Introduction to Direct Hardware Access with Linux Device Drivers

Application Note 582

Logic PD // Products
Published: August 2013

Abstract

This document provides instructions about how to create and use a basic Linux device driver, and explains how the device driver interacts with the Linux operating system.

Revision History

REV	EDITOR	DESCRIPTION	APPROVAL	DATE
A	JA	-Initial Release	RAH, SO	08/08/13

Table of Contents

1 Introduction

Creating a device driver for Linux is a complex task. There are many examples that provide guidance; however, many of the examples are too complex and do not show how to access the hardware directly, which is critical for new projects with Logic PD SOMs.

The example driver provided here is partially based on code from the massively popular Linux Device Drivers, Third Edition[1] publication from O'Reilly Media. It's highly recommended that you begin reading this publication as you being to study Linux device drivers. It's well written, free, and will allow you to modify the simple example provided in this document into a specialized driver for your project.

1.1 Linux_Device_Drivers_Direct_Hardware_Access_Files.tar.gz Directory

Accompanying this application note within the *1024512A_AN582_Linux_Device_Drivers_Direct_Hardware_Access.zip* file is a directory containing software files to be used with these instructions. The *Linux_Device_Drivers_Direct_Hardware_Access_Files.tar.gz* directory should contain the following files that will be referenced throughout this document:

```
ProtoDriver-1.0.tar.gz
ProtoDriver-1.0.tar.gz.md5
protoDriver.spec
```

1.2 Overview

This presentation will cover an example device driver (a kernel module) and a user-mode application that can interact with it.

The kernel module has the following features:

- A minimalist structure of a character device driver
- Load-time device allocation
- Module parameters for dynamic configuration
- Demonstrates direct IO pin use
- Demonstrates modification of the pin mux registers
- Demonstrates the use of interrupts (hard and threaded)
- Demonstrates use of thread-safe data types
- Demonstrates dynamic debug zones

This list of features is enough for the creation of drivers in the prototyping environment.

1.3 Background

One of the critical concepts of the Unix/Linux operating system (OS) is that everything is a file. Device drivers are no different and they can be found in the */dev* directory where a file is created for each instance of a driver.

[1] http://lwn.net/Kernel/LDD3/

As is common with modern OSs, memory space is split and protected between the OS (the kernel) and user applications. This diagram shows the basic break down:

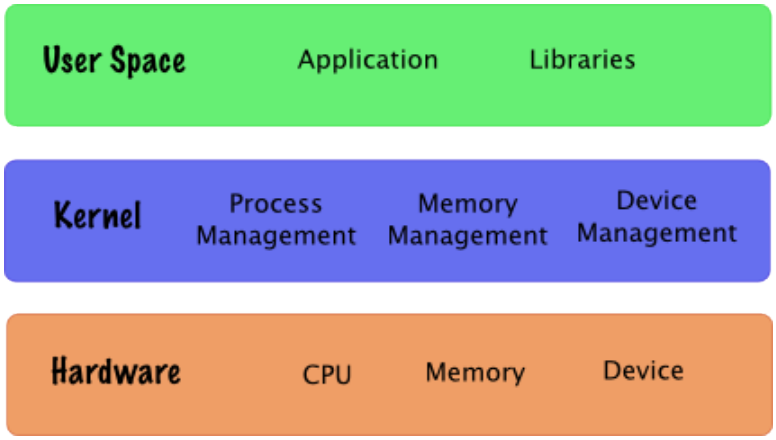

Figure 1.1: Memory Distribution

In order to access the hardware, a program in the user space must go through the kernel. Allowing direct access to the hardware from the user space is a risk to system stability and security. Directly accessing one memory area from the other will cause errors. Part of the purpose of this example is to show the interaction between a user space application and a driver running in the kernel space. The structure of this interaction between the application and the driver allows safe and deterministic access to the system hardware.

2 Load and Build the Source Code

To address the two most common development environments encountered when using Logic PD SOMs, we have provided installation instructions for both the DM3730/AM3703 Torpedo + Wireless SOM and the AM3517 SOM-M2, which use the Logic PD LTIB and the Texas Instruments (TI) PSP environments respectively.

2.1 Prerequisites

For both of the following examples, the Linux kernel must have already been successfully built on the most recent build.

1. First, copy the *Linux_Device_Drivers_Direct_Hardware_Access_Files.tar.gz* file into the *Downloads* directory.
2. Untar the file.

```
logic@logic-desktop-am3517:~/Downloads$ tar xf
Linux_Device_Drivers_Direct_Hardware_Access_Files.tar.gz
logic@logic-desktop-am3517:~/Downloads$
```

For the AM3517 TI PSP environment, a number of symbolic links are used in the path names to shorten them. These are set up as part of the build process provided in the *AM3517 Linux User Guide*.[2]

2.2 TI PSP Environment for AM3517 SOM-M2

In this section, the TI AM3517 05.05.00.00 PSP environment in the Logic PD Virtual Machine for the AM3517 Linux SDK[3] (hereafter, VM) will be used.

2.2.1 Load

1. Switch to the *Projects* directory.

```
logic@logic-desktop-am3517:~$ cd $HOME/TI_SDK/Projects
logic@logic-desktop-am3517:~/TI_SDK/Projects/TI_SDK/$
```

2. Copy the source code to the *Projects* directory.

```
logic@logic-desktop-am3517:~/TI_SDK/Projects$ cp
$HOME/Downloads/Linux_Device_Drivers_Direct_Hardware_Access_Files/ProtoDriv
er-1.0.tar.gz .
logic@logic-desktop-am3517:~/TI_SDK/Projects$
```

3. Unzip the source code and switch into it.

```
logic@logic-desktop-am3517:~/TI_SDK/Projects$ tar xf ProtoDriver-1.0.tar.gz
logic@logic-desktop-am3517:~/TI_SDK/Projects$ cd ProtoDriver-1.0/
logic@logic-desktop-am3517:~/TI_SDK/Projects/ProtoDriver-1.0$ ls
proto  protoUserApp
logic@logic-desktop-am3517:~/TI_SDK/Projects/ProtoDriver-1.0$
```

[2] http://support.logicpd.com/downloads/1434/
[3] http://support.logicpd.com/downloads/1564/

2.2.2 Build

1. Run the environment script.

```
logic@logic-desktop-am3517:~/TI_SDK/Projects/ProtoDriver-1.0$ source
$HOME/TI_SDK/linux-devkit/environment-setup
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0>
```

2. Switch to the kernel module code.

```
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0> cd proto
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/proto>
```

3. Clean the build.

```
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/proto> make clean
rm -rf *.o *~ core .depend .*.cmd *.ko *.mod.c .tmp_versions
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/proto>
```

4. Build the kernel module source.

```
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/proto> make ARCH=arm
CROSS_COMPILE=arm-arago-linux-gnueabi- KERNELDIR=$HOME/TI_SDK/KERNEL
echo CC: arm-arago-linux-gnueabi-gcc
CC: arm-arago-linux-gnueabi-gcc
make -C /home/logic/TI_SDK/KERNEL
M=/home/logic/TI_SDK/Projects/ProtoDriver-1.0/proto
LDDINC=/home/logic/TI_SDK/KERNEL/include  modules
make[1]: Entering directory `/home/logic/ti-sdk-am3517-evm-
05.05.00.00/board-support/linux-2.6.37-psp04.02.00.07.sdk'
  CC [M]  /home/logic/TI_SDK/Projects/ProtoDriver-1.0/proto/main.o
  LD [M]  /home/logic/TI_SDK/Projects/ProtoDriver-1.0/proto/proto.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/logic/TI_SDK/Projects/ProtoDriver-1.0/proto/proto.mod.o
  LD [M]  /home/logic/TI_SDK/Projects/ProtoDriver-1.0/proto/proto.ko
make[1]: Leaving directory `/home/logic/ti-sdk-am3517-evm-
05.05.00.00/board-support/linux-2.6.37-psp04.02.00.07.sdk'
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/proto>
```

5. Edit the module load file *proto_load* to select the IO pins for your platform.

   The file *proto_load* contains two assignments for the *hwOptions* variable, one of which is commented out. The first is for the AM3517 SOM-M2 and the second is for the DM3730/AM3703 Torpedo + Wireless SOM. The file should look like this:

```
...
#Here we perform the processor specific configuration of the driver.
#AM3517
hwOptions="requestedPinInterrupt=11 requestedPinInterruptAddr=0x48002A2 ...

#DM37_TORPEDO
#i2cset -f -y 1 0x49 0x0e 0x04
#hwOptions="requestedPinInterrupt=118 requestedPinInterruptAddr=0x48002 ...
```



```
...
```

6. Switch to the sample application source.

```
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/proto> cd ../protoUserApp/
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/protoUserApp>
```

7. Clean the build.

```
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/protoUserApp> make clean
rm -rf protoUserApp *.o *.out main
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/protoUserApp>
```

8. Build the sample application.

```
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/protoUserApp> make
protoUserApp
arm-arago-linux-gnueabi-gcc -Wl,--rpath=/lib:/usr/lib -Wl,--dynamic-
linker=/lib/ld-linux.so.3   -c -o main.o main.c
arm-arago-linux-gnueabi-gcc -Wl,--rpath=/lib:/usr/lib -Wl,--dynamic-
linker=/lib/ld-linux.so.3 main.o -o protoUserApp
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/protoUserApp>
```

2.2.3 Install

To use the files, you'll need to copy them into the target file system. The *home* directory is an acceptable place to try them out. This can be done via NFS or SCP.

2.2.3.1 NFS

1. Copy the user application.

```
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/protoUserApp> sudo cp
protoUserApp $HOME/TI_SDK/targetNFS/home/root/.
[sudo] password for logic:
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/protoUserApp>
```

   As you see above, the root password was needed to copy files into the NFS directory. The credentials are retained for a few minutes, so they may not be needed for the next commands.

2. Switch back to the kernel module directory.

```
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/protoUserApp> cd ../proto
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/proto>
```

3. Copy the kernel module files.

```
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/proto> make install
TARGET_DEST=$HOME/TI_SDK/targetNFS/home/root
sudo cp proto.ko /home/logic/TI_SDK/targetNFS/home/root/.
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/proto>
```


4. Copy the kernel module script files.

```
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/proto> make
install_scripts TARGET_DEST=$HOME/TI_SDK/targetNFS/home/root
sudo cp proto_load /home/logic/TI_SDK/targetNFS/home/root/.
sudo cp proto_unload /home/logic/TI_SDK/targetNFS/home/root/.
[linux-devkit]:~/TI_SDK/Projects/ProtoDriver-1.0/proto>
```

2.2.3.2 SCP

For this example to work, the target system must be on a network that is accessible from the development environment. You many need to install the SSH server on your host PC to use this method. See Appendix A for the SSH server installation steps.

If you are using the Logic PD VM and have not already installed SSH, do so by following the steps in Appendix A. Also, your target system needs to have SCP installed. For the TI PSP path, there are two root filesystems provided. The SCP utility is only installed in the *tisdk-rootfs-am3517-evm.tar.gz* image, but there are options if you need to use the *base-rootfs-am3517-evm.tar.gz* image.

NOTE: If you need SCP in the *base-rootfs-am3517-evm.tar.gz* image, you can transfer it by unpacking *base-rootfs-am3517-evm.tar.gz* on the host PC, adding the SCP utility and its libraries, and then repacking it.

1. Determine the IP address of your host PC. Look for the *eth0* or *eth1* adaptor; this is highlighted in the example below. Copy the address from the *inet addr* field, which includes X.X.X.X. as a placeholder for the IP address below.

```
[linux-devkit]:~> ifconfig
eth1      Link encap:Ethernet  HWaddr --:--:--:--:--:--
          inet addr:X.X.X.X  Bcast:--.--.--.---  Mask:255.255.255.0
          inet6 addr: ----::----:----:----:----/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:261418 errors:0 dropped:0 overruns:0 frame:0
          TX packets:110319 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:131598602 (131.5 MB)  TX bytes:162267020 (162.2 MB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:16436  Metric:1
          RX packets:112 errors:0 dropped:0 overruns:0 frame:0
          TX packets:112 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:9680 (9.6 KB)  TX bytes:9680 (9.6 KB)

[linux-devkit]:~>
```


2. Identify the path of the example files. This will be different for PSP and LTIB development setups:

   □ For the AM3517 PSP:

```
root@am3517-evm:~# export
EX_BASE_PATH=/home/logic/TI_SDK/Projects/ProtoDriver-1.0
root@am3517-evm:~#
```

   □ For the DM3730/AM3703 LTIB:

```
DM-37x# export EX_BASE_PATH=/home/logic/logic/Logic_BSPs/Linux_3.0/10
22853_LogicPD_Linux_BSP_2.2-2/rpm/BUILD/ProtoDriver-1.0
DM-37x#
```

3. On the target, switch to the home directory.

```
root@am3517-evm:~# cd
root@am3517-evm:~#
```

4. Use the *SCP* command to copy the module file. Be sure to replace the X.X.X.X in the command below with the IP address from Step 1 above.

```
root@am3517-evm:~# scp logic@X.X.X.X:" \
$EX_BASE_PATH/proto/proto.ko  \
$EX_BASE_PATH/proto/proto_load  \
$EX_BASE_PATH/proto/proto_unload"  \
.

Host 'X.X.X.X' is not in the trusted hosts file.
(fingerprint md5 --:--:--:--:--:--:--:--:--:--:--:--:--:--:--:--)
Do you want to continue connecting? (y/n) yes
logic@X.X.X.X's password:
proto.ko                                      100%   89KB  88.7KB/s   00:00
proto_load                                    100% 1005     1.0KB/s   00:00
proto_unload                                  100%  181     0.2KB/s   00:00
root@am3517-evm:~#
```

In the example above, the host PC was not yet listed as a trusted host. You may answer yes to the question to add it when prompted.

5. Use the *SCP* command to copy the user application files. Again, be sure to replace the X.X.X.X in the command below with the IP address from Step 1 above.

```
root@am3517-evm:~# scp
logic@X.X.X.X:$EX_BASE_PATH/protoUserApp/protoUserApp .
logic@X.X.X.X's password:
protoUserApp                                  100%   46KB  46.4KB/s   00:00
root@am3517-evm:~#
```


2.3 Logic PD LTIB Environment for DM3730/AM3703 Torpedo + Wireless SOM

2.3.1 Load

1. If it doesn't already exist, create a symbolic link in the *home* directory to shorten the path to the LTIB environment.

```
logic@logic-desktop:~$ cd
logic@logic-desktop:~$ ln -s
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-2
ltib_3_0
logic@logic-desktop:~$
```

2. Switch to the LTIB environment.

```
logic@logic-desktop:~$ cd ltib_3_0
logic@logic-desktop:~/ltib_3_0$
```

3. From the *Downloads* directory, copy the source code tar file and its accompanying MD5sum file into the LTIB package pool.

```
logic@logic-desktop:~/ltib_3_0$ cp
$HOME/Downloads/Linux_Device_Drivers_Direct_Hardware_Access_Files/ProtoDriv
er-1.0.tar.gz* LTIB-package-pool/.
logic@logic-desktop:~/ltib_3_0$
```

4. Create the directory in the distribution directory for the spec file.

```
logic@logic-desktop:~/ltib_3_0$ mkdir -p dist/lfs-5.1/protoDriver
logic@logic-desktop:~/ltib_3_0$
```

5. Copy the spec file into the distribution directory.

```
logic@logic-desktop:~/ltib_3_0$ cp
$HOME/Downloads/Linux_Device_Drivers_Direct_Hardware_Access_Files/protoDriv
er.spec dist/lfs-5.1/protoDriver/.
logic@logic-desktop:~/ltib_3_0$
```

2.3.2 Build

1. Command LTIB to prep the source.

```
logic@logic-desktop:~/ltib_3_0$ ./ltib -m prep -p protoDriver.spec

Processing: protoDriver
========================
Build path taken because: build key set, no prebuilt rpm,
clobber 0 dir_bld  unpack yes spec_upd 0

rpmbuild --dbpath
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
```



```
2/rootfs//var/lib/rpm --target arm --define
'_unpackaged_files_terminate_build 0' --define '_target_cpu arm' --define
'__strip strip' --define '_topdir
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-2/rpm'
--define '_prefix /usr' --define '_host_prefix /opt/ltib' --define
'_tmppath
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-2/tmp'
--define '_rpmdir
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/RPMS'  --define '_mandir /usr/share/man' --define '_sysconfdir /etc'
--define '_localstatedir /var' -bp
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/dist/lfs-5.1/protoDriver/protoDriver.spec
Building target platforms: arm
Building for target arm
Executing(%prep): /bin/sh -e
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/tmp/rpm-tmp.82298
+ umask 022
+ cd /home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD
+ cd /home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD
+ rm -rf ProtoDriver-1.0
+ /bin/gzip -dc
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/SOURCES/ProtoDriver-1.0.tar.gz
+ tar -xvvf -
drwxr-xr-x logic/logic       0 2012-11-20 15:35 ProtoDriver-1.0/
drwxr-xr-x logic/logic       0 2012-11-26 14:59 ProtoDriver-
1.0/protoUserApp/
-rw-r--r-- logic/logic     266 2012-11-26 14:59 ProtoDriver-
1.0/protoUserApp/Makefile
-rw-r--r-- logic/logic     443 2012-11-20 15:35 ProtoDriver-
1.0/protoUserApp/main.h
-rw-r--r-- logic/logic    2996 2012-11-20 15:35 ProtoDriver-
1.0/protoUserApp/main.c
drwxr-xr-x logic/logic       0 2012-11-26 14:59 ProtoDriver-1.0/proto/
-rw-r--r-- logic/logic    1005 2012-11-20 15:34 ProtoDriver-
1.0/proto/proto_load
-rw-r--r-- logic/logic     466 2012-11-20 15:34 ProtoDriver-
1.0/proto/Makefile
-rw-r--r-- logic/logic     701 2012-11-20 15:34 ProtoDriver-
1.0/proto/proto_ioctl.h
-rw-r--r-- logic/logic    2832 2012-11-20 15:34 ProtoDriver-
1.0/proto/proto.h
-rw-r--r-- logic/logic     181 2012-11-20 15:34 ProtoDriver-
1.0/proto/proto_unload
-rw-r--r-- logic/logic   12288 2012-11-20 15:34 ProtoDriver-
1.0/proto/.proto.h.swp
-rw-r--r-- logic/logic       0 2012-11-20 15:34 ProtoDriver-
1.0/proto/Module.symvers
-rw-r--r-- logic/logic      86 2012-11-20 15:34 ProtoDriver-
1.0/proto/modules.order
-rw-r--r-- logic/logic   12712 2012-11-20 15:34 ProtoDriver-
1.0/proto/main.c
```



```
-rw-r--r-- logic/logic   32768 2012-11-20 15:34 ProtoDriver-
1.0/proto/.main.c.swp
+ STATUS=0
+ '[' 0 -ne 0 ']'
+ cd ProtoDriver-1.0
+ exit 0
Build time for protoDriver: 0 seconds

logic@logic-desktop:~/ltib_3_0$
```

2. Command LTIB to build the source.

```
logic@logic-desktop:~/ltib_3_0$ ./ltib -m scbuild -p protoDriver.spec

Processing: protoDriver
========================
Build path taken because: directory build, build key set, no prebuilt rpm,

rpmbuild --dbpath
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rootfs//var/lib/rpm --target arm --define
'_unpackaged_files_terminate_build 0' --define '_target_cpu arm' --define
'__strip strip' --define '_topdir
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-2/rpm'
--define '_prefix /usr' --define '_host_prefix /opt/ltib' --define
'_tmppath
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-2/tmp'
--define '_rpmdir
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/RPMS'  --define '_mandir /usr/share/man' --define '_sysconfdir /etc'
--define '_localstatedir /var' -bc --short-circuit
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/dist/lfs-5.1/protoDriver/protoDriver.spec
Building target platforms: arm
Building for target arm
Executing(%build): /bin/sh -e
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/tmp/rpm-tmp.69759
+ umask 022
+ cd /home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD
+ cd ProtoDriver-1.0
+
KSRC_DIR=/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2
.2-2/rpm/BUILD/linux
++ eval echo
+++ echo
+ KBOUT=
+
KBOUT=/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD/linux
+
CFG_PATH=/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2
.2-2/rpm/BUILD/linux/.config
```



```
+ '[' arm = ppc -a -f
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD/linux/arch/powerpc/Kconfig ']'
+ '[' '!' -f
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD/linux/.config ']'
+ cd proto
+ make
KERNELDIR=/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_
2.2-2/rpm/BUILD/linux ARCH=arm
echo CC: gcc
CC: gcc
make -C
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD/linux
M=/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD/ProtoDriver-1.0/proto
LDDINC=/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2
-2/rpm/BUILD/linux/include  modules
make[1]: Entering directory
`/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD/linux-3.0'
  CC [M]
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD/ProtoDriver-1.0/proto/main.o
  LD [M]
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD/ProtoDriver-1.0/proto/proto.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD/ProtoDriver-1.0/proto/proto.mod.o
  LD [M]
/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD/ProtoDriver-1.0/proto/proto.ko
make[1]: Leaving directory
`/home/logic/logic/Logic_BSPs/Linux_3.0/1022853_LogicPD_Linux_BSP_2.2-
2/rpm/BUILD/linux-3.0'
+ cd ../protoUserApp
+ make protoUserApp
gcc -Wl,--rpath=/lib:/usr/lib -Wl,--dynamic-linker=/lib/ld-linux.so.3   -c
-o main.o main.c
gcc -Wl,--rpath=/lib:/usr/lib -Wl,--dynamic-linker=/lib/ld-linux.so.3
main.o -o protoUserApp
+ cd ..
+ exit 0
Build time for protoDriver: 3 seconds

logic@logic-desktop:~/ltib_3_0$
```


3. Edit the module load file to select the IO pins for your platform.

   The file *…/rpm/BUILD/ProtoDriver-1.0/proto/proto_load* contains two assignments for the variable *hwOptions*, one of which is commented out. The first is for the AM3517 SOM-M2 and the second is for the DM3730/AM3703 Torpedo + Wireless SOM. The file should look similar to the example below.

```
...
#Here we perform the processor specific configuration of the driver.
#AM3517
#hwOptions="requestedPinInterrupt=11 requestedPinInterruptAddr=0x48002A2
...

#DM37_TORPEDO
i2cset -f -y 1 0x49 0x0e 0x04
hwOptions="requestedPinInterrupt=118 requestedPinInterruptAddr=0x48002 ...
...
```

There's a line unique to the DM37_TORPEDO section which executes an *i2cset* command. On the DM3730/AM3703 Torpedo + Wireless SOM, the interrupt pin used in the examples is part of the audio interface on the PMIC. This command sends a message to the PMIC via I2C that frees that pin for use. It sends the value 0x04 to register 0x0e in control group 0x49. This sets the AIF_TRI_EN bit in in the AUDIO_IF register. For additional information about this, please see the TI *TPS65950 Data Manual*[4].

2.3.3 Install

To use the files, you'll need to copy them into the target filesystem. Since the LTIB environment doesn't come with NFS set up by default, perhaps the easiest way to do this is to SCP the files into the target filesystem once it has finished booting. Please see Section 2.2.3.2 for instructions about how to do this.

[4] http://www.ti.com/product/tps65950

3 Run Demonstration

The example in this section will use the AM3517 Development Kit to explain how to run the demonstration. However, the procedures also apply to the DM3730 Torpedo Development kit unless otherwise specified.

3.1 Connect Multimeter

1. Power down the development kit.
2. Connect the ground lead of the multimeter.
   - □ For the AM3517 Development Kit, J33 is a good choice.
   - □ For the DM3730 Torpedo Development Kit, J37 is a good choice.
3. Connect the positive lead to the “pulse” pin specified in the load script.
   - □ For the AM3517 Development Kit, this will be J39.14.

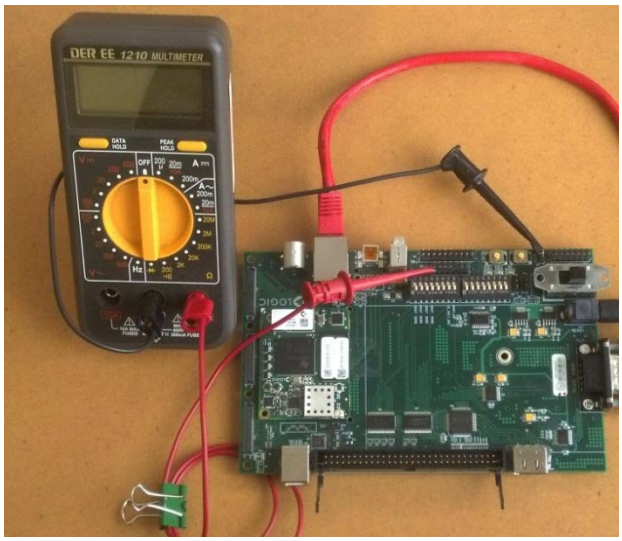

Figure 3.1: AM3517 Development Kit Connected to Multimeter

   - □ For the DM3730 Torpedo Development Kit, this will be J33.2.

Figure 3.2: DM3730 Torpedo Development Kit Connected to Multimeter

3.2 Load Module

1. Mark the scripts as executable.

```
root@am3517-evm:~# chmod +x proto_*
root@am3517-evm:~#
```

2. Run the script to load the module.

   - On the AM3517 Development Kit:

```
root@am3517-evm:~# ./proto_load
[  126.492828] proto  proto_init_module started...
[  126.497924] proto Pulse Pin:     31
[  126.560150] proto Interrupt Pin: 11
[  126.563842] proto IRQ num:       171
[  126.567687] proto  proto_init_module ...complete.
root@am3517-evm:~#
```

   - On the DM3730 Torpedo Development Kit:

```
DM-37x# ./proto_load
[186102.745635] proto  proto_init_module started...
[186102.750732] proto Pulse Pin:     161
[186102.810577] proto Interrupt Pin: 118
[186102.814422] proto IRQ num:       278
[186102.818420] proto  proto_init_module ...complete.
DM-37x#
```

3.3 Run User Application

The user application uses IOCTL (“eye-awk-tol”) calls into the kernel module to request the actions that are provided as an example. The IOCTL call is a way of sending a direct command to a module rather than going through the standard read/write interface. These calls are indexed and the user application example uses a single parameter to select which command it will send to the module. The command indexes are:

- 0 – Reset the interrupt counters
- 1 – Set the IO pin high
- 2 – Set the IO pin low
- 3 – Read the state of the IO pin
- 4 – Show the interrupt counter values
- 5 – Enable the soft (threaded) interrupts
- 6 – Disable the soft (threaded) interrupts

The following steps will illustrate each of these commands and how they interact with each other.

1. Check the initial interrupt count.

```
root@am3517-evm:~# ./protoUserApp 4
Prototype User Application for the 'proto' prototype device driver.
Interrupts> Hard/Soft: 0/0
root@am3517-evm:~#
```


2. Set the pin high.

```
root@am3517-evm:~# ./protoUserApp 1
Prototype User Application for the 'proto' prototype device driver.
root@am3517-evm:~#
```

3. Verify the output on the multimeter.
   - On the AM3517 Development Kit, you should see approximately 3.3Vmeasured on the multimeter.
   - On the DM3730 Torpedo Development Kit, you should see approximately 1.8V on the multimeter.
4. Check the state of the input pin.

```
root@am3517-evm:~# ./protoUserApp 3
Prototype User Application for the 'proto' prototype device driver.
Pin state: 1
root@am3517-evm:~#
```

Due to the internal pull-up, the state of the input pin should be 1, as seen above.

5. Set the pin low.

```
root@am3517-evm:~# ./protoUserApp 2
Prototype User Application for the 'proto' prototype device driver.
root@am3517-evm:~#
```

6. Verify the output on the multimeter. You should see approximately 0 volts measured on the multimeter. You may see an actual reading of 0.00 - 0.05 volts.
7. Check the state of the input pin.

```
root@am3517-evm:~# ./protoUserApp 3
Prototype User Application for the 'proto' prototype device driver.
Pin state: 1
```

Since there is no electrical connection to the input pin, it will still read as 1.

8. Recheck the interrupt count.

```
root@am3517-evm:~# ./protoUserApp 4
Prototype User Application for the 'proto' prototype device driver.
Interrupts> Hard/Soft: 0/0
root@am3517-evm:~#
```

3.4 Unload Module

Run the script to unload the module.

```
root@am3517-evm:~# ./proto_unload
[  831.908966] proto  Started...
[  831.913543] proto  ...complete.
```



```
root@am3517-evm:~#
```

3.5 Connect Feedback Jumper

The feedback jumper will connect the pulse pin which we control to the input pin. We will be able to read the state of the pulse pin with the input pin and low-going edges will trigger an interrupt. You should be able to connect the jumper with your multimeter still connected.

- For the AM3517 Development Kit, connect J39.14 to J39.13.

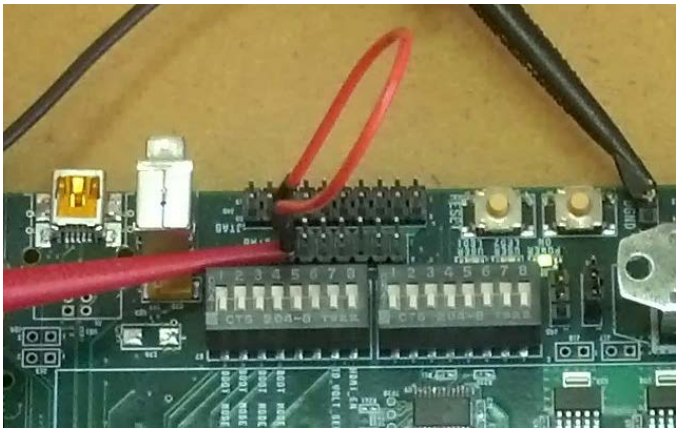

Figure 3.3: Connect J39.14 to J39.13 on AM3517 Development Kit

- For the DM3730 Torpedo Development Kit, connect J33.2 to J33.4.

Figure 3.4: Connect J33.2 to J33.4 on DM3730 Torpedo Development Kit

3.6 Reload Module

Run the script to load the module.

- On the AM3517 Development Kit:

```
root@am3517-evm:~# ./proto_load
[ 2234.423980] proto  proto_init_module started...
[ 2234.429107] proto Pulse Pin:     31
[ 2234.489471] proto Interrupt Pin: 11
[ 2234.493133] proto IRQ num:       171
```



```
[ 2234.496978] proto  proto_init_module ...complete.
root@am3517-evm:~#
```

- On the DM3730 Torpedo Development Kit:

```
DM-37x# ./proto_load
[186102.745635] proto  proto_init_module started...
[186102.750732] proto Pulse Pin:     161
[186102.810577] proto Interrupt Pin: 118
[186102.814422] proto IRQ num:       278
[186102.818420] proto  proto_init_module ...complete.
DM-37x#
```

3.7 Rerun User Application Feedback

1. Check the initial interrupt count.

```
root@am3517-evm:~# ./protoUserApp 4
Prototype User Application for the 'proto' prototype device driver.
Interrupts> Hard/Soft: 0/0
root@am3517-evm:~#
```

2. Set the pin high.

```
root@am3517-evm:~# ./protoUserApp 1
Prototype User Application for the 'proto' prototype device driver.
root@am3517-evm:~#
```

3. Verify the output on the multimeter.
   - On the AM3517 Development Kit, you should see approximately 3.3V measured on the multimeter.
   - On the DM3730 Torpedo Development Kit, you should see approximately 1.8V on the multimeter.
4. Check the state of the input pin. It should be high.

```
root@am3517-evm:~# ./protoUserApp 3
Prototype User Application for the 'proto' prototype device driver.
Pin state: 1
root@am3517-evm:~#
```

5. Set the pin low.

```
root@am3517-evm:~# ./protoUserApp 2
Prototype User Application for the 'proto' prototype device driver.
root@am3517-evm:~#
```

6. Verify the output on the multimeter.

7. Check the state of the input pin. It should be low.

```
root@am3517-evm:~# ./protoUserApp 3
Prototype User Application for the 'proto' prototype device driver.
Pin state: 0
root@am3517-evm:~#
```

8. Check the interrupt count. You should now see a single hard interrupt.

```
root@am3517-evm:~# ./protoUserApp 4
Prototype User Application for the 'proto' prototype device driver.
Interrupts> Hard/Soft: 1/0
root@am3517-evm:~#
```

3.8 Enable Threaded Interrupt

1. Enable the threaded interrupt.

```
root@am3517-evm:~# ./protoUserApp 5
Prototype User Application for the 'proto' prototype device driver.
root@am3517-evm:~#
```

2. Set the pin high.

```
root@am3517-evm:~# ./protoUserApp 1
Prototype User Application for the 'proto' prototype device driver.
root@am3517-evm:~#
```

3. Set the pin low.

```
root@am3517-evm:~# ./protoUserApp 2
Prototype User Application for the 'proto' prototype device driver.
root@am3517-evm:~#
```

4. Check the interrupt count. You should now see two hard interrupts and one threaded interrupt.

```
root@am3517-evm:~# ./protoUserApp 4
Prototype User Application for the 'proto' prototype device driver.
Interrupts> Hard/Soft: 2/1
root@am3517-evm:~#
```

3.9 Clear Interrupt Count

1. Clear the interrupt count.

```
root@am3517-evm:~# ./protoUserApp 0
Prototype User Application for the 'proto' prototype device driver.
root@am3517-evm:~#
```


2. Check the interrupt count. It should be 0/0.

```
root@am3517-evm:~# ./protoUserApp 4
Prototype User Application for the 'proto' prototype device driver.
Interrupts> Hard/Soft: 0/0
root@am3517-evm:~#
```

3.10 Change Debug Message Zones

1. Check the current state of the debug zone module parameter.

```
root@am3517-evm:~# cat /sys/module/proto/parameters/proto_debugLevel
13
root@am3517-evm:~#
```

2. Check the source to find the bit field for the debug zones. They are defined in *…/ProtoDriver-1.0/proto/proto.h*. The Enum is named PROTO_DEBUG_ZONES. Choose the IOCTL zone, which is bit 4.
3. Modify the debug zone module parameter to enable the IOCTL zone.

   This is a trick to let bash do the bit manipulation for you. We can combine the current value with bit 4, which will enable the IOCTL debug zone.

```
root@am3517-evm:~# echo $((13+(1<<4))) >
/sys/module/proto/parameters/proto_debugLevel
root@am3517-evm:~#
```

   Here you see we are shifting 1 by four bit places and adding it to the current value of 13. Bit 4 is 16.

4. Rechecking the value, we now see 13+16=29 as expected.

```
root@am3517-evm:~# cat /sys/module/proto/parameters/proto_debugLevel
29
root@am3517-evm:~#
```

   This seems trivial now, but in larger drivers with larger bit fields, this is a time saver, especially when activating multiple zones.

5. Set the pin low.

```
root@am3517-evm:~# ./protoUserApp 2
Prototype User A[ 3814.823242] proto proto_ioctl
pplication for t[ 3814.828643] proto proto_ioctl IOCTL Set pin LOW.
he 'proto' prototype device driver.
root@am3517-evm:~#
```

   There are several things to notice here. Before, the command to set the pin state had no output; we now see two kernel *printk* statements from our module:

```
[ 3814.823242] proto proto_ioctl
[ 3814.828643] proto proto_ioctl IOCTL Set pin LOW.
```


The *printk* statement is one of the most commonly used debug methods. It is the kernel-space equivalent of adding a *printf* statement to debug a user-space application. The value in square brackets is the number of seconds since the kernel booted.

Notice how they cut into the normal output text of the *protoUserApp* command output. These messages are printed by different threads and the thread for the *printk* statements has a higher priority.

The messages tell us first that the *proto_ioctl* function of the module was called. The second message comes from the code that handles setting the output pins IO state. It received a command to set the pin low. You can see these *printk* statements in *… /ProtoDriver-1.0/proto/main.c* in the *proto_ioctl* function.

4 A Tour of the Source

Now that we have seen the *protoUserApp* and *proto.ko* commands in action, let's walk through some of the highlights in the source code. To prevent this document from being excessively long, please read the first three chapters of the *Linux Device Drivers, Third Edition*[5] publication to understand the boiler plate portions of this code.

4.1 Load Module

4.1.1 Call Script

The process of actually loading a module into the kernel can be quite simple if there are no options being specified. In our case, we have a more complicated process, so we abstracted all of the steps into a script call *proto_load*.

1. To use the script, simply execute it.

```
root@am3517-evm:~# ./proto_load
[   72.094573] proto  proto_init_module started...
[   72.099426] proto Pulse Pin:     31
[   72.164093] proto Interrupt Pin: 11
[   72.167755] proto IRQ num:       171
[   72.171600] proto  proto_init_module ...complete.
root@am3517-evm:~#
```

   The output messages come from the module itself and tell us which IO pins were used and which IRQ was allocated.

2. We can verify that the module was loaded with the *lsmod* command. This tells us the size of the module and what other models depend on it.

```
root@am3517-evm:~# lsmod
Module                  Size  Used by
...
proto                   6064  0
...
root@am3517-evm:~#
```

3. This module creates two entries in the device (*/dev*) directory. We can look at that by listing that directory along with the '-il' flags to show us a list of the entries along with the major/minor numbers.

```
root@am3517-evm:~# ls -il /dev
   2365 drwxr-xr-x    2 root     root          620 Jan  1  2000 block
   2043 drwxr-xr-x    3 root     root          60 Jan  1  2000 bus
   ...
   3361 lrwxrwxrwx    1 root     root          Aug 24 10:09 proto -> proto0
   3360 crw-rw-r--    1 root     staff    251,   0 Aug 24 10:09 proto0
   ...
   2127 crw-------    1 root     root      10, 130 Jan  1  2000 watchdog
   2319 crw-rw-rw-    1 root     root       1,   5 Jan  1  2000 zero
root@am3517-evm:~#
```

[5] http://lwn.net/Kernel/LDD3/

In the output, we can see that our module created */dev/proto0* and a symbolic link to it called */dev/proto*. It was assigned a major number of 251.

4. We can see this major number listed in the */proc/device* file.

```
root@am3517-evm:/proc# cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
  4 tty
  4 ttyS
  5 /dev/tty
  ...
  251 proto
  ...
```

5. There is also a directory created in the *sys* folder for the module.

```
root@am3517-evm:~# ls /sys/module/proto/
holders     notes       refcnt      srcversion
initstate   parameters  sections
root@am3517-evm:~#
```

This directory contains a great deal of information about how the module was loaded. The "parameters" section in particular can modify an already running driver as we have already seen.

4.1.2 Examine Script

This section will take a closer look at the action of the script. The most important part for the example is the *hwOptions* variable. This is passed into the driver when it loads to specify which IO pins we will use and which pin mux registers configure them.

```
hwOptions="requestedPinInterrupt=11 requestedPinInterruptAddr=0x48002A24...
```

Next, the *insmod* command is called to insert the module into the kernel with the options in the *hwOptions* variable.

```
/sbin/insmod ./$module.ko $hwOptions || exit 1
```

Finally, the *mknod* command creates the entries in the */dev* directory so that the interface to the module is made available to the system.

```
mknod /dev/${device}0 c $major 0
```

The *mknod* command is preceded and followed by some commands that do some housekeeping. The *rm* command removes any old entries that may have been abandoned. The *ln* command creates a symbolic link to a non-indexed entry. The *chgrp* and *chmod* commands set the permissions and group membership of the entry.

4.2 Module Source

Aside from the scripts, the body of the source code is in the *proto/main.c* and *proto/proto.h* files.

4.2.1 Driver Instance Structure.

The code and executable for a driver is referred to as a module. Instances of that code are referred to as device drivers. In the *proto/proto.h* header file, there is a structure that contains everything an instance of the device driver will need to know while it is running.

```
struct proto_dev {
        int             pinPulse;
        int             pinInterrupt;
        int             irqNum;
        atomic_t        interruptCountHard;
        atomic_t        interruptCountSoft;
        bool            softInterruptEnabled;
        struct cdev cdev;         /* Char device structure              */

        int             pinPulseSavedMux;
        int             pinInterruptSavedMux;
};
```

Below is a brief explanation of each entry:

- **pinPulse:** Index of the output pin
- **pinInterrupt:** Index of the input pin
- **irqNum:** Stores the IRQ that corresponds to the input pin
- **interruptCountHard:** A count of hard interrupts received
- **interruptCountSoft:** A count of the threaded (soft) interrupts received
- **softInterruptEnabled:** Controls if threaded interrupts are triggered by hard interrupts
- **Cdev:** Required structure for implementing a character driver/module
- **pinPulseSavedMux:** The saved state of the output pin's pin mux register
- **pinInterruptSavedMux:** The saved state of the input pin's pin mux register

4.2.2 *module_init* and *module_exit* Macros

The command that loads the module needs to know what code to call when loading and unloading the module. The *module_init* and *module_exit* macros identify these functions:

```
module_init(proto_init_module);
module_exit(proto_cleanup_module);
```

4.2.3 Device Operations Structure

This structure is passed to the kernel when the module is loaded to define the common API for the module. You can find it in *main.c.*

```
struct file_operations proto_fops = {
        .owner =    THIS_MODULE,
        .llseek =   NULL,
        .read =     NULL,
        .write =    NULL,
        .unlocked_ioctl =    proto_ioctl,
```



```
        .open =     proto_open,
        .release =  proto_release,
};
```

In the example above, the *read/write/seek* calls are not being used. Only the *ioctl* call is used and the *proto_ioctl* function is called.

4.2.4 IO Pin Setup

In the *proto_setup_cdev()* function, you will find the code to configure the IO pins for this example. Below, we will review the process in abstract.

First, the registers that control the pin multiplexer must be adjusted to put the pin in the proper mode. This is done with the *proto_change_mux()* function. In this function, you see that a call is made to *ioremap()* to map the physical register space into virtual memory so it can be accessed. The offset into that virtual address spaces is calculated and the functions *ioread16()* and *iowrite16()* are used to save the current value and write the desired value. The current value is saved so it can be restored when the module is unloaded from memory. This is not required but it is highly recommended as good kernel etiquette.

Next, the pin must be allocated for use. This is done with the *gpio_request_one()* function. The third parameter to this function is a string that is used to identify who requested the IO pin. It is good to add text with this format: <module name><use>. The pin also needs to be released when the module is unloaded by calling the *gpio_free()* function. You can use the debug filesystem (*debugfs*) to view these allocations from the command line. The *gpio* file contains the listing of the pins and the text labels that were used at allocation. Wherever the debug file system is mounted, the file is found at *…/debug/gpio*.

Finally, if the pin needs to be used as an output, it must be specified by calling the *gpio_direction_output()* function and by providing an initial state.

4.2.5 Interrupt Setup

If an IO pin is going to be used as an interrupt, two additional functions must be called to configure it. The *OMAP_GPIO_IRQ()* function will return the processors IRQ number for the pin being set up. This value is then used in the interrupt configuration call.

In this example, the interrupt configuration is done with a call to *request_threaded_irq()*. This sets up a hard and threaded interrupt at the same time.

```
irqflags = IRQF_TRIGGER_FALLING;
result  = request_threaded_irq( dev->irqNum,
                                proto_hardirq,
                                proto_irq,
                                irqflags,
                                DRIVER_NAME_BASE,
                                dev);
```

Hard interrupts are generated first and are where any time-sensitive code can be run. This would include tasks such as setting acknowledge signals or reading a few bytes from a First In, First Out (FIFO). In order not to bog down the kernel's interrupt handler, any additional work is handled by the threaded interrupt handler, which will only run if it is requested. The last parameter of the function is a pointer to the driver instance structure.

4.2.6 Move Data between Kernel and User Space

An important point to remember with Linux drivers is that the memory space for kernel operations and for user operations are kept separate. There are a set of functions that move data between the two memory spaces in a secure and deterministic way. The *__get_user()* and *__put_user* macros will move a single variable, like an integer or a Boolean. The *copy_to_user* and *copy_from_user* functions will move a set number of bytes to or from a specific address. The latter two are used for more complex data structures or buffers. Examples of these functions in action can be found in the IOCTL handling code in *proto_ioctl().*

```
retval = __get_user(pinVal, (int __user *)arg);
if ((0 == retval) && ((1 == pinVal) || (0 == pinVal)))
{
proto_debug(DEBUG_IOCTL, "%s IOCTL Set pin %s.", __func__....
gpio_set_value(dev->pinPulse, pinVal);
}
```

In the example above, we see a call *to __get_user*. Recall that *__get_user* is a macro, so it handles the pointer setup to access its first parameter, which is the destination for the value from a user space. The second parameter casts the third formal parameter to *proto_ioctl* as a pointer to a user space (__user) integer. If the return value is 0, then the transfer was successful and the value in pinVal is valid.

4.2.7 Interrupts

The interrupts in this example are handled by two functions: *proto_hardirq()* and *proto_irq()*. The first is called immediately following the hardware event that triggers the interrupt by the kernel's interrupt processor. To avoid bogging that thread down, only the bare minimum of processing is done in the hard interrupt handler. For example, this is where data could be read from a buffer. When enough data has been read from the buffer, that data can be processed by a separate thread with less priority. This additional processing is requested by the return value from the hard interrupt handler. In the function *proto_hardirq()*, you can see that there are two possible return values shown: IRQ_HANDLED and IRQ_WAKE_THREAD. The latter signals to the kernel that additional work is required and it will release the thread that will run the *proto_irq()* function.

4.2.8 Debug Zones

All throughout the code, debug messages are present in the form of calls to functions like *proto_debug()*. However, their output isn't always shown in the console because they are set up in zones that can be optionally activated. Below is an example.

```
proto_debug(DEBUG_IOCTL, "%s IOCTL Set pin %s.", __func__, ((1 ==
pinVal)?"HIGH":"LOW"));
```

This is from the PROTO_IO_SET_STATE case in *proto_ioctl()*. Its first parameter is the zone it is allocated to. These zones are defined in *proto.h*.

```
enum PROTO_DEBUG_ZONES{
        DEBUG_OFF       = 0,
        DEBUG_ERROR     = BIT(0),
        DEBUG_WARNING   = BIT(1),
        DEBUG_INFO      = BIT(2),
        DEBUG_INIT      = BIT(3),
        DEBUG_IOCTL     = BIT(4),
```



```
        DEBUG_IRQ       = BIT(5),
        DEBUG_UNUSED4   = BIT(6),
        DEBUG_UNUSED5   = BIT(7),
        DEBUG_ALL       = ~0,
};
```

As you can see, the zones are enumerated values that make up a bit field so their states can be stored in a single variable. This variable is a module parameter called *proto_debugLevel* that is defined in *main.c*.

```
int proto_debugLevel            = DEBUG_INFO | DEBUG_ERROR | DEBUG_INIT;
```

Above, you see the initial value of *proto_debugLevel* enables the INFO, ERROR, and INIT zones. Thus, the call to *proto_debug()* we looked at a few lines back wouldn't be shown initially because the DEBUG_IOCTL zone wasn't included. To activate it, we need to change the value of *proto_debugLevel*. This is done through the module's entry in the */sys/module* directory. In this case, it is */sys/module/proto* and the directory where the module parameters are kept is */sys/module/proto/parameters*. Looking at the entry for *proto_debugLevel,* we see:

```
root@am3517-evm:/sys/module/proto/parameters# cat proto_debugLevel
13
root@am3517-evm:/sys/module/proto/parameters#
```

This is bits 0, 2, and 3. If we wanted to activate the DEBUG_IOCTL zone, we would add bit 4 to this. Here's an example of how to do that:

```
root@am3517-evm:/sys/module/proto/parameters# echo $((13+(1<<4))) >
/sys/module/
proto/parameters/proto_debugLevel
root@am3517-evm:/sys/module/proto/parameters# cat proto_debugLevel
29
root@am3517-evm:/sys/module/proto/parameters#
```

Now when you execute an IOCTL, you will access the proto module and all the debug messages that are in the DEBUG_IOCTL zone will be printed.

4.3 User Application Source

The protoUserApp application provides a simple template of how to send IOCTL commands to the proto.ko module once it has been loaded.

4.3.1 Setup and User Input

The application uses simple numerical commands to specify which command to send to the module. These commands are specified in *protoUserApp /main.h.*

```
#define CMD_RESET                       0
#define CMD_FORCE_HIGH                  1
#define CMD_FORCE_LOW                   2
#define CMD_READ_PIN                    3
#define CMD_READ_INT_COUNTS             4
```



```
#define CMD_EN_SOFT_INT                 5
#define CMD_DIS_SOFT_INT                6
```

When the application starts, *argc* and *argv* are checked to verify that proper user input has been specified. The switch block then determines which values are placed in the *driverCmd* and *driverOption* variables. Below is a command to read the interrupt counts; we will use this as an example.

```
case CMD_READ_INT_COUNTS:
driverCmd = PROTO_IO_INT_COUNTS;
driverOption = (unsigned long)&intCounts;
break;
```

In this example, we see *driverCmd* set to PROTO_IO_INT_COUNTS. This is the IOCTL code defined in *../proto/proto_ioctl.h* and included at the top of the file. Next, *driverOption* is set to the address of the *intCounts* function. With these two values prepared, we can open the driver and send the command.

4.3.2 Open Driver

The driver is part of the Linux filesystem and is accessed via the */dev* directory. The macro DRIVER_NAME is defined in *…/protoUserApp/main.h.*

```
#define DRIVER_NAME                     "/dev/proto0"
```

To open the driver, we get a file pointer to it with the *open()* system call.

```
fp = open(DRIVER_NAME, 0);
if (-1 == fp)
{
printf("Error: Cannot open %s.\n", DRIVER_NAME);
return ERROR_CANT_OPEN_DRIVER;
}
```

4.3.3 Send Command to Driver

Once we have a valid file pointer to the driver, we send the values we prepared in Section 4.3.1 using the *ioctl()* system call:

```
res =  ioctl(fp, driverCmd, &driverOption);
if (res)
{
printf("Error: Driver command failed. err: %d\n", res);
return ERROR_COMMAND_FAILED;
}
```

It is important to note here that since not all of the commands send literal values, we send the address of *driverOpion* as the second parameter of *IOCTL()*. This allows the module to perform the address space calculation and move data from user space to kernel space (and vice-versa) regardless of the command.

4.3.4 Responses from Driver

The final switch statement checks if the command sent is expecting a response from the module. You will notice that there are no special functions called here. The module is responsible for the kernel-to-user-space transfer of the data; so, once the IOCTL call is finished, the data has already been transferred if it was successful.

4.3.5 Close Driver

Finally, it is important to remember to close the driver's file pointer when it is no longer needed. Failure to do so is a memory leak.

Appendix A: Install the SSH Server

Your VM workstation image may not have an SSH server installed. Use Aptitude to install the SSH server.

```
logic@logic-desktop-am3517:~ $ sudo apt-get install openssh-server
Reading package lists... Done
Building dependency tree
Reading state information... Done
Suggested packages:
  rssh molly-guard openssh-blacklist openssh-blacklist-extra
The following NEW packages will be installed:
  openssh-server
0 upgraded, 1 newly installed, 0 to remove and 176 not upgraded.
Need to get 285kB of archives.
After this operation, 782kB of additional disk space will be used.
Get:1 http://us.archive.ubuntu.com/ubuntu/ lucid-updates/main openssh-
server 1:5.3p1-3ubuntu7 [285kB]
Fetched 285kB in 0s (539kB/s)
Preconfiguring packages ...
Selecting previously deselected package openssh-server.
(Reading database ... 126695 files and directories currently installed.)
Unpacking openssh-server (from .../openssh-server_1%3a5.3p1-
3ubuntu7_i386.deb) ...
Processing triggers for ureadahead ...
ureadahead will be reprofiled on next reboot
Processing triggers for ufw ...
Processing triggers for man-db ...
Setting up openssh-server (1:5.3p1-3ubuntu7) ...
Creating SSH2 RSA key; this may take some time ...
Creating SSH2 DSA key; this may take some time ...
ssh start/running, process 4787

logic@logic-desktop-am3517:~$
```

