Getting started with GumstixNXT and OpenEmbedded

David Brandt and Ulrik Pagh Schultz
{david.brandt,ups}@mmmi.sdu.dk
USD's Modular Robotics Lab
University of Southern Denmark
http://modular.mmmi.sdu.dk

September 10, 2012

Version and changes

Revision 2.0.2 (September 10, 2012): cleaned up for new iteration of the course, enough checked and verified

1 Introduction

This document contains a very brief introduction to using embedded Linux on the gumstix overo boards. The document is based on various online sources, but mainly on information also available at www.gumstix.org. The hope is that this document will provide enough information to enable the reader to compile Linux for an Overo board and get it to boot, in this document only booting from a micro SD card will be discussed, however many more booting options are available.

This guide is assuming that the development host is running Ubuntu 10.04.04 LTS (all updates installed, see Appendix C for a quick guide on getting Linux up and running). VMWare or VirtualBox can is some cases be used (see Appendix D for some notes on using a virtualized host). We will also assume that the reader has access to the serial console port on the overo board, for instance by using GumstixNXT, the Tobi expansion board or similar.

2 Booting Linux

First a short introduction to what is needed in order to boot Linux on an overo board.

In order to get Linux fully running a total of four elements are needed:

x-load The first level bootloader, it is very small and simple and its purpose is to load the bigger and more advanced second level bootloader.

u-boot The second level bootloader, it is configurable to load many different kinds of operating systems in many different ways including network booting etc.

uImage The actual Linux kernel, including drivers compiled into the kernel, but not loadable modules or programs.

rootfs The entire file system mounted at boot, including modules, libraries, home folders, utility programs etc.

So in order to get Linux up and running we need to build these four elements and transfer them to a SD card in a way that makes it possible to boot from that card. Let us first consider the tools needed.

3 Installing OpenEmbedded

OpenEmbedded (OE) is a build system used for the overo boards to create Linux distributions. OE is used to build all the parts we need for our Linux distribution including cross compilers, the kernel, rootfs etc. First we will take a look at how to setup OE.[1]

The heart of OE is the BitBake program and a collection of recipes used by bitbake. The recipes contains information on how to compile and build a given package, along with dependencies and URLs for where to get the source code. It is used to create both the cross compiler, the bootloaders, the kernel and the root file system.

In order for OE to work on Ubuntu 10.04 LTS you need to install a number of packages, this can be done simply by using apt-get. The needed packages are:

> git-core, subversion, gcc, patch, help2man, diffstat, texi2html, texinfo, libncurses5-dev, cvs, gawk, python-dev, python-pysqlite2, unzip, chrpath, ccache, g++

The command to do this would e.g. be the following:

```
$ sudo apt-get install git-core subversion gcc patch help2man diffstat texi2html texinfo
libncurses5-dev cvs gawk python-dev python-pysqlite2 unzip chrpath ccache g++
```

Although this does not appear to be the case for Ubuntu 10.04, be aware that depending on your version of Ubuntu, it is possible that /bin/sh is linked to /bin/dash. If this is the case you need to change it to be linked to /bin/bash. This can be done by executing the following command and answering 'no' when asked whether to install dash as /bin/bash:

```
$ sudo dpkg-reconfigure dash
```

After installing the needed packages we need to get hold of the OE source files. Be aware that some of the following steps can generate significant amounts of data traffic. Further, when compiling complete distributions a lot of computing time and hard drive space will be needed. It is recommended that you have at least 10GB of free space if you want to compile distributions containing many utility programs etc.[2]

As a first step we will create a directory for OE which will be were we locate all the files needed and generated while we work, through out this document we will assume that this directory is located at the root of your home folder and is named 'overo-oe', however this is by no means mandatory.

```
$ mkdir -p ~/overo-oe
$ cd ~/overo-oe
```

(Note: if you copy-paste these commands, make sure that the "~" character is correctly transferred.) Now we will install the OE data and check out the overo Linux branch (there might be some warnings from the git commands, you should just ignore those):

```
$ git clone git://gitorious.org/gumstix-oe/mainline.git org.openembedded.dev
$ cd org.openembedded.dev
$ git checkout --track -b overo-2011.03 origin/overo-2011.03
```

[1] The instructions here are based on `http://gumstix.org/software-development/open-embedded/61-using-the-open-embedded-build-system.html`

[2] Make sure to use a native Linux file system (not networked or mounted via virtualization) and do not move the installation around after you have completed building.

Next we install BitBake:

```
$ cd ~/overo-oe
$ git clone git://git.openembedded.org/bitbake bitbake
$ cd bitbake
$ git checkout 1.12.0
```

Now we will copy the gumstix profile script files to our build directory:

```
$ cd ~/overo-oe
$ cp -r org.openembedded.dev/contrib/gumstix/build .
```

In order to setup the environment variables for the build system we need to modify the bash profile, this can be done by the following command (you might want to backup .bashrc before issuing it, and if you did not install Open Embedded in the suggested directory you will have to edit .bashrc afterwards):

```
$ cat ~/overo-oe/build/profile >> ~/.bashrc
```

In order for the changes to take effect you need to close your terminal window and open a new one. (Alternatively, you can load the profile file into your current environment using the command `$ source build/profile`)

4 Building Linux

Now we are done installing OE and are getting ready to build our Linux image. In order to build our own custom distribution we need to create a *recipe* which tells bitbake what to include in our distribution. First we create the directory which should contain our custom recipes:

```
$ cd ~/overo-oe
$ mkdir -p user.collection/recipes/images
```

In the directory we just created you should put all recipes you make yourself, you are not supposed to modify the recipes provided as part of the OE installation. Now, in the new directory create a file named *myimage.bb* containing the following:

```
IMAGE_PREPROCESS_COMMAND = "create_etc_timestamp"

IMAGE_INSTALL = "task-boot \
      util-linux-ng-mount util-linux-ng-umount \
      angstrom-version \
      libertas-sd-firmware \
      sshd \
      wireless-tools \
      rt73-firmware \
      zd1211-firmware \
      dhcp-client \
      openssh-misc \
      openssh-scp \
      openssh-ssh \
      wpa-supplicant \
      "

export IMAGE_BASENAME = "minimalist-image"

inherit image
```

This bitbake recipe will build a fairly small, but not minimal, Linux distribution. The distribution includes kernel modules for interfacing to the wireless network, and software packages for various network-related functionalities. In order to build the distribution use the following command:

```
$ cd ~/overo-oe
$ bitbake myimage
```

You should be aware that this command will potentially take a long time (on the order of hours, depending on available processing power and network bandwidth). Briefly stated this small script instructs bitbake to build a small root file system for our Linux distribution, however, since bitbake takes care of all dependencies for us, this might involve compiling and building the cross compiler to compile the programs in the root file system and the Linux kernel itself plus actually compiling the Linux kernel, the needed kernel modules and utility programs. Further, the source code for many of these steps might not be available locally so they have to be downloaded before compiling them. The process will create a directory named *~/overo-oe/tmp/* do not delete or modify the contents of this directory, this is where the temporary files generated during the build and the actual result will be stored. By keeping the temporary files repeated builds will be much faster.

The process might at times appear to have stopped, but can at times be working for an hour before output is displayed. If you want to stop it, note that a single Control-C will stop the current task but not the whole build process, you need to press Control-C twice to stop it. In general, there is a fair amount of error recovery, and it is capable of e.g. trying alternatives for download of software. For example, currently the following error message is generated at some point:

```
Connecting to www.angstrom-distribution.org|188.40.83.200|:80... connected.
HTTP request sent, awaiting response... 404 Not Found
2012-09-07 12:09:21 ERROR 404: Not Found.

Cloning into /mnt/oe/overo-oe/sources/git/www.sakoman.com.git.linux-omap-2.6.git...
```

Here, after failing the initial HTTP request, the process continues using a different (albeit slower to obtain) source, and will normally proceed to the next task eventually.

Once the build is completed you can find the generated files in the *~/overo-oe/tmp/deploy/glibc/images/overo* at this point it should contain the u-boot boot loader (*u-boot-overo.bin*), the kernel image (*uImage-overo.bin*) and the root file system (*minimalist-image-overo.tar.bz2*) along with some other files. So we still need the first level boot loader. In order to generate this use the command:

```
$ cd ~/overo-oe
$ bitbake x-load
```

This will create the file *MLO-overo* in the *~/overo-oe/tmp/deploy/glibc/images/overo* directory.

5 Booting from an SD Card

5.1 Installing Linux on the board

The gumstix boards comes with Linux preinstalled, however, we recommend using the SD card slot for booting the boards. The reason for this is that if the contents of the onboard flash memory gets corrupted due to an error during development it can be difficult to reflash, however, an SD Card can easily be replaced. This means that if you power on the board without the SD card inserted, it will boot using a standard Linux which will let you run various commands, but does not provide dedicated drivers for the board.

The gumstix overo boards are able to boot directly from a micro SD or SDHC card. The following instructions will guide you how to make an SD card bootable an how to install your bootloaders and Linux distribution on it.

In order to create the needed partition layout we will use *fdisk* an during the guide I will assume that the SD card is referred to by the device name */dev/sde*, this might be different on your machine. (This guide assumes an empty partition table, if your card is already partitioned you can delete partitions using fdisk: unmount the partitions, start fdisk as explained in the text, and use the "d" and "w" commands to first delete the partitions and then write the result to the disk.)

First we need to unmount the card in order to modify it using fdisk:

```
$ sudo umount /dev/sde1
```

Next we start fdisk, and start building a new partition table:

```
$ sudo fdisk /dev/sde
Command (m for help):  o
Building a new DOS disklabel.  Changes will remain in memory only, until you decide
to write them.  After that, of course, the previous content won't be recoverable.
Warning:  invalid flag 0x0000 of partition table 4 will be corrected by w(rite)
```

First we need information about the size of the card we are using:

```
Command (m for help):  p
Disk /dev/sde:  2032 MB, 2032664576 bytes
64 heads, 63 sectors/track, 984 cylinders
Units = cylinders of 4032 * 512 = 2064384 bytes
Disk identifier:  0x00aa8e5c
Device Boot Start End Blocks Id System
```

This particular example is for a 2GB card. In order to boot properly we need the to have a card formatted with 255 heads, 63 sectors and as many cylinders as we can fit. To calculate the number of cylinders, take the number of bytes reported above by fdisk, divide by 255 heads, 63 sectors and 512 bytes per sector. In our example: 2032664576 / 255 / 63 / 512 = 247.12 which we round **down** to 247 cylinders.

In order to make these settings in fdisk we enter expert mode and return to normal mode after we are done:

```
Command (m for help):  x
Expert command (m for help):  h
Number of heads (1-256, default 4):  255
Expert command (m for help):  s
Number of sectors (1-63, default 62):  63
Warning:  setting sector offset for DOS compatibility
Expert command (m for help):  c
Number of cylinders (1-1048576, default 984):  247
Expert command (m for help):  r
```

The boot loaders and the kernel image must be placed in the first partition on the card, which must be a FAT partition. Next step is to create a 32MB FAT partition and mark it as bootable:

```
Command (m for help):  n
Command action
e extended
p primary partition (1-4)
```

```
p
Partition number (1-4):  1
First cylinder (1-247, default 1):  1
Last cylinder or +size or +sizeM or +sizeK (1-247, default 15):  +32M
Command (m for help):  t
Selected partition 1
Hex code (type L to list codes):  c
Changed system type of partition 1 to c (W95 FAT32 (LBA))
Command (m for help):  a
Partition number (1-4):  1
```

Next we need to create an ext3 partition on the rest of the card for the root file system and write our changes to the card:

```
Command (m for help):  n
Command action
e extended
p primary partition (1-4)
p
Partition number (1-4):  2
First cylinder (6-247, default 6):  6
Last cylinder or +size or +sizeM or +sizeK (6-247, default 247):  247
Command (m for help):  w
The partition table has been altered!
Calling ioctl() to re-read partition table.
WARNING: If you have created or modified any DOS 6.x partitions, please see the
fdisk manual page for additional information.
Syncing disks
```

Next we need to format the two new partitions. First the FAT partition and then the ext3 partition.

```
$ sudo mkfs.vfat -F 32 /dev/sde1 -n FAT
$ sudo mkfs.ext3 /dev/sde2
```

Now the card is ready to be populated with our newly build Linux distribution. First we mount the FAT partition and copy the boot loaders and the kernel image to it. Note that the first level boot loader **must** be written first. Also note that while we copy the files we also rename them such that the have the names expected by the boot loaders.

```
$ sudo mount /dev/sde1 /media/card
$ cd ~/overo-oe/tmp/deploy/glibc/images/overo
$ sudo cp MLO-overo /media/card/MLO
$ sudo cp u-boot-overo.bin /media/card/u-boot.bin
$ sudo cp uImage-overo.bin /media/card/uImage
$ sudo umount /dev/sde1
```

Now the card contains a bootable Linux kernel, however without a root file system it is not of much use. The final steps installs the root file system in the ext3 partition.

```
$ sudo mount /dev/sde2 /media/card
$ cd /media/card
$ sudo tar xvaf ~/overo-oe/tmp/deploy/glibc/images/overo/minimalist-image-overo.tar.bz2
```

```
$ cd
$ sudo umount /dev/sde2
```

Now you can insert the SD card into the card reader on the gumstix module and power it on, and hopefully it should boot. When booting you might see some error messages such as: *uncorrectable error: end_request: I/O error, dev mtdblock0, sector 0*. These can be ignored.

5.2 Connecting to the board

The board connects to your computer via the USB cable, communication is done using a terminal program, this enables you to run a shell directly on the board with input/output taking place via your computer.

The recommended programs for communicating with the board are minicom and putty. They should be setup for no flow control, 115200 baud 8N1. You can connect via /dev/ttyUSBn, where n is the number assigned by Linux whenever you connect the device (this might change from time to time). Using the terminal program, you can see the messages at boot time, login as "root" and execute commands interactively.

In more detail, the program "minicom" can be used as follows. On a stock Ubuntu you first need to install it (using the software center or apt-get, type "minicom" on the command line for instructions on how to install it). To get the manual for minicom, use "man minicom", this instructs you to first run "minicom -s" to do the configuration, and since you are changing the system-wide defaults you should use "sudo minicom -s" after which you can simply run it by using "minicom". To avoid changing the configuration every time the USB port changes, create multiple configuration files (all however using sudo).

6 Working with the device

We are now ready to start interfacing to the hardware on the device. For this to work, the hardware must be configured correctly at boot-time and there must not be any conflicting use of hardware resources. We use GPIO to interface to the hardware; GPIO can be accessed from user-level using a generic interface and using a dedicated kernel module. We use the generic interface for initial experiments (this section) and the dedicated kernel module to implement the driver (next section).

6.1 Configuring u-boot port setup to enable LED access

To understand how to control the LEDs, we must consult the GumstixNXT schematics (currently available under Blackboard). The LEDs are connected to a set of GPIO pins connected to the CPU via a bidirectional level shifter, named "U2" on page 12 of the revision 1.0 schematics. An excerpt of the relevant part of the diagram is shown in Figure 1, the LEDs are connected on IO1...IO9.

To output on the GPIOs connected to the level shifter, GPIO80 (direction) must be set to 1 (meaning output), GPIO 67 (output enable) must be set to 0 (it is inverted, this means that it is activated), and then GPIO68 (and a number of other GPIOs) can be used to toggle the LED state of the LEDs (writing a zero to the GPIO sets it high, turning on the LED).

To be able to use these GPIOs, the corresponding CPU pins must be configured for GPIO usage. The pins are multiplexed to many different functionalities (modes), the mode must be set correctly on each of the pins that we need to enable GPIO. To find out what pins need to be configured and how they must be configured, we need to consult the datasheet for the CPU "OMAP35x Applications Processor Technical Reference Manual" (available on Blackboard). For example, to find out how to enable GPIO67, we turn to page 772 which contains a part of "Table 7-4. Core Control Module Pad Configuration Register Fields" where we can see that GPIO67 is available on "CONTROL_PADCONF_DSS_PCLK[31:16]" which is configured to "dss_hsync" in Mode 0 must be configured to Mode 4 to enable "gpio_67".

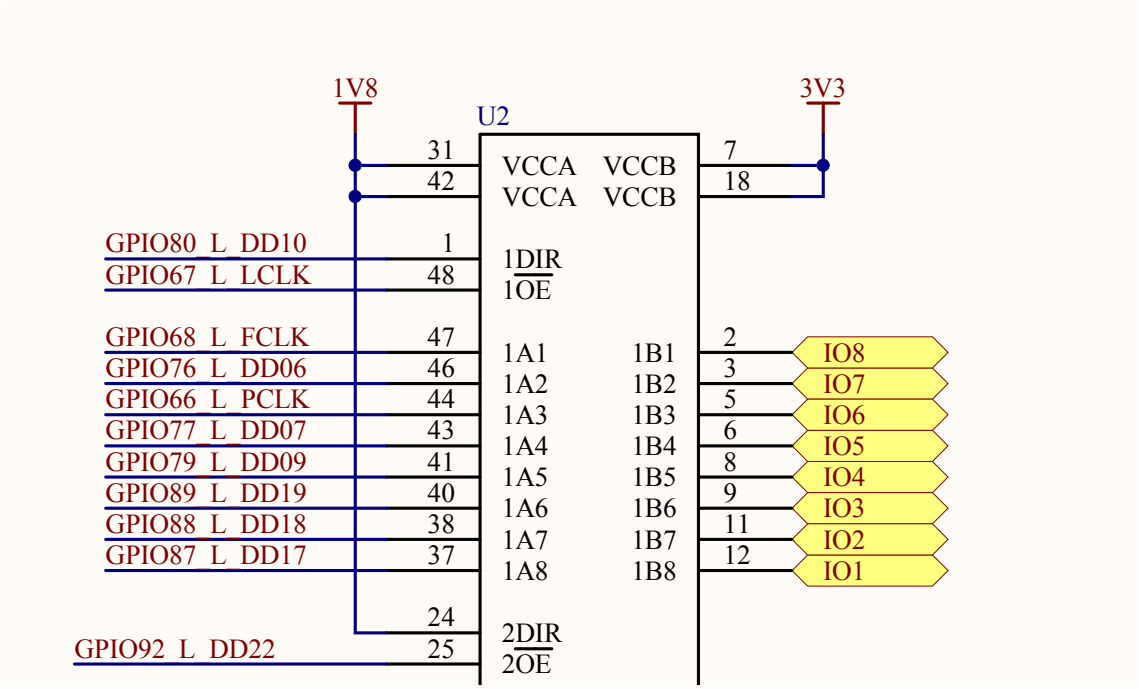

Figure 1: Excerpt from the GumstixNXT schematics, level-shifter controlling LEDs

Following the "Gumstix U-Boot development" guide on jumpnowtek.com, we change the pin multiplexing by modifying the overo.h file, e.g., the line:

```
MUX_VAL(CP(DSS_HSYNC),(IDIS | PTD | DIS | M0)) /*DSS_HSYNC*/\
```

is changed to

```
MUX_VAL(CP(DSS_HSYNC),(IEN  | PTU | DIS | M4)) /*GPIO_67*/\
```

Since GPIO67 is accessed using register CONTROL_...HSYNC in mode 4. All details of how to do this are described on jumpnowtek.com, the recommended process is to modify the file, build a patch, add this patch to the recipe, and rebuild. (Note: the description on the web page contains a few minor errors in the paths.) The same changes must be made for all of the GPIO pins that are needed (reading the schematics in Figure 1, these are GPIOs 80, 67, 68, 76, 66, 77, 79, 89, 88, and 87).

After rebuilding and redeploying, the appropriate ports are in GPIO and can be used for blinking the LEDs, for example by directly accessing the GPIO pins as described next.

If you apply the changes incrementally, for example creating one patch file for the first LED, another for the remaining LEDs, and yet another for the motors (as described later), the end result can be hard to understand and you can end up having conflicts between the patches. (In principle it should work fine, in practice it can cause trouble.)

6.2 Direct hardware access using GPIO

The kernel GPIO interface can be accessed directly from the shell:

```
root@overo:/root# cd /sys/class/gpio
```

```
root@overo:/sys/class/gpio# echo 80 > export
root@overo:/sys/class/gpio# echo 67 > export
root@overo:/sys/class/gpio# echo 68 > export
root@overo:/sys/class/gpio# echo out > gpio80/direction
root@overo:/sys/class/gpio# echo out > gpio67/direction
root@overo:/sys/class/gpio# echo out > gpio68/direction
root@overo:/sys/class/gpio# echo 1 > gpio80/value
root@overo:/sys/class/gpio# echo 0 > gpio67/value
```

These lines first export the appropriate GPIO pins (making the accessible through /sys/class/gpio), set their direction to out, and set GPIO80 and GPIO67 to the appropriate values. Depending on how else your environment is set up, one or more of the LEDs should now light up.

To blink an LED using GPIO68, use e.g.

```
echo 0 > /sys/class/gpio/gpio68/value
```

To turn it off, and to turn it back on again:

```
echo 1 > /sys/class/gpio/gpio68/value
```

Next step is to make a blinking LED program, which for example can be done by creating the following script:

```
#!/bin/sh
for i in 0 1 2 3 4 5 6 7 8 9
do
  echo "Blink cycle $i"
  echo 0 > /sys/class/gpio/gpio68/value
  sleep 1
  echo 1 > /sys/class/gpio/gpio68/value
  sleep 1
done
```

If you do not have a text editor, you can use

```
cat > /root/blink.sh
  (...type the script, end with an empty line, press control-D...)
chmod 755 /root/blink.sh
```

After which running /root/blink.sh will blink the LED 10 times.

6.3 Adding utilities and helper scripts

Next time you format the flash card and deploy the system the helper scripts you just create will be removed. It would be useful to have an easy way of putting these onto the system, as you are working. In addition, using "cat" as a text editor obviously will not get you very far if you need to edit something directly on the board.

We provide a Makefile for automating the task for formatting the SD card, deploying the system, and easily adding extra files. It is available in the "src/home" directory[3] The makefile provides targets for format, building kernel modules (see later), exporting all files stored in the "export" subdirectory, and deploying everything to the SD card.

The default build includes the "vi" text editor. If this is not your cup of tea, there are numerous alternatives, such as the "nano" text editor that has fairly user-friendly keybindings. To add this editor to your image, edit "myimage.bb" (from above) adding the line

[3] Using the current repository organization, the will probably change.

```
nano \
```

to the "IMAGE_INSTALL" variable. Rebuilding will then download, compile and include the nano editor, which will be available after redeployment to the SD card.

7 Kernel Module Development

7.1 Cross-compilation environment

To compile kernel modules, you must set up the shell environment to enable access to the cross-compilers and other tools included with Open Embedded. The process is explained on jumpnowtek.com[4], and basically consists of sourcing (reading into the shell) the file kernel-dev.txt, which can be done by the command

```
. kernel-dev.txt
```

A slightly customized version of the file is available from GumstixNXT github.

7.2 LEDDEV 1-bit kernel module driver

We will now describe the development of a driver for a single-bit LED; this driver is extended in the next subsection to cover all 8 LEDs. The complete source code of the driver can be found in Appendix B.

The basic functionality can be divided into three parts: setup (loading), response to user requests, and unloading. The setup is by far the most complicated part: it starts from the leddev_init function (which is the module initialization function), first sets up the character device, then the supplemental class device, and last initializes the hardware.

Setting up the character device is done in `leddev_init_cdev`. First, the call to the `MKDEV` macro initializes the device structure, after which the major and minor numbers are dynamically assigned using `alloc_chrdev_region`. The specific file operations (the write function) are added to the device using the `cdev_init` function, and last the device is made available to user-space with the call to `cdev_add`.

Setting up the character device is not covered in detail, but follows a standard pattern that will work for any driver (that wishes to export a class interface under /sys/class). Setting up the hardware basically consists of requesting the required GPIOs and configuring the directionality, but is complicated by the need to release them again if something fails.

Response to user requests is implemented by the `leddev_write` function. This function examines the user input (found in the character buffer `buff`) character-by-character to decide what actions to take. Actions are implemented by using the appropriate GPIOs and updating the value representing the LED pattern. Care must be takes to return a value corresponding to the number of bytes read.

Last, unloading is fairly straightforward, and is concerned with unregistering the device from the kernel and freeing the hardware resources.

7.3 LEDDEV 8-bit kernel module driver

The 8-bit kernel driver is a simple generalization of the 1-bit driver. The main difference is that the device now — for generality — handles a set of GPIOs stored in an array. Moreover, parsing the user command is a bit more intricate but not significantly different.

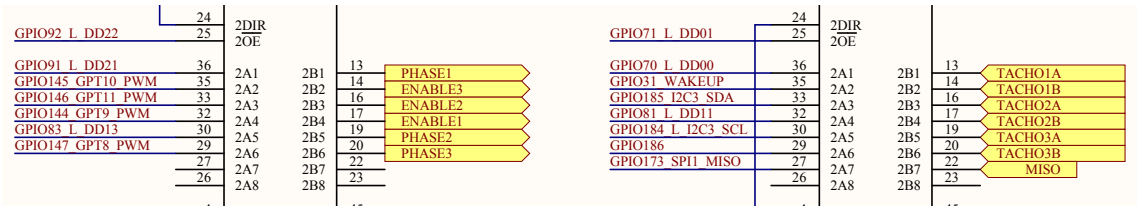

Figure 2: Excerpt from the GumstixNXT schematics, level-shifters for controlling motors

7.4 Developing a motor driver

Each motor has two control signals (PHASEx and ENABLEx) and two tachometer signals (TACHOxA and TACHOxB), where x is the number of the motor. Figure 2 shows the relevant part of the GumstixNXT schematics. Basic motor control is simple: the motor is turned on and off using the ENABLE signal and the direction is controlled using the PHASE signal. The tachometer signals give pulses when the motor moves, so motor speed is equal to number of pulses per time unit. The direction can be read from the phase difference between TACHOxA and TACHOxB.

The simplest control of a motor is thus to set ENABLE to 1 and PHASE to the value indicating direction, ignoring TACHO: this will make the motor rotate full-speed in the given direction. To actuate the motor at less than maximum speed, you must use a PWM signal on ENABLE where the duty-cycle indicates how much torque the motor must yield. More advanced control requires the TACHO signals to be taken into account.

Controlling the motors can only be done once the appropriate multiplexing has been done in u-boot to give access to the appropriate pins as GPIOs. Moreover, the level-shifters have to be set up correctly (PHASE and ENABLE as output, TACHO as input). Pull-up/pull-down in the u-boot configuration can be used to ensure that the motor does not start moving before the driver is ready. Note that one of the TACHO signals is read from GPIO 31, which according to the technical reference manual is bound to a register named "CONTROL_PADCONF_JTAG_EMU0[31:16]", this register however cannot be found in the usual overo.h file. The register definition line can be added manually, to find the correct name use the find command, e.g.:

```
$ cd $UBOOTDIR
$ find . -name '*.h' -exec grep "#define CONTROL_PADCONF_JTAG" {} \; -print
```

(This assumes you have defined UBOOTDIR like on jumpnowtek.com.) This reveals the name CONTROL_PADCONF_JTAG_EMU1 defined in ./git/arch/arm/include/asm/arch-omap3/mux.h so the following line should be added to overo.h:

```
MUX_VAL(CP(JTAG_EMU1),(IEN  | PTU | EN | M4)) /*GPIO_31*/\
```

(The macro "CP" prepends CONTROL_PADCONF to the name.)

An additional complication which must be addressed is that in the default open embedded setup, an LCD driver is using the same CPU pins, which makes it impossible to control the motors. To resolve this issue, the LCD driver must be disabled, which is done by configuring the kernel. Follow the "Gumstix kernel dev" guide on jumpnowtek.com, except that with the default bitbake setting the configuration menu needs to be accessed using the "screen" program, which it a bit cumbersome. To fix this, edit

```
~/overo-oe/org.openembedded.dev/conf/bitbake.conf
```

[4]See for example the article `http://www.jumpnowtek.com/index.php?option=com_content&view=article&id=46&Itemid=54`

by replacing the two lines

```
# Set a default
TERMCMD ?= "${SCREEN_TERMCMD}"
TERMCMDRUN ?= "${SCREEN_TERMCMDRUN}"
```

with the following two lines (that instruct it to open a regular terminal window instead):

```
# Set a default
TERMCMD ?= "${GNOME_TERMCMD}"
TERMCMDRUN ?= "${GNOME_TERMCMDRUN}"
```

Once this is done, configuring the kernel basically amounts to running the command

```
$ bitbake -c menuconfig virtual/kernel
```

This gives you a textual menu for configuring the kernel.

Choose "Device drivers", "Graphics support", and disable everything except "Support for framebuffer devices". After this you need to copy the resulting configuration file and run bitbake again, e.g. (depending on your kernel version):

```
cp tmp/work/overo-angstrom-linux-gnueabi/linux-omap3-3.2-r103/git/.config org.openembedded.dev/r
bitbake -c clean virtual/kernel; bitbake virtual/kernel
bitbake myimage
```

It is recommended that you as a start just configure the kernel and setup u-boot for access to the PHASE and ENABLE signals. You can now test the motors at maximum speed through the kernel user-level /sys/class/gpio interface. The GumstixNXT "apps" directory contains a number of examples that control the motors using this approach, e.g. the remote control car. Once this is confirmed to work, you can implement a PWM control following the guide on jumpnowtek.com (the hardware directly supports PWM control).

7.5 Developing a motor controller

A motor controller requires use of the TACHO signals to read the position of the motor and adjust the PWM control accordingly. Apart from the control algorithm, the main difficulty is that this should happen at regular intervals, meaning it should be run by a timer interrupt. This will be covered in a future version of the guide.

7.6 Developing a sensor controller

...

A Common error messages

A.1 Error messages during boot

`end_request: I/O error, dev mtdblock0, sector...`

This message can be ignored, it apparently has to do with a slight incompatibility between the way the boot partition has been created and what Linux expects.[5] Warning: apparently some people on the internet have rendered their device incapable of booting by trying to fix this problem, so be careful.

B LEDDEV 1-bit kernel module device driver

```
/*

A GumstixNXT LED driver for a single LED using GPIO, loosely based on irqlat by Scott
Ellis

Sample useage (shell commands):
insmod leddev.ko
echo set > /dev/leddev
echo inv > /dev/leddev
echo clr > /dev/leddev

*/

#include <linux/init.h>
#include <linux/module.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <asm/uaccess.h>
#include <linux/string.h>
#include <linux/kernel.h>
#include <linux/device.h>
#include <mach/gpio.h>
#include <linux/interrupt.h>
#include <linux/workqueue.h>


/* GPIOs for controlling the level shifter behavior */
#define GPIO_OE 80
#define GPIO_DIR 67

/* GPIOs for controlling the individual bits */
#define GPIO_BIT0 68

/* Signals to send to GPIOs to turn LEDs on and off */
#define BIT_ON 0
#define BIT_OFF 1
```

[5] Google search for the error message, see for example `http://old.nabble.com/What-is-mtdblock0-on-the-overo-earth-linux-omap3-2.6.29-build-td23179672.html`.

```
/* leddev device structure */
struct leddev_dev {
  /* Standard fields */
  dev_t devt; // kernel data structure for device numbers (major,minor)
  struct cdev cdev; // kernel data structure for character device
  struct class *class; // kernel data structure for device driver class /sys/class/leddev [LDD Chapte
  /* Driver-specific fields */
  int value; // the integer value currently shown, 1 if on, 0 if off
};

/* device structure instance */
static struct leddev_dev leddev_dev;

/* Reset hardware settings and driver state */
static void leddev_reset(void)
{
  // Reset level shifter control
  gpio_set_value(GPIO_OE, 1);
  gpio_set_value(GPIO_DIR, 0);
  // Reset the bit to off state
  gpio_set_value(GPIO_BIT0, BIT_OFF);
  // Reset driver state
  leddev_dev.value = 0;
}

/* Respond to a command that has been written to the device special file */
static ssize_t leddev_write(struct file *filp, const char __user *buff, size_t count, loff_t *f_pos)
{
  char cmd[3]; // Buffer for command written by user
  ssize_t status = 0; // Return status, updated depending on input

  /* Copy the data the user wrote from userspace (how much is read depends on return value) */
  if (copy_from_user(cmd, buff, 3)) {
    printk(KERN_ALERT "Error copy_from_user\n");
    status = -EFAULT;
    goto leddev_write_done;
  }

  /*
    Process the command stored in the data
    'set' means set the bit
    'clr' means clear the bit
    'inv' means invert
  */
  if (cmd[0] == 's' && cmd[1] == 'e' && cmd[2] == 't') { // Set command?
    gpio_set_value(GPIO_BIT0, BIT_ON); // Turn the hardware bit on
    leddev_dev.value = 1; // Store the software bit
    status = 3; // Read 3 bytes, update return status correspondingly
  }
```

```
  else if (cmd[0] == 'c' && cmd[1] == 'l' && cmd[2] == 'r') { // Clear command?
    gpio_set_value(GPIO_BIT0, BIT_OFF); // Turn the selected hardware bit off
    leddev_dev.value = 0; // Turn the software bit off
    status = 3; // Read 3 bytes, update return status correspondingly
  }

  else if (cmd[0] == 'i' && cmd[1] == 'n' && cmd[2] == 'v') { // Invert command?
    leddev_dev.value = !leddev_dev.value; // Invert bit pattern
    gpio_set_value(GPIO_BIT0, leddev_dev.value ? BIT_ON : BIT_OFF);
    status = 3; // Read 3 bytes, update return status correspondingly
  }

  else { // Unrecognized command
    status = 1; // Read one byte, will be called again if there is more input
    if(cmd[0]!=10 && cmd[0]!=' ') // Ignore newline and space
      printk(KERN_ALERT "LED illegal command char %d\n", cmd[0]);
  }

 leddev_write_done:

  return status; // Negative=error, positive=success and indicates #bytes read
}

/* The file operations structure, operations not listed here are illegal */
static const struct file_operations leddev_fops = {
  .owner = THIS_MODULE,
  .write = leddev_write,
};

/* Initialize the character device structure */
static int __init leddev_init_cdev(void)
{
  int error;

  /* Initialize devt structure, major/minor numbers assigned afterwards */
  leddev_dev.devt = MKDEV(0, 0);

  /* Assign major and minor device numbers to devt structure */
  error = alloc_chrdev_region(&leddev_dev.devt, 0, 1, "leddev");
  if (error < 0) {
    printk(KERN_ALERT "alloc_chrdev_region() failed: error = %d \n", error);
    return -1;
  }

  /* Initialize character device using the specific file operations structure */
  cdev_init(&leddev_dev.cdev, &leddev_fops);
  leddev_dev.cdev.owner = THIS_MODULE;

  /* Request kernel to make one device available */
```

```
  error = cdev_add(&leddev_dev.cdev, leddev_dev.devt, 1);
  if (error) {
    printk(KERN_ALERT "cdev_add() failed: error = %d\n", error);
    unregister_chrdev_region(leddev_dev.devt, 1);
    return -1;
  }

  return 0;
}

/* Create a class for the device driver /sys/class/leddev [LDD Ch 14] */
static int __init leddev_init_class(void)
{
  /* Create a class named leddev */
  leddev_dev.class = class_create(THIS_MODULE, "leddev");

  if (!leddev_dev.class) {
    printk(KERN_ALERT "class_create(leddev) failed\n");
    return -1;
  }

  /* Create class representation in the file system */
  if (!device_create(leddev_dev.class, NULL, leddev_dev.devt, NULL, "leddev")) {
    class_destroy(leddev_dev.class);
    return -1;
  }

  return 0;
}

/* Reserve and initialize the GPIO pins needed for the driver */
static int __init leddev_init_pins(void)
{
  /* Request and configure GPIOs for the level shifter */
  if (gpio_request(GPIO_OE, "OutputEnable")) {
    printk(KERN_ALERT "gpio_request failed\n");
    goto init_pins_fail_1;
  }

  if (gpio_direction_output(GPIO_OE, 0)) {
    printk(KERN_ALERT "gpio_direction_output GPIO_OE failed\n");
    goto init_pins_fail_2;
  }

  if (gpio_request(GPIO_DIR, "Direction")) {
    printk(KERN_ALERT "gpio_request(2) failed\n");
    goto init_pins_fail_2;
  }

  if (gpio_direction_output(GPIO_DIR, 0)) {
```

```
    printk(KERN_ALERT "gpio_direction_output GPIO_DIR failed\n");
    goto init_pins_fail_3;
  }

  if (gpio_request(GPIO_BIT0, "Bit0")) {
      printk(KERN_ALERT "gpio_request(0) failed\n");
      goto init_pins_fail_4;
  }

  if (gpio_direction_output(GPIO_BIT0, 0)) {
      printk(KERN_ALERT "gpio_direction_output GPIO_BIT0 failed\n");
      goto init_pins_fail_4;
  }

  return 0;

  /* Error handling code: free in reverse direction */

 init_pins_fail_4:
  gpio_free(GPIO_BIT0);

 init_pins_fail_3:
  gpio_free(GPIO_DIR);

 init_pins_fail_2:
  gpio_free(GPIO_OE);

 init_pins_fail_1:

  return -1;
}

/* Kernel module initialization function */
static int __init leddev_init(void)
{
  /* Zero memory for device struct */
  memset(&leddev_dev, 0, sizeof(leddev_dev));

  /* Run initialization functions in-turn */
  if (leddev_init_cdev())
    goto init_fail_1;

  if (leddev_init_class())
    goto init_fail_2;

  if (leddev_init_pins() < 0)
    goto init_fail_3;

  /* Reset driver state */
  leddev_reset();
```

```
  return 0;

  /* Failure handling: free resources in reverse order (starting at the point we got to) */

 init_fail_3:
  device_destroy(leddev_dev.class, leddev_dev.devt);
  class_destroy(leddev_dev.class);

 init_fail_2:
  cdev_del(&leddev_dev.cdev);
  unregister_chrdev_region(leddev_dev.devt, 1);

 init_fail_1:

  return -1;
}
module_init(leddev_init);

/* Kernel module release function */
static void __exit leddev_exit(void)
{
  /* Free all GPIOs */
  gpio_free(GPIO_OE);
  gpio_free(GPIO_DIR);
  gpio_free(GPIO_BIT0);

  /* Free class device */
  device_destroy(leddev_dev.class, leddev_dev.devt);
  class_destroy(leddev_dev.class);

  /* Free device itself */
  cdev_del(&leddev_dev.cdev);
  unregister_chrdev_region(leddev_dev.devt, 1);
}
module_exit(leddev_exit);

/* Module meta-information */
MODULE_AUTHOR("Ulrik Pagh Schultz");
MODULE_DESCRIPTION("A module for controlling a single GumstixNXT LED");
MODULE_LICENSE("Dual BSD/GPL");
MODULE_VERSION("0.1");
```

C Quick guide to installing Ubuntu

This guide assumes you are using Ubuntu 10.04 LTS (although 10.10 has also been shown to work, and 12.04 can be made to work with a bit of reconfiguration). To get Ubuntu 10.04 LTS running on your computer, there are basically three installation options:

1. Repartitioning your harddisk to make room for a Linux installation on a separate partition.

2. Installing Linux to an external media such as a USB harddrive.

3. Installing Linux using the Wubi installer, which installs Linux in a file on your Windows partition.

If you know what you are doing, the first option gives you the best performance, whereas the third option is both easy and quick and gives fairly good performance. The second option has not been tested and is likely to not give stellar performance for the long-running OpenEmbedded build process, it might work but we have not tested it. The third option is recommended for beginners: it requires you to have enough free space on your harddrive, but then Linux is installed transparently and can be selected when you reboot your machine.

The internet contains extensive information on these options. Go to www.ubuntu.com to read about Ubuntu, and search for "Wubi" on the www.ubuntu.com website to learn more about it. The OpenEmbedded environment recommends having at least 60Gb of space on your installation, but for the options we are using for our system it seems to work with less: 40Gb of space seems to work, and if short on space 30Gb can be used as a minimum.[6]

To install Ubuntu 10.04.04 LTS (also called "Lucid Lynx") on your machine, download the appropriate file from the corresponding repository — typically `http://releases.ubuntu.com/lucid/` — so to use Wubi, download `wubi.exe` from this URL. Linux sometimes has issues with specific peripherals and hardware-specific facilities, in particular on portable PCs. Graphics cards and power management are typical culprits, as is the wireless network. Graphics cards and power management typically work but with a reduced functionality (in which case e.g. graphics will slow and the computer cannot be put to sleep). The wireless network is more of an issue, but here an external USB network card can be used. The ASUS USB-N13 is verified to work with Ubuntu 10.04.04, although unfortunately not out-of-the-box, it requires using appropriate drivers which can be downloaded and transferred via USB-stick:

`http://driverscollection.com/?H=USB-N13%20%28VER.B1%29&By=ASUS`

(There is an error in the "install.sh" script, everywhere it says "su" without "sudo", change it to "sudo su".)

Once the module has been installed, plug in the USB network adapter. You can now connect to eduroam, but make sure to pick "PEAP" for "EAP method" (and don't worry about the lack of a certificate).

D Notes on using a virtualized host for development

You can use a virtualized host for your development, but it puts certain requirements on the operating system running on the physical machine. Compiling open embedded can be done in any host with appropriate resources, e.g. 2Gb of RAM and a 40Gb filesystem works with Ubuntu (although as little as 30Gb can work, and 60Gb is apparently recommended). The hard part is interacting with the SD card, which requires the ability to partition (MBR scheme) and create filesystems (FAT and EXT3). We have tried two different options: attaching the card reader directly to the virtualized host, and using Mac OS to interact with the SD card.

D.1 Attaching card reader to virtualized host

This approach has been verified to work with VMWare 3.x and Mac OS, has been abandoned for Windows 7 due to technical complications, and seems more likely to work with Windows XP (but this is untested). The idea is to simply attach the SD card reader directly to the virtualized host as a USB device, this

[6] Wubi by default imposes af maximum size of 30Gb, this works for what you will normally be doing in this class. If you run out of space, it is fairly easy to expand dynamically afterards by adding another virtual harddisk for storing the open embedded environment. The Wubi main disk can also be expanded after installation, but this is out of the scope of this guide.

allows the virtualized host full control over low-level access to the SD card (but apparently this does not work very well under Windows 7). Note that many internal SD card readers are also attached via USB. Using this approach the SD card becomes directly accessible as a device under `/dev`, and can be used as described earlier in this document..

D.2 Using Mac OS to interact with the SD card

This approach is apparently very close to working, but has not been shown to be completely functional. The prerequisites are that you are using macports with the packages e2fsprogs, ext2fuse, and macfuse installed.

Partitioning can be done using fdisk; the functionality is equivalent to Linux fdisk but the interface is very simplistic. Nevertheless, using the -d and -r options one can save and restore partition schemes from an SD card. So using fdisk with -d one can save the table from one card, and then write it to another card of the same size using the -r option.

Formatting is simple, the required utilities are available either by default or using the aforementioned macports packages:

```
sudo newfs_msdos -F 32 /dev/disk2s1
sudo mkfs.ext3 /dev/disk2s2
```

Mounting the card partitions is however not so easy. The FAT partition mounts automatically and can be mounted manually using e.g.:

```
sudo mount -t msdos /dev/disk2s1 mnt/card
```

The EXT3 partition is however not easily mounted. Using FUSE one can mount EXT2/3 partitions, as follows:

```
sudo ext2fuse /dev/disk2s2 mnt/card -o force,allow_other &
```

This almost works except that apparently it cannot create /dev special files and hard links, both of which is required for creating the basic Linux file system. If this problem could be resolved then the approach would work. (Alternatively the EXT3 filesystem could be created from within Linux as an image that was then copied to the SD card using the dd command, this works but is too slow for e.g. a 4 Gb card since it requires to transfer all the data for the entire partition.)