**Что это за документ**

Этот документ - отчет по работе, проведенной летом-осенью 2014 года. Работа состояла в запуске Linux на вверенной мне плате Zedboard и создании примеров работы с периферией по I2C и SPI.

Zedboard - отладочная плата на младшем кристалле семейства Zynq - Z7020.

Сайт проекта: <http://www.zedboard.org>

Официальная документация:

<http://www.zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf>

<http://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf>

**Подготовительный этап**

**Установка Linux на виртуальную машину**

Если в распоряжении есть десктоп на Linux или уже готовая виртуальная машина с Linux, то этот этап можно пропустить.

Иначе, установим виртуальную машину. Инструкция есть в документе:

*Zynq\_Linux\_Online\_v05\_0/Zynq\_Linux/lab\_instructions/Linux\_on\_Zynq\_2012\_Lab0\_1\_v05.pdf*

Она длинная, я ее тут приводить не буду. Курс взят по этой ссылке:

<http://www.zedboard.org/course/implementing-linux-zynq%C2%AE-7000-all-programmable-soc-ise-142>

Он для наших целей подходит не полностью, потому что построен на XPS, а не на Vivado, но, тем не менее, эта вещь - полезна. Конкретно туториал по настройке окружения нужно пройти полностью. Sourcery CodeBench я устанавливал в домашней директории (юзер у нас тут называется training).

**Форматирование SD-карты**

Я не буду описывать, как это делать. Это популярно написано в туториале *Ubuntu\_on\_Zedboard(...)* на странице 28.

**Построение аппаратной платформы**

За основу для аппаратной платформы я взял дизайн от фирмы Analog Devices, который они построили для демонстрации возможности присутствующего на плате HDMI-чипа ADV7511 их производства. Разумеется, можно построить и свой дизайн - это не очень трудно. Но незачем.

**Скачиваем исходники**

В соответствии с этим туториалом:

<http://wiki.analog.com/resources/fpga/xilinx/kc705/adv7511>

качаем исходники из Git-репозитория: <https://github.com/analogdevicesinc/hdl/tree/master/projects/adv7511/zed>

**Построение IP-ядер**

Для этого запускаем Vivado, в ней активируем TCL-консоль (внизу экрана). В консоли пишем:

*cd Your-path-to-repository/hdl-master/library/axi\_clkgen*

*source ./axi\_clkgen\_ip.tcl*

После того, как проект создан, его можно закрыть и поступить так же с каждой подпапкой в папке library: сделать навигацию в нее командой cd и выполнить лежащий в ней .tcl-скрипт.

**Построение платформы**

Теперь, когда мы построили ядра, надо построить основной проект. Это делается аналогично:

*cd Your-path-to-repository/hdl-master/projects/adv7511/zed/*

*source ./system\_project.tcl*

По идее, никаких проблем этот этап вызывать не должен. Скрипт сам построит дизайн, проведет синтез и имплементацию.

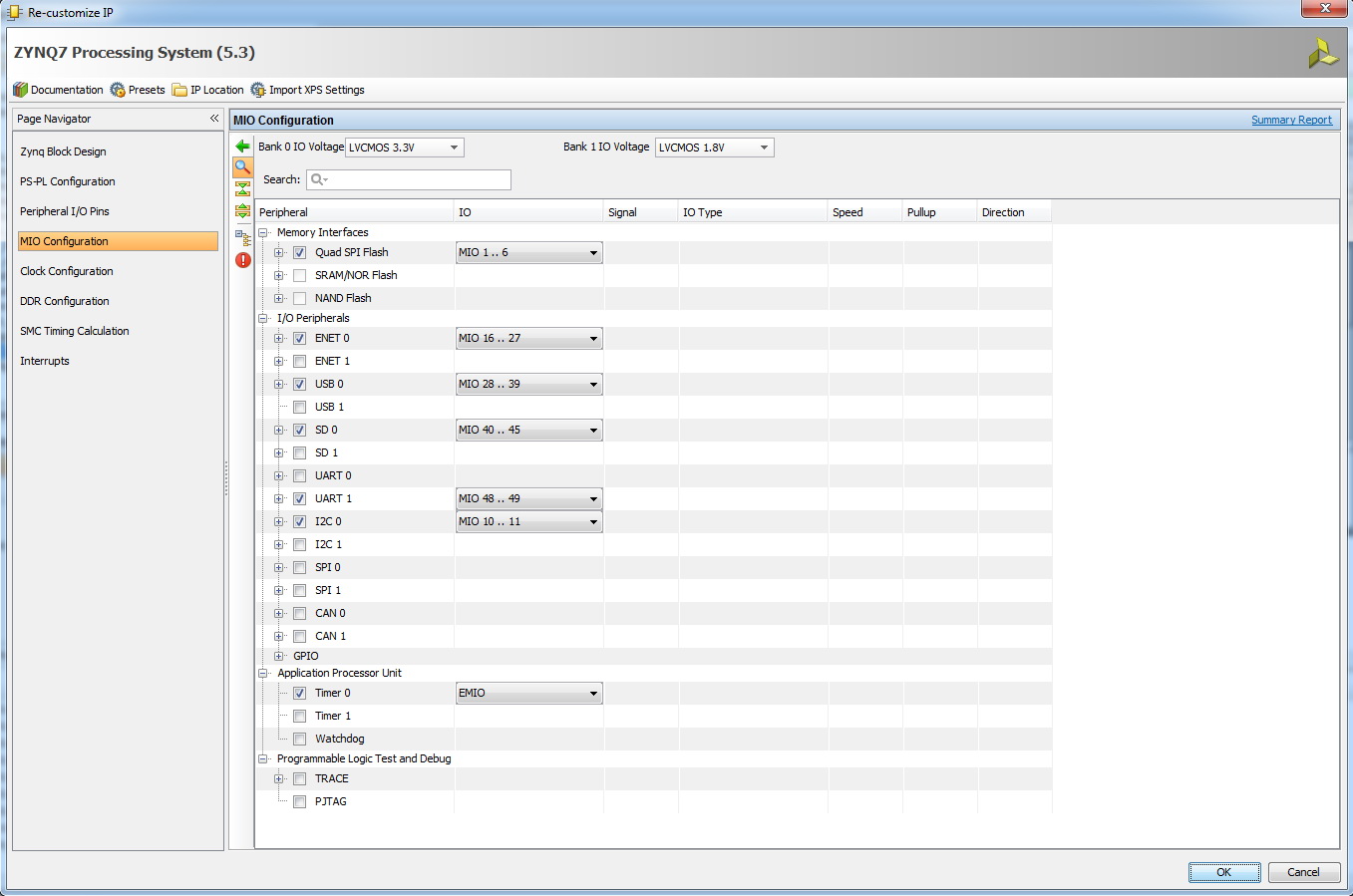
Теперь нам надо включить I2C-выход от процессорной системы (далее PS). Жмем дважды на блок с надписью Zynq (это собственно PS). Открываем вкладку MIO Configuration. Надо бы остановиться на том, что мы видим.

У PS есть два способа общения с внешним миром: MIO и EMIO.

MIO - это 50 ножек, которые идут наружу в обход ПЛИС. Они работают как ножки обычного микроконтроллера.

EMIO - это ножки, которые уходят в программируемую логику. Процессор обычно общается с IP-ядрами по шине AXI, ядра заменяют настоящую железную периферию.

Нам для функционирования шины I2C программируемая логика не нужна. Так что нужно включить I2C0 и назначить ее выходом, например, MIO 10-11. У меня конфигурация выглядит вот так (USB нужен для мышки и клавиатуры, которыми мы пользоваться не планируем - его можно выключить):



После этого нужно сгенерировать *.bit* (*Synthesis*, *Implementation* и *Generate .bit file*). Советую создать в папке проекта подпапку *bootfiles/* и скопировать *.bit* в нее.

Больше общаться с аппаратной платформой в рамках данного туториала нам не придется, Vivado можно закрыть.

Подробно почитать про то, как работают пины Zynq, можно прочитать здесь: <http://zedboard.org/content/zynq-pins-deep-dive>

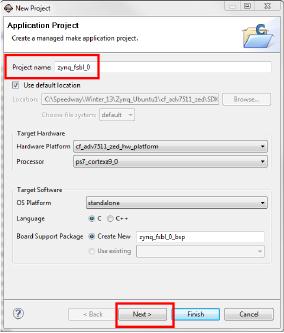
**Загрузчик**

**First Stage Bootloader (FSBL)**

*FSBL* - начальный загрузчик системы. Фактически, это что-то типа BIOS. Он прошивает логику .bit-ом и инициирует вторичный загрузчик *U-boot*.

Чтобы построить *FSBL*, жмем *File -> Export -> Export Hardware for SDK*. Оставляем все настройки как есть, ставим галочку Launch SDK, жмем OK.

Теперь запустится Xilinx SDK с папочкой в левом меню. Жмем *File->New->Application Project*. Назовем его *zynq\_fsbl*. Жмем *Next*.



Из списка выбираем *Zynq FSBL* и жмем *Finish*. Файл загрузчика создастся по этому пути:

*...<Project folder>\SDK\SDK\_Workspace\zynq\_fsbl\Debug\zynq\_fsbl\_0.elf*

Скопируем его в нашу папку *bootfiles* к *.bit*.

**Создание загрузчика U-boot**

Обычно, когда операционная система грузится, ей нужно выполнить несколько низкоуровневых аппаратно-зависимых операций, чтобы привести систему в функциональное состояние. Это операции типа инициализации регистров процессора, инициализации памяти, обнаружения периферии и активации кэша. На этом этапе ресурсов очень мало, так что FSBL должен быть очень компактным.

Когда процессорная система готова к работе, можно загрузить операционную систему. FSBL так сделать не может, поэтому в конце цикла работы он передает управление большому загрузчику U-boot (second stage loader). U-boot уже способен управлять периферией и делать еще кучу всего полезного.

Заметим, что эта часть делается в виртуальной машине Linux. Также заметим, что надо установить пакет *git* (в rpm-дистрибутивах это делается как *sudo yum install git*). Для этого 1) клонируем репозиторий U-boot фирмы Xilinx;

2) сконфигурируем исходники;

3) построим U-boot.

Все действия рекомендую выполнять в домашней директории (*cd ~*)

*$ git clone git://github.com/Xilinx/u-boot-xlnx.git*

*$ cd ~/u-boot-xlnx/*

*$ make distclean*

ВАЖНО!

Чтобы загрузка вообще состоялась, очень важно кое-что поправить в конфиге:

Нужно влезть в этот файл:

*/include/configs/zynq\_common.h*

и там сделать поиск по *sdboot*, после чего исправить абзац после этого слова на этот:

*"sdboot=echo Copying Linux kernel from SD to RAM ..RFS in ext4;" \*

*"mmcinfo;" \*

*"fatload mmc 0 0x3000000 ${kernel\_image};" \*

*"fatload mmc 0 0x2A00000 ${devicetree\_image};" \*

*"bootm 0x3000000 - 0x2A00000\0"* \

Как бороться с еррорами, если они возникают (у меня не возникли) - написано в документе *Ubuntu\_on\_Zedboard\_Tutorial\_v14.4\_02.pdf* на странице 25.

*$ make zynq\_zed\_config*

*$ make*

Теперь он будет долго собираться, в терминале будет много-много букв. Когда это закончится, в папке */u-boot/* лежит файл *u-boot*. Его надо скопировать в папку *bootfiles* в Windows, добавив расширение .elf.

**Построение .bin файла**

Нужно вернуться к SDK.

Там жмем правой кнопкой на наш проект *zynq\_fsbl* и выбираем пункт *Create Boot Image*. Для такого изображения нужно 3 файла (порядок важен): 1) *FSBL* 2) .*bit* 3) *u-boot*. Их нужно добавить в нижнюю менюшку. .*bif* - файл - это файл, который хранит информацию о положении этих трех. В самой нижней строчке настраивается положении будущего файла output.bin. Его надо переименовать в *BOOT.bin* и тоже поместить в *bootfiles*.

**Образ ядра**

Мы соорудили загрузчик. Теперь надо дать ему то, что он, собственно, будет грузить.

Писать конфиги нам необязательно: все нужные драйвера включаются автоматизированным скриптом настройки конфига *zynq\_xcomm\_adv7511\_defconfig*. В конфиги лезть не надо: можно что-нибудь сломать.

Важно: если путь к кросс-компилятору не будет найден, ничего не построится. Несмотря на то, что по идее у нас путь к кросс-компилятору должен быть прописан в *\$PATH* после прохождения туториала по настройке виртуальной машины, у меня он долго не мог найтись. Поэтому в командах, которые написаны ниже, нужно указывать путь к кросс-компилятору. Для меня */path/to/your/arm/cross-compiler = ~/Sourcery/bin/arm-xilinx-linux-gnueabi-.* Если есть ошибки в компиляции (у меня таковых было множество), то, скорее всего, проблема в том, что кросс-компилятор не подцепился.

Создадим образ ядра. Для этого в нашей виртуальной машине проделаем следующие команды в терминале:

*\$ cd ~*

*\$ git clone https://github.com/analogdevicesinc/linux.git*

*\$ cd linux*

*\$ git checkout xcomm\_zynq*

*\$ export ARCH=arm*

*\$ export CROSS\_COMPILE=/path/to/your/arm/cross-compiler*

*\$ make zynq\_xcomm\_adv7511\_defconfig*

*\$ make uImage LOADADDR=0x00008000*

*uImage* готов и ждет в папке *~/linux/arch/arm/boot*. Забираем его себе в *bootfiles*.

Теперь важный момент. В некоторых туториалах люди строят образ памяти *ramdisk*. Нам НЕ НАДО его строить. Чтобы при загрузке наше ядро его не искало, мы удалили адрес в конфиге u-boot (из было 3: *uImage, devicetree, ramdisk*; а оставили мы 2: *uImage* и *devicetree*).

**Device tree**

Теперь самое сложное. При кастомизации системы практически ничего из наших *bootfiles* не меняется: *u-boot* всегда будет таким же, uImage тоже, а .*bit* и *fsbl* строятся автоматизированно. Но device tree всегда придется переделывать вручную.

Вообще *devicetree* - это просто текстовый файлик, в котором фактически перечислены шины, подключенные к нашему процессору, и устройства, подключенные к шинам, с их bus addresses. Подробно прочитать про *device tree* можно вот здесь:

<http://xillybus.com/tutorials/device-tree-zynq-1>

Там научат их читать. Писать их я так и не научился: легче поковырять форумы, найти описание для определенного устройства и скопировать его себе.

К счастью, Xilinx предоставила нам целый набор device tree на все случаи жизни. Они уже есть в папке *linux*, в которой мы строили uImage. Путь: *~/linux/arch/arm/boot/dts*. У нас есть базовый файл *zynq.dtsi* (.*dtsi* - это типа .*h* в Си - он подключается к .*dts*), еще один .*dtsi* - *zynq\_zed.dtsi* и .*dts* - *zynq\_zed\_adv7511.dts*.

*.dts* файл требует компиляции в .*dtb*:

*\$ cd ~/linux*

*\$ make zynq\_zed\_adv7511.dtb*

Лежит он в той же директории, что и .*dts*. Его надо скопировать в *bootfiles*.

Важно: мой *zynq.dtsi* кастомизирован: там изменено описание IIC и QSPI. Лучше вообще не вдаваться в подробности, почему они такие, а просто взять готовое. Исходник приложен в папке *files* с этим документом.

**Root File System rootfs**

Теперь нам надо установить на SD карту файловую систему.

Сначала подключим карту к Windows и зальем в на нее *BOOT.bin*, *zynq\_zed\_adv7511.dtb* (переименовав его в *devicetree.dtb*) и *uImage*. Нет опасности повредить пользовательские файлы: из-под винды видно только загрузочный раздел.

Теперь приделаем ее к линуксовой виртуальной машине (*Player -> Removable Devices -> Integrated Mass Storage Device -> Connect*). Выполним в терминале вот эти команды:

*\$ wget http://releases.linaro.org/12.11/ubuntu/precise-images/ubuntu-desktop/linaro-precise-ubuntu-desktop-20121124-560.tar.gz*

*\$ sudo tar --strip-components=3 -C /media/rootfs -xzpf linaro-precise-ubuntu-desktop-20121124-560.tar.gz binary/boot/filesystem.dir*

*\$ ls /media/rootfs/*

*bin/ boot/ dev/ etc/ home/ lib/ lost+found/ media/ mnt/ opt/*

*proc/ root/ run/ sbin/ selinux/ srv/ sys/ tmp/ usr/ var/*

Выход от последней операции должен быть в точности как тут. Если карта оказывается защищенной от записи, а Lock на ней не стоит, то мы что-то не так отформатировали (см. Подготовительный этап).

**Greetings and welcome**

Теперь можно вставлять SD-карту в Zedboard, подключить USB-UART и включать питание. Должен загореться синенький ЛЕД, а в консоли COM-порта должен отражаться процесс загрузки.

Если все хорошо, то можно и Ethernet подключить. Пинг должен быть.

Надо установить пароль на root, обновить пакеты и список пакетов (если Ubuntu говорит, что надо обновить дистрибутив - это надо сделать, иначе не отстанет):

*\$ sudo passwd*

*\$ sudo apt-get update --fix-missing*

*\$ sudo apt-get upgrade*

SSH по идее должен работать "из коробки". Если не работает (как у меня), надо удалять и устанавливать заново пакеты *openssh-client*, *openssh-server*.

Для работы с i2c нам нужен пакет *i2c-tools*. Его надо *apt-get install*.

Теперь, наш Linux полностью функционален.

**Примеры работы с периферией**

Примеры содержатся в папке *files/src* с этим документом. Подпапка *for\_linux* содержит примеры работы с периферией под Linux, а подпапка *noOs* – без операционной системы.

Думаю, нужны объяснять, как их запускать, нет.

Готовые проекты для Linux и для standalone тоже приложены в папке *files/projects* (в них, в отличие от базового проекта от Analog Devices, приделано IP-ядро из нескольких регистров для тестирования записи/чтения по AXI) и правильно сконфигурированы выходы и .xdc - файл. Папку *library* не прилагаю: она такая же, как и в скачанном из Интернета проекте.

Вопросы, которые я задавал на форуме Xilinx, есть в моем профиле:

<http://forums.xilinx.com/t5/user/viewprofilepage/user-id/57767>

**No-OS**

Запускать bare-metal программу на Цинке можно при помощи создания загрузочного образа и загрузки с SD-карты, либо из SDK (Важно не забыть переставить джамперы на плате в соответствии с [этим](http://zedboard.org/content/zedboard-bring) руководством, которое по сути есть упрощенный документ Zedboard Hardware User Guide).

Алгоритм такой:

1. Когда есть блок-дизайн в Vivado, жмем File->Export->Export Hardware for SDK. Ставим галочки Launch SDK и Include bitstream.
2. Создание нового SW проекта в SDK – по нажатию File->New->Application Project. Там есть несколько готовых шаблонов, но можно сделать пустое приложение.
3. После того, как текст написан, правой кнопкой жмем на проект -> Build Project.
4. Запуск отладки: правой кнопкой на проекте -> Debug As -> Debug configuration. Дважды жмем на Xilinx C\C++ Application (System Debugger).
5. Появится окошко, как на скриншоте ниже. На скриншоте видно, как его настроить.
6. Жмем Debug.
7. SDK предложит запустить Debug perspective (перспективы меняются по нажатию кнопки в правом верхнем углу). Отладочная перспектива – обычный отладчик с брейкпоинтами и прочим.

Скриншот с настроенной Debug Configuration ниже.

Видео-урок о том, как запускать отладку на Zynq:

<http://www.youtube.com/watch?v=Gl8MaQUS1hg>

Загрузочный .bin файл создается так же, как для Linux, только вместо u-boot надо поставить запускаемый .elf-файл.

