Design and Implementation of an Intelligent USB Peripheral Controller

Gianluca Valentino
Department of Communications and Computer Engineering
University of Malta
Msida, Malta
gval0001@um.edu.mt

Saviour Zammit
Department of Communications and Computer Engineering
University of Malta
Msida, Malta
saviour.zammit@um.edu.mt

ABSTRACT

This paper presents the design and implementation of an intelligent, re-programmable device that is capable of automatically detecting USB peripherals on insertion and performing various tasks accordingly. Examples include the automatic transfer of data between pen drives or the automatic printing of a file located on a pen drive. The performance of the system was analyzed and results for the execution time and CPU utilization of the programs performing the tasks were obtained. A comparison was made with the same programs running on a laptop, which was set as a benchmark.

Keywords

USB Controller, Automatic re-programmability, Data transfer, Performance analysis

1. INTRODUCTION

There are various PC (Personal Computer) interfaces, the most popular being USB (Universal Serial Bus), with thousands of peripherals available for connection to PCs. However, it is necessary to have a PC to be able to access or control these peripherals; the need of a user to transfer data between two USB Mass Storage devices, or print an image that has just been scanned, are such examples. In such cases, a low-cost peripheral option is more desirable.

The concept of the Intelligent Peripheral Controller (IPC) [12] is therefore that of a re-programmable device that can detect connected USB peripherals and automatically perform an action accordingly, as shown in Figure 1. The 'intelligence' of the device is derived from its ability to automatically detect peripherals and re-program itself automatically on the insertion of a pen drive containing the new software to be installed, thus increasing the number of tasks it can perform.

Research was conducted to identify any existing devices which might already support the features of the IPC. USB bridges such as the Belkin USB Anywhere and Hitch have been identified [5]. Other devices that have some of the functions contemplated include smart phones, multimedia players and plug computers.

An online search for patents yielded a 'Portable Selective Memory Data Exchange Device' [11], which acknowledges the fact that "there is a need to transfer information from on UFD (USB Flash Drive) to another UFD without the presence of a computer". The invention lacks an operating system, and may be controlled by a set of simple user buttons. Files are transferred depending on their file name, file extension, file size, file location (such as if they are located in a specific folder on the pen drive) or file status.

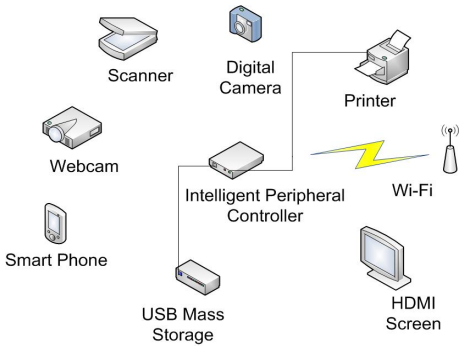

Figure 1: The various peripherals that can be interconnected via the IPC

Existing methods for automatically re-programming embedded systems tend to consider the updating of low-level code (firmware), rather than high-level applications. Yao et al. [13] maintain that as embedded devices are achieving network connectivity, online-update of their software is becoming more of an issue. They present an extended form of the normal boot loader. The new software is obtained from a TFTP server, and is installed by writing it to flash memory.

The risks involved in updating embedded system software are discussed in [7]. In critical systems that have to run continuously, stopping the system for an update is not the ideal way to do things. The contribution made in [7] is the design of a software framework in which an application program can be updated 'dynamically' while running. However, these issues should not be of much concern to the design of the IPC, since the system would not be that critical.

Li et al. [9] have applied the automounting technique to develop an automatic software install/update system for Linux. They placed an executable shell script file into a RPM package, which in turn was transferred to a USB Mass Storage device. When the device was inserted into the embedded system, it was mounted automatically, and the shell script file was executed.

The few devices mentioned do not meet the low-cost and re-programmable criteria of the IPC, which therefore motivates this work. This paper is organized into three sections. The first section discusses the design parameters and hard-

ware/software issues involved in setting up the system. An overview of the system implementation is given in the second section, and evaluation and testing results are provided in the third section.

2. SYSTEM DESIGN

2.1 Hardware

A decision was made to opt for an ARM processor due to its popularity in the design of embedded systems [1]. The BeagleBoard [3] was chosen as the target board on which to implement the IPC. Its features include an OMAP 3530 ARM Cortex-A8 based microprocessor (720 MHz), 256 MB of RAM as well as a support for a substantial number of peripherals, such as USB, HDMI, SD Card and audio connectors.

2.2 Software

The design issues related to software include whether to make use of an existing operating system, which programming language to use for the implementation, and which software tools could be used to develop and test the IPC. A review of the existing literature showed that Linux is the operating system of choice in embedded systems [10]. The C language is ideal since it results in efficient, portable code, and can easily provide access to the underlying hardware [6].

If the IPC is to be implemented on top of the Linux OS, development and evaluation tools making up the GNU toolset can be used. These include GCC (GNU Compiler Collection), GDB (GNU Debugger), gcov (GNU source code coverage) and gprof (GNU profiler). Other tools include time and top, which can be used to analyze the execution times and CPU utilization of the different programs respectively.

3. SYSTEM IMPLEMENTATION

There are three stages involved in implementing the IPC:
(a) the set up of the Linux system;
(b) the design of the system software architecture;
(c) the writing of the programs making up the system.

3.1 Linux System Set-up

A Linux system generally consists of a boot loader, a kernel and a root file system. The task of the boot loader is to boot the Linux kernel at power-up, which in turn mounts the root file system. Ubuntu was chosen as an embedded Linux OS because of its wide support by the BeagleBoard community.

The XLoader and Uboot boot loaders are pre-installed in the BeagleBoard's flash memory. The rootstock package [2] was used to build a kernel and a root file system for Ubuntu. Readily-compiled kernel images are available at [4]. The 2.6.31.5-x5.3 kernel image version was selected for its driver support for USB Wi-Fi and USB webcam.

The rootstock command was then executed in a terminal on the host computer to obtain a Debian package. The booting mechanism supported by the BeagleBoard is via SD Card. A 8 GB SD Card was formatted to have two partitions, the first of which is FAT32-formatted, while the second partition is Ext3-formatted.

The boot script consisting of a number of instructions to be executed by the boot loader, as well as a uImage file were loaded into the first partition. The uImage file was generated by the mkimage tool, which takes the vmlinuz file produced by rootstock as an input. The root file system (also generated by rootstock) was extracted to the Ext3 partition.

Table 1: Software Package Components

File	Naming Convention
Executable run by udev	usb-backup
Executable source code	usb-backup.c
Installation program	usb-backup-installer
Installer source code	usb-backup-installer.c
Program documentation	usb-backup.txt

Table 2: Summary of IPC Functionalities

Functionality	Peripherals involved	Program Name
Data Backup	USB Flash Drive	usb-backup
	USB Camera	camera-backup
Data Transfer	x2 USB Flash Drive	usb-to-usb
Video Display	USB Webcam	webcam-display
	HDMI screen	
Printing	USB Flash Drive	usb-to-printer
	USB Printer	printer-config
Audio	Speakers	N/A
Internet Connection	USB Wi-Fi Dongle	wifi, wifi-config
Human Interface	USB Mouse	N/A
Device	USB Keyboard	N/A
Automatic Software	USB Flash Drive	auto-install
Installation		

3.2 IPC Software Architecture Overview

In order to implement the functionalities of the IPC, a peripheral insertion detection mechanism is required. One such mechanism is udev [8]. Its features include the ability to execute programs when certain device events occur (such as insertion or removal), as well as allowing access to information about currently attached devices. Thus, user programs would not need to continuously poll the device for any activity, and instead would be executed when required.

Therefore, the backbone of the system would be udev, for which rules can be written for in order to automatically run certain programs when certain peripherals are inserted. Each and every feature of the IPC is defined by a software application, and is installed to the IPC from a USB pen drive. The standard format and naming conventions for every software application is listed in Table 1, taking a program named usb-backup (which backs up data on the pen drive to the IPC) as an example.

These five files are compressed using tar and the gzip compression utility into a single file, thus forming a software package.

3.3 Program Implementation

A number of programs were written to implement a range of functionalities which are determined by different combinations of attached peripherals. For example, inserting two pen drives would indicate that the user would like to transfer data from one device to another. A summary of these programs and the respective peripheral combinations is given in Table 2.

The initial software present on the IPC consists of the

mount-usb, *auto-install* and *operation-ready* programs, together with their corresponding udev rules located in the /etc/udev/rules.d/10-IPC.rules file. The *mount-usb* program is responsible for mounting a pen drive to a specific mount-point, located in /home/ubuntu/Mount. A sub-directory is created in this location with a name based on the file in which the device appears in /dev, e.g. 'sda'.

In order to keep production costs low, the current implementation assumes a device (such as the IPC) that is devoid of any user interface, such as a screen. Since this denies the user the option of manually selecting the new software to be installed, the user is therefore constrained to place the new software in a certain pre-established folder on the pen drive. The insertion of the pen drive triggers udev, which in turn runs the the *auto-install* program. This program searches for and extracts any new software packages on the pen drive to the /usr/local/temp directory on the board. The software is determined to be 'new' by checking whether a software package of the same name already exists in the directory.

The various files listed previously in Table 1 are then copied to directories in /usr/local, including 'bin' for the new executable, 'src/bin' for the executable source code, and 'docs' for the program documentation. The new software is then installed by executing the installer program, which writes the appropriate udev rule to the rules file.

The task of the *operation-ready* program is to signal whether the current operation has terminated correctly. Since it is the last program to be executed, it is placed in a separate, appropriately named rules file 20-IPC.rules, such that it is parsed after the first rules file. If the program that has been executed according to the combination of peripherals inserted has terminated successfully, a speech output invites the user to press the button twice to unmount all drives. Once the unmounting process is over, the user is then informed that the peripherals may be removed.

Other programs related to data transfer involving pen drives include the *usb-backup* and *usb-to-usb* programs. The first program creates a new directory with a concatenation of the current date and time as a name in /usr/local/backups. All files in the 'Backups' directory on the pen drive are copied to the newly-created directory. Transfer of data between pen drives is achieved by the second program. In this case, any files not already present in the 'Transfer' directory of the second device to be inserted are transferred from the 'Transfer' folder of the first device.

The remaining features, such as the backup of data on a camera, Wi-Fi connection, printing and webcam display are provided by the *camera-backup*, *wifi-config* and *wifi*, *printer-config* and *usb-to-printer*, and *webcam-display* programs respectively. These programs make use of standard Linux programs, such as gphoto2, wpa supplicant, hplip, lpr and xawtv. If certain peripherals trigger multiple tasks, each task will be performed sequentially according to its position in the rules file.

4. EVALUATION AND TESTING RESULTS

The evaluation and testing of the system consisted of a rigorous performance analysis of the various combination of running programs to calculate the execution times and determine the CPU utilizations of the programs. In addition, a set of benchmarks were developed to compare the speed of execution and CPU utilization of the programs on the IPC with the same parameters on a MSI EX600 laptop (also running Ubuntu Karmic 9.10). The MSI EX600 laptop supports a 2 GHz Intel ® Core ™ 2 Duo processor and 2 GB of DDR2 RAM.

Table 3: Files used to measure Execution Duration

File Type	File Extension	File Size
Image	.jpg	6.3 KB (Small - S)
Document	.pdf	5.5 MB (Medium - M)
Video	.avi	697.8 MB (Large - L)

4.1 Performance Analysis of the IPC

The time taken for each program to complete its task was determined using the *time* command, which is a standard Linux utility. In the case of data transfer, a set of files (listed in Table 3) differing in size and type were used to measure the time taken to transfer them.

To determine the time taken to copy all the media stored on a digital camera to the IPC, a typical scenario consisting of 349 images and 26 video clips was considered. The *time* utility contributed to a significant portion of the results, as the choice whether to use the IPC instead of a PC to carry out certain operations depends on the delays involved in completing them.

Data related to the percentage of the CPU resources used up by the programs were obtained using the *top* command. In the cases of programs which complete in less than a second, the CPU utilization was found to vary as much as 60% for multiple runs.

4.2 Comparison in Performance between IPC and Laptop

The raison d'être of the IPC is to replace the PC in performing daily functions such as transferring of data between peripherals. Thus, a way of testing the IPC's efficiency would be to set the execution times and CPU utilization of the programs running on the standard PC as a benchmark, and to asses the relative performance of the same programs running on the IPC. The programs were all re-compiled on the x86 PC, and were run with *time* and *top*. Figure 2 depicts the times for programs which take less than a second to execute, while Figure 3 shows the timing results for the rest of the programs. A comparison of the CPU utilizations for some of the programs taking more than a second to execute is shown in Figure 4.

The comparison in execution duration and CPU utilization was made to establish whether it would be feasible from a view-point of speed to implement the software technique on a low-cost device which could support multiple features. The results obtained show that the execution duration and CPU utilizations for the implementation on the IPC is comparable or in some cases slightly worse than that for the laptop. However, one has to consider the fact that there is a difference of approximately a factor of 8 in the amount of RAM and a factor of 2.8 in the CPU speed of the two types of hardware.

5. SUGGESTIONS FOR FUTURE WORK

An idea for future work involves allowing for automatic updates of the system software (Linux OS) on the insertion of a pen drive, rather than only the application software.

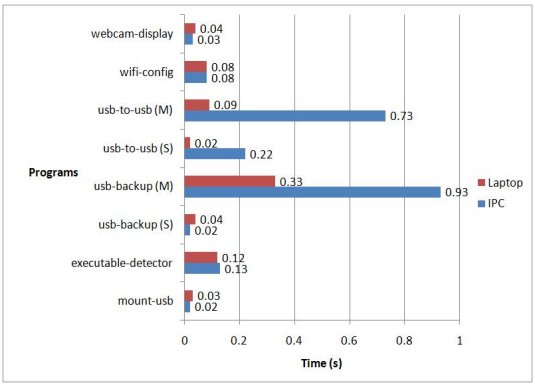

Figure 2: Comparison of Execution Times (a)

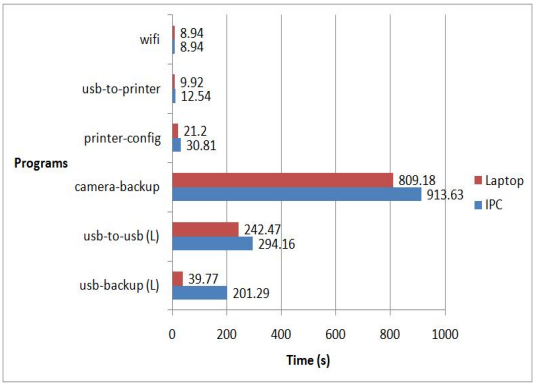

Figure 3: Comparison of Execution Times (b)

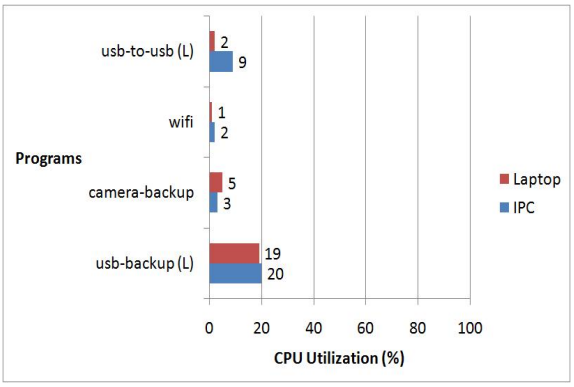

Figure 4: Comparison of CPU Utilizations

This would facilitate the installation of drivers necessary for new USB devices. Further suggestions relate to the provision of an online interface for users to download software packages to their pen drives, as well as the possibility to uninstall unneeded features.

6. CONCLUSION

The contribution of this work was the design and implementation of a low-cost dedicated device, termed as an "Intelligent Peripheral Controller". The current implementation runs a set of programs defining the various functionalities of the system on top of a Ubuntu Linux operating system, together with udev as a hardware detection mechanism. The system is re-programmable by means of a USB pen drive, and its performance has been analyzed and found to be comparable with that of a PC.

7. REFERENCES

[1] Arm press release, arm announces 10 billionth mobile processor, http://www.arm.com/news/24403.html, February 2009.

[2] Rootstock project web page, https://launchpad.net/projectrootstock, March 2010.

[3] Beagleboard system reference manual rev c4, December 2009.

[4] Index of 2.6.31.5 kernel, http://rcn-ee.net/deb/kernel/beagle/karmic/v2.6.31.5-x5.3/, March 2010.

[5] T. Balic. Transfer files from one usb device to another without using a computer, http://www.brighthub.com/computing/hardware/articles/31630.aspx, December 2009.

[6] M. Barr and A. Massa. *Programming Embedded Systems with C and GNU Development Tools.* O'Reilly, 2 edition, 2006.

[7] M. Hicks and S. Nettles. Dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 27(6), November 2005.

[8] G. Kroah-Hartman. udev - a userspace implementation of devfs. In *Proceedings of the Linux Symposium*, July 2003.

[9] T. Li and H. Pei-wei. Automatic software install/update for embedded linux. *Journal of Shanghai Jiaotong University (Science)*, 13(1), February 2008.

[10] C. J. Murray. Embedded linux extends its reach. *Design News*, July 2009.

[11] I. Pomerantz. Portable selective memory data exchange device, u.s. patent 2007/0065119, March 2007.

[12] G. Valentino. *Design and Implementation of an Intelligent Peripheral Controller*. University Malta, June 2010.

[13] G. Yao, W. Zhang, and J. Wang. The design and implementation of online-update on embedded devices. In *IEEE International Conference on Computer Science and Software Engineering*, 2008.