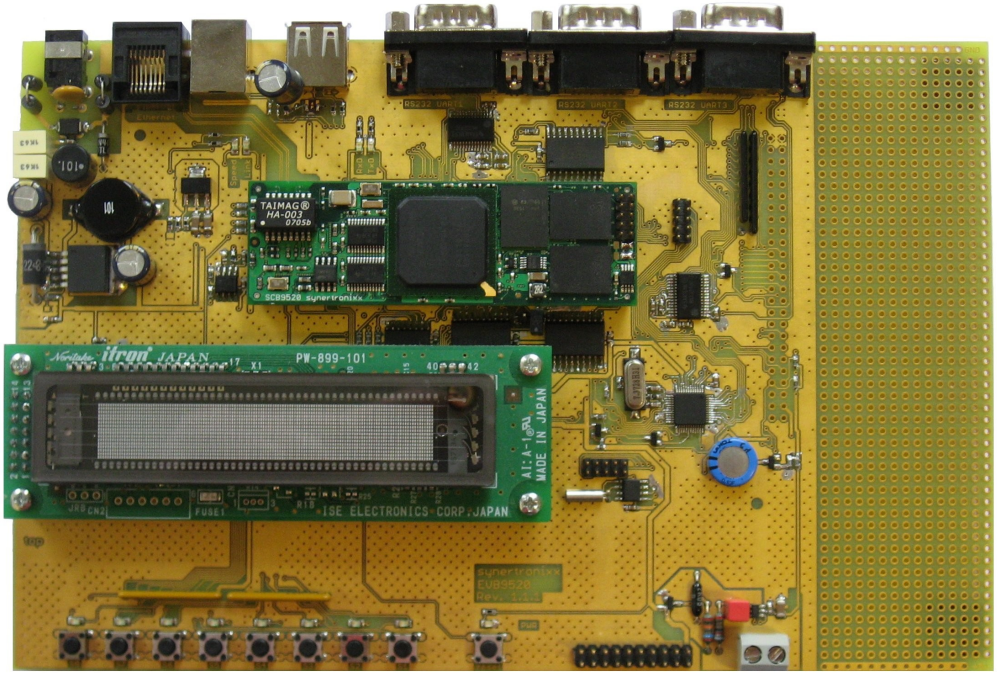
EVB9520
Getting Started



EVB9520 Getting Started

Table of Contents

1 Getting Started

1.1 Set up the Evaluation Board

To put the evaluation board in operation, you have to connect it to a power supply (DC 10V – 30V). To communicate and to control the board, you should connect it with a null modem cable to a PC. Controlling the board is also possible with a network connect EVB9520 on. For the serial connection, you use the COM port on your PC, the debug port on the EVB9520 and a terminal application like *tera term*, *hyperterminal* and on Linux you can use *kermit* or *minicom*.

If you want to control the board with a MS Windows PC, you can also use the application *DeviLANControl*.

The steps in detail:

1. Connect the board with a null modem cable with the PC.
2. Start a terminal application with the following settings: 115,2kBaud, 8 data bit, 1 stop bit and no parity.
3. Now you connect the EVB9520 (without the SCB9520) to the power supply. The PWR-Diode should light. The state of the 8 LEDs is not defined in this case, but in most cases they also light.
4. Now switch off the power supply.
5. If the steps before worked fine, you can go on with this and the next steps. Now mount the SCB9520 carefully on the DIL64 interface. Please note, that it is positioned in the right way. The transformer must be as near as possible to the RJ45 Ethernet interface.
6. Now switch on the power supply again. The LEDs RxD and TxD should flash and the 8 LEDs should light continuously.

The applications *udp_config* and *evb9520* start automatically in the background and now you can read the time on the display. Perhaps the time you see is wrong.

A detailed introduction for using the serial interface with Linux or MS Windows is given in the chapters below.

1.2 Using the serial interface (Linux)

To control the EVB9520 you can use a terminal program on a Linux PC. After starting the board you have to login with the below given username and password. In picture 8 you can see a screenshot.

```
username : root
password : pass
```

Please note, Linux is case sensitive and you must distinguish between upper case and lower case characters. There are several terminal application for Linux available, well

known are *minicom* and *kermit*.

Figure 1: Login with a terminal on a Linux PC.

In screenshot 2 you can see the output you get after a successful login. The commands *ls* (list segment), *pwd* (print working directory) and *cd* (change directory) can help you to have a look at the embedded Linux system and it's file system.

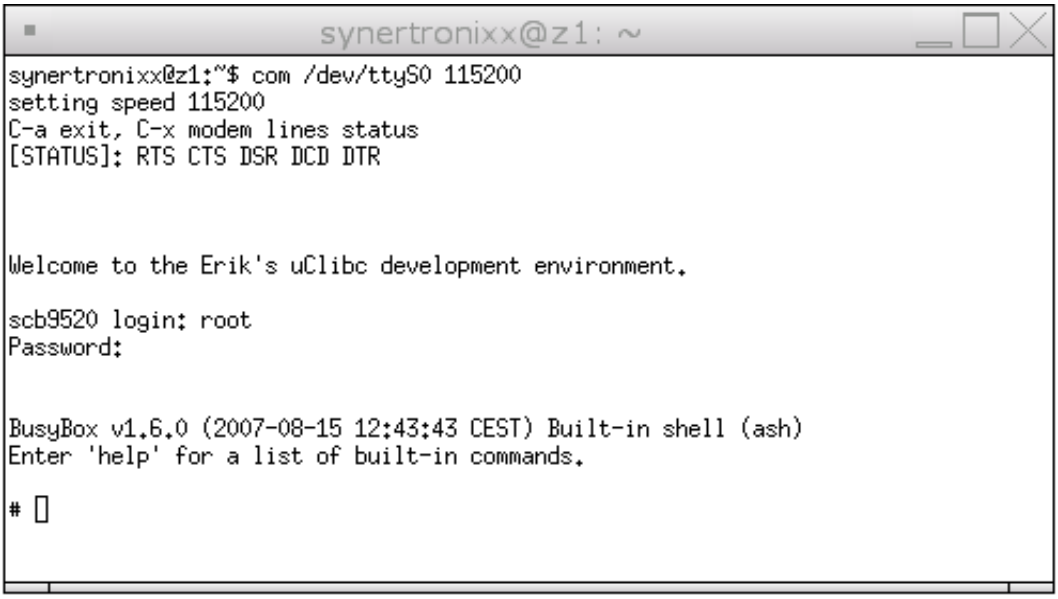


Figure 2: The login succeed.

It's time to configure the network interface. In figure 3 you see the file /etc/network/interfaces. It is opened with the editor *nano*, the editor *vi* also exists on the system.

The entries of this file are maybe not the same, especially “modulname””, “serial”, “tcpport” and “udpport” can vary or even doesn't exist. These entries are made with the application *udp_config*. These entries does not affect the use of the Linux system, they are only used by the application *evb9520*.

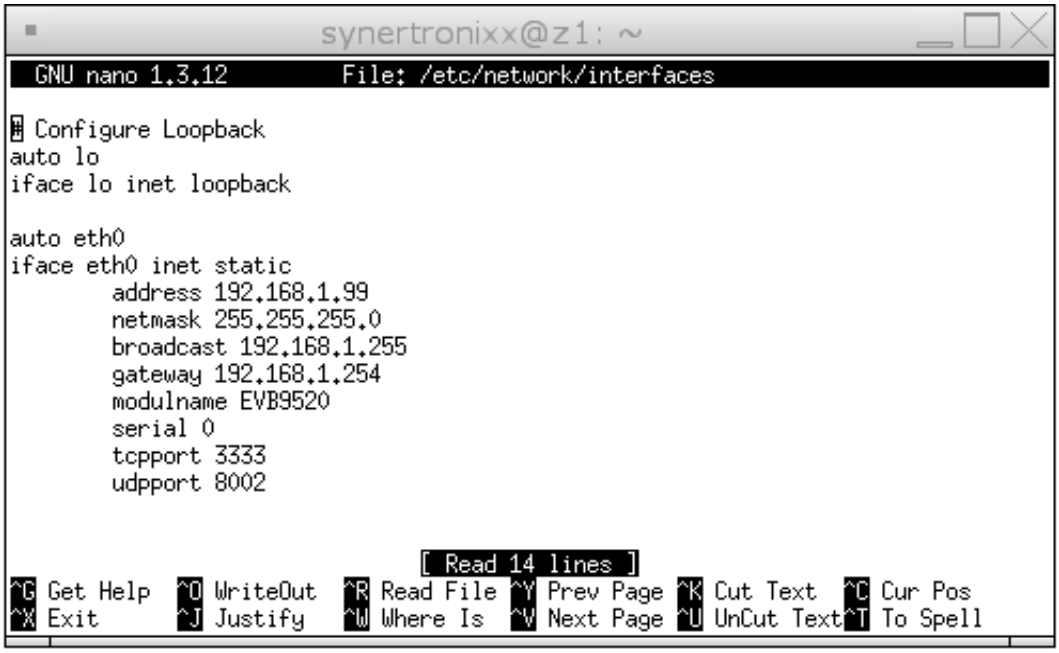


Figure 3: Changing the IP with nano.

By default, the IP address is set to 192.168.1.199. If you want to change it or something else, do it here.

To finish the configuration and use the new entries, use the following commands on the command line.

```
ifdown eth0
ifup eth0
```

The first command stops the network device and the second starts them again. The output of these commands are shown on the figure 9 below.

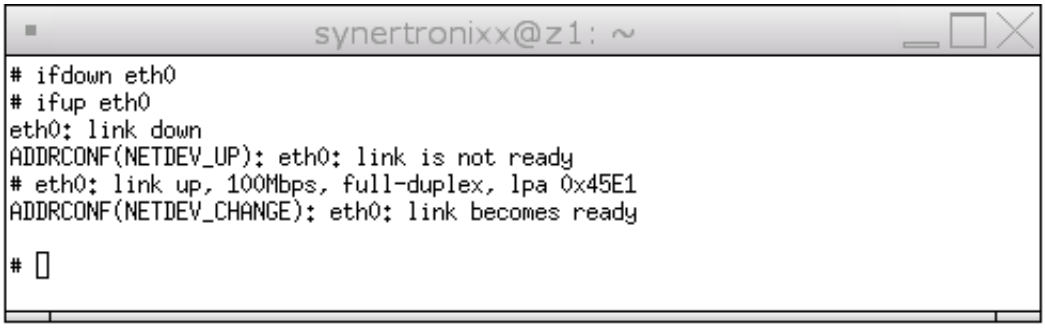


Figure 4: Restart the network.

To test the new configuration, you can use the command *ping*. A possible way to use *ping* is shown in figure 5.

A good strategy to test the network is, first ping your own IP and if that works, ping another member in your network. In the figure 5 you see some additional parameters, the first one is *c* and the second is *4*. With this parameters you only send 4 ping's instead of pinging continuously. The last parameter is the IP address you want to ping.

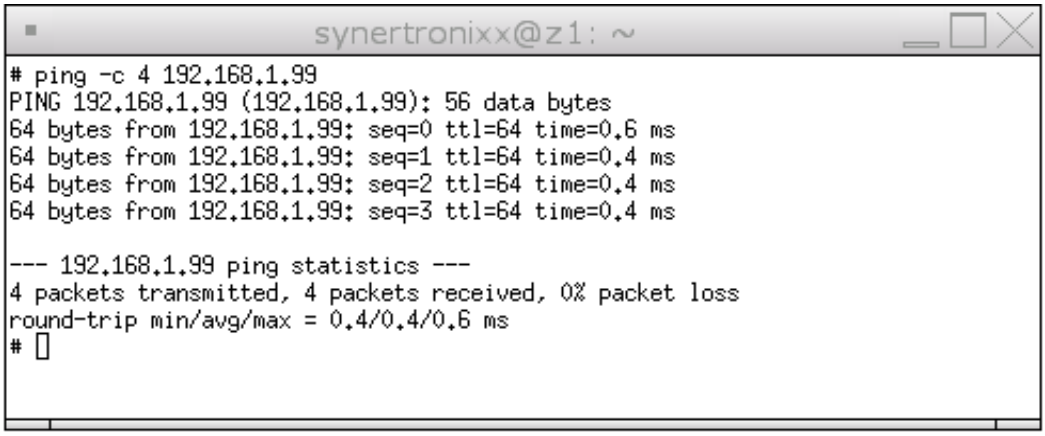


Figure 5: Checking the network configuration ping.

1.3 Using the serial interface (MS Windows)

It is also possible to communicate with MS Windows and a terminal application with the EVB9520.

First of all you have to start the terminal application. In this document we are using the *hyperterminal*. The first step is to give the new connection a name, as shown in figure 6. Click on OK to pass this step.

Figure 6: The name of the new connection.

The next window you will see is shown in figure 7. Here you choose the right COM port and also pass it with OK.

Figure 7: Choosing the right COM port.

In figure 1 you see the settings from the chosen COM port. In Table 1 are the needed settings shown.

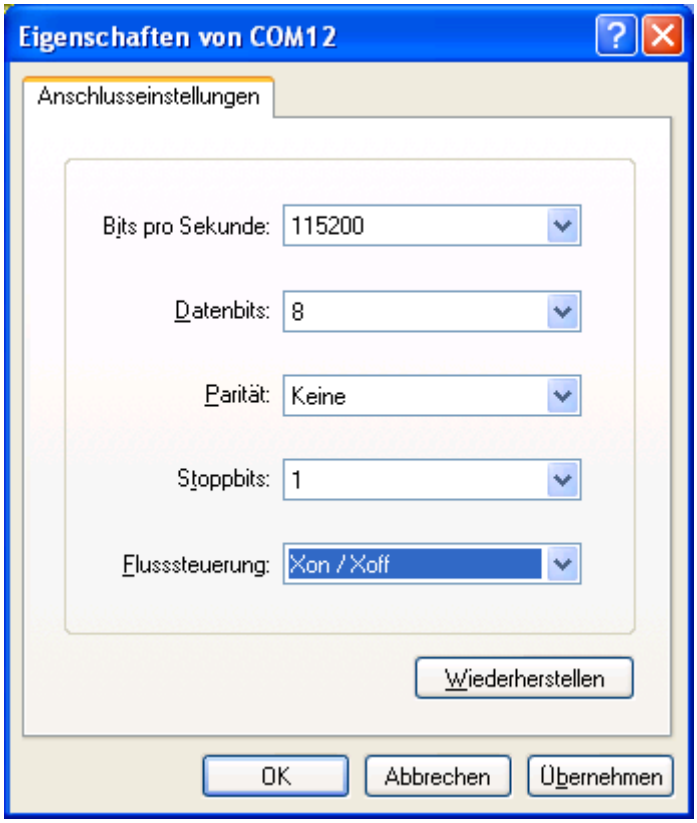


Figure 8: Settings of the serial connection.

Table 1: The parameters for the COM port.

Bits pro Sekunde	115200
Datenbits	8
Parität	keine
Stoppbits	1
Flusssteuerung	Xon/Xoff

In figure 4 is the output on the hypertermnal after boot up shown. The username and the password is the same as already mentioned in chapter 1.3.

```
username : root
password : pass
```

The configuration of the IP address is also the same like in chapter 1.3.

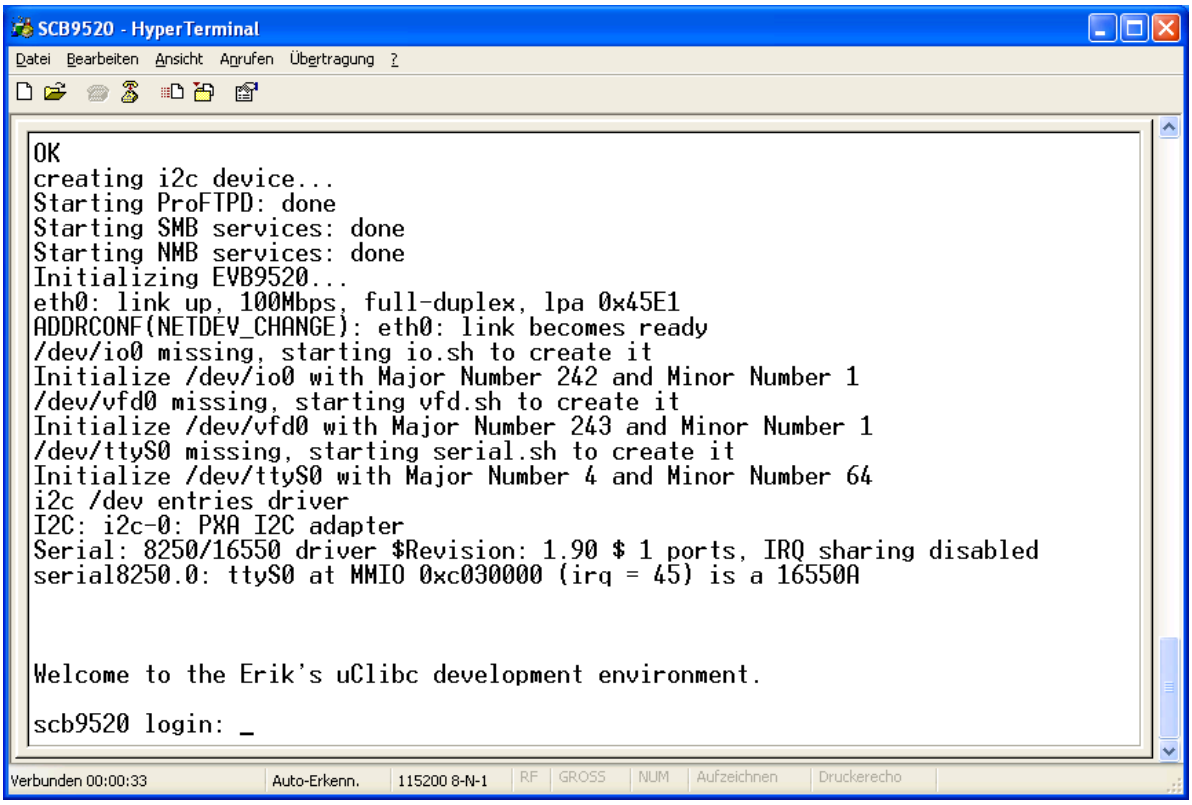


Figure 9: The output of the Hyperterminal.

1.4 Compiling applications for the EVB9520

1.4.1 Compiling "Hello World!"

For compiling applications for the EVB9520 you need a PC equipped with Linux. You

always compile on the PC with the cross compiler and then you copy it to the board. The cross compiler can be found on the provided CD.

First extract the compiler and set it up for using. Below you can see the commands. With *mount* we can use the content of the CD. Some Distributions mount CDs, pen drives and so on automatically. If you use one of these Distributions, you don't need the first and the last command.

```
mount /mnt/cdrom
cd /
tar xjf /mnt/cdrom/home/software/arm-linux-gcc-4.1.2.tar.bz2
umount -l /mnt/cdrom
```

The command *tar* extracts the archive into /usr/local/arm. If this path doesn't exist, it will be automatically created. To use the compiler, we have to extend the PATH variable. To do this, we will use the command *export* as shown in the next line.

```
export PATH=$PATH:/usr/local/arm/arm-linux-gcc-4.1.2/bin
```

The PATH contains all paths to directories with executable binaries or shell scripts. If you don't want to execute this command after every boot up, you should add this command into the /etc/profile.

Again, the compiler will be used on the PC and not on the EVB9520! All these operations you see above, must be done on your development PC and this one must be equipped with Linux.

In listing 1 you can see the source code of "Hello World!".

```
#include <stdio.h>

int main (void)
{
        printf("Hello World!\n");

        return 0;
}
```

Listing 1: The application "Hello World!"

Use the next command and its parameters to compile:

```
arm-linux-gcc -o hello -Wall hello.c
```

The parameters *-o hello* sets the name for the created binary, if you wouldn't set it, the default name “a.out” would be used, which is not very useful. The option *-Wall* shows more warnings than the default. For good programming you should prevent every warning. The last parameter is the file to compile.

After successful compiling you can copy the binary to the EVB9520, this can be done with *ftp* or *scp*. The file has to be copied to the directory /usr/local/bin. An example for copying with *scp* is given below.

```
scp hello root@192.168.1.199:/usr/local/bin
```

If you copy this file the first time, you have to change its rights and permissions with the *chmod* command. After that, you can execute it on the command line. For this, see the next figure 10.

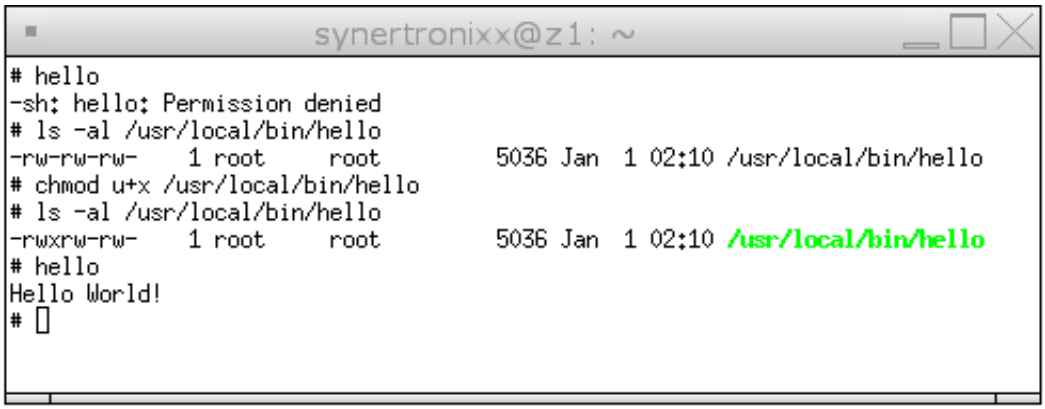


Figure 10: Using chmod and execute "Hello World".

1.4.2 Compiling the example evb9520

On the provided CD you get the source code of one example application. This code can be used for your first applications. To compile it, you can use the following command.

```
make -f evb9520.mak rebuild
```

Note, this compiling also has to be done on a Linux PC and not on the EVB9520 itself.

On the CD there is also provided the project file evb9520.vpj. If you already have the editor *slickedit* you can use this file for comfortable use.

To copy this file also to /usr/local/bin just follow the hints we gave you in chapter 1.3. Executing this application is done with the following command:

```
evb9520
```

2 Controlling the application

2.1 Using buttons to control

The application start with the following command.

```
evb9520
```

After connecting the EVB9520 to the power supply, the application *evb9520* starts in background. The typical behaviour is, all LEDs are lightning and on pushing one button the LED stops lightning until the button is not pushed. The first two buttons can be used to navigate through the menu.

The available points in the menu are:

- Displaying the current time.
- Displaying the current voltage.
- Displaying the input on the device ttySA2.
- Displaying the input on the device ttyS0.

An output on the the screen is not available, because printf outputs are quite heavy for the controller. You can also control the board with the help of *ssh*. First you login into the board, an example command is shown below.

```
ssh root@192.168.1.199
```

2.2 Controlling with the help of DeviLANControl

First you must supply the EVB9520 with power (DC 10V – 30V). Then connect it with a null modem cable to the PC. You have to use the debug interface for this connection. To login, start a terminal application. On figure 11 you can see an example session.

Figure 11: The login prompt

Some features of the EVB9520 can be controlled with a remote PC equipped with MS Windows. For this, we need to start the application *udp_config* with the following command on the board:

```
udp_config
```

If you don't change any settings in the boot up scripts, *udp_config* is already running on the board and you can skip this step.

The screenshot 12 after boot up.

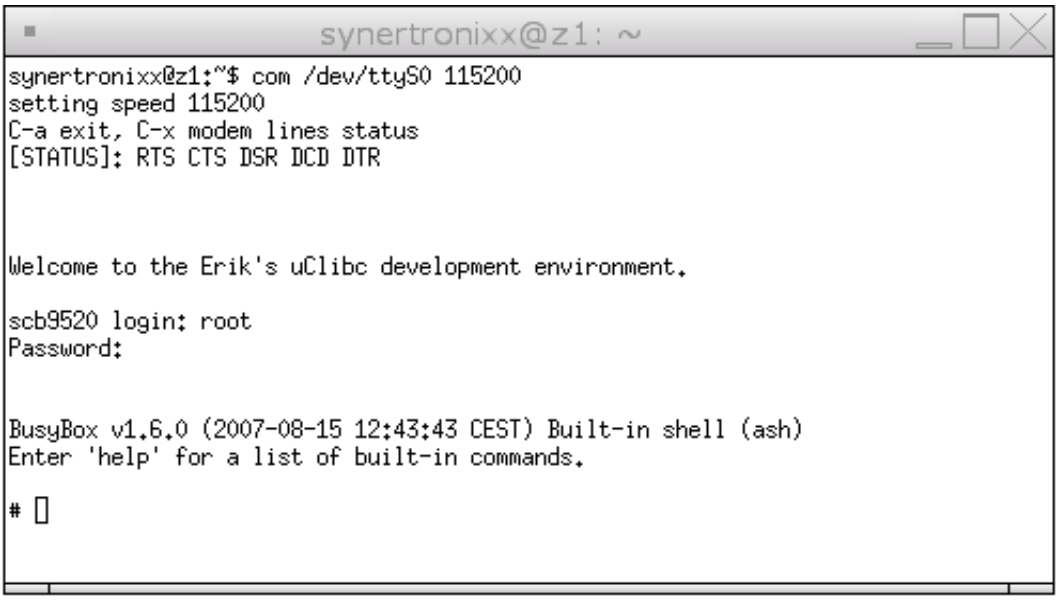


Figure 12: The application udp_config

Now please start DeviLANControl on you MS Windows PC like you can see in figure 13. Here you can see the tab „Verbindungsmanager“. In this tab you can see all active boards in your network. Perhaps you have to change the setting „UDP-Meldung senden an“ to „IP2002/SCB9328 (Port: 80)“. If you cannot see the board, try clicking the button „suchen“.

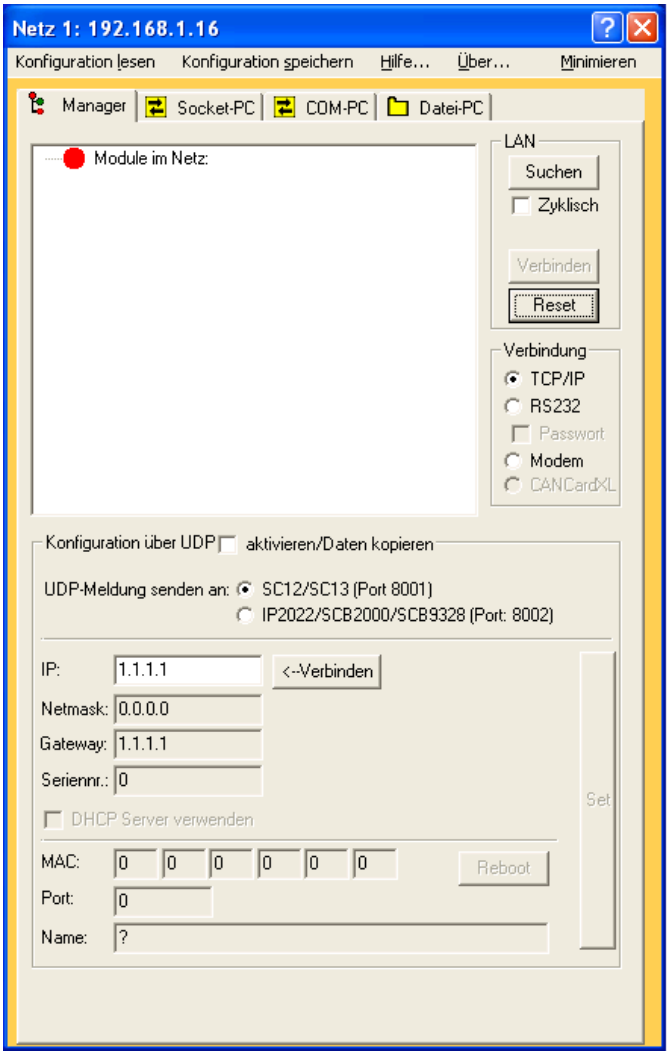


Figure 13: The application DeviLANControl.

In figure 14 you can see the output from *evb9520*. This application doesn't give messages to the stdout, you just get some messages if errors occur.

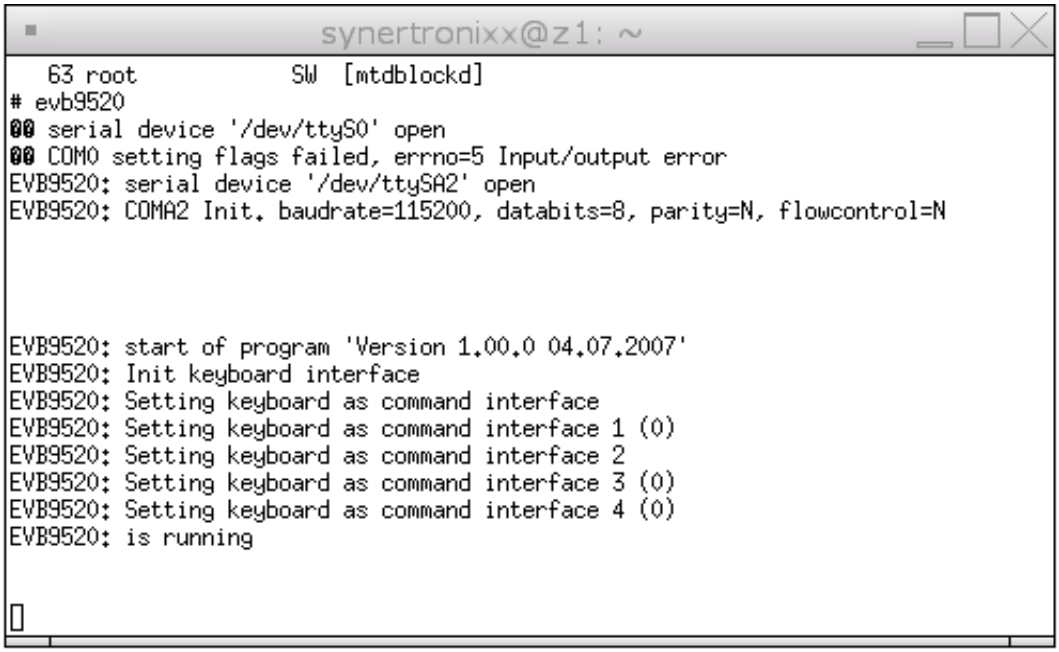


Figure 14: The output of the application evb9520.

In figure 15 you can see the DeviLANControl. In this picture you can see, that DeviLANControl found one board. Now you can connect to it with the button "Verbindung".

Every possible communication partner you see in DeviLANControl is expandable. In expanded view you can see some further information like the IP address, the used port, the net mask and the MAC address.

A short information to *udp_config*:

To identify or configure several running SCB9520 and EVB9520 in your local area network (LAN) you can use the application *udp_config*. For communication there is a command interpreter implemented. It reacts on UDP broadcast messages and answers with the current TCP/IP settings. Over UDP you can set the basic network settings. For this the file /etc/network/interfaces will be modified. It contains the network settings. For safety the old file will be saved as interfaces.pre_udpsocket.

To configure one board you can use DeviLANControl. An example session to configure the network settings is given below.

- Choose the board you want to control in the tree.
- Activate “Konfiguration über UDP” (now, automatically the data from the board gets copied to the input field).
- Activate “UPD-Meldung senden an (Port 8002)”.
- Change or edit the settings you want to.
- Confirm the settings by hitting the button “Set”.
- The new settings will be used after rebooting the board.

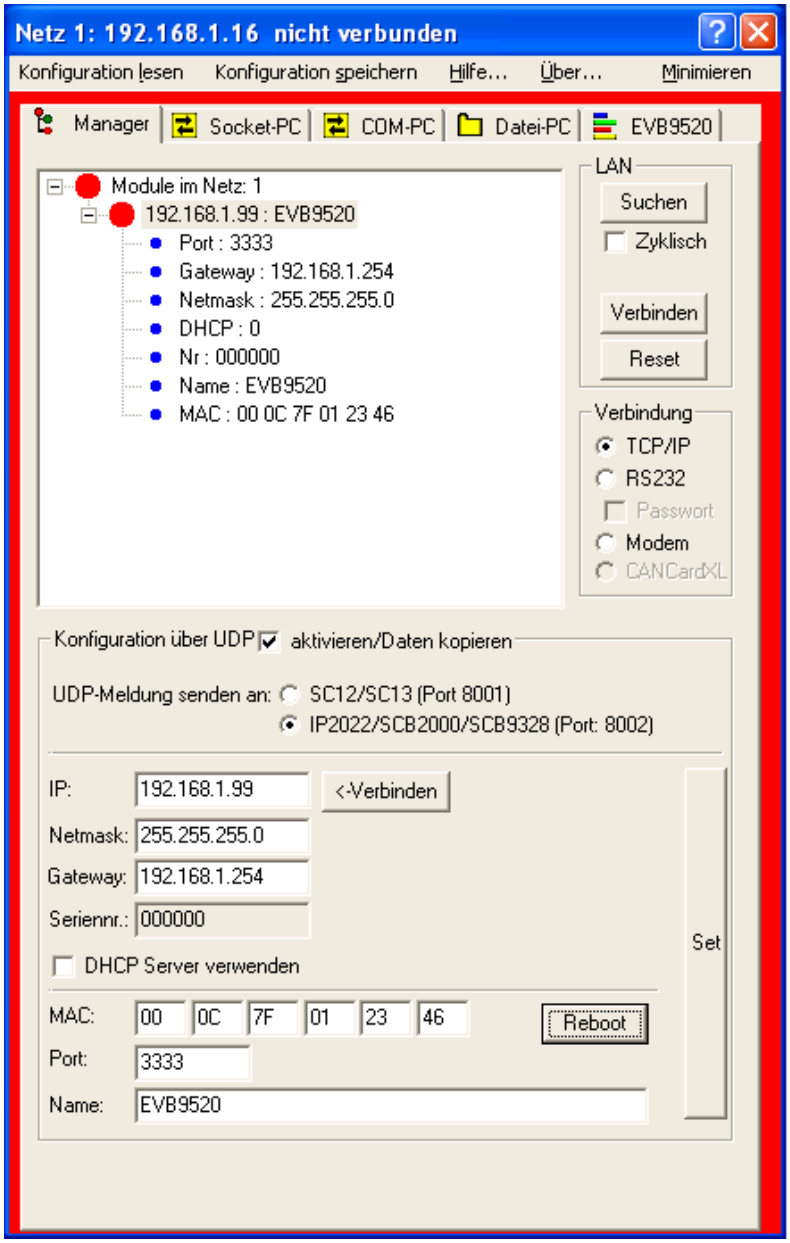


Figure 15: DeviLANControl after connecting with the application udp_config.

After successful login, the string “modul typ 9520” is sent to the board. If the EVB9520 receives this string, it creates a new tab with the name "EVB9520" (see figure 16).

The new tab contains some controlling elements. Under "Port A" you can see the state of the LEDs and under "Port B" you can see the state of the switches. In the middle of the widget you can see the current analogue value in the "ADC" section (dimension is mV). In this area you can also set the current baud rate and gain.

Below the "ADC" section is the "time" area. You can see the current time in it and even set it. You must have a look at the string to set the time, it must be exactly the same as shown

in the widget.

In the "Modus" section you can control the output of the display.

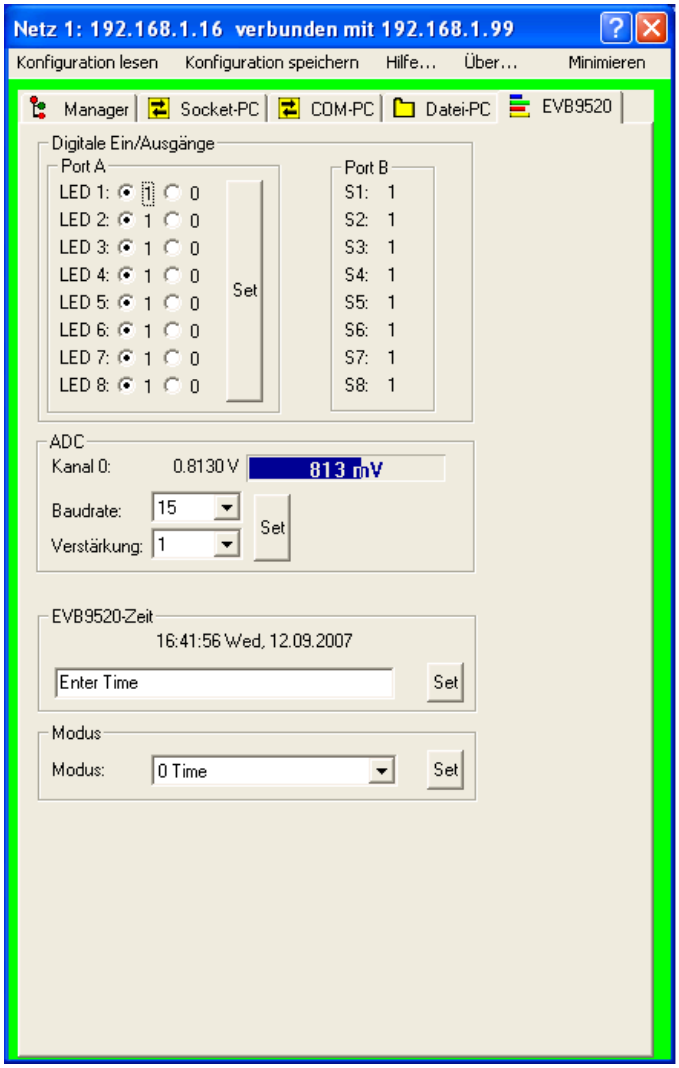


Figure 16: The new register "EVB9520".

If you send the string “data_renew” to the board, the current state of the LEDs and switches, the analogue value and the time will be updated. One example is shown in the figure 17.

DeviLANControl updates specific values in dependency to the current mode. If it runs in time mode, the time is always correct. If it is in ADC mode, the value shown in DeviLANControl is refreshed several times per second.

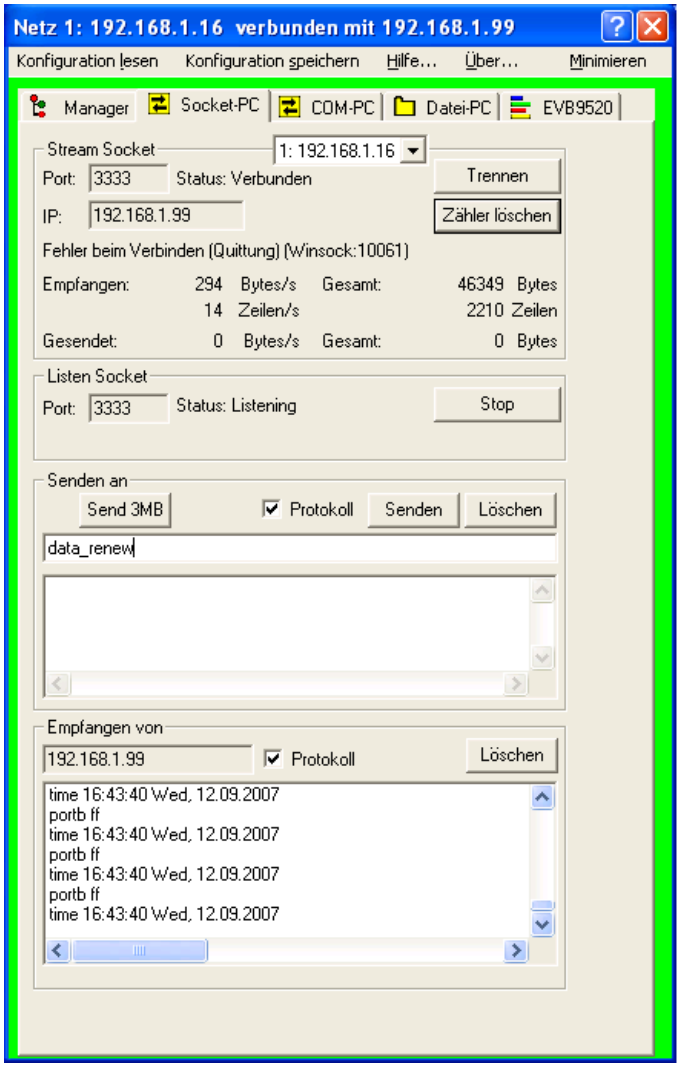


Figure 17: The command data_renew.

3 The hardware

The Evaluation Board EVB9520 is an example application to show you the features of the SCB9520.

Some characteristics of the board:

7. 10 / 100 MBit Fast-Ethernet Interface.
8. USB Device Interface (USB Low- and Full-Speed Support).
9. USB Host-Interface.
10. 16-Bit A/D converter.
11. 3 serial RS232 interface.
12. I²C-Interface (max. 2 MHz).
13. 8 digital inputs and outputs (switches / LEDs).
14. Gold cap buffered real time clock.
15. VFD Display.
16. power supply 8 - 30V DC.
17. 1 carrier for the SCB9520.

The figure 18 shows the position of the most interfaces.

You can find current information like the schematic EVB9520 on the homepage of the company synertronixx.

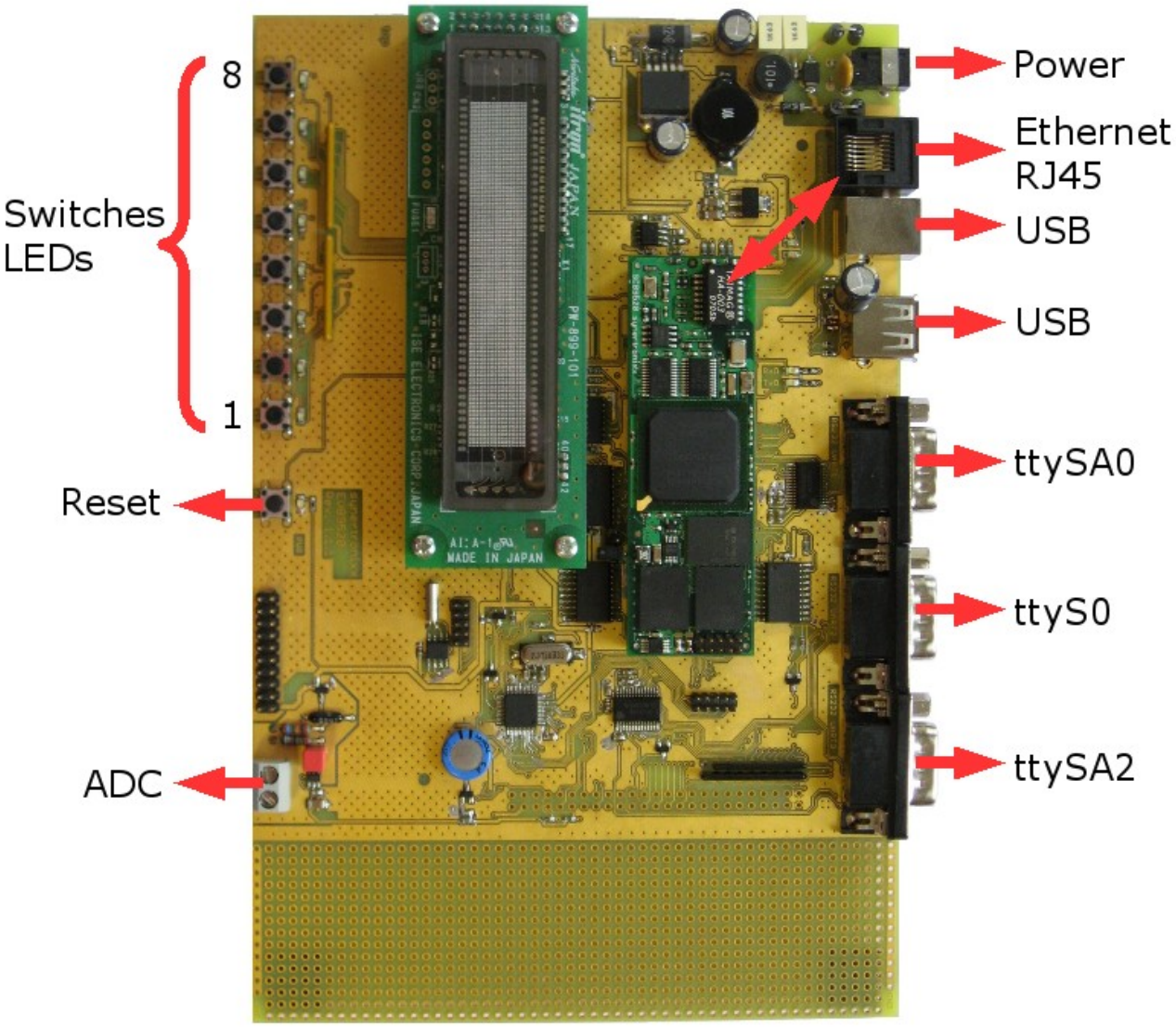


Figure 18: The structure of the Evaluation Board.

3.1 The structure

The real time clock and the ADC are connected to the I²C bus. In order to control or read these parts you have to use their addresses, which stands in their data sheets.

The LEDs and switches are both connected to the same data bus. With the help of the address decoder (shown in figure 19) you can choose to control the VFD, the switches or the LEDs. If the /CS3 is low and A16 and A17 too, the /CS_541 on the decoder output also becomes low.

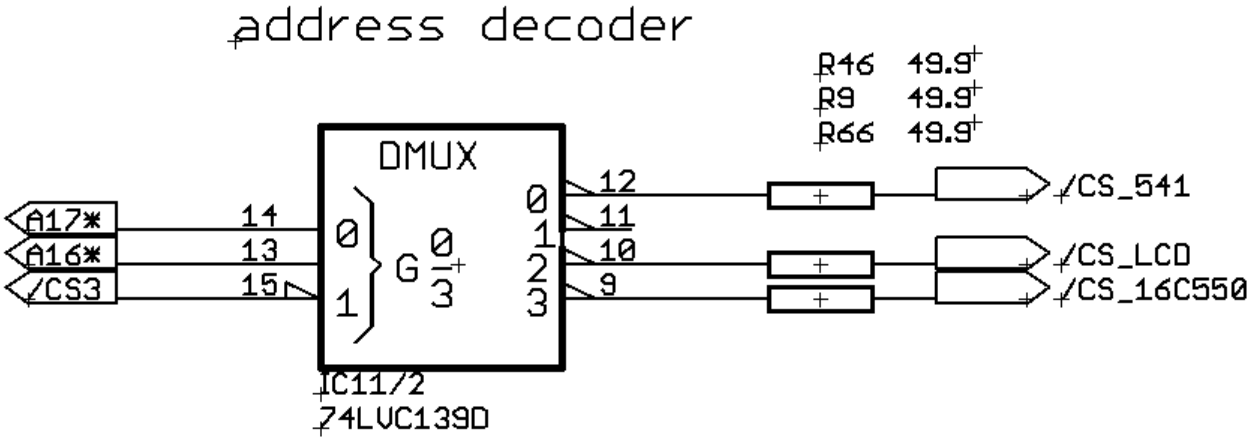


Figure 19: The address decoder.

The /CS_541 leads to another address decoder (see figure 20). This decoder is connected to /RD and /CS3. If /WR, /CS_541 and /CS3 is low, /CS541_WR* becomes low and the LEDs will be set according to the bus shown in figure 21.

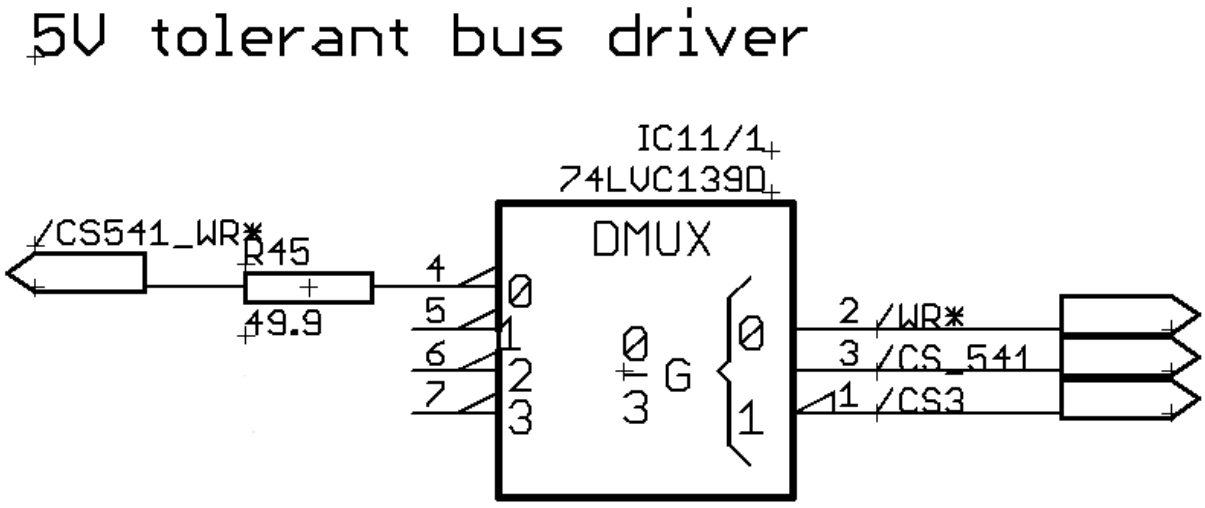


Figure 20: The Differentiation between reading and writing.

To read out the switches /CS_541 and /WR must be low (please have a look at figure 21). The state of /CS_541 can be set by software and the states of /RD and /WR are changing continuously and oppositional, which is implemented in the PXA 270.

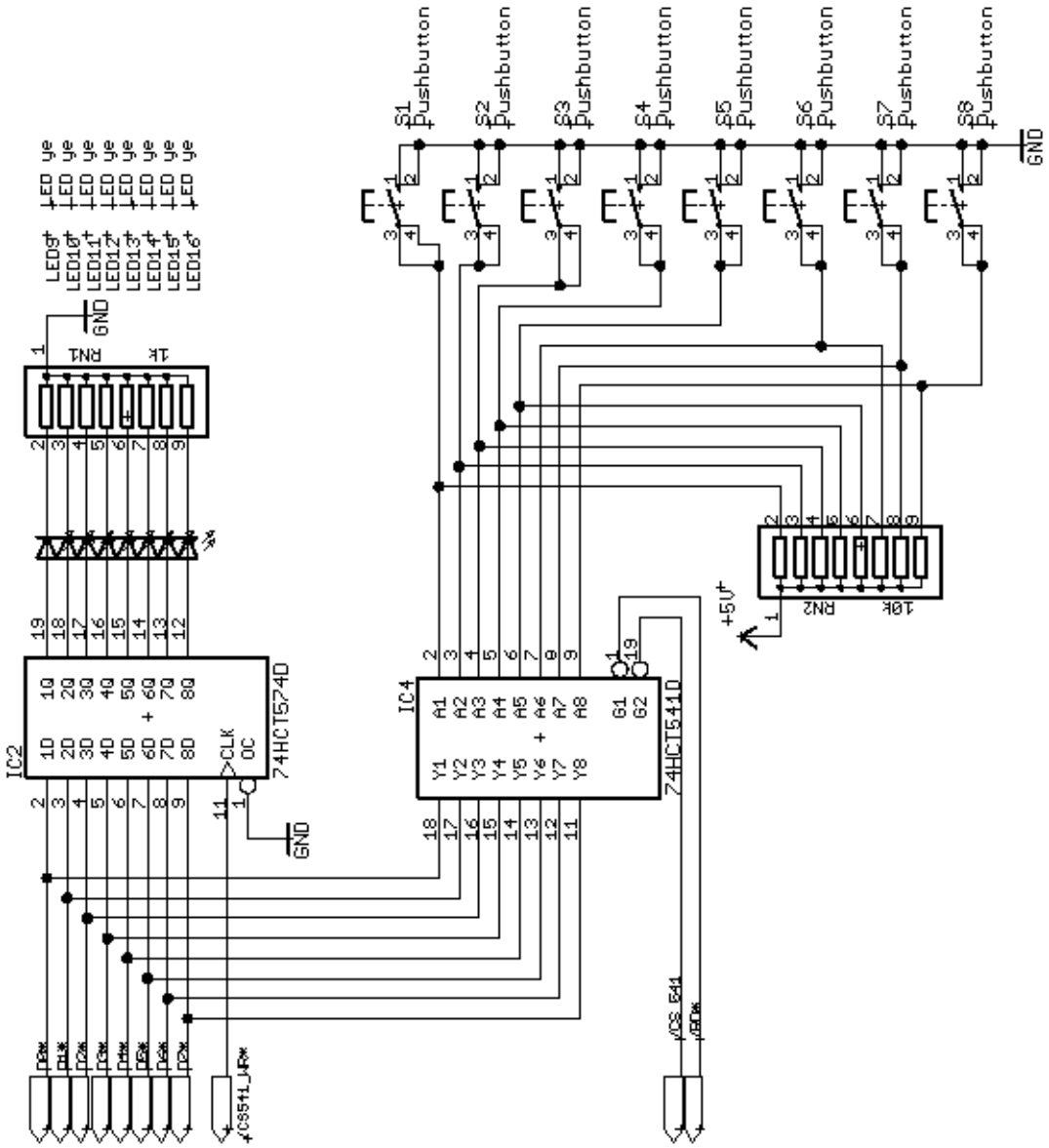


Figure 21: The switches and LEDs.

3.2 The real time clock

The M41T00 uses less current (about 0.3 mA). Please refer to the data sheet for further information. The real time clock can be supplied by the on board gold cap. If the main power supply is not connected, this cap can supply it for up to one week. The RTC offers the century, the year, the month, the hour, the minute, the second and the name of the week day. All is decoded in BCD numbers.

The wiring of the M41T00 is given in the figure 22.

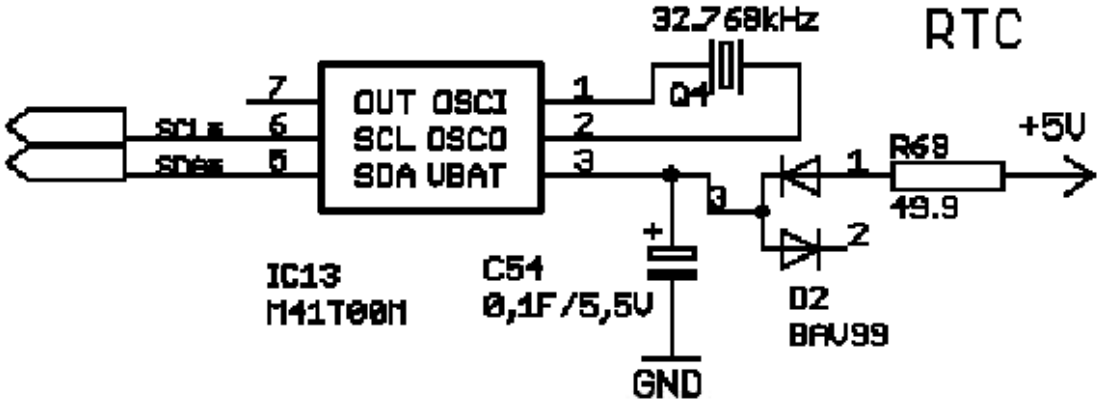


Figure 22: The wiring of the M41T00.

To control or read the M41T00 the generic Linux I²C bus driver is used.

On reading or writing you always get or set an array of 8 bytes.

address	data								function	range
	D7	D6	D5	D4	D3	D2	D1	D0		
0	ST	10 sec			second				second	00-59
1	X	10 min			minute				minute	00-59
2	CEB	CB	10 hour		hour				year/our	0-1/00-23
3	X	X	X	X	X	day			day	1-7
4	X	X	10 date		date				date	1-31
5	X	X	X	X	month				month	1-12
6	10 year				year				year	0-99
7	OUT	FT	S	calibration					control	

Table 2: The register of the M41T00

In table 2 you can see the content of the register of the M41T00. The bits marked with an X contain useless information. If the ST bit is set to one, the oscillator stops and if you set it back to zero, it starts within a minute. If the CEB bit is set, you can recognize a new

century on checking the CB bit. If this bit toggles, another century has gone. The bits S, FT and OUT are used for checking the oscillator, they are not used in this project.

All time information is coded in BCD. Use the following formulas to calculate to BCD or to decimal numbers.

$$X_{10} = \left(\frac{X_{16}}{16} \right) \cdot 10 + X_{16} \, mod \, 10 \tag{1}$$

$$X_{16} = \left(\frac{X_{10}}{10} \right) \cdot 16 + X_{10} \, mod \, 16 \tag{2}$$

3.3 The analogue digital converter

The ADC component ADS1110 requires a current of 0.24 mA. It has a resolution of 16 bit and 4 adjustable steps for the baud rate and the gain. For more information, please refer to the data sheet.

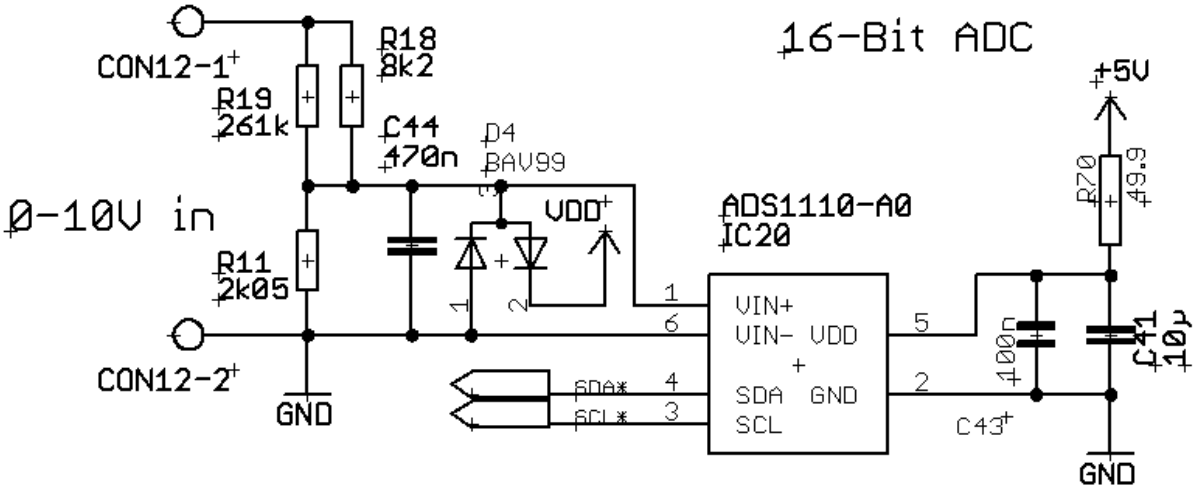


Figure 23: The ADC.

The device is connected to the I²C bus and can also be controlled with the help of the generic Linux driver.

In Figure 23 you can see the schematic. The ADS1110 has an adjustable sample rate from 15, 30, 60 or 240Hz. The gain is adjustable, too. Here you can choose between 1, 2, 4 and 8. The higher gains are used for higher resolutions on lower voltages. The formula to calculate the value on the input of the ADC considers the gain, so the result is always the right one. The formula you should use, is given under 4.

There are three registers. Every register contains 8 bytes. The first two bytes are the data bytes and the last one is the configuration byte.

The module can be read as often as you want to. If you read it faster than the new value is written to the data registers, you just read the same value twice or even more often. A bit, that shows you an ended conversion is not available.

Bit	7	6	5	4	3	2	1	0
Name	ST	0	0	SC	DR1	DR0	PGA1	PGA0

Table 3: The configuration register of the ADS1110.

In Table 3 you can see the configuration register. The seventh bit has no use for the continuous mode. The fourth one must be zero for continuous mode. The bits 3 and 2 are setting the baud rate and the bits 1 and 0 are setting the gain (see table 4 and 5).

DR1	DR0	data rate	maxcode (max. resolution)	mincode (min. resolution)
0	0	240SPS	2047	2048
0	1	60SPS	8191	8192
1	0	30SPS	16383	16384
1	1	15SPS	32767	32768

Table 4: The bits to control the baud rate.

PGA1	PGA0	gain
0	0	1
0	1	2
1	0	4
1	1	8

Table 5: The bits to control the gain.

With formula 3 you can calculate the value on the ADC. To prevent float calculating you can use the factor 1000.

$$\Delta U = \frac{data \cdot 2{,}048}{gain \cdot (-mincode)} \tag{3}$$

$$\Delta U = \frac{data \cdot 10}{gain \cdot (-mincode)} \tag{4}$$

The voltage divider has an proportion from 1 to 4,878 (see formula 7). For the right result use the factor 10 and put it into the formula 4.

$$\Delta U = U_0 \cdot \left(\frac{2050\,\Omega}{2050\,\Omega \cdot \left(\frac{8\text{k}2\,\Omega \cdot 261\,k\,\Omega}{8\text{k}2\,\Omega + 261\,k\,\Omega} \right)} \right) \quad (5)$$

$$\Delta U = U_0 \frac{2{,}05}{10} \quad (6)$$

$$\Delta U = \frac{1}{4{,}878} \quad (7)$$

3.4 The VFD Display

The on board display is a Vacuum Fluorescent Display (VFD). The VFD is connected to the board with an 8 byte data bus. The power supply is 5V and this display offers 11 monospace and proportional fonts.

The schematic is shown in figure 24.

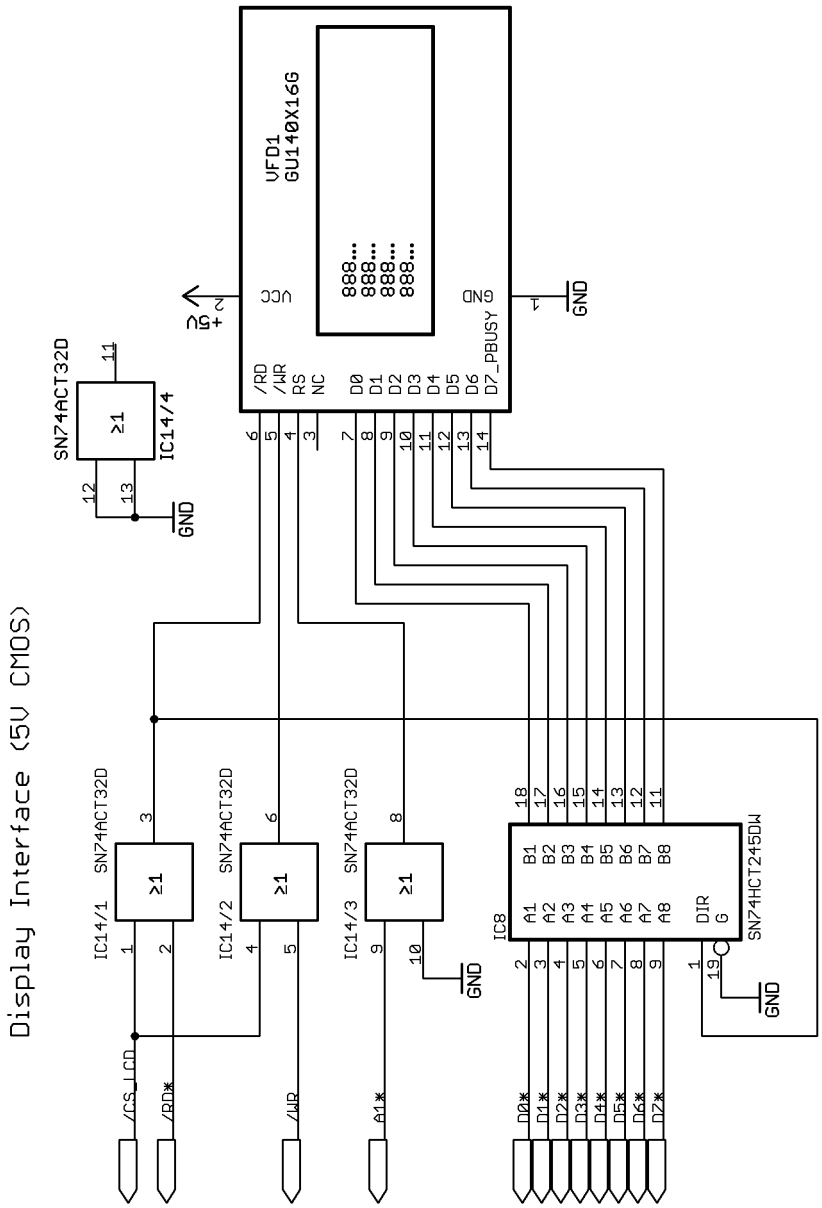


Figure 24: The VFD display.

To write some data to the display /CS_LCD and /WR must be low. /CS_LCD becomes low, if /CS3 and A16 are low and A17 is high.

The VFD has a pin RS, but this cannot be used. On other displays, this bit is normally used to differ between data and command words. Here you just send your commands or data word by word to the display and everything works fine. For further information, please refer to the data sheets.

3.5 The interfaces

There are several available interfaces on the EVB9520. You can find the I²C bus, three RS232, one Ethernet interface, one USB host and one USB device controller.

Two of the serial devices are internal. For the third external one you need the module 8250. You have to load it by yourself, e.g. with a rc-script at boot up.

For the I²C bus you must load the modules *i2c_core*, *i2c_pxa* and *i2c_dev*. The *i2c_core* will be loaded by Linux automatically, but the others you must load by yourself.

4 The software

The application *evb9520* and its source code is an example to show the customer how to program on an embedded Linux system. For this you need the *toolchain buildroot* to cross compile for the EVB9520.

The EVB9520 is equipped with the application *evb9520*, which uses several features from the board.

To develop you should connect the board like it is shown in figure 25. The board is connected with a null modem cable to the development PC and it is connected to a router, hub or switch to get network access. The development PC (Linux) is connected to the network. Perhaps you also want to control the board with the MS Windows application *DeviLANControl.* If yes, you need a Windows equipped PC, which is connected to network, too. After you connect everything in the right way, you can communicate with it with the help of a terminal program. For copying you can use *scp* or *ftp*.

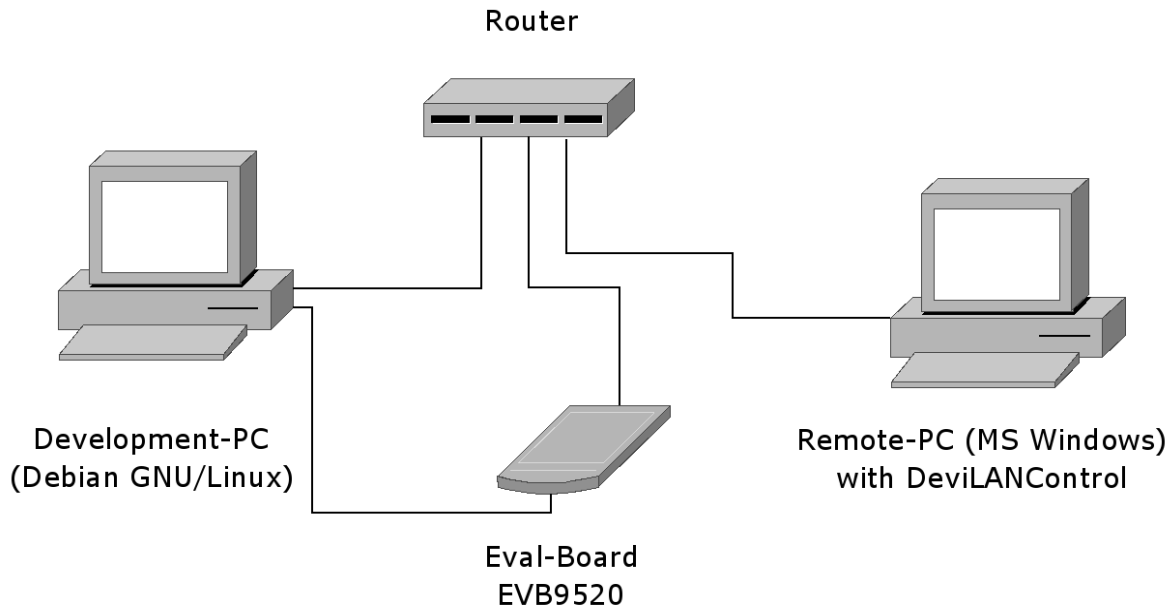


Figure 25: The connection board, development PC and Remote PC.

4.1 The kernel

The board is equipped with a Linux kernel 2.6.31 and *Busybox*. *Busybox* is a set of applications, which provides a shell and tools like *ls* (list segments), *cd* (change directory) and *mkdir* (make directory). It is often used on embedded systems, because of its little size.

In figure 26 you can see an overview of the communication between an application and a kernel module.

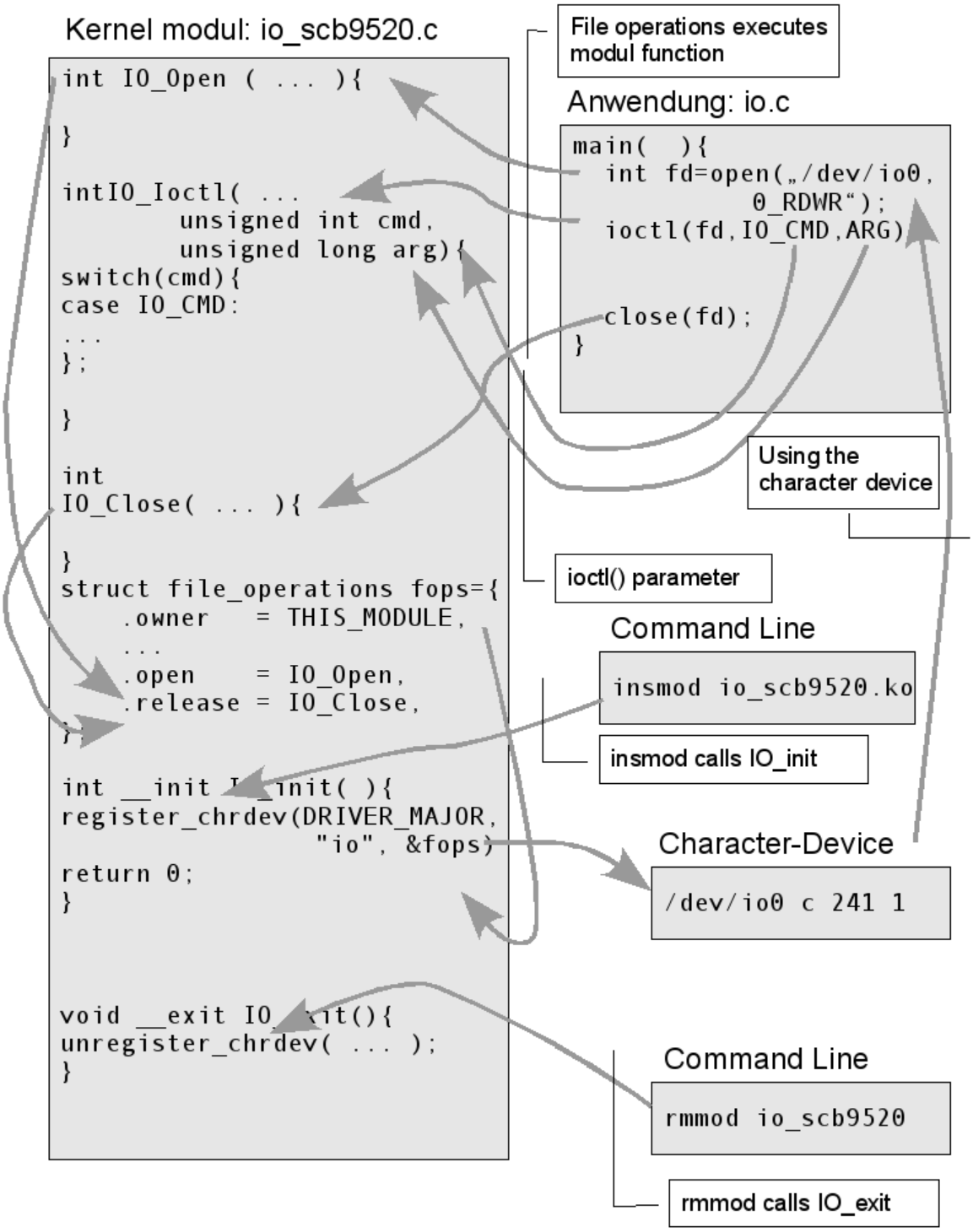


Figure 26: The connection between kernel module and application.

The modules should provide the features of the VFD and the IOs (switches and LEDs). The address of the memory PXA_CS3_PHYS is defined in the header asm/arch/scb9520.h. To access their functions the well known Linux calls *open*, *close*, *write*, *read* and *ioctl* are implemented.

4.2 The I/O module

Developing modules for the Linux kernel follows every time the same procedure. The 5 most important calls are *open*, *close*, *read*, *write* and *ioctl*.

The names of these calls will be connected to functions which must be developed. The connection will be generated with a structure of type *struct file operations,* see listing 2. The names of the functions are not important, but if you want to have well readable code, use self explaining names, like the names you can see below.

- IOOpen
- IOClose
- IORead
- IOWrite
- IOIoctl

```
struct file_operations {
        struct module *owner;
        ssize_t (*read) (struct file *, char __user *, size_t,
                                        loff_t *);
        ssize_t (*write) (struct file *, const char __user *,
                                        size_t, loff_t *);
        int (*ioctl) (struct inode *, struct file *,
                        unsigned int, unsigned long);
        int (*open) (struct inode *, struct file *);
        int (*release) (struct inode *, struct file *);
};
```

Listing 2: Some elements of the structure struct file_operations.

In listing 2 you just see a part of this structure. The components we don't need here are not mentioned.

The assignment of the *IOOpen()* function to the *open* call is done by the addresses of the functions, see listing 3. The other calls will be connected to the other functions in the same way.

The assignment from the macro THIS_MODULE to the *owner* call is done for security. If this assignment has been done, the module cannot be unloaded during it is used.

```
static struct file_operations fops = {
        .owner   = THIS_MODULE,
        .ioctl   = IOIoctl,
        .open    = IOOpen,
        .write   = IOWrite,
        .read    = IORead,
        .release = IOClose,
};
```

Listing 3: The assignment of the functions to the structure fops.

To connect the hardware (kernel module) with the software (application), you need a device file. In this case we create a character device with the following command.

```
mknod /dev/io0 c 242 1
```

The parameter *c* creates a character device with major number *242* and the minor number *1*. The major number binds the module to this file. This number with this minor number is unique in the whole system.

```
module_init( IOinit );
   module_exit( IOexit );
static int __init IOinit(void)
{
        CS3_mem = ioremap(PXA_CS3_PHYS, 0x2);
        if( register_chrdev(DRIVER_MAJOR, "io", &fops) != 0){
                printk("register_chrdev failed\n");
                return -EIO;
        }
        io_infos.amount_leds=8;
        io_infos.amount_switches=8;
        return 0;
}
static void __exit IOexit(void)
{
        unregister_chrdev(DRIVER_MAJOR,"io");
        iounmap(CS3_mem);
}
module_init( IOinit );
module_exit( IOexit );
```

Listing 4: The functions to load and unload the module.

In listing 4 you can see one way to load and unload this module. The command *insmod io_scb9520.ko* calls the function *IOInit(). register_chrdev()* registers the module with its major number to the kernel. The function *ioremap()* allocates some memory.

To unload this module use *rmmod io_scb9520*. Now the function *unregister_chrdev()* unregisters the module and *iounmap()* frees the allocated memory.

```
static int
IOOpen ( struct inode *io_device, struct file *instance )
{
        return 0;
}
```

Listing 5: Die IOOpen-Funktion.

In Listing 5 you can see the dialogue for the *open* call. The parameters are *struct inode *io_device* and *struct file *instance*. The first one contains all elements, which defines the device file, e.g. owner and access rights. The second parameter allows to control the mode you access the device, reading, writing or both. In this function you can handle the access mode, but this is not used in this example.

```
static int
IOClose( struct inode *io_device, struct file *instance )
{
        return 0;
}
```

Listing 6: The IOClose function.

In Listing 6 you can see the *IOClose* function.

```
static int
IOWrite( struct file *file, const char __user *user,\
                            size_t cnt, loff_t *offset )
{
        int ret_val = 0;
        struct IOInOut io_in_out;
        if(copy_from_user( &io_in_out, user, cnt)!=0){
                printk("Too much for 8 LEDs...\n");
                ret_val = -1;
        }
        writeb( io_in_out.leds, CS3_mem );
        return ret_val;
}
```

Listing 7: The IOWrite function.

In Listing 7 you can see how to set the LEDs. The function *copy_from_user()* is used to copy data from user space to kernel space. The parameters of the function are the address of the source variable and the address from the destination variable. The last one

is the amount of bytes to copy. The function returns the amount of bytes, who still need to be copied. If everything could be copied, it returns 0. *writeb()* writes one byte to the address, given as second parameter. The first parameter is the value to be copied.

```
static int
IORead( struct file *file, char __user *user, size_t cnt,\
   loff_t *offset )
{
        int ret_val = 0;
        struct IOInOut io_in_out;
        io_in_out.switches=readw(CS3_mem);
        if(copy_to_user(user, &io_in_out, cnt)!=0){
                printk("Too much for 8 switches...\n");
                ret_val = -1;
        }
        return ret_val;
}
```

Listing 8: The IORead function.

You can see the *IORead()* function in listing 8. In this function one byte will be read from the address CS3_mem. With the *copy_to_user()* function the byte is copied from kernel space into user space. The parameters from *copy_to_user()* are the address out from user space, the address from kernel space and the amount of bytes to copy. This function also gives back the missing bytes. If all could be copied, 0 will be returned.

```
static int
IOIoctl( struct inode *io_device, struct file *instance,\
                           unsigned int cmd, unsigned long arg )
{
        int ret_val;
        switch (cmd){
                case IO_GET_INFOS:
                        copy_to_user( (void*)arg, &io_infos, \
                                       sizeof(io_infos) );
                        ret_val = 1;
                        break;
                default:
                        printk("IO ioctl command not known\n");
                        ret_val = -1;
        }
        return ret_val;
}
```

Listing 9: The IOIoctl function.

To implement simple commands you can use *ioctl()*. To do such things, you just give a special parameter to the call and in the *ioctl()* function it will be analysed. The special parameters are just simple defines out of the header io_scb9520.h. In listing 9 you can see a short example.

4.2.1 The VFD modul

As already mentioned all modules are quite similar, so the vfd_scb9520 module also looks quite similar as the io_scb9520 module. This module controls the VFD module. It also uses the functions *register_chrdev()* and *unregister_chrdev()* for this. One additional function is the *VFDFirstInit()*, which initializes the module and sets settings to the VFD. The bytes we send to it, are given in the listing 10 below. The meaning of these keywords can be read in the data sheet.

```
static void
WaitPBusy (void)
{
        while(readb(CS3_mem+2)&0x80);
        udelay(10);
}
```

Listing 10: The function WaitPBusy() waits for a successful completion of the command.

When you write commands or data to the display, you must be sure, that the command is already completed before you can write the next command or data to it. For this you can check the PBusy bit. If this becomes 0, the command is completed. Unfortunately the SCB9520 is too fast and so we must wait with the delay function.

Normally you try to prevent waiting with delay or sleep, because it is so called active waiting.

4.2.2 The I²C bus

The Linux kernel provides a generic I²C bus kernel module to handle I²C devices. You can access the devices which are used in this application with the well known commands like *open*, *close*, *read*, *write* and *ioctl*.

For further information please refer to Philips and their specifications.

4.3 The application

4.3.1 A description of the application

This application is a demonstration to give a little help into driver and application development under Linux. The application *evb9520* consists of several C and header files. A short list of the files is given below.

- evb9520.c

- ads.c/h
- clock.c/h
- io.c/h
- serial_comA2.c/h
- serial_com0.c/h
- server.c/h
- terminal.c/h
- vfd.c/h

With the buttons you can control the board. With S8 you increment the menu and with S7 you can decrement the menu. In menu 0 you can read the time on the VFD, in menu 1 you can read the current analogue value and on menu 2 and 3 you can read the input from /dev/ttySA2 and /dev/ttyS0.

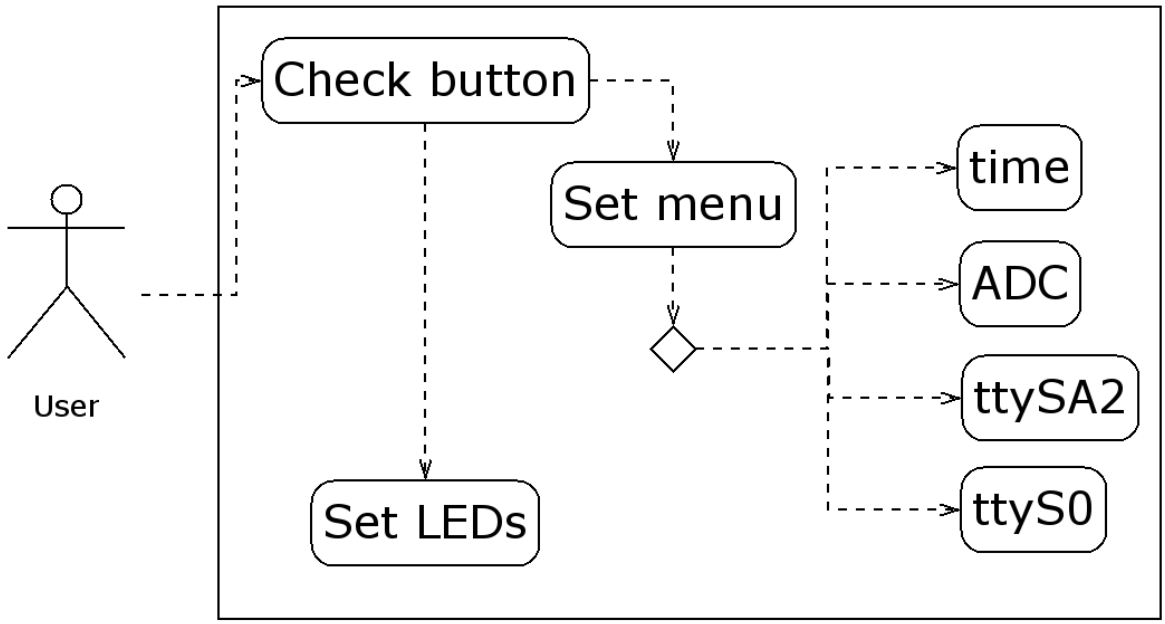


Figure 27: Controlling the board with the buttons.

On the figure 27 above you can see a controlling sequence. With the buttons the menu is set.

Another way to control the board is the TCP/IP protocol. For this you need a MS Windows PC equipped with the application DeviLANControl. To establish the connection, please have a look some chapters before.

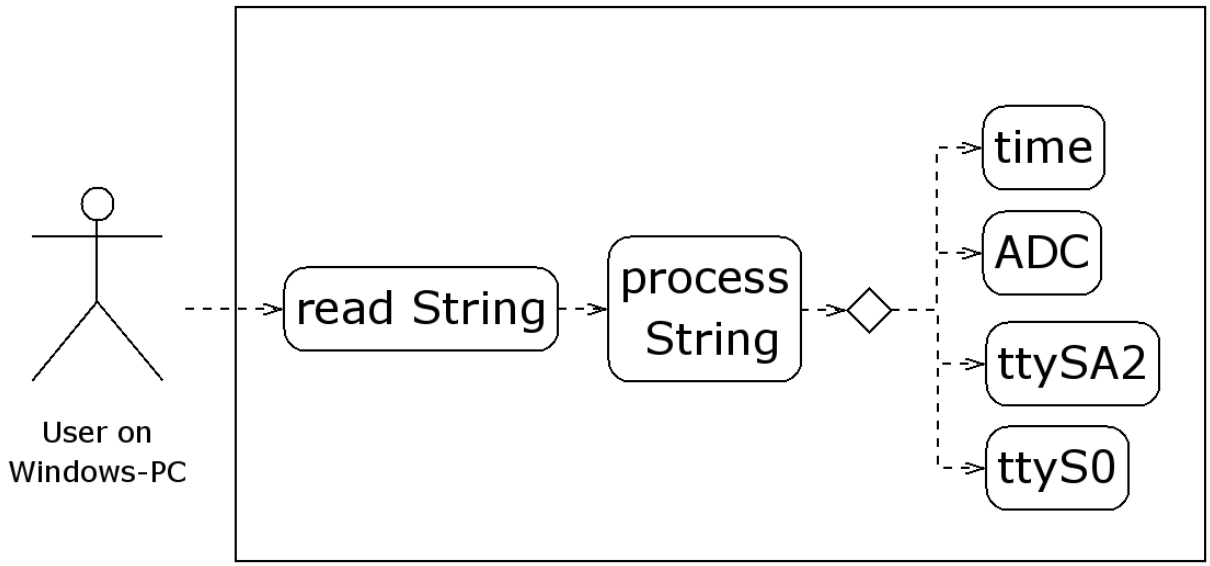


Figure 28: Controlling the board with TCP/IP.

The main routine first opens the following devices vfd0, i2c0, io0, ttyS0 and ttySA2. If one device could not be opened the application breaks. The serial devices are set to none blocking.

In figure 29 you can see an overview of the routine *main()*. In this function a do while loop runs until the variable running becomes 0. With *IOSetLeds()* the LEDs will be set. On hitting one button the function *SetMenu()* will be called. The *select()* function watches the TCP/IP messages. If there is one network message, the routine *ServerGetMsg()* will be called and will give it to the *DataHandler()* function, which analyses the received messages.

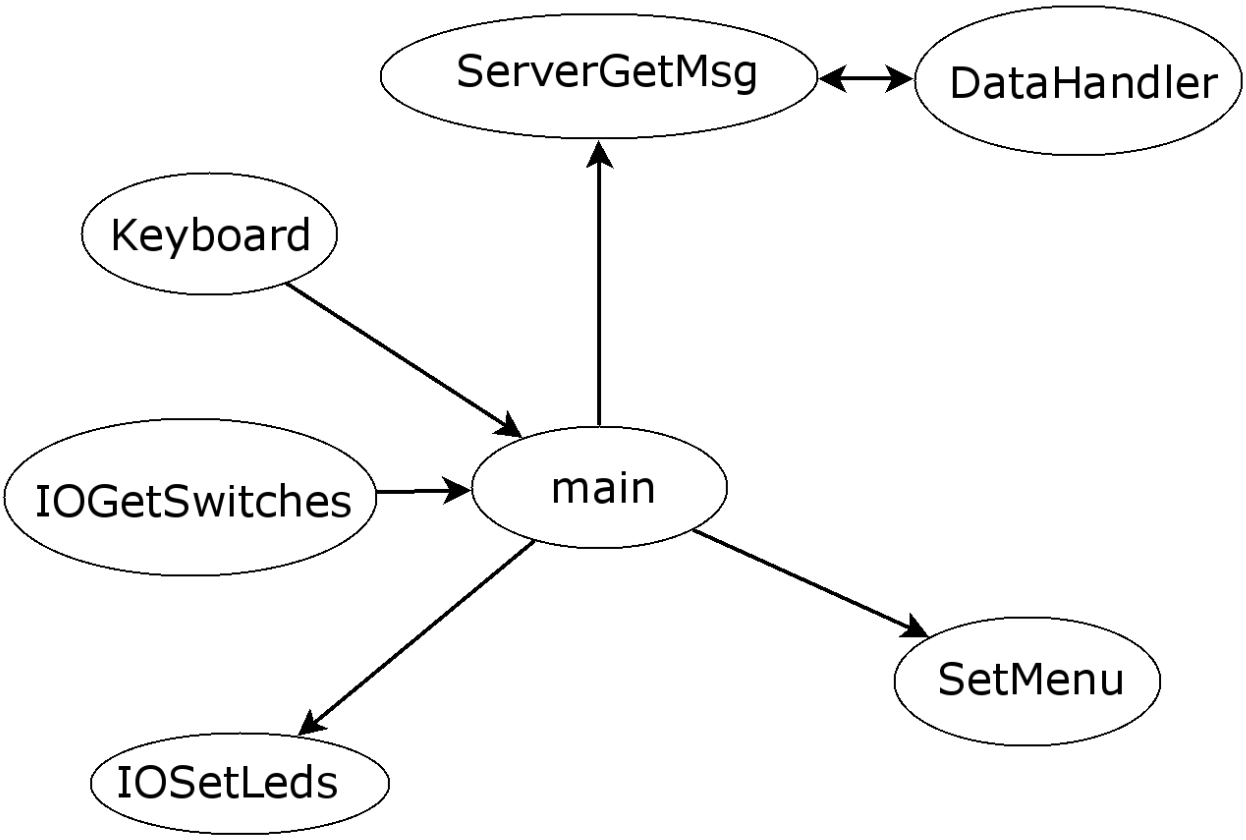


Figure 29: The main routine.

In Figure 30 you can see the *SetMenu()* function. These function creates the menu, which you see on the VFD.

- Menü 0 : Time.
- Menü 1 : Analogue value.
- Menü 2 : Input from ttySA2.
- Menü 3 : Input from ttyS0.
- Menü 4-9 : Free for use.

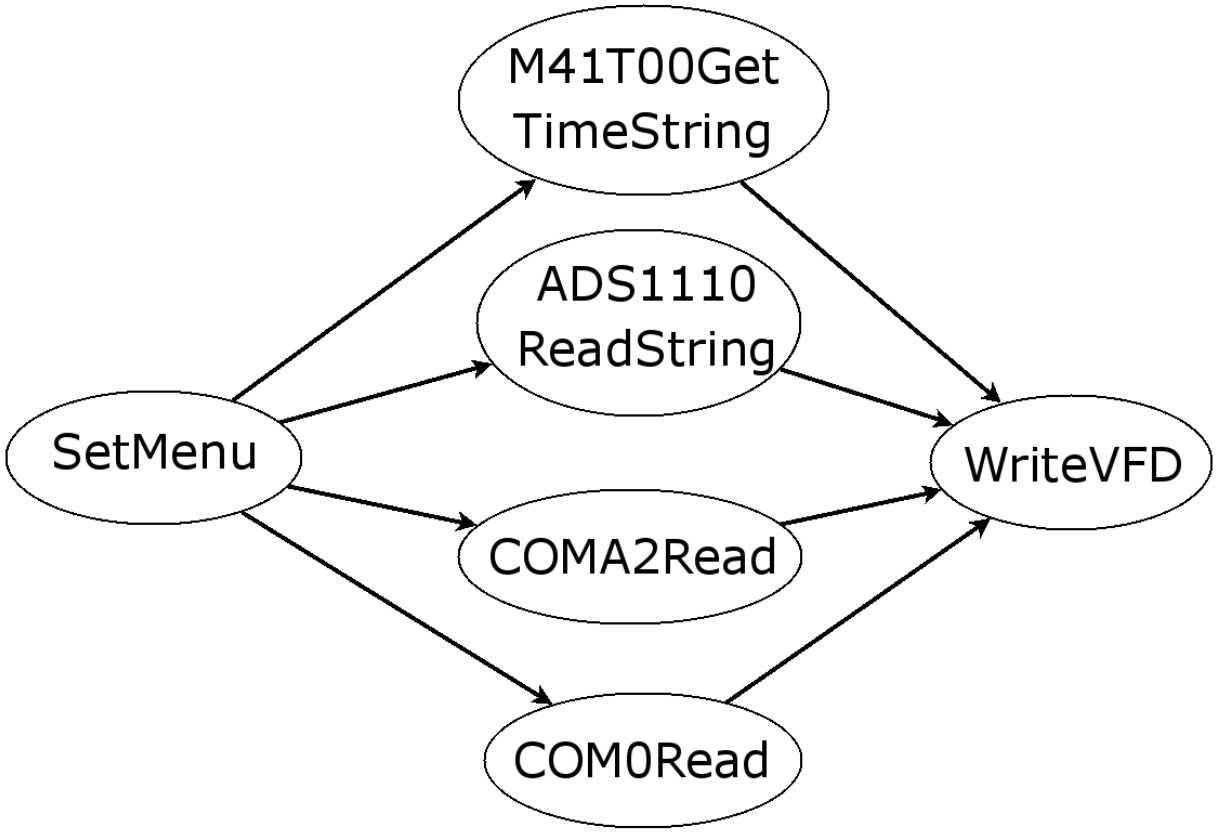


Figure 30: The SetMenu routine.

In figure 31 the *DataHandler()* is explained. The *ServerGetMsg()* reads the TCP stream until EOL (End Of Line). After that, it calls the *DataHandler()*, which analyses the string with the help of the *strstr()* routine.

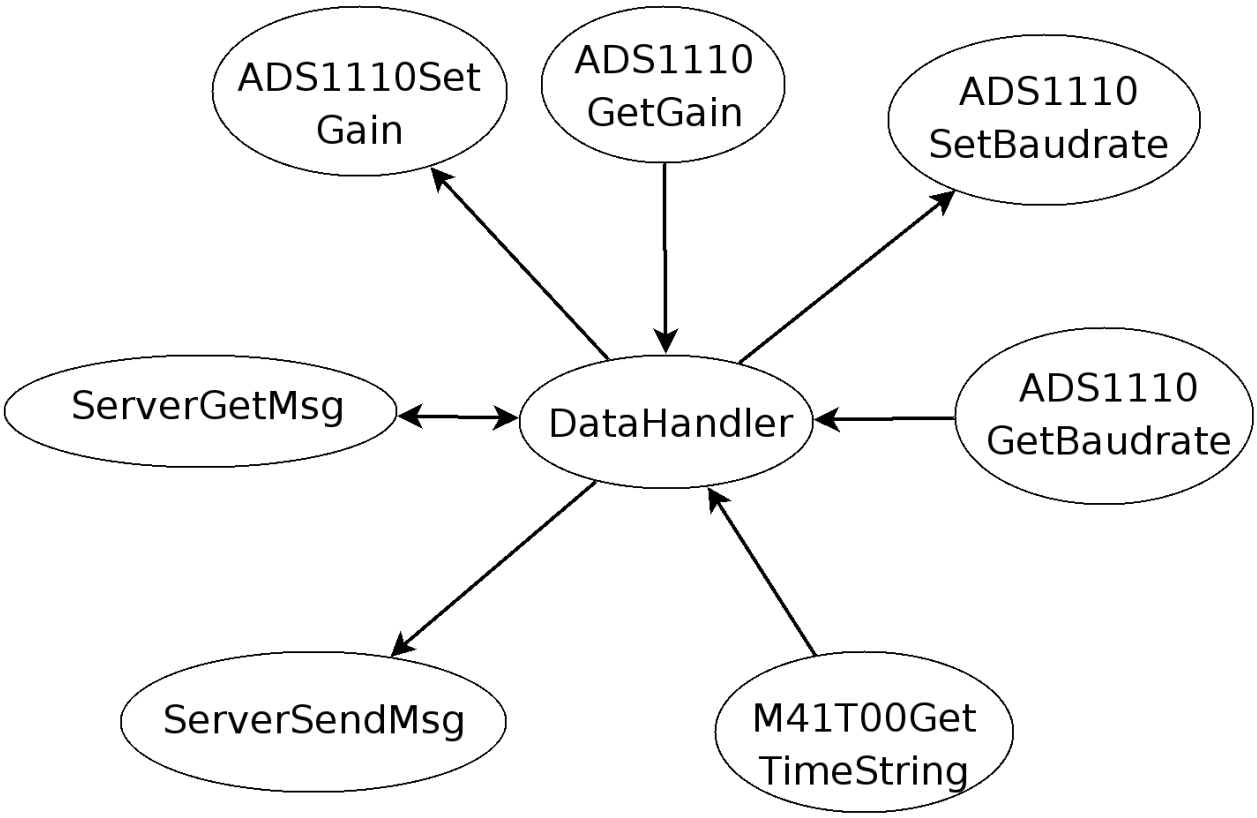


Figure 31: The DataHandler routine.

4.3.2 The buttons and LEDs

The buttons and LEDs are accessed with the help of the io0 device. This device is created at boot up. These IOs can be read and written. For this you also use the commands *open*, *close*, *write*, *read* and *ioctl*.

You can read out every pushed button and set the LEDs. The function *ToggleBit()* helps you to invert the current state of one LED.

4.3.3 The real time clock

The RTC (real time clock) is implemented with the I²C bus. Communication over I²C follows everytime the same procedure. There are two communication partners. One is the master and the other is the slave. On every communication the SCB9520 is the master and the RTC or the ADC is the slave.

The first step is to open the i2c0 device with the help of the open call. In the next line you see the command to set the RTC to slave.

```
ioctl(fd_clock, I2C_SLAVE_FORCE, 0x68)
```

fd_clock is the file descriptor which shows to the I²C device, I2C_SLAVE_FORCE is an *ioctl* command out of the generic Linux driver. The 0x68 or 68H is the address of the real

time clock.

Then you read from the device or write something to it. Everytime you read or write to the device, you read or write 8 bytes. To convert the numbers from decimal to BCD or back, you can use the routines *DecToBcd*() and *BcdToDec*(). Before you use these functions you have to mask the not needed bits. Please have look at the source code or the data sheet for those bits.

You must write to the RTC at least once, before you read the first time from it.

4.3.4 The analogue digital converter

The analogue digital converter (ADC) measures the differential voltage which applies to its input pins VIN- and VIN+. We access the ADC also over the I²C bus. Its address is 4AH. Like the RTC first you have to open the device and set it to slave.

```
ioctl(fd_ads1110, I2C_SLAVE_FORCE, 0x4A)
```

Referring to the data sheet, you have to read three bytes and you can write one byte to it.

The first two bytes contain the data and the last the settings. The settings mainly consist of the baud rate and the gain. The configuration for continuous or single mode is set once in the beginning of the application.

```
int config_reg[4][2]={{2048,1}, {8192,2}, {16384,4},{32768,8}};
```

To identify the gain or the baud rate, the array above is used. The first part of the array is the mincode and the second part is the gain.

First we mask the gain bits and read out the gain with help of the array above. Secondly we mask the baud rate bits, shift them twice left and use also the array above to get the baud rate value.

Below you can see a little example how to mask the right bits.

```
gain     = configure_reg[buf[2]&0x03][0];
baudrate = configure_reg[(buf[2]&0x0C>>2)][1];
```

4.3.5 The VFD module

If you want to use the VFD, you have to load the module vfd_scb9520.ko.

To access the display you can use the usually used calls like *open*, *close*, *read*, *write* und *ioctl*. The *ioctl* command offers several functions, see table 6.

command	function
VFD_GET_INFO	return the manufacturer and name of the display
VFD_OFF	turn display off
VFD_SET_CURSOR_LINE1_POS1	set cursor to line 1 position 1
VFD_SET_CURSOR_LINE2_POS1	set cursor to line 2 position 1
VFD_CLEAR_DISPLAY	clear the display
VFD_GO_XY	set cursor to position X Y
VFD_CLEAR_LINE_1	clear line 1
VFD_CLEAR_LINE_2	clear line 2

Table 6: ioctl commands for the display.

Almost every command gets a NULL as third parameter. Just the commands VFD_GO_XY and VFD_GET_INFO get an address to copy some data from or to kernel space.

4.3.6 Communication with TCP sockets

During the application *evb9520* is running, one socket is open and searches for an incoming connection from another application like *DeviLANControl*. If a connection could be set up, the routine *ServerConnect()* starts and opens a server socket to look for the port in the file /etc/network/interfaces. If there is no port entry, it uses the default port 3333 and writes this one to the file interfaces. The server socket is started with the option SO_REUSEADDR, so the port can be used immediately after disconnecting. Usually, a used port is locked for about two minutes after disconnecting, but in this case, this security feature is not needed.

TCP connections don't loose data like UDP connections. But if the data can be divided into two or more pieces, this depends on traffic and frame size. Because of this, the routine *ServerGetMsg()* reads until an end of line (EOL, 0DH) could be found in the string. If one could be found, the function *DataHandler()* is called and searches for special keywords in the string.

All keywords are listed in table 7.

Table 7: keywords for socket communication.

keyword	parameter	description
get_config	none	sends the name of the module, the state of the LEDs and switches, the time, the gain and the baud rate and the analogue value from the ADC
data_renew	none	sends the current state of the switches, ADC and the time
porta	leds	sets the LEDs to the value of the variable leds
adc_baudrate	ads1110_sps	set the baud rate
adc_gain	ads1110_gain	sets the gain
time	time (s.u.)	sets the time
mode	menu	set the menu

The value for the variable leds must be between 0 and 255 and it has to be sent in hexadecimal. The baud rate can be 15, 30, 60 or 240, the gain must be 1, 2, 4 or 8. The variable menu has a range from 0 to 9, but just 0 to 3 has a function and the rest is free for use. The string to set the time must look like this: 23:12:59 Tue, 31.12.2099

Please beware of case sensitivity.

Here are some examples of how to use the keywords.

```
get_configure
data_renew
adc_baudrate 240
adc_gain 2
time 23:12:59 Tue, 31.12.2099
mode 1
```

4.4 The rc script

At boot up the script S99scb9520 is executed. You can find this script in the directory /etc/init.d. The rules for the names are quite simple. The name must start with a capital 's', then there must be a number and then a random string may follow. The number decides the order in which the scripts will be executed. Our script got number 99, so we can be sure, that it is the last one to be executed.

If you run the script with the parameter 'start', it loads the modules *8250*, *i2c_pxa, i2c_dev, vfd_scb9520* and *io_scb9520*. It creates the device characters and starts the applications *udp_config* and *evb9520* in the background. To stop the applications, use the following commands.

```
killall udp_config
killall evb9520
```

```
#!/bin/sh
if [ "$VERBOSE" != no ]
then
  echo -n "Initializing EVB9520... "
  echo ""
fi
case "$1" in
  start|"starting all modules and apps")
  if [ `ls -l /dev/io0 2>/dev/null|wc -l` -eq 0 ]
  then
    echo /dev/io0 missing, starting io.sh to create it
    /root/io.sh
  else
    echo /dev/io0 already exists, so nothing has to be done
  fi
  if [ `ls -l /dev/vfd0 2>/dev/null|wc -l` -eq 0 ]
  then
    echo /dev/vfd0 missing, starting vfd.sh to create it
    /root/vfd.sh
  else
    echo /dev/vfd0 already exists, so nothing has to be done
  fi
  if [ `ls -l /dev/ttyS0 2>/dev/null|wc -l` -eq 0 ]
  then
    echo /dev/ttyS0 missing, starting serial.sh to create it
    /root/serial.sh
  else
    echo /dev/ttyS0 already exists, so nothing has to be done
  fi
  modprobe i2c-dev
  modprobe i2c-pxa
  modprobe 8250
```

```
  insmod /root/modules/io_scb9520.ko 2>/dev/null
  insmod /root/modules/vfd_scb9520.ko 2>/dev/null
  /usr/local/bin/udp_configure &
  /usr/local/bin/evb9520    &
  ;;
  stop)
  echo "stopping all modules"
  killall evb9520    2>/dev/null
  killall udp_configure 2>/dev/null
  rmmod i2c-dev 2>/dev/null
  rmmod i2c-pxa 2>/dev/null
  rmmod /root/modules/io_scb9520.ko 2>/dev/null
  rmmod /root/modules/vfd_scb9520.ko 2>/dev/null
  echo "removing all files"
  rm /dev/io0 2>/dev/null
  rm /dev/vfd0 2>/dev/null
  ;;
  *)
  echo "Usage:S99scb9520 {start|stop}" >&2
  exit 1
  ;;
esac
```

Listing 11: The rc-script

The script in listing 11 first checks, if the files /dev/io0, /dev/vfd0 and /dev/ttyS0 exist and, if not, creates them.

With the parameter 'stop' it unloads all modules with the help of *rmmod* and stops the applications with the command *killall*. *killall* quits applications depending on their names and not on their PID (process identifier).

To quit applications with *killall* use this command like this:

```
killall evb9520
killall udp_configure
```

To create the files, there are several little shell scripts to execute. For creating the io0 file execute io.sh and so on. All these files are quite similar, they just create the file with the command *mknod*. Look one line below for a simple example.

```
mknod /dev/vfd0 c $IO_MAJOR $IO_MINOR
```