Программирование и отладка C/C++ приложений для микроконтроллеров ARM

Магда Ю. С.

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

Ю. С. Магда

ПРОГРАММИРОВАНИЕ И ОТЛАДКА С/С++ ПРИЛОЖЕНИЙ ДЛЯ МИКРОКОНТРОЛЛЕРОВ ARM

Москва, 2012

УДК 004.42:004.3'144:621.3.049.774ARM
ББК 32.973.26-018.2
М12

М12 **Магда Ю. С.**

Программирование и отладка C/C++ приложений для микроконтроллеров ARM. – М.: ДМК Пресс, 2012. – 168 с.: ил.

ISBN 978-5-94074-745-1

В книге рассмотрены практические аспекты программирования приложений для популярной микропроцессорной платформы ARM.

Материал книги имеет сугубо практическое направление, поэтому в ней приведено множество примеров, иллюстрирующих те или иные подходы при создании программ. Основной упор сделан на практические методы программирования задач на языке программирования C/C++, а также на решение проблем при отладке программ. Создание эффективного программного кода невозможно без применения тех или иных механизмов оптимизации, начиная с разработки эффективного кода в C++ и заканчивая низкоуровневой оптимизацией на уровне команд процессора, поэтому значительная часть материала книги посвящена практическим методам оптимизации приложений.

Для разработки, отладки и оптимизации демонстрационных приложений книги используется свободно распространяемая версия инструментального пакета фирмы Keil, при этом не требуется покупка каких-либо дополнительных аппаратных модулей с микроконтроллерами ARM.

Книга буде полезной в первую очередь разработчикам программного обеспечения систем на базе микроконтроллеров ARM, инженерам, студентам и всем, кто интересуется созданием устройств с ARM микроконтроллерами.

УДК 004.42:004.3'144:621.3.049.774ARM
ББК 32.973.26-018.2

ISBN 978-5-94074-745-1

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Сегодняшние достижения в области электроники и компьютерных технологий открывают все новые и новые перспективы для создания и применения устройств, спроектированных на микроконтроллерах и микропроцессорах, в самых различных сферах человеческой деятельности. Одним из наиболее популярных микропроцессорных платформ в настоящее время является ARM. За последние 10 лет архитектура ARM развивалась особенно интенсивно, опережая все остальные процессорные технологии. В настоящее время 32-разрядные микроконтроллеры ARM используются в наиболее быстро развивающихся сегментах рынка, ориентированных на мобильные устройства и устройства коммуникации. Ежегодно рынок ARM микроконтроллеров захватывает все новые и новые типы устройств и технологических решений, а количество выпускаемых кристаллов ARM исчисляется десятками миллионов.

На сегодняшний день ARM микроконтроллеры применяются практически везде, начиная с мобильных телефонов и заканчивая устройствами электронного управления автомобилями и GPS навигаторами. Более того, ARM микропроцессоры вторгаются в отрасли, ранее считавшиеся недоступными для данной технологии – это настольные и переносные компьютеры и серверные решения. Так, например, уже следующее поколение операционных систем Windows, анонсированное как Windows 8, наряду с процессорами Intel и AMD будет работать и на платформе ARM.

Эта книга посвящена анализу практических методов программирования систем с микроконтроллерами ARM на языке C/C++ . Разработка программной части таких систем занимает львиную долю времени в процессе проектирования, причем в этот процесс вовлечена масса программистов, для которых эта книга и предназначена в первую очередь. Помимо анализа элементов "чистого" программирования на C/C++ в данной книге рассматривается и другой, не менее важный аспект разработки программного обеспечения – отладка и оптимизация программного обеспечения для микропроцессоров ARM.

Производительность приложений, написанных на языке высокого уровня, даже на таком мощном, как C++, будет существенно зависеть от того, насколько хорошо разработчик понимает программную архитектуру ARM и владеет основами программирования на нижнем уровне. По этой причине значительная

часть материала книги посвящена анализу программных интерфейсов языков C/C++ и ассемблера, что позволяет достичь высокой производительности C/C++ приложений.

Важнейшим этапом разработки программного обеспечения является отладка программного кода приложения. Если для самых простых приложений процесс отладки может занять несколько минут, то комплексные приложения могут потребовать от инженера-разработчика значительных усилий и времени, сопоставимого с временем разработки исходных текстов. Для успешной отладки приложения, написанного на языке высокого уровня, программист должен знать основы язык ассемблера микроконтроллеров ARM – в этой книге данная тема рассматривается достаточно широко, так же как и основные методы анализа ошибок и оптимизации программного кода.

Материал книги имеет практическую направленность, поэтому читатели найдут здесь много примеров исходных текстов программ, иллюстрирующих теоретические аспекты тех или иных проблем. Книга будет полезна разработчикам программного обеспечения для микропроцессоров ARM, инженерам и всем желающим ознакомиться с принципами программирования и отладки программного обеспечения для данной архитектуры. Книга содержит несколько глав, краткое описание каждой из которых приводится далее.

- **Глава 1**. ***Программная архитектура микроконтроллеров с ядром ARM7***. В данной главе рассматривается программная архитектура микроконтроллеров ARM и специфика разработки приложений для RISC-процессоров. Часть материала главы посвящена особенностям аппаратной реализации микроконтроллеров ARM и взаимодействия различных функциональных узлов на кристалле микропроцессора. Рассматривается структура программного обеспечения для систем с микроконтроллерами ARM, а также варианты выбора той или иной модели проектирования для реализации программной части.
- **Глава 2**. ***Инструменты программирования микроконтроллеров ARM***. В этой главе дается общая оценка инструментальным средствам программирования систем на базе микроконтроллеров ARM, а также рассматривается один из наиболее популярных пакетов разработки приложений в среде C/C++ для ARM – компилятор Keil C. Значительная часть материала главы посвящена методам создания приложений, положенных в основу инструментального пакета Keil C для ARM. На примере простейшего приложения рассматриваются все этапы разработки пользовательского программного обеспечения, включая компиляцию исходных текстов, генерацию объектных модулей и сборку приложения.
- **Глава 3**. ***Программирование периферийных устройств микроконтроллеров ARM на Keil C***. В этой главе рассматриваются принципы функционирования периферийных устройств (таймеров, портов ввода-вывода, аналого-цифровых и цифро-аналоговых преобразователей), входящих в состав кристаллов современных микроконтроллеров ARM и практические методы программирования таких устройств. На практических примерах рассмот-

рено функционирование системы прерываний микроконтроллеров ARM. Все исходные тексты программ сопровождаются детальным описанием.

- **Глава 4**. ***Программный интерфейс C/C++ и ассемблера для микроконтроллеров ARM***. Материал главы посвящен реализации программного интерфейса между программным кодом, написанным на языке C/C++ и процедурами, разработанными на языке макроассемблера среды Keil. Рассматриваются соглашения о вызовах процедур и передаче параметров процедурам, доступ к общим данным и функциям, использование стека и т.д. Материал главы сопровождается многочисленными примерами программ, иллюстрирующими различные подходы при реализации интерфейса между фрагментами кода на C++ и ассемблере.
- **Глава 5**. ***Отладка программного кода микроконтроллеров ARM***. Данная глава посвящена анализу основных методов отладки программного кода и методике обнаружения часто встречающихся ошибок при программировании микроконтроллеров ARM. Часть материала главы посвящена основам дизассемблирования и анализа программного кода.
- **Глава 6**. ***Анализ и оптимизация программного кода микроконтроллеров ARM***. В этой главе дается подробный анализ методов оптимизации приложений на уровне исходных текстов программ. Рассматриваются такие методы оптимизации кода, как разворачивание циклов, применение инструкций условного выполнения, выбор типов переменных и т.д. Теоретический материал сопровождается многочисленными примерами исходных текстов программ.

ГЛАВА 1

ПРОГРАММНАЯ АРХИТЕКТУРА МИКРОКОНТРОЛЛЕРОВ С ЯДРОМ ARM7

В этой главе будут рассмотрены основы архитектуры ядра ARM7TDMI, которое является базовым модулем для многочисленной линейки процессоров ARM.

Ядро ARM базируется на архитектуре RISC (**R**educed **I**nstruction **S**et **C**omputing, архитектура процессоров, построенная на основе сокращенного набора команд), особенностью которой является использование простых и эффективных инструкций процессора, которые могут выполняться за один цикл. Базовые концепции RISC предполагают перенос основного акцента в разработке приложений с аппаратной части на программную, поскольку производительность приложений поднять значительно проще программными методами, чем используя сложные аппаратные решения.

Вследствие этого программирование RISC-процессоров выдвигает более существенные требования к эффективности компиляторов, по сравнению с архитектурой CISC (**C**omplete **I**nstruction **S**et **C**omputing, архитектура процессоров с широким набором различных машинных команд переменной длины и разным временем их исполнения). Микропроцессоры с CISC архитектурой (например, Intel x86) не столь требовательны к программным средствам разработки – здесь основной упор делается на производительность аппаратной части. На Рис. 1.1 показаны эти отличия.

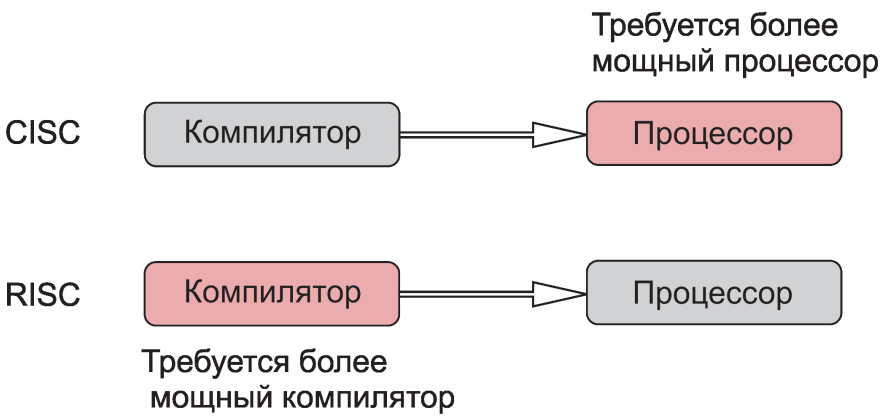

Рис. 1.1.

Если рассматривать архитектуру RISC более детально, то в ее основе лежат следующие базовые принципы, перечисленные далее.

1. **Система инструкций (команд) процессора.** Стандартный RISC-процессор имеет ограниченный набор типов инструкций, причем каждая из этих инструкций выполняется за один машинный цикл процессора. Разработка отдельных программных алгоритмов, например, деления, целиком возлагается на инструментальное средство разработки программ (компилятор) или самого разработчика. Каждая инструкция процессора имеет фиксированную длину, что позволяет успешно использовать принцип конвейера для выборки следующей инструкции в то время, пока предыдущая находится на стадии декодирования. В противоположность этому, в CISC-процессорах инструкции имеют различную длину и могут выполняться за несколько машинных циклов.

   Для иллюстрации этой концепции посмотрим, например, как выполняются операции сложения двух 32-разрядных целых чисел в процессорах CISC и RISC. Если, например, требуется сложить две переменные **i1** и **i2**, находящиеся в оперативной памяти, и сохранить результат в переменной **res**, также находящейся в памяти, то для Intel x86 процессора потребуются следующие команды:

```
MOV  EAX,  dword  ptr  a1
ADD  EAX,  dword  ptr  a2
MOV  dword  ptr  res,  EAX
```

   В эти вычисления вовлечен один регистр общего назначения **EAX**, а вычисление суммы (инструкция **ADD**) выполняется над операндами, расположенными в регистре и в памяти.

   Для RISC-процессора, например, ARM7 LPC2148, та же операция потребует следующей последовательности инструкций:

```
LDR    r0,  =i1
LDR    r1,  =i2
LDR    r3,  =res
LDR    r0,  [r0]
LDR    r1,  [r1]
ADD    r0,  r0,  r1
STR    r0,  [r3]
```

   Для выполнения любой операции в RISC-процессоре (кроме загрузки и сохранения данных, **LDR** и **STR** соответственно) оба операнда должны находиться в регистрах – это видно из мнемоники инструкции **ADD**, где переменные **i1** и **i2** находятся в регистрах **r0 r1** соответственно. Сравнивая два фрагмента кода для CISC и RISC процессоров, мы видим, что для RISC-процессора требуется больше инструкций. С другой стороны, большое количество регистров в ARM процессорах позволяет очень эффективно выполнять вычислительные операции с несколькими переменными, поскольку промежуточные результаты вычислений можно разместить в регистрах. Кроме того, в ARM процессорах можно использовать инструкции множественного доступа к нескольким ячейкам памяти, что повышает производительность операций чтения/записи данных.

Дополнительные возможности повышения производительности ARM микропроцессоров достигаются и при использовании инструкций условного выполнения, когда следующая инструкция выполняется только в том случае, если предыдущая инструкция установила определенные флаги в регистре состояния программы. Эти и другие возможности оптимизации быстродействия программного кода ARM мы рассмотрим далее в этой книге.

2. **Конвейер инструкций.** Процесс обработки каждой инструкции процессора разбивается на несколько этапов, которые выполняются одновременно. В идеальном варианте, для достижения максимальной производительности конвейер инструкций должен продвигаться на один шаг в каждом машинном цикле, при этом декодирование команды может осуществляться на одном шаге конвейера. Этот подход отличается от принятого для CISC-архитектур, в которых декодирование инструкций требует выполнения специальных микропрограмм.
3. **Использование регистров процессора.** В RISC-процессорах имеется набор многочисленных регистров общего применения. Каждый из регистров может содержать данные или адрес данных в памяти, поэтому регистры являются локальными хранилищами данных при выполнении всех операций в процессоре. Для сравнения: в CISC-процессорах имеется ограниченный набор регистров, каждый из которых имеет отдельное функциональное назначение, поэтому многие инструкции CISC в качестве одного из операндов используют ячейку памяти. В примере сложения двух чисел для CISC процессоров Intel x86 использовалась, например, инструкция **`ADD`**, одним из операндов которой являлась ячейка памяти. Такая инструкция требует достаточно много машинных циклов, что снижает производительность приложения, особенно если подобные инструкции используются при циклических вычислениях. Несмотря на существенные различия в архитектурах, в последнее время осуществляется постепенное сближение архитектур RISC и CISC. Так, например, CISC-микропроцессоры на уровне микропрограмм реализованы по принципам RISC, что увеличивает скорость выполнения микроинструкций.

Здесь нужно отметить еще один важный момент: базовое ядро ARM микроконтроллеров не вполне наследует концепции RISC по той причине, что специфика приложений для встроенных и мобильных систем требует большей гибкости и производительности, чем может обеспечить "чистая" RISC-архитектура. Кроме того, одним из основных требований к ARM-процессорам является низкое энергопотребление, поскольку мобильные и переносные устройства работают, как правило, с батарейным питанием.

Программная архитектура современных ARM-микроконтроллеров отличается от той, что реализована в традиционных RISC-устройствах, прежде всего в плане более расширенных возможностей инструкций процессора. Эти отличия обеспечивают более высокую производительность встроенных приложений – они описаны в следующем разделе.

1.1. Особенности выполнения инструкций микроконтроллеров ARM

Некоторые инструкции ARM требуют для выполнения не одного, как классические RISC команды, а нескольких машинных циклов. Так, например, инструкции множественной загрузки/сохранения (load–store–multiple instructions), в зависимости от количества задействованых регистров, используют разное количество машинных циклов. Обмен данными с памятью для таких инструкций осуществляется по последовательным адресам, что повышает производительность системы по сравнению с операциями с произвольным доступом.

Характерной особенностью ARM-процессоров является наличие встроенной многорегистровой схемы циклического сдвига (barrel shifter), которая позволяет выполнять сложные арифметические и логические операции в одной инструкции. Циклический сдвигатель реализован как аппаратное устройство на кристалле процессора и позволяет выполнять арифметические и логические сдвиги содержимого операнда-источника перед выполнением инструкции. С помощью схемы сдвига можно реализовать операции быстрого умножения на степень двух, а также комбинированные операции сложения/вычитания/умножение целых чисел.

Все инструкции процессора ARM могут выполняться в так называемом "условном" режиме (conditional execution). Это означает, что инструкция будет выполнена только в том случае, если условие, указанное в мнемонике инструкции, истинно. Например, команда сложения содержимого двух регистров **r0** и **r1** и результатом в регистре **r0** с безусловным выполнением имеет мнемонику

```
ADD  r0,  r0,  r1
```

Если эта же команда должна выполняться, только если предыдущие инструкции установили флаг **Z** регистре состояния, то мнемоника инструкции сложения будет следующей:

```
ADDEQ  r0,  r0,  r1
```

Инструкция будет пропущена, если требование Z=1 выполняться не будет. Условно выполняемые инструкции позволяют оптимизировать программный код за счет минимизации количества ветвлений, что очень важно при разработке приложений реального времени с ARM-процессорами.

В набор инструкций современных микропроцессоров ARM включена также группа инструкций, позволяющая программировать алгоритмы цифровой обработки сигналов. К этой группе относятся инструкции быстрого умножения 16 r 16 бит, а также инструкции умножения с насыщением. С помощью этой группы инструкций можно создавать эффективные программные алгоритмы без использования специального процессора цифровой обработки сигналов.

1.2. Основы аппаратной архитектуры микроконтроллеров ARM

Встроенные и мобильные устройства и системы широко используются для управления и измерения в самых различных отраслях науки, техники и производства. ARM микропроцессоры как нельзя лучше подходят для реализации таких систем. Кроме ядра, выполняющего функции вычислительного интеллектуального модуля, на кристалле ARM микроконтроллера имеется ряд устройств, которые позволяют строить интерфейс с внешним миром (датчиками, реле и двигателями, компьютерами и т. д.).

Ядро ARM, помимо реализации вычислительных функций, выполняет функции управления периферийными устройствами кристалла, такими, как порты ввода-вывода, таймеры, устройства оцифровки сигналов (аналого-цифровые и цифро-аналоговые преобразователи). Аппаратную часть современного ARM микропроцессора можно представить так, как показано на рис. 1.2.

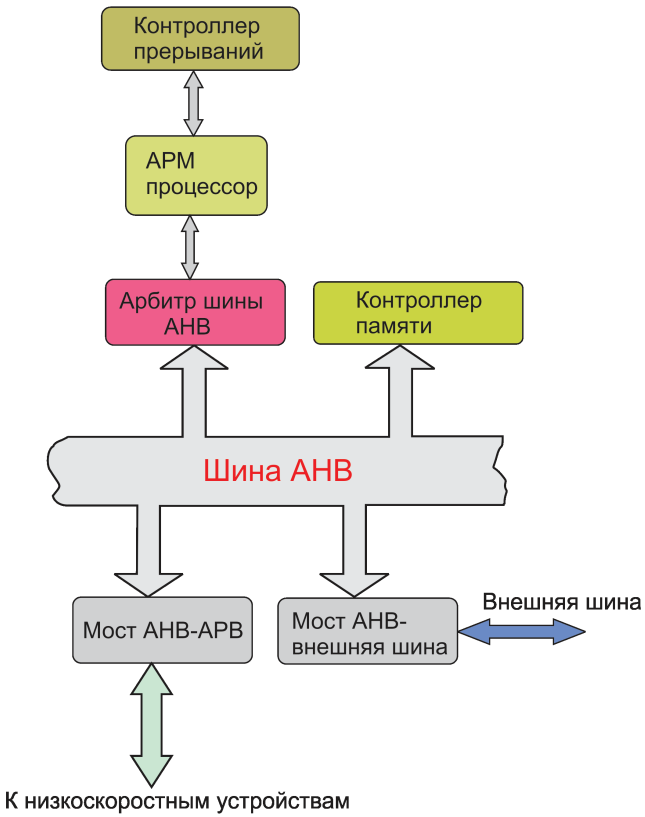

Рис. 1.2.

Рассмотрим эту схему детально, поскольку она отображает основные функциональные узлы и связи между ними, характерные для всех микроконтроллеров ARM. Важнейшим узлом в архитектуре микроконтроллера является контроллер прерываний, который осуществляет захват и приоритезацию внешних сигналов (событий), поступающих от периферийных устройств. Именно наличие этого функционального блока позволяет создавать системы реального времени, хотя существуют и другие подходы к построению таких систем.

Когда какое-либо устройство требует вмешательства со стороны процессора, оно выставляет сигнал прерывания. Контроллер прерывания, в свою очередь, управляет доступом к источнику прерывания со стороны программного обеспечения посредством установки соответствующих бит в регистрах прерываний. В микроконтроллерах ARM применяются два типа контроллеров прерываний – стандартный и векторизованный (Vector Interrupt Controller, VIC).

Стандартный контроллер прерывания просто посылает сигнал прерывания микроконтроллеру, когда какое-либо устройство требует немедленного обслуживания, и устанавливает сигнал прерывания на соответствующей линии. Такой контроллер можно запрограммировать так, чтобы он игнорировал или маскировал запросы на обслуживание от какого-либо устройства или целой группы устройств. Программа-обработчик прерывания определяет, какое устройство выставило запрос на обслуживание, посредством чтения регистра карты прерываний в контроллере прерываний.

Векторизованный контроллер прерываний выполняет более гибкую обработку запроса на обслуживание. Такой контроллер позволяет задавать приоритеты прерываний и упрощает процесс определения устройства, которое вызвало прерывание. После определения источника прерывания и его приоритета VIC-контроллер выставляет сигнал прерывания для микроконтроллера только в том случае, если приоритет только-что инициированного запроса выше, чем у прерывания, которое обслуживается в данный момент. В зависимости от способа реализации, векторизованный контроллер прерываний может либо вызвать стандартный обработчик прерывания, который, в свою очередь, может использовать адрес обработчика прерывания из VIC-контроллера, либо передать обработку прерывания микроконтроллеру. Во втором случае микроконтроллер вызывает обработчик прерывания напрямую.

Важнейшей функцией микроконтроллера является управление ресурсами памяти системы. Современные устройства ARM включают блоки управления памятью, архитектура которых может отличаться в зависимости функций, заложенных в устройство фирмами-производителями, однако принципы их построения одинаковы. В большинстве современных микроконтроллеров для хранения программного кода используется флэш-память, а данные хранятся в статической памяти.

Координация доступа к регионам памяти, занимаемых программным кодом и данными на аппаратном уровне, как раз и осуществляется посредством блока управления памятью. На уровне программы пользователя управление ресурсами памяти осуществляется посредством стандартных библиотечных функций

С (**malloc**, **realloc**, **free** и т. д.), которые оперируют с регионами памяти, находящимися в куче (heap). Эффективность использования и распределения памяти для работающего приложения существенно влияет на производительность системы, поэтому все современные компиляторы (Keil, IAR и др.) имеют целый ряд опций для настройки распределения ресурсов памяти.

Для управления периферийными устройствами и контроллером памяти в микроконтроллерах ARM реализована шинная архитектура. Эта архитектура кардинально отличается от той, что используется, например, в персональных компьютерах на базе процессоров x86 (Intel). В персональных компьютерах все внешние устройства подсоединяются к процессору через одну их шин PCI и/или PCI-Express, которые являются внешними по отношению к процессору. В противоположность этому, все шины микроконтроллеров ARM реализованы на кристалле процессора.

Все устройства, присоединеные к внутренней шине микроконтроллера ARM, могут работать в одном из двух режимов: ведущего (master) или ведомого (slave). Сам ARM микроконтроллер всегда работает в режиме ведущего – это означает, что он инициирует запросы на обмен данными для устройств, находящихся на шине. Периферийные устройства могут работать только в режиме ведомого и осуществлять обмен данными по запросу ведущего.

Архитектуру шины микроконтроллера можно представить в виде двух уровневой модели. Первый уровень реализует непосредственные физические соединения посредством электрических сигналов и характеризуется разрядностью шины, которая может варьироваться от 16 до 64. Второй уровень характеризуется тем, что здесь используются определенные протоколы – система логических правил, в соответствии с которыми выполняется обмен данными между процессором и периферийными устройствами.

В микроконтроллерах ARM применяется архитектура шин под названием AMBA (Advanced Microcontroller Bus Architecture), разработанная около двадцати лет тому назад. Эта архитектура была адартирована для применения с ARM микроконтроллерами. Первыми такими адаптированными шинами были ASB (ARM System Bus) и APB (ARM Peripheral Bus). Позже была разработана шина AHB (ARM High Performance Bus), которая в настоящее время применяется в большинстве микроконтроллеров.

Преимуществом использования шины AMBA является то, что разработчики периферийных устройств микроконтроллеров могут использовать одно и то же устройство во многих проектах без каких-либо изменений аппаратно-архитектурного решения. Если сравнивать различные модификации шин, то AHB способна обеспечить более высокую пропускную способность по сравнению с ASB. Это объясняется тем, что AHB является мультиплексированной шиной с централизованным управлением, в то время как ASB базируется на принципе двунаправленного потока данных. По этой причине шина AHB может функционировать при высоких тактовых частотах, а разрядность данных на этой шине может достигать 64 или 128 бит.

На Рис. 1.2 показана конфигурация кристалла ARM микроконтроллера с

тремя шинами: AHB – для подсоединения высокопроизводительных периферийных устройств, APB – для работы с низкоскоростными устройствами и внешняя шина для подсоединения внешних устройств. Обратите внимание на то, что для устройств на внешней шине требуется мост для перехода на шину AHB.

1.3. Программное обеспечение для систем с ARM микроконтроллерами

Программное обеспечение встроенных систем на микроконтроллерах в самом общем случае может состоять из четырех основных компонентов, показанных на Рис. 1.3. Каждый программный компонент в этом стеке использует определенный уровнь абстракции для разделения программного кода и аппаратного устройства, управляемого этим кодом. Код инициализации выполняется первым при сбросе или перезагрузке системы и определяется специфическими характеристиками данного типа процессора, архитектурой прерываний и системой управления памятью. Код инициализации обычно очень короткий, поскольку его основная функция – сконфигурировать базовые компоненты системы и передать управление загрузчику операционной системы.

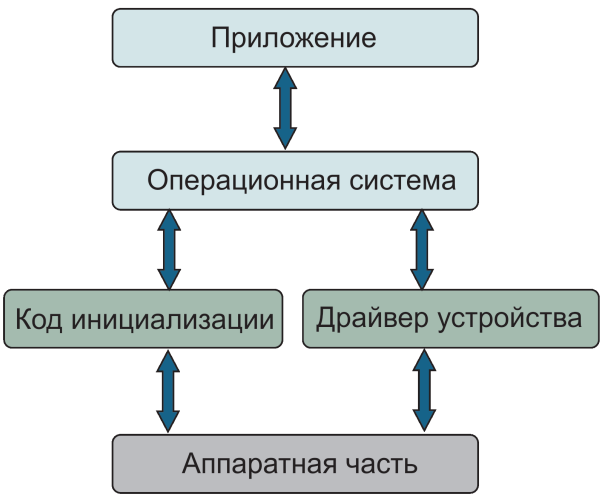

Рис. 1.3.

Операционная система выстраивает необходимую инфраструктуру для управления приложениями и аппаратными ресурсами. Для многих встроенных и переносных систем вообще не требуется комплексная операционная система – в этих случаях можно обойтись каким-либо простым менеджером заданий. Общая структура такого менеджера (Рис. 1.4), управляемого событиями, характерна для многих простых операционных систем реального времени (RTOS).

В такой системе менеджер заданий представляет собой программно реализованный "суперцикл" – это может быть обычный цикл с оператором **`while`** в C++. В нем каждому отдельному заданию для выполнения выделяется определенный промежуток (квант) времени, который контролируется посредством аппаратного таймера, входящего в состав встроенного периферийного оборудования микроконтроллера ARM.

По завершению кванта аремени менеджер заданий передает управление следующему в цепочке заданию и получает статус только-что выполненной операции (завершена, приостановлена и т. д.). Алгоритм управления заданиями может включать и дополнительный программный код, например, для возобновления приостановленного задания или продолжения выполнения незавершившегося задания в следующем кванте времени и т. д.

Рис. 1.4.

Вернемся к Рис. 1.3. Одним из программных компонентов, который управляет непосредственно периферийным устройством, является драйвер устройства. Понятие "драйвер устройства" для встроенных систем может интерпретироваться в широких пределах, в зависимости от структуры программного обеспечения системы в целом.

В простейших случаях, когда приложение пользователя управляет всеми периферийными устройствами системы, драйвер устройства – это фрагмент программного кода в самом приложении. Если приложения пользователя управляются комплексной операционной системой, то драйвер устройства разрабатывается как отдельное приложение с соответствующим программным интерфейсом для взаимодействия с другими уровнями программного обеспечения.

В любом случае конфигурация драйвера устройства во многом определяется требованиями к встроенной системе, а также тем, управляется ли система обычным приложением или же с помощью операционной системы. В следующих главах будут рассмотрены практические аспекты программирования встроенных систем на базе микроконтроллеров ARM, а также методы отладки и оптимизации программного кода.

ГЛАВА 2

ИНСТРУМЕНТЫ ПРОГРАММИРОВАНИЯ МИКРОКОНТРОЛЛЕРОВ ARM

Качество программ, разработанных для любой современной аппаратной платформы, жизненный цикл их разработки, а также возможности расширения и модернизации уже существующих приложений во многом определяются качеством применяемых инструментальных средств разработки. Для каждой платформы имеется множество инструментов разработки приложений на языках высокого уровня, каждое из которых обладает определенными преимуществами и недостатками. Для разработчиков C/C++ приложений созданы многочисленные инструментальные средства, позволяющие не только быстро создавать приложения, но и выполнять многие трудоемкие этапы создания приложений, такие, как отладка и оптимизация в сжатые сроки.

Среди множества инструментальных средств для разработки C/C++ приложений на определенной аппаратной платформе всегда можно выделить наиболее развитые, и по этой причине наиболее популярные инструментальные средства. Например, наиболее популярным инструментом разработки C/C++ приложений в операционных системах Windows на платформе x86 де-факто является Microsoft Visual C++. Многие принципы, используемые в этой среде, так или иначе реализованы в других средствах разработки третьих фирм, поэтому подходы, заложенные в Visual C++, стали фактически стандартными для разработчиков, использующих другие инструментальные средства.

Для платформы ARM7 и последующих в этой линейке фактическим стандартом стала среда разработки Keil C для ARM, работающая в операционных системах Windows. Эта глава посвящена базовым принципам разработки приложений для ARM микропроцессоров в этой среде.

2.1. Среда разработки Keil C и интерфейс пользователя µVision IDE

Средства разработки, входящие в состав пакета Keil C, интегрированы в графический интерфейс пользователя, известный под названием µVision IDE (мы будем работать с версией 4.0). Среда µVision IDE представляет собой многооконный интерфейс, как и большинство подобных пакетов, а также включает текстовый редактор для набора и редактирования исходных текстов программ и мастер проектов, который позволяет легко создавать шаблоны приложения

для конкретного типа ARM процессора. Мастер проектов выполняет создание приложения в несколько шагов, интуитивно понятных разработчику, оставляя, впрочем, возможности для ручной настройки параметров проекта уже после создания шаблона.

Средства разработки C/C++ приложений включают компилятор исходных текстов программ на C/C++, продвинутый макроассемблер, компоновщик (линкер) и генератор файлов в формате HEX (HEX-файлов). Среда µVision IDE включает интегрированную утилиту Make, позволяющую выполнить сборку, компиляцию и компоновку приложения в автоматическом режиме. В большинстве случаев использование Make вполне оправдано, однако разработчики при необходимости могут использовать и свои собственные скрипты, в которых могут быть заложены какие-то спефические требования. Кроме того, разработчик может настраивать параметры компилятора и линкера, вызывая определенные диалоговые окна и устанавливая параметры вручную. Во всех проектах из этой книги мы будем использовать утилиту Make для сборки приложений по умолчанию.

Будучи продвинутой средой разработки приложений, µVision включает отладчик/симулятор, а также специальный интерфейс для загрузчика ULINK Debug Adapter. Здесь я сделаю одно важное замечание. Для демонстрации работы программного кода мы будет использовать симуляторы аппаратных средств, что исключает необходимость приобретения платы (модуля) разработки с микроконтроллером ARM. Конечно, в этом случае вы не сможете записать код программы во флэш-память и запустить его на выполнение на железе. Тем не менее, симулятор Keil позволяет с высокой достоверностью моделировать аппаратный интерфейс микроконтроллера, не говоря уже о выполнении задач вычислительного характера, поэтому программный код, протестированный в симуляторе, будет работать и на реальном железе.

Обычно для адаптации программного кода для конкретного типа микроконтроллера требуются минимальные средства, поскольку среда µVision IDE позволяет генерировать программный код для конкретного типа устройства (как это делается, мы увидим на последующих примерах). Конечно, использование симулятора не может на все 100% заменить реальный физический микроконтроллер, однако большинство задач создания программного кода и его отладки могут быть успешно решены и в отсутствие реальных аппаратных средств.

Для построения всех примеров в среде Keil мы будем использовать демонстрационную версию Keil C с ограничением по размеру исполняемого кода 32K (версия 4.21). Несмотря на ограничение по размеру исполняемого кода, даже в этом случае можно научиться разрабатывать довольно сложные приложения. Вначале мы посмотрим, как можно быстро создать простейшее приложение наподобие “Hello, World” в среде Keil C для ARM, затем в деталях проанализируем те этапы которые требуется выполнить для превращения исходных текстов написанных на языке C/C++ в исполняемый двоичный файл (приложение).

2.2. Программа “Hello, World!” в среде Keil

Вначале посмотрим, как создется программа “Hello, World” в Keil C. Для этого нужно иметь установленный пакет Keil RealView MDK для ARM на вашем ПК (как минимум, демонстрационную версию).

На первом шаге создаем новый проект в среде Keil. В этом и других пакетах программирования проект будет включать несколько файлов (модулей), которые участвуют в построении данного приложения. Итак, приступим к созданию нового проекта (назовем его Hello). Для этого в основном меню выбираем опцию **New µVision Projec**t (Рис. 2.1):

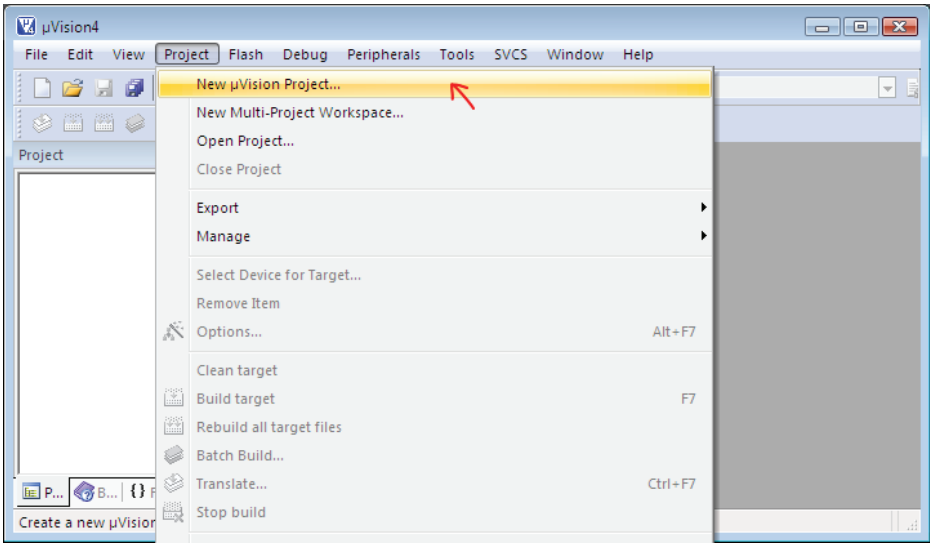

Рис. 2.1.

Данная опция позволяет создать проект, в который в процессе разработки будут добавлены необходимые файлы с исходными текстами программ, а также, при необходимости файлы заголовков и библиотеки функций. Проект можно интерпретировать как контейнер, в который или помещаются какие-то объекты или из него извлекаются (удаляются) ненужные объекты. Структура проекта полностью описывается файлом проекта с расширением .uvproj.

В открывшемся диалоговом окне выберем или создадим каталог, в котором будет сохранен наш проект (пусть это будет каталог Hello) и сохраним проект под именем Hello (Рис. 2.2).

После нажатия кнопки **Sav**e будет создан файл проекта Hello.uvproj. Файл проекта описывает все объекты проекта и связи между ними. На практике файл проекта является не чем иным, как XML-файлом, который, в принципе, может редактироваться вручную, если вдруг исходный файл по каким-то причинам был потерян.

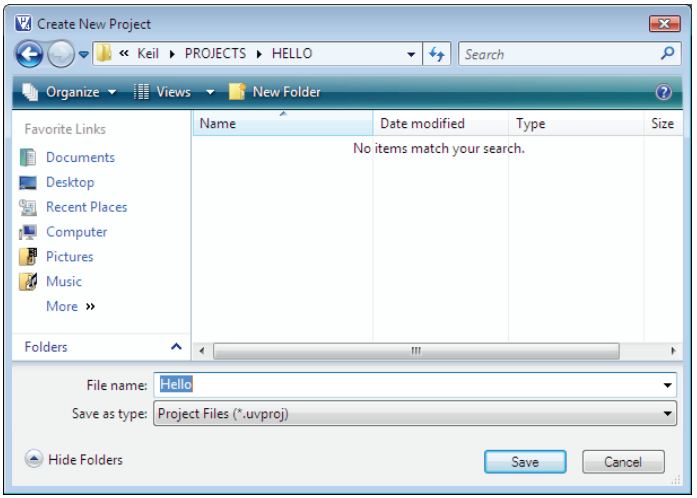

Рис. 2.2.

Содержимое части этого файла, открытого с помощью редактора MS Word, показано на Рис. 2.3. Если разрабатывается сложный проект, содержащий много настроек и много объектов, то полезно иметь сохраненную копию файла проекта, особенно на какой-то определенной стадии. Мастер проектов Keil также создает резервные копии файла проекта после изменений, но вам может понадобиться какая-либо промежуточная версия, которая к этому моменту времени уже могла быть перезаписана!

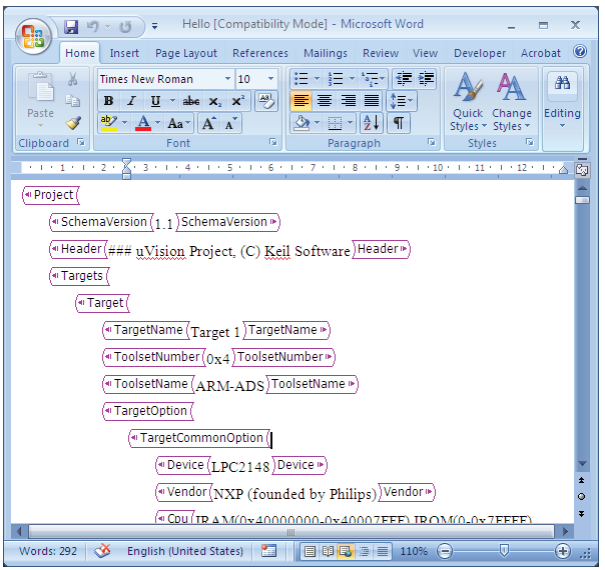

Рис. 2.3.

В меню **Project** (см. Рис. 2.1) имеются еще две опции, которые в данный момент нам не нужны, но которые могут понадобится в дальнейшей работе над проектами. Опция **Open Project...,** как следует из названия, позволяет открыть один из уже существующих проектов, хотя если вы постоянно работаете над каким-то проектом, то быстрее будет сделать то же самое, выбрав один из последних проектов в списке недавно открываемых проектов в нижней части меню.

Еще одна опция, **New Multi-Project Workspace**, предназначена для расширения возможностей по управлению проектами путем создания "рабочей области". Здесь нужно объяснить концепцию "рабочей области" или, что то же самое, "рабочего пространства" для нескольких проектов. В нашем случае при создании проекта Hello новый рабочий проект связан по умолчанию с одним рабочим пространством. Если бы возникла необходимость разработать, скажем, два проекта, в каждом из которых нужно было бы использовать файлы, общие для обоих проектов, то оба проекта можно было бы отладить в одном рабочем пространстве, что сэкономило бы время.

Еще одна ситуация, когда удобно использовать рабочее пространство, возникает, когда разработчик создает приложение для двух или более разнотипных ARM микропроцессоров, используя одни и те же исходные тексты программ. В этом случае можно создать несколько проектов для разных устройств и включить их в одно рабочее пространство.

Вернемся к нашему проекту. В следующем диалоговом окне мастер проектов предложит выбрать тип микроконтроллера от конкретного производителя (Рис. 2.4):

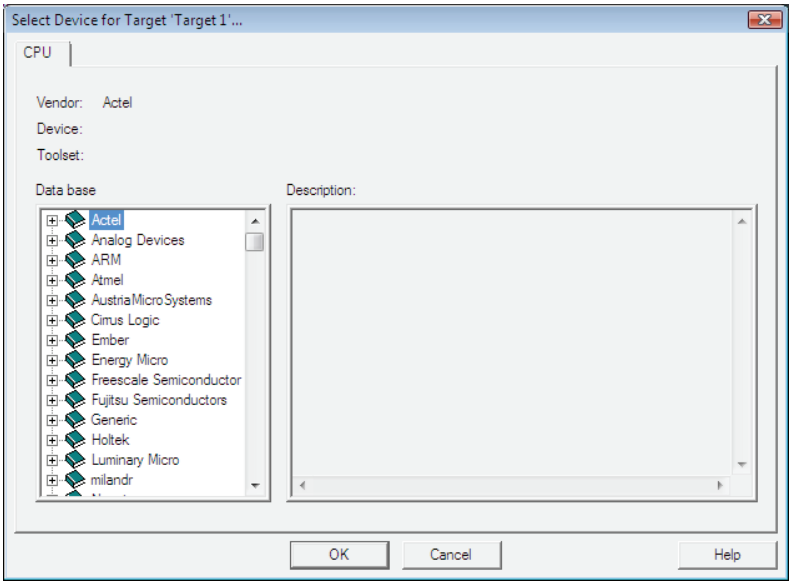

Рис. 2.4.

Если у вас имеется плата разработки с установленным на ней микроконтроллером, то нужно выбрать такое же устройство из списка слева. Если вашего микроконтроллера в списке нет, а такое может случиться, хотя и редко, особенно если данный кристал был запущен в производство буквально несколько недель тому назад, то временно можно воспользоваться похожим по параметрам микропроцессором. Часто значительные по объему фрагменты пользовательских программ отрабатывают какие-либо вычислительные алгоритмы, которые можно временно отлаживать и на других процессорах. В этих случаях можно воспользоваться и "виртуальным" процессором ARM, выбрав слева в списке имя **ARM**.

Если вам все же необходимо выбрать не виртуальное, а конретное устройство, то можно пойти другим путем. Можно попытаться найти похожий по функциональным признакам микропроцессор и разработать базовое программное обеспечение под этот тип устройства, а затем адаптировать программный код для конкретного устройства. Здесь, однако имеется несколько нюансов, которые необходимо принять во внимание. Если программа должна работать с конкретной периферией конкретного микроконтроллера, то никакие замены здесь не помогут – разработку программного кода в этой части придется отложить.

В том случае, если используются стандартные для многих микроконтроллеров периферийные устройства (скажем, последовательный интерфейс), то в принципе возможно написать фрагмент программы для обмена данными через последовательный интерфейс для одного ARM процессора и затем адаптировать его для другого.

Для задач вычислительного характера, в которых задействовано ядро процессора, программное обеспечение для одного типа микроконтроллера в принципе можно адаптировать для другого, но перед этим нужно уточнить используется ли в целевом устройстве сопроцессор, каким образом используется память и какие библиотечные функции задействованы в конкретном алгоритме.

Если для разработки выбран виртуальный микроконтроллер, то в раскрывающемся списке справа вы увидите описание стандартного базового процессора ARM7 (Рис. 2.5).

Использование такого виртуального процессора позволяет успешно продвигать разработку даже при временном отсутствии целевого процессора. Обычно такие крупные производители програмного обеспечения как Keil, IAR и другие обновляют свои базы устройств очень быстро, поэтому в следующих обновления программы процессор, как правило, уже будет в списке.

Для разработки проектов из этой книги будет использоваться процессор LPC2148 фирмы NXP, поэтому выберем его из списка устройств (Рис. 2.6).

Разработать большую часть программного обеспечения для конкретного типа микроконтроллера, перечисленного в списке, можно даже в отсутствие модуля разработки с данным устройством – среда μVision имеет встроенный высококачественный симулятор, который в большинстве случаев помогает протемтировать и отладить большую часть приложения без железа.

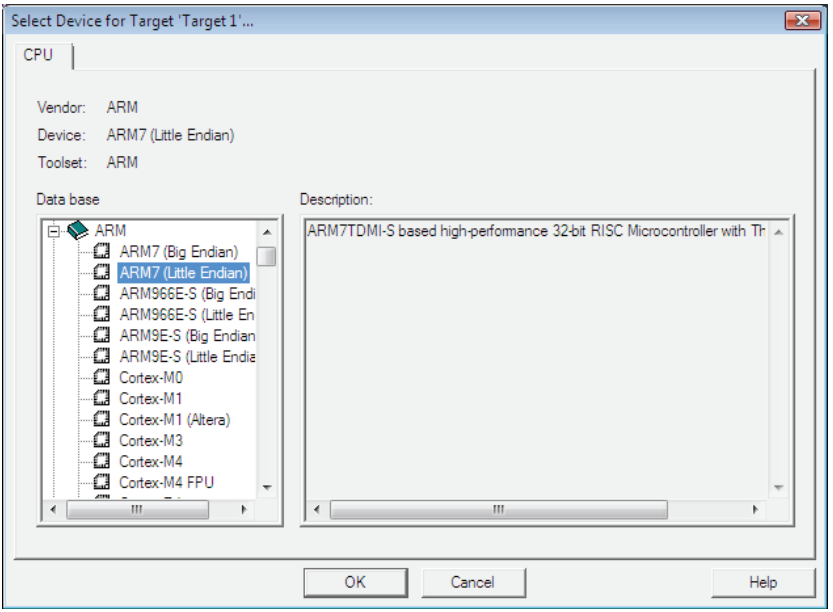
Select Device for Target 'Target 1'...
CPU
Vendor: ARM
Device: ARM7 (Little Endian)
Toolset: ARM
Data base
Description:
ARM
ARM7 (Big Endian)
ARM7 (Little Endian)
ARM966E-S (Big Endi
ARM966E-S (Little En
ARM9E-S (Big Endian
ARM9E-S (Little Endia
Cortex-M0
Cortex-M1
Cortex-M1 (Altera)
Cortex-M3
Cortex-M4
Cortex-M4 FPU
ARM7TDMI-S based high-performance 32-bit RISC Microcontroller with Th
OK
Cancel
Help

Рис. 2.5.

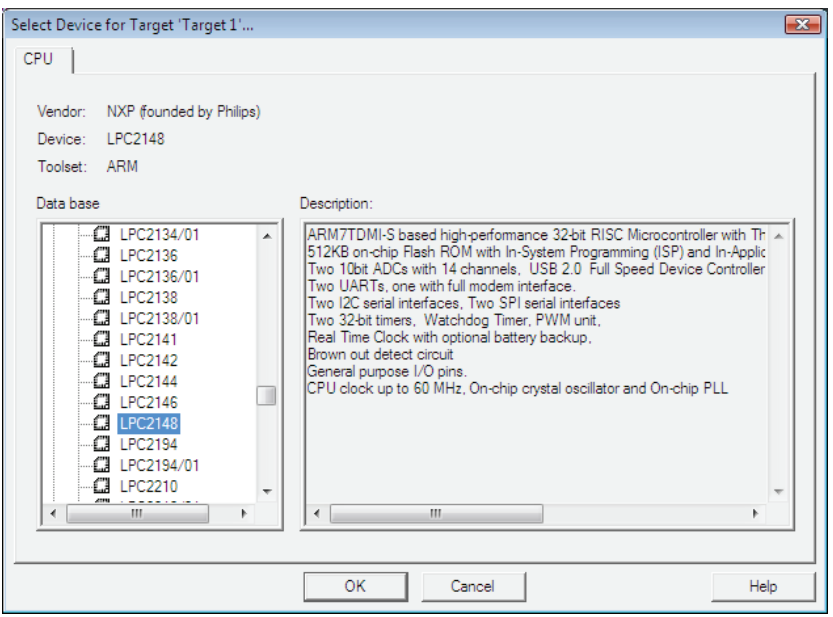
Select Device for Target 'Target 1'...
CPU
Vendor: NXP (founded by Philips)
Device: LPC2148
Toolset: ARM
Data base
Description:
LPC2134/01
LPC2136
LPC2136/01
LPC2138
LPC2138/01
LPC2141
LPC2142
LPC2144
LPC2146
LPC2148
LPC2194
LPC2194/01
LPC2210
ARM7TDMI-S based high-performance 32-bit RISC Microcontroller with Th
512KB on-chip Flash ROM with In-System Programming (ISP) and In-Applic
Two 10bit ADCs with 14 channels, USB 2.0 Full Speed Device Controller
Two UARTs, one with full modem interface.
Two I2C serial interfaces, Two SPI serial interfaces
Two 32-bit timers, Watchdog Timer, PWM unit,
Real Time Clock with optional battery backup,
Brown out detect circuit
General purpose I/O pins.
CPU clock up to 60 MHz, On-chip crystal oscillator and On-chip PLL
OK
Cancel
Help

Рис. 2.6.

Выбрав тип процессора, перейдем в следующее окно, где мастер проектов предложит сгенерировать код инициализации для данного типа прцессора (Рис. 2.7):

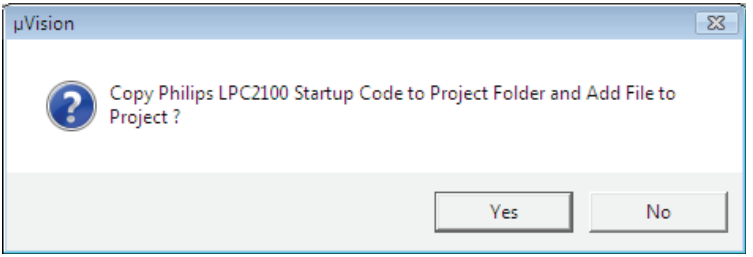

Рис. 2.7.

Мастер проектов предложит сгенерировать код инициализации, с чем мы согласимся нажав **Yes**. Код инициализации выполняется всякий раз при перезагрузке микроконтроллера и содержит секции инициализации стека, памяти и прерываний. Исходный текст программы инициализации написан на языке ассемблера и описывает последовательность операций, выполняемых при инициализации. Для эффективной работы над проектом разработчик программного обеспечения должен ориентироваться в том, как выполняется конфигурирование аппаратных средств. Вот, например, фрагмент исходного текста кода, инициализирующий таблицу векторов прерываний микроконтроллера:

```
Vectors          LDR          PC,  Reset_Addr
                 LDR          PC,  Undef_Addr
                 LDR          PC,  SWI_Addr
                 LDR          PC,  PAbt_Addr
                 LDR          PC,  DAbt_Addr
                 NOP                          ;  Reserved  Vector
;                LDR          PC,  IRQ_Addr
                 LDR          PC,  [PC,  #-0x0FF0]        ;  Vector  from
VicVectAddr
                 LDR          PC,  FIQ_Addr

Reset_Addr       DCD          Reset_Handler
Undef_Addr       DCD          Undef_Handler
SWI_Addr         DCD          SWI_Handler
PAbt_Addr        DCD          PAbt_Handler
DAbt_Addr        DCD          DAbt_Handler
                 DCD          0              ;  Reserved  Address
IRQ_Addr         DCD          IRQ_Handler
FIQ_Addr         DCD          FIQ_Handler

Undef_Handler     B            Undef_Handler
SWI_Handler       B            SWI_Handler
PAbt_Handler      B            PAbt_Handler
DAbt_Handler      B            DAbt_Handler
IRQ_Handler       B            IRQ_Handler
FIQ_Handler       B            FIQ_Handler
```

Программный код инициализации зависит от конкретного типа микроконтроллера и целиком определяется его архитектурными особенностями, поэтому даже для двух устройств, находящихся рядом в линейке, код может заметно отличаться. Даже если вы планируете в дальнейшем каким-то образом откорректировать исходный текст загрузчика, это будет сделать значительно легче, используя готовый исходный текст. После того, как вы согласитесь с мастером проектов, в наш проект будет добавлен файл Startup.s, содержащий исходный текст кода инициализаци (Рис. 2.8):

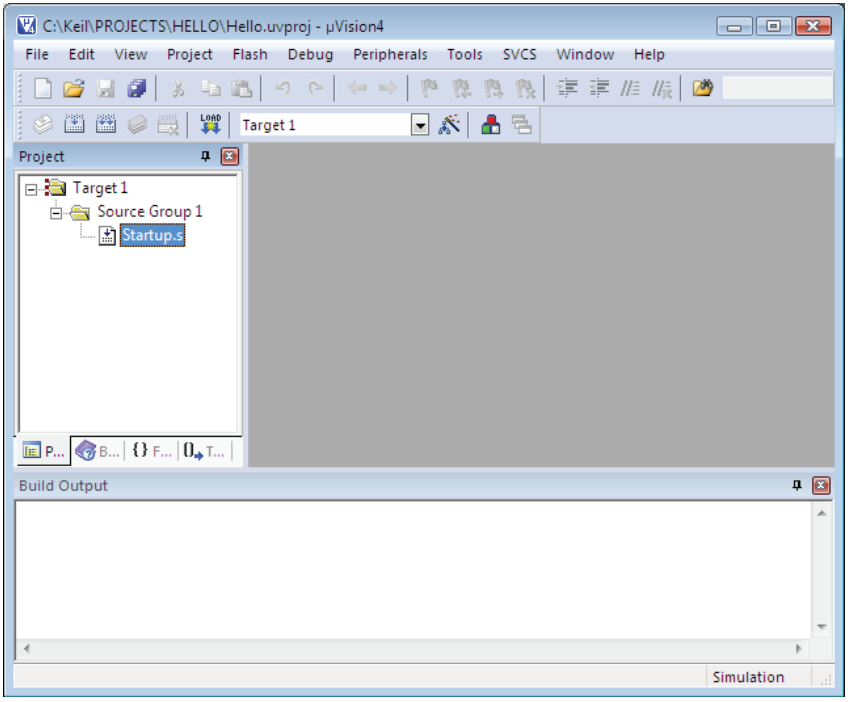

Рис. 2.8.

Кроме Startup.s мы должны включить в наш проект еще два стандартных файла, Serial.c и Retarget.c. Оба файла специфичны для каждого типа процессора, поэтому для включения этих файлов в проект мы должны, во-первых, скопировать подходящие файлы из одного из примеров, разработанных Keil в каталог нашего проекта, а затем добавить оба файла в наш проект. Для того, чтобы вы ориентировались какие файлы Retarget.c и Serial.c нужно выбрать, ниже дается исходный текст обоих файлов. Напомню, что мы выбираем файлы для микроконтроллеров LPC21xx, для других типов нужно использовать такие же файлы, но содержащие другой исходный текст.

Итак, содержимое нашего файла Retarget.c должно выглядеть так, как показано в Листинге 2.1:

Листинг 2.1

```
/*****************************************************************/
/*RETARGET.C: The layer for target-dependent low level functions*/
/*****************************************************************/

#include <stdio.h>
#include <rt_misc.h>

#pragma import(__use_no_semihosting_swi)

extern int sendchar(int ch); /* in serial.c */

struct __FILE { int handle; /* Add whatever you need here*/ };
FILE __stdout;

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

int ferror(FILE *f) {
    /*Your implementation of ferror*/
    return EOF;
}

void _ttywrch(int ch) {
    sendchar(ch);
}


void _sys_exit(int return_code) {
label:  goto label;  /*endless loop*/
}
```

Содержимое файла Serial.c должно быть таким (Листинг 2.2):

Листинг 2.2

```
/*****************************************************************************/
/* SERIAL.C: Low Level Serial Routines                                       */
/*****************************************************************************/
/* This file is part of the uVision/ARM development tools.                   */
/* Copyright (c) 2005-2006 Keil Software. All rights reserved.               */
/* This software may only be used under the terms of a valid,                */
/* current, end user licence from KEIL for a compatible version              */
/* of KEIL software development tools.                                       */
/* Nothing else gives you the right to use this software.                    */
/*****************************************************************************/

#include <LPC21xx.H>                    /* LPC21xx definitions               */

#define CR          0x0D

void init_serial (void)    { /* Initialize Serial Interface                */
```

```
    PINSEL0 = 0x00050000;    /* Enable RxD1 and TxD1      */
    U1LCR = 0x83;    /* 8 bits, no Parity, 1 Stop bit  */
    U1DLL = 97;      /* 9600 Baud Rate @ 12MHz VPB Clock  */
    U1LCR = 0x03;    /* DLAB = 0  */
}

/* implementation of putchar  */
/* (also used by printf function to output data)  */

int sendchar (int ch)
{ /* Write character to Serial Port  */
  if (ch == '\n')
  {
    while (!(U1LSR & 0x20));
    U1THR = CR;  /* output CR  */
  }
  while (!(U1LSR & 0x20));
  return (U1THR = ch);

}

int getkey (void)

{ /* Read character from Serial Port  */

  while (!(U1LSR & 0x01));    /* wait until character ready  */

  return (U1RBR);
}
```

Файл Serial.c содержит вспомогательные функции которые используются стандартными функциями ввода-вывода высокого уровня, такими, например, как широко известные **`printf()`** и **`scanf()`**, хотя операции, выполняемые этими функциями отличаются от тех, которые выполняются в классических версиях C/C++.

Предположим, что файлы Retarget.c и Serial.c уже находятся в каталоге нашего проекта. Теперь нужно добавить оба файла в наш проект. Для этого в окне **Project** нужно щелкнуть правой кнопкой мыши на строке **Source Group 1** и выбрать опцию **Add Files to Group 'Source Group 1'...** (Рис. 2.9).

После того, как файлы было добавлены в проект, окно проекта должно выглядеть так, как показано на Рис. 2.10.

При добавлении файлов в проект с целью упрощения мы поместили все файлы в единственную группу "Source Group 1", хотя при разработке больших проектов часто бывает удобно создавать несколько групп содержащих файлы по определенному функциональному признаку. Таким образом, на данный момент мы имеем все необходимые файлы, кроме одного, который будет содержать нашу программу. Пусть это будет C++ файл Hello.cpp. Для создания этого файла нужно в меню файл выбрать опцию **New...** (Рис. 2.11), затем в открывшемся текстовом редакторе набрать исходный текст файла Hello.cpp.

Исходный текст нашего файла может выглядеть так, как показано в Листинге 2.3:

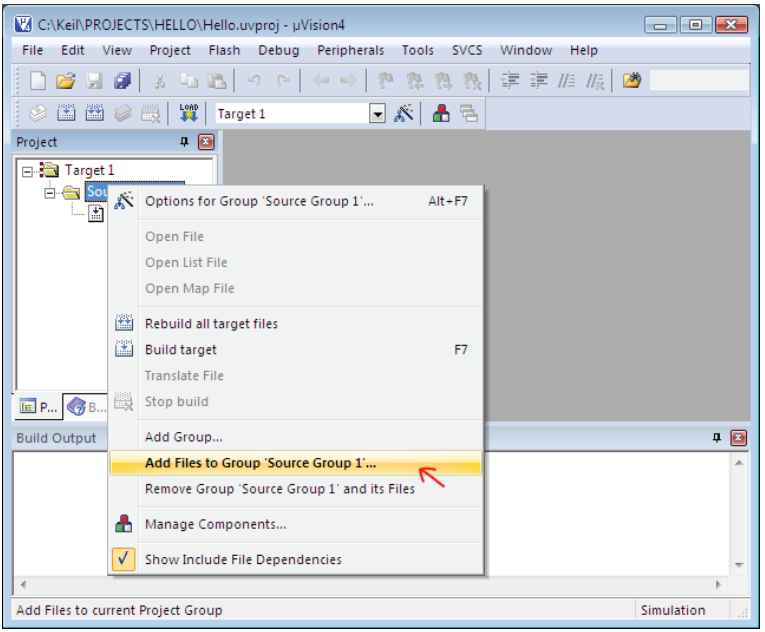

Рис. 2.9.

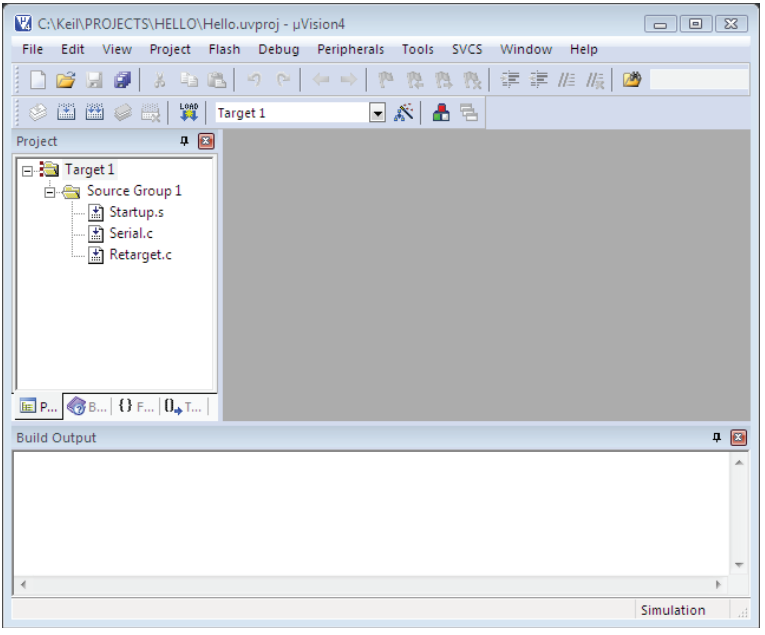

Рис. 2.10.

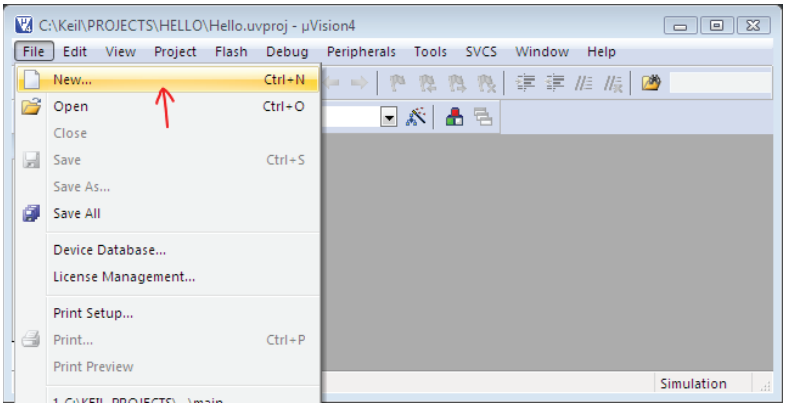

Рис. 2.11.

Листинг 2.3

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

int  main(void)
{
  init_serial();
  printf("  Hello,  world!\n");
  return  0;
  }
```

Наш листинг содержит всего две функции – **`init_serial()`** и **`printf()`**. Первая нужна для инициализации вывода, а вторая используется для непосредственного вывода данных на стандартный вывод. После набора исходного текста нужно сохранить содержимое Листинга 2.3 в файле Hello.cpp, выбрав опцию **Save as…** в меню **File**. На последнем шаге нужно добавить файл Hello.cpp к нашему проекту, как мы это сделали ранее для файлов Retarget.c и Serial.c.

На данном этапе мы имеем все необходимые исходные файлы для создания нашего первого приложения для ARM микроконтроллера. Теперь мы можем создать наше приложение, выбрав опцию **Build** в меню **Project** (Рис. 2.12).

При выборе этого пункта меню начинается процесс сборки приложения, по завершению которого мы получаем (если не было допущено ошибок при наборе исходного текста) исполняемый файл hello.axf в двоичном формате **`ELF`** (Рис. 2.13).

Если планируется генерация двоичного файла для записи во флэш-пямять микроконтроллера, то нужно создавать двоичный исполняемый файл в **`HEX`**-формате (**`HEX`**-файл). В этом случае необходимо указать такую возможность в опциях проекта. Вначале в окне проекта выбираем опцию **Target1** и, после появления выпадающего меню, выбираем опцию **Options for Target 'Target1'…** (Рис. 2.14).

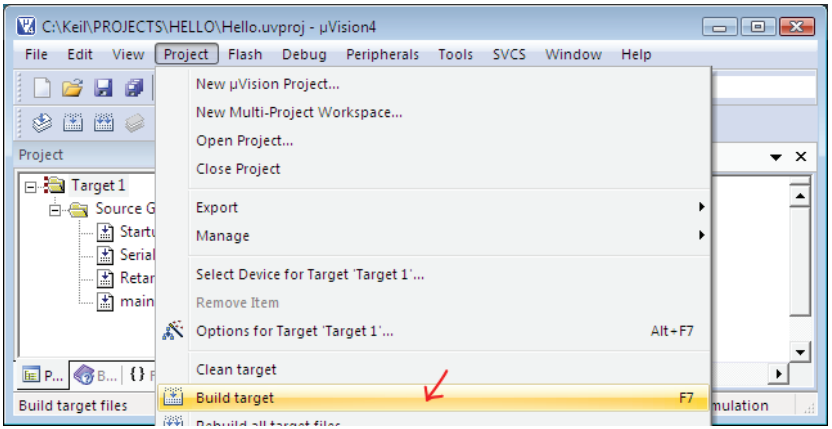

Рис. 2.12.

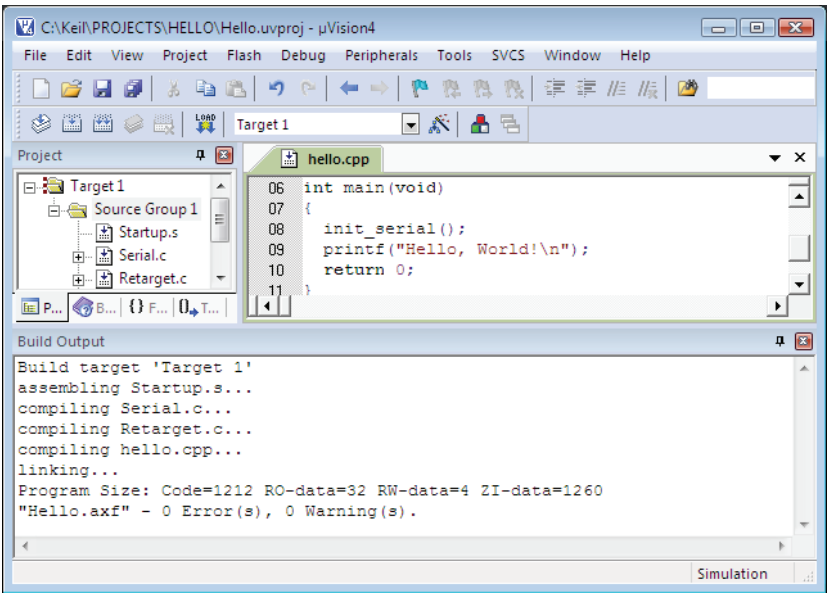

Рис. 2.13.

Затем в раскрывающемся окне выберем закладку **Output** и в раскрывшемся окне установим отметку напротив опции **Create HEX File** (Рис. 2.15).

После этого следует перекомпилировать проект – в результате будет сгенерирован **HEX**-файл. Для повторной компиляции следует воспользоваться опцией **Rebuild all target files** в меню **Project**.

Теперь мы можем проверить работоспособность нашего приложения, используя встроенный симулятор Keil. Вначале следует выбрать симулятор в качестве

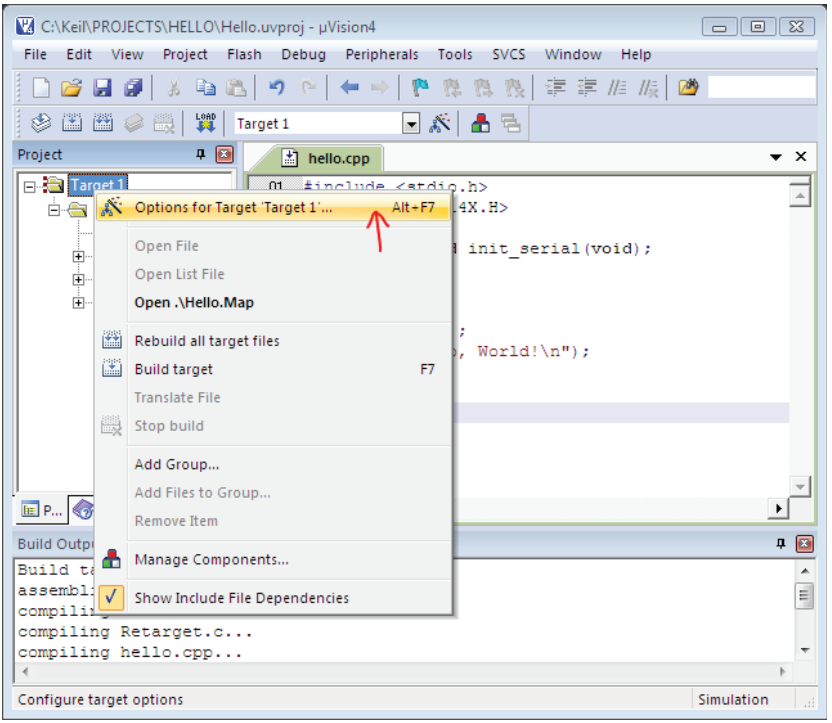

Рис. 2.14.

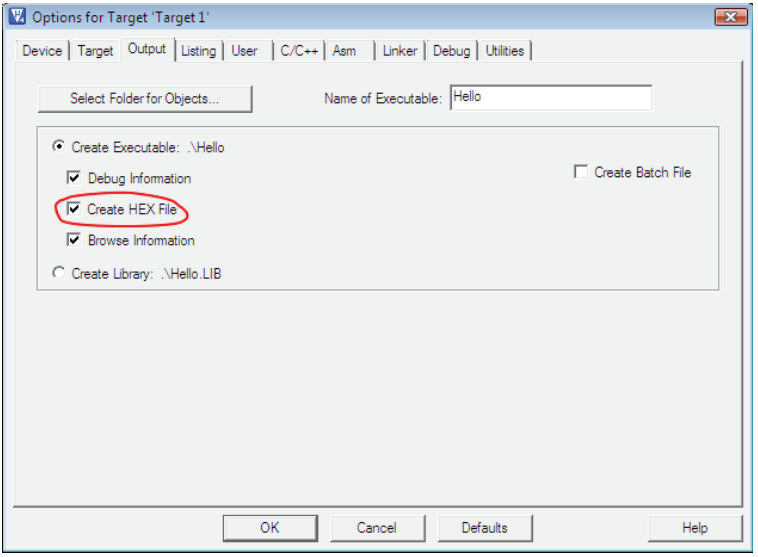

Рис. 2.15.

отладчика для нашего проекта. Чтобы это сделать, вначале выберем **Options for Target ‘Target1’…** в меню **Project**. Далее, в раскрывшемся окне следует выбрать закладку **Debug**, затем отметить опции **Use simulator** и **Run to main()** (Рис. 2.16):

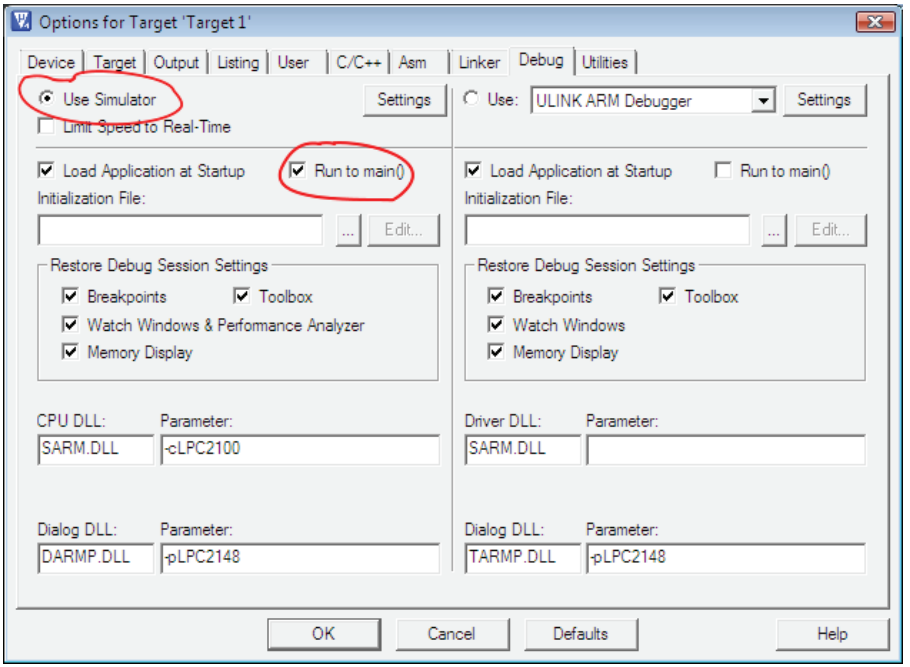

Рис. 2.16.

После установки опций симулятора наше приложение будет работать с виртуальным микроконтроллером, а ввод-вывод будет перенаправлен в терминальное окно ввода-вывода (мы об этом поговорим чуть позже). Для отладки программы на плате микроконтроллера (если таковая имеется) вместо симулятора нужно выбрать опцию **Use <тип отладчика>** в правой части окна (см. Рис. 2.16).

После установки опций симулятора мы можем запустить наше приложение на выполнение, для чего нужно выполнить последовательность шагов, описанную далее. В меню **Debug** нужно выбрать опцию **Start/Stop Debug Session** (Рис. 2.17).

После запуска отладочной сессии должно появиться много окон (Рис. 2.18), в которых будут отображаться состояния переменных, регистров и периферийных устройств. Кроме того, будет присутствовать и окно, в котором будет отображаться машинный код программы в мнемонике языка ассемблера. Нас пока не будут интересовать детали отладки, все что нам сейчас нужно – вывести фразу "Hello, World" на экран и тем самым убедиться, что программа работает нормально. На панели инструментов нужно найти пиктограмму **Serial Windows**

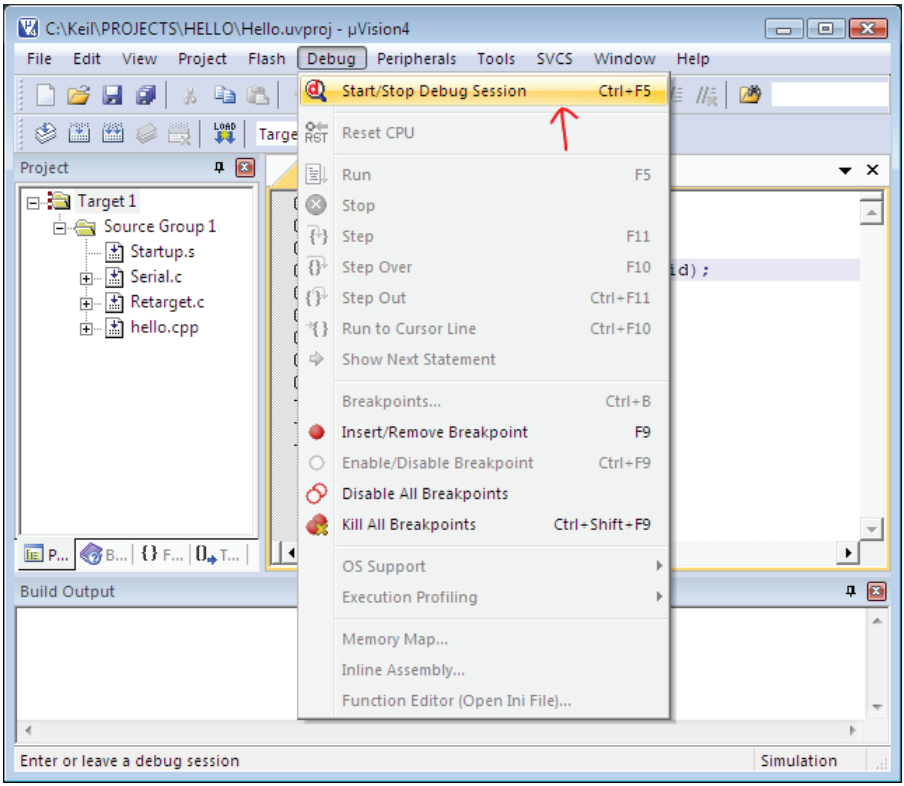

Рис. 2.17.

(показана стрелкой на Рис. 2.18), в которой нужно выбрать виртуальный последовательный интерфейс для вывода данных.

Из предложенного списка выберем интерфейс **UART2 #2** (Рис. 2.19).

После этого в нижней части окна приложения Keil появится окно вывода, обозначенное как **UART 2** –.именно сюда будут выводится выходные данные. Теперь нужно запустить наше приложение на выполнение, нажав пиктограмму **Run** (показана стрелкой на Рис. 2.20).

Если при отладке приложения не указаны точки останова, то программа будет выполняться последовательно, пока не будет выполнена последняя команда. После выполнения программы в симуляторе мы увидим строку "Hello, world!" в окне **UART #2** (Рис. 2.21).

Таким образом, мы протемтировали наше первое C/C++ приложение для ARM микроконтроллера. Для выхода из режима отладки в меню **Debug** нужно снять выделение с опции **Start/Stop Debug Session**. То же самое можно сделать, нажав соответствующую пиктограмку на панели инструментов.

Теперь рассмотрим более детально смысл проведенных манипуляций. Вначале поговорим о вводе-выводе и о том, как он реализован для отладочных

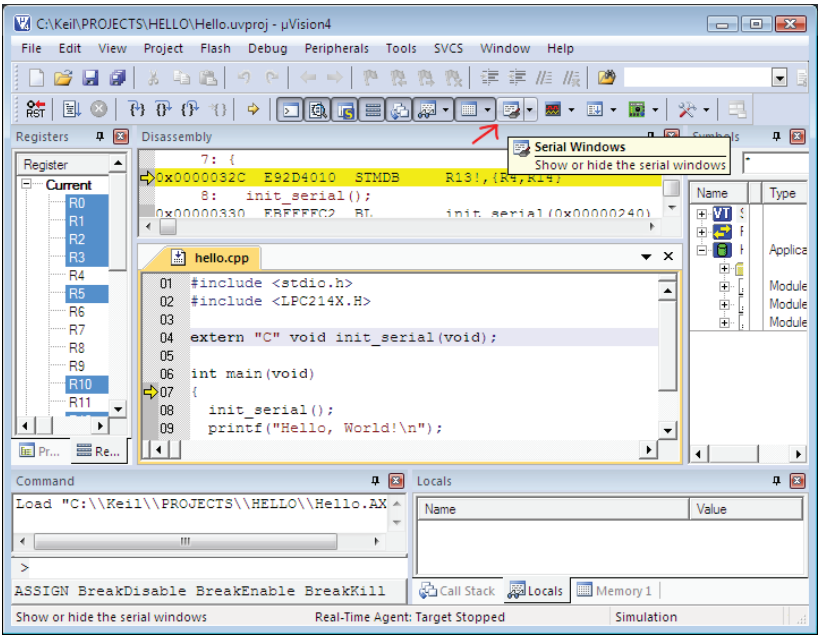

Рис. 2.18.

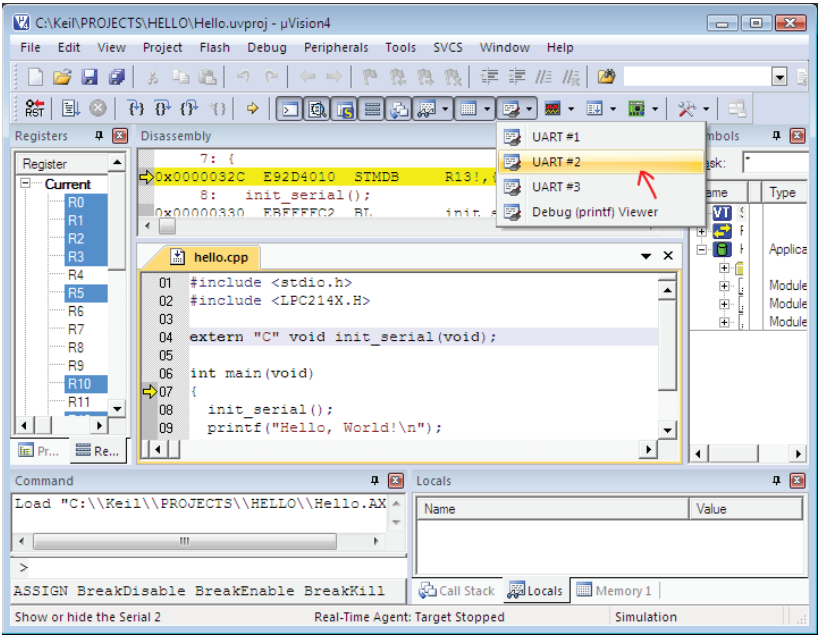

Рис. 2.19.

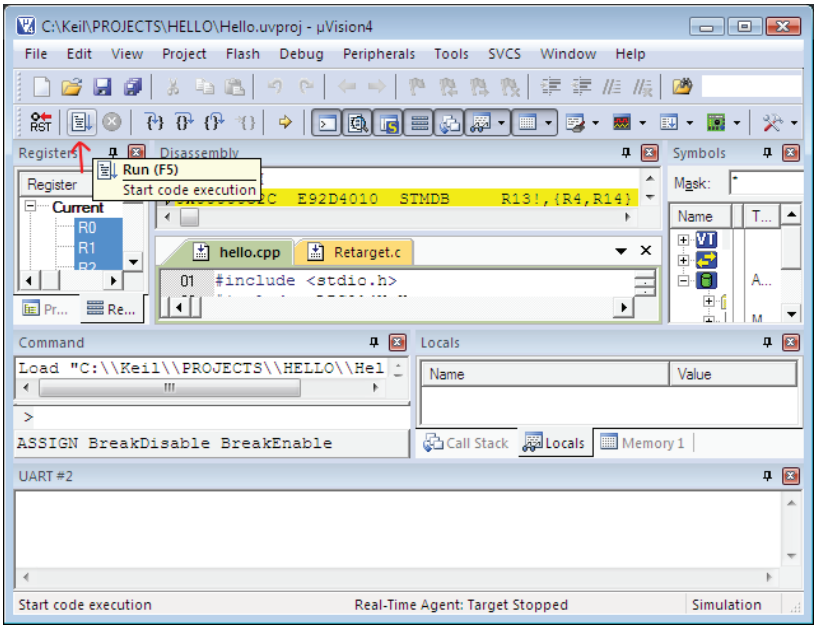

Рис. 2.20.

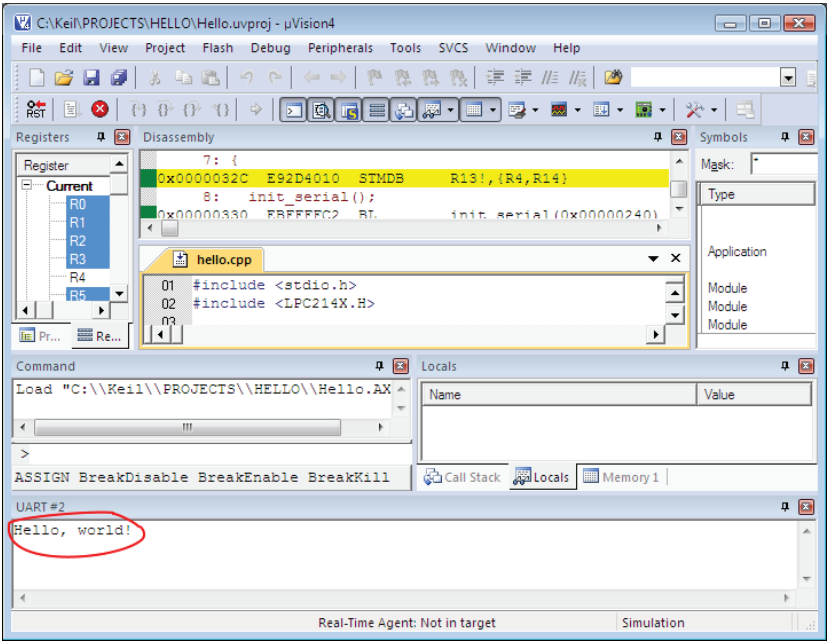

Рис. 2.21.

плат с микроконтроллерами. Для этого вернемся к исходному тексту программы ″Hello, World!″ (Листинг 2.3) и проанализируем его более детально. Для удобства читателей этот листинг приведен ниже.

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

int  main(void)
{
  init_serial();
  printf("  Hello,  world!\n");
  return  0;
  }
```

Первый вопрос, который возникает при детальном анализе этого листинга, связан с функцией **printf()**. Если бы приложение разрабатывалось, скажем, для ПК, то функция **printf()** направляла бы вывод на стандартное устройство вывода по умолчанию, т. е. на экран дисплея. В случае с платой микроконтроллера дело обстоит иначе. Обычно микроконотроллер посылает данные в последовательный интерфейс, который во многих случаях связывает его с системой более высокого уровня, например, с ПК. В этом случае функция **printf()** должна быть адаптирована именно для отправки данных через последовательный порт.

В Keil C функция **printf()** использует вспомогательные функции нижнего уровня для отправки данных в последовательный порт и/или приема данных из последовательного порта. Эти функции определены в файле Serial.c и называются **init_serial**, **sendchar** и **getkey**. Функция **init_serial** устанавливает режим работы последовательного порта и определяет формат получаемых/принимаемых данных. Функция **sendchar** выполняет передачу отдельного байта данных – эта функция имеет классический аналог в C/C++ известный как **putchar**.

Другая функция, **getkey**, выполняет чтение одного байта данных из последовательного порта. Ее аналогами в классическом C/C++ являются функции **getch** и **getchar**. Разработчик ARM систем может, в зависимости от аппарааратной конфигурации системы, переназначить ввод-вывод на другие устройства. Если, например, в системе имеется интерфейс жидкокристаллического индикатора (LCD интерфейс), то, в принципе, возможно написать собственную версию функции **sendchar**, обеспечив тем самым вывод данных на это устройство с помощью **printf()**. По аналогии, ввод данных можно выполнить с сенсорной клавиатуры, написав для нее соответствующую функцию **getkey**.

Вспомним, что для создания законченного приложения (**ELF/HEX** файла) в среде Keil мы можем воспользоваться одной из опций **Build target** или **Rebuild all target files**. Опция **Build** выполняет трансляцию только вновь созданных или

модифицированных файлов с исходными текстами, в то время как при выборе опции **Rebuild** трансляция выполняется для всех файлов проекта независимо от того, были ли внесены в них изменения или нет. Мастер проектов среды μVision отслеживает все ссылки на файлы проекта, поэтому **Build** выполняет все операции корректно.

Создание исполняемого образа программы в Keil C требует, как и в случае классических C++ приложений для обычных ПК, выполнения двух последовательных этапов: компиляции исходных текстов программ и компоновки полученных в результате компиляции объектных модулей в один исполняемый файл. В общем случае для получения исполняемого файла программы требуются, как минимум, две утилиты: компилятор C/C++ (**armcc.exe**) и компоновщик или, по другому, линкер (**armlink.exe**). Компилятор **armcc** обрабатывает файлы с расширениями .c и .cpp, создавая при этом файлы объектных модулей с расширениями .o. Если в состав проекта включены файлы с исходными текстами на языке ассемблера (расширение .s), то для обработки таких файлов вызывается утилита ассемблер **armasm**, в результате чего также генерируются объектные модули.

Часто файлы объектных модулей называют просто объектными файлами. Объектный файл – это файл, содержащий машинные коды инструкций, представленные в исходных текстах. Хотя объектный файл и является двоичным образом, его нельзя запустить на выполнение, поскольку в нем отстутствует информация о том, каким образом исполняемый файл должен загружаться в память при выполнении, а также отсутствует информация о расположении внешних процедур, на которые может ссылаться приложение.

Связывание различных объектных модулей в один исполняемый файд осуществляется на втором этапе программой **armlink**. На этом этапе определяется расположение различных сегментов программного кода и данных в адресном пространстве, а также разрешаются ссылки на внешние по отношению к данному модулю процедуры. В процессе сборки к приложению подключаются и функции, которые могут располагаться в так называемых файлах библиотек или, по другому, в библиотечных модулях (файлы с расширением .lib). Кроме стандартных библиотек к проекту могут подключаться и библиотеки пользователя, в которые разработчик может поместить какие-либо собственные процедуры. В библиотеки очень удобно помещать часто используемые в нескольких проектах процедуры.

Мы продолжим знакомство с особенностями программирования ARM в последующих главах, в которой на практических примерах мы увидим разработку многомодульных приложений, а также принципы взаимодействия отдельных объектных модулей в целостном приложении.

ГЛАВА 3

ПРОГРАММИРОВАНИЕ ПЕРИФЕРИЙНЫХ УСТРОЙСТВ МИКРОКОНТРОЛЛЕРОВ ARM НА KEIL C

Материал этой главы посвящен основам программирования периферийных устройств микроконтроллеров ARM на языке C/C++ в среде Keil. Здесь будут рассмотрены наиболее важные практические аспекты функционирования и программирования портов ввода-вывода, таймеров, устройств цифровой обработки аналоговых сигналов (аналого-цифровых преобразователей) и устройств синтеза аналоговых сигналов (цифро-аналоговых преобразователей), которые являются встроенными устройствами микроконтроллеров ARM. Кроме того, обзор методов программирования таких устройств в среде Keil позволит легко освоить любые другие инструментальные средства разработки ARM-приложений. Многие аспекты программирования являются типичными не только для Keil, но и для остальных популярных средств разработки.

Компилятор Keil C для ARM может выполняться в одном из трех режимов: ISO C90, ISO C99 или ISO C++, каждый из которых можно использовать в зависимости от реализации исходных текстов на C и/или C++. По умолчанию, компилятор Keil выполняется в режиме C90 с опцией **`--c90`**. Для работы в режиме C99 нужно использовать опцию **`--c99`**. Режим C++ будет задействован при указании опции **`--cpp`**. Компилятор Keil C также поддерживает многочисленные расширения языков C и C++. Кроме того, в компиляторе Keil поддерживаются библиотеки стандартных функций C. Keil C также включает библиотеку функций с плавающей точкой **`fplib`**. Она используется с процессорами, у которых отсутствует сопроцессор для работы с плавающей точкой. Особенностью этой библиотеки является то, что все функции оперируют с параметрами с плавающей точкой, используя стандартные целочисленные регистры процессора.

Поскольку подавляющее большинство приложений, написанных для ARM, так или иначе взаимодействует с аппаратными расширениями и периферийными устройствами микроконтроллеров, то в Keil C предусмотрены специальные возможности для решения подобных задач. Так, для записи/чтения регистров периферийных устройств используются обычные операторы присваивания, при этом для чтения регистра устройства нужно поместить его символическое имя или адрес справа от оператора присваивания. Для записи в регистр устройства

символическое имя регистра должно находиться слева от оператора присваивания.

Кроме того, символическое имя или адрес регистра устройства, используемое в операторах, может дополняться операторами унарных и поразрядных логических операций или входить в состав выражений, используемых в управляющих и логических структурах C/C++, таких, например, как операторы **`if`**, **`for`**, **`while`**, **`do`**...**`while`**, **`switch`**...**`case`**.

Для многих типов процессоров доступ к периферийным устройствам и доступ к памяти отличаются в силу различной аппаратной реализации внутренних шин микропроцессора, поэтому адресные пространства устройств ввода-вывода и памяти также отличаются. Кроме этого, и интерфейсы памяти и ввода-вывода могут отличаться, что может потребовать выполнения различных инструкций для доступа к памяти и устройствам ввода-вывода.

Что касается микроконтроллеров ARM, то обращение ко всем периферийным устройствам осуществляется посредством адресов памяти, что позволяет использовать одни и те же инструкции как при обращении к памяти данных/программы, так и при работе с периферией. Для программирования периферийных устройств конкретного микроконтроллера используются адреса и соответствующие им символические имена, определенные в файле заголовка устройства (device header file). Поскольку все примеры данной книги были разработаны и протестированы на плате микроконтроллера LPC2148, то в исходный текст обязательно должен быть включен файл заголовка LPC214X.H.

Кроме того, все приложения должны содержать исходный текст на языке ассемблера файла инициализации микроконтроллера. Этот файл называется Startup.s и генерируется автоматически при разработке нового проекта в среде Keil. В зависимости от специфических требований приложения, в данный файл можно вносить те или иные изменения, однако в наших примерах это не понадобится. Единственное, что нужно будет сделать, это подтвердить включение файла Startup.s в проект на соответствующем этапе создания проекта мастером проектов.

Для иллюстрации работы с периферийными устройствами разработаем несколько демонстрационных проектов, в которых будут задействованы общие для всех микроконтроллеров ARM устройства: цифровые порты ввода-вывода, таймеры-счетчики, цифро-аналоговые и аналого-цифровые преобразователи. Кроме того, в последующих примерах будет продемонстрирована техника программирования прерываний, что существенно упрощает разработку систем реального времени.

Для проверки работоспособности приложения можно использовать либо плату разработки с установленным микроконтроллером ARM, либо симулятор Keil. Напомню, что все примеры книги расчитаны на выполнение в отладчике/симуляторе среды Keil, поэтому нужно установить соответствующие отметки на закладке **Debug**, как показано на Рис. 2.16 из предыдущей главы.

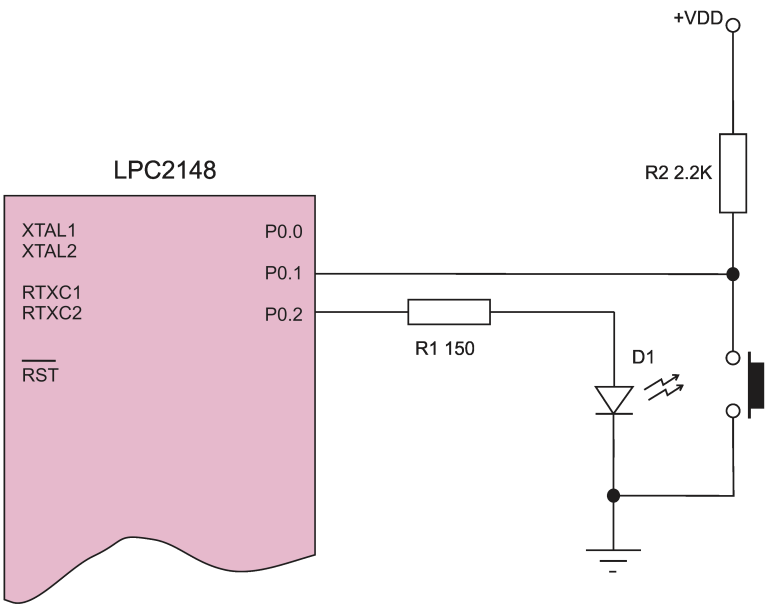

Рис. 3.1.

Пример 1. В этом простейшем примере будет показано, как из программы на C++ можно управлять портами дискретного (цифрового) ввода-вывода. Функциональная схема электрической части нашего проекта показана на Рис. 3.1.

В этой схеме кнопка и резистор R2 моделируют работу дискретного датчика, сигнал от которого поступает на вывод **`P0.1`** порта ввода-вывода **`P0`**. При нажатии кнопки на линию **`P0.1`** подается низкий уровень напряжения (лог. 0), при отпускании кнопки на линии устанавливается напряжение лог. 1. К выводу **`P0.2`** подключен светодиод **`D1`**, который должен переключаться из открытого в закрытое состояние и наоборот всякий раз при нажатии и отпускании кнопки. Для работы нашего приложения необходимо, чтобы вывод **`P0.1`** был сконфигурирован для чтения данных, а вывод **`P0.2`** – для записи.

Программная реализация этого алгоритма показана в исходном тексте программы, написанной на C++, в Листинге 3.1.

Листинг 3.1

```
#include  <stdio.h>
#include  <LPC213X.H>

int  main(void)
{
  unsigned  int  status;
  IO0DIR  =  0x00000004;
```

```
  while  (1)
  {
      while  ((status  =  IO0PIN  &  0x00000002)  !=  0x0);
      while  ((status  =  IO0PIN  &  0x00000002)==  0x0);
      IO0PIN  =  ~(IO0PIN  &  0x00000004);
  }
}
```

В этой программе оператор

```
IO0DIR  =  0x00000004;
```

выполняет конфигурирование вывода **P0.2** для записи, оставляя остальные выводы для чтения. Символическое имя **IO0DIR** в этом операторе обозначает регистр конфигурирования **IODIR** порта **P0**. Данный регистр управляет направлением передачи данных через выводы порта **P0**. Запись единицы в какой-либо разряд этого регистра устанавливает режим работы соответствующего вывода порта на "запись", а запись нуля конфигурирует соответствующий вывод для "чтения".

При сбросе микроконтроллера все выводы порта устанавливаются 0, т. е. в режим для "чтения".

Для чтения-записи дискретного порта ввода-вывода используется регистр **IOPIN**. Для порта **P0** мнемоническое обозначение этого регистра будет **IO0PIN**. Этот порт доступен для чтения-записи, причем можно прочитать данные на выводах порта, независимо от режима их функционирования (чтение или запись).

Вернемся к нашей программе. После конфигурирования порта **P0** в цикле **while** осуществляется непрерывное чтение вывода **P0.1** в переменную **status**. Два идущих подряд оператора

```
while  ((status  =  IO0PIN  &  0x00000002)  !=  0x0);
while  ((status  =  IO0PIN  &  0x00000002)==  0x0);
```

нужны для устранения так называемого "дребезга" входного сигнала и предотвращения ложных срабатываний из-за переходных процессов в переключателе.

Оператор

```
IO0PIN  =  ~(IO0PIN  &  0x00000004);
```

выполняет запись в вывод **P0.2** значения логического уровня, противоположного предыдущему. Если, например, на этом выводе присутствовал низкий уровень напряжения (лог. 0), то будет записан лог. 1, и наоборот, уровень лог. 1 будет изменен на лог. 0.

Пример 2. В этом примере приложение будет выводить сообщение в терминальное окно программы каждые 3 секунды, для чего будет задействован один из таймеров микроконтроллера (таймер 1). Перед разработкой приложения с использованием таймеров кратко ознакомимся с аппаратно-программной архитектурой этих устройств. Следует отметить один важный момент: базовые архитектуры периферийных устройств в микроконтроллерах ARM, независимо от модификаций процессоров, имеют большое сходство.

Основные трудности при программировании периферийных устройств возникают, как правило, при конфигурировании различных режимов работы и параметров, что требует от разработчика понимания принципов работы данного устройства. Упрощенную функциональную схему одного канала отдельного таймера можно представить так, как показано на Рис. 3.2:

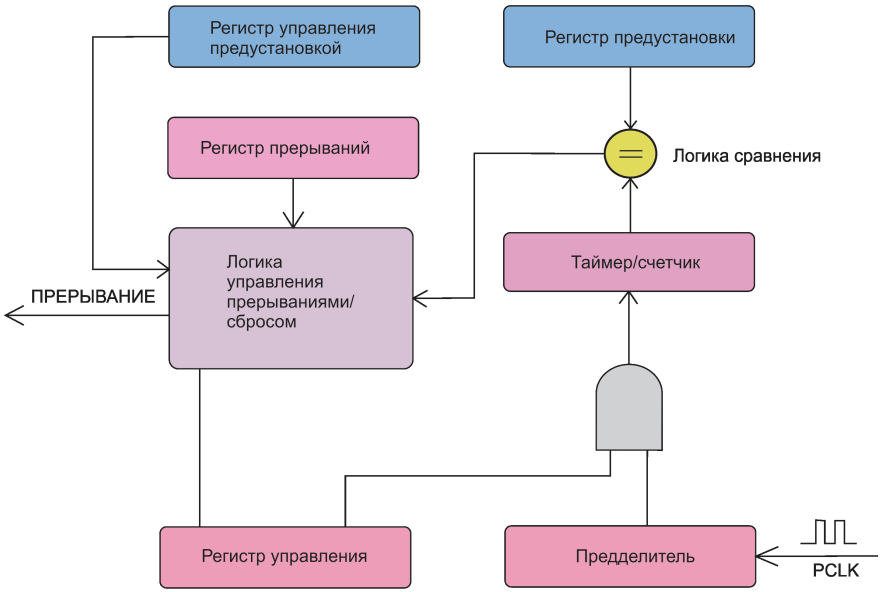

Рис. 3.2.

Эта функциональная схема с небольшими изменениями и дополнениями реализована практически во всех типах микроконтроллеров, не только ARM. В этой схеме сигнал тактовой частоты **PCLK** с выхода синтезатора поступает на схему предделителя, который делит частоту поступающего сигнала в N раз, где N может принимать целые значения от 0 до 2^{32} (для 32-разрядного предделителя). При N = 0 предделитель пропускает сигнал на вход таймера/счетчика без предварительного деления, при N = 1 частота входного сигнала синхронизации делится на 2, при N = 2 осуществляется деление на 3 и т. д. Коэффициент деления N задается программно (в ARM микроконтроллерах значение N записывается в регистр предделителя).

В логике предделителя также имеется счетчик предделителя, который инкрементируется на 1 при поступлении очередного тактового импульса **PCLK**. Когда содержимое счетчика предделителя достигнет значения, записанного в регистре предделителя, то содержимое таймера счетчика инкрементируется на 1, а счетчик предделителя сбрасывается в 0, и цикл начинается сначала. Если частоту сигнала **PCLK** на входе предделителя обозначить как F, частоту сигнала на выходе предделителя как F_N, то получим следующее соотношение:

$$F_N = F/N,$$

где N – коэффициент деления. С частотой F_N будет инкрементироваться таймер/счетчик. В большинстве ARM микроконтроллеров используются 32-разрядные таймеры, поэтому максимальное число тиков таких таймеров достигает 2^{32}. По достижению этого значения следующий инкремент таймера приводит к сбросу содержимого таймера/счетчика в 0. Для 16-разрядных таймеров максимальное число тиков будет равно 2^{16}.

Функциональная схема таймера/счетчика также включает логику предустановки, основными блоками которой являются регистр предустановки и регистр управления предустановкой. В регистр предустановки загружается значение, до которого таймер должен считать. При достижении значения таймера, равного величине, записанной в регистре предустановки, программа может выполнить одно или несколько действий, перечисленных ниже:

1. Сгенерировать сигнал прерывания.
   В этом случае управление будет передано соответствующему данному прерыванию (если таковое разрешено) обработчику;
2. Сбросить содержимое таймера/счетчика в 0;
3. Остановить таймер.

Для выбора возможных вариантов продолжения нужно установить соответствующий бит в регистре управления предустановкой. Регистр управления таймером выполняет две функции: запуск/остановку таймера и сброс содержимого счетчиков таймера (для этого обычно используются первые два бита регистра).

Проанализируем исходный текст программы **main** (файл main.cpp), который показан в Листинге 3.2.

Листинг 3.2

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

unsigned  long  del  =  0;

/*  Timer  1  Interrupt  executes  each  10ms  at  15  MHz  Clock  */
__irq  void  tc1  (void)  {
    T1IR  =  1;
    VICVectAddr  =  0;
    if  (del  <  100)
          del++;
    else
          del  =  0;
}

/*  Setup  the  Timer  Counter  1  Interrupt  */
void  init_timer  (void)  {
    T1MR0  =  150000;
```

```
    T1PR  =  2;
    T1MCR  =  3;
    T1TCR  =  1;
    VICVectAddr0  =  (unsigned  long)tc1;
    VICVectCntl0  =  0x20  |  5;
    VICIntEnable  =  0x00000020;
}
void  wait  (void)
{
    while  (del  <  100);
}

int  main(void)
{
  int  cnt  =  0;
  init_serial();
  init_timer();
  while  (1)
  {
      if  (cnt  ==  3)
        {
          T1TCR  =  0;
          printf("  timer  1  has  been  stopped.\n");
          break;
      }
      printf("  timer  1  interrupt  burst  %d  times.\n",  ++cnt);
      wait();
    }
  while(1);
  }
```

Детально проанализируем исходный текст. Это приложение выводит в терминальное окно сообщения через каждые три секунды — это интервал, через который инициируется прерывание таймера 1. Для конфигурирования прерывания таймера 1 выполняется следующая последовательность действий:

1. В регистр предустановки таймера 1 **T1MR0** записывается значение, по достижению которого таймер перегружается, и генерируется сигнал прерывания. Для нашей программы выберем значение предустановки, равное 150 000, что даст в результате значение задержки, равное 15 г 10^4/ 15 г 10^6 Гц = 10^{-2} с = 10 мс. Следующий оператор записывает величину задержки в регистр **T1MR0**:

   ```
   T1MR0  =  150000;
   ```

   Таким образом, до перезагрузки таймер 1 успеет отсчитать 10 мс (при значении 0 в регистре предделителя).
2. Для дополнительного деления частоты входного сигнала на 3 конфигурируется регистр предделителя

   ```
   T1PR  =  2;
   ```

   Таким образом, интервал между прерываниями будет равен 30 мс.
3. В регистр управления предустановкой **T1MCR** записывается значение 3 — это выполняет оператор

```
T1MCR  =  3;
```

Как только таймер достигнет значения, указанного в регистре предустановки, он будет сброшен и будет инициировано прерывание.

4. Выполняется запуск таймера 1:

```
T1TCR  =  1;
```

Затем программа должна сконфигурировать сам вектор прерывания. Эта операция требует нескольких шагов. Вначале устанавливается адрес обработчика прерывания оператором

```
VICVectAddr0  =  (unsigned  long)tc1;
```

Затем прерыванию назначается конкретный слот в контроллере прерываний с помощью оператора

```
VICVectCntl0  =  0x20  |  5;
```

Наконец, необходимо разрешить работу прерывания:

```
VICIntEnable  =  0x00000020;
```

Сам обработчик прерывания реализован в функции **tc1**, предваряемой квалификатором **__irq**. Функция инкрементирует переменную **del** и проверяет ее равенство значению 100. Как только это значение достигнуто, значение **del** обнуляется. Переменная **del**, в свою очередь, проверяется функцией **wait**, которая выполняет задержку в цикле **while**, где выводятся сообщения.

Обработчик прерывания должен сам сбросить флаг прерывания и сообщить о завершении выполнения обработчика. Это выполняют следующие операторы:

```
T1IR  =  1;
VICVectAddr  =  0;
```

В основной программе после выполнения трех прерываний (цикл **while**) таймер 1 останавливается с помощью оператора

```
T1TCR  =  0;
```

Пример 3. Во этом примере будет разработан простой генератор прямоугольных импульсов, период которого задается интервальным таймером, а выходной сигнал снимается с вывода **P0.2** (бит 2 порта 0) микроконтроллера. Функциональная схема аппаратной части проекта показана на Рис. 3.3.

Исходный текст программы **main** показан в Листинге 3.3.

Листинг 3.3

```
#include  <stdio.h>
#include  <LPC214X.H>

long  volatile  timeval  =  0;

/*  Timer  Counter  1  Interrupt  executes  each  10ms  at  15  MHz  CPU  Clock  */
__irq  void  tc1  (void)  {
    timeval++;
```

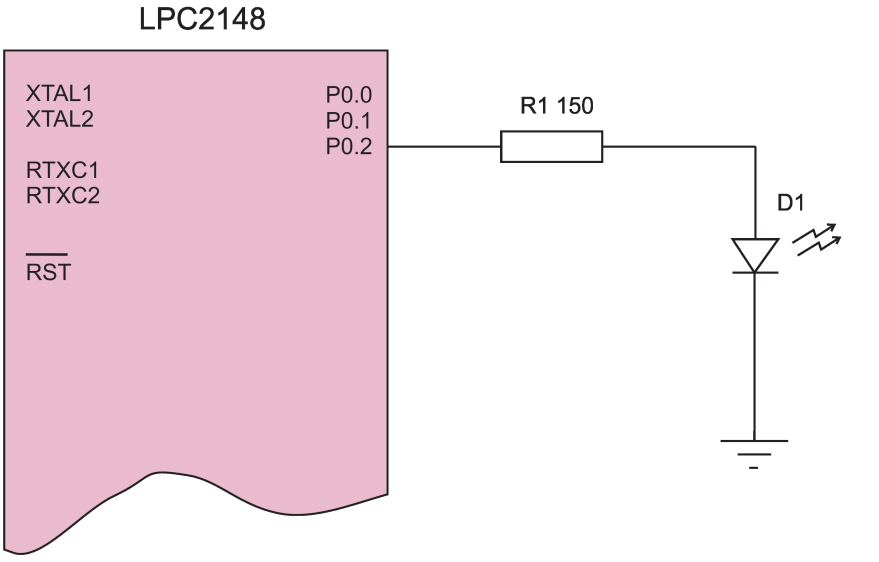

Рис. 3.3.

```
    T1IR  =  1;
    VICVectAddr  =  0;
    if  (timeval  >  100)
    {
      timeval  =  0;
      IO0PIN  =  ~(IO0PIN  &  0x00000004);
    }
}

/*  Setup  the  Timer  1  Interrupt  */
void  init_timer  (void)  {
    T1MR0  =  150000;
    T1PR  =  2;
    T1MCR  =  3;
    T1TCR  =  1;
    VICVectAddr0  =  (unsigned  long)tc1;
    VICVectCntl0  =  0x20  |  5;
    VICIntEnable  =  0x00000020;
}

int  main(void)
{
  IO0DIR  =  0x00000004;
  init_timer();
  while  (1);
}
```

Выходной сигнал генерируется на выводе **P0.2** микроконтроллера и имеет форму прямоугольных импульсов. Уровень напряжения на выводе **P0.2** меняется на противоположный всякий раз, как только переменная **timeval**

превысит значение 100. Эта переменная инкрементируется в обработчике прерывания **tc1** таймера 1. Как только ее значение становится равным 100, **timeval** сбрасывается в 0, а состояние бита 2 порта **P0** переключается на противоположное:

```
IO0PIN  =  ~(IO0PIN  &  0x00000004);
```

Прерывание таймера 1 вызывается каждые 30 мс, поскольку значение предделителя установлено в **T1PR** оператором

```
T1PR  =  2;
```

При данном значении частота инкремента таймера уменьшается в 3 раза, т. е. период сигнала увеличивается в три раза.

Оператор

```
IO0DIR  =  0x00000004;
```

конфигурирует бит 2 порта **P0** на вывод данных. Зная период сигнала, легко подсчитать частоту выходного импульсного сигнала – она будет приблизительно равна $1/(3*20)$ мс$^{-1}$ = 16.6 Гц.

Изменения состояния портов ввода-вывода (как и многих других периферийных устройств) можно проконтролировать в отладчике/симуляторе. Для этого нужно запустить приложение на отладку и в меню **Peripherals** выбрать опции **GPIO Slow Interface → Port 0** (Рис. 3.4).

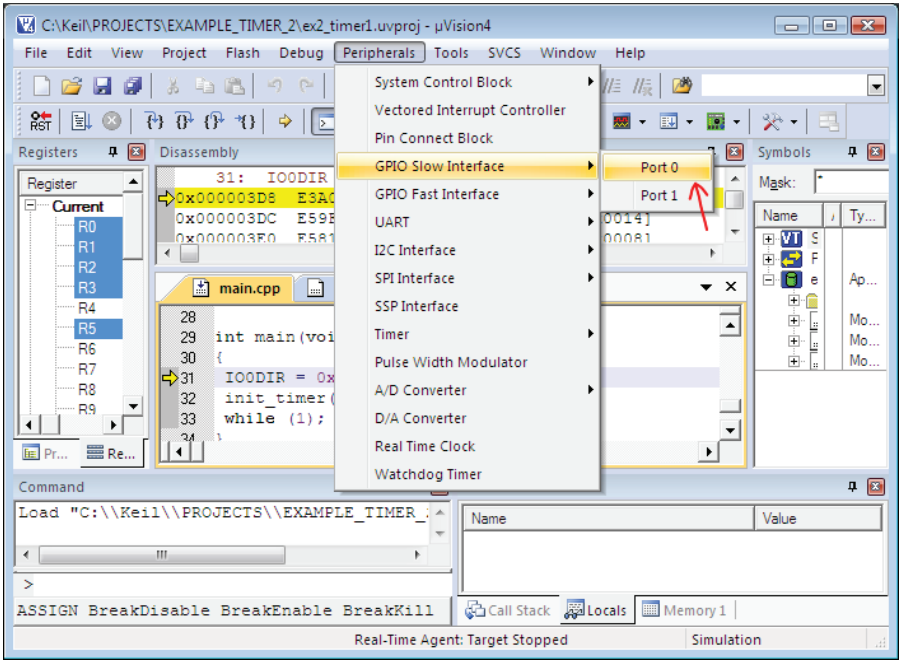

Рис. 3.4.

После нажатия пиктограммы **Run** в левом верхнем углу окна отладчика можно наблюдать изменения состояния бита 2 (показано красной стрелкой на Рис. 3.5) в процессе выполнения программы.

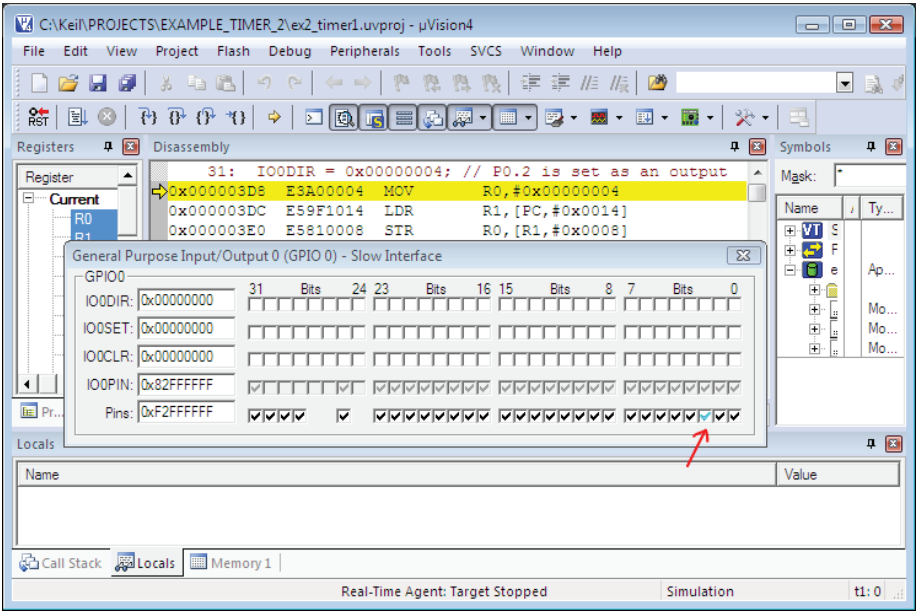

Рис. 3.5.

Большинство приложений реального времени должно обрабатывать внешние сигналы, поступающие на входы микроконтроллера асинхронно с выполнением основной программы. Для обработки цифровых сигналов выделены специальные линии внешних прерываний, к которым можно присоединять внешние источники дискретных сигналов. Так, для микроконтроллера LPC2148 выделено 4 линии **`EINT0`**–**`EINT3`**.

Наш следующий пример показывает программные методы обработки прерывания от дискретного датчика, который может устанавливать выходной сигнал в 1 (высокий уровень напряжения) или в 0 (низкий уровень напряжения).

Пример 4. Будем полагать, что наш дискретный датчик присоединен к линии **`P1.0`** и в активном состоянии генерирует перепад 1 – 0. Это означает, что как только на линии внешнего прерывания будет обнаружен такой перепад, то микроконтроллер генерирует прерывание и передает управление соответствующей процедуре-обработчику прерывания. Обработчик прерывания, в свою очередь, переключит светодиод на выводе **`P0.2`** в противоположное состояние. Функциональная схема для этого проекта такая же, как и в примере 1 (Рис. 3.1). Здесь она приведена еще раз для удобства на Рис. 3.6.

В этой схеме кнопка и резистор R2 моделируют работу дискретного датчика. При нажатии кнопки на линию **`P0.1`** порта **`P0`** подается низкий уровень на-

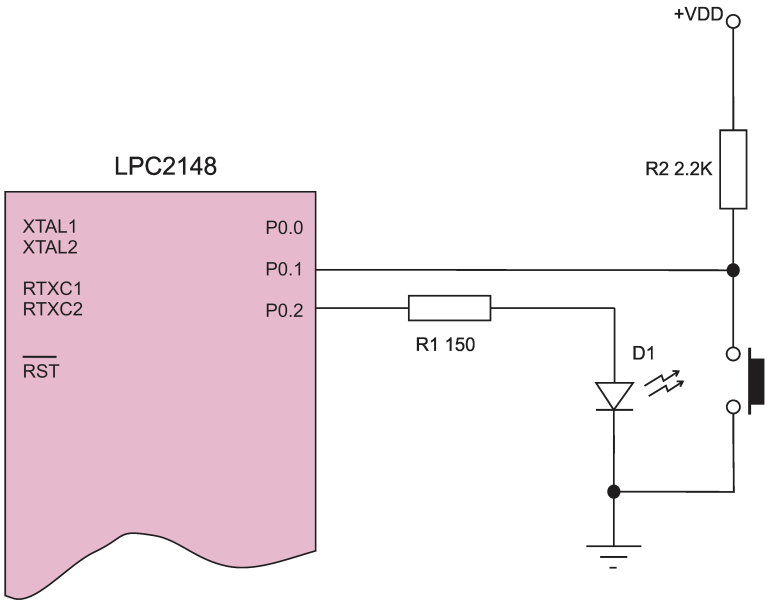

Рис. 3.6.

пряжения (лог. 0), что вынуждает микроконтроллер сгенерировать прерывание **EINT0**. После вызова обработчика прерывания последний считывает состояние регистра–защелки на линии **P0.2**, инвертирует считанное значение и записывает его обратно в защелку **P0.2**. Таким образом на светодиод будет поступать то высокий, то низкий уровень напряжения.

Исходный текст основной программы показан в Листинге 3.4.

Листинг 3.4

```
#include  <stdio.h>
#include  <LPC214X.H>

/*  EINT0  Interrupt  executes  on  P0.1  */
__irq  void  eint0_handler  (void)  {
    EXTINT  =  1;
    VICVectAddr  =  0;
    IO0PIN  =  ~(IO0PIN  &  0x00000004);

  }

/*  Setup  the  EINT0  Interrupt  */
void  init_eint0  (void)  {
    PINSEL0  =  0xC;
    EXTMODE  =  0x1;
    EXTPOLAR  =  0x0;
    VICVectAddr0  =  (unsigned  long)eint0_handler;
    VICVectCntl0  =  0x20  |  14;
```

```
    VICIntEnable  =  1  <<  14;
}

int  main(void)
{
  IO0DIR  =  0x00000004;
  init_eint0();
  while  (1);
}
```

Анализ исходного текста программы начнем с процедуры **init_eint0**, с помощью которой выполняется конфигурирование обработчика внешнего прерывания **EINT0**. Оператор

```
PINSEL0  =  0xC;
```

выполняет привязку линии прерывания к выводу **P0.1** порта **P0**.

Следующие два оператора устанавливают характеристики сигнала прерывания. Оператор

```
EXTMODE  =  0x1;
```

устанавливает режим срабатывания по фронту сигнала. Дополнительно оператор

```
EXTPOLAR  =  0x0;
```

устанавливает чувствительность по перепаду от высокого к низкому уровню сигнала на линии.

Следующие три оператора настраивают контроллер прерываний:

```
VICVectAddr0  =  (unsigned  long)eint0_handler;
VICVectCntl0  =  0x20  |  14;
VICIntEnable  =  1  <<  14;
```

Первый оператор записывает в регистр адреса вектора прерывания адрес процедуры-обработчика прерывания (**eint0_handler**), второй назначает слот прерывания (0, в данном случае) конкретному вектору прерывания. Наконец, последний оператор уставливает бит разрешения данного прерывания в регистре разрешения прерываний.

Обработчик прерывания **eint0_handler** содержит всего три оператора. Оператор

```
EXTINT  =  1;
```

очищает флаг прерывания, чтобы избежать повторного вызова прерывания. Оператор

```
VICVectAddr  =  0;
```

выполняет подтверждение обработки прерывания. Наконец, оператор

```
IO0PIN  =  ~(IO0PIN  &  0x00000004);
```

переключает логический уровень на выводе **P0.2** на противоположное значение. В основной программе направление передачи данных через порт **P0**

конфигурируется с помощью оператора

```
IO0DIR  =  0x00000004;
```

Пример 5. В этом примере будет показано, как сформировать выходной аналоговый сигнал с помощью цифро-аналогового преобразователя микроконтроллера ARM. В данном случае наша программа сформирует непрерывный сигнал треугольной формы, который через буфер подается во внешние цепи. Функциональная схема аппаратной части проекта показана на Рис. 3.7.

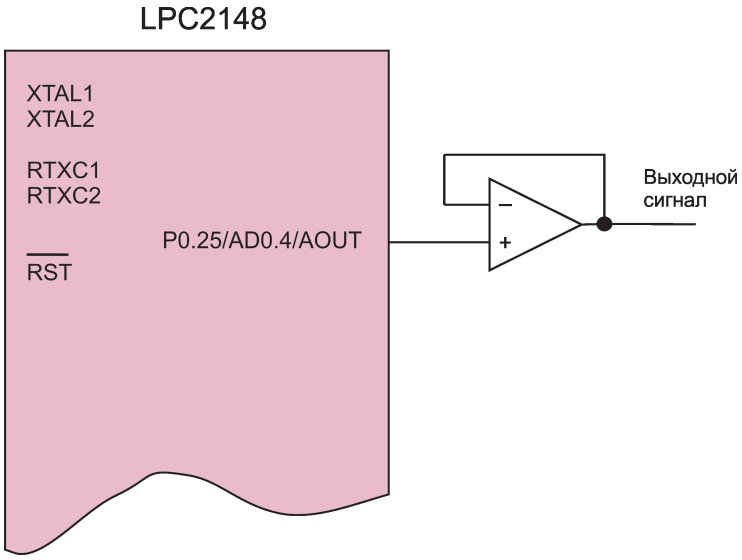

Рис. 3.7.

В микроконтроллере LPC2148 имеется 10-разрядный цифро-аналоговый преобразователь (ЦАП). Напряжение выходного аналогового сигнала ЦАП проходит на вывод **AOUT** микроконтроллера и далее во внешние цепи. Перед подачей сигнала во внешние схемы выход **AOUT** следует буферизовать, для чего можно воспользоваться повторителем напряжения на операционном усилителе (желательно выбрать операционный усилитель с высоким входным сопротивлением и высокой скоростью нарастания выходного напряжения, например, CA3130/CA3140 и т. п.).

Исходный текст основной программы показан в Листинге 3.5.

Листинг 3.5

```
#include  <stdio.h>
#include  <LPC214X.H>

#define  VREF  3.3

extern  "C"  void  init_serial(void);
```

```
void  delay(void)
{
    long  del  =  10;
    while(del--  !=  0);
}

int  main(void)
{
  int  Dout;
  float  Vout;
  int  i1  =  0;
  PINSEL1  =  2  <<  18;  //  DAC  output  is  enabled

  while  (1)
  {
  for  (i1  =  0;  i1  <  500;  i1++)
    {
      Vout  =  0.5  *  VREF  *  (double)i1/500;
      Dout  =  (int)(Vout  *  1024  /  VREF);
      DACR  =  Dout  <<  6;
      delay();
    }
    for  (i1  =  500;  i1  >  0;  i1--)
    {
      Vout  =  0.5  *  VREF  *  (double)i1/500;
      Dout  =  (int)(Vout  *  1024  /  VREF);
      DACR  =  Dout  <<  6;
      delay();
    }
    }
  }
```

В этой программе константа **VREF** определяет значение напряжения смещения (по-другому, образцового напряжения) для схемы цифро-аналогового преобразователя. Для программирования нужного выходного напряжения необходимо знать двоичный код, соответствующий данному напряжению.

Если обозначить V_{REF} – напряжение смещения, V_{OUT} – выходное напряжение и D_{OUT} – двоичный код, ссответствующий заданному выходному напряжению, то получим следующую формулу:

$$V_{OUT} = (V_{REF}/1024) \text{ г } D_{OUT}$$

Из этой формулы легко вычислить D_{OUT} по известным значениям V_{REF} и V_{OUT}. После этого потребуется единственный оператор для записи начения D_{OUT} в регистр **DACR** цифро-аналогового преобразователя:

```
DACR  =  Dout  <<  6;
```

Поскольку выход **AOUT** цифро-аналогового преобразователя мультиплексируется с другими функциями, например, с цифровым выходом, то функцию цифро-аналогового преобразования для данного вывода нужно назначить с помощью записи соотвтетсвующего значения в регистр **PINSEL1**:

```
PINSEL1  =  2  <<  18;
```

Два цикла **for** позволяют сформировать симметричный треугольный сигнал на выходе ЦАП. Для формирования, например, пилообразного напряжения на выводе **AOUT** можно убрать из исходного текста второй цикл **for**.

Для проверки работоспособности приложения можно запустить его на выполнение в симуляторе Keil. Процесс обработки данных цифро-аналоговым преобразователем в симуляторе можно увидеть, если в меню **Peripherals** выбрать опцию **D/A Converter** (показано красной стрелкой) (Рис. 3.8):

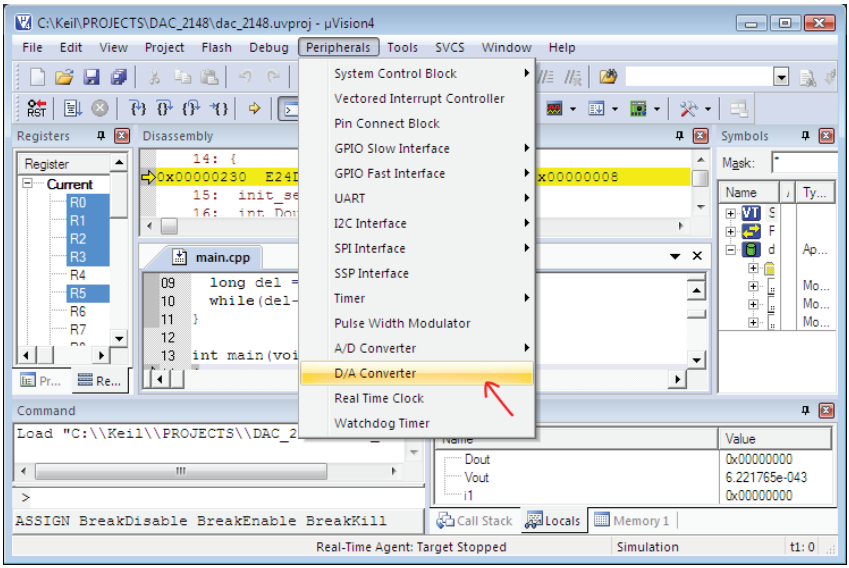

Рис. 3.8.

После нажатия пиктограммы **Run** в левом верхнем углу окна отладчика/симулятора можно наблюдать, как изменяются значения переменных в окне **D/A Converter** (Рис. 3.9).

Поскольку наше приложение генерирует сигнал треугольной формы на выходе **AOUT**, который изменяется с определенной частотой, то в окнах **DACR** и **Analog Output** будут видны быстро меняющиеся значения.

Пример 6. В этом примере будет показано, как выполнять обработку аналогового входного сигнала в микроконтроллере ARM LPC2148. Функционирование встроенных аналого-цифровых преобразователей (АЦП) в большинстве микроконтроллеров, не только ARM, подчиняется практически одним и тем же алгоритмам. Аппаратная часть встроенных АЦП включает узлы, во многом совместимые по своим функциям, поэтому изучив принцип работы одного такого преобразователя, очень легко можно понять работу аналогичного устройства, имеющегося в других микроконтроллерах.

Программный алгоритм функционирования отдельного канала аналого-цифрового преобразователя в упрощенном виде можно представить диаграммой, показанной на Рис. 3.10.

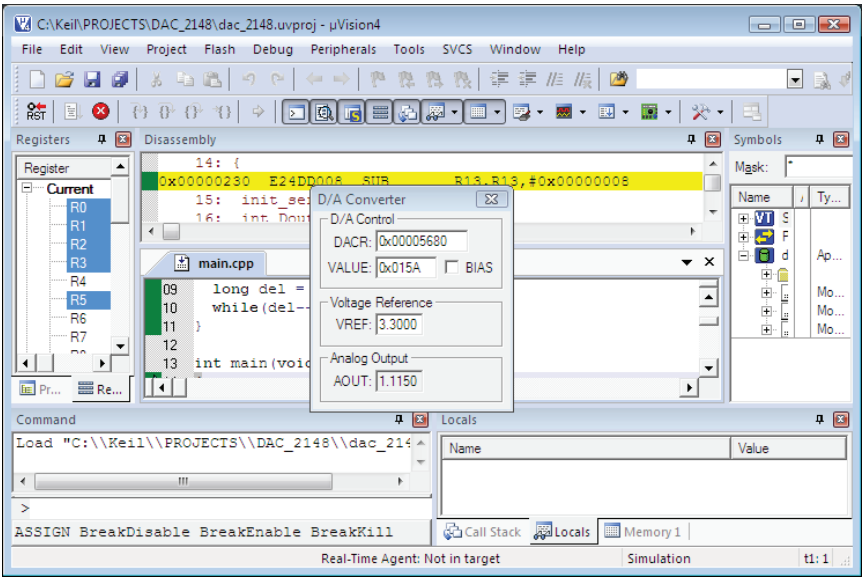

Рис. 3.9.

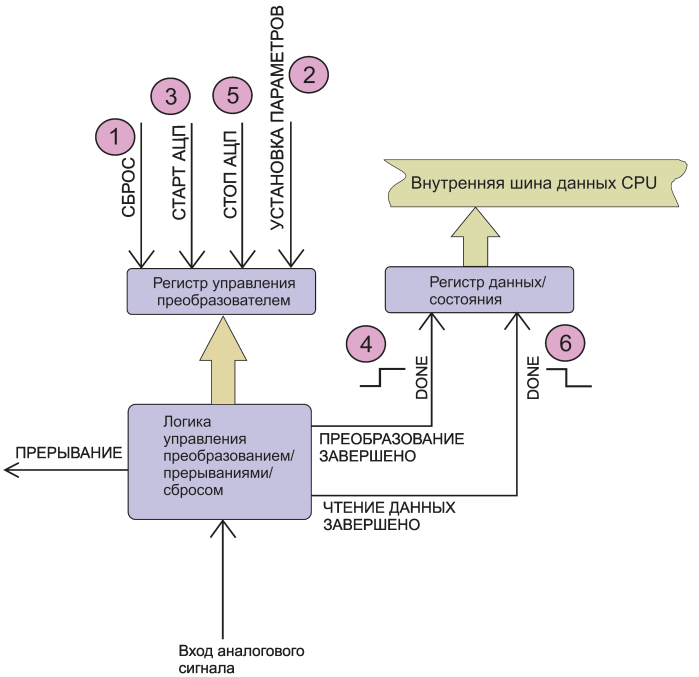

Рис. 3.10.

Перед началом преобразования желательно выполнить начальную инициализацию (сброс) АЦП (1). На этом этапе также сбрасывается флаг завершения преобразования (DONE) в регистре состояния и/или данных. Затем необходимо установить параметры преобразования, для чего выполним установку соответствующих битов в регистре управления преобразователем (2). На этом этапе, если необходимо, выполняется настройка прерывания для АЦП, хотя во многих случаях обработка аналогового сигнала выполняется обычным способом.

Для начала выполнения аналого-цифрового преобразования необходимо установить соответствующий бит в регистре управления (3) и ожидать завершения преобразования. О завершении процедуры преобразования будет свидетельствовать установленный флаг DONE в регистре состояния/данных (4). Если DONE установлен, то работа преобразователя приостанавливается (5), а флаг DONE сбрасывается (6). Двоичный код, соответствующий аналоговому входному сигналу, может быть прочитан программой.

Для тестирования работы преобразователя можно использовать простейшую функциональную схему аппаратной части, показанную на Рис. 3.11.

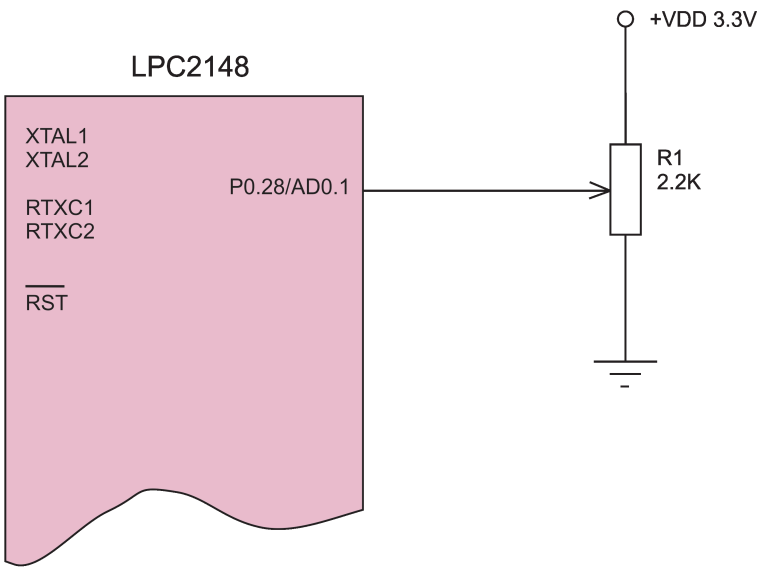

Рис. 3.11.

Здесь аналоговый входной сигнал с резистивного делителя на потенциометре **`R1`** поступает на вход канала 1 (**`AD0.1`**) аналого-цифрового преобразователя микроконтроллера LPC2148. Кроме того, вывод V_{REF} микроконтроллера должен быть подключен к источнику питания с напряжением 3 – 3.3 В (в нашем проекте V_{REF} имеет значение 3.3 В).

Исходный текст программы для измерения и визуализации входного аналогового сигнала показан в Листинге 3.6.

Листинг 3.6

```
#include  <stdio.h>
#include  <LPC214X.H>

#define  VREF  3.3
#define  LSB  VREF/1024.0

extern  "C"  void  init_serial(void);

int  main(void)
{

  unsigned  long  DONE  =  0;
  unsigned  long  res;

  init_serial();
  AD0CR  =  0;
  AD0CR  |=  2;
  AD0CR  |=  3  <<  8;
  AD0CR  |=  1  <<  21;
  AD0CR  |=  1  <<  24;

  while  ((DONE  =  AD0DR1  &  0x80000000)  ==  0x0);
  AD0CR  &=  0  <<  24;
  res  =  (AD0DR1  >>  6)  &  0x3FF;
  double  fres  =  LSB  *  res;
  printf("  ADC0  result:  %5.3f\n",  fres);

  while  (1);
}
```

Перед анализом исходного текста выясним, как двоичный код на выходе АЦП связан с величиной аналогового входного сигнал. Если V_{REF} – это напряжение смещения для АЦП, а V_{IN} – напряжение входного сигнала, то для двоичного кода на выходе преобразователя (обозначим его как D_{OUT}) будет справедливо следующее соотношение:

$$D_{OUT} = V_{OUT}/V_{REF} \text{ г } 1024$$

Число 1024 равно 2^{10} – это разрядность АЦП (в LPC2148 преобразователь имеет разрядность 10).

В нашей программе константа **VREF** численно равна величине напряжения смещения (3.3), а константа **LSB** равна величине младшего значащего разряда преобразователя. Переменная **DONE** будет содержать значение флага завершения преобразования (0 или 1), а переменная **res** – двоичный код на выходе преобразователя.

Оператор

```
AD0CR  =  0;
```

выполняет сброс регистра управления АЦП (**AD0CR**), при этом все биты регистра устанавливаются в значения по умолчанию. Следующий оператор

```
AD0CR  |=  2;
```

выбирает в качестве источника входного аналогового сигнала канал **AD0.1** (всего имеется 8 каналов). Для корректной работы АЦП необходимо правильно выбрать тактовую частоту преобразователя, установив соответствующий коэффициент деления (в данном случае, 3) с помощью оператора

```
AD0CR  |=  3  <<  8;
```

Перед началом преобразования нужно также установить бит **PDN** (power-down), который сигнализирует о том, что на АЦП подано напряжение питания и устройство находится в рабочем состоянии. Это действие выполняет оператор

```
AD0CR  |=  1  <<  21;
```

В принципе, это все базовые установки, которые нужны для выполнения преобразования. Следующий оператор выполняет запуск процесса аналого-цифрового преобразования:

```
AD0CR  |=  1  <<  24;
```

Сам процесс преобразования делится (условно) на два этапа: выборку (sampling) и вывод битового потока. Обе операции требуют определенного времени для завершения, поэтому о готовности результата можно судить по значению флага **DONE**, который непрерывно считывается в цикле **while**:

```
while  ((DONE  =  AD0DR1  &  0x80000000)  ==  0x0);
```

Как только значение **DONE** станет равным 1, преобразователь останавливается:

```
AD0CR  &=  0  <<  24;
```

Теперь можно считать двоичный код результата преобразования в переменную **res**:

```
res  =  (AD0DR1  >>  6)  &  0x3FF;
```

Далее двоичный код преобразуется к формату с плавающей точкой (переменная **fres**):

```
double  fres  =  LSB  *  res;
```

Вывод результата в терминальное окно выполняется функцией **printf**.

Для наблюдения выполнения программы в симуляторе Keil нужно запустить режим отладки и в меню **Peripherals** выбрать опцию **A/D Converter** (Рис. 3.12).

Для моделирования работы аналого-цифрового преобразователя следует запустить приложение на выполнение, но не нажимая пока пиктограмму **Run**. В раскрывшемся окне **A/D Converter 0** в поле **AD01** нужно набрать какое-либо значение из диапазона работы АЦП (0 – 3.3 В). В данном примере значение входного сигнала выбрано 2.944 В (Рис. 3.13).

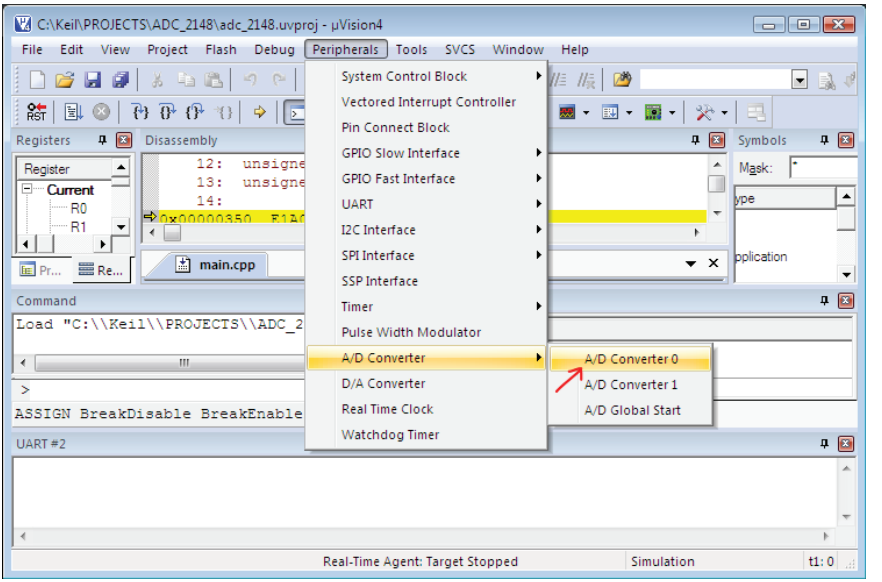

Рис. 3.12.

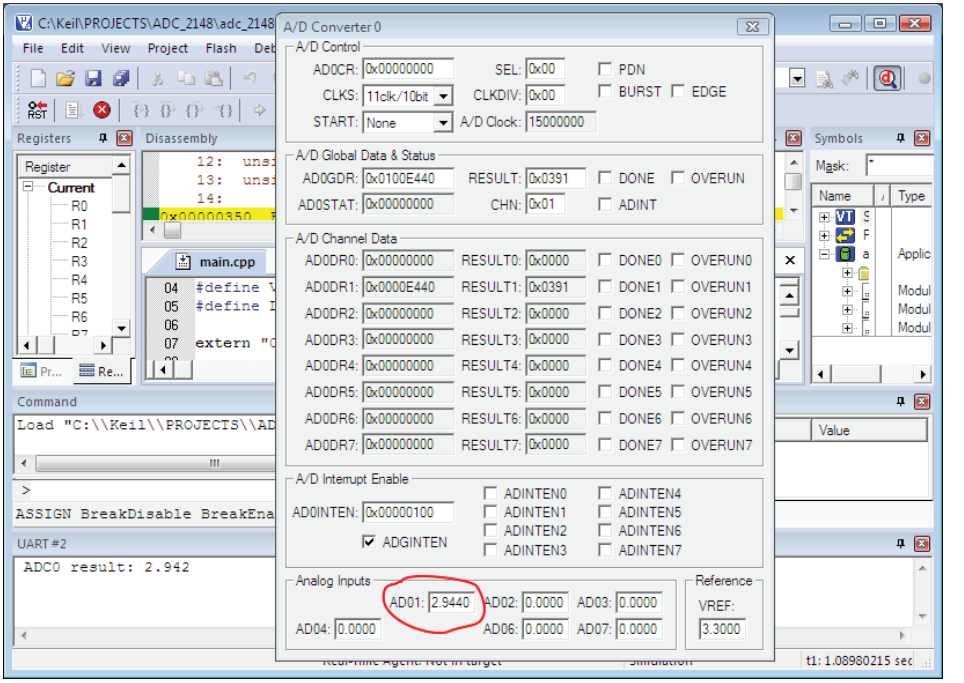

Рис. 3.13.

После этого можно запустить программу на выполнение. В результате выполненного преобразования приложение выведет в терминальное окно **UART #2** (слева внизу) значение входного сигнала (с учетом погрешности преобразования самого АЦП). Выведенное значение, как видно из рисунка, равно 2.942 В.

Если 10-битовой разрешающей способности оказывается недостаточно для получения приемлемого результата, то можно, например, использовать внешний кристалл аналого-цифрового преобразователя с разрядностью 12 или 16 бит и интерфейсом SPI, присоединив его к микроконтроллеру через линии цифрового порта – в этом случае программировать АЦП придется вручную, с использованием временной диаграммы для данного преобразователя.

Приведенные в этой главе примеры программирования периферийных устройств микроконтроллеров ARM могут быть легко усовершенствованы. Кроме этого, исходные тексты программ можно без особого труда модифицировать и впоследствии легко применить для другого процессора ARM.

ГЛАВА 4

ПРОГРАММНЫЙ ИНТЕРФЕЙС С/С++ И АССЕМБЛЕРА ДЛЯ МИКРОКОНТРОЛЛЕРОВ ARM

В этой главе мы рассмотрим принципы создания и функционирования приложений на С/С++, отдельные части которых написаны на языке ассемблера микроконтроллеров ARM7. Использование языка ассемблера (далее по тексту просто "ассемблер") в приложениях для микроконтроллеров (не только ARM) дает ряд существенных преимуществ, которые мы обсудим далее. Но, даже если вы никогда не напишете ни одной строчки программного кода для микроконтроллеров ARM на ассемблере (хотя в это верится с трудом), обойтись без изучения инструкций процессора и принципов функционирования низкоуровневого кода не удастся. Есть по крайней мере одна фундаментальная причина, для чего нужно знать язык ассемблера – он необходим при отладке программного кода. Обойтись без анализа программного кода на низком уровне при отладке более-менее серьезного приложения невозможно. Более того, во многих случаях понимание функционирования ARM-приложения на уровне кодов процессора помогает обнаружить такие ошибки как в самом коде , так и в алгоритме программы, которые никаким другим способом обнаружить не удастся.

Вторая, также веская причина, почему нужно знать ассемблер, связана с оптимизацией приложения. Проблемы оптимизации производительности приложений возникают в тех случаях, когда программа должна работает в реальном времени и выполнять операции, которые требуют максимально высокого быстродействия. Такие приложения работают в системах измерения и управления на производстве и в лабораториях, в мобильных и переносных системах. Каждая такая программа имеет определенные критические участки, на которых выполнение кода должно происходить максимально быстро. Для достижения высокого быстродействия такие участки программного кода целесообразно разрабатывать на языке ассемблера. Во многих случаях именно такая точечная оптимизация помогает существенно повысить быстродействие всего приложения в целом. Вряд ли кому-то придет в голову разрабатывать весь программный код более-менее серьезного приложения на языке ассемблера, поскольку это требует существенных усилий от программиста и наличия времени, но оптимизировать отдельные участки программного кода, зная хотя бы основы программирования на языке ассемблера, можно очень быстро.

В настоящее время разработка встроенных, переносных и мобильных систем переживает настоящий бум. Сегодня от разработчиков программного обеспечения требуется не только знание программирования, скажем, на одном из языков высокого уровня (C, Java, Python и т. д.), но также и понимание основ аппаратной архитектуры системы и интерфейсов. Не случайно в последнее время наметилась тенденция "все в одном", когда разработчик программного обеспечения должен участвовать в разработке интерфейсов системы – специалисты такого уровня очень высоко ценятся на рынке труда как у нас, так и за рубежом. В этом контексте знание архитектуры процессора и выполнения программного кода на уровне инструкций процессора позволяет разработчику достичь высокого уровня универсализации.

Эта глава построена следующим образом. Вначале мы проанализируем, как выполняются некоторые программные алгоритмы на языке макро ассемблера среды Keil ARM, а затем рассмотрим расширенные возможности применения ассемблера в C/C++ приложениях. Для тестирования наших примеров на языке ассемблера будем использовать методику, показанную в следующем примере.

Предположим, мы хотим выполнить операцию вычитания в процедуре на ассемблере и вывести результат в терминальное окно симулятора Keil. Для этого в среде программирования Keil разработаем проект (назовем его **`demoasm1`**), содержащий файл main.cpp (Рис. 4.1):

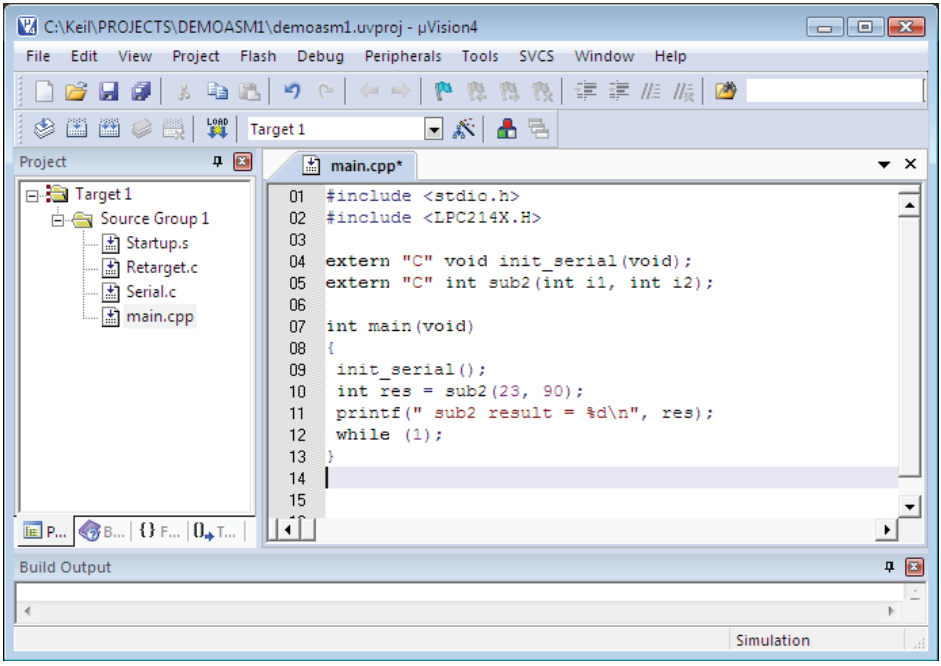

Рис. 4.1.

В исходный текст файла main.cpp (Листинг 4.1) включим объявление внешней процедуры **sub2**, которая будет содержаться в отдельном файле.

Листинг 4.1

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  int  sub2(int  i1,  int  i2);

int  main(void)
{
  init_serial();
  int  res  =  sub2(23,  90);
  printf("  sub2  result  =  %d\n",  res);
  while  (1);
}
```

Как видно из листинга, перед именем процедуры **sub2** указывается спецификатор **extern**, который указывает на то, что данная процедура находится в другом объектном модуле. Спецификатор "C" указывает компоновщику на тип программного интерфейса при вызове данной процедуры. В самой программе значение, возвращаемое процедурой **sub2**, присваивается переменной **res** и выводится в терминальное окно отладчика функцией **printf**. Таким образом, мы сможем проанализировать результат выполнения процедуры на ассемблере.

На втором шаге нам необходимо набрать в редакторе исходный текст процедуры **sub2** и сохранить его в каком-либо файле с расширением .s, например, в файле proc.s. По умолчанию, файл с расширением .s интерпретируется компилятором как содержащий исходный текст на ассемблере, поэтому для компиляции такого файла в объектный модуль будет вызван макро ассемблер Keil (утилита **armasm**). Полученный в результате компиляции объектный модуль будет затем скомпонован с другими объектными модулями проекта.

Вернемся к нашему проекту. Наберем следующий исходный текст на языке ассемблера (Листинг 4.2) и сохраним его в файле proc.s.

Листинг 4.2

```
            AREA  text,  CODE,  READONLY
            EXPORT  sub2
            ENTRY
sub2        SUB      r0,  r0,  r1
            BX        lr
            END
```

Остановимся на анализе этого исходного текста более детально. Директива **AREA** указывает ассемблеру, какой тип секции (код или данные) представлен в данном исходном тексте. Секция представляет собой именованную группу инструкций или данных, которая компонуется линкером (компоновщиком) вместе с другими секциями из других объектных модулей в исполняемый файл.

Область действия директивы распространяется либо до следующей директивы **AREA**, которую встретит макро ассемблер в процессе компиляции, либо (как в нашем случае) до директивы **END**, которая должна быть последней в ассемблерном модуле. Для того, чтобы процедура **sub2** стала доступной из внешних объектных модулей, необходимо указать перед ее именем директиву **EXPORT** – это позволит компоновщику разрешить ссылку на это имя при компоновке объектных модулей.

Директива **ENTRY** в ассемблерной процедуре указывает на первую исполняемую инструкцию, а имя процедуры **sub2** после директивы **ENTRY** обозначает начало этой процедуры.

В соответсвии с соглашением о вызовах процедур (calling conventions) для ARM процессоров при передаче первых четырех параметров в процедуру используются регистры **R0** – **R3**, все остальные параметры (если таковые имеются) передаются через стек. В нашем примере (см. Листинг 4.1) первым параметром процедуры **sub2** является переменная **i1** (регистр **R0**), а вторым – **i2** (регистр **R1**). Если процедура возвращает значение, то оно должно находиться в регистре **R0**. Таким образом, первая инструкция процедуры (**SUB**) вычитает содержимое регистра **R1** из содержимого регистра **R0** и размещает результат в регистре **R0**. Следующая за **SUB** инструкция

```
BX  lr
```

возвращает управление вызывающей C++ программе.

Сохраним исходный текст, показанный в Листинге 4.2, в файле proc.s и включим этот файл в наш проект. Окно проекта должно выглядеть так, как показано на Рис. 4.2 (выделено красной кривой).

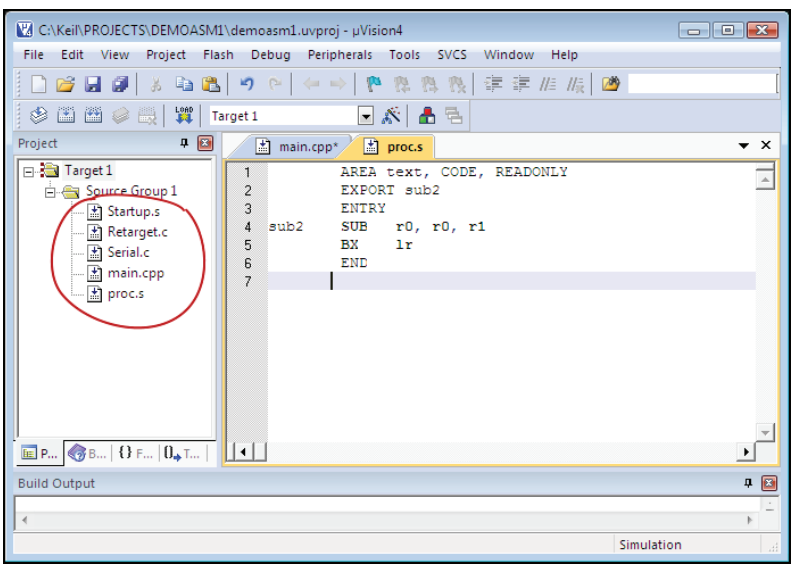

Рис. 4.2.

Откомпилируем наш проект **demoasm1**. Обратите внимание на отчет компилятора/компоновщика в нижней части экрана (окно **Build Output**, Рис. 4.3):

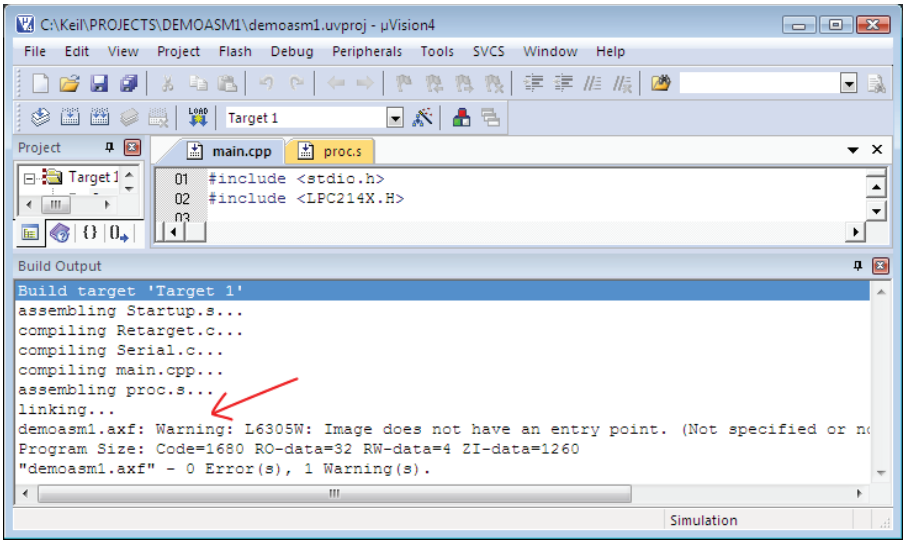

Рис. 4.3.

Наш проект был успешно откомпилирован, в результате чего был создан исполняемый образ в формате ELF (файл demoasm1.axf). Тем не менее, следует обратить внимание на предупреждение (красная стрелка), выданное компоновщиком программы. Его полный текст приводится ниже:

```
. . .
linking...
demoasm1.axf: Warning: L6305W: Image does not have an entry point.
(Not specified or not set due to multiple choices.)
Program Size: Code=1680 RO-data=32 RW-data=4 ZI-data=1260
"demoasm1.axf" - 0 Error(s), 1 Warning(s).
```

Смысл этого сообщения объясняется следующим образом. Каждый исполняемый образ программы при загрузке должен начинать выполняться с некоторого начального адреса, называемого "начальной точкой входа" в программу. Эта точка записывается в заголовок исполняемого файла и используется загрузчиком при вызове программы на выполнение. Только одна точка входа может быть начальной при загрузке программы. Тем не менее, исполняемый файл может содержать несколько точек входа, которые могут быть стартовыми адресами выполняемых процедур, находящихся внутри исполняемого образа программы.

В нашем примере одна из таких точек входа обозначает стартовый адрес процедуры **sub2**, которая будет находиться в объектном модуле proc.o. Если компоновщик обнаруживает несколько точек входа (как в нашем случае), то выдается предупреждение, текст которого показан выше. Несмотря на пре-

дупреждение, компоновщик правильно определяет начальную точку входа в программу (это адрес **main**), но все-таки желательно убрать двусмысленность на этапе компоновки приложения. Сделать это довольно просто – нужно указать начальную точку входа в приложение явным образом, задав значение параметра **entry**.

Для этого в окне проекта нужно выбрать строку **Target1** и в выпадающем меню выбрать опцию **Options for Target 'Target 1'** (показана красной стрелкой на Рис. 4.4).

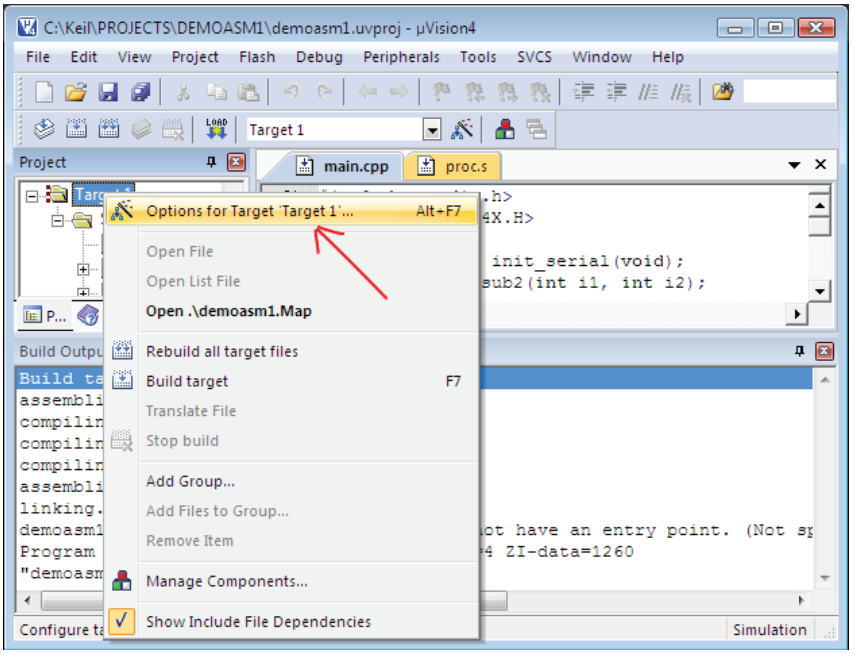

Рис. 4.4.

После выбора этой опции меню появится окно с закладками, в котором нужно выбрать закладку **Linker** и указать явным образом символическое имя начальной точки входа в программу (показано красной стрелкой на Рис. 4.5).

Здесь в окне **Misc controls** нужно ввести строку, в которой указать значение параметра **entry**:

```
--entry=main
```

Когда параметр **entry** получает значение **main**, то компоновщик установит адрес точки входа в программу равным адресу процедуры **main()**, после чего неоднозначность будет устранена и предупреждение компоновщика больше появляться не будет. После запуска приложения на отладку в окно вывода будет выведен результат операции вычитания:

```
sub2  result  =  -67
```

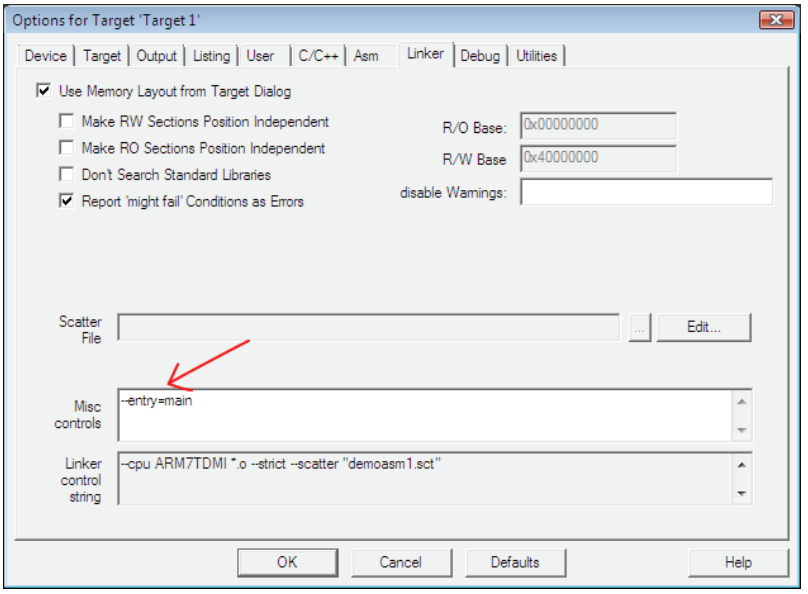

Рис. 4.5.

На этом примере было продемонстрировано, как, используя приложение на C++, проверить результат выполнения программного кода процедуры, разработанной на языке ассемблера. Все последующие примеры из этой главы будут базироваться на этой методике. В следующем разделе рассмотрим несколько примеров программного кода на языке ассемблера, в которых демонстрируется программная техника работы с инструкциями ARM процессора и реализация некоторых программных алгоритмов на языке ассемблера.

4.1. Базовые примеры программного кода на языке ассемблера

Следующий пример демонстрирует вызов нескольких процедур, написанных на ассемблере и находящихся в одном и том же объектном модуле. Здесь вызываются две процедуры, **`sub2`** и **`mul2`**. Первая выполняет сложение двух целых чисел, вторая – умножение. Обе процедуры принимают два параметра в регистрах **`R0`** и **`R1`**, а результат, как обычно, возвращается в регистре **`R0`**. Исходный текст обеих процедур, находящихся в одном файле proc.s, показан в Листинге 4.3.

Листинг 4.3

```
AREA  text,  CODE,  READONLY
EXPORT  sub2
EXPORT  mul2
```

```
            ENTRY
sub2        SUB      r0,  r0,  r1
            BX        lr
mul2        MUL      r2,  r0,  r1
            MOV      r0,  r2
            BX        lr
            END
```

Как видно из листинга, каждая процедура должна быть объявлена со своей директивой **EXPORT**. Процедура **mul2**, как уже было сказано, выполняет умножение двух чисел, переданных ей в качестве параметров, и возвращает результат умножения в регистре **R0**.

Исходный текст C++ приложения, вызывающего обе процедуры, показан в Листинге 4.4.

Листинг 4.4

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  int  sub2(int  i1,  int  i2);
extern  "C"  int  mul2(int  i1,  int  i2);

int  main(void)
{
  init_serial();
  int  res  =  sub2(23,  17);
  printf("  sub2  result  =  %d\n",  res);

  res  =  mul2(23,  17);
  printf("  mul2  result  =  %d\n",  res);

  while  (1);
}
```

Как видно из листинга, каждая из процедур требует отдельного объявления со спецификаторами **extern "C"**.

Процедура на языке ассемблера может содержать, кроме вызываемых из программы на C++, и другие процедуры. При вызове таких внутренних для данного модуля процедур нужно учитывать тот факт, что адрес возврата в вызывающую процедуру находится в регистре связи **LR** (регистр **R14**), поэтому, например, при последовательном вызове двух процедур регистр **LR** будет содержать адрес возврата в последнюю вызывающую процедуру, в то время как самый первый адрес возврата в вызывающую программу будет потерян. В этом случае можно сохранить более "ранний" адрес возврата в стеке с помощью инструкции **STM** (при входе в процедуру) и восстановить этот адрес инструкцией **LDM** по завершению процедуры.

Эта методика продемонстрирована в следующем примере. Основная программа на C++ вызывает две внешние процедуры, **sub2** и **addmul2**, написанные на ассемблере и расположенные в файле proc.s (Листинг 4.5).

Листинг 4.5

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  int  sub2(int  i1,  int  i2);
extern  "C"  int  addmul2(int  i1,  int  i2);

int  main(void)
{
  init_serial();

  int  res  =  sub2(23,  17);
  printf("  sub2  result  =  %d\n",  res);

  res  =  addmul2(23,  17);
  printf("  addmul2  result  =  %d\n",  res);

  while  (1);
}
```

Файл proc.s содержит следующий исходный текст (Листинг 4.6):

Листинг 4.6

```
                    AREA  text,  CODE,  READONLY
                    EXPORT  sub2
                    EXPORT  addmul2
                    ENTRY
sub2                SUB      r0,  r0,  r1
                    BX        lr
;------------------------
addmul2            STMFD  sp!,  {lr}
                    ADD      r0,  r0,  r1
                    BL        mul2
                    MOV      r0,  r1
                    LDMFD  sp!,  {pc}
;------------------------
mul2                MUL  r1,  r0,  r0
                    BX    lr
                    END
```

В этом файле находятся три процедуры. Процедуры **sub2** и **addmul2** объявлены с директивой **EXPORT**, поскольку к ним нужен доступ из другого объектного файла. Процедура **mul2** является внутренней и используется как вспомогательная процедурой **addmul2**. Процедура **addmul2** выполняет две операции – складывает два целых числа, находящихся в регистрах **R0** и **R1**, после чего вызывает процедуру **mul2** посредством инструкции

```
BL        mul2
```

Процедура **mul2**, в свою очередь, вычисляет квадрат целого числа, которое передается как параметр в регистре **R0**. По завершению этой процедуры результат записывается в регистр **R0** инструкцией

```
MOV      r0,  r1
```

для передачи в основную программу.

Обратите внимание на пару инструкций **STMFD/LDMFD**. Инструкция **STMFD** сохраняет адрес возврата в основную программу в стеке, а инструкция **LDMFD** загружает этот адрес в счетчик инструкций программы (регистр **R15**, **pc**) по завершению процедуры **addmul2**. Это очень важный момент, поэтому проанализируем его более подробно.

При вызове процедуры **addmul2** из основной программы регистр связи **LR** содержит адрес инструкции, которая будет выполняться следующей по завершению этой процедуры. Инструкция

```
STMFD  sp!,  {lr}
```

сохраняет содержимое регистра **LR** в стеке, поскольку при вызове **mul2** содержимое регистра связи будет разрушено – там теперь будет содержаться адрес инструкции

```
MOV      r0,  r1
```

которая будет выполнена по завершению процедуры **mul2**. Если не сохранить адрес возврата в основную программу, то по завершению процедуры **addmul2** в счетчике инструкций программы (регистр **R15**) окажется адрес инструкции **MOV r0, r1** вместо адреса инструкции из основной программы. Таким образом, если не откорректировать содержимое регистра **LR**, то возврат в основную программу будет невозможен и приложение будет выполняться непредсказуемым образом.

По вышеизложенным причинам следующая последовательность инструкций микроконтроллера является ошибочной:

```
addmul2             ADD       r0,  r0,  r1
                    BL        mul2
                    MOV       r0,  r1
                    BX        lr
```

Инструкция

```
LDMFD  sp!,  {pc}
```

загружает адрес следующей инструкции непосредственно в регистр-счетчик инструкций программы. Вместо этой одной инструкции можно использовать более понятную последовательность

```
LDMFD  sp!,  {lr}
BX     lr
```

Мы рассмотрели основы построения программных интерфейсов между приложениями, написанными на C/C++, и процедурами, написанными на языке макро ассемблера среды Keil. В следующем разделе вы увидите многочисленные примеры, в которых показаны методы решения практических задач в процедурах на языке ассемблера.

4.2. Примеры решения практических задач программирования на языке ассемблера

Пример 1. В этом примере (процедура **add2**, Листинг 4.7) показано сложение двух целых чисел, значения которых находятся в области памяти, определенной директивами **DCD**. Адреса обеих переменных загружаются в регистры **R0** и **R1** псевдоинструкцией **ADDR**, а результат операции возвращается в регистре **R0**.

Листинг 4.7.

```
add2            ADR         r0,  i1
                ADR         r1,  i2
                LDR         r0,  [r0]
                LDR         r1,  [r1]
                ADD         r0,   r0,  r1
                BX          lr
i1              DCD         32
i2              DCD         -7
                END
```

Пример 2. Здесь показаны фрагменты программного кода на языке макро ассемблера Keil, демонстрирующие операции целочисленного умножения и деления.

Следующая группа инструкций выполняет умножение содержимого регистров **R0** и **R1** и сохраняет результат в регистре **R0**.

```
MUL         r2,  r0,  r1
MOV         r0,  r2
```

Следующая инструкция позволяет выполнить умножение содержимого регистра **R0** на 5, используя аппаратные функции сдвига процессора ARM.

```
ADD         r0,  r0,  r0,  LSL  #2
```

Комбинация операций умножения и сложениея в одной инструкции позволяет вычислять простые выражения, представленные формулой **ab + c:**

```
MLA         r3,  r0,  r1,  r2          ;  r3  =  r0  x  r1  +  r2
```

В соответствии с формулой, показанной выше, значение **a** располагается в регистре **R0**, значение **b** содержится в регистре **R1** и значение **c** находится в **R2**. Результат операции будет сохранен в регистре **R3**.

Более сложное выражение, наподобие $ax^2 + by + c$, можно вычислить, используя программный код, показанный в Листинге 4.8. Здесь процедура **mladd2** вычисляет выражение, представленное как

$$r1 \times r0^2 + r2 \times r3 + r4$$

Таким образом, значение **a** располагается в регистре **R1**, значение **x** находится в регистре **R0**, значение **b** располагается в **R2**, значение **y** находится в **R3** и, наконец, значение **c** загружается из стека. Результат выполнения процедуры возвращается в регистре **R0**.

Листинг 4.8

```
mladd2      MOV         r5,  r0
            MUL         r6,  r0,  r5          ;  r6  =  r0  x  r0  ->  r0²
            MUL         r5,  r6,  r1          ;  r5  =  r6  *  r1  =  r1  x  r0²
            LDR         r4,  [sp]
            MLA         r6,  r2,  r3,  r4        ;  r6  =  r2  x  r3  +  r4
            ADD         r5,  r5,  r6        ;  r5  =  r1  x  r0²  +  r2  x  r3  +  r4
            MOV         r0,  r5                ;  r0  =  r5
            BX          lr
```

Операции деления могут быть реализованы, как комбинации операций сложения, вычитания, умножения и сдвига. Следующая инструкция выполняет деление содержимого регистра **R0** на 16.

```
MOV         r0,  r0,  ASR  #4        ;  r0  =  r0  /  16
```

Следующий пример более сложный и демонстрирует вычисление выражения `– (x – x/4)`.

Процедура **divex** (Листинг 4.9) в процессе вычисления использует регистры **R0** и **R1** и возвращает результат в регистре **R0**.

Листинг 4.9

```
divex       MOV         r1,  r0,  ASR  #2
            SUB         r0,  r1
            RSB         r0,  #0                ;  r0  =  -(r0  -  r0/4)
            BX          lr
```

Пример 3. В этом примере показано, как оперировать данными, расположенными в сегменте данных процедуры на языке ассемблера. Процедуры **add2** и **sub2**, исходный текст которых показан в Листинге 4.10, выполняют сложение и вычитание целых чисел, хранящихся в сегменте данных с именем **myDAT**, определенном как

```
AREA  myDat,  DATA,  READWRITE
```

Директива **AREA** определяет секцию **myDat**, как сегмент данных (ключевое слово **DATA**) с доступом по чтению/записи данных (ключевое слово **READWRITE**).

Листинг 4.10

```
add2        LDR         r1,  =i1
            LDR         r2,  =i2
            LDR         r1,  [r1]
            LDR         r2,  [r2]
            ADD         r1,  r1,  r2
            ;
            LDR         r3,  =myData
            STR         r1,  [r3]
            ;
            LDR         r0,  =myData
            LDR         r0,  [r0]
            BX          lr
```

```
;------------------------------
sub2            LDR         r1,  =i1
                LDR         r2,  =i2
                LDR         r1,  [r1]
                LDR         r2,  [r2]
                SUB         r1,  r1,  r2
                ;
                LDR         r3,  =myData
                STR         r1,  [r3,  #4]
                ;
                LDR         r0,  =myData
                LDR         r0,  [r0,  #4]
                BX          lr
;--------------------------------
              AREA  myDat,  DATA,  READWRITE
i1            DCD         -72
i2            DCD         -19
myData        SPACE       8
              END
```

В секции **myDat**, определенной как сегмент данных (атрибут **DATA)**, располагаются целочисленные переменные **i1** и **i2**, которым присвоены начальные значения −72 и −19 соответственно. В этой же секции располагается область неинициализированных данных **myData**, для которой выделено 8 байт памяти.

Каждая из процедур вначале копирует содержимое переменных **i1** и **i2** в регистры **R1** и **R2** соответственно, используя четыре инструкции **LDR**. Затем выполняется соответствующая операция (сложение или вычитание), после чего результат сохраняется в соответствующей области памяти. Для процедуры **add2** сумма сохраняется по адресу **myData** с помощью следующих инструкций:

```
LDR         r3,  =myData
STR         r1,  [r3]
```

Процедура **sub2** сохраняет результат операции по адресу **myData+4** с помощью инструкций

```
LDR         r3,  =myData
STR         r1,  [r3,  #4]
```

Каждая из процедур возвращает результат операции в регистре **R0**.

В следующих двух примерах показано применение логических инструкций микроконтроллера ARM.

Пример 4. В данном примере процедура **setbit**, исходный текст которой показан в Листинге 4.11, позволяет установить заданный бит в 32-разрядном целом. Процедура принимает два параметра: целое число (регистр **R0**) и номер позиции бита, который должен быть установлен (регистр **R1**). Процедура возвращает результат в регистре **R0**.

Листинг 4.11

```
setbit          MOV       r2,  #1
                MOV       r2,  r2,  LSL  r1
                ORR       r0,  r0,  r2
```

```
                BX        lr
                END
```

Пример 5. Процедура **checkbit**, исходный текст которой показан в Листинге 4.12, демонстрирует, как определить значение отдельного бита в 32-разрядном целом. Процедура принимает два параметра: целое число в регистре **R0** и номер позиции в 32-разрядном числе, которая должна быть протестирована (регистр **R1**). Процедура возвращает значение бита (0 или 1) в данной позиции в регистре **R0**.

Листинг 4.12

```
                    AREA  text,  CODE,  READONLY
                    EXPORT  checkbit
                    ENTRY
checkbit            MOV     r2,  #1
                    MOV     r2,  r2,  LSL  r1
                    TST     r0,  r2
                    MOVEQ   r0,  #0
                    MOVNE   r0,  #1
                    BX      lr
                    END
```

Эту процедуру может вызывать приложение на C++ с исходным текстом, показанным в Листинге 4.13.

Листинг 4.13

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  int  checkbit(int  num,  int  cbit);

int  main(void)
{
  init_serial();
  int  num  =  9179,  cbit  =  7;

  int  res  =  checkbit(num,  cbit);
  if  (res  !=  0)
      printf("  Bit  %d  is  set.\n",  cbit);
  else
      printf("  Bit  %d  is  cleared\n",  cbit);
  while  (1);
}
```

В этом приложении анализируется бит 7 числа 9179. В результате выводится следующее сообщение:

```
Bit  7  is  set
```

Пример 6. Процедура **set_array** из этого примера позволяет инициализировать каждый элемент массива целых чисел определенным значением, ко-

торое может определяться какой-либо формулой. В данном примере первому элементу массива присваивается значение 0, а каждый последующий элемент принимает значение на 3 больше предыдущего, т.е. второй элемент массива будет иметь значение 3, третий 6 и т. д.

Исходный текст процедуры приведен в Листинге 4.14.

Листинг 4.14

```
                    AREA  text,  CODE,  READONLY
                    EXPORT  set_array
                    ENTRY
set_array           LDR         r0,  =i1
                    STMFD       sp!,  {r0}
                    MOV         r1,  #32
                    MOV         r3,  #0
next_term           STR         r3,  [r0],  #4
                    SUBS        r1,  r1,  #1
                    BEQ         exit
                    ADD         r3,  r3,  #3
                    B           next_term
exit                LDMFD       sp!,  {r0}
                    BX          lr
                    AREA       myData,  DATA
i1                  SPACE     128
                    END
```

В этой процедуре определен массив целых чисел **i1**, для чего в секции данных **myData** резервируется буфер памяти размером 128 байт (директива **SPACE**). Адресация массива осуществляется посредством регистра **R0**, а размер массива, равный 32, хранится в регистре **R1** – это значение служит счетчиком цикла при обработке элементов массива.

Регистр **R3** инициализируется нулем – это значение будет записано в первый элемент массива. В каждой последующей итерации содержимое **R3** инкрементируется на 3 инструкцией

```
ADD         r3,  r3,  #3
```

Управление циклом осуществляют инструкции

```
SUBS       r1,  r1,  #1
BEQ         exit
```

Инструкция **SUBS** декрементирует счетчик цикла в **R3** на 1 и, в зависимости от полученного значения, устанавливает соответствующие флаги в регистре состояния программы **cpsr**. Цикл выполняется до тех пор, пока не установлен флаг **Z** – в этом случае следующая инструкция **BEQ** пропускается и выполняется переход на начало цикла (метка **next_term**).

Если флаг **Z** устанавливается в 1 (**R3** = 0), то цикл завершается. Инструкции **STMFD/LDMFD** выполняет сохранение/восстановление адреса массива **i1**. Процедура возвращает адрес массива в регистре **R0**. Вызов процедуры **set_array** из C++ приложения показан в Листинге 4.15.

Листинг 4.15

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  int*  set_array(void);

int  main(void)
{
  init_serial();
int  *pbuf  =  set_array();

  for  (int  i1  =  0;  i1  <  32;  i1++)
    printf("%d  ",  *(pbuf++));
  while  (1);
  }
```

В этом приложении выполняется вызов процедуры **set_array**, после чего содержимое массива выводится в терминальное окно функцией **printf**.

Пример 7. Процедура **max_int** из этого примера выполняет поиск максимального значения в массиве целых чисел (Листинг 4.16).

Листинг 4.16

```
                AREA  text,  CODE,  READONLY
                EXPORT  max_int
                ENTRY
max_int         LDR     r0,  =i1
                LDR     r1,  =i2
                SUB     r1,  r1,  r0
                MOV     r1,  r1,  LSR  #2
                SUB     r1,  r1,  #1
                LDR     r2,  [r0]
next_cmp        LDR     r3,  [r0,  #4]!        ;  pre-indexed  addressing
                CMP     r2,  r3
                MOVLT   r2,  r3
                SUBS    r1,  r1,  #1
                BNE     next_cmp
                MOV     r0,  r2
                BX      lr
                AREA  myData,  DATA
i1              DCD  -42,  -33,  -5,  -12,  -9,  -4,  -34,  -62
i2              EQU  i1+32
                END
```

Здесь в секции данных **myData** определен массив **i1** из 8-ми целых чисел. Константа **i2** содержит значение адреса последнего элемента массива – это значение будет использоваться при определении размера массива. Адрес массива **i1** загружается в регистр **R0** инструкцией **LDR**:

```
LDR     r0,  =i1
```

Через **R0** осуществляется доступ ко всем элементам методом пре-индексирования адреса, при этом значение текущего элемента будет загружено в регистр

R3. Значение локального максимума в каждой итерации хранится в регистре **R2**. Значение текущего элемента в **R3** сравнивается со значением в **R2** при помощи инструкции

```
CMP      r2,  r3
```

Если содержимое регистра **R2** оказывается меньше чем то, что находится в **R3**, содержимое **R3** копируется в регистр **R2** при помощи инструкции

```
MOVLT    r2,  r3
```

Это значение становится локальным максимумом, как минимум, до следующей итерации. По завершению цикла значение локального максимума переписывается в регистр **R0**.

Для выполнения цикла поиска нужно вычислить значение счетчика цикла. Это выполняется следующей группой инструкций:

```
LDR      r1,  =i2
SUB      r1,  r1,  r0
MOV      r1,  r1,  LSR  #2
SUB      r1,  r1,  #1
```

Поскольку в регистре **R0** находится начальный адрес массива целых чисел, а в регистр **R1** записывается адрес последнего элемента, то, вычислив разность **R1** – **R0**, мы получим размер массива в байтах (инструкция **SUB**). Для определения количества целых чисел в массиве необходимо разделить значение, полученное в регистре **R1**, на 4. Эту операцию выполняет инструкция **MOV**. Количество попарных сравнений в массиве будет меньше на 1 размера массива, поэтому счетчик циклов должен быть равен значению **R1** – **1** (эту коррекцию выполняет вторая инструкция **SUB** в этой группе).

Процедура **max_int** должна быть объявлена в вызывающей программе следующим образом:

```
extern  "C"  int  max_int(void);
```

Пример 8. Этот пример демонстрирует как найти максимальное значение в массиве целых чисел, который заканчивается элементом со значением 0xFFF. Исходный текст процедуры (назовем ее **max_int2**) на языке макро ассемблера Keil показан в Листинге 4.17:

Листинг 4.17

```
                    AREA  text,  CODE,  READONLY
                    EXPORT  max_int2
                    ENTRY
max_int2            LDR      r0,  =i1
                    LDR      r1,  =i2
                    LDR      r1,  [r1]
                    LDR      r2,  [r0]
next_cmp            LDR      r3,  [r0,  #4]!
                    CMP      r3,  r1
                    BEQ      exit
                    CMP      r2,  r3
```

```
                    MOVLT    r2,  r3
                    B        next_cmp
exit                MOV      r0,  r2
                    BX       lr
                    AREA  myData,  DATA
i1                  DCD    142,  -33,  9,  -12,  3,  -4,  -34,  62,  211
i2                  DCD    0xFFF
                    END
```

В этой процедуре адрес массива целых чисел **i1** загружается в регистр **R0**. В регистре **R1** хранится признак окончания массива (0xFFF), регистр **R2** содержит локальное значение максимума в текущей итерации. Очередной элемент массива загружается в регистр **R3** и сравнивается с текущим значением локального максимума инструкцией

```
CMP      r2,  r3
```

Инструкция

```
MOVLT    r2,  r3
```

записывает новое значение максимума из регистра **R3** в регистр **R2**, если в результате сравнения оказалось, что **R2 < R3**. Каждая итерация цикла начинается с метки **next_cmp**, где выпоняется загрузка очередного элемента массива для сравнения с локальным максимумом. В каждой итерации проверяется, достигнут ли конец массива с помощью инструкции

```
CMP      r3,  r1
```

Если в результате сравнения устанавливается флаг **Z**, то происходит выход из цикла по инструкции

```
BEQ      exit
```

Инструкция

```
MOV      r0,  r2
```

загружает значение максимума в регистр **R0** для возврата результата в вызывающую процедуру.

Пример 9. Данный пример демонстрирует, как можно изменить знак каждого элемента в массиве целых чисел (процедура **rev_sign**), который заканчивается терминирующим элементом 0xFFF. Исходный текст данной процедуры показан в Листинге 4.18.

Листинг 4.18

```
                    AREA  text,  CODE,  READONLY
                    EXPORT  rev_sign
                    ENTRY
rev_sign            LDR      r0,  =i1
                    LDR      r1,  =i2
                    LDR      r1,  [r1]
                    STMFD    sp!,  {r0}
next_cmp            LDR      r2,  [r0],  #4
```

```
                    CMP     r2,  r1
                    BEQ       exit
                    MVN     r2,  r2
                    ADD     r2,  r2,  #1
                    STR     r2,  [r0,  #-4]
                    B       next_cmp
exit                LDMFD   sp!,  {r0}
                    BX      lr
                    AREA  myData,  DATA
i1                  DCD    242,  -33,  900,  -12,  3,  -4,  -34,  62,  211
i2                  DCD    0xFFF
                    END
```

Обработка элементов происходит в цикле, который начинается с метки **next_cmp**. Очередной элемент массива загружается в регистр **R2** инструкцией

```
LDR     r2,  [r0],  #4
```

Содержимое **R2** проверяется на совпадение с терминальным элементом инструкцией

```
CMP     r2,  r1
```

При равенстве элементов устанавливается флаг **Z** в регистре состояния и происходит выход из цикла по инструкции **BEQ** на метку **exit**.

Инверсия элемента выполняется инструкциями

```
MVN     r2,  r2
ADD     r2,  r2,  #1
```

после чего новое значение записывается по адресу элемента с помощью инструкции

```
STR     r2,  [r0,  #-4]
```

Поскольку инструкция загрузки **LDR** инкрементировала адрес элемента, то для записи по предыдущему адресу нужно указать смещение –4.

Пример 10. Процедура **search_int**, описанная в данном примере, выполняет поиск первого элемента целочисленного массива, превышающего заданное значение (10 в данном случае).

Исходный текст процедуры показан в Листинге 4.19.

Листинг 4.19

```
                        AREA  text,  CODE,  READONLY
                        EXPORT  search_int
                        ENTRY
search_int              LDR       r0,  =i1
                        MOV       r2,  #5
next                    LDR       r1,  [r0],  #4
                        CMP       r1,  #10
                        BGT       exit
                        SUBS      r2,  r2,  #1
                        BNE       next
                        MOV       r0,  #0
                        BX        lr
```

```
exit                        MOV        r0,  r1
                            BX         lr
                            AREA  myData,  DATA
i1                          DCD    8,  4,  -3,  11,  30
                            END
```

Адрес массива **i1** загружается в регистр **R0**, а его размер – в **R2**. Обработка элементов массива выполняется в цикле, который начинается с метки **next**. Текущий элемент массива загружается в регистр **R1** инструкцией

```
LDR        r1,  [r0],  #4
```

Следующая за **LDR** инструкция

```
CMP        r1,  #10
```

выполняет сравнение содержимого **R1** со значением 10. Если число в регистре **R1** превышает 10, то происходит выход из цикла по инструкции

```
BGT  exit
```

на инструкцию

```
MOV        r0,  r1
```

которая сохраняет значение обнаруженного элемента в регистре **R0**. Управление циклом осуществляется инструкциями

```
SUBS       r2,  r2,  #1
BNE        next
```

Если по завершению цикла элемент массива, удовлетворяющий условию, так и не был обнаружен, в регистр **R0** записывается 0:

```
MOV        r0,  #0
```

Процедура завершается, как обычно, инструкцией

```
BX  lr
```

В Листинге 4.20 показан исходный текст программы на Keil C++, которая вызывает процедуру **search_int**.

Листинг 4.20

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

extern  "C"  int  search_int(void);

int  main(void)
{
  init_serial();
  int  res  =  search_int();
  printf("  The  first  integer  greater  than  10  is  %d\n",  res);
```

```
    while  (1);
}
```

Пример 11. Этот пример является расширением предыдущего. Процедура **search_int2** с исходным текстом, показанным в Листинге 4.21, выполняет поиск первого элемента целочисленного массива, попадающего в заданный диапазон (0 – 10 для данного примера).

Листинг 4.21

```
                        AREA  text,  CODE,  READONLY
                        EXPORT  search_int2
                        ENTRY
search_int2             LDR         r0,  =i1
                        MOV         r2,  #5
next                    LDR         r1,  [r0],  #4
                        CMP         r1,  #0
                        BGT         next_cmp
check_cnt               SUBS        r2,  r2,  #1
                        BNE         next
                        B             exit
next_cmp                CMP         r1,  #10
                        BLT         found
                        B           check_cnt
exit                    MOV         r0,  #0
                        BX          lr
found                   MOV         r0,  r1
                        BX          lr
                        AREA  myData,  DATA
i1                      DCD    -8,  6,  -3,  21,  9
                        END
```

В этой процедуре переменная **i1** указывает на массив из пяти целых чисел. Адрес переменной загружается в регистр **R0**, а размер массива в регистр **R2**. Инструкции

```
SUBS       r2,  r2,  #1
BNE         next
```

управляют циклом поиска, который начинается с метки **next**. Содержимое текущего элемента массива загружается в регистр **R1** для анализа инструкцией

```
LDR         r1,  [r0],  #4
```

Анализ содержимого регистра R1 выполняется двумя инструкциями сравнения:

```
                    CMP         r1,  #0
                    BGT         next_cmp
                       .  .  .
next_cmp            CMP         r1,  #10
                    BLT         found
```

Если первый искомый элемент обнаружен, происходит переход по метке **found**, где содержимое регистра **R1** копируется в регистр **R0**. Если по за-

вершинию цикла элемент, удовлетворяющий критерию поиска, так и не был обнаружен, то в регистр **R0** записывается 0.

Управление циклом в данной процедуре осуществляют инструкции

```
SUBS        r2,  r2,  #1
BNE         next
B             exit
```

Пример 12. В этом примере будет продемонстрировано, как установить абсолютные (неотрицательные) значения элементов целочисленного массива. Чтобы это сделать, нужно поменять знак на противоположный только у отрицательных элементов, оставив положительные без изменения. Далее в Листинге 4.22 показан исходный текст процедуры **absproc**, которая выполняет инверсию знака отрицательных элементов массива.

Листинг 4.22

```
                    AREA  text,  CODE,  READONLY
                    EXPORT  absproc
                    ENTRY
absproc             LDR         r0,  =i1
                    STMFD       sp!,  {r0}
                    MOV         r2,  #5
next                LDR         r1,  [r0],  #4
                    CMP         r1,  #0
                    RSBLT       r1,  r1,  #0
                    STRLT       r1,  [r0,  #-4]
                    SUBS        r2,  r2,  #1
                    BNE         next
                    LDMFD       sp!,  {r0}
                    BX          lr
                    AREA  myData,  DATA
i1                  DCD    -78,  4,  -3,  1,  -530
                    END
```

В этой процедуре адрес массива **i1**, состоящего из пяти целых чисел, загружается в регистр **R0**, а его размер – в регистр **R2**. Значение текущего элемента массива загружается в регистр **R1**, после чего сравнивается с нулем инструкцией

```
CMP         r1,  #0
```

Если число оказывается отрицательным (флаг **Z** установлен), то будет выполнена следующая за **CMP** инструкция

```
RSBLT     r1,  r1,  #0
```

которая выполняет вычитание содержимого **R1** из нуля – это означает, что число в регистре **R1** меняет знак на положительный. Инструкция

```
STRLT     r1,  [r0,  #-4]
```

записывает содержимое **R1** по тому же адресу в массиве, который был считан, при этом текущий адрес должен быть декрементирован. Обе инструкции будут

пропущены, если по результату сравнения в инструкции **CMP** число в регистре **R1** оказывается положительным. Обработка элементов массива выполняется в цикле, который начинается с метки **next**, а управление циклом осуществляется инструкциями

```
SUBS        r2,  r2,  #1
BNE         next
```

Следующая группа примеров показывает, как можно реализовать управляющие операторы языка C/C++, такие, например, как **switch**…**case**, **while**, **for**, **do**…**while**, на языке ассемблера.

Пример 13. Данный пример демонстрирует реализацию оператора **switch**… **case** языка C/C++ на ассемблере. Напомню, что оператор **switch**…**case** в общем виде можно представить следующим образом:

```
int  i1;
switch  (i1)
{
  case  1:
    <операторы_1>;
  case  2:
    <операторы_2>;
  case  3:
    <операторы_3>;
  default:
      break;
  }
```

Один из вариантов реализации этого оператора представлен процедурой **swproc**, исходный текст которой показан в Листинге 4.23.

Листинг 4.23

```
                AREA  text,  CODE,  READONLY
                EXPORT  swproc
                ENTRY
swproc          CMP         r0,  #1
                BEQ         label1
                CMP         r0,  #2
                BEQ         label2
                CMP         r0,  #3
                BEQ         label3
                LDR         r0,  =data4
                B             exit
label1          LDR         r0,  =data1
                B             exit
label2          LDR         r0,  =data2
                B             exit
label3          LDR         r0,  =data3
exit            BX          lr
                AREA  myData,  DATA
data1       DCB     "  You  selected  1",  0
data2       DCB     "  You  selected  2",  0
data3       DCB     "  You  selected  3",  0
```

```
data4       DCB     " Nothing between 1-3 was selected", 0
                END
```

Программный код процедуры выполняется следующим образом. Предположим, что в регистре **R0** содержится целое число в диапазоне 1 – 3. Тогда, в зависимости от конкретного значения **R0**, одна из инструкций **BEQ** выполнит передачу управления на одну из меток **label1**, **label2** или **label3**. Затем одна из инструкций **LDR** загружает адрес одной из символьных строк в регистр **R0**. При вызове этой процедуры из C++ приложения адрес строки передается в основную программу по завершению **swproc**.

Пример 14. В этом примере показана программная реализация цикла **while** языка C/C++ на языке ассемблера программной среды Keil. Напомню, что управляющий оператор цикла while может быть представлен следующим образом:

```
while  (условие)
{
      <операторы>
}
```

Операторы внутри фигурных скобок будут выполняться только в том случае, если ***условие*** является истинным. Условие проверяется перед выполнением первого оператора цикла, поэтому возможна ситуация, когда тело цикла не буде выполнено ни разу.

Например, в следующем фрагменте программного кода на C++ подсчитывается сумма элементов до тех пор, пока не будет обнаружен нулевой элемент:

```
int  i  =  0;
int  sum  =  0;
while  (a[i]  !=  0)
{
      sum  =  sum  +  a[i];
      i++;
}
```

Если первый элемент массива a окажется нулевым, цикл не выполнится ни разу. Аналогом этого цикла на языке ассемблера будет следующая процедура (назовем ее **whileproc**) на ассемблере (Листинг 4.24).

Листинг 4.24

```
                    AREA  text,  CODE,  READONLY
                    EXPORT  whileproc
                    ENTRY
whileproc           MOV       r0,  #0
                    LDR       r1,  =data1
next                LDR       r2,  [r1],  #4
                    CMP       r2,  #0
                    ADDNE     r0,  r0,  r2
                    BNE       next
                    BX        lr
                    AREA     myData,  DATA
```

```
data1           DCD    32,  45,  -78,  10,  0,  12
                END
```

В этой процедуре цикл начинается с метки **next**. Вначале проверяется на равенство нулю значение текущего элемента массива **data1**, адресуемого регистром **R1** и загруженного в регистр **R2** инструкцией

```
LDR         r2,  [r1],  #4
CMP         r2,  #0
```

Если элемент не равен нулю, то его значение складывается со значением суммы (хранится в регистре **R0**) с помощью инструкции

```
ADDNE     r0,  r0,  r2
```

Следующая инструкция

```
BNE         next
```

передает управление в начало цикла, только если текущий элемент был ненулевым.

Оператор **do**…**while** выполняется точно так же, как и **while**, за исключением проверки условия – оно проверяется в конце итерации. Общая форма оператора **do**…**while** языка C/C++ представлена ниже:

```
do  {

      <операторы>

}  while  (условие)
```

Операторы в теле цикла будут выполнены как минимум один раз, до того, как состоится проверка условия. Например, в следующем операторе

```
int  i  =  -1;
int  sum  =  0;
do
{
      i++;
      sum  =  sum  +  a[i];
}  while  (a[i]  !=  0)
```

минимальное значение суммы будет равно 0, поскольку начальное значение **sum** равно 0, и оно не изменится, даже если первый элемент массива будет равен 0. Программная реализация цикла **do**…**while** на языке макро ассемблера Keil (процедура **dowhile**) будет выглядеть следующим образом (Листинг 4.25):

Листинг 4.25

```
                    AREA  text,  CODE,  READONLY
                    EXPORT  dowhile
                    ENTRY
dowhile             MOV         r0,  #0
                    LDR         r1,  =data1
next                LDR         r2,  [r1],  #4
```

```
                    ADD         r0,  r0,  r2
                    CMP         r2,  #0
                    BNE         next
                    BX          lr
                    AREA  myData  DATA
data1               DCD    32,  45,  -78,  10,  0,  12
                        END
```

Исходный текст данной процедуры очень похож на тот, что был рассмотрен при анализе цикла **while**. Основное отличие в том, что здесь инструкция

```
ADD         r0,  r0,  r2
```

выполняется безусловно до того, как проверяется условие выполнения цикла (следующая за инструкция **CMP**).

Подобным образом на языке ассемблера может быть реализован и оператор **for** языка C/C++. Следует отметить, что управляющие операторы языков высокого уровня на язык ассемблера транслируются почти одними и теми же группами операторов. Как правило, проверка условия во всех случаях выполняется двумя

```
<метка>
            .  .  .
            CMP  <операнд1,  операнд2>
            BNE/BE  <метка>
            .  .  .
```

или более инструкциями. Здесь возможны самые разные варианты реализации, хотя, при большом числе итераций в цикле, нужно подумать об оптимизации такого участка программы. Вопросы оптимизации кода программы при циклических вычислениях мы рассмотрим в главе 6.

Пример 15. Процедура **cpy_apos**, показанная в этом примере, выполняет копирование всех положительных элементов целочисленного массива в буфер памяти. Адрес исходного массива находится в регистре **R0**, адрес буфера передается в регистре **R1**, а размер массива-источника находится в регистре **R2**. Размер буфера памяти массива-приемника должен быть по крайней мере не меньше размера массива-источника.

Исходный текст процедуры **cpy_apos** показан в Листинге 4.26.

Листинг 4.26

```
cpy_apos            LDR         r3,  [r0],  #4
                    CMP         r3,  #0
                    STRGT       r3,  [r1],  #4
                    SUBS        r2,  r2,  #1
                    BNE         cpy_apos
                    BX           lr
```

Пример расположения элементов в массиве-источнике и в массиве-приемнике показано ниже:

```
источник:{12,-55,9,-34,-19,4,-7}
приемник:{12,9,4,0,0,0,0,0,0,0}
```

Пример 16. Процедура **sum2_arrays**, исходный текст которой показан в Листинге 4.27, выполняет попарное сложение элементов двух целочисленных массивов с одинаковыми индексами и записывает результат в буфер памяти. Исходные массивы адресуются регистрами **R0** и **R1**, адрес буфера памяти содержится в регистре **R2**, а размер любого из исходных массивов (предполагается, что размеры исходных массивов одинаковы) находится в регистре **R3**.

Листинг 4.27

```
sum2_arrays       SUB    r0,  r0,  #4
                  SUB    r1,  r1,  #4
next              LDR    r4,  [r0,  #4]!
                  LDR    r5,  [r1,  #4]!
                  ADD    r4,  r4,  r5
                  STR    r4,  [r2],  #4
                  SUBS   r3,  r3,  #1
                  BNE    next
                  BX     lr
```

В этой процедуре значения текущих элементов исходных массивов загружаются в регистры **R4** и **R5**, затем выполняется операция сложения

```
ADD    r4,  r4,  r5
```

и результат сложения, находящийся в регистре **R4**, копируется в буфер памяти.

Пример 17. Процедура **mul2_arrays**, исходный текст которой показан в Листинге 4.28, выполняет попарное умножение элементов двух целочисленных массивов с одинаковыми индексами и записывает результат в буфер памяти. Исходные массивы адресуются регистрами **R0** и **R1**, адрес буфера памяти содержится в регистре **R2**, а размер любого из исходных массивов (предполагается, что размеры исходных массивов одинаковы) находится в регистре **R3**.

Листинг 4.28

```
mul2_arrays    SUB    r0,  r0,  #4
               SUB    r1,  r1,  #4
next1          LDR    r4,  [r0,  #4]!
               LDR    r5,  [r1,  #4]!
               MUL    r6,  r4,  r5
               STR    r6,  [r2],  #4
               SUBS   r3,  r3,  #1
               BNE    next1
               BX     lr
```

Данная процедура может быть использована для вычисления скалярного произведения двух векторов, представленных своими координатами. Для этого нужно выполнить операцию сложения для произведений отдельных элементов.

Пример 18. В этом примере выполняется сцепление (конкатенация) двух целочисленных массивов. Предполагается, что оба массива заканчиваются нулями. Процедура **concat** с исходным текстом, представленным в Листинге 4.29,

принимает три параметра: адрес первого массива (регистр **R0**), адрес второго массива (регистр **R1**) и адрес буфера-приемника (регистр **R3**). В процессе выполнения этой операции содержимое массива, адресуемого регистром **R0**, записывается с начального адреса буфера, затем добавляется массив с адресом в **R1**. Размер буфера-приемника должен быть достаточным для хранения элементов обоих массивов.

Листинг 4.29

```
                        AREA  text,  CODE,  READONLY
                        EXPORT  concat
                        ENTRY
concat                  LDR      r3,  [r0],  #4
                        CMP      r3,  #0
                        BEQ      next_array
                        STR      r3,  [r2],  #4
                        B          concat
next_array              LDR      r3,  [r1],  #4
                        CMP      r3,  #0
                        BEQ      exit
                        STR      r3,  [r2],  #4
                        B          next_array
exit                    BX        lr
                        END
```

Процедура завершится, когда будет обнаружен нулевой элемент второго массива, адресуемого регистром **R1**. Далее представлен исходный текст основной программы на C++, которая вызывает данную процедуру (Листинг 4.30).

Листинг 4.30

```
#include  <stdio.h>

extern  void  concat(int  *a1,  int  *b1,  int  *c1);

int  main()
{
    int  a1[5]  =  {3,  -5,  90,  -7,  0};
    int  b1[7]  =  {-13,  -45,  21,  8,  -25,  9,  0};
    int  c1[12];
    int  i1;

    concat(a1,  b1,  c1);
    for  (i1  =  0;  i1  <  sizeof(c1)/4  -  2;  i1++)
        printf("%d  ",  c1[i1]);

    return  0;
}
```

Пример 19. В этом примере показана операция сцепления двух байтовых строк, окнчивающихся нулем – такие строки являются стандартными в C/C++ (null-terminated strings). Процедура **concatstr**, выполняющая эту операцию, принимает три параметра: адрес первой строки в регистре **R0**, адрес второй

строки в регистре **R1** и адрес буфера-приемника в регистре **R2**. Исходный текст процедуры **concatstr** показан в Листинге 4.31.

Листинг 4.31

```
concatstr               LDRB        r3,  [r0],  #1
                        CMP         r3,  #0
                        BEQ         next_str
                        STRB        r3,  [r2],  #1
                        B             concatstr
next_str                LDRB        r3,  [r1],  #1
                        CMP         r3,  #0
                        BEQ         exit1
                        STRB        r3,  [r2],  #1
                        B             next_str
exit                    BX          lr
```

В этой процедуре для загрузки-сохранения отдельных байтов строк используются инструкции **LDR/STR** с суффиксом **B**. При этом инкремент адреса для доступа к следующим элементам обеих строк равен 1, в отличие от 4, когда выполняются операции с целыми числами.

Фрагмент исходного текста приложения на C++, которое вызывает процедуру **concatstr**, показан в Листинге 4.32.

Листинг 4.32

```
. . .
char *s1 = "Hello, ";
char *s2 = "World!";
. . .
char *dst = (char*)malloc(32);
memset(dst, '\0', 32);
. . .
concatstr(s1, s2, dst);
```

Пример 20. Процедура **cutstr** с исходным текстом, показанным в Листинге 4.33, позволяет вырезать из строки байтов подстроку и сохранять ее в буфере памяти. Процедура принимает четыре параметра: адрес исходной строки в регистре **R0**, адрес первого элемента вырезаемой подстроки в регистре **R1**, адрес последнего элемента вырезаемой подстроки в регистре **R2** и адрес буфера памяти, в который выполняется копирование подстроки, в регистре **R3**.

Листинг 4.33

```
cutstr                  ADD         r0,  r0,  r1
                        SUBS        r1,  r2,  r1
                        BEQ          exit
next_byte               LDRB        r4,  [r0],  #1
                        STRB        r4,  [r3],  #1
                        SUBS        r1,  r1,  #1
                        BNE          next_byte
exit                    BX          lr
```

В этой процедуре инструкция

```
ADD    r0,  r0,  r1
```

устанавливает в регистре **R0** адрес первого элемента вырезаемой подстроки — этот адрес будет начальным при выполнении копирования подстроки в буфер.

Для выполнения операции копирования нужно знать количество элементов, которые следует переместить в буфер. Следующая инструкция вычисляет это количество:

```
SUBS    r1,  r2,  r1
```

Инструкция

```
BEQ    exit
```

выполняет выход из процедуры, если копируемая подстрока содержит всего один элемент (в качестве упражнения читателю предлагается самостоятельно модифицировать данный пример так, чтобы одиночный элемент мог быть выделен из строки).

Ниже приводится фрагмент исходного текста вызывающей программы (на C++) для процедуры **cutstr** (Листинг 4.34).

Листинг 4.34

```
. . .
char *s3 = "String 1 + String 2";
char *s3cut = (char*)malloc(strlen(s3));
memset(s3cut, '\0', strlen(s3));
. . .
cutstr(s3, 11, 19, s3cut);
printf("\n%s\n", s3cut);
free(s3cut);
```

Пример 21. Процедура **cmpbytes** (исходный текст показан в Листинге 4.35) выполняет сравнение двух C-строк с завершающим нулем и возвращает количество обнаруженных совпадений. Процедура принимает два параметра: первый, в регистре **R0**, содержит адрес первой строки, а второй (регистр **R1**) — адрес второй строки для сравнения. Процедура возвращает значение в регистре **R0**, как обычно.

Листинг 4.35

```
                    AREA  text,  CODE,  READONLY
                    EXPORT  cmpbytes
                    ENTRY
cmpbytes            MOV      r2,  #0
next_byte           LDRB     r3,  [r0],  #1
                    LDRB     r4,  [r1],  #1
                    CMP      r3,  #0
                    BEQ      exit
                    CMP      r4,  #0
                    BEQ      exit
                    CMP      r3,  r4
                    ADDEQ    r2,  r2,  #1
```

```
                    B           next_byte
exit                MOV       r0,  r2
                    BX        lr
                    END
```

В процессе выполнения операции сравнения регистр **R2** хранит количество совпадений элементов строк. Перед завершением процедуры инструкцией

```
BX        lr
```

содержимое **R2** помещается в регистр **R0** инструкцией

```
MOV     r0,  r2
```

Далее приводится исходный текст приложения на Keil C++ (Листинг 4.36), которое вызывает процедуру **cmpbytes**.

Листинг 4.36

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  int  cmpbytes(char  *s1,  char  *d1);

int  main(void)
{
  init_serial();
  char  *s1  =  "12034567ABCDEFA";
  char  *d1  =  "139455709CCD4FA";

  int  res  =  cmpbytes(s1,  d1);
  printf("  %d  matches  are  found\n",  res);
  while  (1);
  }
```

По завершению выполнения приложение выводит следующее сообщение в терминальное окно отладчика:

```
6  matches  are  found
```

Пример 22. В этом примере показан исходный текст (Листинг 4.37) процедуры **seekint**, которая выполняет поиск элемента с заданным значением в массиве целых чисел и возвращает индекс первого обнаруженного элемента. Если же ни один элемент не был обнаружен, процедура возвращает значение −1.

Процедура принимает три параметра. Регистр **R0** содержит адрес массива целых чисел, в регистре **R1** содержится целое число для поиска, а в регистре **R2** содержится размер массива. Процедура возвращает значение, как обычно, в регистре **R0**.

Листинг 4.37

```
                    AREA  text,CODE,  READONLY
                    EXPORT  seekint
```

```
                        ENTRY
seekint                 STMFD  sp!,  {r4}
                        MOV      r4,  #-1
next                    ADD      r4,  r4,  #1
                        LDR      r3,  [r0],  #4
                        CMP      r1,  r3
                        BEQ      found
                        SUBS      r2,  r2,  #1
                        BEQ      not_found
                        B          next
found                   MOV      r0,  r4
                        LDMFD  sp!,  {r4}
                        BX         lr
not_found               MOV      r0,  #-1
                        LDMFD  sp!,  {r4}
                        BX         lr
                        END
```

Далее приведен исходный текст C++ программы, вызывающей процедуру **seekint** (Листинг 4.38).

Листинг 4.38

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  int  seekint(int  *a1,  int  n,  int  cnt);

int  main()
{
    int  a1[11]  =  {-211,  902,  -57,  45,  -126,  434,  27,  -11,  -9,  775,  -76};
    int  num;
    int  a1size  =  sizeof(a1)/4;

    num  =  27;
    int  res  =  seekint(a1,  num,  a1size);
    if  (res  !=  -1)
        printf("The  element  %d  is  found  at  the  position  %d  from  the
                    beginning\n",  num,  res);
    else  if  (res  ==  -1)
        printf("The  element  %d  is  not  found.\n",  num);
    return  0;
}
```

Приложение выводит следующий результат:

```
The  element  27  is  found  at  the  position  6  from  the  beginning
```

Пример 23. В данном примере процедура **seek_range** выполняет подсчет количества элементов целочисленного массива, попадающих в заданный диапазон. Процедура принимает четыре параметра: адрес исходного массива (регистр **R0**), числовое значение нижней границы диапазона поиска (регистр **R1**), числовое значение верхней границы диапазона поиска (регистр **R2**) и размер массива (регистр **R3**). Процедура возвращает количество обнаруженных элементов в регистре **R0**.

Исходный текст процедуры на языке макро ассемблера Keil показан в Листинге 4.39.

Листинг 4.39

```
                        AREA  text,  CODE,  READONLY
                        EXPORT  seek_range
                        ENTRY
seek_range              STMFD  sp!,  {r4-r5}
                        MOV      r5,  #0
next_ch                 LDR      r4,  [r0],  #4
                        CMP      r4,  r1
                        BGE      next_cmp
dec_cnt                 SUBS     r3,  r3,  #1
                        BEQ      exit
                        B          next_ch
next_cmp                CMP      r4,  r2
                        BLE      found
                        B          dec_cnt
found                   ADD      r5,  r5,  #1
                        B          dec_cnt
exit                    MOV      r0,  r5
                        LDMFD  sp!,  {r4-r5}
                        BX        lr
```

В этой процедуре количество элементов из заданного диапазона хранится в регистре **R5**, которому при вызове процедуры присваивается начальное значение 0 инструкцией

```
MOV      r5,  #0
```

Поиск нужного элемента осуществляется в цикле, который начинается с метки **next_ch**. Инструкции

```
dec_cnt          SUBS    r3,  r3,  #1
                  BEQ      exit
```

управляют циклом, счетчик которого находится в регистре **R3** (его начальное значение равно размеру массива). При достижении нулевого значения в **R3** происходит выход из цикла и завершение процедуры.

Группа инструкций

```
CMP      r4,  r1
BGE      next_cmp
```

выполняет сравнение значения текущего элемента массива, загруженного в регистр **R4**, со значением нижней границы диапазона в регистре **R1**. Если значение элемента превышает указанную границу, осуществляется переход на группу инструкций, выполняющих сравнение по верхней границе (метка **next_cmp**):

```
next_cmp         CMP      r4,  r2
                 BLE      found
```

Если значение текущего элемента массива оказывается меньшим верхнего

граничного значения, то управление передается на метку **found**, где инструкция

```
ADD    r5,  r5,  #1
```

выполняет инкремент количества обнаруженных элементов (регистр **R5**). После этого выполняется проверка условия продолжения цикла (метка **dec_cnt**), и если это условие истинно, то очередная итерация начинается с метки **next_ch**.

По завершению цикла содержимое регистра **R5** копируется в **R0** и процедура завершается. Инструкция **STMFD** сохраняет содержимое регистров **R4** – **R5** в стеке, а инструкция **LDMFD** восстанавливает их содержимое перед завершением процедуры.

Вызов процедуры **seek_range** из приложения, написанного на C++, демонстрируется следующим фрагментом кода (Листинг 4.40).

Листинг 4.40

```
extern "C" int seek_range(int *a1, int start,int end, int cnt);
. . .

int a1[11] = {-21, 902, -57, -100, -126, 434, 27, -11, -9, 775, -76};
int a1size = sizeof(a1)/4;

int seek_num = seek_range(a1, -30, 80, a1size);
printf("Found %d coincidences.\n", seek_num);
```

Для данного примера C++ приложение выводит следующий результат:

```
Found 4 coincidences.
```

Пример 24. Процедура **retstr**, исходный текст которой показан в Листинге 4.41, показывает, как можно вернуть адрес массива байт (строки) в вызывающую программу на C++.

Листинг 4.41

```
                AREA text, CODE, READWRITE
                EXPORT retstr
                ENTRY
retstr          ADR     r0, string1
                BX       lr
string1         DCB     "Hello from the procedure!", 0
                END
```

В этой процедуре строка байт идентифицируется переменной **string1**, которая содержит адрес первого элемента строки. Адрес строки загружается в регистр **R0** псевдоинструкцией

```
ADR     r0, string1
```

и передается в вызывающую программу по завершению процедуры. Обратите внимание на то, что строка **string1** завершается нулем, т. е. соответствует стандарту C/C++. Исходный текст программы на C++, вызывающей процедуру **retstr**, показан в Листинге 4.42.

Листинг 4.42

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  char*  retstr(void);

int  main(void)
{
  init_serial();

  char  *s  =  retstr();
  printf("%s\n",  s);
  while  (1);
}
```

Пример 25. Данный пример показывает, как можно копировать строку байт из процедуры на языке ассемблера в программу на C++. Процедура **cpstr** принимает единственный параметр (регистр **R0**), который является адресом буфера памяти, куда будет скопирована строка. Исходный текст процедуры **cpstr** показан в Листинге 4.43.

Листинг 4.43

```
              AREA  text,  CODE,  READONLY
              EXPORT  cpstr
              ENTRY
cpstr         LDR      r1,  =str1
next          LDRB     r2,  [r1],  #1
              CMP      r2,  #0
              BEQ        exit
              STRB     r2,  [r0],  #1
              B          next
exit          BX        lr
;-----------------------------------
              AREA  myData1,  DATA,  READWRITE
str1          DCB    "The  string  to  copy",  0
              END
```

В этой процедуре строка символов с завершающим нулем, которая должна копироваться в буфер памяти вызывающего приложения, идентифицируется переменной **str1**. При вызове процедуры адрес строки **str1** помещается в регистр **R1** для последующей загрузки инструкцией

```
LDR      r1,  =str1
```

Копирование строки выполняется побайтово в цикле, котрый начинается с метки **next**. Первая инструкция цикла

```
LDRB     r2,  [r1],  #1
```

загружает текущий элемент в регистр **R2**. Следующая инструкция

```
CMP      r2,  #0
```

проверяет, является ли данный байт завершающим нулем. Если да, то происходит выход из цикла по инструкции

```
BEQ      exit
```

Если конец строки не обнаружен, то инструкция **BEQ** пропускается и выполняется копирование байта по текущему адресу в буфере-приемнике вызывающей программы:

```
STRB    r2,  [r0],  #1
```

после чего управление передается в начало цикла на метку **next**.

Данная процедура может быть вызвана из программы на языке C++, как показано в Листинге 4.44.

Листинг 4.44

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  void  cpstr(char  *s1);

int  main(void)
{
  init_serial();

  char  s1[32];
  cpstr(s1);
  printf("%s\n",  s1);

  while(1);
  }
```

В этой программе в качестве принимающего буфера для данных используется массив **s1**. Процедуре **cpstr** в качестве параметра передается адрес этого массива. После возврата из процедуры массив **s1** будет содержать строку байт **str1** (см. Листинг 4.43).

Пример 26. Процедура **cpint** из данного примера демонстрирует операции мультизагрузки и мультикопирования данных, когда с помощью соответствующих инструкций обрабатывается сразу несколько элементов данных. Данной процедуре передается единственный параметр, который является адресом буфера памяти, в котором находятся обрабатываемые данные. Процедура выполняет следующие действия:

1. Загружает несколько целых чисел из последовательных адресов в буфере памяти, адресуемом регистром **R0**, в регистры **R1**–**R4**.
2. Каждое число в регистрах **R1**–**R4** инкрементируется на 3.
3. Данные из регистров **R1**–**R4** записываются обратно в память по тем же адресам, из которых они были ранее загружены.

Исходный текст процедуры **cpint** показан в Листинге 4.45.

Листинг 4.45

```
                AREA  text,  CODE,  READONLY
                EXPORT  cpint
                ENTRY
cpint           STMFD  sp!,  {r0}
                LDMIA  r0!,  {r1-r4}
                MOV      r5,  #3
                ADD      r1,  r1,  r5
                ADD      r2,  r2,  r5
                ADD      r3,  r3,  r5
                ADD      r4,  r4,  r5
                LDMFD  sp!,  {r0}
                STMIA  r0!,  {r1-r4}
                BX       lr
                END
```

В этой процедуре инструкции **LDMIA/STMIA** используются для мультизагрузки/мультикопирования данных. Инструкции **STMFD/LDMFD** нужны для сохранения/восстановления оригинального адреса, загруженного в регистр **R1** перед началом операций мультизагрузки/мультикопирования.

Процедура **cpint** может быть вызвана из программы на C++, как показано в Листинге 4.46.

Листинг 4.46

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  void  cpint(int*  i1);

int  main(void)
{
  init_serial();
  int  x1[4]  =  {-11,  56,  9,  -71};

  cpint(x1);
  for  (int  i1  =  0;  i1  <  4;  i1++)
  {
      printf("  %d  ",  x1[i1]);
  }
  while  (1);
  }
```

Приложение выводит следующее содержимое массива **x1**:

```
-8  59  12  -68
```

Пример 27. Процедура **cpint** из предыдущего примера может быть усовершенствована так, чтобы можно было выполнять мультизагрузку/мультикопирование для произвольного числа данных. Модифицированный вариант предыдущей процедуры, которую мы назовем **cpintr**, позволяет обрабатывать произвольное количество элементов массива.

Исходный текст этой процедуры показан в Листинге 4.47.

Листинг 4.47

```
                AREA  text,  CODE,  READONLY
                EXPORT  cpintr
                ENTRY
cpintr          STMFD  sp!,  {r0}
                LDMIA  r0!,  {r2-r4}
                MOV      r5,  #3
                ADD      r2,  r2,  r5
                ADD      r3,  r3,  r5
                ADD      r4,  r4,  r5
                ;
                LDMFD  sp!,  {r0}
                STMIA  r0!,  {r2-r4}
                SUBS     r1,  r1,  #3
                BGE      cpintr
                BX        lr
                END
```

Процедура **cpintr** принимает два параметра: адрес массива целых чисел для обработки (регистр **R0**) и размер массива (регистр **R1**). Операции загрузки и копирования выполняются в цикле, начиная с метки **cpintr**.

Вначале адрес массива, с которого начинается копирование, сохраняется в стеке инструкцией

```
STMFD  sp!,  {r0}
```

Затем в регистры **R2**–**R4** загружаются данные с последовательных адресов, на которые указывает регистр **R0**:

```
LDMIA  r0!,  {r2-r4}
```

Далее к содержимому каждого из регистров **R2**–**R4** прибавляется 3 с помощью инструкций

```
MOV      r5,  #3
ADD      r2,  r2,  r5
ADD      r3,  r3,  r5
ADD      r4,  r4,  r5
```

Для копирования содержимого этих регистров обратно по их адресам в буфере памяти нужно восстановить оригинальный начальный адрес в буфере **R0**, что выполняется инструкцией

```
LDMFD  sp!,  {r0}
```

Следующая за ней инструкция

```
STMIA  r0!,  {r2-r4}
```

записывает обновленное содержимое регистров **R2**–**R4** по прежним адресам в памяти.

Инструкции

```
SUBS    r1,  r1,  #3
BGE      cpintr
```

выполняют управление циклом. Обратите внимание на то, что содержимое счетчика цикла (регистр **R1**) декрементируется на 3 после каждой итерации.

Для демонстрации работы процедуры **cpintr** можно использовать C++ приложение с исходным текстом, показанным в Листинге 4.48.

Листинг 4.48

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  void  cpintr(int*  i1,  int  isize);

int  main(void)
{
  init_serial();
  int  x1[11]  =  {-11,  56,  9,  -71,  33,  -8,  -44,  0,  111,  -77,
191};
  int  nsize  =  sizeof(x1)  /  4;
  cpintr(x1,  nsize);
  for  (int  i1  =  0;  i1  <  11;  i1++)
  {
      printf("  %d  ",  x1[i1]);
  }
  while  (1);
}
```

Пример 28. В этом примере продемонстрировано, как можно выполнить обмен данными между двумя целыми числами в процедуре на языке ассемблера. Процедура **swpint**, которая выполняет эту операцию, принимает два параметра: адрес переменной, в которой хранится первое число, (регистр **R0**) и адрес переменной, в которой хранится второе число (регистр **R1**). Исходный текст процедуры показан в Листинге 4.49.

Листинг 4.49

```
                AREA  text,  CODE,  READONLY
                EXPORT  swpint
                ENTRY
swpint          STMFD  sp!,  {r0-r1}
                LDR      r0,  [r0]
                LDR      r1,  [r1]
                MOV      r2,  r0
                MOV      r3,  r1
                LDMFD  sp!,  {r0-r1}
                STR      r3,  [r0]
                STR      r2,  [r1]
                BX        lr
                END
```

В этой процедуре инструкция **STMFD** помещает содержимое регистров **R0–R1**

в стек для того, чтобы сохранить адреса переменных для дальнейшего использования. Следующие две инструкции

```
LDR      r0,  [r0]
LDR      r1,  [r1]
```

загружают оба значения в эти же регистры.

Затем инструкции

```
MOV      r2,  r0
MOV      r3,  r1
```

помещают содержимое регистров **R0** и **R1** в регистры **R2** и **R3** соответственно. Оба регистра, **R2** и **R3**, выполняют роль временного хранилища данных в процессе обмена.

Инструкция

```
LDMFD  sp!,  {r0-r1}
```

извлекает оригинальные значения регистров **R0**–**R1** из стека для того, чтобы восстановить адреса, куда будут записаны обновленные данные. Наконец, инструкции

```
STR      r3,  [r0]
STR      r2,  [r1]
```

записывают обновленные значения переменных в соответствующие им адреса в памяти.

Исходный текст вызывающей программы, написанной на языке C++, представлен в Листинге 4.50.

Листинг 4.50

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  void  swpint(int  *num1,  int  *num2);

int  main(void)
{
  init_serial();

  int  num1  =  24177,  num2  =  -369;
  swpint(&num1,  &num2);
  printf("  num1  =  %d,  num2  =  %d\n",  num1,  num2);
  while  (1);
  }
```

Приложение выводит в терминально окно следующий результат:

```
num1  =  -369,  num2  =  24177
```

Пример 29. Процедура **revint** из этого примера меняет порядок расположения элементов целочисленного массива на противоположный, т. е. первый элемент массива становится последним, а последний занимает адрес первого

элемента. Процедура работает с массивом, содержащим 10 целых чисел, – для операций с большими массивами следует увеличить размер временного буфера хранения данных внутри процедуры.

Процедура принимает два параметра: адрес массива целых чисел (регистр **R0**) и его размер (регистр **R1**). Исходный текст процедуры на языке макро ассемблера Keil показан в Листинге 4.51.

Листинг 4.51

```
                    AREA  text,  CODE,  READONLY
                    EXPORT  revint
                    ENTRY
revint              CMP      r1,  #2
                    BLT      exit
                    STMFD  sp!,  {r0-r1}
                    LDR      r2,  =tmp
                    ADD      r2,  r2,  #36
next_int            LDR      r3,  [r0],  #4
                    STR      r3,  [r2],  #-4
                    SUBS     r1,  r1,  #1
                    BNE      next_int
                    LDMFD  sp!,  {r0-r1}
                    ADD      r2,  r2,  #4
write_next          LDR      r3,  [r2],  #4
                    STR      r3,  [r0],  #4
                    SUBS     r1,  r1,  #1
                    BNE      write_next
exit                BX        lr
                    AREA    myData,  DATA
tmp                 SPACE  40
                    END
```

Первые две инструкции в начале кода процедуры

```
CMP      r1,  #2
BLT      exit
```

выполняют проверку размера массива (для данной процедуры массив должен состоять как минимум из двух элементов, иначе теряется смысл выполнения данной операции). Если размер массива оказывается меньше 2, происходит немедленный выход из процедуры. Следующая инструкция

```
STMFD  sp!,  {r0-r1}
```

сохраняет содержимое регистров **R0** и **R1** в стеке для последующего использования.

Инструкция

```
LDR      r2,  =tmp
```

загружает адрес буфера **tmp** в регистр **R2**. Этот буфер будет содержать элементы массива, загруженные в обратном порядке. Размер буфера, равный 40 байтам, расчитан на хранение максимум 10 целых чисел, поэтому для работы с массивами больших размеров следует откорректировать размер резервируемой памяти для переменной **tmp**.

После того, как все элементы исходного массива будут скопированы в **tmp**, процедура выполнит копирование содержимого буфера **tmp** обратно в область памяти, занимаемую исходным массивом

Инструкция

```
ADD      r2,  r2,  #36
```

устанавливает указатель адреса во временном буфере на начало последнего элемента для сохранения элементов в обратном порядке.

В следующем цикле, который начинается с метки **next_int**, выполняется копирование элементов исходного массива во временный буфер **tmp**:

```
next_int              LDR      r3,  [r0],  #4
                      STR      r3,  [r2],  #-4
                      SUBS     r1,  r1,  #1
                      BNE      next_int
```

Обратите внимание на то, что элементы исходного массива загружаются по увеличивающимся адресам, в то время как во временном буфере эти же элементы сохраняются по уменьшающимся адресам. По завешению этого цикла временный буфер **tmp** будет содержать элементы исходного массива, записанные в обратном порядке. Управление циклом выполняют инструкции **SUBS** и **BNE**.

По завершению цикла записи во временный буфер инструкция

```
LDMFD  sp!,  {r0-r1}
```

восстанавливает значения параметров из стека для подготовки цикла записи данных из временного буфера обратно в исходный массив, адресуемый регистром **R0**. Кроме того, нам нужно установить указатель адреса в буфере **tmp** на позицию первого элемента, что выполняется инструкцией

```
ADD      r2,  r2,  #4
```

Копирование элементов временного буфера **tmp** обратно в исходный массив выполняется в цикле **write_next**:

```
write_next            LDR      r3,  [r2],  #4
                      STR      r3,  [r0],  #4
                      SUBS     r1,  r1,  #1
                      BNE      write_next
```

Смысл инструкций этого цикла, думаю, понятен. Управление циклом, как обычно, осуществляется инструкциями **SUBS** и **BNE**.

Исходный текст на C++ вызывающей программы для процедуры **revint** показан в Листинге 4.52.

Листинг 4.52

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
```

```
extern "C" void revint(int *a1, int cnt);

int main(void)
{
    init_serial();

    int a1[10] = {19, -56, -45, 34, -7, 11, 771, 0, 338, -92};
    int cnt = sizeof(a1)/4;
    revint(a1, cnt);
    for (int i1 = 0; i1 < cnt; i1++)
        printf(" %d ", a1[i1]);

    while (1);
  }
```

Приложение выводит последовательность элементов массива, как показано ниже:

```
-92  338  0  771  11  -7  34  -45  -56  19
```

Пример 30. В этом примере демонстрируется использование стека при вызове процедур. Напомню, что первые четыре параметра вызываемой процедуры помещаются в регистры **R0**—**R4**, все остальные должны располагаться в области адресов памяти, выделенной под стек. Процедура **add7** показывает, как использовать данные, помещенные вызывающей программой в стек.

Вначале проанализируем исходный текст вызывающей программы на C++ (Листинг 4.53).

Листинг 4.53

```
#include <stdio.h>
#include <LPC214X.H>

extern "C" void init_serial(void);
extern "C" int add7(int i1, int i2, int i3, int i4, int i5, int i6, int i7);

int main(void)
{
  struct myData {
      int i1;
      int i2;
      int i3;
      int i4;
      int i5;
      int i6;
      int i7;
  } dat;

  dat.i6 = -111;
  dat.i7 = -100;

  init_serial();

  int res = add7(0, 0, 0, 0, 0, dat.i6, dat.i7);
  printf("    Add7 result = %d\n", res);
```

```
  while  (1);
}
```

В этой программе для нахождения суммы всех элементов, расположенных в полях структуры **dat** типа **myData**, используется внешняя процедура **add7**, написанная на языке ассемблера. Данная процедура принимает семь параметров, первые пять из которых содержат нулевые значения, а шестой и седьмой параметры солержат значения −111 и −100 соответственно. Исходный текст процедуры **add7** показан в Листинге 4.54.

Листинг 4.54

```
                AREA  text,  CODE,  READONLY
                EXPORT  add7
                ENTRY
add7            LDR    r0,  [sp,  #4]
                LDR    r1,  [sp,  #8]
                ADD    r0,  r0,  r1
                BX      lr
                END
```

Поскольку пятый параметр процедуры адресуется текущим значением регистра-указателя стека (**R13**, **sp**), то значение −111 будет находиться по адресу **sp+4**, а значение −100 – по адресу **sp+8**. Оба значения загружаются в регистры **R0** и **R1** для выполнения операции сложения. Процедура возвращает результат в регистре **R0**, как обычно.

4.3. Использование встроенного ассемблера языка C++ в приложениях Keil

В этом разделе мы рассмотрим встроенный ассемблер (Embedded Asembler) языка C++ среды программирования Keil – чрезвычайно мощное и удобное средство для оптимизации приложений, написанных на языке C++. Ассемблерная процедура на встроенном ассемблере может быть включена непосредственно в исходный текст программы на C++, как обычная процедура.

Объявление процедуры на встроенном ассемблере должно начинаться с ключевого слова **__asm** (здесь идет двойное подчеркивание). Параметры такой процедуре передаются с использованием мнемоники, принятой при вызове обычных процедур, написанных на C++. Для процедур на встроенном ассемблере должны выполняться соглашения о передаче параметров и возвращаемых значениях, принятые в C++. Инструкции макро ассемблера Keil должны помещаться в процедуру между открывающей и закрывающей фигурными скобками. К недостатку встроенного ассемблера следует отнести существенные ограничения по обработке данных, определенных внутри самой ассемблерной процедуры. От этого недостатка свободны процедуры, разрабатываемые на макро ассемблере Keil.

Рассмотрим несколько примеров использования процедур, разработанных с использованием встроенного ассемблера.

Пример 1. Здесь показан пример вычитания двух целых чисел, которые передаются в качестве параметров процедуре **sub2**, написанной с использованием встроенного ассемблера. Исходный текст программы показан в Листинге 4.55.

Листинг 4.55

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

__asm  int  sub2(int  i1,  int  i2)
{
  SUB  r0,  r0,  r1
  BX    lr
}

int  main(void)
{
  init_serial();
  int  a1  =  44,  a2  =  75;

  int  res  =  sub2(a1,  a2);
  printf("  Sub2  result  =  %d\n",  res);

  while  (1);
  }
```

В этой программе процедура **sub2** принимает два параметра, **i1**и **i2**, целого типа. Поскольку для встроенного ассемблера выполняются стандартные соглашения о вызовах, то параметр **i1** передается в регистре **R0**, а **i2** передается в **R1**. Процедура возвращает результат вычитания в регистре **R0**, как обычно.

Пример 2. Здесь показано, как в одной процедуре на встроенном ассемблере можно оперировать несколькими внутренними процедурами. Процедура **add_sub2** содержит две отдельные процедуры, **add2** и **sub2**, выполняющие соответственно операции сложения и вычитания двух целых чисел. Процедура **add_sub2** принимает три параметра: первые два являются целыми числами, над которыми должна выполняться одна из операций сложения/вычитания, а третий параметр определяет тип операции (0 указывает на необходимость выполнения сложения, а 1 предписывает выполнить вычитание).

Исходный текст программы на C++ показан в Листинге 4.56.

Листинг 4.56

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

__asm  int  add_sub2(int  i1,  int  i2,  int  func)
{
```

```
            STMFD    sp!,  {lr}
            CMP        r2,  #0
            BLEQ      add2
            CMP        r2,  #1
            BLEQ      sub2
            B            exit
add2    ADD        r0,  r0,  r1
            BX          lr
sub2    SUB        r0,  r0,  r1
            BX          lr
exit    LDMFD    sp!,  {lr}
            BX    lr
}

int  main(void)
{
  init_serial();
  int  a1  =  44,  a2  =  75;

  int  res  =  add_sub2(a1,  a2,  0);
  printf("  Add2  result  =  %d\n",  res);

  res  =  add_sub2(a1,  a2,  1);
  printf("  Sub2  result  =  %d\n",  res);

  while  (1);
}
```

Здесь в процедуре **add_sub2** выполняется анализ типа операции, переданной в третьем параметре (регистр **R1**). Следующие две инструкции

```
CMP        r2,  #0
BLEQ      add2
```

позволяют передать управление процедуре **add2** в случае, если регистр **R2** содержит нулевое значение. Если условие не выполняется, то следующая пара инструкций

```
CMP        r2,  #1
BLEQ      sub2
```

проверяет равенство третьего параметра единице. Если **R2** содержит 1, то управление передается процедуре **sub2**.

В случае, если ни одно из условий не выполнено, процедура завершается. Инструкции **STMFD/LDMFD** обеспечивают передачу в регистр связи **LR** (**R14**) корректного адреса возврата в основную программу. Это связано с тем, что при выходе из любой из процедур, **add2** или **sub2**, вызывается инструкция

```
BX          lr
```

которая выполняет передачу управления по адресу внутри процедуры **add_sub2**.

Пример 3. Это более сложный пример, в котором показано. как процедура на встроенном ассемблере оперирует с элементами структуры, определенной в основной программе на C++. Процедура **cpstruct** копирует содержимое

одного из полей структуры в буфер памяти, определенный внутри этой же структуры. Для манипуляции с полями структуры процедуре **cpstruct** передается адрес всей структуры, как единственный параметр.

Исходный текст приложения на C++ показан в Листинге 4.57.

Листинг 4.57

```
#include  <stdio.h>
#include  <string.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

__asm  void  cpstruct(char  *struc)
{
            STMFD    sp!,  {r1-r5}
            LDMIA      r0,  {r1-r3}
next        LDRB       r4,  [r1],  #1
            CMP        r4,  #'  '
            ADDNE      r4,  r4,  #2
            STRB       r4,  [r2],  #1
            SUBS       r3,  r3,  #1
            BNE        next
            LDMFD    sp!,  {r1-r5}
            BX          lr
}

int  main(void)
{
  struct    setBytes  {
    char  *src;              /*  Source  buffer  start  address  */
    char  *dst;              /*  Dest  buffer  start  address  */
    int  N;                  /*  Number  of  bytes  to  copy  */
    }  setBytes1;

  struct  setBytes  *psetBytes;
  psetBytes  =  &setBytes1;

  init_serial();
  char  s1[]  =  "0  1  2  3  4  5  6  7  A  B  C  D  E  F  G";

  char  cbuf[30];
  memset(cbuf,  '\0',  sizeof(cbuf));

  psetBytes->src  =  s1;
  psetBytes->dst  =  cbuf;
  psetBytes->N  =  strlen(s1);

  cpstruct((char*)psetBytes);
  printf("      s1  content:  %s\n",  s1);
  printf("      cbuf  content:  %s\n",  cbuf);

  while  (1);
  }
```

В нашем приложении определена структура **setBytes1** типа **setBytes**, содержащая три поля.

Поле **src** содержит указатель на буфер памяти, откуда данные будут копироваться, поле **dst** содержит адрес буфера, куда копируются данные, и поле **N** хранит количество копируемых байтов.

Для выполнения операций со структурой определим указатель **psetBytes** на данную структуру следующим образом:

```
struct  setBytes  *psetBytes;
psetBytes  =  &setBytes1;
```

Указатель **psetBytes** содержит адрес первого элемента структуры **setBytes1**.

Кроме этого, в нашей программе определены некоторые вспомогательные переменные, такие как **s1** и **cbuf**. Переменная **s1** является указателем на строку байтов, которая будет скопирована в буфер, на который указывает переменная **cbuf**. Следующая группа операторов инициализирует поля структуры, определенной указателем **psetBytes**:

```
psetBytes->src  =  s1;
psetBytes->dst  =  cbuf;
psetBytes->N  =  strlen(s1);
```

Здесь в поле **N** записывается размер символьного буфера, который буде копироваться. Процедура **cpstruct**, как уже было сказано, принимает единственный параметр, который является адресом структуры. Тип указателя на структуру про этом приводится к типу **char***:

```
cpstruct((char*)psetBytes);
```

Первая инструкция процедуры

```
STMFD    sp!,  {r1-r5}
```

сохраняет содержимое регистров **R1**–**R5** в стеке на тот случай, если вызывающая программа их использует, и их содержимое не должно быть разрушено.

Инструкция

```
LDMIA    r0,  {r1-r3}
```

загружает регистры **R1**–**R3** соответственно содержимым полей **src**, **dst** и **N** структуры **psetBytes**. Таким образом, после выполнения этой инструкции регистр **R1** будет содержать адрес буфера-источника данных **src**, регистр **R2** будет содержать адрем буфера-приемника данных **dst** и, наконец, регистр **R3** будет содержать количество элементов **N**, равное размеру буфера-источника.

Копирование элеметов из **src** в **dst** выполняется вместе с их преобразованием. Если символ в источнике не является пробелом, то его шестнадцатеричный код увеличивается на 2 и новое значение копируется в буфер-приемник **dst**. Если в буфере **src** обнаруживается символ пробела, то элемент копи-

руется в буфер **dst** без преобразования. Этот алгоритм реализован в цикле **next** процедуры **cpstruct**:

```
next        LDRB        r4,  [r1],  #1
            CMP         r4,  #'  '
            ADDNE       r4,  r4,  #2
            STRB        r4,  [r2],  #1
            SUBS        r3,  r3,  #1
            BNE         next
```

Здесь инструкция **CMP** определяет наличие пробела в потоке байт, а инструкция **ADDNE** прибавляет значение 2 к коду элементов, отличных от пробела. Управление циклом **next** выполняют инструкции **SUBS** и **BNE**.

Работающее приложение выводит следующие строки:

```
  s1  content:  0  1  2  3  4  5  6  7  A  B  C  D  E  F  G
cbuf  content:  2  3  4  5  6  7  8  9  C  D  E  F  G  H  I
```

Пример 4. Этот пример может показаться довольно сложным, но использование программы, показанной здесь, позволит вызывать C++ процедуру из процедуры на встроенном ассемблере, что расширяет возможности реализации сложных программных алгоритмов. Процедура **callc**, написанная на встроенном ассемблере, вызывает для выполнения другую процедуру, **mul2**, написанную на C++. Процедура **mul2** выполняет умножение двух целых чисел, являющихся ее параметрами, и возвращает результат умножения в вызывающую процедуру.

Исходный текст C++ приложения показан в Листинге 4.58.

Листинг 4.58

```
#include  <stdio.h>
#include  <stdlib.h>

#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

int  (*pf)(int,  int);

__asm  int  callc(int  (*p)(int,  int),  int  i1,  int  i2)
{
  STMFD  sp!,  {lr}
  MOV      r3,  r0
  MOV      r0,  r1
  MOV      r1,  r2
  BX        r3
  LDMFD  sp!,  {pc}
}

int  mul2(int  i1,  int  i2)
{
  return  i1*i2;
}
```

```
int  main(void)
{
  init_serial();
  int  i1  =  68,  i2  =  -12;
  pf  =  mul2;
  int  res  =  callc(pf,  i1,  i2);

  printf(" The  mul2  result  =  %d  ",  res);
  while  (1);
  }
```

Для вызова внешней процедуры из **callc** используется следующий программный метод. Для вызываемой процедуры создается указатель, который может быть использован как обычный параметр для другой процедуры.

В нашем случае, создается указатель (глобальный!) **pf**:

```
int  (*pf)(int,  int);
```

В основной программе данному указателю присваивается имя процедуры **mul2** (это эквивалентно присвоению адреса первой выполняемой инструкции):

```
pf  =  mul2;
```

Процедура **callc** принимает три параметра: указатель на процедуру (регистр **R0**) и два целых числа (в регистрах **R1** и **R2**). Процедура **callc** возвращает целочисленное значение, которое (в данном случае) будет являться результатом выполнения процедуры **mul2**.

Следующий оператор вызывает процедуру **callc**:

```
int  res  =  callc(pf,  i1,  i2);
```

Проанализируем теперь смысл инструкций процедуры **callc**.

Инструкция

```
STMFD  sp!,  {lr}
```

сохраняет в стеке корректный адрес возврата в основную программу (который будет разрушен при вызове процедуры **mul2**).

Инструкция

```
MOV      r3,  r0
```

копирует содержимое регистра **R0** (он содержит адрес процедуры **mul2**) в **R3**. Смысл этой операции состоит в том, что процедура **callc** должна будет вызвать процедуру **mul2** и передать ей параметры (два целых числа) в регистрах **R0** и **R1**. Для вызова самой процедуры **mul2** сначала корректируется размещение параметров с помощью инструкций

```
MOV      r0,  r1
MOV      r1,  r2
```

а затем выполняется косвенный вызов самой процедуры через адрес в регистре **R3**:

```
BX    r3
```

Последняя инструкция процедуры

```
LDMFD  sp!,  {pc}
```

восстанавливает корректный адрес возрата в основную программу путем непосредственной загрузки счетчика инструкций программы из стека.

Что же касается результата умножения, выполненного процедурой **mul2**, то он, как обычно, помещается в регистр **R0** по завершению **mul2**. Поскольку процедура **callc** должна возвращать значение, то содержимое регистра **R0** напрямую передается в основную программу.

Этим примером завершается глава 4, материал которой может послужить основой для написания эффективных приложений, в которых будут оптимально сочетаться языки программирования C++ и ассемблер.

ГЛАВА 5

ОТЛАДКА ПРОГРАММНОГО КОДА МИКРОКОНТРОЛЛЕРОВ ARM

После написания программного кода следующим, не менее важным этапом, является отладка. Когда говорят об отладке или, более точно, о процессе отладки, то обычно подразумевают комплекс операций, которые могут потребоваться для получения желаемых рабочих характеристик приложения. Рассмотрим более подробно, в порядке следования, все этапы отладки приложения. Обычно процесс отладки начинается с поиска ошибок в исходном тексте программы, которые препятствуют успешной компиляции и сборке приложения. Проанализируем, какие типичные ситуации могут возникать на этом этапе.

5.1. Компиляция исходных текстов программы

На этапе компиляции исходные тексты обрабатываются компилятором C/C++, который генерирует перемещаемые объектные модули. Многие ошибки, проявляющиеся на этапе компиляции, вызваны, как правило, неточностями в наборе исходного текста в редакторе. Типичными ошибками являются пропущенная точка с запятой в конце или отстутствующие (или недостающие) спецификаторы в комментариях (// и/или /*).

Еще одной характерной ошибкой является несоотвествие количества открывающих и закрывающих круглых скобок в операторах или фигурных скобок в операторах цикла (`while`, `for` и т. д.).

При отсутствии ошибок в исходных текстах программ и успешной компоновке объектных модулей, входящих в состав проекта, в окно **Output** среды программирования Keil будет выведена примерно такая последовательность:

```
Build target 'Target 1'
assembling Startup.s...
compiling Retarget.c...
compiling Serial.c...
compiling main.cpp...
linking...
Program Size: Code=1692 RO-data=32 RW-data=4 ZI-data=1260
"example1.axf" - 0 Error(s), 0 Warning(s).
```

Как видно из этой последовательности, вначале компилируются исходные тексты всех программ, входящих в состав проекта. Затем, если компиляция прошла успешно, компоновщик выполняет связывание различных объектных

модулей в исполнительный файл. К сожалению, крайне редко программа успешно компилируется с первого раза. Рассмотрим несколько типичных ошибок, проявляющихся в процессе компиляции и сборки приложения. Для этого разработаем простое приложение на Keil C++, в исходный текст которого преднамеренно внесем ошибки и проанализируем методику их исправления.

Здесь нужно сделать одно важное замечание. Вспомним, что этапы компиляции исходных текстов и сборка приложения, вообще говоря, не связаны друг с другом. Основная задача компилятора – анализ (parsing) исходных текстов программы и идентификация ошибок. Если компиляция проходит успешно, то на выходе получаем объектный файл, который в дальнейшем может быть использован для компоновки в любом приложении. Компилятор ничего не "знает" о том, как происходит взаимодействие программного кода и обмен данными между различными объектными модулями, это уже задача компоновщика (линкера). Компоновщик выполняет связывание различных объектных модулей и библиотек в исполняемый файл. На этом этапе также могут обнаруживаться ощибки, хотя их идентификация в некоторых случаях может потребовать довольно много времени.

Предположим, что наша программа включает файл main.cpp с исходным текстом с ошибками, показанным далее (Листинг 5.1):

Листинг 5.1

```
1.    #include  <stdio.h>
2.
3.
4.    extern  "C"  void  init_serial(void);
5.    double  quad(double  d1);
6.
7.    {
8.      return  d1  *  d1;
9.    }
10.  int  main(void)
11.  {
12.    init_serial();
13.    double  res;
14.
15.    for  (int  i1  =  1;  i1  <  10;  i++)
16.    {
17.      res  =  guad(i1);
18.    printf("  %5.2f  ",  res)
19.
20.    while  (1);
21.    }
```

Если запустить проект с таким исходным текстом на компиляцию, то получим следующий результат:

```
Build  target  'Target  1'
assembling  Startup.s...
compiling  Retarget.c...
compiling  Serial.c...
```

```
compiling main.cpp...
main.cpp(7): error:  #169: expected a declaration
main.cpp(15): error:  #20: identifier "i" is undefined
main.cpp(17): error:  #20: identifier "guad" is undefined
main.cpp(20): error:  #65: expected a ";"
main.cpp(21): error: At end of source:  #67: expected a "}"
Target not created
```

Из этого отчета видно, что компилятлор обнаружил несколько ошибок в процессе анализа исходного текста программы **main**. Каждая ошибка идентифицируется своим кодом и текстовым описанием. Кроме этого, указывается номер строки исходного текста программы, где была обнаружена ошибка. Так, в строке 7 компилятор обнаружил блок операторов внутри открывающей и закрывающей скобок. Данный блок не идентифицирован ни как управляющий оператор цикла (**for**, **while**, **do**…**while**), ни как оператор **switch**…**case** или **if**. Телом функции этот блок также не является, поскольку отсутствует идентификатор функции перед открывающей скобкой. Очевидно, как следует из исходного текста, данный блок идентифицируется как функция **quad**. Для устранения ошибки достаточно убрать разделитель (точку с запятой) в строке 5.

Ошибка в строке 15 одна из типичных ошибок при быстром наборе исходного текста. Здесь идентификатор переменной **i1** цикла for ощибочно записан как **i** в операторе инкремента. Добавление единицы после **i** устраняет эту ошибку. После устранения этой ошибки можно еще раз перекомпилировать исходный текст программы **main**, несмотря на имеющиеся ошибки. Смысл подобной операции заключается в том, что поэтапное устранение ошибок, особенно если таковых много, позволит систематизировать процесс коррекции и избежать путаницы. Еще один, даже более существенный момент – во многих случаях последующие ошибки в исходном тексте могут являться следствием предыдущих. Часто случается так, что при устранении одной или нескольких ошибок остальные исчезают.

Здесь нужно сделать одно важное замечание: коррекцию ошибок нужно выполнять, начиная с самой первой строки исходного текста, т. е. в том порядке, в каком их обнаружил компилятор. Если эту последовательность нарушить, к существующим проблемам могут добавиться новые. Итак, для нашего примера выполним повторную компиляцию. В результате получим следующий отчет:

```
Build target 'Target 1'
assembling Startup.s...
compiling Retarget.c...
compiling Serial.c...
compiling main.cpp...
main.cpp(17): error:  #20: identifier "guad" is undefined
main.cpp(20): error:  #65: expected a ";"
main.cpp(22): error: At end of source:  #67: expected a "}"
Target not created
```

Как видно из отчета, ошибки были исправлены успешно. Ошибка в строке 17 исходного текста вызвана опечаткой – из исходного текста программы очевидно, что здесь должен находиться идентификатор функции **quad** вместо

guad. В строке 20 забыли разделитель в конце оператора, а в конце исходного текста (в строке 22) нужно добавить закрывающую фигурную скобку.

Повторная компиляция исходного текста показывает, что явных ошибок в исходном тексте программы компилятор не обнаружил:

```
Build target 'Target 1'
assembling Startup.s...
compiling Retarget.c...
compiling Serial.c...
compiling main.cpp...
linking...
Program Size: Code=6732 RO-data=208 RW-data=4 ZI-data=1260
"example1.axf" - 0 Error(s), 0 Warning(s).
```

После успешной сборки приложения можем запустить наш двоичный ELF-файл (example1.axf) на прогон в симуляторе (Рис. 5.1):

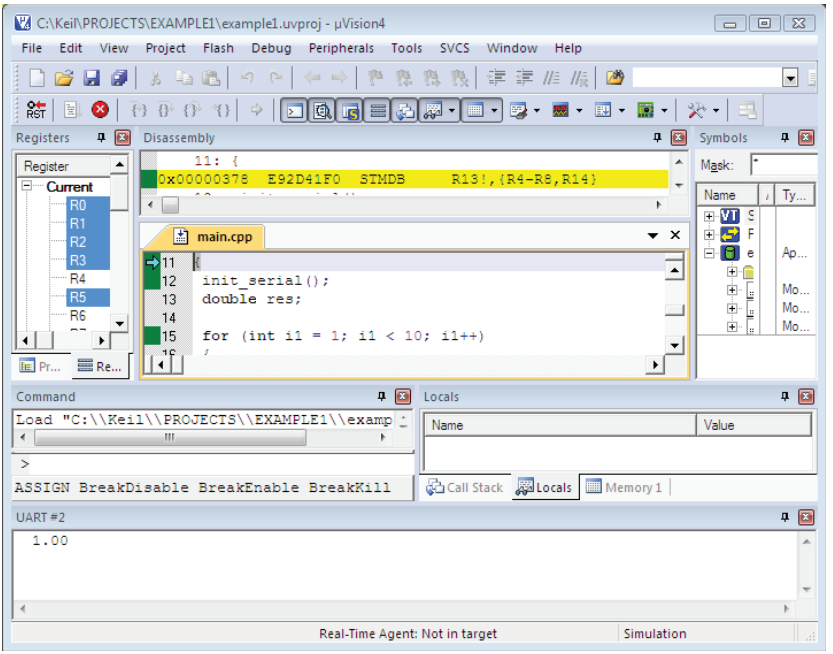

Рис. 5.1.

Полученный в результате выполнения программы результат (1.00 в терминальном окне UART2) несколько обескураживает, поскольку программа должна была вывести последовательность квадратов чисел от 1 до 10. Вернемся к исходному тексту программы. Поскольку вывод значений выполняется в цикле **for**, источник логической ошибки следует искать именно здесь.

После добавления закрывающей фигурной скобки после функции **printf()** и удаления ставшей лишней закрывающей скобки после оператора **while** пе-

рекомпилируем проект и запустим на выполнение. Теперь программа выводит ожидаемый результаты (Рис. 5.2):

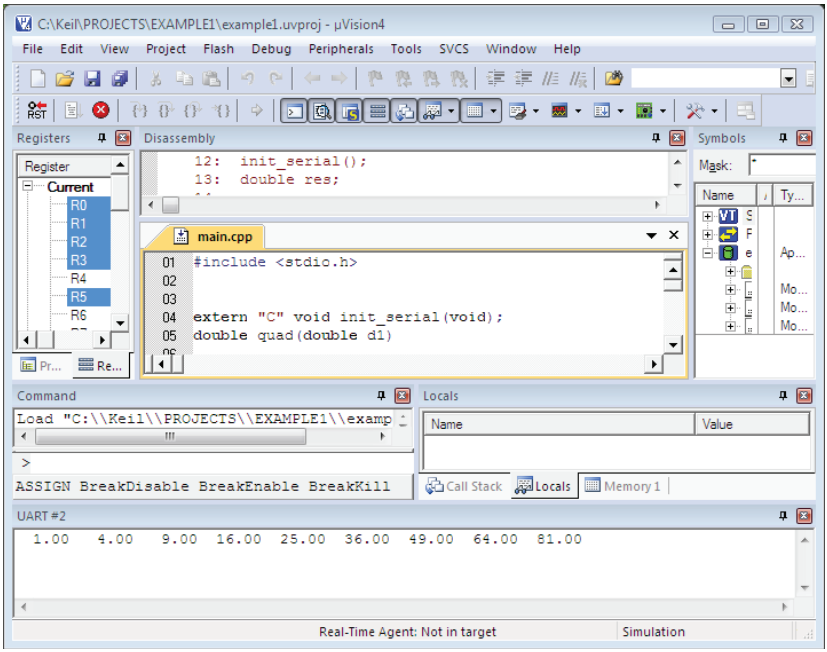

Рис. 5.2.

Этот пример иллюстрирует, помимо всего прочего, еще один важный аспект отладки C++ приложения, а именно, поиск и устранение логических ошибок в самой программе. В относительно простых приложениях, как данный пример, логические ошибки обнаруживаются без труда, но в более сложных программах поиск неправильно работающих участков программы может занять существенный промежуток времени. К счастью, среда программирования Keil (впрочем как и IAR) имеет встроенный отладчик, позволяющий обнаружить и откорректировать неправильно работающие участки программного кода.

5.2. Компоновка объектных модулей и генерация исполняемого файла программы

После успешного выполнения компиляции файлов исходных текстов создаются перемещаемые (relocatable) объектные файлы. Перемещаемые объектные файлы содержат данные/код, которые не имеют привязок к конкретным физическим адресам в памяти, поэтому такие файлы, хотя и содержат откомпилированный двоичный код, не могут быть выполнены. Окончательная привязка всех адресов во всех объектных файлах приложения, а также разрешение символических

ссылок на данные/код выполняется компоновщиком. Результатом работы компоновщика является двоичный исполняемый файл, с фиксированными (абсолютными) адресами загрузки секций кода/данных. Ошибки, возникающие в процессе компоновки объектных модулей и библиотек, в основном связаны с некоррекной информацией (или отсутствием таковой) о размещении участков программного кода и/или данных, вследствие чего невозможно создать ссылки на процедуры /данные, находящиеся в различных объектных модулях. Ошибки компоновки встречаются довольно часто в программах, содержащих несколько отдельных объектных модулей и/или библиотек функций.

Рассмотрим пример типичной ошибки, возникающей при компоновке двух перемещаемых объектных модулей в один исполняемый ELF-файл. Предположим, что в проект включены два файла с исходными текстами основной программы (main.cpp) и процедуры на ассемблере (proc.s).

Исходные тексты обоих файлов показаны в Листингах 5.2 и 5.3 соответственно.

Листинг 5.2

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  int  add2(int  i1,  int  i2);

int  main(void)
{
  init_serial();
  int  i1  =  90,  i2  =  -87;

  int  res  =  add2(i1,  i2);
  printf("  Add2  result:  %d\n",  res);

  while  (1);
}
```

Листинг 5.3

```
            AREA  text,  CODE,  READONLY
            ENTRY
add2        ADD    r0,  r0,  r1
            BX      lr
            END
```

Приложение выполняет сложение двух целых чисел, определенных в основной программе на C++ (см. Листинг 5.2). Саму операцию сложения выполняет вспомогательная процедура **add2**, которая определена в другом модуле (см. Листинг 5.3). В соответствии с соглашением о вызовах (calling conventions) процедура принимает два параметра в регистрах **r0** и **r1** и возвращает результат в регистре **r0**.

Для вызова процедуры из другого модуля в основной программе указывается спецификатор **extern** перед объявлением процедуры **add2**. Если запустить про-

ект (назовем его **example2**) на компиляцию, то получим следующий отчет:

```
Build target 'Target 1'
assembling Startup.s...
compiling Retarget.c...
compiling Serial.c...
compiling main.cpp...
assembling proc.s...
linking...
example2.axf: Error: L6218E: Undefined symbol add2 (referred from
main.o).
Target not created
```

Для лучшего понимания этого отчета рассмотрим простую схему создания исполняемого ELF-файла example2.axf (Рис. 5.3):

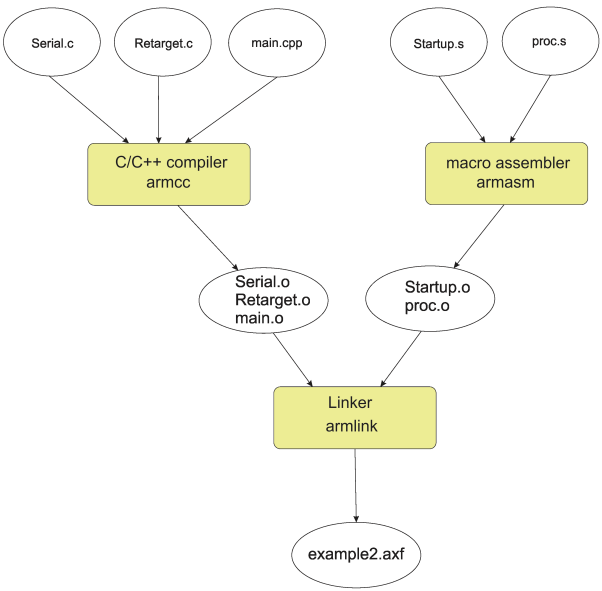

Рис. 5.3.

На первом этапе компилятор **armcc** создает перемещаемые объектные файлы из всех файлов с исходными C/C++ текстами. То же самое выполняет макро ассемблер **armasm** для файлов с исходными текстами на языке ассемблера (расширение .s). После успешной компиляции мы должны получить все пять объектных модулей, включая main.o и proc.o. Как видно из отчета, компиляция была выполнена успешно.

Далее все объектные файлы из нашего проекта должны компоноваться в один исполняемый двоичный файл, которому, в случае успешной компоновки, будет присвоено имя проекта и расширение .axf. Судя по отчету, компоновщик **armlink** не сумел сгенерировать исполняемый двоичный образ из имеющих-

ся объектных модулей. Диагностическое сообщение указывает на неизвестную ссылку **add2** в объектном модуле main.o:

```
example2.axf: Error: L6218E: Undefined symbol add2 (referred from main.o).
```

Вспомним, что в модуле **main** была объявлена внешняя процедура **add2**. Эта процедура присутствует в модуле **proc**, но, тем не менее, компоновщик ее не "увидел". Причина кроется в том, что для получения доступа к какому-либо ресурсу (данные или код), этот ресурс должен быть объявлен как внешний в том модуле, который пытается получить к нему доступ (а), и должен быть указан как общий в том модуле, где он находится (б). Условие (а) в нашем случае выполнено, в то время как условие (б) отсутствует. Поскольку наша процедура находится в ассемблерном модуле **proc**, то для объявления ее доступной из других модулей нужно использовать директиву **EXPORT**, после которой указать имя процедуры:

```
EXPORT  add2
```

Данная строка должна располагаться перед директивой **ENTRY**, указывающей на точку входа в процедуру. Следовательно, нам нужно модифицировать исходный текст файла proc.s, как показано в Листинге 5.4:

Листинг 5.4

```
            AREA  text,  CODE,  READONLY
            EXPORT  add2
            ENTRY
add2        ADD    r0,  r0,  r1
            BX      lr
            END
```

Сохраним изменения и еще раз перекомпилируем исходные тексты. На этот раз генерация исполняемого файла example2.axf выполняется успешно.

Только что рассмотренный пример иллюстрирует одну из наиболее часто встречающихся ошибок в процессе компоновки приложения – неразрешенные ссылки на код/данные при генерации приложения из нескольких модулей.

Рассмотрим еще один пример, в котором процедура умножения двух целых чисел (назовем ее **mul2**), находящаяся во внешнем C++ модуле, использует в качестве параметров данные, находящиеся в основной программе.

Создадим новый пример с именем **example3** и включим в него два файла с исходными текстами, main.cpp и proc.cpp, при этом процедура **mul2** будет находиться в модуле proc.cpp. Исходные тексты файлов main.cpp и proc.cpp показаны в Листингах 5.5.и 5.6 соответственно.

Листинг 5.5

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  int  mul2(void);
int  i1,  i2;
```

```
int  main(void)
{
  init_serial();
  i1  =  -99;
  i2  =  35;

  int  res  =  mul2();
  printf("  Mul2  result:  %d\n",  res);

  while  (1);
}
```

Листинг 5.6

```
extern  int  i1,  i2;

int  mul2(void)
{
    return  i1*i2;
}
```

В основной программе (Листинг 5.5) мы объявили две целочисленные переменные **i1** и **i2** как глобальные; они находятся в разделе деклараций, перед точкой входа в основную программу **main()**, следовательно доступ к ним будет возможен из любого объектного модуля данного приложения. В модуле **proc** эти же переменные объявлены как внешние по отношениею к данному модулю (Листинг 5.6). Что же касается процедуры **mul2**, то по отношению к основной программе она является внешней, поэтому объявлена с директивой **extern**. Используя эту информацию, компоновщик сможет разрешить символические ссылки и правильно разместить секции данных/кода в исполняемом файле.

Следует упомянуть еще один важный аспект, который необходимо учитывать при сборке приложения, состоящего из нескольких объектных файлов. Этот аспект касается спецификатора "C", который указывает на то, что данная процедура при компоновке должна соответствовать требованиям языка C. Этот спецификатор обязательно должен быть указан после спецификатора **extern**, если внешняя функция находится, например, в ассемблерном модуле (файл .s) или в файле .c.

Рассмотрим небольшой пример, который представляет собой небольшую модификацию предыдущего, и в котором показано применение спецификатора "C". Предположим, что кроме операции умножения двух целых чисел выполняемой функцией **mul2**, приложение выполняет операцию вычитания, которая реализуется функцией **sub2**, находящейся в C-файле (назовем его proc.s).

Исходные тексты основной программы, а также функций **mul2** и **sub2** представлены в Листингах 5.7–5.9 соответственно.

Листинг 5.7

```
#include  <stdio.h>
#include  <LPC214X.H>
```

```
extern "C" void init_serial(void);
extern int mul2(void);
extern int sub2(void);
int i1, i2;

int main(void)
{
  init_serial();
  i1 = -99;
  i2 = 35;

  int res = mul2();
  printf(" Mul2 result: %d\n", res);
  res = sub2();
  printf(" Sub2 result: %d\n", res);
  while (1);
}
```

Листинг 5.8

```
extern int i1, i2;

int mul2(void)
{
    return i1*i2;
}
```

Листинг 5.9

```
extern int i1, i2;
int sub2(void)
{
  return i1-i2;
}
```

При компиляции исходных текстов всех файлов проекта генерируется следующий отчет:

```
Build target 'Target 1'
assembling Startup.s...
compiling Retarget.c...
compiling Serial.c...
compiling main.cpp...
compiling proc.cpp...
compiling proc2.c...
linking...
example4.axf: Error: L6218E: Undefined symbol sub2() (referred from main.o).
Target not created
```

Из этого отчета следует, что компоновщик не смог найти соответствие символического имени **sub2** конкретной процедуре или данным. Это произошло из-за того, что исходный текст, содержащий функцию **sub2**, был откомпилириован как C++-код, хотя его нужно было обрабатывать как C-код. Если бы в декларации функции **sub2** в основной программе был указан спецификатор "C", ошибки не произошло бы.

Таким образом, строка с декларацией функции **sub2** в основной программе (см. Листинг 5.7) должна выглядеть так:

```
extern  "C"  int  sub2(void);
```

Если в приложении необходимо получить доступ к данным из нескольких объектных модулей, то данные должны быть объявлены как глобальные в том модуле, где они определены. В других модулях эти же данные должны быть объявлены с директивой **extern**.

Использование общих данных в программах на C/C++ является общепринятой и весьма распространенной практикой, но в то же время здесь кроется и скрытый источник ошибок, поскольку данные легко могут быть разрушены вследствие ошибок в программе. Такие ошибки весьма распространены и трудно обнаруживаются. Рассмотрим два примера обработки общих данных в программе на Keil C/C++.

Пусть, как и в предыдущем примере, наша программа должна умножать и вычитать два целых числа с помощью функций **mul2** и **sub2**. Предположим, что в наш проект включены три файла с исходными текстами (main.cpp, proc.cpp и proc2.cpp), представленными в Листингах 5.10–5.12.

Листинг 5.10

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  int  mul2(void);
extern  "C"  int  sub2(void);
extern  int  i1,  i2;

int  main(void)
{
  init_serial();

  int  res  =  sub2();
  printf("  Sub2  result:  %d\n",  res);

  res  =  mul2();
  printf("  Mul2  result:  %d\n",  res);

  while  (1);
}
```

Листинг 5.11

```
extern  int  i1,  i2;

int  mul2(void)
{
    return  i1*i2;
}
```

Листинг 5.12

```
int  i1,  i2;
```

```
int  sub2(void)
{
  i1  =  −77,  i2  =  −12;
  return  i1-i2;
}
```

Как видно из исходных текстов, переменные **i1** и **i2** определены как глобальные в модуле proc2.c, где определена функция **sub2**. Обеим переменным **i1** и **i2** присваиваются начальные значения −77 и −12 соответственно. В модулях **main** и **proc** обе переменные объявлены как внешние с директивой **extern**. После компиляции и компоновки запущенное приложение выводит в терминальное окно следующий результат:

```
Sub2  result:  −65
Mul2  result:  924
```

Это именно тот результат, который мы ожидали получить.

Теперь изменим исходный текст основной программы таким образом, чтобы первой вызывалась функция **mul2** и только затем **sub2**. Модифицированный исходный текст основной программы теперь будет выглядеть так (Листинг 5.13):

Листинг 5.13

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  int  mul2(void);
extern  "C"  int  sub2(void);
extern  int  i1,  i2;

int  main(void)
{
  init_serial();

  int  res  =  mul2();
  printf("  Mul2  result:  %d\n",  res);

  res  =  sub2();
  printf("  Sub2  result:  %d\n",  res);

  while  (1);
}
```

После генерации исполняемого файла запустим приложение и посмотрим на результат. Он будет следующим:

```
Mul2  result:  0
Sub2  result:  −65
```

Значение **0**, полученное в результате умножения переменных **i1** и **i2**, легко объяснимо: глобальные переменные, объявленные в модуле с функцией **sub2**, получают по умолчанию нулевые значения. Поскольку **mul2** была вызвана раньше, чем **sub2**, то параметрам функции **mul2** были присвоены нулевые значения, следовательно и результат умножения получился равным нулю.

Что касается функции **sub2**, то здесь переменным **i1** и **i2** были присвоены конкретные значения, которые перезаписали предыдущие нули, присвоенные в процессе выполнения функции **mul2**. Последние два примера показывают, что при работе с глобальными переменными, к которым имеют доступ сразу несколько функций, нужно соблюдать особую осторожность.

До сих пор мы рассматривали типичные ошибки, генерируемые компилятором/компоновщиком, и методы их устранения. Если же компоновка приложения выполнена успешно, а приложение работает не так, как было задумано, и выдает странные результаты, то здесь наступает время отладки. В следующем разделе мы рассмотрим общие вопросы отладки приложений в среде программирования Keil для микроконтроллеров ARM.

5.3. Основы отладки приложений в среде Keil

Для отладки приложений и устранения логических (алгоритмических) ошибок в приложении в среде Keil имеется мощный отладчик/симулятор. Все примеры данной книги рассчитаны на отладку в симуляторе, который позволяет выполнить настройку приложения в виртуальной среде, моделирующей поведение реального микроконтроллера ARM. Используя симулятор, можно успешно оттрассировать большую часть программного кода, внести необходимые изменения в исходные тексты, а также выполнить оптимизацию приложения (мы поговорим об этом в следующей главе). Напомню, что для работы с симулятором необходимо установить опцию **Use Simulator** в окне настройки симулятора/отладчика (показана красной стрелкой на Рис. 5.4):

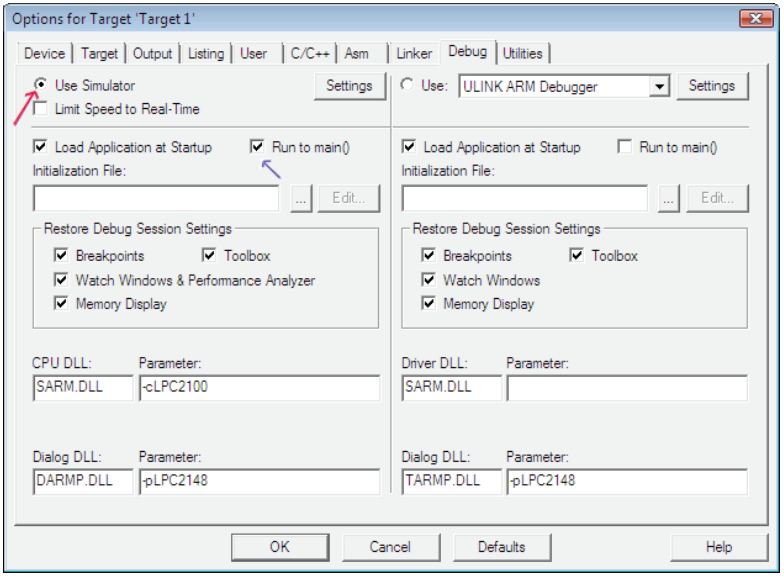

Рис. 5.4.

Кроме того, в панели настройки отладчика должна быть установлена опция **Run to main()**. Для выполнения отладки никаких дополнительных настроек, в принципе, не требуется. Для первоначального ознакомления с работой отладчика разработаем простое приложение, в котором будет единственный файл с исходным текстом программы на C++ (все необходимые файлы, такие, как Retarget.c, Serial.c, и исходный текст загрузчика Startup.s, естественно, должны быть включены в проект).

Исходный текст нашей основной программы на C++ будет таким (Листинг 5.14):

Листинг 5.14

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

int  main(void)
{
  init_serial();

  int  a1[10]  =  {9,  -23,  1,  33,  -51,  6,  18,  -44,  71,  -10};
  int  sum  =  0;
  for  (int  i1  =  0;  i1  <  10;  i1++)
        sum  +=  a1[i1];
  printf("  The  sum  of  elements  of  the  array  =  %d\n",  sum);
  while  (1);
}
```

Данное приложение будет вычислять сумму элементов массива **a1**, состоящего из десяти целых чисел. Текущая сумма сохраняется в переменной sum, начальное значение которой равно 0. Вычислительный алгоритм выполняется в цикле **for**. По завершению цикла значение суммы выводится в терминальное окно с помощью библиотечной функции **printf**.

Перед компиляцией проекта (назовем его **example4**) в него должны быть включены следующие файлы:

Retarget.c, Serial.c, Startup.s, main.cpp.

Окно проекта должно выглядеть так, как показано на Рис. 5.5.

Если в исходном тексте основной программы не было допущено ошибок, то после компиляции/компоновки мы получим исполняемый файл example4.axf в формате ELF. Далее желательно посмотреть, будет ли приложение работать корректно. Для этого можно запустить программу на выполнение в симуляторе, выбрав пиктограмму **Debug** (показана красной стрелкой в верхнем правом углу на Рис. 5.6).

Нажав на эту пиктограмму повторно, можно остановить процесс отладки. Альтернативно можно запустить приложение на отладку, если в меню **Debug** выбрать опцию **Start/Stop Debug Session,** как показано на Рис. 5.7.

После запуска программы на отладку выполним приложение и проанализируем результат. Для этого выберем опцию **Run** (показана красной стрелкой на Рис. 5.8).

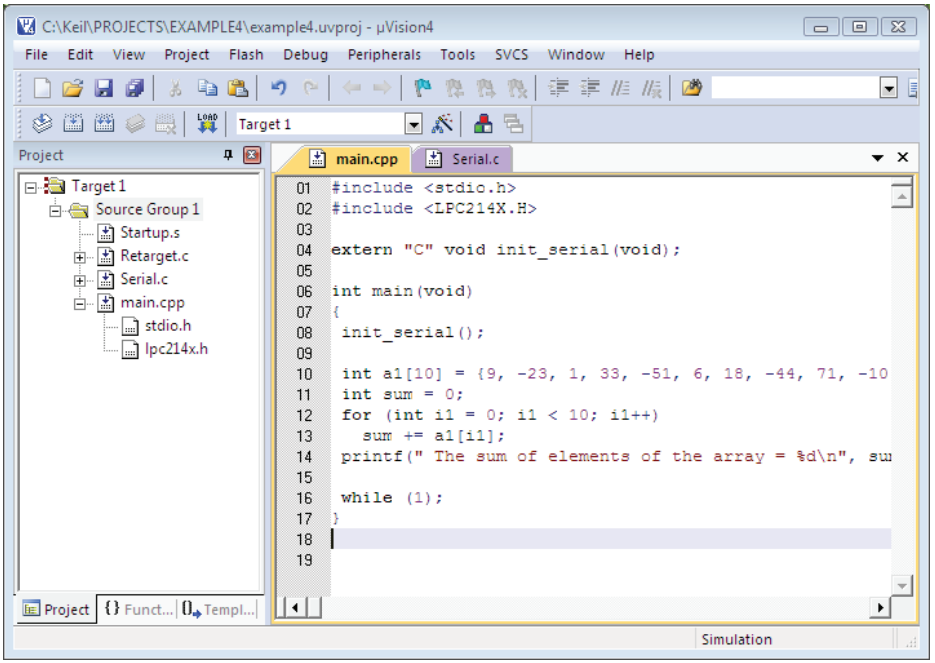
C:\Keil\PROJECTS\EXAMPLE4\example4.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Project
Target 1
Source Group 1
Startup.s
Retarget.c
Serial.c
main.cpp
stdio.h
lpc214x.h
Project Funct... Templ...
main.cpp
Serial.c
01 #include <stdio.h>
02 #include <LPC214X.H>
03
04 extern "C" void init_serial(void);
05
06 int main(void)
07 {
08 init_serial();
09
10 int a1[10] = {9, -23, 1, 33, -51, 6, 18, -44, 71, -10
11 int sum = 0;
12 for (int i1 = 0; i1 < 10; i1++)
13 sum += a1[i1];
14 printf(" The sum of elements of the array = %d\n", su
15
16 while (1);
17 }
18
19
Simulation

Рис. 5.5.

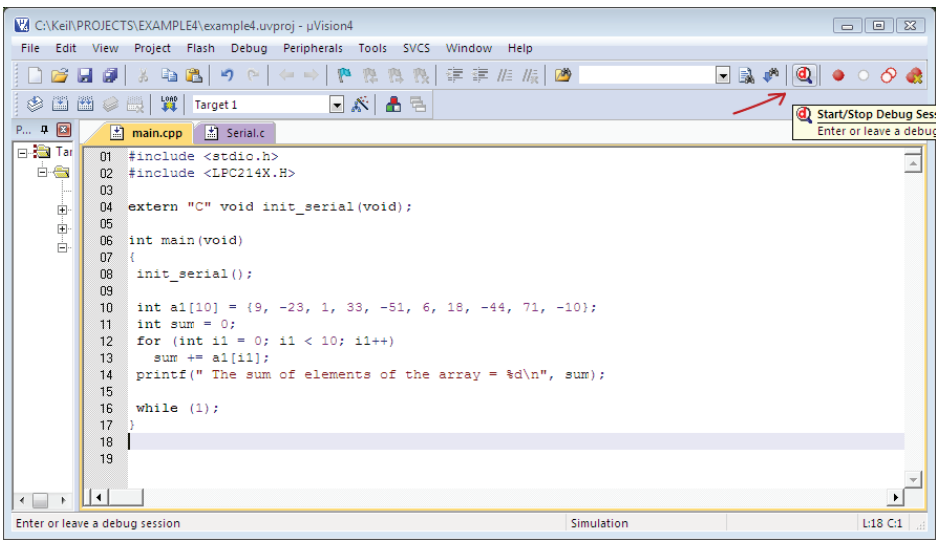
C:\Keil\PROJECTS\EXAMPLE4\example4.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Target 1
Start/Stop Debug Ses
Enter or leave a debug
main.cpp
Serial.c
01 #include <stdio.h>
02 #include <LPC214X.H>
03
04 extern "C" void init_serial(void);
05
06 int main(void)
07 {
08 init_serial();
09
10 int a1[10] = {9, -23, 1, 33, -51, 6, 18, -44, 71, -10};
11 int sum = 0;
12 for (int i1 = 0; i1 < 10; i1++)
13 sum += a1[i1];
14 printf(" The sum of elements of the array = %d\n", sum);
15
16 while (1);
17 }
18
19
Enter or leave a debug session
Simulation
L:18 C:1

Рис. 5.6.

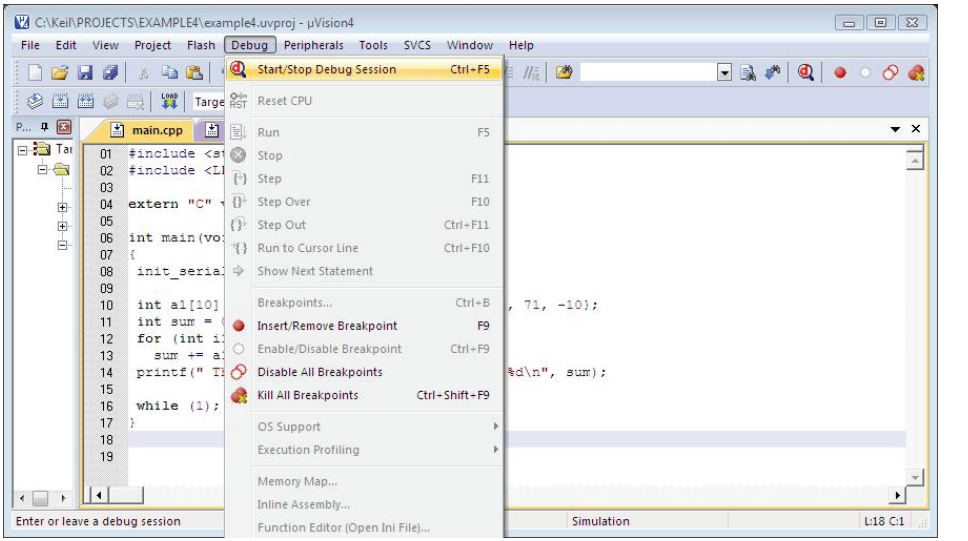
C:\Keil\PROJECTS\EXAMPLE4\example4.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Start/Stop Debug Session Ctrl+F5
Reset CPU
Run F5
Stop
Step F11
Step Over F10
Step Out Ctrl+F11
Run to Cursor Line Ctrl+F10
Show Next Statement
Breakpoints... Ctrl+B
Insert/Remove Breakpoint F9
Enable/Disable Breakpoint Ctrl+F9
Disable All Breakpoints
Kill All Breakpoints Ctrl+Shift+F9
OS Support
Execution Profiling
Memory Map...
Inline Assembly...
Function Editor (Open Ini File)...
main.cpp
Enter or leave a debug session
Simulation
L:18 C:1

Рис. 5.7.

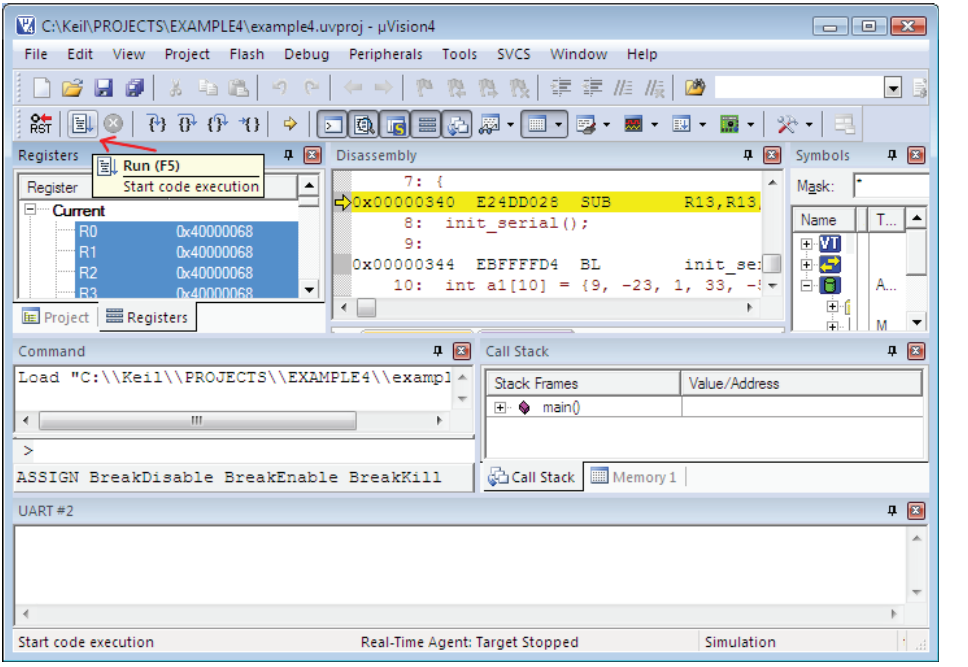
C:\Keil\PROJECTS\EXAMPLE4\example4.uvproj - µVision4
File Edit View Project Flash Debug Peripherals Tools SVCS Window Help
Registers
Run (F5)
Start code execution
Register
Current
R0 0x40000068
R1 0x40000068
R2 0x40000068
R3 0x40000068
Project
Registers
Disassembly
7: {
0x00000340 E24DD028 SUB R13,R13,
8: init_serial();
9:
0x00000344 EBFFFFD4 BL init_se
10: int a1[10] = {9, -23, 1, 33,
Symbols
Mask:
Name
T...
Command
Load "C:\\Keil\\PROJECTS\\EXAMPLE4\\exampl
ASSIGN BreakDisable BreakEnable BreakKill
Call Stack
Stack Frames
Value/Address
main()
Call Stack
Memory 1
UART #2
Start code execution
Real-Time Agent: Target Stopped
Simulation

Рис. 5.8.

При выборе этой опции приложение выполняется, начиная с первой инструкции, и заканчивается выполнением последнего оператора – этот режим полезен при проверке функционирования приложения в целом. Результат выполнения приложения будет выведен в терминальное окно **UART #2** (Рис. 5.9).

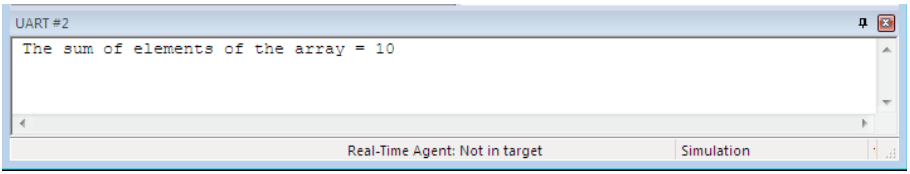

Рис. 5.9.

Результат работы приложения легко проверить вручную, и он совпадает с тем, что мы получили. В подавляющем большинстве случаев разработчика интересует не только конечный результат выполнения программы (даже если все работает правильно, как в нашем случае), но и то, как выполняется обработка данных в конкретных участках программного кода, как используются ресурсы памяти и регистры процессора, можно ли оптимизировать отдельные участки программного кода по быстродействию и т. д. В таких случаях требуется пошаговая отладка приложения, когда можно проанализировать результаты выполнения программного кода после выполнения какого-либо оператора C++ или, что дает больший уровень детализации, после выполнения отдельной инструкции процессора.

Отладчик Keil предлагает целый ряд возможностей по пошаговой отладке приложения. В простейшем варианте перед выполнением приложения можно установить точку останова (breakpoint) непосредственно в исходном тексте основной программы. В этом случае инструкции приложения будут выполняться до оператора или машинной инструкции, отмеченной соответствующей точкой останова. Разработчик может увидеть мгновенный снимок состояния всех ресурсов приложения в точке останова и проанализировать ход выполнения программы.

Предположим, нас интересует состояние переменных приложения, исходный текст которого показан в Листинге 5.14, непосредственно перед выполнением оператора цикла **`for`**.

В этом случае мы устанавливаем точку останова, как показано на Рис. 5.10. Для этого нужно щелкнуть правой кнопкой мыши напротив оператора или номера строки (в данном случае, 12), где выполнение программы должно приостанавливаться, и в выпадающем меню выбрать опцию **Insert/Remove Breakpoint**. Альтернативно можно установить точку останова "быстрой" клавишей **F9**. Точно так же можно и убрать ранее установленную точку останова. Установленная точка останова маркируется красным квадратом (Рис. 5.11).

Теперь мы можем вновь запустить приложение на выполнение как, это было описано ранее. После запуска приложение выполняется до того, как будет

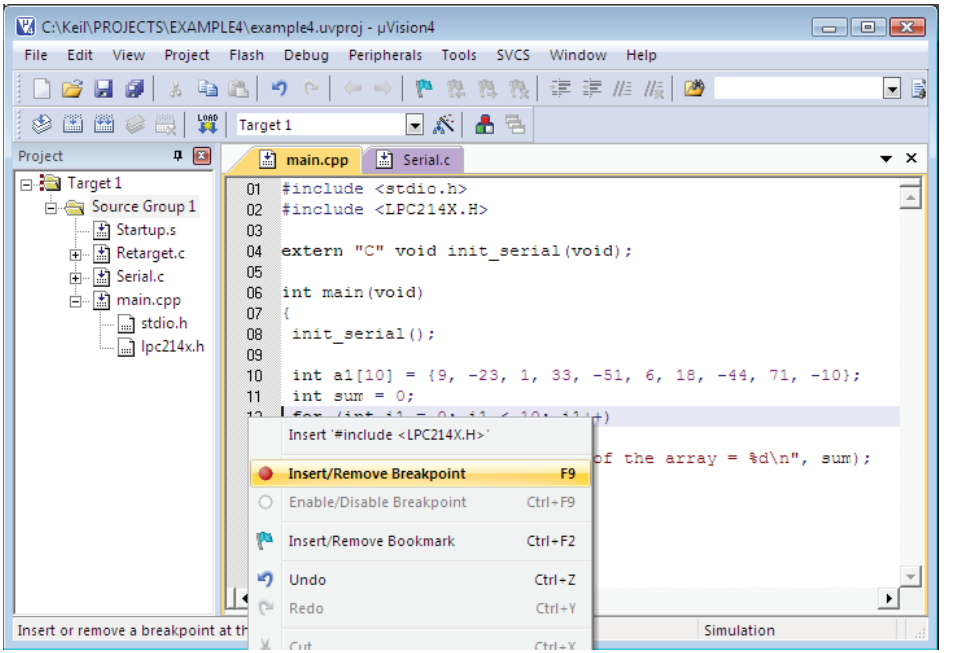

Рис. 5.10.

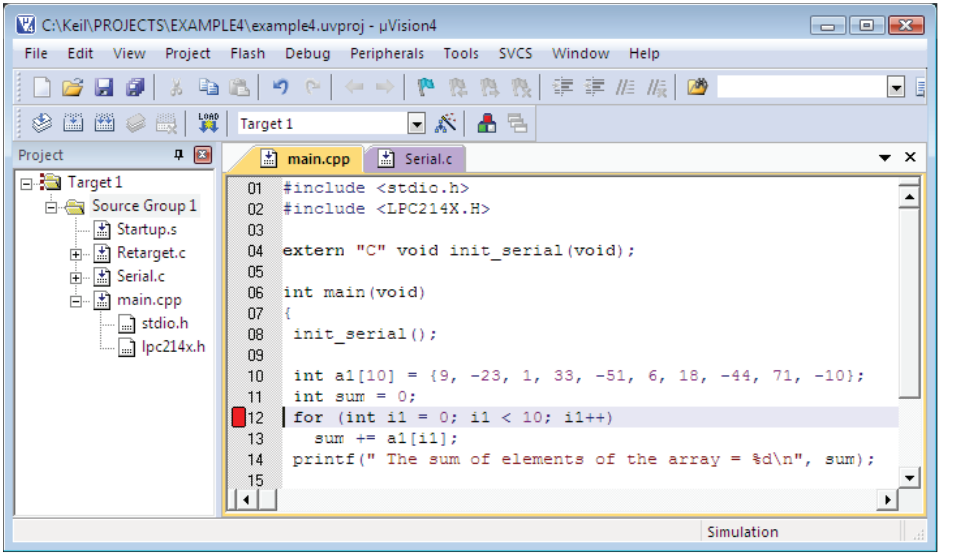

Рис. 5.11.

обнаружен оператор **for**. Мы можем посмотреть на состояние переменных программы, регистров микроконтроллера и стека в соответствующих окнах. По умолчанию в режиме отладки выводится информация о символических именах, состоянии регистров процессора и переменных, а также стека. Кроме того, в окне дизассемблера выводится код программы в форме инструкций микроконтроллера.

Состояние среды Keil в режиме отладки приложения показано на Рис. 5.12:

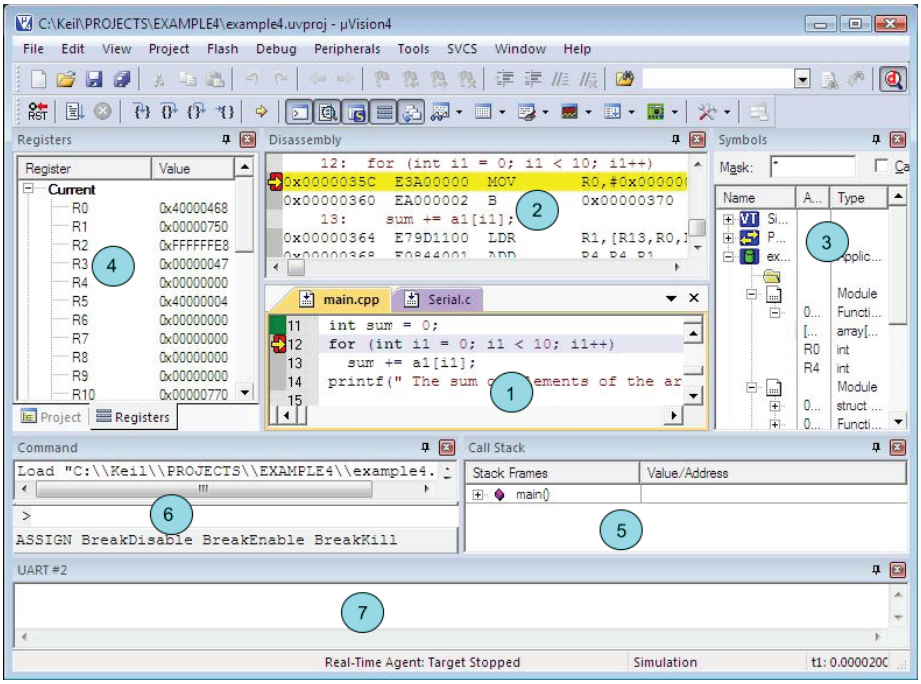

Рис. 5.12

В окне 1 выводится исходный текст приложения на C++ с указанием точки останова программы (желтая стрелка на красном фоне). Окно 2 содержит дизассемблированный код двоичного ELF-файла (example4.axf), в окне 3 показан список символических имен переменных и функций приложения с указанием их атрибутов. В окне 4 показано содержимое регистров процессора на момент останова приложения, а окно 5 отображает состояние стека. В окне 6 отображаются команды, выполняемые отладчиком, а в окно 7 выводится результат выполнения программы.

Список окон для вывода информации можно посмотреть, выбрав опцию **View** основного меню, здесь же можно указать, какую дополнительную информацию нужно выводить. В данном случае нас больше всего интересует состояние переменных нашей программы, поэтому в меню **View** выберем опцию **Watch Windows → Locals** (Рис. 5.13).

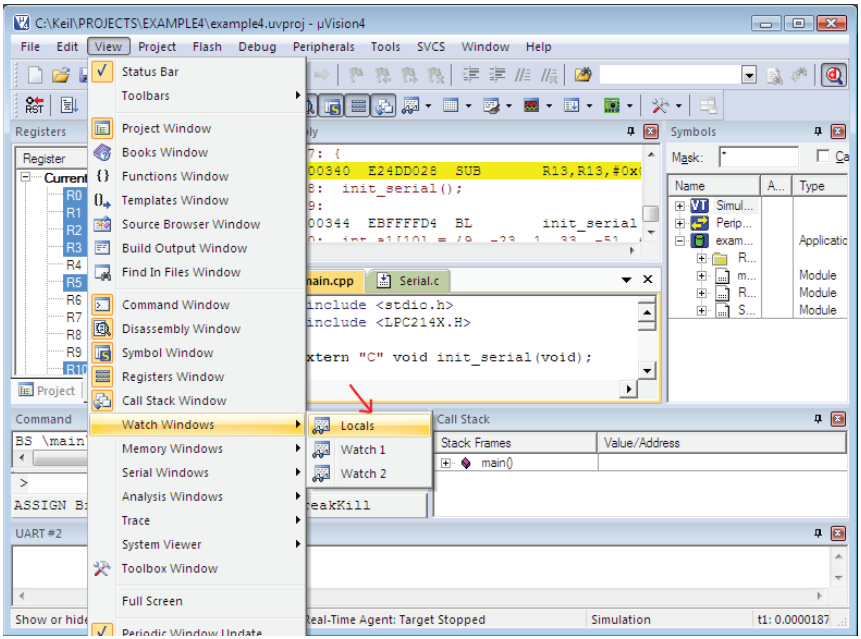

Рис. 5.13.

В окне **Locals** отладчик отображает текущее состояние переменных программы, так что мы сможем наблюдать, как изменяется их содержимое по ходу выполнения приложения. Давайте более подробно ознакомимся с той информацией, которая выводится в окнах отладчика – это существенно упростит навигацию по программному коду нашего приложения. Начнем с окна **Symbols**. В это окно отладчик выводит информацию о символических именах, присутствующих в нашем приложении, и о соответствии этих имен физическим ресурсам микроконтроллера (регистры, память и стек). На Рис. 5.14 окно **Symbols** показано в развернутом виде.

Как видно из рисунка, окно **Symbols** показывает таблицу, в первой колонке которой отоб-

Рис. 5.14.

ражается символическое имя ресурса, во второй колонке отображается привязка данного имени к конкретному физическому адресу/регистру и в третьей колонке показан тип объекта с данным символическим именем. Так, например, переменная **a1**, представляющая собой массив из 10 целых чисел, располагается в стеке (регистр **R13**) со смещением –40 байт от текущего значения.

Переменная **i1**, представляющая собой счетчик цикла **for**, располагается в регистре **R0** микроконтроллера. Переменная sum, содержащая текущее значение суммы, хранит свое значение в регистре **R4**. Точка входа в основную программу main привязана к физическому адресу `0x00000340` – по этому адресу хранится первая инструкция нашего приложения.

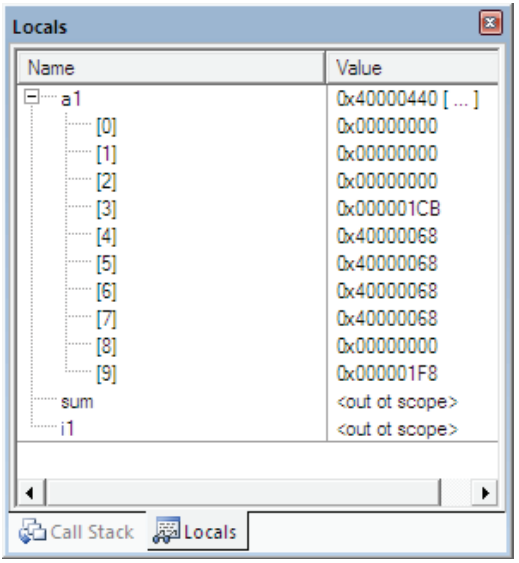

Рис. 5.15.

Эта информация будет как нельзя кстати, когда мы будем рассматривать отладку программного кода на нижнем уровне немного позже в этой и в следующей главах.

В окне **Locals**, как уже было сказано, отображается текущая информация о состоянии переменных программы. В развернутом виде окно **Locals** выглядит так, как показано на Рис. 5.15.

Здесь мы видим, что переменная массива **a1** располагается по адресу `0x40000440`, т. е. содержимое первого элемента **a1[0]** будет находиться по данному адресу. Поскольку наша программа еще не выполняется, элементы массива содержат произвольные значения, так же как и переменные **i1** и **sum**.

Еще одно окно отладчика, в котором отображается содержимое регистров ARM микроконтроллера, называется **Registers** (Рис. 5.16).

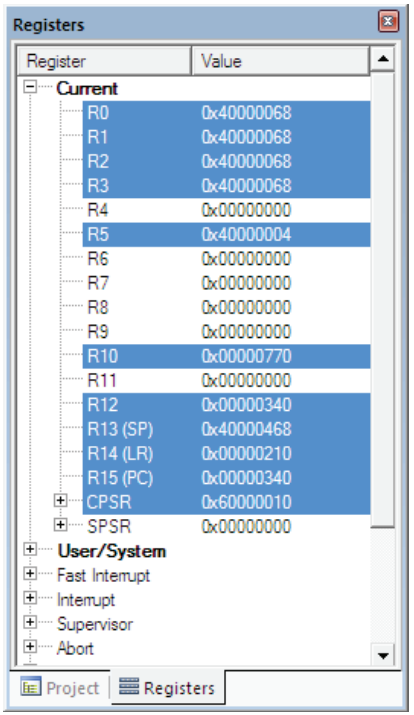

Рис. 5.16.

Здесь следует обратить особое внимание на регистры **R13**–**R15**. Регистр **R13** выполняет функцию указателя стека и содержит адрес вершины стека (в данном случае, перед запуском программы этот адрес равен 0x400000468). Регистр **R14** является регистром связи для вызываемых процедур, поэтому он обычно используется для хранения адреса инструкции, которая будет выполняться после завершения процедуры. Наконец, регистр **R15** является счетчиком инструкций (program counter, **pc**) и содержит адрес следующей инструкции процессора, которая должна выполняться.

В данном случае регистр **R15** содержит адрес 0x00000340, который соответствует точке входа в программу **main()** – этот адрес мы встретили в окне **Symbols** (см. Рис. 5.14).

Для успешной отладки приложений для ARM микроконтроллеров почти всегда требуется анализировать машинный (ассемблерный) код приложения. Именно на этом уровне можно обнаружить ошибки приложения, которые никаким другим образом не идентифицируются. Еще одним существенным моментом является и то, что для оптимизации программ необходимо понимать, как выполняется код приложения на самом нижнем уровне. Для просмотра и анализа ассемблерного кода приложения в отладчике используется окно **Disassembly** (окно 2 на Рис. 5.12). В следующем разделе мы проанализируем работу нашего демонстрационного приложения, используя только что описанные возможности отладчика.

5.4. Методика пошаговой отладки приложения и анализ программного кода

При пошаговой отладке приложения выполнение программы останавливается после каждой выполненной инструкции, что позволяет разработчику проанализировать промежуточные значения переменных в программе и выявить ошибки в программном алгоритме. Здесь можно применить различные методики. Если, например, требуется проанализировать программный код приложения с самого начала, то пошаговое выполнение следует начинать с самого первого оператора (инструкции). Во многих случаях нужно исследовать не весь, а только часть кода, начиная с некоторого оператора. В этом случае имеет смысл установить точку останова и выполнить часть кода до этой точки без отладки, а затем перейти к пошаговому выполнению.

Именно такую методику мы применим для исследования нашего приложения. Напомню, что мы будем заниматься отладкой приложения, исходный текст основной программы которого показан в Листинге 5.14. Для удобства приведу исходный текст программы из данного листинга еще раз:

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

int  main(void)
```

```
{
  init_serial();

  int  a1[10]  =  {9,  -23,  1,  33,  -51,  6,  18,  -44,  71,  -10};
  int  sum  =  0;
  for  (int  i1  =  0;  i1  <  10;  i1++)
      sum  +=  a1[i1];
  printf(" The  sum  of  elements  of  the  array  =  %d\n",  sum);
  while  (1);
}
```

Выберем в качестве точки останова оператор цикла **for** (см. Рис.5.13) и запустим приложение на отладку. Проанализируем, что произошло в программе при выполнении операторов, стоящих перед **for**. В окне **Locals** переменные будут иметь значения, показанные на Рис. 5.17.

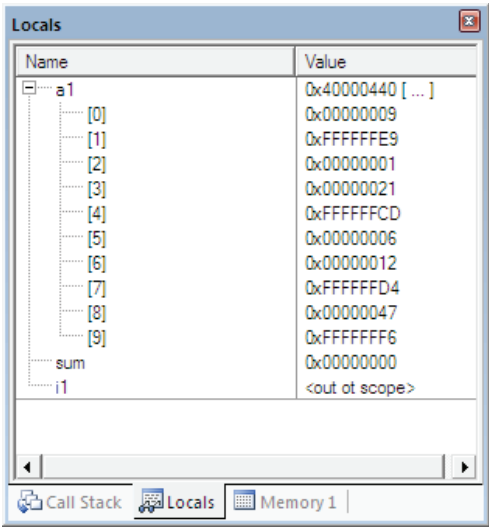

Рис. 5.17.

Как следует из рисунка, элементы массива **a1** проинициализированы значениями, указанными в соответствующем операторе присвоения, а переменная **sum** получила начальное значение 0. Поскольку оператор **for** не был выполнен, переменная **i1**, которая является счетчиком цикла, не была инициализирована.

Содержимое регистров микроконтроллера отображается в окне **Registers** и в точке останова будет таким, как показано на Рис. 5.18.

Обратите внимание на содержимое регистра-указателя стека **R13**, который содержит значение 0x400000440 — данный адрес является адресом массива **a1**. Регистр **R4** содержит текущее значение суммы элементов массива — в данный момент ни один элемент массива не обработан, поэтому содержимое этого регистра равно значению переменной sum, полученному при инициализации, т. е. **0**.

Давайте выполним пошаговую отладку программы, начиная с точки останова, и посмотрим, как будут меняться значения переменных. Для этого нужно нажимать на пиктограмму, на которую указывает красная стрелка на Рис. 5.19.

В результате пошаговой отладки программы становится возможным легко определить значения переменных после выполнения того или иного оператора. Тем не менее, для детального анализа функционирования приложения необходимо понимать, как выполняется программный код на низком уровне, а именно на уровне инструкций микроконтроллера – зачастую это единственный способ выявить скрытые ошибки программного алгоритма и единственный ключ к решению проблемы оптимизации производительности приложения. Вот как выглядит, например, дизассемблированный код цикла **`for`** нашего приложения (Листинг 5.15).

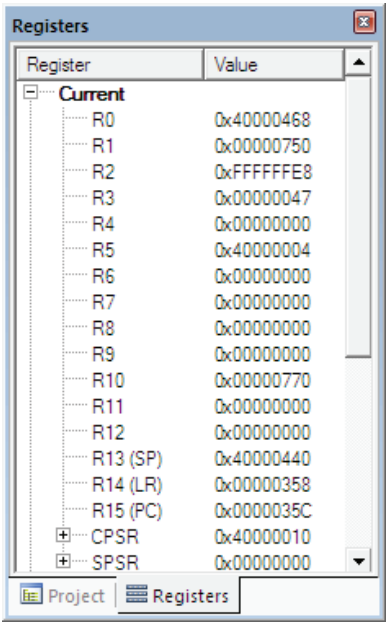

Рис. 5.18.

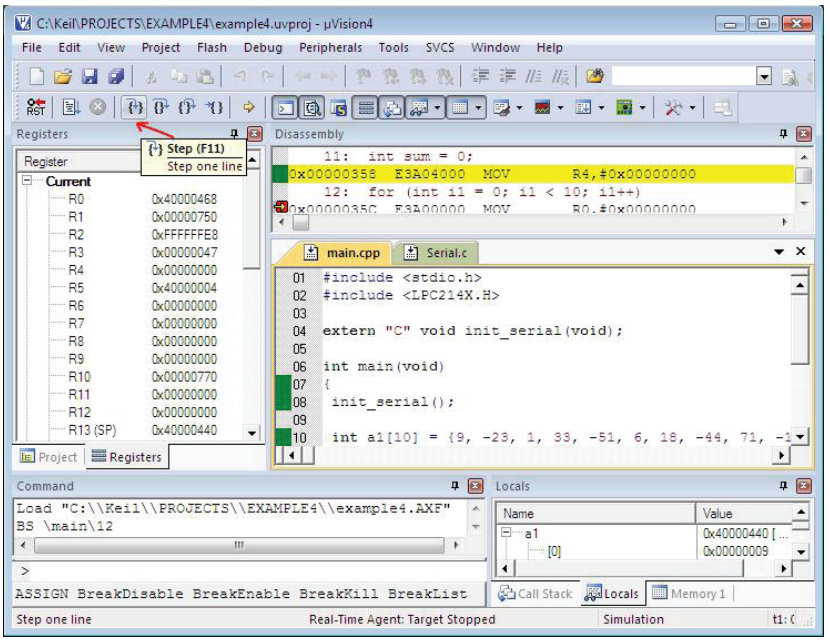

Рис. 5.19.

Листинг 5.15

```
        11:    int  sum  =  0;
0x00000358    E3A04000    MOV             R4,#0x00000000
        12:    for  (int  i1  =  0;  i1  <  10;  i1++)
0x0000035C    E3A00000    MOV             R0,#0x00000000
0x00000360    EA000002    B                 0x00000370
        13:        sum  +=  a1[i1];
0x00000364    E79D1100    LDR             R1,[R13,R0,LSL  #2]
0x00000368    E0844001    ADD             R4,R4,R1
0x0000036C    E2800001    ADD             R0,R0,#0x00000001
0x00000370    E350000A    CMP             R0,#0x0000000A
0x00000374    BAFFFFFA    BLT             0x00000364
```

Как видно из этого листинга, значения переменной sum сохраняются в регистре **R4**, значения переменной цикла **i1** находятся в **R0**, а условие выполнения цикла (**i1 < i0**) проверяется инструкцией **CMP**. Если условие **i1 < i0** истинно, флаг **Z** регистра текущего состояния программы (**cpsr**) остается сброшенным, поэтому инструкция **BLT** передает управление по адресу `0x00000364`. По данному адресу находится инструкция **LDR**, которая загружает в регистр **R1** следующий элемент массива. Инструкция **ADD** добавляет к текущей сумме, находящейся в регистре **R4**, значение элемента массива, загруженное в регистр **R1**. После этого следующая инструкция **ADD**, находящаяся по адресу `0x0000036C`, инкрементирует счетчик цикла в регистре **R0**. Если значение счетчика цикла становится равным 10, то инструкция **CMP** устанавливает флаг **Z**, что приводит к завершению цикла **for**.

ГЛАВА 6

АНАЛИЗ И ОПТИМИЗАЦИЯ ПРОГРАММНОГО КОДА МИКРОКОНТРОЛЛЕРОВ ARM

В процессе разработки и тестирования программного кода для микроконтроллеров ARM рано или поздно возникает вопрос эффективности работы приложения. Для относительно несложных программ с простыми вычислительными алгоритмами и небольшим объемом данных вопрос эффективности и производительности особого значения не имеет, но для комплексных приложений со сложными алгоритмами обработки данных и/или для приложений, выполняющих операции в реальном времени, вопрос производительности является ключевым.

Сейчас мало просто написать приложение – нужно разработать приложение, работающее с максимальной эффективностью и потребляющее, по возможности, минимальный объем ресурсов системы. Второе требование, которое было существенным еще десяток лет тому назад, в настоящее время утратило актуальность. Причина этого кроется в том, что требования по повышению производительности и минимизации объема программного кода требуют диаметрально противоположных подходов. Быстрый код требует манипуляций с типами данных, которые обрабатываются максимально быстро данным процессором. Например, ARM микроконтроллеры максимально оперируют с 32-разрядными типами данных, поскольку их архитектура оптимизирована именно для этого типа данных.

В то же время для выполнения программы при минимальном объеме оперативной и постоянной памяти требуется использовать 16- или 8-разрядные типы данных. В ARM контроллерах предусмотрен специальный 16-разрядный режим (Thumb) для приложений, которые должны выполняться с минимально задействованными ресурсами. Тем не менее, оптимизация программы по объему кода/данных в настоящее время перестает быть актуальной. Причина проста: современные микроконтроллеры и микропроцессоры выпускаются со все большими объемами оперативной и постоянной памяти, что позволяет разрабатывать приложения без оглядки (разве что в редких случаях) на имеющийся ресурс. Таким образом, акцент смещается в сторону оптимизации производительности приложений.

Современные компиляторы языков высокого уровня, таких как C/C++, Java, Python, имеют встроенные алгоритмы оптимизации программного кода приложения. То же самое касается и популярных компиляторов языка C/C++, раз-

работанных Keil и IAR для микроконтроллеров ARM. Это означает, например, что компилятор Keil при указании тех или иных параметров оптимизации будет пытаться реализовать один из предопределенных алгоритмов оптимизации приложения. Но здесь возникает одна существенная проблема, которую ни один компилятор разрешить не в состоянии. Дело в том, что компилятор ничего не "знает" об особенностях работы вашего приложения и пытается выполнить в некотором смысле "усредненную" оптимизацию всего программного кода приложения, которая даст минимальный эффект. Большинство приложений нуждаются в "точечной" оптимизации, когда необходимо добиться максимальной производительности на каких-то отдельных участках кода – эти критические участки знает только разработчик и никакой компилятор здесь не поможет.

Подобная точечная оптимизация программного кода всегда выполняется самим разработчиком и только сам разработчик может добиться максимальной эффективности работы приложения. Оптимизация программного кода – это своего рода искусство, которое требует достаточно хороших знаний особенностей функционирования микроконтроллера/микропроцессора и низкоуровневого программирования. Тем не менее, существует целый ряд методик, базирующихся на особенностях архитектуры конкретного микроконтроллера, которые смогут помочь в оптимизации производительности приложений.

В этой главе мы рассмотрим несколько основных направлений повышения производительности программ, написанных для ARM микроконтроллеров. При анализе дизассемблированных кодов приложений, а также при разработке исходных текстов приложений читателю необходимо знать, по крайней мере, основы программирования на языке ассемблера для микроконтроллеров ARM.

6.1. Выбор типов данных в приложении

Выбор типов данных, с которыми манипулирует приложение, оказывает существенное влияние на производительность программы. Для иллюстрации подходов к выбору типов данных разработаем несколько демонстрационных приложений в Keil C++, проанализируем их дизассемблированный код и выполним оптимизацию.

Пусть наш проект (назовем его **example5**) содержит C++ файл (main.cpp) со следующим исходным текстом (Листинг 6.1):

Листинг 6.1

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

int  main(void)
{
  int  a1[10]  =  {  44,  -7,  21,  -69,  -93,  77,  50,  -11,  9};
  unsigned  short  i1;
  int  sum  =  0;
```

```
init_serial();

for  (i1  =  0;  i1  <  10;  i1++)
  sum  +=  a1[i1];
printf("  The  sum  of  elements  of  the  array  =  %d\n",  sum);
while  (1);
}
```

Эта программа подсчитывает сумму элементов целочисленного массива **a1**, состоящего из 10 элементов. Текущее значение суммы хранится в переменной **sum**, а в качестве переменной цикла **for** используется переменная **i1** типа **unsigned short**. Откомпилируем наш проект и выполним отладку исполняемого файла example5.axf, используя симулятор Keil.

Для представления о том, какие ресурсы процессора использует приложение (эта информация весьма полезна при анализе листинга дизассемблера), нужно открыть или расширить окно **Symbols** (Рис. 6.1):

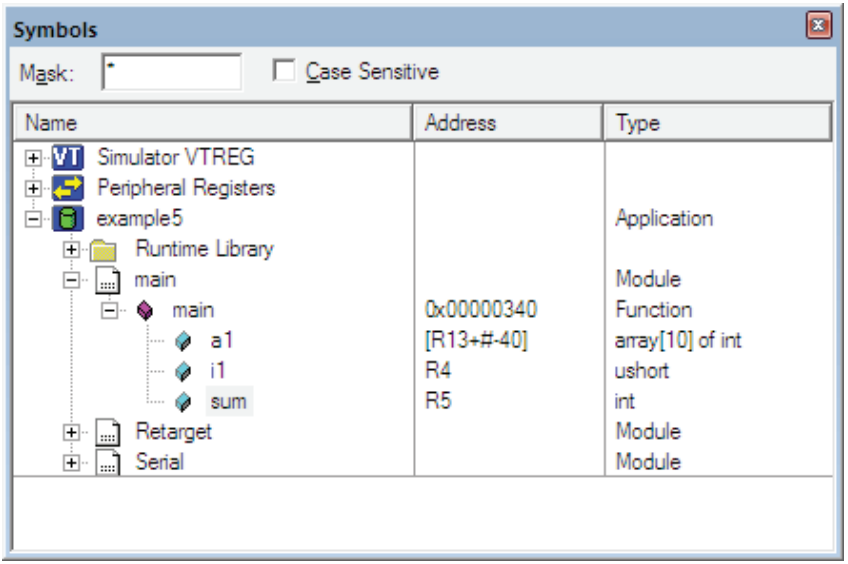

Рис. 6.1.

Наше приложение (функция **main()**), как следует из рисунка, начинает выполняться с адреса 0x00000340. Переменная **a1**, идентифицирующая массив из 10 целых чисел, имеет начальный адрес со смещением –40 в регистре-указателе стека (**R13**). Текущее значение переменной **i1**, выполняющей функцию счетчика цикла, хранится в регистре **R4**, а текущее значение суммы (переменная **sum**) находится в регистре **R5**.

Располагая этой информацией, мы сможем легко ориентироваться в листинге дизассемблированного двоичного кода приложения. Фрагмент дизассемблированного кода, представляющий для нас интерес, показан в Листинге 6.2.

Листинг 6.2

```
          7:  {
0x00000340    E24DD028    SUB              R13,R13,#0x00000028
          8:    int  a1[10]  =  {  44,  -7,  21,  -69,  -93,  77,  50,  -11,  9};
          9:    unsigned  short  i1;
0x00000344    E3A02028    MOV              R2,#0x00000028
0x00000348    E59F1040    LDR              R1,[PC,#0x0040]
0x0000034C    E1A0000D    MOV              R0,R13
0x00000350    EB00005F    BL                __rt_memcpy_w(0x000004D4)
         10:    int  sum  =  0;
0x00000354    E3A05000    MOV              R5,#0x00000000
         11:    init_serial();
         12:
0x00000358    EBFFFFBD    BL                init_serial(0x00000254)
         13:    for  (i1  =  0;  i1  <  10;  i1++)
0x0000035C    E3A04000    MOV              R4,#0x00000000
0x00000360    EA000003    B                  0x00000374
         14:      sum  +=  a1[i1];
0x00000364    E79D0104    LDR              R0,[R13,R4,LSL  #2]
0x00000368    E0855000    ADD              R5,R5,R0
0x0000036C    E2840001    ADD              R0,R4,#0x00000001
0x00000370    E3C04801    BIC              R4,R0,#0x00010000
0x00000374    E354000A    CMP              R4,#0x0000000A
0x00000378    BAFFFFF9    BLT              0x00000364
         15:    printf("  The  sum  of  elements  of  the  array  =  %d\n",  sum);
0x0000037C    E1A01005    MOV              R1,R5
0x00000380    E28F000C    ADD              R0,PC,#0x0000000C
0x00000384    EB00000D    BL                $Ven$AT$I$$__2printf(0x000003C0)
         16:    while  (1);
0x00000388    E1A00000    NOP
0x0000038C    EAFFFFFE    B                  0x0000038C
```

Как следует из листинга дизассемблера, значение переменной **i1** содержится в регистре **R4** микроконтроллера (это легко увидеть, просмотрев содержимое окна **Symbols** в отладчике). Текущее содержимое регистра **R4** помещается в регистр **R0** и инкрементируется на 1 в текущей итерации с помощью инструкции **ADD**:

```
0x0000036C    E2840001    ADD              R0,R4,#0x00000001
```

Затем инструкция **BIC** приводит содержимое регистра **R4** к диапазону, в котором должны находиться беззнаковые целые числа:

```
0x00000370    E3C04801    BIC              R4,R0,#0x00010000
```

После этого выполняется сравнение содержимого регистра **R4** со значением 10 и, если условие истинно, то выполняется переход по адресу `0x00000364` и цикл повторяется.

Модифицируем исходный текст основной программы в Листинге 6.1, изменив тип переменной цикла **i1** с **unsigned short** на **int** (Листинг 6.3):

Листинг 6.3

```
#include  <stdio.h>
```

```
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

int  main(void)
{
  int  a1[10]  =  {  44,  -7,  21,  -69,  -93,  77,  50,  -11,  9};
  int  i1;
  int  sum  =  0;
  init_serial();

  for  (i1  =  0;  i1  <  10;  i1++)
    sum  +=  a1[i1];
  printf("  The  sum  of  elements  of  the  array  =  %d\n",  sum);
  while  (1);
  }
```

После повторной компиляции и сборки запустим приложение в отладчике и проанализируем, как изменился код программы. Интересующий нас фрагмент показан далее (Листинг 6.4):

Листинг 6.4

```
        13:    for  (i1  =  0;  i1  <  10;  i1++)
0x0000035C    E3A04000    MOV             R4,#0x00000000
0x00000360    EA000002    B                 0x00000370
        14:      sum  +=  a1[i1];
0x00000364    E79D0104    LDR             R0,[R13,R4,LSL  #2]
0x00000368    E0855000    ADD             R5,R5,R0
0x0000036C    E2844001    ADD             R4,R4,#0x00000001
0x00000370    E354000A    CMP             R4,#0x0000000A
0x00000374    BAFFFFFA    BLT             0x00000364
```

Как видно из листинга дизассемблера, в результате замены типа в переменной **i1** исчезла инструкция **BIC**. Таким образом, использование 32-разрядной переменной вместо 16-разрядной сэкономило нам одну инструкцию программного кода.

Конечно, для обработки десяти элементов массива это несущественный выигрыш, но что будет, если придется обрабатывать, скажем, 10 000 и более элементов? В этом случае выигрыш в производительности может оказаться весьма существенным, поскольку не понадобится выполнять 10 000 дополнительных инструкций.

Подобная ситуация возникает и в случае присвоения переменной **i1** типа **char**, который использует 8 разрядов. В этом случае компилятор добавил бы в тело цикла **for** дополнительную инструкцию **AND** для ограничения диапазона изменения переменной **i1** значениями в диапазоне 0 – 255 (Листинг 6.5):

Листинг 6.5

```
          13:    for  (i1  =  0;  i1  <  10;  i1++)
0x0000035C    E3A04000    MOV             R4,#0x00000000
0x00000360    EA000003    B                 0x00000374
        14:      sum  +=  a1[i1];
```

```
0x00000364    E79D0104    LDR             R0,[R13,R4,LSL  #2]
0x00000368    E0855000    ADD             R5,R5,R0
0x0000036C    E2840001    ADD             R0,R4,#0x00000001
0x00000370    E20040FF    AND             R4,R0,#0x000000FF
0x00000374    E354000A    CMP             R4,#0x0000000A
0x00000378    BAFFFFF9    BLT             0x00000364
```

Следует отметить один важный факт, и это видно на примерах дизассемблированного кода: ARM микроконтроллер фактически оперирует с 32-разрядными значениями, даже если в программе используются другие типы данных. При этом тип данных приводится к 32-разрядному в процессе компиляции приложения.

6.2. Использование указателей для оптимизации ARM приложений

Значительную долю в вычислительных алгоритмах занимают операции копирования, перемещения или поиска данных. Значительная часть информации организована в форме массивов, структур и записей, доступ к которым осуществляется одним из двух распространенных способов: либо посредством индексации каждого элемента, либо через указатель. Определить, какой способ предпочтительнее, нам поможет простой пример копирования элементов одного целочисленного массива в другой.

Вначале рассмотрим копирование посредством индексации элементов массивов. Исходный текст основной программы приложения на Keil C++ представлен в Листинге 6.6.

Листинг 6.6

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

int  main(void)
{
  init_serial();
  int  src[10]  =  {  44,  -7,  21,  -69,  -93,  77,  50,  -11,  9,  111};
  int  dst[10];

  for  (int  i1  =  0;  i1  <  sizeof(src)/4;  i1++)
       dst[i1]  =  src[i1];
  for  (int  i1  =  0;  i1  <  sizeof(dst)/4;  i1++)
       printf("  %d  ",  dst[i1]);
  while  (1);
  }
```

Это приложение выполняет копирование элементов исходного массива **src** в буфер памяти **dst**, зарезервированный для 10 целых чисел. Операция копирования выполняется в цикле **for** с переменной цикла **i1**. В каждой итерации элемент исходного массива копируется на соответствующую позицию в

буфере назначения. Копирование выполняется с помощью обычного оператора присвоения. Далее содержимое буфера приемника выводится в терминальное окно посредством функции **printf**.

После компиляции и создания исполняемого файла приложения выполним программу в отладчике. Связь переменных нашего приложения и ресурсов процессора показана в окне **Symbols** (Рис. 6.2):

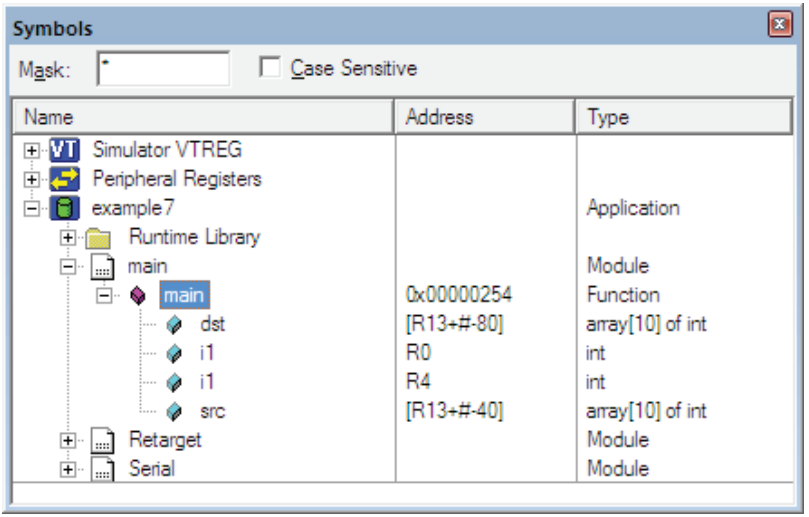

Рис. 6.2.

Как видно из таблицы, представленной в окне **Symbols**, первая инструкция нашего приложения загружена по адресу `0x00000254`. Оба массива (переменные **src** и **dst**) адресуются посредством регистра-указателя стека (**R13**) с соответствующими смещениями относительно базового адреса. Переменная **i1** появляется в нашей программе в двух циклах **for** и ее значения хранятся в регистрах **R0** и **R4**.

Проанализируем интересующий нас фрагмент дизассемблерного кода, представленный в Листинге 6.7:

Листинг 6.7

```
7:  {
0x00000254    E24DD050    SUB             R13,R13,#0x00000050
          8:    init_serial();
0x00000258    EB00002A    BL               init_serial(0x00000308)
          9:    int  src[10]  =  {  44,  -7,  21,  -69,  -93,  77,  50,  -11,
9,  111};
         10:    int  dst[10];
0x0000025C    E3A02028    MOV             R2,#0x00000028
0x00000260    E59F104C    LDR             R1,[PC,#0x004C]
0x00000264    E28D0028    ADD             R0,R13,#0x00000028
0x00000268    EB000095    BL               __rt_memcpy_w(0x000004C4)
         11:    for  (int  i1  =  0;  i1  <  sizeof(src)/4;  i1++)
```

```
0x0000026C    E3A00000    MOV             R0,#0x00000000
0x00000270    EA000003    B                 0x00000284
        12:         dst[i1]  =  src[i1];
0x00000274    E28D1028    ADD             R1,R13,#0x00000028
0x00000278    E7911100    LDR             R1,[R1,R0,LSL  #2]
0x0000027C    E78D1100    STR             R1,[R13,R0,LSL  #2]
0x00000280    E2800001    ADD             R0,R0,#0x00000001
0x00000284    E350000A    CMP             R0,#0x0000000A
0x00000288    3AFFFFF9    BCC             0x00000274
        13:    for  (int  i1  =  0;  i1  <  sizeof(dst)/4;  i1++)
0x0000028C    E3A04000    MOV             R4,#0x00000000
0x00000290    EA000003    B                 0x000002A4
        14:          printf("  %d  ",  dst[i1]);
0x00000294    E79D1104    LDR             R1,[R13,R4,LSL  #2]
0x00000298    E28F0018    ADD             R0,PC,#0x00000018
0x0000029C    EB000043    BL               $Ven$AT$I$$__2printf(0x000003B0)
0x000002A0    E2844001    ADD             R4,R4,#0x00000001
0x000002A4    E354000A    CMP             R4,#0x0000000A
0x000002A8    3AFFFFF9    BCC             0x00000294
        15:    while  (1);
0x000002AC    E1A00000    NOP
0x000002B0    EAFFFFFE    B                 0x000002B0
```

Рассмотрим, как выполняется копирование одного элемента. Начальные адреса обоих массивов хранятся в регистре-указателе стека (**R13**) с различным смещением. Инструкция

```
0x00000274    E28D1028    ADD             R1,R13,#0x00000028
```

загружает в регистр **R1** процессора адрес массива **src**, который вычисляется, как сумма базового адреса в регистре **R13** и смещения к адресу первого элемента массива-источника. Следующая далее инструкция

```
0x00000278    E7911100    LDR             R1,[R1,R0,LSL  #2]
```

загружает в регистр **R1** значение элемента массива. Адрес текущего элемента вычисляется относительно базового адреса в **R13** путем прибавления содержимого регистра **R0**, умноженного на 4, к содержимому регистра **R13**. Регистр **R0** содержит индекс текущего элемента массива, который после умножения прибавляется к базовому адресу – таким образом выполняется проход по массиву.

Содержимое регистра **R1** тут же записывается в буфер-приемник по адресу, который определяется суммой содержимого указателя стека и смещения, кратного четырем. Операция записи в память выполняется инструкцией **STR**:

```
0x0000027C    E78D1100    STR             R1,[R13,R0,LSL  #2]
```

Далее содержимое счетчика цикла, которое находится в регистре **R0** (соответствует значению переменной **i1**), инкрементируется и проверяется на равенство 10. Эти действия выполняют инструкции ARM микроконтроллера, показанные далее:

```
0x00000280    E2800001    ADD             R0,R0,#0x00000001
0x00000284    E350000A    CMP             R0,#0x0000000A
```

Инструкция **CMP** микроконтроллера, в зависимости от результата сравнения операндов, устанавливает или сбрасывает флаги **Z** и **C** в регистре текущего состояния программы (**cpsr**). Если содержимое регистра **R0** превысит 10, будет установлен флаг переноса **C**, что приведет к выходу из цикла (инструкция **BCC** будет пропущена и управление будет передано следующей за ней инструкции). До тех пор, пока значение **R0** меньше 10, флаг **C** остается очищенным и управление передается в начало цикла инструкцией **BCC**.

Как видно из анализа этого фрагмента дизассемблированного кода, в формировании адресов источника и приемника задействованы три инструкции, кроме того, здесь базовым регистром для вычисления адресов является указатель стека, что не совсем удобно. Необходимо учитывать, что регистр-указатель стека активно используется, в частности, для передачи параметров в процедуры, поэтому для доступа к нему могут понадобиться дополнительные инструкции.

Второй метод для обработки данных в массивах – использование указателей. Перепишем исходный C++ текст нашей программы копирования таким образом, чтобы для доступа к элементам массива-источника и массива-приемника можно было использовать указатели. Исходный текст нашей модифицированной программы представлен в Листинге 6.8:

Листинг 6.8

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

int  main(void)
{
  init_serial();
  int  src[10]  =  {  44,  -7,  21,  -69,  -93,  77,  50,  -11,  9,  111};
  int  *psrc  =  src;

  int  dst[10];
  int  *pdst  =  dst;

  for  (int  i1  =  0;  i1  <  sizeof(src)/4;  i1++)
      *(pdst++)  =  *(psrc++);

  for  (int  i1  =  0;  i1  <  sizeof(dst)/4;  i1++)
        printf("  %d  ",  dst[i1]);
  while  (1);
  }
```

В этой программе используются два указателя – **psrc** и **pdst**. Первый содержит адрес массива-источника **src**, а второй – адрес массива-приемника **dst**. Копирование элементов, как и в предыдущем примере, где используются индексы, выполняется в цикле **for**. Оператор

```
*(pdst++)  =  *(psrc++);
```

помимо операции копирования, инкрементирует оба указателя, так что они будут указывать на следующие элементы в обоих массивах.

Откомпилируем наше приложение и запустим его на отладку. Прежде всего посмотрим на используемые нашим приложением ресурсы процессора, отображаемые в окне **Symbols** (Рис. 6.3).

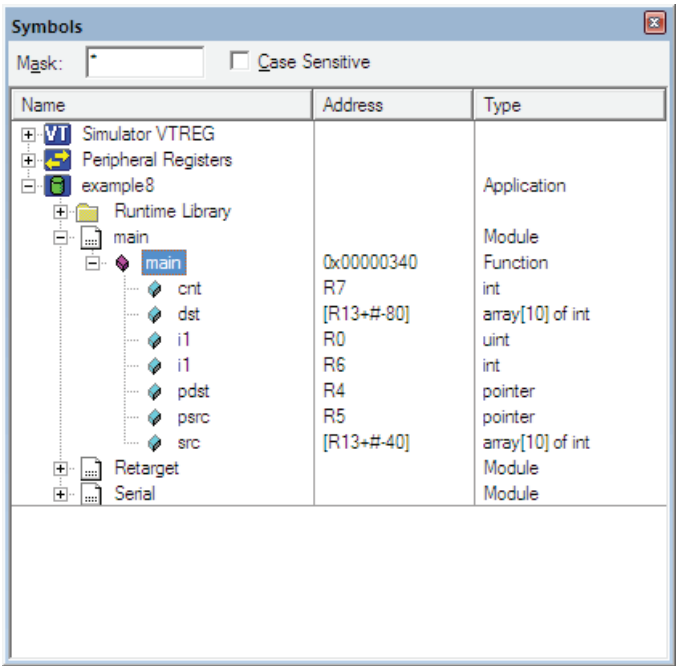

Рис. 6.3.

По сравнению с предыдущим приложением, у нас появились два указателя, **`psrc`** и **`pdst`**, которые содержат адреса массивов **`src`** и **`dst`** соответственно. Указатель **`psrc`** идентифицируется регистром **`R5`**, а указатель **`pdst`** – регистром **`R4`**.

Ниже приведен фрагмент дизассемблированного кода этого приложения (Листинг 6.9).

Листинг 6.9

```
          7:  {
0x00000340    E24DD050    SUB             R13,R13,#0x00000050
          8:    init_serial();
0x00000344    EBFFFFD4    BL               init_serial(0x0000029C)
          9:    int  src[10]  =  {  44,  -7,  21,  -69,  -93,  77,  50,  -11,
9,  111};
0x00000348    E3A02028    MOV             R2,#0x00000028
0x0000034C    E59F1050    LDR             R1,[PC,#0x0050]
0x00000350    E28D0028    ADD             R0,R13,#0x00000028
```

```
0x00000354    EB00005B    BL                __rt_memcpy_w(0x000004C8)
        10:    int  *psrc  =  src;
        11:
        12:    int  dst[10];
0x00000358    E28D5028    ADD               R5,R13,#0x00000028
        13:    int  *pdst  =  dst;
        14:
0x0000035C    E1A0600D    MOV               R6,R13
        15:    for  (int  i1  =  0;  i1  <  sizeof(src)/4;  i1++)
0x00000360    E3A00000    MOV               R0,#0x00000000
0x00000364    EA000002    B                   0x00000374
        16:        *(pdst++)  =  *(psrc++);
        17:
0x00000368    E4951004    LDR               R1,[R5],#0x0004
0x0000036C    E4861004    STR               R1,[R6],#0x0004
0x00000370    E2800001    ADD               R0,R0,#0x00000001
0x00000374    E350000A    CMP               R0,#0x0000000A
0x00000378    3AFFFFFA    BCC               0x00000368
        18:    for  (int  i1  =  0;  i1  <  sizeof(dst)/4;  i1++)
0x0000037C    E3A04000    MOV               R4,#0x00000000
0x00000380    EA000003    B                   0x00000394
        19:          printf("  %d  ",  dst[i1]);
0x00000384    E79D1104    LDR               R1,[R13,R4,LSL  #2]
0x00000388    E28F0018    ADD               R0,PC,#0x00000018
0x0000038C    EB000008    BL                 $Ven$AT$I$$__2printf(0x000003B4)
0x00000390    E2844001    ADD               R4,R4,#0x00000001
0x00000394    E354000A    CMP               R4,#0x0000000A
0x00000398    3AFFFFF9    BCC               0x00000384
        20:    while  (1);
0x0000039C    E1A00000    NOP
0x000003A0    EAFFFFFE    B                   0x000003A0
```

Как и в примере, где используются индексы, адреса обоих массивов помещаются в стек (регистр **R13**). Однако затем адрес массива-источника помещается в регистр **R5** с учетом необходимого смещения с помощью инструкции **ADD**:

```
0x00000358    E28D5028    ADD               R5,R13,#0x00000028
```

Адрес массива-приемника помещается в регистр **R6** с помощью инструкции **MOV**:

```
0x0000035C    E1A0600D    MOV               R6,R13
```

Оба регистра, **R5** и **R6**, будут использоваться. как адресные, в операции копирования. Сама операция копирования для текущего элемента массива-источника требует выполнения всего двух команд, **LDR** и **STR**:

```
0x00000368    E4951004    LDR               R1,[R5],#0x0004
0x0000036C    E4861004    STR               R1,[R6],#0x0004
```

Обратите внимание на то, что обе инструкции работают с пост-индексной адресацией, автоматически инкрементируя соответствующий регистр – указатель адреса. Здесь в операции копирования стек не используется и, кроме того, количество инструкций процессора в цикле уменьшилось на 1 по сравнению с приложением, где используется индексация.

Таким образом, из этого примера можно сделать вывод, что использование указателей в операциях, связанных с перемещением и/или поиском больших объемов данных, дает выигрыш в производительности по сравнению с индексированием данных. Этот выигрыш становится весьма ощутимым при большом количестве итераций цикла **for**.

6.3. Оптимизация циклов

Хорошо известные программистам операторы, управляющие циклическими вычислениями (**for**, **while**, **do**...**while**), могут давать заметный выигрыш в производительности, если использовать методику, известную как разворачивание (unrolling) циклов.

Как показывает анализ дизассемблерованного кода приложений, которые мы проанализировали ранее в этой главе, каждая итерация цикла требует как минимум три дополнительные инструкции. При этом мы не учитываем те операции, для выполнения которых этот цикл собственно и предназначен. В каждом цикле из рассмотренных ранее приложений можно выделить последовательность инструкций, которые выполняют проверку условия продолжения цикла, инкрементируют счетчик цикла и начинают следующую итерацию. Так, например, в последнем примере (см. Листинг 6.9) управляющая последовательность выглядит таким образом (Листинг 6.10):

Листинг 6.10

```
0x00000368
.   .   .
0x00000370    ADD             R0,R0,#0x00000001
0x00000374    CMP             R0,#0x0000000A
0x00000378    BCC             0x00000368
```

Здесь мы оставили только те инструкции, которые будем анализировать. В этом фрагменте инструкция **ADD** инкрементирует на 1 счетчик цикла в регистре **R0**, а инструкция **CMP** сравнивает значение в **R0** со значением 10. Если содержимое **R0** меньше 10, цикл повторяется и следующая итерация начинается с инструкции по адресу `0x00000368`.

Как мы видим, во всех случаях каждая итерация требует как минимум три управляющие инструкции (в оптимизированном варианте, который мы рассмотрим далее в этой главе, таких инструкций будет всего две, но сути дела это не меняет). Если, предположим, в одном цикле требуется пройти, скажем, 10 итераций (как в наших примерах), то количество управляющих инструкций, которые нужно выполнить, будет равно 30. Если же итераций намного больше, то это число может существенно увеличиться. Например, уже при 1000 итераций количество дополнительных инструкций, которые должны выполняться, достигнет 3000!

Очевидно, что выполнение дополнительных инструкций, особенно если таковых много, никак не способствует росту производительности приложения. Еще один скрытый нюанс, который следует принимать во внимание,

связан с конвейерным принципом выполнения инструкций в ARM процессорах (впрочем, как и во всех остальных продвинутых архитектурах). Инструкции ветвления, присутствующие в любом цикле (**BCC** в нашем примере), сбрасывают очередь в конвейере инструкций, что вынуждает микропроцессор восстанавливать и заполнять очередь, начиная с нового адреса. Это требует дополнительного времени и также сказывается на общей производительности приложения.

Для повышения производительности приложения существует методика, позволяющая уменьшить количество итераций в циклических операциях. Эта методика называется "разворачивание" (unrolling) циклов – ее суть мы сейчас рассмотрим на примере. Предположим, какая-либо повторяющаяся операция должна быть выполнена в цикле N раз, причем в каждой итерации операция выполняется (в обычном варианте) всего один раз. Если выполнить такую операцию 4 раза в одной итерации, то количество итераций уменьшится приблизительно в 4 раза, т. е. будет чуть больше или равно N/4. При этом необходимо учитывать, что число N может быть не кратным 4 – в этом случае оставшиеся итерации придется выполнить в обычном режиме.

Математически количество итераций, требуемых, например, при 4-х кратном выполнении операции за один проход, можно выразить следующей формулой:

$$N = X/4 + R,$$

где N – фактическое число итераций, X – требуемое количество итераций в цикле и R – остающееся число итераций после деления на 4.

Например, в наших предыдущих примерах выполнялось 10 итераций. Если выполнять 4 операции в одной итерации, то суммарное количество итераций будет равно 10/4 = 2 + 2 = 4. Таким образом, в модифицированном варианте цикла понадобится всего 4 итерации (2 полных и 2 неполных) вместо 10.

Разворачивание циклов особенно эффективно при большом количестве итераций. Так, например, если требуется выполнить 1990 итераций, то выполнение 4-х операций в одной итерации приведет к уменьшению числа итераций с 1990 до 497 + 2 – это существенный выигрыш в производительности! Очевидно, что для циклов с небольшим количеством итераций выигрыш в производительности будет меньше.

Перейдем к практичекому примеру использования разворачивания цикла. Создадим в среде Keil проект и включим в него файл main.cpp с исходным текстом программы, который показан в Листинге 6.11.

Листинг 6.11

```
#include  <stdio.h>
#include  <stdlib.h>

#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
```

```
int  main(void)
{
  init_serial();
  int  src[11]  =  {  1,  -9,  2,  -79,  -84,  -67,  5,  -111,  249,  -120,
42  };
  int  *psrc  =  src;

  int  dst[11];
  int  *pdst  =  dst;
  int  cnt  =  sizeof(src)/4;

  div_t  res  =  div(cnt,  4);
  int  ibase  =  res.quot;
  int  irem  =  res.rem;

  for  (int  i1  =  0;  i1  <  ibase;  i1++)
  {
    *(pdst++)  =  *(psrc++);
    *(pdst++)  =  *(psrc++);
    *(pdst++)  =  *(psrc++);
    *(pdst++)  =  *(psrc++);
  };
  if  (irem  !=  0)
    for  (int  i1  =  0;  i1  <  irem;  i1++)
      *(pdst++)  =  *(psrc++);

  pdst  -=  cnt;
  for  (int  i1  =  0;  i1  <  cnt;  i1++)
        printf("  %d  ",  *(pdst++));
  while  (1);
  }
```

Это приложение выполняет копирование 11 элементов целочисленного массива **src** в массив **dst**. Для доступа к элементам обоих массивов используются указатели. Указатель **psrc** содержит адрес первого элемента массива **src**, а **pdst** содержит адрес, куда будет скопирован первый элемент массива **dst**. Размер массива-источника (равный размеру приемника) хранится в переменной **cnt**. В данной программе мы будем выполнять 4 операции копирования в одной итерации, поэтому нам нужно вычислить два значения для счетчиков итераций.

Количество итераций с 4 операциями будет храниться в переменной **ibase**, а количество оставшихся итераций — в переменной **irem**. Для вычислений обоих значений нужно выполнить следующие операторы:

```
div_t  res  =  div(cnt,  4);
int  ibase  =  res.quot;
int  irem  =  res.rem;
```

Оператор цикла

```
for  (int  i1  =  0;  i1  <  ibase;  i1++)
{
.  .  .
}
```

выполняет копирование элементов по 4 в одной итерации. Следующий за ним оператор **if** проверяет, есть ли остающиеся элементы для копирования обычным способом. Если таковые обнаружены (**irem > 0**), то выполняется второй цикл **for**:

```
for (int i1 = 0; i1 < irem; i1++)
{
. . .
}
```

Далее приложение выводит содержимое массива **dst** в терминальное окно отладчика.

Откомпилируем наш проект и выполним его отладку. Привязки переменных программы к ресурсам микроконтроллера показаны на Рис. 6.4.

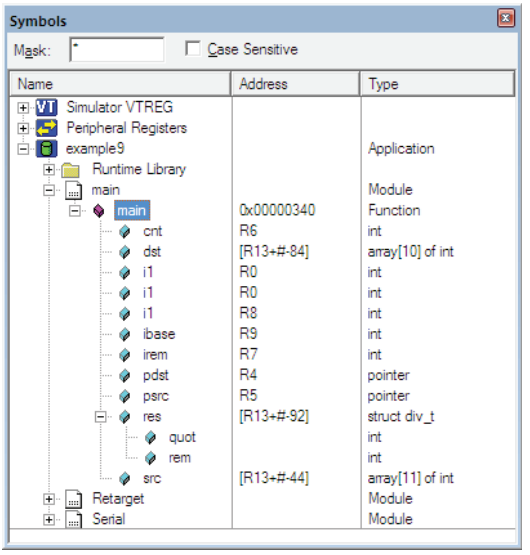

Рис. 6.4.

Как следует из таблицы, показанной в окне **Symbols**, адреса обоих массивов располагаются в регистре-указателе стека (**R13**), а значения указателей **psrc** и **pdst** размещаются в регистрах **R5** и **R4** соответственно. Счетчики циклов, присвоенные переменной **i1**, в ходе выполнения приложения будут храниться в регистрах **R0** и **R8**. Располагая этой информацией, проанализируем фрагмент дизассемблированного кода нашего приложения, представленный в Листинге 6.12.

Листинг 6.12

```
          9:  {
0x00000340    E24DD060    SUB             R13,R13,#0x00000060
         10:    init_serial();
0x00000344    EBFFFFC2    BL               init_serial(0x00000254)
```

```
        11:    int  src[11]  =  {  1,  -9,  2,  -79,  -84,  -67,  5,  -111,
249,  -120,  42  };
0x00000348    E3A0202C    MOV             R2,#0x0000002C
0x0000034C    E59F10AC    LDR             R1,[PC,#0x00AC]
0x00000350    E28D0034    ADD             R0,R13,#0x00000034
0x00000354    EB000079    BL               __rt_memcpy_w(0x00000540)
        12:    int  *psrc  =  src;
        13:
        14:    int  dst[11];
0x00000358    E28D5034    ADD             R5,R13,#0x00000034
        15:    int  *pdst  =  dst;
0x0000035C    E28D400C    ADD             R4,R13,#0x0000000C
        16:    int  cnt  =  sizeof(src)/4;
        17:
0x00000360    E3A0600B    MOV             R6,#0x0000000B
        18:    div_t  res  =  div(cnt,  4);
0x00000364    E3A02004    MOV             R2,#0x00000004
0x00000368    E1A01006    MOV             R1,R6
0x0000036C    E28D0004    ADD             R0,R13,#0x00000004
0x00000370    EB00006B    BL               $Ven$AT$I$$div(0x00000524)
        19:    int  ibase  =  res.quot;
0x00000374    E59D9004    LDR             R9,[R13,#0x0004]
        20:    int  irem  =  res.rem;
        21:
0x00000378    E59D7008    LDR             R7,[R13,#0x0008]
        22:    for  (int  i1  =  0;  i1  <  ibase;  i1++)
        23:    {
0x0000037C    E3A00000    MOV             R0,#0x00000000
0x00000380    EA000008    B                 0x000003A8
        24:          *(pdst++)  =  *(psrc++);
0x00000384    E4951004    LDR             R1,[R5],#0x0004
0x00000388    E4841004    STR             R1,[R4],#0x0004
        25:          *(pdst++)  =  *(psrc++);
0x0000038C    E4951004    LDR             R1,[R5],#0x0004
0x00000390    E4841004    STR             R1,[R4],#0x0004
        26:             *(pdst++)  =  *(psrc++);
0x00000394    E4951004    LDR             R1,[R5],#0x0004
0x00000398    E4841004    STR             R1,[R4],#0x0004
        27:             *(pdst++)  =  *(psrc++);
        28:             };
0x0000039C    E4951004    LDR             R1,[R5],#0x0004
0x000003A0    E4841004    STR             R1,[R4],#0x0004
        22:    for  (int  i1  =  0;  i1  <  ibase;  i1++)
        23:    {
        24:          *(pdst++)  =  *(psrc++);
        25:          *(pdst++)  =  *(psrc++);
        26:          *(pdst++)  =  *(psrc++);
        27:          *(pdst++)  =  *(psrc++);
        28:    };
0x000003A4    E2800001    ADD             R0,R0,#0x00000001
0x000003A8    E1500009    CMP             R0,R9
0x000003AC    BAFFFFF4    BLT             0x00000384
        29:    if  (irem  !=  0)
0x000003B0    E3570000    CMP             R7,#0x00000000
0x000003B4    0A000006    BEQ             0x000003D4
```

```
        30:      for  (int  i1  =  0;  i1  <  irem;  i1++)
0x000003B8    E3A00000    MOV             R0,#0x00000000
0x000003BC    EA000002    B                 0x000003CC
        31:        *(pdst++)  =  *(psrc++);
        32:
0x000003C0    E4951004    LDR             R1,[R5],#0x0004
0x000003C4    E4841004    STR             R1,[R4],#0x0004
0x000003C8    E2800001    ADD             R0,R0,#0x00000001
0x000003CC    E1500007    CMP             R0,R7
0x000003D0    BAFFFFFA    BLT             0x000003C0
        33:    pdst  -=  cnt;
0x000003D4    E0444106    SUB             R4,R4,R6,LSL  #2
        34:    for  (int  i1  =  0;  i1  <  cnt;  i1++)
0x000003D8    E3A08000    MOV             R8,#0x00000000
0x000003DC    EA000003    B                 0x000003F0
        35:          printf("  %d  ",  *(pdst++));
0x000003E0    E28F001C    ADD             R0,PC,#0x0000001C
0x000003E4    E4941004    LDR             R1,[R4],#0x0004
0x000003E8    EB000008    BL               $Ven$AT$I$$__2printf(0x00000410)
0x000003EC    E2888001    ADD             R8,R8,#0x00000001
0x000003F0    E1580006    CMP             R8,R6
0x000003F4    BAFFFFF9    BLT             0x000003E0
        36:    while  (1);
0x000003F8    E1A00000    NOP
0x000003FC    EAFFFFFE    B                 0x000003FC
```

Основной цикл **for**, в котором выполняются 4 операции копирования, начинается с адреса 0x00000384 и содержит 4 пары инструкций **LDR/STR**. Каждая пара инкрементирует текущий адрес для доступа к следующему элементу для считывания (**LDR**) или записи (**STR**):

```
0x00000384    E4951004    LDR             R1,[R5],#0x0004
0x00000388    E4841004    STR             R1,[R4],#0x0004
. . .
```

Адрес текущего элемента массива-источника содержится в регистре **R5**, а адрес элемента с тем же индексом в массиве-приемнике содержится в регистре **R4**. Текущее значение счетчика для данного цикла находится в регистре **R0**, а количество итераций (переменная **ibase**) помещается в регистр **R9**. Проверка окончания цикла осуществляется инструкцией **CMP**:

```
0x000003A8    E1500009    CMP             R0,R9
```

Если содержимое **R0** не превышает значения **ibase** (регистр **R9**), управление передается в начало цикла (адрес 0x00000384) инструкцией **BLT**:

```
0x000003AC    BAFFFFF4    BLT             0x00000384
```

По завершению этого цикла оставшиеся элементы обрабатываются в следующем цикле, для которого количество итераций определяется переменной **irem** (регистр **R7**). Цикл управляется следующей группой инструкций:

```
0x000003C8    E2800001    ADD             R0,R0,#0x00000001
0x000003CC    E1500007    CMP             R0,R7
0x000003D0    BAFFFFFA    BLT             0x000003C0
```

Исходный текст данного примера может быть использован для любого количества элементов в массиве-источнике и массиве-приемнике.

Возможности оптимизации приложения существенно возрастают, если для разработки критических участков программного кода использовать низкоуровневое программирование на языке ассемблера. Во многих случаях, при правильном применении этой методики можно заставить даже относительно медленные приложения работать эффективно. В следующем разделе мы рассмотрим практические примеры оптимизации программного кода приложения, написанного на C/C++, с помощью ассемблерного кода.

6.4. Оптимизация приложений с помощью языка ассемблера

Прежде, чем приступить к обсуждению данной темы. напомню, что передача параметров процедуре (любой, не только написанной на языке ассемблера), равно, как и возврат значения по завершению, подчиняется определенным правилам, которые называют соглашениями о вызовах процедур и передаче параметров (calling conventions).

Для приложений, написанных для микроконтроллеров ARM, первые четыре параметра, передаваемые процедуре, помещаются в регистры **R0**–**R3** процессора, остальные (если таковые имеются) передаются через стек. Процедура, которая возвращает значение, должна помещать результат в регистр **R0**. Передача параметров в регистрах предпочтительнее, поскольку при этом обеспечивается максимальное быстродействие, что особенно важно для часто вызываемых процедур. Во многих случаях для передачи параметров бывает достаточно регистров процессора, так что особо заботиться о производительности здесь не приходится.

Использование языка ассемблера для написания серьезных приложений требует значительных усилий и времени, поэтому в настоящее время акцент сместился в сторону применения ассемблера как очень эффективного средства для оптимизации приложений, написанных на языках высокого уровня, в частности, на C/C++. Любое приложение, написанное на языке высокого уровня, имеет так называемые критические по быстродействию участки кода, производительность выполнения которых влияет на все приложение в целом. Во многих случаях стандартные методы оптимизации таких приложений с использованием оптимизирующих настроек компилятора не дают эффекта.

В предыдущих разделах этой главы мы рассмотрели некоторые эффективные методы оптимизации приложений, в которых использовались возможности языка C++ и некоторые особенности архитектуры ARM микроконтроллеров. Еще большего эффекта можно достичь, если использовать "точечную" оптимизацию программного кода посредством разработки программного кода для критических участков на языке ассемблера.

В этом разделе будут продемонстрированы некоторые практические методы оптимизации приложений с использованием отдельных процедур на языке мак-

ро ассемблера среды разработки Keil, записанных в файлы с расширением .s.

Рассмотрим первый пример, который будет являться модифицированным вариантом приложения, выполняющего копирование файлов. В качестве прототипа возьмем файл с исходным тестом из Листинга 6.11 и модифицируем его так, чтобы часть программы использовала процедуру, написанную на языке ассемблера. Модифицированный исходный текст программы показан в Листинге 6.13.

Листинг 6.13

```
#include  <stdio.h>
#include  <stdlib.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  void  cp4(int  *src,  int  *dst,  int  cnt);

int  main(void)
{
  init_serial();
  int  src[14]  =  {  -1,  -9,  2,  -79,  -84,  -67,  5,  -111,  249,  -10,
-342,  99,  -100,  665  };
  int  *psrc  =  src;

  int  dst[14];
  int  *pdst  =  dst;
  int  cnt  =  sizeof(src)/4;

  div_t  res  =  div(cnt,  4);
  int  ibase  =  res.quot;
  int  irem  =  res.rem;

  cp4(psrc,  pdst,  ibase);

  psrc  +=  ibase  <<  2;
  pdst  +=  ibase  <<  2;
  if  (irem  !=  0)
    for  (int  i1  =  0;  i1  <  irem;  i1++)
      *(pdst++)  =  *(psrc++);

  pdst  -=  cnt;
  for  (int  i1  =  0;  i1  <  cnt;  i1++)
        printf("  %d  ",  *(pdst++));
  while  (1);
  }
```

Напомню, что программа из Листинга 6.11 демонстрировала метод оптимизации, известный как "разворачивание" цикла. В модифицированном варианте, представленном в Листинге 6.13, цикл **for**, в каждой итерации которого выполнялось 4 операции копирования, заменен процедурой **cp4**, написанной на языке ассемблера. Исходный текст этой процедуры записан в файл proc.s, который должен быть включен в состав проекта.

Процедура **cp4** объявлена как внешняя директивой **extern** и принимает три параметра: два указателя на массив-источник и массив-приемник (**src** и **dst**) и размер массива-источника (**cnt**). При вызове процедуры ей передаются фактические параметры (**psrc**, **pdst** и **ibase**). Процедура **cp4** не возвращает значений по завершению. Дополнительный спецификатор "C" указывает компоновщику, как интерпретировать процедуру при компоновке.

Следующие за **cp4** операторы

```
psrc  +=  ibase  <<  2;
pdst  +=  ibase  <<  2;
```

корректируют адреса элементов в обоих массивах для выполнения оставшихся итераций (их количество определяется значением переменной **irem** в операторе **if**). Если значение **irem** больше нуля, то оставшиеся элементы копируются в цикле **for**.

Вывод содержимого буфера-приемника в терминальное окно выполняется в последнем цикле **for**, при этом указатель **pdst** должен быть декрементирован на величину **cnt**, чтобы указывать на первый элемент массива. Эта коррекция выполняется оператором

```
pdst  -=  cnt;
```

Исходный текст ассемблерной процедуры **cp4** показан далее (Листинг 6.14).

Листинг 6.14

```
            AREA  text,  CODE,  READONLY
            EXPORT  cp4
            ENTRY
cp4         CMP      r2,  #0
            BEQ      exit
next        LDR      r3,  [r0],  #4
            STR      r3,  [r1],  #4
            LDR      r3,  [r0],  #4
            STR      r3,  [r1],  #4
            LDR      r3,  [r0],  #4
            STR      r3,  [r1],  #4
            LDR      r3,  [r0],  #4
            STR      r3,  [r1],  #4
            SUBS     r2,  r2,  #1
            BNE      next
exit        BX        lr
            END
```

Вспомним, что означают первые три директивы в исходном тексте процедуры. Директива **AREA** объявляет данную секцию как сегмент кода с доступом по чтению. Следующая за ней директива **EXPORT** объявляет символическое имя **cp4** доступным для программного кода из других объектных модулей. По умолчанию **cp4** интерпретируется как метка в секции программного кода. Директива **ENTRY** указывает на точку входа в процедуру.

Инструкция

```
CMP      r2,  #0
```

проверяет случай, когда третий параметр (размер массива, регистр **R2**) равен нулю. Если регистр **R2** содержит 0, происходит немедленный выход из процедуры по инструкции **BEQ**.

Если содержимое **R2** больше нуля, то, по крайней мере, выполнится хотя бы одна итерация, в которой будут скопированы 4 элемента массива-источника в массив-приемник. Адрес массива-источника передается процедуре в регистре **R0**, а адрес массива-приемника в **R1**. Пара инструкций

```
LDR      r3,  [r0],  #4
STR      r3,  [r1],  #4
```

выполняет выполняет копирование одного элемента. Вначале элемент источника загружается в регистр **R3** инструкцией **LDR**, при этом содержимое регистра **R0** инкрементируется на 4, чтобы указывать на адрес следующего элемента. Инструкция **STR** записывает содержимое регистра **R3** по адресу, находящемуся в регистре **R1**, при этом выполняется инкремент регистра **R1** для продвижения к следующему адресу в массиве-приемнике. Инструкция

```
SUBS     r2,  r2,  #1
```

декрементирует содержимое счетчика итераций в регистре **R2** и устанавливает соответствующие флаги в регистре состояния программы (**cpsr**). Суффикс **S** в конце мнемонического имени инструкции указывает на то, что данная инструкция по результату выполнения операции установит флаги состояния. Если в результате декремента содержимое регистра **R2** станет равным нулю, то будет установлен флаг **Z** и следующая за **SUBS** инструкция

```
BNE      next
```

будет пропущена. Если же содержимое **R2** остается положительным, то инструкция **BNE** передает управление на метку **next**, с которой начинает выполняться очередная итерация.

В процедуре **cp4** реализованы два метода оптимизации: разворачивание циклов, о котором мы уже упоминали, и оптимизация основного цикла. Об оптимизации основного цикла мы сейчас поговорим более подробно. Дизассемблированный код типичной управляющей структуры, такой, как оператор **for** или **while**, обычно включает три инструкции и может быть представлен следующим образом:

```
label        . . . . . .
                 <инструкции  цикла>
                 . . . . . .
                   ADD      rx,  rx,  #N
                   CMP      rx,  #M
                   BNE        label
                   <следующая  инструкция>
```

В соответствии с этим псевдокодом в конце итерации счетчик цикла, содержащийся в регистре **Rx**, инкрементируется на заданное число N (в боль-

шинстве случаев счетчик цикла увеличивается на 1, т. е. N = 1). Инструкция **CMP** сравнивает значение счетчика цикла с каким-то предельным значением, например, 10, как в примерах из этой главы, и затем устанвливает и/или очищает соответствующие флаги в регистре состояния программы. На практике, операция сравнения выполняет вычитание одного операнда из другого и по полученному результату устанавливает флаги. В отличие от операции вычитания, операция сравнения **CMP** не разрушет содержимое регистров, а только устанавливает/очищает флаги процессора.

Следующая за **CMP** инструкция условного ветвления **BNE** анализирует состояние флагов **Z** и **C** и, если код условия выполняется, осуществляет переход на соответствующую метку программы. Так, например, если содержимое регистра **Rx** в нашем фрагменте псевдокода будет больше или равно числа **M**, то происходит выход из цикла. В случае, если содержимое **Rx** остается меньше **M**, то условие в инструкции **BNE** выполняется и управление передается на метку **label**.

Как видно из этого фрагмента псевдокода, управление циклом выполняется тремя инструкциями. Тем не менее, можно оптимизировать программный код цикла, оставив всего две инструкции.

Для этого нужно, чтобы переменная цикла декрементировалась и сравнивалась с нулем – эти операции можно выполнить в одной инструкции **SUBS**. Напомню, что суффикс **S** в конце мнемонического обозначения инструкции разрешает установку флагов состояния по результатам выполнения инструкции.

Это свойство позволяет исключить инструкцию **CMP** из группы команд, управляющих циклом. В результате получим следующую программную конструкцию для управления циклом:

```
label       . . . . . . . . .
              <инструкции  цикла>
              . . . . . . . . .
                SUBS    rx,  rx,  #N
                BNE       label
                <следующая  инструкция>
```

Следует отметить, что управляющие операторы языка C/C++, такие как **for**, **while**, **do**...**while**, на уровне машинных инструкций сводятся к одному из двух только что рассмотренных псевдокодов, возможно, с незначительными модификациями. При реализации на языке ассемблера программный код цикла с инструкцией **SUBS** при большом количестве итерацией дает больший выигрыш в производительности по сравнению с конструкцией, в которой требуется дополнительная инструкция сравнения **CMP**.

Если по какой-либо причине все же нужно использовать инкремент переменной счетчика цикла, но оставить при этом две инструкции управления вместо трех, то можно, например, инвертировать знак счетчика цикла и выполнять проход по циклу с помощью инструкции **ADDS**. Пример исходного текста процедуры **cp4**, в которой используется инкремент счетчика цикла, показан далее (Листинг 6.15).

Листинг 6.15

```
            AREA  text,  CODE,  READONLY
            EXPORT  cp4
            ENTRY
cp4         CMP      r2,  #0
            BEQ      exit1
            RSB      r2,  r2,  #0
next        LDR      r3,  [r0],  #4
            STR      r3,  [r1],  #4
            LDR      r3,  [r0],  #4
            STR      r3,  [r1],  #4
            LDR      r3,  [r0],  #4
            STR      r3,  [r1],  #4
            LDR      r3,  [r0],  #4
            STR      r3,  [r1],  #4
            ADDS     r2,  r2,  #1
            BLE      next1
exit        BX        lr
            END
```

В модифицированном варианте процедуры **cp4** используется инструкция

```
RSB      r2,  r2,  #0
```

которая изменяет знак числа на противоположный. Таким образом, вместо инкремента от нуля до 2 (для нашей конкретной программы) мы будем использовать инкремент от −2 до 0 в инструкции

```
ADDS     r2,  r2,  #1
```

В результате цикл по-прежнему будет управляться двумя инструкциями, но вместо **SUBS** будет применяться **ADDS**.

Напомню, что большинство инструкций ассемблера микроконтроллеров ARM обладает свойством "выполнения при условии" или "условного выполнения" (conditional execution). Например, инструкция

```
ADDEQ    r1,  r1,  #1
```

выполнит инкремент регистра **R1** на 1 только при условии установленного флага **Z**. В противном случае инструкция пропускается и выполняется следующая за ней. Это свойство оказывается весьма полезным при вычислении логических выражений и минимизации количества ветвлений программного кода. В следующем разделе мы рассмотрим несколько примеров оптимизации кода с использованием инструкций условного выполнения.

6.5. Применение инструкций условного выполнения для оптимизации программных алгоритмов

Предположим, что нашей программе необходимо подсчитать количество элементов целочисленного массива, значения которых находятся между 0 и 100.

Создадим в среде Keil проект и включим в него файл main.cpp с исходным текстом, показанным в Листинге 6.16.

Листинг 6.16

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);

int  main(void)
{
  init_serial();
  int  src[11]  =  {  1,  -9,  2,  -79,  -84,  -67,  5,  -111,  249,  -120,  42  };
  int  *psrc  =  src;
  int  cnt  =  sizeof(src)/4;

  int  num  =  0;
  for  (int  i1  =  0;  i1  <  cnt;  i1++)
  {
    if  ((*psrc  >  0)  &&  (*psrc  <  100))num++;
    psrc++;
    }
  printf("  The  number  of  elements  between  0  and  100  =  %d\n",  num);
  while  (1);
  }
```

Наше приложение будет выполнять поиск соответствующих условию элементов в массиве **src**, состоящем из 11 элементов. Поиск выполняется в цикле **for**, внутри которого имеется оператор условия **if**, осуществляющий проверку условия. Элементы массива адресуются посредством указателя **psrc**. Количество обнаруженных элементов, удовлетворяющих условию ″больше 0, но меньше 100″, записывается в переменную **num**. По окончанию поиска значение **num** выводится в терминальное окно функцией **printf**.

После успешной компиляции и компоновки приложения запустим его на отладку. Таблица символических имен и присвоенных им ресурсов процессора показана на Рис. 6.5.

Как видно из таблицы, представленной в окне **Symbols** отладчика Keil, точка входа в программу **main()** (адрес первой инструкции процессора) находится по адресу **0x00000340**, адрес массива **src** располагается в регистре-указателе стека (**R13**) со смещением –44; этот же адрес будет присвоен переменной-указателю **psrc**, значение которой хранится в регистре **R4**. Текущее значение переменной цикла **i1** будет находиться в регистре **R0** микроконтроллера, значения переменных **cnt** и **num** будут располагаться в регистрах **R6** и **R5** соответственно.

Проанализируем интересующий нас фрагмент дизассемблированного кода приложения, показанный в Листинге 6.17.

Листинг 6.17

```
          7:  {
0x00000340    E24DD030    SUB             R13,R13,#0x00000030
```

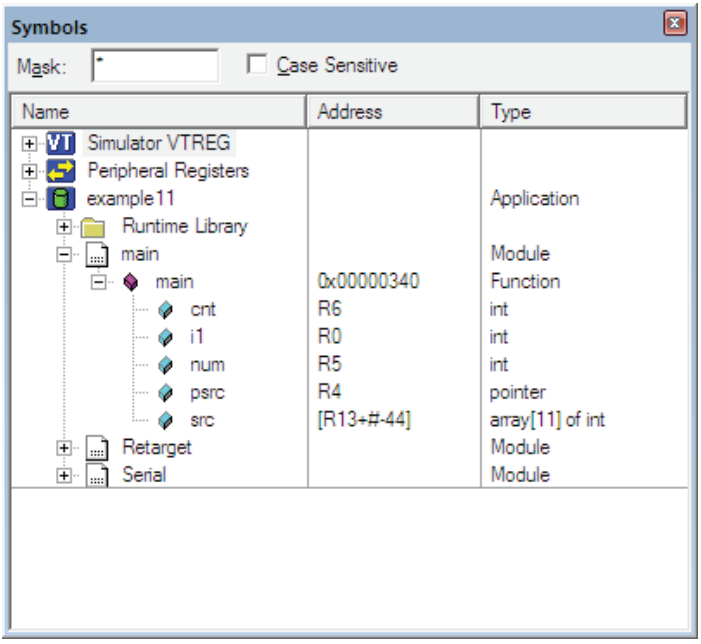

Рис. 6.5.

```
          8:    init_serial();
0x00000344    EBFFFFD4    BL                 init_serial(0x0000029C)
          9:    int  src[11]  =  {  1,  -9,  2,  -79,  -84,  -67,  5,  -111,
249,  -120,  42  };
0x00000348    E3A0202C    MOV                R2,#0x0000002C
0x0000034C    E59F1058    LDR                R1,[PC,#0x0058]
0x00000350    E28D0004    ADD                R0,R13,#0x00000004
0x00000354    EB000067    BL                  __rt_memcpy_w(0x000004F8)
         10:    int  *psrc  =  src;
0x00000358    E28D4004    ADD                R4,R13,#0x00000004
         11:    int  cnt  =  sizeof(src)/4;
         12:
0x0000035C    E3A0600B    MOV                R6,#0x0000000B
         13:    int  num  =  0;
0x00000360    E3A05000    MOV                R5,#0x00000000
         14:    for  (int  i1  =  0;  i1  <  cnt;  i1++)
         15:    {
0x00000364    E3A00000    MOV                R0,#0x00000000
0x00000368    EA000008    B                    0x00000390
         16:      if  ((*psrc  >  0)  &&  (*psrc  <  100))num++;
0x0000036C    E5941000    LDR                R1,[R4]
0x00000370    E3510000    CMP                R1,#0x00000000
0x00000374    DA000003    BLE                0x00000388
0x00000378    E5941000    LDR                R1,[R4]
0x0000037C    E3510064    CMP                R1,#0x00000064
0x00000380    AA000000    BGE                0x00000388
0x00000384    E2855001    ADD                R5,R5,#0x00000001
```

```
        17:      psrc++;
        18:      }
0x00000388    E2844004    ADD             R4,R4,#0x00000004
0x0000038C    E2800001    ADD             R0,R0,#0x00000001
0x00000390    E1500006    CMP             R0,R6
0x00000394    BAFFFFF4    BLT             0x0000036C
        19:    printf(" The number of elements between 0 and 100 =
%d\n", num);
0x00000398    E1A01005    MOV             R1,R5
0x0000039C    E28F000C    ADD             R0,PC,#0x0000000C
0x000003A0    EB00000F    BL               $Ven$AT$I$$__2printf(0x000003E4)
        20:    while (1);
0x000003A4    E1A00000    NOP
0x000003A8    EAFFFFFE    B                 0x000003A8
```

Посмотрим, как выполняется оператор **if** нашего приложения на уровне инструкций процессора. Первые две инструкции в начале оператора **if**

```
0x0000036C    E5941000    LDR             R1,[R4]
0x00000370    E3510000    CMP             R1,#0x00000000
```

выполняют сравнение текущего элемента массива, загруженного в регистр **R4**, с нулем. Если значение элемента оказывается меньше 0, то нет смысла проверять второе условие (***psrc < 100**), поэтому следующая инструкция

```
0x00000374    DA000003    BLE             0x00000388
```

передает управление за пределы оператора **if** по адресу **0x00000388**. Если же первое условие (***psrc > 0**) выполнено, то инструкция **BLE** пропускается и следующая за ней пара инструкций

```
0x00000378    E5941000    LDR             R1,[R4]
0x0000037C    E3510064    CMP             R1,#0x00000064
```

проверяет второе условие (***psrc < 100**). Если это условие не выполнено, то следующая инструкция

```
0x00000380    AA000000    BGE             0x00000388
```

передает управление по адресу **0x00000388**. Инструкция

```
0x00000388    E2844004    ADD             R4,R4,#0x00000004
```

которая находится по этому адресу, инкрементирует адрес в регистре **R4** для доступа к следующему элементу массива **src**.

Если и второе условие (***psrc < 100**) оказывается выполненным, то следующая после **BGE** инструкция

```
0x00000384    E2855001    ADD             R5,R5,#0x00000001
```

инкрементирует на 1 счетчик обнаруженных элементов массива, удовлетворяющих обоим условиям.

При выполнении приложение выводит в виртуальное окно количество обнаруженных элементов, находящихся в диапазоне 0 — 100:

```
The number of elements between 0 and 100 = 4
```

Используя отдельную процедуру, написанную на языке ассемблера, можно существенно упростить как анализ самих условий в операторе условия **if**, так и управление циклом **for**.

Модифицируем наше приложение таким образом, чтобы весь цикл **for** выполнялся внутри ассемблерной процедуры (назовем ее **searchp**).

Эта процедура будет принимать 4 параметра: адрес массива **src** (регистр **R0**), размер массива (регистр **R1**), нижнюю границу диапазона проверки (регистр **R2**) и верхнюю границу диапазона проверки (регистр **R3**). Процедура будет возвращать количество элементов, удовлетворяющих критерию поиска, в регистре **R0**.

Исходный текст процедуры **searchp** представлен в Листинге 6.18.

Листинг 6.18

```
                AREA  text,  CODE,  READONLY
                EXPORT  searchp
                ENTRY
searchp         MOV       r4,  #0
next            LDR       r3,  [r0],  #4
                TEQ       r3,  #0
                RSBGTS    r3,  r3,  #100
                BLT       next
                ADDGT     r4,  r4,  #1
                SUBS      r1,  r1,  #1
                BNE       next
                MOV       r0,  r4
                BX         lr
                END
```

Поскольку процедура **searchp** должна быть доступна из других объектных модулей, она объявляется с директивой **EXPORT**. Регистр **R4** в нашей процедуре будет выполнять функцию счетчика обнаруженных элементов из диапазона 0–100. Аналогом оператора цикла **for** из предыдущего примера является группа инструкций между меткой **next** и инструкцией

```
BNE       next
```

которая является последней инструкцией цикла. Эта инструкция вместе с инструкцией

```
SUBS      r1,  r1,  #1
```

выполняют управление циклом, счетчик которого находится в регистре **R1**. В начале каждой итерации очередной элемент массива, адресуемого регистром **R0**, помещается в регистр **R3** инструкцией

```
LDR       r3,  [r0],  #4
```

Следующая за **LDR** инструкция

```
TEQ       r3,  #0
```

проверяет содержимое **R3** на равенство 0 и устанавливает соответствующие флаги в регистре состояния. Инструкция

```
RSBGTS      r3,  r3,  #100
```

выполняет несколько действий. Во-первых, эта инструкция будет выполнена только в том случае, если содержимое регистра **R3** оказалось положительным и были установлены соответствующие флаги инструкцией **TEQ**. Во-вторых, инструкция **RSBGTS** вычитает содержимое регистра **R3** из 100 и по результату операции устанавливает флаги в регистре состояния.

Если результат вычитания оказался положительным, то следующая инструкция **BLT** не будет выполнена, а счетчик обнаруженных элементов получит приращение на 1 после выполнения инструкции

```
ADDGT       r4,  r4,  #1
```

Следующие за **ADDGT** две инструкции выполняют управление циклом, о чем мы уже упоминали. По завершению цикла содержимое счетчика элементов (регистр **R4**) копируется в **R0** инструкцией

```
MOV         r0,  r4
```

после чего процедура завершается инструкцией

```
BX          lr
```

как обычно.

Как следует из исходного текста процедуры **searchp**, количество ветвлений здесь уменьшилось на единицу по сравнению с тем, что мы видели в Листинге 6.17 для оператора цикла **for**, когда анализировали дизассемблированный код предыдущего примера. При большом количестве итераций применение процедуры на языке ассемблера обеспечит существенный выигрыш в производительности приложения в целом.

Исходный текст модифицированного C++ приложения (файл main.cpp) теперь будет выглядеть следующим образом (Листинг 6.19).

Листинг 6.19

```
#include  <stdio.h>
#include  <LPC214X.H>

extern  "C"  void  init_serial(void);
extern  "C"  int  searchp(int  *src,  int  ssize,  int  highlimit);
int  main(void)
{
  init_serial();
  int  src[11]  =  {1,  -9,  2,  -79,  -84,  -67,  5,  -111,  249,  -120,  42  };
  int  *psrc  =  src;
  int  cnt  =  sizeof(src)/4;

  int  num  =  searchp(psrc,  cnt,  100);

  printf("  The  number  of  elements  between  0  and  100  =  %d\n",  num);
  while  (1);
  }
```

Поскольку мы будем использовать внешнюю по отношению к данному модулю процедуру **searchp**, ее нужно объявить соответствующим образом:

```
extern "C" int searchp(int *src, int ssize, int highlimit);
```

Процедура принимает три параметра: адрес массива целых чисел (переменная-указатель **src**), размер массива (переменная **ssize**) и значение верхней границы диапазона поиска (переменная **highlimit**). После выполнения оператора

```
int num = searchp(psrc, cnt, 100);
```

переменная **num** будет содержать количество обнаруженных элементов, находящихся в диапазоне 0 – 100.

После успешной компиляции проекта выполним отладку нашего приложения. Перед тем, как анализировать дизассемблированный код приложения, нам нужно получить информация о назначениях всех символических имен приложения ресурсам процессора (окно **Symbols**, Рис. 6.6).

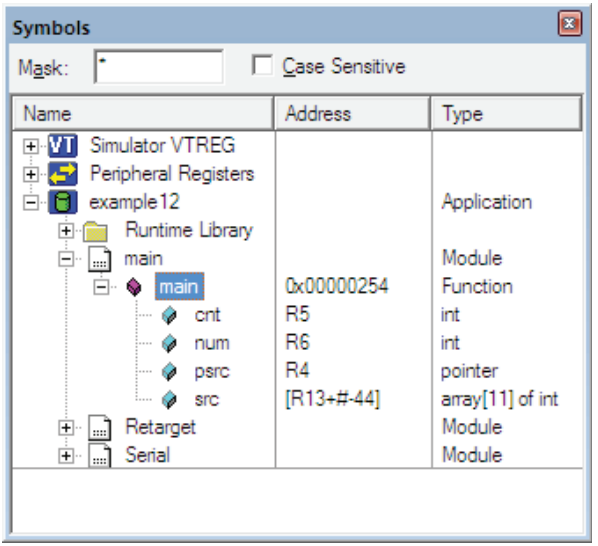

Рис. 6.6.

Поскольку основные операции выполняются в процедуре **searchp**, то количество переменных, задействованных в основной программе, уменьшилось. Дизассемблированный код приложения существенно упростился (Листинг 6.20).

Листинг 6.20

```
          7:  {
0x00000254    E24DD030    SUB             R13,R13,#0x00000030
          8:    init_serial();
0x00000258    EB00002E    BL               init_serial(0x00000318)
```

```
          9:    int  src[11]  =  {1,  -9,  2,  -79,  -84,  -67,  5,  -111,
249,  -120,  42  };
0x0000025C    E3A0202C    MOV             R2,#0x0000002C
0x00000260    E59F1034    LDR             R1,[PC,#0x0034]
0x00000264    E28D0004    ADD             R0,R13,#0x00000004
0x00000268    EB000099    BL               __rt_memcpy_w(0x000004D4)
         10:    int  *psrc  =  src;
0x0000026C    E28D4004    ADD             R4,R13,#0x00000004
         11:    int  cnt  =  sizeof(src)/4;
         12:
0x00000270    E3A0500B    MOV             R5,#0x0000000B
         13:    int  num  =  searchp(psrc,  cnt,  100);
         14:
0x00000274    E3A02064    MOV             R2,#0x00000064
0x00000278    E1A01005    MOV             R1,R5
0x0000027C    E1A00004    MOV             R0,R4
0x00000280    EB000122    BL               searchp(0x00000710)
0x00000284    E1A06000    MOV             R6,R0
         15:    printf("  The  number  of  elements  between  0  and  100  =
%d\n",  num);
0x00000288    E1A01006    MOV             R1,R6
0x0000028C    E28F000C    ADD             R0,PC,#0x0000000C
0x00000290    EB00004A    BL               $Ven$AT$I$$__2printf(0x000003C0)
         16:    while  (1);
0x00000294    E1A00000    NOP
0x00000298    EAFFFFFE    B                 0x00000298
```

Обратите внимание на то, как передаются параметры процедуре **searchp**:

```
0x00000274    E3A02064    MOV             R2,#0x00000064
0x00000278    E1A01005    MOV             R1,R5
0x0000027C    E1A00004    MOV             R0,R4
0x00000280    EB000122    BL               searchp(0x00000710)
```

В регистре **R0** процедуре передается адрес массива (см. Рис. 6.6), в регистре **R1** будет находиться значение счетчика цикла **cnt** и, наконец, в регистр **R2** помещается непосредственное значение **0x00000064** (100). По завершению процедура **searchp** возвращает значение в регистре **R0**, которое помещается в регистр **R6** для дальнейшего использования в программе.

Этим примером мы завершаем обсуждение темы оптимизации приложений для ARM микроконтроллеров, разработанных в среде программирования Keil. Следует отметить, что все принципы и подходы к анализу и оптимизации программного кода остаются одинаковыми для всех инструментальных средств разработки для ARM микроконтроллеров. По этой причине, если вы разрабатываете программный код, скажем, в среде IAR Workbench, вы сможете практически без изменений применить все изложенные выше методы оптимизации в ваших разработках, включая исходные тексты программ, разумеется с учетом того, что некоторые директивы в IAR имеют другие имена. Например, вместо директивы **EXPORT**, используемой в Keil macro assembler, в среде IAR следует применить ее аналог **PUBLIC**. Все исходные тексты, разработанные в Keil, с минимальными изменениями могут использоваться в среде IAR Workbench.

ЗАКЛЮЧЕНИЕ

В этой книге были рассмотрены базовые аспекты эффективного программирования для микроконтроллеров, базирующихся на ядре ARM7. Несмотря на то, что основной упор был сделан именно на ARM7, описанные методики программирования подходят и для более современных типов процессоров на базе ARM, таких как ARM9, ARM11 и Cortex. Новые типы процессоров содержат дополнительные функциональные узлы и включают дополнительные машинные инструкции для улучшения производительности, но базовые принципы функционирования во многом наследуются от ядра ARM7.

Кроме того, все примеры исходных текстов программ, описанные в книге, могут быть легко адаптированы для компиляции в среде разработки, отличной от Keil, например, в IAR Workbench. При переносе исходных текстов из Keil в другую среду программирования необходимо, как правило, заменить директивы, специфичные для Keil, на соответствующие им ключевые слова из этой среды и следовать правилам оформления исходных текстов (это в особенности касается ассемблерных процедур).

Автор надеется, что книга окажет существенную помощь разработчикам-практикам в решении задач программирования ARM-систем.

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «АЛЬЯНС БУКС» наложенным платежом, выслав открытку или письмо по почтовому адресу: **123242, Москва, а/я 20** или по электронному адресу: **orders@alians-kniga.ru**.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в Internet-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. **(495) 258-91-94, 258-91-95**;

Электронный адрес **books@alians-kniga.ru**.

Юрий Степанович Магда

Программирование и отладка C/C++ приложений для микроконтроллеров ARM

Главный редактор *Мовчан Д. А.*
dm@dmk-press.ru
Корректор *Синяева Г. И.*
Верстка *Паранская Н. В.*
Дизайн обложки *Мовчан А. Г.*

Подписано в печать 17.07.2011. Формат 70×100 $^{1}/_{16}$.
Гарнитура «Ньютон». Печать офсетная.
Усл. печ. л. 13,65. Тираж 500 экз.
№

Web-сайт издательства: www.dmk-press.ru

Программирование и отладка C/C++ приложений для микроконтроллеров ARM

В книге рассмотрены практические аспекты программирования приложений для популярной микропроцессорной платформы ARM.

Материал книги имеет сугубо практическое направление, поэтому в ней приведено множество примеров, иллюстрирующих те или иные подходы при создании программ. Основной упор сделан на практические методы программирования задач на языке программирования C/C++, а также на решение проблем при отладке программ. Создание эффективного программного кода невозможно без применения тех или иных механизмов оптимизации, начиная с разработки эффективного кода в C++ и заканчивая низкоуровневой оптимизацией на уровне команд процессора, поэтому значительная часть материала книги посвящена практическим методам оптимизации приложений.

Для разработки, отладки и оптимизации демонстрационных приложений книги используется свободно распространяемая версия инструментального пакета фирмы Keil, при этом не требуется покупка каких-либо дополнительных аппаратных модулей с микроконтроллерами ARM.

Книга будет полезной в первую очередь разработчикам программного обеспечения систем на базе микроконтроллеров ARM, инженерам, студентам и всем, кто интересуется созданием устройств с ARM микроконтроллерами.

Internet-магазин:
www.alians-kniga.ru
Книга - почтой:
e-mail: orders@alians-kniga.ru
Оптовая продажа:
"Альянс-книга"
(495)258-9194, 258-9195
e-mail: books@alians-kniga.ru

ДМК ПРЕСС
ИЗДАТЕЛЬСТВО

ISBN 978-5-94074-745-1
9 785940 747451