СЕРИЯ
МИРОВАЯ ЭЛЕКТРОНИКА

Тревор Мартин

МИКРОКОНТРОЛЛЕРЫ ARM7 СЕМЕЙСТВ LPC2300/2400

Вводный курс разработчика

Перевод с английского
Евстифеева А. В.

Москва
Издательский дом «Додэка-XXI»
2010

УДК 621.316.544.1(035.5)
ББК 32.844.1-04я2
М29

Данное издание подготовлено к печати при участии студентов и преподавателей Уральского Государственного Политехнического университета (УПИ) Агафонова С. А., Букрина И. В., Гусева А. В. при финансовой поддержке Российского представительства компании NXP.

Мартин, Тревор

М29 Микроконтроллеры ARM7 семейств LPC2300/2400. Вводный курс разработчика / Тревор Мартин ; пер. с англ. Евстифеева А. В. — М. : Додэка-XXI, 2010. — 336 с.: ил. — (Серия «Мировая электроника»). —
ISBN 978-5-94120-241-6

Книга представляет собой введение в архитектуру процессора ARM7 TDMI и микроконтроллеров семейств LPC2300 и LPC2400 компании NXP. В ней рассматриваются следующие вопросы: введение в ядро ARM7, средства разработки программного обеспечения, системная архитектура и периферийные устройства микроконтроллеров LPC2300/2400. Кроме того, в книгу включено полное учебное пособие, где на практических примерах закрепляются вопросы, изложенные в основном тексте. Изучая теоретический материал и выполняя сопутствующие упражнения, вы быстро освоите ядро ARM7 и микроконтроллеры семейств LPC2300/2400.

На компакт-диске, прилагаемом к книге, имеются справочные материалы и руководства пользователя по микроконтроллерам семейств LPC2300 и LPC2400, исходные коды всех упражнений и другие полезные материалы.

Предназначена для разработчиков радиоэлектронной аппаратуры, инженеров, студентов технических вузов и радиолюбителей.

УДК 621.316.544.1(035.5)
ББК 32.844.1-04я2

Оригинальное название «The Insider's Guide To The NXP LPC2300/2400 Based Microcontrollers. An Engeneer's Introduction To The LPC2300 & LPC2400 Series».

ISBN 978-5-94120-241-6 (рус.)
ISBN 0-95499-883-9 (англ.)

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Эта книга представляет собой практическое руководство для тех, кто собирается использовать в своих новых разработках те или иные микроконтроллеры семейств LPC2300 и LPC2400 компании NXP Semiconductors. Данная книга не является ни справочником, ни учебным пособием. Предполагается, что читатель имеет некоторый опыт в области программирования микроконтроллеров для встраиваемых систем и знаком с языком программирования Си. Основной объем технической информации содержится в четырех первых главах книги, поэтому если вы совершенно не знакомы с семействами LPC2300/2400 и, в частности, с процессорным ядром ARM7, вам необходимо внимательно прочитать эти главы. Обращаю ваше внимание, что в книге речь пойдет, в основном, о семействе LPC2300, а особенности, присущие семейству LPC2400, будут отмечены особо.

В первой главе рассматриваются основные характеристики процессорного ядра (ЦПУ) ARM7. После прочтения этой главы вы будете знать достаточно, чтобы начать писать программы для любых устройств, построенных на базе ядра ARM7. Если же вы хотите расширить свои знания, к вашим услугам имеется несколько прекрасных книг, описывающих эту архитектуру, часть из которых указана в списке литературы. Во второй главе рассказывается о том, как следует писать программы на языке Си для процессора ARM7. По существу, в этой главе описываются специфические расширения стандарта ANSI C, требуемые для программирования встраиваемых систем. Все примеры, встречающиеся в книге, были написаны с использованием коммерческого компилятора, однако в настоящее время на платформу ARM перенесен и бесплатный пакет программ GCC.

После прочтения первых двух глав книги вы должны хорошо разбираться в процессоре и средствах разработки для него. Третья глава посвящена системной периферии семейства LPC2300. В ней рассказывается о системной архитектуре микроконтроллеров семейства и рассматривается вопрос конфигурирования микросхем для достижения наибольшей производительности. В четвертой главе мы познакомимся со встроенными периферийными устройствами этих микроконтроллеров и узнаем, как их необходимо конфигурировать при использовании в своих программах. В пятой главе описываются более сложные периферийные устройства, такие как контроллер USB и контроллер Ethernet. Шестая и седьмая главы книги посвящены двум популярным операционным системам реального

времени для микроконтроллеров с ядром ARM — платной, разработки компании Keil, и бесплатной FreeRTOS.

На протяжении всей книги вам будут встречаться различные упражнения, которые подробно рассматриваются в восьмой главе, посвященной практическим занятиям. Все эти упражнения можно выполнить, используя ознакомительные версии компилятора и симулятора, имеющиеся на компакт-диске, который прилагается к книге. В продаже также имеется недорогой стартовый набор разработчика (starter kit), используя который вы можете загрузить учебную программу в реальный микроконтроллер и удостовериться, что она действительно работает. Я искренне надеюсь, что, читая эту книгу и выполняя упражнения, вы быстро освоите микроконтроллеры семейств LPC2300 и LPC2400.

ГЛАВА 1

ПРОЦЕССОРНОЕ ЯДРО ARM7

1.1. Основные положения

Все микроконтроллеры семейства LPC2300 построены на основе ЦПУ ARM7. Вообще говоря, чтобы использовать эти микроконтроллеры, вам совершенно не нужно быть экспертом в области программирования процессора ARM7, поскольку заботу о большинстве сложных моментов берет на себя компилятор языка Си. Тем не менее, чтобы разработать надежное устройство, вы должны иметь хотя бы общее представление о том, как работает ЦПУ и какие у него имеются особенности.

В этой главе мы рассмотрим основные характеристики ядра ARM7 вместе с его моделью программирования, а также обсудим набор команд, используемый данным процессором. В результате вы получите всю необходимую информацию о процессоре, являющемся «сердцем» семейства LPC2300. Для более углубленного изучения процессоров ARM рекомендую обратиться к книгам, указанным в списке литературы.

Ключевой принцип, лежащий в основе процессора ARM, — простота. Ядро ARM7 является RISC-машиной, предполагающей использование небольшого числа команд и соответственно состоящей из относительно небольшого количества логических элементов. Благодаря этому процессор ARM7 идеально подходит для использования во встраиваемых системах. Он имеет высокую производительность, низкое энергопотребление и занимает небольшую часть общей площади кристалла.

1.2. Конвейер

Основной элемент ЦПУ ARM7 — конвейер команд, который используется для обработки команд, считанных из памяти программ. Конкретно, в ядре ARM7 реализован трехступенчатый конвейер (**Рис. 1.1**).

Трехступенчатый конвейер является самой простой разновидностью конвейеров и не подвержен возникновению различных опасных ситуаций, таких как «чтение раньше записи», которые встречаются в конвейерах с бóльшим числом ступеней. Конвейер имеет три аппаратно-независимые ступени, благодаря кото-

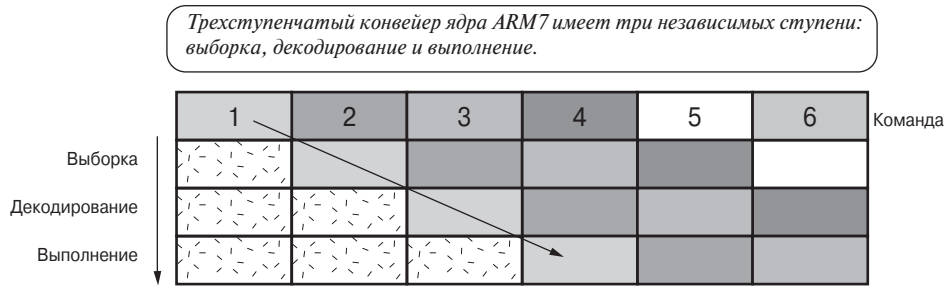


Рис. 1.1. Работа трехступенчатого конвейера.

рым одновременно с выполнением одной команды осуществляется декодирование второй и выборка третьей. Он настолько эффективно ускоряет прохождение команд через ЦПУ, что большинство команд ARM может выполняться за один такт. Конвейер наиболее эффективен при выполнении линейного кода. При обнаружении перехода конвейер сбрасывается, и для возобновления выполнения программы с максимальной скоростью он должен сначала заполниться. Позже мы с вами увидим, что набор команд процессора ARM имеет несколько интересных особенностей, позволяющих исключить из кода короткие переходы для улучшения прохождения кода по конвейеру. Поскольку конвейер является составной частью ЦПУ, он полностью скрыт от программиста. Тем не менее, важно помнить, что значение счетчика команд (Program Counter — PC) на 8 байт превышает значение адреса текущей выполняемой команды. В связи с этим необходимо аккуратно подходить к вычислению смещений в случае относительной адресации с использованием счетчика команд.

Например, команда:

```
0x40000     LDR     PC,[PC,#4]
```

загрузит в счетчик команд PC содержимое, находящееся по адресу PC + 4. Поскольку PC опережает текущую команду на 8 байт, в него будет загружено содержимое по адресу 0x400C, а не 0x4004.

1.3. Регистры

Процессор ARM7 имеет архитектуру «load-and-store» (загрузка — сохранение), поэтому для выполнения любой обработки данных необходимо сначала перенести эти данные из памяти в определенные регистры, выполнить команду обработки данных и затем записать полученные значения обратно в память (**Рис. 1.2**).

Основной регистровый файл состоит из 16 пользовательских регистров R0...R15 (**Рис. 1.3**). Каждый из этих регистров является 32-битным[1)]. Регистры

[1)] В отечественной литературе принято пользоваться понятиями «разряд», «разрядный». Двоичный разряд — это бит. В данном издании мы будем придерживаться зарубежной терминологии («бит», «битный»), что более соответствует современной тенденции в цифровой технике. — *Примеч. пер.*

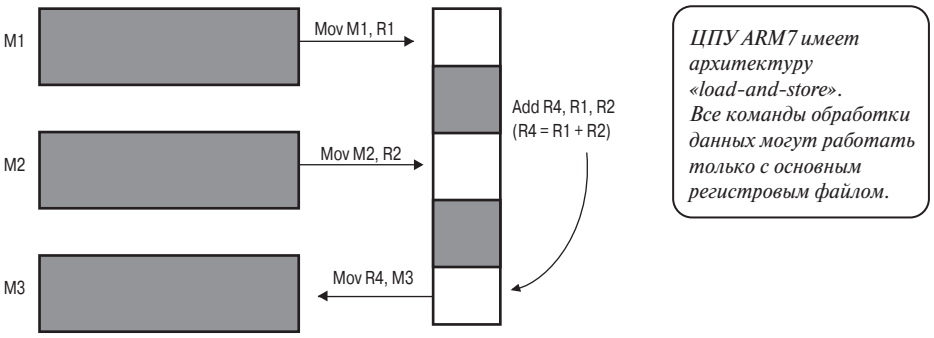


Рис. 1.2. Обработка данных.

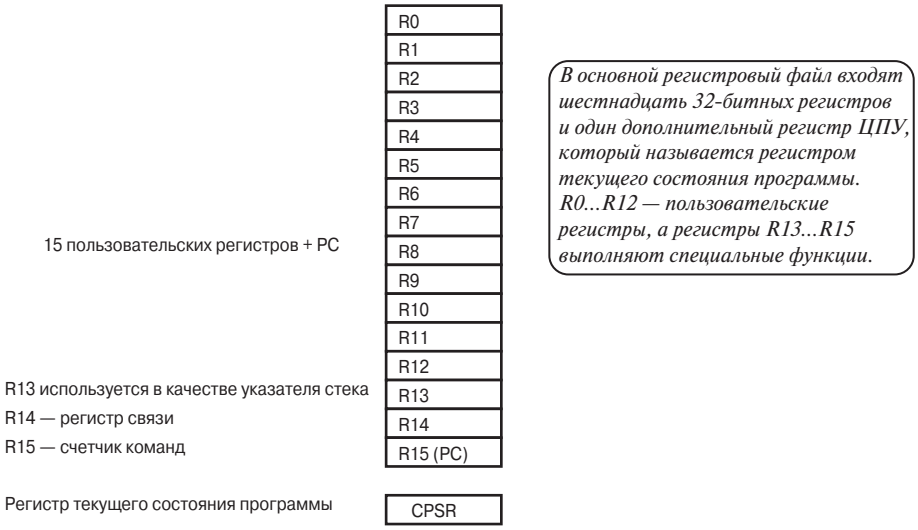


Рис. 1.3. Структура основного регистрового файла.

R0...R12 предназначены исключительно для нужд пользователя и не выполняют никаких других функций, в то время как регистры R13...R15 имеют дополнительные функции. Регистр R13 используется в качестве указателя стека (Stack Pointer — SP). Регистр R14 называется регистром связи (Link Register — LR). При вызове подпрограммы адрес возврата автоматически запоминается в регистре связи, откуда затем считывается при возврате. Такое решение позволяет быстро переходить к «концевым» функциям (функции, которые не вызывают других функций) и возвращаться из них. Если же функция входит в состав «ветви», т.е. вызывает другие функции, содержимое регистра связи необходимо сохранять в стеке (R13). Наконец, регистр R15 выполняет функции счетчика команд (PC). Что интересно, многие команды могут работать с регистрами R13...R15, как с обычными пользовательскими регистрами.

1.4. Регистр текущего состояния программы

Наряду с банком регистров в ЦПУ имеется дополнительный 32-битный регистр, который называется регистром текущего состояния программы (Current Program Status Register — CPSR). Регистр CPSR содержит набор флагов, которые управляют функционированием ЦПУ ARM7 и отображают его состояние (**Рис. 1.4**).

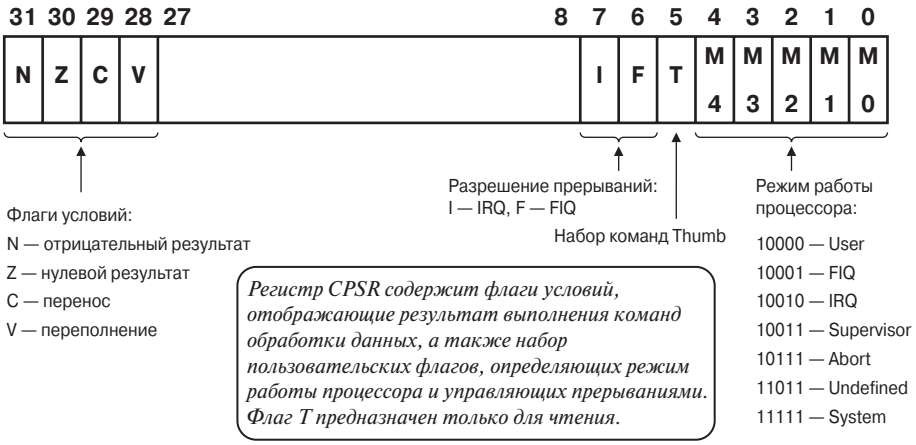


Рис. 1.4. Регистр текущего состояния программы.

В старших четырех битах регистра CPSR хранятся флаги условий, значения которых определяются процессором. Эти флаги отражают результат выполнения очередной команды обработки данных. Благодаря им вы можете узнать, не было ли получено в результате выполнения команды отрицательное или нулевое значение, а также не произошел ли перенос или переполнение. Младшие восемь битов регистра CPSR содержат флаги, значения которых прикладная программа может изменять. Биты 7 и 8 являются флагами I и F соответственно. Эти флаги используются для разрешения и запрещения двух линий прерываний, являющихся внешними по отношению к ЦПУ ARM7. Как мы с вами увидим позже, все периферийные модули микроконтроллеров LPC2300 подключены к этим двум линиям прерываний. При работе с данными битами нужно соблюдать осторожность, поскольку для запрещения любого из источников прерываний в соответствующий бит необходимо записать 1, а не 0, как можно было бы предположить. Пятый бит регистра является флагом THUMB.

ЦПУ ARM7 поддерживает два набора команд — 32-битный набор команд ARM и 16-битный набор команд THUMB. Соответственно флаг T показывает, какой из наборов команд используется. Учтите, что программа не должна пытаться напрямую устанавливать или сбрасывать этот флаг для переключения между наборами команд. Корректный механизм смены текущего набора команд мы рассмотрим чуть позже. Последние пять битов регистра CPSR являются флагами режима. В общей сложности процессор ARM7 поддерживает 7 режимов работы.

Прикладные программы, как правило, выполняются в режиме User. В этом режиме программа может обращаться к регистрам R0...R15 и CPSR. Однако при возникновении исключительных ситуаций (таких как прерывание, ошибка памяти или выполнение команды генерации программного прерывания) режим работы процессора изменяется. При этом регистры R0...R12 и R15 остаются теми же самыми, а регистры R13 (SP) и R14 (LR) заменяются новой парой регистров, уникальной для каждого режима. Таким образом, каждый режим имеет собственные регистр связи и указатель стека. Более того, в режиме быстрых прерываний (FIQ) дублируются и регистры R7...R12. Это позволит вам сразу же приступить к обработке прерывания FIQ, не тратя время на сохранение регистров в стеке.

В каждом из режимов, за исключением режима User, имеется дополнительный регистр, называемый регистром сохраненного состояния программы (Saved Program Status Register — SPSR). Если в момент возникновения исключительной ситуации программа находилась в режиме User, происходит смена режима и текущее содержимое регистра CPSR сохраняется в регистре SPSR. После обработки исключительной ситуации (при возврате из обработчика) содержимое регистра CPSR восстанавливается из SPSR, обеспечивая возобновление выполнения прикладной программы. Режимы работы процессора показаны на **Рис. 1.5**.

ЦПУ ARM7 имеет 6 различных рабочих режимов, которые используются для обработки исключительных ситуаций. Затененные регистры представляют собой дублирующие регистры, которые «включаются» при изменении режима работы. Регистр SPSR используется для сохранения содержимого регистра CPSR при переключении режимов.

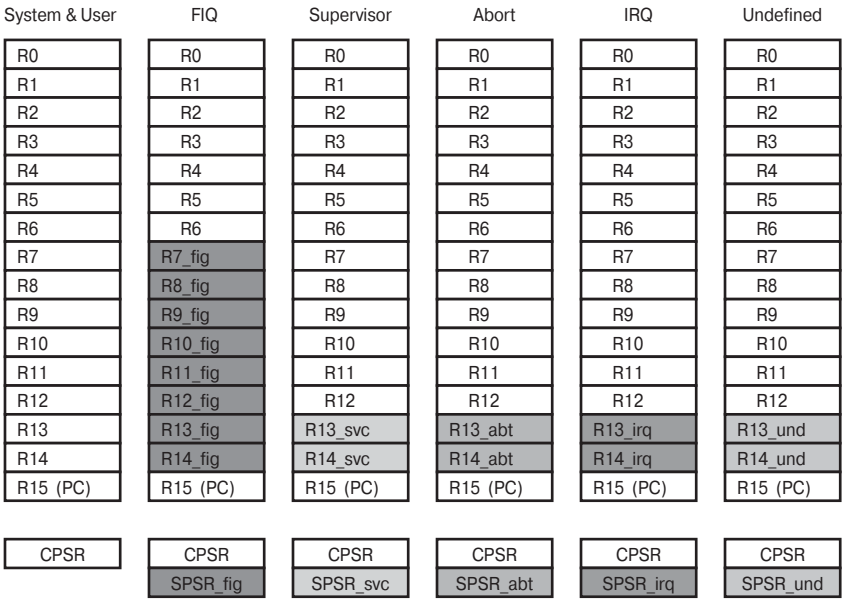


Рис. 1.5. Режимы работы процессора.

1.5. Режимы обработки исключительных ситуаций

При возникновении исключительной ситуации изменяется режим работы ЦПУ, и в PC загружается адрес соответствующего вектора прерывания (**Табл. 1.1**). Таблица векторов начинается с нулевого адреса, первым в таблице расположен вектор сброса, а за ним остальные векторы (по 4 байта на каждый).

Таблица 1.1. **Адреса векторов прерываний**

Исключительная ситуация	Режим	Адрес вектора
Reset (сброс)	Supervisor	0x00000000
Undefined instruction (неопределенная команда)	Undefined	0x00000004
SWI (программное прерывание)	Supervisor	0x00000008
Prefetch Abort (ошибка обращения к памяти при выборке команды)	Abort	0x0000000C
Data Abort (ошибка обращения к памяти при доступе к данным)	Abort	0x00000010
IRQ (прерывание)	IRQ	0x00000018
FIQ (быстрое прерывание)	FIQ	0x0000001C

Каждому режиму работы соответствует свой вектор прерывания. При смене процессором режима производится переход по этому вектору. Обратите внимание! Вектор по адресу 0x00000014 отсутствует!

Замечание. В таблице векторов имеется «дырка», поскольку вектор с адресом 0x00000014 отсутствует. Этот адрес использовался в ранних версиях процессоров ARM, а в процессоре ARM7 он сохранен, чтобы обеспечить программную совместимость между различными архитектурами ARM. Однако, как мы увидим позже, в микроконтроллерах семейства LPC2300, эти четыре байта играют очень важную роль.

При одновременном возникновении нескольких исключительных ситуаций используется метод приоритетов. Приоритеты прерываний приведены в **Табл. 1.2**.

Таблица 1.2. **Приоритеты прерываний**

Приоритет	Исключительная ситуация
Высший 1	Reset
2	Data Abort
3	FIQ
4	IRQ
5	Prefetch Abort
Низший 6	Undefined instruction SWI

Каждый источник исключительной ситуации имеет фиксированный приоритет. Встроенные периферийные устройства обслуживаются прерываниями FIQ и IRQ. Приоритеты прерываний от периферийных устройств можно назначать внутри этих групп.

Когда возникает исключительная ситуация, например, прерывание IRQ, процессор выполняет следующие действия (**Рис. 1.6**). Во-первых, адрес следующей выполняемой команды (PC + 4) сохраняется в регистре связи. Затем регистр CPSR копируется в регистр SPSR конечного режима (в нашем случае SPSR_irq). После этого в PC заносится адрес вектора прерывания режима исключительной ситуации. Для режима IRQ этот адрес — 0x00000018. В то же время режим работы процессора меняется на IRQ, в результате чего регистры R13 и R14 заменяются соответствующими регистрами этого режима. При входе в режим IRQ устанавливается флаг I регистра CPSR, что приводит к отключению линии IRQ. Если требуется использовать вложенные прерывания, то вы должны вручную разрешить

прерывание IRQ в программе и занести содержимое регистра связи в стек, чтобы сохранить исходный адрес возврата. С вектора прерывания программа перейдет к выполнению подпрограммы обработки прерываний. Первое, что необходимо сделать в этой подпрограмме, — сохранить в стеке IRQ все регистры из диапазона R0...R12, которые будут в ней использоваться. Затем можно приступать собственно к обработке исключительной ситуации.

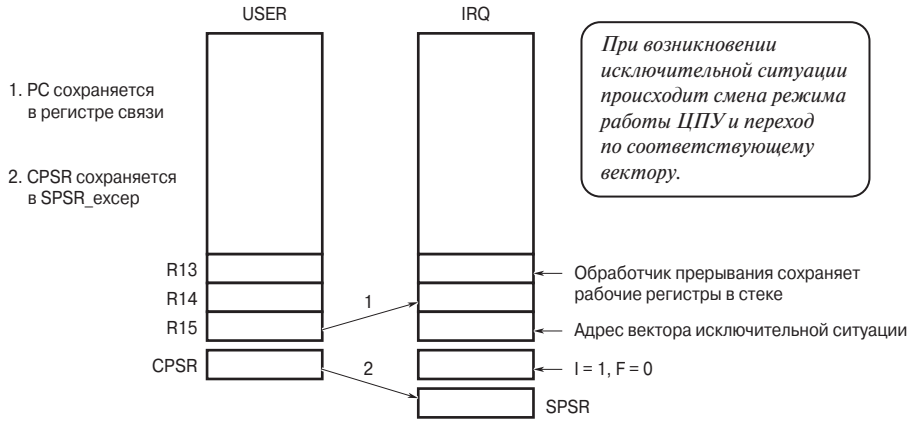


Рис. 1.6. Обработка исключительной ситуации.

После завершения обработки исключительной ситуации необходимо вернуться в режим User и продолжить выполнение программы с прерванного места. Однако в наборе команд ARM отсутствуют команды типа «возврат» или «возврат из подпрограммы», поэтому манипуляции со счетчиком команд PC необходимо осуществлять, используя обычные команды. Ситуация осложняется тем, что существует несколько различных вариантов возврата.

Для начала взглянем на команду `SWI`. При выполнении этой команды адрес следующей выполняемой команды сохраняется в регистре связи, после чего производится обработка исключительной ситуации. Все, что нам нужно сделать для возврата из исключительной ситуации, — это загрузить содержимое регистра связи обратно в PC, и программа продолжит свое выполнение. Однако чтобы ЦПУ при этом переключился обратно в режим User, необходимо использовать специальную команду пересылки `MOVS` (более подробно мы рассмотрим ее чуть позже). Таким образом, команда возврата из программного прерывания будет следующей:

```
MOVS    R15,R14         ; Скопировать регистр связи в PC и переключить режимы
```

А при возникновении исключительной ситуации по прерываниям FIQ и IRQ текущая выполняемая команда сбрасывается и выполняется переход к обработчику исключительной ситуации. При возврате из исключительной ситуации в регистре связи находится адрес отброшенной команды плюс 4. Чтобы возобновить выполнение программы с нужного места, мы должны уменьшить значение, хранящееся в регистре связи, на 4. В данном случае для уменьшения содержимого

регистра связи и сохранения результата в PC мы используем специальную команду вычитания, восстанавливающую также и режим работы ЦПУ. Таким образом, команда возврата из режимов FIQ, IRQ и Abort выглядит следующим образом:

```
SUBS    R15, R14, #4
```

В случае, если произошла ошибка обращения к памяти, исключительная ситуация возникнет через одну команду после той, выполнение которой явилось ее причиной. В идеале, в этом случае мы должны перейти к подпрограмме обработки прерывания Data Abort, выяснить и устранить причину затруднений и снова попытаться выполнить команду, вызвавшую исключительную ситуацию. Соответственно, мы должны «отмотать» PC назад на две команды — отброшенную и вызвавшую возникновение исключительной ситуации. Другими словами, нам нужно вычесть из регистра связи число восемь и сохранить результат в PC. Таким образом, команда возврата из прерывания Data Abort имеет вид:

```
SUBS    R15, R14, #8
```

При выполнении команды возврата модифицированное содержимое регистра связи загружается в счетчик команд, ЦПУ переключается обратно в режим User, а содержимое регистра SPSR переписывается обратно в CPSR. В случае возникновения исключительных ситуаций FIQ или IRQ дополнительно разрешаются соответствующие прерывания. В результате всех этих действий процессор выходит из привилегированного режима и возвращается к выполнению пользовательской программы (**Рис. 1.7**).

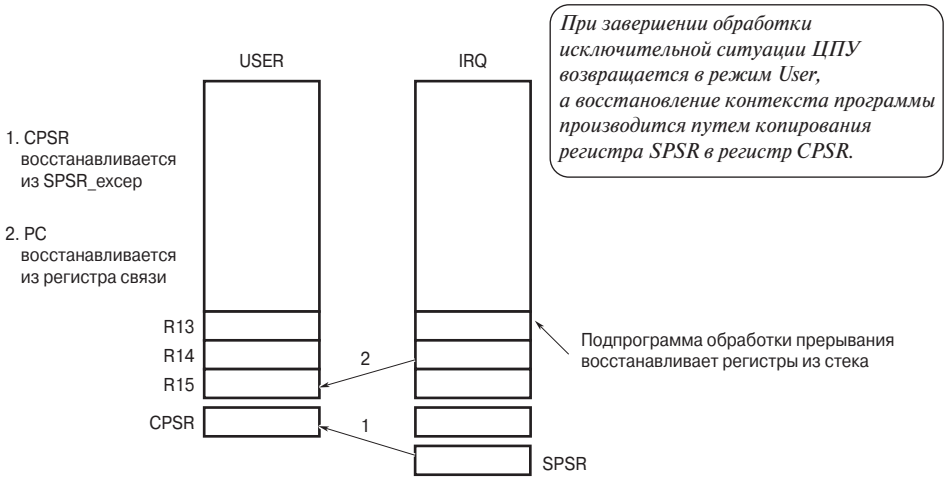


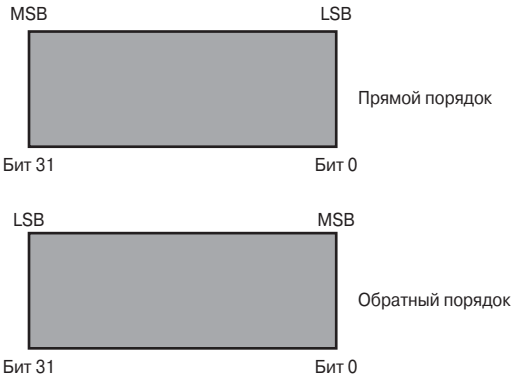
Рис. 1.7. Завершение обработки исключительной ситуации.

1.6. Набор команд ARM7

Теперь, когда мы получили общее представление о ядре ARM7, его модели программирования и режимах работы, пришла пора познакомиться с его набором или, точнее, наборами команд. Поскольку все примеры в книге написаны на

Си, вам нет необходимости быть экспертом в области программирования на ассемблере ARM7. Однако чтобы разрабатывать действительно эффективные программы, очень важно понимать машинный код, скрывающийся за строками программы на языке высокого уровня. Прежде чем мы приступим к изучению команд ARM7, необходимо отметить, что на самом деле ЦПУ ARM7 поддерживает два набора команд: набор команд ARM с 32-битными командами и набор команд THUMB с 16-битными командами. Далее в книге слово ARM будет означать 32-битный набор команд, а слово ARM7 — собственно ЦПУ.

Ядро ARM7 было разработано таким образом, чтобы его можно было использовать как в качестве процессора с обратным порядком байтов (big-endian processor), так и в качестве процессора с прямым порядком байтов (little-endian processor). В первом случае старший бит (Most Significant Bit — MSB) 32-битного слова располагается в начале слова, а во втором случае — в конце (**Рис. 1.8**). Думаю, вы обрадуетесь, узнав, что в семействе LPC2300 используется только прямой порядок байтов (т.е. MSB является битом с самым старшим адресом), что значительно облегчает работу с процессором. Однако используемый вами компилятор для ARM7 должен уметь компилировать код в обоих форматах. В связи с этим необходимо удостовериться, что формат слов задан правильно, в противном случае полученный код будет «вывернут наизнанку».

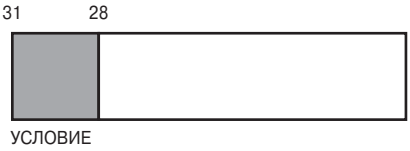


ЦПУ ARM7 поддерживает как прямой, так и обратный порядок байтов. Тем не менее, в микроконтроллерах NXP используется только прямой порядок байтов (little endian).

Рис. 1.8. Прямой и обратный порядок байтов.

Одна из наиболее интересных особенностей набора команд ARM заключается в том, что каждая команда поддерживает условное выполнение. В традиционных микроконтроллерах единственными условными командами являются команды условных переходов, и, быть может, ряд других, таких как команды проверки либо изменения состояния отдельных битов. А в наборе команд ARM старшие 4 бита кода команды всегда сравниваются с флагами условий в регистре CPSR (**Рис. 1.9**). Если их значения не совпадают, команда не выполняется и проходит через конвейер как команда `NOP` (нет операции).

Таким образом, можно выполнить какую-либо команду обработки данных, изменяющую флаги условий в регистре CPSR. Затем, в зависимости от результата, следующая команда может быть выполнена, а может и нет. К базовым мнемо-



Каждая команда ARM (32-битная) является условно выполняемой. Между 4 старшими битами кода команды и флагами условий регистра CPSR производится операция «Логическое И». Если значения не совпадают, выполняется команда NOP.

Рис. 1.9. Расположение битов сравнения в команде ARM.

ническим обозначениям команд ассемблера, таким как `MOV` или `ADD`, можно добавить любой из шестнадцати префиксов, определяющих тестируемые состояния флагов условий (**Табл. 1.3**).

Таблица 1.3. **Префиксы команд**

Префикс	Флаги	Значение
EQ	Z установлен	Равно
NE	Z сброшен	Не равно
CS	C установлен	Выше или равно (беззнаковое)
CC	C сброшен	Ниже (беззнаковое)
MI	N установлен	Отрицательный результат
PL	N сброшен	Положительный результат
VS	V установлен	Переполнение
VC	V сброшен	Нет переполнения
HI	C установлен, Z сброшен	Выше (беззнаковое)
LS	C сброшен, Z установлен	Ниже или равно (беззнаковое)
GE	N равен V	Больше или равно (знаковое)
LT	N не равен V	Меньше (знаковое)
GT	Z сброшен И (N равен V)	Больше (знаковое)
LE	Z установлен ИЛИ (N не равен V)	Меньше или равно (знаковое)
AL	(игнорируются)	Безусловное выполнение

К обозначению любой команды ARM (32-битной) можно добавить один из 16 префиксов, определяющих тестируемые флаги условий. Соответственно, существует 16 вариантов каждой команды

К примеру, команда:

```
EQMOV   R1, #0x00800000
```

выполнит загрузку числа 0x00800000 в регистр R1 только в том случае, если результат выполнения последней команды обработки данных был «равно» и соответственно установлен флаг Z регистра CPSR. Целью такого условного выполнения команд является обеспечение непрерывности потока команд через конвейер, так как при каждом выполнении команд перехода конвейер сбрасывается и на его повторное заполнение требуется время, что резко снижает общую производительность. На практике существует некоторый порог, при котором принудительное «проталкивание» команд `NOP` через конвейер оказывается эффективнее выполнения традиционных команд условного перехода и связанного с этим повторным заполнением буфера. Указанный порог равен трем командам, поэтому короткий переход, такой как:

```
if(x < 100)
{
   x++;
}
```

при использовании условно-выполняемых команд ARM будет реализован более эффективно.

Все множество команд ARM можно разбить на 6 основных групп: команды ветвления, команды обработки данных, команды передачи данных, команды передачи блоков данных, команды умножения и команда программного прерывания.

1.6.1. Команды ветвления

Базовая команда перехода (B), как следует из ее названия, позволяет выполнять переход в диапазоне до 32 Мбайт как вперед, так и назад. Модифицированная версия команды, команда перехода с сохранением адреса (BL), выполняет ту же операцию, однако при этом сохраняет в регистре связи текущее значение PC, увеличенное на четыре (**Рис. 1.10**).

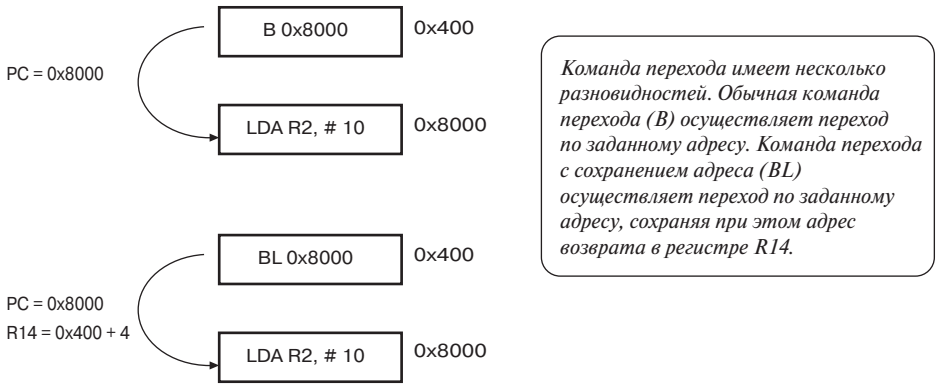


Рис. 1.10. Команды перехода B и BL.

Таким образом, команда перехода с сохранением адреса используется в качестве команды вызова подпрограмм, сохраняющей адрес возврата в регистре связи. Для возврата из подпрограмм можно использовать команду обычного перехода, выполняющую переход по адресу, находящемуся в регистре связи. Используя флаги условий, мы можем выполнять условные переходы и условные вызовы подпрограмм. Существует еще две разновидности команды перехода: «переход со сменой состояния» (BX) и «переход со сменой состояния и сохранением адреса» (BLX). Эти команды выполняют те же операции, что и предыдущие команды, но при этом еще и выполняют переключение с набора команд ARM на THUMB и обратно (**Рис. 1.11**).

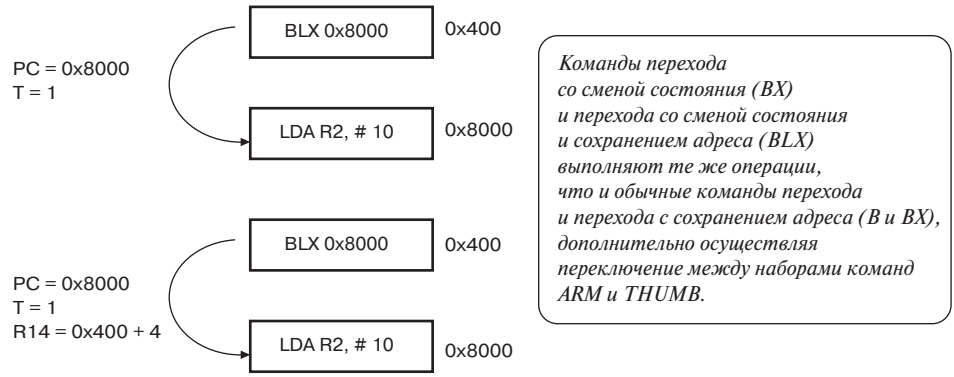


Рис. 1.11. Команды перехода BX и BLX.

Это единственный способ, который вы должны применять для изменения используемого набора команд, так как непосредственные манипуляции с флагом T регистра CPSR могут привести к непредсказуемым результатам.

1.6.2. Команды обработки данных

Обобщенный формат всех команд обработки данных приведен на **Рис. 1.12**. В каждой команде имеется регистр результата и два операнда. Первый операнд обязательно должен быть регистром, тогда как второй может быть и регистром, и константой.

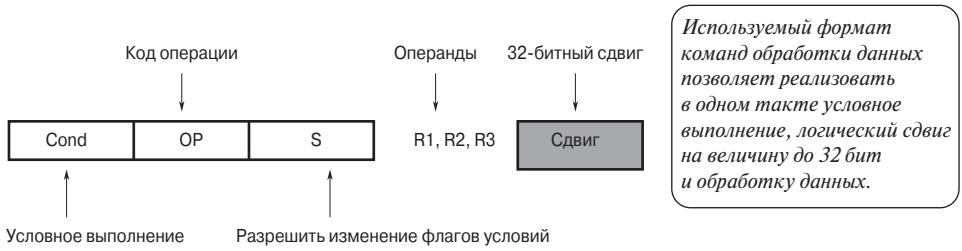


Рис. 1.12. Формат команд обработки данных.

Помимо всего прочего, в ЦПУ ARM7 имеется многорегистровое устройство циклического сдвига (barrel shifter), позволяющее при выполнении команды сдвигать значение 2-го операнда на величину до 32 бит. Бит S используется для управления флагами условий. Если этот бит установлен, флаги условий изменяются в соответствии с результатом выполнения команды. Если этот бит сброшен, состояние флагов условий не изменяется. Однако если при установленном бите S в качестве регистра результата указан счетчик команд (R15), производится копирование содержимого регистра SPSR текущего режима в регистр CPSR. Эта возможность используется для восстановления PC и переключения в исходный ре-

жим в конце обработки исключительных ситуаций. Не пытайтесь выполнить такую команду в режиме User, поскольку в этом режиме отсутствует регистр SPSR и соответственно результат выполнения этой команды невозможно предсказать.

Таблица 1.4. **Команды обработки данных**

Мнемокод	Описание команды	Мнемокод	Описание команды
AND	Логическое побитовое «И»	TST	Проверка битов
EOR	Логическое побитовое «Искл. ИЛИ»	TEQ	Побитовое сравнение
SUB	Вычитание	CMP	Сравнение
RSB	Обратное вычитание	CMN	Сравнение с отрицанием
ADD	Сложение	ORR	Логическое побитовое «ИЛИ»
ADC	Сложение с учетом переноса	MOV	Пересылка
SBC	Вычитание с заемом	BIC	Сброс битов (маскирование)
RSC	Обратное вычитание с заемом	MVN	Пересылка с инверсией

Эти особенности предоставляют нам богатый набор команд обработки данных (**Табл. 1.4**), который, с одной стороны, позволяет создавать очень эффективные программы, а с другой — является источником ночных кошмаров для разработчиков компиляторов. Например, в результате компиляции выражения

```
if(Z == 1)
   R1 = R2 + (R3 × 4);
```

может быть сгенерирована следующая команда:

```
EQADDS  R1,R2,R3,LSL #2
```

Копирование регистров

Следующую группу составляют команды передачи данных (**Табл. 1.5**). ЦПУ ARM7 поддерживает команды загрузки/сохранения, позволяющие пересылать знаковые и беззнаковые числа разного размера (слово, полуслово, байт) в/из заданного регистра.

Таблица 1.5. **Команды передачи данных**

Мнемокод	Описание команды	Мнемокод	Описание команды
LDR	Загрузить слово	STR	Сохранить слово
LDRH	Загрузить полуслово	STRH	Сохранить полуслово
LDRSH	Загрузить полуслово со знаком	STRSH	Сохранить полуслово со знаком
LDRB	Загрузить байт	STRB	Сохранить байт
LDRSB	Загрузить байт со знаком	STRSB	Сохранить байт со знаком

Поскольку набор регистров полностью ортогонален, можно загружать 32-битное значение непосредственно в PC, осуществляя, таким образом, переход в пределах всего адресного пространства процессора. Если конечный адрес лежит

вне диапазона команды перехода, можно просто загрузить сохраненную константу в счетчик команд.

Групповое копирование регистров

Помимо команд загрузки/сохранения содержимого отдельных регистров, в наборе команд ARM имеются команды для загрузки (`LDM`) и сохранения (`STM`) групп регистров (**Рис. 1.13**). Таким образом, с помощью одной команды можно скопировать в память весь блок регистров или его часть, а с помощью другой — восстановить его содержимое.

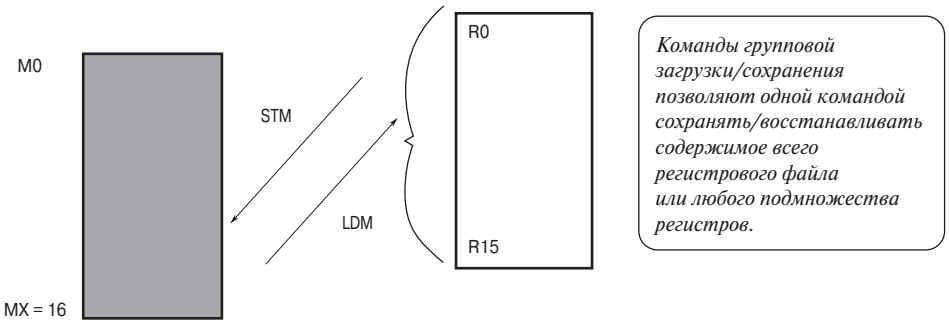


Рис. 1.13. Команды `LDM` и `STM`.

1.7. Команда обмена

В наборе команд ARM имеется также команда обмена (`SWP`), благодаря которой обеспечивается поддержка семафоров реального времени. Эта атомарная команда осуществляет одновременный обмен содержимого двух регистров (**Рис. 1.14**). Благодаря такому решению предотвращается прерывание процесса обмена критическими данными при возникновении исключительной ситуации. В языке Си эта команда напрямую недоступна и поддерживается встроенными функциями библиотек компиляторов

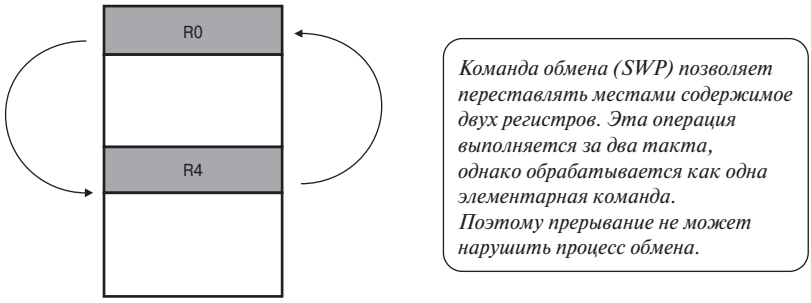
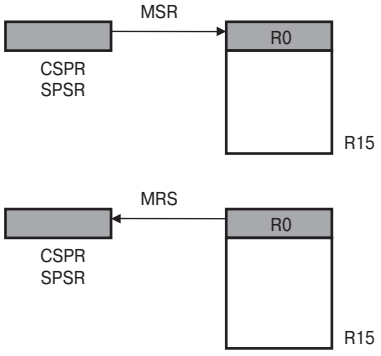


Рис. 1.14. Команда обмена.

1.8. Изменение регистров состояния

Как уже было отмечено в разделе, посвященном архитектуре ARM7, регистры CPSR и SPSR являются регистрами ЦПУ, однако не входят в состав основного банка регистров. Напрямую обращаться к этим регистрам могут только две команды ARM — `MSR` и `MRS`. Указанные команды обеспечивают пересылку содержимого регистра CPSR или SPSR в/из заданного регистра (**Рис. 1.15**). Например, чтобы запретить прерывания IRQ, необходимо скопировать содержимое регистра CPSR в рабочий регистр, установить флаг I (путем выполнения операции «И» между этим регистром и числом 0x00000080) и загрузить полученное значение обратно в регистр CPSR.



Регистры CPSR и SPSR не отображены на память и не входят в состав основного банка регистров. Единственными командами, оперирующими с этими регистрами, являются команды MSR и MRS. Когда ЦПУ находится в режиме User, эти команды недоступны.

Рис. 1.15. Изменение регистров состояния.

Команды `MSR` и `MRS` доступны во всех режимах процессора, за исключением режима User. Таким образом, менять рабочий режим процессора и разрешать/запрещать прерывания можно, только находясь в привилегированном режиме. После входа в режим User вы можете из него выйти только при возникновении исключительной ситуации, сбросе, генерации прерываний FIQ и IRQ или же в результате выполнения команды `SWI`.

1.9. Программное прерывание

Команда программного прерывания `SWI` генерирует исключительную ситуацию, в результате чего процессор переключается в режим Supervisor, а в счетчик команд заносится значение 0x00000008. Как и все остальные команды ARM, команда `SWI` содержит в четырех старших битах флаги условного выполнения, за которыми располагается код операции (**Рис. 1.16**). Остальная часть слова команды остается свободной. Однако в этих неиспользуемых битах может храниться число. Данные биты можно проверять в начале подпрограммы обработки прерывания, чтобы определить, какую именно часть подпрограммы следует выполнять. Таким образом, с помощью команды `SWI` можно переключаться в защищенный режим для выполнения привилегированных участков программы или обработки системных вызовов.

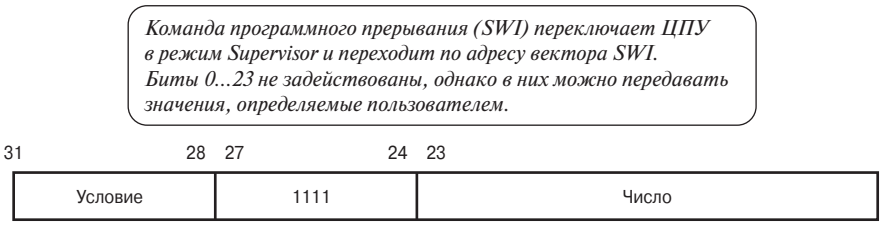


Рис. 1.16. Команда `SWI`.

После компиляции команды

```
SWI     #3
```

в неиспользуемых битах слова команды будет записано число 3. В подпрограмме обработки прерывания `SWI` мы можем проверить значение слова команды следующим образом (текст написан на псевдокоде):

```
switch( *(R14 - 4) & 0x00FFFFFF)  // Вернуться на 4 байта назад
{                                 // Замаскировать старшие 8 битов и
                                  // выполнить переход
                                  // в соответствии с результатом
    case (SWI-1)
                  ......
```

В зависимости от используемого компилятора вам придется реализовывать такую проверку самостоятельно или же компилятор автоматически вставит необходимые команды в код программы.

1.10. Модуль MAC

Наряду с многорегистровым устройством циклического сдвига в ядре ARM7 имеется встроенный модуль умножителя/сумматора (MAC). Модуль MAC поддерживает умножение чисел типа integer и long integer. Команды умножения чисел типа integer выполняют умножение двух 32-битных регистров и помещают результат в третий 32-битный регистр. Команда умножения с накоплением выполняет умножение и прибавляет произведение к промежуточной сумме. Команды умножения чисел типа long integer перемножают содержимое двух 32-битных регистров и помещают 64-битный результат в два регистра. Аналогично, имеется команда длинного умножения с накоплением (**Табл. 1.6**).

Таблица 1.6. **Команды умножения**

Мнемокод	Описание команды	Разрешение
`MUL`	Умножение	32-битный результат
`MULA`	Умножение с накоплением	32-битный результат
`UMULL`	Беззнаковое умножение	64-битный результат
`UMLAL`	Беззнаковое умножение с накоплением	64-битный результат
`SMULL`	Знаковое умножение	64-битный результат
`SMLAL`	Знаковое умножение с накоплением	64-битный результат

1.11. Набор команд THUMB

Несмотря на то, что ARM7 является 32-битным процессором, он поддерживает еще один набор команд (16-битный), называемый THUMB (**Рис. 1.17**). На самом деле, этот набор команд является сжатой формой набора команд ARM.

По сути дела, набор команд THUMB предназначен для увеличения плотности кода, чтобы можно было использовать небольшие однокристальные микроконтроллеры с ядром ARM7.

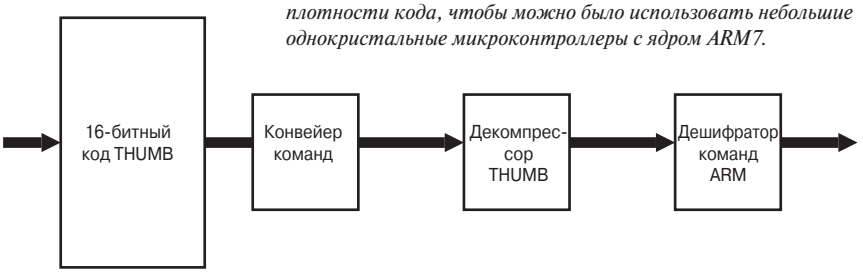


Рис. 1.17. Набор команд THUMB.

За счет этого команды, сохраненные в 16-битном формате, распаковываются в команды ARM, а затем выполняются. Хотя команды THUMB обеспечивают меньшую производительность по сравнению с командами ARM, благодаря им достигается более высокая плотность кода. Таким образом, для создания компактных программ, размещаемых в небольших однокристальных микроконтроллерах, код программ необходимо компилировать в виде совокупности функций THUMB и ARM. Этот процесс называется interworking и элементарно поддерживается всеми компиляторами. При компиляции с использованием набора команд THUMB вы получите 30-процентную экономию памяти программ, в то время как этот же код, скомпилированный с использованием команд ARM, будет выполняться на 40% быстрее.

Набор команд THUMB гораздо больше похож на наборы команд традиционных микроконтроллеров. В отличие от команд ARM, команды THUMB не поддерживают условного выполнения (за исключением команд условных переходов). Команды обработки данных имеют двухадресный формат, причем в качестве регистра результата используется один из регистров-источников:

Команда ARM	Команда THUMB	Действие
`ADD R0,R0,R1`	`ADD R0,R1`	`R0 = R0 + R1`

Команды THUMB не имеют полного доступа ко всем регистрам регистрового файла (**Рис. 1.18**). С регистрами R0...R7 (так называемыми младшими регистрами) могут работать все команды обработки данных. А к регистрам R8....R12 (старшие регистры) могут обращаться только 3 из них:

`MOV, ADD, CMP`

В наборе команд THUMB отсутствуют команды `MSR` и `MRS`, поэтому изменять регистры CPSR и SPSR можно только с помощью косвенной адресации. Если вам

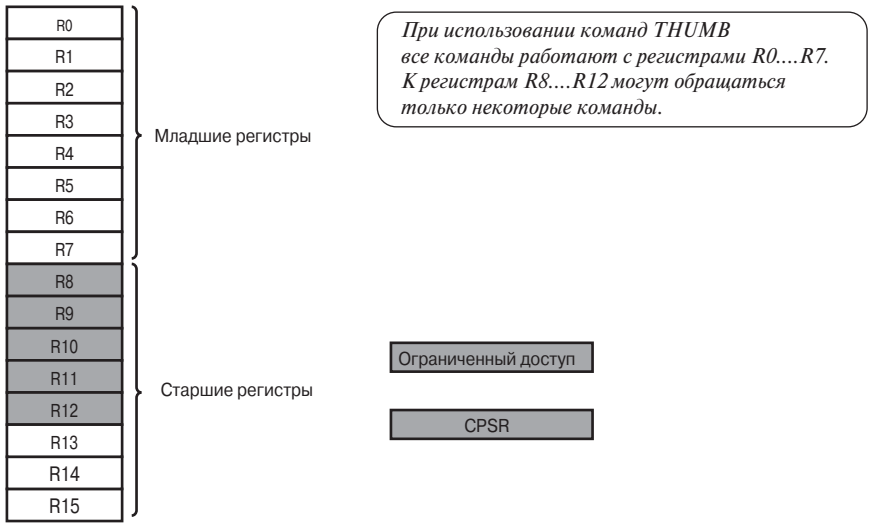


Рис. 1.18. Доступ к регистровому файлу.

требуется изменить какие-либо пользовательские биты в регистре CPSR, необходимо переключиться в режим ARM. Изменять режимы можно с помощью команд `BX` и `BLX` (**Рис. 1.19**). Кроме того, автоматическое переключение в режим ARM происходит при сбросе или при переходе к обработке исключительной ситуации.

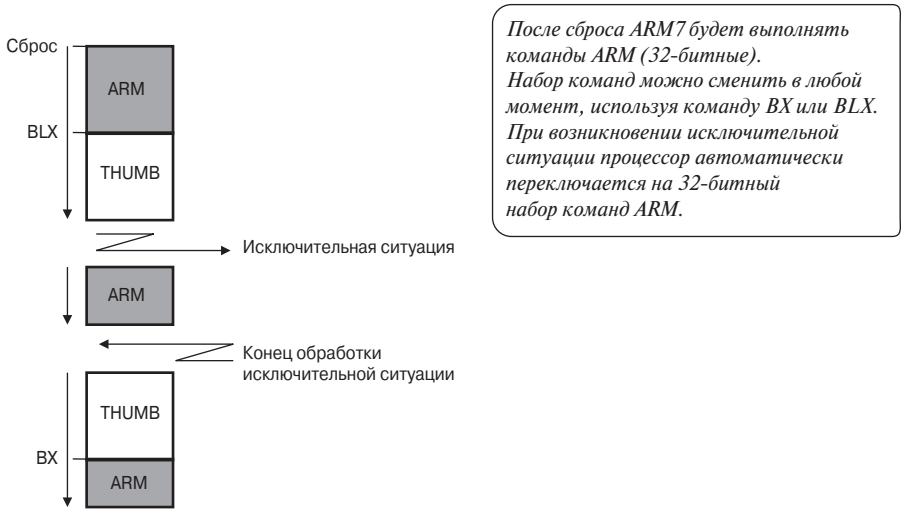


Рис. 1.19. Переключение в режим ARM.

В составе набора команд THUMB имеются более привычные команды работы со стеком PUSH и POP (**Рис. 1.20**). С помощью этих команд реализуется нисходящий стек, жестко привязанный к регистру R13.

В наборе команд THUMB имеются специальные команды PUSH и POP, которые реализуют нисходящий стек, используя в качестве указателя стека регистр R13.

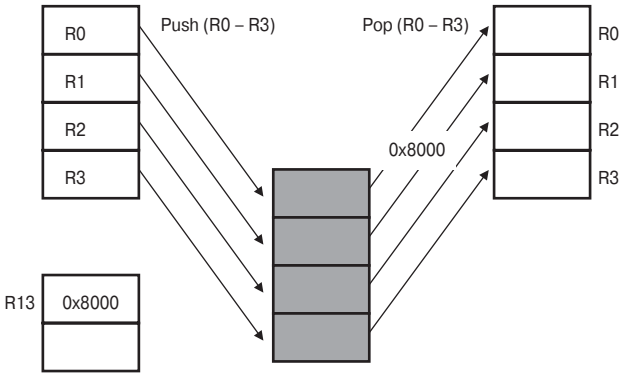


Рис. 1.20. Команды PUSH и POP.

И, наконец, в наборе команд THUMB имеется команда SWI, работающая так же, как и аналогичная команда набора ARM, но содержащая только 8 неиспользуемых битов, что ограничивает максимальное число вызовов SWI до 255.

1.12. Резюме

После прочтения этой главы вы должны иметь общее представление о процессорном ядре ARM7. Книги, в которых ядро ARM7 рассматривается более подробно, приведены в списке литературы. Кроме того, на компакт-диске, прилагающемся к книге, имеется копия полного руководства пользователя по ядру ARM7 («ARM7 TDMI Technical Reference Manual»).

ГЛАВА 2

РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1. Основные положения

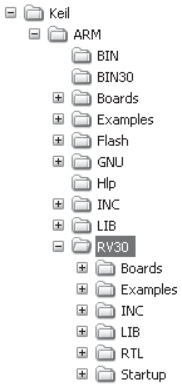
После прочтения первой главы у вас должно сложиться четкое представление о процессорном ядре ARM7. А в этой главе мы рассмотрим, как можно писать и отлаживать программы на языке Си для микроконтроллеров семейства LPC2300. Для микроконтроллеров с ядром ARM7 существует большое количество коммерческих компиляторов и даже бесплатный компилятор GCC с открытым кодом (open source). Однако все учебные программы, встречающиеся в книге, базируются на отладочном комплекте MDK-ARM компании Keil Elektronik. Этот комплект представляет собой законченное средство разработки для микроконтроллеров с ядром ARM7. В состав комплекта входит интегрированная среда разработки (ИСР) (Integrated Development Environment — IDE), которая называется «µVISION» (произносится, как «м*и*кровижн»), компилятор Real View, операционная система реального времени, симулятор с поддержкой симуляции периферийных устройств, а также аппаратный JTAG-отладчик. Вместе с комплектом также поставляется дополнительная библиотека времени выполнения (RTL-ARM), содержащая такие модули, как стек TCP/IP, встраиваемую файловую систему, драйверы шин CAN и USB. Кроме того, как мы увидим позже, комплект MDK-ARM все больше поддерживается программными продуктами сторонних производителей.

Среда HiTOP компании Hitex может работать с самыми разными компиляторами. Однако для микроконтроллеров с архитектурой ARM наибольшую популярность получили два компилятора — Real View компании Keil и свободно распространяемый GCC, которые мы и будем использовать в оставшейся части книги. Несмотря на то, что в данной книге мы рассматриваем только семейство LPC2300, программные средства для процессоров ARM от компаний Keil и Hitex могут использоваться и для других микроконтроллеров, построенных на базе ядра ARM7.

2.1.1. Загрузка и установка пакета программ компании Keil

Все учебные программы из этой книги рассчитаны на работу с отладочным комплектом MDK-ARM компании Keil. Ознакомительную версию программных

средств из этого комплекта можно установить с компакт-диска, прилагаемого к книге. Также эти программы можно загрузить с адреса www.keil.com/arm. Причем, на сайт имеет смысл заглянуть даже в том случае, если у вас есть компакт-диск, — хотя бы для того, чтобы убедиться, что в вашем распоряжении имеется последняя версия. Структура каталогов, формируемая после установки программ, приведена на **Рис. 2.1**.



При инсталляции ПО из комплекта поставки MDK-ARM на жесткий диск компьютера устанавливаются оба компилятора – Real View и CARM. Все примеры из книги и сопутствующие файлы написаны для компилятора Real View и расположены в папке RV30.

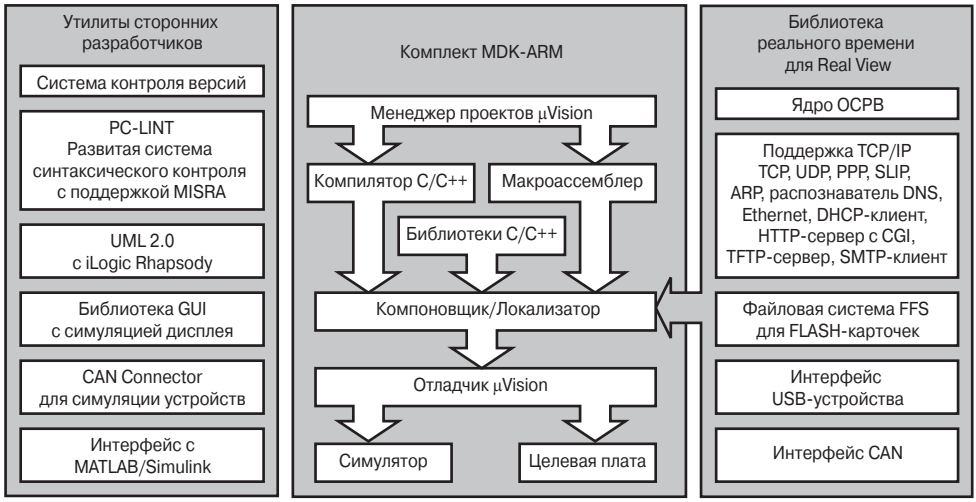
Рис. 2.1. Структура каталогов ПО из комплекта MDK-ARM.

Поскольку с отладочным комплектом поставляется также компилятор CARM, нелишним будет разобраться со структурой папок, создаваемой при инсталляции, чтобы вы легко могли найти исходный текст интересующего вас примера, а также файлы сопутствующих библиотек. Учебные примеры для компилятора CARM находятся в папках «C:\Keil\ARM\Examples» и «C:\Keil\ARM\Boards». А нужные нам файлы для компилятора Real View находятся в папке «C:Keil\ARM\RV30» (подпапки «Examples» и «Boards»). Несмотря на то, что эти примеры в книге не используются, рекомендуется ознакомиться с ними, поскольку в них содержится много дополнительной информации, не раскрытой в книге. Кроме того, компания Keil постоянно увеличивает количество этих примеров с каждой новой версией ПО. После инсталляции программного обеспечения можно приступать к использованию комплекта MDK-ARM, запустив ИСР µVision.

2.1.2. ИСР µVISION

Вместе с комплектом MDK-ARM поставляется ИСР µVision (**Рис. 2.2**). Как уже упоминалось, эта среда может работать с компилятором Real View, «open source» компилятором GCC или «родным» компилятором CARM, который поставлялся со старыми версиями ИСР. Пользователю также предоставляются библиотечные модули операционной системы реального времени. Это вытесняющая многозадачная операционная система была написана специально для использования с небольшими микроконтроллерами на базе ядра ARM. Помимо этого, в среде µVision имеется два средства отладки. Начнем с того, что после компиляции

и компоновки программы ее код можно загрузить в симулятор μVISION. Этот отладчик симулирует работу ядра ARM7 и периферийных устройств поддерживаемого микроконтроллера. Работа с симулятором — хороший способ ознакомления с моделями семейства LPC2300. Поскольку симулятор обеспечивает симуляцию работы ядра и периферии с точностью до такта, он может оказаться очень полезным инструментом для проверки, корректно ли был инициализирован микроконтроллер и правильно ли были вычислены различные стартовые константы, такие как значения коэффициента деления предделителей таймеров.



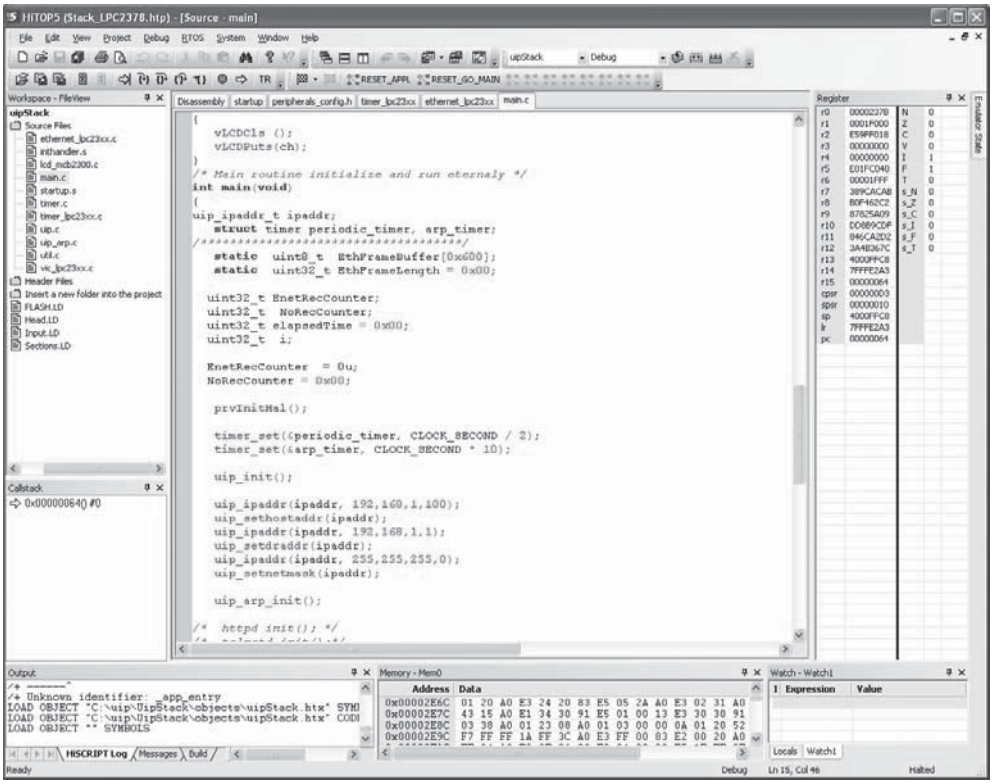
Отладочный комплект MDK-ARM компании Keil представляет собой законченное средство разработки для небольших микроконтроллеров с ядром ARM. В состав комплекта входит интегрированная среда разработки, компилятор с сопутствующими программами, средства отладки, а также операционная система реального времени с расширенной поддержкой связующего ПО.

Рис. 2.2. Состав комплекта MDK-ARM.

Однако, рано или поздно, вам потребуются реальные входные воздействия. Иногда их можно сымитировать с помощью языка сценариев симулятора, но в подавляющем большинстве случаев придется запускать программу на реальном устройстве. Клиентскую часть симулятора можно подключить к вашему устройству через фирменный отладчик компании Keil — ULINK. К ПК кабель ULINK подключается через USB, а к устройству — через интерфейс JTAG микроконтроллера LPC2300. Интерфейс JTAG является отдельным модулем ядра ARM7, который поддерживает команды отладки, посылаемые хостом (ПК). Используя интерфейс JTAG, можно из симулятора μVision управлять работой микроконтроллера. С его помощью можно загружать код в микроконтроллер, запускать программу в пошаговом режиме или в режиме реального времени, устанавливать точки останова и просматривать содержимое памяти.

2.1.3. ИСР HiTOP

ИСР HiTOP (**Рис. 2.3**) поддерживает самые разнообразные отладочные средства. Вы можете как протестировать код, написанный для ядра ARM7, в программном симуляторе, так и провести отладку на реальной плате, используя отладчик Tantino. В отличие от отладчика ULINK компании Keil, Tantino помимо ядра ARM7 поддерживает также ядра ARM9 и ARM11. При работе с объемными программами этот отладчик обеспечивает более высокую скорость записи FLASH-памяти микроконтроллеров. Кроме того, он предоставляет более развитые отладочные возможности, такие как установка и сброс точек останова «на лету».



В составе ИСР HiTOP имеется редактор исходного кода, менеджер проектов, утилита компоновки, а также развитой отладчик. ИСР HiTOP поддерживает большинство распространенных компиляторов для процессоров с ядром ARM, включая компиляторы GCC и Real View.

Рис. 2.3. Главное окно ИСР HiTOP компании Hitex.

Как и ULINK, отладчик Tantino (**Рис. 2.4**) подключается к ПК через USB, а к микроконтроллеру LPC2300 — по интерфейсу JTAG. С помощью этого интерфейса можно выполнять загрузку кода в микроконтроллер, программирование его FLASH-памяти, а также управлять на базовом уровне его функционировани-

Устройство «Tantino» представляет собой JTAG-отладчик, обладающий самыми широкими возможностями. В частности, он позволяет задавать условные точки останова для отладки кода ARM, задавать несколько точек останова в FLASH-памяти, «на лету» обновлять содержимое окон просмотра переменных, а также поддерживает функцию «ассистента исключений», которая позволяет отслеживать ошибки времни выполнения.

Рис. 2.4. JTAG-отладчик Tantino.

ем. Помимо порта JTAG в микроконтроллерах семейства LPC2300 имеется еще один отладочный порт, подключенный к встроенному модулю трассировки (Embedded Trace Module — ETM). Подключившись к модулю ETM, можно с помощью инструмента Trace (трассировщик) записать операции, выполняемые микроконтроллером, а затем воспроизвести их в окне ИСР HiTOP в виде последовательности строк кода на языке высокого уровня, машинных команд или циклов. Также модуль ETM позволяет отслеживать потоки данных в программе — все операции записи и чтения в ОЗУ и регистрах специальных функций (РСФ) можно сохранить в буфере трассировки для последующего анализа. Обычный JTAG-отладчик не имеет доступа к данным модуля ETM, соответственно для этой цели используется более сложная система, называемая Tanto. Особенности данной системы обсуждаются в 8-й главе, посвященной практическим упражнениям, однако сразу можно отметить, что для работы как с Tantino, так и с Tanto используется одна и та же ИСР HiTOP. Вместе с продуктами компании Hitex поставляется также программа StartEasy, представляющая собой инструментальное средство для автоматизированной разработки программ (CASE-средство). С помощью этого средства вы можете определить проект для микроконтроллера семейства LPC2300 и сгенерировать «скелет» будущей программы, содержащий стартовый код, а также строки инициализации периферийных устройств, которые вы собираетесь использовать. Замечу, что вы можете скачать полную версию программы StartEasy с веб-сайта компании Hitex, даже если и не собираетесь использовать другие инструментальные средства этой компании.

2.1.4. Учебные примеры

В соответствующих главах данной книги рассматривается множество учебных программ, которые демонстрируют основные возможности микроконтроллеров семейств LPC2300/2400. В 8-й главе подробно разбираются учебные программы,

иллюстрирующие основные особенности микроконтроллеров LPC2300/2400. В этой же главе содержится вводный урок по среде μVision компании Keil. Исходные тексты всех остальных упражнений наряду с файлами проекта разделены по различным папкам прилагаемого компакт-диска.

С книгой лучше всего работать в следующей последовательности: сначала прочитать раздел, а затем перейти к учебному пособию и выполнить упражнение. Если придерживаться этой схемы, то к тому моменту, когда вы перевернете последнюю страницу книги, вы будете в совершенстве знать ядро ARM7, средства разработки для него и микроконтроллеры семейств LPC2300/2400.

Упражнение 1. Знакомство с ИСР Keil
В нашем первом упражнении рассматривается процесс установки ПО μVision компании Keil и настройки нового проекта.

2.2. Стартовый код

Все учебные проекты включают в себя нескольких файлов с исходным кодом. Фактически весь исходный текст программы хранится в файлах с расширением .c, а файл startup.s является ассемблерным модулем, содержащим стартовый код, который предоставляется компанией — производителем компилятора. Как следует из его названия, стартовый код размещается в памяти микроконтроллера таким образом, чтобы запускаться при переходе по вектору сброса. В этом коде располагается таблица векторов исключительных ситуаций, а также осуществляется инициализация указателей стека различных режимов работы. Кроме того, прежде чем передать управление написанной вами функции `main()`, в нем выполняется инициализация некоторых периферийных модулей и встроенного ОЗУ. Стартовый код будет различным в зависимости от конкретного устройства и используемого компилятора, поэтому при настройке проекта необходимо убедиться, что вы используете корректный файл. Модули со стартовым кодом для компилятора Real View находятся в папке «C:\Keil\ARM\RV30\startup\NXP», а для компилятора GCC — в папке «C:\Keil\GNU\startup».

Прежде всего, в стартовом коде размещается таблица векторов исключительных ситуаций, как показано на **Рис. 2.5**.

Директива `AREA` используется компоновщиком для размещения таблицы векторов по требуемому адресу. В однокристальных системах этот адрес всегда равен 0x00000000, однако, если вы используете внешнюю шину и хотите загружать программу из внешней памяти, то таблица векторов должна размещаться, начиная с адреса 0x80000000. Причины этого будут объяснены в 3-й главе. В таблице векторов используется команда `LDR` (загрузка регистра). Эта команда загружает в счетчик команд 32-битную константу из таблицы, расположенной сразу после таблицы векторов. Таким образом, таблица векторов занимает в общей сложности первые 64 байт памяти. В принципе, вместо команд `LDR` можно использовать команды безусловного перехода. Однако эти команды могут осуществлять переход только в пределах ±32 Мбайт, тогда как команда `LDR` позволяет обращаться ко всему адресному пространству процессора ARM7, составляющему 4 Гбайт. Также вы могли заметить, что по адресу 0x00000014 неиспользуемого вектора размеща-

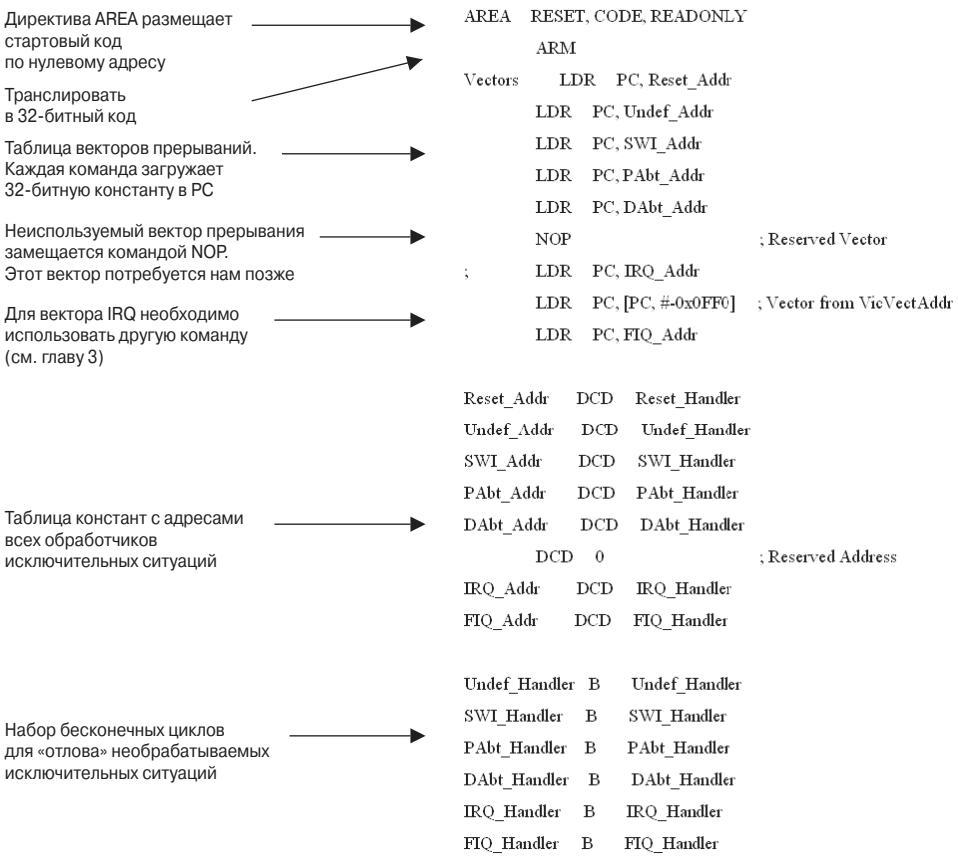
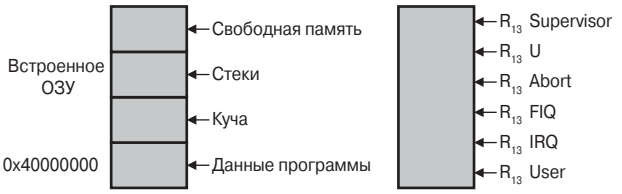


Рис. 2.5. Начало стартового кода.

ется команда `NOP`. В микроконтроллерах LPC2300 фирмы NXP эти 4 байта имеют особое назначение, которое мы обсудим в 3-й главе. И наконец, формат команды, используемой для перехода к обработчику прерывания IRQ, отличается от формата команд для остальных векторов. Этот вопрос тоже будет рассматриваться в 3-й главе.

Поскольку в каждом режиме работы имеется уникальный регистр R13, в процессоре ARM7 присутствует шесть стеков. Эти стеки необходимо инициализировать в стартовом коде до начала выполнения пользовательской программы. Компилятор размещает пользовательские переменные в самых младших адресах встроенного ОЗУ, начиная с его стартового адреса. Затем идет область для динамического выделения памяти, так называемая «куча». Стеки же размещаются в старших адресах ОЗУ и заполняются в направлении младших адресов (**Рис. 2.6**).

В стартовом коде происходит последовательное переключение в различные режимы ARM7 и в каждый из регистров R13 загружается начальный адрес соответствующего стека (**Рис. 2.7**).



По умолчанию компилятор Real View везде, где только можно, размещает локальные переменные в регистрах процессора. Остальные данные программы располагаются, начиная со стартового адреса встроенного ОЗУ. После этих данных выделяется область для «кучи» и только после нее последовательно размещаются все шесть стеков. Размер каждого стека задается пользователем.

Рис. 2.6. Распределение ОЗУ.

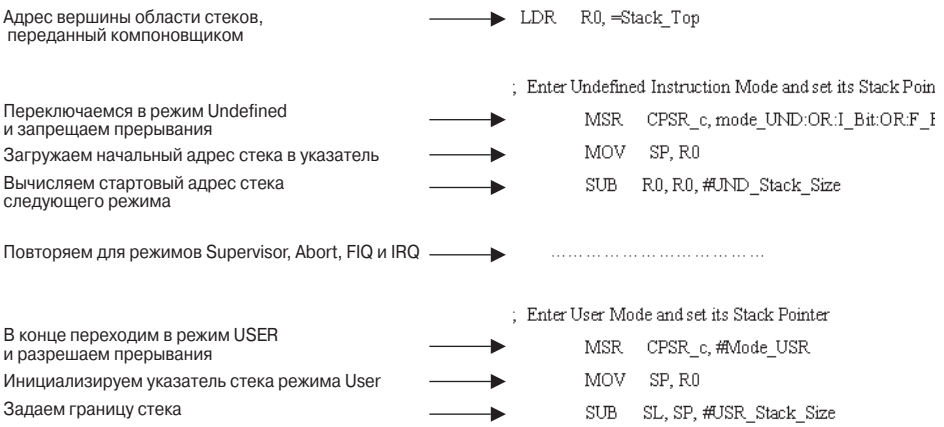


Рис. 2.7. Инициализация стеков в стартовом коде.

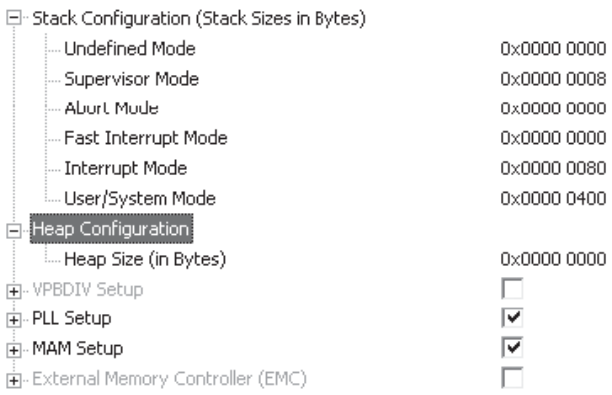


Рис. 2.8. Конфигурирование стеков.

Как и в случае таблицы векторов прерываний, задание размера стеков ложится на ваши плечи. В принципе, это можно сделать, непосредственно изменяя код стартового модуля, однако в составе ИСР предусмотрен специализированный редактор, позволяющий управлять параметрами стеков в графическом виде (**Рис. 2.8**). Более того, этот графический редактор позволяет конфигурировать ряд системных периферийных модулей LPC2300. Позже мы рассмотрим данный вопрос более подробно, но не забывайте, что модули можно конфигурировать путем непосредственного изменения стартового кода.

Упражнение 2. Стартовый код
Во втором упражнении рассматривается процесс выделения памяти под каждый из стеков процессора.

2.2.1. Определение карты памяти проекта

При создании проекта в ИСР µVision автоматически формируются файлы сборщика проекта и компоновщика, соответствующие используемому устройству. В случае однокристальных устройств FLASH-память по умолчанию определяется как область для хранения кода, а ОЗУ — как область данных (**Рис. 2.9**).

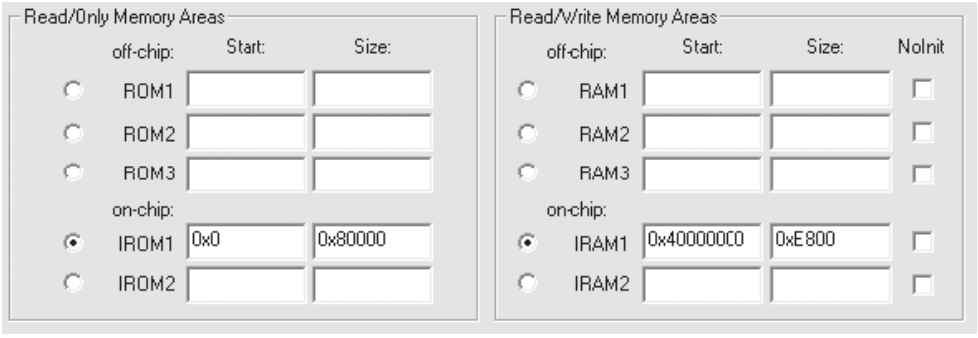


Рис. 2.9. Определение карты памяти в ИСР µVision.

При наличии в схеме внешних микросхем FLASH-памяти и ОЗУ, подключенных к внешней шине устройства, это же диалоговое окно можно использовать для определения дополнительных областей кода и данных. Чуть ниже в этой главе мы коснемся вопроса размещения заданных функций и переменных в таких секциях.

2.2.2. Определение карты памяти для компилятора GCC

Готовые файлы с исходным кодом программы компилируются в объектные модули. Эти модули необходимо скомпоновать вместе для получения конечного файла с абсолютным машинным кодом. В этом подразделе вкратце рассказывается о назначении и структуре файлов, используемых компоновщиком GCC для сборки исполнимого файла программы под микроконтроллеры семейства

LPC2300/2400. Запуск компоновщика осуществляется из командной строки следующим образом:

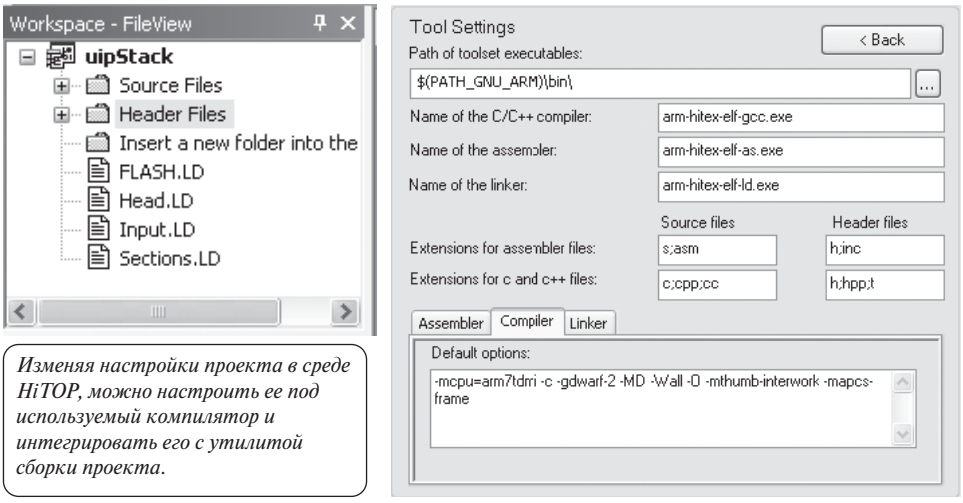
```
ld ld_opt -o <имя_проекта>.elf
```

где под символами `ld_opt` скрываются следующие ключи:

```
-T.\objects\<файл_компоновщика>.ld --cref -t -static -lgcc -lc -lm -nostartfiles -
Map=<project_name>.map
```

Файл сценариев компоновщика с расширением .ld сохраняется в рабочей папке проекта. Очень важно понимать структуру этого файла, поскольку вам может потребоваться модифицировать его по мере роста проекта. Файл сценариев представляет собой обычный текстовый файл, состоящий из нескольких секций, в которых описывается проект с точки зрения компоновщика. В самом начале указываются пути для поиска файлов библиотек компилятора. Эти пути определяются программой установки среды HiTOP и должны указывать на используемые вами библиотеки (**Рис. 2.10**):

```
SEARCH_DIR("C:\Program Files\Hitex\GnuToolPackageARM\ARM-hitex-elf\lib\interwork")
SEARCH_DIR("C:\Program Files\Hitex\GnuToolPackageARM\lib\gcc\ARM-hitexelf\4.0.0\interwork")
```



Изменяя настройки проекта в среде HiTOP, можно настроить ее под используемый компилятор и интегрировать его с утилитой сборки проекта.

Рис. 2.10. Настройка проекта в ИСР HiTOP.

Компилятор GCC поставляется с несколькими наборами библиотек. Указанные выше библиотеки поддерживают взаимодействие кодов ARM и THUMB. В следующей секции файла сценариев приводится список объектных файлов, составляющих проект:

```
GROUP ( objects\startup.o
        objects\main.o
        objects\interrupt.o
        objects\THUMB.o )
```

При добавлении в проект новых модулей вы должны будете вручную откорректировать данный список.

В следующей секции описывается подсистема памяти конечного микроконтроллера:

```
MEMORY
{
    IntCodeRAM (rx) : ORIGIN = 0x00000000, LENGTH = 512K /* Это в FLASH-памяти */
    IntDataRAM (rw) : ORIGIN = 0x40000000, LENGTH = 64k
}
```

В этой секции указываются размер и положение блоков памяти микроконтроллера. В приведенном примере описывается конфигурация подсистемы памяти микроконтроллера LPC2300/2400, имеющего 512 Кбайт FLASH-памяти, начинающейся с адреса 0x00000000, и 64 Кбайт ОЗУ, располагающегося начиная с адреса 0x40000000. Позже, когда мы познакомимся с интерфейсом внешней памяти микроконтроллеров семейства, вы увидите, как выглядит это описание для устройства, использующего внешнюю память. В последней, определяемой пользователем секции описывается размещение кода программы в памяти микроконтроллера.

```
SECTIONS
{
    .text
    {
       …
    }
    .data
    {
       …
    }
}
```

Описание кода программы разбито на две основные секции: `.text` и `.data`. В секции `.text` размещается исполняемый код и константы (да и вообще все, что должно находиться в FLASH-памяти). Секция `.data` используется для размещения переменных программы в пользовательской области ОЗУ.

```
.text :
{
    __code_start__ = .;
    objects\startup.o (.text) /* Стартовый код */
    objects\*.o (.text)
    . = ALIGN(4);
    __code_end__ = .;
    *(.glue_7t) *(.glue_7)
} >IntCodeRAM = 0
```

Первым в секции `.text` описывается стартовый код, который размещается по адресу вектора сброса. Далее указывается положение остальных секций кода программы в FLASH-памяти микроконтроллера. Наличие директивы `ALIGN` гарантирует, что начало любой перемещаемой секции кода будет выровнено по 4-байтной границе.

```
.data : AT (_etext)
{
  /* Используется для инициализированных данных */
  __data_start__ = . ;
  PROVIDE (__data_start__ = .) ;
  *(.data)
  SORT(CONSTRUCTORS)
  __data_end__ = . ;
  PROVIDE (__data_end__ = .) ;
} >IntDataRAM
. = ALIGN(4);
```

В секции `.data` описывается размещение пользовательских переменных в ОЗУ микроконтроллера.

2.3. Взаимодействие кодов ARM и THUMB

Один из наиболее важных вопросов, которые должны быть решены в нашей программе, — взаимодействие между наборами команд ARM и THUMB. Для обеспечения такого взаимодействия компанией ARM был разработан стандарт «ARM Procedure Call Standard» (APCS). Помимо всего прочего, этот стандарт определяет, каким образом производится вызов одних функций из других, порядок передачи параметров и использование стеков (**Рис. 2.11**). При использовании стандарта APCS к коду функций добавляется своеобразная «обертка» на языке ассемблера, благодаря которой обеспечивается поддержка различных возможностей компилятора. Чем больше этих возможностей вы используете, тем больше получается количество дополнительного кода. Теоретически, стандарт APCS обеспечивает совместную работу кода, созданного различными пакетами программ. То есть вы можете взять библиотеку, скомпилированную каким-либо компилятором, и использовать ее совместно с компилятором Keil или GCC.

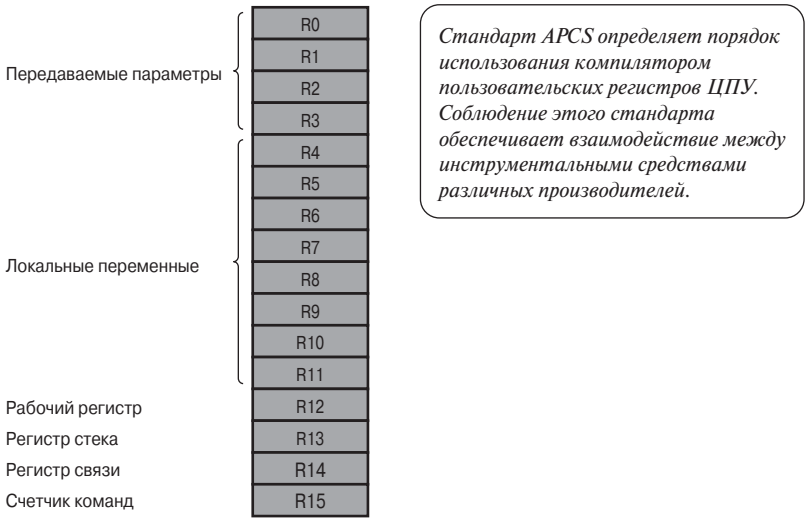


Рис. 2.11. Порядок использования пользовательских регистров.

Согласно стандарту APCS регистровый файл делится на несколько областей. Регистры R0...R3 используются для передачи параметров в подпрограммы. Если необходимо передать больше 16 байт, то остальные параметры передаются через стек. Локальные переменные размещаются в регистрах R4...R11, а регистр R12 резервируется для хранения адреса промежуточного интерфейсного кода. Среда µVision позволяет вам указать желаемый набор команд для любого участка кода. На уровне проекта задание набора команд, используемого по умолчанию, а также разрешение/запрещение взаимодействия кода ARM и THUMB (**Рис. 2.12**) осуществляется на вкладке **Compiler** диалогового окна, вызываемого из меню **Options for Target**.

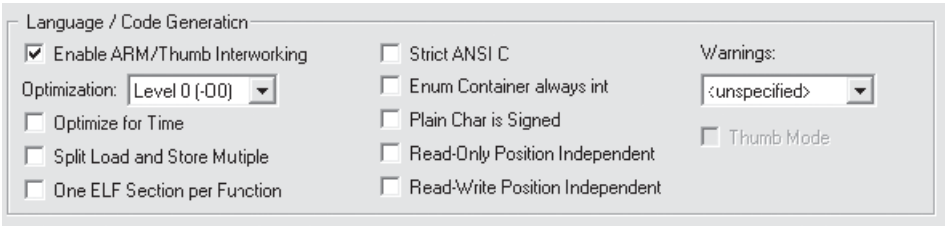


Рис. 2.12. Разрешение взаимодействия кода ARM/THUMB.

Эти же настройки доступны для каждой группы файлов с исходным кодом, а также для каждого Си-модуля. Таким образом, вы можете задать использование набора команд ARM или THUMB как для группы модулей, так и для отдельных модулей.

Кроме того, программист может явно указать, с использованием какого набора команд должна быть скомпилирована та или иная функция. Для этого предназначены две директивы `#pragma ARM` и `#pragma THUMB`, использование которых показано ниже. В этом примере основная функция компилируется с помощью набора команд ARM и вызывает функцию `thumb_function` (как легко догадаться, последняя скомпилирована с использованием 16-битного набора команд THUMB).

```
#pragma ARM                // Переключаемся на набор команд ARM
int main(void)
{
    while(1)
    {
       thumb_function();      // Вызываем THUMB-функцию
    }
}

#pragma THUMB              // Переключаемся на набор команд THUMB
void thumb_function(void)
{
    unsigned long i,delay;

    for (i = 0x00010000;i < 0x01000000 ;i = i<<1)   // Мигалка
    {
```

```
        for (delay = 0;delay<0x000100000;delay++)     // Простой цикл задержки
        {
            ;
        }
        IOSET1 = i;          // Указываем на следующий СИД
    }
}
```

2.3.1. Обеспечение взаимодействия в компиляторе GCC

Компилятор GCC тоже поддерживает взаимодействие между наборами команд ARM и THUMB. Чтобы разрешить такое поведение компилятора, необходимо в командную строку добавить ключ (**Рис. 2.13**)

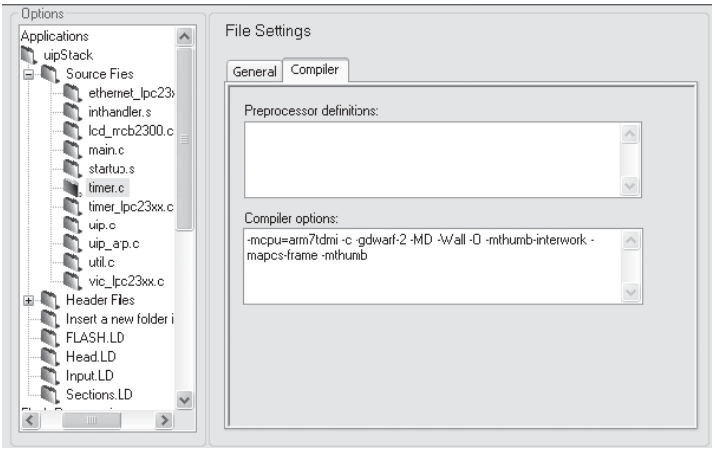
```
-mTHUMB-interwork
```

Наименьшим объектом, для которого компилятор GCC позволяет задать набор команд, является модуль. Соответственно, вы должны продумать структуру программы таким образом, чтобы каждый модуль содержал функции, компилируемые с применением только одного из наборов команд. По умолчанию компилятор для всех функций использует набор команд ARM. Чтобы скомпилировать какой-либо конкретный модуль с помощью набора команд THUMB, в командную строку компилятора следует добавить ключ

```
-mTHUMB
```

Далее компоновщик свяжет промежуточные ARM- и THUMB-объектные файлы в конечный исполняемый файл, содержащий оба набора команд.

После того как вы разберетесь с использованием в проекте обоих наборов команд, можно будет приступать к его компиляции. При этом вопросы взаимодействия ARM/THUMB будут решены компоновщиком автоматически. По мере раз-



В среде HiTOP можно легко задать индивидуальные параметры компиляции для каждого из модулей.

Рис. 2.13. Настройки проекта в ИСР HiTOP.

вития программы вы легко сможете изменить набор команд, используемый в проекте, модуле или даже функции, и перекомпилировать проект — это же так просто!

Упражнение 3. Совместное использование ARM и THUMB
В третьем упражнении рассматривается настройка проекта, в котором осуществляется взаимодействие между кодом ARM и THUMB.

2.4. Библиотека STDIO

Весь код, который будет встречаться в первых главах книги, выполняется без использования операционной системы. В этом случае поддержка стандартных потоков ввода/вывода stdin и stdout отсутствует. А чтобы воспользоваться любой из высокоуровневых библиотечных функций форматного ввода/вывода, нам потребуются низкоуровневые драйверы, обеспечивающие взаимодействие с конкретными устройствами ввода/вывода. Все функции низкого уровня, вызываемые библиотечными функциями из поставки компилятора Real View, описываются в одном единственном файле retarget.c. Шаблон этого файла можно найти в папке «C:\keil\arm\startup\». В частности, в файле retarget.c содержатся две функции, вызываемые высокоуровневыми библиотечными функциями `printf()` и `scanf()`. Эти функции низкого уровня используются для чтения одного символа из стандартного потока ввода stdin и записи символа в стандартный поток вывода stdout.

```
int fputc(int ch, FILE *f) {
    return (sendchar(ch));
}
int fgetc(FILE *f) {
    return (sendchar(getkey()));
}
```

Разумеется, должны быть описаны функции низкого уровня `sendchar()` и `getkey()`, которые будут читать и писать символ из/в потока ввода/вывода. Эти функции, поставляемые по умолчанию с компилятором, работают с модулем UART1, однако вы можете доработать их для использования с другими устройствами, например, модулями ЖКИ и клавиатуры, регистры которых отображены на память микроконтроллера.

```
int sendchar (int ch) {    /* Записать символ в последовательный порт  */
  if (ch == '\n') {
      while (!(U1LSR & 0x20));
      U1THR = CR;         /* Вывести символ CR                         */
  }
  while (!(U1LSR & 0x20));
  return (U1THR = ch);
}

int getkey (void) {        /* Считать символ из последовательного порта */
  while (!(U1LSR & 0x01));

  return (U1RBR);
}
```

2.4.1. Библиотека STDIO и компилятор GCC

Библиотеки компилятора GCC предназначены для работы под управлением операционной системы, соответствующей требованиям стандарта POSIX[1]. Это означает, что при вызове библиотечных функций библиотеки STDIO будут вызываться стандартные системные функции, которые должны были предоставляться операционной системой. Поскольку мы пишем программы на обычном процедурном Си без использования операционной системы, нам придется самим описывать такие функции. Библиотечные функции высокого уровня вывода данных вызывают низкоуровневую функцию `write()`, которую необходимо изменить таким образом, чтобы она выводила один символ в устройство stdout:

```
int write (int file, char * ptr, int len)
{
  int i;

  for (i = 0; i < len; i++) putchar (*ptr++);
  return len;
}
```

Аналогично, функции ввода, такие как `scanf()`, читающие данные из устройства stdin, вызывают для чтения одиночного символа функцию низкого уровня `read()`:

```
int read(int file, char *ptr, int len)
{
  int i;

  for (i=0;i<len;i++) *ptr++ = getchar();
  return len;
}
```

2.5. Организация доступа к периферийным устройствам

Раз уж мы пишем код, который хотим запустить на микроконтроллере, то рано или поздно нам потребуется обратиться к регистрам специальных функций (РСФ) периферийных устройств. Поскольку все периферийные устройства микроконтроллеров семейств LPC2300/2400 отображены на память данных, эти регистры можно рассматривать как обычные ячейки памяти. Таким образом, к любому РСФ можно обращаться посредством «фиксированного» volatile-указателя на занимаемую этим регистром ячейку памяти:

```
#define SFR (*((volatile unsigned long *) 0xFFFFF000))
```

Вместе с компилятором Real View поставляется набор включаемых файлов, в которых определяются РСФ для той или иной модели семейства. Включив соот-

---

[1] POSIX (Portable Operating System Interface (for UniX)) — стандарт переносимых операционных систем, а точнее, набор стандартов, призванных обеспечить как переносимость самих систем на различные архитектуры, так и работу написанных в соответствии со стандартами приложений в различных ОС. — *Примеч. пер.*

ветствующий файл в свою программу, вы сможете напрямую обращаться из нее к регистрам микроконтроллера. Все включаемые файлы имеют имена вида:

```
LPC23xx.h
LPC24xx.h
```

2.5.1. Организация доступа к периферийным устройствам в компиляторе GCC

Метод обращения по указателю, который применяется для доступа к регистрам специальных функций по их абсолютному адресу, полностью соответствует стандарту ANSI C. Поэтому с компилятором GCC можно использовать те же самые включаемые файлы.

2.6. Процедуры обработки прерываний

Помимо обращения к встроенным периферийным устройствам микроконтроллеров, в ваших программах должна осуществляться обработка запросов на прерывание. Превратить обычную функцию в процедуру обработки прерывания можно следующим способом:

```
void fiqint(void) __irq
{
    IOSET1 = 0x00FF0000;    // Включить СИД
    EXTINT = 0x00000002;    // Сбросить флаг прерывания
}
```

Ключевое слово `__irq` указывает на то, что эта функция является процедурой обработки быстрого (FIQ) или обычного прерывания, и в ней будет использован соответствующий механизм возврата. Обработка программных прерываний стоит немного особняком и будет рассмотрена в следующем подразделе.

Помимо объявления Си-функции в качестве процедуры обработки прерывания, вы должны связать ее с вектором прерывания. Как мы уже говорили, по умолчанию в таблице векторов загружаются константы, являющиеся адресами ловушек исключительных ситуаций (бесконечных циклов), расположенных в стартовом коде. Так что если вы просто разрешили обработку исключительной ситуации, не изменив стартовый код, то при возникновении этой ситуации программа зависнет на соответствующей ловушке. Поэтому, коль уж мы объявили функцию как обработчик исключительной ситуации, мы должны связать таблицу векторов с этой функцией, как показано на **Рис. 2.14**.

Поскольку ЦПУ ARM при входе в обработчик исключительной ситуации автоматически переключается на 32-битный набор команд ARM, все процедуры обработки прерываний и прочих исключительных ситуаций должны быть скомпилированы с использованием набора команд ARM. Несколько особняком стоят исключительные ситуации SWI и IRQ. Обработку программных прерываний SWI мы рассмотрим в следующем разделе, а обработку прерываний IRQ — в главе 3.

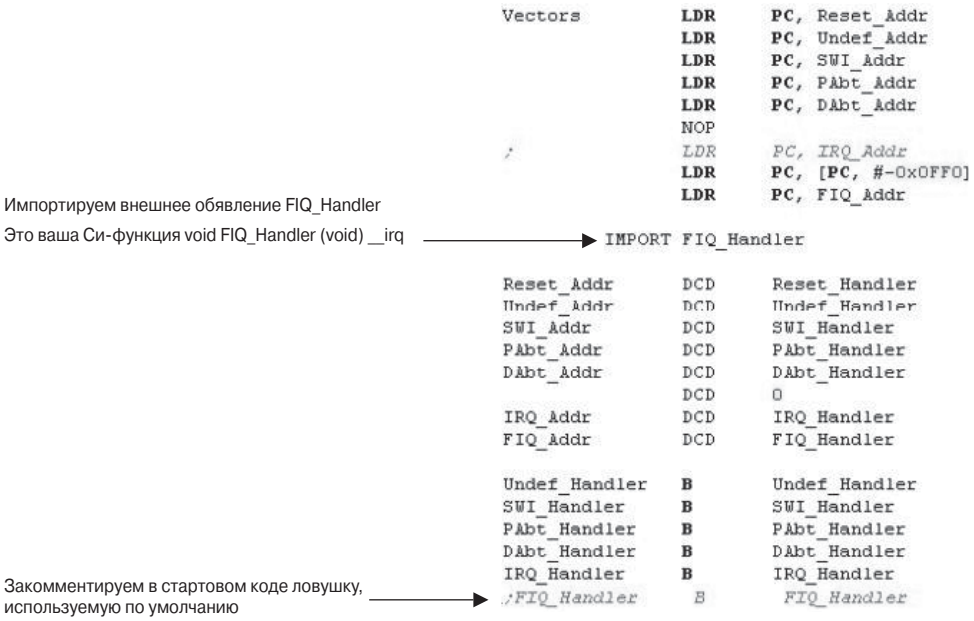


Рис. 2.14. Модификация таблицы векторов прерываний.

2.6.1. Обработка прерываний в компиляторе GCC

По большому счету, обработка исключительных ситуаций реализуется одинаково, что в компиляторе Real View, что в компиляторе GCC. Для объявления функции в качестве процедуры обработки прерывания в последнем используется нестандартное ключевое слово `__attribute__`. Общая форма такого объявления показана ниже.

```
void fiqint (void) __attribute__ ((interrupt("FIQ")));
{
  IOSET1 = 0x00FF0000;   // Включить СИД
  EXTINT = 0x00000002;   // Сбросить флаг прерывания
}
```

Для определения конкретного источника исключительной ситуации совместно с ключевым словом `__attribute__` используются следующие ключевые слова:

```
FIQ, IRQ, SWI, UNDEF
```

2.6.2. Отладка обработчиков системных ошибок

Три из поддерживаемых ядром ARM7 исключительных ситуаций предназначены для перехвата ошибок в программах. Исключительная ситуация «Undefined Instruction» генерируется в том случае, когда код, считанный из памяти, не является кодом какой-либо ARM- или THUMB-команды. Исключительные ситуации «Program Abort» и «Data Abort» генерируются в том случае, если при выборке команды или, соответственно, чтении/записи данных происходит обращение по ад-

ресу, не являющемуся адресом в ОЗУ, ПЗУ или адресом РСФ. При срабатывании любого из этих механизмов выполнение вашей программы прекращается, а управление передается на обработчик соответствующей исключительной ситуации, каждая из которых по умолчанию представляет собой пустые бесконечные циклы. Причем без трассировки программы в реальном времени обнаружить причину возникновения такой ситуации может быть весьма затруднительно.

Если в вашей программе случайно возникла ошибка обращения к памяти, в результате чего произошел переход к одному из упомянутых бесконечных циклов, вы можете проверить содержимое регистра связи режима Abort, чтобы определить адрес команды, выполнение которой вызвало ошибку (адрес команды равен считанному значению минус 4). А в регистре SPSR будет содержаться более подробная информация о режиме работы, в котором находилось ЦПУ на момент возникновения ошибки. Располагая этими данными, вы сможете «отмотать» ход событий и исследовать содержимое стека на момент, непосредственно предшествующий моменту сбоя программы. Таким образом, если вы знаете, что надо искать, из регистров ЦПУ можно извлечь определенную информацию для проведения такой «посмертной» диагностики (**Рис. 2.15**).

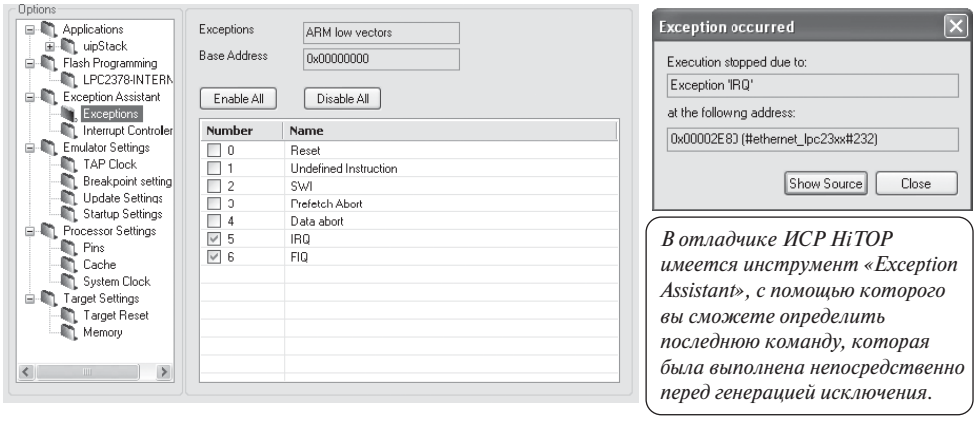


Рис. 2.15. Инструмент «Exception Assistant» отладчика HiTOP.

2.6.3. Программное прерывание

Как мы уже узнали в главе 1, в составе наборов команд ARM и THUMB имеется команда генерации программного прерывания. При выполнении команды `SWI` генерируется исключение, ЦПУ переключается в режим Supervisor, а в счетчик команд загружается адрес вектора программного прерывания. В привилегированном режиме Supervisor ЦПУ может обращаться к регистрам CPSR и SPSR, недоступным в режиме User. Помимо всего прочего, это означает, что код, выполняемый при обработке программного прерывания, будет иметь свой стек и, при необходимости, выделенную область памяти. Это дает возможность разбить весь код на отдельные блоки, выполняющиеся в различных режимах. В качестве примера можно привести прикладную программу, работающую в режиме User, кото-

рая посредством программных прерываний обращается к низкоуровневым драйверам, работающим в режиме Supervisor. Низкоуровневые драйверы могут иметь свой собственный стек и выделенную область памяти, которые не должны затрагиваться прикладной программой. В результате весь код разделяется на два уровня — уровень базового ввода/вывода BIOS и уровень собственно приложения. При этом код каждого из уровней может писаться и отлаживаться отдельно, а впоследствии их можно будет объединить с минимальными затратами. Механизм программных прерываний также может использоваться для связи между двумя отдельно скомпилированными программами, например, между прикладной программой и программой-загрузчиком.

Исключительная ситуация «программное прерывание» отличается от прочих исключительных ситуаций. Как мы уже говорили, в неиспользуемых битах слова команды `SWI` можно закодировать целое число.

```
#define      SWIcall2      asm{ swi #2 }
```

При входе в обработчик программного прерывания адрес возврата будет сохранен в регистре связи R14. Непосредственно в обработчике вы можете считать это значение и вычислить адрес команды `SWI`, сгенерировавшей прерывание. Затем можно будет прочитать код команды по этому адресу и выделить закодированное в данной команде целое число.

Однако в компиляторе Real View поддержка программных прерываний реализована более изящно. Процедуру обработки программного прерывания можно объявить с использованием следующего ключевого слова (не соответствующего стандарту ANSI):

```
void __swi(1) SysCall_1 (int pattern);

void __SWI_1 (void)       // Пустой вызов программного прерывания
{
 ......
}
```

Кроме того, в проект должен быть включен ассемблерный файл SWI.S (**Рис. 2.16**).

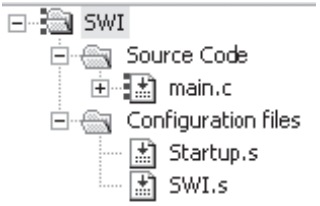


Рис. 2.16. Добавление поддержки программных прерываний.

Теперь при вызове функции будет генерироваться команда `SWI`. В результате процессор будет переходить в привилегированный режим Supervisor и выполнять код, содержащийся в файле SWI.S. Этот код определяет, какая из функций была

вызвана и обеспечивает передачу необходимых параметров. Такой механизм существенно облегчает использование структуры исключительных ситуаций процессора ARM7 и разбиение кода на второстепенный, выполняющийся в режиме User, и критический, такой как BIOS или функции операционной системы. В главе 8 этот вопрос рассматривается более подробно.

2.6.4. Программное прерывание и компилятор GCC

В компиляторе GCC отсутствует непосредственная поддержка механизма программных прерываний. Тем не менее, вызов программного прерывания можно описать, воспользовавшись встроенным ассемблером:

```
#define SoftwareInterrupt2 asm ("swi #2" )
```

В месте использования этого макроопределения компилятор вставит команду `SWI`, в неиспользуемых битах кода которой будет находиться заданное значение. Соответственно, при выполнении этой строки программы будет сгенерировано программное прерывание, ЦПУ переключится в режим Supervisor и перейдет по вектору SWI, в результате чего управление будет передано на обработчик программного прерывания. После входа в обработчик нам необходимо принять решение, какую из его секций следует выполнить. Разумеется, можно предусмотреть некоторую глобальную переменную и, используя оператор `switch`, выполнять те или иные функции в зависимости от значения этой переменной. Однако существует более изящное решение. Компилятор GCC позволяет объявлять указатели на регистры ЦПУ с использованием очередного нестандартного расширения:

```
register unsigned *link_ptr asm ("r14");
```

С помощью данного указателя мы сможем в процедуре обработки программного прерывания прочитать код команды `SWI`, вызвавшей прерывание, и, соответственно, хранящееся в нем целое число. И уже на основании этого значения определить, какая из функций должна быть вызвана обработчиком. Для извлечения данного числа можно использовать выражение:

```
temp = *(link_ptr - 1) & 0x00FFFFFF;
```

Значение, сохраненное в регистре связи, «откатывается» назад на одну команду (на 4 байта, так как используется указатель на 32-битное целое), чтобы указать на адрес команды `SWI`, сгенерировавшей исключительную ситуацию. Старшие 8 бит кода команды маскируются, и значение битов 0...23 заносится в переменную `temp`. В результате в переменной `temp` окажется содержащееся в коде команды целое число (в данном случае 2). Это значение можно будет использовать для определения кода, который необходимо выполнить. Команда `SWI` является удобным механизмом для перехода из режима User в режим Supervisor и выполнения какого-либо привилегированного кода. Вызовы этой команды могут содержаться в BIOS, в которой обращение к регистрам специальных функций осуществляется посредством вызова программных прерываний. В сущности, это позволяет разбить вашу прикладную программу таким образом, что все драйверы периферийных устройств будут выполняться в режиме Supervisor со своим индивидуальным стеком и, при необходимости, с отдельной областью памяти. Разумеется, вы вов-

се не обязаны проектировать код таким образом, однако если вы хотите это сделать, то у вас для этого есть все возможности.

Упражнение 4. Программное прерывание
В этом упражнении мы определим встраиваемую ассемблерную функцию для вызова программного прерывания и поместим в команду вызова число 0x01. В подпрограмме обработки программного прерывания мы декодируем эту команду, чтобы определить номер вызванной функции программного прерывания, а затем используем оператор switch() для перехода к соответствующей секции обработчика.

2.6.5. Размещение переменных по абсолютным адресам

Обращаю ваше внимание на то, что возможности, описанные в этом и следующем подразделах, в ознакомительной версии компилятора заблокированы.

Часто возникает необходимость размещения переменных по определенным адресам в памяти. Компилятор Real View позволяет разместить различные сегменты программы в пределах одного Си-модуля по любым адресам. Для этого достаточно выделить в окне менеджера проектов требуемый модуль и из контекстного меню выбрать пункт **Option**. В появившемся диалоговом окне вы сможете разместить сегменты данного модуля в любой из областей памяти, определенных в настройках проекта (**Рис. 2.17**).

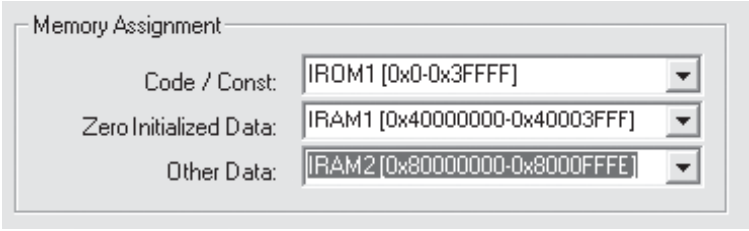


Рис. 2.17. Размещение сегментов модуля.

Если вы хотите разместить какую-либо переменную по конкретному адресу, к примеру, структуру поверх отображенных в память регистров внешнего периферийного устройства, то вы должны определить область памяти в настройках проекта. После этого надо будет создать модуль-«пустышку», содержащий только определение структуры, и разместить инициализированные данные (Zero Initialized Data) модуля в этом сегменте памяти.

2.6.6. Размещение кода в ОЗУ

Как мы увидим позже, самым узким местом в ЦПУ ARM7, ограничивающим его производительность, является выборка исполняемых команд из FLASH-памяти. В микроконтроллерах семейства LPC2300 имеется специальный модуль для решения этой проблемы при использовании встроенной FLASH-памяти. Однако если ваша программа хранится во внешней FLASH-памяти, вы в буквальном

смысле завязнете в ней из-за ее большого времени доступа. Одним из способов решения этой проблемы является перегрузка исполняемого кода в быстродействующее ОЗУ и последующее выполнение данного кода из ОЗУ. Для этого вам нужно будет скомпилировать либо переместимый код, который можно скопировать в ОЗУ, либо код, который можно запускать из ОЗУ и который загружается туда отдельной программой-загрузчиком. Оба этих решения будут работать, однако для их реализации придется приложить дополнительные усилия. К счастью, ИСР µVision позволяет легко перегружать код из FLASH-памяти в ОЗУ. Для этого достаточно просто указать, что секция кода и констант модуля (или группы модулей) должна располагаться в ОЗУ (**Рис. 2.18**).

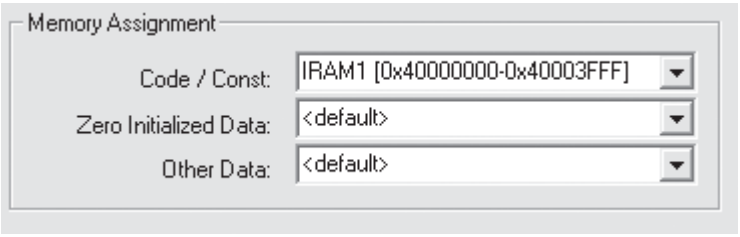


Рис. 2.18. Размещение кода в ОЗУ.

В результате после компиляции и компоновки выбранный код будет способен выполняться из ОЗУ, а стартовый код автоматически перегрузит его из FLASH-памяти в ОЗУ при выполнении программы. Обратите внимание, что компилятор никак не контролирует ситуацию, при которой ваша подпрограмма в ОЗУ вызывает другие, не находящиеся в ОЗУ. Поэтому если ваша «быстрая» функция вызывает подпрограмму математической библиотеки, находящуюся в FLASH-памяти, вы можете не получить ожидаемого быстродействия. Указанный способ размещения подпрограмм в ОЗУ не только прост и легок в использовании, он еще и облегчает работу компоновщику, так как последний знает итоговое место размещения подпрограммы и может разместить отладочную информацию по корректному адресу. В результате мы получим автономно выполняемый образ, пригодный не только для записи в ПЗУ и выполнения оттуда, но и для отладки.

2.6.7. Загрузка кода и данных в ОЗУ при использовании компилятора GCC

В компиляторе GCC отсутствует непосредственная возможность размещения объектов по заданным адресам, однако существует, по меньшей мере, несколько способов обойти это ограничение. Метод, рассматриваемый ниже, может использоваться как для загрузки кода в ОЗУ, так и для размещения переменных по фиксированным адресам. Этот метод основан на использовании директивы `SECTION __attribute`, которая позволяет разместить любой объект (код или данные) в секции с именем, определенным пользователем. Впоследствии указанная секция размещается в памяти по требуемому адресу, однако адрес времени вы-

полнения задается в произвольном месте ОЗУ посредством файла сценариев компоновщика.

В пошаговом виде этот процесс выглядит следующим образом:

1) Определяем макрос, скажем, так:

```
#define RAMCODE __attribute__ ((section (".ramcode")))
```

Имя RAMCODE было выбрано исключительно из соображений удобочитаемости, разумеется, оно может быть любым.

2) Помещаем макроопределение RAMCODE после объявления той функции, которую мы хотим хранить в ПЗУ, а запускать из ОЗУ:

```
// Функция стирания FLASH-памяти
// К объявлению функции добавляется атрибут
unsigned int erase_flash (unsigned int sector) RAMCODE ;
```

Замечание. Сама функция не меняется. Наличие в объявлении функции макроопределения RAMCODE и, соответственно, строки `__attribute__((section(".ramcode")))` приведет к тому, что код, сгенерированный для данной функции, будет помещен в секцию с именем .ramcode.

3) Добавляем в файл сценариев компоновщика с расширением .ld после строки `PROVIDE (etext=.);` следующий текст:

```
/* Секция .rodata, которая используется для хранения констант */

.rodata . :
{
*(.rodata)
} >IntCodeRAM

. = ALIGN(4);

/* Создаем символ, доступный из C */

_ramcode_rom_image = . ;
PROVIDE (ramcode_rom_image = .) ;

_etext = . ;
PROVIDE (etext = .);

/* ДОБАВЛЯЕМ СЛЕДУЮЩИЕ СТРОКИ */
/*****************************************/
/* Код, который будет записан в ПЗУ */
/* и скопирован в ОЗУ в main() */
/* Он размещается в IntDataRAM (SRAM), */
/* определенной в секции MEMORY { } в самом начале */
/* этого файла */

/* Секция .ramcode берется из файла RAMCODE.C */

.ramcode :

/* Помещаем образ ПЗУ в конец области .rodata */
AT ( ADDR (.rodata) + SIZEOF (.rodata))
{
/* Задаем значения публичных идентификаторов, названных нами */
/* _ramcode и _eramcode, которые будут использоваться в Си-программе */
```

```
_ramcode = . ; *(.ramcode) ; _eramcode = . ;

} >IntDataRAM

/* Создаем идентификатор _eramcode */
PROVIDE (_eramcode = .);
```

Теперь функция будет размещена в ПЗУ за секцией `.rodata`. Обычно эта секция находится в самом конце области Code/Constant.

В добавленных строках определяются публичные идентификаторы `_eramcode` (конец секции) и `_ramcode` (начало секции), которые используются процедурой копирования для определения конечного адреса. Идентификатор `ramcode_rom_image` используется для определения положения функции в ПЗУ. В Си-модулях ссылки на эти идентификаторы объявляются следующим образом:

```
extern unsigned long _eramcode ; // Конечный адрес области ОЗУ
extern unsigned long _ramcode ;  // Начальный адрес области ОЗУ, по которому будет
                                 // загружена функция
extern unsigned long _ramcode_rom_image ; // Начальный адрес области ПЗУ, в которой
                                          // находится код функции
```

4) Добавляем следующий код в начало вашей функции `main()`. Эти строки копируют код функции из ПЗУ в ОЗУ:

```
// Копируем функции стирания и записи FLASH-памяти в ОЗУ
// Указываем на начало образа функции в ПЗУ
func_copy_ptr = (unsigned long *) &_ramcode_rom_image ;
// Указываем на адрес, начиная с которого должен будет располагаться код функции
ramcode_ptr = (unsigned long *) &_ramcode ;
// Вычисляем размер копируемого участка памяти
func_copy_length = (unsigned long) &_eramcode - (unsigned long) &_ramcode ;
// Выполняем копирование
for(i = 0 ; i < func_copy_length/sizeof(unsigned int) ; i++)
{
    ramcode_ptr[i] = func_copy_ptr[i] ;
}
```

При этом уже должны быть объявлены следующие указатели:

```
unsigned long *func_copy_ptr ;
unsigned long *ramcode_ptr ;
unsigned long func_copy_length ;
```

5) Из-за огромной разницы между адресом в ПЗУ, из которого будет вызвана функция (0x00000000), и адресом в ОЗУ, по которому находится ее код (0xA0000000), напрямую эту функцию вызвать нельзя. Поэтому нам придется объявить указатель на функцию и вызвать ее уже по указателю (косвенный вызов):

```
// Берем стартовый адрес функции, уже расположенной в ОЗУ
// Мы не можем непосредственно вызвать функцию, расположенную по
// адресу 0xA0000000, поэтому нам придется воспользоваться указателем на нее
SRAM_erase_FLASH_func = (unsigned long (*)(unsigned int))&erase_flash ;
SRAM_write_FLASH_func = (unsigned long (*)(unsigned int, unsigned int))&write_flash ;
```

Собственно вызов функции, находящейся в ОЗУ, осуществляется следующим образом:

```
// Мы в 0-м банке
// Стираем сектор 4 по адресу 0x8000
error_status = SRAM_erase_FLASH_func(0x10) ; // Функция размещена в ОЗУ
```

6) Проверим map-файл и убедимся, что секция `.ramcode` расположена в требуемом месте, а образ функции находится в ПЗУ сразу за секцией `.rodata`:

```
.rodata 0x00000618 0x0
*(.rodata)
0x00000618 . = ALIGN (0x4)
0x00000618 _ramcode_rom_image = .
0x00000618 PROVIDE (ramcode_rom_image, .)
0x00000618 _etext = .
0x00000618 PROVIDE (etext, .)

.ramcode 0xa0000000 0x1dc load address 0x00000618
0xa0000000 _ramcode = .
*(.ramcode)
.ramcode 0xa0000000 0x1dc objects\ramcode.o
0xa0000000 erase_flash
0xa00000ec write_flash
0xa00001dc _eramcode = .
0xa00001dc PROVIDE (_eramcode, .)
```

Вот, в общем-то, и все!

2.7. Встраиваемые функции

Увеличения производительности программы можно также достичь, используя встраиваемые функции (inlining). Любую функцию можно сделать встраиваемой, указав при ее объявлении ключевое слово `__inline`:

```
__inline void NoSubroutine (void)
{
 ......
}
```

Если функцию объявить подобным образом, то подпрограммы как таковой создано не будет; вместо этого везде, где вызывается функция, будет вставлен ее код. При этом не потребуется выполнять код входа в подпрограмму и выхода из нее, что ускорит выполнение функции. Однако, поскольку код функции дублируется при каждом вызове, такое решение приводит к большим затратам FLASH-памяти.

2.7.1. Встраиваемые функции в компиляторе GCC

Компилятор GCC тоже поддерживает встраиваемые функции, причем они объявляются практически так же, как и в компиляторе Real View:

```
inline void NoSubroutine (void)
{
 .....
}
```

Замечание. Обратите внимание, компилятор не будет использовать встраиваемые функции до тех пор, пока не будет выбран уровень оптимизации «O2».

2.8. Встроенный ассемблер

Компилятор также позволяет вставлять ARM- и THUMB-команды ассемблера непосредственно в программы на языке Си. Для этого используется ключевое слово `__asm`:

```
__asm {
  loop:  LDRB ch, [src], #1
         STRB ch, [dst], #1
         CMP  ch, #0
         BNE  loop
}
```

Это может пригодиться в том случае, если требуется выполнить какие-либо действия, не поддерживаемые языком Си, например, вызвать команды `MRS` и `MSR`.

2.8.1. Встроенный ассемблер компилятора GCC

Компилятор GCC, как и компилятор Real View, тоже позволяет вставлять ассемблерные команды непосредственно в текст Си-программ:

```
asm ( "mov r15,r12" );
```

2.8.2. Импортирование программ для компилятора GCC

У компилятора Real View есть одна интересная функция — работа в режиме эмуляции GCC. При добавлении в строку вызова компилятора ключа `-gnu` (**Рис. 2.19**) компилятор Real View сможет компилировать проект, написанный под компилятор GCC. Поскольку оба рассматриваемых компилятора соответствуют стандарту двоичного интерфейса ARM, становится возможным создавать смешанные проекты, содержащие код, рассчитанный как на компилятор GCC, так и на компилятор Real View. В частности, благодаря такой функции, для компиляции существующих проектов со свободным исходным кодом (open source) можно использовать ведущие коммерческие компиляторы.

Рис. 2.19. Импортирование кода GCC.

2.9. Аппаратные средства отладки

При разработке семейства LPC2300 компания NXP позаботилась о наличии в микроконтроллерах семейства как можно более полной поддержки отладки. Причем эта поддержка осуществляется на нескольких уровнях. Самый простой

уровень — порт отладки по интерфейсу JTAG. С его помощью можно подключить микроконтроллер к ПК для проведения сеанса отладки (**Рис. 2.20**). Интерфейс JTAG предоставляет базовые средства управления функционированием кристалла. Так, вы можете пошагово выполнять программу, запускать и приостанавливать ее выполнение в реальном времени, устанавливать точки останова, а также просматривать при останове программы значения переменных и ячеек памяти.

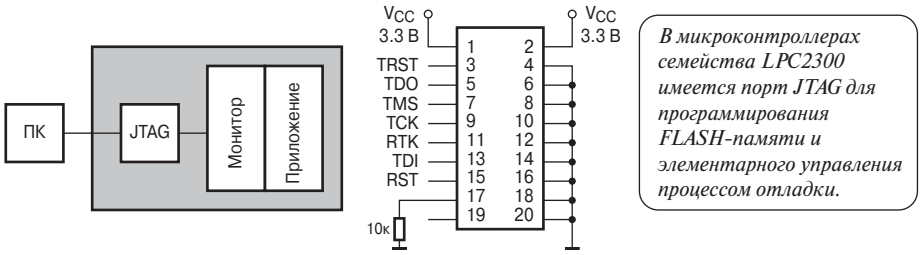


Рис. 2.20. Подключение микроконтроллера к ПК для отладки.

Кроме того, в составе микроконтроллеров семейства имеется модуль встроенной трассировки (ETM) от компании ARM. Этот модуль (**Рис. 2.21**) предоставляет гораздо более развитые средства отладки и трассировки в реальном времени, исследования кода программы и анализа эффективности. Помимо использования расширенных средств отладки, модуль ETM позволяет выполнять подробную верификацию кода и углубленное тестирование программного обеспечения, что невозможно сделать при использовании простого интерфейса JTAG. Это особенно важно в том случае, если вы разрабатываете приложения, для которых безопасность является критическим фактором.

Последним из встроенных отладочных средств является монитор реального времени (Real Time Monitor). Он представляет собой код, постоянно находящий-

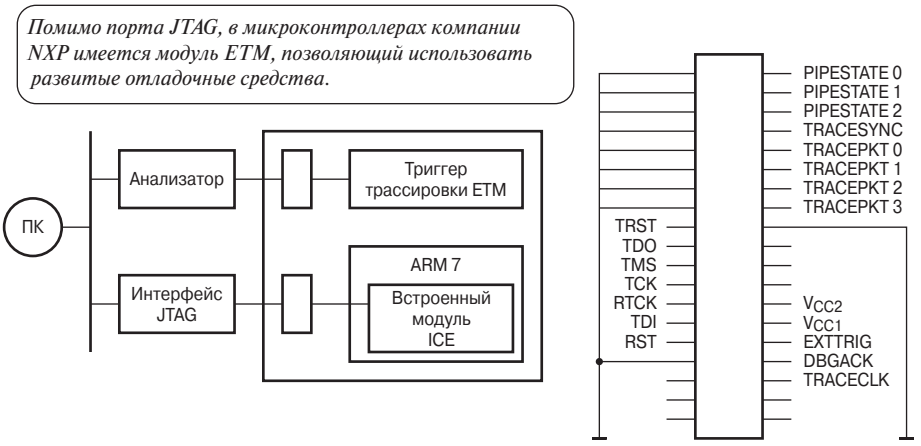


Рис. 2.21. Модуль ETM.

ся в зарезервированной области памяти. Во время сеанса отладки отладчик может запустить этот монитор через интерфейс JTAG. Монитор реального времени можно использовать для внесения изменений «на ходу», в процессе выполнения программы. На самом деле этот процесс не является процессом реального времени, поскольку монитор прерывает выполнение прикладной программы. Кроме того, ему требуется время для считывания и передачи отладочной информации на ПК.

2.9.1. Важное замечание!

Отладочные интерфейсы JTAG и ETM представляют собой довольно «тупые» средства отладки ядра ARM7 по последовательному каналу. Инструментальные средства ARM общего назначения, использующие интерфейс JTAG, не имеют никакого понятия обо всей архитектуре микроконтроллеров семейства LPC2300. Это означает, что при их использовании после сброса микроконтроллера всегда будет запускаться загрузчик, поскольку они не записывают в FLASH-память так называемую «сигнатуру программы» (эта особенность микроконтроллеров семейства будет обсуждаться позже). Соответственно, при их использовании ваша программа никогда не запустится. Если вы только начинаете изучать микроконтроллеры семейства LPC2300, такое поведение программ элементарно может поставить вас в тупик. Поскольку программные средства компании Keil разрабатываются специально для микроконтроллеров общего назначения с ядром ARM7, в ИСР μVISION заложена информация об архитектуре памяти семейства LPC2300, поэтому отладка будет осуществляться безо всяких проблем.

2.9.2. Еще более важное замечание!

Как уже было отмечено, порт JTAG является простым средством отладки ядра ARM7 по последовательному каналу. Очень важно хорошо представлять себе его поведение при сбросе. Когда ЦПУ ARM7 находится в состоянии сброса, все периферийные модули, включая модуль JTAG, тоже сброшены. При этом отладчик ULINK теряет контроль над кристаллом и должен восстановить его после того, как микроконтроллер выйдет из состояния сброса. Для этого потребуется определенное количество тактов. Все это время программа, содержащаяся в устройстве, будет выполняться в нормальном режиме. Как только отладчик перехватывает управление кристаллом, он выполняет программный сброс (soft reset) путем обнуления счетчика команд. Однако встроенные периферийные устройства микроконтроллера уже не находятся в состоянии сброса, т.е. периферия к этому моменту будет уже проинициализирована, прерывания разрешены и т.д. Вы должны учитывать это обстоятельство, если такое поведение микроконтроллера может воспрепятствовать нормальному функционированию разрабатываемой вами программы. Простейшим решением указанной проблемы будет вставка цикла задержки в стартовый код или в начало функции `main()`. После сброса ЦПУ будет «висеть» в этом цикле до тех пор, пока ULINK не восстановит контроль над кристаллом. При этом собственно приложение выполняться не будет, т.е. микроконтроллер останется в инициализированном состоянии.

2.10. Резюме

Итак, после прочтения этой главы вы должны уметь выполнять настройку проекта в средах компаний Keil и Hitex, задавать используемый компилятор и используемую модель микроконтроллера, конфигурировать стартовый код, обеспечивать взаимодействие между наборами команд ARM и THUMB, обращаться к периферийным устройствам микроконтроллеров семейства и объявлять Си-функции, являющиеся обработчиками исключительных ситуаций. Обладая этими знаниями, мы можем приступить к изучению системных периферийных устройств микроконтроллеров семейства LPC2300.