GNU/Linux and EZ-USB®: Controlling USB Devices From User-Space

Introduction

This application note and accompanying software demonstrate how simple user-mode GNU/Linux applications can communicate directly to EZ-USB® based USB devices, without any need to write a kernel device driver. The author assumes the reader has a working knowledge of GNU/Linux.

Some USB devices require the installation of kernel-mode drivers to function properly under the Linux operating system; writing kernel-mode drivers can be a complicated process. But for other specialized USB devices, such as scanners, digital cameras, mass-storage devices, etc., communication between the host and the device can be accomplished directly, without the need for device driver development, by utilizing standard features of GNU/Linux starting with the 2.4 kernel. These new features include “Hotplug” software, used to perform dynamic reconfiguration of USB devices, pre-written generic or class drivers, and the USB Device File System APIs.

The included application “HexPad”, consists of several components that combine to communicate with an EZ-USB or EZ-USB FX™ development board directly from user-space using these features. The “HexPad” user interface is shown in *Figure 1.*

When an EZ-USB or EZ-USB FX board is plugged into a USB port, clicking on one of the “HexPad” buttons will cause the corresponding character to light on the development board's seven-segment LED.

The Hexpad components have been successfully installed, built and executed on both a PC laptop running Redhat 7.2 Linux and an Apple® Macintosh® PowerBook® G4 computer running Yellowdog Linux 2.2.

GNU/Linux Hotplug Operations

When GNU/Linux first boots, a “hotplug” system service is invoked, which initializes the hotplugging functionality for USB, PCI, networking, and other removable device categories.

When one of these devices is attached to the system, the hotplug system executes /sbin/hotplug, which, in turn, invokes /etc/hotplug/<device_type>.agent. The agent files use environment variables, shell scripts, and text “database” files to get information about loading drivers and firmware, performing configuration tasks, and handling administrative functions.

More detailed information about Hotplug functions can be found at http://linux-hotplug.sourceforge.net.

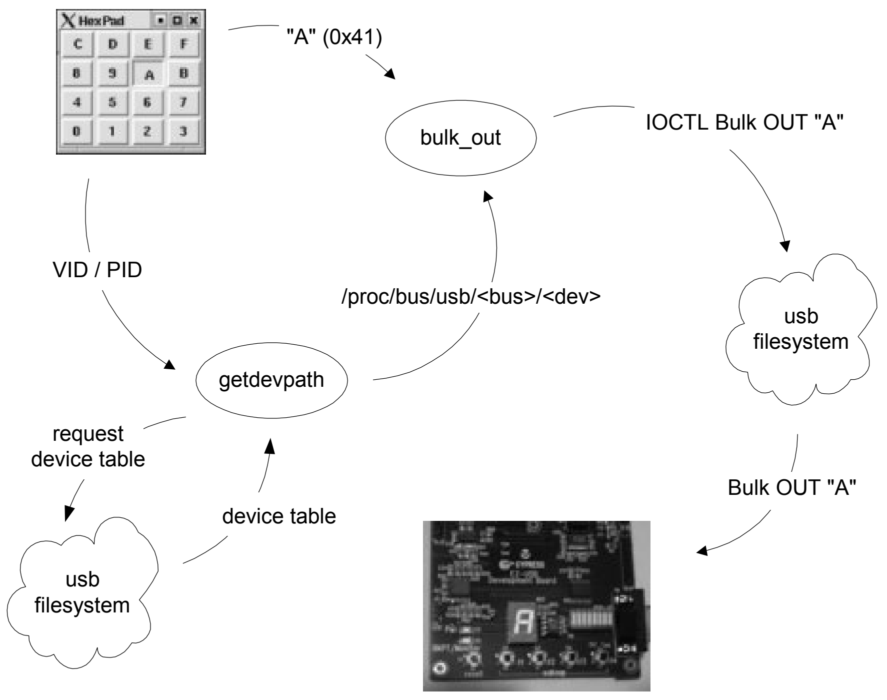

Figure 1.

Preparing Your GNU/Linux System

The software described in this document was developed and tested on Linux platforms running GNU/Linux based on the RedHat 7.2 distribution. As Linux is an “always evolving” OS, it is important that you obtain the latest Hotplug components from http://linux-hotplug.sourceforge.net.

Installing “HexPad”

The archive “hexpad.tar.gz” consists of a “hexpad” directory containing several files, listed in *Table 1*.

Table 1.

Bulk7Seg.hex	8051 firmware in “hex” format
Bulk7Seg.tar.gz	firmware source files
bulk_out.c	Source code of interface to usbdevfs
ez.map	Data added to usb.usermap
getdevpath	Shell script
Hexpad_script	Hotplug script
Hexpad	Tcl/tk application

These directions assume you are using a windowing environment such as GNOME or KDE. They also assume you have Tcl/Tk properly installed on your system.

Expand the gzipped tar file into your home directory and set the resulting “hexpad” directory as the current working directory:

```
% tar zxf hexpad.tar.gz[1]
% cd ~/hexpad
```

Next, SU to root, and make a directory to hold the firmware file and move the firmware file to the new directory:

```
% su
Password:
# mkdir -p /etc/hotplug/usb/ezusb.fw
# mv Bulk7Seg.hex /etc/hotplug/usb/ezusb.fw/
```

Then, move the HexPad hotplug script to the location where the hotplug usb agent will find it, and make the script executable.

```
# mv hexpad_script /etc/hotplug/usb/
# chmod 0775 /etc/hotplug/usb/hexpad_script
```

Next, add a line to the usb.usermap file to tell the Hotplug USB agent to execute the HexPad script when the EZ-USB development board is hotplugged into the system. You can add the required line with this command:

```
# cat ez.map >> /etc/hotplug/usb.usermap
```

Now you are ready to see if the hotplug scripts can discover your EZ-USB board. Using this terminal window and type:

```
# tail -f /var/log/messages
```

then attach a development board; if the proceeding steps have been done correctly, you should see new messages being written to the log file. In particular, you should see a line that contains “fxload... Bulk7Seg.hex”. Once the firmware file downloads to the board, re-numeration occurs and the green “monitor” LED lights up.

Now, open a new user terminal window, and change to the “hexpad” directory). Then type

```
% ./getdevpath -v547 -p1002
```

and the system should respond with

```
/proc/bus/usb/bbb/ddd
```

where bbb is a three-digit number corresponding to the USB root hub and ddd is the three-digit device number assigned to your device by the USB subsystem. (If you have more than one of these devices, this will only show the first one.)

Next, build the bulk_out tool:

```
% gcc -o bulk_out bulk_out.c
```

Finally, run the hexpad application

```
% ./hexpad
```

As you click on the application's buttons, the corresponding characters appear on the seven-segment LED.

Now, detach and re-attach the development board. Notice that, in the messages log file being displayed in the root terminal, the USB subsystem has assigned a new device number to the device. With the HexPad application still running, click on a button and verify that the application still finds the dev board.

Configuring the Dev Board For HexPad

When the development board is plugged into the system, the kernel detects the event and invokes the hotplug helper application (by default, /sbin/hotplug) with the single parameter usb, as well as setting certain environment variables, such as the vendor and product id's for the device. The helper application then launches the USB agent script. The function of this script is to open and read various USB map files, looking for a match to our device. We added the line

```
hexpad_script 0x0003 0x0547 0x0080 0x0001 0x0000 0x00 0x00
0x00 0x00 0x00 0x00 0x00000000
```

to the usb.usermap file. The agent script line above translates as follows: use the first hexadecimal number as a bit mask that in conjunction with the next two hex numbers identifies the board’s VID and PID[2]. When the board is hotplugged, the script named in the 1st field is executed to perform device configuration. The “hexpad_script” file is listed below.

```
#!/bin/sh
# hexpad_script
FIRMWARE=usb/ezusb.fw/Bulk7Seg.hex
/sbin/fxload -I $FIRMWARE
```

Configuration scripts executed by the Hotplug system in the way described can perform any task necessary to prepare a

Note:

1. The “%” prompt in code listings indicates normal user-level login, while “#” indicates super-user-, or root-, level login.
2. “VID” is Vendor ID; “PID” is Product ID.

USB device for use. In the case of the HexPad application, our script calls the HotPlug “fxload” utility to download our firmware file to the EZ-USB dev board. In more complex cases, it can do quite a lot more, such as tell system services about the new device.

The Bulk7Seg Firmware

The Bulk7Seg.hex file was developed on a Windows® 2000 platform using the Keil µVision2 IDE and the Cypress/Anchor Firmware Frameworks, both of which are included in the EZ-USB Development Kit installation.

The principal routine of Bulk7Seg is shown here:

```
#define LED_ADDR0x21

BYTE xdata Digit[] = { 0xc0, 0xf9, 0xa4, 0xb0, 0x99, 0x92,
0x82, 0xf8, 0x80, 0x98, 0x88, 0x83, 0xc6, 0xa1, 0x86, 0x8e
};

void TD_Poll(void) // Called repeatedly while the device is
idle
{
   BYTE  c;

   // Is there something in the OUT2BUF buffer?

   if (! (EPIO[OUT2BUF_ID].cntrl & bmEPBUSY) )
     if ( EPIO[OUT2BUF_ID].bytes > 0)
     {
       // convert ascii char to 0..15

       c = OUT2BUF[0] - 0x30;

       if (c > 9)
               c = c - 0x11 + 10;

       EZUSB_WriteI2C(LED_ADDR, 0x01, &(Digit[c] ));

       EZUSB_WaitForEEPROMWrite(LED_ADDR);

       EPIO[OUT2BUF_ID].bytes = 0;
     }
}
```

The TD_Poll routine looks for single-byte input at endpoint 2. When a new byte is available, the routine converts the byte from an ascii hex character in the range ['0'..'F'] to a single byte integer from 0 to 15. This integer becomes the index to Digit, an array of codes corresponding to the individual segments of the seven-segment array. When the specific code is written to LED_ADDR, the character appears on the LED.

The HexPad User Interface

The HexPad application has these functional components: hexpad, getdevpath, and bulk_out. Full listings for each of these components will be found in the “Listings” section of this document.

The hexpad component is a Tcl/Tk script file that constructs and displays the HexPad user interface. In this application, arguments to the routine MakeButtons, the characters '0' through 'F', become both the text displayed on the buttons as well as the parameter passed to the procedure b_out.

```
proc b_out {c} {
  exec ./getdevpath -v547 -p1002 | ./bulk_out -c$c
}

proc MakeButtons {args} {
  set b 0
  set c 1
  set r 4
foreach val $args {
    button .$b -text $val -command "b_out $val"
    grid .$b -column $c -row $r
    incr b
    incr c
    if {$c == 5} {
      set c 1
      set r [expr $r - 1 ]
    }
  }
}

MakeButtons 0 1 2 3 4 5 6 7 8 9 A B C D E F
```

The b_out procedure executes this shell command:

```
./getdevpath -v547 -p1002 | ./bulk_out -c$c
```

The getdevpath script obtains a list of all devices attached to the system’s root hubs by reading the USB file system's /proc/bus/usb/devices file. By comparing the vendor and product ID's in the list to the -v and -p parameters, the script extracts the corresponding bus and device numbers, builds the file system path to our dev board, and writes the path string to stdout. The format of the /proc/bus/usb/devices file is documented in most Linux distributions in /Documentation/usb/proc_usb_info.txt, in your Linux distribution.

The bulk_out program provides the interface to the usb file system. The dev board's file system path is read from stdin, and the character to be displayed comes in through the -c argument.

After declaring the data structure to contain information about the bulk_out transfer

```
struct usbdevfs_bulktransfer bulk;
```

and moving the character to be displayed to a “holding” buffer, the program reads the path name (that was constructed by the getdevpath script) from stdin.

```
scanf("%s",fname);
```

Next, the device is opened, the bulk data structure is filled in, and a bulk transfer ioctl call is made

```
ioctl(fd, USBDEVFS_BULK, &bulk);
```

This ioctl call sends the one-byte buffer to the dev board by a BULK_OUT transfer. causing the character to be displayed on the LED, as described on page 3.

Conclusion

In a similar fashion, many standard USB functions can be performed from user-space. See the file /usr/src/<linux>/include/linux/usbdevice_fs.h for details, and the kernel source in /usr/src/<linux>/drivers/usb/devio.c for information about how to use each ioctl request. **Note:** “<linux>” should be replaced with the partial path to the GNU/Linux source directory on your system; on my system, this is “linux-2.4.7.10”.

We have discussed the USB file system and Hotplug facilities introduced in the 2.4 Linux kernel. Then we demonstrated how several types of components (C programs, shell scripts and Tcl/Tk) can be used together to achieve desired results.

This user-space solution can be used for several purposes. Shell scripts can be used for repeated device testing; device driver functionality can be prototyped quickly; finally, end-user applications can be developed and deployed without requiring extensive device driver development or installation.

References and Links

Information about USB programming for GNU/Linux can be found here: http://www.linux-usb.org

This URL: http://linux-hotplug.sourceforge.net provides details on hotplugging, as well as links to up-to-date downloads for hotplug scripts and the fxload utility.

Listings

bulk_out.c

```
/*
 *  Bulk Transfer
 *  © 2002 Cypress Semiconductor
 *
 */
# include  <stdlib.h>
# include  <stdio.h>
# include  <getopt.h>
# include  <string.h>
# include  <sys/ioctl.h>
# include  <sys/types.h>
# include  <sys/stat.h>
# include  <fcntl.h>
# include  <linux/ioctl.h>
# include  <linux/usbdevice_fs.h>

int main(int argc, char*argv[])
{
int            opt;
char           fname[256];
constchar      *buf;

struct usbdevfs_bulktransfer bulk;
int            fd, rc;

    while ((opt = getopt(argc, argv, "c:")) != EOF)
      switch (opt) {
        case 'c':
        buf = optarg;
        break;
    default:
        break;
    }

    scanf("%s",fname);

    // open the device
    if ((fd = open(fname,O_RDWR)) < 0)
    {
        printf("file open error\n");
        return -1;
    }

    // set up bulk-transfer data structure
    bulk.ep = 2;
    bulk.len = 1;
    bulk.timeout = 100;
    bulk.data = (void *)buf;

    // pass bulk-transfer struct to usb file system
    if (ioctl(fd, USBDEVFS_BULK, &bulk) < 0)
    {
        printf("bulk xfer error\n");
        return -1;
    }

    return 0;
}
```

getdevpath

```
#!/bin/sh
# getdevpath
#
# Copyright(c) 2002 Cypress Semiconductor
#
# this script reads /proc/bus/usb/devices and builds the
# proper usb file system path to the device corresponding
# to the caller's vid and pid arguments.  The pathname is
# written to stdout.
#
# usage: getdevpath -vVID -pPID
#   where VID is Vendor ID in hex
#         and PID is Product ID in hex

# set variables
MYVID=
MYPID=
TLINE=
BUS=
DEV=
EURIKA=

function usage() {
   cat <<EOF
Usage: $0 -vVID -pPID
   -v   Vendor ID in hex
   -p   Product ID in hex
   -?   this help message
EOF
   exit 0
}

# parse command-line arguments
while getopts ":v:p:" opt; do
  case $opt in
    v) MYVID=$OPTARG ;;
    p) MYPID=$OPTARG ;;
    \?) usage ;;
  esac
done

# pad VID and PID with left-end zeros
COUNT=`echo $MYVID | wc -c`
while [ $COUNT -lt 5 ]
  do
    MYVID=`echo 0$MYVID`
    let COUNT=COUNT+1
  done

COUNT=`echo $MYPID | wc -c`
while [ $COUNT -lt 5 ]
  do
    MYPID=`echo 0$MYPID`
    let COUNT=COUNT+1
  done

# VID and PID both equal to "0000" is invalid
if [ "$MYPID" == "0000" ]
then
   if [ "$MYVID" == "0000" ]
   then
      usage;
   fi
fi

# set result in temp file for later checking
echo "false" > eurika

# read usb devices list, taking the lines with useful info

cat /proc/bus/usb/devices | grep "T:\|P:" |

while
   read LINE
do
   if [ `expr substr "$LINE" 1 1` == "T" ]
   then
      TLINE=$LINE
   else
      VID=`expr substr "$LINE" 12 4`
      PID=`expr substr "$LINE" 24 4`
      if [ $VID == $MYVID ]
      then
         if [ $PID == $MYPID ]
         then
            echo $TLINE > tline
            BUS=`awk '{print $2}' < tline | cut -c5-`
            COUNT=`echo $BUS | wc -c`
            while [ $COUNT -lt 4 ]
              do
                 BUS=`echo 0$BUS`
                 let COUNT=COUNT+1
              done
            DEV=`awk '{print $8}' < tline`
            COUNT=`echo $DEV | wc -c`
            while [ $COUNT -lt 4 ]
              do
                 DEV=`echo 0$DEV`
                 let COUNT=COUNT+1
              done
            rm tline
            echo "true" > eurika
            echo /proc/bus/usb/$BUS/$DEV
         fi
      fi

   fi

done

if [ `cat eurika` == "false" ]
then
   echo "device not found"
fi

rm eurika

# end of file
```

Hexpad

```
#!/bin/sh
# \
exec wish "$0" "$@"

wm title . HexPad

proc b_out {c} {
  exec ./getdevpath -v547 -p1002 | ./bulk_out -c$c
}

proc MakeButtons {args} {
  set b 0
  set c 1
  set r 4
  foreach val $args {
    button .$b -text $val -command "b_out $val"
    grid .$b -column $c -row $r
    incr b
    incr c
    if {$c == 5} {
      set c 1
      set r [expr $r - 1 ]
    }
  }
}

MakeButtons 0 1 2 3 4 5 6 7 8 9 A B C D E F

# end of file
```

approved dsg 9/12/02