Министерство образования и науки Российской Федерации

Федеральное государственное бюджетное образовательное учреждение «Томский государственный университет систем управления и радиоэлектроники»

(ТУСУР)

УТВЕРЖДАЮ

Заведующий кафедрой

«Управление инновациями»

_____ А.Ф. Уваров

«____» _____ 2012 г.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К ЛАБОРАТОРНОМУ ПРАКТИКУМУ И САМОСТОЯТЕЛЬНОЙ РАБОТЕ

по дисциплине

«Архитектура систем на кристалле»

Составлены кафедрой «Управление инновациями»

Для студентов, обучающихся по направлению подготовки 222000.68 «Инноватика»

Магистерская программа «Управление инновациями в электронной технике»

Форма обучения – очная

Составитель:

Ассистент кафедры УИ _____ Н.В. Милованов

«____» _____ 2012 г.

Томск 2012 г.

СОДЕРЖАНИЕ

Введение

Цель дисциплины «Архитектура систем на кристалле» – дать будущим специалистам знания о развитии и технологиях современной микропроцессорной индустрии, знания об архитектурах современных систем на кристалле ведущих мировых производителей микропроцессоров. Познакомить с технологиями разработки программного обеспечения для встраиваемых устройств под управлением ОС GNU/Linux.

Задачей курса является ознакомление с современными микропроцессорами и технологиями разработки программного обеспечения для них.

Настоящее пособие содержит подробные инструкции по выполнению лабораторных работ на оборудовании компании «Элекард Девасез». Также в пособии даются темы для самостоятельной проработки, изучение которых позволит студентам закрепить материал теоретического курса и лабораторного практикума.

Лабораторная работа № 1. Командная оболочка bash. Создание и выполнение скриптов

Цель работы.

Научиться пользоваться командной оболочкой bash. Разобраться с принципом работы переменных окружения.

Теория.

Командный язык shell (в переводе - оболочка) фактически есть язык программирования очень высокого уровня. На этом языке пользователь осуществляет управление компьютером. Обычно, после входа в систему вы начинаете взаимодействовать с командной оболочкой (если угодно - она начинает взаимодействовать с вами). Признаком того, что оболочка (shell) готова к приему команд, служит выдаваемый ею на экран промптер (символ-приглашение). В простейшем случае это знак доллар ("$").

Обратите внимание: shell - это одна из многих команд UNIX. То есть в набор команд оболочки (интерпретатора) "shell" входит команда "sh" - вызов интерпретатора "shell". Первый "shell" вызывается автоматически при вашем входе в систему и выдает на экран промптер. После этого вы можете вызывать на выполнение любые команды, в том числе и снова сам "shell", который вам создаст новую оболочку внутри прежней.

Команды в shell обычно имеют следующий формат:

<имя команды> <флаги> <аргумент(ы)>

Например:

ls -ls /usr/bin

ls имя команды выдачи содержимого директория,

-ls флаги ("-" - признак флагов, l - длинный формат, s - об'ем файлов в блоках).

/usr/bin директорий, для которого выполняется команда.

Эта команда выдаст на экран в длинном формате содержимое директория /usr/bin, при этом добавит информацию о размере каждого файла в блоках.

К сожалению, такая структура команды выдерживается далеко не всегда. Не всегда перед флагами ставится минус, не всегда флаги идут одним словом. Есть разнообразие и в представлении аргументов. К числу команд, имеющих экзотические форматы, относятся и такие "ходовые" команды, как cc, tar, dd, find и ряд других.

Как правило (но не всегда), первое слово (т.е. последовательность символов до пробела, табуляции или конца строки) shell воспринимает, как команду. Поэтому в командной строке

cat cat

первое слово будет расшифровано shell, как команда (конкатенации), которая выдаст на экран файл с именем "cat" (второе слово), находящийся в текущем директории.

Управление заданиями в Bash

Одна из наиболее мощных возможностей Bash - это возможность помочь пользователю в управлении исполнением различных команд. Каждая команда имеет стандартный механизм для обработки ввода-вывода:

STDIN (стандартное устройство ввода) позволяет программе получать входные данные из командной оболочки.

STDOUT (стандартное устройство вывода) позволяет программе передавать свои выходные данные командной оболочке.

STDERR (стандартное устройство вывода ошибок) позволяет программе передавать командной оболочке информацию о возникающих ошибках.

Обычно STDIN - это клавиатура, с помощью которой передается информация команде (как это делается в строке приглашения Bash), а STDOUT и STDERR - это экран, на который выводятся результаты. Однако можно изменить этот механизм ввода-вывода, например, так, чтобы команда читала и записывала данные в файл.

Для изменения стандартного механизма ввода-вывода в Bash используются команды перенаправления:

Последовательность	Описание
command < file	перенаправляет STDIN на чтение из файла.
command > file	перенаправляет STDOUT на запись в файл.
command >> file	перенаправляет STDOUT на дозапись в файл.
command 2> file	перенаправляет STDERR на запись в файл.
command1 | command2	подсоединяет STDOUT команды command1 к STDIN команды command2.

Последний пункт в таблице известен как конвейер и используется для объединения команд. Объединение команд с помощью конвейера - это ключевая техника программирования в командной оболочке UNIX, использующаяся, как продемонстрировано на рисунке 2. В этой модели каждая команда выполняет одну простую задачу и передает результат своей работы другой команде, которая также выполняет простую задачу, и так далее. Например, можно с помощью программы cat передать через конвейер содержимое файла команде grep, которая через конвейер передает все найденные строки команде wc, которая подсчитывает и выводит на экран количество строк в файле, содержащих заданную подстроку.

Асинхронное выполнение

Исполнение команд, которое описывалось выше, является синхронным, то есть в каждый момент времени выполняется только одна программа. Иногда команда или программа должна работать в течение долгого времени. Чтобы не отказываться от интерактивного использования командной оболочки, можно выполнять команды асинхронно. Для этого нужно дополнить команду знаком амперсанда (&). Это указывает командной оболочке запустить команду в фоновом режиме, что позволит продолжить работу в Bash. Пример демонстрирует эту и другие техники управления заданиями.

```
rb$ grep paper.pdf /var/log/httpd/access.log | wc -l
5
rb$ python demo.py &
[1] 20451
rb$ jobs
[1]+  Running                 python demo.py &
rb$ fg 1
python demo.py
```

В первом примере команда grep ищет строку paper.pdf в лог-файле Web-сервера Apache Web. Результат работы этой команды передается через конвейер команде wc -l, которая считает количество строк. Таким образом, целиком команда подсчитывает число строк, содержащих paper.pdf в лог-файле.

Во втором примере программа на языке Python запускается в фоне для долговременной работы. Bash запускает это задание асинхронно в фоне и указывает идентификатор задания. Используя команду jobs, можно просмотреть список всех запущенных команд. В этом примере запущена только одна программа, она выводится из фоновой работы в синхронное выполнение с помощью команды fg.

Задание на выполнение.

1. Написать скрипт, который выводит список переданных ему аргументов в виде таблицы.
2. Написать скрипт, который передает часть аргументов другому скрипту через переменные окружения.
3. Написать скрипт, который выполняет собранное в нестандартной директории приложение с зависимостями, при этом дублирует функционал инициализационного скрипта (/etc/rc.d)..

Лабораторная работа № 2. Работа с Elecard STB SDK

Цель работы.

Получение навыков работы с Elecard STB SDK. Удаленная отладка приложений на приставке.

Теория.

Установка среды для сборки приложений STB.

Для установки среды потребуются следующие файлы:

- **eldk-4.0.tgz** – бесплатный пакет Embedded Linux Development Kit, содержащий компиляторы C/C++, утилиты и стандартные библиотеки;
- **stb810-SP7.2.003.elc.200806171335.tgz** – пакет компонентов, необходимых для разработки приложений для STB.

Архив eldk-4.0.tgz необходимо извлечь в директорию /opt. Например:

mkdir –p /opt

cd /opt

tar –zvxf <путь к eldk-4.0.tgz>

После выполнения данных команд в директории /opt появится директория eldk-4.0 и символическая ссылка eldk.

Далее, необходимо извлечь пакет stb810-SP7.2.003.elc.200806171335.tgz в директорию /opt/elecard (извлечение пакета надо проводить от имени root, иначе пакет не будет корректно распакован). Например:

mkdir –p /opt/elecard

cd /opt/elecard

tar –zvxf <путь к stb810-SP7.2.001.elc.tgz >

После выполнения этих команд в директории /opt/elecard появится директория stb810-SP7.2, в которой находится система сборки приложений для STB.

Изменять название папки stb810-SP7.2 нельзя, иначе система сборки приложений не будут работать.

Подготовка среды к компиляции.

Прежде чем приступить к компиляции приложений, необходимо настроить переменные окружения. Настройка производится при помощи следующих команд:

cd /opt/elecard /stb810-SP7.2

source ./setup.sh

Последняя команда выполняет инициализацию переменных окружения. Эту команду достаточно выполнять один раз за сессию.

Компиляция приложений.

Компиляция приложения StbMainApp возможна двумя путями:

1. Скриптом mkFirmware, который автоматически компилирует приложение, создаёт образы файловых систем и генерирует файл обновления. При использовании данного скрипта файлы приложения хранятся в файловой системе, которая записывается во Flash-память при помощи файла обновлений, а файлы конфигурации приложения хранятся в специальной файловой системе, монтируемой в директорию /config.
2. В ручную, при помощи утилиты prjmake. Этот метод следует использовать в случае загрузки по NFS для более удобной разработки приложений, однако для подготовки конечного варианта следует пользоваться первым вариантом.

Компиляция при помощи mkFirmware.

Для компиляции приложения данным методом достаточно выполнить следующую команду:

mkFirmware

При выполнении данной команды будет автоматически пере собрано приложение StbMainApp и помещено в основную файловую систему. Так же будут сгенерированы образы файловых систем и создан файл обновления вида STB820.dev.256.rev0001.<timestamp>.<hostname>.efp, где <timestamp> - это штамп времени когда был создан файл обновлений (этот же штамп является идентификатором обновления), а <hostname> - это имя компьютера, на котором был запущен скрипт. ВАЖНО! С генерированное данным скриптом обновление помечается как версия для разработки. Это значит, что при наличии этой прошивки на USB-Flash носителе STB будет обновляться вне зависимости от текущей установленной прошивки.

Для генерации релизного обновления необходимо задать следующие переменные окружения и запустить скрипт сборки

BUILD_MODE=release SYSREV=`cat ${BUILDROOT}/.version` mkFirmware

Для генерации обновления содержащего только скрипт, необходимо задать следующие переменное окружение и запустить (скрипт располагается в папке /opt/elecard/stb810-SP7.2/build_256M_NOIP/script/ , empty указывает на название вложенной папки со скриптом, сам файл скрипта должен обязательно называться update):

BUILD_SCRIPT_FW=empty BUILD_COMPONENT_FW=0

mkFirmware

Обновление содержащее только скрипт занимает мало места и обновление с помощью неё происходит гораздо быстрее. В скрипте можно описать действия выполняемые по изменению например config файлов (с помощью hwconfigmanager).

Для генерации комбинированного обновления (основное обновление и скрипт в одном файле) необходимо задать следующее переменное окружение:

BUILD_SCRIPT_FW=empty BUILD_COMPONENT_FW=1

mkFirmware

Переменное окружение задавать используя команду export.

Компиляция при помощи prjmake.

При использовании утилиты prjmake процесс компиляции может быть разбит на 3 этапа:

Очистка объектных файлов. Данная операция может быть полезна в случае, если изменения в коде не имеют эффекта при попытке компиляции приложения (например, изменения в файлах .h могут быть проигнорированы системой сборки).

prjmake StbMainApp.lnx.clean

Компиляция приложения. После компиляции бинарные файлы помещаются в специальную папку /opt/elecard/stb810-SP7.2/build_256M_NOIP/apps/StbMainApp/, из которой они позже могут быть скопированы в файловую систему.

prjmake StbMainApp.lnx.make

Копирование приложения и сопутствующих файлов в основную файловую систему (/opt/elecard/stb810-SP7.2/build_256M_NOIP/rootfs, откуда в последствии может загружаться по NFS STB, при сборке обновления файлы из этой папки попадают в файл обновления).

prjmake StbMainApp.lnx.install

Так же приложение может быть собрано следующими командами:

make clean

make

make install

Эти комманды необходимо вводить, находясь в директории с исходными кодами StbMainApp (/opt/elecard/stb810-SP7.2/src/project/elecard/apps/StbMainApp).

Загрузка по NFS.

Для разработки приложений удобней пользоваться загрузкой по NFS. Для этого необходимо, чтобы на персональном компьютере была установлена ОС Linux с поддержкой NFS, а в файле /etc/exports необходимо дописать:

/opt/elecard/rootfs_sp7.256

*(rw,no_root_squash,no_all_squash,sync,nohide,subtree_check,crossmnt)

По-умолчанию ядро Linux для STB настроено на загрузку с адреса 192.168.200.3 и пути /opt/elecard/rootfs_sp7.256. Адрес сервера и путь могут быть изменены в файле /opt/elecard/stb810-SP7.2/src/project/romfs/boot.xml в строке:

<param name="nfsroot" value="192.168.200.3:/opt/elecard/rootfs_sp7.256"/>

Для того, чтобы обновить образ ядра необходимо выполнить команду mkRomFs (предварительно должно быть подключено переменное окружение командой source ./setup.sh описанной выше, если ранее оно не было подключено) и затем обновить прошивку командой mkFirmware:

mkRomFs

mkFirmware

После этого необходимо вновь собранное обновление необходимо прошить в STB, в результате файл boot.xml на STB заменится на необходимый нам (если он был изменён). Загружаем STB, предварительно убедившись, что NFS сервер на ПК запущен (при подключении напрямую к компьютеру необходимо также использовать DHCP). В начале загрузки необходимо нажать любую клавишу (предварительно подключившись к STB описанными ниже методами), в результате должно появится список вариантов загрузки STB. Выбираем в данном списке нужный нам способ загрузки.

Задание на выполнение.

Создание и отладка тестовой программы «Hello world»для приставки.

Лабораторная работа № 3. Сборка модулей (драйверов) ядра

Цель работы.

Получение навыков работы с модулями (драйверами) ядра операционной системы Linux. Получение навыков обработки сообщений ядра.

Задание на выполнение.

1. Распаковка исходников ядра Linux. Обзор make системы сборки ядра. Создание makefil-а для модуля ядра. Сборка модуля и его установка в рабочую систему.

2. Просмотр информации о модуле (modinfo). Получение описания модулей, версии ядра и прочей информации. Загрузка модуля ядра с помощью утилиты insmod с учетом зависимостей. Прослеживание сообщений от загружаемого модуля ядра. Выгрузка модуля и его зависимостей.

3. Загрузка модуля ядра с помощью утилиты modprobe. Отслеживание загруженных модулей с помощью утилиты lsmod. Прослеживание сообщений от загружаемого модуля и его зависимостей.

4. Добавление вывода printk уровня ядра в модуль. Отслеживание добавленных сообщений уровня ядра в системном логе. Использование предопределенных макросов ядра LINUX_VERSION_CODE, KERNEL_VERSION и KERN_INFO.

Лабораторная работа №4. Реализация драйвера с обменом данными между пространствами ядра и приложений

Цель работы.

Получение и закрепление навыков разработки драйверов ядра, обмена данными с пользовательскими приложениями.

Задание на выполнение.

Реализовать драйвер с созданием файла в procfs. Драйвер должен обрабатывать чтение и запись файла.

1. Драйвер должен: хранить до 8-ми значений типа char, которые передаются при попытке записи, а при чтении файла драйвера возвращать запомненные значения.

Например: файл драйвера - /proc/buffer

user@mach~$ echo "H" > /proc/buffer

user@mach~$ echo "e" > /proc/buffer

user@mach~$ echo "l" > /proc/buffer

user@mach~$ echo "l" > /proc/buffer

user@mach~$ echo "o" > /proc/buffer

user@mach~$ echo "!" > /proc/buffer

user@mach~$ cat /proc/buffer

Hello!

user@mach~$ echo "1" > /proc/buffer

user@mach~$ echo "2" > /proc/buffer

user@mach~$ echo "3" > /proc/buffer

user@mach~$ echo "4" > /proc/buffer

user@mach~$ echo "5" > /proc/buffer

user@mach~$ echo "6" > /proc/buffer

user@mach~$ echo "7" > /proc/buffer

user@mach~$ echo "8" > /proc/buffer

user@mach~$ echo "9" > /proc/buffer

user@mach~$ cat /proc/buffer

23456789

2. Драйвер должен выполнять арифметические действия: сложение, вычитание и умножение. При записи в файл драйвера передается строка вида <операнд1> <оператор> <операнд2>, где <операнд1> и <операнд2> - положительные числа в десятичном представлении в пределах [0..99999999], <оператор> - '+', '-' или '*'. При чтении файла драйвера должен возвращаться результат операции.

Пример: файл драйвера - /proc/calc

*user@mach~$ echo "2 * 2" > /proc/calc*

user@mach~$ cat /proc/calc

4

*user@mach~$ echo "12345679 * 8" > /proc/calc*

user@mach~$ cat /proc/calc

98765432

3. Драйвер должен выполнять побитовые логические операции: | (или) и & (и). При записи в файл драйвера передается строка вида <операнд1> <оператор> <операнд2>, где <операнд1> и <операнд2> - положительные числа в шестнадцатеричном представлении (int - 4 байта, с обязательным префиксом 0x), <оператор> - '|' или '&'. При чтении файла драйвера должен возвращаться результат операции.

Пример: файл драйвера - /proc/logical

user@mach~$ echo "0x23 | 0x56" > /proc/logical

user@mach~$ cat /proc/logical

0x77

user@mach~$ echo "0x78 & 0xC5" > /proc/logical

user@mach~$ cat /proc/calc

0x40

Самостоятельная работа № 1. Установка ОС GNU/Linux на виртуальную машину

Необходимо подготовить доклад об известных дистрибутивах Linux. Возможности использования подобных систем во встроенных системах.

Самостоятельная работа № 2. Файловые системы Linux. Сравнение возможностей

Необходимо подготовить доклад об используемых файловых системах. Сделать их сравнительный анализ. Описать процесс монтирование файловых систем.

Самостоятельная работа № 3. Bash. Создание скриптов. Загрузочные скрипты

Необходимо подготовить доклад об использовании командной оболочки. Создание скриптов. Загрузочные скрипты.

Самостоятельная работа № 4. Процесс загрузки Linux. Initramfs, загрузка по сети

Необходимо подготовить доклад о возможностях удаленной загрузки Linux. Описать весь процесс подготовки и настройки.

Самостоятельная работа № 5. Кросс-компиляция программ

Этап 1.

Цель работы.

Ознакомиться с технологией создания мультимедийных приложений NDK фирмы NXP-Trident и получить навыки разработки TSSA модулей. Изучение примера программы, работающей на двух ядрах PNX8950 MIPS и TriMedia.

Задание на выполнение.

1. Создать TSSA модуль.
2. Создать приложение, использующее данный модуль.
3. Организовать пересылку команд между созданными модулями.

Этап 2.

Цель работы.

Получить навыки профилирования и усовершенствования TSSA модулей. Создание специальных вставок на ассемблере TriMedia.

Задание на выполнение.

1. Разработать ассемблерную функцию. Отладить ее на симуляторе и на устройстве.

2. Провести профилирование TSSA модуля. Найти его «тонкое» место. Выделить его в отдельную функцию. Запустить функцию на симуляторе. Ускорить ее написание ассемблера.

3. Провести профилирование результата на устройстве.

Самостоятельная работа № 6. Модули ядра.

Цель работы.

Закрепление навыков кросс-компиляции.

Получение навыков компиляции модулей ядра под кросс-системы и установки модулей в целевую файловую систему.

Получение навыков работы с регистрами (физической памятью) и портами.

Навыки работы с портами ввода/вывода общего назначения.

Задание на выполнение.

1. Кросс-компилирование модуля ядра реализующее символьное устройство «Hello world». Запуск на платформе DaVinci и отладка.

2. Кросс-компилирование модуля ядра, реализующего обмен данными между пространством приложений и пространством ядра при помощи файловой системы procfs. Запуск на платформе DaVinci и отладка.

3. Изучение блока портов ввода/вывода общего назначения (GPIO) процессора TI DaVinic. Реализация драйвера конфигурации произвольного GPIO. Задание параметров GPIO осуществляется через запись в файл устройства. Формат записи: conf <номер банка> <номер пина> <вход=0/выход=1> - конфигурация GPIO

set <номер банка> <номер пина> <выходное значение 0,1> - задание выхода

net <номер банка> <номер пина> - чтение входа

Например:

user@mach~$ echo «conf 2 5 0» > /dev/sample_gpio.

user@mach~$ echo «get 2 5» > /dev/sample_gpio
0