|  |
| --- |
|  |
|  |
|  |
| **EMBEDDED LINUX**  **ONLY FOR FACULTY** |

**PhyBOARD-RANA-AM335x**

**Single BoardComputer**

**System Development User Manualfor Linux**

A description...

**Table of Contents**

***Prepare Host Environment***

**Configuring Services (TFTP, NFS**

Configuring TFTP

Configuring NFS Service

**Partitioning Micro SD card in Ubuntu**

Method 1: Using Graphical User Interface

Method 2: Using Command line interface

**Setting Serial Console (Minicom)**

**Installing Toolchain**

Downloading the Toolchain

Export the Toolchain to Shell Path

***Download prebuilt images***

***Flashing binaries***

**Using Barebox bootloader**

Booting from Network

Booting from Micro SD card

Booting from NAND

**Prepare Host Environment:**

|  |
| --- |
| In Order to follow this manual the host requires 32-bit Ubuntu, better go with LTS versions of Ubuntu. To install Ubuntu follow the link below.  [*https://help.ubuntu.com/community/GraphicalInstall*](https://help.ubuntu.com/community/GraphicalInstall) |

Ubuntu doesn’t come with all the pre-requisite archives for the development, to prepare the host for development download the bash shell script which can be available from the below link. Download and execution steps of the script are as follows.

*$wget*[*ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/tools/scripts/elinux\_pkg.sh*](ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/tools/scripts/elinux_pkg.sh)

*$ sudo chmod +x elinux\_pkg.sh*

*$ sh elinux\_pkg.sh*

**Configuring Services (TFTP, NFS)**

***Configuring TFTP***

|  |
| --- |
| Installation of the TFTP Server is as follows; execute the command on Linux shell.  ***$ sudo apt-get -y install xinetd tftpd tftp***  ***$sudo mkdir /var/lib/tftpboot***  ***$ sudo vim /etc/xinetd.d/tftp*** |

Add the following structure in /etc/xinetd.d/tftp file if it is not there or edit if it exists.

service tftp

{

protocol = udp

port = 69

socket\_type = dgram

wait = yes

user = nobody

server = /usr/sbin/in.tftpd

server\_args = /var/lib/tftpboot -s

disable = no

|  |
| --- |
| }  ***$ sudo service xinetd restart*** |

**Testing TFTP Server**:

***$ sudo cp <file\_name> /var/lib/tftpboot***

***$ sudo chmod +wr –R /var/lib/tftpboot***

***$ sudo tftp localhost***

***>>> get <file\_name>***

If the same file is received without any error from the tftp server then the tftp service is configured correctly.

To transfer file through tftp between host and target follow steps listed below, configure the IP address for host by selecting the option Network Connection and modify the related option. The same can be done through command line interface by issuing the following command.

***Host Side Setup:***

***$ sudo ifconfig eth0 192.168.1.12 up***

*This will be used as server IP address. Note that 192.168.1.12 is not mandatory you can give any IP but IP for server and target should not be same*

***Target Side Setup:***

|  |
| --- |
| On Targetside issue the command with a different IP as below  ***$ ifconfig eth0 192.168.1.11 up*** |

Configure the gateway address for target

***$ route add default gw 192.168.1.1***

Check the default gateway by command ***$ route***

***or***

***$ /sbin/route -n***

Issue the tftp command to transfer the file

***$ tftp -r <file\_name > -g <server\_ip>***

First step in NFS service configuration is to decide the path of a directory which is going to share the RFS. Here in this manual it has been taken as /nfsroot which is to be created by user by issuing the following command.

***$ sudo mkdir /nfsroot***

Edit the/etc/exports file and add the below line at the end of the file, save and close it

/nfsroot \*(rw, sync, no\_subtree\_check, no\_root\_squash)

***$ sudo gedit /etc/exports***

To restart the service with made changes issue the following commands on the terminal.

***$ sudo exportfs–ra***

***$ sudo /etc/init.d/nfs-kernel-serverrestart***

**Partitioning Micro SD cardin Ubuntu**

***Method 1: Using Graphical User Interface***

Use the application Disk Utility in Ubuntu.

 Select Mass storage device

 Unmounts the volume

 Delete the partition

 Create the two partition boot and rootfs with *fat* and *ext3*support respectively

 Select the first *fat* partition and then select “edit partition” and mark it as bootable.

Figure 2

A description...

***Method 2: Using Command line interface***

|  |
| --- |
| Download the script and follow the steps.  ***$wget****ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/tools/scripts/mkcard.sh* |

Change the mode of script

***$ chmod 777 mkcard.sh***

Insert the Micro SD card and issue the following command and check Micro SD card device interface

***$ dmeg | tail***

Execute the following script with that device interface like /dev/sda ***$ sudo ./mkcard.sh /dev/sdx/\* /dev/sdx is device interface name \*/***

**Setting Serial Console (Minicom)**

***$ sudo minicom –s***

Select the option serial port setup from menu on terminal and press 'Enter'

A description...

Using the options at left; make the setting as shown in the figure 3.2. For ex: Serial device -/dev/ttsyS0, first press ‘A’ the cursor blinks at the particular line then change the device name to /dev/ttyS0 and press ‘ENTER’.

Note : The /dev/ttyS0 is not fixed it may vary depending upon the interface which You are using for example if you are using usb to serial converter then the node will be /dev/ttyUSB\*

A description...

After making all changes like baud rate -115200, hardware control flow -No etc. that are required press ‘ENTER’ to come out from the screen, now the previous screen will appear.

A description...

Now select the ‘Save setup as dfl’ it saves the setting as default.

Now select ‘Exit’

***Installing Toolchain***

This section explains how to install prebuilt toolchain binaries on the Host. Whereas building of Toolchain is explained in build system section.

***Downloading the Toolchain:***

|  |
| --- |
| Download the prebuild toolchain from the public ftp of PHYTEC from the link below.  *$wget*[*ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/tools/toolchain/arm-cortexa8-linux-gnueabihf.tar.bz2*](ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/tools/toolchain/arm-cortexa8-linux-gnueabihf.tar.bz2) |

You can also download the OSELAS.Toolchain-2012.12.1 source code compile it and get the toolchain from the below link.

*$wgetftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/tools/toolchain/OSELAS.Toolchain-2012.12.1.tar.bz2*

***Export the Toolchain to Shell Path:***

Extract the toolchain that has downloaded in the above section to the HOME directory. For extracting the toolchain from GUI you can ‘right click’ on the toolchain and select ‘Extract’. Same can be done from terminal issue the following:

***$ tar –xvf arm-cortexa8-linux-gnueabihf.tar.bz2***

Now write a script that export the target Architecture and toolchain prefix as below.

**$ vim env.sh**

In the file add the following lines

#!/bin/sh

export PATH=$PATH:/home/<user\_name>/<path to toolchain bin>

export ARCH=arm

export CROSS\_COMPILE=arm-cortexa8-linux-gnueabihf-

Save and exit from the editor. Execute the script by issuing

***$ . env.sh***

Note:The Command is “dot space env.sh”. Execute the above command for every new shell where you are trying to start compilation of Linux Kernel or Boot loader

|  |
| --- |
| Check whether the toolchain path is set or not by issuing  ***$ echo $PATH***  Above command displays the entire paths that are added to Shell PATH variable separated by colon ‘:’. See if your path is printed on the terminal. If you are not able to see your toolchain path repeat the steps from the beginning of this section.  ***Download prebuilt images:***  This section gives the links to all the required binaries to boot the phyBOARD-RANA-AM335x using the simplest method using Micro SD card. Partition your Micro SD card.Follow the Partitioning Micro SD cardin Ubuntu section.  All the tested binaries are placed at the public PHYTEC ftp.Follow the links below to download the binaries.  MLO:  [*ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/images/MLO*](ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/images/MLO)  Barebox:  *ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/images/barebox-image*  Linux:  *ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/images/linuximage*  Root.tar  *ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/images/root.tgz*  Root.ubi  *ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/images/root.ubi*  Note:Please change the barebox-image to barebox.bin  **$mv barebox-imagebarebox.bin**  **Flashing binaries**  A description...  Figure shows two methods. The first method used to provide all needed components to run on the target itself. The Linux kernel image and the root file-system image are persistent in the media the target features. This means the only connection needed is the serial cable to see what is happening on our target. This method is called standalone. This works after writing image into NAND which is explained later in the Document.  The other method is to provide needed components via network. In this case the development host is connected to the phyBOARD-RANA-AM335xwith a serial cable and via Ethernet; the embedded board boots into the bootloader, then issues a TFTP request on the network and boots the kernel from the TFTP server on the host. Then, after decompressing the kernel into the RAM and starting it, the kernel mount sits root file system atphyBOARD-RANA-AM335xBoard/root directory.  The latter one is especially for development purposes, as it provides a very quick turnaround while testing the kernel and the root file system. Other methods like Micro SD card and NAND have also been discussed.  ***Using Barebox bootloader:***  ***Booting from Network:***  Now flash Kernel image and rootfs with tftp & nfs. Before using this method make sure that tftp & nfs-server is working fine. To test and configure refer section “Configuring Services (TFTP, NFS)”Copy the uImage (kernel image) and root filesystem into /var/lib/tftpboot & /nfsrootrespectively.  On Host: ***$ sudo cp linuximage/var/lib/tftpboot***  ***$ sudo tar -xvf rootfs.tar -C /nfsroot***  Now minicom terminal, reset the board & press ‘m’ for bare-box menu or type exit in the shell & select option 3to boot from network.  ***Booting from Micro SD card***  For partitioning the Micro SD card refer section *“Partitioning a Micro SD card in Linux”*  Copy MLO, barebox.bin, Linux image in first partition (boot) and Root file system in second partition (rootfs).  ***On HOST:***  Insert the Micro SD card.  ***$ cp MLO /media/boot***  ***$ cp barebox.bin /media/boot***  ***$ cp linuximage/media/boot***  ***$ sudo tar -xvf rootfs.tar -C /media/rootfs***  ***Target:***  Insert the Micro SD cardand make sure that the pins 3+4 of JP5 is shorted, Power On the board,press ‘m’ for barebox menu & select option 2 to boot from SD Card.  ***Booting from NAND:***  If booting from SD Card start from here:  ***barebox@Phytec phyBOARD RANAAM335x:/erase dev/nand0***  ***barebox@Phytec phyBOARD RANAAM335x:/cp boot/MLO dev/nand0.xload.bb***  ***barebox@Phytec phyBOARD RANAAM335x:/cp boot/barebox.bin dev/nand0.barebox.bb***  ***barebox@Phytec phyBOARD RANAAM335x:/cp boot/linuximage dev/nand0.kernel.bb***  ***barebox@Phytec phyBOARD RANAAM335x:/cp boot/root.ubi dev/nand0.root.bb***  Remove the Micro SD card and make sure that JP5 is open, Power On theboard, press ‘m’ for barebox menu & select option 1 to boot from NAND.  **Preparing phyBOARD-RANA-AM335x Bootloader& Kernel**  This is required if board is not flashed with correct boot loader. Even you can skip to the flashing section if you want to use prebuilt images.  ***Preparing phyBOARD-RANA-AM335x Barebox bootloader:***  Note: Refer section 1.4 Installing Toolchain before going ahead.  Download the source of barebox version 2013.07.0from the following link.  *http://barebox.org/download/barebox-2013.07.0.tar.bz2*  After getting the source copy it to working directory  ***$ cp barebox-2013.07.0.tar.bz2~/work***  ***$ cd ~/work***  Extract the source here by following command  ***$ tar -xvf barebox-2013.07.0.tar.bz2***  Go to extracted folder and issue the following command  ***$ cd ~/work/<directory path to source>***  Note: We had already applied the patches and uploaded the barebox source code on Phytec Ftp you can directly download it from the below link.  ***$wget****ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/src/bootloaders/barebox/barebox-2013.07.0.tar.bz2*  If you want to do it manually then download the patches for phyBOARD-RANA-AM335x from the public ftp from the link below. Apply the patches.  ***$wget****ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/src/patches/barebox-2013.07.0/barebox/\**  Apply the patches one by one using following command.  ***$patch -p1 <<patch\_file\_name.patch>/\* apply all patches one byone \*/*** |

Cleaning all the binaries and other configuration files

***$ make clean***

***$ make distclean***

***$ make rana\_am335\_mlo\_defconfig /\*configuring the MLO for target \*/***

***$ make /\*It will create MLO for target \*/***

***$ make rana\_am335\_defconfig /\*configuring the Barebox for target \*/$ make /\*It will create barebox.bin for target \*/***

If the compilation goes well you will find the files MLO and barebox.bin in the source directory. If you experience any error act accordingly or post the errors at our forums you will get help accordingly.

***PreparingphyBOARD-RANA-AM335xLinux kernel***

Note: Refer section 1.4 Installing Toolchain before going ahead.

|  |
| --- |
| Download the source of Linux kernel version 3.2from the following link.  [*ftp://ftp.kernel.org/pub/linux/kernel/v3.x/linux-3.2.tar.bz2*](ftp://ftp.kernel.org/pub/linux/kernel/v3.x/linux-3.2.tar.bz2) |

After getting the source copy it to working directory

***$ cp linux-3.2.tar.bz2~/work***

***$ cd ~/work***

Extract the source here by following command

***$ tar -xvf linux-3.2.tar.bz2***

Go to extracted folder and issue the following command

***$ cd ~/work/linux-3.2/***

Note: We had already applied the patches and uploaded the linuxsource code on Phytec Ftp you can directly download it from the below link.

***$wget****ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/src/linux/linux-3.2.tar.bz2*

If you want to do it manually then download the patches for phyBOARD-RANA-AM335x from the public ftp from the link below. Apply the patches.

***$wget****ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/src/patches/linux-3.2/linux/\**

Apply the patches one by one using following command.

***$ patch -p1 <<patch\_file\_name.patch>/\* apply all patches one by one \*/***

Cleaning all the binaries and other configuration files

***$ make clean***

***$ make distclean***

***$ make rana\_am335x\_defconfig/\*configuring the Kernelfor target \*/***

***$ make uImage***

If the compilation goes well you will find the files zImage and uImage in arch/arm/boot directory. If you experience any error act accordingly or post the errors at our forums you will get help accordingly.

**Compiling User-application:**

Note: Refer section 1.4 Installing Toolchainbefore going ahead.

Make a helloworld program fortarget. Open file helloworld.c via command

***$ gedit helloworld.c***

Write a simple helloworld C program Compile it with toolchain

***$ arm-cortexa8-linux-gnueabihf-gcc helloworld.c -o helloworld***

Transfer it on target by tftp-server or by USB or Micro SD card(Please don't disturb mmc if booting form mmc).

***Host Side****:****TFTP***

*$ cp helloworld /val/lib/tftpboot/*

***SCP***

*$ scp helloworld root@<target ip>:~*

***Target Side:TFTP***

*$ tftp -r helloworld -g <host\_ip>*

*Then you able to see your ‘helloworld’ app in the current directory.*

*$ ./helloworld <to run the application>*

***SCP***

*No need to do anything it will be there in home directory just run the application using below command.*

*$ ./helloworld <to run the application>*

Insert the USB or MMC to target and switch to minicom terminal

It will display the device like sdb: sdb1. Use that device for mounting.

***$ mount /dev/sdb1 /mnt(sdb1) is USB-device interface***

***$ cd /mnt***

***$ ./helloworld/\* it is dot slash***

**Compiling RFS with PTX-dist:**

|  |
| --- |
| Downloadthe PTX-dist packetfrom the following link:$wget*ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/buildsystems/ptxdist/ptxdist-2013.01.0.tar.bz2* |

Extract the above downloaded source

***$ tar -xvf ptxdist-2013.01.0.tar.bz2***

Go to downloaded folder

***$ cd ptxdist-2013.01***

Configure the packet

***$./configure***

When the configure script is finished successfully, to compile run

***$ make***

***$ sudo make install***

Download the toolchain from the public ftp of PHYTEC from the link below.

$wget*ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/tools/toolchain/arm-cortexa8-linux-gnueabihf.tar.bz2*

Extract the toolchain that has been downloaded. Issue following

***$ tar –xvf arm-cortexa8-linux-gnueabihf.tar.bz2***

Download the PTX-dist BSP source from following link

$wget*ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/buildsystems/ptxdist/RANA-AM335x-PD13.0.0.tar.bz2*

Extract the source

***$ tar -xvf Rana-AM335x.PD13.0.0.tar.gz***

Go to extracted folder

***$ cd Rana-AM335x.PD13.0.0***

Select the toolchain

***$ ptxdist toolchain<path\_of\_toolchain\_bin>***

Start the build

***$ ptxdist go***

***$ptxdist images***

Check the images at platform-Rana-AM335x/images/

***$ lsplatform-Rana-AM335x/images/***

**Compiling RFS with Buildroot:**

|  |
| --- |
| Download the Buildroot source from following link$ wget *ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/buildsystems/buildroot/buildroot-2013.11.tar.bz2* |

Extract the source

***$ tar -xvf buildroot-2013.11.tar.bz2***

Go to extracted folder

***$ cd buildroot-2013.11***

Configure the buildroot by issuing following command

***$ make rana\_am335x\_defconfig***

Compile the buildroot

***$ make***

Check the output files at output/images

***$ ls output/images***

**Compiling RFS with Yocto:**

|  |
| --- |
| Download the Yocto buidsystem source from following link  $ wget [*ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/buildsystems/yocto/poky-dora-10.0.0.tar.bz2*](ftp://ftp.phytec.de/pub/Products/India/phyBOARD-RANA-AM335x/Linux/PD13.0.0/buildsystems/yocto/poky-dora-10.0.0.tar.bz2)  Extract the source  ***$ tar –xvf poky-dora-10.0.0.tar.bz2***  Go to extracted folder  ***$ cdpoky-dora-10.0.0***  Configure poky-dora-10.0.0by issuing following command  ***$ . ./oe-init-build-env***  Then type the following command to get hob GUI interface.  ***$ hob***  Then under “select machine” option select RANA\_AM335x to build images for phyBOARD-RANA-AM335x board.  To use hob refer the link given below.  *https://www.yoctoproject.org/documentation/hob-*  *-manual*  Check the output files at tmp/deploy/images  ***$ ls tmp/deploy/images*** |

**Expansion Selection for phyBOARD-RANA-AM335x**

In the barebox we are providing a menu so that it is easy for the user to select what feature he want to provide on the expansion for this power up the board and press “m” for the menu or type “exit” in the barebox shell. You will get the something like that.

A description...

Once you type “m” or “exit” in the barebox shell you will get a menu like this.

A description...

Here are a lot of option are provided to boot from different devices such as MMC, NAND, TFTP. You can select any one to boot but if you want to modify the default functionality on the expansion you have to go inside “Setting” option as shown in below diagram.

A description...

As you had seen that in “Settings” a lot of options are there you can set the network related things such as IP address in Network settings, etc. You can modify the default Expansion functionality by changing the 3rdoption “Expansion settings” as shown below.

A description...

By default we are providing support for LCD, SPI0, UART1, UART2, UART3, CAN0, CAN1, 28 GPIO’s. As it is mention a lot of options you can pass whatever you want on the expansion just by passing:

expansion=<Functionality>

 Eg. expansion=ALLGPIO (It will provide 70 gpio’son the Expansion)

Next if you need Max GPIO’s and 1 UART then you have to pass.

 expansion=ALLGPIO,UART1

(It will provide 68 gpio’s and UART1 tx, UART1 rxon the Expansion)

Next if you need Max GPIO’s, 1 UART and 1 CAN then you have to pass.

 expansion=ALLGPIO,UART1,CAN0

(It will provide 66 GPIO’s and UART1 Tx, UART1 Rx, CAN0 Tx, CAN0 Rxon the Expansion)

While booting the Kernel you will be able to see whatever devices you have selected.

A description...

As you had seen we had selected LCD, SPI0, UART1, UART2, UART3, CAN0, CAN1, 28 UART2, UART3, CAN0, CAN1, 28 GPIO’s so these devices has been initialised and are available on the expansion.

**Display Selection for phyBOARD-RANA-AM335x**

As it is shown in the below diagram that we are supporting these many LCD’s As it is shown in the below diagram that we are supporting these many LCD’s so whatever LCD you want to connect with phyBOARD-RANA-AM335x Board you had to just uncomment the line such as display=ZQ\_ZQ3506(It will select 3.5” LCD which we are providing by default). You can change the LCD as per your wish.

A description...

**What is GNU ‘Makefile’?**

Large projects can contain thousands of lines of code, distributed in multiple source files, written by many developers and arranged in several subdirectories. A project may contain several component divisions. These components may have complex inter-dependencies — for example, in order to compile component A, you have to first compile B; in order to compile B, you have to first compile C; and so on.

For a large project, when a few changes are made to the source, manually recompiling the entire project each time is tedious, error-prone and time-consuming.Make is a solution to these problems.

It can be used to specify dependencies between components, so that it will compile components in the order required to satisfy dependencies. An important feature is that when a project is recompiled after a few changes, it will recompile only the files which are changed, and any components that are dependent on it. This saves a lot of time. Make is, therefore, an essential tool for a large software project.

Each project needs a Makefile — a script that describes the project structure, namely, the source code files, the dependencies between them, compiler arguments, and how to produce the target output (normally, one or more executables). Whenever the make command is executed, the Makefile in the current working directory is interpreted, and the instructions executed to produce the target outputs. The Makefile contains a collection of rules, macros, variable assignments, etc. (‘Makefile’ or ‘makefile’ are both acceptable.)

**General Understanding of Makefile**

All variable names used in a Makefile are in upper-case. A common variable assignment in a Makefile is CC = gcc, which can be used later on as ${CC} or $(CC).

Makefiles use # as the comment-start marker, just like in shell scripts.

The general syntax of a Makefile rule is as follows:

|  |
| --- |
| target: dependency1 dependency2 ...  [TAB] action1  [TAB] action2      ... |

Let’s take a look at a simple Makefile for our sample project:

|  |
| --- |
| all: main.o module.o      gcc main.o module.o -o target\_bin  main.o: main.c module.h      gcc -I . -c main.c  module.o: module.c module.h      gcc -I . -c module.c  clean:      rm -rf \*.o      rm target\_bin |

We have four targets in the Makefile:

* All is a special target that depends on main.o and module.o, and has the command (from the “manual” steps earlier) to make GCC link the two object files into the final executable binary.
* main.o is a filename target that depends on main.c and module.h, and has the command to compile main.c to produce main.o.
* module.o is a filename target that depends on module.c and module.h; it calls GCC to compile the module.c file to produce module.o.
* clean is a special target that has no dependencies, but specifies the commands to clean the compilation outputs from the project directories.

The make command accepts a target parameter (one of those defined in the Makefile), so the generic command line syntax is make <target>. However, make also works if you do not specify any target on the command line, saving you a little typing; in such a case, it defaults to the first target defined in the Makefile. In our Makefile, that is the target all, which results in the creation of the desired executable binary target\_bin!

**Makefile processing, in general**

When the make command is executed, it looks for a file named makefile or Makefile in the current directory. It parses the found Makefile, and constructs a dependency tree. Based on the desired make target specified (or implied) on the command-line, make checks if the dependency files of that target exist.

If make finds that files in the dependency tree are newer than their target, then all the targets in the affected branch of the tree are executed, starting from the “lowest”, to update the dependency files. When make finally returns from its recursive checking of the tree, it completes the final comparison for the desired make target. If the dependency files are newer than the target (which is usually the case), it runs the command(s) for the desired make target.This process is how make saves time, by executing only commands that need to be executed, based on which of the source files (listed as dependencies) have been updated, and have a newer timestamp than their target.

For the execution of each target, make prints the actions while executing them. Note that each of the actions (shell commands written on a line) are executed in a separate sub-shell. If an action changes the shell environment, such a change is restricted to the sub-shell for that action line only. For example, if one action line contains a command like cd newdir, the current directory will be changed only for that line/action; for the next line/action, the current directory will be unchanged.

**Processing ‘Makefile’**

After understanding how make processes Makefiles. In the project directory, we run the following command:

|  |
| --- |
| prolific@pro:~$ make  gcc -I . -c main.c  gcc -I . -c module.c  gcc main.o module.o -o target\_bin |

When we ran make without specifying a target on the command line, it defaulted to the first target in our Makefile — that is, the target all. This target’s dependencies are module.o andmain.o. Since these files do not exist on our first run of make for this project, make notes that it must execute the targets main.o and module.o. These targets, in turn, produce the main.o andmodule.o files by executing the corresponding actions/commands. Finally, make executes the command for the target all. Thus, we obtain our desired output, target\_bin.If we immediately run make again, without changing any of the source files, we will see that only the command for the target all is executed:

|  |
| --- |
| prolific@pro:~$ make  gcc main.o module.o -o target\_bin |
|  |

Though make checked the dependency tree, neither of the dependency targets (module.o andmain.o) had their own dependency files bearing a later timestamp than the dependency target filename. Therefore, make rightly did not execute the commands for the dependency targets.

Now, we update module.c by adding a statement

printf("\nfirst update one"); inside thesample\_func() function. We then run make again:

|  |
| --- |
| Prolific @pro:~$ make  gcc -I . -c module.c  gcc main.o module.o -o target\_bin |

Since module.c in the dependency tree has changed make runs the action for the module.o target, which recompiles the changed source file. It then runs the action for the all target.

We can explicitly invoke the clean target to clean up all the generated .o files and target\_bin:

|  |
| --- |
| $ make clean  rm -rf \*.o  rm target\_bin |

More bytes on Makefiles

Make provides many interesting features that we can use in Makefiles. Let’s look at the most essential ones.

**‘Assignments’ Operators Ideas**

There are different ways of assigning variables in a Makefile. They are (type of assignment, followed by the operator in parentheses):

**Simple assignment (:=)**

We can assign values (RHS) to variables (LHS) with this operator, for example: CC := gcc. With simple assignment (:=), the value is expanded and stored to all occurrences in the Makefile when its first definition is found.

For example, when a CC := ${GCC} ${FLAGS} simple definition is first encountered, CC is set togcc -W and wherever ${CC} occurs in actions, it is replaced with gcc -W.

**Recursive assignment (=)**

Recursive assignment (the operator used is =) involves variables and values that are not evaluated immediately on encountering their definition, but are re-evaluated every time they are encountered in an action that is being executed. As an example, say we have:

|  |
| --- |
| GCC = gcc  FLAGS = -W |

With the above lines, CC = ${GCC} {FLAGS} will be converted to gcc -W only when an action like${CC} file.c is executed somewhere in the Makefile. With recursive assignation, if the GCC variable is changed later (for example, GCC = c++), then when it is next encountered in an action line that is being updated, it will be re-evaluated, and the new value will be used; ${CC} will now expand to c++ -W.

**Conditional assignment (?=)**

Conditional assignment statements assign the given value to the variable only if the variable does not yet have a value.

Appending (+=)

The appending operation appends texts to an existing variable. For example:

|  |
| --- |
| CC = gcc  CC += -W |

CC now holds the value gcc -W.

Though variable assignments can occur in any part of the Makefile, on a new line, most variable declarations are found at the beginning of the Makefile.Using patterns and special variables.The % character can be used for wildcard pattern-matching, to provide generic targets.

For example:

|  |
| --- |
| %.o: %.c  [TAB] actions |

When % appears in the dependency list, it is replaced with the same string that was used to perform substitution in the target., we can use special variables for matching filenames.

Some of them are:

$@ (full target name of the current target)

$? (returns the dependencies that are newer than the current target)

$\* (returns the text that corresponds to % in the target)

$< (name of the first dependency)

$^ (name of all the dependencies with space as the delimiter)

Instead of writing each of the file names in the actions and the target, we can use shorthand notations based on the above, to write more generic Makefiles.

**Action modifiers**

We can change the behaviour of the actions we use by prefixing certain action modifiers to the actions. Two important action modifiers are:

- (minus) — Prefixing this to any action causes any error that occurs while executing the action to be ignored. By default, execution of a Makefile stops when any command returns a non-zero (error) value.

Looking at the Makefile from our sample project: in the clean target, the rm target\_bin command will produce an error if that file does not exist (this could happen if the project had never been compiled, or if make cleanis run twice consecutively).

To handle this, we can prefix the rm command with a minus, to ignore errors: -rm target\_bin.

@ (at) suppresses the standard print-action-to-standard-output behaviour of make, for the action/command that is prefixed with @.

For example, to echo a custom message to standard output, we want only the output of the echo command, and don’t want to print the echo command line itself. @echo Message will print “Message” without the echo command line being printed.

Use PHONY to avoid file-target name conflicts

Remember the all and clean special targets in our Makefile? The conflicts will cause errors. Use the .PHONYdirective to specify which targets are not to be treated as files — for example: .PHONY: all clean.

**Installing GNU Make**

You have to install it, either using the package-management system, or by manually compiling fromsource, by running:

Example: sudo apt-get install build-essential.

**A sample project**

let’s use a simple C “Hello world” project, and a Makefile that handles building of the target binary. We have three files (below): module.h, the header file that contains the declarations; module.c, which contains the definition of the function defined in module.h; and the main file, main.c, in which we call the sample\_func() defined inmodule.c.

Since module.h includes the required header files like stdio.h, we don’t need to include stdio.h in every module; instead, we just include module.h. Here, module.c and main.ccan be compiled as separate object modules, and can be linked by GCC to obtain the target binary.

module.h:

|  |
| --- |
| #include <stdio.h>  void sample\_func(); |

module.c:

|  |
| --- |
| #include "module.h"  void sample\_func()  {      printf("Hello world!");  } |

main.c:

|  |
| --- |
| #include "module.h"  void sample\_func();  int main()  {      sample\_func();      return 0;  } |

The following are the manual steps to compile the project and produce the target binary:

|  |
| --- |
| prolific@pro:~$ gcc -I . -c main.c # Obtain main.o  prolific@pro:~$ gcc -I . -c module.c # Obtain module.o  prolific@pro:~$ gcc main.o module.o -o target\_bin #Obtain target binary |

(-I is used to include the current directory (.) as a header file location.)

|  |  |  |
| --- | --- | --- |
| hellomake.c | hellofunc.c | hellomake.h |
| #include “hellofunc.c”  int main() {  // call a function in another file  myPrintHelloMake();  return(0);  } | #include <stdio.h>  #include “hellomake.h”  void myPrintHelloMake(void) {  printf("Hello makefiles!\n");  return;  } | /\*  example include file  \*/  void myPrintHelloMake(void); |

Normally, you would compile this collection of code by executing the following command:

***gcc -o hellomake hellomake.c hellofunc.c -I.***

This compiles the two .c files and names the executable hellomake. The -I. is included so that gcc will look in the current directory (.) for the include file hellomake.h.

Unfortunately, this approach to compilation has two downfalls.

* First, if you lose the compile command or switch computers you have to retype it from scratch, which is inefficient at best.
* Second, if you are only making changes to one .c file, recompiling all of them every time is also time-consuming and inefficient. So, it's time to see what we can do with a makefile.

The simplest makefile you could create would look something like:

[part- 1](http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/makefile.1)

hellomake: hellomake.c hellofunc.c

gcc -o hellomake hellomake.c hellofunc.c -I.

* If you put this rule into a file called Makefile or makefile and then type make on the command line it will execute the compile command as you have written it in the makefile.
* Furthermore, by putting the list of files on which the command depends on the first line after the :, make knows that the rule **hellomake** needs to be executed if any of those files change.
* Immediately, you have solved problem-1 and can avoid using the up arrow repeatedly, looking for your last compile command.
* However, the system is still not being efficient in terms of compiling only the latest changes.One very important thing to note is that there is a tab before the gcc command in the makefile.

[part- 2](http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/makefile.2)

CC=gcc

CFLAGS=-I.

hellomake: hellomake.o hellofunc.o

$(CC) -o hellomake hellomake.o hellofunc.o -I.

So now we've defined some constants CC and CFLAGS. It turns out these are special constants that communicate to make how we want to compile the files hellomake.c and hellofunc.c. In particular, the macro CCis the C compiler to use, and CFLAGS is the list of flags to pass to the compilation command. By putting the object files--hellomake.o and hellofunc.o--in the dependency list and in the rule, make knows it must first compile the .c versions individually, and then build the executable hellomake.

Using this form of makefile is sufficient for most small scale projects.

* However, there is one thing missing: dependency on the include files. If you were to make a change to hellomake.h, for example, **make**would not recompile the .c files, even though they needed to be.

In order to fix this, we need to tell make that all .c files depend on certain .h files. We can do this by writing a simple rule and adding it to the makefile.

[part- 3](http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/makefile.3)

CC=gcc

CFLAGS=-I.

DEPS = hellomake.h

%.o: %.c $(DEPS)

$(CC) -c -o $@ $< $(CFLAGS)

hellomake: hellomake.o hellofunc.o

gcc -o hellomake hellomake.o hellofunc.o -I.

This addition first creates the macro DEPS, which is the set of .h files on which the .c files depend. Then we define a rule that applies to all files ending in the .o suffix. The rule says that the .o file depends upon the .c version of the file and the .h files included in the **DEPS macro**.

The rule then says that to generate the .o file, make needs to compile the .c file using the compiler defined in the CC macro. The -c flag says to generate the object file, the -o $@ says to put the output of the compilation in the file named on the left side of the :, the $< is the first item in the dependencies list, and the CFLAGS macro is defined as above.

As a final simplification, let's use the special macros $@ and $^, which are the left and right sides of the :, respectively, to make the overall compilation rule more general. In the example below, all of the include files should be listed as part of the macro DEPS, and all of the object files should be listed as part of the macro OBJ.

[part- 4](http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/makefile.4)

CC=gcc

CFLAGS=-I.

DEPS = hellomake.h

OBJ = hellomake.o hellofunc.o

%.o: %.c $(DEPS)

$(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)

gcc -o $@ $^ $(CFLAGS)

So what if we want to start putting our .h files in an include directory, our source code in a src directory, and some local libraries in a lib directory? Also, can we somehow hide .o files that hang around all over the place? The answer, of course, is yes. The following makefile defines paths to the include and lib directories, and places the object files in an obj subdirectory within the src directory.

It also has a macro defined for any libraries you want to include, such as the math library -lm. This makefile should be located in the src directory. Note that it also includes a rule for cleaning up your source and object directories if you type make clean. The .PHONY rule keeps make from doing something with a file named clean.

[part- 5](http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/makefile.5)

IDIR =../include

CC=gcc

CFLAGS=-I$(IDIR)

ODIR=obj

LDIR =../lib

LIBS=-lm

\_DEPS = hellomake.h

DEPS = $(patsubst %,$(IDIR)/%,$(\_DEPS))

\_OBJ = hellomake.o hellofunc.o

OBJ = $(patsubst %,$(ODIR)/%,$(\_OBJ))

$(ODIR)/%.o: %.c $(DEPS)

$(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)

gcc -o $@ $^ $(CFLAGS) $(LIBS)

.PHONY: clean

clean:

rm -f $(ODIR)/\*.o \*~ core $(INCDIR)/\*~

So now you have a perfectly good makefile that you can modify to manage small and medium-sized software projects. You can add multiple rules to a makefile; you can even create rules that call other rules.

**Build Linux Kernel Module**

There is no need of new full source tree in order to just compile or build module against the running kernel i.e an exploded source tree is not required to build kernel driver or module.

**Linux Kernel headers**

This is essential because if you just want to compile and install driver for new hardware such as camera or any device etc. Install the Linux-kernel-headers package which provides headers from the Linux kernel. These headers are used by the installed headers for GNU glibc and other system libraries as well as compiling modules. Use following command to install kernel headers:

**# apt-get install kernel-headers-2.6.xx.xx.xx**

Replace xx.xx with your actual running kernel version (e.g. 2.6.8.-2) and architecture name (e.g. 686/em64t/amd64). Use uname -r command to get actual kernel version name.

A more generic (recommend) and accurate way is as follows:

**# apt-get install kernel-headers-$(uname -r)**

All you need to do is change Makefile to use current kernel build directory.

$ **ls -d /lib/modules/$(uname -r)/build**

Sample output:

/lib/modules/2.6.27-7-generic/build

Create hello.c kernel module file:

#include <linux/module.h>

#include <linux/kernel.h>

int init\_module(void)

{

printk(KERN\_INFO "init\_module() called**\n**");

return0;

}

void cleanup\_module(void)

{

printk(KERN\_INFO "cleanup\_module() called**\n**");

}

Create a Makefile:

obj-m += hello.o

all:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules

 clean:

make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean

To build kernel module enter:  
$ make  
Sample output:

make -C /lib/modules/2.6.27-7-generic/build M=/tmp/test2 modules

make[1]: Entering directory `/usr/src/linux-headers-2.6.27-7-generic'

CC [M] /tmp/test2/hello.o

Building modules, stage 2.

MODPOST 1 modules

CC /tmp/test2/hello.mod.o

LD [M] /tmp/test2/hello.ko

make[1]: Leaving directory `/usr/src/linux-headers-2.6.27-7-generic'

**Run** ls command to see newly build kernel module:

$ls

Sample output:

hello.c hello.ko hello.mod.c hello.mod.o hello.o Makefile Module.markers modules.order Module.symvers

hello.ko is kernel module file. To see information about module, enter:  
$ modinfo hello.ko

Sample output:

filename: hello.ko

srcversion: 4F856ABA1F3290D5F81D961

depends:

vermagic: 2.6.27-7-generic SMP mod\_unload modversions 586

To load kernel module, enter:  
$ sudo insmod hello.ko  
OR  
$ sudo modprobe hello  
To list installed Linux kernel module, enter:  
$ lsmod  
$ lsmod | grep hello  
  
To remove hello Linux kernel module, enter:  
$ rmmod hello  
This module just logs message to a log file called /var/log/messages OR /var/log/syslog.

enter:  
$ tail -f /var/log/messages

Sample output:

Nov 5 00:36:36 vivek-desktop kernel: [52488.923000] init\_module() called

Nov 5 00:36:50 vivek-desktop kernel: [52503.065252] cleanup\_module() called

# UNDERSTANDING OF KERNEL

This is one the essential and important task. Many times we upgrade our kernel and some precompiled drivers won't work with Linux. Especially if you have weird hardware then vendor may send you, driver code aka C files to compile. Or even you can write your own Linux kernel driver. Compiling kernel driver is easy. Kernel 2.6.xx makes it even much easier. Following steps are required to compile driver as module: AND if you don't have a source code download it from [kernel.org](http://kernel.org/). Untar kernel source code (tar ball) in /usr/src using tar command:

$ tar -zxvf kernel\* -C /usr/src

Kernel headers are more than sufficient to compile kernel modules / drivers.

Example: hello.c module

1) hello.c C source code. Copy following code and save to hello.c  
$ mkdir demo; cd demo  
$ vi hello.c

2)Add following c source code to it:

#include <linux/module.h> /\* Needed by all modules \*/

#include <linux/kernel.h> /\* Needed for KERN\_INFO \*/

#include <linux/init.h> /\* Needed for the macros \*/

static int hello\_start(void) //if compiler not supported then write \_\_init hello\_start(void)

{

printk(KERN\_INFO "Loading hello module...\n");

printk(KERN\_INFO "Hello world\n");

return 0;

}

static void \_\_exit hello\_end(void)

{

printk(KERN\_INFO "Goodbye Mr.\n");

}

module\_init(hello\_start);

module\_exit(hello\_end);

This is an example modified for demonstration purpose.

3) Save the file. Create new Makefile as follows:  
$ vi Makefile

Append following make commands:

obj-m = hello.o

KVERSION = $(shell uname -r)

all:

make -C /lib/modules/$(KVERSION)/build M=$(PWD) modules

clean:

make -C /lib/modules/$(KVERSION)/build M=$(PWD) clean

4) Save and close the file.

5) Compile hello.c module:  
$ make

6) Become a root user (use su or sudo) and load the module:  
$ su -  
# insmod hello.ko

Note you can see message on screen if you are logged in as root

7) Verify that module loaded:  
# lsmod | less

8) See message in /var/log/message file:  
# tail -f /var/log/message

9) Unload the module:  
# rmmod hello

10) Load module when Linux system comes up. File /etc/modules use to load kernel boot time. This file should contain the names of kernel modules that are to be loaded at boot time, one per line. First copy your module to /lib/modules/$(uname -r)/kernel/drivers. Following are suggested steps:

(a) Create directory for hello module:  
# mkdir -p /lib/modules/$(uname -r)/kernel/drivers/hello  
(b) Copy module:  
# cp hello.ko /lib/modules/$(uname -r)/kernel/drivers/hello/  
(c) Edit /etc/modules file under Debian Linux:  
# vi /etc/modules  
(d) Add following line to it:  
hello  
(e) Reboot to see changes. Use lsmod or dmesg command to verify module loaded or not.  
# cat /proc/modules  
OR  
# lsmod | less

**part 2**

you can rename the init and cleanup functions of your modules; they no longer have to be called init\_module() and cleanup\_module() respectively.

This is done with module\_init() andmodule\_exit() macros. These macros are defined in linux/init.h. The only caveat is that your init and cleanup functions must be defined before calling the macros, otherwise you'll get compilation errors. Here's an example of this technique:

**hello-2.c**

|  |
| --- |
| /\*  \* hello-2.c - Demonstrating the module\_init() and module\_exit() macros.  \* This is preferred over using init\_module() and cleanup\_module().  \*/  #include <linux/module.h> /\* Needed by all modules \*/  #include <linux/kernel.h> /\* Needed for KERN\_INFO \*/  #include <linux/init.h> /\* Needed for the macros \*/  static int hello\_2\_init(void) //if compiler not supporting then \_\_init hello\_2\_init(void)  {  printk(KERN\_INFO "Hello, world 2\n");  return 0;  }  static void hello\_2\_exit(void) // if compiler not supporting then \_\_exit hello\_2\_exit(void)  {  printk(KERN\_INFO "Goodbye, world 2\n");  }  module\_init(hello\_2\_init);  module\_exit(hello\_2\_exit); |

So now we have two real kernel modules under our belt. Adding another module is as simple as this:

**Makefile for both our modules**

|  |
| --- |
| obj-m += hello-1.o  obj-m += hello-2.o  all:  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules  clean:  make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean |

**part 3: The \_\_init and \_\_exit Macros**

Notice the change in the definitions of the init and cleanup functions. The \_\_init macro causes the init function to be discarded and its memory freed once the init function finishes for built-in drivers, but not loadable modules. If you think about when the init function is invoked, this makes perfect sense.There is also an \_\_initdata which works similarly to \_\_init but for init variables rather than functions.

The \_\_exit macro causes the omission of the function when the module is built into the kernel, and like \_\_exit, has no effect for loadable modules. These macros are defined in linux/init.h and serve to free up kernel memory.

**Example 2-5. hello-3.c**

|  |
| --- |
| /\*  \* hello-3.c - Illustrating the \_\_init, \_\_initdata and \_\_exit macros.  \*/  #include <linux/module.h> /\* Needed by all modules \*/  #include <linux/kernel.h> /\* Needed for KERN\_INFO \*/  #include <linux/init.h> /\* Needed for the macros \*/  static int hello3\_data \_\_initdata = 3;  static int hello\_3\_init(void) //write \_\_init hello\_3\_init(void) if not supported  {  printk(KERN\_INFO "Hello, world %d\n", hello3\_data);  return 0;  }  static void hello\_3\_exit(void) //write \_\_exit hello\_3\_exit(void) if not supported  {  {  printk(KERN\_INFO "Goodbye, world 3\n");  }  module\_init(hello\_3\_init);  module\_exit(hello\_3\_exit); |

**part 4: Licensing and Module Documentation**

If you're running kernel 2.4 or later, when you loaded proprietary modules:

|  |
| --- |
| # insmod xxxxxx.o  Warning: loading xxxxxx.ko will taint the kernel: no license  See http://www.tux.org/lkml/#export-tainted for information about tainted modules  Module xxxxxx loaded, with warnings |

This license mechanism is defined and documented inlinux/module.h:

|  |
| --- |
| /\*  \* The following license idents are currently accepted as indicating free  \* software modules  \*  \* "GPL" [GNU Public License v2 or later]  \* "GPL v2" [GNU Public License v2]  \* "GPL and additional rights" [GNU Public License v2 rights and more]  \* "Dual BSD/GPL" [GNU Public License v2  \* or BSD license choice]  \* "Dual MIT/GPL" [GNU Public License v2  \* or MIT license choice]  \* "Dual MPL/GPL" [GNU Public License v2  \* or Mozilla license choice]  \*  \* The following other idents are available  \*  \* "Proprietary" [Non free products]  \*  \* There are dual licensed components, but when running with Linux it is the  \* GPL that is relevant so this is a non issue. Similarly LGPL linked with GPL  \* is a GPL combined work.  \*  \* This exists for several reasons  \* 1. So modinfo can show license info for users wanting to vet their setup  \* is free  \* 2. So the community can ignore bug reports including proprietary modules  \* 3. So vendors can do likewise based on their own policies  \*/ |

Similarly, MODULE\_DESCRIPTION() is used to describe what the module does, MODULE\_AUTHOR() declares the module's author, and MODULE\_SUPPORTED\_DEVICE() declares what types of devices the module supports.

**Example 2-6. hello-4.c**

|  |
| --- |
| /\*  \* hello-4.c - Demonstrates module documentation.  \*/  #include <linux/module.h> /\* Needed by all modules \*/  #include <linux/kernel.h> /\* Needed for KERN\_INFO \*/  #include <linux/init.h> /\* Needed for the macros \*/  #define DRIVER\_AUTHOR "Peter Jay Salzman <p@dirac.org>"  #define DRIVER\_DESC "A sample driver"  static int init\_hello\_4(void) //or write static int \_\_init init\_hello\_4(void)  {  printk(KERN\_INFO "Hello, world 4\n");  return 0;  }  static void cleanup\_hello\_4(void) //or write static void \_\_exit cleanup\_hello\_4(void)  {  printk(KERN\_INFO "Goodbye, world 4\n");  }  module\_init(init\_hello\_4);  module\_exit(cleanup\_hello\_4);  /\*  \* You can use strings, like this:  \*/  /\*  \* Get rid of taint message by declaring code as GPL.  \*/  MODULE\_LICENSE("GPL");  /\*  \* Or with defines, like this:  \*/  MODULE\_AUTHOR(DRIVER\_AUTHOR); /\* Who wrote this module? \*/  MODULE\_DESCRIPTION(DRIVER\_DESC); /\* What does this module do \*/  /\*  \* This module uses /dev/testdevice. The MODULE\_SUPPORTED\_DEVICE macro might  \* be used in the future to help automatic configuration of modules, but is  \* currently unused other than for documentation purposes.  \*/  MODULE\_SUPPORTED\_DEVICE("testdevice"); |

/\* ########################################################################################

\* Program Details

\*

\* Date : 15 march 2015

\*

\* Author : PROLIFIC SYTEMS & TECHNOLOGIES PVT LTD

\*

\* Purpose : sample code for program for cross compilation

\*

\* Compilation Option : \*\*\*\*\*\*-gcc gpioOperate.c -o gpioOperate (where \*\*\*\* is cross compiler prefix)

\*

\* Work to be done : Error handling to be done for return values

\*

\* #########################################################################################

\*/

#include<stdlib.h>

#include<stdio.h>

#include<string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#define MAX\_BUF 50

#define MAX\_GPIO 26

#define TRUE 1

#define FALSE 0

int fd;

char buf[MAX\_BUF];

int exportGpio(int gpio)

{

fd = open("/sys/class/gpio/export", O\_WRONLY);

sprintf(buf, "%d", gpio);

write(fd, buf, strlen(buf));

close(fd);

return 0;

}

int unexportGpio(int gpio)

{

fd = open("/sys/class/gpio/unexport", O\_WRONLY);

sprintf(buf, "%d", gpio);

write(fd, buf, strlen(buf));

close(fd);

return 0;

}

int getDirection(int gpio, int \*isDirectionIn)

{

char buff[5];

memset(buff,'\0',sizeof(buff));

sprintf(buf, "/sys/class/gpio/gpio%d/direction", gpio);

fd = open(buf, O\_RDONLY);

read(fd,buff,4);

printf("String = %s\n",buff);

if(0 == strncmp(buff,"in",2))

{

printf("IN Direction\n");

\*isDirectionIn = 0;

}

else

{

\*isDirectionIn = 1;

}

close(fd);

return 0;

}

int setDirection(int gpio, int isDirectionIn)

{

sprintf(buf, "/sys/class/gpio/gpio%d/direction", gpio);

fd = open(buf, O\_WRONLY);

if(isDirectionIn == TRUE)

{

// Set out direction

write(fd, "out", 3);

}else

{

// Set in direction

write(fd, "in", 2);

}

close(fd);

return 0;

}

int setValue(int gpio, int value)

{

sprintf(buf, "/sys/class/gpio/gpio%d/value", gpio);

fd = open(buf, O\_WRONLY);

if(value == TRUE)

{

// Set out direction

write(fd, "1" , 1);

}else

{

// Set in direction

write(fd, "0" , 1);

}

close(fd);

return 0;

}

//lseek(fp, 0, SEEK\_SET);

int getValue(int gpio, int \*value)

{

sprintf(buf, "/sys/class/gpio/gpio%d/value", gpio);

fd = open(buf, O\_WRONLY);

read(fd, value, 1);

close(fd);

return 0;

}

unsigned int printMenu()

{

unsigned int option;

printf("1. register gpio\n");

printf("2. free gpio\n");

printf("3. Set direction for gpio\n");

printf("4. Get Direction for gpio\n");

printf("5. Set Value for Gpio\n");

printf("6. Get value for Gpio\n");

printf("\nEnter Choice : \n");

scanf("%d",&option);

return option;

}

int main()

{

unsigned int gpio=0;

unsigned int option=0;

int value=FALSE;

while(1)

{

option=printMenu();

printf("Enter Gpio Number\n");

scanf("%d",&gpio);

switch(option)

{

case 5 :

{

printf("Enter Value to be set \n");

scanf("%d",&value);

setValue(gpio,value);

}

break;

case 6 :

{

getValue(gpio,&value);

printf("Value for Gpio %d = %d \n",gpio,value);

}

break;

case 1:

{

printf("Registering gpio = %d \n",gpio);

exportGpio(gpio);

break;

}

case 2 :

{

printf("Registering gpio = %d \n",gpio);

unexportGpio(gpio);

break;

}

case 3 :

{

printf("Enter Direction to be set 1 : out, 0 : in \n");

scanf("%d",&value);

setDirection(gpio,value);

break;

}

case 4 :

{

printf("Direction for the port is 1 : out, 0 : in \n");

getDirection(gpio,&value);

printf("Direction of the Port is %s \n",(value==1)?"OUT":"IN");

break;

}

}

value=0;

gpio=0;

}

return 0;

}