the file is being written to
  if ((clusCount >= 0) && (file->write || file->append))
  {
    // Set flag to allocate a new cluster
    file->curSect = filesysSecPerClus;
    file->curByte = 0;
  }
  file->curClus = cluster;

  // Reload sector buffer for new position in file, if it is a different sector
  if ((curPos ^ position) >= BYTE_PER_READ)
  {
    disc_ReadSector( file->curSect + FAT_ClustToSect(file->curClus), file->readBuffer);
  }

  FAT_unlock();
  return 0;
}

/*-----------------------------------------------------------------
FAT_fread(buffer, size, count, file)
Reads in size * count bytes into buffer from file, starting
  from current position. It then sets the current position to the
  byte after the last byte read. If it reaches the end of file
  before filling the buffer then it stops reading.
void* buffer OUT: Pointer to buffer to fill. Should be at least as
  big as the number of bytes required
u32 size IN: size of each item to read
u32 count IN: number of items to read
FAT_FILE* file IN: Handle of an open file
u32 OUT: returns the actual number of bytes read
-----------------------------------------------------------------*/
u32 FAT_fread (void* buffer, u32 size, u32 count, FAT_FILE* file)
{
  int curByte;
  int curSect;
  u32 curClus;
  u32 tempNextCluster;

  int tempVar;

  char* data = (char*)buffer;

  u32 length = size * count;
  u32 remain;

  bool flagNoError = true;

  // Can't read non-existant files
  if ((file == NULL) || (file->inUse == false) || size == 0 || count == 0 || buffer == NULL)
    return 0;

  // Can only read files openned for reading
  if (!file->read)
    return 0;

  FAT_lock();

  // Don't read past end of file
  if (length + file->curPos > file->length)
    length = file->length - file->curPos;

  remain = length;

  curByte = file->curByte;
  curSect = file->curSect;
  curClus = file->curClus;

  // Align to sector
  tempVar = BYTE_PER_READ - curByte;
  if (tempVar > remain)
    tempVar = remain;

  if ((tempVar < BYTE_PER_READ) && flagNoError)
  {
    memcpy(data, &(file->readBuffer[curByte]), tempVar);
    remain -= tempVar;
    data += tempVar;

    curByte += tempVar;
    if (curByte >= BYTE_PER_READ)
    {
      curByte = 0;
      curSect++;
    }
  }
```

```
// align to cluster
// tempVar is number of sectors to read
if (remain > (filesysSecPerClus - curSect) * BYTE_PER_READ)
{
  tempVar = filesysSecPerClus - curSect;
} else {
  tempVar = remain / BYTE_PER_READ;
}

if ((tempVar > 0) && flagNoError)
{
  disc_ReadSectors ( curSect + FAT_ClustToSect(curClus), tempVar, data);
  data += tempVar * BYTE_PER_READ;
  remain -= tempVar * BYTE_PER_READ;

  curSect += tempVar;
}

// Move onto next cluster
// It should get to here without reading anything if a cluster is due to be allocated
if (curSect >= filesysSecPerClus)
{
  tempNextCluster = FAT_NextCluster(curClus);
  if ((remain == 0) && (tempNextCluster == CLUSTER_EOF))
  {
    curSect = filesysSecPerClus;
  } else {
    curSect = 0;
    curClus = tempNextCluster;
    if (curClus == CLUSTER_FREE)
    {
      flagNoError = false;
    }
  }
}

// Read in whole clusters
while ((remain >= filesysBytePerClus) && flagNoError)
{
  disc_ReadSectors (FAT_ClustToSect(curClus), filesysSecPerClus, data);
  data += filesysBytePerClus;
  remain -= filesysBytePerClus;

  // Advance to next cluster
  tempNextCluster = FAT_NextCluster(curClus);

  if ((remain == 0) && (tempNextCluster == CLUSTER_EOF))
  {
    curSect = filesysSecPerClus;
  } else {
    curSect = 0;
    curClus = tempNextCluster;
    if (curClus == CLUSTER_FREE)
    {
      flagNoError = false;
    }
  }
}

// Read remaining sectors
tempVar = remain / BYTE_PER_READ; // Number of sectors left
if ((tempVar > 0) && flagNoError)
{
  disc_ReadSectors (FAT_ClustToSect(curClus), tempVar, data);
  data += tempVar * BYTE_PER_READ;
  remain -= tempVar * BYTE_PER_READ;
  curSect += tempVar;
}

// Last remaining sector
```

```
  // Check if sector wanted is different to the one started with
  if ( ((file->curByte + length) >= BYTE_PER_READ) && flagNoError)
  {
    disc_ReadSector( curSect + FAT_ClustToSect( curClus), file->readBuffer);
    if (remain > 0)
    {
      memcpy(data, file->readBuffer, remain);
      curByte += remain;
      remain = 0;
    }
  }

  // Length read is the wanted length minus the stuff not read
  length = length - remain;

  // Update file information
  file->curByte = curByte;
  file->curSect = curSect;
  file->curClus = curClus;
  file->curPos = file->curPos + length;

  FAT_unlock();

  return length;
}

#ifdef CAN_WRITE_TO_DISC
/*-----------------------------------------------------------------
FAT_fwrite(buffer, size, count, file)
Writes size * count bytes into file from buffer, starting
  from current position. It then sets the current position to the
  byte after the last byte written. If the file was openned in
  append mode it always writes to the end of the file.
const void* buffer IN: Pointer to buffer containing data. Should be
  at least as big as the number of bytes to be written.
u32 size IN: size of each item to write
u32 count IN: number of items to write
FAT_FILE* file IN: Handle of an open file
u32 OUT: returns the actual number of bytes written
-----------------------------------------------------------------*/
u32 FAT_fwrite (const void* buffer, u32 size, u32 count, FAT_FILE* file)
{
  int curByte;
  int curSect;
  u32 curClus;

  u32 tempNextCluster;
  int tempVar;
  u32 length = size * count;
  u32 remain = length;
  char* data = (char*)buffer;

  char* writeBuffer;

  bool flagNoError = true;
  bool flagAppending = false;

  if ((file == NULL) || (file->inUse == false) || length == 0 || buffer == NULL)
    return 0;

  if (file->write)
  {
    FAT_lock();

    // Write at current read pointer
    curByte = file->curByte;
    curSect = file->curSect;
    curClus = file->curClus;

    // Use read buffer as write buffer
    writeBuffer = file->readBuffer;
    // If it is writing past the current end of file, set appending flag
    if (length + file->curPos > file->length)
    {
      flagAppending = true;
    }
  }
  else if (file->append)
  {
    FAT_lock();

    // Write at end of file
    curByte = file->appByte;
    curSect = file->appSect;
    curClus = file->appClus;
    flagAppending = true;

    // Use global buffer as write buffer, don't touch read buffer
    writeBuffer = (char*)globalBuffer;
    disc_ReadSector(curSect + FAT_ClustToSect(curClus), writeBuffer);
  }
  else
  {
    return 0;
  }

  // Move onto next cluster if needed
  if (curSect >= filesysSecPerClus)
  {
    curSect = 0;
    tempNextCluster = FAT_NextCluster(curClus);
    if ((tempNextCluster == CLUSTER_EOF) || (tempNextCluster == CLUSTER_FREE))
    {
      // Ran out of clusters so get a new one
      curClus = FAT_LinkFreeCluster(curClus);
      if (curClus == CLUSTER_FREE) // Couldn't get a cluster, so abort
      {
        flagNoError = false;
      }
      memset(writeBuffer, 0, BYTE_PER_READ);
    } else {
      curClus = tempNextCluster;
      disc_ReadSector( FAT_ClustToSect( curClus), writeBuffer);
    }
  }

  // Align to sector
  tempVar = BYTE_PER_READ - curByte;
  if (tempVar > remain)
    tempVar = remain;

  if ((tempVar < BYTE_PER_READ) && flagNoError)
  {
    memcpy(&(writeBuffer[curByte]), data, tempVar);
    remain -= tempVar;
    data += tempVar;
    curByte += tempVar;

    // Write buffer back to disk
    disc_WriteSector (curSect + FAT_ClustToSect(curClus), writeBuffer);

    // Move onto next sector
    if (curByte >= BYTE_PER_READ)
    {
      curByte = 0;
      curSect++;
    }
  }

  // Align to cluster
  // tempVar is number of sectors to write
```

```
if (remain > (filesysSecPerClus - curSect) * BYTE_PER_READ)
{
  tempVar = filesysSecPerClus - curSect;
} else {
  tempVar = remain / BYTE_PER_READ;
}

if ((tempVar > 0) && flagNoError)
{
  disc_WriteSectors ( curSect + FAT_ClustToSect(curClus), tempVar, data);
  data += tempVar * BYTE_PER_READ;
  remain -= tempVar * BYTE_PER_READ;
  curSect += tempVar;
}

if (((curSect >= filesysSecPerClus) && flagNoError) && (remain > 0))
{
  curSect = 0;
  tempNextCluster = FAT_NextCluster(curClus);
  if ((tempNextCluster == CLUSTER_EOF) || (tempNextCluster == CLUSTER_FREE))
  {
    // Ran out of clusters so get a new one
    curClus = FAT_LinkFreeCluster(curClus);
    if (curClus == CLUSTER_FREE) // Couldn't get a cluster, so abort
    {
      flagNoError = false;
    }
  } else {
    curClus = tempNextCluster;
  }
}

// Write whole clusters
while ((remain >= filesysBytePerClus) && flagNoError)
{
  disc_WriteSectors (FAT_ClustToSect(curClus), filesysSecPerClus, data);
  data += filesysBytePerClus;
  remain -= filesysBytePerClus;
  if (remain > 0)
  {
    tempNextCluster = FAT_NextCluster(curClus);
    if ((tempNextCluster == CLUSTER_EOF) || (tempNextCluster == CLUSTER_FREE))
    {
      // Ran out of clusters so get a new one
      curClus = FAT_LinkFreeCluster(curClus);
      if (curClus == CLUSTER_FREE) // Couldn't get a cluster, so abort
      {
        flagNoError = false;
        break;
      }
    } else {
      curClus = tempNextCluster;
    }
  } else {
    // Allocate a new cluster when next writing the file
    curSect = filesysSecPerClus;
  }
}

// Write remaining sectors
tempVar = remain / BYTE_PER_READ; // Number of sectors left
if ((tempVar > 0) && flagNoError)
{
  disc_WriteSectors (FAT_ClustToSect(curClus), tempVar, data);
  data += tempVar * BYTE_PER_READ;
  remain -= tempVar * BYTE_PER_READ;
  curSect += tempVar;
}

// Last remaining sector
// Check if sector wanted is different to the one started with
```

```
  if ( (( (file->append ? file->appByte : file->curByte) + length) >= BYTE_PER_READ) &&
flagNoError)
  {
    if (flagAppending)
    {
      // Zero sector before using it
      memset (writeBuffer, 0, BYTE_PER_READ);
    } else {
      // Modify existing sector
      disc_ReadSector( curSect + FAT_ClustToSect( curClus), writeBuffer);
    }
    if (remain > 0) {
      memcpy(writeBuffer, data, remain);
      curByte += remain;
      remain = 0;
      disc_WriteSector( curSect + FAT_ClustToSect( curClus), writeBuffer);
    }
  }

  // Amount read is the originally requested amount minus stuff remaining
  length = length - remain;

  // Update file information
  if (file->write)  // Writing also shifts the read pointer
  {
    file->curByte = curByte;
    file->curSect = curSect;
    file->curClus = curClus;
    file->curPos = file->curPos + length;
    if (file->length < file->curPos)
    {
      file->length = file->curPos;
    }
  }
  else if (file->append) // Appending doesn't affect the read pointer
  {
    file->appByte = curByte;
    file->appSect = curSect;
    file->appClus = curClus;
    file->length = file->length + length;
  }

  FAT_unlock();
  return length;
}
#endif


/*-----------------------------------------------------------------
FAT_feof(file)
Returns true if the end of file has been reached
FAT_FILE* file IN: Handle of an open file
bool return OUT: true if EOF, false if not
-----------------------------------------------------------------*/
bool FAT_feof(FAT_FILE* file)
{
  if ((file == NULL) || (file->inUse == false))
    return true; // Return eof on invalid files

  return (file->length == file->curPos);
}


#ifdef CAN_WRITE_TO_DISC
/*-----------------------------------------------------------------
FAT_remove (path)
Deletes the file or empty directory sepecified in path
const char* path IN: Path of item to delete
int return OUT: zero if successful, non-zero if not
-----------------------------------------------------------------*/
int FAT_remove (const char* path)
```

```
{
  DIR_ENT dirEntry;
  u32 oldWorkDirCluster;
  char checkFilename[13];
  FILE_TYPE checkFiletype;

  FAT_lock();
  dirEntry = FAT_DirEntFromPath (path);

  if (dirEntry.name[0] == FILE_FREE) {
    FAT_unlock();
    return -1;
  }

  // Only delete directories if the directory is entry
  if (dirEntry.attrib & ATTRIB_DIR)
  {
    // Change to the directory temporarily
    oldWorkDirCluster = curWorkDirCluster;
    FAT_chdir(path);

    // Search for files or directories, excluding the . and .. entries
    checkFiletype = _FAT_FindFirstFile (checkFilename);
    while ((checkFilename[0] == '.')  && (checkFiletype != FT_NONE))
    {
      checkFiletype = _FAT_FindNextFile (checkFilename);
    }

    // Change back to working directory
    curWorkDirCluster = oldWorkDirCluster;

    // Check that the directory is empty
    if (checkFiletype != FT_NONE)
    {
      // Directory isn't empty
      FAT_unlock();
      return -1;
    }
  }

  // Refresh directory information
  dirEntry = FAT_DirEntFromPath (path);

  // Free any clusters used
  FAT_ClearLinks (dirEntry.startCluster | (dirEntry.startClusterHigh << 16));

  // Remove Directory entry
  disc_ReadSector ( (wrkDirCluster == FAT16_ROOT_DIR_CLUSTER ? filesysRootDir :
FAT_ClustToSect(wrkDirCluster)) + wrkDirSector , globalBuffer);
  ((DIR_ENT*)globalBuffer)[wrkDirOffset].name[0] = FILE_FREE;
  disc_WriteSector ( (wrkDirCluster == FAT16_ROOT_DIR_CLUSTER ? filesysRootDir :
FAT_ClustToSect(wrkDirCluster)) + wrkDirSector , globalBuffer);

  // Flush any sectors in disc cache
  disc_CacheFlush();

  FAT_unlock();
  return 0;
}
#endif

#ifdef CAN_WRITE_TO_DISC
/*-----------------------------------------------------------------
FAT_mkdir (path)
Makes a new directory, so long as no other directory or file has
  the same name.
const char* path IN: Path and filename of directory to make
int return OUT: zero if successful, non-zero if not
-----------------------------------------------------------------*/
int FAT_mkdir (const char* path)
{
```

```
u32 newDirCluster;
u32 parentDirCluster;
DIR_ENT dirEntry;
DIR_ENT* entries = (DIR_ENT*)globalBuffer;
int i;

int pathPos, filePos;
char pathname[MAX_FILENAME_LENGTH];
u32 oldDirCluster;

if (FAT_FileExists(path) != FT_NONE)
{
  return -1; // File or directory exists with that name
}

// Find filename within path and change to that directory
oldDirCluster = curWorkDirCluster;
if (path[0] == '/')
{
  curWorkDirCluster = filesysRootDirClus;
}

pathPos = 0;
filePos = 0;

while (path[pathPos + filePos] != '\0')
{
  if (path[pathPos + filePos] == '/')
  {
    pathname[filePos] = '\0';
    if (FAT_chdir(pathname) == false)
    {
      curWorkDirCluster = oldDirCluster;
      return -1; // Couldn't change directory
    }
    pathPos += filePos + 1;
    filePos = 0;
  }
  pathname[filePos] = path[pathPos + filePos];
  filePos++;
}

// Now grab the parent directory's cluster
parentDirCluster = curWorkDirCluster;
curWorkDirCluster = oldDirCluster;

// Get a new cluster for the file
newDirCluster = FAT_LinkFreeCluster(CLUSTER_FREE);

if (newDirCluster == CLUSTER_FREE)
{
  return -1; // Couldn't get a new cluster for the directory
}
// Fill in directory entry's information
dirEntry.attrib = ATTRIB_DIR;
dirEntry.reserved = 0;
// Time and date set to system time and date
dirEntry.cTime_ms = 0;
dirEntry.cTime = getRTCtoFileTime();
dirEntry.cDate = getRTCtoFileDate();
dirEntry.aDate = getRTCtoFileDate();
dirEntry.mTime = getRTCtoFileTime();
dirEntry.mDate = getRTCtoFileDate();
// Store cluster position into the directory entry
dirEntry.startCluster = (newDirCluster & 0xFFFF);
dirEntry.startClusterHigh = ((newDirCluster >> 16) & 0xFFFF);
// The file has no data in it - its over written so should be empty
dirEntry.fileSize = 0;

if (FAT_AddDirEntry (path, dirEntry) == false)
{
    return -1; // Couldn't add the directory entry
  }

  // Create the new directory itself
  memset((void*)entries, FILE_LAST, BYTE_PER_READ);

  // Create . directory entry
  dirEntry.name[0] = '.';
  // Fill name and extension with spaces
  for (i = 1; i < 11; i++)
  {
    dirEntry.name[i] = ' ';
  }

  memcpy((void*)entries, (void*)&dirEntry, sizeof(dirEntry));

  // Create .. directory entry
  dirEntry.name[1] = '.';
  dirEntry.startCluster = (parentDirCluster & 0xFFFF);
  dirEntry.startClusterHigh = ((parentDirCluster >> 16) & 0xFFFF);

  memcpy((void*)&entries[1], (void*)&dirEntry, sizeof(dirEntry));

  // Write entry to disc
  disc_WriteSector(FAT_ClustToSect(newDirCluster), (void*)entries);

  // Flush any sectors in disc cache
  disc_CacheFlush();
  return 0;
}
#endif

/*-----------------------------------------------------------------
FAT_fgetc (handle)
Gets the next character in the file
FAT_FILE* file IN: Handle of open file
bool return OUT: character if successful, EOF if not
-----------------------------------------------------------------*/
char FAT_fgetc (FAT_FILE* file)
{
  char c;
  return (FAT_fread(&c, 1, 1, file) == 1) ? c : EOF;
}

#ifdef CAN_WRITE_TO_DISC
/*-----------------------------------------------------------------
FAT_fputc (character, handle)
Writes the given character into the file
char c IN: Character to be written
FAT_FILE* file IN: Handle of open file
bool return OUT: character if successful, EOF if not
-----------------------------------------------------------------*/
char FAT_fputc (char c, FAT_FILE* file)
{
  return (FAT_fwrite(&c, 1, 1, file) == 1) ? c : EOF;
}
#endif

/*-----------------------------------------------------------------
FAT_fgets (char *tgtBuffer, int num, FAT_FILE* file)
Gets a up to num bytes from file, stopping at the first
 newline.

CAUTION: does not do strictly streaming. I.e. it's
 reading more then needed bytes and seeking back.
 shouldn't matter for random access

char *tgtBuffer OUT: buffer to write to
int num IN: size of target buffer
FAT_FILE* file IN: Handle of open file
bool return OUT: character if successful, EOF if not
```

```
  Written by MightyMax
  Modified by Chishm - 2005-11-17
  * Added check for unix style text files
  * Removed seek when no newline is found, since it isn't necessary
-------------------------------------------------------------------*/
char *FAT_fgets(char *tgtBuffer, int num, FAT_FILE* file)
{
  u32 curPos;
  u32 readLength;
  char *returnChar;

  // invalid filehandle
  if (file == NULL)
  {
    return NULL ;
  }

  // end of file
  if (FAT_feof(file)==true)
  {
    return NULL ;
  }

  // save current position
  curPos = FAT_ftell(file);

  // read the full buffer (max string chars is num-1 and one end of string \0
  readLength = FAT_fread(tgtBuffer,1,num-1,file) ;

  // mark least possible end of string
  tgtBuffer[readLength] = '\0' ;

  if (readLength==0) {
    // return error
    return NULL ;
  }

  // get position of first return '\r'
  returnChar = strchr(tgtBuffer,'\r');

  // if no return is found, search for a newline
  if (returnChar == NULL)
  {
    returnChar = strchr(tgtBuffer,'\n');
  }

  // Mark the return, if existant, as end of line/string
  if (returnChar!=NULL) {
    *returnChar++ = 0 ;
    if (*returnChar=='\n') { // catch newline too when jumping over the end
      // return to location after \r\n (strlen+2)
      FAT_fseek(file,curPos+strlen(tgtBuffer)+2,SEEK_SET) ;
      return tgtBuffer ;
    } else {
      // return to location after \r (strlen+1)
      FAT_fseek(file,curPos+strlen(tgtBuffer)+1,SEEK_SET) ;
      return tgtBuffer ;
    }
  }

  return tgtBuffer ;
}

#ifdef CAN_WRITE_TO_DISC
/*-----------------------------------------------------------------
FAT_fputs (const char *string, FAT_FILE* file)
Writes string to file, excluding end of string character
const char *string IN: string to write
FAT_FILE* file IN: Handle of open file
bool return OUT: number of characters written if successful,
  EOF if not

  Written by MightyMax
  Modified by Chishm - 2005-11-17
  * Uses FAT_FILE instead of int
  * writtenBytes is now u32 instead of int
-------------------------------------------------------------------*/
int FAT_fputs (const char *string, FAT_FILE* file)
{
   u32 writtenBytes;
  // save string except end of string '\0'
   writtenBytes = FAT_fwrite((void *)string, 1, strlen(string), file);

   // check if we had an error
   if (writtenBytes != strlen(string))
   {
      // return EOF error
      return EOF;
   }

   // return the charcount written
   return writtenBytes ;
}
#endif
```

(ff_uart.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

// UART driver for the debug console

#include "pen.h"

//FFUART mostly used for output, so TX queue is bigger
#define RX_SIZE 64  //power of 2 so wrapping is easier
static u8 rx_queue[RX_SIZE];
static int rx_head; //write to here
static int rx_tail; //read from here

#define TX_SIZE 2048
static u8 tx_queue[TX_SIZE];
static int tx_head; //write to here
static int tx_tail; //read from here

//pull char from incoming queue, -1 if queue empty
int ff_uart_getc() {
  int ch;
  int cpumode;

  if(rx_head==rx_tail)
    return -1;

  cpumode=IRQS_OFF(); //critical section, don't let interrupts mess up the queue
  ch=rx_queue[rx_tail];
  rx_tail=(rx_tail+1)&(RX_SIZE-1);
  IRQS_SET(cpumode);
  return ch;
}

//called by main interrupt handler
void ff_uart_irq_handler() {
  int txcount=0;
  int iir=FFIIR&0x0f;

  if(iir==2) {  //Tx interrupt
    do {
      if(tx_head==tx_tail) { //nothing left to send, turn off Tx interrupt
```

```
        FFIER = FFIER & ~IER_TIE;
        break;
      } else {
        FFTHR=tx_queue[tx_tail];
        tx_tail=(tx_tail+1)&(TX_SIZE-1);
        txcount++;
      }
    } while(txcount<31);//stuff as many bytes into tx fifo as possible
  } else if(iir==4 || iir==12) {  //Rx interrupt
    do {
      rx_queue[rx_head]=FFRBR;   //read incoming char into rx queue
      rx_head=(rx_head+1)&(RX_SIZE-1);
      if(rx_head==rx_tail)
        rx_tail=(rx_tail+1)&(RX_SIZE-1);  //queue's full, old char gets dropped
    } while(FFLSR & LSR_DR); //loop until Rx FIFO is empty
    setsig(SIG_UART);
  }
}

//clear uart buffers
void ff_uart_flush() {
  int cpumode;
  cpumode=IRQS_OFF(); //critical section, don't let interrupts mess up the queue

  rx_head=0;    //reset queues
  rx_tail=0;
  tx_head=0;
  tx_tail=0;
  FFFCR=FCR_TRFIFOE | FCR_RESETTF | FCR_RESETRF | FCR_ITL_16;  //clear uart fifos

  IRQS_SET(cpumode);
}

//setup FFUART for interrupt-driven Rx+Tx
void ff_uart_init() {
  pxa_gpio_mode(GPIO39_FFTXD_MD);//grab FFUART GPIO pins
  pxa_gpio_mode(GPIO34_FFRXD_MD);

  CKEN|=CKEN_FFUART;  //UART clock enable
  FFIER=0;   //FFUART off
  FFMCR=MCR_OUT2;   //no loopback, UART interrupts enabled
  FFLCR=LCR_DLAB;   //get access to divisor
  FFDLL=DLL_BAUD(115200);  //set baud rate (lo)
  FFDLH=0;   //set baud late (hi)
  FFLCR=LCR_WLS(3); //no parity, 8 data bits, 1 stop bit
  FFFCR=FCR_TRFIFOE | FCR_RESETTF | FCR_RESETRF | FCR_ITL_16;  //enable FIFO
  FFIER=IER_UUE | IER_RTOIE | IER_RAVIE;//FFUART on, RX interrupts enabled
  ff_uart_flush();
  irq_enable(IRQ_FFUART);
}

//put char into Tx queue, let interrupts do the rest
int ff_uart_putc(int c) {
  int cpumode;
  int head;
  cpumode=IRQS_OFF(); //critical section

  tx_queue[tx_head]=c;//put char into queue
  head=(tx_head+1)&(TX_SIZE-1);
  if(head!=tx_tail) { //chars are dropped if queue is full
    tx_head=head;
  }

  FFIER |= IER_TIE; //Tx interrupt enable

  IRQS_SET(cpumode);
  return c;
}

/*** define stdio stuff so printf(), etc goes to FFUART ***/
```

```
#include <stdio.h>

struct __FILE {
  int handle;
};

FILE __stdout;

int fputc(int ch, FILE *f) {
  if(ch=='\n')
    ff_uart_putc('\r');
  return ff_uart_putc(ch);
}

int ferror(FILE *f) {
  return 0;  //no file error
}
```

(flash.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

// code for writing to gumstix flash memory

#include <stdio.h>
#include <string.h>
#include "pen.h"

void flashthread(void);

extern u8 diskimage[];   //image of 12MB formatted disk
extern int diskimagesize;

//#define FATIMAGESIZE 12MB (in pen.h)
#define BLOCKSIZE 0x20000
#define BLOCKS (FATIMAGESIZE/BLOCKSIZE)
#define FATBASE (4*0x100000) //base address of FAT image in flash
#define BLOCK(x) (*(vu16*)(BLOCKSIZE*(x)))

#define DIRTYTIMEOUT (30*1000)  //milliseconds to wait before flashing a dirty block
#define ERASETIMEOUT MTICKS(10) //flash timer interrupt rate
#define FLASHTIMEOUT UTICKS(30)

#define CMD_READ  0xFF
#define CMD_READID  0x90
#define CMD_READSTATUS 0x70
#define CMD_LOCK  0x60
#define CMD_WRITEBUF 0xE8
#define CMD_ERASE 0x20
#define CMD_CONFIRM 0xd0

#define STATUS_READY 0x80 //not busy erasing or programming
#define STATUS_ERROR 0x30 //erase or program error

#pragma arm section zidata="nocache"
u8 flashcache[FATIMAGESIZE];
u8 flashbuffer[BLOCKSIZE]; //data is copied here before flashing, so normal writing can continue
#pragma arm section zidata

int flashlock;         //controls access to physical flash
vu32 flashdirty;       //=1 if any blocks need to be flashed
static u32 writetime[BLOCKS];   //when block was first dirtied (0=clean block)
static char valid[BLOCKS];   //cache block is valid? (0/1)

u16 *flashsrc, *flashdst, *flashend;  //stuff for flash timer interrupt..
vu32 flashing;  //something is being flashed? (0/1)
u32 timeout;  //timeout for flash timer (ERASETIMEOUT or FLASHTIMEOUT)
u32 laststatus; //status from last flash operation (error code)
```

```
//u32 flashed,eraseticks,writeticks;  //some stats for tuning the timer

//overwrite the cache with an empty disk image
void flash_format() {
  flashdirty=0;
  memset(valid,1,sizeof(valid));
  memset(writetime,0,sizeof(writetime));
  memset(flashcache,0,BLOCKSIZE);
  flash_writecache(0,diskimage,diskimagesize);
}

void flash_init() {
  unlock(&flashlock);
  flashing=0;
  flashdirty=0;
  memset(writetime,0,sizeof(writetime));  //reset dirty flags

  //if(1) {
  if(memcmp((u8*)FATBASE,diskimage,512)) { //disk image is corrupt/missing
    flash_format();
  } else {
    //memset(valid,0,sizeof(valid));

    //preload the whole flash cache (FIXME... USB code expects flashcache to be valid)
    memset(valid,1,sizeof(valid));
    DMAcopy(flashcache,(u8*)FATBASE,FATIMAGESIZE);
  }
  irq_enable(IRQ_OST1); //enable flash timer interrupt
  newthread((void*)flashthread,0); //start thread to watch for dirty blocks and flash them
}

/*
//wait til flash is ready
void flash_busywait() {
  BLOCK(0)=CMD_READSTATUS;
  while(!(BLOCK(0) & 0x80)) {
    //idle();  //don't use this inside interrupt processing
  }
}
*/

//returns busy status
int flash_busy() {
  BLOCK(0)=CMD_READSTATUS;
  return (BLOCK(0)&STATUS_READY)==0;
}

//get status register contents
int flash_status() {
  BLOCK(0)=CMD_READSTATUS;
  return BLOCK(0);
}

//start erasing a block
int flash_erase(vu16 *dst) {
  if(flash_busy())
    return 0;
  *dst=CMD_ERASE;
  *dst=CMD_CONFIRM;
  return 1;
}

//fill the write buffer and start programming
void flash_writebufr(vu16 *src, vu16 *dest) {
  int i;
  do {       //wait til buffer is ready
    *dest=CMD_WRITEBUF;
  } while(!*dest&0x80);

  *dest=0x0f;     //write 16 words (32 bytes)
  for (i=0; i<16; i++) {
    dest[i] = src[i];
  }
  *dest = CMD_CONFIRM;   //start dumping the buffer to flash
}

//make flash readable
void flash_readmode() {
  BLOCK(0)=CMD_READ;
}

/*
//make block "locked" (prevent erasing or programming)
void flash_lock(int block) {
  flash_busywait();
  BLOCK(block)=CMD_LOCK;
  BLOCK(block)=1;
  while(flash_busy()); //wait til finished
}
*/

//write one block (wait til flashing completes)
int flash_writeblock(u8 *src, u8 *dst) {
  if(flashing)
    return 0;

  ff_uart_putc('.');

  DMAcopy(flashbuffer,src,BLOCKSIZE);

  flashing=1;
  flashsrc=(u16*)flashbuffer;
  flashdst=(u16*)dst;
  flashend=(u16*)(dst+BLOCKSIZE);
  if(!flash_erase(flashdst))      //start the erase cycle
    return 0;

  timeout=ERASETIMEOUT;
  OSMR1=OSCR+ERASETIMEOUT;     //start the flash timer
  OSSR=2;
  OIER|=2;

  do {            //wait til erase+program is all done
    wait(SIG_FLASH);
  } while(flashing);

  if(laststatus&STATUS_ERROR)
    printf("Flash error.\n");
  return (laststatus&STATUS_ERROR)==0;
}

//write a block of the FAT cache
int writeFATblock(u32 block) {
  block*=BLOCKSIZE;
  return flash_writeblock(flashcache+block,(u8*)FATBASE+block);
}

//watch for dirty blocks, commit them to flash after X seconds
void flashthread() {
  u32 i,time;
  while(1) {
    wait(SIG_ANY);
    //msleep(?); don't use big delay, so you can do immediate flash if necessary (i.e. shutdown)

    time=RTC;
    for(i=0;i<BLOCKS;i++) {
      if(writetime[i] && time-writetime[i]>DIRTYTIMEOUT) {  //dirty time elapsed?
        lock(&flashlock);
        writetime[i]=0;   //make clean
        writeFATblock(i);
        unlock(&flashlock);
        break;
      }
    }
    if(i==BLOCKS)
      flashdirty=0; //show that the whole flash is clean (for flashall)
  }
}

//do flash erasing+writing
void flashtimer_irq_handler() {
  int status=flash_status();

  /* if(timeout==FLASHTIMEOUT)
    writeticks++;
  else
    eraseticks++; */

  if(status&STATUS_READY) {
    if((status&STATUS_ERROR) || (flashdst>=flashend)) { //done flashing or error occurred?
      laststatus=status;
      flashing=0;
      OIER&=~2;       //disable timer interrupt
      setsig(SIG_FLASH);
    } else { //not finished yet, write the next piece
      timeout=FLASHTIMEOUT;    //change timer frequency for writing
      flash_writebufr(flashsrc, flashdst);
      flashsrc+=16;
      flashdst+=16;
    }
  }
  OSSR=2;         //clear interrupt
  OSMR1=OSCR+timeout;    //set next timeout
}

int flash_readcache(void *dst, u32 src, int size) {
  u32 i,block;

  if(src+size>FATIMAGESIZE) {
    printf("Flash read out of range (%x)\n",src);
    return 0;
  }

  //load invalid blocks with good data first
  for(i=src;i<src+size;i+=BLOCKSIZE) {
    if(!valid[i/BLOCKSIZE]) {
      lock(&flashlock);
      flash_readmode();
      block=i&~(BLOCKSIZE-1);
      memcpy(flashcache+block,(u8*)(FATBASE+block),BLOCKSIZE);
      unlock(&flashlock);
      valid[i/BLOCKSIZE]=1;
      break;
    }
  }
  memcpy(dst,flashcache+src,size);
  return 1;
}

int flash_writecache(u32 dst, void *src, int size) {
  u32 i;
  u32 block;

  if(dst+size>FATIMAGESIZE) {
    printf("Write out of range.\n");
    return 0;
  }

  flashdirty=1;
  for(i=dst;i<dst+size;i+=BLOCKSIZE) {  //scan thru all blocks
    if(!writetime[i/BLOCKSIZE])
      writetime[i/BLOCKSIZE]=RTC;  //set dirty time
    if(!valid[i/BLOCKSIZE]) {  //if this block is empty, load the whole block first
```

```
      valid[i/BLOCKSIZE]=1;
      lock(&flashlock);
      flash_readmode();
      block=i&~(BLOCKSIZE-1);
      memcpy(flashcache+block,(u8*)FATBASE+block,BLOCKSIZE);
      unlock(&flashlock);
    }
  }
  memcpy(flashcache+dst,src,size);  //don't use DMAcopy (src might be in cpu cache)
  return 1;
}

//print some statistics
void flashinfo() {
  int i,dirty=0;
  u32 nextwrite=-1;

  printf("Flash status: 0x%X\n",flash_status());
  printf("Last error: 0x%X\n",laststatus);
  // printf("flashed=%i eraseticks=%i writeticks=%i\n",flashed,eraseticks,writeticks);

  //count dirty blocks
  for(i=0;i<BLOCKS;i++) {
    if(writetime[i]) {
      dirty++;
      if(nextwrite>writetime[i])
        nextwrite=writetime[i];
    }
  }
  printf("%i/%i flash blocks dirty.\n",dirty,i);
  if(dirty)
    printf("Next write in %i seconds.\n",(DIRTYTIMEOUT-RTC+nextwrite)/1000);
}

//write back all dirty pages
void flashall() {
  int i,dirty;
  flashdirty=1;   //reset global flash status
  dirty=0;
  for(i=0;i<BLOCKS;i++) {
    if(writetime[i]) {  //if block is dirty,
      dirty++;
      writetime[i]=1; //force a write timeout
    }
  }
  printf("Flashing %i dirty pages",dirty);
  do {
    wait(SIG_ANY);
  } while(flashdirty);
  printf("\n");
}
```

(font.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

// helper functions for printing with the FreeType font library

#include <stdio.h>
#include <ft2build.h>
#include FT_FREETYPE_H
#include "pen.h"

static FT_Library library;
static FT_Face face;

extern FT_Byte font[];
extern int fontsize;
```

```
//draw monochrome font
void draw_1bpp(int x, int y, FT_Bitmap *fontbmp) {
  u8 *src,*rowsrc;
  int i,j;
  int width,height,pitch;
  u8 c;
  int bit;

  width=fontbmp->width;
  height=fontbmp->rows;
  rowsrc=fontbmp->buffer;
  pitch=fontbmp->pitch;

  for(j=0;j<height;j++) {
    src=rowsrc;
    bit=8;
    for(i=0;i<width;i++) {
      if(bit>7) {bit=0; c=*src++;}

      if(c&0x80) putpix(x+i, y+j,BLACK);
      c<<=1;
      bit++;
    }
    rowsrc+=pitch;
  }
}

// x,y is the bottom left corner of text
//returns end of string (null char)
char *print(char *c, int x, int y) {
  FT_GlyphSlot slot=face->glyph;  // a small shortcut
  int error;

  while(*c) {
    //load glyph image into the slot (erase previous one)

    error = FT_Load_Char( face, *c++, FT_LOAD_RENDER | FT_LOAD_TARGET_MONO );
    if ( error ) continue; //ignore errors

    //now, draw to our target surface
    draw_1bpp(x + slot->bitmap_left, y - slot->bitmap_top,
    &slot->bitmap);

    x += slot->advance.x >> 6; //increment pen position
  }
  return c;
}

void font_init() {
  if(FT_Init_FreeType(&library)) {
    printf("font_init failed.\n");
    return;
  }
  if(FT_New_Memory_Face(library, font, fontsize, 0, &face)) {
    printf("font_init failed.\n");
    return;
  }
  if(FT_Set_Pixel_Sizes(face, 0, 16)) { //face, width, height
    printf("font_init failed.\n");
    return;
  }
}
```

(gfx.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////
```

```
// miscellaneous drawing routines

#include <stdio.h>
#include "pen.h"

//draw INCBIN'd bitmap with embedded dimensions
void draw_bmp(int x, int y, u8 *src) {
  u8 *dst;
  int width,height;
  int i,j;

  width=*(u32*)src;
  height=*(u32*)(src+4);
  src+=8;
  dst=frame+PORTRAIT_OFFSET(x+width-1,y);

  //do some range checking?

  //this is not efficient.. should check for word alignment & do block copy
  for(j=0;j<width;j++) {
    for(i=0;i<height;i++) {
      *dst++ = *src++;
    }
    dst+=Y_RES-height;
  }
}

//draw antialiased line
//stolen from http://www.codeproject.com/gdi/antialias.asp
void aline (int X0, int Y0, int X1, int Y1) {

#define BaseColor 0
#define IntensityBits 4
#define NumLevels 16

   unsigned short IntensityShift, ErrorAdj, ErrorAcc;
   unsigned short ErrorAccTemp, Weighting, WeightingComplementMask;
   short DeltaX, DeltaY, Temp, XDir;

   /* Make sure the line runs top to bottom */
   if (Y0 > Y1) {
      Temp = Y0; Y0 = Y1; Y1 = Temp;
      Temp = X0; X0 = X1; X1 = Temp;
   }
   /* Draw the initial pixel, which is always exactly intersected by
      the line and so needs no weighting */
   putpix(X0, Y0, BaseColor);

   if ((DeltaX = X1 - X0) >= 0) {
      XDir = 1;
   } else {
      XDir = -1;
      DeltaX = -DeltaX; /* make DeltaX positive */
   }
   /* Special-case horizontal, vertical, and diagonal lines, which
      require no weighting because they go right through the center of
      every pixel */
   if ((DeltaY = Y1 - Y0) == 0) {
      /* Horizontal line */
      while (DeltaX-- != 0) {
         X0 += XDir;
         putpix(X0, Y0, BaseColor);
      }
      return;
   }
   if (DeltaX == 0) {
      /* Vertical line */
      do {
         Y0++;
         putpix(X0, Y0, BaseColor);
      } while (--DeltaY != 0);
      return;
   }
   if (DeltaX == DeltaY) {
      /* Diagonal line */
      do {
         X0 += XDir;
         Y0++;
         putpix(X0, Y0, BaseColor);
      } while (--DeltaY != 0);
      return;
   }
   /* Line is not horizontal, diagonal, or vertical */
   ErrorAcc = 0;  /* initialize the line error accumulator to 0 */
   /* # of bits by which to shift ErrorAcc to get intensity level */
   IntensityShift = 16 - IntensityBits;
   /* Mask used to flip all bits in an intensity weighting, producing the
      result (1 - intensity weighting) */
   WeightingComplementMask = NumLevels - 1;
   /* Is this an X-major or Y-major line? */
   if (DeltaY > DeltaX) {
      /* Y-major line; calculate 16-bit fixed-point fractional part of a
         pixel that X advances each time Y advances 1 pixel, truncating the
         result so that we won't overrun the endpoint along the X axis */
      ErrorAdj = ((unsigned long) DeltaX << 16) / (unsigned long) DeltaY;
      /* Draw all pixels other than the first and last */
      while (--DeltaY) {
         ErrorAccTemp = ErrorAcc;   /* remember currrent accumulated error */
         ErrorAcc += ErrorAdj;      /* calculate error for next pixel */
         if (ErrorAcc <= ErrorAccTemp) {
            /* The error accumulator turned over, so advance the X coord */
            X0 += XDir;
         }
         Y0++; /* Y-major, so always advance Y */
         /* The IntensityBits most significant bits of ErrorAcc give us the
            intensity weighting for this pixel, and the complement of the
            weighting for the paired pixel */
         Weighting = ErrorAcc >> IntensityShift;
         putpix(X0, Y0, BaseColor + Weighting);
         putpix(X0 + XDir, Y0,
               BaseColor + (Weighting ^ WeightingComplementMask));
      }
      /* Draw the final pixel, which is always exactly intersected by the line
         and so needs no weighting */
      putpix(X1, Y1, BaseColor);
      return;
   }
   /* It's an X-major line; calculate 16-bit fixed-point fractional part of a
      pixel that Y advances each time X advances 1 pixel, truncating the
      result to avoid overrunning the endpoint along the X axis */
   ErrorAdj = ((unsigned long) DeltaY << 16) / (unsigned long) DeltaX;
   /* Draw all pixels other than the first and last */
   while (--DeltaX) {
      ErrorAccTemp = ErrorAcc;   /* remember currrent accumulated error */
      ErrorAcc += ErrorAdj;      /* calculate error for next pixel */
      if (ErrorAcc <= ErrorAccTemp) {
         /* The error accumulator turned over, so advance the Y coord */
         Y0++;
      }
      X0 += XDir; /* X-major, so always advance X */
      /* The IntensityBits most significant bits of ErrorAcc give us the
         intensity weighting for this pixel, and the complement of the
         weighting for the paired pixel */
      Weighting = ErrorAcc >> IntensityShift;
      putpix(X0, Y0, BaseColor + Weighting);
      putpix(X0, Y0 + 1,
            BaseColor + (Weighting ^ WeightingComplementMask));
   }
   /* Draw the final pixel, which is always exactly intersected by the line
      and so needs no weighting */
   putpix(X1, Y1, BaseColor);
```

```
}

void line(int x0, int y0, int x1, int y1, plotfn plot) {
        int dy = y1 - y0;
        int dx = x1 - x0;
        int stepx, stepy;
        int fraction;

        if (dy < 0) { dy = -dy;  stepy = -1; } else { stepy = 1; }
        if (dx < 0) { dx = -dx;  stepx = -1; } else { stepx = 1; }
        dy <<= 1;                                                  // dy is now 2*dy
        dx <<= 1;                                                  // dx is now 2*dx

  plot(x0,y0);

        if (dx > dy) {
            fraction = dy - (dx >> 1);                         // same as 2*dy - dx
            while (x0 != x1) {
                if (fraction >= 0) {
                    y0 += stepy;
                    fraction -= dx;                                // same as fraction -= 2*dx
                }
                x0 += stepx;
                fraction += dy;                                    // same as fraction -= 2*dy
                plot(x0,y0);
            }
        } else {
            fraction = dx - (dy >> 1);
            while (y0 != y1) {
                if (fraction >= 0) {
                    x0 += stepx;
                    fraction -= dy;
                }
                y0 += stepy;
                fraction += dx;
                plot(x0,y0);
            }
        }
}

void putpix(int x,int y, char c) {
  frame[y+(X_MAX-x)*Y_RES]=c;
}

//black dot
void point(int x,int y) {
  frame[y+(X_MAX-x)*Y_RES]=0;
}

//erase one 16x16 square - used with line()
void erase(int x,int y) {
  int i,j, offset;
  u8 *src, *dst;

  x-=8; y-=8;
  if(x<0) x=0;
  if(y<0) y=0;
  if(x>X_RES-16) x=X_RES-16;
  if(y>Y_RES-16) y=Y_RES-16;

  offset=PORTRAIT_OFFSET(x+15,y);
  src=background+offset;
  dst=frame+offset;
  for(i=0;i<16;i++) {
    for(j=0;j<16;j++) {
      *dst++=*src++;
    }
    src+=Y_RES-16;
    dst+=Y_RES-16;
  }
}
```

```
//draw a filled rect
//x0,y0=top left, x1,y1=bottom right
void rect(int x0, int y0, int x1, int y1, int color) {
  u8 *dst;
  int width,height;
  int x,y;

  dst=frame+PORTRAIT_OFFSET(x1,y0);

  width=x1-x0+1;
  height=y1-y0+1;

  for(x=0;x<width;x++) {
    for(y=0;y<height;y++) {
      *dst++=color;
    }
    dst+=Y_RES-height;
  }
}

buttonstruct yes_no_buttons[]={
  {0,0,        0,0, -1,-1}, //first parameter is the drawing surface
  {NoU, NoP,      530,980, 0,0},
  {YesU, YesP,      458,980, 0,0},
  {0,}
};

//draw text to status area
void statustext(char *str) {
  char *s=str;
  int y=945;
  rect(STATUS_X, 925, 607, 1023, RED);
  line(STATUS_X, 924, 607, 924,point);
  while(*s) {
    s=print(s,STATUS_X+7,y)+1;
    y+=20;
  }
}

//show yes/no dialog, wait for button input
int yesno(char *str) {
  pen_event pen;
  statustext(str);
  set_buttons(yes_no_buttons);   //for now, assume the caller will restore its buttons
  while(1) {
    wait(SIG_GPIO|SIG_PEN|SIG_UART);
    if(get_pen_event(&pen)) {
      if(pen.event==BUTTON_EVENT) {
        return pen.buttonID-1;   //no=0, yes=1;
      } else {
        return 0;
      }
    }
  }
}

//redraw page status area
void redraw_status(void) {
  char s[32];
  set_buttons(dashboard);  //redraw buttons?
  rect(STATUS_X,925, 607, 1023, WHITE);
  line(STATUS_X,924, 607,924,point);
  sprintf(s,"%i of %i", book.currentpagenumber+1, book.pages);
       print(s, STATUS_X+7, 1017);
  if(clipboard_full)
    print("Clip",STATUS_X+7,970);
  if(book.currentpage->bookmarked)
    print("Bookmarked",STATUS_X+7,990);
}
```

(gifdecode.c)

```
/* DECODE.C - An LZW decoder for GIF
 * Copyright (C) 1987, by Steven A. Bennett
 *
 * Permission is given by the author to freely redistribute and include
 * this code in any program as long as this credit is given where due.
 *
 * In accordance with the above, I want to credit Steve Wilhite who wrote
 * the code which this is heavily inspired by...
 *
 * GIF and 'Graphics Interchange Format' are trademarks (tm) of
 * Compuserve, Incorporated, an H&R Block Company.
 *
 * Release Notes: This file contains a decoder routine for GIF images
 * which is similar, structurally, to the original routine by Steve Wilhite.
 * It is, however, somewhat noticably faster in most cases.
 *
 */
#include "pen.h"

typedef short WORD;
typedef unsigned short UWORD;
typedef unsigned char UTINY;
typedef long LONG;
typedef unsigned long ULONG;
//ok to change, just keep these negative
#define OUT_OF_MEMORY -10
#define BAD_CODE_SIZE -20
#define READ_ERROR -1

WORD decoder(int linewidth);

//************** user defined stuff ******************

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "fat.h"

static char buff[512]; //file buffer
static int p;   //next byte to read from buff
static int left;  //bytes left in buff
static char *dest;  //where to decode to
static FAT_FILE *f;

//get next byte from file.  return an error code if there's a problem...
int get_byte() {
  if(!left) {
    p=0;
    left=FAT_fread(buff,1,sizeof(buff),f);
    if(!left) return READ_ERROR;
  }
  --left;
  return buff[p++];
}

//decode GIF file to buffer
int gif_read(char *filename, u8 *buffer) {
  dest=(char*)buffer;
  f=FAT_fopen(filename,"r");
  if(!f)
    return 0;
  FAT_fseek(f,0x317,SEEK_SET); //skip past header
  p=left=0;     //reset the file buffer
  decoder(Y_RES); //decode the whole thing, out_line will be called for each line
  FAT_fclose(f);
  return 1;
}

//called when decoder wants to feed us some decoded pixels
```

```
int out_line(unsigned char *pixels, int linelen) {
  memcpy(dest,pixels,linelen);
  dest+=linelen;
  return 0;
}

/* Incremented each time an out of range code is read by the decoder.
 * When this value is non-zero after a decode, your GIF file is probably
 * corrupt in some way... */
int bad_code_count;

//*********************************************************

//#define NULL   0
#define MAX_CODES   4095

/* Static variables */
static WORD curr_size;                    /* The current code size */
static WORD clear;                        /* Value for a clear code */
static WORD ending;                       /* Value for a ending code */
static WORD newcodes;                     /* First available code */
static WORD top_slot;                     /* Highest code for current size */
static WORD slot;                         /* Last read code */

/* The following static variables are used
 * for seperating out codes
 */
static WORD navail_bytes = 0;             /* # bytes left in block */
static WORD nbits_left = 0;               /* # bits left in current byte */
static UTINY b1;                          /* Current byte */
static UTINY byte_buff[257];              /* Current block */
static UTINY *pbytes;                     /* Pointer to next byte in block */

static LONG code_mask[13] = {
     0,
     0x0001, 0x0003,
     0x0007, 0x000F,
     0x001F, 0x003F,
     0x007F, 0x00FF,
     0x01FF, 0x03FF,
     0x07FF, 0x0FFF
     };


/* This function initializes the decoder for reading a new image.
 */
static WORD init_exp(int size)
   {
   curr_size = size + 1;
   top_slot = 1 << curr_size;
   clear = 1 << size;
   ending = clear + 1;
   slot = newcodes = ending + 1;
   navail_bytes = nbits_left = 0;
   return(0);
   }

/* get_next_code()
 * - gets the next code from the GIF file.  Returns the code, or else
 * a negative number in case of file errors...
 */
static WORD get_next_code()
   {
   WORD i, x;
   ULONG ret;

   if (nbits_left == 0)
      {
      if (navail_bytes <= 0)
         {
```

```
         /* Out of bytes in current block, so read next block
          */
         pbytes = byte_buff;
         if ((navail_bytes = get_byte()) < 0)
            return(navail_bytes);
         else if (navail_bytes)
            {
            for (i = 0; i < navail_bytes; ++i)
               {
               if ((x = get_byte()) < 0)
                  return(x);
               byte_buff[i] = x;
               }
            }
         }
      b1 = *pbytes++;
      nbits_left = 8;
      --navail_bytes;
      }

   ret = b1 >> (8 - nbits_left);
   while (curr_size > nbits_left)
      {
      if (navail_bytes <= 0)
         {

         /* Out of bytes in current block, so read next block
          */
         pbytes = byte_buff;
         if ((navail_bytes = get_byte()) < 0)
            return(navail_bytes);
         else if (navail_bytes)
            {
            for (i = 0; i < navail_bytes; ++i)
               {
               if ((x = get_byte()) < 0)
                  return(x);
               byte_buff[i] = x;
               }
            }
         }
      b1 = *pbytes++;
      ret |= b1 << nbits_left;
      nbits_left += 8;
      --navail_bytes;
      }
   nbits_left -= curr_size;
   ret &= code_mask[curr_size];
   return((WORD)(ret));
   }


/* The reason we have these seperated like this instead of using
 * a structure like the original Wilhite code did, is because this
 * stuff generally produces significantly faster code when compiled...
 * This code is full of similar speedups...  (For a good book on writing
 * C for speed or for space optomisation, see Efficient C by Tom Plum,
 * published by Plum-Hall Associates...)
 */
static UTINY stack[MAX_CODES + 1];            /* Stack for storing pixels */
static UTINY suffix[MAX_CODES + 1];           /* Suffix table */
static UWORD prefix[MAX_CODES + 1];           /* Prefix linked list */

/* WORD decoder(linewidth)
 *    WORD linewidth;               * Pixels per line of image *
 *
 * - This function decodes an LZW image, according to the method used
 * in the GIF spec.  Every *linewidth* "characters" (ie. pixels) decoded
 * will generate a call to out_line(), which is a user specific function
 * to display a line of pixels.  The function gets it's codes from
 * get_next_code() which is responsible for reading blocks of data and
 * seperating them into the proper size codes.  Finally, get_byte() is
 * the global routine to read the next byte from the GIF file.
 *
 * It is generally a good idea to have linewidth correspond to the actual
 * width of a line (as specified in the Image header) to make your own
 * code a bit simpler, but it isn't absolutely necessary.
 *
 * Returns: 0 if successful, else negative.
 *
 */

WORD decoder(int linewidth)
   {
   UTINY *sp, *bufptr;
   UTINY *buf;
   WORD code, fc, oc, bufcnt;
   WORD c, size, ret;

   /* Initialize for decoding a new image...
    */
   if ((size = get_byte()) < 0)
      return(size);
   if (size < 2 || 9 < size)
      return(BAD_CODE_SIZE);
   init_exp(size);

   /* Initialize in case they forgot to put in a clear code.
    * (This shouldn't happen, but we'll try and decode it anyway...)
    */
   oc = fc = 0;

   /* Allocate space for the decode buffer
    */
   if ((buf = (UTINY *)malloc(linewidth + 1)) == NULL)
      return(OUT_OF_MEMORY);

   /* Set up the stack pointer and decode buffer pointer
    */
   sp = stack;
   bufptr = buf;
   bufcnt = linewidth;

   /* This is the main loop.  For each code we get we pass through the
    * linked list of prefix codes, pushing the corresponding "character" for
    * each code onto the stack.  When the list reaches a single "character"
    * we push that on the stack too, and then start unstacking each
    * character for output in the correct order.  Special handling is
    * included for the clear code, and the whole thing ends when we get
    * an ending code.
    */
   while ((c = get_next_code()) != ending)
      {

      /* If we had a file error, return without completing the decode
       */
      if (c < 0)
         {
         free(buf);
         return(0);
         }

      /* If the code is a clear code, reinitialize all necessary items.
       */
      if (c == clear)
         {
         curr_size = size + 1;
         slot = newcodes;
         top_slot = 1 << curr_size;

         /* Continue reading codes until we get a non-clear code
          * (Another unlikely, but possible case...)
```

```
             */
            while ((c = get_next_code()) == clear)
               ;

            /* If we get an ending code immediately after a clear code
             * (Yet another unlikely case), then break out of the loop.
             */
            if (c == ending)
               break;

            /* Finally, if the code is beyond the range of already set codes,
             * (This one had better NOT happen...  I have no idea what will
             * result from this, but I doubt it will look good...) then set it
             * to color zero.
             */
            if (c >= slot)
               c = 0;

            oc = fc = c;

            /* And let us not forget to put the char into the buffer... And
             * if, on the off chance, we were exactly one pixel from the end
             * of the line, we have to send the buffer to the out_line()
             * routine...
             */
            *bufptr++ = c;
            if (--bufcnt == 0)
               {
               if ((ret = out_line(buf, linewidth)) < 0)
                  {
                  free(buf);
                  return(ret);
                  }
               bufptr = buf;
               bufcnt = linewidth;
               }
            }
         else
            {

            /* In this case, it's not a clear code or an ending code, so
             * it must be a code code...  So we can now decode the code into
             * a stack of character codes. (Clear as mud, right?)
             */
            code = c;

            /* Here we go again with one of those off chances...  If, on the
             * off chance, the code we got is beyond the range of those already
             * set up (Another thing which had better NOT happen...) we trick
             * the decoder into thinking it actually got the last code read.
             * (Hmmn... I'm not sure why this works...  But it does...)
             */
            if (code >= slot)
               {
               if (code > slot)
                  ++bad_code_count;
               code = oc;
               *sp++ = fc;
               }

            /* Here we scan back along the linked list of prefixes, pushing
             * helpless characters (ie. suffixes) onto the stack as we do so.
             */
            while (code >= newcodes)
               {
               *sp++ = suffix[code];
               code = prefix[code];
               }

            /* Push the last character on the stack, and set up the new
             * prefix and suffix, and if the required slot number is greater
```

```
          * than that allowed by the current bit size, increase the bit
          * size.  (NOTE - If we are all full, we *don't* save the new
          * suffix and prefix...  I'm not certain if this is correct...
          * it might be more proper to overwrite the last code...
          */
         *sp++ = code;
         if (slot < top_slot)
            {
            suffix[slot] = fc = code;
            prefix[slot++] = oc;
            oc = c;
            }
         if (slot >= top_slot)
            if (curr_size < 12)
               {
               top_slot <<= 1;
               ++curr_size;
               }

         /* Now that we've pushed the decoded string (in reverse order)
          * onto the stack, lets pop it off and put it into our decode
          * buffer...  And when the decode buffer is full, write another
          * line...
          */
         while (sp > stack)
            {
            *bufptr++ = *(--sp);
            if (--bufcnt == 0)
               {
               if ((ret = out_line(buf, linewidth)) < 0)
                  {
                  free(buf);
                  return(ret);
                  }
               bufptr = buf;
               bufcnt = linewidth;
               }
            }
         }
      }
   ret = 0;
   if (bufcnt != linewidth)
      ret = out_line(buf, (linewidth - bufcnt));
   free(buf);
   return(ret);
   }
```

(gifsave.c)

```
/**************************************************************************
 *
 *  FILE:           GIFSAVE.C
 *
 *  MODULE OF:      GIFSAVE
 *
 *  DESCRIPTION:    Routines to create a GIF-file. See GIFSAVE.DOC for
 *                  a description . . .
 *
 *                  The functions were originally written using Borland's
 *                  C-compiler on an IBM PC -compatible computer, but they
 *                  are compiled and tested on SunOS (Unix) as well.
 *
 *  WRITTEN BY:     Sverre H. Huseby
 *                  Bjoelsengt. 17
 *                  N-0468 Oslo
 *                  Norway
 *
 *                  sverrehu@ifi.uio.no
 *
 *  LAST MODIFIED:  26/9-1992, v1.0, Sverre H. Huseby
 *                    * Version 1.0, no modifications
 *
 ***************************************************************************/

#include <stdlib.h>
#include "fat.h"
#include "pen.h"

enum GIF_Code { GIF_OK,  GIF_ERRCREATE,  GIF_ERRWRITE,  GIF_OUTMEM };

/***************************************************************************
 *                                                                         *
 *                       P R I V A T E    D A T A                          *
 *                                                                         *
 ***************************************************************************/

typedef unsigned short Word;    /* At least two bytes (16 bits) */
typedef unsigned char Byte;     /* Exactly one byte (8 bits) */

/*========================================================================*
 =                                                                        =
 =                             I/O Routines                               =
 =                                                                        =
 *========================================================================*/

static FAT_FILE *OutFile;           /* File to write to */

/*========================================================================*
 =                                                                        =
 =                      Routines to write a bit-file                      =
 =                                                                        =
 *========================================================================*/

static Byte Buffer[256];        /* There must be one to much !!! */
static int  Index,              /* Current byte in buffer */
            BitsLeft;           /* Bits left to fill in current byte. These */
                                /* are right-justified */



/*========================================================================*
 =                                                                        =
 =                Routines to maintain an LZW-string table                =
 =                                                                        =
 *========================================================================*/

#define RES_CODES 2

#define HASH_FREE 0xFFFF
#define NEXT_FIRST 0xFFFF

#define MAXBITS 12
#define MAXSTR (1 << MAXBITS)

#define HASHSIZE 9973
#define HASHSTEP 2039

#define HASH(index, lastbyte) (((lastbyte << 8) ^ index) % HASHSIZE)

static Byte *StrChr = NULL;
static Word *StrNxt = NULL,
            *StrHsh = NULL,
            NumStrings;



/*========================================================================*
 =                                                                        =
 =                             Main routines                              =
 =                                                                        =
 *========================================================================*/
```

```
typedef __packed struct {
    Word LocalScreenWidth,
         LocalScreenHeight;
    Byte Flags;
/* GlobalColorTableSize : 3,
         SortFlag             : 1,
         ColorResolution      : 3,
         GlobalColorTableFlag : 1;*/
    Byte BackgroundColorIndex;
    Byte PixelAspectRatio;
} ScreenDescriptor;

typedef __packed struct {
    Byte Separator;
    Word LeftPosition,
         TopPosition;
    Word Width,
         Height;
    Byte Flags; //not used...
} ImageDescriptor;

static int  BitsPrPrimColor,    /* Bits pr primary color */
            NumColors;          /* Number of colors in color table */
static Byte *ColorTable = NULL;
static Word ScreenHeight,
            ScreenWidth,
            ImageHeight,
            ImageWidth,
            ImageLeft,
            ImageTop,
            RelPixX, RelPixY;         /* Used by InputByte() -function */
static int  (*GetPixel)(int x, int y);









/*************************************************************************
 *                                                                       *
 *                  P R I V A T E    F U N C T I O N S                   *
 *                                                                       *
 *************************************************************************/


/*========================================================================*
 =                                                                        =
 =                         Routines to do file IO                         =
 =                                                                        =
 *========================================================================*/

/*-------------------------------------------------------------------------
 *
 *  NAME:           Create()
 *
 *  DESCRIPTION:    Creates a new file, and enables referencing using the
 *                  global variable OutFile. This variable is only used
 *                  by these IO-functions, making it relatively simple to
 *                  rewrite file IO.
 *
 *  PARAMETERS:     filename - Name of file to create
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_ERRWRITE - Error opening the file
 *
 */
```

```
static int Create(char *filename) {
  OutFile=FAT_fopen(filename, "wb");
  if(!OutFile)
    return GIF_ERRCREATE;
  else
    return GIF_OK;
}




/*-------------------------------------------------------------------------
 *
 *  NAME:           Write()
 *
 *  DESCRIPTION:    Output bytes to the current OutFile.
 *
 *  PARAMETERS:     buf - Pointer to buffer to write
 *                  len - Number of bytes to write
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_ERRWRITE - Error writing to the file
 *
 */
static int Write(void *buf, unsigned len)
{
    if (FAT_fwrite(buf, sizeof(Byte), len, OutFile) < len)
        return GIF_ERRWRITE;

    return GIF_OK;
}




/*-------------------------------------------------------------------------
 *
 *  NAME:           WriteByte()
 *
 *  DESCRIPTION:    Output one byte to the current OutFile.
 *
 *  PARAMETERS:     b - Byte to write
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_ERRWRITE - Error writing to the file
 *
 */
static int WriteByte(Byte b)
{
    if (FAT_fwrite(&b, 1, 1, OutFile) < 1)
        return GIF_ERRWRITE;
    return GIF_OK;
}


/*-------------------------------------------------------------------------
 *
 *  NAME:           Close()
 *
 *  DESCRIPTION:    Close current OutFile.
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        Nothing
 *
 */
static void Close(void)
{
    FAT_fclose(OutFile);
}




/*======================================================================*
 =                                                                      =
 =                      Routines to write a bit-file                    =
 =                                                                      =
 *======================================================================*/

/*----------------------------------------------------------------------
 *
 *  NAME:           InitBitFile()
 *
 *  DESCRIPTION:    Initiate for using a bitfile. All output is sent to
 *                  the current OutFile using the I/O-routines above.
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        Nothing
 *
 */
static void InitBitFile(void)
{
    Buffer[Index = 0] = 0;
    BitsLeft = 8;
}




/*----------------------------------------------------------------------
 *
 *  NAME:           ResetOutBitFile()
 *
 *  DESCRIPTION:    Tidy up after using a bitfile
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        0 - OK, -1 - error
 *
 */
static int ResetOutBitFile(void)
{
    Byte numbytes;


    /*
     *  Find out how much is in the buffer
     */
    numbytes = Index + (BitsLeft == 8 ? 0 : 1);

    /*
     *  Write whatever is in the buffer to the file
     */
    if (numbytes) {
        if (WriteByte(numbytes) != GIF_OK)
            return -1;

        if (Write(Buffer, numbytes) != GIF_OK)
            return -1;

        Buffer[Index = 0] = 0;
        BitsLeft = 8;
    }

    return 0;
}
```

```
/*-------------------------------------------------------------------------
 *
 *  NAME:           WriteBits()
 *
 *  DESCRIPTION:    Put the given number of bits to the outfile.
 *
 *  PARAMETERS:     bits    - bits to write from (right justified)
 *                  numbits - number of bits to write
 *
 *  RETURNS:        bits written, or -1 on error.
 *
 */
static int WriteBits(int bits, int numbits)
{
    int  bitswritten = 0;
    Byte numbytes = 255;


    do {
        /*
         *  If the buffer is full, write it.
         */
        if ((Index == 254 && !BitsLeft) || Index > 254) {
            if (WriteByte(numbytes) != GIF_OK)
                return -1;

            if (Write(Buffer, numbytes) != GIF_OK)
                return -1;

            Buffer[Index = 0] = 0;
            BitsLeft = 8;
        }

        /*
         *  Now take care of the two specialcases
         */
        if (numbits <= BitsLeft) {
            Buffer[Index] |= (bits & ((1 << numbits) - 1)) << (8 - BitsLeft);
            bitswritten += numbits;
            BitsLeft -= numbits;
            numbits = 0;
        } else {
            Buffer[Index] |= (bits & ((1 << BitsLeft) - 1)) << (8 - BitsLeft);
            bitswritten += BitsLeft;
            bits >>= BitsLeft;
            numbits -= BitsLeft;

            Buffer[++Index] = 0;
            BitsLeft = 8;
        }
    } while (numbits);

    return bitswritten;
}




/*========================================================================*
 =                                                                        =
 =                Routines to maintain an LZW-string table                =
 =                                                                        =
 *========================================================================*/

/*-------------------------------------------------------------------------
```

```
 *
 *  NAME:           FreeStrtab()
 *
 *  DESCRIPTION:    Free arrays used in string table routines
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        Nothing
 *
 */
static void FreeStrtab(void)
{
    if (StrHsh) {
        free(StrHsh);
        StrHsh = NULL;
    }

    if (StrNxt) {
        free(StrNxt);
        StrNxt = NULL;
    }

    if (StrChr) {
        free(StrChr);
        StrChr = NULL;
    }
}




/*-------------------------------------------------------------------------
 *
 *  NAME:           AllocStrtab()
 *
 *  DESCRIPTION:    Allocate arrays used in string table routines
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        GIF_OK     - OK
 *                  GIF_OUTMEM - Out of memory
 *
 */
static int AllocStrtab(void)
{
    /*
     *  Just in case . . .
     */
    FreeStrtab();

    if ((StrChr = (Byte *) malloc(MAXSTR * sizeof(Byte))) == 0) {
        FreeStrtab();
        return GIF_OUTMEM;
    }

    if ((StrNxt = (Word *) malloc(MAXSTR * sizeof(Word))) == 0) {
        FreeStrtab();
        return GIF_OUTMEM;
    }

    if ((StrHsh = (Word *) malloc(HASHSIZE * sizeof(Word))) == 0) {
        FreeStrtab();
        return GIF_OUTMEM;
    }

    return GIF_OK;
}
```

```
/*----------------------------------------------------------------------
 *
 *  NAME:           AddCharString()
 *
 *  DESCRIPTION:    Add a string consisting of the string of index plus
 *                  the byte b.
 *
 *                  If a string of length 1 is wanted, the index should
 *                  be 0xFFFF.
 *
 *  PARAMETERS:     index - Index to first part of string, or 0xFFFF is
 *                          only 1 byte is wanted
 *                  b     - Last byte in new string
 *
 *  RETURNS:        Index to new string, or 0xFFFF if no more room
 *
 */
static Word AddCharString(Word index, Byte b)
{
    Word hshidx;


    /*
     *  Check if there is more room
     */
    if (NumStrings >= MAXSTR)
        return 0xFFFF;

    /*
     *  Search the string table until a free position is found
     */
    hshidx = HASH(index, b);
    while (StrHsh[hshidx] != 0xFFFF)
        hshidx = (hshidx + HASHSTEP) % HASHSIZE;

    /*
     *  Insert new string
     */
    StrHsh[hshidx] = NumStrings;
    StrChr[NumStrings] = b;
    StrNxt[NumStrings] = (index != 0xFFFF) ? index : NEXT_FIRST;

    return NumStrings++;
}




/*----------------------------------------------------------------------
 *
 *  NAME:           FindCharString()
 *
 *  DESCRIPTION:    Find index of string consisting of the string of index
 *                  plus the byte b.
 *
 *                  If a string of length 1 is wanted, the index should
 *                  be 0xFFFF.
 *
 *  PARAMETERS:     index - Index to first part of string, or 0xFFFF is
 *                          only 1 byte is wanted
 *                  b     - Last byte in string
 *
 *  RETURNS:        Index to string, or 0xFFFF if not found
 *
 */
static Word FindCharString(Word index, Byte b)
{
    Word hshidx, nxtidx;
    /*
     *  Check if index is 0xFFFF. In that case we need only
     *  return b, since all one-character strings has their
     *  bytevalue as their index
     */
    if (index == 0xFFFF)
        return b;

    /*
     *  Search the string table until the string is found, or
     *  we find HASH_FREE. In that case the string does not
     *  exist.
     */
    hshidx = HASH(index, b);
    while ((nxtidx = StrHsh[hshidx]) != 0xFFFF) {
        if (StrNxt[nxtidx] == index && StrChr[nxtidx] == b)
            return nxtidx;
        hshidx = (hshidx + HASHSTEP) % HASHSIZE;
    }

    /*
     *  No match is found
     */
    return 0xFFFF;
}




/*-------------------------------------------------------------------------
 *
 *  NAME:           ClearStrtab()
 *
 *  DESCRIPTION:    Mark the entire table as free, enter the 2**codesize
 *                  one-byte strings, and reserve the RES_CODES reserved
 *                  codes.
 *
 *  PARAMETERS:     codesize - Number of bits to encode one pixel
 *
 *  RETURNS:        Nothing
 *
 */
static void ClearStrtab(int codesize)
{
    int q, w;
    Word *wp;


    /*
     *  No strings currently in the table
     */
    NumStrings = 0;

    /*
     *  Mark entire hashtable as free
     */
    wp = StrHsh;
    for (q = 0; q < HASHSIZE; q++)
        *wp++ = HASH_FREE;

    /*
     *  Insert 2**codesize one-character strings, and reserved codes
     */
    w = (1 << codesize) + RES_CODES;
    for (q = 0; q < w; q++)
        AddCharString(0xFFFF, q);
}
```

```
/*========================================================================*
 =                                                                        =
 =                         LZW compression routine                        =
 =                                                                        =
 *========================================================================*/

/*-------------------------------------------------------------------------
 *
 *  NAME:           LZW_Compress()
 *
 *  DESCRIPTION:    Perform LZW compression as specified in the
 *                  GIF-standard.
 *
 *  PARAMETERS:     codesize  - Number of bits needed to represent
 *                              one pixelvalue.
 *                  inputbyte - Function that fetches each byte to compress.
 *                              Must return -1 when no more bytes.
 *
 *  RETURNS:        GIF_OK     - OK
 *                  GIF_OUTMEM - Out of memory
 *
 */
static int LZW_Compress(int codesize, int (*inputbyte)(void))
{
    register int c;
    register Word index;
    int  clearcode, endofinfo, numbits, limit, errcode;
    Word prefix = 0xFFFF;


    /*
     *  Set up the given outfile
     */
    InitBitFile();

    /*
     *  Set up variables and tables
     */
    clearcode = 1 << codesize;
    endofinfo = clearcode + 1;

    numbits = codesize + 1;
    limit = (1 << numbits) - 1;

    if ((errcode = AllocStrtab()) != GIF_OK)
        return errcode;
    ClearStrtab(codesize);

    /*
     *  First send a code telling the unpacker to clear the stringtable.
     */
    WriteBits(clearcode, numbits);

    /*
     *  Pack image
     */
    while ((c = inputbyte()) != -1) {
        /*
         *  Now perform the packing.
         *  Check if the prefix + the new character is a string that
         *  exists in the table
         */
        if ((index = FindCharString(prefix, c)) != 0xFFFF) {
            /*
             *  The string exists in the table.
             *  Make this string the new prefix.
             */
            prefix = index;

        } else {
            /*
             *  The string does not exist in the table.
             *  First write code of the old prefix to the file.
             */
            WriteBits(prefix, numbits);

            /*
             *  Add the new string (the prefix + the new character)
             *  to the stringtable.
             */
            if (AddCharString(prefix, c) > limit) {
                if (++numbits > 12) {
                    WriteBits(clearcode, numbits - 1);
                    ClearStrtab(codesize);
                    numbits = codesize + 1;
                }
                limit = (1 << numbits) - 1;
            }

            /*
             *  Set prefix to a string containing only the character
             *  read. Since all possible one-character strings exists
             *  int the table, there's no need to check if it is found.
             */
            prefix = c;
        }
    }

    /*
     *  End of info is reached. Write last prefix.
     */
    if (prefix != 0xFFFF)
        WriteBits(prefix, numbits);

    /*
     *  Write end of info -mark.
     */
    WriteBits(endofinfo, numbits);

    /*
     *  Flush the buffer
     */
    ResetOutBitFile();

    /*
     *  Tidy up
     */
    FreeStrtab();

    return GIF_OK;
}




/*========================================================================*
 =                                                                        =
 =                             Other routines                             =
 =                                                                        =
 *========================================================================*/

/*-------------------------------------------------------------------------
 *
 *  NAME:           BitsNeeded()
 *
 *  DESCRIPTION:    Calculates number of bits needed to store numbers
 *                  between 0 and n - 1
 *
 *  PARAMETERS:     n - Number of numbers to store (0 to n - 1)
 *
 *  RETURNS:        Number of bits needed
 *
 */
static int BitsNeeded(Word n)
{
    int ret = 1;


    if (!n--)
        return 0;

    while (n >>= 1)
        ++ret;

    return ret;
}





/*-------------------------------------------------------------------------
 *
 *  NAME:           InputByte()
 *
 *  DESCRIPTION:    Get next pixel from image. Called by the
 *                  LZW_Compress()-function
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        Next pixelvalue, or -1 if no more pixels
 *
 */
static int InputByte(void)
{
    int ret;


    if (RelPixY >= ImageHeight)
        return -1;

    ret = GetPixel(ImageLeft + RelPixX, ImageTop + RelPixY);

    if (++RelPixX >= ImageWidth) {
        RelPixX = 0;
        ++RelPixY;
    }

    return ret;
}





/*-------------------------------------------------------------------------
 *
 *  NAME:           WriteScreenDescriptor()
 *
 *  DESCRIPTION:    Output a screen descriptor to the current GIF-file
 *
 *  PARAMETERS:     sd - Pointer to screen descriptor to output
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_ERRWRITE - Error writing to the file
 *
 */
static int WriteScreenDescriptor(ScreenDescriptor *sd)
```

```
{
  return Write((void*)sd,sizeof(ScreenDescriptor));
}




/*-------------------------------------------------------------------------
 *
 *  NAME:           WriteImageDescriptor()
 *
 *  DESCRIPTION:    Output an image descriptor to the current GIF-file
 *
 *  PARAMETERS:     id - Pointer to image descriptor to output
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_ERRWRITE - Error writing to the file
 *
 */
static int WriteImageDescriptor(ImageDescriptor *id)
{
  return Write((void*)id,sizeof(ImageDescriptor));
}


/**************************************************************************
 *                                                                        *
 *                   P U B L I C    F U N C T I O N S                     *
 *                                                                        *
 **************************************************************************/


/*-------------------------------------------------------------------------
 *
 *  NAME:           GIF_Create()
 *
 *  DESCRIPTION:    Create a GIF-file, and write headers for both screen
 *                  and image.
 *
 *  PARAMETERS:     filename  - Name of file to create (including extension)
 *                  width     - Number of horisontal pixels on screen
 *                  height    - Number of vertical pixels on screen
 *                  numcolors - Number of colors in the colormaps
 *                  colorres  - Color resolution. Number of bits for each
 *                              primary color
 *
 *  RETURNS:        GIF_OK        - OK
 *                  GIF_ERRCREATE - Couldn't create file
 *                  GIF_ERRWRITE  - Error writing to the file
 *                  GIF_OUTMEM    - Out of memory allocating color table
 *
 */
int GIF_Create(char *filename, int width, int height,
               int numcolors, int colorres)
{
    int q, tabsize;
    Byte *bp;
    ScreenDescriptor SD;


    /*
     *  Initiate variables for new GIF-file
     */
    NumColors = numcolors ? (1 << BitsNeeded(numcolors)) : 0;
    BitsPrPrimColor = colorres;
    ScreenHeight = height;
    ScreenWidth = width;

    /*
     *  Create file specified
     */
    if (Create(filename) != GIF_OK)
        return GIF_ERRCREATE;

    /*
     *  Write GIF signature
     */
    if ((Write("GIF87a", 6)) != GIF_OK)
        return GIF_ERRWRITE;

    /*
     *  Initiate and write screen descriptor
     */
    SD.LocalScreenWidth = width;
    SD.LocalScreenHeight = height;

    SD.Flags=0x80 | (BitsNeeded(NumColors) - 1) | ((colorres-1)<<4);

    SD.BackgroundColorIndex = 0;
    SD.PixelAspectRatio = 0;
    if (WriteScreenDescriptor(&SD) != GIF_OK)
        return GIF_ERRWRITE;

    /*
     *  Allocate color table
     */
    if (ColorTable) {
        free(ColorTable);
        ColorTable = NULL;
    }
    if (NumColors) {
        tabsize = NumColors * 3;
        if ((ColorTable = (Byte *) malloc(tabsize * sizeof(Byte))) == NULL)
            return GIF_OUTMEM;
        else {
            bp = ColorTable;
            for (q = 0; q < tabsize; q++)
                *bp++ = 0;
        }
    }

    return 0;
}




/*-------------------------------------------------------------------------
 *
 *  NAME:           GIF_SetColor()
 *
 *  DESCRIPTION:    Set red, green and blue components of one of the
 *                  colors. The color components are all in the range
 *                  [0, (1 << BitsPrPrimColor) - 1]
 *
 *  PARAMETERS:     colornum - Color number to set. [0, NumColors - 1]
 *                  red      - Red component of color
 *                  green    - Green component of color
 *                  blue     - Blue component of color
 *
 *  RETURNS:        Nothing
 *
 */
void GIF_SetColor(u16 *pal)
{
  int i;
  Byte *p = ColorTable;
  for(i=0;i<256;i++) {
    *p++ = (pal[i]&0x001f)<<3;
    *p++ = (pal[i]>>3)&0xfc;
    *p++ = (pal[i]>>8)&0xf8;
  }
}




/*-------------------------------------------------------------------------
 *
 *  NAME:           GIF_CompressImage()
 *
 *  DESCRIPTION:    Compress an image into the GIF-file previousely
 *                  created using GIF_Create(). All color values should
 *                  have been specified before this function is called.
 *
 *                  The pixels are retrieved using a user defined callback
 *                  function. This function should accept two parameters,
 *                  x and y, specifying which pixel to retrieve. The pixel
 *                  values sent to this function are as follows:
 *
 *                    x : [ImageLeft, ImageLeft + ImageWidth - 1]
 *                    y : [ImageTop, ImageTop + ImageHeight - 1]
 *
 *                  The function should return the pixel value for the
 *                  point given, in the interval [0, NumColors - 1]
 *
 *  PARAMETERS:
 *                  width    - Width of the image, or -1 if as wide as
 *                             the screen
 *                  height   - Height of the image, or -1 if as high as
 *                             the screen
 *                  getpixel - Address of user defined callback function.
 *                             (See above)
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_OUTMEM   - Out of memory
 *                  GIF_ERRWRITE - Error writing to the file
 *
 */
int GIF_CompressImage(int width, int height,
                      int (*getpixel)(int x, int y))
{
    int codesize, errcode;
    ImageDescriptor ID;


    if (width < 0) {
        width = ScreenWidth;
    }
    if (height < 0) {
        height = ScreenHeight;
    }

    /*
     *  Write global colortable if any
     */
    if (NumColors)
        if ((Write(ColorTable, NumColors * 3)) != GIF_OK)
            return GIF_ERRWRITE;

    /*
     *  Initiate and write image descriptor
     */
    ID.Separator = ',';
    ID.LeftPosition = ImageLeft = 0;
    ID.TopPosition = ImageTop = 0;
    ID.Width = ImageWidth = width;
    ID.Height = ImageHeight = height;
    ID.Flags = 0;
    if (WriteImageDescriptor(&ID) != GIF_OK)
        return GIF_ERRWRITE;

    /*
     *  Write code size
     */
    codesize = BitsNeeded(NumColors);
    if (codesize == 1)
        ++codesize;
    if (WriteByte(codesize) != GIF_OK)
        return GIF_ERRWRITE;

    /*
     *  Perform compression
     */
    RelPixX = RelPixY = 0;
    GetPixel = getpixel;
    if ((errcode = LZW_Compress(codesize, InputByte)) != GIF_OK)
        return errcode;

    /*
     *  Write terminating 0-byte
     */
    if (WriteByte(0) != GIF_OK)
        return GIF_ERRWRITE;

    return GIF_OK;
}




/*-------------------------------------------------------------------------
 *
 *  NAME:           GIF_Close()
 *
 *  DESCRIPTION:    Close the GIF-file
 *
 *  PARAMETERS:     None
 *
 *  RETURNS:        GIF_OK       - OK
 *                  GIF_ERRWRITE - Error writing to file
 *
 */
int GIF_Close(void)
{
    ImageDescriptor ID;


    /*
     *  Initiate and write ending image descriptor
     */
    ID.Separator = ';';
    if (WriteImageDescriptor(&ID) != GIF_OK)
        return GIF_ERRWRITE;

    /*
     *  Close file
     */
    Close();

    /*
     *  Release color table
     */
    if (ColorTable) {
        free(ColorTable);
        ColorTable = NULL;
    }

    return GIF_OK;
```

```
}

//***************************************

static u8 *gif_savebuffer;
int getpix(int x,int y) {
  return gif_savebuffer[x+y*Y_RES];
}

int gif_save(char *name, u8 *buff, u16 *pal) {
  int err;

  gif_savebuffer=buff;
  err=GIF_Create(name, Y_RES, X_RES, 256, 8);  //filename, width, height, numcolors, bpp
  GIF_SetColor(pal);
  err|=GIF_CompressImage(-1, -1, getpix);
  err|=GIF_Close();

  return err==0;
}
```

(gpio.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

// GPIO input handling

#include <string.h>
#include <stdio.h>
#include "pen.h"

vu32 gpios;

void gpio_irq_handler() {
  //if(!(GPLR(GPIO51_nPIOW)&GPIO_bit(GPIO51_nPIOW))) {  //HWUART RTS
  //gpios|=GPIO_PGUP;
  //}
  //if(!(GPLR(GPIO50_nPIOR)&GPIO_bit(GPIO50_nPIOR))) {  //HWUART CTS
  //gpios|=GPIO_PGDN;
  //}
  if(!(GPLR(GPIO81_NSCLK)&GPIO_bit(GPIO81_NSCLK))) {  //NSSP CLOCK
    gpios|=GPIO_CLEAR;
  }
  setsig(SIG_GPIO);
  GEDR0=-1;     //clear all GPIO interrupts
  GEDR1=-1;
  GEDR2=-1;
}

//set GPIO function + direction + default state
void pxa_gpio_mode(int gpio_mode) {
  int gpio = gpio_mode & GPIO_MD_MASK_NR;
  int fn = (gpio_mode & GPIO_MD_MASK_FN) >> 8;
  int gafr;

  if (gpio_mode & GPIO_DFLT_LOW)
    GPCR(gpio) = GPIO_bit(gpio);
  else if (gpio_mode & GPIO_DFLT_HIGH)
    GPSR(gpio) = GPIO_bit(gpio);
  if (gpio_mode & GPIO_MD_MASK_DIR)
    GPDR(gpio) |= GPIO_bit(gpio);
  else
    GPDR(gpio) &= ~GPIO_bit(gpio);
  gafr = GAFR(gpio) & ~(0x3 << (((gpio) & 0xf)*2));
  GAFR(gpio) = gafr |  (fn  << (((gpio) & 0xf)*2));
}

//reset GPIO stuff
```

```
void gpio_init() {
  //PSSR |= PSSR_PH;  //GPIOs stay active in sleep mode

  GRER0=0; //disable all GPIO interrupts
  GRER1=0;
  GRER2=0;
  GFER0=0;
  GFER1=0;
  GFER2=0;
  GEDR0=-1;  //clear pending GPIO interrupts
  GEDR1=-1;
  GEDR2=-1;
}

void gpio_buttoninit() {
  gpios=0;

  //pageup button
  //pxa_gpio_mode(GPIO51_nPIOW | GPIO_IN);   //HWUART RTS
  //GFER(GPIO51_nPIOW)|=GPIO_bit(GPIO51_nPIOW);  //interrupt on falling edge

  //clear button
  pxa_gpio_mode(GPIO81_NSCLK | GPIO_IN);   //NSSP CLK
  GFER(GPIO81_NSCLK)|=GPIO_bit(GPIO81_NSCLK);  //interrupt on falling edge

  irq_enable(IRQ_GPIO_2_x);
}

/*
initialized by u-Boot:
6 MMCCLK
8 MMCCS0
12 32KHz out
15 2
18 1
23 1
24 1
25 1
26 2
28 AC97
29 AC97
30 AC97
31 AC97
42 BTUART RX
43 BTUART TX
44 BTUART CTS
45 BTUART RTS
46 STUART RX
47 STUART TX
50 2
53 MMCCLK
54 2
55 2
56 1
57 1
78 2
79 2
*/
```

(incbin.s)

```
; PEN (Personal Electronic Notebook)
; CS4710 Fall 2006
; Neal Tew
;-------------------------------

  AREA |.data|, DATA, READONLY

  EXPORT diskimage
  EXPORT diskimagesize
  EXPORT font
  EXPORT fontsize
font
  incbin verdana.ttf
fontsize
  dcd fontsize-font;
diskimage
  INCBIN 12M.ima
diskimagesize
  DCD diskimagesize-diskimage

  align 4
  EXPORT palette_inc
palette_inc
  INCBIN gfx\palette.bin

  include gfx\gfx.inc      ;include all graphics
;-------------------------------
  END
```

(interrupts.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

// some helper code for working with interrupts

#include <stdio.h>
#include "pen.h"

//divide-by-zero exception
void __rt_raise() {
  printf("\a\nDivide by zero!");
  while(1);
}
//CPU exceptions
char *d_err[]={
  "Vector","Alignment","Terminal","Alignment",
  "External","Translation","External","Translation",
  "External","Domain","External","Domain","External",
  "Permission","External","Permission"};
void data_abort() {
  int fault,faultaddr;
  __asm { mrc p15,0,fault,c5,c0; mrc p15,0,faultaddr,c6,c0 }
  printf("\a\nCrash! %s fault at 0x%08X",d_err[fault&15],faultaddr);
  while(1);
}
void prefetch_handler() {
  printf("\a\nCrash! Prefetch fault (bad PC).");
  while(1);
}
void undefined_handler() {
  printf("\a\nCrash! Undefined instruction.");
  while(1);
}

//main handler for all IRQs
int master_irq_handler() {
  int flags=ICIP;
  if(flags&BIT(IRQ_OST0)) {
    return 1;       //thread timeout is handled by threads.s
  }
  if(flags&BIT(IRQ_DMA))
    DMAirq();
  if(flags&BIT(IRQ_GPIO_2_x))
    gpio_irq_handler();
  if(flags&BIT(IRQ_HWUART))
    wacom_irq_handler();
  if(flags&BIT(IRQ_FFUART))
```

```
    ff_uart_irq_handler();
  if(flags&BIT(IRQ_LCD))
    lcd_irq_handler();
  if(flags&BIT(IRQ_OST1))
    flashtimer_irq_handler();
  if(flags&BIT(IRQ_USB))
    usb_irq_handler();
  return 0;
}

void irq_init() {
  ICMR=0;    //disable all interrupts
  ICLR=0;    //interrupts trigger IRQ (not FIQ)
  ICCR=1;    //cpu wakes up only on enabled interrupts
}

void irq_enable(int irqnum) {
  ICMR|=1<<irqnum;
}
```

(io_gstix.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

// Interface to flash memory for GBA_NDS_FAT driver

#include <string.h>
#include "io_gstix.h"
#include "pen.h"

//media inserted?
bool GSTIX_IsInserted (void) {
  return true;
}

/*-----------------------------------------------------------------
Tries to make the interface go back to idle mode (what?)
bool return OUT:  true if a CF card is idle
-----------------------------------------------------------------*/
bool GSTIX_ClearStatus (void) {
  //flush out the cache?
  return false;
}

/*-----------------------------------------------------------------
Read 512 byte sector numbered "sector" into "buffer"
u32 sector IN: address of first 512 byte sector on SD card to read
u8 numSecs IN: number of 512 byte sectors to read,
 1 to 256 sectors can be read, 0 = 256
void* buffer OUT: pointer to 512 byte buffer to store data in
bool return OUT: true if successful
-----------------------------------------------------------------*/
bool GSTIX_ReadSectors (u32 sector, u8 numSecs, void* buffer) {
  int size=numSecs ? numSecs*512 : 256*512;
  return flash_readcache(buffer,512*sector,size);
}

/*-----------------------------------------------------------------
Write 512 byte sector numbered "sector" from "buffer"
u32 sector IN: address of 512 byte sector on SC card to read
u8 numSecs IN: number of 512 byte sectors to read,
 1 to 256 sectors can be read, 0 = 256
void* buffer IN: pointer to 512 byte buffer to read data from
bool return OUT: true if successful
-----------------------------------------------------------------*/
bool GSTIX_WriteSectors (u32 sector, u8 numSecs, void* buffer) {
  int size=numSecs ? numSecs*512 : 256*512;
  return flash_writecache(512*sector,buffer,size);
}

//unload the interface
bool GSTIX_Shutdown(void) {
  return GSTIX_ClearStatus() ;
}

//initialize the interface
bool GSTIX_StartUp(void) {
  return true; //flash_init();
}

//the actual interface structure
IO_INTERFACE io_gstix = {
  DEVICE_TYPE_GSTIX,
  FEATURE_MEDIUM_CANREAD | FEATURE_MEDIUM_CANWRITE | FEATURE_SLOT_GBA,
  (FN_MEDIUM_STARTUP)&GSTIX_StartUp,
  (FN_MEDIUM_ISINSERTED)&GSTIX_IsInserted,
  (FN_MEDIUM_READSECTORS)&GSTIX_ReadSectors,
  (FN_MEDIUM_WRITESECTORS)&GSTIX_WriteSectors,
  (FN_MEDIUM_CLEARSTATUS)&GSTIX_ClearStatus,
  (FN_MEDIUM_SHUTDOWN)&GSTIX_Shutdown
};

//returns the interface structure to host
LPIO_INTERFACE GSTIX_GetInterface(void) {
  return &io_gstix ;
}
```

(io_gstix.h)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

// Interface to flash memory for GBA_NDS_FAT driver

#ifndef IO_GSTIX_H
#define IO_GSTIX_H

#define DEVICE_TYPE_GSTIX 0xaabbccdd

#include "disc_io.h"

// export interface
extern LPIO_INTERFACE GSTIX_GetInterface(void) ;

#endif
```

(lcd.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

// LCD initialization code

#include <string.h>
#include <stdio.h>
#include "pen.h"

#pragma arm section zidata="nocache"
u32 paldesc[4];
u32 framedesc[4];
u16 palette[256];
u8 framebuffer[FRAMESIZE]; //temp frame buffer for ???
#pragma arm section zidata

u8 *frame; //points to active framebuffer
```

```
volatile int vblanks;

void vblankwait() {
  u32 vbl=vblanks;
  do {
    wait(SIG_VBLANK);
  } while(vbl==vblanks);
}

void lcd_irq_handler() {
  int status=LCSR;
  if(status&LCSR_EOF) {    //end of frame
    vblanks++;
    LCSR=LCSR_EOF;
    setsig(SIG_VBLANK);
  } else if(status&LCSR_OU) {  //fifo underrun (starved for data)
    printf("DMA underrun!\n");
    LCSR=LCSR_OU;
  } else {      //??? unknown irq
    printf("LCDirq 0x%x\n",status);
    LCSR=-1;
  }
}

void lcd_init() {

  CKEN|=CKEN_LCD|CKEN_PWM1;  //LCD, PWM (backlight control) clock enable

  if(LCCR0&LCCR0_ENB) {  //LCD is already on?
    printf("^");    //debug... shutdown code hangs sometimes??
    LCCR0=LCCR0_QDM;  //disable LCD
    while(!(LCSR&LCSR_QD));  //wait til it's done shutting down
    LCSR=LCSR_QD;
    printf("^");
  }

  pxa_gpio_mode(GPIO17_PWM1_MD);
  PWM_CTRL1=0;    //PWM 3.xxMHz divisor
  PWM_PERVAL1=255;  //PWM resolution
  backlight(0);   //backlight off

  pxa_gpio_mode(GPIO58_LDD_0_MD);
  pxa_gpio_mode(GPIO59_LDD_1_MD);
  pxa_gpio_mode(GPIO60_LDD_2_MD);
  pxa_gpio_mode(GPIO61_LDD_3_MD);
  pxa_gpio_mode(GPIO62_LDD_4_MD);
  pxa_gpio_mode(GPIO63_LDD_5_MD);
  pxa_gpio_mode(GPIO64_LDD_6_MD);
  pxa_gpio_mode(GPIO65_LDD_7_MD);
  pxa_gpio_mode(GPIO66_LDD_8_MD);
  pxa_gpio_mode(GPIO67_LDD_9_MD);
  pxa_gpio_mode(GPIO68_LDD_10_MD);
  pxa_gpio_mode(GPIO69_LDD_11_MD);
  pxa_gpio_mode(GPIO70_LDD_12_MD);
  pxa_gpio_mode(GPIO71_LDD_13_MD);
  pxa_gpio_mode(GPIO72_LDD_14_MD);
  pxa_gpio_mode(GPIO73_LDD_15_MD);
  pxa_gpio_mode(GPIO74_LCD_FCLK_MD);
  pxa_gpio_mode(GPIO75_LCD_LCLK_MD);
  pxa_gpio_mode(GPIO76_LCD_PCLK_MD);
  pxa_gpio_mode(GPIO77_LCD_ACBIAS_MD);

  LCCR1= LCCR1_EndLnDel(4) | //front porch (1..256)
    LCCR1_HorSnchWdth(64) |  //HSYNC width (1..64)
    LCCR1_BegLnDel(232U) |//back porch (1..256)
    LCCR1_DisWdth(1024);//line width (1..1024)
  LCCR2= LCCR2_EndFrmDel(3) |//front porch (0..255)
    LCCR2_VrtSnchWdth(7) |//VSYNC width (1..64)
    LCCR2_BegFrmDel(29) |  //back porch (0..255)
    LCCR2_DisHght(768); //display height (1..1024)
  memcpy(&palette[0],&palette_inc,512); //use a fixed palette for now...

  //setup palette DMA descriptor
  paldesc[0]=(u32)&framedesc;    //next descriptor (16-byte align)
  paldesc[1]=(u32)&palette;    //frame source (8-byte align)
  paldesc[2]=0;       //frame ID
  paldesc[3]=256*2 | LDCMD_PAL;  //load to internal palette RAM

  //setup frame DMA descriptor
  framedesc[0]=(u32)&framedesc;  //next descriptor (16-byte align)
  //framedesc[0]=(u32)&paldesc;  //next descriptor (16-byte align)
  lcd_setframebuffer(framebuffer); //frame source (8-byte align)
  framedesc[2]=0;       //frame ID
  framedesc[3]=1024*768 | LDCMD_EOFINT; //trigger interrupt on frame end

  FDADR0=(int)&paldesc;      //first DMA starts from here

  LCCR3= LCCR3_VSP|LCCR3_HSP|LCCR3_PCP| //invert line, frame, pixel clocks
    LCCR3_Bpp(3) |    //8 bits per pixel
    LCCR3_PixClkDiv(0); //pixel clock divisor (~50MHz)

  LCSR=-1;   //clear any pending interrupts

  LCCR0= //end-of-frame, out-fifo underrun interrupts enabled
    LCCR0_Act |//active mode
    LCCR0_BM | //ignore branch start interrupt
    LCCR0_QDM |//ignore quick-disable interrupt
    LCCR0_LDM |//ignore normal-disable interrupt
    LCCR0_IUM |//ignore input underrun interrupt
    LCCR0_SFM |//ignore start-of-frame interrupt
    LCCR0_ENB; //enable LCD controller

  irq_enable(IRQ_LCD);
  backlight(255);   //backlight on
}

//tell LCD controller where to draw from
void lcd_setframebuffer(u8 *fb) {
  framedesc[1]=(u32)fb;
  frame=fb;
}

//255=full bright
void backlight(int bright) {

  msleep(10);//PWM pin screws up if you change it too fast

  //turn lamp off if bright=0
  //HWUART CTS is backlight enable
  pxa_gpio_mode(bright?
    GPIO50_nPIOR | GPIO_OUT | GPIO_DFLT_HIGH :
    GPIO50_nPIOR | GPIO_OUT | GPIO_DFLT_LOW
  );

  PWM_DUTY1=bright;
}
```

(main.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

// main program loop and some button handlers

#include <stdio.h>
#include <string.h>
#include "pen.h"
#include <fat.h>
```

```
void dash_cut(void);
void dash_clear(void);
void markpage(void);
void nextpage(void);
void prevpage(void);
void prevmark(void);
void nextmark(void);
void nothing(void);
void font_test(void);
void deletepage(void);

voidfn dash_jumptbl[]={
  0,
  next_background,  //background
  dash_clear,   //clear
  nothing, //new_page,   //ins blank
  nothing, //copy_page,    //ins copy
  nothing, //dash_cut,   //cut
  nothing, //paste_page,   //paste
  markpage,     //bookmark
  nothing, //deletepage,   //delete
  prevpage,
  prevmark,
  nextpage,
  nextmark };

buttonstruct dashboard[]={
  {0,0,        0,0, X_MAX, Y_MAX-100}, //first parameter is the drawing surface
  {BackgroundU, BackgroundP, 0,924, 0,0},
  {ClearU, ClearP,    0,974, 0,0},
  {InsBlankU, InsBlankP,   80,924, 0,0},
  {InsCopyU, InsCopyP,   80,974, 0,0},
  {CutU, CutP,      160,924, 0,0},
  {PasteU, PasteP,    160,974, 0,0},
  {BookmarkU, BookmarkP,   240,924, 0,0},
  {DeleteU, DeleteP,     240,974, 0,0},
  {BackU, BackP,      608,924, 0,0},
  {LastU, LastP,      608,974, 0,0},
  {NextU, NextP,      688,924, 0,0},
  {NextBU, NextBP,    688,974, 0,0},
  {0,}
};

void PENmain() {
  u8 *fb;  //frame buffer
  pen_event pen;

  //----------- system init, do these first

  sysclock_init();
  irq_init();
  gpio_init();
  threadtimer_init();

  //----------- init everything else

  ff_uart_init();   //do this first, so printf's will work

  printf("PEN BIOS (built " __DATE__ " " __TIME__ ") ");
  printf("."); flash_init();
  printf("."); if(!FAT_InitFiles()) printf("FAT_InitFiles() failed.\n");
  printf("."); load_settings();
  printf("."); lcd_init();
  printf("."); wacom_init();
  printf("."); font_init();
  printf("."); gpio_buttoninit();
  printf("."); pagemanager_init();
  printf("."); console_init();
  //printf("."); usb_init();
  printf("\nType '?' for help.\n>");
  //-----------

  fb=requestpage(book.firstpage);
  set_buttons(dashboard);
  redraw_status();
  lcd_setframebuffer(fb);
  while(1) {
    wait(SIG_GPIO|SIG_PEN);
/*
    if(gpios) {
      msleep(100); //debounce..
      gp=gpios;
      if(gp&GPIO_CLEAR)
        clearpage();
      if(gp&GPIO_PGUP)
        pageinc(1);
      if(gp&GPIO_PGDN)
        pageinc(-1);
      gpios=0;
      set_buttons(dash);
    }
*/
    while(get_pen_event(&pen)) {
      if(pen.event==DRAW_EVENT) {
        dirty();
        if(pen.erase) {
          line(pen.x0, pen.y0, pen.x1, pen.y1, erase);
        } else {
          if(settings.antialias)
            aline(pen.x0, pen.y0, pen.x1, pen.y1);
          else
            line(pen.x0, pen.y0, pen.x1, pen.y1, point);
        }
      } else if(pen.event==BUTTON_EVENT) {
        dash_jumptbl[pen.buttonID]();
      }
    }
  }
}

////////////////////////// button functions

void nextpage(void) {
  u8 *fb=pageinc(1);
  redraw_status();
  lcd_setframebuffer(fb);
}
void prevpage(void) {
  u8 *fb=pageinc(-1);
  redraw_status();
  lcd_setframebuffer(fb);
}

void prevmark(void) {
  u8 *fb;
  page *p=book.currentpage;
  while(p->prev) {
    p=p->prev;
    book.currentpagenumber--;
    if(p->bookmarked) break;
  }
  fb=requestpage(p);
  redraw_status();
  lcd_setframebuffer(fb);
}

void nextmark(void) {
  u8 *fb;
  page *p=book.currentpage;
  while(p->next) {
    p=p->next;
    book.currentpagenumber++;
    if(p->bookmarked) break;
  }
  fb=requestpage(p);
  redraw_status();
  lcd_setframebuffer(fb);
}

//toggle page bookmark
void markpage()  {
  book.currentpage->bookmarked^=1;
  book.dirty=1;
  redraw_status();
}
void deletepage() {
  u8 *fb;

  if(!yesno("Delete page:\0Are you sure?\0")) {
    redraw_status();
  } else {
    fb=delete_currentpage();
    redraw_status();
    lcd_setframebuffer(fb);
  }
}
void dash_clear() {
  u8 *old=frame;
  int yes;

  yes=yesno("Clear page:\0Are you sure?\0");
  if(!yes) {
    redraw_status();
    return;
  }

  frame=background;
  redraw_status();  //draw status area onto the _background_ first
  frame=old;

  DMAcopy(frame,background,FRAMESIZE);  //then copy the whole thing to visible page
  dirty();
}
void dash_cut() {
  u8 *fb;
  fb=cut_page();
  redraw_status();
  lcd_setframebuffer(fb);
}
void nothing() {
}
```

(misc.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

//miscellaneous routines I don't know where else to put

#include <stdio.h>
#include "pen.h"
#include "fat.h"

settings_struct settings;

void save_settings() {
  FAT_FILE *f=FAT_fopen("/SETTINGS","w");
  if(f) {
    FAT_fwrite(&settings,1,sizeof(settings),f);
    FAT_fclose(f);
  }
}

void load_settings() {
  FAT_FILE *f=FAT_fopen("/SETTINGS","r");
  if(f) {
    FAT_fread(&settings,1,sizeof(settings),f);
    FAT_fclose(f);
  } else {
    settings.antialias=1;
    settings.calibrated=0;
  }
}

void reboot(u32 addr) {
  if(LCCR0&LCCR0_ENB) {    //LCD shutdown
    printf("Stopping LCD..");
    LCCR0&=~LCCR0_ENB;
    while(!(LCSR&LCSR_QD));
    LCSR=LCSR_QD;
  }

  flash_readmode();
  CKEN|=CKEN_MMC;   //turn MMC clock on for bootloader
  ((voidfn)addr)();
}

#define DMAMAX 0x1ff8
vu32 dmadst; //cleared when dma is done
u32 dmasrc;
u32 dmasize;
void DMAirq() {
  int dmalen=dmasize;
  DCSR8=DCSR_NODESC|DCSR_ENDINTR; //no descriptor mode, clear interrupt status
  if(dmalen) {
    if(dmalen>DMAMAX) dmalen=DMAMAX;
    dmasize-=dmalen;
    DSADR8=dmasrc;
    DTADR8=dmadst;
    dmasrc+=dmalen;
    dmadst+=dmalen;
    DCMD8=DCMD_INCSRCADDR|DCMD_INCTRGADDR|DCMD_BURST32|DCMD_ENDIRQEN|dmalen;
    DCSR8=DCSR_NODESC|DCSR_RUN;
  } else { //last dma is finished
    DINT=0;
    dmadst=0;
    setsig(SIG_DMA);
  }
}

void DMAcopy(void *dst, void *src, int size) {
  int dmalen=size>DMAMAX?DMAMAX:size;

  irq_enable(IRQ_DMA);

  DINT=0x100;
  DCSR8=DCSR_NODESC|DCSR_ENDINTR; //no descriptor mode, clear interrupt status
  DSADR8=(u32)src;
  DTADR8=(u32)dst;
  DCMD8=DCMD_INCSRCADDR|DCMD_INCTRGADDR|DCMD_BURST32|DCMD_ENDIRQEN|dmalen;

  dmasrc=(u32)src+dmalen;
  dmadst=(u32)dst+dmalen;
  dmasize=size-dmalen;

  DCSR8=DCSR_NODESC|DCSR_RUN;

  do {
    wait(SIG_DMA);
  } while(dmadst);
}
```

```
//generate random unused filename
void tempfile(char *s) {
  FAT_FILE *f;
  while(1) {
    sprintf(s,"%X",OSCR);
    f=FAT_fopen(s,"r");
    if(!f) {
      return;
    } else {
      FAT_fclose(f);
    }
  }
}
```

(pagemanager.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

// page caching and "booklet" management code

#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "pen.h"
#include "fat.h"

void newbook(int newpages);
void pageload_thread(void);
void pagesave_thread(void);

#define BACKGROUNDS 3
#define DEFAULT_BG 1   //0=blank, 1=ruled, 2=graph

#define CACHESIZE 4    //pages to keep cached in ram
#define NEWBOOKSIZE 10   //default book size

typedef struct {
  page *page_p; //pointer to page data (0=unused)
  u32 dirty; //timestamp of when page was first dirtied (0=clean)
  u32 lastseen; //timestamp (for page replacement)
  u8 *buffer;
} pcache_entry;
pcache_entry pcache[CACHESIZE];

pcache_entry *visible;   //currently visible page (cache entry)

#pragma arm section zidata="nocache"
u8 background_buff[FRAMESIZE*BACKGROUNDS];
u8 pagebuffers[FRAMESIZE*CACHESIZE];
u8 clipboard[FRAMESIZE];
#pragma arm section zidata

int clipboard_full;

u8 *background;   //background of current page
u8 *bg_override;  //see loadbitmap()

page *requested;  //page request for pageload_thread (changed to page buffer address after
loading)
volatile int direction;  //pageup/pagedown direction (hint for precaching)

booklet book;   //one global booklet for now

//-------------------------------------------------------------

void draw_backgrounds(void);
void read_book(void);
void write_book(void);

//--------------------------------------------------------------

//make active page dirty
void dirty() {
  if(!visible->dirty)
    visible->dirty=RTC;
}

void pagemanager_init() {
  int i;

  clipboard_full=0;
  visible=0;
  requested=0;
  bg_override=0;

  draw_backgrounds();

  //clear the page cache
  for(i=0;i<CACHESIZE;i++) {
    pcache[i].page_p=0;
    pcache[i].lastseen=0;
    pcache[i].dirty=0;
    pcache[i].buffer=pagebuffers+FRAMESIZE*i;
  }

  read_book();  //read in our book (or create if it doesn't exist)

  //threads for saving/loading pages
  newthread((void*)pageload_thread,0);
  newthread((void*)pagesave_thread,0);
}

//------ backround stuff --------------------------------------------------------

//have all backgrounds pre-drawn, so we don't need to deal with it during page flipping
void draw_backgrounds() {
  int x,y;
  u8 *old=frame;
  memset(background_buff,WHITE,FRAMESIZE*BACKGROUNDS);

  //BG0: all white
  frame=background_buff;
  set_buttons(dashboard);    //draw buttons onto background

  //BG1: ruled paper
  frame=background_buff+FRAMESIZE;
  for(y=64;y<Y_RES;y+=26)
    for(x=0;x<X_RES;x++)
      putpix(x,y,BLUE);
  for(y=0;y<Y_RES;y++)
    putpix(80,y,RED);
  set_buttons(dashboard);

  //BG2: graph paper
  frame=background_buff+FRAMESIZE*2;
  for(y=0;y<Y_RES;y+=20)
    for(x=0;x<X_RES;x++)
      putpix(x,y,BLUE);
  for(x=0;x<X_RES;x+=20)
    for(y=0;y<Y_RES;y++)
      putpix(x,y,BLUE);
  set_buttons(dashboard);

  background=background_buff+FRAMESIZE*DEFAULT_BG;
  frame=old;
}

//copy background onto current frame (excluding drawn pixels)
```

```
void redraw_background() {
  u32 x,y;
  u32 *src=(u32*)background;
  u32 *dst=(u32*)frame;
  u32 d,s,mask;
  for(x=0;x<X_RES;x++) {
    for(y=0;y<(Y_RES-100)/4;y++) {
      d=*dst;
      s=*src++;
      mask=0;
      if(d&0x000000f0) mask=0x000000ff; //process 4 pixels together (faster)
      if(d&0x0000f000) mask|=0x0000ff00;
      if(d&0x00f00000) mask|=0x00ff0000;
      if(d&0xf0000000) mask|=0xff000000;
      s=(d&~mask)|(s&mask);
      *dst++ = s;
    }
    src+=100/4;//skip the button area
    dst+=100/4;
  }
}

//change the background of the current page
void next_background() {
  int i=book.currentpage->background+1;
  if(i>=BACKGROUNDS) i=0;
  book.currentpage->background=i;
  background=background_buff+i*FRAMESIZE;
  redraw_background();
  dirty();
  book.dirty=1;
}


//---booklet stuff---------------------------------------------------------------

//remove page from book and delete its .gif
//cache/display stuff is handled elsewhere
void delete_page(page *p) {
  page *next, *prev;
  next=p->next;
  prev=p->prev;
  if(next) next->prev=prev;
  if(prev) {
    prev->next=next;
  } else {
    book.firstpage=next;
  }
  if(FAT_remove(p->filename)) {
    printf("delete %s failed\n",p->filename);
  }
  free(p);
  book.pages--;
  book.dirty=1;
}

//insert new after p
void append_page(page *p, page *new) {
  new->next=p->next;
  new->prev=p;
  if(new->next)
    new->next->prev=new;
  p->next=new;
  book.pages++;
}

//insert new before p
void prepend_page(page *p, page *new) {
  new->next=p;
  new->prev=p->prev;
  p->prev=new;
  if(new->prev)
    new->prev->next=new;
  book.pages++;
}

static page *newpage() {
  page *p=malloc(sizeof(page));
  p->prev=0;
  p->next=0;
  p->background=DEFAULT_BG;
  p->bookmarked = 0;
  tempfile(p->filename);
  return p;
}

void newbook(int pages) {
  page *p, *new;
  int i;
  p=newpage();
  book.pages=1;
  for(i=1;i<pages;i++) {
    new=newpage();
    prepend_page(p,new);
    p=new;
  }
  book.currentpagenumber=0;
  book.firstpage=p;
  book.currentpage=p;
  book.dirty=1;
}

//delete all dynamically allocated memory associated with book
void freebook() {
  page *p, *next;
  p=book.firstpage;
  while(p) {
    next=p->next;
    free(p);
    p=next;
  };
  book.pages=0;
  book.firstpage=0;
  book.currentpage=0;
}

//book.inf struct {
//  u32 pages;
//  page pages[...];
//  u32 book_id;
//}
#define BOOK_ID 0x12344321 //guards against changing page struct size, corrupt files, etc

//write booklet INF file
void write_book() {
  page *p;
  FAT_FILE *f;
  u32 pages, id;
  f=FAT_fopen("BOOK.INF","w");
  if(!f)
    return;
  pages=book.pages;
  FAT_fwrite(&pages,1,4,f);
  p=book.firstpage;
  while(p) {
    FAT_fwrite(p, sizeof(page), 1, f);
    p=p->next;
  }
  id=BOOK_ID;
  FAT_fwrite(&id, 1, 4, f);
  FAT_fclose(f);
  book.dirty=0;
```

```
}

//read booklet INF
void read_book() {
  page *p, *new;
  FAT_FILE *f;
  u32 i, id, pages;
  f=FAT_fopen("BOOK.INF","r");
  if(!f) {        //missing INF file, make new book
    newbook(NEWBOOKSIZE);
  } else {
    FAT_fread(&pages,1,4,f);   //get page count
    p=newpage();
    FAT_fread(p, sizeof(page), 1, f); //assume there's always at least one page..
    p->prev=0;
    p->next=0;
    book.pages=1;
    book.firstpage=p;
    book.currentpage=p;
    book.currentpagenumber=0;
    for(i=1;i<pages;i++) {
      new=newpage();
      FAT_fread(new, sizeof(page), 1, f); //assume there's at least one page
      append_page(p,new);
      p=new;
    }
    FAT_fread(&id, 1, 4, f);
    FAT_fclose(f);
    if(id!=BOOK_ID) { //uh oh... something went wrong
      freebook();
      newbook(NEWBOOKSIZE);
      return;
    }
  }
}

//------------------------------------

//save a cache entry to disk
void savepage(pcache_entry *p) {
  char *s=p->page_p->filename;
  if(!gif_save(s,p->buffer,palette)) {
    printf("Write error (%s).\n",s);
  }
  p->dirty=0;
}

//watch for any offscreen dirty pages and save them
void pagesave_thread() {
  int i=0;
  while(1) {
    wait(SIG_ANY);
    if(pcache[i].page_p && &pcache[i]!=visible && pcache[i].dirty) {
      savepage(&pcache[i]);
    }
    i=(i==CACHESIZE-1) ? 0 : i+1;

    if(!i && book.dirty) { //take care of the booklet too... (background / bookmark settings)
      write_book();
    }
  }
}

//find a cache entry to replace.
//if everything's dirty, wait til one is clean first
pcache_entry *replacepage() {
  u32 oldest;
  int i,page;

  oldest=-1;
  do {
    for(i=0;i<CACHESIZE;i++) {
      if(!pcache[i].dirty && &pcache[i]!=visible) {  //not dirty, not active
        if(oldest>=pcache[i].lastseen) {  //find least recently used
          oldest=pcache[i].lastseen;
          page=i;
        }
      }
    }
    if(oldest==-1) {  //everything's dirty, wait for pagesave_thread to clean up
      wait(SIG_ANY);
    }
  } while(oldest==-1);

  if(oldest==-1) {
    printf("Can't find a free cache entry!?");
    while(1) wait(SIG_ANY);
  }

  pcache[page].page_p=0; //mark this page unused
  return &pcache[page];
}

//write everything to disk
void clean_all_pages() {
  int i=0;
  pcache_entry *tmp=visible;
  visible=0; //this allows pagesave_thread to save the active page
  do {   //wait til pagesave_thread cleans all pages
    if(pcache[i].dirty) {
      wait(SIG_ANY);
    } else {
      i++;
    }
  } while(i<CACHESIZE);
  visible=tmp;

  while(book.dirty);  //need to write this too..
}

//---------------------------------------------------------

//unpack a file to dst, make a blank page if file doesn't exist
int loadbitmap(char *filename, u8 *dst) {
  u8 *bg;
  if(!gif_read(filename,dst)) {//file doesn't exist? (or read error)
    bg=bg_override;   //override is used with page copy and paste to clone pages.
    if(!bg)       //if override isn't defined, go with the standard background
      bg=background;
    DMAcopy(dst,bg,FRAMESIZE); //make blank page
    bg_override=0;
    return 0;
  } else {
    return 1;
  }
}

//check if a page is cached, return its cache entry ptr
pcache_entry *findpage(page *p) {
  int i;
  for(i=0;i<CACHESIZE;i++) {
    if(p==pcache[i].page_p)
      return &pcache[i];
  }
  return 0;
}

//read a new page into the cache, if it's not there yet
//return the cache entry for the loaded page
pcache_entry *loadpage(page *pg) {
  pcache_entry *pc=findpage(pg);    //is page already in cache?
  if(!pc) {
    pc=replacepage(); //find a cache entry to replace
    loadbitmap(pg->filename, pc->buffer);
    pc->page_p=pg;
  }
  return pc;
}

//waits for main thread to request a new page.
void pageload_thread() {
  page *next;
  pcache_entry *pc;
  direction=1;
  while(1) {
    if(requested) { //main wants to switch pages?
      pc=loadpage(requested);
      //lcd_setframebuffer(p->buffer);  let main thread take care of this
      pc->lastseen=RTC;
      visible=pc;

      //find another page to preload
      if(direction>0)
        next=requested->next;
      else
        next=requested->prev;

      requested=0;  //let someone know the page is ready now

      //start grabbing the next page early
      if(next) {
        loadpage(next);
      }
    }
    wait(SIG_ANY);
  }
}

//--------- remaining functions are called from the main thread ---------------

//tell loader thread which page we need, and wait until it's ready.
//returns buffer of new page
u8 *requestpage(page *p) {
  background = background_buff + p->background * FRAMESIZE;  //update background ptr
  requested=p;
  do {
    wait(SIG_ANY);
  } while(requested);
  frame=visible->buffer;   //all gfx operations now go to the new page
  book.currentpage=p;
  //book.currentpagenumber must be changed elsewhere
  return frame;
}

//+1=pageup, -1=pagedown
//returns buffer of new page
u8 *pageinc(int increment) {
  page *p;
  direction=increment;//hint for precaching
  if(direction>0)
    p=book.currentpage->next;
  else
    p=book.currentpage->prev;
  if(p) {
    book.currentpagenumber+=increment;
    return requestpage(p);
  } else
    return visible->buffer;
}

//remove current page from book, and switch to next page
u8 *delete_currentpage() {
  page *p, *next;
  u8 *ret;

  p=book.currentpage;
  if(book.pages<2)    //keep at least one page
    return visible->buffer;

  //pull current page out of the cache
  visible->dirty=0;
  visible->page_p=0;

  //get next page
  if(p->next) {   //next page exists - move forward
    next=p->next;
  } else {   //on the last page - move backward
    next=p->prev;
    book.currentpagenumber--;
  }

  //delete the old page
  delete_page(p);

  //switch to next page
  ret=requestpage(next);

  return ret;
}

//insert a blank page into booklet and switch to it
void new_page() {
  u8 *fb;
  page *p;

  p=newpage();
  append_page(book.currentpage, p);
  p->background=book.currentpage->background;
  fb=requestpage(p);  //this causes a new page to be created, since file doesn't exist yet
  book.currentpagenumber++;
  redraw_status();
  lcd_setframebuffer(fb);
  book.dirty=1; //page DOESN'T need to be marked dirty.  if it's not saved, background will be
loaded automatically
}

//insert a copy of the current page and switch to it
void copy_page() {
  bg_override=visible->buffer;
  new_page();
  dirty();
}
void paste_page() {
  if(clipboard_full) {
    bg_override=clipboard;
    clipboard_full=0;
    new_page();
    dirty();
  }
}

//copy current page to clipboard, then delete it
u8 *cut_page() {
  DMAcopy(clipboard, frame, FRAMESIZE);
  clipboard_full=1;
  return delete_currentpage();
}

//print some useful stats
void pageinfo() {
  page *p;
  int i,dirty;

  printf("%i pages in book: ",book.pages);
  p=book.firstpage;
  i=0;
  if(p->prev) printf("!! Book consistency check failed !!\n");
  while(p) {
    i++;
    printf("%s ",p->filename);
    if(p->next && p->next->prev!=p)
      printf("!! Book consistency check failed !!\n");
    p=p->next;
  }
  if(i!=book.pages)
    printf("!! Book consistency check failed !!\n");
  printf("\nPage cache:\n");
  for(i=0;i<CACHESIZE;i++) {
    p=pcache[i].page_p;
    if(p) {
      dirty=pcache[i].dirty;
      printf("\t%s, %s\n",
        dirty?"dirty":"clean", p->filename);
    } else {
      printf("\tempty\n");
    }
  }
}
```

(pen.h)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

/*
 Throw everything in here so we don't have to manage a dozen header files.
 ..Normally not a big deal, but visual studio isn't smart enough
 to figure out header dependancies for the ARM compiler.
*/

#ifndef __PEN_H
#define __PEN_H

#include <asm/types.h>
#include <asm/arch/hardware.h>
#include <asm/arch/pxa-regs.h>
#include <asm/arch/bitfield.h>
#include <asm/arch/irqs.h>

#define BIT(x) (1<<(x))

#define X_MAX 767   //screen dimensions..
#define Y_MAX 1023
#define X_RES (X_MAX+1)    //yeah, I'm lazy
#define Y_RES (Y_MAX+1)

/////////////////// LCD.C

#define FRAMESIZE 1024*768
#define PORTRAIT_OFFSET(x,y) ((y)+(X_MAX-(x))*Y_RES) //get offset for portrait coords (rotate
90deg CCW)

extern u8 *frame;   //points to active framebuffer
extern volatile int vblanks;
extern u16 palette[256];
extern u8 framebuffer[FRAMESIZE];

void lcd_init(void);
void lcd_setframebuffer(u8 *fb);
void lcd_irq_handler(void);
void backlight(int bright);
void vblankwait(void);
```

```
/////////////////// FF_UART.C

void ff_uart_irq_handler(void);
void ff_uart_init(void);
int ff_uart_putc(int c);
int ff_uart_getc(void);

/////////////////// WACOM.C

//#define PEN_X_MAX 15980  //TC1100 digitizer range
//#define PEN_Y_MAX 21240
#define PEN_X_MAX 21760    //gateway digitizer range
#define PEN_Y_MAX 28800

enum {IN_RANGE=0x20, ERASER=4, PENDOWN=1, BUTTON=2};  //pen_flag bits

//for set_buttons()
typedef struct {
  u8 *up_bmp, *down_bmp; //bitmaps
  int x0,y0; //upper left
  int x1,y1; //lower right (filled automatically by set_buttons)
} buttonstruct;

extern buttonstruct dashboard[];  //dashboard buttons

//for get_pen_event()
enum { DRAW_EVENT, //pen is being dragged across drawing surface
  BUTTON_EVENT, //pen touched a button (check buttonID)
  NO_EVENT //pen touched an undefined area
};
typedef struct {
  int event;
  int erase; //nonzero for eraser
  int buttonID; //an index into buttonstruct, when button is pushed
  u32 x0,y0,x1,y1;
} pen_event;

//for wacom_read
typedef struct {
  int x,y,raw_x,raw_y,flags,btn;
} peninfo;

void wacom_init(void);
void wacom_irq_handler(void);
void wacom_flush(void);    //clear the pen queue
void wacom_calibrate(void);
int wacom_read(peninfo *p);     //get raw pen event
int get_pen_event(pen_event *p);  //get cooked pen event (check software buttons etc)
void set_buttons(buttonstruct *p);  //setup button regions

/////////////////// GPIO.C

extern vu32 gpios;
enum {GPIO_CLEAR=1, GPIO_PGUP=2, GPIO_PGDN=4};
void gpio_init(void);
void gpio_irq_handler(void);
void gpio_buttoninit(void);

/////////////////// TIMERS.C

#define MTICKS(x) (((x)*36864)/10)     //OS timer ticks in X msecs
#define UTICKS(x) (((x)*36864)/10000)    //OS timer ticks in X usecs

#define RTC RCNR         //RTC is a millisecond counter

void sysclock_init(void);
void threadtimer_init(void);
void msleep(u32 msecs);    //sleep for x millisecs
void flashmgr_irq_handler(void);
void flashtimer_irq_handler(void);
```

```
////////////////// INTERRUPTS.C

typedef void(*voidfn)(void);
void irq_init(void);
void irq_enable(int irqnum);

__inline int IRQS_OFF(void) {//for critical sections
  int oldPSR,newPSR;
  __asm {
    mrs oldPSR, CPSR
    orr newPSR, oldPSR, #0x80
    msr CPSR_c, newPSR
  }
  return oldPSR;
}

__inline void IRQS_SET(int oldPSR) {
  __asm {
    msr CPSR_c, oldPSR
  }
}

///////////////// CONSOLE.C

void console_init(void);

//////////////// PAGEMANAGER.C

extern u8 *background;   //current background buffer

typedef struct _page {
  char filename[128];
  u32 bookmarked;
  u32 background;
  struct _page *prev, *next;
} page;

typedef struct {
  u32 dirty;    //when book was last modified (0=clean)
  int currentpagenumber;//which page we're looking at now
  int pages;    //number of pages in book
  page *firstpage;
  page *currentpage;
} booklet;

extern booklet book;   //one global booklet for now

void next_background(void);  //change the background of the current page
void pagemanager_init(void);

u8 *delete_currentpage(void);
void copy_page(void);
void paste_page(void);
void new_page(void);
u8 *requestpage(page*);    //tell pagemanager to switch pages.
u8 *pageinc(int increment);  //pageup/pagedown (+1,-1).
        //** These functions just precache the page and return the frame buffer ptr.
        //** This lets buttons, etc be refreshed before the page is made visible.

void dirty(void);   //make active page dirty
void pageinfo(void);  //print some useful info to the console
void clean_all_pages(void);  //write all dirty pages to disk
extern u8 background_buff[];
extern int clipboard_full; //set after page cut
u8 *cut_page(void);    //copy current page to clipboard, then delete it


////////////////// FLASH

#define FATIMAGESIZE (12U*0x100000)
```

```
extern u8 flashcache[FATIMAGESIZE];   //make visible for USB code

void flash_init(void);
void flash_readmode(void);
void flash_format(void);
int flash_writeblock(u8 *src, u8 *dst);    //write one block (wait til flashing completes)
int flash_readcache(void *dst, u32 src, int size);
int flash_writecache(u32 dst, void *src, int size);
void flashinfo(void);
void flashall(void);

/////////////////// THREADS

#define SIG_ANY   -1
#define SIG_GPIO  (1<<0) //GPIO button was pushed
#define SIG_PEN   (1<<1) //wacom packet received
#define SIG_UART  (1<<2) //byte was received
#define SIG_TIMER (1<<3)
#define SIG_DMA   (1<<4) //DMAcopy finished
#define SIG_FLASH (1<<5) //flashing completed
#define SIG_VBLANK  (1<<6) //end of frame

#define MAXTHREADS 8
extern u32 currentthread;
extern struct {
  u32 regs[17];
  u32 prev, next, ticks, blocked, sig, id;
  u32 unused[32-23];
} threadstate[MAXTHREADS];

int newthread(void *func, void *args);//returns thread ID
void wait(u32 sig); //make thread wait for signal
#define setsig(x)
void lock(int*);
void unlock(int*);
void killthread(int id);
void suspend(void);
void resume(int id);

//////////////////////////////// USB

void usb_init(void);
void usb_irq_handler(void);

//////////////////////////////// MISC

typedef struct {
  int antialias;
  int calibrated;
  int px0,py0,px1,py1,dx0,dy0,dx1,dy1;  //calibration stuff
} settings_struct;
extern settings_struct settings;

void reboot(u32 addr);
void DMAcopy(void *dst, void *src, int size);
void DMAirq(void);
void save_settings(void);
void load_settings(void);

int gif_save(char *filename, u8 *buff, u16 *pal);
int gif_read(char *filename, u8 *buffer);
void tempfile(char *s);      //generate random unused filename

//////////////////////////////// GFX

#define STATUS_X 320//status area position
#define WHITE 0x3f
#define BLUE 0x3e
#define RED 0x3d
#define BLACK 0x3c
```

```
extern u8 palette_inc[];
#include "..\gfx\gfx.h"    //include all gfx externs

void draw_bmp(int x, int y, u8 *src);
void point(int x,int y);
void erase(int x,int y);
void putpix(int x,int y,char c);

typedef void(*plotfn)(int,int);
void line(int x0, int y0, int x1, int y1, plotfn p);
void aline (int X0, int Y0, int X1, int Y1);
void rect(int x0, int y0, int x1, int y1, int color);
int yesno(char *str);
void statustext(char *str);
void redraw_status(void);

///////////////////////////////// FONT

void font_init(void);
char *print(char *c, int x, int y);

////////////////////////////////////////////////////////

#endif //__PEN_H
```

(threads.s)

```
; PEN (Personal Electronic Notebook)
; CS4710 Fall 2006
; Neal Tew
;---------------------------------------------

; simple round-robin thread manager
;---------------------------------------------
  AREA |.text|, CODE

  IMPORT master_irq_handler
  IMPORT PENmain

  EXPORT swi_handler
  EXPORT irq_handler
  EXPORT threadmanager_init
  EXPORT newthread
  EXPORT threadstate
  EXPORT currentthread
  EXPORT killthread
  EXPORT wait
  EXPORT lock
  EXPORT unlock
  EXPORT suspend
  EXPORT resume

TIMEOUT equ ((10)*36864)/10  ;10 msec thread timeout

;--------------------------------
threadmanager_init
  ldr r1,=threadstate
  add r2,r1,#MAXTHREADS*THREADSTATESIZE

  str r1,[r1,#T_NEXT]    ;reset linked list
  str r1,[r1,#T_PREV]
  str r1,currentthread

  mov r0,#0
tmi0
  str r0,[r1,#T_USED]    ;mark all threads free
  str sp,[r1,#T_SP]    ;preallocate thread stacks
  sub sp,sp,#STACKSIZE
  add r1,r1,#THREADSTATESIZE
  cmp r1,r2
  bne tmi0
  ;launch PENmain() as the first thread

  ldr r0,=PENmain
  bl newthread
  mov r0,r1     ;r0=main thread struct
  msr cpsr_c,#0xd3  ;supervisor mode
  b runthread
;---------------------------------
newthread  ;(void *funcp, void *args)
    ;returns thread ID of new thread (0 on error)

  stmfd sp!,{r4-r7,lr}

  mrs r7,cpsr
  orr r3,r7,#0xc0
  msr cpsr_c,r3   ;block interrupts

  ;find a free thread struct

  ldr r4,=threadstate
  add r6,r4,#MAXTHREADS*THREADSTATESIZE
nt0    ldr r5,[r4,#T_USED]
    cmp r5,#0  ;thread unused?
    beq nt1
    add r4,r4,#THREADSTATESIZE
    cmp r4,r6
    bne nt0
  mov r0,#0     ;failed
  b nt9
nt1
  ;r4=new thread struct ptr
  ;set the initial cpu state

  str r1,[r4,#T_R0] ;args -> r0
  str r0,[r4,#T_PC] ;funcp -> PC
  mov r0,#0x1f
  str r0,[r4,#T_PSR]  ;system mode -> CPSR
  ldr r1,[r4,#T_SP]
  ldr r3,=STACKSIZE-1
  add r1,r1,r3
  bic r1,r1,r3
  str r1,[r4,#T_SP] ;reset stack -> SP
  adr r2,stopthread
  str r2,[r4,#T_LR] ;return to stopthread -> LR
  mov r2,#0
  str r2,[r4,#T_SUSPEND] ;thread is active

  ;assign a thread ID

  ldr r2,nextID
  add r1,r2,#1
  str r2,[r4,#T_ID]
  str r1,nextID

  ;insert thread into linked list, following currentthread

  ldr r0,currentthread;r0=current, r4=new
  ldr r1,[r0,#T_NEXT] ;r1=current.next
  str r0,[r4,#T_PREV] ;new.prev=current
  str r1,[r4,#T_NEXT] ;new.next=current.next
  str r4,[r1,#T_PREV] ;current.next.prev=new
  str r4,[r0,#T_NEXT] ;current.next=new

  mov r0,r2     ;return with r0=thread ID
  mov r1,r4     ;r1=thread struct
nt9 msr cpsr_c,r7   ;restore interrupts
  ldmfd sp!,{r4-r7,pc};return to caller
;--------------------------------
;Automatically called when a thread exits its main function.
;Can also be called directly, if a thread wants to kill itself
```

```
stopthread
  msr cpsr_c,#0xd3  ;block interrupts and go to supervisor mode

  ldr r3,currentthread

  mov r0,#0
  str r0,[r3,#T_USED] ;mark thread unused
  ldr r1,[r3,#T_PREV]
  ldr r0,[r3,#T_NEXT] ;r0=next, r1=prev
  str r0,[r1,#T_NEXT] ;update linked list..
  str r1,[r0,#T_PREV]

  str r0,currentthread
  b runthread
;-------------------------------
;look up struct from ID
;r0=TID, returns threadstruct (0=fail)
findthread
  mov r12,r0
  mov r0,#0
  ldr r2,=threadstate
  add r3,r2,#MAXTHREADS*THREADSTATESIZE
  b kt0
kt1    add r2,r2,#THREADSTATESIZE
    cmp r2,r3
    bxeq lr
kt0    ldr r1,[r2,#T_USED]
    cmp r1,#0  ;thread used?
    beq kt1
    ldr r1,[r2,#T_ID]
    cmp r1,r12 ;IDs match?
    bne kt1
  mov r0,r2
  bx lr
;-------------------------------
killthread      ;forcibly end another thread
  stmfd sp!,{r4,lr}

  mrs r4,cpsr   ;keep old PSR
  orr r1,r4,#0xc0   ;disable interrupts
  msr cpsr_c,r1

  bl findthread
  cmp r0,#0
  beq kt9

  mov r3,#0
  str r3,[r0,#T_USED] ;mark thread unused
  ldr r1,[r0,#T_PREV]
  ldr r3,[r0,#T_NEXT] ;r0=next, r1=prev
  str r3,[r1,#T_NEXT] ;update linked list..
  str r1,[r3,#T_PREV]
kt9
  msr cpsr_c,r4   ;restore interrupts
  ldmfd sp!,{r4,pc}
;---------------------------------
irq_handler
  sub lr,lr,#4
  stmfd sp!,{r0-r3,r12,lr}

  bl master_irq_handler

  cmp r0,#0
  ldmeqfd sp!,{r0-r3,r12,pc}^     ;normal irq was handled, exit

  ;thread timed out...

OSMR0  equ 0x00
OSSR equ 0x14
OSCR equ 0x10
```

```
nextthread

  ldr r3,=0x40a00000
  ldr r0,[r3,#OSCR]
  ldr r1,=TIMEOUT
  add r0,r0,r1
  str r0,[r3,#OSMR0]     ;OSMR0=OSCR+TIMEOUT
  mov r0,#1
  str r0,[r3,#OSSR]   ;OSSR=1 - clear interrupt

  ;todo:  if all threads are blocked, go to sleep

  ldr r1,currentthread
  mov r0,r1

nt2    ldr r0,[r0,#T_NEXT]    ;find next unsuspended thread
    ldr r2,[r0,#T_SUSPEND]
    cmp r2,#0
    bne nt2
    str r0,currentthread   ;current=next

  ;save current thread state:
  mrs r3,spsr
  stmia r1,{r3-r11,r13-r14}^ ;save psr,r4-11,r13-14
  add r1,r1,#11*4
  ldmfd sp!,{r4-r7,r12,lr} ;restore regs saved at top of interrupt
  stmia r1,{r4-r7,r12,lr}    ;save r0-r3,r12,pc

  ldr r1,[r0,#T_TIME] ;keep track of how long thread's been alive (for debugging purposes..)
  add r1,r1,#1
  str r1,[r0,#T_TIME]

runthread     ;r0=threadstruct
  ldmia r0,{r3-r11,r13-r14}^
  msr spsr_cxsf,r3
  add r0,r0,#11*4
  ldmia r0,{r0-r3,r12,pc}^

;signal interrupt???
; ldr r0,[rx,#T_SIGNAL]
; ldr r1,[rx,#T_SIGNAL_TRAP]
; ands r0,r0,r1
; bne interruptthread
;interruptthread
; clz r0,r0
; add ???,???,???,lsl#???
; ldr r0,[rx,rx,#T_SIGHANDLERS]

  ;push r0-r3,r12 onto thread's stack
  ;lr=pc
  ;pc=handler
;-------------------------------
;unsuspend a thread.  r0=TID
resume
  stmfd sp!,{lr}
  bl findthread
  mov r1,#0
  cmp r0,#0
  strne r1,[r0,#T_SUSPEND]
  ldmfd sp!,{pc}
;-------------------------------
suspend  ;put self to sleep indefinitely (resume from another thread)
  ldr r1,currentthread
  mov r0,#-1
  str r0,[r1,#T_SUSPEND]
;-----------------------------------
wait ;make thread wait for a signal:
  ;if signal is already set, return immediately
  ;otherwise move self into blocked thread list
  ;signal mask is in r0; returns with signals in r0?
  ;mov r0,#1
  ;mcr p14,0,r0,c7,c0 ;cpu idle

  swi 0

  bx lr
;-------------------------------------
swi_handler
  stmfd sp!,{r0-r3,r12,lr}
  b nextthread
;----------------------------------------
;semaphore stuff ... wait til word is available, then take control of it
lock
  ;do {
  ; temp=1
  ; swap(v,t)
  ; if(temp)
  ;   wait
  ;} while(temp)

  stmfd sp!,{r4,lr}
  mov r4,r0
lockwait
  mov r1,#1
  swp r1,r1,[r4]
  cmp r1,#0
  ldmeqfd sp!,{r4,pc}

  mov r0,#-1 ;SIG_LOCK ??
  bl wait
  b lockwait
;---------------------------
; release semaphore
unlock
  mov r1,#0
  str r1,[r0]
  bx lr
;-------------------------------------------------------
  AREA |.text|, DATA

currentthread
  dcd 0      ;points to a thread struct in threadstate
nextID
  dcd 1      ;next thread ID that will be assigned
;-------------------------------------------------------
  AREA |.bss|, NOINIT

MAXTHREADS    equ 8
THREADSTATESIZE   equ 0x80
STACKSIZE     equ 0x10000

T_PSR  equ  0
T_USED equ  T_PSR    ;0=thread unused
T_R4 equ  T_PSR+4
T_R5 equ  T_R4+4
T_R6 equ  T_R5+4
T_R7 equ  T_R6+4
T_R8 equ  T_R7+4
T_R9 equ  T_R8+4
T_R10  equ  T_R9+4
T_R11  equ  T_R10+4
T_SP equ  T_R11+4    ;r13=sp
T_LR equ  T_SP+4   ;r14=lr
T_R0 equ  T_LR+4
T_R1 equ  T_R0+4
T_R2 equ  T_R1+4
T_R3 equ  T_R2+4
T_R12  equ  T_R3+4
T_PC equ  T_R12+4
T_PREV equ  T_PC+4
T_NEXT equ  T_PREV+4
```

```
T_TIME equ  T_NEXT+4
T_BLOCKED equ T_TIME+4 ;signal bitmask
T_SIG  equ  T_BLOCKED+4
T_ID equ  T_SIG+4    ;unique thread ID number
T_SUSPEND equ T_ID+4   ;1=suspended
;T_
;T_
;T_
;T_
;T_
;T_
;T_
;T_

threadstate
  % MAXTHREADS*THREADSTATESIZE
;--------------------------
  END
```

(timers.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

#include <stdio.h>
#include "pen.h"

//set up system clocks (and memory waitstates)
void sysclock_init() {
  u32 i;

  //mimic u-boot settings.
  //only do this while running inside SDRAM?
  MSC0=0x128c26ab;  //set flash memory timings
  MSC1=0x0000128c;
  MSC2=0x7ff07ff0;
  i=MSC0;
  i=MSC1;
  i=MSC2;
  MECR=0;      //no CF interface

  CKEN=0;    //shut off peripheral clocks (save power)
      //turn each periph.clock back on in its respective init routine
  OIER=0;    //disbale OS timer interrupts
  OSSR=-1; //clear timer interrupt flags

  RTC=1000*600; //set clock to 10min (simplifies flashall..)
  RTTR=31; //change RTC to a millisecond counter
}

void threadtimer_init() {
  OIER|=1;     //enable timer 0,3
  OSMR0=OSCR+MTICKS(10);   //timeout here is arbitrary ... the real thread time is set in
threads.s
  irq_enable(IRQ_OST0);

  //OSMR3=OSCR+MTICKS(10000);
  //OWER=1;         //enable watchdog!
}

//wait (at least) N milliseconds
void msleep(u32 msecs) {
  u32 stoptime=msecs+RTC;
  do {
    wait(SIG_ANY);
  } while(RTC<stoptime);
}
```

(usb.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

// USB driver code for mimicking a mass storage device

#include <string.h>
#include <stdio.h>
#include "pen.h"

#define BIGEND(x) ((((x)&0xff000000)>>24) |(((x)&0xff0000)>>8) | (((x)&0xff00)<<8) |
(((x)&0xff)<<24))

enum { EP0_IDLE, EP0_IN_DATA_PHASE, EP0_END_XFER, EP0_OUT_DATA_PHASE};
int usb_state;

typedef struct {
  u32 sig;
  u32 tag;
  u32 length;//expected bulk transfer size
  u8 flags;  //0x80: host to device (OUT), 0x00: device to host (IN)
  u8 LUN;
  u8 CBlength;
  u8 CB[16];
  u8 align;
} CBW_struct;

struct {
  u32 sig; //0x53425355
  u32 tag; //copy of CBW tag
  int residue; //difference between expecetd length & processed length
  u8 status; //0=ok, 1=fail, 2=phase error
} CSW;

struct {
  u8 type;
  u8 request;
  u16 value;
  u16 index;
  u16 length;
  u8 unused[8];
} setup;

u32 rx_idx;//EP2 assumes this gets reset to 0
u8 rx_buff[1024];
u8 tx_buff[1024]; //EP1 xmit buffer

enum {GET_DESCRIPTOR=6, SET_CONFIGURATION=9};       //setup request types
enum {DEVICE=1, CONFIGURATION, STRING};    //descriptor types

//void *descriptors[]={ 0, device_descriptor, config_descriptor };
char device_descriptor[]={
  18,  //length
  DEVICE,  //type
  0x10,0x01,//1.10 USB spec
  0,//class (specified in interface descriptor)
  0,//subclass
  0,//protocol
  16,  //EP0 packetsize
  0x11,0x11, //vendor ID
  0x22,0x22, //product ID
  1,0, //device release number
  1,//manufacturer string
  2,//product string
  3,//serial number string
  1 //number of configurations
};
char config_descriptor[]={
  9,//desc.size
  2,//desc.type
  9+9+7+7, 0,//total config size (config+IF+ep descriptors)
  1,//interfaces
  1,//this configuration's id?
  4,//config string index
  0xc0,  //attributes (self-powered)
  25,  //power requirement (*2mA)

  //interface descriptor

  9,//size
  4,//desc.type
  1,//IF number
  0,//alt setting
  2,//endpoints (excluding EP0)
  8,//mass storage class
  6,//mass storage subclass (SCSI transparent)
  0x50,  //bulk-only transport
  5,//interface string

  //EP1 descriptor (bulk IN x64)

  7,//size
  5,//desc.type
  0x81,  //EP address
  2,//attribute
  64,0,  //packet size
  0,//unused

  //EP2 descriptor (bulk OUT x64)

  7,//size
  5,//desc.type
  0x02,  //EP address
  2,//attribute
  64,0,  //packet size
  0,//unused
};
char langID[]={ 4, 3, 9, 4 }; //supported languages (en-us = 0x0409)
char manufacturer[]={ 10, 3, 'M',0,'a',0,'n',0,'u',0 };
char product[]={ 20, 3, 'P',0,'E',0,'N',0,' ',0,'D',0,'r',0,'i',0,'v',0,'e',0 };
char serial_str[]={ 26, 3,
'3',0,'3',0,'3',0,'3',0,'3',0,'3',0,'3',0,'3',0,'3',0,'3',0,'3',0,'3',0 };
char config_str[]={ 10, 3, 'C',0,'o',0,'n',0,'f',0 };
char if_str[]={ 10, 3, 'I',0,'n',0,'t',0,'f',0 };
char *string_descriptor[]={
  langID,
  manufacturer,
  product,
  serial_str,
  config_str,
  if_str
};

void ep1_send(u8 *p, int size);
void ep2_process(void);

//if p is nonzero, send a new packet; otherwise continue sending the previous one
void ep0_send(char *p, int size, int max) {
  int sent;
  static char *src;
  static int remaining;
  if(p) {
    src=p;
    if(!size)  //figure out size if it wasn't specified
      size=p[0];
    remaining=(size>max?max:size);
    usb_state=EP0_IN_DATA_PHASE;
  }
  sent=0;
  while(remaining && sent<16) {
    UDDR0=*src++;
    sent++;
    remaining--;
  }
  printf("x%i ",sent);
  if(sent<16) {
    UDCCS0=UDCCS0_IPR;
    usb_state=EP0_END_XFER;
  }
}

void setup_standard() {
  if(setup.type==0x80) { //host is asking us for data
    printf("SI ");  //setup in
    switch(setup.request) {
      case GET_DESCRIPTOR:
        switch(setup.value>>8) {
          case DEVICE:
            printf("getdevice ");
            ep0_send(device_descriptor, 0, setup.length);
            break;
          case CONFIGURATION:
            printf("getconfig ");
            ep0_send(config_descriptor, sizeof(config_descriptor), setup.length);
            break;
          case STRING:
            printf("getstring(%i) ",setup.value&0xff);
            ep0_send(string_descriptor[setup.value&0xff], 0, setup.length);
            break;
          default:
            printf("bad desc ");
        }
        break;
      default:
        printf("bad req ");
    }
  } else if(setup.type==0x00 && setup.request==9) {
    printf("set_cfg "); //no response needed
    rx_idx=0;
    usb_state=EP0_OUT_DATA_PHASE;
  } else {
    UDCCS0=UDCCS0_FST;  //send stall
    printf("stall ");
  }
}

void setup_class() {
  char maxLUN=0;
  if(setup.type==0xa1 && setup.request==0xfe) {  //get max LUN
    printf("getmaxLUN ");
    ep0_send(&maxLUN,1,1);
  } else {
    UDCCS0=UDCCS0_FST;  //send stall
    printf("stall ");
  }
}

void usb_irq_handler() {
  int i;
  u8 c;
  i=UDCCR;
  if(i & (UDCCR_RSTIR | UDCCR_SUSIR | UDCCR_RESIR)) {
    if(i & UDCCR_RSTIR) {  //reset interrupt
      rx_idx=0;
      printf("USB reset\n");
    } else if(i & UDCCR_SUSIR) { //suspend interrupt
    } else if(i & UDCCR_RESIR) { //resume interrupt
    }
    UDCCR=i; //clear interrupt
  }
  if(USIR0 & USIR0_IR0) {  //endpoint 0:
     if(UDCCS0 & UDCCS0_OPR) {  //rx fifo has data
       if(UDCCS0 & UDCCS0_SA) { //this is a SETUP packet
         usb_state=EP0_IDLE;
         i=0;
         while(UDCCS0 & UDCCS0_RNE) { //pull data til FIFO is emptied
           ((u8*)(&setup))[i++]=UDDR0;
         }
         printf("\n(%i,%i,%i,%i,%i)
",setup.type,setup.request,setup.value,setup.index,setup.length);
         switch(setup.type&0x60) {
           case 0x00: setup_standard(); break;
           case 0x20: setup_class();    break;
           //case 0x40: setup_vendor();   break;
           default:
             UDCCS0=UDCCS0_FST;  //send stall
             printf("stall ");
         }
         UDCCS0=UDCCS0_SA; //ack setup packet
       } else if(usb_state==EP0_OUT_DATA_PHASE) { //reading setup OUT packet
         i=0;
         while(UDCCS0 & UDCCS0_RNE) { //pull data til FIFO is empty
           i++;
           c=UDDR0;
           if(rx_idx<sizeof(rx_buff))
             rx_buff[rx_idx++]=c;
         }
         printf("o%i ",i);
         UDCCS0=UDCCS0_IPR;  //send status in
       } else { //host sent status out
         if(usb_state==EP0_END_XFER) {
           printf(".");
         } else {   //premature status? flush tx fifo
           printf("*");
           UDCCS0=UDCCS0_FTF;
         }
         usb_state=EP0_IDLE;
       }
       UDCCS0=UDCCS0_OPR;     //rx ack
    } else if(usb_state==EP0_OUT_DATA_PHASE) { //status in (done reading OUT packet)
       printf("STI ");
       if(rx_idx==setup.length) { //packet is good?
         printf("OUT ");
         //go handle the packet..
       } else {
         printf("?3");
       }
       rx_idx=0;     //done handling packet, reset size to 0
       usb_state=EP0_IDLE;
    } else if(usb_state==EP0_IN_DATA_PHASE) {  //still sending data
       ep0_send(0,0,0);
    } else {
       printf("^%x ",UDCCS0);
    }
    USIR0=USIR0_IR0;  //irq ack
  }
  if(USIR0 & USIR0_IR1) {  //endpoint 1: tx complete
    ep1_send(0,0);    //send something if we need to
    UDCCS1=UDCCS_BI_TPC;//ep1 ack
    USIR0=USIR0_IR1;  //irq ack
  }
  if(USIR0 & USIR0_IR2) {  //endpoint 2:
    if(UDCCS2 & UDCCS_BO_RNE) {  //need to pull data from FIFO
       printf("e%i ",UBCR2+1);
       for(i=UBCR2;i>=0;i--) {
         c=UDDR2;
         if(rx_idx<sizeof(rx_buff))
           rx_buff[rx_idx++]=c;
       }
    }
    if(UDCCS2 & UDCCS_BO_RSP) {  //short packet (i.e. the last packet)
       ep2_process();
    }
    UDCCS2=UDCCS_BO_RPC;//ep2 irq ack
    USIR0=USIR0_IR2;  //irq ack
  }
}

void usb_init() {
  CKEN|=CKEN_USB;   //USB clock enable

  UDCCFR=  UDCCFR_MB1;//respond to SET_CONFIG & SET_INTERFACE automatically
//UDCCFR=  UDCCFR_MB1| UDCCFR_ACM;  //don't respond to SET_CONFIG & SET_INTERFACE automatically
(warn us first)

  UDCCS0=UDCCS0_FTF;  //EP0: flush FIFO
  UDCCS1=UDCCS_BI_FTF;//EP1: flush FIFO
  UDCCS2=0;    //EP2: no DMA

  USIR0=0xff;  //clear EP interrupts
  USIR1=0xff;
  UICR0=0xf8;  //EP0-2 interrupt enable
  UICR1=0xff;  //EP8-15 interrupt mask

  UDCCR= UDCCR_SRM |//mask suspend/resume interrupt
    UDCCR_RSTIR | UDCCR_SUSIR | UDCCR_RESIR | //clear interrupts
    UDCCR_UDE; //USB enable

  irq_enable(IRQ_USB);

  usb_state=EP0_IDLE;
}

/*------ BULK IN / BULK OUT interface ----------------------
**  process CBW's, etc
*/

//if p is nonzero, send a new packet; otherwise continue sending the previous one
void ep1_send(u8 *p, int size) {
  #define EP1_MAX 64

  CBW_struct *cbw=(CBW_struct*)rx_buff; //command block wrapper
  int sent;
  static u8 sentstatus;
  static u8 *src;
  static int remaining;

  if(p) {
    if(size>cbw->length)
      size=cbw->length;
    CSW.residue=cbw->length-size;
    sentstatus=0;
    src=p;
    remaining=size;
  } else {
    if(!remaining) {  //send status as final packet
      if(!sentstatus) {
        sentstatus=1;
        src=(u8*)&CSW;
        remaining=13; //sizeof(CSW)
      } else {
        return;
      }
    }
  }

  //check if fifo is ok to receive?
  sent=0;
  while(remaining && sent<EP1_MAX) {
    UDDR1=*src++;
    sent++;
    remaining--;
  }
  if(sent<EP1_MAX) {
    UDCCS1=UDCCS_BI_TSP;
    printf("@");
  }
}

#define BLOCKSIZE 2048
//rx_buff is ready for processing
void ep2_process() {
  int LBA,blocks;
  CBW_struct *cbw=(CBW_struct*)rx_buff; //command block wrapper
  u8 *cdb=(u8*)&(cbw->CB[0]);  //SCSI command descriptor
  if(cbw->sig==0x43425355 && rx_idx==31) { //this is a valid CBW packet
    CSW.sig=0x53425355; //prepare status pkt
    CSW.tag=cbw->tag;
    CSW.status=0;
    memset(tx_buff,0,64);
    switch(cdb[0]) {  //SCSI opcode:
      case 0x12:      //INQUIRY
        printf("inquiry ");
        //tx_buff[0]=0;   //direct-access device
        //tx_buff[1]=0;   //non-removable media
        tx_buff[2]=4;   //supports SBC-2 command set
        tx_buff[4]=36-4;  //remaining length of inquiry response length
        ep1_send(tx_buff,36);
        break;
      case 0x23:      //MMC READ FORMAT CAPACITIES
        printf("readFMT ");
        tx_buff[3]=16;
        ((u32*)tx_buff)[1]=BIGEND(FATIMAGESIZE/BLOCKSIZE);  //num blocks
        ((u32*)tx_buff)[2]=BIGEND(BLOCKSIZE) | 2;
        ((u32*)tx_buff)[3]=BIGEND(FATIMAGESIZE/BLOCKSIZE);
        ((u32*)tx_buff)[4]=BIGEND(BLOCKSIZE);
        ep1_send(tx_buff,20);
        break;
      case 0x28:    //READ
        LBA=((cdb[3]<<16) | (cdb[4]<<8) | cdb[5])*BLOCKSIZE;
        blocks=((cdb[7]<<8) | cdb[8])*BLOCKSIZE;
        //if((LBA+blocks)>FATIMAGESIZE)
        //return error status
        printf("read(x%x,x%x) ",LBA,blocks);
        ep1_send(flashcache+LBA, blocks);
        break;
      case 0x25:    //READ CAPACITY
        printf("readCAP ");
        ((u32*)tx_buff)[0]=BIGEND(FATIMAGESIZE/BLOCKSIZE-1);
        ((u32*)tx_buff)[1]=BIGEND(BLOCKSIZE);
        ep1_send(tx_buff,8);
        break;
      case 0x1A:    //MODE SENSE
        if(cdb[2]==0x1c) {    //page 1C: informational exceptions control mode
          printf("sense1C ");
          //mode parameter header
            ((u32*)tx_buff)[0]=0x08000017;
          //block descriptor
            ((u32*)tx_buff)[1]=BIGEND(FATIMAGESIZE/BLOCKSIZE);
            ((u32*)tx_buff)[2]=BIGEND(BLOCKSIZE);
          //mode page
            ((u32*)tx_buff)[3]=0x00000a1c;
            ((u32*)tx_buff)[4]=BIGEND(0); //report count
            ((u32*)tx_buff)[5]=BIGEND(0); //interval timer
          ep1_send(tx_buff,24);
        } else if(cdb[2]==0x3f) { //page 3F: return all subpages (none?)
          printf("sense3F ");
          tx_buff[0]=3;
          ep1_send(tx_buff,4);
        } else {
          printf("sense(%x,%x,%x)",cdb[1],cdb[2],cdb[3]);
          //return ILLEGAL REQUEST (see mode sense, SPC-2)
        }
        break;
      case 0x03:      //REQUEST SENSE (no error reporting for now)
        printf("reqSense ");
        if(cdb[1]&1) {  //descriptor format
          tx_buff[0]=0x72;
        } else { //fixed format
          tx_buff[0]=0x70;
        }
        ep1_send(tx_buff,8);
        break;
      default:      //unsupported (send null packet & fail)
        CSW.status=1;
        ep1_send((u8*)1,0);
        printf("err(x%x,%i) ", cdb[0], cbw->length);
        break;

    }
  } else {
    printf("?1");
    //stall
  }
  rx_idx=0;
}
```

(wacom.c)

```
// PEN (Personal Electronic Notebook)
// CS4710 Fall 2006
// Neal Tew
///////////////////////////////////////

// Wacom digitizer input over gumstix HW_UART port

#include <stdio.h>
#include "pen.h"

//digitizer commands
#define WACOM_START '1'
#define WACOM_STOP '0'
#define WACOM_QUERY '*'    //returns max resolution

#define PACKETLENGTH 9
static u8 packet[PACKETLENGTH]; //incoming packet buffer
static int pkt_idx;

#define PQSIZE 128     //queue holds about 1 second of pen data
struct {
  int x,y,pressure,flags;
} pen_queue[PQSIZE];  //raw values from the digitizer (not translated to screen coords yet)
int pen_head,pen_tail;

//-----------------------------------------------------
// higher level pen routines for checking software buttons, etc

buttonstruct *buttonlist=0;

// set the active buttonlist (pass in NULL for no buttons)
void set_buttons(buttonstruct *p) {
  buttonlist=p;
  if(!p)
    return;
  p++; //skip drawing surface
  do {
    p->x1=p->x0-1+*(int*)(p->up_bmp);
    p->y1=p->y0-1+*(int*)(p->up_bmp+4);
    draw_bmp(p->x0, p->y0, p->up_bmp);
    p++;
  } while(p->up_bmp);
}

//check where pen is at.  return -1 for nothing, 0=drawing, >0 = button
int checkbutton(int x, int y) {
  int i=0;
  buttonstruct *bl=buttonlist;

  if(!bl)
    return -1;
  do {
    if(x>=bl->x0 && x<=bl->x1 && y>=bl->y0 && y<=bl->y1)
      return i;
    i++;
    bl++;
  } while(bl->up_bmp);
  return -1;
}

//fill a pen_event struct
//return false if nothing interesting happened
enum {IDLE, BUTTONDOWN, DRAWING, NOTHING};
int get_pen_event(pen_event *p) {
  buttonstruct *bs;
  static int state=IDLE;
  static peninfo oldpen;
  peninfo pen;
  int x,y;
  int btn;
  int retval=0;

  if(!wacom_read(&pen))
    return 0;

  x=pen.x;
  y=pen.y;
  pen.btn=btn=checkbutton(x,y);
  retval=0;
  switch(state) {
    case IDLE:
      if( (pen.flags&PENDOWN) && !(oldpen.flags&PENDOWN) ) {
        p->erase=(pen.flags&ERASER);
        if(!btn) {
          state=DRAWING;
        } else if(btn>0) {
          draw_bmp(buttonlist[btn].x0, buttonlist[btn].y0, buttonlist[btn].down_bmp);
          p->buttonID=btn;
          state=BUTTONDOWN;
        } else {
          state=NOTHING;
        }
      }
      break;
    case BUTTONDOWN:
      bs=&buttonlist[p->buttonID];
      if(oldpen.btn!=btn) {  //button changed
        draw_bmp(bs->x0, bs->y0, btn==p->buttonID ? bs->down_bmp : bs->up_bmp);
      }
      if(!(pen.flags&PENDOWN)) { //button released
        draw_bmp(bs->x0, bs->y0, bs->up_bmp);
        state=IDLE;
        if(btn==p->buttonID) { //pen is still inside original button
          p->event=BUTTON_EVENT;
          retval=1;
        }
      }
      break;
    case DRAWING:
      if(!(pen.flags&PENDOWN)) { //pen's up. stop drawing
        state=IDLE;
      } else if(btn==0 && oldpen.btn==0) {  //pen is on a drawing surface
        p->event=DRAW_EVENT;
        p->x0=oldpen.x;
        p->y0=oldpen.y;
        p->x1=x;
        p->y1=y;
        retval=1;
      }
      break;
    case NOTHING: //pen touched in an undefined area, just wait for it to go up
      if(!(pen.flags&PENDOWN)) {
        state=IDLE;
        p->event=NO_EVENT;
        retval=1;
      }
      break;
  }
  oldpen=pen;
  return retval;
}
//------------------------------------------------------

//fill struct with next value from queue
//returns false if queue is empty
int wacom_read(peninfo *p) {
  int x,y;
  int cpumode;
  if(pen_head==pen_tail)
    return 0;

  cpumode=IRQS_OFF(); //critical section, don't let interrupts mess up the queue

  //p->pressure=pen_queue[pen_tail].pressure;
  p->flags=pen_queue[pen_tail].flags;
  p->raw_x=x=pen_queue[pen_tail].x;
  p->raw_y=y=pen_queue[pen_tail].y;

  if(!settings.calibrated) {
    x=x*X_RES/PEN_X_MAX;
    y=y*Y_RES/PEN_Y_MAX;
  } else {
    x=settings.px0+(x-settings.dx0)*(settings.px1-settings.px0)/(settings.dx1-settings.dx0);
    y=settings.py0+(y-settings.dy0)*(settings.py1-settings.py0)/(settings.dy1-settings.dy0);
  }

  //range check..
  if(x<0) x=0;
  else if(x>X_MAX) x=X_MAX;
  if(y<0) y=0;
  else if(y>Y_MAX) y=Y_MAX;

  p->x=x;
  p->y=y;

  pen_tail=(pen_tail+1)&(PQSIZE-1); //pop off the queue
  IRQS_SET(cpumode);
  return 1;
}

//full wacom packet was received, decode it and push it in the queue
void parse_packet() {
  u32 x,y;
  if ((packet[0]&0x60)==IN_RANGE) {
    x=((packet[6] & 0x60) >> 5) | (packet[2] << 2) | (packet[1] << 9);
    y=((packet[6] & 0x18) >> 3) | (packet[4] << 2) | (packet[3] << 9);

    //translate landscape to portrait coords
    pen_queue[pen_head].x=(PEN_X_MAX-y);
    pen_queue[pen_head].y=x;

    pen_queue[pen_head].pressure = ((packet[6] & 0x01) << 7) | (packet[5] & 0x7F);
    pen_queue[pen_head].flags=packet[0] & (IN_RANGE | ERASER | PENDOWN | BUTTON);
  } else {
    pen_queue[pen_head].x=0;
    pen_queue[pen_head].y=0;
    pen_queue[pen_head].pressure=0;
    pen_queue[pen_head].flags=0;
  }

  pen_head=(pen_head+1)&(PQSIZE-1);
  if(pen_head==pen_tail)     //queue is full? oldest value gets dropped
    pen_tail=(pen_tail+1)&(PQSIZE-1);
  setsig(SIG_PEN);
}

//called by main interrupt handler
void wacom_irq_handler() {
  int incoming;
  int iir=HWIIR&0x0f;

  if(iir==4 || iir==12) {  //Rx interrupt
    do {
      incoming=HWRBR;
      if((!pkt_idx)^(incoming>>7)) {  //bad byte? reset the queue
        pkt_idx=0;
      } else {      //good byte? add to queue
        packet[pkt_idx]=incoming;
        pkt_idx++;
        if(pkt_idx>=PACKETLENGTH) {
          parse_packet();
          pkt_idx=0;
        }
      }
    } while(HWLSR & LSR_DR); //loop until Rx FIFO is empty
  }
}


//send byte to digitizer.  don't bother with interrupts, since we rarely use this
void wacom_putc(int c) {
  while(!(HWLSR&LSR_TDRQ));
  HWTHR=c;
}


//clear buffers
void wacom_flush() {
  int cpumode;
  cpumode=IRQS_OFF(); //critical section, don't let interrupts mess up the queue

  pen_head=pen_tail=0;

  pkt_idx=0;    //reset packet queue
  HWFCR=FCR_TRFIFOE | FCR_RESETTF | FCR_RESETRF | FCR_ITL_16;  //clear UART fifos

  IRQS_SET(cpumode);
}


//setup HWUART for Wacom digitizer reading
void wacom_init() {
  pxa_gpio_mode(GPIO48_HWTXD_MD); //grab HWUART GPIO pins
  pxa_gpio_mode(GPIO49_HWRXD_MD);
  pxa_gpio_mode(GPIO16_PWM0 | GPIO_OUT | GPIO_DFLT_LOW);  //use PWM0 for digitizer reset pin

  CKEN|=CKEN_HWUART;  //UART clock enable
  HWIER=0;   //HWUART off
  HWMCR=MCR_OUT2;   //no loopback, enable UART interrupts
  HWLCR=LCR_DLAB;   //get access to divisor
  HWABR=0;   //no autobaud
  HWDLL=DLL_BAUD(19200); //set baud rate (lo)
  HWDLH=0;   //set baud late (hi)
  HWLCR=LCR_WLS(3); //no parity, 8 data bits, 1 stop bit
  HWFCR=FCR_TRFIFOE | FCR_RESETTF | FCR_RESETRF | FCR_ITL_16;  //enable FIFO
  HWIER=IER_UUE | IER_RTOIE | IER_RAVIE; //HWUART on, RX interrupts enabled
  wacom_flush();    //reset buffers
  irq_enable(IRQ_HWUART);
  pxa_gpio_mode(GPIO16_PWM0 | GPIO_OUT | GPIO_DFLT_HIGH); //release digitizer reset signal
  msleep(100);      //give digi some time to wake up
  wacom_putc(WACOM_START); //put wacom in send mode
}

void wacom_calibrate() {
  u8 *old;
  int i, bmp_offset;
  peninfo p;

  settings.calibrated=0;

  old=frame;
  DMAcopy(framebuffer, background_buff, FRAMESIZE);
  lcd_setframebuffer(framebuffer);

  #define X0 200
  #define Y0 200
  #define W 8

  bmp_offset=*((u32*)calib_x)/2;
  draw_bmp(X0-bmp_offset, Y0-bmp_offset, calib_x);

  statustext("Touch the X...\0");

  while(1) {
    wait(SIG_PEN);
    i=wacom_read(&p);
    if(i && (p.flags&PENDOWN))
      break;
  }

  settings.px0=X0;
  settings.py0=Y0;
  settings.dx0=p.raw_x;
  settings.dy0=p.raw_y;

  while(1) {
    wait(SIG_PEN);
    i=wacom_read(&p);
    if(i && !(p.flags&PENDOWN))
      break;
  }

////////////////////////

  rect(X0-50,Y0-50,X0+50,Y0+50,WHITE);

  #define X1 (X_RES-200)
  #define Y1 (Y_RES-200)
  draw_bmp(X1-bmp_offset, Y1-bmp_offset, calib_x);

  while(1) {
    wait(SIG_PEN);
    i=wacom_read(&p);
    if(i && (p.flags&PENDOWN))
      break;
  }

  settings.px1=X1;
  settings.py1=Y1;
  settings.dx1=p.raw_x;
  settings.dy1=p.raw_y;

  while(1) {
    wait(SIG_PEN);
    i=wacom_read(&p);
    if(i && !(p.flags&PENDOWN))
      break;
  }
```

```
   lcd_setframebuffer(old);

   settings.calibrated=1;
   save_settings();
}
```