Введение в ARM7. Урок 1

igorkov / 2008-06-20 / fsp@igorkov.org

Что дает ARM7

Первым делом, человек, приняв решение к освоению микроконтроллера, что-то ожидает от него, желает получить какие-то возможности. Думаю, для большинства, читающих этот материал ARM является не первым микроконтроллером и до этого был опыт работы с какими-то другими контроллерами, PIC или AVR к примему. По этой причине, я часто буду ссылаться на явные отличия архитектур и не буду углубляться в понятия, такие как прерывания, периферия, интерфейсы и их применение. Будут просто описываться их возможности и даваться примеры использования.

Так какие же возможности получает человек, переходящий на ARM (конкретно ARM7TDMI):

1. 32 битная архитектура. Дает 32 битные арифметические операции, 32 битную адресацию и эффективную реализацию алгоритмов.
2. Однотактное ядро. Т.е. минимальное число тактов на команду - 1. Обычно частота микроконтроллеров на ARM7TDMI около 60Мгц.
3. RISC система команд. Точнее 2 системы команд: ARM (32 битные команды), THUMB - упрощенные и укороченные команды по 16 бит. Соответственно фиксированная длина команд.
4. Фон-Неймановская архитектура памяти. Плоская модель памяти размером 4Гб. Очень удобно, в одном адресном пространстве лежит все: и оперативная память, и FLASH, и адреса периферии. Так же ядро не видит разницы из какой области выполнять код. Не знаю, можно ли выполнить его из области периферии, не пробовал, но вот из оперативной памяти или из FLASH-а разницы никакой (разве что в скорости, FLASH часто имеет бОльшие задержки).
5. Развитая периферия, но это уже особенности конкретных контроллеров.

Архитектура ARM7. Память, ядро.

Первым делом скажу. Основной источник информации по ядру, системе команд и синтаксису ассемблера для меня - ARM7TDMI Datasheet DDI0029E (ссылка внузу стать или легко найти на www.arm.com).

Как было сказано выше ARM7TDMI - это 32 битная RISC архитектура, включающая 2 системы команд (ARM и THUMB). Всего в ядре около 32 регистров, но не все из них доступны одновременно. В ARM имеется понятие о режиме работы. Таких режимов работы - 7 штук, это:

1. User (0x10) - обычный режим выполнения.
2. FIQ (0x11) - быстрое прерывание.
3. IRQ (0x12) - обычное векторное прерывание.
4. Supervisor (0x13) - программное прерывание.
5. Abort (0x17) - ошибка доступа к памяти.
6. Undefined (0x1C) - недопустимая инструкция.
7. System (0x1F) - в этом режиме находимся в момент, когда происходит сброс.

Переход между режимами осуществляется автоматически, при входе в прерывание и возврате из него. Так же при желании его можно переключить вручную модификацие статусного регистра. Статусный регистр - CPSR содержит флаги процесса работы (переполнение, перенос, ноль, знак), режим работы (то, что описано выше) и флаг набора команд.Работа с этим регистром рассмотрю позже, пока лишь

скажу, что в принципе непосредственно читать и модифицировать его почти никогда не требуется, редкий случай - вложенные прерывания.

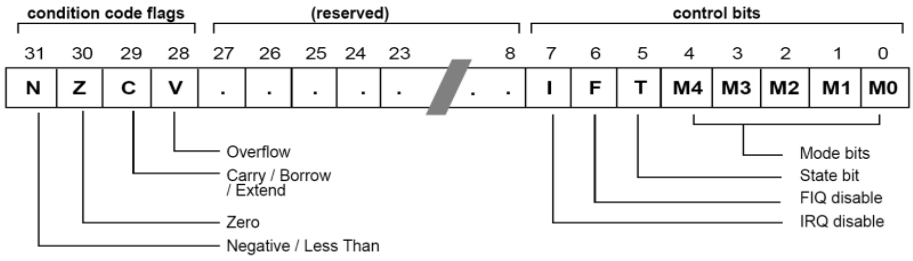

Рис. 1 - Регистр CPSR/SPSR

Теперь подробнее про режимы работы: вместе с переключением режима, переключается и часть банка регистров. Чтобы понять этот процесс посмотрим рис. 2[1]

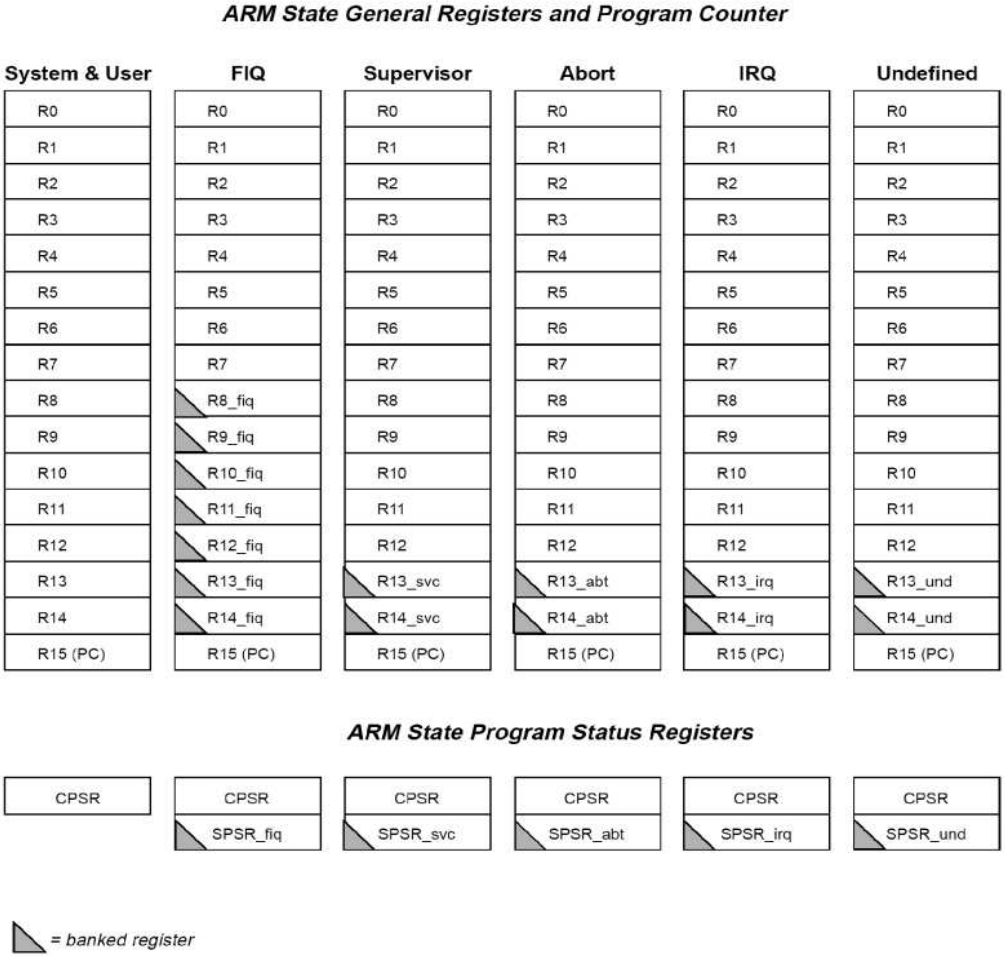

Рис. 2 – Регистр CPSR/SPSR

Непосредственно в каждом режиме работы имеется основной банк из 16 регистров: R0-R15 и статусный регистр CPSR. При написании кода обычно должно интересовать только 16 основных регистров и флаги из CPSR, необходимые для условных команд. Допустим мы находимся в основном режиме (User) и происходит прерывание, пусть это будет IRQ. Статусных регистр сохраняется в SPSR_irq и регистры R13, R14 заменяются на R13_irq, R14_irq. R14 – это регистр, используемый

[1] Рисунок взят из ARM 7TDMI Data Sheet (ARM DDI 0029E). Документ на сайте www.arm.com в настоящий момент найти не удается, в конце приведена ссылка на копию документа.

для хранения адреса возврата, точнее при вызове функции туда аппаратно может поместиться адрес возврата. R13 обычно (именно обычно, аппаратной привязки здесь нет) используется для хранения указателя стека.

В других режимах происходит такая же замена на соответствующие регистры, однако в FIQ-режиме меняется банк регистров R8-R14. Это позволяет оптимизировать функцию обработки FIQ прерывания и избавить ее от предватительного сохраненния регистров в стек для размещения локальных переменных, что уменьшит время реакции прерывания (потому и Fiq – Fast Irq).

Теперь о размещении программы. Таблица векторов прерываний располагается в начале FLASH по адресу 0x00000000. Там кладутся команды, осуществляющие прыжки в соответствующие точки входа. Таблица прерываний представлена на рисунке (рисунок взят из официального User Manual на LPC214x[2]):

Table 3. ARM exception vector locations

Address	Exception
0x0000 0000	Reset
0x0000 0004	Undefined Instruction
0x0000 0008	Software Interrupt
0x0000 000C	Prefetch Abort (instruction fetch memory fault)
0x0000 0010	Data Abort (data access memory fault)
0x0000 0014	Reserved **Note:** Identified as reserved in ARM documentation, this location is used by the Boot Loader as the Valid User Program key. This is described in detail in "Flash Memory System and Programming" chapter on page 295.
0x0000 0018	IRQ
0x0000 001C	FIQ

Рис. 3 – Таблица векторов прерываний

Можно заметить два интересных факта: первый, здесь нет никаких векторов прерываний от периферии. Только системные прерывания, такие как прерывание по ошибке доступа, некорректной инструкции, попытки выполения из недопустимой области памяти, программного прерывания, прерывания FIQ. Все прерывания периферии <<висят>> на векторе IRQ. В общем за прерывания периферии отвечает модуль, называемый VIC (векторный контроллер прерываний), так же у ST встречал название EIC (расширенный контроллер прерываний). Настройка прерываний периферии осуществляется путем конфигурации этого модуля.

Вторая особенность: вектор 0x00000014 свободен. Формально по описанию ядра ARM там ничего не определено, просто пусто. Но, допустим в NXP LPC по этому адресу хранит контрольную сумму таблицы прерываний и, если она не совпадает, ваше приложение не запустится. Но обычно все загрузчики знают об этом и модифицируют занчение по этому адресу.

По адресу 0x00000014 вектор прерывания отсутствует, однако NXP LPC2000 используют его для хранения контрольной суммы векторов прерывания. Причем если это значение будет установлено некорректно, тогда ваша программа банально не запустится. Констроллер будет вести себя так, словно в него ничего не загружено.

Остается еще одна важная особенность ядра: поддержка двух различных наборов команд. Это полноценные, очень гибкий набор ARM команд длиной 32 бита и упрощенный набор THUMB половинной длины. Соответственно второй имеет

[2] http://www.standardics.nxp.com/support/documents/microcontrollers/pdf/user.manual.lpc2141.lpc2142.lpc2144.lpc2146.lpc2148.pdf

огромное количество ограничений. Нет, THUMB код так же полнофункционален и позволяет реализовать любой алгоритм, просто порой для этого может потребоваться намного больше инструкций (следует учесть, что каждая инструкция имеет половинную длину, так что итоговый размер обычно на 20-30% меньше).

Фактически исполнение THUMB кода происходит так: контроллер памяти забирает инструкцию, потом она поступает на вход специального модуля, который «разворачивает» команду в набор обычную ARM и передает на исполнение.

Пара слов про ассемблер.

Здесь я сделаю небольшое введение в ассеблер ARM. Это пригодится, когда будем рассматривать исходники HelloWorld, а конкретно Startup-файла, в который вынесен некоторый код инициализации. Сразу поясню, выше были введены названия регистров R0-R15. однако 3 последних имеют псевдонимы, так как используются для определенных целей. Так R13 – обычно указатель стека имеет псевдоним SP. Регистр R14 – обычно хранит адрес возврата, и его псевдоним LR. R15 – счетчик команд, обзывается PC. Дальше рассмотрены только команды ARM-режима. THUMB я пока не затрагиваю, так как и сам никогда не писал на ассемблере в THUMB.

Про PC регистр стоит заметить, что он имеет значение не текущей выполняемой команды, не следующей, а <<Адрес текущей>> + 8. Это вызвано 3х ступенчатым конвеером в ARM7. То есть, если выполняется инструкция:

```
0x00000100: MOV R0, PC
```

то в R0 положится значение 0x00000108.

Теперь пройдемся по основным классам инструкций:

1. **Data Processing команды (обработка данных).** Архитектура ARM при обработке данных предусматриывает так называемый Load-and-Store подход (может немного не корректно, вроде сама подобная архитектура называется Load-and-Store). Это диктует нам следующую последовательность действий при обработке данных: сначала загрузили в регистры, затем обработали, затем сохранили обратно. Поэтому мы имеем достаточно большой банк регистров и команды обработки данных оперируют ТОЛЬКО с регистрами. Примеры команд:

```
ADD R0, R1, R2  //; R0 = R1 + R2
SUB R0, R1, R2  //; R0 = R1 - R2
MOV R0, R1      //; R0 = R1
ADD R0, R1, #10 //; R0 = R1 + 10
CMP R0, R1      //; R0 ? R1 -> CPSR
CMP R0, #10     //; R0 ? 10 -> CPSR

ADD R0, R1, R2, lsl #2 //; R0 = R1 + (R2 << 2)
ADD R0, R1, R2, lsl R3 //; R0 = R1 + (R2 << (R3 & 0x01F))

// Без констант и сдвигов:
MUL R1, R2, R3        //; R1    = R2 * R3
MULL R0, R1, R2, R3   //; R0:R1 = R2 * R3
```

Огорчает то, что на константы в командах есть очень специфичное ограничение: они должны быть представимы в виде: x*2^1/2/3/4/5/6.../31, где x в пределах от 0 до 255. У всех этих команд есть флажок, указывающих, будут ли по результату выполения модифицироваться флаги в CPSR.

2. **Single Data Load команды (загрузка данных).** Это команды, загружающие данные в регистр. Конкретно команда LDR/STR и ее вариации. Сразу стоит

сказать, в ARM отсутствует как таковая абсолютная адресация или адресация по константе. При задании адреса в DP-команде обязательно присутствует регистр и уже относительно него смещение. Смещения бывают разные. В общем они делятся на Immediate offset и Register Offset. Первый случай:

```
LDR R0, <R1, #4> //; R0 = <R1 + 4>
```

Здесь задано смещение +4 относительно значения в регистре R1. Значение такого смещения не должно превышать +/-2^12 (+/-4096). Очень часто встречается конструкция:

```
LDR R0, <PC, #xxx> //; R0 = <PC + xxx>
```

Это элементарная загрузка константы. Т.е. есть модуль на ассемблере или Си. В нем используются константы, которые сохраняются в конце модуля. Если размер модуля превышает 4к, то этот модуль разбивается, так как аппаратно не получается «дотянуться» (к слову, адрес переменной в RAM так же является константой и сохраняется компилятором в конце модуля). Но это специфика построения компиляторов. Нам же, надо просто знать, как загрузить какую-то константу. Для этого достаточно написать:

```
LDR R0, =0x12345678
```

при этом все дополнительные действия будут проведены автоматически.

Второй способ адресации – Register Offset. Здесь смещение задается через регистр, при этом значение во втором регистре может подвергнуться допомнительным преобразованиям (сдвигам). Вот примеры:

```
LDR R0, <R1, R2> //; R0 = <R1 + R2>
LDR R0, <R1, -R2> //; R0 = <R1 - R2>
LDR R0, <R1, R2, asl #2> //; R0 = <R1 + (R2 << 2)>
```

Так же при адресации мы можем записать значение смещение обратно (WriteBack-бит) в базовый регистр:

```
LDR R0, [R1, R2]! //; R0 = <R1+R2>, R1 = R1 + R2
```

И последний вариант с «пост» инкрементом адреса:

```
LDR R0, <R1>, R2 //; R0 = <R1>, R1 = R1 + R2
```

отличается он тем, что сначала загружается адрес из R1 и только потом происходит инкремент. Соответственно бит WriteBack здесь не актуален (установлен всегда).

Так же есть команды LDRH, LDRB для загрузки слова 16 бит и байта (8 бит).

Очень серьезное ограничение команд загрузки/сохранения данных - это требования к выравниванию слов (16 бит) и двойных слов (32 бита) по границам в 2 байта и 4 байта соответственно. Это может составить много неприятностей при портировании кода со структурами, где существуют привязки к размеру структур и просто это надо учитывать. Попытке чтения/записи невыровненных заначений ARM7 ядро не генерирует прерываний (data abort к примеру), а читает значение со сдвигом из выровненного слова. Данная особенность подробно описана в докуменатции на ядро (см. ссылки в конце документа) в разделе с описанием команд.

4. **Multiple Data Load команды (загрузка блока данных, стековые операции).** Еще один темный лес. Это набор команд для блочной загрузки данных: LDM/STM. То есть такие команды позволяют загрузить данные не в один регистр, а хоть сразу во все 16. Они не имеют развитой адресации, такой как SDP, грузят данные просто по адресу из регистра, однако есть 4 режима загрузки. Эти режимы достаточно муторные и в них легко путаться. Так как команды в основном используются для реализации стека, а иногда для загрузки блока данных, рассмотрю именно это их применение. Сначала стек:

```
STMFD SP!, {R4, R5, R8-R10}
....
LDMFD SP!, {R4, R5, R8-R10}
```

Этот код сначала сохранит в стеке значения R4, R5, R8-R10, а затем восстановит их обратно. При этом при реализации функций иногда делают так:

```
STMFD SP!, {R4, R5, R8-R10, LR}
....
LDMFD SP!, {R4, R5, R8-R10, PC}
```

Это сразу же осуществит и возвтрат из процедуры. «!» означает записать братно значение SP (иначе данные в стек попадут, а сам указатель стека не сместится).

Для работы с блоком данных можно использовать такой код:

```
//; В R1 адрес какого-то буфера из 4*4=16 байт, который загружаем,
а затем сохраняем
LDMIA R1, {R2-R5}
....
STMIA R1, {R2-R5}
```

Раз уж заговорили о стеке, сразу скажу каким образом передаются параметры в Си-коде процедурам: для передачи первых 4х параметров используются регистры R0-R3, все параметры после 4ого передаются через стек. В R0 происходит возврат значения. Так же стоит сказать, что для R0-R3 Си компилятор обычно предполагает, что эти регистры могут быть модифицированы в функции (даже есть функция вида void a(void);).

5. Branch команды (прыжки, вызовы функций). Команды ветвления, передачи управления, вызова функций. Первый класс относительные:

```
B label   //; относительный прыжек (+/- 32Мб) (PC = label).
BL label  //; вызов подпрограммы по относительному адресу (+/- 32Мб).
//; Помещает в LR адрес следующей за BL инструкции (LR = PC - 4, PC =
label).
```

Второй класс, по регистру, здесь всего одна команда BX (есть еще BLX с сохранением адреса возврата, но она не реализована в системе команд ARM7, на сколько мне известно, присутсвует только в ARM9/11):

```
BX R0     //; PC = R0
```

При этом длинный вызов процедуры надо оформлять так:

```
MOV LR, PC
BX  R0    //; PC = R0
```

6. **Другие команды.** К этому классу можно отнести команды MRS, MSR, которые предназначены для работы со статусным регистром и команду SWI, которая генерирует программное прерывание и может использоваться для реализации системных вызовов в своей ОС :)

Теперь основная особенность: ЛЮБАЯ команда поддерживает условное выполнение. Всего таких условий 16. Все условия можно найти в шпоргалке или справочнике.

Семейство микроконтроллеров NXP LPC от ARM7TDMI.

Вообще разработкой ядра и самой архитектуры занималась английская компания ARM Ltd. Сама она ничего связанного с контроллерами не производит, периферию не разрабатывает, а только сидит и лицензирует свои разработки другим производителям. И сказать очень успешно лицензирует. Решения на ARM выпускает не один десяток производителей, а то, что ядро стандартизировано, один и тот же компилятор может генерировать код для любого из чипов. Главное учесть специфику периферии и карты памяти.

Я умышленно ничего не говорил про карту памяти в ARM до этого момента, так как она достаточно отличается у разных произовдителей. Теперь, когда в плотную подошли к конкретным контроллерам, можно рассказать и про нее. Конкретно все что будет ниже будет применительно к микроконтроллерам NXP LPC.

Сначала начнем с распределением памяти, на рисунке представлена карта памяти LPC214x, однако на других контроллерах семейства она не сильно отличается, во всяком случае совпадают адреса периферии, адрес RAM и FLASH.

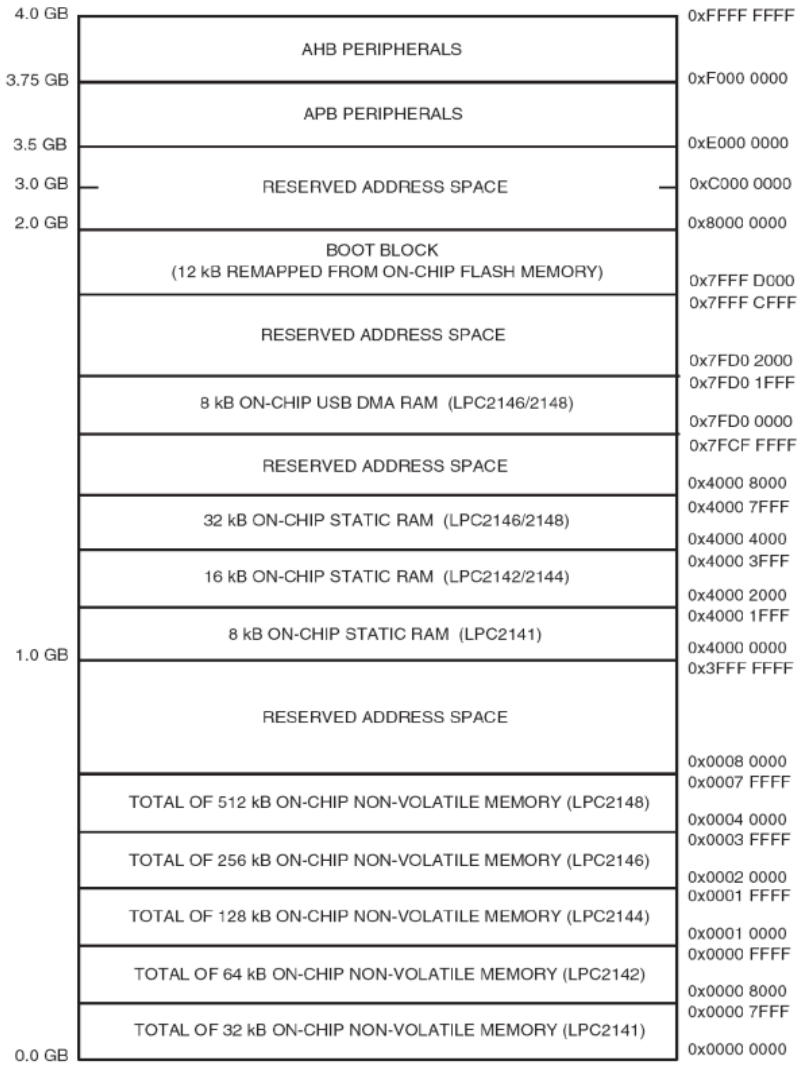

Рис. 2 – Карта памяти NXP LPC214x

В двух словах: в верхней части адресного пространства лежит вся периферия, в самом начале – FLASH, а по адресу 0x40000000 начинается RAM.

Важное замечание по хранению WORD/DWORD значений в оперативной памяти. Порядок следования байтов аналогичен таковому в x86, т.е. имеем little-endian. Это означает, что в памяти сначала хранится младший байт числа и далее по возрастанию. Например, имеем 0x12345678, лежащее по адресу 0x04. В памяти оно расположится так (адрес: значение байта): 0x04: 0x78 | 0x05: 0x56 | 0x06: 0x34 | 0x07: 0x12.

Теперь краткие технические характеристики LPC214x:

1. Ядро: ARM7TDMI.
2. Максимальная тактовая частота ядра: 60Мгц.
3. Размер основного ОЗУ: 8/16/32к, в зависимости от модели. LPC2146/8 имеют 32к, LPC2142/44 – 16к, LPC2141 – 8к.
4. Размер FLASH: 32/64/128/256/512к (LPC2141/2/4/6/8 соответственно).
5. Периферия:
   5.1. 2 UART;
   5.2. SPI;
   5.3. SSP;
   5.4. USB (LPC2146/8 с DMA, которое имеет дополнительные 8к ОЗУ);

5.5. I²C;
5.6. 2 32х битных счетчика/таймера;
5.7. ADC (аналогово-цифроовой преобразователь), у старших есть DAC (цифро-аналоговый преобразователь);
5.8. RTC с возможностью тактирования как от отдельного, так и от основного кварца;
5.9. PLL модуль;
5.10. GPIO, всякие WDT, PWM и т.д. прилагаются.

6. Как отдельную периферию можно выделить VIC.
7. Корпус: LQFP-64.

Сразу же хочу сказать основные отличия от, допустим, ATMEL AVR семейства:

1. Напряжение питания только 3.3В.
2. Наличие PLL модуля. PLL (Phase Locked Loop) – фазовая автоподстройка частоты. Модуль PLL позволяет имея кварц на 12Мгц получить частоты ядра 12, 24, 48, 30, 60Мгц (в ряде случаев можно настроить и на большую, у меня, к примеру, прекрасно работало на 72Мгц). Встроенные генераторы отсутствуют.
3. Так же отсутствует отдельное понятие битов конфигурации. Некоторое, что конфигурируется в AVR с помощью FUSE-ов, здесь нужно конфигурировать при инициализации (например задать частоту работы). Такая функция как защита FLASH-а реализуется путем записи вектора 0x87654321 по адресу 0x1FC во FLASH.
4. Интересная особенность: периферия тактируется через делитель. Обычно устанавливаю частоту периферии равную частоте ядра, но возможно разделить ее на 2 или 4. Основные цели – экономия энергии.
5. Программирование. Интересная особенность всех ARM. На сколько я знаю, нет способов (во всяком случае public) прошить контроллер используя какие-то его аппаратные средства (по аналогии ISP у AVR). Например, последовательный ISP интерфейс загрузки в LPC реализуется с помощью заранее прошитой микропрограммы (загрузчика). А как же JTAG, спросите вы? А JTAG FLASH писать не умеет, во всяком случае непосредственно. То есть мы можем прочитать, записать любой адрес, НО не FLASH. Для записи FLASH нужны дополнительные пляски. Само прошивание по JTAG работает следующим образом: в ОЗУ помещается программка, буфер и происходит передача управления этой программке, она в свою очередь делает IAP вызов того же загрузчика (IAP – In Application Programming). NXP вообще не разглашает секретов работы с FLASH в своих ARM контроллерах, оставляя только IAP-вызовы (но никто не мешает дизассеблировать загрузчик, особенно учитывая то, что снять его дамп нет никаких проблем).

P.S. Все картинки и таблички нагло сперты из приложенной к статье документации.

Ссылки

1. Оригинал статьи:
   http://igorkov.org/pdf/arm1.pdf
2. Подробное описание ARM7TDMI (ARM7TDMI Datasheet DDI0029E):
   http://igorkov.org/pdf/arm1_arm7tdmi.pdf
3. User Manual для LPC214x:
   http://www.nxp.com/acrobat/usermanuals/UM10139_1.pdf
4. Шпоргалка по командам ARM:
   http://igorkov.org/pdf/arm1_instrset.pdf