Exercise 8:
Cross Platform Development with NGW100

In this exercise, we will set up and test the tools necessary for the Miniproject. We will be using the NGW100 development card which has an AVR32 AP7000 processor, and we will be running Linux 2.6 as the operating system.

Since this system does not have the user interface or computational power required for development, we will instead do the programming work on the lab computer. This means we need a compiler toolchain that is properly configured to run on the lab computer, but generates code for AVR32. We also need to compile Linux and the necessary drivers to target AVR32 and the hardware peripherals available on the NGW100.

In order to do this, we will be using a tool called *Buildroot*, which is a set of scripts for configuring Linux and GCC (and a few other tools) for cross-platform development.

1. Buildroot

1.1. Building Buildroot

Download "buildroot" from Blackboard, or from

```
scp student@10.100.23.170:buildroot.tar.bz2 .
```

and unpack it. Open a terminal in the newly created folder and type

```
make atngw100_defconfig
```

This will set the default configuration for the NGW100. Next, 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 set 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.

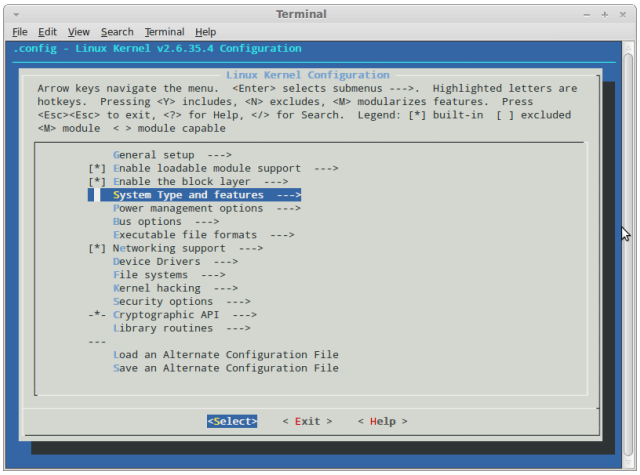
Exit menuconfig and save the configuration.

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 can increase the real-time performance of Linux by making some changes in “System Type and Features”. The first change is to change the timer frequency to 1000Hz, increasing the scheduler tick rate. The second thing is changing the Preemption Model to “Preemptible Kernel”, letting userspace tasks interrupt kernel tasks. However, this last option doesn’t work that well with kernel modules (which is what we will be making later in

this exercise), so leave it as is for now, and maybe come back to it later for the miniproject. You can also make other changes, but try not to break anything.



Exit menuconfig and save the new configuration. Now that all the configuration is complete, type

```
make
```

and wait for the compilation to complete. This could take about 15-20 minutes.

If you have made any changes to Buildroot or Linux, you should make sure to save the output folder for next time, so you don't have to re-compile everything again. If you plan to making other changes later, you should save the entire folder, as re-compiling is faster than compiling from scratch.

1.2. The Buildroot output files

The three most important things Buildroot creates are

- **Cross-compiling toolchain:** The files in `output/staging/usr/bin/` let you compile code for the NGW100 on the host computer
- **Linux kernel:** The linux operating system is found in `output/images`, in a file called `uImage`. (This file is called an "image" in the sense of the mathematics definition of the word - this file is directly mapped/copied from storage to RAM during the boot sequence)
- **File system:** The file system required by Linux is also found in `output/images`, and is found in the file `rootfs.tar` or `rootfs.tar.gz`

Since the NGW100 has limited storage, the two options for booting an operating system is with an SD card or via the network. The easiest option is via the network, as we don't have to fiddle around with SD cards. This means that we need to make the kernel image and the file system available over the network, by hosting these files on the lab PC and connecting the NGW100 to it with a network cable.

Several steps are required to configure netboot:

- **Connecting the NGW100:**
  Connect the crossed network cable between the WAN port on the NGW100 and the PCI network card on the lab PC. Connect the serial cable to the motherboard of the lab PC (close to the usb ports)
- **Configure U-boot:**
  Run `minicom -D /dev/ttyS0` in a new terminal window, and press '`Ctrl+A, Z`', then '`O`' to change the serial port setup to `115200 8N1`, with no hardware flow control. Power on the NGW100. You should see 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 environment 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, change them by using `setenv` like this:

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

If you had to change anything, save the changes with `saveenv`.

- **TFTP:**
  The Trivial File Transfer Protocol is used when transferring the kernel image from the host to the target during the boot sequence. The NGW100 bootloader will search for the bootfile (which should be set as `uImage` in the config above) in `/export/tftp` on the host computer. Copy `uImage` from the buildroot output to `/export/tftp`.
- **NFS:**
  A Network File System is used to share the root file system between the host and the target. Copy `rootfs.tar.gz` to `/export/nfs`, and extract its contents with `sudo tar xzf rootfs.tar.gz`. If there are any warnings or errors, you can safely ignore them. You may have to use `sudo` to copy files to `/export/nfs`.

We are now ready to boot the NGW100. Click on the reset button on the card, and look at the boot sequence on minicom. You should see a "Welcome to Buildroot" message appear. You are now asked for a login, but unfortunately the Buildroot documentation does not tell us what the default password is. But fortunately we can modify the password file from the host PC, bypassing the admin requirements we would need if we tried to do this from the NGW100. Open the password file

```
sudo nano /export/nfs/etc/shadow
```

and change the root password line to

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

Press Ctrl+X to exit, Y to save, and Enter to confirm the file name. This will disable the root password. In order to set a new root password, restart the NGW100, log in as root (there should be no password now), and set the new password with

```
passwd root
```

Note that the root user does not work properly if no password is set, so this step is not optional.

Now that you have logged in, you can type `pwd` to see your current directory in `/export/nfs`. If you want to go to `/export/nfs` directly, type '`cd /`'.

2. Cross-developing for NGW100

Now that we are running Linux on the NGW100, the next step is to use the cross-compile toolchain from Buildroot to create some programs.

Task A:

Write a simple "hello world" type program, and run it on the NGW100.

Instead of using `gcc` from the terminal, you must use `avr32-linux-gcc` found in `output/staging/usr/bin` from the Buildroot folder. Once you have compiled the program, you have to copy the executable to `/export/nfs`. Then swap to the minicom terminal window, and run the program on the NGW100.

3. Creating a kernel module

Kernel modules are pieces of code that extend the functionality of the kernel, and can be loaded and unloaded on demand. This lets you get access to everything in kernel space, letting you deal with low-level hardware features like interrupts or crating device drivers, or more high-level Linux features that are designed for extension (like netfilter or sysfs). However, we cannot call standard C library functions from inside kernel modules, due to a combination of the way kernel modules are loaded (specifically linking to symbols of the stdlib functions), and the kernel not being reentrant (we can't use syscalls while already in kernel space).

This means that kernel module code is quite different from "normal" user-space application code, both in terms of how the program is structured (no "main" function) and what the program can do (no standard library).

3.1. Kernel module makefile

Compiling a kernel module requires a few extra compiler options, so below (and on blackboard) is a makefile that will compile a kernel module with the name mymodule.ko from the file mymodule.c. Make sure to check the path to the toolchain folder, and make sure that the rules for all and clean are indented with TABs (not spaces).

```
obj-m += mymodule.o
BUILDROOTDIR := /home/student/Desktop/buildroot-precompiled
KERNELDIR := $(BUILDROOTDIR)/output/build/linux-2.6.35.4
CROSS := $(BUILDROOTDIR)/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
```

Task B:

Create a "hello world" kernel module. You should search online to find more information than what is given below, but here's a quick intro:

The start and stop function should be created like this:

```
#include <linux/module.h>   // included for all kernel modules
#include <linux/init.h>     // included for __init and __exit macros

static int __init mymodule_init(void){
    return 0;    // Non-zero return means that the module couldn't be loaded.
}

static void __exit mymodule_cleanup(void){
}

module_init(hello_init);
module_exit(hello_cleanup);
```

Since you cannot use `printf()`, you will have to use `printk()` instead:

```
#include <linux/kernel.h> // included for KERN_INFO

printk(KERN_INFO "Hello world!\n");
```

The output of `printk` is logged to the kernel message buffer, which can be displayed to the terminal by calling `dmesg`.

Once you have compiled the module, copy it to /export/nfs. Then from the minicom terminal, load the module with

```
insmod mymodule.ko
```

and remove it with

```
rmmod mymodule
```

Task C:

Create any other slightly more advanced kernel module.

Below is a short walkthrough on how to create an entry in /proc, but you are free to do something completely different. If you want to use a kernel module on the Miniproject, this is where you should start thinking about what that module should be. Some ideas: See if you can create a new thread (it won't be a pthread!), schedule something to run periodically, or capture network packets.

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 can 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 assign the `procfile_read` function to the the `read_proc` field of the struct that was returned to you.

```
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);
```