Exercise 10: Developing for NGW100 target on Linux host

In this exercise we will create the necessary tools for cross-platform development with a NGW100 (AVR32) target and configure an Linux host. NGW100 is a development card with an AVR32 AP7 CPU that is able to run Linux. For this exercise you will need to run Linux Mint on the host computer.

To develop and compile on a host computer and run the code on a target system is an often used method for embedded system development. An embedded system has little or no user interface and limited computational power, so it is difficult or impossible to use that system for development.

1. Buildroot

Buildroot is a set of scripts and a menu system that can be used to build everything you need for cross-platform development. It creates a linux kernel, file system and basic tools for the target, and it also creates tools for cross-compiling that means that you can build programs for the target on the host computer.

1.1. Building Buildroot

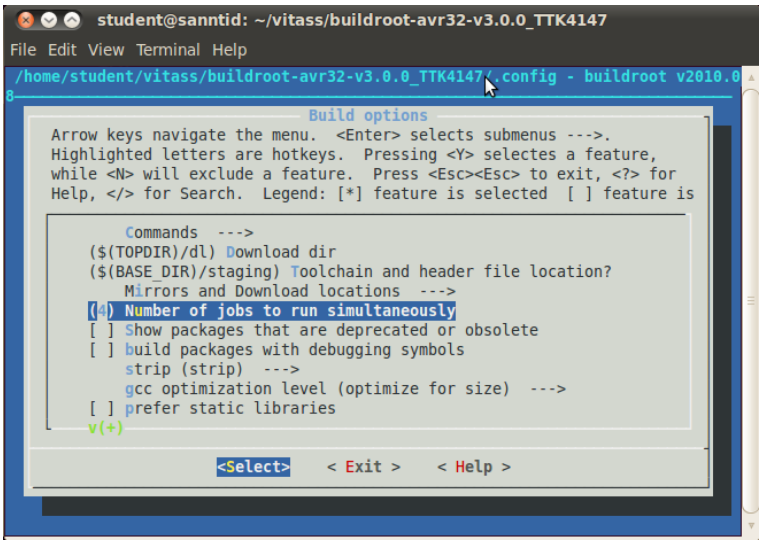
To start you should download a custom made buildroot from the link in It's learning. Unpack it and move into the newly created folder in the terminal and type:

```
make atngw100_defconfig
```

This will set the default buildroot configuration for NGW100. Next you should type:

```
make menuconfig
```

This will show you a menu that allows you to modify the build configuration of Buildroot. For now you will only do one change here. Select "Build options" and select the "Number of jobs to run simultaneously" to 4. This will ensure that you use all of the four CPU cores of the computer, which is important because it is quite time consuming to build Buildroot.



Exit menuconfig and save the new configuration.

You have now seen the configuration of the buildroot itself. We can also look at the configuration of the linux kernel, by running:

```
make linux26-menuconfig
```

We will not do any changes to the kernel configuration, but we can give the kernel a custom name. Enter “General setup” and select “Local version…”. The text you add here is attached to the end of the kernel’s name. Exit menuconfig and save the new configuration.

Type:

```
make
```

Wait for the process to complete. It should take between 15 and 20 minutes.

When you run “make”, you will probably get an error message about “variable ‘ret’ set but not used…”. To solve this replace the `<buildroot>/output/build/host-makedevs/makedevs.c` with the `makedevs.c` file found on it’s learning. Do a new “make” command, and the build process should finish in 10 seconds. This must be done AFTER the initial “make” command, as the file is created during the first build.s

While you are waiting you should read the rest of the exercise and also take a look at the Buildroot documentation. A good place to start is to open buildroot.html in firefox, the file is located in the docs folder.

1.2. What did Buildroot create

When Buildroot completes it is time to inspect what it did create for us.

- Filesystem – Buildroot created a filesystem for the target (in our case the NGW100). This can be found as either rootfs.tar or rootfs.tar.gz in the `output/images` folder of Buildroot.
- Linux kernel – The linux kernel created by Buildroot is also found in `output/images`. It is called uImage.

- Cross-compiling toolchain – Buildroot also created a cross-compile toolchain that lets you compile code for the NGW100 on the host computer. The toolchain executables can be found in the `output/staging/usr/bin` folder.

2. Netboot

We have now created the tools for running Linux on NGW100 and develop programs for it. We now need to boot the NGW100. There are two ways to boot this card, either with a SD card or via network. The most useful method for development is to boot via the network, so we will do that.

Several steps are necessary to configure netboot.

2.1. Connecting the NGW100 to the host computer

The NGW100 should be connected to the host computer with a serial cable and a crossed Ethernet cable. The serial cable should be connected to the back of the host computer which is COM1. The Ethernet cable should be connected to the WAN port on the target and the free network port on the host.

2.2. U-boot

Run `minicom` on the host computer. Make sure that minicom is configured to use `/dev/ttyS0`, '115200 8N1', and verify that “hardware flow control” is turned off. Press 'Ctrl-A Z' then 'O' to change the serial port setup.

```
minicom -D /dev/ttyS0
```

Power the NGW100, and see the text indicating that the NGW100 is booting. Press space when asked to abort autoboot. You will now get a U-boot prompt where you can configure the U-boot bootloader. Write `printenv` to see the environmental variables, and check that they are as follows:

```
bootdelay=1
baudrate=115200
hostname=atngw100
ethact=macb0
ethaddr=00:11:22:33:44:55
serverip=192.168.0.1
tftpserver=192.168.0.1
bootcmd=set bootfile uImage;dhcp;bootm
bootargs=root=/dev/nfs nfsroot=192.168.0.1:/export/nfs ip=dhcp console=ttyS0
stdin=serial
stdout=serial
stderr=serial
```

If your variables are not like these, then you need to change them using `set` like this:

```
setenv bootcmd 'set bootfile uImage;dhcp;bootm'
```

If you have changed the variables, they must be saved with `saveenv`.

2.3. TFTP

TFTP stands for Trivial File Transfer Protocol and is used to transfer the kernel image from the host to the target. You should copy your uImage file from Buildroot (see section 1.2) to `/export/tftp` on your host. The NGW100 will then fetch the uImage file from the host when it boots.

2.4. NFS

NFS stands for Network File System and is used to share the root file system to the target. The folder /export/nfs have been set up with a file system generated with Buildroot, but you should update to make sure it has the newest files from buildroot. Copy the `rootfs.tar.gz` file from `output/images` (in the buildroot folder) to `/export/nfs`. Unpack the files with `tar xzf rootfs.tar.gz`.

You can safely ignore ALL errors and warnings!

2.5. Get root access

To install a kernel module, you must log into the target as root. Unfortunately the Buildroot documentation does not tell us what the default root password is, so we need to change this ourselves.

Password hashes are stored in /etc/shadow. If you open up /export/nfs/etc/shadow in an editor it will list the different users of the Linux system on the target. You need to change the part for the root into this:

```
root::10933:0:99999:7:::
```

2.6. Boot Linux on NGW100

You are now ready to boot the NGW100. Click on reset button on the card and look at the boot sequence on minicom. You should end up with a “Welcome to Buildroot” message. Log in as root, you will not be asked for a password, since you just set the user to have no password. You should however set a password, with the following command:

```
passwd root
```

Run the command to see which kernel you are running. You should see the text you added in the kernel configuration here.

```
uname –a.
```

Run the following command to find your IP address.

```
ifconfig
```

3. Cross-developing for NGW100

We have now booted the NGW100 with the files provided by Buildroot. The next step is to use the cross-compile toolchain from Buildroot to develop applications for the NGW100. To set up a cross-compile development environment is a useful skill, and you can learn a thing or two about how gcc and other tools works.

Assignment A:

Test cross-compiling of a simple program for the NGW100, both using a Makefile and in Eclipse. In addition, you should test remote debugging with Eclipse.

In the miniproject, you can choose for yourself whether you want to use and editor and a Makefile or Eclipse for your developing.

3.1. Cross-compiling with Makefile

You can use the same makefile that we have used earlier as a starting point. But where this makefile uses `gcc`, you must use the `avr32-linux-gcc` from Buildroot instead. Since the folder with the crosscompile executables isn't in the PATH variable of the Linux Mint system, you must reference the executable with its full path. Alternatively, you can add the folder to the PATH variable.

After you have modified the makefile, you can create a small program that prints a message. The executable created when running the makefile should NOT run on the desktop computer, but it should run on the NGW100.

3.2. Cross-compiling with Eclipse

Start eclipse, and change your workspace location to `/export/nfs/root`. This has the advantage of having all your project files available for your NGW100 target. Create a new C project of the type "Cross-Compile Project". Set Tool command prefix to "avr32-linux-", and the Tool command path to the folder where your cross-compile toolchain binary files are. Click the finish. You have now told Eclipse that you want to use the toolchain buildroot created to compile your program.

Add a source file to the project, and create a simple program. Right click the project and select Build Project. Open the terminal of your target, and find the compiled program in the file system and run it.

Running from Eclipse

We would like to use Eclipse for running and debugging the program we are developing. Right click your project, and select "Run As->Run Configurations". Add a new C/C++ Remote Application, and give it a name. Create a new Connection with the "New" button. Select SSH only and give the targets IP address as "Host name". In the "Remote Absolute File Path…" you should put the path to the program file from the targets view. It will typically be something like this: `/root/Hello World/Debug/Hello World`. You should also check the "Skip download to target path", since the file is already on the target's filesystem.

Click run, and provide the root username and password when asked for it. Say yes to the messages that appear. Your program output should be shown in the Eclipse Console. If not, try to run it again. Make sure you run the configuration you just created, since if you just say run, then Eclipse will create a new local application that will not work.

Debugging

A big advantage for using Eclipse for developing on the NGW100 is to use the built-in debugging tools. While setting up debugging for Eclipse is relatively straight forward, it is unfortunately somewhat unstable to use.

Right-click your project and select "Debug As->Debug Configurations". You keep the same configuration as you used for running, but we must in addition configure the debugger. In the Debugger panel you must change the GDB debugger to `avr32-linux-gdb` in your toolchain binary folder. You can use the "Browse" button to find it.

On the bottom of the Debug Configuration window there will be a text saying "Using GDB (DSF) Automatic Remote Debugging Launcher". Click the "Select other" link next to this text. Check for "Use configuration specific settings" and select "GDB (DSF) Manual Remote Debugging Launcher". This means that you must manually start the gdbserver on the NGW100 each time you want to debug. Ideally this should be started automatically by Eclipse, but it tends to mess things up. Select "Debugger – connection", change "Host name or IP" to 192.168.0.10 and "port number to 2345. To start the debugging server on the target, use the following command: `gdbserver 192.168.0.1:2345 <application name>`

After starting the gdbserver, you can click the "Debug" button in the Debug Configuration windows, and say yes to the request to change perspective. You should now be able to step through your program as if it was a local program.

> If Eclipse is having trouble finding shared libraries. Add `/export/nfs/lib` to `shared libraries` in the `debugger configuration`.

4. Creating a kernel module

A Linux kernel module is a small program that can be inserted into the running kernel. This program will then run in kernel mode. You must be root to insert a kernel module, which you are on the NGW100 system (but not on the desktop system). Programming kernel modules becomes really complicated, really fast, so we will only create a very small and simple module in this exercise.

4.1. Kernel module Makefile

Kernel modules are compiled differently than normal C-program. We must therefore use another Makefile. It must specify the cross-compiler to use, and also the location of the kernel source, as the kernel module code will use code from other parts of the Linux kernel. Below is a simple Makefile that will compile a kernel module with the name `mymodule.ko` from a `mymodule.c` source file. You must verify that the paths are correct and that there are TABs and not spaces in front of make and rm, or the Makefile will not work.

```
obj-m := mymodule.o

KERNELDIR := /home/student/buildroot-avr32-v3.0.0_TTK4147/output/build/linux-2.6.35.4
CROSS := /home/student/buildroot-avr32-v3.0.0 TTK4147/output/staging/usr/bin/avr32-linux-

all:
	make ARCH=avr32 CROSS_COMPILE=$(CROSS) -C $(KERNELDIR) M=$(shell pwd) modules

clean:
	rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
```

4.2. Kernel module source

You must search for how to program a kernel module yourself. It should be easy to find information about this. The C file for a kernel module that runs on avr32 should be the same as for another kernel module. Only the Makefile differs. The makefile template expects a C file called `mymodule.c`.

A typical source file contains two functions, one that will run when the module is installed, and one that runs when the module is removed from the kernel.

> *You can't use* `printf()` *in kernel mode, use* `printk(KERN_INFO "text", args..)` *instead. The output from* `printk()` *is found with* `dmesg`.

4.3. Install kernel module

Locate the `mymodule.ko` file on an avr32 terminal and can then install the module with:

```
insmod mymodule.ko
```

and removed it again with

```
rmmod mymodule
```

To see what your module printed to the kernel log, run `dmesg`.

Assignment B:

The `/proc` filesystem is normally used to present information about the kernel to user space modules. If you write `cat /proc/cpuinfo` in a Linux terminal, you will get some information about the computers CPU. You should make a small kernel module that creates a file in the `/proc` filesystem that you can read with `cat`.

In your init function, you should call the `create_proc_entry()` function, which returns a pointer to a `proc_dir_entry` struct. Store this pointer in a variable. The function has three parameters, the first is a string to the name of your `/proc` entry, the second defines the permission of the file and the third can be NULL. If you run `create_proc_entry("myproc", 0644, NULL)` you will create an entry called `/proc/myproc`.

You must also define a function that runs when the `/proc` entry is read. You can use the function given below. When ran, it inserts some text into the buffer, which will be sent to the process reading the file. To register this function, you must set the `read_proc` field, of the struct that was returned to you, equal to the name of the function.

```
int procfile_read(char *buffer, char **buffer_location, off_t offset,
                  int buffer_length, int *eof, void *data)
{
     if (offset > 0)
     {
          return 0;
     }
     else
     {
```

```
            return sprintf(buffer, "Hello world\n");
      }
}
```

When the module exits, you should remove the `/proc` entry with the following function call:

```
remove_proc_entry("myproc", NULL);
```

You should be able to find help and examples on this topic on the Internet.

5. Store your Toolchain

You will use your toolchain in the miniproject, so you should store it so you don't have to do the long compile again. It is not guaranteed that files on the real-time lab computer will be there next time…

6. Approving the exercise

For this exercise you need to show the following to a student assistant to get the exercise approved.

1. Cross compile a simple program for the NGW100 with Makefile and Eclipse, and run the resulting programs on the board.
2. Demonstrate remote debugging with Eclipse
3. Install the kernel mode, and show that you can read the `/proc` file with `cat`.