Building a linux module

Interfacing a character LCD display

by Carlos Becker – http://carlosbecker.com.ar

REVISION A - 24th November, 2007

Introduction

The idea behind this text is to present the basic guidelines needed to interface a character LCD display from an embedded linux. In this particular example the target architecture is avr32, running Linux Kernel 2.6.x and uClibc.

The approach chosen is to build a kernel module and implement a char device on `/dev/lcddev`, in order to be able to do something like "`echo 'hello there' > /dev/lcdddev`" to display text on the LCD module.

It is true that a kernel module might not be the best solution to this particular case, since a non-preemptive kernel could lead to our kernel driver doing some blocking, avoiding some other important kernel tasks or other I/O modules from performing their functions. However, this example is a good start point to demonstrate how easy and straight forward it is to develop a kernel module for I/O.

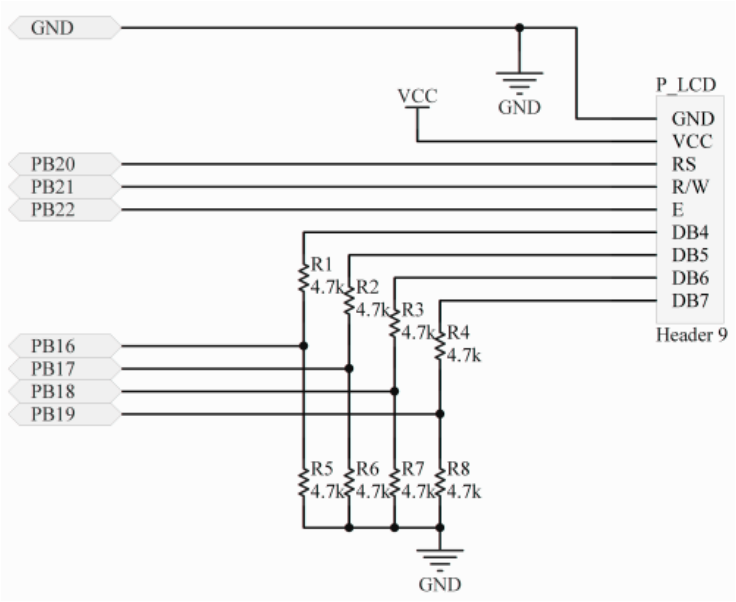
Interface compatibility

Most character LCDs get powered from 5V. The avr32 I/O pads work at 3.3V, so some kind of conversion is expected.

If we were to just send data to the LCD and expect nothing back from it, everything would be fine, since the LCD would recognize 3.3V as High and 0V as Low on its inputs. However, we need to acquire data regarding busy states.

Working at not-so-high switching frequencies lets us use a simple R-R divider which allows bidirectional I/O. The problem with higher frequencies is that the RC circuit composed by the R-R divider and the pin's stray capacitance can distort digital signals importantly.

Here it is the schematic used for this project:



The resistors have to be large enough to avoid drawing too much current from the LCD's output, but not so high in order to avoid the RC equivalent circuit from distorting the digital signals significantly. 4.7k seems to work alright with my 08x02 character LCD.

Note that the other lines (E, R/W and RS) do not need the R-R networks, since they always act as inputs from the LCD side.

Back to the AVR32 side, I chose PB19 to PB22 as I/O pins just for convenience, but any other free pins could be used (this was tested on a NGW100 board, I don't know about free pins on the STK1000).

Don't forget to wire the GND line!

Module squeleton

Before writing any code, it is important to divide the module code in a few files, according to functionality. This allows better understanding and easier enhancements in the future.

The files will be:

- module.c — *basic module code, initialization and termination*
- chardev.c, chardev.h — *character device implementation*
- lcd.c, lcd.h — *low level lcd routines, called from the other two files*
- portdefs.h — *port address definition*

To ease the compiling tasks requiered to build the module, we use the "framework" provided by the kernel developers. Next are the files to be created:

Makefile

```
#linux source path, avr32 --- REPLACE TO FIT YOUR BUILD
LINUX_SOURCE = /root/compile/avr32/linux-2.6.22.atmel.2


default:
    make ARCH=avr32 CROSS_COMPILE=avr32-linux- -C $(LINUX_SOURCE) M=`pwd`
                    modules


clean:
     make -C $(LINUX_SOURCE) M=`pwd` clean
```

Kbuild

```
obj-m := lcdchar.o
lcdchar-y = module.o chardev.o lcd.o
```

Module basic functions

Now we are ready to implement the basic module functions, namely `init_module()` and `cleanup_module()`.

module.c

```
#include <linux/module.h>

#include <asm/io.h>
#include <linux/timer.h>
#include <linux/jiffies.h>
#include <linux/clk.h>
#include <linux/interrupt.h>
#include <linux/irq.h>
#include <linux/workqueue.h>
#include <linux/fs.h>

#include "lcd.h"    //lcd low level routines


int init_module(void)
{
    int ret_val;

        printk("Starting up\n");


    if ( !LCD_init() )      //initialize LCD
    {
        printk( KERN_ALERT "Couldn't init LCD\n" );
        printk( KERN_ALERT "Module not initialized!\n");

        return -5;
    }


    if ( ! LCD_clear() )        //clear contents
    {
        printk( KERN_ALERT "Coudln't clear the LCD\n");
    }

    if ( !LCD_printstr("Init OK") )
    {
        printk( KERN_ALERT "Couldn't write to LCD\n");
    }


    //register char device
    ret_val = register_lcd_chrdev();

    if ( ret_val < 0 )
    {
        printk( KERN_ALERT "Couldn't register LCD device\n" );
        return ret_val;
    }

    return 0;   //ok
}

void cleanup_module(void)
{
        printk("Closing LCD device\n");

    //unregister character device from kernel
    unregister_lcd_chrdev();
}


MODULE_LICENSE("GPL");
```

The code is practically self explaining. If everything goes alright, "Init OK" should appear at the LCD screen.

The LCD routines return zero when there was an error when waiting the BUSY flag from the module to get cleared. There is a timeout defined in lcd.c.

Character device callbacks and registration

In order to provide a character device interface to the kernel it must be registered as such first and report information about itself. Here is the source code:

chardev.h

```
#ifndef _LCDCHAR_H_
#define _LCDCHAR_H_

#include <linux/ioctl.h>

//MAJOR number, static here, used with mknod in user space
#define MAJOR_NUM   101

//Device file name, to register in /dev/lcddev
// (still needs mknod)
#define DEVICE_FILE_NAME    "lcddev"

//Prototypes for (un)register functions
int  register_lcd_chrdev( void );
int  unregister_lcd_chrdev( void );


#endif
```

MAJOR_NUM is the major number to be used by the module, so that mknod can be called later by using that same number. On the other side, "lcddev" is the device name to be passed to mknod too.

chardev.c

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <asm/uaccess.h>    //para get_user y put_user

#include "chardev.h"    //some useful definitions

#include "lcd.h"    //low level functions

#define SUCCESS 0

//holds whether the device was already opened by someone
//and wasn't released
static int  Device_Open = 0;

//called when user calls open()
static int  device_open( struct inode *inode, struct file *file )
{
    if ( Device_Open )
        return  -EBUSY; //solamente lo puede usar 1 al mismo tiempo

    Device_Open++;  //flag it!

    //incr usage count
    try_module_get( THIS_MODULE );

    return SUCCESS; //ok!
}
```

```
//called when user calls close()
static int  device_release( struct inode *inode, struct file *file )
{
    Device_Open--;  //flag that

    //decr usage count
    module_put( THIS_MODULE );
    return SUCCESS;
}

//called when user calls read() -- NOT IMPLEMENTED
/** static ssize_t  device_read( struct file *file, \
                char __user *buffer, \
                size_t length, \
                loff_t  *offset )
{
    return 0;
} **/

//called when user calls write() -- just writes data to display
static ssize_t  device_write( struct file *file,
                const char __user *buffer,
                size_t  length,
                loff_t *offset )
{
    //neccesary chars + some other just in case
    char k_buf[ LCD_CHARPERLINE*LCD_LINES + 4 ];

    //truncate if neccesary
    if ( length > sizeof( k_buf ) - 1 )
      length = sizeof( k_buf ) - 1;

    //copy from user space to k_buf
    copy_from_user( k_buf, buffer, length );

    //end string, just in case..
    kbuf[ length ] = '\0';

    //write data to LCD...
    if ( ! LCD_printstr( k_buf ) )
        return 0;

    return length;  //return length
}

//called when user calls ioctl() -- NOT IMPLEMENTED
/** int     device_ioctl( struct inode *inode,
                struct file *file,
                unsigned int ioctl_num,
                unsigned long ioctl_param )
{
    return SUCCESS; //ehem..just say OK?
}**/

//neccesary to register as character device
struct file_operations  Fops = {
    .read   = NULL,
    .write  = device_write,
    .ioctl  = NULL,
    .open   = device_open,
    .release= device_release,
};

//exported function, used by module.c
int register_lcd_chrdev(void)
{
    int ret;

    ret = register_chrdev( MAJOR_NUM, DEVICE_FILE_NAME , &Fops );

    if ( ret < 0 )
        return ret;

    return ret;
}

//exported function, used by module.c
```

```
int unregister_lcd_chrdev(void)
{
    unregister_chrdev( MAJOR_NUM, DEVICE_FILE_NAME );

    return 0;
}

MODULE_LICENSE("GPL");
```

The basic char dev structure can be found at http://www.faqs.org/docs/kernel/x571.html along with some explanations.

An important function is `register_chrdev()` which tells the kernel that we want to implement a character device in our module. Fops is a structure which contains function pointers to our callbacks, so when an `open()`, `read()`, `write()`, `close()` or `ioctl()` call is made over our device, we will get noticed.

Since we want to be able to do something like " `echo 'hello' > /dev/lcddev` ", we must implement `open()` and `write()`. The `read()` callback is left empty, since we don't have any information to send to the linux user. Finally, the `release()` callback is processed when someone calls `close()` on the device.

The `Device_Open` variable is to avoid several users from opening the device at the same time. It is controlled by `device_open()` and `device_release()`.

We also help the kernel keep track of the module's usage count. This is done by `try_module_get` and `module_put` in `device_open()` and `device_release()`. This way, the kernel won't let the user do a `rmmod` on this module when it's being used and hasn't been closed yet.

LCD low level routines

First we create an include file with port address definitions:

port_defs.h

```
#ifndef _portdefs_
#define _portedfs_

#define PORTB   (void*)0xffe02C00

#define PIO_PER 0x0000
#define PIO_PDR 0x0004
#define PIO_PSR 0x0008
#define PIO_OER 0x0010
#define PIO_ODR 0x0014
#define PIO_OSR 0x0018

#define PIO_SODR 0x0030
#define PIO_CODR 0x0034

#define PIO_PDSR 0x003C

#define PIO_PUDR 0x0060
#define PIO_PUER 0x0064

#define PIO_ASR 0x0070

#define PIO_OWER 0x00A0

#define PIO_ODSR 0x0038

#endif
```

In order to make things show up on the display, here are the neccesary routines

lcd.h

```
#ifndef __lcd_h
#define __lcd_h

// Port used both as DATA and CONTROL
#define LCD_PORT    PORTB
      // PB16:19 -> BITS4:7 LCD bits

// bits where these signals are mapped
#define LCD_RS 20
#define LCD_RW 21
#define LCD_E 22

//LCD timeout in milliseconds when waiting for busy information
#define LCD_TIMEOUT_MS  2


//clear LCD, just a macro
#define LCD_clear() LCD_writeCMD(LCD_CMD_CLEAR)

//define some basic constants
#define LCD_LINES   2
#define LCD_CHARPERLINE 8   //info we need
#define LCD_LINEOFFSET  0x40    // DDR address, 1st line


// LCD command list
#define LCD_CMD_INIT4 (1<<1)

#define LCD_CMD_CLEAR 0x01
#define LCD_CMD_HOME  0x02

#define LCD_CMD_ENTRYMODE 0x06  // increase cursor, do notshift display

// on, cursor, blinking = 0 or 1
#define LCD_CMD_DISPLAY(on,cursor,blinking) ( (1<<3) | (on<<2) | (cursor<<1) |
blinking<<0 )

#define LCD_on(aa)  LCD_writeCMD(LCD_CMD_DISPLAY(aa,0,0))

// display=~cursor
// right= ~left
#define LCD_CMD_SHIFT(display,right)  ((1<<4) | (display<<3) | (right<<2) )

//bits8 = ~bits4
//line2=~line1
//size5x10=~size5x7
#define LCD_CMD_FUNC(bits8,line2,size5x10) ( (1<<5) | (bits8<<4) | (line2<<3) |
(size5x10<<2) )

//addr is 6 bits, bit 6 and 7 = 0!
#define LCD_CMD_SET_CGRAM_ADDR(addr) ( (1<<6) | addr )   // address is 6 bits long

//addr is 7 bits, bit7=0!
#define LCD_CMD_SET_DDRAM_ADDR(addr)  ( (1<<7) | addr)


/** Starts/Stop cursor blinking **/
unsigned char    LCD_blinkCursor(unsigned char on);

/** Writes command to LCD, or DATA **/
unsigned char  LCD_writeCMD(unsigned char data);
unsigned char  LCD_writeData(unsigned char data);

/** Init LCD **/
unsigned char  LCD_init(void);

/** Wait for LCD not busy
 * returns !=0 if alright, 0 if timeout */
unsigned char  LCD_waitBusy(void);

/** Print ascii string on LCD **/
```

```
unsigned char  LCD_printstr(char *str);

/** Print ascii string to a given LCD position **/
unsigned char  LCD_printstr_at(char *str, unsigned char line, unsigned char
pos);

/** Deletes just one LCD line **/
unsigned char   LCD_clearLine(unsigned char line);


#endif
```

lcd.h contains bit and port definitions used by the LCD module. By changing them, the kernel module can be adapted to different wirings.

It's imperative to call `LCD_init()` before doing any operation on the display to make sure that the LCD is present and ready to accept commands.

Function implementation follows here:

lcd.c

```
#include "lcd.h"

#include <asm/io.h>
#include "port_defs.h"
#include <linux/delay.h>

/** Time counting functions, for waitBusy() timeout */
#include <linux/jiffies.h>


/*** FUNCTIONS TO SET/CLR PORT BITS ***/
#define clrbit(x)   __raw_writel( (1<<(x))  ,LCD_PORT + PIO_CODR )
#define setbit(x)   __raw_writel( (1<<(x))  ,LCD_PORT + PIO_SODR )

/** Reads outputted data **/
#define LCD_DATA_READ_OUTPUT()  __raw_readl( LCD_PORT + PIO_ODSR )

/** Sets DATA bits as input, the others remain intact */
#define LCD_DATA_SET_AS_INPUT() __raw_writel( (1L<<16) | (1L<<17) | (1L<<18) |
(1L<<19) , LCD_PORT + PIO_ODR )

/** Reads Input from DATA bits **/
#define LCD_DATA_READ_INPUT()   ( ( __raw_readl( LCD_PORT + PIO_PDSR ) >> 16 )
& 0x0F )

/** Sets DATA bits as output, the others remain intact ***/
#define LCD_DATA_SET_AS_OUTPUT()    \
    __raw_writel( (1L<<16) | (1L<<17) | (1L<<18) | (1L<<19) \
            , LCD_PORT + PIO_OER );

/** Clears DATA bits when configured as outputs **/
#define CLEAR_OUTPUT()  __raw_writel( ~( (~LCD_DATA_READ_OUTPUT()) | (0x000FL
<< 16 ) ), LCD_PORT + PIO_ODSR )

/** Sets data bits to a given nibble **/
#define SETDATA(c)  CLEAR_OUTPUT(); __raw_writel( LCD_DATA_READ_OUTPUT() |
( ((c)&0x0FL)<<16) , LCD_PORT + PIO_ODSR )

//millisecond delays
#define _delay_ms(x)    msleep(x)


/* This is a inter-message delay, neccesary to let the LCD read the data **/
#define WAIT  ndelay(100)
```

```
unsigned char LCD_clearLine(unsigned char line)
{
    return LCD_printstr_at( "        ",line,0);
}

void  LCD_AwriteCMD(unsigned char data)

{
    clrbit(LCD_RS);  //command!
    clrbit(LCD_RW);  //write!

    SETDATA( (data>>4) & 0xF); // mask, MSB nibble sent
    setbit( LCD_E );


    WAIT;


    clrbit(LCD_E);


    SETDATA(data & 0xF); // mask, LSB nibble sent
    setbit(LCD_E);

    WAIT;

    clrbit(LCD_E);

}



unsigned char    LCD_blinkCursor(unsigned char on)
{
    if (on)
        return LCD_writeCMD( LCD_CMD_DISPLAY(1,1,1) ); // on, cursor, blinking!
    else
        return LCD_writeCMD( LCD_CMD_DISPLAY(1,0,0) ); // on, nocursor, no blinking!
}




unsigned char  LCD_init()
{

    unsigned char data;

    // Configure so writing on ODSR will modify IO values
    __raw_writel( (1L<<16) | (1L<<17) | (1L<<18) | (1L<<19) | \
            (1L<<LCD_RS) | (1L<<LCD_RW) | (1L<<LCD_E), LCD_PORT + PIO_OWER );

    //Activate PIO control for this pads
    __raw_writel( (1L<<16) | (1L<<17) | (1L<<18) | (1L<<19) | \
            (1L<<LCD_RS) | (1L<<LCD_RW) | (1L<<LCD_E), LCD_PORT + PIO_PER );
    //Everything as output
    __raw_writel( (1L<<16) | (1L<<17) | (1L<<18) | (1L<<19) | \
            (1L<<LCD_RS) | (1L<<LCD_RW) | (1L<<LCD_E), LCD_PORT + PIO_OER );

    //Initially zero
    clrbit( LCD_RS ); clrbit( LCD_E ); clrbit( LCD_RW );




    ////////////////////////////////


    // delay for 45 ms.. not neccesary if LCD was already powered up before
    _delay_ms(10);_delay_ms(10);_delay_ms(10);_delay_ms(10);
    _delay_ms(5);
```

```
    ////////////////RESETTT!!!

    for (data = 0; data < 3; data++ )
    {
        clrbit(  LCD_RS);
        clrbit(  LCD_RW);

        SETDATA( 3 );

        setbit( LCD_E );
        WAIT;
        clrbit(LCD_E); //end

        //30ms
        _delay_ms(10);_delay_ms(10);_delay_ms(10);_delay_ms(10);
    }

    ///////////END RESET


    data = 0x22;
    clrbit(LCD_RS);  //command!
    clrbit(LCD_RW);  //write!

    SETDATA( (data>>4) & 0xF); // mask, MSB nibble sent
    setbit(LCD_E); //written
    WAIT;
    WAIT;
    clrbit(LCD_E); //end

    SETDATA(data & 0xF); // mask, MSB nibble sent
    setbit(LCD_E); //written
    WAIT;
    WAIT;
    clrbit(LCD_E); //end



    data=0xC;
    WAIT;
    WAIT;
    SETDATA(data & 0xF); // mask, MSB nibble sent
    setbit(LCD_E); //written

    WAIT;
    WAIT;
    clrbit(LCD_E); //end

    WAIT;
    WAIT;


    _delay_ms(5);



    LCD_AwriteCMD( LCD_CMD_DISPLAY(1,0,0) ); // on, nocursor, no blinking!

    //DELAY!
    _delay_ms(5);



    LCD_AwriteCMD( LCD_CMD_CLEAR );  //clear!

    _delay_ms(10);

    return LCD_writeCMD( LCD_CMD_ENTRYMODE );
}
```

```
unsigned char  LCD_writeCMD(unsigned char data)
{
    clrbit(LCD_RS);  //command!
    clrbit(LCD_RW);  //write!

    SETDATA( (data>>4) & 0xF); // mask, MSB nibble sent
    setbit(LCD_E); //written
    WAIT;
    clrbit(LCD_E); //end

    WAIT;

    SETDATA( data & 0xF); // mask, LSB nibble sent
    setbit(LCD_E); //written
    WAIT;
    clrbit(LCD_E); //end

    WAIT;

    return LCD_waitBusy();
}

unsigned char  LCD_writeData(unsigned char data)
{

    SETDATA( (data>>4) & 0xF); // mask, MSB nibble sent
    setbit(LCD_RS);  //data!
    clrbit(LCD_RW);  //write!
    setbit(LCD_E); //written
    WAIT;
    clrbit(LCD_E); //end

    WAIT;

    SETDATA( data & 0xF); // mask, LSB nibble sent
    setbit(LCD_RS);  //data!
    setbit(LCD_E); //written
    WAIT;
    clrbit(LCD_E); //end

    WAIT;
    return LCD_waitBusy();
}
unsigned char LCD_printstr(char* str)
{
    return LCD_printstr_at(str,0,0);//ok!
}
```

```
//prints a str. if no '\n' present then it will automatically jump
// function will print the max characters it can
unsigned char  LCD_printstr_at(char* str, unsigned char line, unsigned char
pos)
{
    unsigned char i,j,k;
    unsigned char curaddr;

    if (line >= LCD_LINES)
        return 0;
    if (pos >= LCD_CHARPERLINE)
        return 0;

    curaddr = LCD_CMD_SET_DDRAM_ADDR(0) + line*LCD_LINEOFFSET + pos;
    i=0;
    for (j=0; j<LCD_LINES; j++)
    {
        k=0;
        if ( ! LCD_writeCMD( curaddr ) )
            return 0;

        while( (k<LCD_CHARPERLINE) && (str[i] != '\n') && (str[i] != '\0') )
        {
            if  ( ! LCD_writeData(str[i]) )
                return 0;
            i++;
            k++;
        }
        if (str[i] == '\0')
            break;
        //i++;
        curaddr+= LCD_LINEOFFSET;   //add it!
    }

    return 1;
}

unsigned char LCD_waitBusy()
{
    unsigned char lnibble, hnibble;
    unsigned long timeout = jiffies + msecs_to_jiffies( LCD_TIMEOUT_MS );

    do
    {
        //timeout?
        if ( jiffies > timeout )
        {
            LCD_DATA_SET_AS_OUTPUT();   //prepare to exit
            return 0;   //bad :(
        }

        //Turn to inputs first
        LCD_DATA_SET_AS_INPUT();

        setbit( LCD_RW);
        clrbit(LCD_RS);  //read status

        setbit(LCD_E);
        WAIT;
        hnibble = LCD_DATA_READ_INPUT() & 0x0F; // filter
        clrbit(LCD_E); //ready!


        setbit( LCD_RW);
        clrbit(LCD_RS);  //read status
        WAIT;
        setbit(LCD_E);
        WAIT;

        lnibble = LCD_DATA_READ_INPUT() & 0x0F; // filter

        clrbit(LCD_E); //ready!
```

```
        clrbit( LCD_RW);      //as output again
    } while ( ( (1<<3) & hnibble )  != 0);  //testbit 3, ready?

    //Back to outputs
    LCD_DATA_SET_AS_OUTPUT();

    return 1;   //ok
}
```

Compile and test the new device!

So that's it! We are done!

Now just invoke 'make' and wait until it compiles and links the module. You should have a 'lcddev.ko' file in the Makefile's directory. If something fails, please check that `LINUX_SOURCE` variable in `Kbuild` fits your system.

Now connect the LCD to the NGW100 and invoke '`insmod lcddev.ko`' to load the kernel module. The message "Init OK" should appear in the LCD module. It is useful to track kernel messages (from printk) to do debugging. If you can't see kernel messages do a '`dmesg|tail`' to see if everything went alright.

Before writing anything to the LCD we have to make an entry in /dev. Just type:

```
mknod /dev/lcddev c 101 0
```

The arguments tell mknod to make a character device with major number 101 and minor number 0. Type '`man mknod`' to see detailed information.

OK! Now it's the time..... "`echo 'hello!' > /dev/lcddev`"

If something goes wrong, you can look for debug information with `dmesg` and doing something like:

```
cat /proc/devices
```

`/proc/devices` contains a list of registered devices, where lcdddev should appear if initialization went ok.

Writing a C program to communicate with the LCD

Here is a simple program which basically does the same as " `echo 'hello' > /dev/lcddev` ".

test_me.c

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <string.h>

int main( int argc, char *argv[] )
{
    if ( argc != 2 )
    {
        printf("Run me as:\n\t./test_me <string>\n");
        return -1;
    }

    int fd = open("/dev/lcddev", O_RDWR );

    if ( fd < 0 )
    {
        printf("Couldn't open lcddev!!\n");
        return -1;
    }


    write( fd, argv[1], strlen(argv[1]) );

    return 1;
}
```

Just compile it with

```
avr32-linux-gcc   test_me.c -o test_me
```

and then run, from the NGW

```
./test_me 'hello!'
```

Further improvements

First and more important, this kernel module currently calls `ndelay()` and `msleep()`. The first of them just keep the execution path looping around for the amount of time specified, avoiding other kernel functions from using the CPU. On the other side, `msleep()` doesn't. The reason ndelay() is used throughout the program is because there is no such `nsleep()` or `usleep()` function provided by the kernel, so using `msleep()` implies a minimum sleep time of 1 millisecond, which is extremely large.

A quite useless delay loop is the 45msec one found in `LCD_init()` which I kept since it was in one of my AVR 8-bit project, but seems to be totally useless with AVR32 since the LCD module was (supposedly) powered up long time ago. Anyway, since `msleep()` is used there, it will cause no harm to other kernel modules or devices.

Back to CPU apropiation: even if the LCD display functions are not called quite frequently, the loop inside `LCD_waitBusy()` is an absurd CPU time consuming task.

The solutions to this matter was pointed to me by AVRFreaks forum users at avrfreaks.net, see http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&p=381268 . Making a kind of "user-space driver" looks promisingm althou it can be dangerous to what security implies.

Maybe the next version of this document includes a driver using mmap() and /dev/mem. I have already tried those and successfully "bypassed" the kernel to get to the PIO registers. See next section below!

Other minor improvements may include:

- use custom LCD characters
- read from LCD's RAM
- make IOCTL entry for `LCD_setCursorPos()`, `LCD_blinkCursor()`, etc
- ask linux for `MAJOR_NUM` rather than arbitrary selection
- Anything you want! Just remember to have fun :P

Using mmap() and /dev/mem to <bypass> the kernel

As said before, `mmap` can be used to to get directly to uP registers. Care must be taken, since touching what we should not can cause a system crash or a security hole.

As said in the avrfreaks.net post mentioned above, given that permissions are correctly set up on the NGW, only root can access `/dev/mem`, and that's essential for this method to success.

Here it is a simple but effective example

test_mmap.c

```
#include <stdio.h>
#include <sys/mman.h>
#include <malloc.h>
#include <unistd.h>
#include <fcntl.h>

/* Amount of memory we need to
 * access all PIO registers    */
#define IOM_SIZE    (0x00AC+4)
```

```c
/* Where that amount of memory starts
 * (PIOB chosen here )                  */
#define IOM_BASE    0xFFE02C00

/* Page size used by the kernel on the MMU
 * It is needed since mmap asks for a multiple
 * of PAGE_SIZE as the base address              */
#include <asm/page.h>   // <--- PAGE_SIZE defined there

int main()
{
    // /dev/mem file descriptor
    int mem_fd;

    // open /dev/mem
    mem_fd = open("/dev/mem", O_RDWR | O_SYNC );
    if ( mem_fd < 0 )
    {
        printf("Error opening /dev/mem\n");
        return -1;
    }

    unsigned long *io_mem = NULL;

    unsigned long int   iom_base = IOM_BASE;
    unsigned long int   iom_offset = 0;

    //fix to fit a page start address
    iom_base -= IOM_BASE % PAGE_SIZE;

    /* offset of IOM_BASE with respect to
     * iom_base                           */
    iom_offset = IOM_BASE % PAGE_SIZE;

    // print some debug info
    printf("Start address request: %lu\n", iom_base );
    printf("Offset %lu\n", iom_offset );
    printf("Page size is %d\n", PAGE_SIZE );

    // call mmap()
    io_mem = (unsigned long*) mmap( NULL,
                            IOM_SIZE + iom_offset ,
                            PROT_READ | PROT_WRITE,
                            MAP_SHARED,
                            mem_fd,
                            iom_base );


    // succeeded?
    if ( io_mem == MAP_FAILED )
    {
        printf("mmap couldn't success!\n");

        return -1;
    }

    // create a pointer to IOM_BASE (PIOB in this example)
    unsigned long *portb = (unsigned long*) ( ( (void*)io_mem ) + iom_offset  );

    printf("Setting PB16 as output\n");
    *portb = (1L<<16);       //enable for PIO control
    *(portb+4) = (1L<<16);  //set output

    // toggle on and off at 0.5Hz
    while(1)
    {
        *(portb+12) = (1L<<16); //high
        sleep(1);
        *(portb+13) = (1L<<16); //low
        sleep(1);
    }

    // will never reach here, but..
    printf("Goodbye\n");

    return 1;
}
```

The code explains itself through the comments. The most important piece is to call `mmap()` with a start address which is a multiple of `PAGE_SIZE`, since the MMU is involved there.

To understand the code faster, consider that `portb` is a *pointer to unsigned long*. That way, the following are true:

- `*portb` refers to PIO_PER
- `*(portb+4)` refers to PIO_OER (IOM_BASE + 4*0x04)
- `*(portb+12)` refers to PIO_SODR (IOM_BASE + 12*0x04)
- `*(portb+13)` refers to PIO_CODR (IOM_BASE + 13*0x04)

Of course a bunch of #defines wouldn't have hurt :P, but maybe using void pointers would be easier, since we could take some advantage by reusing the file `port_defs.h`. It's up to you.

To compile just write:

```
avr32-linux-gcc   test_mmap.c -o test_mmap
```

This code will never return to the command line by itself, but since it is a user-space program you can kill it with CONTROL-C.

Revisions and credits

- Revision A:
  - Thanks to *hardy* at avrfreaks.net for pointing out that there was a `msleep()` function.
  - *hardy* also proposed to remove the read() and ioctl() functions to avoid confussion, so they are now commented.
  - Added `mmap()`, `/dev/mem` example.

References

- The Linux Kernel Module Programming Guide
  http://www.tldp.org/LDP/lkmpg/2.6/html/index.html
- ParamodCE robot
  http://www.avr32linux.org/twiki/bin/view/Main/PramodeCE
- AVRFreaks forum
  http://avrfreaks.net