Ядро Cortex-M3 компании ARM
Полное руководство

Joseph Yiu

THE DEFINITIVE GUIDE TO THE ARM CORTEX-M3

Newnes

СЕРИЯ
МИРОВАЯ ЭЛЕКТРОНИКА

Джозеф Ю

ЯДРО CORTEX-M3 КОМПАНИИ ARM

Полное руководство

Перевод А. В. Евстифеева

Москва
ДМК Пресс, Додэка-XXI
2015

УДК 004.31(035.3)
ББК 32.973-04я81
Ю11

Данное издание подготовлено при участии компании «Компэл» и российского представительства компании Texas Instruments. На сайтах www.compel.ru и www.ti.com/ru вы можете получить консультацию, а также заказать бесплатные образцы микросхем.

Телефон горячей линии технической поддержки TI +7-495-981-07-01.

Ю, Джозеф.

Ю11 Ядро Cortex-M3 компании ARM. Полное руководство / Джозеф Ю ; пер. с англ. А. В. Евстифеева. — М. : Додэка-XXI, 2012. — 552 с. : ил. — (Мировая электроника). — Доп. тит. л. англ. — ISBN 978-5-97060-307-9.

Настоящая книга представляет собой исчерпывающее руководство по новому 32-битному процессору компании ARM — Cortex-M3. В данном руководстве подробно описана архитектура процессорного ядра Cortex-M3 и его подсистемы памяти. Также подробно рассмотрены остальные узлы процессора, в том числе контроллер векторных прерываний NVIC, модуль защиты памяти MMU и разнообразные компоненты отладки. Приводится детальное описание новой системы команд Thumb-2, поддерживаемой данным процессором.

Книга содержит большое число примеров программного кода как на языке Си, так и на ассемблере.

Это руководство должно присутствовать на столе любого разработчика, использующего в своей работе микроконтроллеры с ядром Cortex-M3. Полнота и ясность изложения материала книги также позволяет рекомендовать её студентам соответствующих специальностей и подготовленным радиолюбителям.

УДК 004.31(035.3)
ББК 32.973-04я81

Книга «Ядро Cortex-M3 компании ARM. Полное руководство» Дж. Ю подготовлена и издана по договору с Elsevier Inc., 30 Corporste Drive, 4th Floor, Burlington, MA 01803, USA.

ISBN 978-1-85617-963-8 (англ.)
ISBN 978-5-94120-243-0 (Додэка)
ISBN 978-5-97060-307-9 (ДМК Пресс)

СОДЕРЖАНИЕ

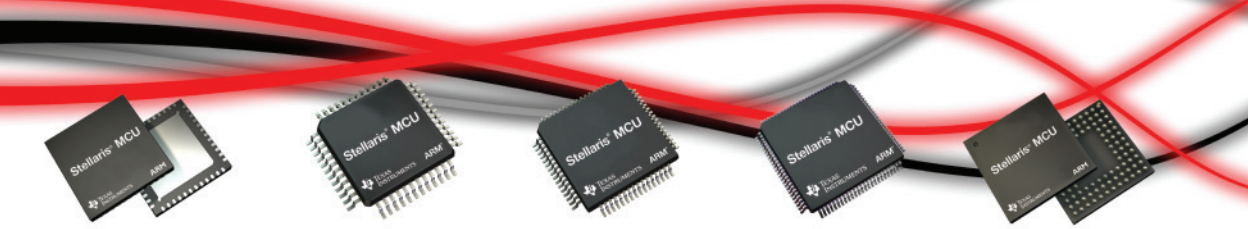
Четыре основных преимущества Stellaris®

1. Расширенные коммуникационные возможности: встроенные 10/100 Ethernet MAC/PHY, CAN, USB-контроллеры
2. DSP-инструкции и модуль вычислений с плавающей точкой (FPU) (Cortex-M4F) NEW
3. Простота разработки с использованием библиотек верхнего уровня API Stellaris Peripheral Driver Library для программирования встроенных периферийных модулей
4. Возросшая энергоэффективность за счет нового техпроцесса 65 нм (Cortex-M4F) NEW

Москва
Тел.: (495) 995-0901
E-mail: ti@compel.ru

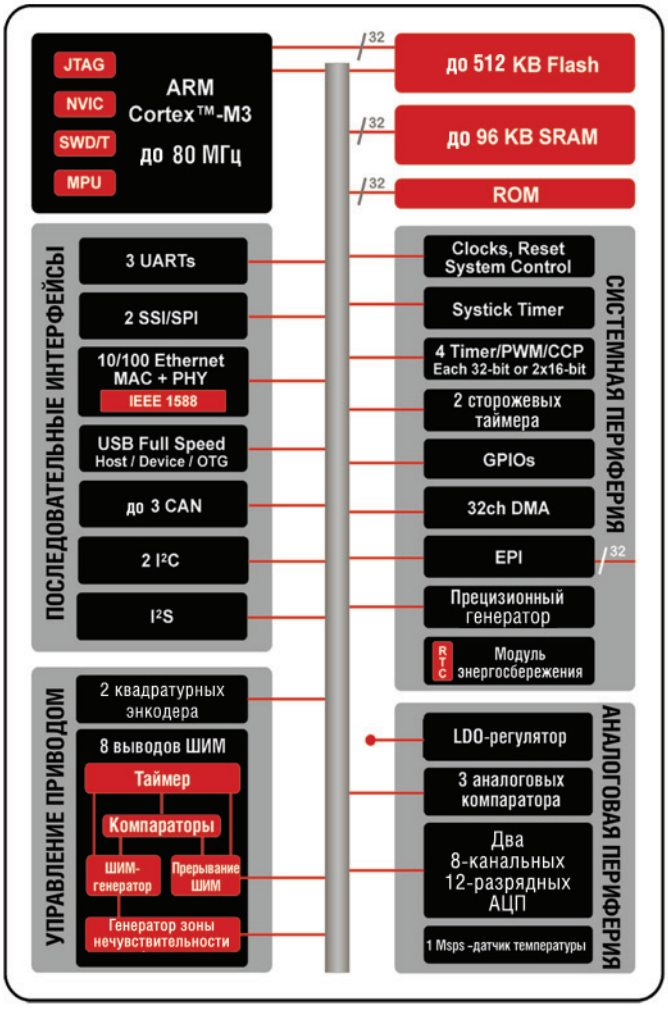
Санкт-Петербург
Тел.: (812) 327-9404
E-mail: ti@compel.ru

Компэл
www.compel.ru



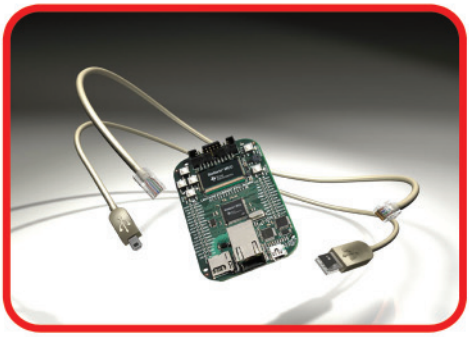
Cortex-M3 от Texas Instruments – многообразие интерфейсов

TEXAS INSTRUMENTS



КЛЮЧЕВЫЕ ОСОБЕННОСТИ:

- Производительность до 100 MIPS
- Загруженные в ROM библиотека драйверов периферии и таблицы AES и CRC
- Физический уровень Ethernet на кристалле
- BootLoader для обновления пользовательского ПО, том числе для Ethernet
- 32-разрядная шина для подключения внешней периферии и памяти
- Усовершенствованный алгоритм управления приводом
- Три аналоговых компаратора
- Два 8-канальных 12-разрядных АЦП



Москва
Тел.: (495) 995-0901
E-mail: ti@compel.ru

Санкт-Петербург
Тел.: (812) 327-9404
E-mail: ti@compel.ru

Компэл
www.compel.ru

Решения по питанию для микроконтроллеров на базе ядра Cortex-M3/Cortex-M4

Texas Instruments

Семейство Stellaris от Texas Instruments – ведущее в отрасли микроконтроллеров реального времени (MCU), основанных на революционной Cortex-M3/M4 технологии от ARM.

Управление питанием для типовых применений

Высокая эффективность

TPS62237
500 мА
DC/DC-конвертор
Фикс. $U_{вых.}$ 3,3 В

- Размер решения 12 мм²
- Высокий PSRR (до 90 дБ)
- Режимы энергосбережения на малой нагрузке

Ультранизкое входное напряжение

TPS61201
600 мА
Повышающий конвертор
Фикс. $U_{вых.}$ 3,3 В

- Входное напряжение от 0,3 В
- Автоматическое переключение между режимами повышающего и понижающего преобразователя
- Режимы энергосбережения на малой нагрузке

Широкий диапазон входного напряжения

TPS63001
800 мА
Повышающе-понижающий DC/DC-конвертор

- Диапазон входного напряжения: 1,8 – 5,5 В
- До 96% КПД
- Режимы энергосбережения на малой нагрузке

Широчайший диапазон входного напряжения

TPS54231
2000 мА
DC/DC-конвертор
Фикс. $U_{вых.}$ 3,3 В

- Диапазон входного напряжения: 3,5 – 28 В
- Расширенный температурный диапазон: 150°C
- Режимы энергосбережения на малой нагрузке

Высокоинтегрированное решение по управлению питанием

TPS65000x
3-канальная DC/DC-система питания

- Диапазон входного напряжения: 2,3 – 6 В
- Один 600мА DC/DC
- Два 200 мА LDO
- Функция «размытия» частоты тактирования для уменьшения ЭМИ

Высокое входное напряжение

TPS62140/150
2/1 А
DC/DC-конвертор
Настр. $U_{вых.}$

- Диапазон входного напряжения: 3,0 –17 В
- КПД до 95%
- Ток потребления < 20 мА

Простейшее решение

TPS7A8001
1000 мА
LDO
Настр. $U_{вых.}$

- Лучший в индустрии PSRR
- Достаточно конденсатора 4,7 мкФ
- Корпус 3x3-мм SON-8

Решение для питания через Ethernet (POE)

TPS23753
DC/DC-конвертор с упрвлением по току

- Совместим с IEEE 802.3
- Интегрированный изолированный конвертор 13 Вт
- Поддержка O-ring диода

Устройство	Uвх(В)	Iвых(мА)	Описание	Корпус
TPS7A8001	2.2 – 5.5	1000	Малошумящий шумоподавляющий в широкой полосе частот (PSRR)	SON-8
TPS23753	36 – 60	1000	Интерфейс питания через Ethernet с изолированным конвертором	TSSOP-14
TPS62237	2.05 – 6.0	500	КПД до 94%, 3 МГц понижающий конвертер	1x1,5x0,6 SON-6
TPS61201	0.3 – 5.5	600	Повышающий конвертер с 0,3 В входного и 3,3 В выходного напряжениями	3x3 SON-10
TPS65000x	2.3 – 6.0	600/200/200	3-канальная DC/DC-система питания , 2.25 МГц DC-DC и два LDO	3x3 QFN
TPS62150/140	3.0 – 17	1000	КПД до 95%, 3МГц понижающий конвертер	3x3 16-QFN
TPS63001	1.8 – 5.5	1500	КПД 96%, повышающе-понижающий конвертор	3x3 SON-10
TPS54231	3.5 – 28	2000	Понижающий конвертор с Эко-режимом и расширенным температурным диапазоном	SOIC-8

Для заказа образцов, демонстрационных плат и референс-дизайнов обращайтесь на сайт www.compel.ru или по электронной почте: ti@compel.ru

Москва
Тел.: (495) 995-0901
E-mail: ti@compel.ru

Санкт-Петербург
Тел.: (812) 327-9404
E-mail: ti@compel.ru

Компэл
www.compel.ru

ВСТУПИТЕЛЬНОЕ СЛОВО

С момента выхода первого издания книги прошло не так уж много времени, а темпы развития сообщества пользователей микроконтроллеров с процессорами ARM уже превзошли самые смелые ожидания. Безо всякого преувеличения можно сказать, что продукция нашей компании произвела настоящую революцию в мире микроконтроллеров. На сегодняшний день в мире насчитывается тысячи и тысячи конечных пользователей микроконтроллеров, построенных на процессорах ARM, что даёт все основания считать данную технологию наиболее быстро развивающейся из представленных на рынке. Поэтому второе издание книги Джозефа, содержащее наиболее актуальную информацию о данной технологии МК, появилось как нельзя вовремя.

О развитии сообщества можно судить по таким фактам, как увеличение числа компаний, предлагающих свои изделия на базе процессора Cortex-M3 (на сегодняшний день насчитывается более 30 таких компаний), разработка стандарта CMSIS, облегчающего перенос приложений как между различными вариантами процессора Cortex, так и между устройствами разных производителей, а также появление более совершенных средств разработки. Нельзя не упомянуть и о выпуске процессора Cortex-M0, который открыл перед микроконтроллерами ARM нишу чрезвычайно дешёвых устройств.

Всё это свидетельствует о наступлении эры встраиваемых систем на базе процессора Cortex-M3!

Ричард Йорк (Richard York)
Руководитель подразделения маркетинга продукции, компания ARM

ВСТУПИТЕЛЬНОЕ СЛОВО

Люди, пишущие программы для микроконтроллеров, в чём-то подобны божествам. Подчиняя микроконтроллеры своей воле, они вдыхают жизнь в застывшие конструкции и в итоге создают фантастические изделия. Далеко не последнюю роль в этом акте творения играют средства разработки — вот почему в группу, основной задачей которой было упрощение и в то же время усовершенствование процессора ARM7TDMI, помимо разработчиков ЦПУ, вошли специалисты отдела разработки программных средств компании ARM.

В результате такого совместного творчества на свет появился процессор Cortex™-M3, явивший собой потрясающее развитие оригинальной архитектуры ARM. Новый процессор органично сочетает в себе все преимущества 32-битной архитектуры ARM с поддержкой чрезвычайно эффективного набора команд Thumb-2, обеспечивая при этом ряд новых возможностей. Однако, несмотря на все усовершенствования, процессор Cortex-M3 сохранил упрощённую модель программирования, которая хорошо знакома всем приверженцам архитектуры ARM.

Уэйн Лайонз (Wayne Lyons)
Руководитель подразделения встраиваемых решений, компания ARM

ВСТУПИТЕЛЬНОЕ СЛОВО

Сегодня многие российские разработчики и специалисты хорошо знакомы или начинают знакомиться с продукцией компании ARM, в том числе с новыми продуктами серии Cortex (M0, M3, M4…). На все вопросы, связанные с преимуществами архитектуры ядра Cortex-M3, призвана ответить данная книга. Это первый и пока единственный технический материал на русском языке, рассказывающий о данной архитектуре, выпущенный при содействии компаний Texas Instruments и КОМПЭЛ.

В предисловиях автора и сотрудников компании ARM говорится о тенденциях в мире микроконтроллеров и актуальных темах, связанных с архитектурой ядра. В свою очередь, я бы хотела сфокусировать внимание непосредственно на компании Texas Instruments, которая также использует продукты ARM в своих разработках, в том числе — в микроконтроллерах на ядре Cortex-M3 (семейство Stellaris, см. Приложение И).

Итак, Texas Instruments (TI) — один из самых крупных производителей полупроводниковых компонентов с номенклатурой более 80 000 наименований, которая значительно расширилась в 2011 в связи с приобретением компании National Semiconductor.

Компания TI была основана в 1930 г. и изначально занималась сейсмографической разведкой нефти, но уже с 1952 г. переориентировалась на электронику, а в 1958 г. сотрудник TI Джек Килби изобрел первую в мире интегральную микросхему. С этого момента началась новая эпоха в развитии электроники. Компания TI самостоятельно разрабатывала микроконтроллеры и цифровые сигнальные процессоры. Тем не менее, важной особенностью развития бизнеса TI была и покупка других фирм, в основном в области аналоговых компонентов. Это помогло компании вырасти из нишевой в гиганта с широчайшей номенклатурой полупроводниковых компонентов. Из самых значимых приобретений — Silicon Systems в 1996 г., Unitrode и Power Trends в 1999 г., Burr-Brown в 2000 г., Chipcon в 2007 г., Luminary Micro в 2009 г. и самая большая покупка на сегодня — компания National Semiconductor в 2011 г. Последние приобретения открывают для TI новые технологии и продукты. Например, с покупкой Luminary Micro компания приобрела микроконтроллеры семейства Stellaris на базе ядра Cortex-M3. Его описанию посвящено отдельное приложение в рамках этого издания. TI не останавливается просто на покупке: с момента присоединения Luminary Micro линейка продуктов продолжает расти. Совсем недавно анонсирована линейка новых микроконтроллеров LM4F на базе ядра Cortex-M4. Данная архитектура расширила семейство Stellaris, которое получило не только новые вычислительные возможности, но и фирменную технологию производства Texas Instruments. Топологической нормой для производства новых контроллеров стала отработанная в течение последних пяти лет 65-нанометровая технология. В результате стало возможным достичь небывалого компромисса между производительностью и энергопотреблением.

Надеюсь, эта книга станет для вас не просто настольным справочником, а настоящим помощником в работе с микроконтроллерами, сделанными на базе ядра Cortex-M3 от компании ARM.

Мария Рудяк
Руководитель направления по работе с продукцией Texas Instruments
КОМПЭЛ

ПРЕДИСЛОВИЕ АВТОРА

Данная книга предназначена как для разработчиков, так и для программистов, заинтересовавшихся процессором Cortex™-M3 компании ARM. Разумеется, в официальных документах, таких как «*Cortex-M3 Technical Reference Manual*» и «*ARMv7-M Architecture Application Level Reference Manual*», содержится практически вся информация по этому процессору. Однако указанные документы излишне подробны и могут оказаться слишком сложными для понимания.

Эта же книга писалась в расчёте на программистов, разработчиков встраиваемых устройств, разработчиков систем на кристалле, радиолюбителей, учёных — в общем, самых разных людей, изучающих процессор Cortex-M3 и хоть в какой-то мере знакомых с микроконтроллерами либо микропроцессорами. В книге достаточно подробно рассматриваются архитектура процессора Cortex-M3, набор команд с примерами использования некоторых из них, различные аппаратные возможности, а также развитая система отладки процессора. Кроме того, в книге также приведены примеры программ, позволяющие читателю освоить азы разработки ПО для процессора Cortex-M3 с использованием инструментариев ARM и GNU. Эта книга также пригодится разработчикам, переносящим свои проекты с процессора ARM7TDMI на Cortex-M3, поскольку описывает как различия между двумя указанными процессорами, так и собственно процесс переноса прикладных программ с процессора ARM7TDMI на Cortex-M3.

Благодарности

Прежде всего, я хотел бы поблагодарить всех тех, кто своими советами, консультациями и отзывами оказал мне огромную помощь в написании первого и второго изданий книги: Ричарда Йорка (Richard York), Эндрю Фрейма (Andrew Frame), Рейнхарда Кейла (Reinhard Keil), Ника Сампейза (Nick Sampays), Дэва Банерджи (Dev Banerjee), Роберта Бойза (Robert Boys), Доминика Паджака (Dominic Pajak), Алана Трингхэма (Alan Tringham), Стивена Теобальда (Stephen Theobald), Дэна Брука (Dan Brook), Дэвида Браша (David Brash), Гайдна Поуви (Haydn Povey), Гэри Кэмпбелла (Gary Campbell), Кевина Макдермотта (Kevin McDermott), Ричарда Ирншоу (Richard Earnshaw), Шияма Садасивана (Shyam Sadasivan), Саймона Краске (Simon Craske), Саймона Аксфорда (Simon Axford), Такаши Угаджина (Takashi Ugajin), Уэйна Лайонза (Wayne Lyons), Самина Иштиака (Samin Ishtiaq) и Саймона Смита (Simon Smith).

Я хотел бы особо поблагодарить Яна Белла (Ian Bell) и Джейми Бреттль (Jamie Brettle) из компании National Instruments за помощь в написании главы, посвящённой пакету LabVIEW, и за их поддержку. Также я хотел бы выразить мою признательность Карлосу О'Донеллу (Carlos O'Donell), Брайану Баррере (Brian Barrera) и Дэниелу Якобовицу (Daniel Jakobowitz) из компании CodeSourcery за их поддержку и помощь в подборе материалов, касающихся разработки ПО в пакете Sourcery G++. И, конечно же, огромное спасибо всем сотрудникам издательства Elsevier за их профессионализм, проявленный при подготовке данной книги к публикации.

Наконец, я хотел бы высказать благодарность Питеру Коулю (Peter Cole) и Ивану Ярдли (Ivan Yardley) за их постоянную поддержку и заинтересованность в этом проекте.

ОБОЗНАЧЕНИЯ

В данной книге используются следующие обозначения и правила оформления:

Обычный ассемблерный код

```
MOV R0, R1; Копируем содержимое регистра R1 в регистр R0
```

Ассемблерный код с использованием обобщённого синтаксиса

Элементы, обозначенные угловыми скобками, необходимо заменить названиями регистров:

```
MRS <reg>, <special_reg>
```

Тексты программ на языке Си

```
for (i=0;i<3;i++) { func1(); }
```

Псевдокод

```
if (a > b) { ...
```

Значения

1. 4'hC, 0x123 — шестнадцатеричные значения.
2. #3 — элемент №3 (например, IRQ #3 означает IRQ с номером 3).
3. `#immed _ 12` — 12-битное непосредственное значение (константа).

Биты регистров

Обычно используются для указания части содержимого регистра; например, запись «биты [15:12]» относится к битам с 15-го по 12-й.

Доступность битов регистров обозначается следующим образом:

1. R — доступен только для чтения.
2. W — доступен только для записи.
3. R/W — доступен для чтения и для записи.
4. R/Wc — доступен для чтения, при записи сбрасывается.

ГЛОССАРИЙ

ADK	AMBA Design Kit Набор разработки AMBA
AHB	Advanced High-Performance Bus Усовершенствованная высокопроизводительная шина (шина AHB)
AHB-AP	AHB Access Port Порт доступа к шине AHB
AMBA	Advanced Microcontroller Bus Architecture Усовершенствованная шинная архитектура для микроконтроллеров
APB	Advanced Peripheral Bus Усовершенствованная шина периферии (шина APB)
ARM ARM	ARM Architecture Reference Manual Справочное руководство по архитектуре ARM
ASIC	Application-Specific Integrated Circuit Заказная интегральная схема
ATB	Advanced Trace Bus Усовершенствованная шина трассировки (шина ATB)
BE-8	Byte-invariant big Endian mode Обратный порядок байтов с неизменным расположением байтов (формат хранения данных)
CMSIS	Cortex Microcontroller Software Interface Standard Стандарт программного интерфейса микроконтроллеров с ядром Cortex
CPI	Cycles Per Instruction Число тактов на команду
CPU	Central Processing Unit Центральный процессор, ЦПУ
CS3	CodeSourcery Common Start-up Code Sequence Общий стартовый код ИСР CodeSourcery
DAP	Debug Access Port Порт доступа к модулю отладки (порт DAP)
DSP	Digital Signal Processor/Digital Signal Processing Процессор цифровой обработки сигналов / Цифровая обработка сигналов
DWT	Data Watchpoint and Trace unit Модуль трассировки и поддержки контрольных точек данных
EABI/ABI	Embedded Application Binary Interface Двоичный интерфейс встраиваемых приложений (интерфейс EABI)
ETM	Embedded Trace Macrocell Встроенная макроячейка трассировки
FPB	Flash Patch and Breakpoint unit Модуль коррекции флэш-памяти и задания точки останова
FPGA	Field Programmable Gate Array Программируемая вентильная матрица

FSR	Fault Status Register Регистр состояния отказа
HTM	CoreSight AHB Trace Macrocell Макроячейка трассировки AHB
ICE	In-Circuit Emulator Внутрисхемный эмулятор
IDE	Integrated Development Environment Интегрированная среда разработки, ИСР
IRQ	Interrupt ReQuest Запрос прерывания (обычно применяется с внешними прерываниями)
ISA	Instruction Set Architecture Архитектура набора команд
ISR	Interrupt Service Routine Процедура обработки прерывания
ITM	Instrumentation Trace Macrocell Макроячейка инструментальной трассировки
JTAG	Joint Test Action Group Объединённая рабочая группа по автоматизации тестирования; название стандарта интерфейсов тестирования и отладки
JTAG-DP	JTAG Debug Port Порт отладки JTAG
LR	Link Register Регистр связи
LSB	Least Significant Bit Младший значащий бит
MCU	MicroController Unit Микроконтроллер (МК)
MDK-ARM	Keil Microcontroller Development Kit for ARM Пакет разработки для ARM компании Keil
MMU	Memory Management Unit Модуль управления памятью
MPU	Memory Protection Unit Модуль защиты памяти
MSB	Most Significant Bit Старший значащий бит
MSP	Main Stack Pointer Основной указатель стека
NMI	NonMaskable Interrupt Немаскируемое прерывание
NVIC	Nested Vectored Interrupt Controller Контроллер вложенных векторных прерываний
OS	Operating System Операционная система (ОС)

PC	Program Counter Счётчик команд
PMU	Power Management Unit Модуль управления питанием
PSP	Process Stack Pointer Указатель стека процесса
PPB	Private Peripheral Bus Шина собственных периферийных устройств (шина PPB)
PSR	Program Status Register Регистр состояния программы
SCB	System Control Block Блок управления системой
SCS	System Control Space Пространство управления системой
SIMD	Single Instruction, Multiple Data Один поток команд — несколько потоков данных (архитектура SIMD)
SoC	System-on-Chip Система на кристалле
SP	Stack Pointer Указатель стека
SRPG	State Retention Power Gating Технология SRPG
SW	Serial-Wire Интерфейс Serial-Wire
SW-DP	Serial-Wire Debug Port Порт отладки Serial-Wire
SWJ-DP	Serial-Wire JTAG Debug Port Порт отладки Serial-Wire/JTAG
SWV	Serial-Wire Viewer Модуль наблюдения за шиной Serial-Wire (один из режимов работы модуля TPIU)
TCM	Tightly Coupled Memory Тесно связанная память (характеристика Cortex-M1)
TPA	Trace Port Analyzer Анализатор порта трассировки
TPIU	Trace Port Interface Unit Модуль интерфейса порта трассировки
UAL	Unified Assembly Language Унифицированный язык ассемблера
UART	Universal Asynchronous Receiver Transmitter Универсальный асинхронный приёмопередатчик
WIC	Wakeup Interrupt Controller Контроллер пробуждающих прерываний

ГЛАВА 1
ВВЕДЕНИЕ

1.1. Процессор ARM Cortex-M3 — что же это такое?

Рынок микроконтроллеров поистине огромен — по прогнозам аналитиков в 2010 году будет продано более 20 миллиардов данных устройств. На этом рынке идёт непрерывная конкурентная борьба между различными производителями, моделями и архитектурами микроконтроллеров. Рост запросов со стороны промышленного сектора вызвал потребность в более производительных микроконтроллерах; в частности, возникла необходимость в микроконтроллерах, которые при той же частоте или потребляемой мощности выполняли бы большее число операций. Кроме того, микроконтроллеры становятся всё более «коммуникабельными», используя для связи с окружающим миром шину USB, Ethernet или радиоканал, и вполне естественно, что для поддержки этих каналов связи и развитых периферийных устройств требуются дополнительные вычислительные ресурсы. Одновременно растёт сложность самих приложений, что обусловлено использованием более изощрённых пользовательских интерфейсов, необходимостью поддержки мультимедиа и увеличением функциональности устройств.

Процессор ARM Cortex™-M3 — первый представитель процессоров семейства Cortex, выпущенных компанией ARM в 2006 году — изначально был нацелен на рынок 32-битных микроконтроллеров. Данный процессор, несмотря на небольшое число логических вентилей, требуемых для его реализации, обладает великолепной производительностью и предлагает много новых возможностей, которые ранее были доступны только в самых «навороченных» процессорах. Процессор Cortex-M3 удовлетворяет самым разным требованиям рынка 32-битных процессоров для встраиваемых систем, предлагая:

- *Бóльшую производительность* — позволяет выполнять больший объём вычислений без необходимости увеличения частоты или потребляемой мощности.
- *Низкое энергопотребление* — обеспечивает большее время автономной работы, что особенно критично для портативных устройств, в том числе использующихся в беспроводных сетях.
- *Улучшенный детерминизм* — гарантирует, что переход к обслуживанию критических задач и прерываний будет осуществляться за минимально возможное и, главное, точно определённое время.

- *Увеличенную плотность кода* — позволяет разместить необходимый код даже в памяти небольшого объёма.
- *Простоту использования* — обеспечивает лёгкость программирования и отладки для растущего числа пользователей, переходящих с 8- и 16-битных платформ на 32-битную.
- *Низкую стоимость* — позволяет приблизить стоимость 32-битных систем к стоимости классических 8/16-битных устройств и предлагать 32-битные микроконтроллеры начального уровня по цене, составляющей менее одного доллара США.
- *Большой выбор средств разработки* — от недорогих или вообще бесплатных компиляторов до развитых пакетов от различных производителей средств разработки.

Микроконтроллеры, созданные на базе процессора Cortex-M3, уже напрямую конкурируют с устройствами, имеющими другие архитектуры. Причём, если раньше разработчики обращали внимание на стоимость отдельных устройств, то теперь они, в первую очередь, стремятся к уменьшению стоимости системы в целом. По существу, происходит агрегирование устройств, благодаря чему появляется потенциальная возможность замены трёх или четырёх традиционных 8-битных устройств одним более мощным.

Ещё одним из направлений снижения себестоимости является увеличение объёмов кода, используемого повторно в различных изделиях. Поскольку микроконтроллеры с процессорным ядром Cortex-M3 рассчитаны на программирование с использованием языков высокого уровня, в частности языка Си, и имеют установившуюся архитектуру, это значительно упрощает перенос и повторное использование программ, уменьшая тем самым время разработки и затраты на тестирование.

Следует отметить, что Cortex-M3 — не первый процессор компании ARM, предназначенный для создания микроконтроллеров общего назначения. До сих пор на этом рынке пользуется успехом несколько устаревший процессор ARM7, получивший огромную популярность благодаря различным партнёрам компании ARM, таким как NXP (Philips), Texas Instruments, Atmel, OKI, а также многим другим производителям, предлагающим надёжные 32-битные микроконтроллеры. Вообще говоря, процессор ARM7 является самым распространённым из когда-либо выпускавшихся 32-битных встраиваемых процессоров — каждый год выпускалось и продолжает выпускаться более 1 миллиарда процессоров, находящих применение в самых различных устройствах, от мобильных телефонов до автомобилей.

Предполагается, что процессор Cortex-M3 будет иметь не меньший успех, поскольку он позволяет создавать микроконтроллеры, предлагающие более простую модель программирования и отладки и при этом имеющие бóльшие вычислительные возможности. Кроме того, процессор Cortex-M3 предоставляет ряд функциональных возможностей и технологий, востребованных разработчиками микроконтроллеров. Это и наличие немаскируемых прерываний для поддержки критических задач, и поддержка вложенных векторных прерываний с высокой степенью детерминизма, и атомарные битовые операции, и даже опциональный

модуль защиты памяти (Memory Protection Unit — MPU). Всё это делает процессор Cortex-M3 привлекательным не только для тех, кто уже использует процессоры ARM, но и для массы новых пользователей, задумывающихся о применении 32-битных микроконтроллеров в своих устройствах.

1.2. ARM — компания и архитектура

1.2.1. Историческая справка

Чтобы нам было легче разобраться в многообразии процессоров ARM и версий их архитектур, совершим краткий экскурс в историю компании ARM.

Компания ARM (Advanced RISC Machines Ltd.) была основана в 1990 году как совместное предприятие компаний Apple Computer, Arcon Computer Group и VLSI Technology. В 1991 году компания ARM представила семейство процессоров ARM6, лицензию на выпуск которых первой получила компания VLSI. После того как лицензии на использование процессора ARM приобрели и другие компании, в том числе Texas Instruments, NEC, Sharp и ST Microelectronics, эти процессоры стали широко применяться в мобильных телефонах, компьютерных жёстких дисках, КПК, бытовой аудио- и видеоаппаратуре и прочих потребительских товарах.

Процессор Cortex-M3 и МК с ядром Cortex-M3 — в чём различие?

Процессор Cortex-M3 является центральным процессором (ЦПУ) микроконтроллера, т.е. всего лишь одним из его многочисленных узлов. После того как производитель микросхем приобретает лицензию на процессор Cortex-M3, он может использовать этот процессор в своих изделиях, добавляя к нему память, периферийные устройства, устройства ввода/вывода и другие блоки. Микроконтроллеры с ядром Cortex-M3 от разных производителей будут иметь различные объёмы и типы памяти, различные наборы периферии и разные возможности. Основное внимание в книге уделяется архитектуре именно процессорного ядра. Для получения подробной информации об остальных узлах микроконтроллера читателю предлагается обратиться к фирменной документации на конкретный микроконтроллер.

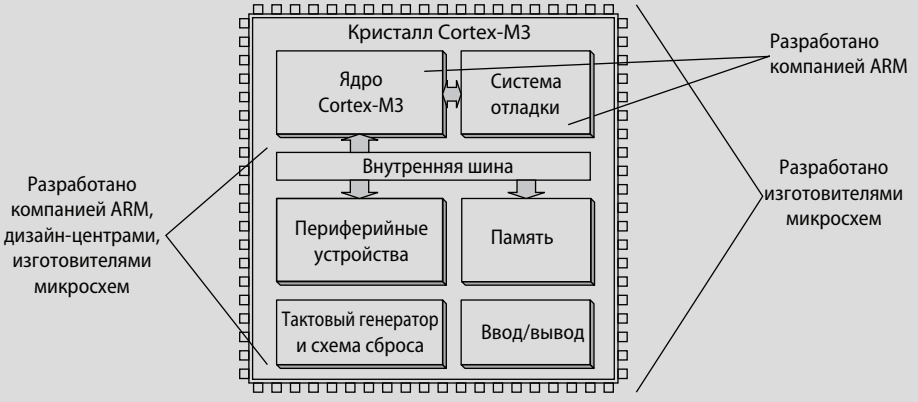


Рис. 1.1. *Процессор Cortex-M3 и МК с ядром Cortex-M3.*

В наши дни партнёры компании ARM ежегодно поставляют более двух миллиардов процессоров ARM. В отличие от многих других полупроводниковых компаний, компания ARM не занимается ни изготовлением процессоров, ни продажей микросхем. Вместо этого ARM предлагает своим бизнес-партнёрам, в числе которых большинство ведущих мировых полупроводниковых компаний, лицензии на использование разработанных ею процессорных ядер. На основе дешёвых и экономичных решений от ARM её партнёры создают собственные процессоры, микроконтроллеры и системы на кристалле. Такая бизнес-модель называется лицензированием интеллектуальной собственности (Intellectual Property — IP).

Помимо процессорных ядер, компания ARM также лицензирует IP-блоки системного уровня и различные программные решения. Для поддержки своей продукции компания ARM предлагает разнообразные аппаратные и программные средства, призванные облегчить партнёрам создание собственных продуктов.

1.2.2. Версии архитектуры

На протяжении всего времени своего существования компания ARM продолжала разработку новых процессоров и системных блоков. Эволюция функциональных возможностей и постепенное совершенствование процессоров привело к последовательному появлению нескольких версий архитектуры ARM. Причём, что интересно, номера версий архитектуры никак не привязаны к обозначениям процессоров. Скажем, в основе процессора ARM7TDMI лежит архитектура ARMv4T (буква «T» указывает на поддержку набора команд Thumb®).

Архитектура ARMv5E была реализована в процессорах семейства ARM9, в частности в процессорах ARM936E-S и ARM946E-S. В этой версии архитектуры появилась поддержка «расширенных» команд цифровой обработки сигналов, предназначенных для реализации мультимедиа-приложений.

Процессоры следующего поколения ARM11 имели уже новую архитектуру — ARMv6. В данной версии архитектуры была улучшена подсистема памяти и появилась поддержка мультимедийных SIMD-команд. Архитектуру ARMv6, в частности, имеют процессоры ARM1136J(F)-S, ARM1156T2(F)-S и ARM1176JZ(F)-S.

Буквально сразу же после выпуска на рынок семейства ARM11 руководство компании пришло к выводу, что многие новые технологии, в частности оптимизированный набор команд Thumb-2, были бы востребованы и в бюджетных сегментах рынков микроконтроллеров и компонентов для автомобильной электроники. Также было принято решение о необходимости разработки процессорных архитектур, которые бы максимально полно удовлетворяли требованиям конкретных классов приложений. Наличие таких архитектур позволило бы создавать как содержащие малое число вентилей процессоры для сегментов рынка, критичных к стоимости компонентов, так и высокопроизводительные и многофункциональные процессоры для устройств верхнего ценового диапазона.

За последние несколько лет компания ARM значительно расширила номенклатуру своей продукции путём диверсификации процесса разработки ЦПУ.

В новой версии в рамках единой архитектуры было выделено три подсемейства (профиля):

- *Профиль A* — для высокопроизводительных открытых платформ.
- *Профиль R* — для многофункциональных встраиваемых систем, работающих в режиме реального времени.
- *Профиль M* — для встраиваемых систем на базе микроконтроллеров.

Рассмотрим сферы применения различных профилей более подробно:

- *Профиль A (ARMv7-A)* — прикладные процессоры, предназначенные для поддержки сложных приложений, в частности «тяжёлых» встраиваемых операционных систем, таких как Symbian, Linux и Windows Embedded. От этих процессоров требуются максимальная вычислительная мощность, поддержка системы виртуальной памяти посредством модулей управления памятью (MMU) и, по возможности, расширенная поддержка языка Java и обеспечение безопасной среды выполнения программы. В качестве примера целевых устройств можно указать дорогие мобильные телефоны и электронные кошельки для проведения финансовых операций.
- *Профиль R (ARMv7-R)* — высокопроизводительные процессоры, предназначенные для создания устройств, работающих в условиях жёсткого реального времени[1)], таких как системы торможения современных автомобилей и контроллеры жёстких дисков. Подобные приложения требуют большой вычислительной мощности, высокой надёжности и как можно меньшей латентности.
- *Профиль M (ARMv7-M)* — процессоры для бюджетных приложений, в которых, помимо высокой производительности, критичными являются такие параметры, как стоимость, энергопотребление, время реакции на прерывания и простота использования, а также процессоры для систем управления производственными процессами, в том числе систем управления реального времени.

Семейство Cortex является первым семейством, имеющим архитектуру v7, при этом процессор Cortex-M3 основан на профиле ARMv7-M, предназначенном для микроконтроллеров.

Эта книга посвящена процессору Cortex-M3, но не стоит забывать, что данный процессор является всего лишь одним из представителей семейства, имеющего архитектуру ARMv7. Помимо Cortex-M3, существует ещё Cortex-A8 (прикладной процессор), основанный на профиле ARMv7-A, и Cortex-R4 (процессор реального времени), основанный на профиле ARMv7-R (**Рис. 1.2**).

[1)] Вообще-то это большой вопрос, можно ли получить систему «реального времени», используя процессоры общего назначения. По определению, термин «реальное время» означает, что система может получить отклик в течение гарантированного интервала времени. В любых системах, основанных на использовании процессоров, возможность получения или неполучения такого отклика будет зависеть от выбранной ОС, величины задержки обработки прерываний или от времени доступа к памяти, не говоря уже о том, что в требуемый момент времени процессор может быть занят обработкой прерывания с более высоким приоритетом.

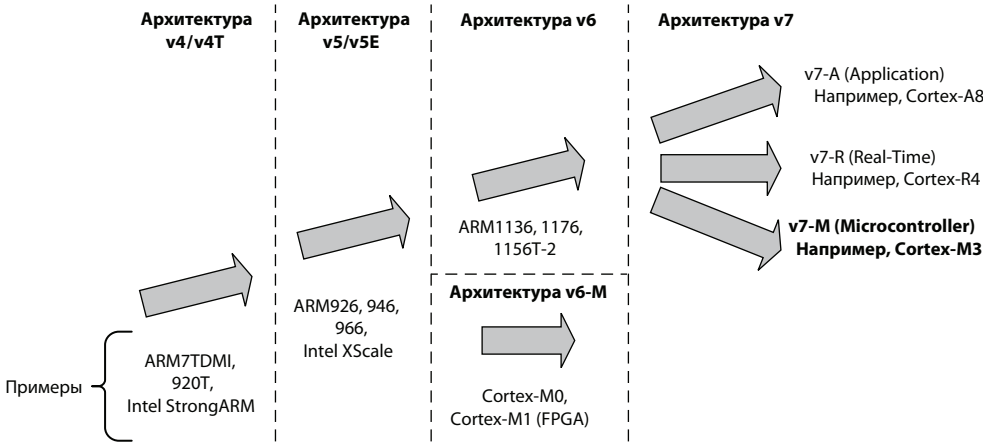


Рис. 1.2. *Эволюция архитектуры процессоров ARM.*

Архитектура ARMv7-M подробно описана в руководстве [2]. Этот документ можно свободно загрузить с веб-сайта компании ARM после несложной регистрации. В данном документе подробно рассмотрены следующие ключевые элементы архитектуры:

- модель программирования;
- набор команд;
- модель памяти;
- архитектура отладки.

Информация, относящаяся к рассматриваемому нами процессору, скажем детальное описание его интерфейса и значения временных параметров, содержится в справочном руководстве [1]. Данный документ можно свободно загрузить с веб-сайта компании. В руководстве по процессору Cortex-M3 также описываются некоторые особенности реализации ядра, не указанные в спецификации архитектуры. В частности, в этом документе приводится список поддерживаемых команд, поскольку не все команды, описанные в спецификации ARMv7-M, являются обязательными к реализации в устройствах, имеющих данную архитектуру.

1.2.3. Обозначения процессоров

С самого начала компания ARM использовала для обозначения своих процессоров порядковые номера. Для указания индивидуальных особенностей процессоров в 90-х годах использовались буквенные суффиксы. Например, в обозначении процессора ARM7TDMI буква «T» указывает на поддержку набора команд Thumb, «D» указывает на возможность отладки по интерфейсу JTAG, «M» означает наличие быстрого умножителя, а «I» — наличие встроенного модуля ICE. Позже было принято решение сделать все эти возможности стандартными для будущих процессоров, и надобность в подобных суффиксах отпала. Взамен была разработана новая схема обозначений, отражающая различия в реализации интерфейса памяти, кэш-памяти и тесно связанной памяти (TCM).

Например, процессорам ARM с кэш-памятью и MMU были присвоены коды 26 или 36, тогда как процессоры с MPU получили код 46 (например, ARM946E-S). Кроме того, были введены новые суффиксы, отражающие использование технологии синтезируемого[1)] кода (S) и технологии Jazelle (J). Различные названия процессоров сведены в **Табл. 1.1**.

***Таблица 1.1.* Обозначения процессоров ARM**

Название процессора	Версия архитектуры	Возможности по управлению памятью	Прочие возможности
ARM7TDMI	ARMv4T	—	—
ARM7TDMI-S	ARMv4T	—	—
ARM7EJ-S	ARMv5E	—	DSP, Jazelle
ARM920T	ARMv4T	MMU	—
ARM922T	ARMv4T	MMU	—
ARM926EJ-S	ARMv5E	MMU	DSP, Jazelle
ARM946E-S	ARMv5E	MPU	DSP
ARM966E-S	ARMv5E	DSP	—
ARM968E-S	ARMv5E	—	DMA, DSP
ARM966HS	ARMv5E	MPU (опция)	DSP
ARM1020E	ARMv5E	MMU	DSP
ARM1022E	ARMv5E	MMU	DSP
ARM1026EJ-S	ARMv5E	MMU или MPU	DSP, Jazelle
ARM1136J(F)-S	ARMv6	MMU	DSP, Jazelle
ARM1176JZ(F)-S	ARMv6	MMU + TrustZone	DSP, Jazelle
ARM11 MPCore	ARMv6	MMU + поддержка многопроцессорного кэша	DSP, Jazelle
ARM1156T2(F)-S	ARMv6	MPU	DSP
Cortex-M0	ARMv6-M	—	NVIC
Cortex-M1	ARMv6-M	FPGA TCM-интерфейс	NVIC
Cortex-M3	ARMv7-M	MPU (опция)	NVIC
Cortex-R4	ARMv7-R	MPU	DSP
Cortex-R4F	ARMv7-R	MPU	DSP + поддержка операций с плавающей точкой
Cortex-A8	ARMv7-A	MMU + TrustZone	DSP, Jazelle, NEON + поддержка операций с плавающей точкой
Cortex-A9	ARMv7-A	MMU + TrustZone + поддержка многопроцессорных конфигураций	DSP, Jazelle, NEON + поддержка операций с плавающей точкой

[1)] Синтезируемое процессорное ядро поставляется в виде поведенческого описания, выполненного на языке описания аппаратуры, таком как Verilog или VHDL. Посредством программы-синтезатора это описание может быть преобразовано в список соединений (электрическую схему) процессора.

Начиная с 7-й версии архитектуры, компания решила полностью отказаться от этой сложной числовой схемы, требующей расшифровки, и перешла к использованию названий семейств процессоров, первым из которых стало семейство Cortex. Помимо указания на совместимость между отдельными процессорами, эта система исключает путаницу между номером версии архитектуры и числовым кодом, обозначающим семейство. К примеру, популярный процессор ARM7TDMI имеет архитектуру v4T, а вовсе не v7, как можно ошибочно предположить.

1.3. Развитие набора команд

Совершенствование и расширение наборов команд, используемых в процессорах ARM, было одной из основных причин появления новых версий архитектуры (**Рис. 1.3**).

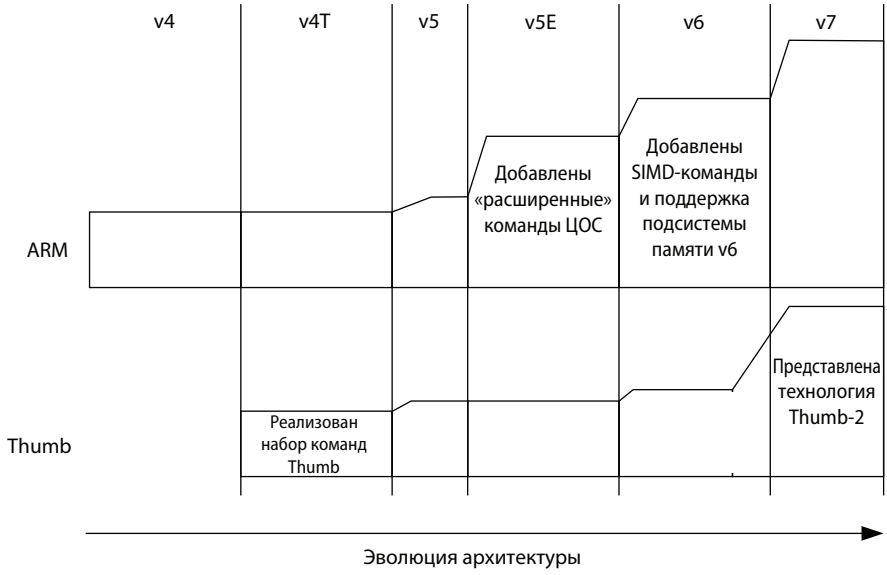


Рис. 1.3. *Развитие набора команд.*

Исторически сложилось (начиная с ARM7TDMI), что процессоры ARM поддерживают два различных набора команд: 32-битный набор команд ARM и 16-битный набор команд Thumb. При выполнении программы процессор может «на лету» переключаться между состояниями ARM и Thumb для использования того или иного набора команд. Набор команд Thumb является всего лишь подмножеством команд ARM, однако он обеспечивает бóльшую плотность кода, что немаловажно для устройств, имеющих память небольшого объёма.

По мере появления новых версий архитектуры в обоих наборах команд появлялись новые инструкции. Некоторые сведения об изменении набора команд Thumb в процессе эволюции архитектуры приведены в Приложении Б. В 2003 году компания ARM анонсировала набор команд Thumb-2, являющийся расширением набора команд Thumb и содержащий как 16-, так и 32-битные команды.

Подробная информация о системе команд содержится в справочном руководстве по архитектуре ARM «*The ARM Architecture Reference Manual*», также называемом «*ARM ARM*». Этот документ обновлялся одновременно с выпуском на рынок очередной архитектуры ARMv5, ARMv6 и ARMv7. С появлением архитектуры ARMv7 это руководство было разбито на отдельные документы из-за появления различных профилей. Собственно набор команд процессора Cortex-M3 подробно описан в руководстве [2]. Вся информация, касающаяся набора команд, которая необходима для разработки программного обеспечения, также приведена в Приложении А данной книги.

1.4. Технология Thumb-2 и архитектура набора команд

Внедрение технологии Thumb-2[1)] значительно расширило архитектуру системы команд (Instruction Set Architecture — ISA) Thumb и позволило получить высокоэффективный и мощный набор команд, обладающий серьёзными преимуществами перед своим предшественником в части простоты использования, плотности кода и производительности (**Рис. 1.4**). Расширенный набор команд Thumb-2 является надмножеством 16-битного набора команд Thumb, в который были добавлены как дополнительные 16-битные, так и новые 32-битные команды. Новый набор команд позволяет выполнять в состоянии Thumb более сложные операции, обеспечивая, таким образом, бóльшую эффективность за счёт уменьшения числа переключений между состояниями ARM и Thumb.

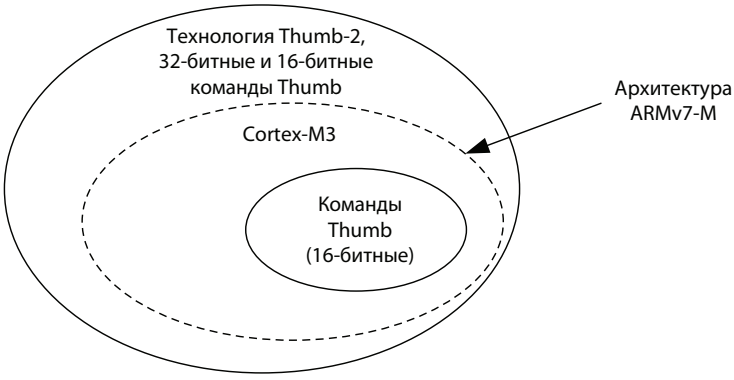


Рис. 1.4. *Взаимосвязь между наборами команд Thumb и Thumb-2.*

В связи с тем, что процессор Cortex-M3 рассчитан на применение в устройствах с малыми объёмами памяти, таких как микроконтроллеры, а также с целью уменьшения занимаемой им площади кристалла данный процессор поддерживает только набор команд Thumb-2 (наряду с традиционным Thumb). Поэтому Cortex-M3 использует команды набора Thumb-2 для выполнения любых операций, тогда как прежним процессорам для выполнения некоторых операций приходилось задействовать команды ARM. Как следствие, между процессором

[1)] Thumb и Thumb-2 являются зарегистрированными товарными знаками компании ARM.

Cortex-M3 и традиционными процессорами ARM отсутствует обратная совместимость. То есть процессор Cortex-M3 не сможет выполнить код, предназначенный для процессора ARM7. В то же время Cortex-M3 может выполнять практически все 16-битные команды набора Thumb, включая все команды, поддерживаемые семейством ARM7, что облегчает перенос приложений.

С появлением набора Thumb-2, содержащего как 16-, так и 32-битные команды, отпала необходимость в переключении процессора между состояниями Thumb (16-битные команды) и ARM (32-битные команды). Так, в случае использования процессора семейства ARM7 или ARM9, для выполнения сложных вычислений или большого числа условных операций без потери производительности могло потребоваться переключение в состояние ARM. При использовании же процессора Cortex-M3 вы можете свободно смешивать 32-битные и 16-битные команды без необходимости смены состояния, получая без всяких проблем высокую плотность кода и высокую производительность.

Набор команд Thumb-2 — очень важный элемент архитектуры ARMv7. По сравнению с командами, поддерживаемыми процессорами семейства ARM7 (архитектура ARMv4T), набор команд процессора Cortex-M3 имеет много новых возможностей. Прежде всего, это команда аппаратного деления и несколько команд аппаратного умножения, позволяющие ускорить сложные вычисления. Кроме того, процессор Cortex-M3 поддерживает обращение к невыровненным данным (ранее эта возможность имелась только в самых старших процессорах).

1.5. Области применения процессора Cortex-M3

Имея высокую производительность, обеспечивая высокую плотность кода и занимая небольшую площадь на кристалле, процессор Cortex-M3 является идеальным выбором для самых различных приложений:

- *Недорогие микроконтроллеры.* Процессор Cortex-M3 замечательно подходит для создания недорогих микроконтроллеров, повсеместно используемых в самых разных потребительских товарах — от игрушек до электроприборов. На этом рынке существует очень сильная конкуренция со стороны широко распространенных 8- и 16-битных микроконтроллеров других производителей. Малая мощность, потребляемая процессором Cortex-M3, его высокая производительность и простота применения стимулирует переход разработчиков встраиваемых устройств на 32-битную платформу и разработку изделий уже на базе архитектуры ARM.
- *Автомобильная электроника.* Ещё одной областью применения, как будто специально созданной для процессора Cortex-M3, является автомобильная промышленность. Процессор Cortex-M3 имеет очень высокую производительность и малое время реакции на прерывания, что позволяет использовать его в системах реального времени. Поддержка процессором до 240 внешних прерываний, наличие в нём встроенного контроллера с поддержкой вложенных прерываний, а также наличие опционального модуля MPU делает Cortex-M3 идеальным кандидатом на использование в чувствительных к сто-

имости устройствах автомобильной электроники с высокой степенью интеграции.

- *Передача данных.* Малое энергопотребление процессора и его высокая эффективность, а также наличие команд работы с битовыми полями, появившимися в наборе Thumb-2, делает Cortex-M3 идеальным выбором для большинства коммуникационных приложений, таких как Bluetooth и ZigBee.
- *Автоматизация производства.* Ключевыми факторами для устройств управления промышленным оборудованием являются простота, время реакции и надёжность. И опять же, наличие в процессоре Cortex-M3 продвинутого контроллера прерываний, малое время реакции на прерывание и расширенные средства обеспечения отказоустойчивости делают этот процессор основным кандидатом на использование в данной области.
- *Потребительские товары.* Во многих потребительских товарах используется один или несколько высокопроизводительных микропроцессоров. Процессор Cortex-M3, будучи небольшим по размеру, имеет высокую эффективность и малое энергопотребление. Кроме того, этот процессор поддерживает выполнение сложного программного обеспечения, рассчитанного на использование модуля MPU, обеспечивая надёжную защиту памяти.

На рынке уже предлагается много изделий, содержащих процессор Cortex-M3, включая устройства начального уровня, стоимость которых не превышает одного доллара США. То есть стоимость микроконтроллеров ARM сравнялась или даже опустилась ниже стоимости большинства 8-битных микроконтроллеров.

1.6. Структура книги

Данная книга содержит общую информацию о процессоре Cortex-M3, которая структурирована следующим образом:

- Главы 1 и 2 — Введение и обзор Cortex-M3.
- Главы 3…6 — Основы Cortex-M3.
- Главы 7…9 — Исключительные ситуации и прерывания.
- Главы 10 и 11 — Программирование Cortex-M3.
- Главы 11…14 — Аппаратные особенности Cortex-M3.
- Главы 15 и 16 — Поддержка отладки в Cortex-M3.
- Главы 17…21 — Разработка приложений с использованием Cortex-M3.
- Приложения.

1.7. Дополнительная литература

Эта книга никоим образом не является исчерпывающим руководством по процессору Cortex-M3. Цель данной книги — предоставить начальные сведения тем, кто впервые сталкивается с Cortex-M3, а также служить дополнительным справочником для разработчиков, использующих микроконтроллеры с ядром Cortex-M3. Для получения более подробных сведений о процессоре Cortex-M3 следует обращаться к фирменной документации, которую можно загрузить с веб-сайта компании ARM (*www.arm.com*) и веб-сайтов её партнёров:

- Справочное руководство «*The Cortex-M3 Technical Reference Manual*» [1] содержит полную информацию о процессоре, в том числе о модели программирования, карте памяти, а также о времени выполнения команд.
- Справочное руководство «*The ARMv7-M Architecture Application Level Reference Manual*» [2] содержит детальную информацию о наборе команд и модели памяти.
- Справочная документация (datasheet) на конкретные микроконтроллеры с ядром Cortex-M3. Эту документацию можно найти на веб-сайте изготовителя того микроконтроллера, который вы намереваетесь использовать.
- Руководства пользователя «*Cortex-M3 User Guide*» предоставляются производителями микроконтроллеров. В ряде случаев этот документ является частью общего руководства на микроконтроллер. В руководстве пользователя, которое адаптируется каждым производителем в соответствии с конкретными реализациями их микроконтроллеров, описывается модель программирования процессора Cortex-M3 и приводится подробное описание набора команд.
- Спецификация «*AMBA Specification 2.0*» [4] содержит полную информацию о реализации протокола внутренней системной шины AMBA.
- Руководство по применению «*AN179. Cortex-M3 Embedded Software Development*» [7], предлагаемое компанией ARM, содержит полезные советы, касающиеся программирования процессора Cortex-M3 на языке Си.

Данная книга рассчитана на тех читателей, которые уже имеют некоторый опыт в программировании встраиваемых систем, предпочтительно для процессоров ARM. Если же вы, будучи менеджером или студентом, хотите всего лишь ознакомиться с процессором Cortex-M3 и не желаете тратить время на чтение всей книги или изучение справочного руководства по процессору, то рекомендую прочитать вторую главу, в которой приводятся основные сведения о процессоре Cortex-M3.

ГЛАВА 2
ОБЗОР CORTEX-M3

2.1. Основные сведения

Процессор Cortex™-M3 представляет собой 32-битный микропроцессор. Он имеет 32-битную шину данных, 32-битный банк регистров и 32-битные интерфейсы памяти (**Рис. 2.1**).

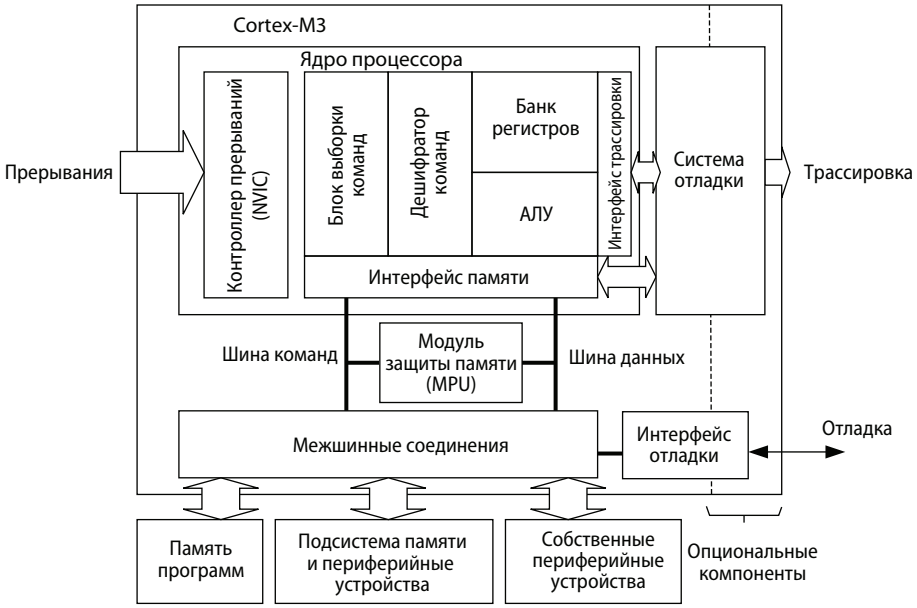


Рис. 2.1. *Упрощённая блок-схема Cortex-M3.*

Процессор выполнен по Гарвардской архитектуре, т.е. имеет раздельные шины команд и данных. Это позволяет осуществлять выборку команд одновременно с обращением к данным. В результате увеличивается производительность процессора, поскольку операции доступа к данным никак не влияют на конвейер команд. Это позволило реализовать в процессоре Cortex-M3 несколько шинных интерфейсов, каждый из которых оптимизирован для выполнения определённых функций и, в то же время, может использоваться одновременно с другими

интерфейсами. При этом шины команд и данных разделяют одно и то же адресное пространство (единая система памяти). Другими словами, наличие раздельных шинных интерфейсов вовсе не означает, что вы сможете использовать память размером 8 Гбайт.

Для поддержки сложных приложений, требующих более развитой системы памяти, в процессоре Cortex-M3 предусмотрены опциональный модуль защиты памяти (Memory Protection Unit — MPU) и возможность использования внешней кэш-памяти. Поддерживаются системы памяти, использующие как прямой (little endian), так и обратный (big endian) порядок байтов.

В составе процессора Cortex-M3 имеется ряд встроенных компонентов отладки. Эти компоненты осуществляют поддержку основных отладочных операций и возможностей, таких как точки останова (breakpoints) и точки наблюдений (watchpoints).

Предусмотрены и опциональные компоненты, поддерживающие расширенные возможности отладки, в частности трассировку команд, а также различные типы отладочных интерфейсов.

2.2. Регистры

Процессор Cortex-M3 имеет 16 регистров с R0 по R15 (**Рис. 2.2**). При этом регистр R13 (указатель стека) реализован в виде банка из двух регистров (в каждый момент времени доступен только один из регистров).

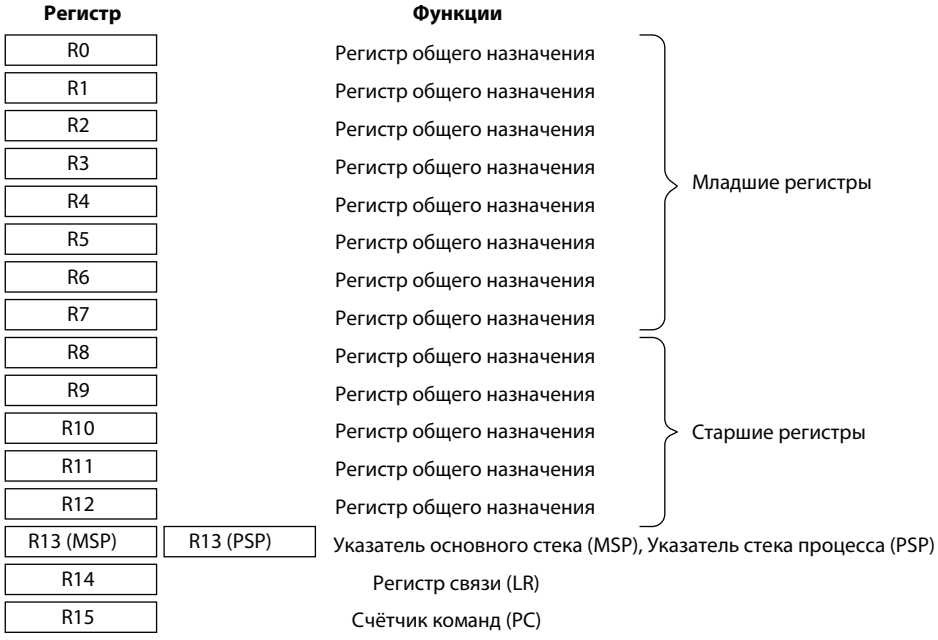


Рис. 2.2. *Регистры процессора Cortex-M3.*

2.2.1. R0…R12 — регистры общего назначения

Регистры с R0 по R12 являются 32-битными регистрами общего назначения, предназначенными для хранения обрабатываемых данных. Некоторые 16-битные команды Thumb® могут обращаться только к младшим регистрам (R0…R7).

2.2.2. R13 — указатели стека

Процессор Cortex-M3 содержит два указателя стека (R13). Они объединены в банк, поэтому в каждый момент времени виден только один из них:

- *Основной указатель стека* (Main Stack Pointer — MSP) — указатель стека, используемый ядром операционной системы и обработчиками исключительных ситуаций.
- *Указатель стека процесса* (Process Stack Pointer — PSP) — указатель стека, используемый прикладной программой.

Два младших бита указателей стека всегда сброшены в 0, т.е. эти указатели всегда выровнены на границу 32-битного слова.

2.2.3. R14 — регистр связи

В этом регистре при вызове подпрограммы запоминается адрес возврата.

2.2.4. R15 — счётчик команд

Счётчик команд (Program Counter — PC) содержит адрес выполняемой в данный момент команды. Этот регистр может быть изменён для управления ходом выполнения программы.

2.2.5. Регистры специального назначения

В процессоре Cortex-M3 также имеется несколько регистров специального назначения (**Рис. 2.3**):

- регистры состояния программы (xPSR);
- регистры маскирования прерываний (PRIMASK, FAULTMASK и BASEPRI);
- регистр управления (CONTROL).

Эти регистры выполняют специальные функции и для обращения к ним необходимо использовать особые команды. Указанные регистры не могут задействоваться для обработки и хранения обычных данных (**Табл. 2.1**).

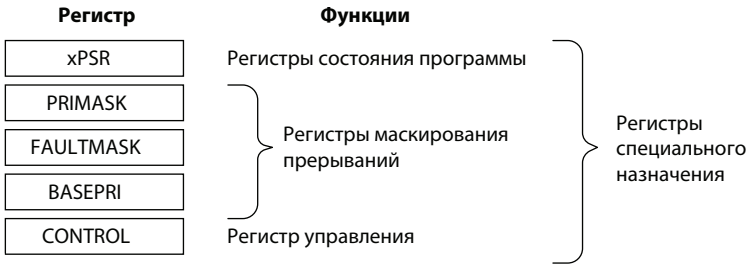


Рис. 2.3. *Регистры специального назначения процессора Cortex-M3.*

Таблица 2.1. **Регистры специального назначения и их функции**

Регистр	Назначение
xPSR	Содержат флаги результатов выполнения арифметических и логических операций (флаг нуля и флаг переноса), состояние выполнения программы и номер обрабатываемого в данный момент прерывания
PRIMASK	Запрещает все прерывания, за исключением немаскируемого прерывания (NMI) и исключения Hard Fault
FAULTMASK	Запрещает все прерывания, за исключением NMI
BASEPRI	Запрещает все прерывания, имеющие уровень приоритета, равный или меньший заданного
CONTROL	Определяет уровень доступа и используемый указатель стека

Примечание. Более подробно эти регистры рассмотрены в Главе 3.

2.3. Режимы работы

Процессор Cortex-M3 имеет два режима работы и поддерживает два уровня доступа к коду программы. Режимы работы Thread (режим потока) и Handler (режим обработчика) определяют, какой код выполняет процессор в данный момент времени — код обычной программы или же код обработчика исключительной ситуации, такой как прерывание или системное исключение (**Рис. 2.4**). Два уровня доступа к коду (привилегированный и пользовательский) обеспечивают безопасное обращение к критическим областям памяти, а также реализуют базовую модель механизма защиты.

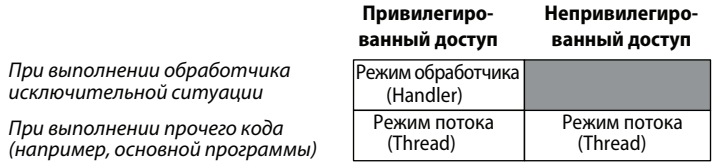


Рис. 2.4. *Режимы работы и уровни доступа к коду процессора Cortex-M3.*

При работе процессора в режиме потока допускается как привилегированное, так и непривилегированное выполнение программы, тогда как обработка исключительных ситуаций всегда осуществляется на привилегированном уровне. После сброса процессор находится в режиме потока с правами привилегированного доступа к коду. В состоянии привилегированного доступа программа может обращаться к любым областям памяти (за исключением тех, доступ к которым запрещён настройками модуля MPU) и использовать все поддерживаемые команды.

Программное обеспечение, выполняющееся на привилегированном уровне, может переключиться на пользовательский уровень, используя регистр управления. В случае возникновения исключительной ситуации процессор автоматически переключится на привилегированный уровень, а при выходе из обработчика — вернётся на исходный. Пользовательская программа не может самостоятельно переключиться на привилегированный уровень посредством записи в регистр управления (**Рис. 2.5**). Для этого необходим обработчик исключительной ситуации, который загрузит в регистр управления такое значение, чтобы при

возврате в режим потока процессор переключился на привилегированный уровень.

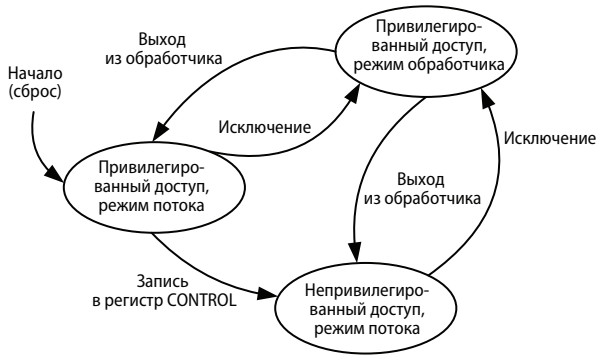


Рис. 2.5. *Диаграмма допустимых переходов между режимами работы процессора Cortex-M3.*

Наличие двух уровней доступа увеличивает надёжность системы, запрещая непроверенному коду доступ к регистрам, определяющим конфигурацию системы. При наличии модуля MPU он может использоваться совместно с привилегированным уровнем доступа для защиты критических секций памяти, например содержащих исполняемый код и данные ядра ОС.

Так, на привилегированном уровне доступа, обычно используемом ядром операционной системы, допускается обращение к любым областям памяти, не заблокированным настройками модуля MPU. При запуске системой пользовательского приложения, оно, в большинстве случаев, выполняется на непривилегированном (пользовательском) уровне, что позволяет защитить систему от возникновения сбоев из-за некорректного функционирования пользовательских программ.

2.4. Встроенный контроллер вложенных векторных прерываний

В составе процессора Cortex-M3 имеется контроллер вложенных векторных прерываний (Nested Vectored Interrupt Controller — NVIC). Он тесно связан с ядром процессора и выполняет следующие функции:

- поддержка вложенных прерываний;
- поддержка векторных прерываний;
- поддержка динамического изменения приоритетов;
- уменьшение задержки обработки прерывания;
- маскирование прерываний.

2.4.1. Поддержка вложенных прерываний

Контроллер NVIC обеспечивает поддержку вложенных прерываний. Всем внешним прерываниям и большинству системных исключений могут быть назначены различные уровни приоритета. При возникновении прерывания контроллер сравнивает его приоритет с приоритетом прерывания, обрабатываемого

в данный момент. Если новое прерывание имеет более высокий приоритет, то обработка текущего прерывания приостанавливается и запускается обработчик нового прерывания.

2.4.2. Поддержка векторных прерываний

В процессоре Cortex-M3 реализована поддержка векторных прерываний. В случае возникновения разрешённого прерывания стартовый адрес соответствующей процедуры обработки прерывания (Interrupt Service Routine — ISR) берётся из таблицы векторов, расположенной в памяти. Причём, определение стартового адреса ISR и переход на него осуществляется полностью аппаратно, что ускоряет обслуживание запроса прерывания.

2.4.3. Поддержка динамического изменения приоритетов

Уровни приоритета прерываний могут изменяться программно. При этом активация обслуживаемых в данный момент прерываний блокируется до выхода из процедуры обработки прерывания, что исключает нежелательный повторный вызов обработчиков прерываний при изменении приоритетов.

2.4.4. Уменьшение времени реакции на прерывание

В процессоре Cortex-M3 также реализованы определённые решения, позволяющие уменьшить задержку обработки прерываний. Это автоматическое сохранение и восстановление содержимого некоторых регистров, уменьшение задержки при переходе от одной процедуры обработки прерывания к другой, а также обработка «опоздавших» прерываний. Более подробно эти возможности процессора рассматриваются в Главе 9.

2.4.5. Маскирование прерываний

Прерывания и системные исключения могут быть маскированы в соответствии с их уровнями приоритета или же полностью при помощи регистров маскирования BASEPRI, PRIMASK и FAULTMASK. Эти регистры могут использоваться для того, чтобы исключить прерывание строго ограниченных во времени задач, тем самым гарантируя их своевременное завершение.

2.5. Карта памяти

В процессоре Cortex-M3 используется фиксированное распределение адресного пространства. Это позволяет обращаться к встроенным периферийным устройствам, таким как контроллер прерываний и компоненты системы отладки, посредством обычных команд доступа к памяти. То есть большинство системных функций можно использовать напрямую из программ, написанных на языке Си. Предопределённая карта памяти также обеспечивает чрезвычайно высокую скорость работы процессора и облегчает его интеграцию в системы на кристалле (System on a Chip — SoC).

Распределение общего адресного пространства размером 4 Гбайт показано на **Рис. 2.6**.

Рис. 2.6. *Карта памяти процессора Cortex-M3.*

Процессор Cortex-M3 имеет внутреннюю шинную инфраструктуру, которая оптимизирована для использования памяти, распределённой в соответствии с **Рис. 2.6**. Кроме того, конструкция процессора позволяет задействовать указанные области различным образом. Так, память данных может быть размещена в секции кода, а код программы может запускаться из секции внешнего ОЗУ.

В системных областях памяти располагаются контроллер прерываний и компоненты отладки. Эти устройства имеют фиксированные адреса, полная информация о которых приведена в Главе 5. Размещение этих периферийных устройств по фиксированным адресам значительно облегчает перенос приложений между микроконтроллерами различных производителей.

2.6. Интерфейсы шин

В процессоре Cortex-M3 реализовано несколько шин, что позволяет ему осуществлять выборку команд одновременно с обращением к данным. Можно выделить следующие основные интерфейсы шин:

- шины памяти кода;
- системная шина;
- шина собственных периферийных устройств.

Шины памяти кода предназначены для обеспечения доступа к одноимённой области памяти и физически реализованы в виде двух шин, называемых I-Code и D-Code. Такое решение позволило уменьшить время выборки команд, увеличив тем самым быстродействие процессора.

Системная шина используется для доступа к памяти и периферийным устройствам. С её помощью производятся обращения к статическому ОЗУ (СОЗУ), периферии, внешнему ОЗУ, внешним устройствам, а также к некоторым системным областям памяти.

Шина собственных периферийных устройств обеспечивает доступ к определённой части памяти, зарезервированной для использования встроенными периферийными устройствами процессора, такими как компоненты отладки.

2.7. Модуль защиты памяти MPU

В процессоре Cortex-M3 предусмотрен опциональный модуль защиты памяти MPU. Этот модуль позволяет задавать правила доступа к памяти на привилегированном и пользовательском уровнях. При нарушении правил доступа генерируется исключение отказа, в обработчике которого можно проанализировать проблему и, по возможности, скорректировать её.

Модуль MPU можно задействовать различным образом. В общем случае ОС может настроить MPU для защиты данных, используемых ядром ОС и другими привилегированными процессами, от недостаточно надёжных пользовательских программ. Помимо этого, модуль MPU может применяться для перевода определённых областей памяти в режим «только для чтения», позволяя тем самым предотвратить случайное повреждение данных или же изолировать области памяти различных задач в многозадачной системе. В общем и целом, модуль MPU помогает повысить надёжность и отказоустойчивость встроенных систем.

Модуль MPU является необязательным компонентом процессора, и решение о его использовании принимается на этапе реализации микроконтроллера или системы на кристалле. Более подробно о модуле MPU рассказывается в Главе 13.

2.8. Набор команд

Процессор Cortex-M3 поддерживает набор команд Thumb-2. Это одна из наиболее важных особенностей процессора, поскольку позволяет совместно использовать 16- и 32-битные команды, обеспечивая одновременно как высокую эффективность, так и высокую плотность кода. Набор Thumb-2 — гибкий, мощный и при этом простой в использовании набор команд.

В предыдущих процессорах компании ARM центральный процессор (ЦПУ) мог находиться в одном из двух состояний: 32-битном состоянии ARM и 16-битном состоянии Thumb. В состоянии ARM все команды являются 32-битными, что обеспечивает очень высокую производительность. В состоянии Thumb команды являются 16-битными, позволяя получить более высокую плотность кода. Однако эти команды имеют ограниченную функциональность по сравнению с командами ARM, поэтому для выполнения определённых операций может потребоваться большее число команд.

Для использования преимуществ обоих состояний код большинства приложений состоит из смеси команд ARM и Thumb. Однако такой подход не всегда себя оправдывает. На переключение процессора между состояниями расходуется как время, так и место в памяти (**Рис. 2.7**). К тому же наличие двух наборов команд может потребовать разбиения исходного кода на отдельные файлы, каждый из которых будет скомпилирован с использованием соответствующего набора. Это усложняет разработку программного обеспечения и уменьшает максимальную производительность ядра ЦПУ.

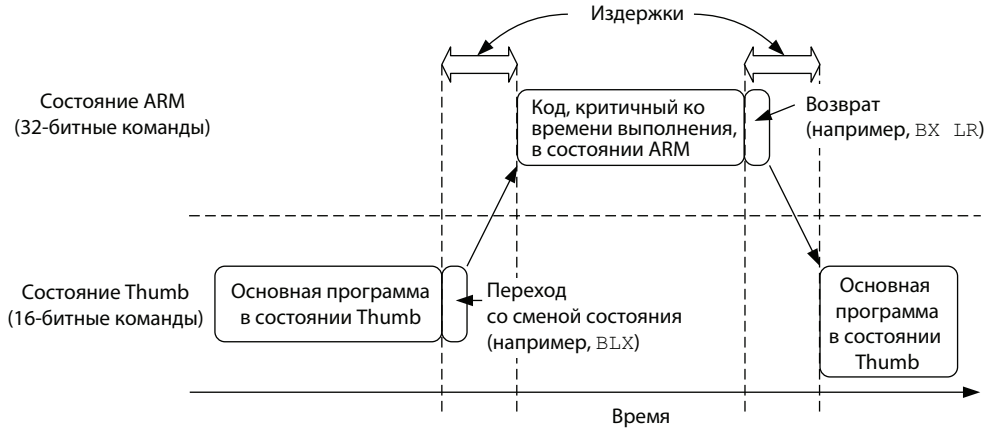


Рис. 2.7. *Переключение между состояниями ARM и Thumb в классическом процессоре ARM, таком как ARM7.*

С появлением набора команд Thumb-2 стало возможным выполнять все необходимые операции, находясь в одном состоянии, — необходимость в переключении между двумя рабочими состояниями полностью исчезла. Следует отметить, что процессор Cortex-M3 вообще не поддерживает набор команд ARM. Даже прерывания теперь обрабатываются в состоянии Thumb (ранее при входе в процедуры обработки прерываний процессор переключался в состояние ARM). Поскольку процессору Cortex-M3 не требуется переключаться между рабочими состояниями, он имеет ряд преимуществ по сравнению с традиционными процессорами ARM:

- отсутствуют накладные расходы на переключения между состояниями, в результате чего уменьшается время выполнения и размер кода программы;
- отсутствует необходимость разделения исходных файлов на файлы с кодом ARM и файлы с кодом Thumb, что облегчает разработку программ и их дальнейшее сопровождение;
- упрощается достижение максимальной эффективности и производительности, что, в свою очередь, облегчает разработку программного обеспечения (не требуется перепрыгивать с одного набора команд на другой, пытаясь достичь лучшего соотношения между размером кода и производительностью).

Процессор Cortex-M3 поддерживает ряд интересных и мощных команд. Вот только некоторые из них:

- `UBFX`, `BFI` и `BFC` — извлечение, вставка и очистка битового поля;

- `UDIV` и `SDIV` — беззнаковое и знаковое деление;
- `WFE`, `WFI` и `SEV` — ожидание события, ожидание прерывания и генерация события; эти команды используются для перевода процессора в режим пониженного энергопотребления и для поддержки синхронизации между задачами в многопроцессорных системах;
- `MSR` и `MRS` — пересылка данных между регистрами общего назначения и регистрами специального назначения.

Поскольку процессор Cortex-M3 поддерживает только набор команд Thumb-2, то для использования существующего кода, предназначенного для других процессоров ARM, требуется его перенос на новую архитектуру. В большинстве случаев процедура переноса заключается в перекомпиляции исходных текстов на языке Си с применением нового компилятора, поддерживающего процессор Cortex-M3. Некоторые фрагменты программы, написанные на ассемблере, придётся переписать с учётом новой архитектуры и нового унифицированного языка ассемблера.

Обратите внимание, что в процессоре Cortex-M3 реализованы не все команды из набора Thumb-2. В соответствии с [2] обязательным к реализации является только определённое подмножество команд Thumb-2. В частности, в процессоре Cortex-M3 отсутствует поддержка команд сопроцессора (могут быть подключены внешние устройства обработки данных), а также не реализованы SIMD-команды (один поток команд — несколько потоков данных). Кроме того, не поддерживаются некоторые команды Thumb, такие как команда перехода `BLX` (использовалась для переключения процессора из состояния Thumb в состояние ARM), ряд команд изменения состояния процесса (`CPS`), а также команды изменения представления многобайтных чисел (`SETEND`), появившиеся в архитектуре v6. Полный список поддерживаемых команд приведён в Приложении А.

2.9. Прерывания и исключения

В процессоре Cortex-M3 реализована новая модель исключений, разработанная для архитектуры ARMv7-M. Данная модель отличается от классической модели исключений ARM и обеспечивает очень эффективную поддержку исключительных ситуаций. В этой модели предусмотрено несколько системных исключений плюс некоторое количество внешних запросов прерываний (входы внешних прерываний). В процессоре Cortex-M3 отсутствует быстрое прерывание FIQ, имеющееся в процессорах ARM7/ARM9/ARM10/ARM11. С другой стороны, он поддерживает приоритеты прерываний, а также вложенные прерывания. Поэтому не составляет никакого труда реализовать систему с поддержкой вложенных прерываний (прерывание с более высоким приоритетом может приостановить выполнение обработчика прерывания с более низким приоритетом), которые будут вести себя аналогично прерыванию FIQ в предыдущих процессорах ARM.

За поддержку прерываний в процессоре Cortex-M3 отвечает контроллер прерываний NVIC. Помимо внешних прерываний, процессор также поддерживает несколько внутренних источников исключений, предназначенных, в частности, для обработки системных отказов. Соответственно, в процессоре имеется несколько предопределённых типов исключений (**Табл. 2.2**).

Таблица 2.2. **Типы исключений Cortex-M3**

Номер исключения	Тип исключения	Приоритет*	Описание
0	—	—	Исключение отсутствует
1	Reset	–3 (Наивысший)	Сброс
2	NMI	–2	Немаскируемое прерывание (вход внешнего немаскируемого прерывания)
3	Hard Fault	–1	Любой отказ, если соответствующий обработчик не разрешён
4	MemManage Fault	Программируемый	Отказ системы управления памятью; нарушение правил доступа, заданных модулем MPU, или обращение по некорректному адресу
5	Bus Fault	Программируемый	Отказ шины (отказ предвыборки или отказ данных)
6	Usage Fault	Программируемый	Отказ программы
7…10	Зарезервировано	—	Зарезервировано
11	SVCall	Программируемый	Вызов супервизора
12	Debug monitor	Программируемый	Исключение монитора отладки (точки останова, точки наблюдения или внешняя команда отладки)
13	Зарезервировано	—	Зарезервировано
14	PendSV	Программируемый	Запрос системной службы
15	SYSTICK	Программируемый	Системный таймер
16	IRQ #0	Программируемый	Внешнее прерывание №0
17	IRQ #1	Программируемый	Внешнее прерывание №1
…	…	…	…
255	IRQ #239	Программируемый	Внешнее прерывание №239
* Если допускает программирование, то по умолчанию равен 0.			

Примечание. Количество входов внешних прерываний определяется изготовителями микросхем (поддерживается до 240 входов). Кроме того, Cortex-M3 имеет вход немаскируемого прерывания NMI, при активации которого в обязательном порядке запускается обработчик немаскируемого прерывания.

2.9.1. Низкое энергопотребление и высокая энергоэффективность

В процессоре Cortex-M3 применён ряд решений, позволяющих разработчикам создавать экономичные и энергоэффективные изделия. Прежде всего, это наличие двух режимов пониженного энергопотребления, позволяющих использовать различные стратегии для уменьшения потребления во время простоя.

Во-вторых, снижению потребляемой мощности способствует относительно небольшое число логических вентилей, образующих процессор, а также определённые схемотехнические решения, позволяющие уменьшить активность отдельных его узлов. Кроме того, процессор Cortex-M3 обеспечивает высокую плотность кода, что снижает требования к объёму памяти программ. В то же время все эти особенности позволяют ускорить выполнение различных задач обработ-

ки данных и, соответственно, возврат процессора в спящий режим для сокращения энергопотребления. В результате энергоэффективность процессора Cortex-M3 оказывается лучшей, нежели у большинства 8- и 16-битных микроконтроллеров.

Во второй ревизии процессора Cortex-M3 появился новый модуль — контроллер «пробуждающих» прерываний (Wakeup Interrupt Controller — WIC). Этот модуль позволяет отключать питание процессорного ядра с сохранением состояния процессора, а также обеспечивает практически мгновенный возврат процессора в активное состояние при возникновении прерывания. Подобная возможность позволяет использовать процессор Cortex-M3 во многих приложениях со сверхнизким потреблением, которые прежде могли быть выполнены только на 8- или 16-битных микроконтроллерах.

2.10. Возможности отладки

Процессор Cortex-M3 поддерживает различные функции отладки, такие как управление процессом выполнения программы, включая останов и пошаговое исполнение, точки останова и точки наблюдения данных, обращение к регистрам процессора и к памяти «на лету», профилирование и трассировку.

Аппаратные средства отладки процессора Cortex-M3 базируются на архитектуре CoreSight™. В отличие от традиционных процессоров ARM, в самом ядре процессора интерфейс JTAG отсутствует. Вместо этого интерфейс отладки реализован в виде отдельного модуля, для связи с которым в процессоре предусмотрен специальный интерфейс, называемый *портом доступа к средствам отладки* (Debug Access Port — DAP). С помощью указанного интерфейса внешние отладчики могут обращаться как к регистрам управления аппаратных средств отладки, так и к системе памяти даже во время выполнения процессором программы. Управление данным интерфейсом осуществляется через внешний *порт отладки* (Debug Port — DP). В настоящее время реализованы следующие порты: Serial-Wire JTAG Debug Port (SWJ-DP), поддерживающий как традиционный протокол JTAG, так и протокол Serial-Wire, и SW-DP (поддерживает только протокол Serial-Wire). Также можно использовать модуль JTAG-DP из семейства продукции CoreSight™ от ARM. Производители микроконтроллеров могут реализовать интерфейс отладки на базе любого из указанных модулей.

Для обеспечения возможности трассировки команд производители микросхем могут также включать в свои изделия *модуль встроенной макроячейки трассировки* (Embedded Trace Macrocell — ETM). Трассировочная информация выводится через *модуль интерфейса порта трассировки* (Trace Port Interface Unit — TPIU). Далее информация о выполненных командах с помощью внешнего аппаратного трассировщика передаётся хосту отладки, в качестве которого обычно применяется персональный компьютер.

В самом процессоре для запуска отладочных действий могут использоваться различные события. Источниками этих событий могут служить точки останова, точки наблюдения, отказы или сигналы от внешнего отладчика. При возникновении любого из указанных событий процессор может либо перейти в режим останова, либо запустить обработчик исключения монитора отладки.

Функция точек наблюдения данных реализуется *модулем просмотра и трассировки данных* (Data Watchpoint and Trace — DWT) процессора Cortex-M3. Этот модуль может быть задействован для останова процессора (или для запуска обработчика исключения монитора отладки) или же для генерации информации о трассировке данных. При осуществлении трассировки соответствующая информация выводится через модуль TPIU. Следует отметить, что в архитектуре CoreSight единственный порт трассировки может использоваться совместно несколькими устройствами трассировки.

Помимо описанных базовых средств отладки, в процессоре Cortex-M3 также имеется *модуль коррекции флэш-памяти и задания точки останова* (Flash Patch and Breakpoint — FPB). Этот модуль может использоваться для установки простых точек останова или же для переназначения адресов команд из области флэш-памяти в область ОЗУ.

Макроячейка инструментальной трассировки (Instrumentation Trace Macrocell — ITM) предоставляет разработчику новый канал для передачи данных в отладчик. Данные, записываемые в регистры модуля ITM, могут быть получены отладчиком по интерфейсу трассировки для их последующего отображения или обработки. Этот метод прост в использовании и более быстрый, нежели вывод по интерфейсу JTAG.

Управление всеми указанными компонентами отладки осуществляется по шине интерфейса DAP процессора Cortex-M3 или же программой, выполняемой процессором. Вся информация о процессе трассировки может быть получена из модуля TPIU.

2.11. Резюме

Почему процессор Cortex-M3 считается революционным устройством? Какие преимущества даёт его использование? Ответы на эти и аналогичные вопросы можно найти в данном разделе.

2.11.1. Высокая производительность

Процессор Cortex-M3 обеспечивает высокое быстродействие микроконтроллеров:

- Большинство команд, включая команды умножения, выполняются за один такт. По данному параметру Cortex-M3 опережает большинство популярных микроконтроллеров.
- Раздельные шины команд и данных позволяют одновременно выполнять операции выборки команд и обращения к данным.
- Набор команд Thumb-2 исключает непроизводительные издержки на переключение состояний процессора. Больше не нужно тратить время на переход между 32-битным состоянием ARM и 16-битным состоянием Thumb, что увеличивает скорость выполнения программы и уменьшает её размер. Это новшество также упрощает разработку программного обеспечения, что, в свою очередь, сокращает время выхода продукции на рынок и облегчает последующее сопровождение кода.

- Набор Thumb-2 является чрезвычайно гибким. Многие операции могут быть выполнены с использованием меньшего числа команд. Как следствие, Cortex-M3 обеспечивает бóльшую плотность кода и требует меньше памяти для хранения программ.
- Выборка команд осуществляется 32-битными словами, соответственно, за один такт может быть выбрано до двух команд. В результате обеспечивается более высокая пропускная способность для передачи данных.
- Конструкция процессора Cortex-M3 позволяет создавать микроконтроллеры, работающие на высокой частоте (при использовании современных техпроцессов — более 100 МГц). Но даже при работе на той же частоте, что и большинство других микроконтроллеров, Cortex-M3 имеет лучшее соотношение числа тактов на одну команду (CPI). Это позволяет выполнять большее число операций в пересчёте на мегагерц тактовой частоты или же даёт возможность снизить тактовую частоту для уменьшения энергопотребления.

2.11.2. Развитые средства поддержки прерываний

Средства поддержки прерываний в процессоре Cortex-M3 легки в использовании, обладают большой гибкостью и обеспечивают высокую производительность при обработке прерываний:

- Встроенный контроллер прерываний NVIC поддерживает до 240 входов внешних прерываний. Поддержка векторных прерываний значительно уменьшает задержку обработки прерываний, поскольку выбор необходимого обработчика осуществляется полностью аппаратно. Кроме того, не требуется прибегать к программным ухищрениям для обеспечения поддержки вложенных прерываний.
- Процессор Cortex-M3 при входе в процедуру обработки прерывания автоматически сохраняет в стеке регистры R0…R3, R12, LR, PSR и PC и извлекает их из стека при выходе из обработчика. Это уменьшает задержку обработки запроса прерывания и даёт возможность описывать обработчик прерывания в виде обычной функции на языке Си (см. Главу 8).
- Система прерываний является чрезвычайно гибкой, поскольку NVIC позволяет задавать приоритет индивидуально для каждого прерывания. Поддерживаются не менее 8 уровней приоритета, причём приоритет может изменяться динамически.
- Для уменьшения задержки обработки прерываний используются такие специальные методы, как принятие «опоздавших» прерываний и прямой переход (tail-chain) от одного обработчика прерывания к другому.
- Допускается прерывание некоторых операций, выполняемых за несколько тактов, в том числе операций загрузки/сохранения нескольких регистров (`LDM`/`STM`) и операций сохранения в стеке и извлечения из стека (`PUSH`/`POP`).

При появлении запроса немаскируемого прерывания гарантируется немедленный запуск обработчика этого прерывания, если только система не является полностью заблокированной. Наличие немаскируемого прерывания является

очень важным фактором для большинства приложений, предъявляющих повышенные требования к безопасности.

2.11.3. Низкое энергопотребление

Процессор Cortex-M3 может использоваться в самых разных приложениях, требующих низкого энергопотребления:

- Процессор Cortex-M3 позволяет создавать устройства с низким энергопотреблением, поскольку содержит относительно небольшое число логических вентилей.
- Процессор Cortex-M3 имеет два режима пониженного потребления (SLEEPING и SLEEPDEEP). Для перехода в эти режимы используются команды `WFI` или `WFE`. Конструкцией процессора предусмотрено раздельное тактирование его основных блоков, что позволяет останавливать большинство узлов процессора на время спящего режима.
- Полностью статическое, синхронное и синтезируемое ядро позволяет легко изготавливать данный процессор как по специальному (low-power), так и по стандартному техпроцессам.

2.11.4. Системные возможности

В процессоре Cortex-M3 реализованы определённые возможности, позволяющие использовать его в самых разных приложениях:

- Поддерживается прямой доступ к битам данных (метод bit-band), хранение данных в формате с обратным порядком и неизменным расположением байтов (byte-invariant big endian), а также обращения к невыровненным данным.
- Расширенная поддержка обработки системных отказов, в том числе наличие разных типов исключений и регистров состояния отказов, облегчает обнаружение источника проблемы.
- Благодаря наличию теневого указателя стека, области стека ядра ОС и пользовательских процессов могут быть изолированы друг от друга. При использовании опционального модуля MPU возможностей процессора оказывается более чем достаточно для разработки отказоустойчивого программного обеспечения и надёжных устройств.

2.11.5. Поддержка отладки

Процессор Cortex-M3 имеет богатые функции отладки, облегчающие пользователям разработку своих устройств:

- Поддерживает отладочные интерфейсы JTAG или Serial-Wire.
- Благодаря использованию технологии отладки CoreSight, состояние процессора и содержимое памяти может быть получено без останова ядра процессора.
- Имеется встроенная поддержка до шести точек останова и до четырёх точек наблюдения.
- Имеется опциональный модуль ETM для трассировки команд и модуль DWT для трассировки данных.

- Новые возможности отладки, включая регистры состояния отказов, новые исключения отказов и поддержка функций коррекции флэш-памяти (Flash Patch) значительно облегчают процесс отладки.
- Модуль ITM предоставляет удобный способ вывода отладочной информации из тестового кода.
- Схема выборки значений счётчика команд и счётчики, имеющиеся в модуле DWT, предоставляют информацию, необходимую для профилирования кода.

ГЛАВА 3
ОСНОВЫ CORTEX-M3

3.1. Регистры

Как мы уже знаем, процессор Cortex™-M3 имеет 16 регистров с R0 по R15 и несколько регистров специального назначения. Регистры с R0 по R12 являются регистрами общего назначения. Ряд команд набора Thumb® могут обращаться только к младшим регистрам R0…R7, тогда как 32-битные команды Thumb-2 могут работать со всеми регистрами. Регистры специального назначения выполняют предопределённые функции и для обращения к ним необходимо использовать особые команды.

3.1.1. Регистры общего назначения с R0 по R7

Регистры общего назначения с R0 по R7 также называются *младшими регистрами*. Обращаться к этим регистрам могут все команды как 16-битного набора Thumb, так и 32-битного набора Thumb-2. Все указанные регистры имеют разрядность 32 бита, состояние регистров после сброса может быть любым.

3.1.2. Регистры общего назначения с R8 по R12

Регистры общего назначения с R8 по R12 также называются *старшими регистрами*. Обращаться к этим регистрам могут все команды набора Thumb-2 и некоторые команды набора Thumb. Все указанные регистры имеют разрядность 32 бита, состояние регистров после сброса может быть любым (**Рис. 3.1**).

3.1.3. Указатель стека R13

Регистр процессора R13 используется в качестве указателя стека, причём в процессоре Cortex-M3 имеется два таких указателя. Наличие двух указателей позволяет реализовать два независимых стека. Используя идентификатор R13, можно обращаться только к текущему указателю, другой указатель при этом будет недоступен. Обратиться к недоступному в данный момент указателю можно только с помощью команд пересылки между регистром общего назначения и регистром специального назначения (`MSR` и `MRS`). Итак, в процессоре Cortex-M3 имеются следующие указатели стека:

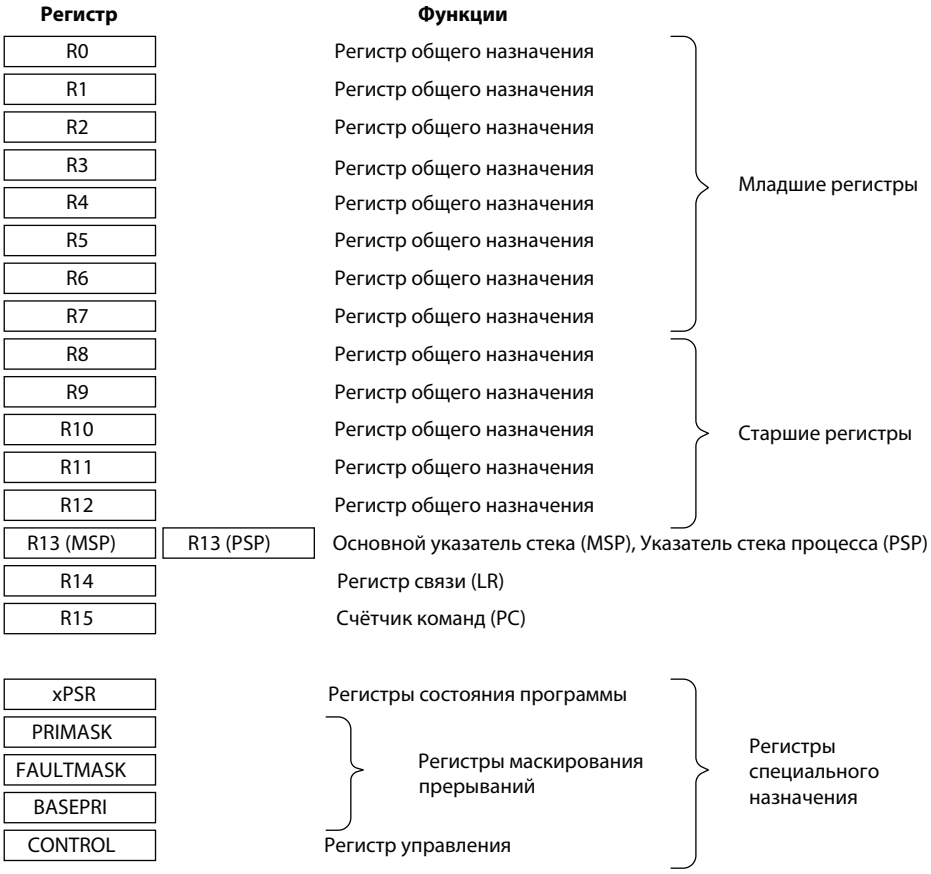


Рис. 3.1. *Регистры процессора Cortex-M3.*

- *Основной указатель стека MSP* (в документации ARM он обозначается как *SP_main*). Это указатель, используемый по умолчанию. Он используется ядром операционной системы, обработчиками исключительных ситуаций, а также всеми программными модулями, требующими привилегированного доступа к памяти.
- *Указатель стека процесса PSP* (в документации ARM он обозначается как *SP_process*). Этот указатель используется прикладной программой (если только не выполняется обработчик исключительной ситуации).

Разумеется, совсем необязательно использовать оба указателя стека. В простых приложениях вполне можно обойтись одним указателем, MSP. В любом случае указатели стека применяются при операциях со стеком, таких как помещение в стек и извлечение из стека.

Загрузка данных в стек и извлечение данных из стека

Под стеком понимается определённая модель использования памяти. Вообще говоря, стеком является просто некий участок системной памяти, который за счёт использования регистра указателя, имеющегося в процессоре, работает как буфер магазинного типа (LIFO-буфер). Чаще всего стек задействуется для сохранения содержимого регистров перед выполнением каких-либо операций над данными и последующего восстановления содержимого этих регистров по окончании обработки данных.

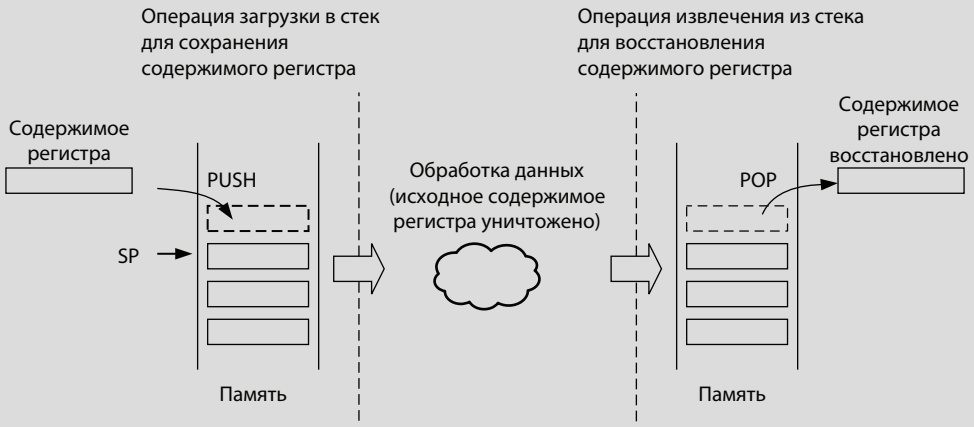


Рис. 3.2. *Стек, основные концепции.*

При выполнении операций загрузки в стек и извлечения из стека регистр указателя, обычно называемый указателем стека, автоматически изменяется таким образом, чтобы последующие операции со стеком не повредили ранее сохранённые данные. Более подробно о стековых операциях рассказывается ниже в этой главе.

В процессоре Cortex-M3 для работы со стеком предусмотрены команды `PUSH` (загрузка регистров в стек) и `POP` (извлечение регистров из стека). Эти команды имеют следующий синтаксис (текст после каждого символа «;» является комментарием):

```
PUSH {R0}  ; R13 = R13 - 4, затем Memory[R13] = R0
POP  {R0}  ; R0 = Memory[R13], затем R13 = R13 + 4
```

В процессоре Cortex-M3 реализован «полный» убывающий стек (более подробно об этом можно узнать из раздела 3.6 «Операции со стеком»). Соответственно, указатель стека при помещении нового значения в стек декрементируется. Команды `PUSH` и `POP` обычно используются для сохранения содержимого регистров в стеке при входе в подпрограмму и последующего восстановления содержимого этих регистров из стека перед выходом из подпрограммы. С помощью одной команды можно загрузить в стек или восстановить из стека сразу несколько регистров:

```
subroutine_1
    PUSH   {R0-R7, R12, R14} ; Сохраняем регистры
    ...                      ; Выполняем требуемые действия
    POP    {R0-R7, R12, R14} ; Восстанавливаем регистры
    BX     R14               ; Возвращаемся в вызвавшую функцию
```

Вместо названия регистра R13 в программе можно использовать символическое имя SP, эти обозначения полностью эквивалентны. И к указателю MSP, и к указателю PSP из программы можно обращаться как к R13/SP. Тем не менее, используя команды доступа к регистрам специального назначения (MRS/MSR), можно обратиться к конкретному указателю.

Указатель MSP, также называемый *SP_main* в документации ARM, является указателем стека по умолчанию после включения процессора; он используется ядром ОС и обработчиками исключений. Указатель PSP, также называемый *SP_process* в документации ARM, обычно задействуется пользовательскими процессами при наличии встроенной операционной системы.

Поскольку операции сохранения регистров в стеке и восстановления регистров из стека всегда выровнены на границу 32-битного слова (их адреса должны быть равны 0x00, 0x04, 0x08, …), биты 0 и 1 регистра SP/R13 аппаратно сброшены в 0.

3.1.4. Регистр связи R14

Регистр R14 используется в качестве регистра связи (Link Register — LR). В ассемблерной программе вы можете использовать любое из обозначений регистра — R14 или LR. Регистр связи применяется для сохранения адреса возврата при вызове процедуры или функции, например при выполнении команды перехода со ссылкой (BL):

```
    main          ; Основная программа
    ...
    BL function1  ; Вызываем function1, используя команду перехода со ссылкой
                  ; PC = адрес function1, а
                  ; LR = адрес следующей команды в main
    ...
function1
    ...           ; Тело подпрограммы function1
    BX LR         ; Возвращаемся
```

Несмотря на то что бит 0 счётчика команд всегда сброшен в 0 (поскольку команды выровнены на границу слова или полуслова), бит 0 регистра LR доступен как для чтения, так и для записи. Это связано с тем, что в наборе команд Thumb 0-й бит часто используется для указания состояния ARM/Thumb процессора. Чтобы программы, написанные для процессора Cortex-M3, могли запускаться на других процессорах ARM, поддерживающих технологию Thumb-2, младший бит регистра связи сделан доступным и для чтения, и для записи.

3.1.5. Счётчик команд R15

Регистр R15 служит в качестве счётчика команд PC. Для обращения к данному регистру можно использовать любое из обозначений — R15 или PC. Из-за наличия в процессоре Cortex-M3 конвейера значение, считанное из этого регистра, будет отличаться от адреса исполняемой в данный момент команды (как правило, на 4).

```
    0x1000 : MOV R0, PC ; R0 = 0x1004
```

В других командах, в частности выполняющих загрузку констант (чтение ячейки памяти по адресу, задаваемому относительно текущего значения PC), действительное значение PC может не быть равным адресу команды, увеличенному на 4, из-за выравнивания при вычислении адреса. Но в любом случае значение PC, как минимум, на 2 байта превышает значение адреса исполняемой команды.

Запись в счётчик команд вызовет переход (LR при этом не изменяется). Поскольку адрес команды должен быть выровнен на границу полуслова, младший бит значения, считываемого из PC, всегда сброшен в 0. Однако при выполнении перехода, как при помощи команды перехода, так и путём непосредственной записи в PC, младший бит конечного адреса должен быть установлен в 1, поскольку он используется для индикации состояния Thumb. Если этот бит окажется сброшенным в 0, то процессор попытается переключиться в состояние ARM, что вызовет генерацию исключения отказа.

3.2. Регистры специального назначения

В процессоре Cortex-M3 имеются следующие регистры специального назначения (**Рис. 3.3** и **3.4**):

- регистры состояния программы (xPSR);
- регистры маскирования прерываний (PRIMASK, FAULTMASK и BASEPRI);
- регистр управления (CONTROL).

	31	*30*	*29*	*28*	*27*	*26:25*	*24*	*23:20*	*19:16*	*15:10*	*9*	*8*	*7*	*6*	*5*	*4:0*
APSR	N	Z	C	V	Q											
IPSR												Номер исключения				
EPSR						ICI/IT	T			ICI/IT						

Рис. 3.3. *Отдельные регистры состояния программы процессора Cortex-M3.*

	31	*30*	*29*	*28*	*27*	*26:25*	*24*	*23:20*	*19:16*	*15:10*	*9*	*8*	*7*	*6*	*5*	*4:0*
xPSR	N	Z	C	V	Q	ICI/IT	T			ICI/IT		Номер исключения				

Рис. 3.4. *Объединённые регистры состояния программы (xPSR) процессора Cortex-M3.*

Указанные регистры не имеют собственных адресов, поэтому для обращения к ним необходимо использовать команды `MSR` и `MRS`:

```
MRS <reg>, <special_reg>; Чтение регистра специального назначения
MSR <special_reg>, <reg>; Запись в регистр специального назначения
```

3.2.1. Регистры состояния программы

Группу регистров состояния программы составляют три регистра:

- регистр состояния приложения APSR;
- регистр состояния прерывания IPSR;
- регистр состояния выполнения EPSR.

Обращения к этим трём регистрам производятся посредством команд `MSR` и `MRS`, причём обращаться можно как к каждому регистру по отдельности, так и одновременно ко всем регистрам группы. В последнем случае используется обозначение *xPSR*.

С помощью команды `MSR` можно выполнять чтение всех регистров состояния программы, а используя команду `MRS`, можно осуществлять запись в регистр APSR (регистры IPSR и EPSR доступны только для чтения). Например:

```
MRS r0, APSR ; Считать состояние флагов в R0
MRS r0, IPSR ; Считать состояние исключения/прерывания
MRS r0, EPSR ; Считать состояние выполнения
MSR APSR, r0 ; Записать состояние флагов
```

При одновременном обращении ко всем регистрам состояния используется обозначение *PSR*:

```
MRS   r0, PSR    ; Чтение всех регистров состояния программы
MSR   PSR, r0    ; Запись во все регистры состояния программы
```

Назначение битов регистров состояния программы указано в **Табл. 3.1**.

Таблица 3.1. **Биты регистров состояния программы**

Бит	Описание
N	Отрицательное значение
Z	Нулевое значение
C	Перенос/заём
V	Переполнение
Q	Насыщение
ICI/IT	Состояние команды, возобновляемой после прерывания (ICI); состояние команды `IF-THEN` (`IT`)
T	Признак состояния Thumb, всегда установлен в 1. Попытка сброса этого бита вызовет генерацию исключения отказа
Номер исключения	Содержит номер исключения, обрабатываемого процессором в данный момент

Если сравнить эти регистры с регистром текущего состояния программы CPSR процессора ARM7, то можно заметить отсутствие некоторых битов. Так, были убраны биты режима (M), поскольку Cortex-M3 не имеет режимов работы в том виде, в котором они были определены в процессоре ARM7. Бит Thumb (T) был перемещён в 24-й бит регистра. Биты состояния прерывания (I и F) были заменены новым регистром PRIMASK. Для сравнения, на **Рис. 3.5** показаны форматы регистра CPSR традиционных процессоров ARM.

	31	*30*	*29*	*28*	*27*	*26:25*	*24*	*23:20*	*19:16*	*15:10*	*9*	*8*	*7*	*6*	*5*	*4:0*
ARM (обобщённый)	N	Z	C	V	Q	IT	J	Зарезер-вировано	GE[3:0]	IT	E	A	I	F	T	M[4:0]
ARM7TDMI	N	Z	C	V	Зарезервировано								I	F	T	M[4:0]

Рис. 3.5. *Регистры состояния программы в традиционных процессорах ARM.*

3.2.2. Регистры PRIMASK, FAULTMASK и BASEPRI

Регистры PRIMASK, FAULTMASK и BASEPRI используются для запрещения исключений (**Табл. 3.2**).

Таблица 3.2. **Регистры маскирования прерываний**

Регистр	Описание
PRIMASK	Однобитный регистр, который, будучи установлен в 1, разрешает генерацию только немаскируемого прерывания и исключения Hard Fault; все остальные прерывания и исключения маскируются. По умолчанию этот регистр содержит 0, что означает отсутствие маскирования
FAULTMASK	Однобитный регистр, который, будучи установлен в 1, разрешает генерацию только немаскируемого прерывания; все прерывания и системные исключения маскируются. По умолчанию этот регистр содержит 0, что означает отсутствие маскирования
BASEPRI	Регистр разрядностью до 8 бит (зависит от разрядности регистров, используемых для задания уровня приоритета). Данный регистр определяет уровень приоритета маскируемых прерываний. Запись в этот регистр значения, отличного от нуля, запрещает все прерывания, имеющие такой же или меньший приоритет (большее значение уровня приоритета). Прерывания с более высоким приоритетом при этом разрешены. При наличии в регистре нулевого значения (состояние по умолчанию) маскирование прерываний отключено

Регистры PRIMASK и BASEPRI полезны для временного запрещения прерываний в задачах, критичных ко времени исполнения. Операционная система может использовать регистр FAULTMASK для временного запрещения обработки отказов при аварийном завершении задачи. В этом случае могут возникать различные исключения. Однако после того, как ядром ОС был запущен процесс восстановления из некорректного состояния, прерывание его работы другим исключением, вызванным «упавшим» процессом, может оказаться нежелательным. Таким образом, регистр FAULTMASK предоставляет ядру ОС время, необходимое для принятия соответствующих мер при возникновении сбоя.

Для обращения к регистрам PRIMASK, FAULTMASK и BASEPRI используются специальные функции, которые входят в состав библиотек, предоставляемых изготовителями микроконтроллеров. Например:

```
x = __get_BASEPRI();   // Чтение регистра BASEPRI
x = __get_PRIMARK();   // Чтение регистра PRIMASK
x = __get_FAULTMASK(); // Чтение регистра FAULTMASK
__set_BASEPRI(x);      // Запись нового значения в BASEPRI
__set_PRIMASK(x);      // Запись нового значения в PRIMASK
__set_FAULTMASK(x);    // Запись нового значения в FAULTMASK
__disable_irq();       // Очистка PRIMASK, разрешение прерываний
__enable_irq();        // Установка PRIMASK, запрещение прерываний
```

Более подробно эти функции доступа к регистрам процессорного ядра описаны в Приложении Ж. Достаточно полные сведения о стандарте CMSIS (Cortex Microcontroller Software Interface Standard — стандарт программного интерфейса микроконтроллеров с ядром Cortex) можно найти в Главе 10.

В ассемблерных программах для этих же целей используются команды `MRS` и `MSR`, например:

```
MRS  r0, BASEPRI   ; Считать регистр BASEPRI в R0
MRS  r0, PRIMASK   ; Считать регистр PRIMASK в R0
MRS  r0, FAULTMASK ; Считать регистр FAULTMASK в R0
MSR  BASEPRI, r0   ; Записать R0 в регистр BASEPRI
MSR  PRIMASK, r0   ; Записать R0 в регистр PRIMASK
MSR  FAULTMASK, r0 ; Записать R0 в регистр FAULTMASK
```

Регистры PRIMASK, FAULTMASK и BASEPRI недоступны на пользовательском (непривилегированном) уровне доступа.

3.2.3. Регистр управления CONTROL

Регистр управления используется для задания уровня доступа и выбора указателя стека. Этот регистр содержит два бита, назначение которых указано в **Табл. 3.3**.

Таблица 3.3. **Регистр управления Cortex-M3**

Бит	Описание
CONTROL[1]	Состояние стека: 1 — используется альтернативный стек; 0 — используется основной стек (MSP). При работе процессора в режиме потока альтернативным стеком является PSP. Для режима обработчика дополнительный стек не определён, поэтому при работе процессора в режиме обработчика этот бит должен быть сброшен в 0
CONTROL[0]	0 — привилегированный уровень доступа в режиме потока; 1 — пользовательский уровень доступа в режиме потока. В режиме обработчика процессор всегда работает на привилегированном уровне доступа

CONTROL[1]

При нахождении процессора Cortex-M3 в режиме обработчика бит CONTROL[1] всегда сброшен в 0. В режиме потока этот бит может быть равным как 0, так и 1.

Этот бит доступен для записи только при работе ядра процессора в режиме потока на привилегированном уровне. При работе на пользовательском уровне или в режиме обработчика запись в указанный бит запрещена. Изменить данный бит можно не только записью в регистр, но и изменив 2-й бит регистра LR при возврате из обработчика исключения. Более подробно об этом будет сказано в Главе 8, посвящённой исключениям.

CONTROL[0]

Бит CONTROL[0] доступен для записи только в привилегированном состоянии. После переключения процессора на пользовательский уровень доступа единственный способ возврата в прежнее состояние состоит в генерации прерывания и изменении состояния процессора в обработчике этого прерывания.

Для обращения к регистру управления из программ на языке Си в CMSIS-совместимых библиотеках драйверов устройств предусмотрены следующие функции:

```
x = __get_CONTROL(); // Считать текущее значение регистра CONTROL
```

```
__set_CONTROL(x);    // Загрузить в регистр CONTROL значение x
```

В ассемблерных программах для обращения к регистру управления используются команды MRS и MSR:

```
MRS  r0, CONTROL   ; Прочитать содержимое регистра CONTROL в R0
MSR  CONTROL, r0   ; Записать R0 в регистр CONTROL
```

3.3. Режимы работы

Процессор Cortex-M3 поддерживает два режима работы и два уровня доступа к коду программы (**Рис. 3.6**).

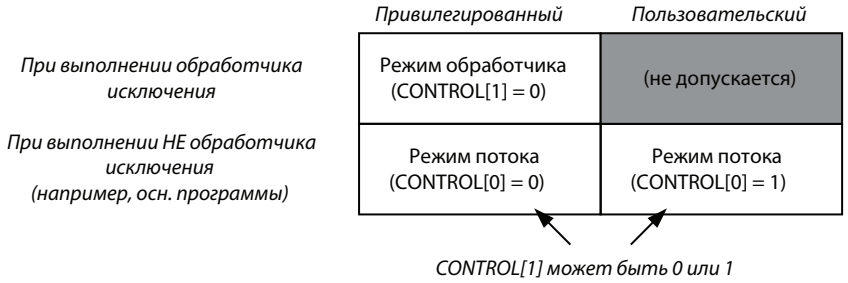


Рис. 3.6. *Режимы работы и уровни доступа к коду процессора Cortex-M3.*

При работе процессора в режиме потока он может находиться как на привилегированном, так и на пользовательском уровне, тогда как обработка исключительных ситуаций всегда осуществляется на привилегированном уровне. После сброса процессор находится в режиме потока с правами привилегированного доступа к коду.

На пользовательском уровне доступа (режим потока) обращения к пространству управления системой (System Control Space — SCS) — области памяти, содержащей регистры конфигурации и компоненты отладки, — заблокированы. Более того, на этом уровне запрещено даже использование команд, обращающихся к регистрам специального назначения (таких как MSR, за исключением обращений к регистру APSR). Если программа, выполняющаяся на пользовательском уровне доступа, попытается обратиться к пространству SCS или к регистрам специального назначения, то будет сгенерировано исключение отказа.

Программное обеспечение, работающее на привилегированном уровне, может переключиться на пользовательский уровень, используя регистр управления. При возникновении исключительной ситуации процессор всегда переключается на привилегированный уровень, а при выходе из обработчика исключения — возвращается на исходный. Пользовательская программа не может сама переключиться на привилегированный уровень, выполнив запись в регистр управления. Она должна воспользоваться обработчиком исключения, который запрограммирует регистр CONTROL так, чтобы процессор переключился на привилегированный уровень при возврате в режим потока (**Рис. 3.7**).

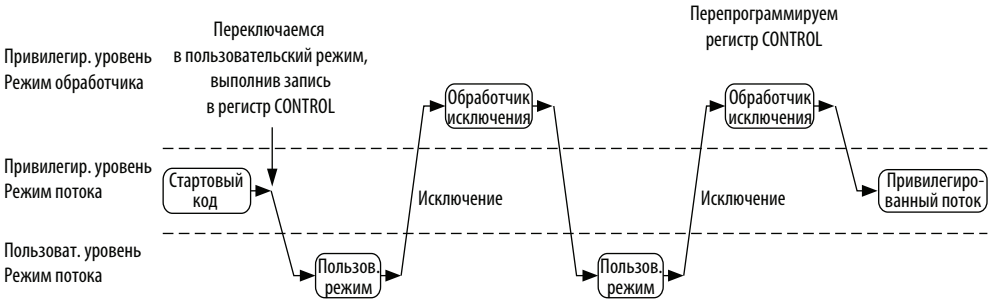


Рис. 3.7. *Переключение режимов работы процессора.*

Поддержка процессором двух уровней доступа обеспечивает более безопасную и надёжную архитектуру. Например, при некорректной работе пользовательской программы она никогда не сможет повредить содержимое регистров управления контроллера прерываний. А при наличии модуля MPU можно вообще заблокировать пользовательским программам доступ к областям памяти, используемым привилегированными процессами.

В простых приложениях нет нужды разделять привилегированный и пользовательский уровни доступа. В таких случаях пользовательский уровень доступа не используется и, соответственно, программирование регистра управления не требуется.

Вы можете отделить стек пользовательского приложения от стека ядра операционной системы, чтобы предотвратить крах системы из-за некорректной работы со стеком в пользовательской программе. В этом случае пользовательская программа, работающая в режиме потока, будет использовать указатель PSP, а обработчики исключений — указатель MSP. Переключение между указателями стеков осуществляется автоматически при входе в обработчики и при выходе из них (см. подраздел 3.6.3). Более подробно этот вопрос рассматривается в Главе 8.

Режим работы процессора и уровень доступа к коду определяются регистром управления. Если 0-й бит регистра управления сброшен в 0, то при генерации исключения режим процессора будет изменён (**Рис. 3.8** и **3.9**).

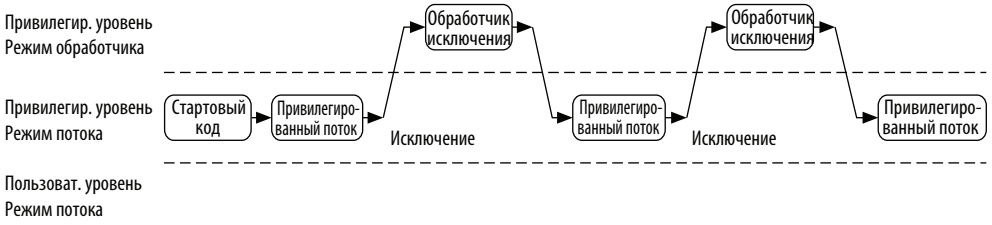


Рис. 3.8. *Простая программа, не использующая пользовательский уровень доступа в режиме потока.*

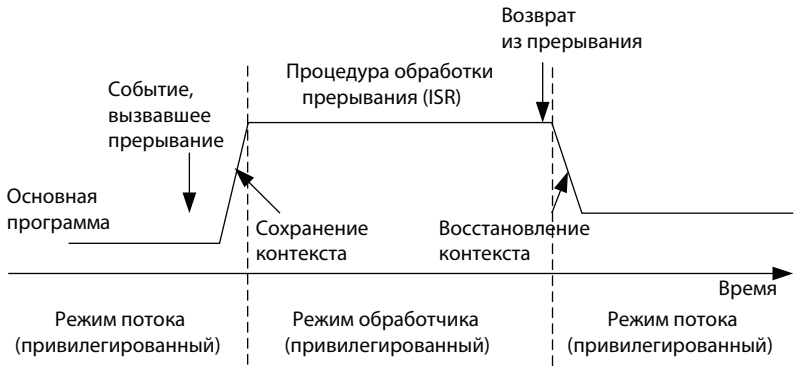


Рис. 3.9. *Переключение режима работы процессора в прерывании.*

Если 0-й бит регистра управления установлен в 1, то при генерации исключения будут изменены и режим процессора, и уровень доступа (**Рис. 3.10**).

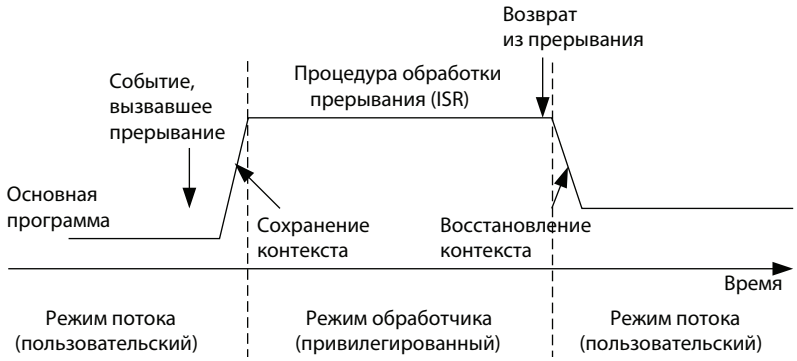


Рис. 3.10. *Переключение режима работы процессора и уровня доступа в прерывании.*

Запись в 0-й бит регистра управления допускается только на привилегированном уровне (см. **Рис. 2.5**). Чтобы программа, выполняющаяся на пользовательском уровне, смогла переключиться в привилегированное состояние, она должна сгенерировать прерывание (например, посредством команды вызова супервизора `SVC`) и изменить бит CONTROL[0] в обработчике прерывания.

3.4. Исключения и прерывания

Процессор Cortex-M3 поддерживает определённое число исключений, в том числе фиксированное количество системных исключений и ряд прерываний, обычно называемых *IRQ*. Количество входов прерываний в микроконтроллерах с процессором Cortex-M3 зависит от конкретной реализации. Прерывания, генерируемые периферийными устройствами процессора, за исключением системного таймера, тоже подключаются к входам прерываний. Как правило, используется 16 или 32 входа прерываний. Однако вы вполне можете встретить микроконтроллеры, имеющие большее (или меньшее) число прерываний.

Помимо входов обычных прерываний, в процессоре также имеется вход немаскируемого прерывания (NMI). Фактическое использование немаскируемого прерывания зависит от конкретного микроконтроллера или системы на кристалле. В большинстве случаев вход NMI может быть подключён к сторожевому таймеру или блоку контроля напряжения питания, который сообщит процессору о падении напряжения ниже заданного уровня. Немаскируемое прерывание может быть активировано в любой момент времени, в том числе сразу же после выхода процессора из состояния сброса.

Перечень исключений, поддерживаемых процессором Cortex-M3, приведён в **Табл. 3.4**. Некоторые системные исключения предназначены для обработки отказов процессора и могут генерироваться при возникновении различных нештатных ситуаций. В контроллере NVIC также имеется несколько регистров состояния отказов, с помощью которых обработчики исключений могут определить причину возникновения исключительной ситуации.

Более подробно исключения процессора Cortex-M3 рассматриваются в Главах 7…9.

Таблица 3.4. **Типы исключений в процессоре Cortex-M3**

Номер исключения	Тип исключения	Приоритет	Описание
1	Reset	–3 (наивысший)	Сброс
2	NMI	–2	Немаскируемое прерывание
3	Hard Fault	–1	Любой отказ, если соответствующий обработчик не может быть запущен (исключение в данный момент запрещено или маскировано)
4	MemManage Fault	Программируемый	Отказ системы управления памятью; нарушение правил доступа, заданных модулем MPU, или обращение по некорректному адресу (например, при выборке команды из области памяти, не предназначенной для хранения исполняемого кода)
5	Bus Fault	Программируемый	Отказ шины; возникает при отказе предвыборки команды или при ошибке доступа к данным
6	Usage Fault	Программируемый	Отказ программы; обычно возникает при попытке исполнить неверную команду или переключиться в недопустимое состояние (скажем, переключить процессор Cortex-M3 в состояние ARM)
7…10	—	—	Зарезервировано
11	SVCall	Программируемый	Вызов супервизора посредством команды `SVC`
12	Debug monitor	Программируемый	Исключение монитора отладки
13	—	—	Зарезервировано
14	PendSV	Программируемый	Запрос системной службы
15	SYSTICK	Программируемый	Системный таймер
16…255	IRQ	Программируемый	Вход прерывания №0…239

3.5. Таблица векторов

При возникновении события, генерирующего исключение (если данное исключение разрешено), запускается обработчик соответствующего исключения. Для определения начального адреса обработчика предназначена таблица векторов. *Таблица векторов* представляет собой массив 32-битных значений, расположенных в системной области памяти, каждое из которых является начальным адресом обработчика одного конкретного исключения. Положение таблицы векторов можно изменять — оно определяется регистром смещения таблицы векторов VTOR контроллера NVIC. После сброса этот регистр содержит нулевое значение, поэтому при включении процессора таблица векторов располагается, начиная с адреса 0x00 (см. **Табл. 3.5**).

Таблица 3.5. **Таблица векторов после сброса**

Номер исключения	Смещение	Вектор обработчика исключения
18…255	0x48…0x3FF	IRQ №2…239
17	0x44	IRQ №1
16	0x40	IRQ №0
15	0x3C	SYSTICK
14	0x38	PendSV
13	0x34	Зарезервировано
12	0x30	Debug monitor
11	0x2C	SVCall
7…10	0x1C…0x28	Зарезервировано
6	0x18	Usage Fault
5	0x14	Bus Fault
4	0x10	MemManage Fault
3	0x0C	Hard Fault
2	0x08	NMI
1	0x04	Сброс
0	0x00	Начальное значение MSP

К примеру, если событие сброса является исключением с номером 1, то адрес вектора сброса равен 1 × 4 (каждое слово содержит 4 байта), т.е. 0x00000004. Вектор NMI (номер 2) расположен по адресу 2 × 4 = 0x00000008. Адрес 0x00000000 используется для хранения начального значения основного указателя стека.

Младший бит каждого вектора указывает, в каком состоянии должен выполняться соответствующий обработчик. Поскольку процессор Cortex-M3 поддерживает только команды Thumb, младшие биты всех векторов исключений должны быть установлены в 1.

3.6. Стек

Процессор Cortex-M3 поддерживает обычные программно-управляемые операции загрузки в стек и извлечения из стека. Кроме того, операции загрузки в стек и извлечения из стека выполняются автоматически при входе и выходе из

обработчика исключения/прерывания. В данном разделе мы рассмотрим программную работу со стеком (стековые операции при обработке исключений рассматриваются в Главе 9).

3.6.1. Основные стековые операции

В общем случае стековые операции представляют собой операции записи или чтения памяти по адресу, определяемому указателем стека. Содержимое регистров сохраняется в стеке посредством операции загрузки в стек и может быть восстановлено позже посредством операции извлечения из стека. При выполнении этих операций указатель стека изменяется автоматически таким образом, чтобы многократная загрузка данных в стек не привела к стиранию ранее сохранённых данных.

Стек предназначен для временного сохранения содержимого регистров в памяти и последующего его восстановления после завершения некоторой задачи. При обычном использовании стека каждой операции загрузки должна соответствовать своя операция извлечения, причём адрес, используемый в этих операциях, должен быть одним и тем же (**Рис. 3.11**). При выполнении команд PUSH/POP указатель стека инкрементируется и декрементируется автоматически.

Основная программа

```
    ...
; R0   X, R1   Y, R2   Z
BL    function1                    Подпрограмма

                function1
                    PUSH    {R0} ; Сохраняем R0 в стеке (SP инкрементируется)
                    PUSH    {R1} ; Сохраняем R1 в стеке (SP инкрементируется)
                    PUSH    {R2} ; Сохраняем R2 в стеке (SP инкрементируется)
                    ... ; Выполняем задачу (R0, R1 и R2
                        ; можно изменять)
                    POP     {R2} ; Восстанавливаем R2 (SP декрементируется)
                    POP     {R1} ; Восстанавливаем R1 (SP декрементируется)
                    POP     {R0} ; Восстанавливаем R0 (SP декрементируется)
                    BX      LR   ; Возврат

; Вернулись в основную программу
; R0   X, R1   Y, R2   Z
... ; Следующие команды
```

Рис. 3.11. *Основные стековые операции: одна операция — один регистр.*

При возврате управления в основную программу содержимое регистров R0…R2 будет таким же, что и до вызова подпрограммы. Обратите внимание на порядок выполнения операций: операции извлечения из стека выполняются в обратном порядке по сравнению с операциями загрузки в стек.

Данный процесс можно упростить, так как команды PUSH и POP поддерживают сохранение и восстановление нескольких регистров. В этом случае порядок

регистров, восстанавливаемых из стека, автоматически изменяется процессором на обратный (**Рис. 3.12**).

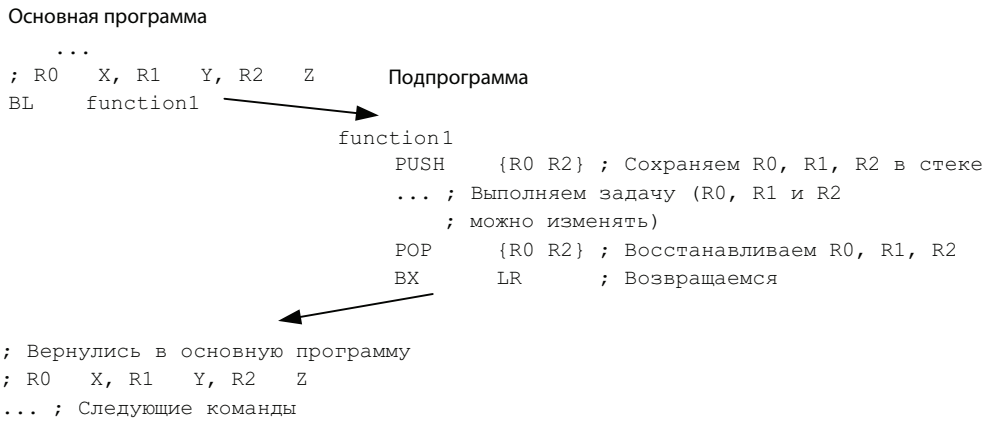


Рис. 3.12. *Основные стековые операции: одна операция — несколько регистров.*

Вы также можете объединить операцию возврата из подпрограммы с операцией извлечения из стека. Для этого необходимо сохранить содержимое регистра LR в стеке и извлечь его в счётчик команд в конце подпрограммы (**Рис. 3.13**).

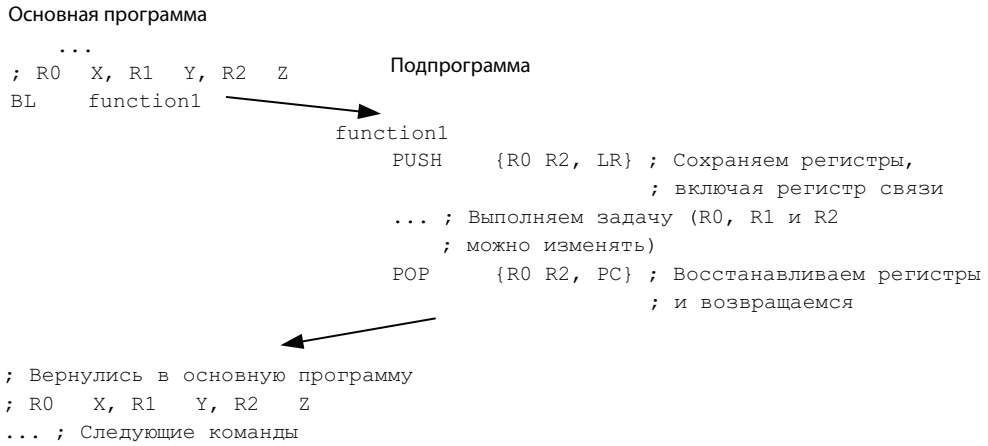


Рис. 3.13. *Основные стековые операции: объединение операции извлечения из стека с операцией возврата.*

3.6.2. Реализация стека в процессоре Cortex-M3

В процессоре Cortex-M3 используется модель «полного» убывающего стека. Указатель стека указывает на последнее значение, помещённое в стек, и инкрементируется перед выполнением новой операции загрузки в стек. В качестве примера на **Рис. 3.14** показано выполнение команды `PUSH {R0}`.

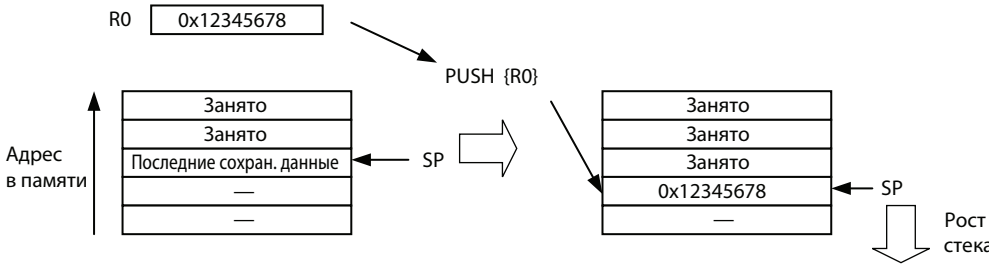


Рис. 3.14. *Реализация операции загрузки в стек.*

Во время операции извлечения из стека данные считываются из памяти по адресу, определяемому указателем стека, после чего указатель инкрементируется. Содержимое ячейки памяти не изменяется, однако оно будет перезаписано при выполнении последующей операции загрузки в стек (**Рис. 3.15**).

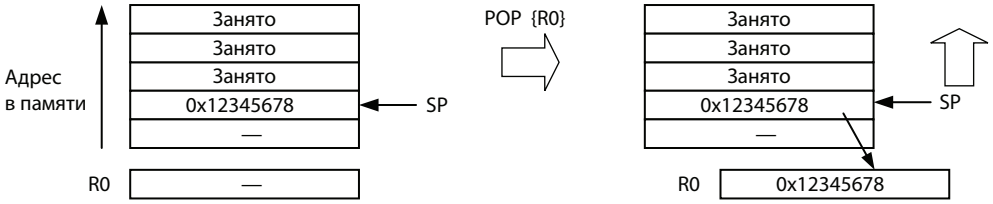


Рис. 3.15. *Реализация операции извлечения из стека.*

Поскольку при выполнении каждой операции загрузки в стек и извлечения из стека осуществляется пересылка 4 байт данных (каждый регистр содержит одно слово, или 4 байт), значение указателя стека уменьшается/увеличивается на 4 или, при одновременном сохранении/восстановлении нескольких регистров, на величину, кратную четырём.

В процессоре Cortex-M3 в качестве указателя стека используется регистр R13. При возникновении прерывания некоторые регистры будут автоматически сохранены в стеке. При возврате из обработчика прерывания эти регистры также автоматически будут восстановлены из стека, а значение указателя стека изменится соответствующим образом.

3.6.3. Два стека процессора Cortex-M3

Как уже было сказано, в процессоре Cortex-M3 имеется два указателя стека: основной MSP и дополнительный PSP. Используемый в настоящий момент указатель определяется битом 1 регистра управления (далее этот бит обозначается как CONTROL[1]).

Если бит CONTROL[1] сброшен в 0, то указатель MSP используется в обоих режимах работы процессора (**Рис. 3.16**). В этом случае и основная программа, и обработчики исключений задействуют под стек одну и ту же область памяти. Такое поведение используется по умолчанию после включения процессора.

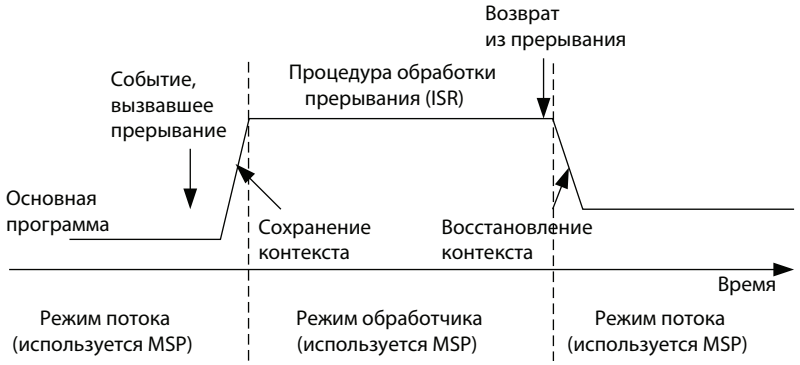


Рис. 3.16. *CONTROL[1] = 0; основной стек используется и в режиме потока, и в режиме обработчика.*

Если бит CONTROL[1] установлен в 1, то в режиме потока используется указатель PSP (**Рис. 3.17**). В этом случае основная программа и обработчики прерываний задействуют под стеки разные области памяти. Это позволяет предотвратить порчу стека, используемого операционной системой, при наличии ошибок в пользовательской программе (предполагается, что пользовательская программа выполняется только в режиме потока, а ядро ОС выполняется в режиме обработчика).

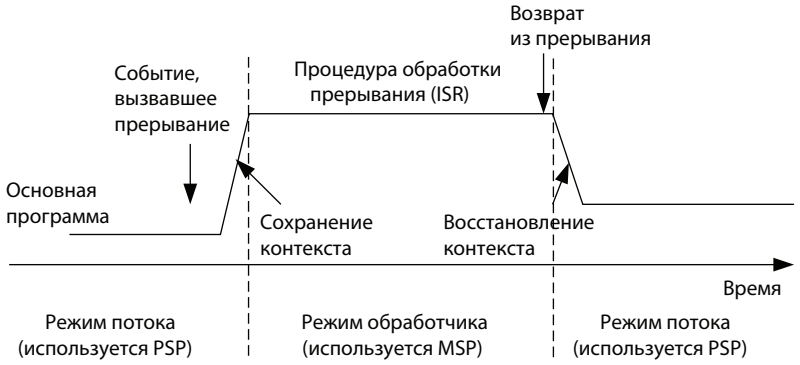


Рис. 3.17. *CONTROL[1] = 1; в режиме потока используется стек процесса, а в режиме обработчика — основной стек.*

Заметьте, что в этом случае при автоматическом сохранении и восстановлении регистров будет использоваться дополнительный стек, тогда как при стековых операциях внутри обработчиков — основной стек.

В процессоре предусмотрена возможность непосредственной записи/чтения указателей MSP и PSP, независимо от того, на какой из них ссылается в данный момент регистр R13. При работе на привилегированном уровне для доступа к содержимому указателей MSP и PSP можно использовать следующие функции:

```
x = __get_MSP(); // Чтение значения MSP
__set_MSP(x);    // Установка значения MSP
x = __get_PSP(); // Чтение значения PSP
```

```
__set_PSP(x);    // Установка значения PSP
```

Вообще говоря, изменять значение текущего указателя стека внутри Си-функций крайне не рекомендуется, поскольку в стеке могут храниться локальные переменные. Для обращения к указателям стека из ассемблерного кода можно использовать уже известные нам команды MRS и MSR:

```
MRS  R0, MSP ; Чтение основного указателя стека в R0
MSR  MSP, R0 ; Запись R0 в основной указатель стека
MRS  R0, PSP ; Чтение указателя стека процесса в R0
MSR  PSP, R0 ; Запись R0 в указатель стека процесса
```

Считав значение PSP посредством команды MRS, ОС может прочитать данные, помещённые в стек пользовательской программой (скажем, содержимое регистров перед вызовом супервизора командой SVC). Более того, ОС может изменить значение указателя PSP, например при переключении контекста в многозадачной системе.

3.7. Цикл сброса

После выхода процессора из состояния сброса он считывает из памяти два 32-битных значения (**Рис. 3.18**):

- по адресу 0x00000000 — начальное значение R13 (указателя стека);
- по адресу 0x00000004 — вектор сброса (адрес, с которого начинается выполнение программы; младший бит значения должен быть установлен в 1 для указания состояния Thumb).

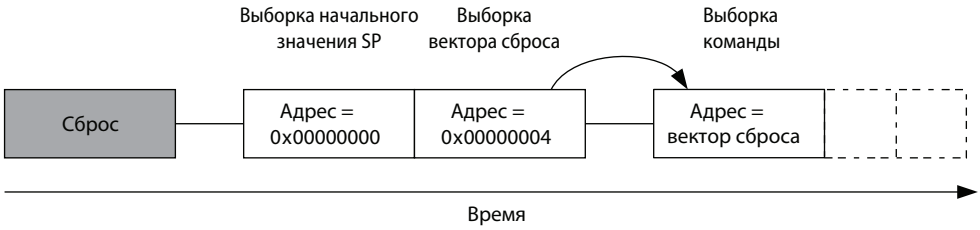


Рис. 3.18. *Цикл сброса.*

Эта последовательность отличается от поведения традиционных процессоров ARM. Предыдущие процессоры компании ARM выполняли программный код, начиная с нулевого адреса. Более того, таблица векторов предыдущих процессоров ARM содержала команды (вы должны были поместить в таблицу команду перехода на начало обработчика исключения, расположенного в каком-нибудь другом месте).

В процессоре же Cortex-M3 в самом начале карты памяти размещается начальное значение MSP, после него располагается таблица векторов (позже, во время выполнения программы, таблица векторов может быть перемещена в другое место). К тому же в таблице векторов содержатся не команды перехода, а значения адресов. Первым вектором в таблице является вектор сброса, который представляет собой второе значение, считываемое процессором при выходе из состояния сброса.

Поскольку в процессоре Cortex-M3 реализована модель «полного» убывающего стека (указатель стека декрементируется перед загрузкой значения в стек), на-

чальное значение указателя должно указывать на первую ячейку памяти, расположенную выше области стека. Например, если вы отвели под стек область памяти с адресами 0x20007C00…0x20007FFF (1 Кбайт), то начальное значение указателя стека будет равно 0x20008000.

Таблица векторов располагается сразу же после начального значения указателя стека. Первым вектором в таблице является вектор сброса. Не забывайте, что младшие биты адресов в таблице векторов процессора Cortex-M3 должны быть установлены в 1, указывая на использование кода Thumb. По этой причине значение вектора сброса в предыдущем примере равно 0x101, тогда как код загрузчика располагается, начиная с адреса 0x100 (**Рис. 3.19**). После выборки вектора сброса процессор может приступить к выполнению программы, начиная с адреса, расположенного по указанному вектору. Столь ранняя инициализация указателя стека необходима, потому что некоторые исключения (такие как NMI) могут возникнуть сразу же после сброса процессора, а обработчикам этих исключений может потребоваться стек.

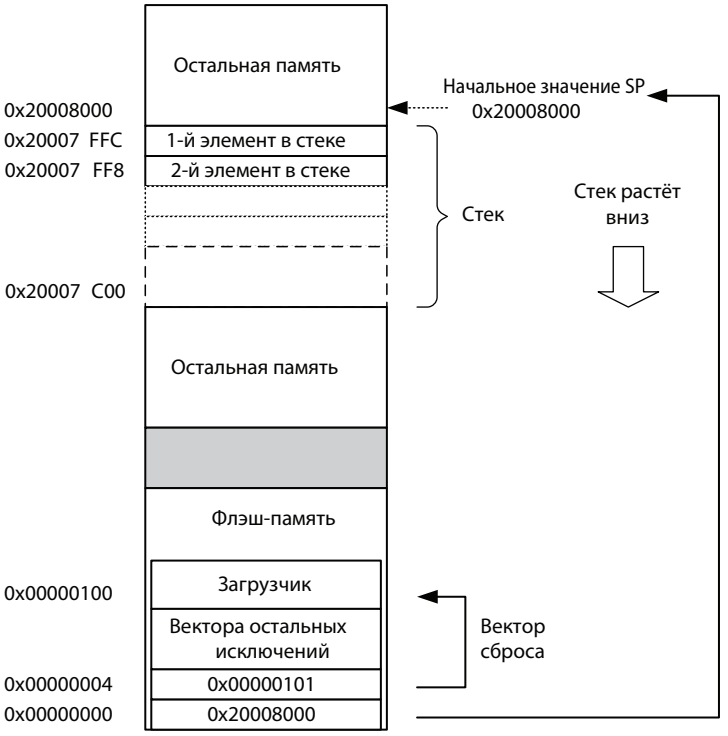


Рис. 3.19. *Начальные значения указателя стека и счётчика команд.*

В различных средствах разработки программного обеспечения могут использоваться разные способы задания начального значения указателя стека и значения вектора сброса. Если вам необходима более подробная информация по этому вопросу, то лучше всего обратиться к примерам, входящим в состав используемого пакета. В Главах 10 и 20 данной книги приведены простые примеры для средств разработки компании ARM, а в Главе 19 — для инструментария GNU.

ГЛАВА 4
НАБОР КОМАНД

4.1. Основы языка ассемблера

В данном разделе вы познакомитесь с базовыми синтаксическими конструкциями языка ассемблера ARM, что поможет вам понять фрагменты кода, содержащиеся в последующих главах книги. Бóльшая часть ассемблерного кода из данной книги рассчитана на использование средств разработки компании ARM. Исключение составляет ряд примеров из Главы 19, демонстрирующих применение инструментария GNU.

4.1.1. Язык ассемблера: основы синтаксиса

В ассемблерных программах при записи команд обычно используется следующее форматирование:

```
<метка>
    <код_операции> операнд1, операнд2, ...; Комментарий
```

Элемент <метка> является необязательным. Метки могут размещаться перед некоторыми командами для того, чтобы с их помощью можно было определять адреса этих команд. После необязательной метки располагается <код_операции> (команда), сопровождаемый некоторым количеством операндов. Как правило, первый операнд является приёмником результата операции. Число операндов зависит от типа команды, формат записи операндов также может быть различным. Скажем, константы обычно записываются в виде #число, как показано в следующем примере:

```
MOV R0, #0x12 ; Загрузить в R0 число 0x12 (шестнадцатеричная константа)
MOV R1, #'A'  ; Загрузить в R1 символ ASCII 'A'
```

Текст после символа «;» является комментарием. Эти комментарии не влияют на выполнение программы, однако позволяют сделать текст программы более понятным для человека.

Вы можете определить константу с помощью директивы ассемблера EQU, а затем использовать её в программе. Например:

```
NVIC_IRQ_SETEN0  EQU 0xE000E100
NVIC_IRQ0_ENABLE EQU 0x1
    ...
```

```
LDR R0, =NVIC_IRQ_SETEN0  ; LDR является псевдокомандой,
                          ; которая преобразуется ассемблером
                          ; в команду загрузки, использующую адресацию
                          ; относительно PC
MOV R1, #NVIC_IRQ0_ENABLE ; Пересылка константы в регистр
STR R1,[R0]               ; Разрешаем IRQ0, записывая R1 по адресу,
                          ; содержащемуся в R0
```

Имеется также несколько директив описания данных, предназначенных для вставки констант непосредственно в ассемблерный код. Так, директива `DCI` (Define Constant Instruction) может применяться для вставки команды в том случае, если используемый вами ассемблер не может сгенерировать нужную команду и если вы знаете её двоичный код.

```
DCI 0xBE00 ; Точка останова (BKPT 0), 16-битная команда
```

Для определения в коде программы однобайтных констант, например символов, можно использовать директиву `DCB` (Define Constant Byte), а для определения 32-битных констант — директиву `DCD` (Define Constant Data):

```
    LDR R3,=MY_NUMBER ; Получаем адрес константы MY_NUMBER
    LDR R4,[R3]       ; Считываем значение 0x12345678 в R4
    ...
    LDR R0,=HELLO_TXT ; Получаем начальный адрес строки HELLO_TXT
    BL  PrintText     ; Вызываем функцию PrintText
                      ; для отображения строки
    ...
MY_NUMBER
    DCD 0x12345678
HELLO_TXT
    DCB «Hello\n»,0   ; Строка, завершающаяся нулевым символом
```

Следует заметить, что синтаксис языка ассемблера зависит от используемого инструментария. В данном случае описывается синтаксис ассемблера ARM. Изучение синтаксиса других ассемблеров лучше всего начинать с рассмотрения примеров программ, поставляемых вместе со средствами разработки.

4.1.2. Язык ассемблера: использование суффиксов

В ассемблере для процессоров ARM при записи мнемоник команд могут использоваться дополнительные суффиксы, указанные в **Табл. 4.1**.

Таблица 4.1. **Суффиксы мнемоник команд**

Суффикс	Описание
S	Изменить регистр состояния приложения APSR (флаги операций), например: `ADDS  R0, R1  ; Эта команда повлияет на содержимое APSR`
EQ, NE, LT, GT и т.п.	Условное выполнение; EQ = «Равно», NE = «Не равно», LT = «Меньше», GT = «Больше» и т.д. Например: `BEQ  <Label>  ; Перейти, если «равно»`

Суффиксы условного выполнения в процессоре Cortex-M3 обычно используются в командах переходов. Тем не менее, эти суффиксы могут применяться и в других командах, если те располагаются внутри блока `IF-THEN` (мы познако-

мимся с данной конструкцией чуть позже). При этом суффикс S и суффиксы условного выполнения могут использоваться одновременно. В общей сложности имеется 15 условий выполнения команд, которые будут описаны далее.

4.1.3. Язык ассемблера: унифицированный язык ассемблера

Для поддержки набора команд Thumb®-2 и использования всех предоставляемых этим набором преимуществ, компания ARM разработала *унифицированный язык ассемблера* (Unified Assembler Language — UAL). Применение данного языка облегчает переход между кодами ARM и Thumb за счёт использования одинакового синтаксиса для обоих наборов команд.

```
ADD R0, R1     ; R0 = R0 + R1, используется традиционный синтаксис Thumb
ADD R0, R0, R1 ; Эквивалентная команда, использующая синтаксис UAL
```

При этом, как и прежде, можно использовать традиционный синтаксис Thumb. Для указания того, каким образом ассемблер должен интерпретировать текст программы, обычно применяются специальные директивы. Так, для ассемблера ARM наличие директивы CODE16 в заголовке программы означает использование в программе традиционного синтаксиса Thumb, тогда как директива THUMB указывает на использование нового синтаксиса UAL.

При применении традиционных команд Thumb не забывайте, что некоторые из них изменяют флаги регистра APSR независимо от наличия суффикса S. В синтаксисе UAL влияние команды на флаги состояния всегда определяется этим суффиксом. Например:

```
AND  R0, R1     ; Традиционный синтаксис Thumb
ANDS R0, R0, R1 ; Эквивалентный синтаксис UAL (указан суффикс S)
```

В связи с появлением в наборе Thumb-2 новых команд некоторые операции теперь могут быть выполнены как при помощи команд Thumb, так и при помощи команд Thumb-2. К примеру, операция инкрементирования R0 = R0 + 1 может быть реализована посредством 16-битной команды Thumb или же 32-битной команды Thumb-2. Для указания разрядности команды в UAL предусмотрены специальные суффиксы:

```
ADDS   R0, #1 ; Используем 16-битную команду Thumb по умолчанию
              ; для получения более компактного кода
ADDS.N R0, #1 ; Используем 16-битную команду Thumb; N = Narrow (узкая)
ADDS.W R0, #1 ; Используем 32-битную команду Thumb-2; W = Wide (широкая)
```

Суффикс .W указывает на использование 32-битной команды. При отсутствии суффикса ассемблер может вставить команду любой разрядности; как правило, по умолчанию применяются 16-битные команды Thumb для уменьшения размера кода. Для явного указания 16-битных команд Thumb вы также можете использовать суффикс .N, при условии, что этот суффикс поддерживается ассемблером.

Ещё раз подчеркну, что всё сказанное выше относится к ассемблеру ARM. Прочие ассемблеры могут использовать другой синтаксис. Если суффикс не указан, то ассемблер может сам выбрать необходимую команду.

В большинстве случаев приложения пишутся на языке Си, а компиляторы с этого языка везде, где только можно, задействуют 16-битные команды для полу-

чения наиболее компактного кода. Команды Thumb-2 вставляются компилятором при использовании непосредственных значений, выходящих за границы определённого диапазона, а также в тех случаях, когда 32-битные команды позволяют выполнить требуемую операцию наиболее эффективным образом.

Команды Thumb-2 могут быть выровнены на границу полуслова, например:

```
0x1000 : LDR    r0, [r1] ; 16-битная команда (располагается по адресам
                         ; 0x1000-0x1001)
0x1002 : RBIT.W r0       ; 32-битная команда Thumb-2 (располагается
                         ; по адресам 0x1002...0x1005)
```

Большинство 16-битных команд могут обращаться только к регистрам R0…R7, тогда как 32-битные команды не имеют подобного ограничения. Тем не менее, в некоторых командах не допускается использование счётчика команд (R15). Более подробно этот вопрос рассмотрен в [2] (раздел A.4.6).

4.2. Список команд

Команды, поддерживаемые процессором Cortex-M3, перечислены в **Табл. 4.2…4.9**. Подробное описание всех команд процессора можно найти в [2]. Кроме того, основная информация о поддерживаемых командах содержится в Приложении А данной книги.

Таблица 4.2. **Команды обработки данных (16-битные)**

Мнемоника	Описание
ADC	Сложение с переносом
ADD	Сложение
ADR	Сложение PC и константы с загрузкой результата в регистр
AND	Логическое И
ASR	Арифметический сдвиг вправо
BIC	Очистка битов (операция «Логическое И» между одним значением и результатом логической инверсией другого значения)
CMN	Сравнение с отрицательным операндом (сравнение одного значения с дополнительным кодом другого значения и изменение флагов)
CMP	Сравнение (сравнение двух значений и изменение флагов)
CPY	Копирование (появилась в архитектуре v6; пересылает содержимое одного старшего или младшего регистра в другой старший или младший регистр); синоним команды MOV
EOR	Исключающее ИЛИ
LSL	Логический сдвиг влево
LSR	Логический сдвиг вправо
MOV	Пересылка (может использоваться для пересылки данных между регистрами или для загрузки констант)
MUL	Умножение
MVN	Пересылка с инверсией (пересылает результат логической инверсии значения)
NEG	Отрицание (вычисляет дополнительный код значения), эквивалент команды RSB
ORR	Логическое ИЛИ
RSB	Обратное вычитание

***Таблица 4.2.* Команды обработки данных (16-битные)** *(продолжение)*

Мнемоника	Описание
ROR	Циклический сдвиг вправо
SBC	Вычитание с переносом
SUB	Вычитание
TST	Проверка битов (выполняет операцию «Логическое И»; влияет на флаг Z, результат операции не сохраняется)
REV	Перестановка байтов в 32-битном регистре (появилась в архитектуре v6)
REV16	Перестановка байтов в каждом полуслове 32-битного регистра (появилась в архитектуре v6)
REVSH	Перестановка байтов в младшем полуслове 32-битного регистра и расширение знака результата (появилась в архитектуре v6)
SXTB	Расширение знака байта (появилась в архитектуре v6)
SXTH	Расширение знака полуслова (появилась в архитектуре v6)
UXTB	Дополнение нулями байта (появилась в архитектуре v6)
UXTH	Дополнение нулями полуслова (появилась в архитектуре v6)

***Таблица 4.3.* Команды для ветвления (16-битные)**

Мнемоника	Описание
B	Переход
B<cond>	Условный переход
BL	Переход со ссылкой; вызывает подпрограмму и сохраняет адрес возврата в LR (в действительности это 32-битная команда, но она также имеется в наборе Thumb традиционных процессоров ARM)
BLX	Косвенный переход со ссылкой и изменением состояния (только BLX <reg>)*
BX <reg>	Косвенный переход с изменением состояния
CBZ	Сравнение и переход, если ноль (архитектура v7)
CBNZ	Сравнение и переход, если не ноль (архитектура v7)
IT	Блок IF-THEN (архитектура v7)

* Команда BLX с использованием непосредственного значения не поддерживается, поскольку при её выполнении происходит переключение в состояние ARM, не поддерживаемое процессором Cortex-M3. Попытка использования команды BLX <reg> для переключения процессора в состояние ARM вызовет генерацию исключения Usage Fault.

***Таблица 4.4.* Команды загрузки и сохранения (16-битные)**

Мнемоника	Описание
LDR	Загрузка слова из памяти в регистр
LDRH	Загрузка полуслова из памяти в регистр
LDRB	Загрузка байта из памяти в регистр
LDRSH	Загрузка полуслова из памяти в регистр с расширением знака
LDRSB	Загрузка байта из памяти в регистр с расширением знака
STR	Сохранение слова из регистра в памяти
STRH	Сохранение полуслова из регистра в памяти
STRB	Сохранение байта из регистра в памяти

***Таблица 4.4.* Команды загрузки и сохранения (16-битные)** *(продолжение)*

Мнемоника	Описание
LDM/LDMIA	Загрузка нескольких регистров/Загрузка нескольких регистров с постинкрементом
STM/STMIA	Сохранение нескольких регистров/Сохранение нескольких регистров с постинкрементом
PUSH	Загрузка регистров в стек
POP	Извлечение регистров из стека

***Таблица 4.5.* Прочие 16-битные команды**

Мнемоника	Описание
SVC	Вызов супервизора
SEV	Генерация события
WFE	Ожидание события при нахождении в спящем режиме
WFI	Ожидание прерывания при нахождении в спящем режиме
BKPT	Точка останова; если отладка разрешена, то эта команда переводит процессор в режим отладки (останова) или же, если разрешено исключение монитора отладки, вызывает исключение отладки. В противном случае, генерируется исключение отказа
NOP	Нет операции
CPSIE	Разрешение прерываний, очищает регистр PRIMASK (CPSIE i)/FAULTMASK (CPSIE f)
CPSID	Запрещение прерываний, устанавливает регистр PRIMASK (CPSIE i)/FAULTMASK (CPSIE f) в 1

***Таблица 4.6.* Команды обработки данных (32-битные)**

Мнемоника	Описание
ADC	Сложение с переносом
ADD	Сложение
ADDW	Сложение с 12-битной константой
ADR	Сложение PC и константы с загрузкой результата в регистр
AND	Логическое И
ASR	Арифметический сдвиг вправо
BIC	Очистка битов (операция «Логическое И» между одним значением и результатом логической инверсии другого)
BFC	Очистка битового поля
BFI	Вставка битового поля
CMN	Сравнение с отрицательным операндом (сравнение одного значения с дополнительным кодом другого и изменение флагов)
CMP	Сравнение (сравнение двух значений и изменение флагов)
CLZ	Подсчёт ведущих нулевых битов
EOR	Исключающее ИЛИ
LSL	Логический сдвиг влево

***Таблица 4.6.* Команды обработки данных (32-битные)** *(продолжение)*

Мнемоника	Описание
LSR	Логический сдвиг вправо
MLA	Умножение со сложением
MLS	Умножение с вычитанием
MOV	Пересылка
MOVW	Пересылка 16-битной константы в регистр
MOVT	Пересылка старшего полуслова (загрузка константы в старшее полуслово регистра)
MVN	Пересылка с инверсией
MUL	Умножение
ORR	Логическое ИЛИ
ORN	Логическое ИЛИ с инверсией
RBIT	Перестановка битов
REV	Перестановка байтов слова
REV16	Перестановка байтов в каждом полуслове
REVSH	Перестановка байтов в младшем полуслове и расширение знака
ROR	Циклический сдвиг вправо
RSB	Обратное вычитание
RRX	Расширенный циклический сдвиг вправо
SBFX	Извлечение битового поля со знаком
SDIV	Знаковое деление
SMLAL	Длинное знаковое умножение со сложением
SMULL	Длинное знаковое умножение
SSAT	Знаковое насыщение
SBC	Вычитание с переносом
SUB	Вычитание
SUBW	Вычитание 12-битной константы
SXTB	Расширение знака байта
SXTH	Расширение знака полуслова
TEQ	Проверка на равенство (выполняет операцию «Исключающее ИЛИ»; влияет на флаги, результат операции не сохраняется)
TST	Проверка битов (выполняет операцию «Логическое И»; влияет на флаг Z, результат операции не сохраняется)
UBFX	Извлечение беззнакового битового поля
UDIV	Беззнаковое деление
UMLAL	Длинное беззнаковое умножение со сложением
UMULL	Длинное беззнаковое умножение
USAT	Беззнаковое насыщение
UXTB	Дополнение нулями байта
UXTH	Дополнение нулями полуслова

Таблица 4.7. **Команды загрузки и сохранения (32-битные)**

Мнемоника	Описание
LDR	Загрузка слова из памяти в регистр
LDRT	Загрузка слова из памяти в регистр (непривилегированный уровень доступа)
LDRB	Загрузка байта из памяти в регистр
LDRBT	Загрузка байта из памяти в регистр (непривилегированный уровень доступа)
LDRH	Загрузка полуслова из памяти в регистр
LDRHT	Загрузка полуслова из памяти в регистр (непривилегированный уровень доступа)
LDRSB	Загрузка байта из памяти в регистр с расширением знака
LDRSBT	Загрузка байта из памяти в регистр с расширением знака (непривилегированный уровень доступа)
LDRSH	Загрузка полуслова из памяти в регистр с расширением знака
LDRSHT	Загрузка полуслова из памяти в регистр с расширением знака (непривилегированный уровень доступа)
LDM/LDMIA	Загрузка нескольких регистров с постинкрементом
LDMDB	Загрузка нескольких регистров с преддекрементом
LDRD	Загрузка двух слов из памяти в регистры
STR	Сохранение слова из регистра в памяти
STRT	Сохранение слова из регистра в памяти (непривилегированный уровень доступа)
STRB	Сохранение байта из регистра в памяти
STRBT	Сохранение байта из регистра в памяти (непривилегированный уровень доступа)
STRH	Сохранение полуслова из регистра в памяти
STRHT	Сохранение полуслова из регистра в памяти (непривилегированный уровень доступа)
STM/STMIA	Сохранение нескольких регистров с постинкрементом
STMDB	Сохранение нескольких регистров с преддекрементом
STRD	Сохранение двух слов из регистров в памяти
PUSH	Загрузка регистров в стек
POP	Извлечение регистров из стека

Таблица 4.8. Команды ветвления (32-битные)

Мнемоника	Описание
B	Переход
B<cond>	Условный переход
BL	Переход со ссылкой
TBB	Табличный переход с однобайтными смещениями; переход в прямом направлении с использованием таблицы, содержащей однобайтные смещения
TBH	Табличный переход с 2-байтными смещениями; переход в прямом направлении с использованием таблицы, содержащей 2-байтные смещения

Таблица 4.9. **Прочие 32-битные команды**

Мнемоника	Описание
LDREX	Монопольная загрузка в регистр
LDREXH	Монопольная загрузка полуслова в регистр
LDREXB	Монопольная загрузка байта в регистр
STREX	Монопольное сохранение регистра
STREXH	Монопольное сохранение полуслова из регистра
STREXB	Монопольное сохранение байта из регистра
CLREX	Блокирование монопольного доступа для локального процессора
MRS	Пересылка из РСН в РОН
MSR	Пересылка из РОН в РСН
NOP	Нет операции
SEV	Генерация события
WFE	Перевод в спящий режим и ожидание события
WFI	Перевод в спящий режим и ожидание прерывания
ISB	Барьер синхронизации команд
DSB	Барьер синхронизации данных
DMB	Барьер памяти данных

4.2.1. Неподдерживаемые команды

Некоторые команды Thumb не поддерживаются процессором Cortex-M3; эти команды перечислены в **Табл. 4.10**.

Таблица 4.10. **Неподдерживаемые команды Thumb**

Мнемоника	Описание
BLX label	Это команда перехода со ссылкой и изменением состояния. При использовании в качестве операнда непосредственного значения эта команда всегда переводит процессор в состояние ARM. Поскольку процессор Cortex-M3 не поддерживает состояние ARM, все команды, пытающиеся переключить его в данное состояние, вызывают генерацию исключения Usage Fault
SETEND	Эта команда Thumb, появившаяся в архитектуре v6, изменяет порядок байтов многобайтных значений во время выполнения программы. Поскольку процессор Cortex-M3 не поддерживает динамическое изменение порядка байтов, попытка выполнения команды SETEND вызовет генерацию исключения Usage Fault

Ряд команд, описанных в [2], также не поддерживаются процессором Cortex-M3. Так, в архитектуре ARM v7-M предусмотрены команды сопроцессора, однако в процессоре Cortex-M3 поддержка сопроцессора полностью отсутствует. Соответственно, попытка исполнения любой из команд сопроцессора, перечисленных в **Табл. 4.11**, вызовет генерацию исключения Usage Fault с одновременной установкой бита NOCP в регистре UFSR контроллера NVIC.

Таблица 4.11. **Неподдерживаемые команды сопроцессора**

Мнемоника	Описание
MCR	Пересылка данных из процессора ARM в сопроцессор
MCR2	Пересылка данных из процессора ARM в сопроцессор
MCRR	Пересылка в сопроцессор двух регистров ARM
MRC	Пересылка данных из сопроцессора в регистр ARM
MRC2	Пересылка данных из сопроцессора в регистр ARM
MRRC	Пересылка данных из сопроцессора в два регистра ARM
LDC	Загрузка сопроцессора; загрузка последовательно расположенных данных из памяти в сопроцессор
STC	Сохранение сопроцессора; сохранение данных из сопроцессора в памяти по последовательно расположенным адресам

Некоторые из команд изменения состояния процесса (CPS) также не поддерживаются процессором Cortex-M3 (см. **Табл. 4.12**). Это связано с изменением концепции регистра состояния программы (PSR) и, соответственно, с отсутствием в процессоре Cortex-M3 некоторых битов, определённых в архитектуре ARM v6.

Таблица 4.12. **Неподдерживаемые команды изменения состояния процесса**

Мнемоника	Описание
CPS<IEIID>.W A	В процессоре Cortex-M3 бит A отсутствует
CPS.W #mode	В регистре PSR процессора Cortx-M3 биты режима отсутствуют

Кроме того, команды-подсказки (hint-команды), перечисленные в **Табл. 4.13**, исполняются процессором Cortex-M3 как команда NOP.

Таблица 4.13. **Неподдерживаемые Hint-команды**

Мнемоника	Описание
DBG	Hint-команда системы отладки и трассировки
PLD	Предварительная загрузка данных; hint-команда для кэш-памяти. Поскольку в процессоре Cortex-M3 кэш-память отсутствует, эта команда эквивалентна команде NOP
PLI	Предварительная загрузка команды; hint-команда для кэш-памяти. Поскольку в процессоре Cortex-M3 кэш-память отсутствует, эта команда эквивалентна команде NOP
YIELD	Hint-команда, позволяющая многопоточному приложению информировать процессор о завершении задачи, которая может быть выгружена для улучшения общей производительности системы

При попытке исполнения всех остальных неопределённых команд генерируется исключение Usage Fault.

4.3. Описание команд

В этом разделе мы познакомимся с основными командами, которые наиболее широко применяются в ассемблерных программах. Некоторые команды могут иметь различные опции, в частности использовать предварительный сдвиг опе-

рандов; подобные варианты использования команд в этом разделе практически не рассматриваются.

4.3.1. Язык ассемблера: пересылка данных

Одной из основных функций процессора является пересылка данных. Процессор Cortex-M3 поддерживает следующие типы пересылок:

- пересылка данных между регистрами общего назначения;
- пересылка данных между памятью и регистром общего назначения;
- пересылка данных между регистром специального назначения и регистром общего назначения;
- пересылка непосредственного значения в регистр общего назначения.

Для пересылки данных между регистрами предназначена команда `MOV`. Скажем, пересылка содержимого регистра R3 в регистр R8 записывается следующим образом:

```
MOV R8, R3
```

Имеется ещё одна команда пересылки, которая пересылает инверсное значение исходного содержимого регистра, — это команда `MVN`.

Основными командами, посредством которых выполняются обращения к памяти, являются команды загрузки и сохранения. Команда загрузки (`LDR`) пересылает данные из памяти в регистры процессора, а команда сохранения (`STR`) пересылает содержимое регистров в память. При этом поддерживается пересылка данных любого размера (байт, полуслово, слово и двойное слово), как указано в **Табл. 4.14**.

Таблица 4.14. **Наиболее часто используемые команды обращения к памяти**

Пример	Описание
`LDRB Rd, [Rn, #offset]`	Чтение байта из памяти по адресу Rn + offset
`LDRH Rd, [Rn, #offset]`	Чтение полуслова из памяти по адресу Rn + offset
`LDR  Rd, [Rn, #offset]`	Чтение слова из памяти по адресу Rn + offset
`LDRD Rd1, Rd2, [Rn, #offset]`	Чтение двойного слова из памяти по адресу Rn + offset
`STRB Rd, [Rn, #offset]`	Сохранение байта в памяти по адресу Rn + offset
`STRH Rd, [Rn, #offset]`	Сохранение полуслова в памяти по адресу Rn + offset
`STR  Rd, [Rn, #offset]`	Сохранение слова в памяти по адресу Rn + offset
`STRD Rd1, Rd2, [Rn, #offset]`	Сохранение двойного слова в памяти по адресу Rn + offset

Команды групповой загрузки (`LDM`) и сохранения (`STM`) дают возможность за один раз выполнить несколько одноимённых операций, как указано в **Табл. 4.15**. При этом символ «!» после первого операнда указывает на необходимость обновления регистра *Rd* после выполнения команды. Пусть в R8 содержится значение 0x8000, тогда:

```
STMIA.W R8!, {R0-R3} ; После сохранения данных R8 станет равным 0x8010
                     ; (инкрементируется на 4 для каждой операции пересылки)
STMIA.W R8 , {R0-R3} ; R8 не изменится
```

Таблица 4.15. **Команды групповой загрузки/сохранения**

Пример	Описание
`LDMIA Rd!, <reg list>`	Чтение нескольких слов из памяти, начиная с адреса, находящегося в *Rd*; адрес инкрементируется после (Increment After — IA) каждой пересылки (16-битная команда Thumb)
`STMIA Rd!, <reg list>`	Сохранение нескольких слов в памяти, начиная с адреса, находящегося в *Rd*; адрес инкрементируется после (Increment After — IA) каждой пересылки (16-битная команда Thumb)
`LDMIA.W Rd(!), <reg list>`	Чтение нескольких слов из памяти, начиная с адреса, находящегося в *Rd*; адрес инкрементируется после каждой операции чтения (суффикс `.W` говорит о том, что это 32-битная команда Thumb-2)
`LDMDB.W Rd(!), <reg list>`	Чтение нескольких слов из памяти, начиная с адреса, находящегося в *Rd*; адрес декрементируется перед (Decrement Before — DB) каждой операцией чтения (суффикс `.W` говорит о том, что это 32-битная команда Thumb-2)
`STMIA.W Rd(!), <reg list>`	Сохранение нескольких слов в памяти, начиная с адреса, находящегося в *Rd*; адрес инкрементируется после каждой операции записи (суффикс `.W` говорит о том, что это 32-битная команда Thumb-2)
`STMDB.W Rd(!), <reg list>`	Сохранение нескольких слов в памяти, начиная с адреса, находящегося в *Rd*; адрес декрементируется перед каждой операцией записи (суффикс `.W` говорит о том, что это 32-битная команда Thumb-2)

Также процессоры ARM поддерживают обращение к памяти с использованием пред- и постиндексации. В случае прединдексации содержимое регистра, в котором хранится адрес ячейки памяти, перед выполнением операции изменяется. То есть пересылка данных будет осуществлена уже по новому адресу. Например:

```
LDR.W R0,[R1, #offset]! ; Читаем из памяти по адресу [R1+offset],
                        ; после операции R1 становится равным R1+offset
```

Наличие символа «!» указывает на изменение базового регистра R1. Этот символ является необязательным, при его отсутствии команда выполнит обычную операцию пересылки из ячейки памяти, определяемой смещением относительно базового адреса. В группе команд обращения к памяти с прединдексацией имеются команды загрузки и сохранения данных различной разрядности (**Табл. 4.16**).

Таблица 4.16. **Примеры команд обращения к памяти с прединдексацией**

Пример	Описание
`LDR.W Rd, [Rn, #offset]!`	Команды загрузки с прединдексацией (слово, байт, полуслово, двойное слово)
`LDRB.W Rd, [Rn, #offset]!`	
`LDRH.W Rd, [Rn, #offset]!`	
`LDRD.W Rd1, Rd2, [Rn, #offset]!`	
`LDRSB.W Rd, [Rn, #offset]!`	Команды загрузки с прединдексацией и расширением знака (байт, полуслово)
`LDRSH.W Rd, [Rn, #offset]!`	
`STR.W Rd, [Rn, #offset]!`	Команды сохранения с прединдексацией (слово, байт, полуслово, двойное слово)
`STRB.W Rd, [Rn, #offset]!`	
`STRH.W Rd, [Rn, #offset]!`	
`STRD.W Rd1, Rd2, [Rn, #offset]!`	

Команды обращения к памяти с постиндексацией осуществляют пересылку данных с использованием базового адреса, определяемого регистром, после чего изменяют содержимое этого регистра. Например,

```
LDR.W R0, [R1], #offset ; Читаем из памяти по адресу [R1], после операции
                        ; R1 становится равным R1+offset
```

При применении команд с постиндексацией нет необходимости в использовании знака «!», поскольку команды данного типа в любом случае изменяют регистр, содержащий базовый адрес, тогда как в случае команд с прединдексацией вы сами решаете, изменять этот регистр или нет.

Как и в группе команд с прединдексацией, в группе команд с постиндексацией имеются команды пересылки данных различной разрядности (**Табл. 4.17**).

Таблица 4.17. **Примеры команд обращения к памяти с постиндексацией**

Пример	Описание
`LDR.W Rd, [Rn], #offset`	Команды загрузки с постиндексацией (слово, байт, полуслово, двойное слово)
`LDRB.W Rd, [Rn], #offset`	
`LDRH.W Rd, [Rn], #offset`	
`LDRD.W Rd1, Rd2, [Rn], #offset`	
`LDRSB.W Rd, [Rn], #offset`	Команды загрузки с постиндексацией и расширением знака (байт, полуслово)
`LDRSH.W Rd, [Rn], #offset`	
`STR.W Rd, [Rn], #offset`	Команды сохранения с постиндексацией (слово, байт, полуслово, двойное слово)
`STRB.W Rd, [Rn], #offset`	
`STRH.W Rd, [Rn], #offset`	
`STRD.W Rd1, Rd2, [Rn], #offset`	

К классу операций с памятью относятся также операция загрузки данных в стек и операция извлечения данных из стека. Для выполнения этих операций предназначены команды `PUSH` и `POP` соответственно. Например,

```
PUSH {R0, R4-R7, R9} ; Загружаем R0, R4, R5, R6, R7, R9 в стек
POP  {R2,R3}         ; Извлекаем R2 и R3 из стека
```

Как правило, команде `PUSH` соответствует команда `POP` с тем же списком аргументов, однако это не является обязательным. В качестве примера подобного исключения из правил можно указать использование команды `POP` для возврата из подпрограммы:

```
PUSH {R0-R3, LR} ; Сохраняем содержимое регистров в начале подпрограммы
....             ; Выполняем требуемые действия
POP  {R0-R3, PC} ; Восстанавливаем содержимое регистров и возвращаемся
```

В этом случае вместо восстановления содержимого регистра LR и последующего перехода по данному адресу мы загружаем адрес возврата непосредственно в счётчик команд.

Как было сказано в Главе 3, процессор Cortex-M3 имеет несколько регистров специального назначения. Для обращения к этим регистрам используются команды `MRS` и `MSR`. Например,

```
MRS R0, PSR     ; Считать состояние процессора в R0
MSR CONTROL, R1 ; Записать содержимое R1 в регистр управления
```

Ещё раз напомню, что обращение ко всем регистрам специального назначения, кроме регистра APSR, допускается только в привилегированном режиме.

Очень часто возникает необходимость загрузки в регистр непосредственного значения. Скажем, для того чтобы обратиться к регистру периферийного устройства, вам необходимо сначала загрузить в регистр процессора требуемый адрес. Для загрузки значений разрядностью 8 бит и менее вы можете использовать команду MOVS. Например,

```
MOVS R0, #0x12 ; Загружаем 0x12 в R0
```

Для загрузки значений большей разрядности (более 8 бит) вам придётся воспользоваться командой пересылки из набора Thumb-2. Например,

```
MOVW.W R0, #0x789A ; Загружаем 0x789A в R0
```

В случае 32-битной константы вы можете использовать две команды для загрузки старшей и младшей частей значения:

```
MOVW.W R0,#0x789A ; Загружаем 0x789A в младшие 16 бит R0
MOVT.W R0,#0x3456 ; Загружаем 0x3456 в старшие 16 бит R0
                  ; Теперь R0 = 0x3456789A
```

В качестве альтернативы вы можете воспользоваться псевдокомандой LDR, имеющейся в ассемблере ARM. Например,

```
LDR R0, =0x3456789A
```

Это не ассемблерная команда; ассемблер преобразует указанную инструкцию в команду загрузки значения с использованием адресации относительно PC. Данный способ генерации 32-битных констант является более предпочтительным, нежели применение пары команд MOVW.W и MOVT.W, поскольку обеспечивает лучшую читаемость, а также позволяет уменьшить расход памяти в случае, если данное значение используется в разных местах программы.

4.3.2. Псевдокоманды LDR и ADR

Обе команды LDR и ADR могут применяться для загрузки в регистры значений адреса памяти. Однако синтаксис и поведение этих команд различаются. При использовании команды LDR, если адрес расположен в области кода программы, ассемблер автоматически установит младший бит загружаемого значения в 1. Например,

```
    LDR R0, =address1 ; Загружаем 0x4001 в R0
    ...
address1              ; Метка соответствует адресу 0x4000
    MOV R0, R1        ; address1 содержит код программы
    ...
```

Вы сможете увидеть, что команда LDR загрузит в R1 значение 0x4001; младший бит устанавливается в 1, указывая на использование кода Thumb. Если address1 является адресом в области данных, то младший бит значения не изменяется. Например,

```
    LDR R0, =address1 ; Загружаем 0x4000 в R0
    ...
address1              ; Метка соответствует адресу 0x4000
```

```
    DCD 0x0          ; address1 содержит данные
    ...
```

Используя команду ADR, вы можете загрузить в регистр значение адреса программы без автоматического изменения младшего бита. Например,

```
    ADR R0, address1
    ...
address1             ; Метка соответствует адресу 0x4000
    MOV R0, R1       ; address1 содержит код программы
    ...
```

При использовании команды ADR в регистр будет загружено значение 0x4000. Обратите внимание на отсутствие в выражении знака равенства.

Команда LDR размещает константу в коде программы и загружает её значение в регистр, используя адресацию относительно PC. Команда ADR пытается сформировать значение константы с помощью команд сложения и вычитания (например, основываясь на текущем значении PC). Как следствие, используя команду ADR, нельзя получить любые значения, т.е. искомый адрес должен находиться на небольшом расстоянии от команды. Тем не менее, команда ADR может генерировать более компактный код по сравнению с командой LDR.

Команда ADR является 16-битной и требует, чтобы конечный адрес был выровнен на границу слова (кратен четырём). Если конечный адрес не выровнен на границу слова, то вы можете использовать 32-битный вариант команды — ADR.W. Если конечный адрес расположен за пределами области в ±4095 байт относительно текущего значения PC, то можно задействовать псевдокоманду ADRL, которая позволяет обращаться к адресам в диапазоне ±1 Мбайт.

4.3.3. Язык ассемблера: обработка данных

Процессор Cortex-M3 поддерживает множество самых разных команд, предназначенных для обработки данных. В настоящем подразделе будут рассмотрены наиболее часто используемые из этих команд. Следует отметить, что большинство команд обработки данных могут иметь несколько форматов записи. Например, команда сложения ADD может выполнять операцию между двумя регистрами или же между регистром и непосредственным значением:

```
    ADD   R0, R0, R1    ; R0 = R0 + R1
    ADDS  R0, R0, #0x12 ; R0 = R0 + 0x12
    ADD.W R0, R1, R2    ; R0 = R1 + R2
```

Все команды из приведённого примера являются командами ADD, однако они имеют разные форматы записи и разное двоичное представление.

При использовании 16-битного кода Thumb и традиционного синтаксиса для записи команд Thumb команда ADD влияет на флаги в регистре состояния процессора. В то же время 32-битный код Thumb-2 может изменять эти флаги, а может и не изменять. Если выполнение последующих операций зависит от состояния флагов, то следует использовать суффикс S, указывающий на необходимость их изменения:

```
    ADD.W  R0, R1, R2 ; Не влияет на состояние флагов
    ADDS.W R0, R1, R2 ; Влияет на состояние флагов
```

Помимо команды ADD, процессор Cortex-M3 поддерживает такие команды арифметических операций, как команда вычитания (SUB), команда умножения (MUL), а также команды беззнакового и знакового деления (UDIV/SDIV). Некоторые наиболее широко используемые команды арифметических операций приведены в **Табл. 4.18**.

***Таблица 4.18.* Примеры команд арифметических операций**

Команда	Операция
`ADD Rd, Rn, Rm ; Rd = Rn + Rm`	Сложение
`ADD Rd, Rd, Rm ; Rd = Rd + Rm`	
`ADD Rd, #immed ; Rd = Rd + #immed`	
`ADD Rd, Rn, # immed ; Rd = Rn + #immed`	
`ADC Rd, Rn, Rm ; Rd = Rn + Rm + перенос`	Сложение с переносом
`ADC Rd, Rd, Rm ; Rd = Rd + Rm + перенос`	
`ADC Rd, #immed ; Rd = Rd + #immed + перенос`	
`ADDW Rd, Rn, #immed ; Rd = Rn + #immed`	Сложение регистра и 12-битной константы
`SUB Rd, Rn, Rm ; Rd = Rn – Rm`	Вычитание
`SUB Rd, #immed ; Rd = Rd – #immed`	
`SUB Rd, Rn, #immed ; Rd = Rn – #immed`	
`SBC Rd, Rm ; Rd = Rd – Rm – заём`	Вычитание с заёмом (инверсия переноса)
`SBC.W Rd, Rn, #immed ; Rd = Rn – #immed – заём`	
`SBC.W Rd, Rn, Rm ; Rd = Rn – Rm – заём`	
`RSB.W Rd, Rn, #immed ; Rd = #immed – Rn`	Обратное вычитание
`RSB.W Rd, Rn, Rm ; Rd = Rm – Rn`	
`MUL Rd, Rm ; Rd = Rd * Rm`	Умножение
`MUL.W Rd, Rn, Rm ; Rd = Rn * Rm`	
`UDIV Rd, Rn, Rm ; Rd = Rn/Rm`	Деление
`SDIV Rd, Rn, Rm ; Rd = Rn/Rm`	

Эти команды могут применяться как с суффиксом S, указывающим на то, что команда воздействует на регистр APSR, так и без данного суффикса. В большинстве случаев при использовании синтаксиса UAL и отсутствии в мнемонике команды суффикса S в код будет вставлен 32-битный вариант команды, поскольку практически все 16-битные команды Thumb влияют на состояние регистра APSR.

Также процессор Cortex-M3 поддерживает команды 32-битного умножения и умножения со сложением, формирующие 64-битный результат. Имеются как знаковые, так и беззнаковые варианты этих команд (**Табл. 4.19**).

***Таблица 4.19.* Команды 32-битного умножения**

Команда	Операция
`SMULL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} = Rn * Rm`	Команды 32-битного умножения чисел со знаком
`SMLAL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} += Rn * Rm`	
`UMULL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} = Rn * Rm`	Команды 32-битного умножения чисел без знака
`UMLAL RdLo, RdHi, Rn, Rm ; {RdHi,RdLo} += Rn * Rm`	

Ко второй группе команд обработки данных относятся команды логических операций и сдвига. Некоторые из наиболее широко используемых команд данной группы приведены в **Табл. 4.20**. Эти команды тоже могут применяться с суффиксом `S`, указывающим на необходимость изменения регистра APSR. При использовании синтаксиса UAL и отсутствии в записи команды суффикса `S` в код будет вставлен 32-битный вариант команды, поскольку все 16-битные команды логических операций влияют на состояние регистра APSR.

Таблица 4.20. **Команды логических операций**

Команда	Операция
`AND Rd, Rn ; Rd = Rd & Rn`	Побитовое И
`AND.W Rd, Rn,#immed ; Rd = Rn & #immed`	
`AND.W Rd, Rn, Rm ; Rd = Rn & Rd`	
`ORR Rd, Rn ; Rd = Rd | Rn`	Побитовое ИЛИ
`ORR.W Rd, Rn,#immed ; Rd = Rn | #immed`	
`ORR.W Rd, Rn, Rm ; Rd = Rn | Rd`	
`BIC Rd, Rn ; Rd = Rd & (~Rn)`	Очистка битов
`BIC.W Rd, Rn,#immed ; Rd = Rn & (~#immed)`	
`BIC.W Rd, Rn, Rm ; Rd = Rn & (~Rd)`	
`ORN.W Rd, Rn,#immed ; Rd = Rn | (~#immed)`	Побитовое ИЛИ с инверсией
`ORN.W Rd, Rn, Rm ; Rd = Rn | (~Rd)`	
`EOR Rd, Rn ; Rd = Rd ^ Rn`	Побитовое «Исключающее ИЛИ»
`EOR.W Rd, Rn,#immed ; Rd = Rn | #immed`	
`EOR.W Rd, Rn, Rm ; Rd = Rn | Rd`	

Как уже было сказано, в процессоре Cortex-M3 имеются команды простого и циклического сдвига. В некоторых случаях операция циклического сдвига может объединяться с другими операциями, например при вычислении смещения адреса для команд загрузки/сохранения. Примеры команд, выполняющих операции сдвига, приведены в **Табл. 4.21**. И опять же, при использовании синтаксиса UAL и отсутствии в мнемонике команды суффикса `S` в код вставляется 32-битный вариант команды.

Таблица 4.21. **Команды сдвига**

Команда	Операция
`ASR Rd, Rn, #immed ; Rd = Rn » immed`	Арифметический сдвиг вправо
`ASR Rd, Rn ; Rd = Rd » Rn`	
`ASR.W Rd, Rn, Rm ; Rd = Rn » Rm`	
`LSL Rd, Rn, #immed ; Rd = Rn « immed`	Логический сдвиг влево
`LSL Rd, Rn ; Rd = Rd « Rn`	
`LSL.W Rd, Rn, Rm ; Rd = Rn « Rm`	
`LSR Rd, Rn, #immed ; Rd = Rn » immed`	Логический сдвиг вправо
`LSR Rd, Rn ; Rd = Rd » Rn`	
`LSR.W Rd, Rn, Rm ; Rd = Rn » Rm`	

Таблица 4.21. **Команды сдвига** *(продолжение)*

Команда	Операция
`ROR Rd, Rn ; Rd = Rd rot by Rn`	Циклический сдвиг вправо
`ROR.W Rd, Rn, #immed ; Rd = Rn rot by immed`	
`ROR.W Rd, Rn, Rm ; Rd = Rn rot by Rm`	
`RRX.W Rd, Rn ; {C, Rd} = {Rn, C}`	Расширенный циклический сдвиг вправо

При использовании синтаксиса UAL и при наличии суффикса `S` команды обычного и циклического сдвигов также могут изменять состояние флага переноса (16-битные команды Thumb всегда влияют на этот флаг), как показано на **Рис. 4.1**. При сдвиге содержимого регистра на несколько битов состояние флага переноса C будет равно значению последнего бита, выдвинутого из регистра.

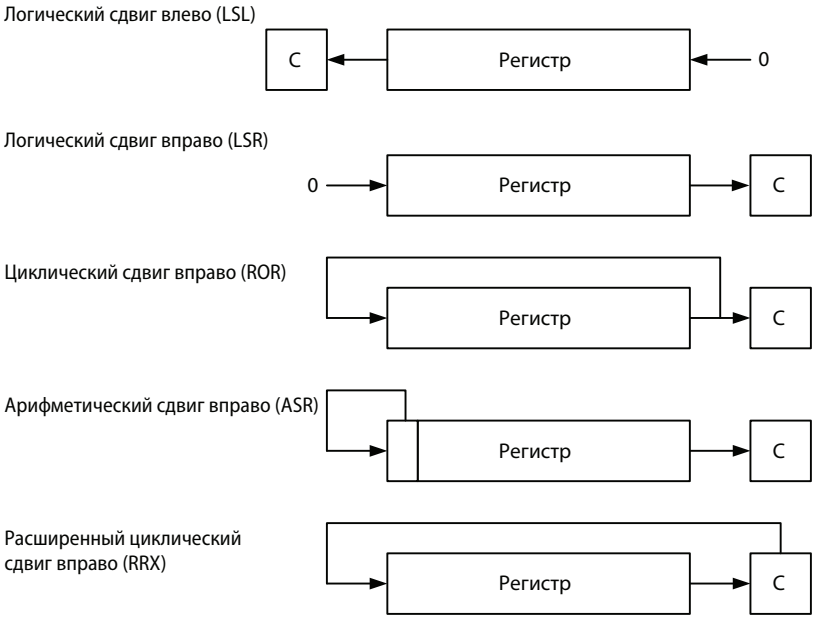


Рис. 4.1. *Команды обычного и циклического сдвига.*

Почему имеются команды циклического сдвига вправо, но нет команд циклического сдвига влево?

Операция циклического сдвига влево может быть заменена операцией циклического сдвига вправо с другим значением величины сдвига. Так, операция циклического сдвига на 4 бита влево может быть выполнена с помощью команды циклического сдвига на 28 бит вправо. Обе команды приводят к одинаковому результату и выполняются за одно и то же время.

Для изменения разрядности знакового значения с байта или полуслова до слова в процессоре Cortex-M3 предусмотрено две команды, приведённые в **Табл. 4.22**. Имеются как 16-битный, так и 32-битный варианты этих команд, причём 16-битные команды могут обращаться только к младшим регистрам.

Таблица 4.22. **Команды расширения знака**

Команда	Операция
`SXTB Rd, Rm ; Rd = signext(Rm[7:0])`	Расширение знака байта до слова
`SXTH Rd, Rm ; Rd = signext(Rm[15:0])`	Расширение знака полуслова до слова

Ещё одна группа команд обработки данных предназначена для перестановки байтов содержимого регистра (**Табл. 4.23**). Эти команды обычно используются для преобразования данных с прямым порядком байтов в данные с обратным порядком байтов, и наоборот (**Рис. 4.2**). Имеются как 16-битный, так и 32-битный варианты этих команд, причём 16-битные команды могут обращаться только к младшим регистрам.

Таблица 4.23. **Команды изменения порядка байтов**

Команда	Операция
`REV   Rd, Rn  ; Rd = rev(Rn)`	Перестановка байтов слова
`REV16 Rd, Rn  ; Rd = rev16(Rn)`	Перестановка байтов каждого полуслова
`REVSH Rd, Rn  ; Rd = revsh(Rn)`	Перестановка байтов младшего полуслова и расширение знака результата

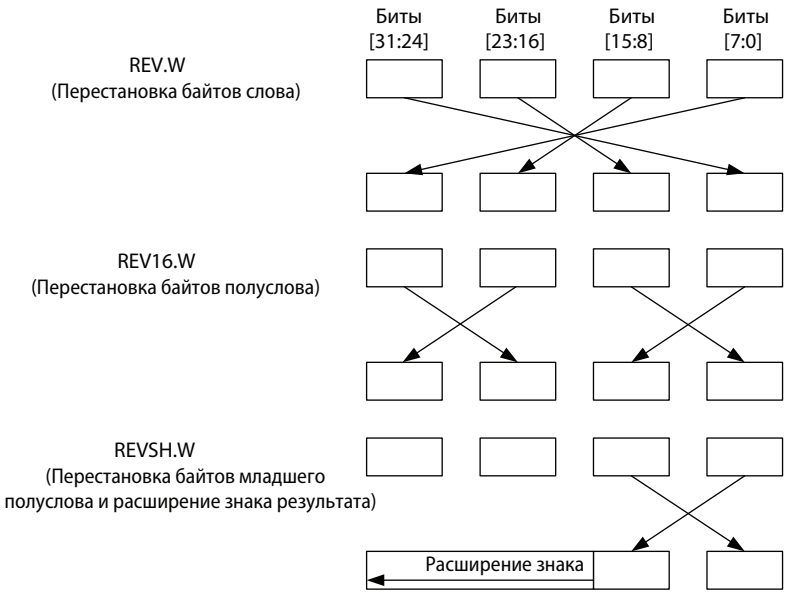


Рис. 4.2. *Команды перестановки байтов.*

Последнюю группу команд обработки данных составляют команды операций с битовыми полями. Все команды из этой группы перечислены в **Табл. 4.24**. Примеры использования этих команд приводятся в последнем разделе главы.

Таблица 4.24. **Команды операций с битовыми полями**

Команда	Операция
BFC.W Rd, Rn, #<width>	Очистка битового поля в регистре
BFI.W Rd, Rn, #<lsb>, #<width>	Вставка битового поля в регистр
CLZ.W Rd, Rn	Подсчёт ведущих нулевых битов
RBIT.W Rd, Rn	Перестановка битов в регистре
SBFX.W Rd, Rn, #<lsb>, #<width>	Копирование битового поля из регистра-источника и расширение его знака
UBFX.W Rd, Rn, #<lsb>, #<width>	Копирование битового поля из регистра-источника

4.3.4. Язык ассемблера: вызов подпрограмм и безусловный переход

Наиболее часто в программах используются следующие команды ветвления:

```
B label ; Переход по адресу, обозначенному меткой
BX reg  ; Переход по адресу, определяемому регистром
```

В случае команды BX младший бит значения, находящегося в регистре, определяет следующее состояние (Thumb/ARM) процессора. Поскольку процессор Cortex-M3 всегда работает в состоянии Thumb, этот бит должен быть установлен в 1. Если он окажется сброшенным, то произойдёт генерация исключения Usage Fault, поскольку в данном случае команда попытается переключить процессор в состояние ARM.

Для вызова функции следует использовать команду перехода со ссылкой.

```
BL label ; Переход по адресу, обозначенному меткой, с сохранением
         ; адреса возврата в LR
BLX reg  ; Переход по адресу, определяемому регистром, с сохранением
         ; адреса возврата в LR
```

При выполнении этих команд адрес возврата сохраняется в регистре связи (LR), что даёт возможность вернуться в вызвавший подпрограмму процесс с помощью команды BX LR. При использовании команды BLX необходимо убедиться, что младший бит содержимого регистра установлен в 1. В противном случае, будет сгенерировано исключение Usage Fault, поскольку команда попытается переключить процессор в состояние ARM.

Операции перехода также можно выполнять, используя команды MOV и LDR, например

```
MOV R15, R0   ; Переход по адресу, содержащемуся в R0
LDR R15, [R0] ; Переход по адресу, содержащемуся в ячейке памяти
              ; с адресом, определяемым R0
POP {R15}     ; Извлекаем значение адреса возврата из стека
              ; в счётчик команд
```

Выполняя переходы с помощью указанных методов, не забывайте о необходимости установки младшего бита нового значения счётчика команд в 1. В противном случае, будет генерироваться исключение Usage Fault, поскольку команды будут пытаться переключить процессор в состояние ARM, которое им не поддерживается.

Сохраняйте LR при вызове вложенных подпрограмм

Команда `BL` портит текущее содержимое регистра LR. Соответственно, если это значение потребуется далее в программе, его необходимо сохранить перед выполнением команды `BL`. Как правило, содержимое регистра LR сохраняется в стеке в самом начале подпрограммы. Например,

```
    main
    ...
    BL functionA
    ...
functionA
    PUSH {LR} ; Сохраняем содержимое LR в стеке
    ...
    BL functionB
    ...
    POP {PC}  ; Используем значение LR, находящееся в стеке,
              ; для возврата в main
functionB
    PUSH {LR}
    ...
    POP {PC}  ; Используем значение LR, находящееся в стеке,
              ; для возврата в functionA
```

Если вызываемая вами подпрограмма написана на языке Си, то вам может также потребоваться сохранить содержимое регистров R0…R3 и R12, если предполагается последующее использование этих значений. В соответствии со стандартом AAPCS [5] подпрограммы на языке Си могут изменять содержимое указанных регистров.

4.3.5. Язык ассемблера: условное выполнение и переходы

Большинство команд условных переходов в процессорах ARM принимают решение о выполнении перехода на основании состояний флагов регистра APSR. Этот регистр содержит пять флагов, четыре из которых используются командами условных переходов (**Табл. 4.25**).

Таблица 4.25. **Флаги регистра APSR, используемые при выполнении условных переходов**

Флаг	Позиция	Описание
N	31	Отрицательное значение (результатом последней операции было отрицательное значение)
Z	30	Ноль (результатом последней операции было нулевое значение)
C	29	Перенос (последняя операция вызвала перенос или заём)
V	28	Переполнение (последняя операция вызвала переполнение)

В 27-м бите регистра расположен пятый флаг — *Q*. Этот флаг предназначен для выполнения математических операций с насыщением и не применяется в командах условных переходов.

Флаги в процессорах ARM

Очень часто команды обработки данных изменяют флаги в регистре состояния программы. Значения этих флагов могут использоваться для выполнения условных переходов или же в качестве аргументов последующих команд. В процессорах ARM содержатся, как минимум, флаги Z, N, C и V, состояние которых изменяется в зависимости от результатов выполнения команд обработки данных:

- *Z* (ноль) — этот флаг устанавливается, если в результате выполнения команды было получено нулевое значение или при сравнении двух одинаковых значений.
- *N* (отрицательное значение) — этот флаг устанавливается, если результатом выполнения команды является отрицательное значение (бит 31 установлен в 1).
- *C* (перенос) — этот флаг используется при обработке беззнаковых данных. Например, при сложении (команда `ADD`) данный флаг устанавливается в случае переполнения; при вычитании (команда `SUB`) этот флаг устанавливается в случае отсутствия заёма (признак заёма является инверсией признака переноса).
- *V* (переполнение) — этот флаг используется при обработке данных со знаком. Так, данный флаг устанавливается, если при сложении двух положительных значений получается отрицательное значение или же если при сложении двух отрицательных значений получается положительное значение.

При использовании указанных флагов с командами сдвига они могут изменяться и по другим правилам. Более подробно это описано в [2].

Различные комбинации четырёх флагов (N, Z, C и V) позволяют определить 15 условий переходов (**Табл. 4.26**). Используя эти условия, команды ветвления могут быть записаны следующим образом:

```
BEQ label ; Переход по адресу 'label', если флаг Z установлен
```

***Таблица 4.26.* Условия для команд переходов и прочих условно выполняемых операций**

Мнемоника	Условие	Флаг
EQ	Равно	Z установлен
NE	Не равно	Z сброшен
CS/HS	Перенос/выше или то же (беззнаковое «≥»)	C установлен
CC/LO	Нет переноса/ниже (беззнаковое «<»)	C сброшен
MI	Минус/отрицательное значение	N установлен
PL	Плюс/положительное значение или ноль	N сброшен
VS	Переполнение	V установлен
VC	Нет переполнения	V сброшен
HI	Выше (беззнаковое «>»)	C установлен и Z сброшен
LS	Ниже или то же (беззнаковое «≤»)	C сброшен или Z установлен
GE	Больше или равно (знаковое «≥»)	N установлен и V установлен или N сброшен и V сброшен (N == V)
LT	Меньше (знаковое «<»)	N установлен и V сброшен или N сброшен и V установлен (N != V)

Таблица 4.26. **Условия для команд переходов и прочих условно выполняемых операций** *(продолжение)*

Мнемоника	Условие	Флаг
GT	Больше (знаковое «>»)	Z сброшен и либо N установлен и V установлен, либо N сброшен и V сброшен (Z == 0, N == V)
LE	Меньше или равно (знаковое «≤»)	Z сброшен или либо N установлен и V сброшен, либо N сброшен и V установлен (Z == 1 или N != V)
AL	Всегда (безусловное выполнение)	—

Если конечный адрес расположен достаточно далеко, то можно использовать 32-битные варианты этих команд, например

```
BEQ.W label ; Переход по адресу 'label', если флаг Z установлен
```

Условия переходов также могут использоваться в условных блоках IF-THEN-ELSE. Например,

```
CMP   R0, R1 ; Сравниваем R0 и R1
ITTEE GT     ; Проверяем условие R0 > R1:
             ; если истина, то выполняются две первые команды,
             ; если ложь, то выполняются две оставшиеся команды
MOVGT R2, R0 ; R2 = R0
MOVGT R3, R1 ; R3 = R1
MOVLE R2, R1 ; R2 = R1
MOVLE R3, R0 ; R3 = R0
```

Состояние флагов APSR может изменяться в следующих случаях:

- при выполнении большинства 16-битных команд арифметических и логических операций;
- при выполнении 32-битных (Thumb-2) команд в случае указания суффикса S, например ADDS.W;
- при выполнении команд сравнения (например, CMP) и проверки (например, TST и TEQ);
- при непосредственной записи в регистр APSR/xPSR.

Большинство 16-битных команд Thumb влияют на состояние флагов N, Z, C и V. Команды Thumb-2 могут влиять на эти флаги, а могут и не влиять. Например:

```
ADDS.W R0, R1, R2 ; Эта 32-битная команда Thumb влияет на состояние флагов
ADD.W  R0, R1, R2 ; Эта 32-битная команда Thumb не влияет на состояние флагов
```

Будьте внимательны при повторном использовании ранее написанного кода из старых проектов. Если в старом проекте использовался традиционный синтаксис Thumb, то

```
ADD R0, R1   ; Эта 16-битная команда Thumb влияет на состояние флагов
ADD R0, #0x1 ; Эта 16-битная команда Thumb влияет на состояние флагов
```

Если же вы вставите этот код в программу, использующую синтаксис UAL (в программе имеется директива ассемблера THUMB), то

```
ADD R0, R1   ; Эта 16-битная команда Thumb влияет на состояние флагов
ADD R0, #0x1 ; Эта запись соответствует 32-битной команде Thumb,
             ; которая не влияет на состояние флагов
```

Чтобы гарантировать получение корректно работающего кода при использовании различных средств разработки, вы должны всегда явно указывать суффикс S, если для выполнения последующих команд, таких как команды условного перехода, требуются актуальные значения флагов.

Команда сравнения (CMP) вычитает одно значение из другого и обновляет флаги (аналогично команде SUBS), однако результат вычитания нигде не сохраняется. Команда CMP имеет следующие форматы записи:

```
CMP R0, R1     ; Вычисляет R0 - R1 и обновляет флаги
CMP R0, #0x12  ; Вычисляет R0 - 0x12 и обновляет флаги
```

Похожая команда CMN сравнивает одну величину с отрицательным значением (дополнением до двух) другой величины; флаги обновляются, однако результат нигде не сохраняется.

```
CMN R0, R1     ; Вычисляет R0 - (-R1) и обновляет флаги
CMN R0, #0x12  ; Вычисляет R0 - (-0x12) и обновляет флаги
```

Команда проверки TST очень похожа на команду AND. Она выполняет побитовую операцию И между двумя значениями и обновляет флаги, однако результат этой операции нигде не сохраняется. Как и команда CMP, данная команда имеет два формата записи:

```
TST R0, R1     ; Вычисляет R0 AND R1 и обновляет флаги
TST R0, #0x12  ; Вычисляет R0 AND 0x12 и обновляет флаги
```

4.3.6. Язык ассемблера: объединение операций сравнения и условного перехода

В процессоре Cortex-M3 имеются две новые команды, выполняющие сравнение с нулем и последующий переход в зависимости от результата сравнения. Это команды CBZ (сравнение и переход, если ноль) и CBNZ (сравнение и переход, если не ноль).

Команды сравнения и перехода поддерживают переходы только в прямом направлении. Возьмём, к примеру, такой фрагмент программы на языке Си:

```
i = 5;
while (i != 0 )
{
    func1();       // Вызов функции
    i--;
}
```

В результате компиляции может быть получен следующий код:

```
      MOV R0, #5        ; Инициализируем счётчик цикла
loop1 CBZ R0, loop1exit ; Если счётчик = 0, то выходим из цикла
      BL  func1         ; Вызываем функцию
      SUB R0, #1        ; Декрементируем счётчик цикла
      B   loop1         ; К следующей итерации
loop1exit
```

Использование команды CBNZ полностью аналогично использованию команды CBZ, за исключением того, что переход выполняется при сброшенном флаге Z (результат не равен нулю). К примеру, фрагмент исходного текста:

```
status = strchr(email_address, '@');
if (status == 0) {    // status = 0, если в email_address отсутствует символ @
    show_error_message();
    exit(1);
}
```

может быть скомпилирован следующим образом:

```
    ...
    BL   strchr
    CBNZ R0, email_looks_okay ; Переходим, если результат не равен нулю
    BL   show_error_message
    BL   exit
email_looks_okay
    ...
```

Команды CBZ и CBNZ не влияют на содержимое регистра APSR.

Язык ассемблера: условное выполнение с использованием команд IT

Такие конструкции, как IT-блоки (IF-THEN), очень полезны для реализации небольших по объёму фрагментов условно выполняемого кода. Они позволяют избавиться от недостатков, связанных с использованием команд перехода, поскольку не вмешиваются в ход выполнения программы. Эти блоки могут содержать до четырёх условно выполняемых команд.

В первой строке IT-блока располагается команда IT, после которой указывается проверяемое условие. Первое выражение после команды IT выполняется в случае истинности проверяемого условия. Последующие выражения (со второго по четвёртое) могут выполняться либо в случае истинности условия, либо при его ложности. В команде IT это указывается в виде ITxyz, где истинности соответствует символ «T» (THEN), а ложности — символ «E» (ELSE):

```
    IT<x><y><z> <cond>                  ; Команда IT (<x>, <y>, <z>
                                        ; могут быть T или E)
    instr1<cond> <operands>             ; 1-я команда (условие <cond> должно
                                        ; быть таким же, как в команде IT)
    instr2<cond или not cond> <operands> ; 2-я команда (условие может быть таким
                                        ; же, как в команде IT, или обратным)
    instr3<cond или not cond> <operands> ; 3-я команда (условие может быть таким
                                        ; же, как в команде IT, или обратным)
    instr4<cond или not cond> <operands> ; 4-я команда (условие может быть таким
                                        ; же, как в команде IT, или обратным)
```

Если выражение должно выполняться в случае ложности условия <cond>, то суффикс условия команды должен быть обратным тому условию, которое указано в команде IT. Так, для условия EQ обратным является условие NE, для условия GT — LE и т.д. Ниже приведён простой пример использования условного выполнения команд:

```
    if (R1<R2) then
       R2=R2-R1
       R2=R2/2
    else
```

```
R1=R1-R2
R1=R1/2
```

На языке ассемблера этот код будет выглядеть следующим образом:

```
CMP     R1, R2  ; Если R1 < R2 (меньше чем),
ITTEE   LT      ; то выполняются команды 1 и 2
                ; (на это указывает символ T),
                ; иначе выполняются команды 3 и 4
                ; (на это указывает символ E)
SUBLT.W R2,R1   ; 1-я команда
LSRLT.W R2,#1   ; 2-я команда
SUBGE.W R1,R2   ; 3-я команда (обратите внимание, что условие GE
                ; противоположно условию LT)
LSRGE.W R1,#1   ; 4-я команда
```

Допускается использование меньшего числа условно выполняемых команд, но не менее одной. При этом количество символов «T» и «E» в мнемонике команды IT должно соответствовать числу используемых команд.

При возникновении исключения внутри IT-блока состояние процесса выполнения блока будет запоминаться в битовом поле ICI регистра PSR, значение которого сохраняется в стеке. Благодаря этому после завершения обработчика исключения и возобновления выполнения IT-блока остальные команды блока будут выполнены корректно. Если исключение возникнет во время исполнения команды, выполняющейся за несколько тактов (например, команды групповой загрузки и сохранения), то выполнение данной команды будет полностью прервано и запущено повторно после завершения обработки прерывания.

4.3.7. Язык ассемблера: команды барьерной синхронизации

Процессор Cortex-M3 поддерживает несколько команд барьерной синхронизации. Необходимость в этих командах вызвана постоянно растущей сложностью систем памяти процессоров. В ряде случаев, если не использовать команды барьерной синхронизации, возможно возникновение «гонок».

Например, если допускается переключение карты памяти с помошью аппаратного регистра, то после записи в этот регистр вы должны использовать команду DSB. В противном случае, если операция записи в регистр переключения памяти буферирована и выполняется за несколько тактов, а следующая команда сразу же обращается к переключённой области памяти, то обращение произойдёт с использованием старой карты памяти. В ряде случаев, если переключение памяти произойдёт одновременно с обращением к ней, это может вызвать ошибку доступа. Использование в данной ситуации команды DSB позволит гарантировать, что запись в регистр переключения карты памяти будет завершена до выполнения следующей команды.

Процессором Cortex-M3 поддерживаются три команды барьерной синхронизации:

- DMB;
- DSB;
- ISB.

Все эти команды описаны в **Табл. 4.27**.

Таблица 4.27. **Команды барьерной синхронизации**

Команда	Описание
DMB	Барьер памяти данных; обеспечивает завершение всех операций с памятью до выполнения нового обращения к ней
DSB	Барьер синхронизации данных; гарантирует, что все операции с памятью будут завершены до исполнения следующей команды
ISB	Барьер синхронизации команд; очищает конвейер и гарантирует завершение всех предыдущих команд перед выполнением новых команд

Команды барьерной синхронизации могут использоваться в Си-программах посредством вызова соответствующих функций CMSIS-совместимой библиотеки драйвера устройства:

```
void __DMB(void); // Барьер памяти данных
void __DSB(void); // Барьер синхронизации данных
void __ISB(void); // Барьер синхронизации команд
```

Команды DSB и ISB могут потребоваться в случае самомодифицирующегося кода. Предположим, к примеру, что программа изменяет свой собственный код, и следующая исполняемая команда должна соответствовать новому коду. Однако, поскольку в процессоре имеется конвейер, в нём может уже находиться старое содержимое модифицированной ячейки памяти. Использование команды DSB, а затем ISB позволит гарантировать, что команда модифицированного кода будет считана в конвейер повторно.

С точки зрения архитектуры, команда ISB должна применяться после каждого изменения регистра CONTROL. В процессоре Cortex-M3 это правило не является строго обязательным к выполнению. И всё же, если вы хотите обеспечить переносимость своего приложения, вы должны обеспечить наличие команды ISB после каждого изменения регистра CONTROL.

Команда DMB чрезвычайно полезна в многопроцессорных системах. Например, задача, выполняемая отдельным процессором, может использовать для взаимодействия с другими задачами разделяемую область памяти. В такой среде порядок доступа к разделяемой области памяти может оказаться чрезвычайно важным. Команда DMB может применяться между обращениями к разделяемой памяти, обеспечивая необходимую последовательность этих операций.

Более подробно вопрос использования команд барьерной синхронизации рассмотрен в [2].

4.3.8. Язык ассемблера: операции насыщения

Процессор Cortex-M3 поддерживает две команды, обеспечивающие выполнение операции насыщения для знаковых (SSAT) и беззнаковых (USAT) данных.

Наиболее часто арифметика с насыщением используется при обработке сигналов, скажем, при их усилении. При усилении сигнала всегда существует вероятность того, что выходной сигнал выйдет за пределы допустимого диапазона. Если корректировка значений осуществляется простым отбрасыванием неис-

пользуемых старших битов, то в результате переполнения мы получим совершенно искажённый сигнал (**Рис. 4.3**).

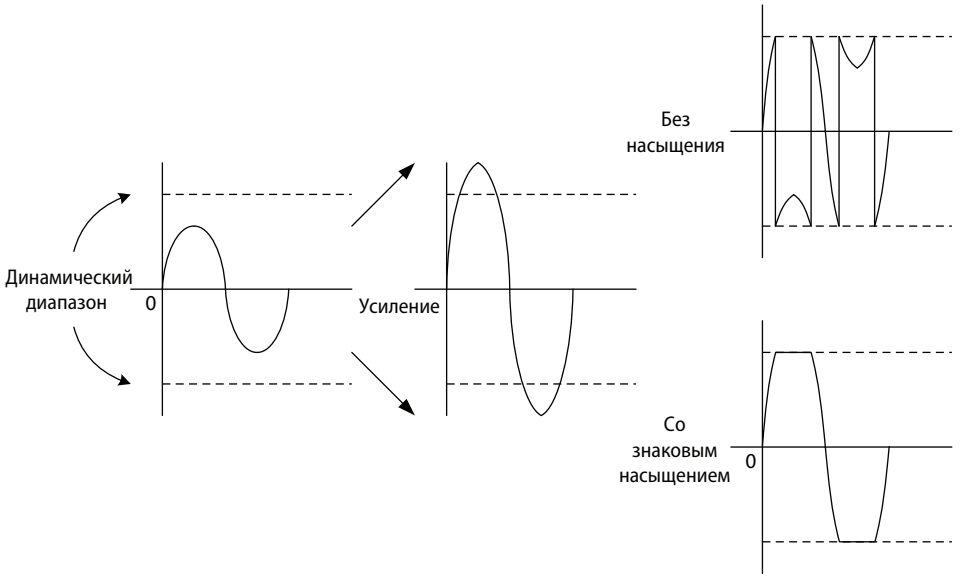


Рис. 4.3. *Операция знакового насыщения.*

Операция насыщения не предотвращает искажение сигнала, однако она, по крайней мере, значительно уменьшает степень этих искажений.

Синтаксис команд `SSAT` и `USAT` описан далее, а также в **Табл. 4.28**.

Таблица 4.28. **Команды насыщения**

Команда	Описание
`SSAT.W <Rd>, #<immed>, <Rn>, {,<shift>}`	Насыщение знакового значения
`USAT.W <Rd>, #<immed>, <Rn>, {,<shift>}`	Насыщение знакового значения в пределах беззнакового диапазона
`Rn` — входное значение; `shift` — операция сдвига входного значения перед выполнением насыщения; необязательная; может быть `#LSL N` или `#ASR N`; `immed` — позиция бита, при достижении которой производится насыщение; `Rd` — регистр-приёмник.	

Операция влияет не только на содержимое регистра-приёмника, но и на состояние флага Q регистра APSR. Этот флаг устанавливается в 1, если при выполнении операции произошло насыщение (**Табл. 4.29**). Флаг Q может быть сброшен непосредственной записью в регистр APSR. Например, если необходимо выполнить насыщение 32-битного значения в пределах 16-битного знакового диапазона, то можно использовать следующую команду:

```
SSAT.W  R1, #16, R0
```

Таблица 4.29. **Некоторые результаты выполнения операции знакового насыщения**

Вход (R0)	Выход (R1)	Бит Q
0x00020000	0x00007FFF	Устанавливается
0x00008000	0x00007FFF	Устанавливается
0x00007FFF	0x00007FFF	Не изменяется
0x00000000	0x00000000	Не изменяется
0xFFFF8000	0xFFFF8000	Не изменяется
0xFFFF7FFF	0xFFFF8000	Устанавливается
0xFFFE0000	0xFFFF8000	Устанавливается

Аналогично, если необходимо выполнить насыщение 32-битного значения в пределах 16-битного беззнакового диапазона, то можно использовать команду:

```
USAT.W  R1, #16, R0
```

Последняя команда выполняет операцию насыщения в соответствии с **Рис. 4.4**. Некоторые выходные значения для этой команды приведены в **Табл. 4.30**.

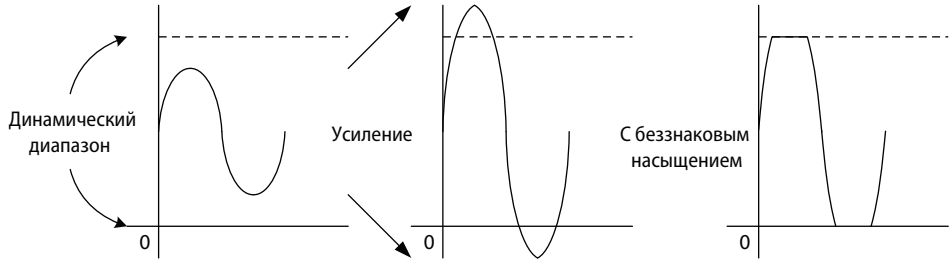


Рис. 4.4. *Операция беззнакового насыщения.*

Таблица 4.30. **Некоторые результаты выполнения операции беззнакового насыщения**

Вход (R0)	Выход (R1)	Бит Q
0x00020000	0x0000FFFF	Устанавливается
0x00008000	0x00008000	Не изменяется
0x00007FFF	0x00007FFF	Не изменяется
0x00000000	0x00000000	Не изменяется
0xFFFF8000	0x00000000	Устанавливается
0xFFFF8001	0x00000000	Устанавливается
0xFFFFFFFF	0x00000000	Устанавливается

Команды насыщения также могут использоваться для преобразования типов данных. В частности, их можно использовать для преобразования 32-битного целого значения в 16-битное. Однако компиляторы языка Си могут оказаться не в состоянии напрямую использовать эти команды, что потребует использования встроенных либо ассемблерных функций (или же встроенного ассемблера) для выполнения преобразования.

4.4. Некоторые полезные команды процессора Cortex-M3

В этом разделе мы познакомимся с несколькими полезными командами Thumb-2, появившимися в архитектурах v6 и v7.

4.4.1. Команды MSR и MRS

Эти две команды обеспечивают доступ к регистрам специального назначения процессора Cortex-M3. Они имеют следующий синтаксис:

```
MRS <Rn>, <SReg> ; Пересылка из регистра специального назначения
MSR <SReg>, <Rn> ; Запись в регистр специального назначения
```

где в качестве <SReg> может использоваться одно из символических имён, перечисленных в **Табл. 4.31**.

***Таблица 4.31.* Обозначения регистров специального назначения для команд MRS и MSR**

Обозначение	Описание
IPSR	Регистр состояния прерывания
EPSR	Регистр состояния выполнения программы (читается как 0)
APSR	Флаги результата предыдущей операции
IEPSR	Объединённые регистры IPSR и EPSR
IAPSR	Объединённые регистры IPSR и APSR
EAPSR	Объединённые регистры EPSR и APSR
PSR	Объединённые регистры APSR, EPSR и IPSR
MSP	Основной указатель стека
PSP	Указатель стека процесса
PRIMASK	Регистр маскирования исключений
BASEPRI	Регистр маскирования низкоприоритетных исключений
BASEPRI_MAX	Тот же регистр маскирования низкоприоритетных исключений, но с функцией условной записи (новый уровень приоритета должен быть выше старого уровня)
FAULTMASK	Регистр маскирования исключений отказов (также запрещает обычные прерывания)
CONTROL	Регистр управления

Например, для инициализации указателя стека процесса может применяться следующий код:

```
LDR R0,=0x20008000 ; Новое значение для указателя стека процесса (PSP)
MSR PSP, R0
```

Обращения ко всем регистрам специального назначения, за исключением регистра APSR, с помощью команд `MSR` и `MRS` допускаются только в привилегированном режиме. В противном случае, операция будет проигнорирована, а возвращаемое значение (в случае команды `MRS`) будет равно нулю.

После изменения содержимого регистра CONTROL рекомендуется вставлять команду синхронизации `ISB`, чтобы обеспечить немедленное вступление в силу сделанных изменений. Правда, при использовании процессора Cortex-M3 это

следует делать исключительно из соображений переносимости кода (если его предполагается использовать с другими процессорами ARM).

4.4.2. Ещё раз об IT-блоке

Мы уже вкратце ознакомились с командой IF–THEN в подразделе «Язык ассемблера: условное выполнение с использованием команд IT». В этом подразделе мы рассмотрим данную команду более детально.

Команда IF–THEN (IT) позволяет сделать условно выполняемыми до четырёх последовательно расположенных команд, образующих IT-блок. Эта команда может иметь различные форматы, представленные в **Табл. 4.32**, в которой:

- <x> определяет условие выполнения второй команды блока;
- <y> определяет условие выполнения третьей команды блока;
- <z> определяет условие выполнения четвёртой команды блока;
- <cond> определяет базовое условие для блока команд; первая команда блока, расположенная после команды IT, выполняется в том случае, если условие <cond> истинно.

***Таблица 4.32.* Различные размеры IT-блока**

Число команд в блоке	Формат команды (каждый из <x>, <y> и <z> может быть T (THEN), либо E (ELSE))	Примеры
Одна условно выполняемая команда	IT <cond> instr1<cond>	IT EQ ADDEQ R0, R0, R1

Две условно выполняемые команды	IT<x> <cond>
	instr1<cond>
	instr2<cond или ~(cond)>
		ITE GE
		ADDGE R0, R0, R1
		ADDLT R0, R0, R3

Три условно выполняемые команды	IT<x><y> <cond>
	instr1<cond>
	instr2<cond или ~(cond)>
	instr3<cond или ~(cond)>
		ITET GT
		ADDGT R0, R0, R1
		ADDLE R0, R0, R3
		ADDGT R2, R4, #1

Четыре условно выполняемые команды	IT<x><y><z> <cond>
	instr1<cond>
	instr2<cond или ~(cond)>
	instr3<cond или ~(cond)>
	instr4<cond или ~(cond)>
		ITETT NE
		ADDNE R0, R0, R1
		ADDEQ R0, R0, R3
		ADDNE R2, R4, #1
		MOVNE R5, R3

Для задания условия <cond> используют такие же аббревиатуры, что и в командах условных переходов. Если в качестве <cond> задано условие AL, то для условного управления нельзя использовать символ «E», поскольку в этом случае он означает, что соответствующая команда никогда не будет выполнена.

На месте каждого из элементов <x>, <y> и <z> может располагаться символ «T» (THEN) или «E» (ELSE), которые относятся к базовому условию <cond>, тогда как для задания <cond> используются традиционные обозначения, такие как EQ, NE, GT и т.п.

Приведём пример использования команды IT:

```
if (R0 equal R1) then {
   R3 = R4 + R5
   R3 = R3/2
```

```
} else {
   R3 = R6 + R7
   R3 = R3/2
}
```

На языке ассемблера данную конструкцию можно записать следующим образом:

```
CMP   R0, R1     ; Сравниваем R0 и R1
ITTEE EQ         ; Если R0 = R1, «То»-«То»-«Иначе»-«Иначе»
ADDEQ R3, R4, R5 ; Сложить, если «равно»
ASREQ R3, R3, #1 ; Сдвинуть вправо, если «равно»
ADDNE R3, R6, R7 ; Сложить, если «не равно»
ASRNE R3, R3, #1 ; Сдвинуть вправо, если «не равно»
```

Команда `IT` используется также при переносе ассемблерного кода с процессора ARM7TDMI на процессор Cortex-M3. Ассемблер ARM (входящий, в том числе, в состав пакета RVMDK компании Keil, которому посвящена Глава 20) при обнаружении в коде «отдельно стоящей» условно выполняемой команды может автоматически вставить требуемую команду `IT` (см. пример в **Табл. 4.33**). Эта особенность ассемблера позволяет, не модифицируя старый код, использовать его с процессором Cortex-M3.

Таблица 4.33. **Автоматическая вставка команды IT ассемблером ARM**

Исходный ассемблерный код	Дизассемблированный код из сгенерированного ассемблером объектного файла
`CMP  R1, #2`	`CMP  R1, #2`
`ADDEQ  R0, R1, #1`	`IT  EQ`
...	`ADDEQ R0, R1, #1`

Заметьте, что 16-битные команды обработки данных не влияют на флаги регистра APSR при их использовании внутри `IT`-блока. Если вы укажете в условно выполняемой команде суффикс `S`, то ассемблер вставит 32-битную версию команды.

4.4.3. Команды SDIV и UDIV

Команды знакового и беззнакового деления имеют следующий синтаксис:

```
SDIV.W <Rd>, <Rn>, <Rm>
UDIV.W <Rd>, <Rn>, <Rm>
```

Результат деления Rn/Rm сохраняется в регистре *Rd*, например:

```
LDR   R0,=300   ; 300 десятичное
MOV   R1, #5
UDIV.W R2, R0, R1
```

В итоге мы получим в R2 число 60 (0x3C).

Если установить бит DIVBYZERO в регистре управления конфигурацией NVIC, то при делении на ноль будет генерироваться исключение Usage Fault. В противном случае, при делении на ноль в регистре *Rd* будет просто возвращаться нулевое значение.

4.4.4. Команды REV, REVH и REVSH

Команда REV изменяет порядок байтов в 32-битном слове, а команда REVH переставляет байты в полусловах. Например, если в регистре R0 содержится значение 0x78563412, то после выполнения команд

```
REV  R1, R0
REVH R2, R0
```

содержимое регистра R1 будет равно 0x78563412, а регистра R2 — 0x34127856. Команды REV и REVH используются, в частности, для изменения формата хранения данных (с прямым/обратным порядком байтов).

Команда REVSH похожа на команду REV, если не считать того, что она переставляет байты только в младшем полуслове, а затем расширяет знак результата. Например, если в R0 содержится 0x33448899, то после выполнения команды

```
REVSH R1, R0
```

содержимое регистра R1 будет равно 0xFFFF9988.

4.4.5. Перестановка битов

Команда RBIT изменяет порядок битов в слове данных. Эта команда имеет следующий синтаксис:

```
RBIT.W <Rd>, <Rn>
```

Команда RBIT часто используется для обработки битовых потоков при передаче данных. К примеру, если R0 равен 0xB4E10C23 (двоичное значение 1011_0100_1110_0001_0000_1100_0010_0011), то после выполнения команды

```
RBIT.W R0, R1
```

в регистре R1 окажется значение 0xC430872D (двоичное значение 1100_0100_0011_0000_1000_0111_0010_1101).

4.4.6. Команды SXTB, SXTH, UXTB и UXTH

Четыре команды SXTB, SXTH, UXTB и UXTH используются для расширения одно- и двухбайтных значений до слова. Эти команды имеют следующий синтаксис:

```
SXTB <Rd>, <Rn>
SXTH <Rd>, <Rn>
UXTB <Rd>, <Rn>
UXTH <Rd>, <Rn>
```

При помощи команд SXTB/SXTH производится расширение знака значения, используя 7-й и 15-й биты содержимого *Rn* соответственно. Команды USTB/UXTH дополняют исходное значение нулями до 32 бит.

Например, если R0 содержит 0x55AA8765, то:

```
SXTB R1, R0 ; R1 = 0x00000065
SXTH R1, R0 ; R1 = 0xFFFF8765
UXTB R1, R0 ; R1 = 0x00000065
UXTH R1, R0 ; R1 = 0x00008765
```

4.4.7. Очистка и вставка битового поля

Команда очистки битового поля BFC обнуляет от 1 до 31 расположенных подряд битов, начиная с любого бита. Команда имеет следующий синтаксис:

```
BFC.W <Rd>, <#lsb>, <#width>
```

Например:

```
LDR   R0,=0x1234FFFF
BFC.W R0, #4, #8
```

В результате мы получим R0 = 0x1234F00F.

Команда вставки битового поля BFI копирует от 1 до 31 бита (#width) из одного регистра в любое место (#lsb) другого регистра. Эта команда имеет следующий синтаксис:

```
BFI.W <Rd>, <Rn>, <#lsb>, <#width>
```

Например:

```
LDR    R0,=0x12345678
LDR    R1,=0x3355AACC
BFI.W  R1, R0, #8, #16 ; Вставляем R0[15:0] в R1[23:8]
```

В результате мы получим R0 = 0x335678CC.

4.4.8. Команды UBFX и SBFX

Команды UBFX и SBFX используются для извлечения беззнакового и знакового битового поля соответственно. Команды имеют следующий синтаксис:

```
UBFX.W <Rd>, <Rn>, <#lsb>, <#width>
SBFX.W <Rd>, <Rn>, <#lsb>, <#width>
```

Команда UBFX извлекает из регистра битовое поле любого размера (#width), начиная с любого бита (определяется значением #lsb), дополняет это значение нулями до 32 бит и помещает его в регистр-приёмник. Например:

```
LDR    R0,=0x5678ABCD
UBFX.W R1, R0, #4, #8
```

В результате мы получим R1 = 0x000000BC.

Аналогично, команда SBFX извлекает из регистра битовое поле, расширяет знак полученного значения до 32 бит и помещает результат в регистр-приёмник. Например:

```
LDR    R0,=0x5678ABCD
SBFX.W R1, R0, #4, #8
```

В результате мы получим R1 = 0xFFFFFFBC.

4.4.9. Команды LDRD и STRD

Команды LDRD и STRD предназначены для пересылки двух слов данных между двумя регистрами и памятью. Команды имеют следующий синтаксис:

```
LDRD.W <Rxf>, <Rxf2>, [Rn, #+/-offset]{!} ; Прединдексация
LDRD.W <Rxf>, <Rxf2>, [Rn], #+/-offset    ; Постиндексация
STRD.W <Rxf>, <Rxf2>, [Rn, #+/-offset]{!} ; Прединдексация
STRD.W <Rxf>, <Rxf2>, [Rn], #+/-offset    ; Постиндексация
```

где <Rxf> — первый регистр-приёмник/регистр-источник, а <Rxf2> — второй. Из-за ошибки в ядре Cortex-M3 ревизий 0…2 при использовании команды LDRD в качестве <Rxf> и <Rxf2> нельзя указывать один и тот же регистр.

В следующем фрагменте выполняется чтение 64-битного значения, расположенного по адресу 0x1000, в регистры R0 и R1:

```
LDR     R2,=0x1000
LDRD.W  R0, R1, [R2] ; В результате R0 = memory[0x1000],
                     ; а R1 = memory[0x1004]
```

Точно так же мы можем использовать команду STRD для сохранения 64-битного значения в памяти. В следующем примере применяется адресация с преиндексацией:

```
LDR    R2,=0x1000          ; Базовый адрес
STRD.W R0, R1, [R2, #0x20] ; В результате memory[0x1020] = R0,
                           ; а memory[0x1024] = R1
```

4.4.10. Команды табличного перехода TBB и TBH

Команды TBB и TBH предназначены для реализации таблиц переходов. Команда TBB использует таблицу с однобайтными значениями смещений, а команда TBH — с двухбайтными значениями. Поскольку 0-й бит счётчика команд всегда равен нулю, значение из таблицы перед прибавлением к PC умножается на два. Ну, а поскольку текущее значение PC равно адресу текущей команды плюс четыре, диапазон переходов для команды TBB составляет (2 × 255) + 4 = 514 байт, а для команды TBH — (2 × 65535) + 4 = 131 074 байт. Обе команды поддерживают переходы только в прямом направлении.

Команда TBB имеет следующий синтаксис:

```
TBB.W [Rn, Rm]
```

где Rn — адрес начала таблицы, а Rm — индекс в таблице. Искомый элемент таблицы переходов располагается по адресу Rn + Rm. Выполнение этой команды при использовании в качестве Rn счётчика команд показано на **Рис. 4.5**.

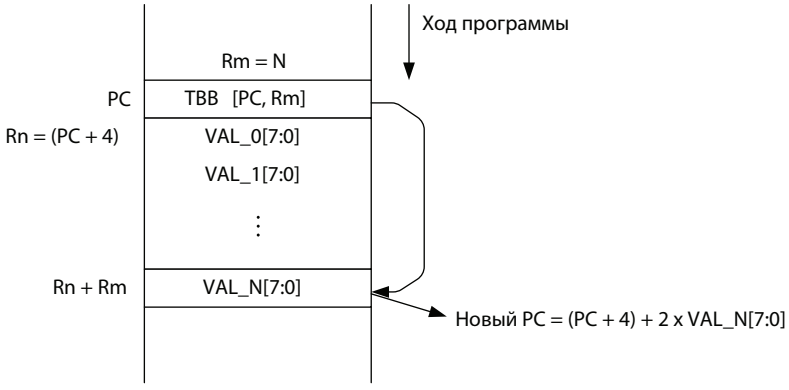


Рис. 4.5. *Выполнение команды* TBB.

Команда TBH выполняет аналогичные операции, за исключением того, что элемент таблицы располагается по адресу Rn + 2 × Rm, а смещения могут иметь

бóльшие значения. Выполнение команды TBH при использовании в качестве Rn счётчика команд показано на **Рис. 4.6**.

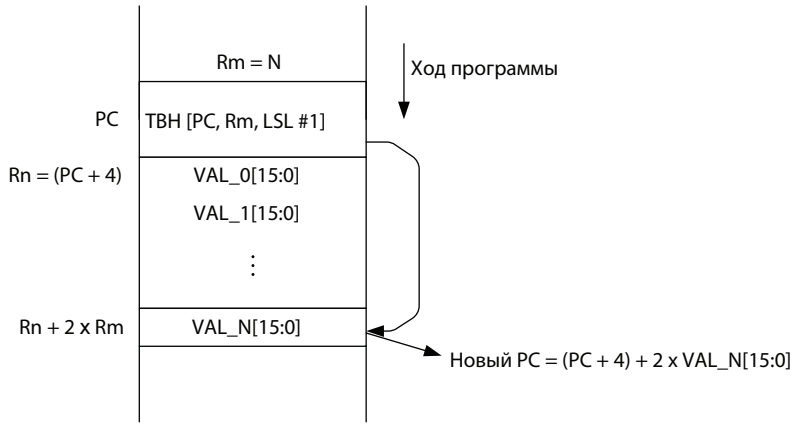


Рис. 4.6. *Выполнение команды* TBH.

Если в командах табличного перехода в качестве операнда *Rn* указывается регистр R15, то из-за наличия конвейера значение, используемое при выполнении команды, будет равно PC + 4. Эти две команды в большей степени ориентированы на использование Си-компиляторами для генерации кода операторов ветвления switch. Поскольку значения, хранящиеся в таблице переходов, задаются относительно текущего значения счётчика команд, то ручное кодирование этой таблицы на ассемблере является довольно сложной задачей. Основные трудности связаны с тем, что на этапе ассемблирования/компиляции подчас невозможно определить величины смещений, особенно если место перехода находится в другом исходном файле. Синтаксические конструкции, используемые для задания содержимого таблиц переходов, зависят от применяемого средства разработки. Так, при использовании ассемблера ARM таблица переходов для команды TBB может быть описана следующим образом:

```
    TBB.W [pc, r0] ; Во время исполнения этой команды PC равен
                   ; branchtable
branchtable
    DCB ((dest0 - branchtable)/2) ; Поскольку значения смещений 8-битные,
                                  ; используются директивы DCB
    DCB ((dest1 - branchtable)/2)
    DCB ((dest2 - branchtable)/2)
    DCB ((dest3 - branchtable)/2)
dest0
    ... ; Выполняем, если r0 = 0
dest1
    ... ; Выполняем, если r0 = 1
dest2
    ... ; Выполняем, если r0 = 2
dest3
    ... ; Выполняем, если r0 = 3
```

При исполнении команды `TBB` текущее значение PC равно адресу, помеченному меткой `branchtable` (из-за наличия в процессоре конвейера).

Команда `TBH` может использоваться аналогичным образом:

```
    TBH.W [pc, r0, LSL #1]
branchtable
    DCI ((dest0 - branchtable)/2) ; Поскольку значения смещений 16-битные,
                                  ; используются директивы DCI
    DCI ((dest1 - branchtable)/2)
    DCI ((dest2 - branchtable)/2)
    DCI ((dest3 - branchtable)/2)
dest0
    ... ; Выполняем, если r0 = 0
dest1
    ... ; Выполняем, если r0 = 1
dest2
    ... ; Выполняем, если r0 = 2
dest3
    ... ; Выполняем, если r0 = 3
```

ГЛАВА 5
СИСТЕМА ПАМЯТИ

5.1. Основные особенности системы памяти

Архитектура памяти, реализованная в процессоре Cortex-M3, отличается от архитектуры, использующейся в традиционных процессорах ARM. Во-первых, процессор Cortex-M3 имеет фиксированную карту памяти, которая определяет, какой шинный интерфейс должен использоваться при обращении к тому или иному участку памяти. Эта особенность также позволяет процессору оптимизировать операции обращения к различным устройствам.

Ещё одной особенностью системы памяти процессора Cortex-M3 является поддержка доступа к отдельным битам памяти (метод bit-band). Это позволяет выполнять атомарные операции с битами памяти или битами регистров периферийных устройств. Правда, необходимо заметить, что операции прямого доступа к битам поддерживаются только для определённых областей памяти. Чуть позже мы рассмотрим данный вопрос более подробно.

Система памяти процессора Cortex-M3 также поддерживает пересылку невыровненных данных и операции монопольного доступа; эти возможности являются частью архитектуры v7-M. И наконец, процессор Cortex-M3 поддерживает память, использующую как прямой, так и обратный порядок хранения байтов.

5.2. Карта памяти

В процессоре Cortex-M3 используется фиксированная карта памяти (**Рис. 5.1**). Это облегчает перенос кода между устройствами на базе Cortex-M3, выпускаемыми различными производителями. Например, компоненты, описанные в предыдущих главах, такие как контроллер прерываний NVIC и модуль защиты памяти MPU, всегда будут располагаться по одним и тем же адресам, независимо от производителя. В то же время используемое распределение карты памяти обеспечивает гибкость, достаточную для выделения продукции одних производителей среди продукции других.

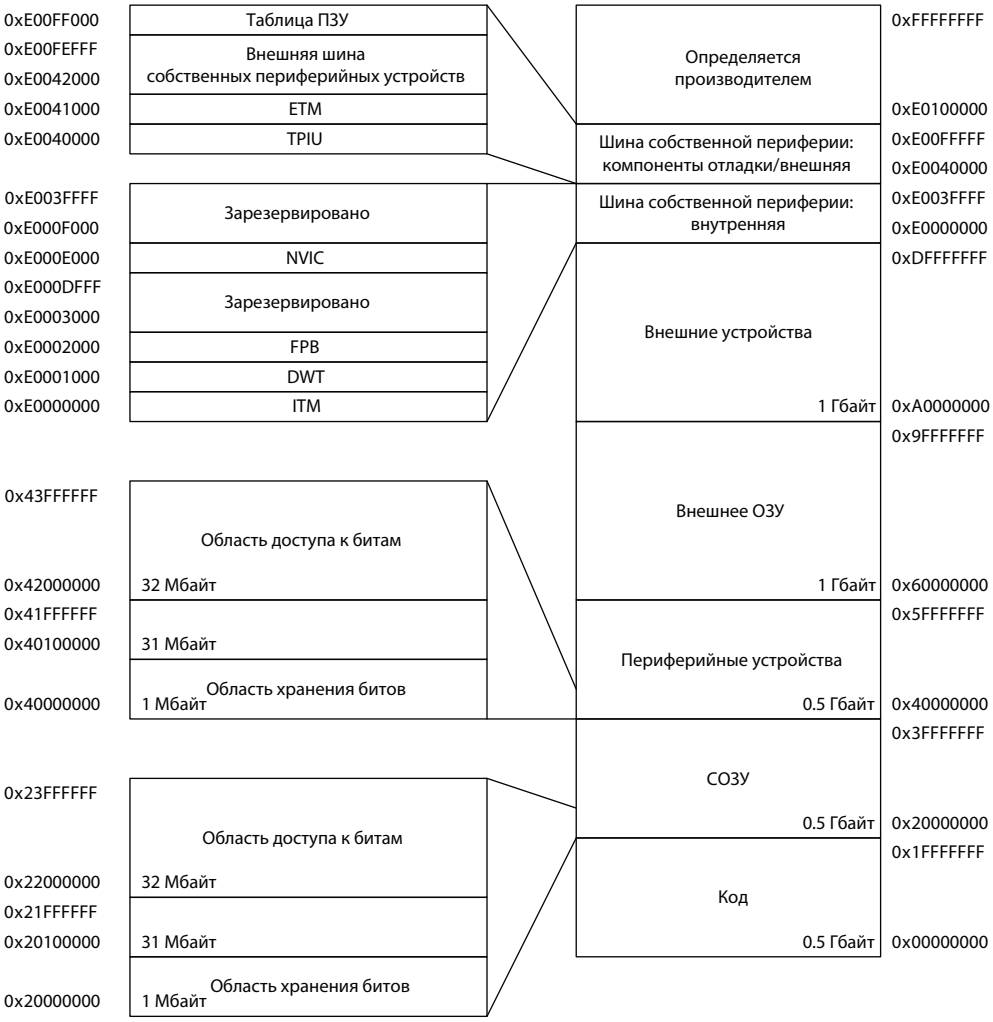


Рис. 5.1. *Предопределённая карта памяти процессора Cortex-M3.*

Некоторые участки памяти зарезервированы для использования встроенными периферийными устройствами, такими как компоненты отладки. Эти устройства располагаются в одноимённой области памяти. К указанным компонентам отладки относятся следующие модули:

- модуль коррекции флэш-памяти и задания точки останова (FPB);
- модуль просмотра и трассировки данных (DWT);
- модуль трассировки (ITM);
- модуль встроенной ячейки трассировки (ETM);
- модуль интерфейса порта трассировки (TPIU);
- таблица ПЗУ.

Более детально эти компоненты будут рассмотрены в других главах, посвящённых возможностям отладки.

Процессор Cortex-M3 имеет адресное пространство размером 4 Гбайт. Программа может располагаться в области кода, области статического ОЗУ (СОЗУ) или области внешнего ОЗУ. Однако наилучшим местом для размещения программы является всё же область кода, поскольку в этом случае операции выборки команд и обращения к данным смогут выполняться одновременно, используя различные интерфейсы шин.

Область статического ОЗУ предназначена для подключения внутренней оперативной памяти. Обращение к области СОЗУ осуществляется по системной шине. В этой области имеется участок размером 32 Мбайт, предназначенный для обращения к области с побитовой адресацией (область хранения битов). Каждое слово внутри указанного диапазона соответствует одному биту в 1-Мбайт области хранения битов. Все операции записи в область доступа к битам преобразуются в атомарные операции вида «чтение—модификация—запись» в области хранения битов, воздействующие на состояния отдельных битов физической памяти. Обратите внимание, что операции побитового доступа поддерживаются только для данных, но никак не для команд! Размещая булевы переменные (одиночные биты) в области с побитовой адресацией, мы можем «упаковать» несколько переменных в одно слово данных. При этом каждую из этих переменных можно будет изменять независимо от других, используя область доступа к битам. В результате экономится память микроконтроллера, поскольку нам не требуется программно реализовывать операции вида «чтение—модификация—запись». Более подробно работа с бит-адресуемой памятью будет рассмотрена ниже.

Ещё одна область памяти размером 0.5 Гбайт выделена для периферийных устройств. Как и область статического ОЗУ, область периферийных устройств поддерживает доступ к отдельным битам с использованием метода bit-band, а обращения к этой области осуществляются по системной шине. Однако размещение исполняемого кода в указанной области не допускается. Наличие в области периферийных устройств секции с побитовой адресацией значительно упрощает работу с битами регистров периферийных устройств, облегчая написание функций для управления данными устройствами.

Две области памяти размером по 1 Гбайт выделены для внешнего ОЗУ и внешних периферийных устройств. Различие между этими областями заключается в том, что из области внешних периферийных устройств не допускается исполнение кода программы. Также имеются некоторые различия в стратегии кэширования.

Последняя распределённая область памяти размером 0.5 Гбайт предназначена для системных компонентов, внешних и внутренних шин встроенных периферийных устройств процессора, а также для периферийных устройств системного уровня, определяемых производителем. При этом шина собственных периферийных устройств (Private Peripheral Bus — PPB) состоит из двух сегментов:

- *усовершенствованная высокопроизводительная шина* (шина AHB), предназначенная для подключения исключительно внутренних AHB-совместимых периферийных устройств процессора, к которым относятся контроллер NVIC и модули FPB, DWT и ITM;

- *усовершенствованная шина периферии* (шина APB), предназначенная для подключения как внутренних APB-совместимых модулей, так и внешней (по отношению к процессору Cortex-M3) периферии. Процессор Cortex-M3 позволяет производителям микросхем подключать к данной шине дополнительные периферийные устройства, поддерживающие спецификацию APB.

Контроллер NVIC располагается в области памяти, называемой пространством управления системой (System Control Space — SCS), см. **Рис. 5.2**. Помимо контроллера NVIC, в этой области располагаются регистры управления модулей SYSTICK и MPU, а также компоненты управления отладкой.

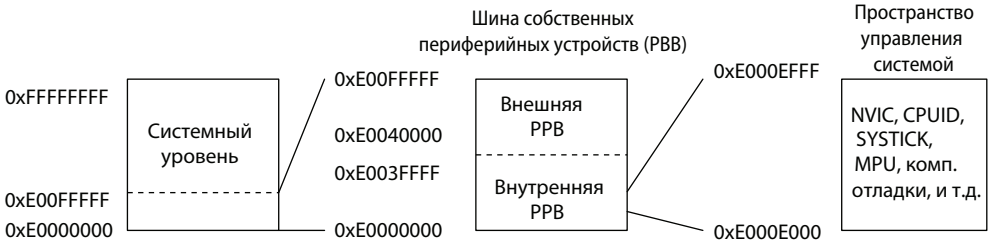


Рис. 5.2. *Пространство управления системой (SCS).*

Оставшуюся часть адресного пространства производители микросхем могут задействовать по своему усмотрению. Для обращения к данной области памяти используется интерфейс системной шины, однако исполнение кода из этой области не допускается.

В процессоре Cortex-M3 предусмотрен опциональный модуль защиты памяти MPU. Наличие или отсутствие указанного модуля в конкретном устройстве зависит от производителя данного микроконтроллера.

Описанное распределение карты памяти процессора можно считать своего рода шаблоном; более подробные карты памяти, в том числе реальное положение и размеры областей ПЗУ и ОЗУ, а также расположение периферийных устройств приводятся изготовителями микросхем в документации на конкретные изделия.

5.3. Атрибуты доступа к памяти

Карта памяти показывает, что именно расположено по тем или иным адресам. При этом она не только предоставляет информацию, к какому блоку памяти или устройству производится доступ, но и определяет атрибуты доступа к каждой из областей памяти. Процессором Cortex-M3 используются следующие атрибуты:

- *Буферируемая* — запись в память может производиться из буфера, в то время как процессор будет выполнять следующую команду.
- *Кэшируемая* — данные, полученные при чтении из памяти, могут быть сохранены в кэш-памяти. В итоге при следующем обращении к этим данным их можно будет считать из кэша, ускоряя тем самым процесс выполнения программы.
- *Исполняемая* — из этой области памяти процессор может осуществлять выборку и исполнение кода программы.

- *Разделяемая* — данные в этой области памяти могут совместно использоваться несколькими контроллерами шины. Система памяти должна обеспечивать когерентность данных, хранящихся в разделяемой области памяти, для всех контроллеров шины.

Интерфейсы шин процессора Cortex-M3 предоставляют системе памяти информацию о перечисленных атрибутах при каждой пересылке команды или данных. Значения атрибутов памяти, присущих отдельным областям, могут переопределяться модулем MPU (при его наличии), если его конфигурация отличается от принятой по умолчанию. Несмотря на то что в процессоре Cortex-M3 нет ни кэша, ни контроллера кэш-памяти, в микроконтроллер может быть добавлен блок внешней кэш-памяти, характер обращений к которой будет определяться атрибутами соответствующей области памяти. Более того, атрибуты кэш-памяти могут также влиять на работу контроллеров встроенной и внешней памяти — это зависит от архитектуры контроллеров памяти, используемых производителями микросхем.

Ниже перечислены атрибуты доступа к памяти для каждой из областей карты памяти:

- *Область кода* (0x00000000…0x1FFFFFFF) — является исполняемой, а атрибут кэширования имеет значение «кэшируемая со сквозной записью» (Write Through — WT). Вы также можете размещать в этой области данные. Обращения к данным, расположенным в указанной области, осуществляются по интерфейсу шины данных. Операции записи в данную область памяти буферизуются.
- *Область СОЗУ* (0x20000000…0x3FFFFFFF) — предназначена для подключения встроенного ОЗУ. Операции записи в эту область буферизуются, а атрибут кэширования имеет значение «кэшируемая с обратной записью и размещением записываемых данных» (WB-WA). Указанная область является исполняемой, так что вы можете скопировать в неё код программы и исполнять его уже оттуда.
- *Область периферийных устройств* (0x40000000…0x5FFFFFFF) — предназначена для размещения периферийных устройств. Обращения к этой области памяти не кэшируются, и исполнение кода из неё не допускается (в документации ARM неисполняемая память имеет атрибут XN, являющийся сокращением от eXecute Never).
- *Область внешнего ОЗУ* (0x60000000…0x7FFFFFFF) — предназначена для встроенной или внешней памяти данных. Обращения к указанной области памяти могут кэшироваться (WB-WA), из этой области также допускается выполнение кода.
- *Область внешнего ОЗУ* (0x80000000…0x9FFFFFFF) — предназначена для встроенной или внешней памяти данных. Обращения к этой области памяти могут кэшироваться (WT), из данной области также допускается выполнение кода.
- *Область внешних устройств* (0xA0000000…0xBFFFFFFF) — предназначена для внешних устройств и/или разделяемой памяти, к которой необходимо

обеспечить упорядоченный и небуферизованный доступ. Эта область также является исполняемой.

- *Область внешних устройств* (0xC0000000…0xDFFFFFFF) — предназначена для внешних устройств и/или разделяемой памяти, к которой необходимо обеспечить упорядоченный и небуферизованный доступ. Область также является исполняемой.
- *Системная область* (0xE0000000…0xFFFFFFFF) — предназначена для собственных периферийных устройств процессора, а также для модулей, определяемых производителем. Исполнение кода из этой области памяти не допускается. В диапазоне адресов, выделенных шине PPB, обеспечивается строго упорядоченный доступ к памяти (некэшируемая и небуферируемая память). В диапазоне адресов, выделенных для изготовителей кристаллов, память является буферируемой, но некэшируемой.

Замечу, что, начиная с 1-й ревизии процессора Cortex-M3, атрибуты памяти области кода (кэшируемая и небуферируемая), транслируемые внешней подсистеме памяти, задаются аппаратно и не могут быть переопределены модулем MPU. Это касается только подсистем памяти, расположенных вне процессора (например, кэш-памяти 2-го уровня и некоторых типов контроллеров памяти с поддержкой кэширования). В процессоре в любом случае имеется внутренний буфер, который может использоваться при операциях записи в область кода.

5.4. Права доступа к памяти, принятые по умолчанию

Каждой из областей карты памяти процессора Cortex-M3 назначены определённые права доступа, используемые по умолчанию. Это позволяет предотвратить обращение пользовательской программы, работающей на непривилегированном уровне, к системным областям памяти, например к регистрам контроллера NVIC. Права доступа, принятые по умолчанию, используются при отсутствии модуля MPU или же в том случае, если он отключён.

Если модуль MPU присутствует и включён, то возможность доступа со стороны пользователя к той или иной области памяти определяется конфигурацией модуля.

Права доступа к различным областям памяти, установленные по умолчанию, указаны в **Табл. 5.1**.

Таблица 5.1. **Права доступа к памяти, установленные по умолчанию**

Область памяти	Диапазон адресов	Доступ из пользовательской программы
Область изготовителя	0xE0100000…0xFFFFFFFF	Полный доступ
Таблица ПЗУ	0xE00FF000…0xE00FFFFF	Заблокирован; при попытке обращения генерируется исключение Bus Fault
Внешняя шина PPB	0xE0042000…0xE00FEFFF	Заблокирован; при попытке обращения генерируется исключение Bus Fault

Таблица 5.1. **Права доступа к памяти, установленные по умолчанию** *(продолжение)*

Область памяти	Диапазон адресов	Доступ из пользовательской программы
ETM	0xE0041000…..0xE0041FFF	Заблокирован; при попытке обращения генерируется исключение Bus Fault
TPIU	0xE0040000…0xE0040FFF	Заблокирован; при попытке обращения генерируется исключение Bus Fault
Внутренняя шина PPB	0xE000F000…0xE003FFFF	Заблокирован; при попытке обращения генерируется исключение Bus Fault
NVIC	0xE000E000…0xE000EFFF	Заблокирован; при попытке обращения генерируется исключение Bus Fault. Исключение — регистр программного запуска прерывания STIR, доступ к которому со стороны пользователя может быть разрешён
FPB	0xE0002000…..0xE0003FFF	Заблокирован; при попытке обращения генерируется исключение Bus Fault
DWT	0xE0001000…0xE0001FFF	Заблокирован; при попытке обращения генерируется исключение Bus Fault
ITM	0xE0000000…0xE0000FFF	Разрешено чтение; запись игнорируется, за исключением операций записи в порты стимулов при условии, что доступ к ним со стороны пользователя разрешён
Внешние устройства	0xA0000000…0xDFFFFFFF	Полный доступ
Внешнее ОЗУ	0x60000000…0x9FFFFFFF	Полный доступ
Периферийные устройства	0x40000000…0x5FFFFFFF	Полный доступ
Статическое ОЗУ	0x20000000…0x3FFFFFFF	Полный доступ
Код	0x00000000…0x1FFFFFFF	Полный доступ

Примечание. Если доступ со стороны пользователя заблокирован, то исключение отказа генерируется немедленно.

5.5. Операции побитового доступа

Поддержка метода побитового доступа bit-band позволяет использовать обычные операции загрузки/сохранения для обращения (чтение/запись) к отдельным битам данных. В процессоре Cortex-M3 побитовый доступ к памяти поддерживается только для двух предопределённых областей памяти, называемых областями хранения битов. Одна из этих областей расположена в 1-м мегабайте адресного пространства СОЗУ, а другая — в 1-м мегабайте адресного пространства периферийных устройств. В принципе, эти области можно использовать как обычную память, однако к ним можно обращаться и посредством специальной области памяти, называемой областью доступа к битам (**Рис. 5.3**). При использовании адреса, расположенного в области доступа к битам, младший бит адресуемого значения обращается к отдельному биту области хранения битов.

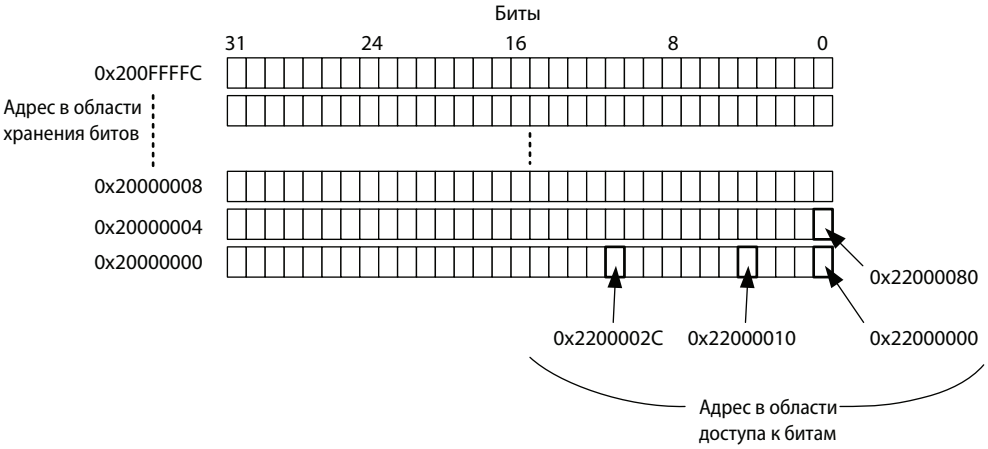


Рис. 5.3. *Обращение к области хранения битов посредством области доступа к битам.*

Например, для установки 2-го бита слова данных, расположенного по адресу 0x20000000, вместо трёх команд (чтение значения, установка бита и сохранение результата), можно использовать всего одну команду (**Рис. 5.4**). Фрагменты ассемблерного кода, соответствующие обоим вариантам, приведены на **Рис. 5.5**.

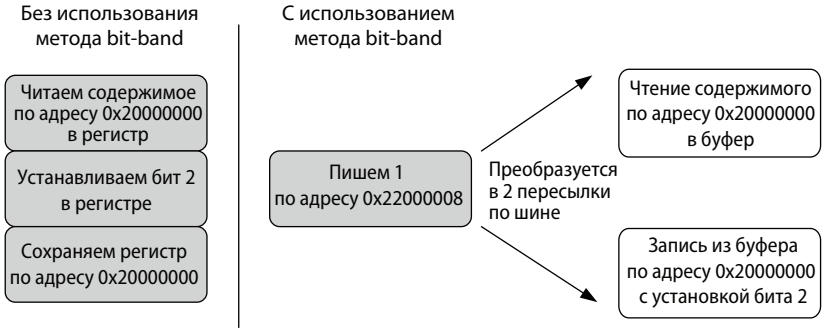


Рис. 5.4. *Запись в область доступа к битам.*

Без использования метода bit-band

```
LDR   R0,=0x20000000 ; Задаём адрес
LDR   R1, [R0]       ; Читаем
ORR.W R1, #0x4       ; Изменяем бит
STR   R1, [R0]       ; Сохраняем результат
```

С использованием метода bit-band

```
LDR   R0,=0x22000008 ; Задаём адрес
MOV   R1, #1         ; Задаём данные
STR   R1, [R0]       ; Пишем
```

Рис. 5.5. *Запись бита с использованием метода bit-band и без него.*

Аналогичным образом, использование метода bit-band может упростить код программы при необходимости чтения отдельного бита памяти. Так, если нам нужно узнать состояние 2-го бита слова, расположенного по адресу 0x20000000, мы можем воспользоваться одним из способов, показанных на **Рис. 5.6**. Фрагменты ассемблерного кода, соответствующие обоим способам, приведены на **Рис. 5.7**.

Рис. 5.6. *Чтение из области доступа к битам.*

Без использования метода bit-band

```
LDR    R0,=0x20000000 ; Задаём адрес
LDR    R1, [R0]       ; Читаем
UBFX.W R1,  R1, #2, #1 ; Выделяем бит 2
```

С использованием метода bit-band

```
LDR   R0,=0x22000008 ; Задаём адрес
LDR   R1, [R0]       ; Читаем
```

Рис. 5.7. *Чтение бита с использованием метода bit-band и без него.*

Метод bit-band, реализованный в процессоре Cortex-M3, не является чем-то новым. На самом деле, аналогичная возможность уже более 30 лет имеется в 8-битных микроконтроллерах, в частности в микроконтроллерах семейства 8051. Хотя в процессоре Cortex-M3 и нет специальных команд для манипуляций с битами, в его адресном пространстве предусмотрены особые области, обращения к которым автоматически преобразуются в обращения к отдельным битам.

Ещё раз напомню, что при адресации памяти посредством метода bit-band используются следующие понятия:

- *Область хранения битов* (bit-band region) — область памяти, поддерживающая доступ по методу bit-band.
- *Адрес доступа к биту* (bit-band alias) — обращение по этому адресу вызывает обращение к области хранения битов (т.е. производится переадресация).

Каждое слово области хранения битов представлено младшими битами 32 слов области доступа к битам. При обращении по адресу, находящемуся в области доступа к битам, производится переадресация на область хранения битов. При операциях чтения производится чтение слова, после чего искомый бит выдвигается в младший бит возвращаемого значения. При записи значение сохраняемого бита сдвигается на заданную позицию, после чего выполняется операция «чтение—модификация—запись».

В адресном пространстве процессора имеется две области памяти, поддерживающие побитовую адресацию:

- 0x20000000…0x200FFFFF (СОЗУ, 1 Мбайт);
- 0x40000000…0x400FFFFF (периферийные устройства, 1 Мбайт).

Соответствие адресов в области хранения битов, расположенной в адресном пространстве СОЗУ, адресам области доступа к битам указано в **Табл. 5.2**.

Таблица 5.2. **Переназначение адресов области хранения битов на область СОЗУ**

Область хранения битов	Область доступа к битам
0x20000000, бит [0]	0x22000000, бит [0]
0x20000000, бит [1]	0x22000004, бит [0]
0x20000000, бит [2]	0x22000008, бит [0]
…	…
0x20000000, бит [31]	0x2200007C, бит [0]
0x20000004, бит [0]	0x22000080, бит [0]
…	…
0x20000004, бит [31]	0x220000FC, бит [0]
…	…
0x200FFFFC, бит [31]	0x23FFFFFC, бит [0]

Аналогичным образом, к области с побитовой адресацией, расположенной в адресном пространстве периферийных устройств, можно обращаться посредством соответствующей области доступа к битам, как указано в **Табл. 5.3**.

Таблица 5.3. **Переназначение адресов области хранения битов на область периферифных устройств**

Область хранения битов	Область доступа к битам
0x40000000, бит [0]	0x42000000, бит [0]
0x40000000, бит [1]	0x42000004, бит [0]
0x40000000, бит [2]	0x42000008, бит [0]
…	…
0x40000000, бит [31]	0x4200007C, бит [0]
0x40000004, бит [0]	0x42000080, бит [0]
…	…
0x40000004, бит [31]	0x420000FC, бит [0]
…	…
0x400FFFFC, бит [31]	0x43FFFFFC, бит [0]

Рассмотрим простой пример:

1. Запишем по адресу 0x20000000 значение 0x3355AACC.
2. Прочитаем слово с адреса 0x22000008. При этом в действительности будет выполнено чтение с адреса 0x20000000. Возвращаемое значение равно 1 (2-й бит значения 0x3355AACC).
3. Запишем 0x0 по адресу 0x22000008. Эта операция преобразуется в операцию вида «чтение—модификация—запись» по адресу 0x20000000. Значение 0x3355AACC считывается из памяти, 2-й бит сбрасывается и полученное значение (0x3355AAC8) записывается обратно по адресу 0x20000000.
4. Теперь выполним чтение с адреса 0x200000. В результате получим число 0x3355AAC8 (2-й бит сброшен).

При обращении к адресам, расположенным в области доступа к битам, используются только младшие (нулевые) биты значений. Кроме того, обращения

к области доступа к битам всегда должны быть выровнены; в противном случае, результат операции будет непредсказуем.

5.5.1. Преимущества использования метода bit-band

Итак, что же нам даёт возможность выполнения операций с отдельными битами? В частности, мы можем использовать эти операции для того, чтобы организовать обмен данными с последовательными устройствами при помощи портов ввода/вывода общего назначения (General-Purpose Input/Output — GPIO). Поскольку обращения к линии данных и линии тактового сигнала могут быть разделены, код программы значительно упрощается.

> **Bit-Band против Bit-Bang**
>
> При описании процессора Cortex-M3 мы используем термин bit-band, чтобы подчеркнуть наличие специальной области памяти (band), поддерживающей побитовую адресацию. Термином же bit-bang обычно обозначается программное управление контактами ввода/вывода с целью реализации функций последовательной передачи данных. Несмотря на то что bit-bang-операции могут быть реализованы с использованием bit-band-операций, поддерживаемых процессором Cortex-M3, эти термины обозначают совершенно разные понятия.

Битовые операции могут также использоваться при организации ветвлений. Например, если переход должен осуществляться на основании значения одного из битов регистра состояния периферийного устройства, то вместо трёх операций:

- чтение всего содержимого регистра;
- маскирование остальных битов;
- сравнение и переход;

вам достаточно выполнить всего две операции:

- чтение бита состояния с использованием метода bit-band (получаем 0 или 1);
- сравнение и переход.

Помимо того, что метод bit-band ускоряет выполнение битовых операций, позволяя обойтись меньшим числом команд, он также играет важную роль при организации совместного использования каких-либо ресурсов несколькими процессами. Одним из важнейших достоинств битовых операций с применением метода bit-band является их атомарность. Другими словами, выполнение последовательности «чтение—модификация—запись» не может быть прервано никакими другими операциями на шине. В противном случае, скажем, при программной реализации последовательности «чтение—модификация—запись», могут возникнуть определённые проблемы. Предположим, что 0-й бит обычного порта вывода используется основной программой, а 1-й бит — обработчиком прерывания. При программной реализации операции «чтение—модификация—запись» могут возникать конфликты по данным, как показано на **Рис. 5.8**.

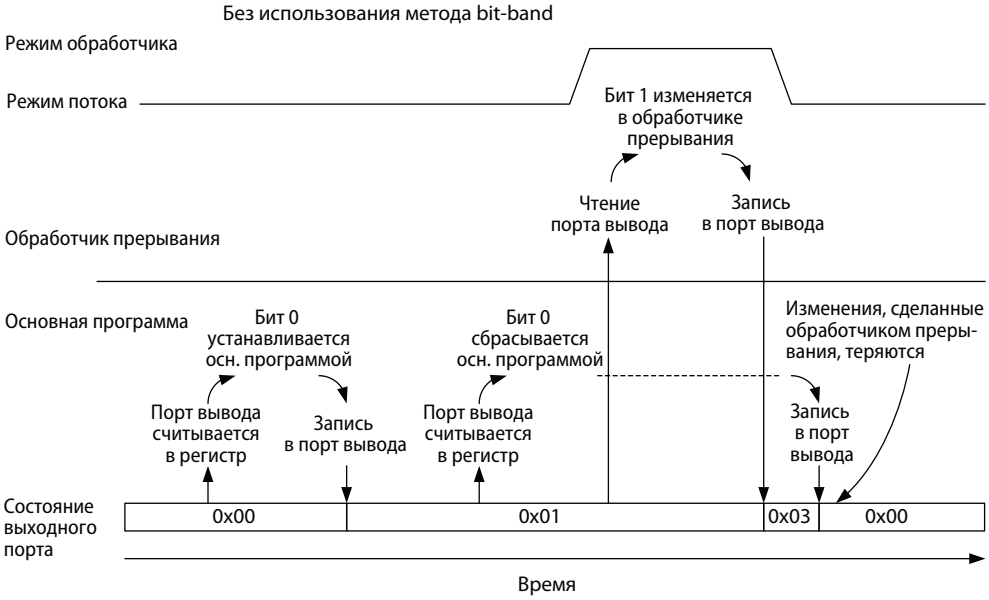


Рис. 5.8. *Потеря данных при изменении содержимого разделяемой ячейки памяти обработчиком исключения.*

Благодаря поддержке процессором Cortex-M3 метода bit-band можно избежать подобных состояний гонок, поскольку операции «чтение—модификация—запись» реализуются на аппаратном уровне и являются атомарными — обе пересылки, осуществляемые при выполнении операции, неотделимы друг от друга и возникновение прерываний между этими пересылками исключено (**Рис. 5.9**).

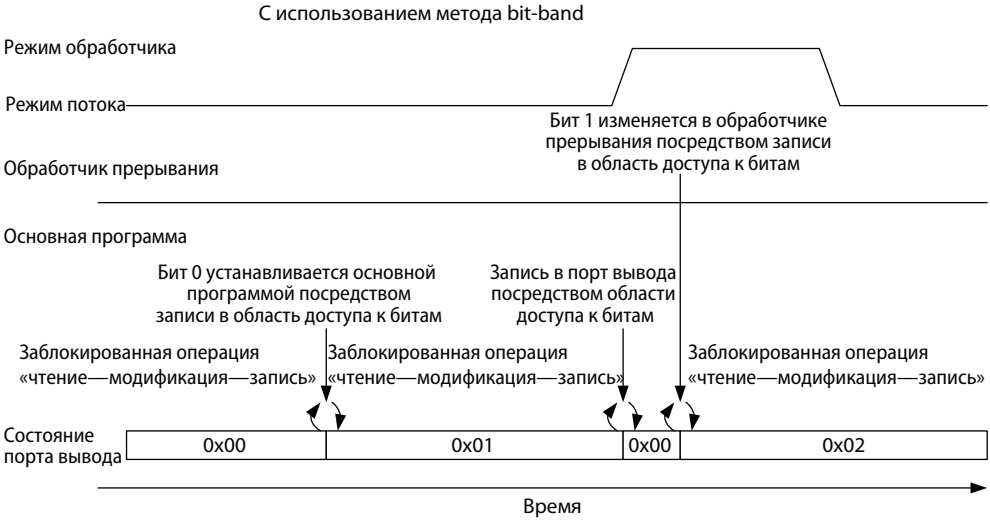


Рис. 5.9. *Предотвращение потери данных благодаря использованию метода bit-band.*

Похожую проблему можно наблюдать в многозадачных системах. Так, если 0-й бит порта вывода используется процессом A, а 1-й бит порта — процессом B, то в случае программной реализации операции «чтение—модификация—запись» точно так же может возникнуть конфликт по данным (**Рис. 5.10**).

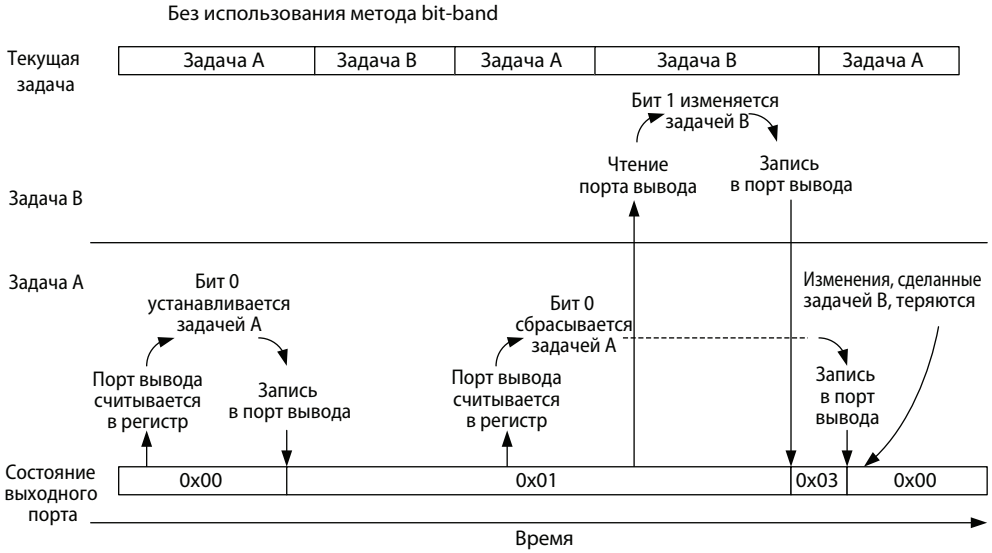


Рис. 5.10. *Потеря данных при изменении разными задачами содержимого разделяемой ячейки памяти.*

И точно так же использование метода bit-band позволяет обеспечить полную независимость битовых операций, выполняемых разными задачами. Это гарантирует отсутствие конфликта по данным (**Рис. 5.11**).

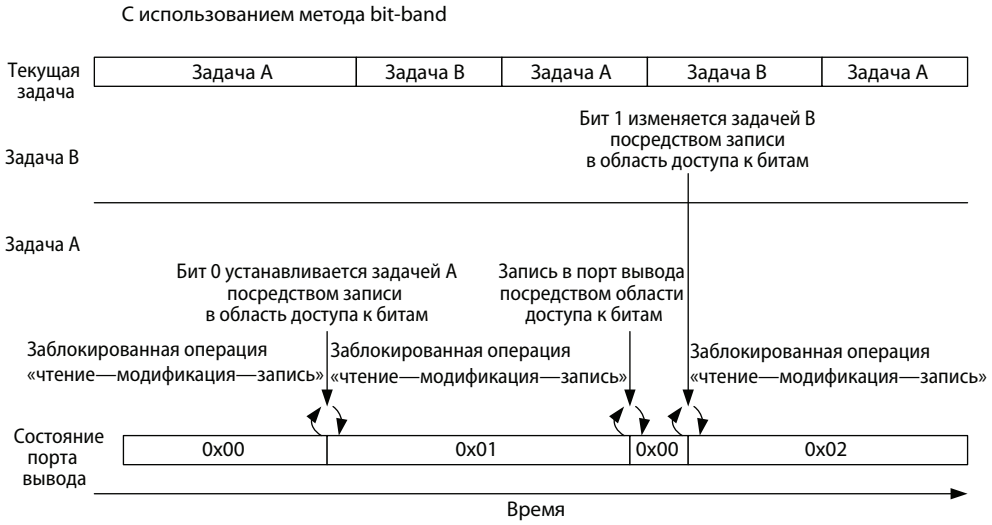


Рис. 5.11. *Предотвращение потери данных благодаря использованию метода bit-band.*

Метод bit-band может применяться не только для ввода/вывода данных, но и для хранения и обработки булевых данных в области СОЗУ. В частности, для экономии памяти несколько булевых переменных можно «упаковать» в одно слово данных. И в то же время, используя область доступа к битам, каждым битом можно будет оперировать независимо от остальных.

И в заключение пара замечаний для разработчиков систем на кристалле. При создании устройства, поддерживающего побитовый доступ с использованием метода bit-band, адрес области памяти этого устройства должен располагаться в области хранения битов. При этом необходимо контролировать сигнал блокировки (HMASTLOCK) от интерфейса шины AHB, чтобы при выполнении заблокированных пересылок изменение содержимого регистров устройства, доступных для записи, осуществлялось бы только шиной.

5.5.2. Битовые операции с данными разной разрядности

При выполнении битовых операций могут использоваться не только 4-байтные, но также двухбайтные и однобайтные пересылки. Скажем, при обращении к области доступа к битам с помощью однобайтных команд (`LDRB`/`STRB`) итоговая пересылка в/из области хранения битов будет также однобайтной. То же справедливо и для двухбайтных пересылок (`LDRH`/`STRH`). Тем не менее, даже если размер пересылки, осуществляемой при обращении к области доступа к битам, не равен слову, адрес всё равно должен быть выровнен на границу слова.

5.5.3. Битовые операции в Си-программах

В большинстве компиляторов с языка Си отсутствует поддержка битовых операций с применением метода bit-band. В частности, компиляторы не понимают, как к одной и той же ячейке памяти можно обращаться, используя два разных адреса. Также они не могут знать, что при адресации области доступа к битам в действительности производится обращение только к младшему биту ячейки памяти. Чтобы воспользоваться в Си-программах возможностями, обеспечиваемыми методом bit-band, можно объявить пару указателей — на адрес в области хранения битов и на соответствующий ему адрес в области доступа к битам. Например:

```
#define DEVICE_REG0      *((volatile unsigned long *) (0x40000000))
#define DEVICE_REG0_BIT0 *((volatile unsigned long *) (0x42000000))
#define DEVICE_REG0_BIT1 *((volatile unsigned long *) (0x42000004))
...
DEVICE_REG0 = 0xAB; // Обращение к регистру периферийного устройства
                    // с использованием обычной адресации
...
DEVICE_REG0 = DEVICE_REG0 | 0x2; // Установка бита 1 без использования
                                 // метода bit-band
...
DEVICE_REG0_BIT1 = 0x1; // Установка бита 1 с использованием метода bit-band
                        // (обращение к адресу в области доступа к битам)
```

Можно упростить использование бит-адресуемых областей памяти, написав пару макроопределений. Один из макросов будет вычислять адрес в области доступа к битам на основе адреса слова из области хранения битов и номера бита, а второй — преобразовывать значение адреса ячейки памяти в указатель:

```
// Вычисляет адрес в области хранения битов, используя адрес
// в бит-адресуемой области и номер бита
#define BITBAND(addr,bitnum) ((addr & 0xF0000000) + 0x2000000 +
                              ((addr & 0xFFFFF)<<5) + (bitnum <<2))
// Преобразует адрес в указатель
#define MEM_ADDR(addr) *((volatile unsigned long *) (addr))
```

Перепишем код предыдущего примера, используя эти макросы:

```
#define DEVICE_REG0 0x40000000
#define BITBAND(addr,bitnum) ((addr & 0xF0000000) + 0x02000000 +
                              ((addr & 0xFFFFF)<<5) + (bitnum<<2))
#define MEM_ADDR(addr) *((volatile unsigned long *) (addr))
...
MEM_ADDR(DEVICE_REG0) = 0xAB; // Обращение к регистру периферийного устройства
                              // с использованием обычной адресации
...
// Установка бита 1 без использования метода bit-band
MEM_ADDR(DEVICE_REG0) = MEM_ADDR(DEVICE_REG0) | 0x2;
...
// Установка бита 1 с использованием метода bit-band
MEM_ADDR(BITBAND(DEVICE_REG0,1)) = 0x1;
```

Обратите внимание, что при использовании метода bit-band переменные, размещаемые в бит-адресуемой области, должны быть объявлены как `volatile`. Компиляторы ведь не знают, что к одним и тем же данным можно обращаться по двум различным адресам, а применение модификатора `volatile` позволяет гарантировать, что при каждом обращении к такой переменной процессор будет обращаться непосредственно к ячейке памяти, а не к локальной копии этой переменной.

Для поддержки бит-адресуемой памяти в компилятор ARM был добавлен новый атрибут для переменных `__attribute__((bit-band))` и новый ключ командной строки `__bitband` (см. [6]). Пакет ARM RVDS начал поддерживать эти расширения языка, начиная с версии 4.0, а пакет Keil MDK-ARM — с версии 3.8. В руководстве по применению компилятора ARM RealView Compiler Tools [7] вы найдёте дополнительные примеры реализации обращений к бит-адресуемой памяти с использованием макросов.

5.6. Обращения к невыровненным данным

Процессор Cortex-M3 поддерживает пересылку невыровненных данных за одно обращение к памяти. Как известно, классические процессоры ARM, такие как ARM7/ARM9/ARM11, поддерживают обращения только к выровненным данным. Это означает, что при обращении к слову два младших бита адреса должны быть сброшены в 0, а при обращении к полуслову младший бит адреса должен быть сброшен в 0. Например, 32-битное значение может располагаться по адресу

0x1000 или 0x1004, но никак не по адресу 0x1001, 0x1002 или 0x1003. Аналогично, 16-битное значение может иметь адрес 0x1000 или 0x1002, но не 0x1001.

Что же собой представляет невыровненная пересылка? Ответ на этот вопрос можно получить из **Рис. 5.12**…**5.16**. При использовании 32-битной памяти невыровненная пересылка производится при выполнении любой операции чтения/записи слова, адрес которого не кратен 4 (**Рис. 5.12**…**5.14**). Точно так же, невыровненной является пересылка полуслова, адрес которого не кратен двум (**Рис. 5.15**, **5.16**).

Рис. 5.12. *Невыровненные данные, пример 1.*

Рис. 5.13. *Невыровненные данные, пример 2.*

Рис. 5.14. *Невыровненные данные, пример 3.*

Рис. 5.15. *Невыровненные данные, пример 4.*

Рис. 5.16. *Невыровненные данные, пример 5.*

Любые однобайтные пересылки в процессоре Cortex-M3 являются выровненными, поскольку минимальная разница между двумя адресами равна одному байту.

В процессоре Cortex-M3 невыровненные пересылки осуществляются обычными командами обращения к памяти, такими как `LDR`, `LDRH`, `STR` и `STRH`. При этом имеются некоторые ограничения:

- невыровненные пересылки не поддерживаются командами групповой загрузки/сохранения;
- операции со стеком (команды `PUSH`/`POP`) должны быть выровнены;

- команды монопольного доступа, такие как LDREX или STREX, должны обращаться только к выровненным данным; в противном случае, будет возникать исключение Usage Fault;
- невыровненные пересылки не поддерживаются при побитовых операциях с использованием метода bit-band. Если вы всё же попытаетесь так сделать, то результат будет непредсказуемым.

Любое обращение к невыровненным данным преобразуется интерфейсом шины в последовательность из нескольких выровненных пересылок, причём это преобразование является абсолютно прозрачным для пользователя. Поскольку при любом обращении к невыровненным данным в действительности осуществляется несколько пересылок, то для выполнения такой операции требуется большее число тактов, что в ряде случаев может оказаться нежелательным. Следовательно, для достижения максимальной производительности лучше всё же выравнивать данные соответствующим образом.

Кроме того, можно сконфигурировать контроллер NVIC так, чтобы при обращении к невыровненным данным возникало исключение. Для этого необходимо установить бит UNALIGN_TRP регистра управления конфигурацией CCR (0xE000ED14) в 1. В таком случае при каждой попытке обращения к невыровненным данным процессор будет генерировать исключение Usage Fault. Указанная возможность используется на этапе разработки и отладки ПО для того, чтобы отслеживать формирование программой невыровненных пересылок.

5.7. Монопольный доступ

Вы могли заметить, что в наборе команд процессора Cortex-M3 отсутствует команда SWP, которая в традиционных процессорах ARM, таких как ARM7TDMI, применялась для реализации семафоров. В новом процессоре для этих же целей используются команды монопольного доступа. Впервые поддержка операций монопольного доступа появилась в архитектуре v6 (в частности, в процессоре ARM1136).

Семафоры обычно используются для управления доступом к разделяемым ресурсам. Если разделяемый ресурс может обслуживать только одного клиента или процесс, то такой семафор также называется семафором взаимного исключения, или мьютексом (от *англ.* MUTual EXclusion). В таких случаях при использовании ресурса одним из процессов данный ресурс блокируется этим процессом и не может обслуживать другие процессы до тех пор, пока не будет разблокирован. Для реализации мьютекса требуется одна ячейка памяти, которая задействуется в качестве флага блокировки, указывающего, заблокирован соответствующий разделяемый ресурс каким-либо процессом или нет. Если процесс или приложение хотят задействовать такой ресурс, то они должны сначала проверить его состояние. Если ресурс не используется, то процесс или приложение устанавливает флаг, показывая, что ресурс теперь заблокирован. В традиционных процессорах ARM обращения к флагу блокировки осуществлялись командой SWP. Эта команда обеспечивала атомарность операций записи и чтения, что позволяло исключить одновременную блокировку ресурса двумя процессами.

В новых процессорах ARM операции чтения/записи могут осуществляться по разным шинам. Это делает невозможным использование команды `SWP` для организации атомарного доступа к памяти, поскольку операции чтения и записи при выполнении блокированной пересылки должны осуществляться по одной и той же шине. Соответственно, блокированные пересылки заменяются операциями монопольного доступа. Концепция, лежащая в основе таких операций, достаточно проста и при этом отличается от концепции команды `SWI`: обращения к ячейке памяти, в которой расположен семафор, допускаются со стороны другого ведущего шины или другого процесса, выполняемого тем же процессором (**Рис. 5.17**).

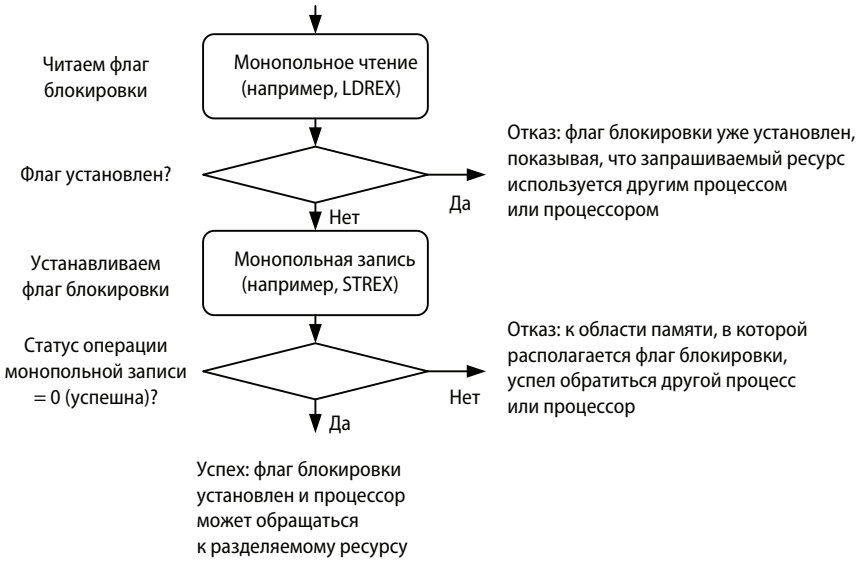


Рис. 5.17. *Команды монопольного доступа и мьютексы.*

Чтобы обеспечить корректное выполнение операций монопольного доступа в многопроцессорной среде, требуется дополнительный аппаратный модуль, называемый монитором монопольного доступа. Этот монитор контролирует обращения к разделяемым ячейкам памяти и сообщает процессору об успешном выполнении операций. В интерфейсах шин процессора также предусмотрены дополнительные сигналы управления[1)] этим монитором, которые используются для индикации пересылок, осуществляемых в монопольном режиме.

Если между операциями монопольного чтения и монопольной записи к памяти обратится другой ведущий шины, то при попытке процессора выполнить монопольную запись монитор передаст по шине сообщение об отказе в монопольном доступе. В итоге статус выполнения операции монопольной записи окажет-

[1)] Сигналы монопольного доступа имеются в системной шине (сигналы EXREQS и EXRESPS) и шине D-Code (сигналы EXREQD и EXRESPD) процессора Cortex-M3. Шина I-Code, используемая для выборки команд, не может генерировать транзакции монопольного доступа.

ся равным 1. В случае неудачной попытки монопольной записи монитор также блокирует запись по адресу, используемому для монопольного доступа.

В процессоре Cortex-M3 предусмотрено три команды монопольного чтения — LDREX (слово), LDREXB (байт), LDREXH (полуслово) и три команды монопольной записи — STREX (слово), STREXB (байт), STREXH (полуслово). Эти команды имеют следующий синтаксис:

```
LDREX <Rxf>, [Rn, #offset]
STREX <Rd>, <Rxf>,[Rn, #offset]
```

В регистре *Rd* возвращается статус выполнения монопольной записи (0 — успех, 1 — неудача).

Пример использования команд монопольного доступа можно найти в Главе 10. Для вызова команд монопольного доступа из программ на языке Си можно задействовать встроенные функции из CMSIS-совместимых библиотек, предоставляемых производителями микроконтроллеров: __LDREX, __LEDEXH, __LDREXB, __STREX, __STREXH, __STREXB. Более подробно эти функции описаны в Приложении Ж.

При осуществлении монопольного доступа внутренние буферы записи интерфейсов шин процессора Cortex-M3 не используются, даже если модуль MPU определяет указанную область памяти как буферируемую. Это гарантирует, что информация о семафоре, хранящаяся в физической памяти, всегда будет актуальной и непротиворечивой для разных контроллеров шин. Разработчики систем на кристалле, использующие процессор Cortex-M3 в многопроцессорных системах, должны обеспечить непротиворечивость данных при осуществлении пересылок монопольного доступа.

5.8. Порядок расположения байтов

Собственно процессор Cortex-M3 поддерживает как прямой, так и обратный порядок байтов. Однако тип памяти, используемый конкретным микроконтроллером, зависит и от остальных его компонентов (шинной инфраструктуры, контроллеров памяти, периферии и т.п.). Не забудьте внимательно изучить справочную документацию на применяемый микроконтроллер, прежде чем приступать к разработке программного обеспечения. В большинстве случаев в микроконтроллерах с процессором Cortex-M3 используется *прямой порядок байтов* (little endian). В этом режиме первый байт слова данных располагается в младшем байте 32-битной ячейки памяти (**Табл. 5.4**).

Таблица 5.4. **Процессор Cortex-M3, прямой порядок байтов (little endian) — данные в памяти**

Адрес	Биты 31…24	Биты 23…16	Биты 15…8	Биты 7…0
0x1003…0x1000	Байт — 0x1003	Байт — 0x1002	Байт — 0x1001	Байт — 0x1000
0x1007…0x1004	Байт — 0x1007	Байт — 0x1006	Байт — 0x1005	Байт —0x1004
…	Байт — 4xN+3	Байт — 4xN+2	Байт — 4xN+1	Байт — 4xN

Также встречаются микроконтроллеры, использующие *обратный порядок байтов* (big endian). В таких микроконтроллерах первый байт слова данных располагается в старшем байте 32-битной ячейки памяти (**Табл. 5.5**).

Таблица 5.5. **Процессор Cortex-M3, обратный порядок байтов (big endian) — данные в памяти**

Адрес	Биты 31…24	Биты 23…16	Биты 15…8	Биты 7…0
0x1003…0x1000	Байт — 0x1000	Байт — 0x1001	Байт — 0x1002	Байт — 0x1003
0x1007…0x1004	Байт — 0x1004	Байт — 0x1005	Байт — 0x1006	Байт — 0x1007
…	Байт — 4xN	Байт — 4xN+1	Байт — 4xN+2	Байт — 4xN+3

В процессоре Cortex-M3 по-другому определяется обратный порядок байтов, нежели в процессоре ARM7. Если в процессоре ARM7TDMI используется формат *с неизменным расположением слов* (word-invariant big endian), обозначаемый в документации ARM как BE-32, то в процессоре Cortex-M3 — формат *с неизменным расположением байтов* (byte-invariant big endian), обозначаемый в документации как BE-8 (этот формат поддерживается архитектурами ARM v6 и v7). С точки зрения расположения байтов в памяти, эти форматы полностью идентичны — отличается только использование байтовых трактов шины при пересылке данных (**Табл. 5.6** и **5.7**).

Таблица 5.6. **Процессор Cortex-M3 (обратный порядок с неизменным расположением байтов, BE-8) — данные на шине AHB**

Адрес	Биты 31…24	Биты 23…16	Биты 15…8	Биты 7…0
0x1000, слово	Биты данных [7:0]	Биты данных [15:8]	Биты данных [23:16]	Биты данных [31:24]
0x1000, полуслово	—	—	Биты данных [7:0]	Биты данных [15:8]
0x1002, полуслово	Биты данных [7:0]	Биты данных [15:8]	—	—
0x1000, байт	—	—	—	Биты данных [7:0]
0x1001, байт	—	—	Биты данных [7:0]	—
0x1002, байт	—	Биты данных [7:0]	—	—
0x1003, байт	Биты данных [7:0]	—	—	—

Таблица 5.7. **Процессор ARM7TDMI (обратный порядок с неизменным расположением слов, BE-32) — данные на шине AHB**

Адрес	Биты 31…24	Биты 23…16	Биты 15…8	Биты 7…0
0x1000, слово	Биты данных [7:0]	Биты данных [15:8]	Биты данных [23:16]	Биты данных [31:24]
0x1000, полуслово	Биты данных [7:0]	Биты данных [15:8]	—	—
0x1002, полуслово	—	—	Биты данных [7:0]	Биты данных [15:8]
0x1000, байт	Биты данных [7:0]	—	—	—
0x1001, байт	—	Биты данных [7:0]	—	—
0x1002, байт	—	—	Биты данных [7:0]	—
0x1003, байт	—	—	—	Биты данных [7:0]

Обратите внимание, что при пересылке данных по шине AHB в режиме BE-8 используются те же байтовые тракты шины, что и в режиме с прямым порядком байтов. Однако байты данных внутри полуслова или слова располагаются в обратном порядке (**Табл. 5.8**).

***Таблица 5.8.* Процессор Cortex-M3, прямой порядок байтов — данные на шине AHB**

Адрес	Биты 31…24	Биты 23…16	Биты 15…8	Биты 7…0
0x1000, слово	Биты данных [31:24]	Биты данных [23:16]	Биты данных [15:8]	Биты данных [7:0]
0x1000, полуслово	—	—	Биты данных [15:8]	Биты данных [7:0]
0x1002, полуслово	Биты данных [15:8]	Биты данных [7:0]	—	—
0x1000, байт	—	—	—	Биты данных [7:0]
0x1001, байт	—	—	Биты данных [7:0]	—
0x1002, байт	—	Биты данных [7:0]	—	—
0x1003, байт	Биты данных [7:0]	—	—	—

В процессоре Cortex-M3 формат хранения данных фиксируется в момент выхода процессора из состояния сброса, после чего изменить его уже нельзя (динамическое изменение формата хранения данных не допускается и команда `SETEND` не поддерживается). При выборке команд всегда используется прямой порядок байтов, так же как и при обращении к данным, расположенным в пространстве управления системой (скажем, при обращении к модулям NVIC и FPB) и в диапазоне адресов внешней шины PPB (в диапазоне адресов от 0xE0000000 до 0xE00FFFFF данные всегда хранятся в формате с прямым порядком байтов).

Если ваша система на кристалле не поддерживает обратный порядок байтов, а одно или несколько из используемых периферийных устройств используют этот формат, то вы можете легко преобразовать данные из одного формата в другой с помощью специальных команд преобразования данных, имеющихся в процессоре Cortex-M3. В частности, для подобных преобразований часто используют команды `REV` и `REV16`.

ГЛАВА 6
ОСОБЕННОСТИ РЕАЛИЗАЦИИ CORTEX-M3

Данная глава предназначена, главным образом, для разработчиков систем на кристалле (SoC), которые хотели бы использовать процессор Cortex™-M3 в своих проектах. Обычным разработчикам знать тонкости реализации ядра Cortex™-M3, вообще говоря, нет никакой необходимости. Однако если вам действительно интересно, как устроен и как работает процессор Cortex-M3, то вы сможете почерпнуть из настоящей главы немало полезного.

6.1. Конвейер

В процессоре Cortex-M3 реализован трёхступенчатый конвейер. Обработка каждой команды в конвейере осуществляется в три этапа: выборка, декодирование и выполнение (**Рис. 6.1**).

Рис. 6.1. *Трёхступенчатый конвейер процессора Cortex-M3.*

Кто-то может возразить, что таких этапов на самом деле четыре, поскольку есть ещё этап взаимодействия конвейера с интерфейсом шины при обращении к памяти. Однако выполнение данного этапа происходит за пределами процессора, поэтому конвейер всё же считается трёхступенчатым.

Если понаблюдать за процессом выполнения программы, содержащей в основном 16-битные команды, то можно заметить, что выборка команд не всегда происходит на каждом такте. Это связано с тем, что процессор может выбирать до двух команд за один раз (32-битная шина), так что после выборки одной команды следующая тоже окажется в процессоре. В такой ситуации интерфейс шины процессора может либо попытаться осуществить выборку последующей команды, либо, если буфер команд полон, перейти в состояние ожидания. Для выполнения некоторых команд требуется несколько тактов; в этом случае работа конвейера приостанавливается.

При выполнении команды ветвления конвейер очищается, после чего процессор должен будет вновь заполнить его командами, начиная с того адреса, по которому был осуществлён переход. Однако процессор Cortex-M3 поддерживает несколько команд архитектуры v7-M, которые в ряде случаев позволяют заменить короткие переходы на условно выполняемые команды[1)].

Из-за наличия в процессоре конвейера, а также для обеспечения совместимости программ с набором команд Thumb® при чтении счётчика команд во время исполнения команды возвращаемое значение будет равно адресу команды плюс четыре. Если счётчик команд задействуется для формирования адреса при обращении к памяти, то будет использовано значение адреса команды, выровненное на границу слова и увеличенное на четыре. Это смещение постоянно и не зависит от сочетания 16-битных команд Thumb и 32-битных команд Thumb-2. Такой подход гарантирует совместимость обоих наборов команд.

В блоке выборки команд процессора предусмотрен специальный буфер (**Рис. 6.2**), который позволяет организовать очередь из предварительно выбранных команд. Этот буфер предотвращает останов конвейера при наличии в последовательности команд 32-битных команд Thumb-2, не выровненных на границу слова. Однако он не является дополнительной ступенью конвейера и, следовательно, не увеличивает штраф ветвления.

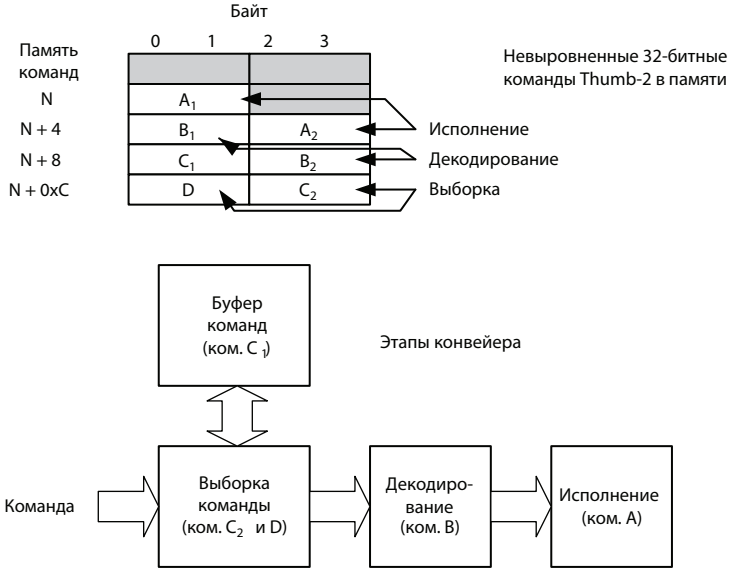


Рис. 6.2. *Использование буфера блока выборки команд для более эффективной обработки 32-битных команд.*

[1)] См. разделы Главы 4, посвящённые команде `IF-THEN` (`IT`).

6.2. Подробная блок-схема

В состав процессора Cortex-M3 входит не только процессорное ядро, но и различные компоненты управления системой, а также компоненты поддержки отладки (**Рис. 6.3**). Все эти компоненты соединены между собой шинами AHB и APB. Указанные шины являются составной частью усовершенствованной шинной архитектуры для микроконтроллеров (Advanced Microcontroller Bus Architecture — AMBA) [4].

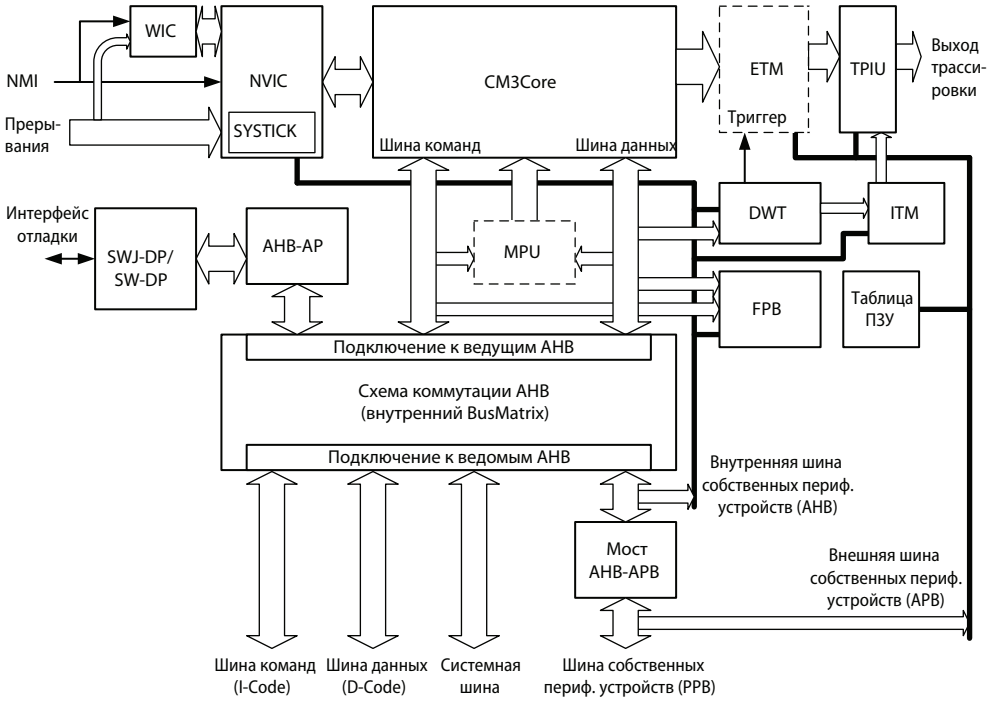


Рис. 6.3. *Блок-схема процессора Cortex-M3.*

Обратите внимание, что модули MPU, WIC и ETM являются опциональными блоками, которые могут быть включены в микроконтроллер на этапе его реализации. Некоторые новые компоненты перечислены в **Табл. 6.1**.

Таблица 6.1. **Сокращения и обозначения, использованные в блок-схеме**

Обозначение	Описание
CM3Core	Ядро центрального процессора Cortex-M3
NVIC	Контроллер вложенных векторных прерываний
SYSTICK	Системный таймер, который может использоваться операционными системами
WIC	Контроллер «пробуждающих» прерываний (опция)
MPU	Модуль защиты памяти (опция)
BusMatrix	Внутренняя схема коммутации шин AHB
Мост AHB-APB	Мост, преобразующий сигналы шины AHB в сигналы шины APB

Таблица 6.1. **Сокращения и обозначения, использованные в блок-схеме** *(продолжение)*

Обозначение	Описание
Интерфейс SW-DP/SWJ-DP	Интерфейс порта отладки Serial Wire/Serial Wire + JTAG. Связь с интерфейсом отладки осуществляется по протоколу Serial Wire или по стандартному протоколу JTAG (для SWJ-DP)
AHB-AP	Порт доступа AHB; преобразует команды от интерфейса SW/SWJ в транзакции шины AHB
ETM	Встроенная макроячейка трассировки; модуль, обеспечивающий возможность трассировки команд при отладке (опциональный)
DWT	Модуль просмотра и трассировки данных; модуль, реализующий поддержку точек наблюдения данных при отладке
ITM	Модуль трассировки Instrumentation Trace Macrocell
TPIU	Модуль интерфейса порта трассировки; интерфейсный модуль, осуществляющий передачу данных отладки внешнему аппаратному трассировщику
FPB	Модуль коррекции флэш-памяти и задания точки останова
Таблица ПЗУ	Небольшая таблица, содержащая информацию о конфигурации процессора

Процессор Cortex-M3 предлагается в виде процессорной подсистемы (**Рис. 6.3**). Собственно ядро ЦПУ тесно связано с контроллером прерываний NVIC и различными блоками поддержки отладки:

- *CM3Core* — ядро Cortex-M3, содержащее регистры, АЛУ, тракт данных и интерфейс шины.
- *NVIC* — встроенный контроллер прерываний. Число поддерживаемых прерываний определяется конкретным изготовителем микросхем. Контроллер NVIC тесно связан с ядром ЦПУ и содержит несколько регистров управления системой. Этот контроллер поддерживает обработку вложенных прерываний, что значительно упрощает обработку таких прерываний процессором. Кроме того, контроллер NVIC является контроллером векторных прерываний. Это означает, что при возникновении прерывания можно сразу же перейти к процедуре обработки этого прерывания (нет необходимости использовать общий обработчик для определения типа прерывания).
- *SYSTICK* — системный таймер, представляющий собой обычный вычитающий счётчик, который может использоваться для генерации прерываний через равные интервалы времени, даже во время нахождения системы в спящем режиме. Наличие такого таймера значительно упрощает перенос операционных систем на различные микроконтроллеры с процессором Cortex-M3, поскольку исключает необходимость изменения кода, отвечающего за поддержку системного таймера ОС. Таймер SYSTICK входит в состав контроллера NVIC.
- *WIC* — модуль, взаимодействующий с контроллером прерываний NVIC и при этом отделённый от основной схемы процессора. Этот модуль предназначен для вывода системы из спящего режима при появлении прерывания даже в том случае, если процессор (включая контроллер NVIC) полностью останов-

лен или отключён. Модуль WIC появился во второй ревизии ядра Cortex-M3 и является опциональным.

- *MPU* — модуль MPU является опциональным. Это означает, что в каких-то реализациях процессора Cortex-M3 этот модуль может присутствовать, а в каких-то нет. При наличии модуля MPU он может использоваться для защиты содержимого памяти, например, запрещая запись в определённые области памяти или предотвращая доступ пользовательских программ к данным привилегированных приложений.
- *BusMatrix* — этот модуль является «сердцем» системы внутренних шин процессора Cortex-M3. Он представляет собой схему коммутации шин AHB, обеспечивающую возможность одновременной пересылки данных по различным шинам при обращении обоих ведущих шины к разным областям памяти. Модуль BusMatrix также осуществляет дополнительное управление передачей данных, обеспечивая, помимо всего прочего, буфер записи, а также осуществляет поддержку бит-ориентированных операций (bit-band).
- *AHB-APB* — мост между шинами AHB и APB; используется для подключения различных APB-устройств, таких как компоненты отладки, к шине встроенных периферийных устройств процессора Cortex-M3. Кроме того, конструкция процессора позволяет производителям микросхем подключать дополнительные APB-устройства к внешней шине встроенных периферийных устройств (PPB) с помошью этой шины APB.

Остальные компоненты, изображённые на блок-схеме, предназначены для поддержки отладки и, как правило, не используются прикладными программами:

- *SW-DP/SWJ-DP* — порт отладки Serial-Wire (SW-DP)/порт отладки Serial-Wire и JTAG (SWJ-DP); работает совместно с портом доступа AHB (AHB-AP), который позволяет внешним отладчикам формировать транзакции на шине AHB для управления процессом отладки. В самом процессорном ядре Cortex-M3 нет тракта сканирования JTAG; управление большинством отладочных функций осуществляется регистрами NVIC по шине AHB. Модуль SWJ-DP поддерживает протоколы Serial-Wire и JTAG, тогда как модуль SW-DP поддерживает только протокол Serial Wire.
- *AHB-AP* — модуль AHB-AP обеспечивает доступ ко всей памяти процессора Cortex-M3 посредством нескольких регистров. Этот блок управляется модулем SW-DP/SWJ-DP по отладочному интерфейсу общего назначения, который называется *портом доступа к средствам отладки* (DAP). Для выполнения своих функций внешний аппаратный отладчик должен обращаться к модулю AHB-AP через SW-DP/SWJ-DP, чтобы генерировать требуемые транзакции на шине AHB.
- *ETM* — модуль ETM является опциональным компонентом, обеспечивающим возможность трассировки команд, поэтому в некоторых устройствах на базе Cortex-M3 поддержка трассировки команд в реальном времени может отсутствовать. Данные трассировки выводятся в порт трассировки через модуль

TPIU. Регистры управления ETM отображены на адресное пространство памяти процессора и могут управляться отладчиком через интерфейс DAP.

- *DWT* — модуль DWT реализует функции точек наблюдения данных. При обнаружении равенства адреса или величины переменной заданному значению этот модуль может генерировать события точки наблюдения, используемые для активации отладчика, генерации информации о трассировке данных или для активации модуля ETM.
- *ITM* — модуль ITM может использоваться двумя способами. Во-первых, программное обеспечение может напрямую выполнять запись в этот модуль для вывода информации в порт TPIU. Во-вторых, по событию совпадения от модуля DWT в модуле ITM могут генерироваться пакеты с информацией о трассировке данных для их вывода в выходной поток.
- *TPIU* — модуль TPIU обеспечивает взаимодействие с внешними аппаратными устройствами трассировки. Трассировочная информация, являющаяся внутренней для процессора Cortex-M3, имеет вид пакетов, передаваемых по усовершенствованной шине трассировки (Advanced Trace Bus — ATB), а модуль TPIU переформатирует данные таким образом, чтобы они могли быть считаны внешним устройством.
- *FPB* — модуль FPB используется для реализации функций Flash Patch и задания точки останова. Функция Flash Patch означает, что при обращении центрального процессора по некоторому адресу операция может быть переадресована в другую область памяти, в результате будет выбрано другое значение. Также совпадение адреса может быть использовано для генерации события точки останова.
- *Таблица ПЗУ* — это небольшая таблица, в которой содержится информация о распределении памяти для различных системных устройств и компонентов отладки. Средства отладки используют эту таблицу для определения адресов, по которым расположены компоненты отладки. В большинстве случаев применяется стандартная карта памяти, описанная в [1]. Однако поскольку некоторые компоненты отладки являются опциональными и, кроме того, возможно использование дополнительных компонентов, отдельные производители могут реализовывать отладочные возможности своих микросхем в соответствии со своими предпочтениями. В этом случае таблица ПЗУ должна быть скорректирована таким образом, чтобы программные средства отладки смогли определить реальное распределение карты памяти и, соответственно, имеющиеся в устройстве компоненты отладки.

6.3. Интерфейсы шин в процессоре Cortex-M3

Если только вы не занимаетесь разработкой систем на кристалле с использованием процессора Cortex-M3, то вряд ли когда-нибудь сможете напрямую использовать сигналы интерфейсов шин, описанных в этом разделе. Как правило, изготовители микросхем подключают все сигналы шин к блокам памяти и периферийным устройствам. И лишь в очень редких случаях изготовители решают

соединить системную шину с мостом, позволяющим подключать к устройству внешние системы шин. Интерфейсы шин процессора Cortex-M3 базируются на протоколах AHB-Lite и APB, спецификация которых содержится в [4].

6.3.1. Шина I-Code

Шина I-Code — это 32-битная шина, использующая протокол AHB-Lite, которая предназначена для выборки команд из памяти в диапазоне адресов от 0x00000000 до 0x1FFFFFFF. Выборка производится 32-битными словами, даже в случае 16-битных команд Thumb. Соответственно, во время выполнения программы ЦПУ может одновременно выбирать до двух команд Thumb.

6.3.2. Шина D-Code

Шина D-Code является 32-битной шиной, использующей протокол AHB-Lite, которая предназначена для обращения к данным в диапазоне адресов от 0x00000000 до 0x1FFFFFFF. Несмотря на то что процессор Cortex-M3 поддерживает обращения к невыровненным данным, вы не сможете обнаружить таких пересылок на шине, поскольку интерфейс шины в процессорном ядре преобразует их в выровненные. Соответственно, устройства (такие как память), подключаемые к этой шине, должны поддерживать только выровненные пересылки согласно протоколу AHB-Lite (AMBA 2.0).

6.3.3. Системная шина

Системная шина — это 32-битная шина, использующая протокол AHB-Lite, которая предназначена для выборки команд и обращения к данным в диапазоне адресов от 0x20000000 до 0xDFFFFFFF и от 0xE0100000 до 0xFFFFFFFF. Аналогично шине D-Code, пересылки по системной шине всегда выравниваются на границу слова.

6.3.4. Внешняя шина PPB

Внешняя шина собственных периферийных устройств работает по протоколу APB. Эта шина предназначена для обращений к периферийным устройствам в диапазоне адресов от 0xE0040000 до 0xE00FFFFF. В связи с тем, что некоторые области памяти в указанном диапазоне адресов уже используются модулями TPIU, ETM и таблицей ПЗУ, для подключения дополнительных устройств к этой шине может задействоваться только участок памяти в диапазоне адресов от 0xE0042000 до 0xE00FF000. Пересылки по шине PBB всегда выравниваются на границу слова.

6.3.5. Шина DAP

Шина DAP — это 32-битная шина, поддерживающая расширенную версию спецификации APB. Эта шина предназначена исключительно для подключения модулей интерфейса отладки, таких как SWJ-DP или SW-DP. Использовать данную шину для других целей нельзя. Дополнительную информацию об интерфейсе шины DAP можно найти в Главе 15, а также в [3].

6.4. Другие интерфейсы процессора Cortex-M3

Помимо интерфейсов шин, в процессоре Cortex-M3 имеется ряд других интерфейсов, используемых для разных целей. Сигналы данных интерфейсов вряд ли окажутся на выводах микросхемы, поскольку их подключают, большей частью, к различным узлам SoC или же оставляют неподключёнными. Подробная информация об указанных сигналах содержится в [1]. Краткое описание некоторых из них приведено в **Табл. 6.2**.

***Таблица 6.2.* Некоторые сигналы интерфейсов**

Группа сигналов	Назначение
Межпроцессорный обмен (TXEV, RXEV)	Сигналы синхронизации задач, выполняющихся на разных процессорах
Сигналы перехода в спящий режим (SLEEPING, SLEEPDEEP)	Состояние режима пониженного энергопотребления для системы управления питанием
Сигналы внутреннего состояния (ETMINTNUM, ETMINTSTATE, CURRPRI)	Состояние прерывания, используются для работы модуля ETM и для отладки
Запрос сброса (SYSRESETREQ)	Выход запроса на сброс от контроллера NVIC
Состояние блокировки* и останова (LOCKUP, HALTED)	Показывают, что процессорное ядро перешло в состояние блокировки (из-за ошибок при выполнении обработчика исключения Hard Fault или обработчика немаскируемого прерывания) или состояние останова (для отладочных операций)
Формат хранения данных (ENDIAN)	Задаёт формат хранения многобайтных значений (прямой/обратный порядок байтов), используемый процессором Cortex-M3 после выхода из состояния сброса
Интерфейс ETM	Подключается к модулю ETM для трассировки команд
Интерфейс ATB модуля ITM	ATB представляет собой один из протоколов архитектуры отладки CoreSight компании ARM, предназначенный для передачи данных трассировки. В данном случае этот интерфейс обеспечивает вывод данных трассировки из модуля ITM процессора Cortex-M3, подключённого к модулю TPIU
*Более подробно состояние блокировки рассматривается в Главе 12.	

6.5. Внешняя шина PPB

Процессор Cortex-M3 имеет интерфейс внешней шины PPB. Данный интерфейс базируется на протоколе APB из спецификации AMBA версии 2.0 (для ревизий 0 и 1 процессора Cortex-M3) или версии 3.0 (для ревизии 2 процессора). Шина PPB предназначена для подключения системных устройств, не являющихся разделяемыми ресурсами, таких как компоненты отладки.

Данный интерфейс поддерживает использование CoreSight-совместимых компонентов отладки. В связи с этим он несколько отличается от интерфейса обычной шины APB тем, что имеет дополнительный сигнал, называемый

PADDR31, который указывает на источник пересылки. Если указанный сигнал равен 0, значит, пересылка была сгенерирована программой, выполняемой процессором. Если этот сигнал равен 1, значит, пересылка была сгенерирована аппаратными средствами отладки. Благодаря наличию данного сигнала периферийное устройство может быть спроектировано таким образом, чтобы его мог задействовать только отладчик или же чтобы при использовании устройства программными средствами была доступна лишь часть его функций.

Эта шина не предназначена для обычного использования, скажем для подключения периферийных устройств общего назначения. В принципе, ничто не мешает разработчику микросхем спроектировать и подключить к данной шине обычное периферийное устройство. Однако в дальнейшем пользователи такого устройства могут столкнуться с проблемами, связанными с особенностями управления доступом на привилегированном уровне, например при программировании устройства на пользовательском уровне или же при отделении устройства от остальных областей памяти посредством MMU.

Внешняя шина PPB не поддерживает обращения к невыровненным данным. Поскольку шина содержит 32 линии данных и базируется на протоколе APB, при разработке периферийных устройств, предназначенных для подключения к этой области памяти, необходимо предусмотреть выравнивание адресов всех регистров устройства на границу слова. Кроме того, при разработке устройства, допускающего обращения со стороны программного обеспечения, рекомендуется, чтобы все обращения к данному устройству были 32-битными. При работе с шиной PPB всегда используется прямой порядок байтов.

6.6. Типичная схема подключения процессора

Из-за того, что в процессоре Cortex-M3 имеется множество интерфейсов шин, можно легко запутаться, пытаясь понять, как соединить его с другими компонентами системы, такими как память или периферийные устройства. В качестве примера на **Рис. 6.4** приведена блок-схема простой системы на базе процессора Cortex-M3.

Поскольку обращения к области памяти Code могут осуществляться как по шине команд (при выборке команд), так и по шине данных (при обращении к данным), необходим некий переключатель шин AHB, называемый *BusMatrix*[1)], или же обычный мультиплексор шины AHB. При использовании компонента BusMatrix, входящего в состав пакета разработки AMBA[2)] (ADK), обращения к флэш-памяти и дополнительному СОЗУ (при его наличии) могут производиться по любой шине. В случае одновременного обращения обеих шин к одному и тому же устройству памяти запросы шины данных должны иметь более высокий приоритет — при этом обеспечивается бóльшая производительность.

[1)] Компонент BusMatrix, требуемый в данном случае, отличается от одноимённого компонента, входящего в состав процессора Cortex-M3 (**Рис. 6.4**). Внутренний BusMatrix спроектирован специально для использования в процессоре и отличается от стандартного компонента из состава пакета разработки AMBA (AMBA Design Kit — ADK).

[2)] Пакет ADK содержит набор компонентов для поддержки архитектуры AMBA и примеры готовых систем, написанные на VHDL/Verilog.

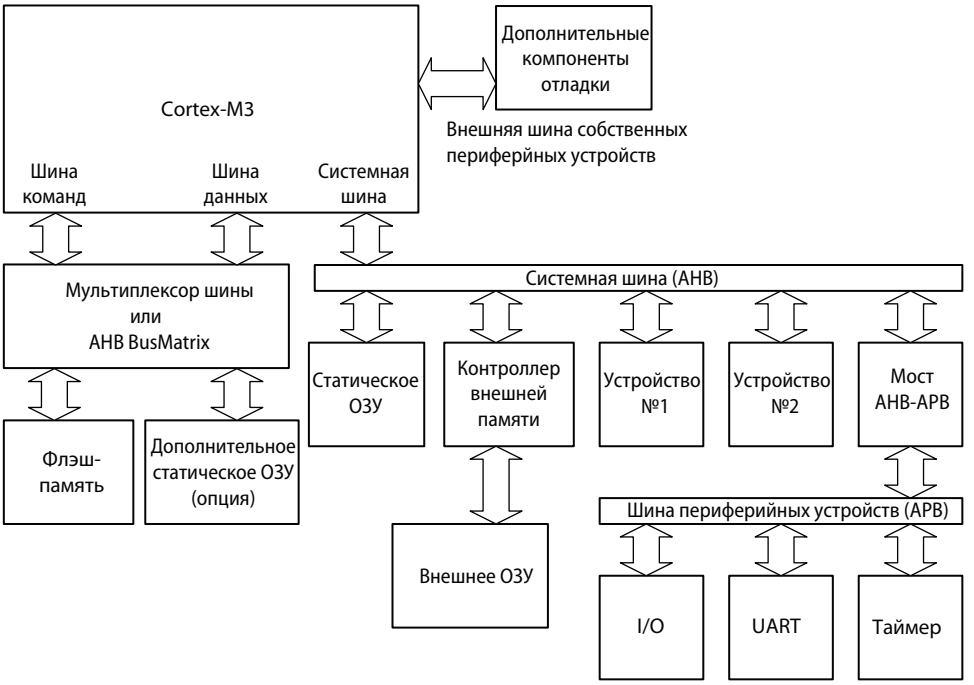


Рис. 6.4. *Блок-схема системы на базе процессора Cortex-M3.*

Компонент BusMatrix позволяет при обращении шин к разным устройствам памяти (например, при выборке команды из флэш-памяти по шине команд с одновременным чтением данных из дополнительного СОЗУ по шине данных) пересылать данные по обеим шинам одновременно. При использовании мультиплексора шины одновременная пересылка данных по обеим шинам будет невозможна, однако схема при этом будет иметь меньшие размеры. В обычных микроконтроллерах с ядром Cortex-M3 оперативная память подключается к системной шине.

Основной блок СОЗУ должен подключаться через интерфейс системной шины в диапазоне адресов, выделенных под оперативную память. Это даёт возможность обращаться к данным одновременно с выборкой команд. Кроме того, такое подключение СОЗУ позволяет работать с булевыми типами данных, используя побитовый доступ по методу bit-band.

В некоторых микроконтроллерах может присутствовать внешний интерфейс памяти. Это требует использования контроллера внешней памяти, поскольку вы не можете подключать внешние запоминающие устройства напрямую к шине AHB. Контроллер внешней памяти может быть подключён к системной шине процессора. Дополнительные AHB-совместимые устройства также могут быть легко подключены к системной шине без использования компонента BusMatrix.

Простые периферийные устройства могут подключаться к процессору через мост AHB-APB. Это позволяет использовать для обращения к ним более простой протокол APB.

Блок-схема, приведённая на **Рис. 6.4**, является всего лишь простым примером; разработчики микросхем могут реализовывать и другие варианты подключения к шинам процессора. Однако вам это не потребуется — единственное, что вам необходимо для разработки программного обеспечения, это информация о карте распределения памяти.

Компоненты, изображённые на блок-схеме, такие как BusMatrix, мост AHB-APB, контроллер памяти, интерфейс ввода/вывода, таймер и модуль универсального асинхронного приёмопередатчика (UART), предлагаются компанией ARM, а также другими компаниями, разрабатывающими IP-блоки. Поскольку в микроконтроллерах могут использоваться периферийные устройства разных производителей, при разработке программного обеспечения необходимо внимательно изучить справочную документацию на применяемый микроконтроллер.

6.7. Виды сброса и сигналы сброса

В системах с процессором Cortex-M3 используются разные виды сброса. Некоторые устройства могут поддерживать большее число разновидностей сброса по сравнению с другими — это определяется реализацией схемы сброса микроконтроллера или SoC (**Рис. 6.5**). Однако в любой системе имеется, по крайней мере, три вида сброса, перечисленные в **Табл. 6.3**.

***Таблица 6.3.* Виды сброса, имеющиеся в любой системе**

Вид сброса	Сигнал сброса процессора Cortex-M3	Описание
Сброс по включению питания	PORESETn	Сброс, выполняемый при включении питания устройства; сбрасывается ядро процессора, периферийные устройства и система отладки. Активируется при включении устройства
Сброс системы	SYSRESETn	Системный сброс; сбрасывается всё устройство, кроме системы отладки, в том числе ядро процессора, контроллер NVIC (за исключением регистров управления отладкой), модуль MPU, периферия. Активируется при включении устройства, а также по запросу от отладчика через регистр AIRCR контроллера NVIC
Сброс процессора	Бит VECTRESET в регистре AIRCR контроллера NVIC	Сброс процессорного ядра; сбрасывается весь процессор, кроме системы отладки, в том числе ядро процессора, контроллер NVIC (за исключением регистров управления отладкой) и модуль MPU. Активируется по запросу от отладчика через регистр AIRCR контроллера NVIC — этот вид сброса предназначен для использования отладчиком
Сброс JTAG	nTRST	Сигнал сброса от контроллера порта JTAG (только при наличии интерфейса JTAG)

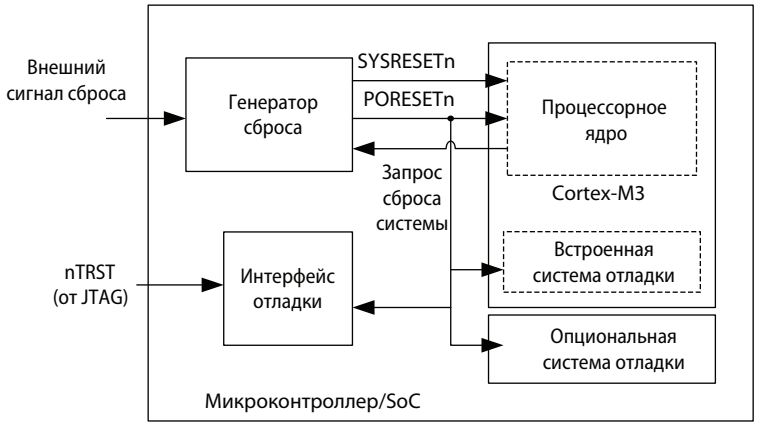


Рис. 6.5. *Внутренние сигналы сброса типичного микроконтроллера с ядром Cortex-M3.*

Более подробно сигналы сброса процессора Cortex-M3 описаны в [1]. Все сигналы сброса процессора подключаются к схеме сброса внутри микроконтроллера или SoC. Готовое устройство имеет всего один или два внешних сигнала сброса.

ГЛАВА 7
ИСКЛЮЧЕНИЯ

7.1. Типы исключений

Архитектура исключений, реализованная в процессоре Cortex-M3, поддерживает несколько системных исключений и большое число внешних прерываний. Каждое исключение имеет свой порядковый номер, при этом исключения с номерами от 1 до 15 являются системными, а с номерами 16 и более — соответствуют входам внешних прерываний. Большинство исключений имеют программируемый приоритет и лишь некоторые — фиксированный приоритет.

Кристаллы с процессором Cortex-M3 могут иметь разное количество входов внешних прерываний (от 1 до 240) и разное число уровней приоритета. Это позволяет разработчикам микросхем конфигурировать исходный код процессора в соответствии со своими нуждами.

Как уже говорилось, исключения с номерами от 1 до 15 являются системными исключениями (исключения с номером 0 не существует), которые перечислены в **Табл. 7.1**. Исключения с номерами 16 и более соответствуют входам внешних прерываний (**Табл. 7.2**).

Номер исключения, обрабатываемого в данный момент времени, содержится в регистре состояния прерывания IPSR процессора, а также в регистре управления и состояния прерывания ICSR контроллера NVIC (поле VECTACTIVE).

Обратите внимание, что в данной главе номера прерываний (скажем, прерывание №0) соответствуют номерам входов прерываний контроллера NVIC. В реальных микроконтроллерах или системах на кристалле номера входов внешних прерываний могут не соответствовать номерам входов прерываний контроллера. Например, несколько первых входов прерываний могут быть назначены внутренним периферийным устройствам, а выводы внешних прерываний могут быть подключены к следующей группе входов контроллера NVIC. Для уточнения нумерации прерываний необходимо обратиться к документации на конкретное устройство.

При возникновении разрешённого исключения, которое не может быть обработано немедленно (например, в том случае, когда выполняется процедура обработки прерывания с более высоким приоритетом или установлен регистр маски-

рования прерываний), данное исключение будет отложено (это не касается ряда системных исключений[1]).

Таблица 7.1. **Системные исключения**

Номер исключения	Тип исключения	Приоритет	Описание
1	Reset	–3 (Наивысший)	Сброс
2	NMI	–2	Немаскируемое прерывание (вход внешнего немаскируемого прерывания)
3	Hard Fault	–1	Любой отказ, если соответствующий обработчик не разрешён
4	MemManage Fault	Программируемый	Отказ системы управления памятью; нарушение правил доступа, заданных модулем MPU, или обращение по некорректному адресу
5	Bus Fault	Программируемый	Отказ шины; происходит при получении интерфейсом шины AHB сигнала ошибки от ведомого устройства на шине (также называется *отказом предвыборки*, если ошибка возникла при выборке команды, или *отказом данных*, если ошибка возникла при обращении к данным)
6	Usage Fault	Программируемый	Ошибка в программе или попытка обращения к сопроцессору (процессор Cortex-M3 не поддерживает сопроцессор)
7…10	Зарезервировано	—	—
11	SVCall	Программируемый	Вызов супервизора
12	Debug monitor	Программируемый	Исключение монитора отладки (точки останова, точки наблюдения или внешняя команда отладки)
13	Зарезервировано	—	—
14	PendSV	Программируемый	Запрос системной службы
15	SYSTICK	Программируемый	Системный таймер

Таблица 7.2. **Внешние прерывания**

Номер исключения	Тип исключения	Приоритет	Описание
16	IRQ #0	Программируемый	Внешнее прерывание №0
17	IRQ #1	Программируемый	Внешнее прерывание №1
…	…	…	…
255	IRQ #239	Программируемый	Внешнее прерывание №239

[1] В некоторых случаях описанный сценарий нарушается. Если при возникновении отказа соответствующий обработчик не может быть сразу же запущен из-за того, что выполняется обработчик исключения с более высоким приоритетом, то вместо требуемого обработчика отказа возможен запуск обработчика исключения Hard Fault (исключение отказа с наивысшим приоритетом). Более подробно этот вопрос мы рассмотрим чуть позже, когда будем говорить об исключениях отказов; исчерпывающая же информация, как обычно, содержится в [2].

Откладывание исключения означает, что запрос на его обработку будет храниться в специальном регистре до тех пор, пока не появится возможность обработки данного исключения. Такое поведение отличается от поведения традиционных процессоров ARM. Ранее компоненты схемы, которые генерировали запросы обычного/быстрого (IRQ/FIQ) прерываний, должны были удерживать данный запрос до начала его обработки. Теперь же, благодаря наличию в контроллере NVIC регистров, обеспечивающих работу с отложенными исключениями, возникшее прерывание будет обработано даже в том случае, если источник прерывания снимет свой сигнал запроса.

7.2. Приоритеты исключений

В процессоре Cortex-M3 очерёдность обработки исключений, а также сама возможность обработки конкретного исключения определяется приоритетом этого исключения. Исключения с более высоким приоритетом (меньшим значением уровня приоритета) могут прерывать обработку исключения, имеющего меньший приоритет (большее значение уровня приоритета), т.е. поддерживаются вложенные исключения/прерывания. Некоторые типы исключений (Reset, NMI и Hard Fault) имеют фиксированные значения приоритета, которые представлены отрицательными числами, отражая тот факт, что данные исключения обладают наивысшим приоритетом среди всех остальных. Прочие исключения имеют программируемые уровни приоритета.

Процессор Cortex-M3 поддерживает три фиксированных уровня с наивысшим приоритетом и до 256 уровней с программируемым приоритетом (до 128 уровней вложенности). Однако большинство микросхем с ядром Cortex-M3 имеют гораздо меньшее число уровней, например 8, 16, 32 и т.д. Требуемое число уровней задаётся разработчиком при проектировании микроконтроллера или системы на кристалле. Уменьшение числа уровней приоритета осуществляется отключением соответствующего числа младших битов регистров приоритета.

Например, если для задания уровня приоритета в устройстве используется только три бита, то формат регистра приоритета будет таким, как показано на **Рис. 7.1**.

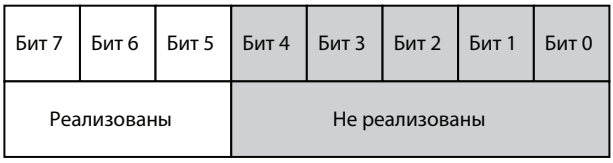


Рис. 7.1. *Регистр приоритета, в котором реализовано 3 бита.*

Поскольку биты 0…4 регистра не реализованы, при их чтении всегда возвращается ноль, а запись в эти биты игнорируется. При такой конфигурации процессора в нашем распоряжении будут следующие уровни приоритета: 0x00 (наивысший приоритет), 0x20, 0x40, 0x60, 0x80, 0xA0, 0xC0 и 0xE0 (наименьший приоритет).

Аналогичным образом, при использовании четырёх битов для задания уровня приоритета формат регистра приоритета будет соответствовать показанному на **Рис. 7.2**.

Рис. 7.2. *Регистр приоритета, в котором реализовано 4 бита.*

Чем больше битов регистров приоритета будет реализовано, тем больше окажется в нашем распоряжении уровней приоритета (**Рис. 7.3**). Однако увеличение числа используемых битов регистров приоритета влечёт за собой увеличение числа вентилей и, соответственно, потребляемой мощности. Для процессора Cortex-M3 минимальное число реализованных битов регистров приоритета равно трём (8 уровней).

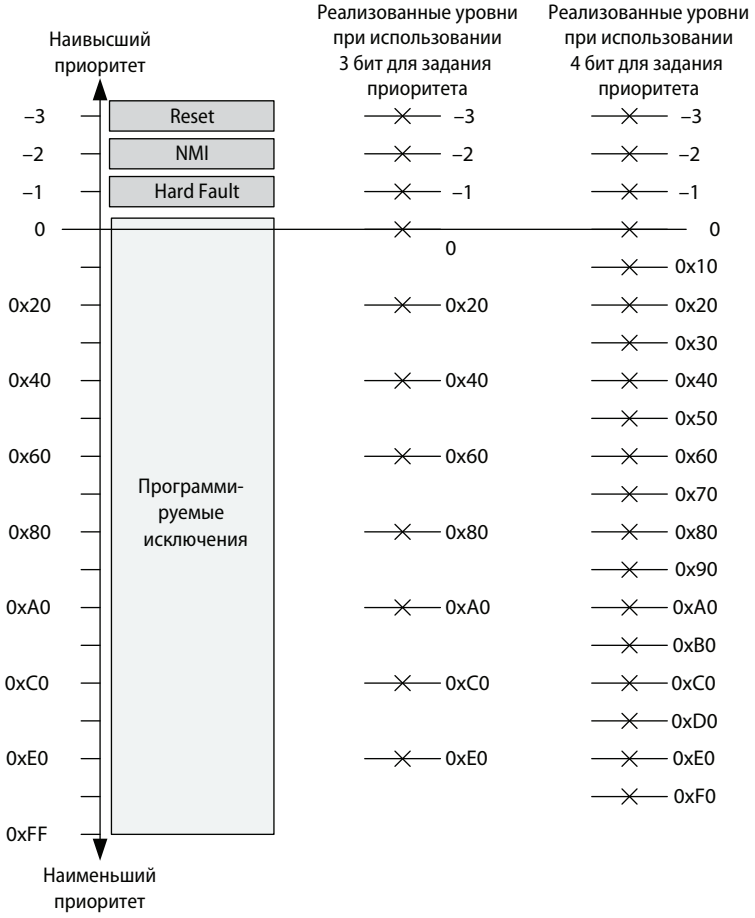


Рис. 7.3. *Возможные уровни приоритета при 3- и 4-битных регистрах приоритета.*

Отключение младших, а не старших битов регистров позволяет упростить перенос программы с одного устройства с ядром Cortex-M3 на другое. Очевидно,

что при таком подходе программа, написанная для устройства с 4-битными регистрами приоритета, нормально запустится на устройстве с 3-битными регистрами приоритета. Если бы вместо младших битов отключались старшие биты, то при переносе программы с одного устройства на другое могла бы произойти инверсия приоритетов. Предположим, что в программе используется прерывание IRQ #0 с уровнем приоритета 0x05 и прерывание IRQ #1 с уровнем приоритета 0x03, т.е. прерывание IRQ #1 имеет более высокий приоритет. Однако при удалении 2-го бита (старшего) уровень приоритета прерывания IRQ #0 стал бы равен 0x01, в результате чего это прерывание получило бы более высокий приоритет, чем IRQ #1.

Примеры возможных значений уровней приоритета для устройств с 3-, 5- и 8-битными регистрами приоритета приведены в **Табл. 7.3**.

***Таблица 7.3.* Доступные уровни приоритета для устройств с 3-, 5- и 8-битными регистрами приоритета**

Уровень приоритета	Тип исключения	Устройства с 3-битными регистрами приоритета	Устройства с 5-битными регистрами приоритета	Устройства с 8-битными регистрами приоритета
–3 (наивысший)	Reset	–3	–3	–3
–2	NMI	–2	–2	–2
–1	Hard Fault	–1	–1	–1
0, 1, … 0xFF	Исключения с программируемым уровнем приоритета	0x00, 0x20, … 0xE0	0x00, 0x08, … 0xF8	0x00, 0x01, 0x02, 0x03, … 0xFE, 0xFF

Наиболее внимательные читатели могут спросить: если регистры приоритета являются 8-битными, то почему поддерживается только 128 уровней вложенности? Это связано с тем, что содержимое 8-битных регистров делится на две части: *приоритет группы* и *субприоритет*.

Разбиение регистров приоритета исключений, имеющих программируемый приоритет, определяется полем PRIGROUP регистра AIRCR контроллера NVIC (**Табл. 7.4**). При этом старшая часть регистра (левые биты) определяет приоритет группы, а младшая (правые биты) — субприоритет (**Табл. 7.5**).

Значение *уровня приоритета группы* определяет возможность возникновения прерывания, если в данный момент уже выполняется обработчик другого прерывания. Значение *уровня субприоритета* используется только в случае одновременного возникновения нескольких исключений с одинаковыми значениями уровня приоритета группы. При этом исключение с более высоким субприоритетом (с меньшим значением уровня) будет обработано первым.

Из-за *группирования приоритетов* максимальное число битов, которое может использоваться для задания приоритета группы, равно 7, что и даёт нам уже упомянутые 128 уровней вложенности. При значении PRIGROUP = 7 все исключения с программируемым уровнем приоритета будут иметь один и тот же уровень. В результате ни одно из этих исключений не сможет быть прервано никаким другим исключением, кроме Hard Fault, NMI и Reset, которые имеют приоритеты –3, –2 и –1 соответственно.

Таблица 7.4. **Регистр управления прерываниями и сбросом AIRCR (0xE000ED0C)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:16	VECTKEY	R/W	—	Ключ доступа; при записи в регистр в этом поле должно быть записано значение 0x05FA, иначе запись будет игнорирована. При чтении регистра в данном поле возвращается значение 0xFA50
15	ENDIANNESS	R	—	Отображает формат хранения данных: 1 — обратный порядок байтов (BE-8), 0 — прямой порядок байтов; этот бит может быть изменён только в момент сброса
10:8	PRIGROUP	R/W	0	Установки группирования приоритетов
2	SYSRESETREQ	W	—	Формирует запрос на генерацию сигнала сброса
1	VECTCLRACTIVE	W	—	Очищает всю активную информацию о состоянии исключений; обычно используется при отладке или операционной системой для восстановления после системной ошибки (сброс является более безопасным методом)
0	VECTRESET	W	—	Сбрасывает процессор Cortex-M3 (за исключением компонентов отладки), однако не сбрасывает блоки, внешние по отношению к процессору

Таблица 7.5. **Поля приоритета группы и субприоритета в регистрах приоритета**

Значение PRIGROUP	Поле приоритета группы	Поле субприоритета
0	Биты [7:1]	Бит [0]
1	Биты [7:2]	Биты [1:0]
2	Биты [7:3]	Биты [2:0]
3	Биты [7:4]	Биты [3:0]
4	Биты [7:5]	Биты [4:0]
5	Биты [7:6]	Биты [5:0]
6	Бит [7]	Биты [6:0]
7	Отсутствует	Биты [7:0]

При выборе необходимых значений уровня приоритета группы и уровня субприоритета необходимо учитывать следующее:

- разрядность регистров приоритета;
- параметры группирования приоритетов (значение PRIGROUP).

Так, если разрядность регистров приоритета равна 3 (доступны биты 7…5), а в поле PRIGROUP будет записано 5, то в вашем распоряжении окажется 4 уровня приоритета группы (биты 7…6), в каждом из которых будет по два уровня субприоритета (бит 5).

Уровни приоритета, доступные при такой конфигурации регистров приоритета (**Рис. 7.4**), показаны на **Рис. 7.5**. Если же в поле PRIGROUP записать 0x01, то в нашем распоряжении окажется всего 8 уровней приоритета группы и ни одно-

го уровня субприоритета (биты 1:0 регистра, используемые для задания субприоритета, всегда равны 0). Формат регистров приоритета для такой конфигурации показан на **Рис. 7.6**, а возможные уровни приоритета — на **Рис. 7.7**.

Бит 7	Бит 6	Бит 5	Бит 4	Бит 3	Бит 2	Бит 1	Бит 0
Приоритет группы		Субприо-ритет	Не реализованы				

Рис. 7.4. *Определение полей в 3-битном регистре приоритета при PRIGROUP = 5.*

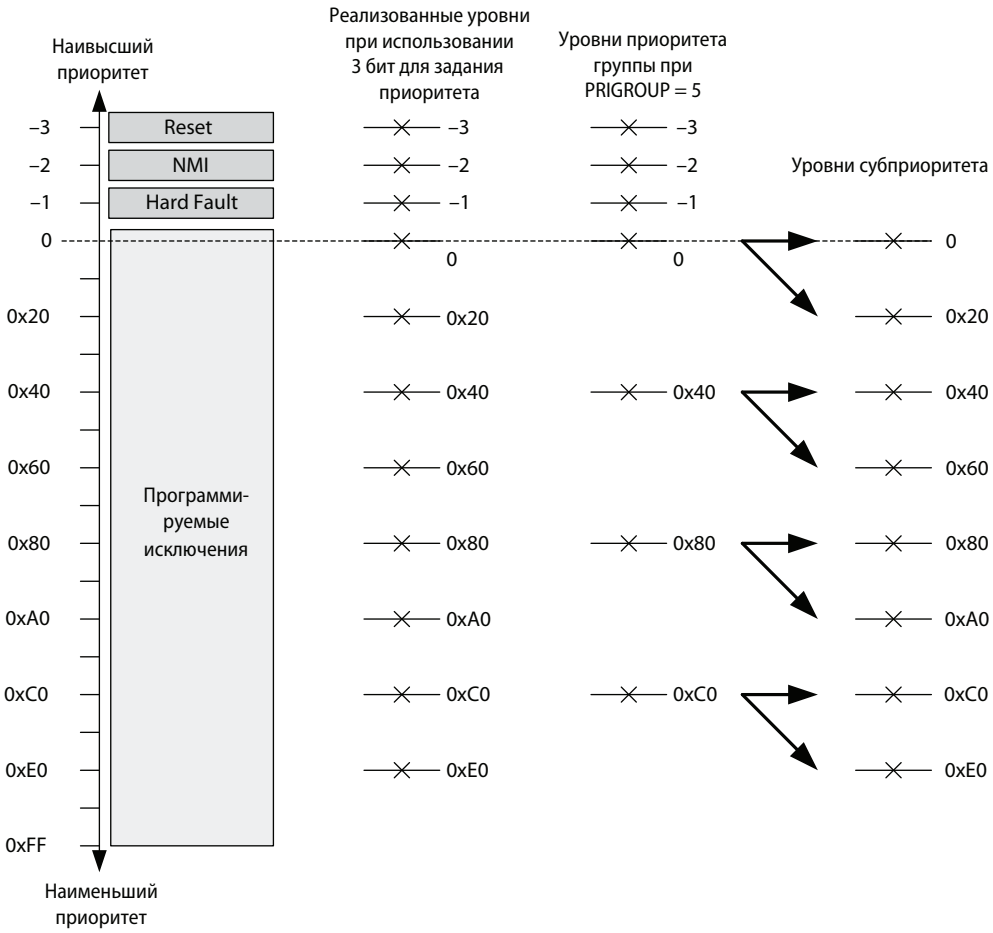


Рис. 7.5. *Возможные уровни приоритета при 3-битном регистре приоритета и PRIGROUP = 5.*

Бит 7	Бит 6	Бит 5	Бит 4	Бит 3	Бит 2	Бит 1	Бит 0
Приоритет группы [5:3]			Приоритет группы [2:0] (всегда 0)			Субприоритет [1:0] (всегда 0)	

Рис. 7.6. *Определение полей в 3-битном регистре приоритета при PRIGROUP = 1.*

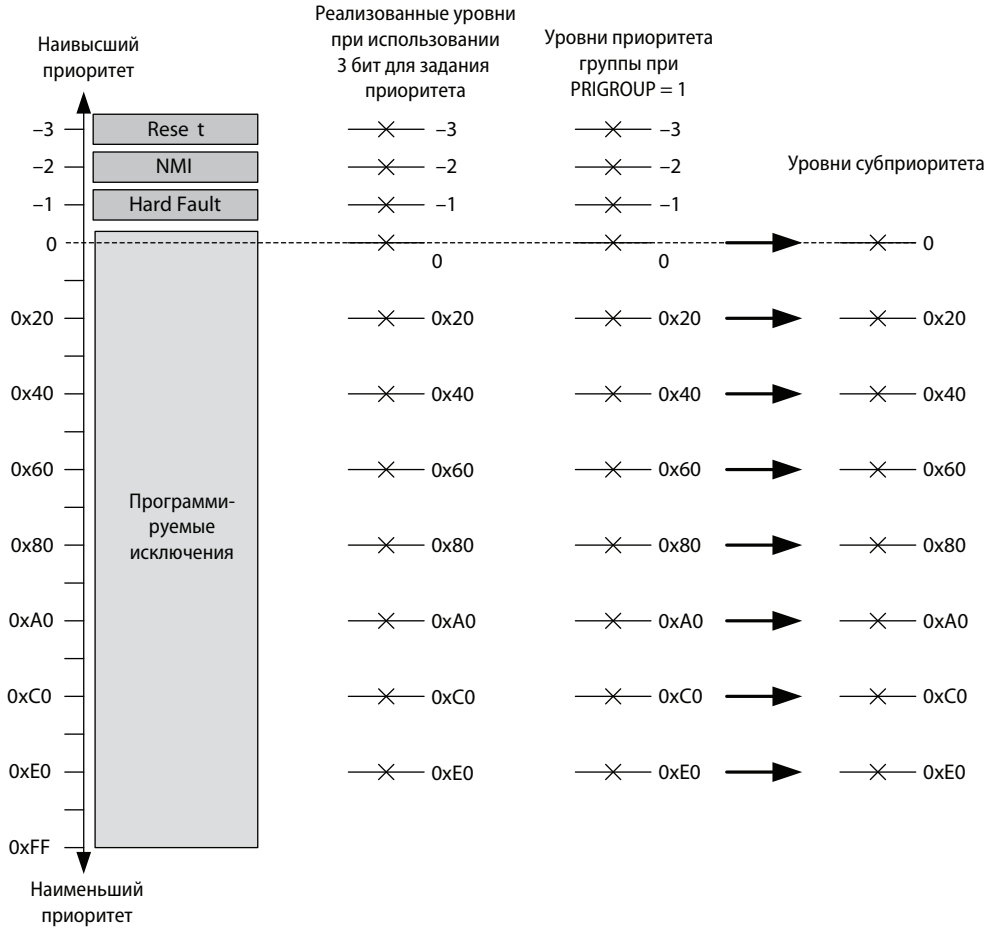


Рис. 7.7. *Возможные уровни приоритета при 3-битном регистре приоритета и PRIGROUP = 1.*

Если в регистрах приоритета устройства реализованы все 8 бит, то максимальное число уровней вложенности, которое может быть использовано в этом устройстве, будет равно 128 (при PRIGROUP = 0). Формат регистров приоритетов для данного случая приведён на **Рис. 7.8**.

Бит 7	Бит 6	Бит 5	Бит 4	Бит 3	Бит 2	Бит 1	Бит 0
Приоритет группы							Субприоритет

Рис. 7.8. *Определение полей в 8-битном регистре приоритета при PRIGROUP = 0.*

При одновременном появлении двух прерываний, имеющих одинаковые значения как уровня приоритета группы, так и уровня субприоритета, более высо-

кий приоритет будет иметь прерывание с меньшим порядковым номером (IRQ #0 имеет более высокий приоритет, нежели IRQ #1).

Чтобы не сбить случайно настройки группирования приоритетов прерываний, соблюдайте осторожность при записи в регистр AIRCR. В большинстве случаев после настройки группирования приоритетов трогать этот регистр нет никакой необходимости, кроме как для генерации сброса (см. **Табл. 7.4**).

7.3. Таблица векторов

При возникновении исключения процессору необходимо определить начальный адрес обработчика этого исключения. Данная информация хранится в таблице векторов, расположенной в памяти процессора. По умолчанию таблица векторов располагается, начиная с нулевого адреса, а адрес каждого вектора в указанной таблице равен порядковому номеру исключения, умноженному на 4 (**Табл. 7.6**).

Таблица 7.6. **Таблица векторов исключений после включения питания**

Адрес	Номер исключения	Значение (32-битное)
0x00000000	—	Начальное значение MSP
0x00000004	1	Вектор сброса (начальное значение счётчика команд)
0x00000008	2	Стартовый адрес обработчика NMI
0x0000000C	3	Стартовый адрес обработчика исключения Hard Fault
…	…	Стартовые адреса остальных обработчиков

Поскольку адрес 0x00 должен находиться в области загрузочного кода, он, как правило, соответствует флэш-памяти или ПЗУ, и его содержимое не может изменяться во время выполнения программы. Однако таблица векторов может быть перемещена по другому адресу в области кода или в области ОЗУ. В последнем случае мы получим возможность изменять обработчики исключений в процессе выполнения программы. Положение таблицы векторов в памяти определяется регистром смещения таблицы векторов VTOR контроллера NVIC, который расположен по адресу 0xE000ED08. Величина смещения должна быть выровнена на размер таблицы векторов и приведена к ближайшему большему значению, являющемуся степенью двойки. Например, если в процессоре используется 32 входа прерываний, то общее число исключений с учётом системных будет равно 32 + 16 = 48. Приводя это значение к значению, являющемуся степенью двойки, получаем число 64. Умножая результат на 4 (4 байта на вектор), получаем 256 (0x100). Таким образом, величина смещения таблицы векторов может быть равна 0x00, 0x100, 0x200 и т.д. Описание полей регистра VTOR приведено в **Табл. 7.7**.

Таблица 7.7. **Регистр смещения таблицы векторов VTOR (0xE000ED08)**

Биты	Обозначение	Тип	Значение после сброса	Описание
29	TBLBASE	R/W	0	Таблица расположена в области кода (0) или в области ОЗУ (1)
28:7	TBLOFF	R/W	0	Смещение таблицы относительно начала области кода или области ОЗУ

Если вы хотите динамически менять обработчики исключений, то в начале секции загрузчика должны располагаться, как минимум, следующие элементы:

- начальное значение основного указателя стека;
- вектор сброса;
- вектор NMI;
- вектор исключения Hard Fault.

Это требование связано с тем, что немаскируемое прерывание, как и исключение Hard Fault, может произойти при выполнении кода загрузчика. Остальные исключения не смогут возникнуть до тех пор, пока не будут разрешены.

После завершения процесса загрузки вы можете выделить часть области ОЗУ под таблицу векторов и переместить таблицу на новое место.

7.4. Входы прерываний и отложенная обработка прерываний

В этом разделе описывается поведение входов IRQ и отложенная обработка прерываний. Всё, сказанное в данном разделе, справедливо и для входа NMI, за исключением того, что обработка немаскируемого прерывания, как правило, запускается сразу же после его возникновения. Откладывание немаскируемого прерывания происходит только в следующих случаях:

- процессор уже выполняет обработчик NMI;
- процессор находится в состоянии останова, вызванного командой отладчика;
- процессор находится в состоянии блокировки из-за какого-либо серьёзного системного сбоя.

При активации входа прерывания это прерывание откладывается, что означает перевод прерывания в состояние ожидания обработки запроса процессором. Даже если источник снимет запрос прерывания, наличие такого состояния обеспечит последующий запуск обработчика этого прерывания в соответствии с заданным приоритетом. После запуска обработчика признак отложенного прерывания снимается автоматически. Данный процесс показан на **Рис. 7.9**.

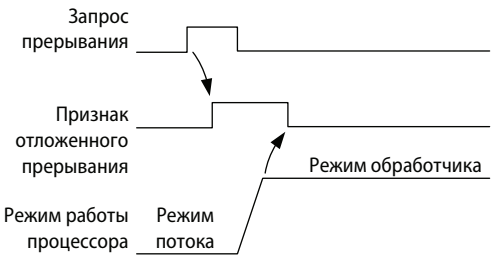


Рис. 7.9. *Отложенная обработка прерывания.*

Если же признак отложенного прерывания окажется сброшен до того, как процессор приступит к обработке этого прерывания (например, прерывание не было обслужено сразу после его возникновения из-за установленного регистра PRIMASK/FAULTMASK, а признак отложенного прерывания был сброшен программно записью в регистр управления прерыванием контроллера NVIC), то

прерывание будет аннулировано (**Рис. 7.10**). Признак отложенного прерывания хранится в регистрах контроллера NVIC и доступен для записи, т.е. вы можете программно как сбросить отложенное прерывание, так и перевести новое прерывание в состояние ожидания обработки.

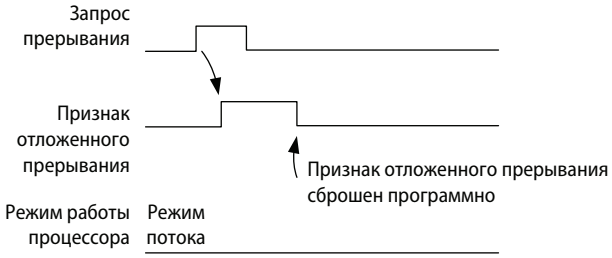


Рис. 7.10. *Сброс признака отложенного прерывания до начала обработки этого прерывания.*

Когда процессор приступает к обработке прерывания, оно становится активным и признак отложенного прерывания автоматически сбрасывается (**Рис. 7.11**). После того как прерывание станет активным, вы не сможете повторно запустить обработчик этого же прерывания до тех пор, пока процедура обработки прерывания не будет завершена командой возврата из прерывания (также называемой *выходом из исключения*, см. Главу 9). После выполнения данной команды признак активности прерывания сбрасывается, и прерывание может быть обработано повторно, если бит признака отложенного прерывания установлен в 1. Прерывание можно повторно перевести в состояние ожидания непосредственно перед выходом из процедуры обработки.

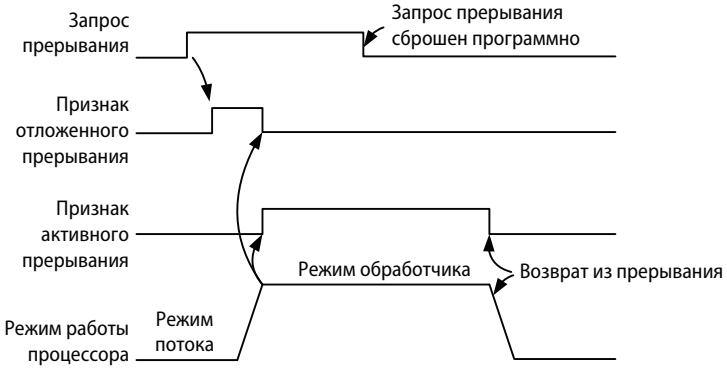


Рис. 7.11. *Перевод прерывания в активное состояние.*

Если источник прерывания продолжает удерживать сигнал запроса прерывания в активном состоянии, то в конце процедуры обработки прерывание будет отложено повторно, как показано на **Рис. 7.12**. Именно так ведёт себя классический процессор ARM7TDMI.

Многократная установка и снятие запроса прерывания до момента начала его обработки интерпретируется процессором как единственный запрос прерывания (**Рис. 7.13**). Если запрос прерывания будет снят и на короткое время установ-

лен во время выполнения обработчика прерывания, то прерывание будет снова отложено, как показано на **Рис. 7.14**.

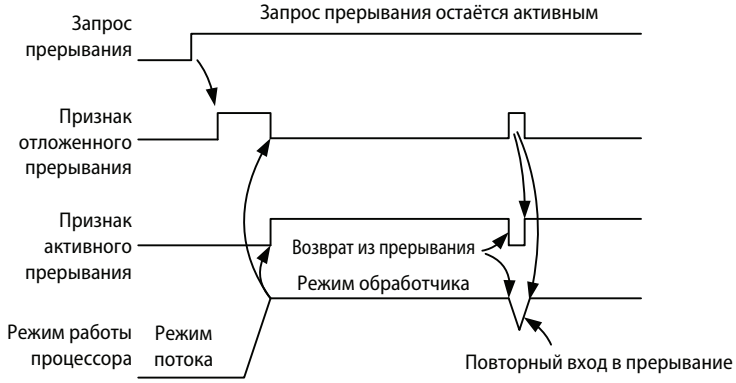


Рис. 7.12. *Удерживание запроса прерывания.*

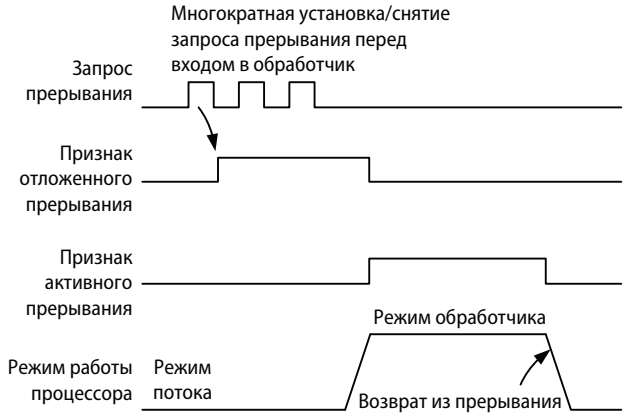


Рис. 7.13. *Многократная установка и снятие запроса прерывания.*

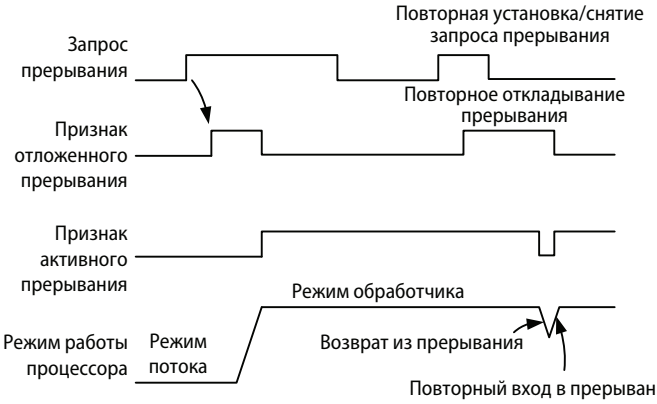


Рис. 7.14. *Повторная установка запроса прерывания во время выполнения обработчика.*

Прерывание откладывается даже в том случае, если оно запрещено; при последующем разрешении данного прерывания оно будет обработано. Поэтому, прежде чем разрешить какое-либо прерывание, никогда не помешает убедиться в том, что оно не находится в состоянии ожидания (источник указанного прерывания мог ранее выставить запрос и установить признак отложенного прерывания). При необходимости вы можете перед разрешением прерывания сбросить этот признак.

7.5. Исключения отказов

Некоторые системные исключения предназначены для обработки отказов. Процессор Cortex-M3 различает следующие виды отказов:

- отказы шины (Bus Fault);
- отказы системы управления памятью (MemManage Fault);
- отказы программы (Usage Fault);
- тяжёлые отказы (Hard Fault).

7.5.1. Отказы шины

Отказы шины возникают при получении сигнала ошибки во время обмена по шине AHB. Это может произойти в следующие моменты:

- во время предварительной выборки команды — такой отказ называется *отказом предвыборки* (prefetch abort);
- во время обращения к данным — такой отказ называется *отказом данных* (data abort).

Также отказ шины может возникнуть во время следующих операций:

- при сохранении данных в стеке в начале обработки прерывания — такой отказ называется *ошибкой загрузки в стек* (stacking error);
- при извлечении данных из стека в конце обработки прерывания — такой отказ называется *ошибкой извлечения из стека* (unstacking error);
- при чтении адреса вектора прерывания (выборке вектора) во время запуска ядром процесса обработки прерывания (эта особая ситуация классифицируется как тяжёлый отказ).

При возникновении любого из этих отказов (за исключением ошибки при выборке вектора) будет запущен обработчик исключения Bus Fault, при условии, что он разрешён и на момент возникновения отказа не выполнялась обработка других исключений с таким же или более высоким приоритетом. Если обработчик разрешён, но на момент возникновения отказа процессор был занят обработкой исключения с более высоким приоритетом, то исключение Bus Fault будет отложено. И наконец, если обработчик исключения Bus Fault не разрешён или же если это исключение возникло во время обработки другого исключения с таким же или более высоким приоритетом, то будет запущен обработчик исключения Hard Fault. Если же отказ шины возникнет во время выполнения обработчика Hard Fault, то процессорное ядро перейдёт в состояние блокировки[1)].

[1)] Более подробно состояние блокировки рассматривается в Главе 12.

Причины появления сигнала ошибки на шине AHB

Отказ шины возникает при получении по шине AHB сигнала ошибки. Обычно это происходит в следующих случаях:

- при попытке обращения по недопустимому адресу (например, при обращении по адресу, который не задействован ни одним из устройств);
- в случае, если устройство не готово к обмену (например, при попытке обращения к динамическому ОЗУ до инициализации контроллера динамической памяти);
- при попытке обмена с разрядностью, не поддерживаемой конечным устройством (например, при однобайтном обращении к регистру периферийного устройства, который может быть считан или записан только как слово);
- в случае, если устройство по какой-либо причине не принимает данные (например, при обращении к устройству, допускающему программирование только на привилегированном уровне доступа).

Для разрешения обработчика исключения Bus Fault следует установить бит BUSFAULTENA регистра управления и состояния системных обработчиков SHCSR контроллера NVIC. Если таблица векторов была перемещена в ОЗУ, то перед установкой бита необходимо убедиться, что начальный адрес обработчика исключения присутствует в таблице.

Как же нам определить причину, по которой процессору пришлось запустить обработчик исключения Bus Fault? Для этого в контроллере NVIC предусмотрено несколько регистров состояния отказов (xFSR), одним из которых является регистр состояния отказа шины BFSR. С помощью указанного регистра обработчик исключения Bus Fault может определить, что послужило причиной возникновения отказа: выборка команды, обращение к данным или же прерывание операции загрузки в стек/извлечения из стека.

В случае «точного» отказа шины адрес команды, вызвавшей ошибку, может быть определён из значения, сохранённого в стеке счётчика команд. Если при этом установлен бит BFARVALID регистра BFSR, то дополнительно можно будет определить адрес памяти, обращение к которому вызвало отказ шины. Значение данного адреса содержится в регистре адреса отказа шины BFAR контроллера NVIC. Для «неточных» отказов шины подобная информация недоступна, поскольку на момент получения сообщения об ошибке процессор мог уже выполнить несколько других команд.

«Точные» и «неточные» отказы шины

Отказы шины, вызванные обращением к данным, можно разделить на «точные» и «неточные». «Неточный» отказ — это отказ, вызванный выполнением уже завершённой операции (такой как запись с буферизацией), которая могла произойти несколько тактов назад. «Точный» отказ шины — это отказ, вызванный последней завершённой операцией. Например, операция чтения вызовет в процессоре Cortex-M3 «точный» отказ, поскольку команда чтения не может быть завершена до тех пор, пока не будут получены данные.

Регистр BFSR (**Табл. 7.8**) является 8-битным регистром, расположенным по адресу 0xE000ED29. Также к этому регистру можно обратиться посредством 32-битной пересылки по адресу 0xE000ED28 (регистр будет расположен во вто-

ром байте слова). Сброс любого бита, используемого для индикации ошибки, осуществляется записью в него 1.

Таблица 7.8. **Регистр состояния отказа шины BFSR (0xE000ED29)**

Биты	Обозначение	Тип	Значение после сброса	Описание
7	BFARVALID	—	0	Индицирует корректность содержимого регистра BFAR
6:5	—	—	—	—
4	STKERR	R/Wc	0	Ошибка загрузки в стек
3	UNSTKERR	R/Wc	0	Ошибка извлечения из стека
2	IMPRECISERR	R/Wc	0	«Неточное» нарушение доступа к данным
1	PRECISERR	R/Wc	0	«Точное» нарушение доступа к данным
0	IBUSERR	R/Wc	0	Нарушение доступа к команде

7.5.2. Отказы системы управления памятью

Отказы системы управления памятью могут быть вызваны обращением к памяти, конфликтующим с настройкам модуля MPU, или же некорректным обращением (например, при попытке выполнить код из области памяти, не имеющей такой возможности), которое может генерировать отказ даже в случае отсутствия модуля MPU.

Наиболее частыми отказами, связанными с модулем MPU, являются:

- обращение к областям памяти, не заданным в настройках модуля MPU;
- запись в области памяти, предназначенные только для чтения;
- обращение на пользовательском уровне к области, допускающей обращение только на привилегированном уровне.

При возникновении любой из этих ситуаций будет запущен обработчик исключения MemManage Fault, при условии, что он разрешён. Если отказ системы управления памятью произойдёт одновременно с возникновением исключения, имеющего более высокий приоритет, то это исключение будет обработано первым, а исключение MemManage Fault будет отложено. Наконец, если обработчик исключения MemManage Fault не разрешён или же если это исключение возникло во время обработки другого исключения с таким же или более высоким приоритетом, то будет запущен обработчик исключения Hard Fault. Если же отказ системы управления памятью возникнет во время выполнения обработчика Hard Fault, то процессорное ядро перейдёт в состояние блокировки.

Перед использованием обработчика исключения MemManage Fault его необходимо разрешить. Для этого нужно установить бит MEMFAULTENA регистра SHCSR контроллера NVIC. Если таблица векторов была перемещена в ОЗУ, то перед разрешением обработчика необходимо прописать его начальный адрес в таблице.

Для определения причины возникновения отказа системы управления памятью в контроллере NVIC имеется соответствующий регистр состояния отказа MMFSR. Если этот регистр показывает, что отказ произошёл при нарушении прав доступа во время обращения к данным (бит DACCVIOL) или к команде (бит

IACCVIOL), то адрес команды, вызвавшей ошибку, может быть определён из значения счётчика команд, сохранённого в стеке. Если при этом установлен бит MMARVALID регистра MMFSR, то из регистра адреса отказа системы управления памятью MMFAR контроллера NVIC можно будет прочитать адрес памяти, обращение к которому вызвало отказ.

Регистр MMFSR (**Табл. 7.9**) является 8-битным регистром, расположенным по адресу 0xE000ED28. Также к данному регистру можно обратиться посредством 32-битной пересылки по тому же адресу; в этом случае регистр будет расположен в младшем байте слова. Как и во всех регистрах xFSR, сброс любого бита, используемого для индикации ошибки, осуществляется записью в него 1.

Таблица 7.9. **Регистр состояния отказа системы управления памятью MMFSR (0xE000ED28)**

Биты	Обозначение	Тип	Значение после сброса	Описание
7	MMARVALID	—	0	Индицирует корректность содержимого регистра MMAR
6:5	—	—	—	—
4	MSTKERR	R/Wc	0	Ошибка загрузки в стек
3	MUNSTKERR	R/Wc	0	Ошибка извлечения из стека
2	—	—	—	—
1	DACCVIOL	R/Wc	0	Нарушение доступа к данным
0	IACCVIOL	R/Wc	0	Нарушение доступа к команде

7.5.3. Отказы программы

Отказы программы могут быть вызваны следующими причинами:

- вызов неопределённой команды;
- вызов команды сопроцессора (в процессоре Cortex-M3 поддержка сопроцессора отсутствует; однако, используя механизм исключений, можно обеспечить выполнение программы, скомпилированной для другого процессора Cortex, посредством программной эмуляции сопроцессора);
- попытка переключения в состояние ARM (программа может использовать данный механизм для проверки, поддерживает ли выполняющий её процессор код ARM; поскольку Cortex-M3 не поддерживает состояние ARM, то при попытке переключения в это состояние произойдёт отказ программы);
- некорректный выход из прерывания (регистр связи содержит неверное или некорректное значение);
- обращение к невыровненным данным в командах групповой загрузки или сохранения.

Установкой определённых битов в регистрах контроллера NVIC можно обеспечить генерацию отказа программы также в следующих ситуациях:

- при делении на ноль;
- при любом невыровненном доступе к памяти.

В случае возникновения отказа программы будет запущен обработчик исключения Usage Fault, при условии, что он разрешён. Если отказ программы произой-

дёт одновременно с возникновением исключения, имеющего более высокий приоритет, то исключение Usage Fault будет отложено. Наконец, если обработчик исключения Usage Fault не разрешён или же если это исключение возникло во время обработки другого исключения с таким же или более высоким приоритетом, то будет запущен обработчик исключения Hard Fault. Если же отказ программы возникнет во время выполнения обработчика Hard Fault или обработчика NMI, то процессор перейдёт в состояние блокировки.

Разрешение обработчика исключения Usage Fault осуществляется установкой бита USGFAULTENA регистра SHCSR контроллера NVIC. Если таблица векторов была перемещена в ОЗУ, то перед разрешением обработчика необходимо поместить в таблицу его начальный адрес.

Для определения причины возникновения отказа в контроллере NVIC предусмотрен соответствующий регистр состояния отказа программы UFSR. В самом обработчике адрес команды, вызвавшей отказ, можно определить из значения счётчика команд, сохранённого в стеке.

Случайное переключение в состояние ARM

Одной из наиболее распространённых причин возникновения отказов программы является попытка переключения процессора в режим ARM. В частности, это может произойти при загрузке в PC нового значения со сброшенным младшим битом. Как правило, это происходит в следующих случаях:

- при попытке перехода по команде `BX` или `BLX` по адресу, содержащемуся в регистре, без предварительной установки младшего бита значения;
- при сброшенном младшем бите значения вектора в таблице векторов;
- при ручной модификации сохранённого в стеке значения PC, вызвавшей сброс младшего бита, и последующем извлечении PC из стека.

Во всех указанных ситуациях одновременно с генерацией исключения Usage Fault устанавливается бит INVSTATE регистра UFSR.

Регистр UFSR (**Табл. 7.10**) занимает 2 байта в памяти и обратиться к нему можно посредством 16-битной пересылки по адресу 0xE000ED2A или 32-битной пересылки по адресу 0xE000ED28. В последнем случае значение регистра будет содержаться в старшем полуслове. Как и в остальных регистрах xFSR, сброс любого бита, используемого для индикации ошибки, осуществляется записью в этот бит 1.

Таблица 7.10. **Регистр состояния отказа программы UFSR (0xE000ED2A)**

Биты	Обозначение	Тип	Значение после сброса	Описание
9	DIVBYZERO	R/Wc	0	Попытка выполнения операции деления на ноль (может быть установлен только при установленном бите DIV_0_TRP)
8	UNALIGNED	R/Wc	0	Отказ невыровненного доступа
7:4	—	—	—	—
3	NOCP	R/Wc	0	Попытка выполнения команды сопроцессора
2	INVPC	R/Wc	0	Попытка выхода из обработчика исключения с некорректным значением EXC_RETURN

Таблица 7.10. **Регистр состояния отказа программы UFSR (0xE000ED2A)** *(продолжение)*

Биты	Обозначение	Тип	Значение после сброса	Описание
1	INVSTATE	R/Wc	0	Попытка переключения в некорректное состояние (например, в состояние ARM)
0	UNDEFINSTR	R/Wc	0	Попытка выполнения неопределённой команды

7.5.4. Тяжёлые отказы

Исключение Hard Fault может быть вызвано отказом программы, отказом шины или отказом системы управления памятью в том случае, если выполнение обработчиков указанных системных исключений невозможно. Также тяжёлый отказ может быть вызван отказом шины во время выборки вектора (чтения значения из таблицы векторов). Регистр состояния тяжёлого отказа HFSR контроллера NVIC позволяет определить, был ли вызван данный отказ выборкой вектора. Если нет, то обработчик исключения Hard Fault должен будет проверить содержимое других регистров xFSR для определения причины возникновения отказа.

Биты регистра HFSR указаны в **Табл. 7.11**. Как и в прочих регистрах xFSR, сброс любого бита, используемого для индикации ошибки, осуществляется записью в него 1.

Таблица 7.11. **Регистр состояния тяжёлого отказа HFSR (0xE000ED2C)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31	DEBUGEVT	R/Wc	0	Исключение Hard Fault было вызвано отладочным событием
30	FORCED	R/Wc	0	Исключение Hard Fault было вызвано отказом шины, отказом системы управления памятью или отказом программы
29:2	—	—	—	—
1	VECTBL	R/Wc	0	Исключение Hard Fault было вызвано ошибкой выборки вектора
0	—	—	—	—

7.5.5. Обработка отказов

Если ошибка в программе выявляется на этапе разработки, то мы можем определить причину возникновения данной ошибки, используя регистры xFSR, после чего исправить её. Наиболее распространённые причины возникновения различных отказов указаны в Приложении Д настоящей книги. В реальной работающей системе ситуация несколько иная. После определения причины отказа программа должна решить, что делать дальше. Если система работает под управлением ОС, то проблемные задачи и приложения могут просто останавливаться

(выгружаться из памяти). В других случаях может потребоваться сброс системы. Вообще говоря, требования, касающиеся восстановления после сбоев, определяются конкретным приложением. При правильной реализации этой функции изделие можно сделать более надёжным, однако лучшим выходом всё же будет предотвращение самой возможности возникновения отказов. Итак, существуют следующие методы восстановления после сбоев:

- *Сброс.* Эту операцию можно выполнить, используя бит SYSRESETREQ регистра управления прерываниями и сбросом AIRC контроллера NVIC. В результате установки указанного бита в 1 будет сброшено большинство узлов системы, за исключением компонентов отладки. Если вам не требуется сбрасывать всю систему, то вы можете сбросить только процессор, используя бит VECTERSET того же регистра.
- *Восстановление.* В ряде случаев проблему, вызвавшую исключение отказа, можно разрешить. Скажем, если причиной отказов являются команды сопроцессора, то проблему можно устранить, программно эмулируя эти команды.
- *Завершение задачи.* В системах, использующих ОС, возможно завершение и, при необходимости, повторный запуск задачи, вызвавшей отказ.

Содержимое регистров xFSR сохраняется до тех пор, пока они не будут очищены программно. После обработки любого бита состояния отказа обработчик исключения должен его сбросить. Если этого не сделать, то при повторном вызове обработчика в случае возникновения другого отказа он может сделать неправильный вывод о наличии первого отказа и, соответственно, попытается обработать его повторно. В регистрах xFSR используется механизм «сброс посредством записи» (write-to-clear), при котором сброс бита осуществляется записью в него 1. Разработчики микроконтроллеров также могут включать в свои изделия дополнительный регистр состояния отказов AFSR, предназначенный для индикации прочих отказов. Реализация такого регистра определяется конструктивными особенностями конкретного микроконтроллера.

7.6. Вызов супервизора и системных служб

Исключения вызова супервизора (SVCall) и вызова системной службы (PendSV) предназначены для использования системным программным обеспечением, а также операционными системами. Исключение SVCall применяется для генерации вызовов системных функций. Например, вместо того, чтобы позволить пользовательской программе напрямую обращаться к оборудованию, операционная система может предоставлять доступ к оборудованию посредством исключения SVCall. То есть, если пользовательской программе необходимо задействовать некое оборудование, то она, используя команду `SVC`, генерирует исключение SVCall, в результате чего запускается обработчик данного исключения из состава операционной системы, который и обслуживает запрос пользовательского приложения. В этом случае обращение к аппаратуре находится под управлением ОС, что увеличивает надёжность системы, предотвращая некорректный доступ пользовательского ПО к оборудованию.

Исключение SVCall также позволяет сделать программное обеспечение более переносимым, поскольку пользовательской программе не требуется досконально знать, каким образом осуществляется работа с оборудованием. Она должна «знать» только идентификатор API-функции и её параметры; собственно взаимодействие с оборудованием на аппаратном уровне осуществляется драйверами устройств (**Рис. 7.15**).

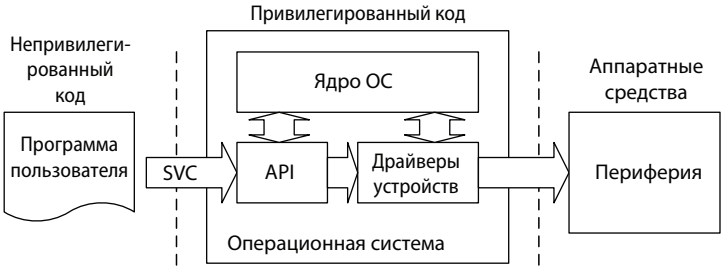


Рис. 7.15. *Исключение SVCall как шлюз для функций ОС.*

Как уже говорилось, генерация исключения SVCall осуществляется вызовом команды `SVC`. Данная команда требует параметра в виде константы, посредством которой осуществляется передача информации в обработчик исключения. Обработчик может извлечь это значение и определить требуемое действие. Например:

```
SVC  #0x3   ; Вызов SVC-функции с ID = 3
```

Можно использовать и традиционный синтаксис указанной команды (без символа «#»):

```
SVC  0x3    ; Вызов SVC-функции с ID = 3
```

В программах на языке Си вызов команды `SVC` можно сгенерировать, используя встроенную функцию `__svc` (при работе в среде RVDS компании ARM или MDK-ARM компании Keil) или же встроенный ассемблер (при работе с другими компиляторами).

В обработчике исключения SVCall вы можете определить значение, переданное в команде. Для этого следует прочитать из стека сохранённое значение счётчика команд, после чего считать код команды по данному адресу, маскируя ненужные биты. Если в системе для пользовательских приложений используется указатель PSP, то вначале вам может потребоваться определить, в каком из стеков был сохранён счётчик команд. Это можно узнать из значения, находящегося в регистре связи, при входе в обработчик исключения (более подробно данный вопрос рассматривается в Главе 8).

SVC и команда программного прерывания (ARM7)

Если вы уже работали с традиционными процессорами ARM, такими как ARM7, то вы наверняка знаете, что в них имеется команда программного прерывания `SWI`. Команда `SVC` выполняет схожие функции и даже более того, машинный код команды `SVC` идентичен коду команды `SWI` процессора ARM7. Однако в связи с переработкой модели исключений команда была переименована. Это позволяет гарантировать, что программистом будут выполнены определённые действия по переносу кода программы с процессора ARM7 на процессор Cortex-M3.

Из-за приоритетной модели прерываний, реализованной в процессоре Cortex-M3, вы не можете использовать команду `SVC` внутри обработчика исключения SVCall (поскольку приоритет нового исключения будет таким же, что и у выполняющегося в данный момент). Если попытаться так сделать, то будет сгенерировано исключение Usage Fault. По этой же причине нельзя использовать команду `SVC` в обработчике NMI или в обработчике исключения Hard Fault.

Исключение PendSV, как и исключение SVCall, предназначено для использования в ОС. Однако в то время как исключение SVCall, вызванное командой `SVC`, не может быть отложено (приложение, вызвавшее `SVC`, будет ожидать немедленного выполнения требуемой задачи), исключение PendSV может быть отложено. Это позволяет операционной системе отложить выполнение требуемых операций до того момента, когда будут завершены другие важные задачи. Исключение PendSV генерируется записью 1 в бит PENDSVSET регистра ICSR контроллера прерываний.

Типичным использованием исключения PendSV является *переключение контекста* (переключение между задачами). Пусть, к примеру, система имеет две активных задачи, переключение между которыми может быть инициировано:

- вызовом функции `SVC`;
- системным таймером (SYSTICK) .

Рассмотрим вариант переключения контекста по сигналу системного таймера (**Рис. 7.16**).

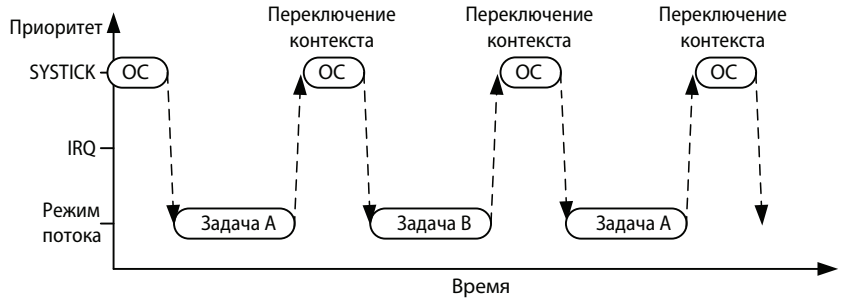


Рис. 7.16. *Использование SYSTICK для переключения между двумя задачами.*

Если запрос прерывания будет выставлен до возникновения исключения SYSTICK, то последнее вытеснит обработчик IRQ. При этом операционной системе уже не нужно будет переключать контекст, иначе обработка прерывания будет приостановлена на некоторое время. В процессоре Cortex-M3 также предусмотрена возможность генерации исключения Usage Fault при попытке переключения в режим потока при наличии активного прерывания (**Рис. 7.17**).

Чтобы исключить задержки при обработке запросов IRQ, некоторые ОС выполняют переключение контекста только в том случае, если ни один из обработчиков IRQ не выполняется. Однако это может привести к появлению очень больших задержек между переключениями задач, особенно если частота появления прерываний близка к частоте генерации исключения SYSTICK.

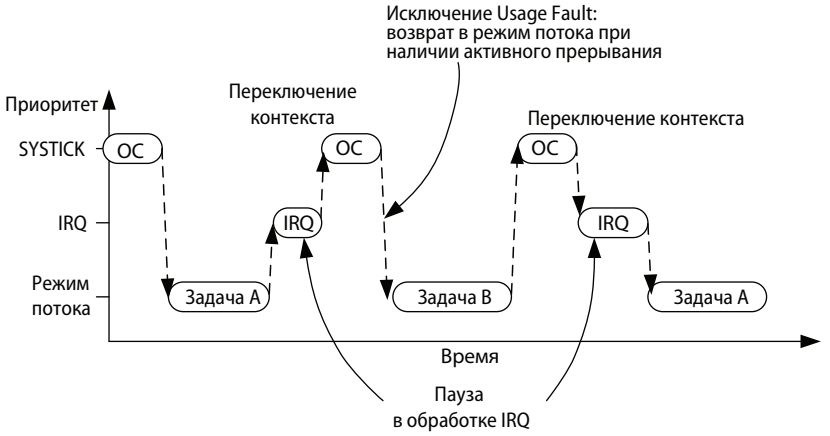


Рис. 7.17. *Проблемы переключения контекста при возникновении IRQ.*

Исключение PendSV решает указанную проблему, приостанавливая обработку запроса на переключение контекста на время, необходимое для завершения всех обработчиков IRQ. Для этого исключению PendSV назначается минимальный приоритет. Если ОС обнаруживает, что какое-то прерывание находится в активном состоянии (выполнение обработчика IRQ было прервано исключением SYSTICK), то она задерживает переключение контекста, откладывая исключение PendSV. В качестве примера на **Рис. 7.18** показан процесс переключения контекста в случае следующей последовательности событий:

1. Задача A выполняет команду `SVC` для переключения задач (например, ожидая завершения какого-либо процесса).
2. ОС получает запрос, выполняет подготовку к переключению контекста и генерирует исключение PendSV.
3. После выхода из обработчика SVCall ЦПУ сразу же запускает обработчик исключения PendSV, в котором производится переключение контекста.
4. После завершения обработчика исключения PendSV и возврата в режим потока начинает выполняться задача B.
5. Возникает прерывание и запускается его обработчик.
6. Во время обработки прерывания происходит исключение SYSTICK (тик ОС).
7. Операционная система выполняет необходимые действия, после чего откладывает исключение PendSV, готовясь к переключению контекста.
8. При выходе из исключения SYSTICK управление передаётся обратно в обработчик прерывания.
9. После завершения процедуры обработки прерывания запускается обработчик исключения PendSV, в котором производится переключение контекста.
10. После завершения обработчика исключения PendSV и возврата в режим потока начинает выполняться задача A.

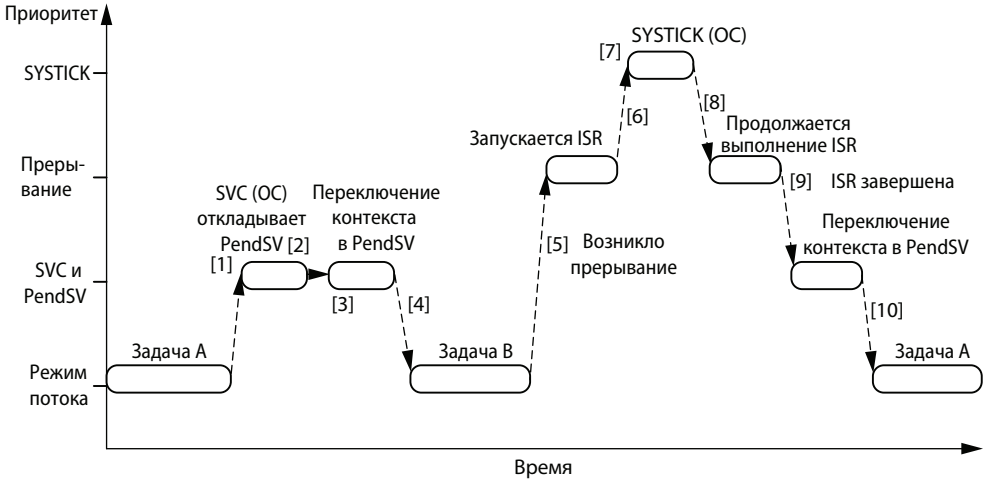


Рис. 7.18. *Переключение контекста с использованием исключения PendSV.*

ГЛАВА 8
КОНТРОЛЛЕР ВЛОЖЕННЫХ ВЕКТОРНЫХ ПРЕРЫВАНИЙ И УПРАВЛЕНИЕ ПРЕРЫВАНИЯМИ

8.1. Общие сведения о контроллере прерываний

Контроллер вложенных векторных прерываний (Nested Vectored Interrupt Controller — NVIC) представляет собой один из компонентов процессора Cortex™-M3, тесно связанный с логикой ядра процессора. Все регистры контроллера прерываний отображены на адресное пространство процессора. Помимо регистров управления обработкой прерываний и соответствующих узлов, контроллер NVIC также содержит регистры управления системным таймером SYSTICK и компонентами отладки. В этой главе мы познакомимся с узлами контроллера, осуществляющими поддержку обработки прерываний. Модуль защиты памяти и узлы, выполняющие управление отладкой, обсуждаются в последующих главах.

Контроллер NVIC поддерживает от 1 до 240 входов внешних прерываний, обычно называемых *входами запросов прерываний* (IRQ). Точное число поддерживаемых прерываний определяется производителями конкретных микросхем на базе процессора Cortex-M3. Кроме того, в контроллере NVIC имеется вход немаскируемого прерывания (NonMaskable Interrupt — NMI). Каким образом он будет использоваться, тоже определяется производителем микросхемы. В некоторых устройствах данный вход может быть даже недоступен для внешних сигналов.

Регистры контроллера NVIC расположены в пространстве управления системой, начиная с адреса 0xE000E000. Большинство регистров управления и состояния прерываний доступны только в привилегированном режиме, за исключением регистра программной генерации прерывания STIR, который может быть сконфигурирован для работы в пользовательском режиме. Для обращения к регистрам управления и состояния прерываний могут использоваться 1-, 2- и 4-байтные пересылки.

С прерываниями также связаны несколько регистров маскирования прерываний. Эти регистры относятся к группе регистров специального назначения и были рассмотрены в Главе 3. Для обращения к указанным регистрам предназначены специальные команды: пересылка содержимого регистра специального назначения в регистр общего назначения (`MRS`) и пересылка содержимого регистра общего назначения в регистр специального назначения (`MSR`).

8.2. Базовые средства конфигурации прерываний

С каждым входом внешнего прерывания связаны несколько регистров:

- регистры разрешения и отмены разрешения прерывания;
- регистры установки и сброса признака отложенного прерывания;
- регистр уровня приоритета прерывания;
- регистр активного состояния прерывания.

Кроме того, на процесс обработки прерываний оказывают влияние и другие регистры:

- регистры маскирования прерываний (PRIMASK, FAULTMASK и BASEPRI);
- регистр смещения таблицы векторов (VTOR);
- регистр программной генерации прерывания (STIR);
- регистр управления прерываниями и сбросом (AIRCR).

8.2.1. Разрешение и запрещение прерываний

Регистр разрешения прерывания программируется по двум адресам. Для установки бита разрешения прерывания необходимо выполнить запись по адресу регистра NVIC_ISER, а для сброса этого же бита — по адресу регистра NVIC_ICER. Благодаря такому решению разрешение или запрещение конкретного прерывания не влияет на состояние битов разрешения остальных прерываний. Регистры NVIC_ISER/NVIC_ICER являются 32-битными; каждый бит регистра соответствует одному входу прерывания.

Поскольку в процессоре Cortex-M3 может быть больше 32 внешних прерываний, он может иметь несколько регистров NVIC_ISER и NVIC_ICER, например NVIC_ISER0, NVIC_ISER1 и т.д. (**Табл. 8.1**). Причём, в регистрах реализуются биты разрешения только для существующих прерываний. Соответственно, если у вас всего 32 входа прерываний, то вам будут доступны только регистры NVIC_ISER0 и NVIC_ICER0. Для обращения к этим регистрам можно использовать 1-, 2- и 4-байтные пересылки. Поскольку первые 16 исключений являются системными, внешнему прерыванию №0 соответствует исключение с номером 16 (**Табл. 8.2**).

Таблица 8.1. **Регистры разрешения прерываний NVIC_ISERx (0xE000E100…0xE000E11C) и регистры отмены разрешения прерываний NVIC_ICERx (0xE000E180…0xE000E19C)**

Адрес	Регистр	Тип	Значение после сброса	Описание
0xE000E100	NVIC_ISER0	R/W	0	Разрешает внешние прерывания с номерами от 0 до 31: бит [0] — прерывание №0 (исключение №16); бит [1] — прерывание №1 (исключение №17); … бит [31] — прерывание №31 (исключение №47). Для установки бита в него необходимо записать 1; запись 0 не имеет смысла. При чтении возвращается текущее состояние
0xE000E104	NVIC_ISER1	R/W	0	Разрешает внешние прерывания с номерами от 32 до 63. Для установки бита в него необходимо записать 1; запись 0 не имеет смысла. При чтении возвращается текущее состояние
0xE000E108	NVIC_ISER2	R/W	0	Разрешает внешние прерывания с номерами от 64 до 95. Для установки бита в него необходимо записать 1; запись 0 не имеет смысла. При чтении возвращается текущее состояние
…	—	—	—	—
0xE000E180	NVIC_ICER0	R/W	0	Отменяет разрешение внешних прерываний с номерами от 0 до 31: бит [0] — прерывание №0 (исключение №16); бит [1] — прерывание №1 (исключение №17); … бит [31] — прерывание №31 (исключение №47). Для сброса бита в него необходимо записать 1; запись 0 не имеет смысла. При чтении возвращается текущее состояние
0xE000E184	NVIC_ICER1	R/W	0	Отменяет разрешение внешних прерываний с номерами от 32 до 63. Для сброса бита в него необходимо записать 1; запись 0 не имеет смысла. При чтении возвращается текущее состояние
0xE000E188	NVIC_ICER2	R/W	0	Отменяет разрешение внешних прерываний с номерами от 64 до 95. Для сброса бита в него необходимо записать 1; запись 0 не имеет смысла. При чтении возвращается текущее состояние
…	—	—	—	—

Таблица 8.2. **Регистры установки признака отложенного прерывания NVIC_ISPRx (0xE000E200…0xE000E21C) и регистры сброса признака отложенного прерывания NVIC_ICPRx (0xE000E208…0xE000E280)**

Адрес	Регистр	Тип	Значение после сброса	Описание
0xE000E200	NVIC_ISPR0	R/W	0	Устанавливает признак отложенного прерывания для внешних прерываний с номерами от 0 до 31: бит [0] — прерывание №0 (исключение №16); бит [1] — прерывание №1 (исключение №17); … бит [31] — прерывание №31 (исключение №47). Для установки бита в него необходимо записать 1; запись 0 не имеет смысла. При чтении возвращается текущее состояние
0xE000E204	NVIC_ISPR1	R/W	0	Устанавливает признак отложенного прерывания для внешних прерываний с номерами от 32 до 63. Для установки бита в него необходимо записать 1; запись 0 не имеет смысла. При чтении возвращается текущее состояние
0xE000E208	NVIC_ISPR2	R/W	0	Устанавливает признак отложенного прерывания для внешних прерываний с номерами от 64 до 95. Для установки бита в него необходимо записать 1; запись 0 не имеет смысла. При чтении возвращается текущее состояние
…	—	—	—	—
0xE000E280	NVIC_ICPR0	R/W	0	Сбрасывает признак отложенного прерывания для внешних прерываний с номерами от 0 до 31: бит [0] — прерывание №0 (исключение №16); бит [1] — прерывание №1 (исключение №17); … бит [31] — прерывание №31 (исключение №47). Для сброса бита в него необходимо записать 1; запись 0 не имеет смысла. При чтении возвращается текущее состояние
0xE000E284	NVIC_ICPR1	R/W	0	Сбрасывает признак отложенного внешнего прерывания для внешних прерываний с номерами от 32 до 63. Для сброса бита в него необходимо записать 1; запись 0 не имеет смысла. При чтении возвращается текущее состояние
0xE000E288	NVIC_ICPR2	R/W	0	Сбрасывает признак отложенного внешнего прерывания для внешних прерываний с номерами от 64 до 95. Для сброса бита в него необходимо записать 1; запись 0 не имеет смысла. При чтении возвращается текущее состояние
…	—	—	—	—

8.2.2. Установка/сброс признака отложенного прерывания

Если возникшее прерывание не может быть обработано немедленно (например, из-за выполнения обработчика прерывания с более высоким приоритетом), то оно будет отложено. Для работы с признаком отложенного прерывания предназначены регистры установки признака отложенного прерывания (NVIC_ISPR) и регистры сброса признака отложенного прерывания (NVIC_ICPR). Аналогично регистрам разрешения/запрещения прерываний, если в процессоре реализовано более 32 входов внешних прерываний, то таких пар регистров установки и сброса признака отложенного прерывания будет несколько.

Значение признака отложенного прерывания может быть изменено программно, т.е. вы можете как аннулировать ожидающее своей очереди прерывание, используя регистр NVIC_ICPR, так и программно сгенерировать прерывание посредством регистра NVIC_ISPR (см. **Табл. 8.2**).

8.2.3. Уровни приоритета

Каждому внешнему прерыванию соответствует свой регистр уровня приоритета, разрядность которого в зависимости от конкретной реализации процессора может варьироваться от 3 до 8 бит. Как уже было сказано в предыдущей главе, содержимое этих регистров может быть разделено на две части (уровень приоритета группы и уровень субприоритета) в соответствии с настройками группирования приоритетов. Для обращения к указанным регистрам можно использовать 1-, 2- и 4-байтные пересылки. Общее число регистров уровня приоритета зависит от количества прерываний, реализованных в конкретной микросхеме (**Табл. 8.3**). Адреса регистров уровня приоритета для системных исключений приведены в Приложении Г (**Табл. Г.19**).

Таблица 8.3. **Регистры уровня приоритета прерываний PRI_x (0xE000E400…0xE000E4EF)**

Адрес	Регистр	Тип	Значение после сброса	Описание
0xE000E400	PRI_0	R/W	0 (8 бит)	Уровень приоритета внешнего прерывания №0
0xE000E401	PRI_1	R/W	0 (8 бит)	Уровень приоритета внешнего прерывания №1
…	—	—	—	—
0xE000E41F	PRI_31	R/W	0 (8 бит)	Уровень приоритета внешнего прерывания №31
…	—	—	—	—

8.2.4. Активное состояние

Каждое внешнее прерывание имеет бит активного состояния. Когда процессор начинает выполнение обработчика прерывания, этот бит устанавливается в 1, а при выходе из прерывания он сбрасывается. Однако во время выполнения процедуры обработки прерывания может возникнуть прерывание с более высо-

ким приоритетом, которое приостановит выполнение этого обработчика. Во время выполнения обработчика более высокоприоритетного прерывания первое прерывание будет продолжать считаться активным. Регистры активного состояния являются 32-битными, однако к ним также можно обращаться посредством 2- или 1-байтных пересылок. Если в конкретной реализации процессора имеется более 32 внешних прерываний, то таких регистров будет несколько. Регистры активного состояния доступны только для чтения (**Табл. 8.4**).

Таблица 8.4. **Регистры активного состояния прерывания NVIC_IABRx (0xE000E300…0xE000E31C)**

Адрес	Регистр	Тип	Значение после сброса	Описание
0xE000E300	NVIC_IABR0	R	0	Признак активности внешних прерываний с номерами от 0 до 31: бит [0] — прерывание №0; бит [1] — прерывание №1; … бит [31] — прерывание №31.
0xE000E204	NVIC_IABR1	R	0	Признак активности внешних прерываний с номерами от 32 до 63
…	—	—	—	—

8.2.5. Регистры PRIMASK и FAULTMASK

Регистр PRIMASK используется для запрещения всех исключений, кроме немаскируемого прерывания и исключения Hard Fault. В действительности, этот регистр изменяет текущий уровень приоритета, делая его равным нулю (наивысший программируемый уровень). При программировании на языке Си для установки и сброса регистра PRIMASK можно использовать встроенные функции из состава CMSIS-совместимой библиотеки драйвера устройства или же имеющиеся в компиляторе:

```
void __enable_irq();                  // Очистить PRIMASK
void __disable_irq();                 // Установить PRIMASK
void __set_PRIMASK(uint32_t priMask); // Записать значение в PRIMASK
uint32_t __get_PRIMASK(void);         // Прочитать значение PRIMASK
```

В программах на ассемблере текущее состояние регистра PRIMASK можно изменить, используя команды изменения состояния процессора:

```
CPSIE I ; Очистить PRIMASK (разрешить прерывания)
CPSID I ; Установить PRIMASK (запретить прерывания)
```

Для обращения к этому регистру также можно использовать команды `MRS` и `MSR`. Например:

```
MOV R0, #1
MSR PRIMASK, R0 ; Записать 1 в PRIMASK для запрещения всех прерываний
```

и

```
MOV R0, #0
MSR PRIMASK, R0 ; Записать 0 в PRIMASK для разрешения прерываний
```

Регистр PRIMASK полезен для временного запрещения всех прерываний на время выполнения критических задач. При установленном регистре PRIMASK любое исключение ошибки будет обрабатываться обработчиком Hard Fault.

Регистр FAULTMASK по своему назначению аналогичен регистру PRIMASK, за исключением того, что он изменяет текущий уровень приоритета на –1, т.е. блокируется даже исключение Hard Fault. Соответственно, при установленном регистре FAULTMASK могут обрабатываться только немаскируемые прерывания. Обработчики исключений отказов могут использовать данный регистр для увеличения своего уровня приоритета до значения –1, в результате чего они получают некоторые возможности обработчика исключения Hard Fault (более подробно этот вопрос рассматривается в Главе 12). При программировании на языке Си для установки и сброса регистра FAULTMASK можно использовать функции CMSIS-совместимой библиотеки:

```
void __set_FAULTMASK(uint32_t faultMask);
uint32_t __get_FAULTMASK(void);
```

Те, кто использует язык ассемблера, могут изменять состояние регистра FAULTMASK с помощью команд CPS:

```
CPSIE F ; Очистить FAULTMASK
CPSID F ; Установить FAULTMASK
```

Для обращения к регистру FAULTMASK также можно использовать команды MRS и MSR.

При выходе из обработчика исключения (кроме обработчика немаскируемого прерывания) регистр FAULTMASK очищается автоматически. Ни FAULTMASK, ни PRIMASK не могут быть установлены на пользовательском уровне доступа.

8.2.6. Регистр BASEPRI

В некоторых случаях вам может потребоваться запретить только те прерывания, приоритет которых меньше некоторого значения. Это можно сделать с помощью регистра BASEPRI — для получения требуемого результата достаточно просто записать необходимое значение уровня приоритета в данный регистр. К примеру, если вы хотите заблокировать все исключения с уровнем приоритета не больше 0x60, то достаточно вставить в программу следующие строки:

```
__set_BASEPRI(0x60);  // Запрещаем прерывания с приоритетом 0x60...0xFF,
                      // используя функцию CMSIS
```

Или, на языке ассемблера:

```
MOV R0, #0x60
MSR BASEPRI, R0 ; Запрещаем прерывания с приоритетом 0x60...0xFF
```

Вы также можете прочитать содержимое регистра BASEPRI:

```
x = __get_BASEPRI(void); // Читаем значение BASEPRI
```

То же, на языке ассемблера:

```
MRS R0, BASEPRI
```

Чтобы отключить маскирование, просто обнулите регистр BASEPRI:

```
__set_BASEPRI(0x0);  // Отключаем маскирование BASEPRI
```

То же на языке ассемблера:

```
MOV R0, #0x0
MSR BASEPRI, R0  ; Отключаем маскирование BASEPRI
```

Для обращения к регистру BASEPRI можно также использовать идентификатор BASEPRI_MAX. В принципе, он обозначает тот же самый регистр, однако при использовании этого имени операция записи становится условной. (Хотя имена BASEPRI и BASEPRI_MAX относятся к одному регистру, в машинном коде команды они кодируются разными значениями.) При обращении к регистру BASEPRI_MAX процессор автоматически сравнивает текущее значение приоритета с новым и разрешает изменение регистра только в случае увеличения приоритета; уменьшить приоритет таким образом невозможно.

Например, рассмотрим следующий фрагмент:

```
MOV R0, #0x60
MSR BASEPRI_MAX, R0   ; Запрещаем прерывания с приоритетом 0x60, 0x61, ... и т.д.
MOV R0, #0xF0
MSR BASEPRI_MAX, R0   ; Эта операция записи будет проигнорирована, поскольку
                      ; устанавливаемый приоритет меньше 0x60
MOV R0, #0x40
MSR BASEPRI_MAX, R0   ; Эта операция записи будет выполнена, в результате чего
                      ; уровень маскирования станет равным 0x40
```

Для изменения уровня приоритета на более низкий или для полного отключения маскирования при обращении к регистру необходимо использовать имя BASEPRI. Изменение регистра BASEPRI/BASEPRI_MAX допускается только на привилегированном уровне доступа.

Формат регистра BASEPRI зависит от разрядности регистров приоритета. Так, если в регистрах приоритета реализовано всего 3 бита, то регистр BASEPRI может содержать только следующие значения: 0x00, 0x20, 0x40, …, 0xC0, 0xE0.

8.2.7. Конфигурационные регистры остальных исключений

Исключения Usage Fault, MemManage Fault и Bus Fault разрешаются регистром SHCSR (0xE00ED24). Признаки состоянии ожидания исключений отказов и признаки активного состояния большинства системных исключений также хранятся в этом регистре (**Табл. 8.5**).

***Таблица 8.5.* Регистр состояния и управления обработчиков системных исключений SHCSR (0xE000ED24)**

Биты	Обозначение	Тип	Значение после сброса	Описание
18	USGFAULTENA	R/W	0	Разрешение обработчика исключения Usage Fault
17	BUSFAULTENA	R/W	0	Разрешение обработчика исключения Bus Fault
16	MEMFAULTENA	R/W	0	Разрешение обработчика исключения MemManage Fault
15	SVCALLPENDED	R/W	0	Исключение SVCall отложено; обработчик SVCall выполнялся, но был вытеснен исключением с более высоким приоритетом

Таблица 8.5. **Регистр состояния и управления обработчиков системных исключений SHCSR (0xE000ED24)** *(продолжение)*

Биты	Обозначение	Тип	Значение после сброса	Описание
14	BUSFAULTPENDED	R/W	0	Исключение Bus Fault отложено; обработчик Bus Fault выполнялся, но был вытеснен исключением с более высоким приоритетом
13	MEMFAULTPENDED	R/W	0	Исключение MemManage Fault отложено; обработчик MemManage Fault выполнялся, но был вытеснен исключением с более высоким приоритетом
12	USGFAULTPENDED	R/W	0	Исключение Usage Fault отложено; обработчик Usage Fault выполнялся, но был вытеснен исключением с более высоким приоритетом
11	SYSTICKACT	R/W	0	Читается как 1, если исключение SYSTICK активно
10	PENDSVACT	R/W	0	Читается как 1, если исключение PendSV активно
8	MONITORACT	R/W	0	Читается как 1, если исключение Debug Monitor активно
7	SVCALLACT	R/W	0	Читается как 1, если исключение SVCall активно
3	USGFAULTACT	R/W	0	Читается как 1, если исключение Usage Fault активно
1	BUSFAULTACT	R/W	0	Читается как 1, если исключение Bus Fault активно
0	MEMFAULTACT	R/W	0	Читается как 1, если исключение MemManage Fault активно

Будьте аккуратны при записи в этот регистр, не измените случайно биты активности системных исключений. В противном случае, если вы вдруг сбросите бит активности системного исключения, находящегося в активном состоянии, то в момент выхода из обработчика системного исключения будет сгенерировано исключение отказа.

Перевод исключений NMI, SYSTICK и PendSV в состояние ожидания осуществляется посредством регистра ICSR. Многие битовые поля данного регистра предназначены исключительно для отладки. В прикладных программах, как правило, используются только биты, связанные с признаками отложенного прерывания (**Табл. 8.6**).

Таблица 8.6. **Регистр управления и состояния прерываний ICSR (0xE000ED04)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31	NMIPENDSET	R/W	0	Исключение NMI отложено
28	PENDSVSET	R/W	0	Запись 1 в этот бит переводит исключение PendSV в состояние ожидания. При чтении возвращается текущее состояние

Таблица 8.6. **Регистр управления и состояния прерываний ICSR (0xE000ED04)** *(продолжение)*

Биты	Обозначение	Тип	Значение после сброса	Описание
27	PENDSVCLR	W	0	Запись 1 в этот бит очищает признак отложенного прерывания для исключения PendSV
26	PENDSTSET	R/W	0	Запись 1 в этот бит переводит исключение SYSTICK в состояние ожидания. При чтении возвращается текущее состояние
25	PENDSTCLR	W	0	Запись 1 в этот бит очищает признак отложенного прерывания для исключения SYSTICK
23	ISRPREEMPT	R	0	Отложенное прерывание станет активным на следующем шаге (используется для отладки)
22	ISRPENDING	R	0	Имеется отложенное внешнее прерывание
21:12	VECTPENDING	R	0	Номер отложенного исключения с наибольшим приоритетом
11	RETTOBASE	R	0	Устанавливается в 1 при выполнении процессором обработчика исключения; обеспечивает переход процессора в режим потока после возврата из прерывания при отсутствии других отложенных исключений
9:0	VECTACTIVE	R	0	Номер обрабатываемого в данный момент исключения

8.3. Примеры инициализации прерывания

В большинстве простых приложений программа располагается в ПЗУ, поэтому при отсутствии необходимости динамического изменения обработчиков исключений вся таблица векторов может быть размещена в области Code (начиная с адреса 0x00000000). При этом смещение таблицы всегда будет равно нулю, а вектора прерываний будут располагаться в ПЗУ. Соответственно, для настройки любого из прерываний необходимо выполнить следующее:

1. Задать настройки группирования приоритетов (этот этап не обязателен). По умолчанию в поле PRIGROUP регистра AIRCR содержится нулевое значение — для задания уровня субприоритета используется только 0-й бит регистров уровня приоритета.
2. Задать уровень приоритета прерывания (этот этап также не обязателен). По умолчанию все прерывания имеют 0-й уровень (наивысший приоритет).
3. Разрешить прерывание.

Ниже приведён пример простой процедуры, выполняющей перечисленные операции:

```
NVIC_SetPriorityGrouping(5);
NVIC_SetPriority(7, 0xC0);    // Задаём для IRQ#7 уровень приоритета, равный 0xC0
NVIC_EnableIRQ(7);
```

Кроме того, если вы планируете использовать вложенные прерывания, не забудьте выделить достаточное количество памяти под стек. Поскольку обработчики исключений всегда используют указатель MSP, размер основного стека должен быть достаточным для поддержки максимально возможного числа вложенных прерываний.

Если обработчики прерываний должны изменяться в процессе выполнения программы, то нам придётся переместить таблицу векторов из ПЗУ в ОЗУ — только так мы сможем модифицировать значения векторов прерываний. В результате процедура инициализации немного усложнится и нам потребуется:

1. Изменить при необходимости настройки группирования приоритетов. По умолчанию в поле PRIGROUP регистра AIRCR содержится нулевое значение (биты [7:1] регистра уровня приоритета определяют уровень приоритета группы, а бит [0] — уровень субприоритета).
2. Скопировать вектора обработчиков исключений Hard Fault, NMI, а также других требуемых исключений в новую таблицу векторов, расположенную в ОЗУ.
3. Изменить содержимое регистра смещения таблицы векторов VTOR (см. **Табл. 7.7**) так, чтобы он указывал на новую таблицу векторов.
4. Поместить вектор прерывания в новую таблицу.
5. Задать уровень приоритета прерывания.
6. Разрешить прерывание.

Ниже приведён пример подобной процедуры, написанной с использованием функций из CMSIS-совместимой библиотеки драйвера устройства (предполагается, что начальный адрес новой таблицы векторов задаётся константой NEW _ VECTOR _ TABLE):

```
// HW_REG - макрос для преобразования значения адреса в указатель
#define HW_REG(addr) (*((volatile unsigned long *)(addr)))
#define NEW_VECT_TABLE 0x20008000 // Область СОЗУ для размещения таблицы векторов
NVIC_SetPriorityGrouping(5);
...
HW_REG((NEW_VECT_TABLE +0x8)) = HW_REG(0x8); // Копируем вектор NMI
HW_REG((NEW_VECT_TABLE +0xC)) = HW_REG(0xC); // Копируем вектор Hard Fault
...
SCB->VTOR = NEW_VECT_TABLE; // Перемещаем таблицу векторов в СОЗУ
...
HW_REG(4*(7+16)) = (unsigned) IRQ7_Handler; // Устанавливаем вектор
...
NVIC_SetPriority(7, 0xC0); // Задаём для IRQ#7 уровень приоритета, равный 0xC0
...
NVIC_EnableIRQ(7);
```

Аналогичный код на ассемблере может выглядеть следующим образом:

```
LDR R0, =0xE000ED0C        ; Регистр AIRCR
LDR R1, =0x05FA0500        ; Поле PRIGROUP = 5 (2/6)
STR R1, [R0]               ; Задаём группирование приоритетов
...
MOV R4,#8                  ; Таблица векторов в ПЗУ
LDR R5,=(NEW_VECT_TABLE+8)
```

```
LDMIA R4!,{R0-R1}          ; Считываем вектора исключений NMI и Hard Fault
STMIA R5!,{R0-R1}          ; Копируем вектора в новую таблицу
...
LDR R0,=0xE000ED08         ; Регистр VTOR
LDR R1,=NEW_VECT_TABLE
STR R1,[R0]                ; Указываем новое положение таблицы векторов
...
LDR R0,=IRQ7_Handler       ; Получаем стартовый адрес обработчика IRQ#7
LDR R1,=0xE000ED08         ; Регистр VTOR
LDR R1,[R1]
ADD R1, R1, #(4*(7+16))    ; Вычисляем адрес вектора обработчика IRQ#7
STR R0,[R1]                ; Устанавливаем вектор IRQ#7
...
LDR R0,=0xE000E400         ; Начальный адрес блока регистров приоритета
                           ; внешних IRQ
MOV R1, #0x0
STRB R1,[R0,#7]            ; Устанавливаем приоритет IRQ#7, равный 0x0
...
LDR R0,=0xE000E100         ; Регистр SETEN
MOV R1,#(1<<7)             ; Бит разрешения IRQ#7 (1 сдвигается на 7 бит влево)
STR R1,[R0]                ; Разрешаем прерывание
```

Если программа должна запускаться на различных устройствах, то она, как правило, должна уметь определять:

- число прерываний, поддерживаемых устройством;
- разрядность регистров приоритета.

В процессоре Cortex-M3 предусмотрен регистр типа контроллера прерываний ICTR, который содержит число групп входов прерываний, по 32 входа каждая (**Табл. 8.7**). В качестве альтернативы вы можете определить точное число внешних прерываний, выполняя операцию чтения/записи в регистры конфигурации прерываний, такие как регистры разрешения прерываний или регистры приоритета.

Таблица 8.7. **Регистр типа контроллера прерываний ICTR (0xE000E004)**

Биты	Обозначение	Тип	Значение после сброса	Описание
4:0	INTLINESNUM	R	—	Количество входов прерываний с шагом 32: 0 = от 1 до 32; 1 = от 33 до 64; …

Для определения числа реализованных битов регистров приоритета вы можете записать 0xFF в любой из регистров, а затем прочитать его содержимое и посмотреть число установленных битов. Минимальное число установленных битов равно трём, в этом случае при чтении регистра должно возвращаться число 0xE0.

8.4. Программные прерывания

Программные прерывания могут формироваться различными способами. Первый из этих способов заключается в использовании регистров NVIC_ISPRx;

второй — в использовании регистра программной генерации прерывания STIR (**Табл. 8.8**).

***Таблица 8.8.* Регистр программной генерации прерывания STIR (0xE000EF00)**

Биты	Обозначение	Тип	Значение после сброса	Описание
8:0	INTID	W	—	При записи в это поле какого-либо числа устанавливается бит признака отложенного прерывания для прерывания с соответствующим номером; так, для перевода в состояние ожидания прерывания №0 необходимо записать 0

Например, вы можете генерировать прерывание №3 посредством выражения:

```
NVIC->STIR = 3; /* NVIC->STIR определено в CMSIS-совместимой библиотеке
                   драйвера устройства */
```

С тем же результатом запись в регистр NVIC_ISPRx можно выполнить, используя функцию из CMSIS-совместимой библиотеки:

```
NVIC_SetPendingIRQ(3);
```

Системные исключения (NMI, исключения отказов, PendSV и т.д.) не могут быть отложены с использованием данного регистра. По умолчанию пользовательская программа не может выполнять запись в регистры контроллера NVIC, однако при необходимости ей можно предоставить доступ к регистру STIR — для этого следует установить бит USERSETMPEND регистра управления конфигурацией CCR (0xE000ED14).

8.5. Системный таймер SYSTICK

Таймер SYSTICK входит в состав контроллера NVIC и может использоваться для генерации исключения SYSTICK (исключение №15). Многие операционные системы задействуют отдельный аппаратный таймер, генерирующий периодические прерывания. Благодаря этим прерываниям ОС может осуществлять управление задачами, например выделять каждой задаче свой временной интервал или же предотвращать блокирование всей системы какой-либо одной задачей. Для этого таймер должен быть способен генерировать прерывания и, по возможности, должен быть защищён от воздействия со стороны пользовательских задач, чтобы пользовательский код не мог влиять на его работу.

В связи с этим в процессор Cortex-M3 был включён простой таймер SYSTICK. Поскольку данный таймер имеется во всех кристаллах с процессором Cortex-M3, значительно упрощается перенос программ между подобными устройствами. Этот таймер представляет собой 24-битный вычитающий счётчик, который может тактироваться как от внутреннего тактового сигнала процессора, так и от внешнего тактового сигнала (в [1] этот сигнал обозначен как STCLK). Поскольку источник тактового сигнала STCLK определяется разработчиком конкретной микросхемы, частота данного сигнала в разных устройствах может быть различ-

ной. Соответственно, при выборе источника тактового сигнала необходимо внимательно изучить документацию на используемую микросхему.

Таймер SYSTICK может применяться для генерации прерываний — он имеет своё исключение и отдельный вектор исключения. Указанный таймер облегчает перенос операционных систем и программного обеспечения, поскольку во всех устройствах используется одинаковым образом. Таймер SYSTICK управляется четырьмя регистрами, формат которых указан в **Табл. 8.9**…**8.12**.

Таблица 8.9. **Регистр управления и состояния системного таймера SYST_CSR (0xE000E010)**

Биты	Обозначение	Тип	Значение после сброса	Описание
16	COUNTFLAG	R	0	Читается как 1, если с момента предыдущего чтения регистра счётчик успел досчитать до 0; автоматически сбрасывается в 0 после чтения регистра или при обнулении текущего значения счётчика
2	CLKSOURCE	R/W	0	0 — внешний тактовый сигнал (STCLK); 1 — тактовый сигнал процессора
1	TICKINT	R/W	0	1 — разрешает генерацию прерывания SYSTICK при достижении таймером нулевого значения; 0 — прерывание не генерируется
0	ENABLE	R/W	0	Разрешение таймера SYSTICK

Таблица 8.10. **Регистр значения перезагрузки системного таймера SYST_RVR (0xE000E014)**

Биты	Обозначение	Тип	Значение после сброса	Описание
23:0	RELOAD	R/W	0	Значение, загружаемое в таймер при достижении им нулевого значения

Таблица 8.11. **Регистр текущего значения системного таймера SYST_CVR (0xE000E018)**

Биты	Обозначение	Тип	Значение после сброса	Описание
23:0	CURRENT	R/Wc	0	При чтении возвращается текущее значение таймера. При записи в регистр текущее значение счётчика обнуляется. Также при записи сбрасывается бит COUNTFLAG регистра SYST_CSR

Таблица 8.12. **Регистр калибровочного значения системного таймера SYST_CALIB (0xE000E01C)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31	NOREF	R	—	1 — внешний тактовый сигнал отсутствует (сигнал STCLK недоступен); 0 — внешний тактовый сигнал доступен

Таблица 8.12. **Регистр калибровочного значения системного таймера SYST_CALIB (0xE000E01C)** *(продолжение)*

Биты	Обозначение	Тип	Значение после сброса	Описание
30	SKEW	R	—	1 — калибровочное значение не соответствует 10-мс интервалу; 0 — калибровочное значение точное
23:0	TENMS	R/W	0	Калибровочное значение для периода 10 мс. Разработчик кристалла должен подать это значение (в двоичном коде) на соответствующие входы процессора. Если при чтении данного поля возвращается 0, значит, калибровочное значение недоступно

Регистр калибровки SYST_CALIB позволяет генерировать прерывания SYSTICK с одним и тем же интервалом при работе программы на различных устройствах с ядром Cortex-M3. Для использования этой возможности просто запишите содержимое поля TENMS в регистр значения перезагрузки SYST_RVR; в результате вы получите интервал между прерываниями, равный примерно 10 мс. Для получения других интервалов программа должна будет сама рассчитать новое значение из калибровочного. Однако битовое поле TENMS может быть реализовано не во всех устройствах с процессором Cortex-M3 (в этом случае на входы калибровочного значения подаётся сигнал НИЗКОГО уровня), поэтому перед использованием указанной возможности убедитесь в её наличии по документации производителя.

Вообще говоря, таймер SYSTICK может использоваться не только как системный таймер для операционных систем. В частности, он может применяться в качестве сигнального таймера, для измерения временных интервалов, а также для других целей. Обратите внимание, что при останове процессора (во время отладки) таймер SYSTICK тоже останавливается. В зависимости от реализации микроконтроллера таймер SYSTICK также может останавливаться при переходе процессора в определённые режимы пониженного энергопотребления.

Инициализацию системного таймера рекомендуется выполнять в следующей последовательности:

1. Запретить таймер SYSTICK, записав 0 в регистр управления и состояния SYST_CSR.
2. Записать новое значение перезагрузки в регистр SYST_RVR.
3. Записать любое значение в регистр SYST_CVR для обнуления текущего значения таймера.
4. Записать в регистр управления и состояния SYST_CSR соответствующее значение для запуска таймера SYSTICK.

Эта последовательность применима ко всем реализациям процессора Cortex-M3. Более подробно процесс настройки SYSTICK рассматривается в Главе 14.

ГЛАВА 9
ПРЕРЫВАНИЯ

9.1. Последовательность обработки прерываний/исключений

При возникновении исключительной ситуации (исключения) процессор выполняет следующие операции:

- сохраняет контекст в стеке (загружает содержимое восьми регистров в стек);
- осуществляет выборку вектора (считывает адрес обработчика исключительной ситуации из таблицы векторов);
- модифицирует указатель стека, регистр связи (LR) и счётчик команд (PC).

9.1.1. Сохранение контекста

При возникновении исключительной ситуации содержимое регистров R0…R3, R12, LR, PC и PSR сохраняется в стеке. Если в исполняемом коде используется указатель PSP, то регистры будут сохранены в стеке процесса, иначе — в основном стеке. Обработчики исключительных ситуаций всегда используют основной стек, поэтому во вложенных прерываниях будет задействован именно он.

Блок из восьми слов данных, сохраняемых в стеке, обычно называется стековым фреймом. До появления 2-й ревизии процессора Cortex-M3 стековый фрейм по умолчанию мог начинаться с любого адреса, кратного слову. Во второй ревизии процессора стековый фрейм по умолчанию выравнивается на границу двойного слова, однако такое выравнивание можно отключить, сбросив бит STKALIGN регистра управления конфигурацией CCR контроллера NVIC. Такое выравнивание стекового фрейма доступно и в 1-й ревизии Cortex-M3, только его надо разрешить, установив бит STKALIGN. Более подробно использование регистра CCR рассматривается в Главе 12.

Расположение данных внутри стекового фрейма исключения показано на **Рис. 9.1**, а последовательность загрузки данных в стек представлена на **Рис. 9.2** (предполагается, что после возникновения исключительной ситуации значение указателя стека равно *N*). Из-за конвейерного характера интерфейса шины AHB адрес и данные оказываются сдвинутыми друг относительно друга на одну ступень конвейера.

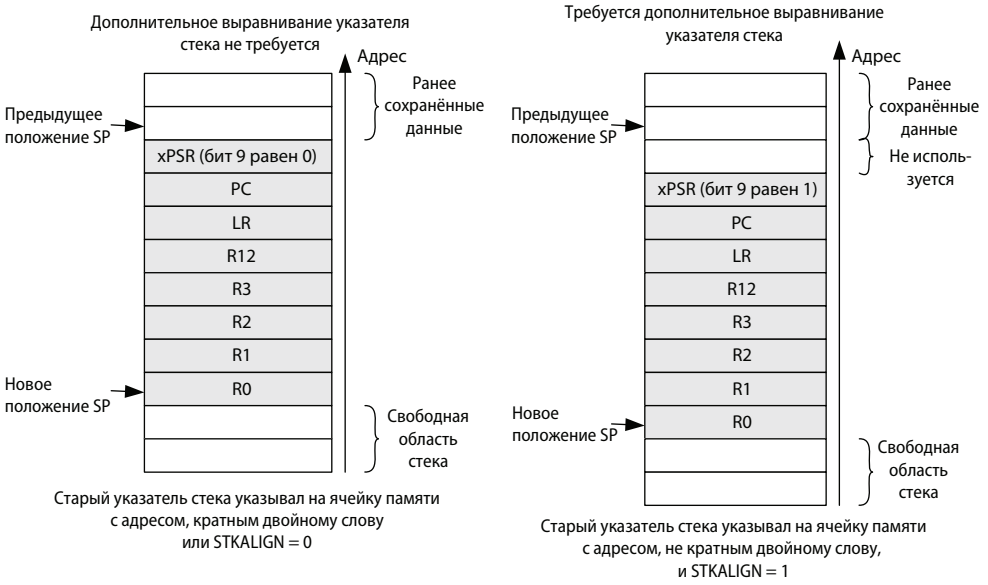


Рис. 9.1. *Стековый фрейм исключения.*

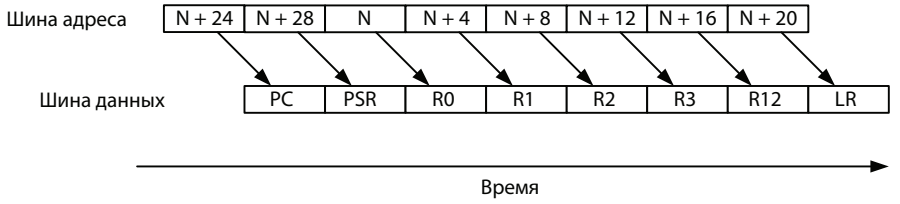


Рис. 9.2. *Последовательность сохранения контекста в стеке.*

Первыми в стек загружаются значения PC и PSR — это позволяет уже на самом начальном этапе начинать выборку команд (что требует модификации PC) и изменять содержимое регистра IPSR. После сохранения указанных регистров значение указателя стека корректируется, и в стек загружаются остальные данные (см. **Рис. 9.1**).

Сохранение содержимого именно регистров R0…R3, R12, R13, LR, PC и PSR регламентируется соглашением AAPCS [5], которое определяет эти регистры как сохраняемые вызывающей процедурой (caller-saved registers). Используемое соглашение позволяет описывать обработчики прерываний в виде обычных Си-функций, поскольку регистры, которые могут быть изменены обработчиком исключительной ситуации, в любом случае сохраняются в стеке.

Регистры общего назначения R0…R3 и R12 располагаются в конце стекового фрейма, что облегчает обращение к ним с использованием адресации относительно SP. Как следствие, с помощью этих регистров можно легко организовать передачу параметров в программные прерывания.

9.1.2. Выборка вектора

В то время, пока шина данных «занимается» сохранением регистров в стек, шина команд выполняет другую важную задачу — по этой шине производится выборка вектора (начального адреса обработчика исключительной ситуации) из таблицы векторов прерываний. Поскольку сохранение контекста и выборка вектора осуществляются разными интерфейсами шин, эти операции могут выполняться одновременно.

9.1.3. Обновление регистров

После того как будет сохранён контекст программы и выбран вектор из таблицы векторов, процессор приступит к обработке исключительной ситуации. На момент входа в обработчик будут изменены следующие регистры:

- *SP* — во время сохранения контекста указатель стека (MSP или PSP) устанавливается на новую позицию. Если процедура обработки исключительной ситуации обращается к стеку, то она будет использовать основной стек и указатель MSP.
- *PSR* — в регистр IPSR (младший байт регистра PSR) заносится номер возникшегоисключения.
- *PC* — после выборки вектора его значение будет загружено в счётчик команд, в результате чего начнётся выборка команд обработчика исключительной ситуации.
- *LR* — в регистр связи заносится специальное значение, называемое EXC_RETURN[1)]. Это значение управляет процессом возврата из прерывания (реально используются только 4 младших бита регистра). Более подробно об EXC_RETURN будет рассказано в разделе 9.3 данной главы.

Помимо рассмотренных регистров, также изменяются некоторые регистры контроллера NVIC. В частности, сбрасывается бит признака отложенного прерывания и устанавливается бит активного состояния.

9.2. Выход из исключения

После завершения обработки исключительной ситуации необходимо осуществить выход из обработчика (в некоторых процессорах эта операция называется *возвратом из прерывания*) для восстановления исходного состояния системы, что позволит возобновить нормальное выполнение прерванной программы. Имеется три способа возврата из прерывания; все они используют значение EXC_RETURN, сохраняемое в регистре LR при входе в обработчик исключения (**Табл. 9.1**).

Некоторые микропроцессорные архитектуры имеют специальные команды для возврата из прерывания (например, команда `reti` в процессоре Intel 8051). В процессоре Cortex-M3 для этой цели применяется обычная команда возврата, что, при использовании языка Си, позволяет описывать обработчик исключительной ситуации как обыкновенную подпрограмму.

[1)] Биты [31:4] значения EXC_RETURN всегда установлены в 1, т.е. это значение имеет вид 0xFFFFFFFx. Информация для возврата содержится в 4 младших битах. Более подробно значение EXC_RETURN будет рассмотрено ниже в этой главе.

Таблица 9.1. **Команды для запуска процесса возврата из прерывания**

Команда возврата	Описание
`BX reg`	Если значение EXC_RETURN всё ещё хранится в регистре LR, то мы можем использовать для возврата из прерывания команду `BX LR`
`POP {PC}` или `POP {…, PC}`	Очень часто содержимое LR сохраняется в стеке после входа в обработчик исключения. В этом случае для загрузки значения EXC_RETURN в счётчик команд мы можем использовать команду `POP`. В результате процессор выполнит возврат из прерывания
`LDR` или `LDM`	Возврат из прерывания также можно выполнить, используя команду `LDR`/`LDM` с PC в качестве регистра-приёмника

При выполнении команды возврата из прерывания производится восстановление контекста из стека и изменение регистров контроллера NVIC (**Табл. 9.1**):

1. *Восстановление контекста из стека* — восстанавливаются значения регистров, помещённые в стек. Также восстанавливается исходное значение указателя стека.
2. *Обновление регистров NVIC* — бит активности исключения сбрасывается. В случае внешнего прерывания, если вход этого прерывания всё ещё активен, будет повторно установлен бит признака отложенного прерывания, что приведёт к повторному запуску обработчика прерывания.

9.3. Вложенные прерывания

Поддержка вложенных прерываний встроена непосредственно в процессорное ядро Cortex-M3 и контроллер NVIC. То есть для использования вложенных прерываний больше не требуется написание дополнительного ассемблерного кода. Нужно только задать соответствующий уровень приоритета для каждого источника прерывания. Во-первых, декодирование приоритетов осуществляется контроллером NVIC автоматически. Так что при обработке какой-либо исключительной ситуации все остальные исключения с таким же или меньшим приоритетом блокируются. Во-вторых, аппаратно-реализованное сохранение и восстановление контекста позволяет обработчикам вложенных прерываний не заботиться о сохранности данных в регистрах.

Необходимо помнить только об одном — о выделении достаточного количества памяти под основной стек при наличии нескольких уровней вложенных прерываний. Каждый уровень использует восемь слов стека, да ещё и самим обработчикам исключений может потребоваться стек — в итоге расход стека может оказаться бóльшим, нежели предполагалось изначально.

Повторный вход в обработчик исключительной ситуации в процессоре Cortex-M3 не допускается. Поскольку каждому исключению соответствует определённый уровень приоритета, а во время обработки исключительной ситуации все прерывания с таким же или меньшим приоритетом блокируются, то обработать это же исключение можно будет только после выхода из текущего обработчика. В частности, поэтому нельзя выполнить команду `SVC` в обработчике исключения SVCall — при такой попытке будет сгенерирована исключительная ситуация Hard Fault.

9.4. «Цепочечная» обработка прерываний

В процессоре Cortex-M3 применены различные решения, позволяющие уменьшить задержку обработки прерываний. Первым решением, которое мы рассмотрим, является механизм «цепочечной» (tail-chaining) обработки прерываний (**Рис. 9.3**).

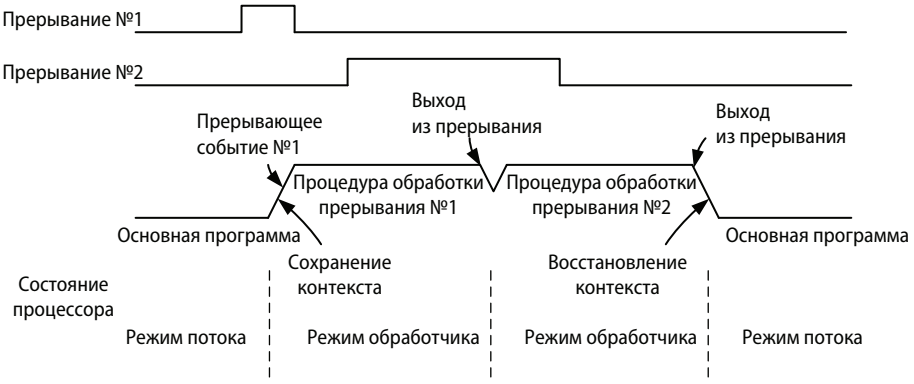


Рис. 9.3. *«Цепочечная» обработка исключений.*

Если исключение возникает в тот момент, когда процессор занят обработкой исключительной ситуации с таким же или более высоким приоритетом, то оно переводится в состояние ожидания (откладывается). После завершения текущего обработчика процессор может приступить к обработке отложенного прерывания. И вот здесь вместо того, чтобы восстанавливать содержимое регистров из стека, а затем снова сохранять его в стеке, процессор, пропуская эти операции, сразу переходит к обработчику отложенного прерывания. В результате значительно уменьшается временной интервал между последовательным выполнением двух обработчиков прерываний.

9.5. «Опоздавшие» исключения

Ещё одним механизмом, увеличивающим эффективность подсистемы обработки прерываний, является поддержка «опоздавших» исключений. Если во время сохранения контекста для обработки какой-либо исключительной ситуации возникнет исключение с более высоким приоритетом, то процессор сначала обработает это «опоздавшее» исключение.

Например, если исключение №1 (меньший приоритет) возникнет на несколько тактов раньше исключения №2 (больший приоритет), то процессор поведёт себя так, как показано на **Рис. 9.4**, т.е. после сохранения контекста будет запущен обработчик исключения №2.

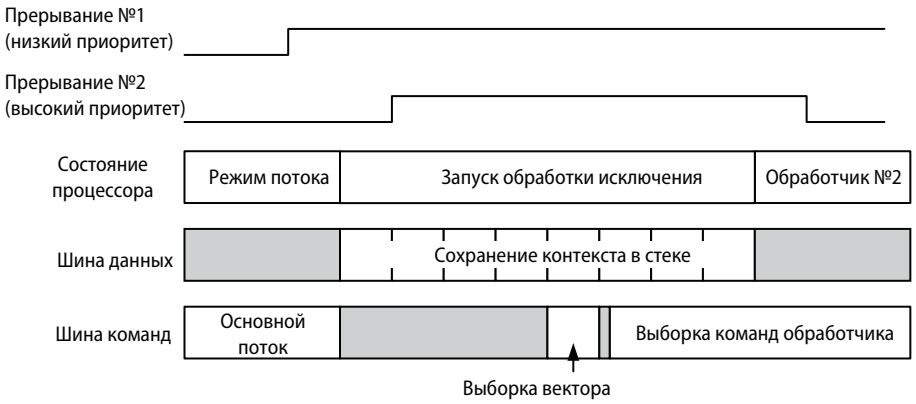


Рис. 9.4. *Обработка «опоздавшего» исключения.*

9.6. Ещё раз о значении EXC_RETURN

При входе в обработчик исключительной ситуации в регистр LR заносится специальное значение, называемое EXC_RETURN, старшие 28 бит которого установлены в 1. При завершении обработчика это значение загружается в счётчик команд, запуская процесс выхода из исключения.

Для выхода из обработчика исключительной ситуации могут использоваться следующие команды:

- `POP/LDM`;
- `LDR` с PC в качестве регистра-приёмника;
- `BX` с любым регистром в качестве операнда.

Биты [31:4] значения EXC_RETURN всегда установлены в 1, а биты [3:0] содержат информацию, необходимую для выполнения операции возврата (**Табл. 9.2**). При входе в обработчик прерывания содержимое LR обновляется автоматически, так что вручную записывать эти значения не нужно.

Таблица 9.2. **Назначение битов EXC_RETURN**

Биты	31:4	3	2	1	0
Описание	0xFFFFFFF	Режим возврата (потока/обработчика)	Стек возврата	Зарезервировано; должен быть 0	Состояние процессора (ARM/Thumb)

Бит 0 значения EXC_RETURN определяет состояние, в котором должен оказаться процессор после возврата из обработчика исключения. Поскольку процессор Cortex-M3 поддерживает только состояние Thumb, бит 0 всегда должен быть установлен в 1. Допустимые для процессора Cortex-M3 значения EXC_RETURN указаны в **Табл. 9.3**.

Таблица 9.3. **Допустимые значения EXC_RETURN для Cortex-M3**

Значение	Описание
0xFFFFFFF1	Возврат в режим обработчика
0xFFFFFFF9	Возврат в режим потока с использованием основного стека при возврате
0xFFFFFFFD	Возврат в режим потока с использованием стека процесса при возврате

Если в режиме потока используется основной стек, то при входе в обработчик первого прерывания в регистр LR будет записано значение 0xFFFFFFF9, а при входе в обработчики последующих вложенных прерываний — 0xFFFFFFF1 (**Рис. 9.5**).

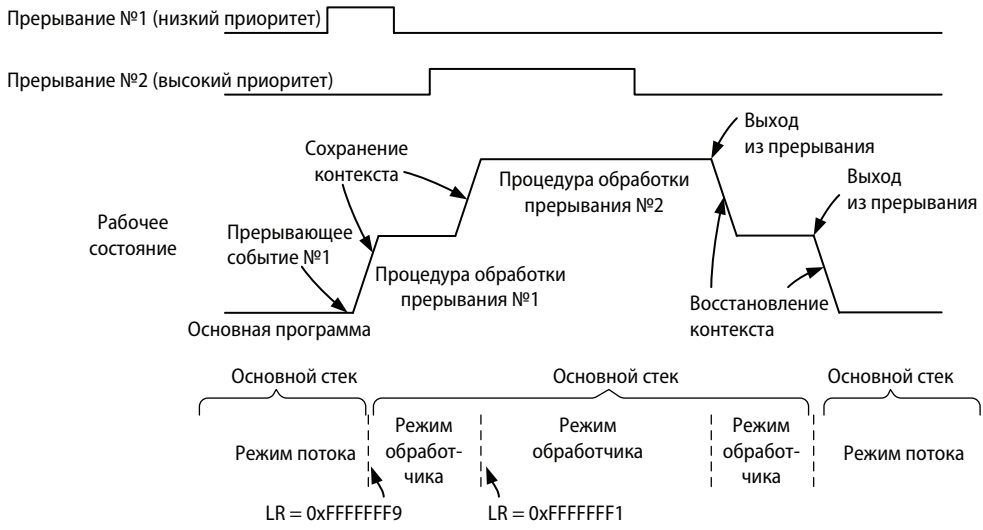


Рис. 9.5. *Загрузка EXC_RETURN в LR при входе в обработчик (в режиме потока используется основной стек).*

Если в режиме потока используется стек процесса, то содержимое LR будет равно 0xFFFFFFFD при входе в обработчик первого прерывания и 0xFFFFFFF1 при входе в обработчики последующих вложенных прерываний, как показано на **Рис. 9.6**.

Из-за используемого формата EXC_RETURN мы не сможем выполнить возврат по адресу из диапазона 0xFFFFFFF0…0xFFFFFFFF. Но, поскольку данные адреса расположены в неисполняемой области памяти, в этом нет ничего страшного.

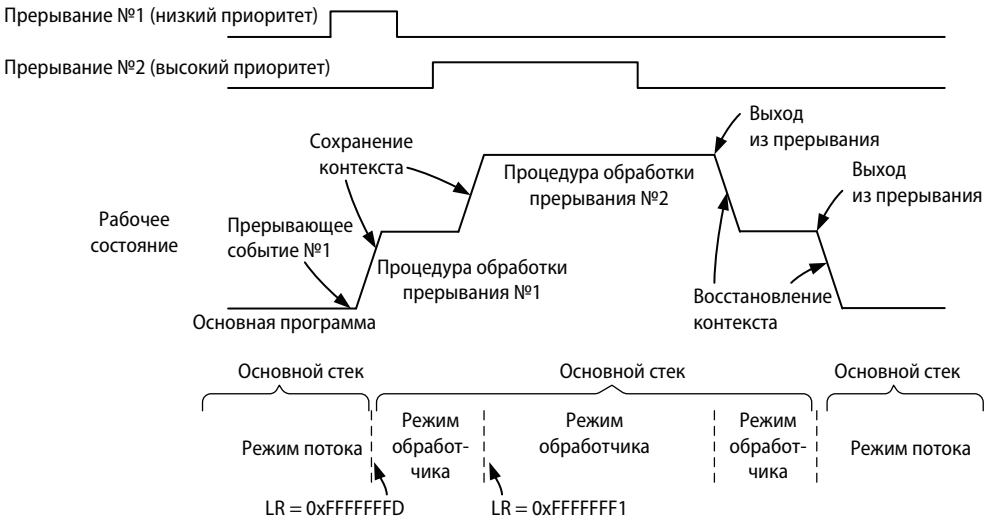


Рис. 9.6. *Загрузка EXC_RETURN в LR при входе в обработчик (в режиме потока используется стек процесса).*

9.7. Задержка обработки прерывания

Термин *задержка обработки прерывания* (interrupt latency) обозначает интервал между появлением запроса прерывания и началом исполнения обработчика этого прерывания. В процессоре Cortex-M3 при условии, что память имеет нулевую латентность, и учитывая, что архитектура шин позволяет одновременно осуществлять выборку вектора и сохранение контекста программы, минимальная величина задержки обработки прерывания составляет 12 тактов. Это время уходит на сохранение регистров в стеке, выборку вектора и выборку первых команд обработчика прерывания. В то же время указанная величина зависит от наличия циклов ожидания при обращении к памяти и ряда других факторов.

При «цепочечной» обработке прерываний время перехода от одного обработчика прерывания к другому может быть уменьшено до 6 тактов, поскольку в этом случае не требуется выполнять лишние операции сохранения/восстановления содержимого регистров.

Если процессор исполняет команду, для выполнения которой требуется несколько тактов, например команду деления, то выполнение подобной команды может быть прервано и возобновлено после завершения обработки прерывания. Это также касается команд загрузки/сохранения двойных слов `LDRD`/`STRD`.

Для уменьшения задержки обработки исключений процессор Cortex-M3 допускает возникновение исключения во время выполнения команд групповой загрузки и сохранения (`LDM`/`STM`). В этом случае при возникновении исключительной ситуации процессор завершит текущее обращение к памяти и запомнит номер следующего регистра в поле ICI регистра xPSR, который будет сохранён в стеке. После завершения обработчика исключения выполнение команды загрузки/восстановления возобновится с того места, где был остановлен процесс пересыл-

ки. В крайнем случае, если прерванная команда групповой загрузки/сохранения была условно выполняемой (т.е. входила в состав IT-блока), то её выполнение будет прекращено, а после завершения обработчика прерывания запущено заново. Это связано с тем, что поле ICI расположено в тех же битах регистра EPSR, которые используются для хранения состояния процесса выполнения команды `IT`.

Кроме того, при наличии незавершённой пересылки по шине, скажем при выполнении буферированной записи, процессор будет ожидать её завершения. Это необходимо для того, чтобы в случае нештатной ситуации обработчик исключения Bus Fault прервал бы корректный процесс.

Разумеется, прерывание может оказаться блокированным, если процессор уже занят обработкой исключительной ситуации с таким же или более высоким приоритетом или если запрос прерывания маскируется регистром маскирования прерываний. В таких случаях обработка прерывания будет отложена до момента снятия блокировки.

9.8. Отказы, связанные с прерываниями

Во время обработки исключений могут возникать различные отказы. Давайте познакомимся с ними.

9.8.1. Сохранение контекста

При возникновении отказа шины во время сохранения регистров в стеке, процесс сохранения контекста прерывается и запускается (или откладывается) выполнение обработчика исключения Bus Fault. Если это исключение запрещено, то будет запущен обработчик исключения Hard Fault. В противном случае, если исключение Bus Fault имеет приоритет, который выше приоритета текущего исключения, то будет запущен обработчик исключения Bus Fault; если же нет — то обработка этого исключения будет отложена до завершения текущего обработчика. Такое состояние, называемое *ошибкой загрузки в стек* (stacking error), индицируется битом STKERR (бит 4) регистра состояния отказа шины BFSR (0xE000ED29).

Если ошибка загрузки в стек была вызвана конфликтом с настройками модуля защиты памяти, то будет запущен обработчик исключения MemManage Fault с одновременной установкой бита MSTKERR (бит 4) регистра состояния отказа управления памятью MMFSR (0xE000ED28). Если исключение MemManage Fault запрещено, то будет запущен обработчик исключения Hard Fault.

9.8.2. Восстановление контекста

При возникновении отказа шины во время извлечения регистров из стека (при возврате из прерывания) процесс восстановления контекста прерывается и запускается (или откладывается) выполнение обработчика исключения Bus Fault. Если это исключение запрещено, то будет запущен обработчик исключения Hard Fault. В противном случае, если приоритет исключения Bus Fault выше приоритета текущего исключения (при наличии вложенных прерываний процессор к этому моменту уже может начать обработку нового исключения), то будет запущен обработчик исключения Bus Fault. Данная ситуация, называемая *ошибкой*

извлечения из стека (unstacking error), индицируется битом UNSTKERR (бит 3) регистра состояния отказа шины BFSR (0xE000ED29).

Аналогично, если ошибка извлечения из стека была вызвана конфликтом с настройками модуля защиты памяти, то будет запущен обработчик исключения MemManage Fault с одновременной установкой бита MUNSTKERR (бит 3) регистра состояния отказа системы управления памятью MMFSR (0xE000ED28). Если исключение MemManage Fault запрещено, то будет запущен обработчик исключения Hard Fault.

9.8.3. Выборка вектора

Если во время выборки вектора произойдёт отказ шины или отказ системы управления памятью, то будет запущен обработчик исключения Hard Fault. Эта ситуация индицируется битом VECTTBL (бит 1) регистра состояния тяжёлого отказа HFSR (0xE000ED2C).

9.8.4. Некорректный возврат

Если значение EXC_RETURN некорректно или не соответствует состоянию процессора (скажем, равно 0xFFFFFFF1 при возврате в режим потока), то будет сгенерировано исключение Usage Fault. Если данное исключение не разрешено, то будет запущен обработчик исключения Hard Fault. При возникновении отказа устанавливается бит INVPC (бит 2) или INVSTATE (бит 1) регистра UFSR (0xE000ED2A), в зависимости от причины отказа.

ГЛАВА 10
ПРОГРАММИРОВАНИЕ CORTEX-M3

10.1. Общие сведения

Для программирования процессора Cortex-M3 можно использовать язык ассемблера, язык Си или любой другой язык высокого уровня, наподобие языка среды LabVIEW компании National Instruments. Программный код большинства встраиваемых приложений может быть целиком написан на Си. Разумеется, встречаются разработчики, предпочитающие программировать исключительно на ассемблере или же использующие в своих проектах смесь языков Си и ассемблера. Процесс создания программного кода и загрузки полученных двоичных образов в конечное устройство в значительной мере определяется используемым набором инструментов. Хотя это и немного выходит за рамки данной книги, в главах 19 и 20 приведены простые примеры использования набора инструментов GNU и среды разработки компании Keil. А в Главе 21 содержатся базовые сведения о применении среды LabVIEW для программирования устройств с ядром Cortex-M3.

10.2. Типичный процесс разработки ПО

Для разработки приложений под процессор Cortex-M3 существует множество различных программных средств. Однако во всех этих средствах при генерации программного кода применяются одни и те же принципы. Как минимум, вам потребуется ассемблер, компилятор с языка Си, компоновщик (редактор связей), а также утилиты для генерации двоичного файла. Процесс создания программ с использованием таких решений компании ARM, как RealView Development Suite (RVDS) или RealView Compiler Tools (RVCT), показан на **Рис. 10.1**.

Помимо указанных компонентов, в состав пакета RVDS также входит большое число разнообразных утилит, включая интегрированную среду разработки (ИСР) и отладчики. Для получения более подробной информации посетите сайт компании ARM (*www.arm.com*).

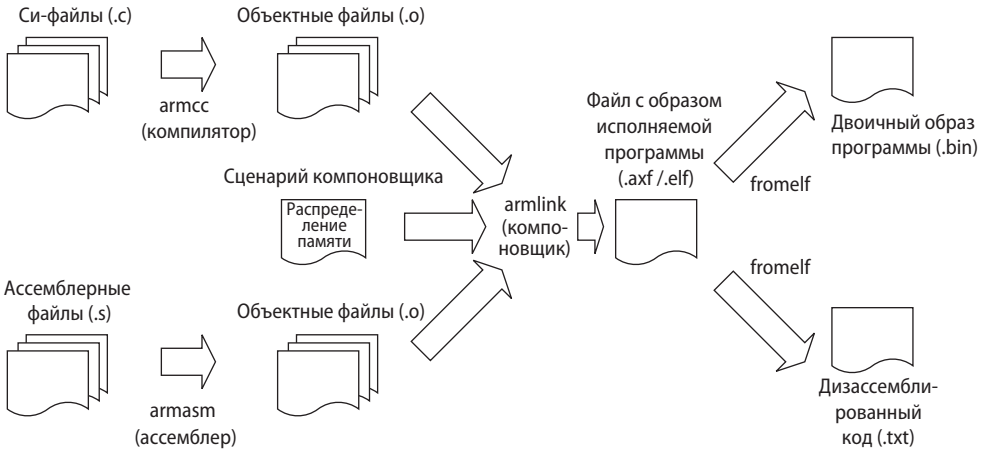


Рис. 10.1. *Создание ПО с использованием средств разработки компании ARM.*

10.3. Использование языка Си

Для тех, кто делает первые шаги в программировании для встраиваемых систем, наилучшим выбором для разработки ПО под процессор Cortex-M3 будет язык Си. Программирование на Си для устройств с процессором Cortex-M3 весьма облегчается тем, что подавляющее большинство производителей микроконтроллеров предоставляют готовые библиотеки для управления периферийными устройствами. В дальнейшем разработчик может подключать эти библиотеки к своим проектам. Поскольку современные компиляторы языка Си генерируют очень эффективный код, лучше программировать на Си, чем тратить кучу времени, пытаясь написать сложные подпрограммы на ассемблере, которые к тому же подвержены ошибкам и обладают меньшей переносимостью.

В этой главе мы сначала рассмотрим процесс создания простой программы на языке Си. Затем коснёмся некоторых аспектов программирования на языке Си, включая использование библиотек драйверов устройств и стандарт CMSIS.

По сравнению с языком ассемблера язык Си обеспечивает лучшую переносимость программ и облегчает реализацию сложных операций. Поскольку язык Си является языком программирования общего назначения, в нём отсутствуют какие-либо средства для определения процесса инициализации процессора. Эта задача решается по-разному, в зависимости от используемого инструментария.

Для начала лучше всего будет ознакомиться с примерами программ. Те, кто работает в средах разработки на базе компилятора ARM C, таких как RVDS или MDK-ARM, могут воспользоваться учебными программами, входящими в состав пакетов. Для тех же, кто предпочитает компилятор GNU, в Главе 19 будет описан процесс создания простой программы в пакете Sourcery G++ компании CodeSourcery.

10.3.1. Компиляция простой Си-программы в пакете RVDS

Нормальная программа на языке Си для процессора Cortex-M3 содержит, как минимум, «основную» процедуру и таблицу векторов. Давайте попробуем написать программу, которая включала бы светодиод в мигающем режиме:

```
#define LED *((volatile unsigned int *)(0xDFFF000C))
int main (void)
{
int i;          /* Счётчик циклов для функции задержки                    */
volatile int j; /* Переменная описана как volatile, чтобы                 */
                /* во время оптимизации компилятор не убрал бы пустые циклы,
                   формирующие задержки*/
  while (1) {
    LED = 0x00;                 /* Переключаем СИД    */
    for (i=0;i<10;i++) {j=0;}   /* Формируем задержку */
    LED = 0x01;                 /* Переключаем СИД    */
    for (i=0;i<10;i++) {j=0;}   /* Формируем задержку */
  }
  return 0;
}
```

Назовём этот файл `blinky.c`. Таблицу векторов мы опишем в отдельном файле, который назовём `vectors.c`. Помимо таблицы векторов, в данном файле также будут описаны несколько пустых обработчиков исключений (их можно будет определить позже при написании конкретного приложения):

```
typedef void(* const ExecFuncPtr)(void) __irq;
extern int __main(void);
/*
* Пустые обработчики исключений
*/
__irq void NMI_Handler(void)
{ while(1); }
__irq void HardFault_Handler(void)
{ while(1); }
__irq void SVC_Handler(void)
{ while(1); }
__irq void DebugMon_Handler(void)
{ while(1); }
__irq void PendSV_Handler(void)
{ while(1); }
__irq void SysTick_Handler(void)
{ while(1); }
__irq void ExtInt0_IRQHandler(void)
{ while(1); }
__irq void ExtInt1_IRQHandler(void)
{ while(1); }
__irq void ExtInt2_IRQHandler(void)
{ while(1); }
__irq void ExtInt3_IRQHandler(void)
```

```
{ while(1); }
#pragma arm section rodata=»exceptions_area»
ExecFuncPtr exception_table[] = { /* Таблица векторов */
    (ExecFuncPtr)0x20002000,
    (ExecFuncPtr)__main,
    NMI_Handler, /* NMI */
    HardFault_Handler,
    0, /* MemManage_Handler в Cortex-M3 */
    0, /* BusFault_Handler в Cortex-M3 */
    0, /* UsageFault_Handler в Cortex-M3 */
    0, /* Зарезервировано */
    0, /* Зарезервировано */
    0, /* Зарезервировано */
    0, /* Зарезервировано */
    SVC_Handler,
    0, /* DebugMon_Handler в Cortex-M3 */
    0, /* Зарезервировано */
    PendSV_Handler,
    SysTick_Handler,
    /* Внешние прерывания */
    ExtInt0_IRQHandler,
    ExtInt1_IRQHandler,
    ExtInt2_IRQHandler,
    ExtInt3_IRQHandler
};
#pragma arm section
```

При использовании компилятора RVDS для компиляции указанных файлов можно использовать следующие командные строки:

```
$> armcc -c -g -W blinky.c -o blinky.o
$> armcc -c -g -W vectors.c -o vectors.o
```

Для генерации образа программы необходима отдельная программа, называемая компоновщиком, или линкером. Чтобы сообщить этой программе о распределении памяти и разместить таблицу векторов в начале образа, используется файл сценария компоновщика led.scat, представляющий собой обычный текстовый файл:

```
#define HEAP_BASE  0x20001000
#define STACK_BASE 0x20002000
#define HEAP_SIZE  ((STACK_BASE-HEAP_BASE)/2)
#define STACK_SIZE ((STACK_BASE-HEAP_BASE)/2)
LOAD_REGION 0x00000000 0x00200000
{
  VECTORS 0x0 0xC0
  {
    ; Provided by the user in vectors.c
    * (exceptions_area)
  }
  CODE 0xC0 FIXED
  {
    * (+RO)
  }
```

```
  DATA 0x20000000 0x00010000
  {
    * (+RW, +ZI)
  }
  ;; Heap starts at 4KB and grows upwards
  ARM_LIB_HEAP HEAP_BASE EMPTY HEAP_SIZE
  {
  }
  ;; Stack starts at the end of the 8KB of RAM
  ;; And grows downwards for 2KB
  ARM_LIB_STACK STACK_BASE EMPTY -STACK_SIZE
  {
  }
}
```

А командная строка для запуска линкера будет иметь вид:

```
$> armlink -scatter led.scat «--keep=vectors.o(exceptions_area)»
blinky.o vectors.o -o blinky.elf
```

В результате мы получим образ исполняемой программы `blinky.elf`. Используя утилиту `fromelf`, мы можем преобразовать его в двоичный файл, а также дизассемблировать:

```
/* Создаём двоичный файл */
$> fromelf --bin blinky.elf -output blinky.bin
/* Создаём дизассемблированный файл */
$> fromelf -c blinky.elf > list.txt
```

В предыдущих процессорах ARM, поддерживающих два состояния (ARM и Thumb), код для каждого из состояний необходимо было компилировать по отдельности. Процессор Cortex-M3 всегда находится в состоянии Thumb, что значительно упрощает управление файлами, входящими в проект.

При разработке приложений на языке Си рекомендуется использовать стек, выровненный на границу двойного слова (определяется битом STKALIGN регистра CCR контроллера NVIC). Начиная со 2-й ревизии процессора Cortex-M3, этот бит установлен по умолчанию. При использовании процессора 1-й ревизии указанный бит необходимо устанавливать вручную, скажем в самом начале программы. Более подробно об использовании данного бита было рассказано в Главе 9.

```
SCB->CCR = SCB->CCR | 0x200; /* Установили STKALIGN */
/* SCB->CCR определено в библиотеке драйвера устройства */
```

Следующие строки выполняют ту же операцию, но уже без использования CMSIS-совместимого драйвера устройства:

```
#define NVIC_CCR *((volatile unsigned long *)(0xE000ED14))
NVIC_CCR = NVIC_CCR | 0x200; /* Установили STKALIGN */
```

Выравнивание стека на границу двойного слова обеспечивает соответствие кода программы соглашениям AAPCS (более подробно об этом будет рассказано в Главе 12).

10.3.2. Компиляция простой Си-программы в пакете MDK-ARM

Приведённую выше программу можно скомпилировать и в пакете MDK-ARM компании Keil. Необходимо будет только скорректировать формат командной строки и некоторые определения в файле сценария компоновщика. Так, файл `led.scat` должен выглядеть следующим образом:

```
#define HEAP_BASE 0x20001000
#define STACK_BASE 0x20002000
#define HEAP_SIZE ((STACK_BASE-HEAP_BASE)/2)
#define STACK_SIZE ((STACK_BASE-HEAP_BASE)/2)
LOAD_REGION 0x00000000 0x00200000
{
  VECTORS 0x0 0xC0
  {
    ; Provided by the user in vectors.c
    * (exceptions_area)
  }

   CODE 0xC0 FIXED
  {
    * (+RO)
  }

  DATA 0x20000000 0x00010000
  {
    * (+RW, +ZI)
  }
  ;; Heap starts at 4KB and grows upwards
  Heap_Mem HEAP_BASE EMPTY HEAP_SIZE
  {
  }
  ;; Stack starts at the end of the 8KB of RAM
  ;; And grows downwards for 2KB
  Stack_Mem STACK_BASE EMPTY -STACK_SIZE
  {
  }
}
```

А для компиляции программы можно создать пакетный файл DOS, содержащий следующие команды:

```
SET PATH=C:\Keil\ARM\BIN40\;%PATH%
SET RVCT40INC=C:\Keil\ARM\RV31\INC
SET RVCT40LIB=C:\Keil\ARM\RV31\LIB
SET CPU_TYPE=Cortex-M3
SET CPU_VENDOR=ARM
SET UV2_TARGET=Target 1
SET CPU_CLOCK=0x00000000
C:\Keil\ARM\BIN40\armcc -c -O3 -W -g -Otime --device DLM vectors.c
C:\Keil\ARM\BIN40\armcc -c -O3 -W -g -Otime --device DLM blinky.c
```

```
C:\Keil\ARM\BIN40\armlink --device DLM «--keep=Startup.o(RESET)»
  «--first=Startup.o(RESET)» -scatter led.scat --map vectors.o
  blinky.o -o blinky.elf
C:\Keil\ARM\BIN40\fromelf --bin blinky.elf -o blinky.bin
```

Конечно же, для создания и компиляции проектов гораздо удобнее использовать ИСР µVision, нежели командную строку. Для начинающих разработчиков, собирающихся разрабатывать ПО с использованием среды MDK-ARM компании Keil, предназначена Глава 20 данной книги.

10.3.3. Отображённые в память регистры и язык Си

Существуют различные способы для обращения к регистрам периферийных устройств, отображённых в память процессора, из программ на языке Си. Продемонстрируем их на примере регистров системного таймера (SYSTICK) процессора Cortex-M3. Это обычный 24-битный таймер, который имеет всего 4 регистра. Конкретно функциональные возможности таймера SYSTICK будут рассмотрены в Главе 14. В предыдущих примерах мы уже использовали простейший метод обращения к регистрам, заключающийся в описании каждого регистра как указателя. Опишем таким образом и регистры модуля SYSTICK (**Рис. 10.2**).

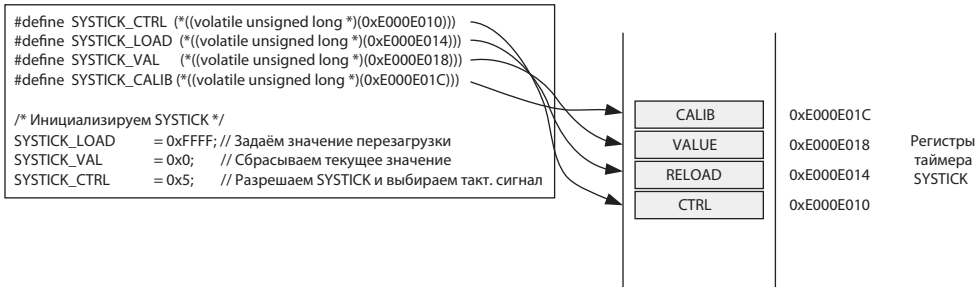


Рис. 10.2. *Обращение к регистрам по указателю.*

Этот метод можно немного модифицировать, написав макроопределение, преобразующее значение адреса регистра в указатель. В результате текст программы будет выглядеть немного иначе, однако генерируемый код будет точно таким же, как и в предыдущей реализации (**Рис. 10.3**).

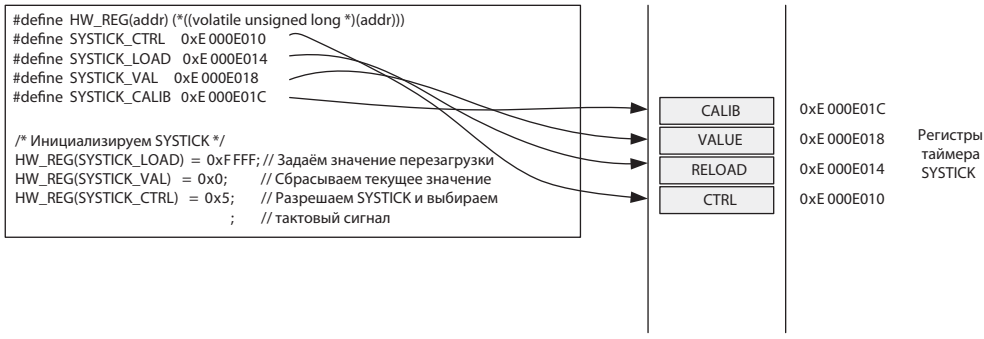


Рис. 10.3. *Обращение к регистрам по указателю (альтернативный вариант).*

Второй метод заключается в описании всех регистров периферийного устройства в виде структуры с последующим определением указателя на данную структуру (**Рис. 10.4**). Этот метод используется в CMSIS-совместимых библиотеках драйверов устройств.

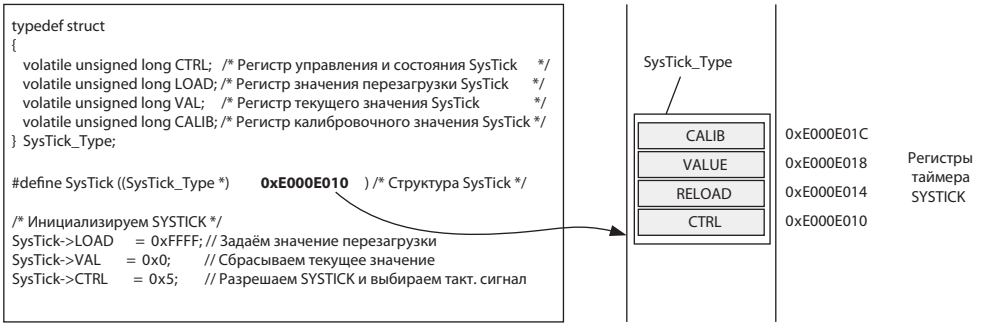


Рис. 10.4. *Обращение к регистрам как к полям структуры.*

Третий метод тоже основан на использовании структуры, однако базовый адрес периферийного устройства определяется в файле сценария компоновщика и подставляется в код на этапе компоновки программы (**Рис. 10.5**).

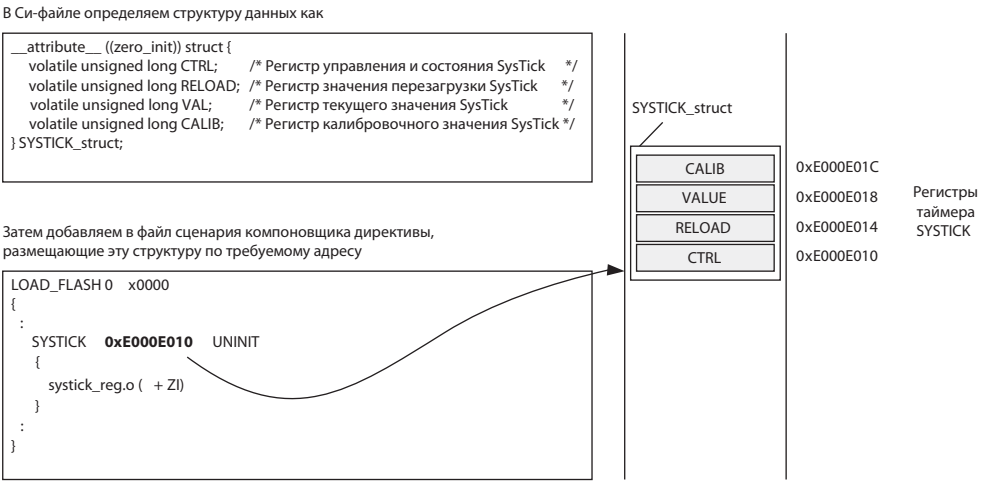


Рис. 10.5. *Определение базового адреса периферийного устройства в файле сценария компоновщика.*

В последнем случае блок регистров периферийного устройства должен быть описан в программе как внешний указатель. При этом обращение к конкретным регистрам производится так же, как и при использовании второго метода.

Первый метод, применение которого показано на **Рис. 10.2** и **Рис. 10.3**, является самым простым, но в то же время и наименее эффективным по сравнению с прочими методами, поскольку значения адресов регистров хранятся в виде констант. В итоге увеличивается размер кода программы и может замедлиться её выполнение, поскольку для считывания значений адресов потребуются дополни-

тельные обращения к памяти программ. В то же время при использовании только одного регистра для управления периферийным устройством эффективность данного метода оказывается такой же, как и других методов.

Второй метод (использование указателя на структуру) применяется, пожалуй, чаще всего. Этот метод позволяет обращаться ко всем регистрам периферийного устройства, используя единственную константу — базовый адрес устройства. Для обращения к конкретным регистрам в данном случае можно задействовать режим адресации с непосредственным смещением. Указанный метод применяется в стандарте CMSIS, который будет рассмотрен в следующем разделе.

Третий метод, в котором задействуется файл сценария компоновщика (**Рис. 10.5**), по эффективности эквивалентен 2-му методу, однако обладает гораздо меньшей переносимостью из-за необходимости использования отдельного файла сценария (формат и синтаксис этого файла зависят от конкретного средства разработки). Применение данного метода оправдано в том случае, если вы разрабатываете библиотеку драйверов для периферийного устройства, которое будет использоваться в разных изделиях и базовый адрес которого в каждом случае становится известным только на этапе компоновки.

10.3.4. Встроенные функции

Использование языка высокого уровня, такого как Си, значительно ускоряет разработку приложений. Однако в ряде случаев возникает потребность в использовании определённых команд, которые не могут быть сгенерированы с помощью стандартных конструкций языка. Поэтому в некоторых компиляторах Си имеются встроенные (intrinsic) функции, позволяющие использовать в программе такие команды. Обращение к встроенным функциям производится так же, как и к обычным пользовательским функциям. В качестве примера в **Табл. 10.1** перечислены встроенные функции, поддерживаемые компиляторами ARM.

Таблица 10.1. **Встроенные функции, поддерживаемые компиляторами ARM**

Команда ассемблера	Встроенная функция
CLZ	unsigned char __clz(unsigned int val)
CLREX	void __clrex(void)
CPSID I	void __disable_irq(void)
CPSIE I	void __enable_irq(void)
CPSID F	void __disable_fiq(void)
CPSIE F	void __enable_fiq(void)
LDREX/LDREXB/LDREXH	unsigned int __ldrex(volatile void *ptr)
LDRT/LDRBT/LDRSBT/LDRHT/LDRSHT	unsigned int __ldrt(const volatile void *ptr)
NOP	void __nop(void)
RBIT	unsigned int __rbit(unsigned int val)
REV	unsigned int __rev(unsigned int val)
ROR	unsigned int __ror(unsigned int val, unsigned int shift)
SSAT	int __ssat(int val, unsigned int sat)
SEV	void __sev(void)

Таблица 10.1. **Встроенные функции, поддерживаемые компиляторами ARM** *(продолжение)*

Команда ассемблера	Встроенная функция
STREX/STREXB/STREXH	int __strex(unsigned int val, volatile void *ptr)
STRT/STRBT/STRHT	void int __strt(unsigned int val, const volatile void *ptr)
USAT	int __usat(unsigned int val, unsigned int sat)
WFE	void __wfe(void)
WFI	void __wfi(void)
BKPT	void __breakpoint(int val)

10.3.5. Встроенный и inline-ассемблер

Вместо использования встроенных функций мы можем напрямую вставлять команды ассемблера в текст программ на Си. Как правило, потребность в этом возникает при управлении системой на низком уровне или же при необходимости реализации критичной ко времени исполнения процедуры. Большинство компиляторов языка Си для процессоров ARM позволяют внедрять ассемблерный код в текст программы, используя *inline-ассемблер*.

Компилятор ARM позволяет выполнять вставку ассемблерных команд в текст программы на языке Си. Традиционно для этого используется inline-ассемблер, однако inline-ассемблер компилятора RealView не поддерживает команды Thumb-2. Начиная с версии 3.0, в компиляторе появилась поддержка так называемого встроенного ассемблера, поддерживающего команды из набора Thumb-2. Например, вы можете вставить в свою программу ассемблерную функцию, описав её следующим образом:

```
__asm void SetFaultMask(unsigned int new_value)
{
    // Используем ассемблерный код
    MSR FAULTMASK, new_value // Заносим новое значение в FAULTMASK
    BX LR                    // Возвращаемся в вызывающую программу
}
```

Подробно встроенный ассемблер компилятора RealView описан в руководстве [6]. Применительно к процессору Cortex-M3 встроенный ассемблер полезен для решения таких задач, как непосредственная манипуляция стеком и реализация критичных ко времени выполнения процедур (кодеки).

10.4. Стандарт CMSIS

10.4.1. Предпосылки появления стандарта CMSIS

На сегодняшний день микроконтроллеры с процессором Cortex-M3 активно захватывают рынок встраиваемых приложений — появляется всё больше и больше устройств, основанных на этом процессоре, а также программного обеспечения, поддерживающего данные устройства. К концу 2008 года процессор Cortex-M3 поддерживали уже более пяти компиляторов разных производителей и более 15 различных встраиваемых ОС. Помимо этого, на рынке присутствует

множество компаний, предлагающих собственные программные решения, такие как кодеки, библиотеки обработки данных, а также различные средства программирования и отладки. Стандарт CMSIS (Cortex Microcontroller Software Interface Standard — стандарт программного интерфейса микроконтроллеров с ядром Cortex), разработанный компанией ARM, позволяет пользователям микроконтроллеров с ядром Cortex-M3 извлечь максимальную выгоду из всех этих программных решений и быстро разрабатывать собственные высоконадёжные встраиваемые приложения (**Рис. 10.6**).

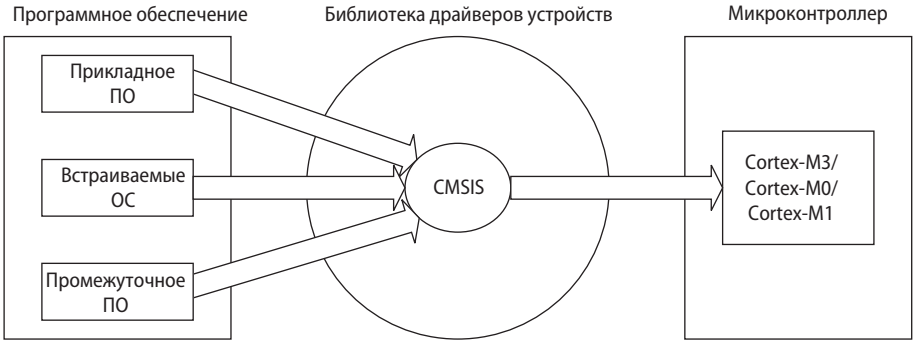


Рис. 10.6. *Роль CMSIS в разработке встраиваемых приложений.*

Разработка стандарта CMSIS, который позволил бы улучшить потребительские свойства и функциональную совместимость программного обеспечения для микроконтроллеров ARM, была начата в 2008 году. Этот стандарт нашёл применение в библиотеках драйверов, предлагаемых производителями микроконтроллеров, обеспечивая стандартизованный программный интерфейс для доступа к функциональным возможностям процессора Cortex-M3, а также различные системные функции и функции ввода/вывода. Подобные библиотеки также поддерживаются компаниями-разработчиками ПО, включая разработчиков встроенных ОС, а также производителей компиляторов.

При создании CMSIS преследовались следующие цели:

- улучшить переносимость и степень повторного использования готового кода;
- предоставить поставщикам программных решений возможность разработки продуктов, которые могли бы без всяких проблем работать с библиотеками различных производителей микроконтроллеров;
- уменьшить время разработки ПО за счёт простого и стандартизованного программного интерфейса;
- обеспечить возможность создания встроенного ПО с использованием различных компиляторов;
- исключить проблемы с совместимостью программных модулей из различных источников.

Первая версия стандарта CMSIS была представлена на суд общественности в 4-м квартале 2008 года и в настоящее время входит в состав библиотек драйверов устройств, предлагаемых различными производителями микроконтроллеров. Стандарт CMSIS также поддерживает процессорное ядро Cortex-M0.

10.4.2. Области стандартизации

Действие стандарта CMSIS распространяется на следующие области:

- *Уровень аппаратной абстракции для регистров процессора Cortex-M*. Это касается унифицированных обозначений регистров контроллера NVIC, блока управления системой SBC, системного таймера SYSTICK, модуля MPU, а также ряда функций для использования возможностей контроллера NVIC и ядра процессора.
- *Унифицированные названия системных исключений*. Это позволяет операционным системам и промежуточному ПО использовать системные исключения без всяких проблем с совместимостью.
- *Унифицированная структура заголовочных файлов*. Это облегчает переход пользователей на новые микроконтроллеры с ядром Cortex и улучшает переносимость ПО.
- *Единообразная инициализация системы*. Каждый производитель микроконтроллеров предоставляет в библиотеке драйверов для своих микроконтроллеров функцию `SystemInit()`, выполняющую основные операции по конфигурированию микроконтроллера, такие как инициализация системы тактирования. И опять же, это помогает новичкам освоить микроконтроллеры с ядром Cortex-M и облегчает перенос ПО.
- *Стандартные встроенные функции*. Встроенные функции, как правило, используются для вызова команд процессора, которые не могут быть сгенерированы стандартными конструкциями языка Си[1)]. При наличии стандартных встроенных функций значительно увеличиваются возможности повторного использования кода и переносимость ПО.
- *Общие функции для передачи данных*. Это набор интерфейсных функций для типовых коммуникационных интерфейсов, таких как UART, Ethernet и SPI. Благодаря наличию в библиотеках драйверов устройств таких функций улучшается переносимость встроенного ПО и увеличивается возможность повторного использования кода. На момент написания книги данные функции ещё находились в процессе разработки.
- *Унифицированный метод определения тактовой частоты системы*. В драйвере устройства определена глобальная переменная `SystemFrequency`, с помощью которой встраиваемые ОС могут настраивать модуль SYSTICK в соответствии с тактовой частотой системы.

В стандарте CMSIS определены основные требования к ПО, обеспечивающие переносимость и возможность повторного использования кода. Производители микроконтроллеров могут включать в библиотеки дополнительные функции для поддержки своей периферии. Соответственно, использование CMSIS никоим образом не ограничивает возможности встраиваемых устройств.

10.4.3. Структура CMSIS

Собственно стандарт CMSIS описывает несколько взаимосвязанных уровней.

[1)] Возможности языка Си/Си++ определены в стандарте ISO/IEC 14882, подготовленном международной организацией стандартов (ISO) совместно с международной электротехнической комиссией (IEC).

Core Peripheral Access Layer (уровень доступа к периферии ядра)

- Определения имён, определения адресов, а также вспомогательные функции для доступа к регистрам и периферии ядра процессора.

Middleware Access Layer (уровень доступа промежуточного ПО)

- Общие методы для обращения к периферийным устройствам для разработчиков ПО (в процессе разработки).
- Поддержка коммуникационных интерфейсов, таких как Ethernet, UART и SPI.
- Позволяет переносимому ПО осуществлять обмен данными на любом микроконтроллере с ядром Cortex, поддерживающем требуемый коммуникационный интерфейс.

Device Peripheral Access Level (уровень доступа к периферии устройства, определяется производителем микроконтроллера)

- Определения имён, определения адресов и вспомогательные функции для обращения к периферийным устройствам.

Access Functions for Peripherals (функции доступа к периферии, определяются производителем микроконтроллера)

- Опциональные дополнительные функции для работы с периферийными устройствами.

Место каждого из уровней в иерархии CMSIS показано на **Рис. 10.7**.

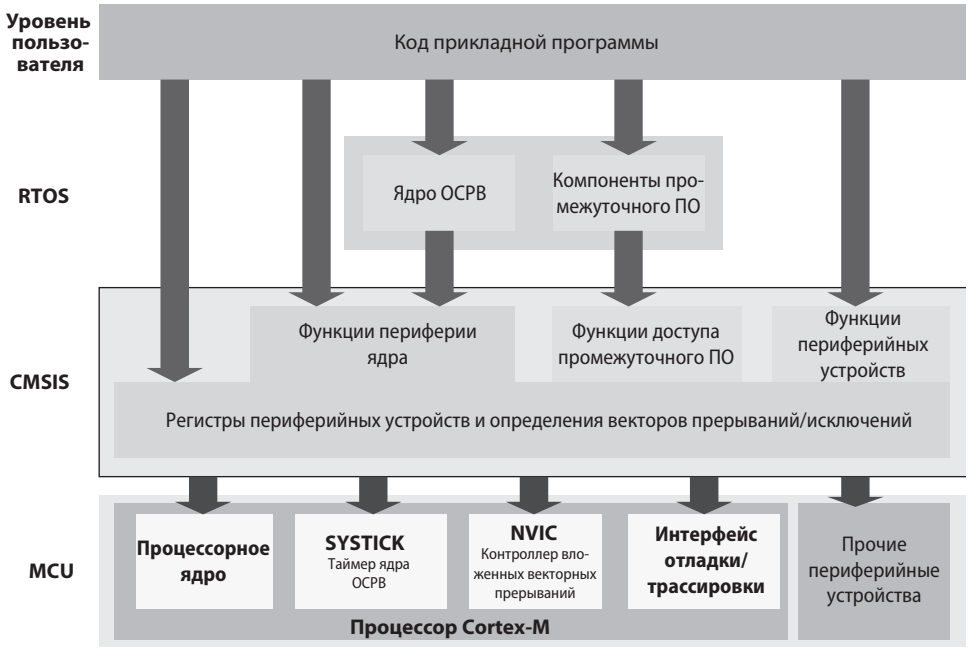


Рис. 10.7. *Структура CMSIS.*

10.4.4. Использование стандарта CMSIS

Поскольку стандарт CMSIS, условно говоря, «внедрён» в библиотеки драйверов устройств, то для его использования в проекте не требуется предпринимать никаких специальных мер. Для каждого микроконтроллера производитель предоставляет заголовочный файл <device>.h, который, в свою очередь, содержит директивы вставки дополнительных заголовочных файлов, требуемых библиотекой драйвера устройства, в том числе и заголовочный файл уровня доступа к периферии ядра, разработанный компанией ARM (**Рис. 10.8**).

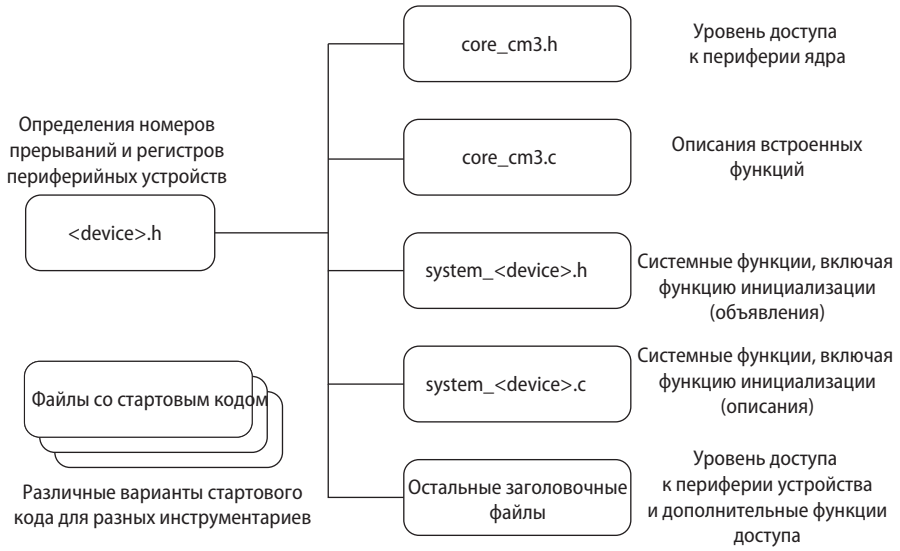


Рис. 10.8. *Файлы CMSIS.*

Файл `core_cm3.h` содержит определения регистров периферийных устройств и описания функций доступа к периферии процессора Cortex-M3, такой как контроллер NVIC, регистры управления системой и регистры системного таймера SYSTICK. Файл `core_cm3.h` также содержит объявления встраиваемых функций стандарта CMSIS, позволяющих использовать команды процессора, которые не могут быть сгенерированы при использовании стандартных конструкций языка. Кроме того, этот файл содержит описание функции для вывода отладочной информации через модуль ITM.

Следует заметить, что имена некоторых встроенных функций CMSIS могут совпасть с именами встроенных функций компилятора, однако функции CMSIS не зависят от используемого компилятора.

Файл `core_cm3.c` содержит описания встроенных функций CMSIS, которые не могут быть реализованы в файле `core_cm3.h` в виде макроопределений.

Файл `system_<device>.h` содержит объявление, а файл `system_<device>.c` — описание функции `SystemInit()`, предназначенной для инициализации системы (реализация этой функции зависит от конкретного микроконтроллера).

В самом файле <device>.h содержатся определения номеров прерываний и регистров периферийных устройств конкретного устройства.

Кроме того, CMSIS-совместимые драйверы устройств также содержат стартовый код (с таблицей векторов) для всех поддерживаемых компиляторов, а также CMSIS-версии встроенных функций, что позволяет встроенному ПО использовать все возможности ядра независимо от применяемого компилятора.

Примеры использования стандарта CMSIS можно найти на сайтах производителей микроконтроллеров. Подобные примеры также можно найти в самих библиотеках драйверов. Кроме того, вы можете загрузить с сайта *www.onarm.com* пакет CMSIS, содержащий примеры и документацию. В этом же пакете можно найти документацию на функции общего назначения.

Простейший пример использования стандарта CMSIS при разработке собственного приложения показан на **Рис. 10.9**. Чтобы воспользоваться функциями CMSIS для конфигурирования прерываний и исключений, необходимо задействовать константы, определённые в файле <device>.h. Значения указанных констант отличаются от номеров исключений, используемых регистрами ядра, например регистром IPSR. В CMSIS для системных исключений используются отрицательные значения, а для прерываний периферии — положительные.

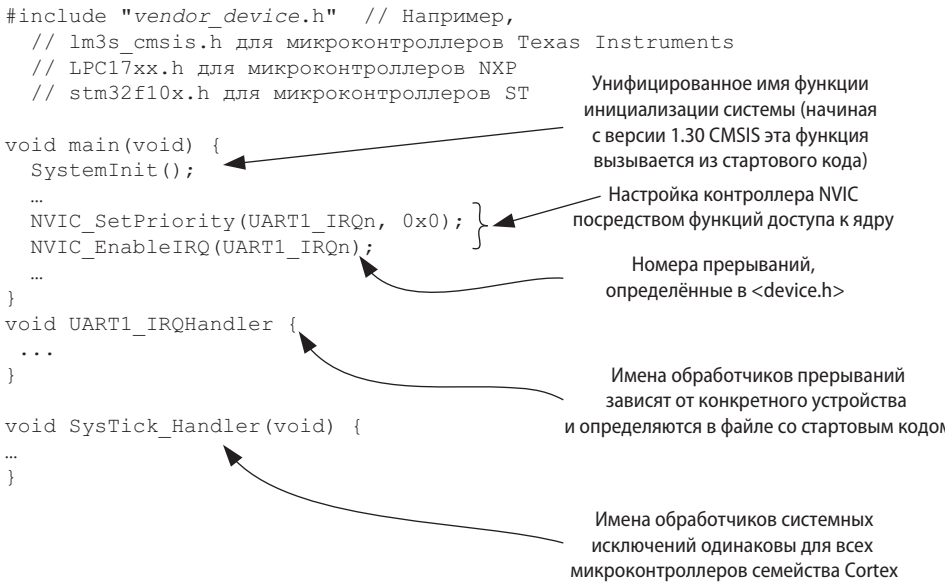


Рис. 10.9. *Пример использования CMSIS.*

Чтобы создать переносимое приложение, вы должны использовать для обращения к процессору и периферийным устройствам функции доступа к ядру и функции промежуточного ПО соответственно. Это позволит вам перенести приложения с одного микроконтроллера на другой с минимальными усилиями.

Более подробно CMSIS-функции, как общего назначения, так и встроенные, описаны в Приложении Ж.

10.4.5. Выгода от использования CMSIS

И всё же, что стандарт CMSIS даёт конечному пользователю?

Прежде всего, стандарт CMSIS значительно улучшает переносимость и увеличивает возможность повторного использования программного кода. Следование этому стандарту не только облегчает переход между разными микроконтроллерами с ядром Cortex-M3, но и позволяет ускорить процесс переноса программного обеспечения с процессора Cortex-M3 на другие процессоры линейки Cortex-M, тем самым сокращая время вывода продукта на рынок.

Для разработчиков встраиваемых ОС и промежуточного ПО выгода от использования стандарта CMSIS гораздо существеннее. Применение этого стандарта позволит обеспечить совместимость их продукции с драйверами устройств для микроконтроллеров самых разных производителей, в том числе и для тех устройств, которые пока существуют только на бумаге (**Рис. 10.10**). Если не использовать CMSIS, то поставщики ПО должны либо предоставлять небольшую библиотеку функций для работы с ядром Cortex-M3, либо разрабатывать множество конфигураций для своей продукции, позволяющих ей работать с библиотеками от различных производителей микроконтроллеров.

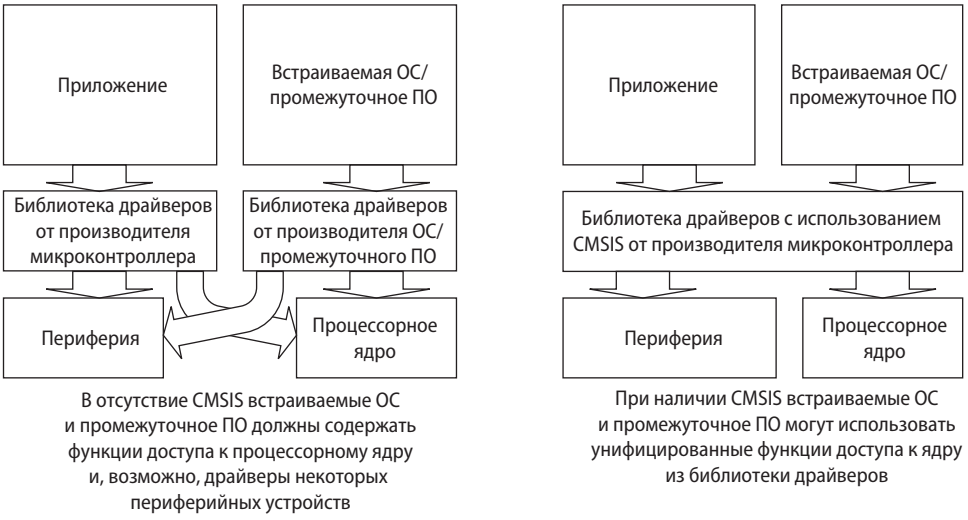


Рис. 10.10. *Стандарт CMSIS как средство против дублирования кода.*

Стандарт CMSIS очень нетребователен к памяти (необходимо менее 1 Кбайт для всех функций доступа к ядру и несколько байтов ОЗУ). Он также позволяет избежать дублирования кода драйвера периферии ядра при повторном использовании кода из другого проекта.

Поскольку стандарт CMSIS поддерживается многими разработчиками компиляторов, для компиляции встраиваемого ПО можно использовать самые разные компиляторы. Это позволяет разрабатывать встраиваемые ОС и промежуточное ПО, которые были бы независимы как от производителей микроконтроллеров, так и от производителей средств разработки. До появления стандарта

CMSIS в каждом компиляторе были, как правило, свои собственные встроенные функции, что затрудняло перенос ПО на другой компилятор.

Поскольку все CMSIS-совместимые библиотеки драйверов устройств имеют одинаковую структуру, значительно облегчается изучение новых микроконтроллеров Cortex-M3 (программный интерфейс практически не меняется, т.е. нет необходимости изучать новый API).

В тестировании стандарта CMSIS принимало участие множество людей; кроме того, он совместим с требованиями MISRA[1]. Всё это позволяет уменьшить затраты на тестирование ПО, необходимое при разработке собственных функций работы с ядром или контроллером NVIC.

10.5. Использование ассемблера

Небольшие проекты могут быть целиком и полностью описаны на ассемблере. Однако следует учесть, что для начинающих разработчиков это часто оказывается довольно сложной задачей. Используя ассемблер, вы можете по максимуму оптимизировать код приложения так, как считаете необходимым; правда, такой подход чреват увеличением времени разработки, а также появлением различных ошибок. Кроме того, обработка сложных структур данных на языке ассемблера, так же как и управление библиотеками функций, может оказаться весьма нетривиальной задачей. И тем не менее, даже при написании приложения на языке Си может возникнуть необходимость в использовании ассемблера:

- для реализации функций, которые не могут быть написаны на Си, например функций, напрямую манипулирующих содержимым стека или использующих специальные команды, которые не могут быть получены с помощью стандартных конструкций языка;
- для написания процедур, критичных ко времени выполнения;
- при ограниченных объёмах памяти, что требует написания части программы на ассемблере для получения как можно более компактного кода.

10.5.1. Интерфейс между ассемблером и Си

Рано или поздно ассемблерному коду приходится взаимодействовать с программой, написанной на языке Си:

- при использовании встроенного ассемблера (или, в случае средств GNU, inline-ассемблера) в тексте программы на Си;
- при вызове из программы на Си функции или процедуры, написанной на ассемблере и реализованной в отдельном файле;
- при вызове из ассемблерной программы процедуры или функции, написанной на Си.

При этом необходимо чётко представлять себе, каким образом осуществляется передача параметров в вызываемую функцию и возврат результата её работы в вызывающую программу. Механизм такого взаимодействия детально описан в стандарте AAPCS [5].

[1] Motor Industry Software Reliability Association — Ассоциация по вопросам надёжности программного обеспечения в автомобильной промышленности. — *Примеч. пер.*

В простейших случаях для передачи параметров используются регистры R0…R3 процессора, при этом в регистре R0 передаётся 1-й параметр, в регистре R1 — 2-й и т.д. Аналогично, регистр R0 используется для передачи значения, возвращаемого вызываемой функцией. Регистры R0…R3 и R12 могут изменяться в функции или процедуре, тогда как содержимое регистров R4…R11 должно сохраняться при входе в подпрограмму и восстанавливаться при выходе из неё. Обычно для этой цели используют стек.

Необходимо отметить, что примеры, имеющиеся в книге, не в полной мере следуют рекомендациям стандарта AAPCS — это было сделано для того, чтобы облегчить их понимание. При вызове Си-функции из ассемблерной программы нужно учитывать возможность изменения регистров R0…R3 и R12. Если содержимое этих регистров потребуется в дальнейшем, то их необходимо сохранить в стеке перед вызовом функции и восстановить из стека после возврата из неё. Поскольку в примерах вызываются, главным образом, только ассемблерные процедуры и функции, которые изменяют лишь некоторые регистры или же восстанавливают содержимое регистров перед возвратом, сохранять регистры R0…R3 и R12 нет необходимости.

10.5.2. Программирование на ассемблере — первые шаги

В этой главе приводится несколько учебных программ, написанных на ассемблере. Конечно же, в своей практической деятельности вы в большинстве случаев будете программировать на Си. Однако изучая ассемблерные программы, можно лучше разобраться в том, как использовать процессор Cortex-M3. Приведённые ниже примеры написаны на ассемблере ARM (armasm) из состава пакета разработки RVDS. Ассемблер, имеющийся в пакете программ MDK-ARM компании Keil, задействует немного другие параметры командной строки. При использовании других ассемблеров потребуется изменить соответствующим образом формат файла и синтаксис команд. Необходимо также отметить, что вам, как правило, не нужно беспокоиться о создании стартового кода на ассемблере, поскольку многие средства разработки предоставляют его уже в готовом виде.

Наша первая программа будет выглядеть следующим образом:

```
STACK_TOP EQU 0x20002000 ; Начальное значение SP (константа)
     AREA |Header Code |, CODE
     DCD STACK_TOP ; Вершина стека
     DCD Start     ; Вектор сброса
     ENTRY         ; Обозначает точку входа в программу
Start ; Начало основной программы
     ; Инициализируем регистры
     MOV  r0, #10  ; Начальное значение счётчика цикла
     MOV  r1, #0   ; Начальное значение результата
     ; Будем вычислять 10 + 9 + 8 +...+ 1
loop
     ADD  r1, r0   ; R1 = R1 + R0
     SUBS r0, #1   ; Декрементируем R0, обновляя флаги (суффикс S)
     BNE  loop     ; Если R0 не равен 0, переходим к метке loop
     ; Результат сложения - в R1
deadloop
```

```
B    deadloop ; Бесконечный цикл
END           ; Конец файла
```

В этой простейшей программе задаётся начальное значение указателя стека SP, начальное значение счётчика команд PC, после чего в цикле выполняются требуемые вычисления.

При использовании инструментальных средств пакета RealView компании ARM ассемблирование данной программы можно выполнить с помощью следующей командной строки:

```
$> armasm --cpu cortex-m3 -o test1.o test1.s
```

Ключ `-o` применяется для указания имени выходного объектного файла (`test1.o`). Для создания образа исполняемой программы (ELF) необходимо использовать компоновщик, который запускается из командной строки со следующими параметрами:

```
$> armlink --rw_base 0x20000000 --ro_base 0x0 --map -o test1.elf test1.o
```

Параметр `--ro _ base` определяет начало области, предназначенной только для чтения (ПЗУ памяти программ); в данном случае эта область начинается с адреса 0x0. Аналогично, параметр `--rw _ base` определяет начальный адрес области памяти, доступной для чтения и для записи (память данных). Учтите, что в нашей программе нет данных, которые располагались бы в ОЗУ. Ключ `--map` указывает на необходимость генерации map-файла, который может пригодиться для изучения распределения памяти в скомпилированном образе.

И в завершение, создадим двоичный образ:

```
$> fromelf --bin --output test1.bin test1.elf
```

Чтобы убедиться в том, что полученный образ соответствует исходному тексту, можно сгенерировать файл листинга, содержащий дизассемблированный код программы:

```
$> fromelf -c --output test1.list test1.elf
```

Если всё прошло нормально, то вы можете загрузить полученный образ (ELF или двоичный) в микроконтроллер или симулятор для тестирования программы.

10.5.3. Вывод результатов работы программы

Разумеется, гораздо интереснее, если наш микроконтроллер может взаимодействовать с окружающим миром. Самым простым вариантом такого взаимодействия является включение/выключение светодиода. Однако этот способ находит довольно ограниченное применение, поскольку его информативность чрезвычайно мала. Одним из наиболее широко используемых способов вывода информации является передача текстовых сообщений. При разработке встраиваемых устройств для этого, как правило, применяется интерфейс UART. Для отображения получаемой информации можно задействовать, например, компьютер под управлением ОС Windows[1)] с программой Hyper-Terminal, работающей в режиме консоли (**Рис. 10.11**).

[1)] Windows и Hyper-Terminal являются товарными знаками компании Microsoft Corporation.

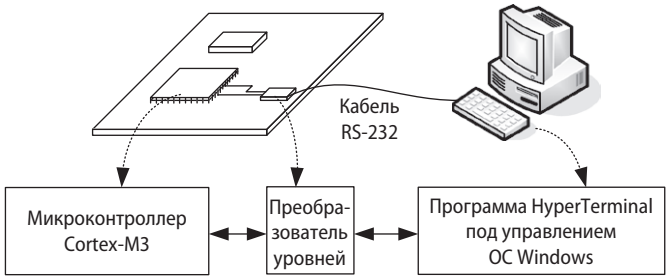


Рис. 10.11. *Недорогая тестовая система, в которой используется вывод текстовых сообщений.*

В самом процессоре Cortex-M3 нет интерфейса UART, однако соответствующие модули имеются в большинстве микроконтроллеров с процессором Cortex-M3. Спецификации данных модулей определяются производителями конкретных микроконтроллеров, поэтому мы не будем заострять на этом внимание. Наш следующий пример написан в расчёте на то, что в модуле UART используемого микроконтроллера имеется флаг состояния, показывающий готовность буфера передачи к загрузке новых данных. Для подключения микроконтроллера к компьютеру потребуется также преобразователь уровней, поскольку уровни напряжений интерфейса RS-232 отличаются от логических уровней, имеющихся на выводах микроконтроллера.

Интерфейс UART — далеко не единственное решение для вывода текстовых сообщений. В процессоре Cortex-M3 реализовано несколько механизмов, облегчающих вывод отладочных сообщений:

- *Semihosting.* Механизм полухостинга (вывод сообщений с помощью функции `printf` через отладчик) может быть задействован при помощи регистра управления отладкой контроллера NVIC (возможность применения этого механизма зависит от конкретного отладчика и используемой библиотеки). Более подробно данный вопрос рассматривается в Главе 15. При использовании данного механизма вы можете задействовать в своей программе стандартную функцию `printf` для вывода информации в консоль/стандартный поток вывода (STDOUT) программы отладчика.
- *Instrumentation trace.* Если в микроконтроллере с ядром Cortex-M3 реализован специальный порт трассировки, а в вашем распоряжении имеется анализатор порта трассировки (Trace Port Analyzer — TPA), то вместо модуля UART вы можете использовать для вывода отладочных сообщений модуль ITM. Порт трассировки работает гораздо быстрее UART и позволяет реализовать большее число каналов передачи данных.
- *Instrumentation trace via Serial-Wire Viewer (SVW).* В качестве альтернативы процессор Cortex-M3 (1-й и последующих ревизий) также поддерживает работу модуля интерфейса порта трассировки TPIU в режиме SVW. Этот интерфейс позволяет считывать данные с выхода модуля ITM, используя недорогие устройства вместо анализаторов порта. Однако модуль TPIU в режиме SVW имеет ограниченную полосу пропускания, поэтому данный механизм не

очень годится в случае больших объёмов данных (скажем, при трассировке исполняемых команд).

10.5.4. Программа «Hello World»

Прежде чем приступить к написанию программы «Hello World», мы должны понять, каким образом производится передача одиночных символов по интерфейсу UART. Код, выполняющий эту операцию, можно оформить в виде подпрограммы, которая, в свою очередь, будет вызываться другими функциями вывода сообщений. Тогда при смене устройства вывода нам потребуется изменить только эту подпрограмму, после чего все текстовые сообщения можно будет выводить на другое устройство. Такого рода модификацию кода обычно называют перенаправлением (retargeting).

Ниже приведён код простейшей подпрограммы для вывода одного символа:

```
UART0_BASE    EQU   0x4000C000
UART0_FLAG    EQU   UART0_BASE+0x018
UART0_DATA    EQU   UART0_BASE+0x000
Putc       ; Подпрограмма вывода одного символа через UART
           ; Вход: R0 – передаваемый символ
           PUSH {R1,R2, LR}    ; Сохраняем регистры
           LDR  R1,=UART0_FLAG
PutcWaitLoop
           LDR  R2,[R1]        ; Читаем флаги состояния
           TST  R2, #0x20      ; Проверяем флаг буфера передачи
           BNE  PutcWaitLoop   ; Если полон, то проверяем снова
           LDR  R1,=UART0_DATA ; Иначе
           STRB R0, [R1]       ; Загружаем данные в буфер передачи
           POP  {R1,R2, PC}    ; Возвращаемся
```

Адреса регистров и определения битов в данном коде приведены лишь для примера; вам, скорее всего, придётся изменить эти значения в соответствии с используемым устройством. Кроме того, в некоторых модулях UART может потребоваться более сложная процедура проверки состояния перед загрузкой символа в буфер передачи. Для инициализации модуля UART вызывается отдельная подпрограмма (`Uart0Initialize`), реализация которой будет зависеть от конкретного модуля, в связи с чем она не приводится в этой главе. Пример инициализации модуля UART для микроконтроллера LM3S811 производства Texas Instruments, написанный на языке Си, приведён в Главе 20.

Теперь мы можем использовать нашу подпрограмму для создания различных функций, осуществляющих вывод текстовых сообщений:

```
Puts      ; Подпрограмма вывода строки через UART
          ; Вход: R0 = адрес начала строки
          ; Строка должна завершаться  нулевым символом
          PUSH {R0 ,R1, LR}      ; Сохраняем регистры
          MOV  R1, R0            ; Копируем адрес в R1, поскольку R0 будет
                                 ; использоваться для передачи символа в Putc
PutsLoop
          LDRB R0,[R1],#1        ; Читаем один символ и инкрементируем адрес
          CBZ  R0, PutsLoopExit  ; Если символ нулевой, то выходим из цикла
```

```
          BL   Putc              ; Выводим символ в UART
          B    PutsLoop          ; Следующий символ
PutsLoopExit
          POP  {R0, R1, PC}      ; Возвращаемся
```

Используя эту подпрограмму, мы, наконец, можем написать нашу первую программу «Hello World»:

```
STACK_TOP     EQU   0x20002000 ; Константа, содержащая начальное значение SP
UART0_BASE    EQU   0x4000C000
UART0_FLAG    EQU   UART0_BASE+0x018
UART0_DATA    EQU   UART0_BASE+0x000
              AREA | Header Code|, CODE
              DCD STACK_TOP       ; Начальное значение указателя стека
              DCD Start           ; Вектор сброса
              ENTRY
Start         ; Начало основной программы
              MOV  r0, #0         ; Инициализируем регистры
              MOV  r1, #0
              MOV  r2, #0
              MOV  r3, #0
              MOV  r4, #0
              BL   Uart0Initialize ; Инициализируем UART0
              LDR  r0,=HELLO_TXT   ; Заносим в R0 адрес начала строки
              BL   Puts
deadend
              B    deadend         ; Бесконечный цикл
              ;--------------------------------
              ; Подпрограммы
              ;--------------------------------
Puts          ; Подпрограмма вывода строки через UART
              ; Вход: R0 = адрес начала строки
              ; Строка должна завершаться  нулевым символом
              PUSH {R0 ,R1, LR}     ; Сохраняем регистры
              MOV  R1, R0           ; Копируем адрес в R1, поскольку R0 будет
                                    ; использоваться для передачи символа в Putc
PutsLoop
              LDRB R0,[R1],#1       ; Читаем один символ и инкрементируем адрес
              CBZ  R0, PutsLoopExit ; Если символ нулевой, то выходим из цикла
              BL   Putc             ; Выводим символ в UART
              B    PutsLoop         ; Следующий символ
PutsLoopExit
              POP  {R0, R1, PC}     ; Возвращаемся
              ;--------------------------------
Putc          ; Подпрограмма вывода одного символа через UART
              ; Вход: R0 - передаваемый символ
              PUSH {R1, R2, LR}     ; Сохраняем регистры
              LDR  R1,=UART0_FLAG
PutcWaitLoop
              LDR  R2, [R1]         ; Читаем флаги состояния
              TST  R2, #0x20        ; Проверяем флаг буфера передачи
              BNE  PutcWaitLoop     ; Если полон, то проверяем снова
              LDR  R1,=UART0_DATA   ; Иначе
```

```
            STR  R0, [R1]         ; Загружаем данные в буфер передачи
            POP  {R1, R2, PC}     ; Возвращаемся
            ;-------------------------------
Uart0Initialize
            ; Зависит от устройства, здесь не приведено
            BX   LR               ; Возвращаемся
            ;-------------------------------
            HELLO_TXT
            DCB «Hello world\n»,0 ; Строка, завершающаяся нулевым символом
            END                   ; Конец файла
```

Единственное, что вам осталось сделать, — это вставить в программу код подпрограммы `Uart0Initialize` и откорректировать значения адресов регистров UART, расположенные в начале программы.

Также нелишне было бы иметь ряд подпрограмм для вывода содержимого регистров. Эти подпрограммы могут базироваться на уже написанных подпрограммах `Putc` и `Puts`. Первая подпрограмма предназначена для вывода шестнадцатеричных значений.

```
PutHex ; Подпрограмма вывода содержимого регистра в шестнадцатеричном виде
       ; Вход: R0 = выводимое значение
     PUSH  {R0-R3, LR}
     MOV   R3, R0      ; Сохраняем входной параметр в R3, поскольку R0
                       ; используется для передачи параметров в другие
                       ; подпрограммы
     MOV   R0,#'0'     ; Выводим «0x»
     BL    Putc
     MOV   R0,#'x'
     BL    Putc
     MOV   R1, #8      ; Инициализируем счётчик цикла
     MOV   R2, #28     ; Задаём смещение
PutHexLoop
     ROR   R3, R2      ; Сдвигаем значение на 4 бит влево (на 28 бит вправо)
     AND   R0, R3,#0xF ; Выделяем младшие 4 бит
     CMP   R0, #0xA    ; Преобразуем в ASCII
     ITE   GE
     ADDGE R0, #55     ; Если больше или равно 10, преобразуем в «A»-«F»,
     ADDLT R0, #48     ; иначе преобразуем в «0»-«9»
     BL    Putc        ; Выводим один шестнадцатеричный разряд
     SUBS  R1, #1      ; Декрементируем счётчик цикла
     BNE   PutHexLoop  ; Если переданы все 8 шестнадцатеричных разрядов,
     POP  {R0-R3, PC}  ; то возвращаемся, иначе обрабатываем следующие 4 бит
```

Эта подпрограмма полезна для вывода значений регистров. Однако иногда желательно выводить содержимое регистров в десятичном виде. Не пугайтесь, в процессоре Cortex-M3 данная операция реализуется легко и быстро благодаря наличию команд аппаратного умножения и деления. Ещё одна проблема заключается в том, что выходные символы, получаемые в процессе преобразования, будут располагаться в обратном порядке. Поэтому нам потребуется промежуточный буфер для хранения указанных символов. После завершения преобразования мы выведем полученную строку при помощи функции `Puts`. В данном примере текстовый буфер размещается в стеке:

```
PutDec   ; Подпрограмма вывода содержимого регистра в десятичном виде
         ; Вход: R0 = выводимое значение
         ; Поскольку регистр 32-битный, то максимальное число символов
         ; в десятичном формате, включая завершающий нулевой символ, равно 11
         PUSH {R0-R5, LR}    ; Сохраняем регистры
         MOV  R3, SP         ; Копируем указатель стека в R3
         SUB  SP, SP, #12    ; Резервируем 12 байт под текстовый буфер
         MOV  R1, #0         ; Нулевой символ
         STRB R1, [R3, #-1]! ; Кладём нулевой символ в конец буфера,
                             ; используем операцию пересылки с прединдексацией
         MOV  R5, #10        ; Задаём делитель
PutDecLoop
         UDIV R4, R0, R5     ; R4 = R0 / 10
         MUL  R1, R4, R5     ; R1 = R4 * 10
         SUB  R2, R0, R1     ; R2 = R0 - (R4 * 10) = остаток
         ADD  R2, #48        ; Преобразуем в ASCII (R2 должен содержать
                             ; от 0 до 9)
         STRB R2, [R3, #-1]! ; Кладём ASCII-символ в буфер (с прединдексацией)
         MOVS R0, R4         ; Заносим в R0 результат деления и устанавливаем
                             ; флаг Z, если R4 = 0
         BNE  PutDecLoop     ; Если R0 (R4) уже равно 0, значит, больше
                             ; десятичных разрядов нет
         MOV  R0, R3         ; Заносим в R0 адрес начала буфера
         BL   Puts           ; Выводим строку, используя подпрограмму Puts
         ADD  SP, SP, #12    ; Восстанавливаем указатель стека
         POP  {R0-R5, PC}    ; Возвращаемся
```

Благодаря различным возможностям, предоставляемым набором команд процессора Cortex-M3, мы смогли написать очень компактную процедуру преобразования числовых значений в десятичный текстовый формат.

10.5.5. Использование памяти данных

Вернёмся к нашему первому примеру: на этапе компоновки мы указали область памяти, доступную как для чтения, так и для записи. Как же нам разместить в ней данные? Для этого в ассемблерном файле необходимо определить область данных. Взяв за основу пример, приведённый в начале главы, мы можем сохранить данные в памяти данных по адресу 0x20000000 (область СОЗУ). Расположение секции данных определяется ключами командной строки при вызове компоновщика:

```
STACK_TOP     EQU 0x20002000   ; Константа, содержащая начальное значение SP
              AREA | Header Code|, CODE
              DCD  STACK_TOP   ; Начальное значение SP
              DCD  Start       ; Вектор сброса
              ENTRY
Start         ; Начало основной программы
              ; Инициализируем регистры
              MOV  r0, #10     ; Начальное значение счётчика цикла
              MOV  r1, #0      ; Начальное значение результата
                               ; Будем вычислять 10 + 9 + 8 +...+ 1
```

```
loop
                ADD  r1, r0      ; R1 = R1 + R0
                SUBS r0, #1      ; Декрементируем R0, обновляя флаги (суффикс S)
                BNE  loop        ; Если R0 не равен 0, то переходим к метке loop
                                 ; Результат сложения - в R1
                LDR  r0,=MyData1 ; Загружаем адрес ячейки MyData1 в R0
                STR  r1, [r0]    ; Сохраняем результат в MyData1
deadloop
                B deadloop       ; Бесконечный цикл
                AREA | Header Data|, DATA
                ALIGN 4
MyData1         DCD 0 ; Ячейка памяти для хранения результата
MyData2         DCD 0
                END ; Конец файла
```

Компоновщик разместит секцию `DATA` в области перезаписываемой памяти, поэтому в данном случае переменная `MyData1` будет располагаться по адресу 0x20000000.

10.6. Монопольный доступ и семафоры

Команды монопольного доступа применяются для реализации семафоров, в частности семафоров взаимных исключений (мьютексов), позволяющих гарантировать использование какого-либо ресурса только одной задачей. В качестве примера представим себе, что переменная `DeviceALocked`, расположенная в памяти, предназначена для индикации использования устройства A. Если задача хочет задействовать устройство A, то она должна сначала проверить его состояние, прочитав переменную `DeviceALocked`. Если её значение равно нулю, то задача может записать в неё 1 для блокирования устройства. После того как надобность в устройстве отпадёт, задача может обнулить переменную `DeviceALocked`, предоставляя возможность использовать это устройство другим задачам.

Что произойдёт, если две задачи попытаются одновременно обратиться к устройству A? В этом случае существует вероятность, что обе задачи, прочитав переменную, получат нулевое значение. После этого обе попытаются записать в неё 1 для блокирования устройства, и в результате каждая задача будет пребывать в полной уверенности, что она имеет исключительный доступ к устройству. Чтобы описанной ситуации не возникало, используются команды исключительного доступа. Команда `STREX` возвращает значение, показывающее, была ли операция сохранения успешной. Если обе задачи одновременно попытаются заблокировать устройство, то будет возвращена 1 (отказ доступа), и задача будет знать, что ей необходимо повторить блокирование устройства.

Некоторые базовые сведения о командах монопольного доступа приведены в Главе 5. Блок-схема, отображающая рассмотренную выше ситуацию, приведена на **Рис. 10.12**.

Ниже приведена подпрограмма, выполняющая указанные операции с использованием функций стандарта CMSIS. Заметьте, что при отказе доступа операция записи данных командой `STREX` не выполняется, что предотвращает установку бита блокировки:

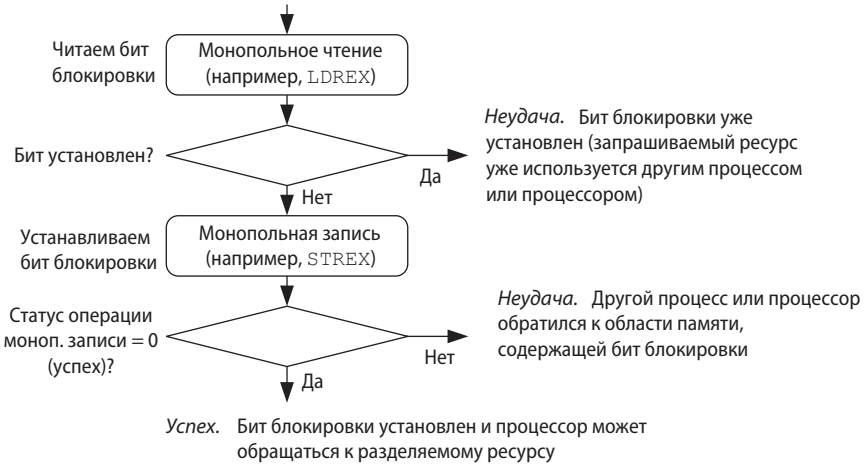


Рис. 10.12. *Использование команд монопольного доступа для реализации семафоров.*

```
volatile unsigned int DeviceALocked; // Флаг блокировки
int LockDeviceA(void){
  unsigned int status; // Переменная для хранения статуса выполнения STREX
  // Читаем флаг блокировки и проверяем его состояние
  if (__LDREXW(&DeviceALocked) = 0) {
    // Если блокировки нет, то пытаемся установить флаг
    status = __STREXW(1, &DeviceALocked);
    if (status != 0) return (1); // Возвращаем «отказ доступа»
    else return(0); // Возвращаем «успех»
  } else {
    return(1); // Возвращаем «отказ доступа»
  }
}
```

Аналогичную функцию можно написать и на ассемблере:

```
LockDeviceA
            ; Простая функция для блокирования устройства A
            ; Возвращаемое значение (R0) : 0 = «успех», 1 = «отказ доступа»
            ; В случае успеха в переменную DeviceALocked заносится 1
            PUSH  {R1, R2, LR}
TryToLockDeviceA
            LDR   R1,=DeviceALocked ; Получаем состояние блокировки
            LDREX R2,[R1]
            CMP   R2,#0             ; Заблокировано?
            BNE   LockDeviceAFailed
DeviceAIsNotLocked
            MOV   R0,#1             ; Пытаемся записать 1 в DeviceALocked
            STREX R2,R0,[R1]        ; Монопольная запись
            CMP   R2, #0
            BNE   LockDeviceAFailed ; STREX не прошла
LockDeviceASucceed
            MOV   R0,#0             ; Возвращаем «успех»
            POP   {R1, R2, PC}      ; Возвращаемся
```

```
LockDeviceAFailed
            MOV   R0,#1             ; Возвращаем «отказ доступа»
            POP   {R1, R2, PC}      ; Возвращаемся
```

Если функция возвращает 1 (отказ в монопольном доступе), то программа должна выждать некоторое время и повторить попытку позже. В однопроцессорных системах наиболее частой причиной отказа в монопольном доступе служит появление прерывания между командами монопольной загрузки и монопольного сохранения. Если код выполняется на привилегированном уровне доступа, то возникновение такой ситуации можно предотвратить, установив на некоторое время регистр маскирования, такой как PRIMASK, — это увеличит вероятность успешного блокирования ресурса.

В многопроцессорных системах возможна и другая причина отказа в выполнении команды монопольного сохранения — обращение другого процессора к той же области памяти. Факт обращения к памяти других процессоров определяется отдельным устройством, которое называется монитором монопольного доступа. Этот монитор отслеживает обращения к памяти со стороны других ведущих шины между двумя операциями монопольного доступа. Однако, поскольку в большинстве недорогих микроконтроллеров задействуется только один процессор Cortex-M3, в таком мониторе нет никакой необходимости.

Используя описанный механизм, мы можем быть твёрдо уверены, что в каждый момент времени к конкретному разделяемому ресурсу обращается только одна задача. Если приложение не может захватить ресурс в течение какого-то времени, оно должно завершить работу с сообщением об ошибке тайм-аута. Такая ситуация может возникнуть, если задача, заблокировавшая ресурс, аварийно завершится, не сбросив признак блокировки. В подобных случаях использование ресурса конкретными задачами должно отслеживаться операционной системой. Если задача завершилась или была прервана, не сбросив признак блокировки, то ОС должна будет разблокировать ресурс.

Если процесс начал операцию обращения в монопольном режиме, используя команду `LDREX`, а затем обнаружил, что монопольный доступ больше не требуется, он может использовать команду `CLREX` для очистки локальной записи в мониторе монопольного доступа. Эту операцию можно выполнить с помощью функции CMSIS:

```
void __CLREX(void);
```

При программировании на ассемблере просто вставляется команда `CLREX`:

```
CLREX
```

или

```
CLREX.W
```

В процессоре Cortex-M3 все операции монопольного доступа должны выполняться последовательно. Если же код, управляющий монопольным доступом, должен работать и на других процессорах ARM, то между монопольными пересылками необходимо вставлять команду барьера памяти данных `DMB`, чтобы гарантировать корректный порядок обращений к памяти. Пример использования команд барьерной синхронизации с командами монопольного доступа можно найти в разделе 14.3, посвящённом коммуникациям в многопроцессорных системах.

10.7. Метод bit-band и семафоры

Для реализации семафоров можно также воспользоваться побитовым доступом по методу bit-band при условии, что система памяти поддерживает блокированные пересылки или же к шине памяти подключён только один ведущий. Используя метод bit-band, мы можем реализовать функционал семафоров на языке Си, не прибегая к помощи операций монопольного доступа. В данном случае для управления ресурсами используется переменная (скажем, слово), расположенная в области хранения битов, каждый бит которой показывает, что соответствующий ресурс используется какой-то задачей.

Операции записи в область доступа к битам представляют собой блокированные пересылки типа «чтение—модификация—запись» (смена ведущего шины между пересылками не допускается). Поэтому если каждая задача изменяет только свой бит, то значения битов блокировки других задач не будут искажены даже в том случае, если две задачи одновременно попытаются выполнить запись по одному и тому же адресу. В отличие от варианта с использованием команд монопольного доступа, возможно кратковременное «блокирование» ресурса одновременно двумя задачами до тех пор, пока одна из них не обнаружит конфликт и не снимет блокировку (**Рис. 10.13**).

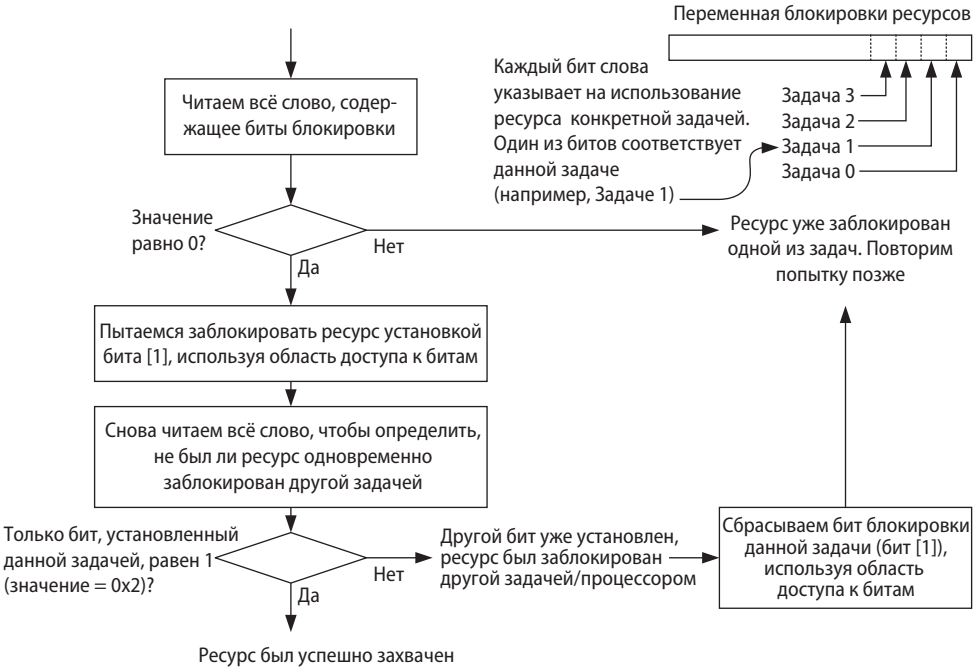


Рис. 10.13. *Реализация мьютекса с использованием метода bit-band.*

Использование метода bit-band для реализации семафоров будет работать только в том случае, если все задачи в системе будут изменять свои биты блокировки исключительно посредством области доступа к битам. Если же любая из задач изменит ячейку с битами блокировки, используя обычную операцию запи-

си, то работа семафора может быть нарушена: в случае, если непосредственно перед записью в эту ячейку какой-либо задачей будет установлен бит блокировки, то бит блокировки, установленный ранее другой задачей, окажется сброшенным.

10.8. Использование команд извлечения битового поля и команд табличных переходов

Мы познакомились с командами извлечения беззнакового битового поля (UBFX) и командами табличного перехода (TBB/TBH) в Главе 4. Эти две команды могут использоваться вместе для реализации очень развитых деревьев переходов. Такие деревья находят широкое применение в приложениях передачи данных, в которых передаваемые данные могут иметь различный смысл в зависимости от заголовка сообщения. В качестве примера, давайте рассмотрим ассемблерную процедуру декодирования дерева решений, изображённого на **Рис. 10.14**.

```
DecodeA
    LDR  R0,=A                  ; Считываем значение A из памяти
    LDR  R0, [R0]
    UBFX R1, R0, #6, #2         ; Извлекаем биты [7:6] в R1
    TBB  [PC, R1]
BrTable1
    DCB  ((P0 -BrTable1)/2)     ; Переходим к P0, если A[7:6] = 00
    DCB  ((DecodeA1-BrTable1)/2) ; Переходим к DecodeA1, если A[7:6] = 01
    DCB  ((P1 -BrTable1)/2)     ; Переходим к P1, если A[7:6] = 10
    DCB  ((DecodeA2-BrTable1)/2) ; Переходим к DecodeA1, если A[7:6] = 11
DecodeA1
    UBFX R1, R0, #3, #2         ; Извлекаем биты [4:3] в R1
    TBB  [PC, R1]
BrTable2
    DCB  ((P2 -BrTable2)/2)     ; Переходим к P2, если A[4:3] = 00
    DCB  ((P3 -BrTable2)/2)     ; Переходим к P3, если A[4:3] = 01
    DCB  ((P4 -BrTable2)/2)     ; Переходим к P4, если A[4:3] = 10
    DCB  ((P4 -BrTable2)/2)     ; Переходим к P4, если A[4:3] = 11
DecodeA2
    TST  R0, #4                 ; Проверяем только 1 бит, поэтому команда UBFX
                                ; не нужна
    BEQ  P5
    B    P6
P0 ... ; Process 0
P1 ... ; Process 1
P2 ... ; Process 2
P3 ... ; Process 3
P4 ... ; Process 4
P5 ... ; Process 5
P6 ... ; Process 6
```

Как видите, мы смогли реализовать достаточно большое дерево решений с помощью весьма компактного фрагмента. Если конечные адреса переходов нахо-

дятся на более далёком расстоянии, то некоторые из команд `TBB` необходимо будет заменить на команды `TBH`.

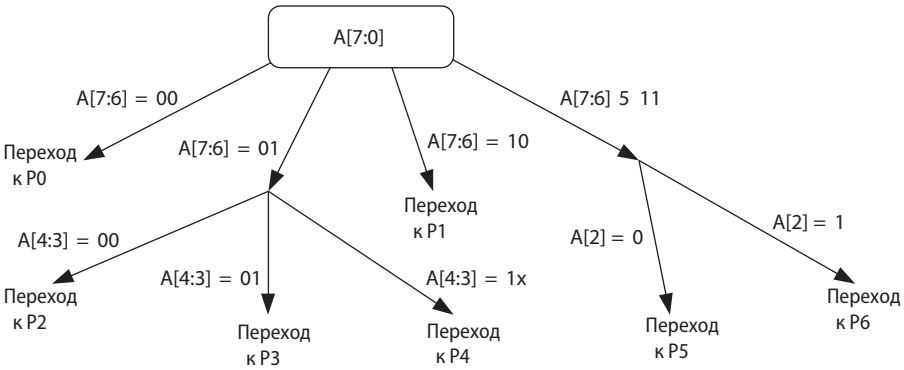


Рис. 10.14. *Декодер битового поля: пример использования команд* `UBFX` *и* `TBB`.

ГЛАВА 11
РАБОТА С ПРЕРЫВАНИЯМИ/ ИСКЛЮЧЕНИЯМИ

11.1. Использование прерываний

Наверное, сложно найти такое встраиваемое приложение, в котором вообще не использовались бы прерывания. В процессоре Cortex™-M3 большую часть работы по обслуживанию прерываний выполняет контроллер прерываний NVIC. В частности, он осуществляет контроль приоритетов прерываний, а также сохранение/восстановление контекста в стеке. Однако, прежде чем вы сможете использовать прерывания в своей программе, вам необходимо будет выполнить ряд операций:

- настроить стек;
- настроить таблицу векторов прерываний;
- задать приоритеты прерываний;
- разрешить прерывания.

11.1.1. Конфигурирование стека

В простых приложениях, как правило, используется один (основной) стек. При разработке таких приложений необходимо только зарезервировать под стек область памяти достаточного размера и установить указатель MSP на вершину этой области. Не забудьте: при расчёте требуемого размера стека необходимо учитывать не только степень его использования программой, но и максимально возможное число уровней вложенности прерываний.

Каждый уровень вложенности требует, как минимум, восемь слов стека. Кроме того, стек может использоваться и самими обработчиками прерываний.

Поскольку в процессоре Cortex-M3 реализован «полный» убывающий стек, указатель стека обычно устанавливают на конец области статического ОЗУ, чтобы исключить фрагментацию свободной памяти (**Рис. 11.1**).

Рис. 11.1. *Пример простого распределения памяти.*

В более сложных приложениях, задействующих разные стеки для пользовательского кода и ядра системы, размер основного стека должен быть достаточным для поддержки вложенных вызовов обработчиков прерываний, а также для работы ядра. Размер стека процесса должен быть на восемь слов больше, чем требуется для работы пользовательского кода. Этот «довесок» обусловлен тем, что при переходе к обработчику прерывания первого уровня контекст пользовательской программы сохраняется в стеке процесса.

11.1.2. Настройка таблицы векторов прерываний

В простых приложениях, использующих неизменяемые обработчики прерываний, таблица векторов может располагаться во флэш-памяти или в ПЗУ. В этом случае необходимость в какой-либо настройке таблицы во время выполнения программы отсутствует. Однако существуют приложения, в которых обработчики прерываний требуется изменять в зависимости от каких-либо условий. В таком случае вам придётся переместить таблицу векторов в область перезаписываемой памяти.

Перед перемещением таблицы векторов необходимо будет скопировать её часть в то место, где будет располагаться новая таблица. Копируются адреса векторов обработчиков отказов, немаскируемого прерывания, системных вызовов и т.п. Если этого не сделать, то при возникновении любого из указанных исключений после перемещения таблицы процессор выберет неверное значение адреса вектора.

После определения требуемых элементов таблицы векторов и перемещения таблицы на новое место мы можем добавлять в неё новые векторы. В CMSIS-совместимых библиотеках для конфигурирования векторов таблицы предусмотрена следующая функция (для обращения к регистру смещения таблицы векторов в данной функции используется операция `SCB->VTOR`):

```
void SetVector(unsigned int ExcpType, unsigned int VectorAddress)
{ // Место расположения вектора вычисляется как VTOR + (Exception_Type * 4)
    *((volatile unsigned int *) (SCB->VTOR + (ExcpType << 2))) =
      VectorAddress | 0x1;
    // LSB вектора устанавливается в 1 (Thumb)
    return;
}
```

Те, кто предпочитает программировать на ассемблере, могут использовать подпрограмму, текст которой приведён в следующем примере:

```
    ; Подпрограмма для задания вектора исключения по его номеру
    ; (Для пользовательских прерываний прибавляем 16: IRQ #0 = исключение №16)
SetVector
    ; Вход: R0 = Номер исключения
    ;       R1 = Значение адреса вектора
    PUSH {R2, LR}
    LDR R2,=0xE000ED08       ; Регистр смещения таблицы векторов
    LDR R2, [R2]
    ORR R1, R1, #1           ; Устанавливаем LSB вектора, так как используем
                             ; команды Thumb
    STR R1, [R2, R0, LSL #2] ; Заносим вектор по адресу
                             ; VectTblOffset + ExcpType*4
    POP {R2, PC}             ; Возвращаемся
```

В большинстве случаев устанавливать младший бит вектора в 1 не требуется, поскольку компилятор или ассемблер должны распознать задаваемое значение как адрес команды Thumb и установить этот бит автоматически.

11.1.3. Назначение приоритетов прерываний

По умолчанию (после сброса) все исключения с программируемым приоритетом имеют нулевой уровень приоритета. Исключения Hard Fault и NMI имеют уровни –1 и –2 соответственно. При использовании CMSIS-совместимых библиотек для задания уровня приоритета можно задействовать соответствующую библиотечную функцию. К примеру, установка приоритета прерывания №4, равного 0xC0, производится следующим образом:

```
NVIC_SetPriority(IRQ4_IRQn, 0xC); // Эта функция
// автоматически сдвигает значение приоритета на число
// битов, реализованных в регистрах уровня приоритета
```

Константа `IRQ4 _ IRQn`, упомянутая в примере, является идентификатором прерывания. При вызове функций управления прерываниями стандарта CMSIS рекомендуется применять идентификаторы, определённые в заголовочном файле `device.h` (см. **Рис. 10.8**), — это улучшает читабельность программы и её переносимость.

С функцией `NVIC _ SetPriority` можно использовать другую функцию стандарта CMSIS, которая вычисляет значение уровня приоритета, исходя из значений приоритета группы, субприоритета и настроек группирования приоритетов:

```
NVIC_SetPriority(IRQ4_IRQn, NVIC_EncodePriority(PriorityGroup,
  PreemptPriority, SubPriority));
```

Дополнительная информация об этих функциях приведена в Приложении Ж.

При программировании на ассемблере мы можем облегчить себе работу, используя побайтную адресацию регистров, например:

```
; Установка приоритета IRQ #4, равного 0xC0
LDR  R0, =0xE000E400 ; Базовый адрес регистров приоритета внешних прерываний
LDR  R1, =0xC0       ; Уровень приоритета
STRB R1, [R0, #4]    ; Устанавливаем приоритет IRQ #4 (пишем один байт)
```

В процессоре Cortex-M3 разрядность регистров приоритета прерываний задаётся изготовителем микросхемы. Минимальная разрядность равна трём, а максимальная — восьми битам. В CMSIS-совместимых драйверах устройств разрядность регистров уровня приоритета определяется константой __NVIC_PRIO_BITS. Число реализованных битов также можно определить, записав 0xFF в один из регистров приоритета и прочитав затем его содержимое. Описанную процедуру можно реализовать следующим образом:

```
; Определение разрядности поля приоритета
LDR  R0,=0xE000E400 ; Регистр приоритета внешнего прерывания №0
LDR  R1,=0xFF
STRB R1,[R0]        ; Пишем 0xFF (однобайтная запись)
LDRB R1,[R0]        ; Считываем обратно (например, 0xE0 при трёх битах)
RBIT R2, R1         ; Переставляем биты и сохраняем в R2 (например,
                    ; 0x07000000 при трёх битах)
CLZ  R1, R2         ; Считаем число ведущих нулей (например, 0x5
                    ; при трёх битах)
MOV  R2, #8
SUB  R2, R2, R1     ; Вычисляем реальную разрядность регистров приоритета
                    ; (например, 8 - 5 = 3 при трёх битах)
MOV  R1, #0x0
STRB R1,[R0]        ; Восстанавливаем значение по умолчанию (0x0)
```

Если вы хотите, чтобы ваше приложение было переносимым, рекомендуется использовать только следующие значения уровней приоритета: 0x00, 0x20, 0x40, 0x60, 0x80, 0xA0, 0xC0 и 0xE0. Это пожелание связано с тем, что данные уровни приоритета имеются в любом устройстве на базе процессора Cortex-M3.

Не забывайте также задавать приоритеты системных исключений и обработчиков отказов. Если по каким-либо причинам некоторые важные прерывания в приложении должны иметь приоритет, который выше приоритета остальных системных исключений или обработчиков отказов, то необходимо снизить уровень приоритета последних с тем, чтобы эти обработчики могли прерываться обработчиками важных прерываний.

11.1.4. Разрешение прерываний

После настройки таблицы векторов и задания приоритета мы наконец-то можем разрешить требуемое прерывание. Однако перед этим необходимо выполнить следующее:

1. Если таблица векторов располагается в области памяти с буферированием записи, то возможно потребуется применить команду барьера синхронизации данных (DSB), чтобы быть уверенными, что содержимое таблицы векторов полностью обновлено. В большинстве случаев операции

записи в память должны завершаться за несколько тактов. Однако если вы хотите обеспечить переносимость своей программы на другие процессоры ARM, то указанная операция сможет гарантировать выборку корректного значения вектора при возникновении прерывания сразу же после его разрешения.

2. Поскольку прерывание уже могло быть отложено или активировано, перед разрешением прерывания настоятельно рекомендуется сбрасывать бит признака отложенного прерывания. В частности, запросы прерываний могут случайно генерироваться из-за паразитных импульсов, возникающих при подаче питания. Кроме того, в некоторых периферийных устройствах, таких как UART, шум на неподключённом входе приёмника может быть ошибочно принят устройством за данные и вызвать отложенное прерывание. Поэтому из соображений безопасности рекомендуется проверять признак отложенного прерывания и, при необходимости, сбрасывать его перед разрешением прерывания. К тому же существуют такие устройства, которые могут потребовать повторной инициализации в случае, если при их разрешении признак отложенного прерывания уже был установлен.

Обращения к регистрам контроллера NVIC

Чтобы улучшить совместимость ПО, для обращения к регистрам контроллера NVIC (в том числе и к регистрам конфигурации прерываний) следует использовать соответствующие функции CMSIS. Более подробно функции CMSIS, предназначенные для работы с периферией ядра процессора, описаны в Приложении Ж.

При необходимости, вы можете написать собственные варианты функций для работы с контроллером NVIC (учтите, что правильно выбранный размер пересылок может облегчить разработку вашей программы). В процессоре Cortex-M3 к большинству регистров контроллера NVIC можно обращаться с помощью однобайтных, двухбайтных и 4-байтных операций пересылки. В частности, к регистрам уровня приоритета лучше обращаться побайтно — это позволит не беспокоиться о случайном изменении приоритетов других исключений. Однако для процессора Cortex-M0 данный метод не годится, поскольку контроллер NVIC этого процессора допускает только пословное обращение.

Для разрешения и запрещения прерываний в контроллере NVIC предусмотрены отдельные регистры. Это позволяет разрешать/запрещать отдельные прерывания, не затрагивая биты разрешения остальных прерываний. В противном случае, при использовании программных операций типа «чтение—модификация—запись» существовала бы вероятность потери изменений, сделанных в регистрах разрешения обработчиками прерываний. Для разрешения прерывания программа должна определить положение бита, соответствующего данному прерыванию, в группе регистров NVIC_ISERx и записать в этот бит 1. Аналогичным образом, для запрещения прерывания необходимо записать 1 в соответствующий бит одного из регистров запрещения NVIC_ICERx.

Те, кто пользуются CMSIS-совместимыми библиотеками, могут задействовать для разрешения/запрещения прерываний функции `NVIC _ EnableIRQ` и `NVIC _ DisableIRQ`, например:

```
NVIC_EnableIRQ(UART1_IRQn);  // Разрешаем прерывание модуля UART1
```

```
                                    // Значение UART1_IRQn зависит от МК и
                                    // определяется в файле device.h
    NVIC_DisableIRQ(UART1_IRQn); // Запрещаем прерывание модуля UART1
```

Более подробно эти функции описаны в Приложении Ж.

Те, кто использует ассемблер, могут сами написать функции, выполняющие аналогичные операции. Так на ассемблере может выглядеть аналог функции NVIC _ EnableIRQ:

```
    ; Подпрограмма разрешения прерывания по его номеру
EnableIRQ
    ; Вход: R0 = Номер IRQ
    PUSH    {R0-R2, LR}
    AND.W   R1, R0, #0x1F  ; Формируем битовую маску для разрешения прерывания
    MOV     R2, #1
    LSL     R2, R2, R1     ; Маска = (0x1 << (N & 0x1F))
    AND.W   R1, R0, #0xE0  ; Вычисляем смещение, если номер прерывания больше 31
    LSR     R1, R1, #3     ; Смещение = (N/32)*4 (каждое слово содержит
                           ; биты разрешения для 32 прерываний)
    LDR     R0,=0xE000E100 ; Адрес регистра NVIC_ISER0 для внешних прерываний
                           ; с номерами 31...0
    STR     R2, [R0, R1]   ; Пишем маску в соответствующий регистр NVIC_ISERx
    POP     {R0-R2, PC}    ; Восстанавливаем регистры и возвращаемся
```

А так может выглядеть аналог функции NVIC _ DisableIRQ:

```
    ; Подпрограмма запрещения прерывания по его номеру
DisableIRQ
    ; Вход: R0 = Номер IRQ
    PUSH    {R0-R2, LR}
    AND.W   R1, R0, #0x1F  ; Формируем битовую маску для разрешения прерывания
    MOV     R2, #1
    LSL     R2, R2, R1     ; Маска = (0x1 << (N & 0x1F))
    AND.W   R1, R0, #0xE0  ; Вычисляем смещение, если номер прерывания больше 31
    LSR     R1, R1, #3     ; Смещение = (N/32)*4 (каждое слово содержит
                           ; биты разрешения для 32 прерываний)
    LDR     R0,=0xE000E180 ; Адрес регистра NVIC_ICER0 для внешних прерываний
                           ; с номерами 31...0
    STR R2, [R0, R1]       ; Пишем маску в соответствующий регистр NVIC_ICERx
    POP     {R0-R2, PC}    ; Восстанавливаем регистры и возвращаемся
```

Также можно написать аналогичные подпрограммы для установки и сброса битов регистров признака отложенного прерывания.

11.2. Обработчики исключений/прерываний

В процессоре Cortex-M3 обработчики прерываний могут быть полностью написаны на Си, тогда как в процессоре ARM7 в большинстве случаев применяются обработчики, написанные на ассемблере, которые гарантируют сохранение всех необходимых регистров. А при использовании вложенных прерываний необходимо к тому же переключать режимы работы процессора во избежание потери информации. В процессоре Cortex-M3 ничего подобного не требуется, что значительно упрощает разработку программного обеспечения.

На языке Си обработчик прерывания может быть описан следующим образом:

```
void UART1_Handler(void) {
  ... // Обработка запроса периферийного устройства
  return;
}
```

Для корректной инициализации содержимого таблицы векторов при использовании CMSIS-совместимых библиотек обработчики прерываний должны иметь имена, определённые изготовителем микроконтроллера. Эти имена можно узнать, посмотрев на описание таблицы векторов в файле со стартовым кодом. В пакете MDK-ARM компании Keil, например, этот файл называется `startup _ <device>.s`.

Пользователи компилятора RealView компании ARM или пакета MDK-ARM компании Keil могут для удобочитаемости добавлять к объявлению обработчика ключевое слово `_ _ irq`. Например:

```
__irq void UART1_Handler(void) {
   ... // Обрабатываем запрос IRQ от периферийного устройства
   ... // Снимаем запрос IRQ
   return;
}
```

На языке ассемблера простейший обработчик исключения может иметь следующий вид:

```
irq1_handler
    ; Обрабатываем запрос IRQ
    ...
    ; Снимаем запрос IRQ
    ...
    ; Возвращаемся из прерывания
    BX LR
```

Необходимость деактивации IRQ в процедуре обработки прерывания определяется реализацией конкретного периферийного устройства. Если устройство формирует запросы прерываний в виде импульсов, то эта операция не требуется. Что же касается процессора Cortex-M3, то при формировании периферийным устройством запросов прерываний в виде импульсов контроллер NVIC может сохранить запрос, установив бит признака отложенного прерывания. При входе в обработчик исключения этот бит автоматически сбрасывается. Данная логика отличается от поведения традиционных процессоров ARM, которые требовали, чтобы периферийные устройства удерживали запрос до тех пор, пока он не будет обработан. Это связано с тем, что контроллеры прерываний, разработанные для предыдущих процессоров ARM, таких как ARM7TDMI, не имели памяти для хранения признака отложенного прерывания.

В тех случаях, когда периферийное устройство может генерировать несколько запросов в течение короткого промежутка времени, это необходимо учитывать при деактивации запроса прерывания, чтобы исключить пропуск какого-либо запроса.

Часто обработчику прерывания помимо регистров R0…R3 и R12 требуются дополнительные регистры процессора; их тоже необходимо сохранять. При написании программы на Си программисту не нужно задумываться об этом, поскольку в Си-функциях все используемые регистры сохраняются автоматически, когда это необходимо. При использовании же ассемблера в обработчиках прерываний необходимо предусматривать команды загрузки в стек и извлечения из стека для сохранения используемых регистров из диапазона R4…R11.

В следующем примере сохраняются все регистры, которые не были помещены в стек при сохранении контекста. Если какие-то из регистров не используются в обработчике исключения, то их можно исключить из списка сохраняемых регистров:

```
irq1_handler
    PUSH {R4-R11, LR} ; Сохраняем все регистры, которые не были сохранены
                      ; при помещении контекста в стек
    ; Обрабатываем запрос прерывания
    ...
    ; Снимаем запрос прерывания в периферийном устройстве (опционально)
    ...
    POP {R4-R11, PC}  ; Восстанавливаем регистры и выходим из обработчика
```

Поскольку команда `POP` является одной из команд, позволяющих осуществить возврат из прерывания, мы можем объединить в одной команде операцию восстановления регистров с операцией возврата.

В зависимости от конструкции периферийного устройства, обработчику прерывания от этого устройства может потребоваться обратиться к нему для деактивации запроса прерывания. Если периферийное устройство формирует запрос прерывания в виде импульса, то сбрасывать указанный запрос в обработчике прерывания не требуется. В противном случае, обработчик должен сбрасывать запрос прерывания, чтобы после выхода из обработчика данное прерывание не было отложено повторно. Ещё раз напомню, что в традиционных процессорах ARM периферийные устройства должны удерживать запрос до тех пор, пока он не будет обработан.

Если периферийное устройство генерирует запросы прерываний в виде импульсов, то контроллер NVIC может сохранить запрос, используя признак отложенного прерывания. После перехода процессора к выполнению обработчика исключения бит признака отложенного прерывания сбрасывается автоматически. Естественно, что в этом случае обработчик не должен обращаться к периферийному устройству для сброса запроса прерывания.

11.3. Программные прерывания

Прерывания могут инициироваться:

- подачей сигнала на вход внешнего прерывания;
- записью в регистр признака отложенного прерывания контроллера NVIC (см. Главу 8);
- записью в регистр программной генерации прерывания STIR контроллера NVIC (см. Главу 8).

В большинстве случаев ряд прерываний оказываются незадействованными и могут использоваться в качестве программных прерываний. Программные прерывания могут служить для тех же целей, что и команда вызова супервизора (SVC), т.е. для доступа к системным службам. Однако по умолчанию пользовательские программы не могут обращаться к контроллеру NVIC. Обращение к регистру STIR контроллера NVIC допускается только в том случае, если установлен бит USERSETMPEND регистра управления конфигурацией NVIC (см. **Табл. Г.18** в Приложении Г).

В отличие от исключения SVCall, программные прерывания являются «неточными». Другими словами, обработка программного прерывания не обязательно произойдёт сразу же после его генерации, даже при отсутствии блокировки со стороны регистров маскирования прерываний или других процедур обработки прерывания. Поэтому если команда, следующая за командой записи в регистр STIR, зависит от результата программного прерывания, то результат её выполнения может оказаться некорректным, поскольку обработка программного прерывания может произойти уже после исполнения команды.

Для решения указанной проблемы можно использовать команду барьерной синхронизации DSB. Так, пользователи CMSIS-совместимых библиотек могут вставить в программу следующие строки:

```
NVIC_SetPendingIRQ(SOFTWARE_INTERRUPT_NUMBER);
__DSB();
```

При использовании ассемблера:

```
MOV R0, #SOFTWARE_INTERRUPT_NUMBER
LDR R1,=0xE000EF00 ; Адрес регистра генерации программного прерывания (STIR)
                   ; контроллера NVIC
STR R0, [R1]       ; Генерируем программное прерывание
DSB                ; Барьер синхронизации данных
...
```

Однако существует ещё одна возможная проблема. Если установлен регистр маскирования прерываний или если программное прерывание генерируется непосредственно из обработчика исключения, то существует вероятность того, что программное прерывание вообще не будет выполнено. Соответственно, программный код, генерирующий программное прерывание, должен контролировать выполнение данного прерывания. Для этой цели можно использовать программный флаг, устанавливаемый в обработчике программного прерывания.

Кстати говоря, даже сама установка бита USERSETMPEND может привести к появлению проблем. После установки указанного бита пользовательская программа сможет программно генерировать любое прерывание, кроме системных исключений. Вследствие этого, если в системе используется непроверенный пользовательский код, то обработчики исключений должны контролировать допустимость возникновения исключений, поскольку они могут быть сгенерированы пользовательской программой. В идеале, если в системе имеются недостаточно надёжные пользовательские программы, то доступ к системным сервисам должен предоставляться только через исключение SVCall.

11.4. Пример перемещения таблицы векторов

В Главе 7 мы уже говорили о том, что начальная таблица векторов должна содержать, как минимум, вектор сброса, вектор немаскируемого прерывания и вектор исключения Hard Fault, поскольку оба последних исключения не требуют какого-либо разрешения. После запуска программы мы можем при необходимости переместить таблицу векторов в область ОЗУ. Сразу необходимо отметить, что в большинстве простых приложений этого не требуется.

Для перемещения таблицы векторов необходимо выполнить следующие операции:

- *Зарезервировать память для новой таблицы векторов.* Возможно, вам придётся использовать сценарии компоновщика для резервирования участка памяти. Адрес таблицы векторов должен быть выровнен на величину, равную размеру таблицы, который приведён к ближайшему большему значению, являющемуся степенью двойки.
- *Скопировать существующую таблицу на новое место.* Перед перемещением таблицы векторов необходимо убедиться, что новая таблица содержит корректные значения векторов для всех требуемых исключений, включая NMI, Hard Fault и все разрешённые исключения.
- *Записать новый вектор исключения в новую таблицу векторов, после чего выполнить запись в регистр смещения таблицы векторов VTOR для перемещения таблицы.*

Пример программного кода, выполняющего собственно перемещение таблицы векторов, был приведён в Главе 8. В следующем примере, написанном на ассемблере, мы продемонстрируем процесс резервирования памяти в начале области ОЗУ для таблицы векторов и нескольких переменных:

```
STACK_TOP       EQU 0x20002000    ; Константа, содержащая начальное значение SP
NVIC_SETEN      EQU 0xE000E100    ; Адрес блока регистров разрешения прерываний
NVIC_VECTTBL    EQU 0xE000ED08    ; Регистр смещения таблицы векторов
NVIC_AIRCR      EQU 0xE000ED0C    ; Регистр управления прерываниями и сбросом
NVIC_IRQPRI     EQU 0xE000E400    ; Регистр уровня приоритета прерываний
      AREA | Header Code |, CODE
      DCD   STACK_TOP       ; Начальное значение SP
      DCD   Start           ; Вектор сброса
      DCD   Nmi_Handler     ; Обработчик NMI
      DCD   Hf_Handler      ; Обработчик Hard Fault
      ENTRY
Start ; Начало основной программы
      ; Инициализируем регистры
      MOV   r0, #0          ; Инициализируем регистры
      MOV   r1, #0
      ...
      ; Копируем старую таблицу векторов на новое место
      LDR   r0, =0
      LDR   r1, =VectorTableBase
      LDMIA r0!, {r2-r5}    ; Копируем 4 слова
      STMIA r1!, {r2-r5}
```

```
      DSB                   ; Барьер синхронизации данных

      ; Пишем смещение в регистр смещения таблицы векторов
      LDR   r0, =NVIC_VECTTBL
      LDR   r1, =VectorTableBase
      STR   r1, [r0]
      ...
      ; Конфигурируем регистр приоритета
      LDR   r0, =NVIC_AIRCR
      LDR   r1, =0x05FA0500 ; PRIGROUP = 5
      STR   R1, [r0]
      ; Настраиваем вектор IRQ0
      MOV   r0, #0          ; IRQ#0
      LDR   r1, =Irq0_Handler
      BL    SetupIrqHandler
      ; Задаём приоритет
      LDR   r0, =NVIC_IRQPRI
      LDR   r1, =0xC0       ; Приоритет IRQ0
      STRB  r1, [r0,#0]     ; Пишем приоритет IRQ0 со смещением = 0
          ; Примечание : 1-байтная операция
          ; (смещение для IRQ1 будет равно 1)
      DSB ; Барьер синхронизации данных. Гарантирует завершение операций записи
          ; до разрешения прерываний
      MOV   r0, #0 ; Выбираем IRQ0
      BL    EnableIRQ
      ...
      ;------------------------
      ; Функции
      SetupIrqHandler
      ; Вход: R0 = Номер IRQ
      ;       R1 = Обработчик IRQ
      PUSH  {R0, R2, LR}
      LDR   R2, =NVIC_VECTTBL ; Получаем смещение таблицы векторов
      LDR   R2, [R2]
      ADD   R0, #16           ; Номер исключения = номер IRQ + 16
      LSL   R0, R0, #2        ; Умножаем на 4 (размер каждого вектора – 4 байт)
      ADD   R2, R0            ; Находим адрес вектора
      STR   R1, [R2]          ; Записываем адрес обработчика
      POP   {R0, R2, PC}      ; Возвращаемся
EnableIRQ
      ; Вход: R0 = Номер IRQ
      PUSH  {R0 – R3, LR}
      AND   R1, R0, #0x1F     ; Извлекаем младшие 5 бит для формирования
                              ; битовой маски
      MOV   R2, #1
      LSL   R2, R2, R1        ; Маска - в R2
      BIC   R0, #0x1F
      LSR   R0, #3            ; Смещение слова (номер IRQ может быть больше 32)
      LDR   R1, =NVIC_SETEN
      STR   R2, [R1, R0]      ; Устанавливаем бит разрешения
      POP   {R0 – R3, PC}     ; Возвращаемся
```

```
      ;------------------------
      ; Обработчики исключений
Hf_Handler
      ...       ; Здесь должен быть ваш код
      BX    LR  ; Возвращаемся
Nmi_Handler
      ...       ; Здесь должен быть ваш код
      BX    LR  ; Возвращаемся
Irq0_Handler
      ...       ; Здесь должен быть ваш код
      BX    LR  ; Возвращаемся
      ;------------------------
      AREA | Header Data|, DATA
      ALIGN 4
      ; Перемещённая таблица векторов
VectorTableBase SPACE 256 ; Размер (в байтах)
VectorTableEnd            ; (256 / 4 = до 64 исключений)
MyData1 DCD 0             ; Переменные
MyData2 DCD 0
      END                 ; Конец файла
```

Это довольно длинный пример, поэтому будем рассматривать его с конца — с определения области данных.

В секции данных (практически в самом конце программы) мы выделяем под таблицу векторов участок памяти размером 256 байт (`SPACE 256`). В таблице такого размера мы сможем хранить до 64 векторов. При необходимости размер этой области можно скорректировать. Остальные переменные располагаются сразу после таблицы векторов, поэтому адрес переменной `MyData1` равен 0x20000100.

В самом начале приведённого кода мы определяем несколько констант. Использование таких именованных констант вместо непосредственных значений значительно увеличивает читабельность программы.

Итак, исходная таблица векторов содержит вектора сброса, немаскируемого прерывания и исключения Hard Fault. Приведённый пример предназначен для иллюстрации процесса настройки векторов исключений и поэтому не содержит рабочего кода соответствующих обработчиков — операции, выполняемые этими обработчиками, будут зависеть от конкретного приложения. Для возврата из обработчиков исключений в примере используется команда перехода с изменением состояния `BX LR`, однако она может быть заменена любой другой командой, позволяющей выполнить возврат из обработчика (см. **Табл. 9.1**).

После инициализации регистров мы копируем вектора обработчиков в новую таблицу, расположенную в СОЗУ. Данная операция выполняется одной командой групповой загрузки и одной командой группового сохранения. Если необходимо скопировать большее число векторов, то мы можем вставить дополнительные команды групповой загрузки/сохранения и/или увеличить количество слов, пересылаемых при выполнении каждой пары этих команд.

После того как новая таблица будет готова, мы можем переместить её на новое место в ОЗУ. Чтобы быть уверенными в том, что все операции пересылки векторов завершены, мы используем команду синхронизации `DSB`.

После перемещения таблицы векторов мы должны окончательно сконфигурировать прерывание. Прежде всего — задать требуемые установки группирования приоритетов (эту операцию необходимо выполнить всего один раз в начале программы). Для настройки прерываний в примере используются две подпрограммы, `SetupIrqHandler` и `EnableIRQ`, которые позволяют значительно упростить этот процесс. Заменив в коде последней подпрограммы `NVIC_SETEN` на `NVIC_CLREN`, мы получим ещё одну функцию, `DisableIRQ`. После настройки обработчика и задания уровня приоритета прерывания мы наконец-то можем разрешить его.

11.5. Использование команды SVC

Команда `SVC` является общепринятым методом предоставления пользовательским приложениям доступа к программному интерфейсу приложения (API) операционной системы. При этом пользовательские приложения должны «знать» только то, какие параметры следует передавать ОС (информация о расположении API-функций в памяти приложениям не требуется).

Команда `SVC` имеет параметр, представляющий собой 8-битную константу, значение которой хранится в коде команды. Например:

```
SVC  #3   ; Вызов системного сервиса №3
```

Можно использовать и альтернативный синтаксис (без символа «#»):

```
SVC   3   ; Вызов системного сервиса №3
```

Обработчик исключения SVCall может извлечь значение этого параметра из кода команды `SVC`, определив её адрес по значению счётчика команд, сохранённому в стеке при входе в обработчик. Для выполнения указанной операции можно использовать процедуру, блок-схема алгоритма которой приведена на **Рис. 11.2**.

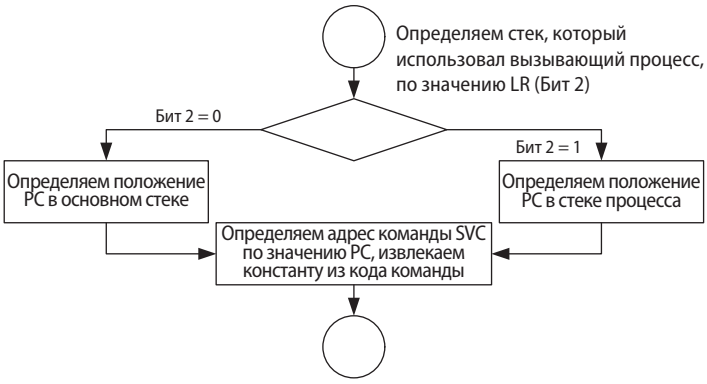


Рис. 11.2. *Алгоритм извлечения параметра команды* `SVC`.

Ниже приведена одна из простейших реализаций данной процедуры:

```
svc_handler
     TST   LR, #0x4         ; Проверяем бит 2 значения EXC_RETURN в LR
     ITE   EQ               ; Если ноль («равно»), то был использован
     MRSEQ R0, MSP          ; основной стек, помещаем MSP в R0
     MRSNE R0, PSP          ; Иначе был использован стек процесса,
```

```
                        ; помещаем PSP в R0
    LDR   R1,[R0,#24]   ; Читаем из стека сохранённое значение PC
    LDRB  R0,[R1,# -2]  ; Извлекаем константу из кода команды
    ; Переданное значение - в R0
    ...
    BX    LR            ; Возвращаемся в вызывающую функцию
```

Определив значение параметра, обработчик исключения может выполнить соответствующую системную функцию. Для реализации переходов к требуемым фрагментам кода удобнее всего использовать команды табличного перехода TBB и TBH. Не забывайте только выполнять проверку на соответствие параметра допустимым значениям, чтобы избежать краха системы из-за вызова некорректной системной функции. Впрочем, если вы абсолютно уверены в корректности передаваемых значений, то эту проверку можно опустить.

Обратите внимание: передача параметров обработчику исключения SVCall и возврат результата выполнения обработчика должны осуществляться через стек. Причины данного требования объясняются в следующем подразделе.

Поскольку одна системная служба не может вызвать другую, используя механизм исключений, обработчик системных вызовов должен напрямую вызывать другие SVC-функции (скажем, используя команду BL).

11.6. Пример использования команды SVC: функции вывода текстовых сообщений

Мы уже разрабатывали с вами различные подпрограммы для вывода информации. Однако не всегда имеется возможность вызова подпрограмм с помощью команды BL — например, если код выполняется на непривилегированном уровне доступа, а функции ввода/вывода текстовых сообщений требуют привилегированного доступа. В подобных случаях мы можем воспользоваться командой SVC в качестве точки входа в функции вывода. Например, пользовательская программа может задействовать команду SVC с разными параметрами для вызова различных сервисов:

```
LDR  R0,=HELLO_TXT
SVC  #0   ; Отобразить строку, адрес которой содержится в R0
MOV  R0,#'A'
SVC  #1   ; Отобразить символ, содержащийся в R0
LDR  R0,=0xC123456
SVC  #2   ; Отобразить шестнадцатеричное значение, содержащееся в R0
MOV  R0,#1234
SVC  #3   ; Отобразить десятичное значение, содержащееся в R0
```

Если таблица векторов была перемещена в ОЗУ, то для использования команды SVC нам необходимо настроить обработчик исключения SVCall. Для этого можно воспользоваться модифицированным вариантом функции SetupIrqHandler из предыдущего раздела. Единственным отличием новой функции будет то, что в качестве входного параметра она будет принимать номер исключения, а не номер прерывания (исключение SVCall имеет номер 11). Заодно оптимизируем код функции, переписав её с использованием 32-битных команд Thumb-2:

```
SetupExcpHandler ; Настройка вектора в таблице векторов, перемещённой в ОЗУ
     ;Вход: R0 = Номер исключения
     ;      R1 = Обработчик исключения
     PUSH   {R0, R2, LR}
     LDR    R2, =NVIC_VECTTBL       ; Получаем смещение таблицы векторов
     LDR    R2, [R2]
     STR.W  R1, [R2, R0, LSL #2]    ; Сохраняем вектор по адресу [R2+R0<<2]
     POP    {R0, R2, PC}            ; Возвращаемся
```

Сам обработчик исключения SVCall мы напишем, взяв за основу код, приведённый в конце предыдущего раздела. Добавим в него команду чтения из стека параметра системного вызова, а также команды переходов к различным системным функциям в зависимости от значения этого параметра:

```
svc_handler
     TST   LR, #0x4      ; Проверяем бит 2 значения EXC_RETURN в LR
     ITE   EQ            ; Если ноль («равно»), то был использован
     MRSEQ R1, MSP       ; основной стек, помещаем MSP в R1
     MRSNE R1, PSP       ; Иначе был использован стек процесса,
                         ; помещаем PSP в R1
     LDR   R0, [R1,#0]   ; Читаем из стека сохранённое значение R0
     LDR   R1, [R1,#24]  ; Читаем из стека сохранённое значение PC
     LDRB  R1, [R1,#-2]  ; Извлекаем константу из кода команды
     ; Теперь значение константы находится в R1, значение параметра - в R0
     PUSH  {LR}          ; Сохраняем LR в стеке
     CBNZ  R1, svc_handler_1
     BL    Puts          ; Вызываем процедуру Puts
     B     svc_handler_end
svc_handler_1
     CMP   R1, #1
     BNE   svc_handler_2
     BL    Putc          ; Вызываем процедуру Putc
     B     svc_handler_end
svc_handler_2
     CMP   R1, #2
     BNE   svc_handler_3
     BL    PutHex        ; Вызываем процедуру PutHex
     B     svc_handler_end
svc_handler_3
     CMP   R1, #3
     BNE   svc_handler_4
     BL    PutDec        ; Вызываем процедуру PutDec
     B     svc_handler_end
svc_handler_4
     B     error         ; Некорректный номер системного вызова
     ...
svc_handler_end
     POP   {PC}          ; Возвращаемся
```

Код обработчика `svc _ handler` должен располагаться рядом с кодом функций вывода, с тем чтобы эти функции оказались в пределах досягаемости команд `BL`.

Заметьте, что для передачи параметров используется не текущее содержимое регистров процессора, а их содержимое, сохранённое в стеке. Это сделано специ-

ально, так как если во время исполнения команды SVC возникнет прерывание с более высоким приоритетом, то обработчик SVCall будет запущен сразу же после завершения остальных обработчиков («цепочечная» обработка). Содержимое регистров R0…R3 и R12 к этому моменту может отличаться от исходного, поскольку при «цепочечной» обработке прерываний отсутствуют промежуточные операции восстановления/сохранения контекста. Например:

1. Параметр помещается в регистр R0.
2. Исполняется команда SVC и одновременно с этим возникает более высокоприоритетное прерывание.
3. Производится сохранение контекста, в результате чего содержимое регистров R0…R3, R12, LR, PC и xPSR оказывается в стеке.
4. Выполняется обработчик прерывания. В процессе выполнения он может изменять содержимое регистров R0…R3 и R12. Это допускается, поскольку данные регистры будут восстановлены при восстановлении контекста.
5. Запускается обработчик исключения SVCall. При входе в обработчик содержимое регистров R0…R3 и R12 может отличаться от того, которое было на момент выполнения команды SVC. Однако корректное значение параметра содержится в стеке, откуда обработчик его и берёт.

Используйте по максимуму возможности режимов адресации

Сравнивая подпрограммы SetupIrqHandler и SetupExcpHandler, можно заметить, как за счёт применения различных режимов адресации, поддерживаемых процессором Cortex-M3, уменьшается размер итогового кода. Так, в процедуре SetupIrqHandler вычисляется адрес вектора IRQ, после чего выполняется запись по этому адресу:

```
SetupIrqHandler /* R0 = номер IRQ, R1 = обработчик IRQ */
   PUSH  {R0, R2, LR}
   LDR   R2, =NVIC_VECTTBL ; Получаем смещение таблицы векторов   ; Шаг 1
   LDR   R2, [R2]                                                 ; Шаг 2
   ADD   R0, #16      ; Номер исключения = номер IRQ + 16         ; Шаг 3
   LSL   R0, R0, #2   ; Умножаем на 4 (каждый вектор – 4 байт)    ; Шаг 4
   ADD   R2, R0       ; Находим адрес вектора                     ; Шаг 5
   STR   R1, [R2]     ; Записываем адрес обработчика              ; Шаг 6
   POP   {R0, R2, PC} ; Возвращаемся
```

В то же время в процедуре SetupExcpHandler для выполнения операций, осуществляемых на шагах 4…6, используется всего одна команда:

```
SetupExcpHandler /* R0 = номер IRQ, R1 = обработчик IRQ */
   PUSH  {R0, R2, LR}
   LDR   R2, =NVIC_VECTTBL    ; Получаем смещение таблицы векторов
   LDR   R2, [R2]
   STR.W R1, [R2, R0, LSL #2] ; Записываем значение вектора по адресу
                              ; [R2 + R0<<2]
   POP   {R0, R2, PC}         ; Возвращаемся
```

В общем случае, мы можем уменьшить число требуемых команд, если адрес ячейки памяти определяется выражениями:

- Rn + (2^N) × Rm
- Rn +/– непосредственное_смещение

Подпрограмму `SetupIrqHandler` можно переписать следующим образом:

```
SetupIrqHandler
   PUSH   {R0, R2, LR}
   LDR    R2, =NVIC_VECTTBL     ; Получаем смещение таблицы векторов  ; Шаг 1
   LDR    R2, [R2]                                                    ; Шаг 2
   ADD    R2, #(16*4)  ; Вычисляем начальный адрес векторов IRQ       ; Шаг 3
   STR.W  R1, [R2, R0, LSL #2]  ; Сохраняем значение вектора          ; Шаг 4
   POP    {R0, R2, PC}          ; Возвращаемся
```

11.7. Использование команды SVC в программах на языке Си

В большинстве случаев для передачи параметров в `SVC`-функции необходимо использовать обработчик, написанный на ассемблере. Это связано с тем, что параметры передаются через стек, а не через регистры (см. предыдущий раздел). Если обработчик исключения SVCall должен быть написан на Си, то для определения адреса, по которому регистр был сохранён в стеке, и для передачи этого значения в обработчик можно использовать простую «обёртку», написанную на ассемблере. Используя указанную информацию, обработчик сможет впоследствии извлечь номер системного вызова и значения параметров. Если вы работаете в среде разработки RVDS или MDK-ARM, то данную «обёртку» можно написать с использованием встроенного ассемблера:

```
// Ассемблерный код-«обёртка» для получения начального адреса стекового фрейма
// Начальный адрес фрейма помещается в регистр R0, после чего осуществляется
// переход к собственно обработчику SVCall
__asm void svc_handler_wrapper(void)
{
  TST   LR, #4
  ITE   EQ
  MRSEQ R0, MSP
  MRSNE R0, PSP
  B __cpp(svc_handler)
} // Команда возврата (BX LR) в конце этой «обёртки» не нужна, поскольку
  // при возврате из svc_handler управление будет передано
  // на то место, откуда была вызвана команда SVC
```

Основная часть обработчика исключения SVCall может быть реализована в виде процедуры на языке Си с одним аргументом (регистр R0), в котором передаётся стартовый адрес стекового фрейма исключения. Это значение используется для выделения номера системного сервиса и передачи реальных параметров исключения (регистры R0…R3):

```
// Обработчик SVCall, написанный на Си, принимающий в виде параметра начальный
// адрес стекового фрейма, который используется в качестве указателя на массив
```

```
// аргументов
// svc_args[0] = R0 , svc_args[1] = R1
// svc_args[2] = R2 , svc_args[3] = R3
// svc_args[4] = R12, svc_args[5] = LR
// svc_args[6] = Адрес возврата (сохранённое значение PC)
// svc_args[7] = xPSR
void svc_handler(unsigned int * svc_args)
{
  unsigned int svc_number;
  unsigned int svc_r0;
  unsigned int svc_r1;
  unsigned int svc_r2;
  unsigned int svc_r3;
  svc_number = ((char *) svc_args[6])[-2]; // Memory[(Stacked PC)-2]
  svc_r0 = ((unsigned long) svc_args[0]);
  svc_r1 = ((unsigned long) svc_args[1]);
  svc_r2 = ((unsigned long) svc_args[2]);
  svc_r3 = ((unsigned long) svc_args[3]);
  printf («Номер SVC = %xn», svc_number);
  printf («0-й параметр SVC = %x\n», svc_r0);
  printf («1-й параметр SVC = %x\n», svc_r1);
  printf («2-й параметр SVC = %x\n», svc_r2);
  printf («3-й параметр SVC = %x\n», svc_r3);
  return;
}
```

Обратите внимание, что супервизор не может возвращать результаты своей работы в вызывающую функцию так, как это делают обычные Си-функции. Обычные функции имеют тип, отличный от `void`, например `unsigned int func()`, и используют для передачи возвращаемого значения оператор `return`, который в действительности помещает это значение в регистр R0. Если при выходе из обработчика исключения SVCall загрузить возвращаемые значения в любой из регистров R0…R3, то они будут утеряны при восстановлении контекста. Соответственно, если обработчик должен возвращать какие-то результаты в вызывающую программу, то он должен модифицировать непосредственно стековый фрейм, чтобы эти значения оказались в регистрах при восстановлении контекста.

При работе в пакетах RVDS компании ARM или MDK-ARM компании Keil мы можем использовать для вызова супервизора ключевое слово компилятора `__svc`. Например, если нам необходимо передать четыре параметра в SVC-функцию с номером 3, то мы можем объявить такую функцию с именем `call_svc_3` как:

```
void __svc(0x03) call_svc_3(unsigned long svc_r0, unsigned long
svc_r1, unsigned long svc_r2, unsigned long svc_r3);
```

Вызывается эта функция так же как и любая другая:

```
int main(void)
{
  unsigned long p0, p1, p2, p3; // Параметры, передаваемые обработчику SVCall
  ...
  call_svc_3(p0, p1, p2, p3);   // Вызываем SVC №3 с аргументами p0, p1, p2, p3
  ...
```

```
  return;
}
```

Дополнительную информацию об использовании ключевого слова `__svc` можно найти в руководстве [4].

Поскольку в компиляторе GCC инструментария GNU ключевое слово `__svc` отсутствует, то при использовании данного инструментария для вызова супервизора из Си-программ нам придётся прибегнуть к помощи inline-ассемблера. Так, если нам необходимо вызвать функцию супервизора с номером 3, которая имеет один параметр и возвращает результат в регистре R0 (согласно спецификации AAPCS [5] первое значение всегда передаётся через регистр R0), то мы можем использовать следующий код:

```
int MyDataIn = 0x123;
__asm __volatile («mov R0, %0\n»
                  «svc 3 \n» : «» : «»r» (MyDataIn) );
```

Приведённый выше код можно разбить на три части, где входные данные определяются параметром `r (MyDataIn)`, а выходные данные отсутствуют (обозначено как «»):

```
__asm ( ассемблерный_код : список_выходных_параметров : список_входных_параметров
)
```

Другие примеры использования inline-ассемблера при работе с инструментарием GNU вы сможете найти в Главе 19. Для получения детальной информации о передаче параметров в/от inline-ассемблера обратитесь к документации GNU.

ГЛАВА 12
ПРОДВИНУТЫЕ ПРОГРАММНЫЕ ВОЗМОЖНОСТИ И ПОВЕДЕНИЕ СИСТЕМЫ

12.1. Реализация системы с двумя раздельными стеками

Одной из наиболее важных особенностей архитектуры ARMv7-M является возможность отделения стека пользовательского приложения от стека, используемого привилегированным кодом, таким как ядро системы. Если в устройстве реализован опциональный модуль защиты памяти (MPU), то его можно задействовать для того, чтобы запретить пользовательским приложениям доступ к стеку ядра. Это не позволит им нарушить работу ядра, испортив содержимое памяти.

Любая надёжная система на базе процессора Cortex-M3, как правило, имеет следующие характеристики:

- обработчики исключительных ситуаций, использующие основной стек (MSP);
- код ядра, запускаемый с определённой периодичностью в исключении системного таймера (SYSTICK). Этот код работает на привилегированном уровне доступа, осуществляя диспетчеризацию задач и управление системой;
- пользовательские приложения, выполняющиеся в виде отдельных потоков с пользовательским (непривилегированным) уровнем доступа; эти приложения используют стек процесса (PSP);
- ядро системы и обработчики исключительных ситуаций, задействующие основной стек, адресуемый указателем MSP; при наличии модуля MPU доступ к этой области памяти предоставляется только на привилегированном уровне;
- стек пользовательских приложений, адресуемый указателем PSP.

Возьмём систему с модулем MPU, использующую в качестве системной памяти встроенное статическое ОЗУ. Мы можем так сконфигурировать модуль MPU, что всё СОЗУ окажется разделено на две области, одна из которых будет доступна для обращений на пользовательском уровне, а другая — на привилегированном уровне (**Рис. 12.1**). Каждая из областей используется как для хранения данных, так и для размещения стека. Поскольку в процессоре Cortex-M3 применяется модель полного «убывающего» стека, то перед началом работы каждый указатель стека должен быть установлен на вершину соответствующей области памяти.

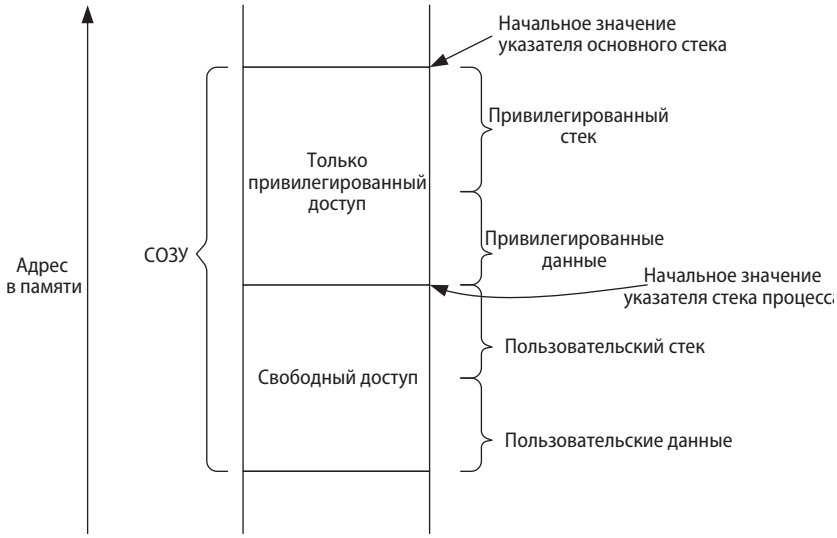


Рис. 12.1. *Разбиение памяти на две области.*

После подачи питания инициализируется только MSP (в процессе сброса в него заносится значение, расположенное по 0-му адресу). Для конфигурирования отказоустойчивой системы с двумя стеками необходимо выполнить дополнительные операции. При написании приложения на ассемблере код инициализации может выглядеть следующим образом:

```
; Код начинает выполняться на привилегированном уровне (он располагается
; в области памяти, доступной на пользовательском уровне)
BL   MpuSetup          ; Задаём области MPU и разрешаем защиту памяти
LDR  R0, =PSP_TOP      ; Устанавливаем PSP на вершину стека процесса
MSR  PSP, R0
BL   SystickSetup      ; Конфигурируем таймер SYSTICK и его исключение
                       ; для периодического вызова ядра ОС
MOV  R0, #0x3          ; Конфигурируем регистр CONTROL таким образом,
                       ; чтобы пользовательская программа использовала PSP
MSR  CONTROL, R0       ; и переключаем текущий уровень доступа на пользовательский
ISB                    ; Барьер синхронизации команд
B    UserApplicationStart  ; Теперь мы работаем на пользовательском уровне
                           ; доступа
                           ; Переходим к выполнению пользовательского кода
```

Всё это прекрасно работает в случае ассемблера, однако при программировании на Си переключение указателей стека в середине функции или процедуры может привести к порче локальных переменных, поскольку эти переменные могут располагаться в стеке. В руководстве [1] предполагается использовать процедуру обработки какого-либо исключения (например, SVCall) для вызова ядра с последующим изменением указателя стека модификацией значения EXC_RETURN (**Рис. 12.2**).

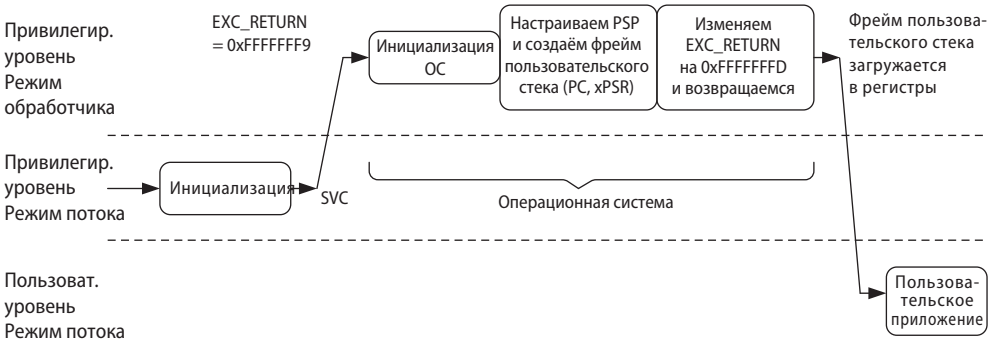


Рис. 12.2. *Инициализация нескольких стеков в простой ОС.*

В большинстве случаев модификация значения EXC_RETURN и переключение стеков осуществляется ядром операционной системы (ОС). После запуска пользовательского приложения исключение SYSTICK может использоваться для периодического обращения к ядру ОС, осуществляющему управление системой и, при необходимости, инициирующему переключение контекста (**Рис. 12.3**).

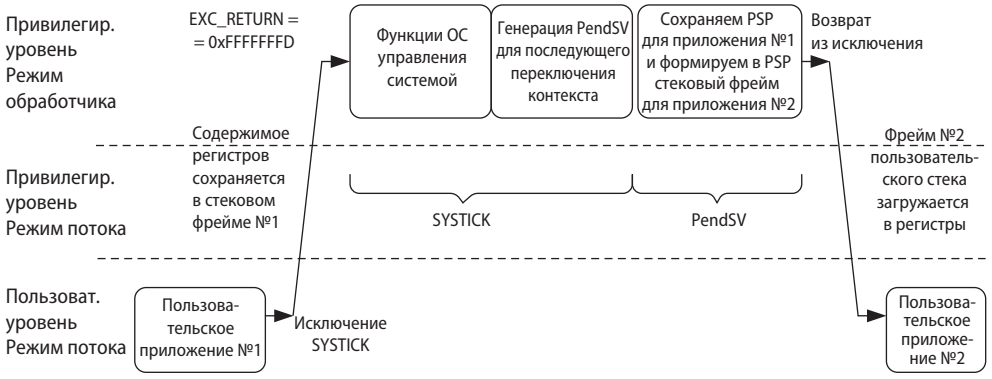


Рис. 12.3. *Переключение контекста в простой ОС.*

Обратите внимание, что собственно переключение контекста осуществляется в низкоприоритетном исключении PendSV, чтобы исключить выполнение данной операции в середине обработчика прерывания.

Многие приложения прекрасно обходятся без ОС, однако и в этом случае использование отдельных стеков для разных секций программного кода позволяет увеличить надёжность устройства. Для реализации такой системы можно при запуске процессора Cortex-M3 установить указатель MSP на область стека процесса. В этом случае будет инициализирован стек процесса, но с использованием указателя MSP. А перед запуском пользовательского приложения надо будет выполнить следующий код:

```
; Код начинает выполняться на привилегированном уровне,
; MSP указывает на пользовательский стек
MpuSetup();              // Задаём области MPU и разрешаем защиту памяти
SystickSetup();          // Конфигурируем таймер SYSTICK и его исключение,
                         // реализующее функции управления системой
```

```
SwitchStackPointer();     // Вызываем ассемблерную подпрограмму для переключения
                          // указателей стека
   /*; ------Тело подпрограммы SwitchStackPointer -----
   PUSH {R0, R1, LR}
   MRS  R0, MSP          ; Сохраняем текущий указатель стека
   LDR  R1, =MSP_TOP     ; Устанавливаем MSP на другую область памяти
   MSR  MSP, R1
   MSR  PSP, R0          ; Сохраняем текущее значение указателя стека в PSP
   MOV  R0, #0x3
   MSR  CONTROL, R0      ; Переключаемся в пользовательский режим и используем
                         ; стек процесса в качестве текущего стека
   POP  {R0, R1, PC}     ; Возвращаемся
   ; ------ Обратно в Си-программу -----*/
; Теперь мы находимся в пользовательском режиме и используем PSP
; Все локальные переменные остались на месте
UserApplicationStart();   // Запускаем приложение в пользовательском режиме
```

12.2. Выравнивание стека на границу двойного слова

В приложениях, написанных с учётом требований стандарта AAPCS[1)], указатель стека при входе в функцию должен быть выровнен на границу двойного слова. Для этого при обработке исключений соответствующим образом корректируются адреса стека, по которым сохраняются регистры. Чтобы включить эту возможность, необходимо установить бит STKALIGN регистра управления конфигурацией CCR контроллера NVIC (см. **Табл. Г.18** в Приложении Г). При использовании CMSIS-совместимого драйвера это можно сделать с помощью выражения:

```
SCB->CCR = SCB->CCR | 0x200;
```

При программировании на Си без использования CMSIS:

```
#define NVIC_CCR *((volatile unsigned long *) (0xE000ED14))
NVIC_CCR = NVIC_CCR | 0x200; /* Устанавливаем бит STKALIGN в NVIC */
```

Эту операцию можно выполнить и на языке ассемблера:

```
LDR   R0, =0xE000ED14 ; Загружаем в R0 адрес регистра CCR
LDR   R1, [R0]
ORR.W R1, R1, #0x200  ; Устанавливаем бит STKALIGN
STR   R1, [R0]        ; Пишем новое значение в регистр CCR
```

Если на момент сохранения контекста при обработке исключительной ситуации бит STKALIGN был установлен, то бит 9 сохранённого в стеке регистра xPSR (объединённый регистр состояния программы) показывает, корректировался ли указатель стека при сохранении контекста. При восстановлении контекста из стека процессор проверяет состояние 9-го бита xPSR и соответствующим образом корректирует значение указателя стека.

[1)] Соглашения по вызову процедур для архитектуры ARM (Procedure Call Standard for ARM Architecture — AAPCS) [5]. Соответствующие рекомендации, касающиеся выравнивания указателя стека и стандарта AAPCS, были опубликована на сайте компании ARM; см. *http://infocenter.arm.com/help/topic/com.arm.doc.ihi0046a/IHI0046A_ABI_Advisory_1.pdf.*

Во избежание повреждения данных, хранящихся в стеке, бит STKALIGN не следует изменять в обработчике исключений. Это может привести к несоответствию значений указателя до и после исключения.

Возможность выравнивания стека появилась в 1-й ревизии процессора Cortex-M3. Старые устройства, построенные на процессоре ревизии 0, такой возможности не имеют. Во 2-й ревизии процессора эта возможность разрешена по умолчанию, тогда как в 1-й ревизии её необходимо разрешать программно.

Ещё раз повторю, что выравнивание стека на границу двойного слова необходимо, если ваше приложение должно удовлетворять требованиям стандарта AAPCS.

12.3. Переход в режим потока с любого уровня вложенности

В процессоре Cortex-M3 предусмотрена возможность переключения выполняющегося обработчика прерывания с привилегированного уровня на пользовательский. Это необходимо в тех случаях, когда код обработчика прерывания является частью пользовательского приложения и не должен иметь привилегированных прав доступа к данным. Указанная функция называется «Nonbase Thread Enable» и включается установкой бита NONBASETHRDENA регистра управления конфигурацией CCR контроллера NVIC.

Используйте эту возможность с осторожностью

Поскольку данная функция требует ручной корректировки указателя стека и модификации находящихся в стеке данных, её не следует использовать при обычном прикладном программировании. Если же без этой возможности никак не обойтись, то применять её следует очень аккуратно, не забывая обеспечивать корректное завершение процедуры прерывания. В противном случае, прерывания с таким же или меньшим уровнем приоритета могут оказаться маскированными.

Для использования данной функции применяется перенаправление обработчика исключительной ситуации. Вектор в таблице векторов указывает на обработчик, выполняющийся на привилегированном уровне, но размещённый в области памяти, доступной в пользовательском режиме.

```
redirect_handler
    PUSH {LR}
    SVC  0              ; SVC-функция для переключения из привилегированного
                        ; режима в пользовательский
    BL   User_IRQ_Handler
    SVC  1              ; SVC-функция для возврата из пользовательского режима
                        ; в привилегированный
    POP  {PC}           ; Возвращаемся
```

Обработчик исключения SVCall состоит из трёх частей:

- в первой части определяется значение параметра, переданного в команде SVC;
- во второй части (обработчик системного сервиса №0) устанавливается бит NONBASETHRDENA, выравнивается пользовательский стек, корректируется значение EXC_RETURN и выполняется возврат в обработчик (уже на пользовательском уровне доступа) с использованием стека процесса;

- в третьей части (обработчик системного сервиса №1) сбрасывается бит NONBASETHRDENA, восстанавливается значение указателя пользовательского стека и выполняется возврат в обработчик (на привилегированном уровне доступа) с использованием основного стека.

```
svc_handler
    TST   LR, #0x4          ; Проверяем бит 2 значения EXC_RETURN
    ITE   EQ                ; Если = 0, то
    MRSEQ R0, MSP           ; загружаем корректный указатель стека в R0
    MRSNE R0, PSP
    LDR   R1,[R0, #24]      ; Читаем из стека PC
    LDRB  R0,[R1, #-2]      ; Извлекаем параметр с адреса PC - 2
    CBZ   R0, svc_service_0
    CMP   R0, #1
    BEQ   svc_service_1
    B.W   Unknown_SVC_Request
svc_service_0    ; Сервис переключения обработчика из привилегированного режима
                 ; в пользовательский
    MRS   R0, PSP           ; Корректируем PSP
    SUB   R0, R0, #0x20     ; PSP = PSP - 0x20
    MSR   PSP, R0
    MOV   R1, #0x20         ; Копируем стековый фрейм из основного стека
                            ; в стек процесса
svc_service_0_copy_loop
    SUBS  R1, R1, #4
    LDR   R2, [SP, R1]
    STR   R2, [R0, R1]
    CMP   R1, #0
    BNE   svc_service_0_copy_loop
    STRB  R1, [R0, #0x1C]   ; Сбрасываем в 0 регистр IPSR, находящийся в стеке
                            ; процесса
    LDR   R0, =0xE000ED14   ; Разрешаем функцию «Nonbase thread enable» в CCR
    LDR   R1, [r0]
    ORR   R1, #1
    STR   R1, [r0]
    ORR   LR, #0xC          ; Меняем LR для возврата в режим потока, используя PSP
    BX    LR
svc_service_1    ; Сервис переключения обработчика из пользовательского режима
                 ; в привилегированный
    MRS   R0, PSP           ; Изменяем PC в стеке привилегированного режима
                            ; так, чтобы при возврате попасть на команду,
    LDR   R1, [R0, #0x18]   ; расположенную после 2-й команды SVC обработчика
    STR   R1, [SP, #0x18]   ; перенаправления
    MRS   R0, PSP           ; Заносим в PSP значение, которое в нём было
                            ; до выполнения 1-й команды SVC
    ADD   R0, R0, #0x20
    MSR   PSP, R0
    LDR   R0, =0xE000ED14   ; Сбрасываем бит NONBASETHRDENA в CCR
    LDR   R1, [r0]
    BIC   R1, #1
    STR   R1, [r0]
```

```
BIC   LR, #0xC          ; Возвращаемся в режим обработчика,
                        ; используя основной стек
BX    LR
```

Исключение SVCall используется потому, что регистр состояния прерывания (IPSR) может быть изменён только при возврате из обработчика исключительной ситуации. Разумеется, можно использовать и другие исключения, например программно генерируемые прерывания. Однако этого делать не рекомендуется, поскольку такие прерывания являются «неточными» и могут оказаться маскированными, т.е. существует вероятность задержки выполнения требуемых операций копирования и переключения стека. Последовательность выполнения кода показана на **Рис. 12.4**. На этом же рисунке показано изменение указателей стека и текущего приоритета исключений. Моменты ручной корректировки указателя PSP в обработчике SVCall помечены пунктирными овалами.

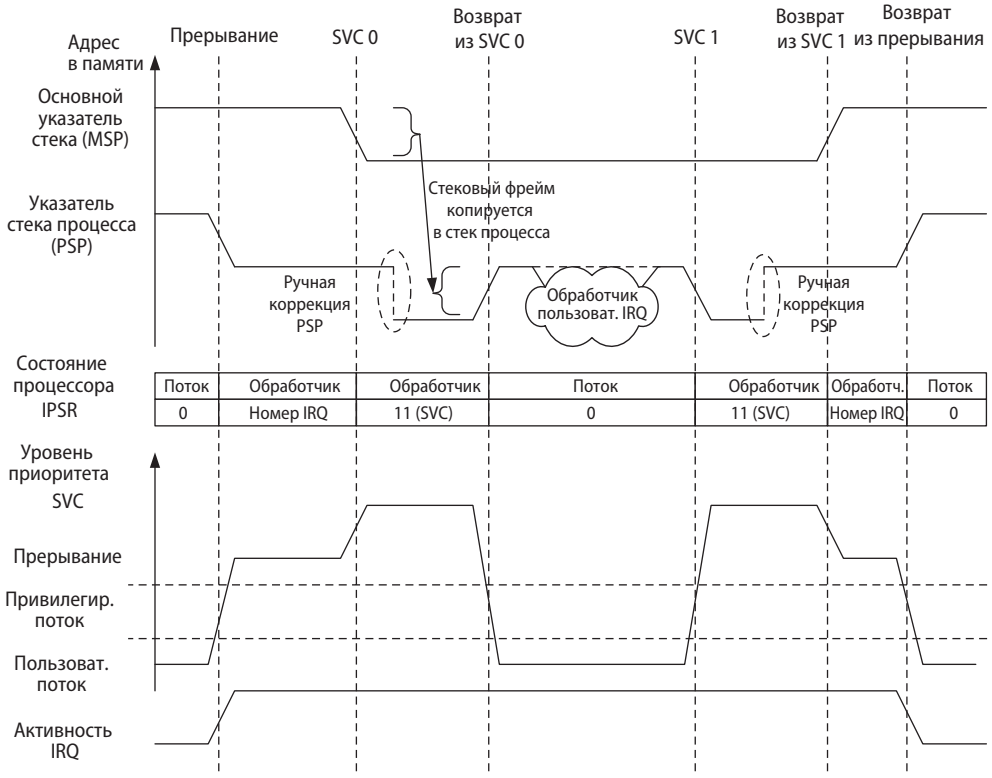


Рис. 12.4. *Использование бита NONBASETHRDENA.*

12.4. Пара слов о производительности

Чтобы выжать из процессора Cortex-M3 максимум возможного, нужно учитывать некоторые моменты. Во-первых, необходимо исключить циклы ожидания при обращении к памяти. При проектировании микроконтроллера или SoC разработчик должен так оптимизировать систему памяти, чтобы обращения к командам

и данным выполнялись за одинаковое время, а также использовать 32-битную память. Карта памяти должна быть такой, чтобы программный код исполнялся бы из области кода, а большинство обращений к данным шло бы по системной шине. Это позволит обращаться к данным одновременно с выборкой команд.

Во-вторых, таблица векторов прерываний также должна быть, по возможности, размещена в области кода. Это позволит одновременно выполнять выборку вектора и сохранение контекста в стеке. При нахождении таблицы векторов в ОЗУ задержка прерываний может увеличиться на несколько тактов, поскольку и для выборки вектора, и для сохранения контекста в этом случае будет задействоваться системная шина (если только стек не размещён в области кода, где используется шина D-Code).

По возможности избегайте обращений к невыровненным данным. Невыровненные пересылки преобразуются в две и более пересылок по шине AHB), что замедляет выполнение программы. Так что уделите должное внимание структуре данных своей программы.

Те, кто пишет программы на ассемблере, могут также использовать ряд трюков, позволяющих ускорить выполнение определённых операций.

1. Используйте команды доступа к памяти со смещением. Если необходимо обращаться к различным ячейкам памяти в пределах некоторого диапазона адресов, то вместо кода

```
LDR    R0, =0xE000E400       ; Устанавливаем приоритет прерываний №3...№0
LDR    R1, =0xE0C02000       ; Уровни приоритетов
STR    R1, [R0]
LDR    R0, =0xE000E404       ; Устанавливаем приоритет прерываний №7...№4
LDR    R1, =0xE0E0E0E0       ; Уровни приоритетов
STR    R1, [R0]
```

   можно написать следующее:

```
LDR    R0, =0xE000E400       ; Устанавливаем приоритет прерываний №3...№0
LDR    R1, =0xE0C02000       ; Уровни приоритетов
STR    R1, [R0]
LDR    R1, =0xE0E0E0E0       ; Уровни приоритетов
STR    R1, [R0,#4]           ; Устанавливаем приоритет прерываний №7...№4
```

   Вторая команда сохранения использует смещение относительно первого адреса, уменьшая таким образом число команд.

2. Применяйте для обращения к памяти команды групповой загрузки/сохранения. Так, используя команду `STM`, предыдущий пример можно переписать следующим образом:

```
LDR    R0, =0xE000E400       ; Базовый адрес группы регистров приоритета
LDR    R1, =0xE0C02000       ; Уровни приоритетов прерываний №3...№0
LDR    R2, =0xE0E0E0E0       ; Уровни приоритетов прерываний №7...№4
STMIA  R0, {R1, R2}
```

3. Используйте `IT`-блоки для замены коротких условных переходов. Поскольку процессор Cortex-M3 имеет конвейер, при выполнении операции перехода возникает штраф ветвления. Если условный переход применяется для пропуска нескольких команд, то его с успехом можно заменить `IT`-блоком, что позволит сэкономить несколько тактов.

4. Если операцию можно выполнить с помощью двух команд Thumb либо с помощью одной команды Thumb-2, то необходимо использовать последний вариант, поскольку он обеспечивает меньшее время исполнения при том же размере кода.

12.5. Состояние блокировки

При возникновении какого-либо отказа генерируется соответствующее исключение. Если ещё один отказ произойдёт во время выполнения обработчика Usage Fault/Bus Fault/MemManage Fault, то будет запущен обработчик исключения Hard Fault. А что будет, если отказ произойдёт внутри обработчика Hard Fault? В этом случае процессор перейдёт в состояние блокировки (**Рис. 12.5**).

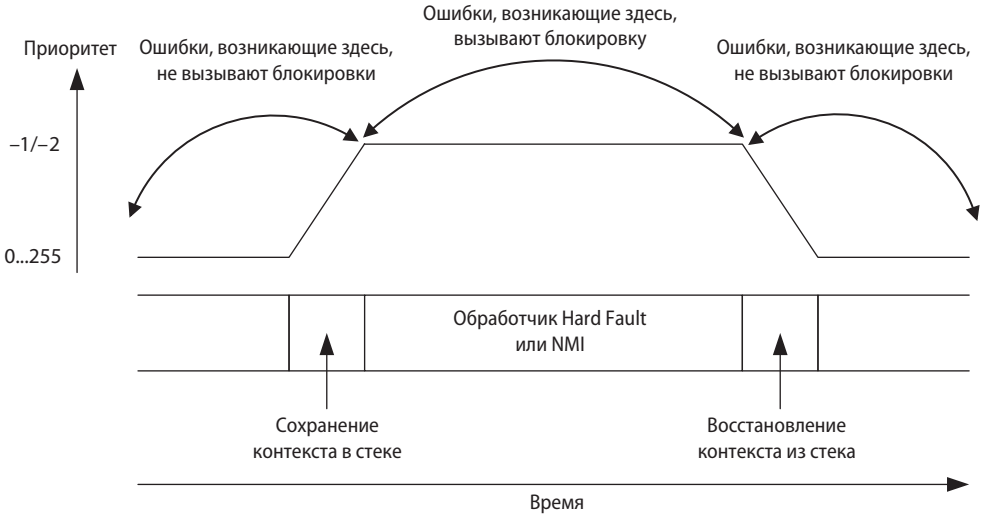


Рис. 12.5. *Возникновение состояния блокировки.*

12.5.1. Что происходит во время блокировки?

При переходе процессора в состояние блокировки в счётчик команд загружается значение 0xFFFFFFFX, после чего процессор продолжает выборку команд с этого адреса. Кроме того, активируется выходной сигнал LOCKUP процессора, сигнализируя о данной ситуации. Разработчики устройств могут использовать этот сигнал для активации схемы сброса системы.

Состояние блокировки может наступить, если:

- отказ возникнет внутри обработчика исключения Hard Fault (двойной отказ);
- отказ возникнет внутри обработчика немаскируемого прерывания;
- отказ шины возникнет во время цикла сброса (при выборке начального значения SP или PC).

В случае двойного отказа процессорное ядро сохраняет возможность реагирования на немаскируемое прерывание и запуска соответствующего обработчика. Однако после завершения обработчика процессор вернётся в состояние бло-

кировки, при этом значение счётчика команд снова станет равным 0xFFFFFFFX. В данном случае происходит блокировка системы с уровнем приоритета, равным –1. При возникновении немаскируемого прерывания процессор перейдёт к его обработке, поскольку приоритет этого прерывания (–2) больше текущего уровня приоритета (–1). После завершения обработки NMI и возврата в состояние блокировки текущий уровень приоритета вновь станет равным –1.

Как правило, наилучшим способом выхода из состояния блокировки является сброс устройства. При подключённом отладчике также можно остановить процессор, записать в PC другое значение и продолжить выполнение программы с нового адреса. Однако в большинстве случаев это может оказаться не лучшим вариантом, поскольку ряд регистров, в том числе регистры системы прерываний, могут потребовать повторной инициализации, прежде чем система сможет вернуться в рабочее состояние.

Вы спросите, почему же мы просто не сбрасываем ядро при возникновении блокировки? В готовой системе, возможно, так и придётся делать, однако при разработке программного обеспечения мы прежде всего должны выявить источник проблемы. Если мы сразу же сбросим ядро, то мы не сможем понять причину отказа, поскольку регистры будут сброшены и состояние системы изменится. В большинстве микроконтроллеров на базе процессора Cortex-M3 для сброса ядра при его попадании в состояние блокировки можно использовать сторожевой таймер.

Обратите внимание, что отказ шины во время сохранения контекста при входе в обработчик Hard Fault или обработчик NMI не вызовет перехода в состояние блокировки, однако исключение Bus Fault всё же будет отложено.

12.5.2. Предотвращение блокировки

При написании обработчика NMI или Hard Fault необходимо сделать всё возможное, чтобы избежать блокировки процессора. К примеру, можно исключить все необязательные обращения к стеку в обработчике Hard Fault до тех пор, пока мы не убедимся, что память работает нормально и указатель стека имеет корректное значение. При разработке сложных систем одной из возможных причин возникновения отказов шины или отказов системы управления памятью является порча указателя стека. Если мы поместим в начале обработчика Hard Fault код, наподобие следующего:

```
hard_fault_handler
    PUSH {R4-R7, LR} ; Не лучшая идея до тех пор, пока вы не убедитесь
                     ; в безопасности работы со стеком!
    . . .
```

то при возникновении отказа из-за ошибки во время обращения к стеку, мы сразу же перейдём в состояние блокировки. Вообще говоря, в обработчиках исключений Hard Fault, Bus Fault и MemManage Fault желательно всегда проверять указатель стека на корректность, прежде чем продолжать работу со стеком. При написании обработчика NMI мы можем попытаться уменьшить вероятность появления ошибок, вызванных операциями со стеком, используя только регистры R0…R3 и R12, которые сохраняются в стеке автоматически.

Одно из основных правил, которому необходимо следовать при написании обработчиков исключения Hard Fault и немаскируемого прерывания, звучит следующим образом: в обработчике следует выполнять только самые необходимые операции, а для выполнения остальных задач, скажем для вывода сообщений об ошибках, использовать отдельное исключение (например, PendSV) или программное прерывание. Это позволит сделать указанные обработчики небольшими по размеру и надёжными.

Не вздумайте вызывать в обработчиках NMI и Hard Fault команду `SVC`. Поскольку приоритет исключения SVCall всегда меньше приоритетов Hard Fault и немаскируемого прерывания, выполнение команды `SVC` в этих обработчиках вызовет блокировку системы. Данное требование может показаться простым, однако при разработке сложных систем, когда используются функции, описанные в разных файлах, вы можете случайно, сами того не желая, вызвать в обработчике NMI или Hard Fault функцию, содержащую команду `SVC`. Поэтому, прежде чем приступить к написанию программы, необходимо тщательно продумать реализацию обработчика системных вызовов.

12.6. Регистр FAULTMASK

Регистр FAULTMASK используется для перевода конфигурируемых обработчиков исключений отказов (Bus Fault, Usage Fault или MemManage Fault) на уровень исключения Hard Fault, не дожидаясь возникновения последнего из-за реального отказа. Это позволяет конфигурируемым обработчикам исключений отказов выступать в роли обработчика Hard Fault. При этом обработчик исключения отказа получает следующие возможности:

1. Маскирование отказов шины установкой бита BFHFNMIGN регистра управления конфигурацией. Данная возможность может использоваться для тестирования шинной системы без риска возникновения блокировки. Например, для проверки корректной работы моста.
2. Обход модуля MPU. Данная возможность позволяет обработчику отказа обращаться к областям памяти, защищённым модулем MPU, не требуя, таким образом, перепрограммирования последнего для выполнения нескольких пересылок, предназначенных для исправления ошибки.

Использование регистра FAULTMASK отличается от использования регистра PRIMASK. Последний обычно задействуется в коде, критичном ко времени исполнения, однако он не позволяет маскировать отказы шины или обходить установки модуля MPU. При установленном регистре PRIMASK все конфигурируемые обработчики исключительных ситуаций поднимаются на уровень обработчика Hard Fault. В то же время регистр FAULTMASK позволяет конфигурируемым обработчикам исключений отказов разрешать проблемы, связанные с системой памяти, за счёт возможностей, в обычном режиме доступных лишь обработчику исключения Hard Fault. Однако даже при установленном регистре FAULTMASK отказы, вызванные такими операциями, как выборка некорректной команды или использование команды `SVC` на неверном уровне приоритета, всё равно могут вызвать блокировку процессора.

ГЛАВА 13
МОДУЛЬ ЗАЩИТЫ ПАМЯТИ MPU

13.1. Общие сведения

Архитектура процессора Cortex-M3™ предусматривает наличие опционального модуля защиты памяти (Memory Protection Unit — MPU). Будучи реализованным в микроконтроллере или системе на кристалле, этот модуль осуществляет защиту памяти, что позволяет повысить надёжность разрабатываемых изделий. Перед использованием модуля MPU его необходимо сконфигурировать и включить. При отключённом модуле MPU система памяти будет функционировать так же, как и при его отсутствии.

Модуль MPU может повысить надёжность встраиваемой системы следующим образом:

- не позволяя пользовательским приложениям повреждать данные, используемые операционной системой;
- разделяя данные между задачами посредством блокирования доступа к данным любой задачи со стороны остальных задач;
- позволяя назначать областям памяти атрибут «только для чтения» для защиты наиболее важных данных;
- обнаруживая неожиданные обращения к памяти (скажем, при порче стека).

Кроме того, модуль MPU может использоваться для назначения различным областям памяти атрибутов доступа, определяющих возможности кэширования и буферизации.

Для реализации функций защиты памяти модуль MPU разбивает всё адресное пространство процессора на несколько областей. В общей сложности можно задать до восьми областей. Помимо этого, можно определить «фоновую» карту памяти, которая будет использоваться по умолчанию на привилегированном уровне доступа. Обращения к ячейкам памяти, которые не входят ни в одну из областей модуля MPU или доступ к которым запрещён параметрами области, вызовут генерацию исключения MemManage Fault.

Области MPU могут перекрываться. Если ячейка памяти относится к двум областям, то атрибуты доступа к памяти и права доступа, используемые при обращении к этой ячейке, будут определяться установками области с наибольшим номером. Так, если адрес пересылки находится в диапазоне адресов, соответствующем 1-й и 4-й областям, то будут использоваться установки 4-й области.

13.2. Регистры модуля MPU

Модуль MPU имеет ряд специальных регистров. Один из них — это регистр типа модуля MPU_TYPE, который позволяет определить наличие данного модуля в устройстве. Если при чтении поля DREGION возвращается нулевое значение, значит, модуль MPU не реализован (**Табл. 13.1**).

Таблица 13.1. **Регистр типа MPU MPU_TYPE (0xE000ED90)**

Биты	Обозначение	Тип	Значение после сброса	Описание
23:16	IREGION	R	0	Число областей команд, поддерживаемых данным MPU; поскольку в архитектуре ARMv7-M используется унифицированный модуль MPU, это поле всегда содержит 0
15:8	DREGION	R	0 или 8	Число областей, поддерживаемых данным MPU; в процессоре Cortex-M3 это значение равно либо 0 (MPU отсутствует), либо 8 (MPU присутствует)
0	SEPARATE	R	0	Этот бит всегда равен 0, поскольку используется унифицированный модуль MPU

Для управления модулем предназначено несколько регистров. Прежде всего, это регистр управления модулем MPU_CTRL (**Табл. 13.2**). В данном регистре расположены три управляющих бита. При сбросе в регистр заносится нулевое значение, что соответствует отключённому модулю MPU. Для разрешения работы модуля MPU необходимо задать параметры всех областей MPU, после чего установить бит ENABLE регистра управления.

Таблица 13.2. **Регистр управления MPU MPU_CTRL (0xE000ED94)**

Биты	Обозначение	Тип	Значение после сброса	Описание
2	PRIVDEFENA	R/W	0	Разрешение доступа к карте памяти в привилегированном режиме. При установленном бите PRIVDEFENA и включённом модуле MPU вся карта памяти рассматривается как «фоновая» область, доступная для привилегированных обращений. Если данный бит сброшен, то «фоновая» область отсутствует, а обращение к ячейке памяти, не входящей в какую-либо область MPU, вызывает отказ системы управления памятью
1	HFNMIENA	R/W	0	Этот бит разрешает использование модуля MPU в обработчиках NMI и исключения Hard Fault. Если бит сброшен, то при выполнении данных обработчиков модуль MPU не используется (блокируется)
0	ENABLE	R/W	0	Установка этого бита в 1 разрешает работу модуля MPU

Если не задано ни одной области MPU, то установленный бит PRIVDEFENA позволяет привилегированному коду обращаться к любым ячейкам памяти в преде-

лах всего адресного пространства, в то время как обращения из пользовательского кода будут блокироваться. Если же имеются сконфигурированные и разрешённые области MPU, то их настройки будут иметь более высокий приоритет. В качестве примера можно рассмотреть две системы с одинаковыми настройками MPU, в одной из которых бит PRIVDEFENA будет установлен в 1 (правая часть **Рис. 13.1**). Установленный бит PRIVDEFENA разрешает в последней системе привилегированный доступ к ячейкам памяти, не входящим ни в одну из областей MPU.

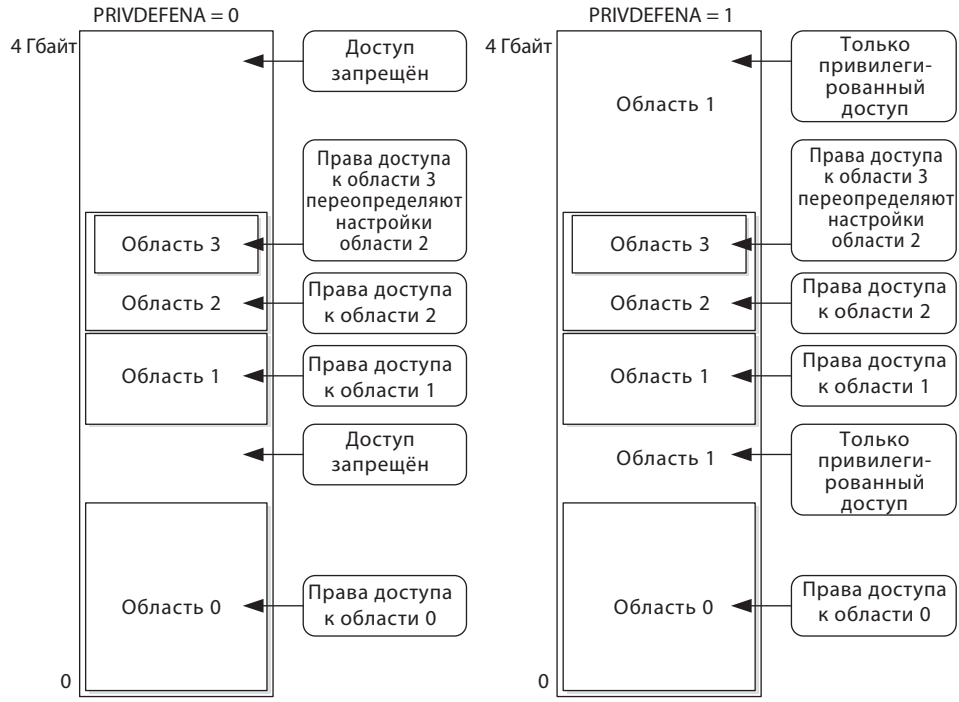


Рис. 13.1. *Влияние бита PRIVDEFENA.*

Установка бита разрешения в регистре управления обычно является последней операцией при настройке модуля MPU. Если не следовать этому правилу, то в процессе конфигурирования областей MPU могут случайным образом генерироваться различные исключения отказов.

Следующим регистром, используемым для управления модулем MPU, является регистр номера области MPU_RNR (**Табл. 13.3**). Перед заданием параметров каждой из областей в этот регистр заносится номер конфигурируемой области.

Таблица 13.3. **Регистр номера области MPU MPU_RNR (0xE000ED98)**

Биты	Обозначение	Тип	Значение после сброса	Описание
7:0	REGION	R/W	—	Задаёт номер программируемой области. Поскольку модуль MPU процессора Cortex-M3 поддерживает всего 8 областей, в этом регистре реализованы только биты [2:0]

Начальный адрес области MPU определяется регистром базового адреса MPU_RBAR (**Табл. 13.4**). Используя поля VALID и REGION данного регистра, мы сможем исключить этап программирования регистра номера области. Это позволит упростить код программы, особенно в том случае, если все параметры MPU хранятся в программе в виде таблицы.

Таблица 13.4. **Регистр базового адреса области MPU MPU_RBAR (0xE000ED9C)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:5	ADDR	R/W	—	Базовый адрес области
4	VALID	R/W	—	Если бит установлен в 1, то номер области MPU, программируемой на данном этапе, будет определяться полем REGION этого регистра. Если бит сброшен в 0, то используется область, заданная регистром MPU_RNR
3:0	REGION	R/W	—	Это поле используется вместо регистра номера области MPU_RNR, если бит VALID установлен в 1; в противном случае, содержимое данного поля игнорируется. Поскольку модуль MPU процессора Cortex-M3 поддерживает всего 8 областей, то это поле также игнорируется, если его значение больше 7

Для задания параметров области предназначен регистр атрибутов и размера области MPU_RASR (**Табл. 13.5**). В частности, размер области MPU определяется содержимым 5-битного поля REGION_SIZE этого регистра (**Табл. 13.6**).

Таблица 13.5. **Регистр атрибутов и размера области MPU MPU_RASR (0xE000EDA0)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:29	Зарезервировано	—	—	—
28	XN	R/W	—	Запрет выборки команд (1 = запрещает выборку команд из данной области памяти; при попытке это сделать генерируется отказ системы управления памятью)
27	Зарезервировано	—	—	—
26:24	AP	R/W	—	Поле прав доступа к данным
23:22	Зарезервировано	—	—	—
21:19	TEX	R/W	—	Поле расширения типа
18	S	R/W	—	Разделяемая
17	C	R/W	—	Кэшируемая
16	B	R/W	—	Буферируемая
15:8	SRD	R/W	—	Запрет подобластей
7:6	Зарезервировано	—	—	—
5:1	SIZE	R/W	—	Размер области MPU
0	ENABLE	R/W	—	Разрешение области

Таблица 13.6. **Значение поля SIZE регистра MPU_RASR и размер области MPU**

Поле SIZE	Размер области
b00000	Зарезервировано
b00001	Зарезервировано
b00010	Зарезервировано
b00011	Зарезервировано
b00100	32 байта
b00101	64 байта
b00110	128 байт
b00111	256 байт
b01000	512 байт
b01001	1 Кбайт
b01010	2 Кбайт
b01011	4 Кбайт
b01100	8 Кбайт
b01101	16 Кбайт
b01110	32 Кбайт
b01111	64 Кбайт
b10000	128 Кбайт
b10001	256 Кбайт
b10010	512 Кбайт
b10011	1 Мбайт
b10100	2 Мбайт
b10101	4 Мбайт
b10110	8 Мбайт
b10111	16 Мбайт
b11000	32 Мбайт
b11001	64 Мбайт
b11010	128 Мбайт
b11011	256 Мбайт
b11100	512 Мбайт
b11101	1 Гбайт
b11110	2 Гбайт
b11111	4 Гбайт

Поле запрета подобластей (биты [15:8] регистра MPU_RASR) используется для разбиения области MPU на восемь подобластей равного размера с последующим разрешением или запрещением каждой из них. Если подобласть запрещена и при этом частично перекрывается другой подобластью, то используются права доступа указанной другой подобласти. Если подобласть запрещена и не перекрывается с другой областью, то обращение к данному участку памяти вызовет исключение MemManage Fault. Использование подобластей возможно только в том случае, если размер области превышает 128 байт. Права доступа к области MPU определя-

ются содержимым битового поля AP (биты [26:24] регистра MPU_RASR) в соответствии с **Табл. 13.7**.

***Таблица 13.7.* Содержимое поля AP и соответствующие права доступа**

Значение AP	Привилегированный доступ	Пользовательский доступ	Описание
000	Доступ запрещён	Доступ запрещён	Доступ запрещён
001	Чтение/запись	Доступ запрещён	Доступ только на привилегированном уровне
010	Чтение/запись	Только чтение	Запись в пользовательской программе вызовет отказ
011	Чтение/запись	Чтение/запись	Полный доступ
100	Непредсказуемо	Непредсказуемо	Непредсказуемо
101	Только чтение	Доступ запрещён	Только чтение на привилегированном уровне
110	Только чтение	Только чтение	Только чтение
111	Только чтение	Только чтение	Только чтение

Поле XN (бит 28) определяет допустимость выборки команд из данной области памяти. Если этот бит установлен в 1, то при исполнении любой команды, выбранной из указанной области, будет сгенерировано исключение MemManage Fault.

Назначение полей TEX, S, B, C (биты [21:16]) более сложное. Несмотря на то что процессор Cortex-M3 не имеет кэш-памяти, он всё же построен по архитектуре ARMv7-M, которая поддерживает внешнюю кэш-память, а также более развитые системы памяти. Поэтому атрибуты доступа к памяти для конкретной области могут задаваться в соответствии с используемой моделью управления памятью.

В архитектурах v6 и v7 система памяти может иметь два уровня кэш-памяти: внутренний и внешний. Эти уровни могут иметь различные политики кэширования. Поскольку сам процессор Cortex-M3 не имеет контроллера кэш-памяти, различные политики кэширования влияют только на буферирование операций записи во внутреннем коммутаторе BusMatrix и, возможно, в контроллере памяти (**Табл. 13.8**). Причём, в большинстве микроконтроллеров можно ограничиться всего несколькими типами памяти с соответствующими атрибутами (**Рис. 13.2**).

***Таблица 13.8.* Атрибуты памяти в архитектуре ARMv7-M**

TEX	C	B	Описание	Совместное использование области
b000	0	0	Строго упорядоченная (выполнение и завершение пересылок производится в программируемом порядке)	Разделяемая
b000	0	1	Разделяемое устройство (операции записи могут буферироваться)	Разделяемая
b000	1	0	Внешний и внутренний кэш со сквозной записью; без размещения записываемых данных	[S]
b000	1	1	Внешний и внутренний кэш со сквозной записью; без размещения записываемых данных	[S]
b001	0	0	Внешний и внутренний кэш отключены	[S]
b001	0	1	Зарезервировано	Зарезервировано

***Таблица 13.8.* Атрибуты памяти в архитектуре ARMv7-M** *(продолжение)*

TEX	C	B	Описание	Совместное использование области
b001	1	0	Зависит от реализации	—
b001	1	1	Внешний и внутренний кэш со сквозной записью; с размещением записываемых данных	[S]
b010	0	0	Неразделяемое устройство	Неразделяемая
b010	0	1	Зарезервировано	Зарезервировано
b010	1	X	Зарезервировано	Зарезервировано
b1BB	A	A	Кэшируемая память; BB = политика внешнего кэша, AA = политика внутреннего кэша	[S]

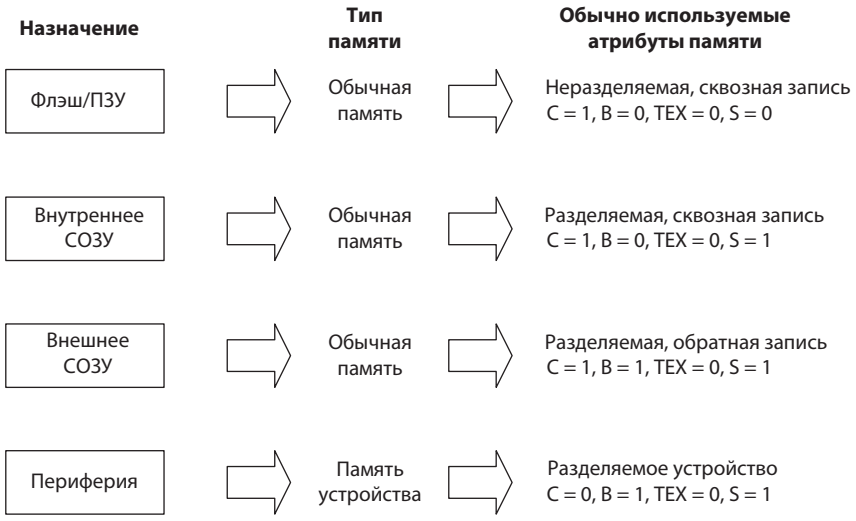


Рис. 13.2. *Наиболее часто используемые в микроконтроллерах атрибуты памяти.*

Если вы используете микроконтроллер с кэш-памятью, то вы должны сконфигурировать модуль MPU в соответствии с желаемой политикой кэширования (например, запрет кэширования/кэширование со сквозной записью/кэширование с обратной записью). Если бит TEX[2] установлен в 1, то политика кэширования для внутренней и внешней кэш-памяти соответствует **Табл. 13.9**.

***Таблица 13.9.* Политика внутреннего и внешнего кэша при TEX[2] = 1**

Биты атрибутов памяти (AA и BB)	Политика кэширования
00	Кэширование отключено
01	Обратная запись; размещение записываемых данных
10	Сквозная запись; нет размещения записываемых данных
11	Обратная запись; нет размещения записываемых данных

Дополнительную информацию о кэш-памяти и политиках кэширования можно почерпнуть из руководства [2].

13.3. Настройка модуля MPU

На первый взгляд, регистры модуля MPU могут показаться сложными для понимания, однако если вы чётко осознаёте, какие именно области памяти нужны для вашего приложения, то использование этих регистров не вызовет никаких затруднений. Как правило, необходимы следующие области:

- область программного кода для привилегированных программ, таких как ядро ОС и обработчики исключений;
- область программного кода для пользовательских программ;
- область данных для привилегированных и пользовательских программ в различных областях памяти (например, данные и стек приложения, расположенные в области статического ОЗУ с адресами от 0x20000000 до 0x30000000);
- прочая периферия.

Выделять отдельную область памяти в диапазоне адресов шины собственных периферийных устройств не обязательно. Модуль MPU автоматически распознаёт соответствующие адреса и разрешает привилегированному коду обращаться к этой области памяти.

В устройствах с ядром Cortex-M3 для большинства областей памяти могут быть заданы следующие атрибуты: TEX = 000, C = 1, B = 1. Системные устройства, такие как контроллер прерываний NVIC, должны располагаться в области со строго упорядоченным доступом, а области для периферии должны иметь атрибуты «разделяемые устройства» (TEX = 000, C = 0, B = 1). Если же вы хотите, чтобы любые отказы шины в области были «точными», то для неё необходимо будет использовать атрибуты строго упорядоченной памяти (TEX = 000, C = 0, B = 0), чтобы запретить буферизацию записи. Однако при этом может снизиться производительность системы.

При использовании CMSIS-совместимых драйверов устройств к регистрам MPU можно обращаться как к элементам структуры (см. **Табл. 13.10**). Алгоритм простейшей процедуры инициализации модуля MPU приведён на **Рис. 13.3**.

Таблица 13.10. **Названия регистров модуля MPU в стандарте CMSIS**

Поле	Регистр модуля MPU	Адрес
MPU->TYPE	Регистр типа MPU	0xE000ED90
MPU->CTRL	Регистр управления MPU	0xE000ED94
MPU->RNR	Регистр номера области MPU	0xE000ED98
MPU->RBAR	Регистр базового адреса области MPU	0xE000ED9C
MPU->RASR	Регистр атрибутов и размера области MPU	0xE000EDA0
MPU->RBAR_A1	Псевдоним 1 регистра базового адреса области MPU	0xE000EDA4
MPU->RBAR_A2	Псевдоним 2 регистра базового адреса области MPU	0xE000EDAC
MPU->RBAR_A3	Псевдоним 3 регистра базового адреса области MPU	0xE000EDB4
MPU->RASR_A1	Псевдоним 1 регистра атрибутов и размера области MPU	0xE000EDA8

***Таблица 13.10.* Названия регистров модуля MPU в стандарте CMSIS** *(продолжение)*

Поле	Регистр модуля MPU	Адрес
MPU->RASR_A2	Псевдоним 2 регистра атрибутов и размера области MPU	0xE000EDB0
MPU->RASR_A3	Псевдоним 3 регистра атрибутов и размера области MPU	0xE000EDB8

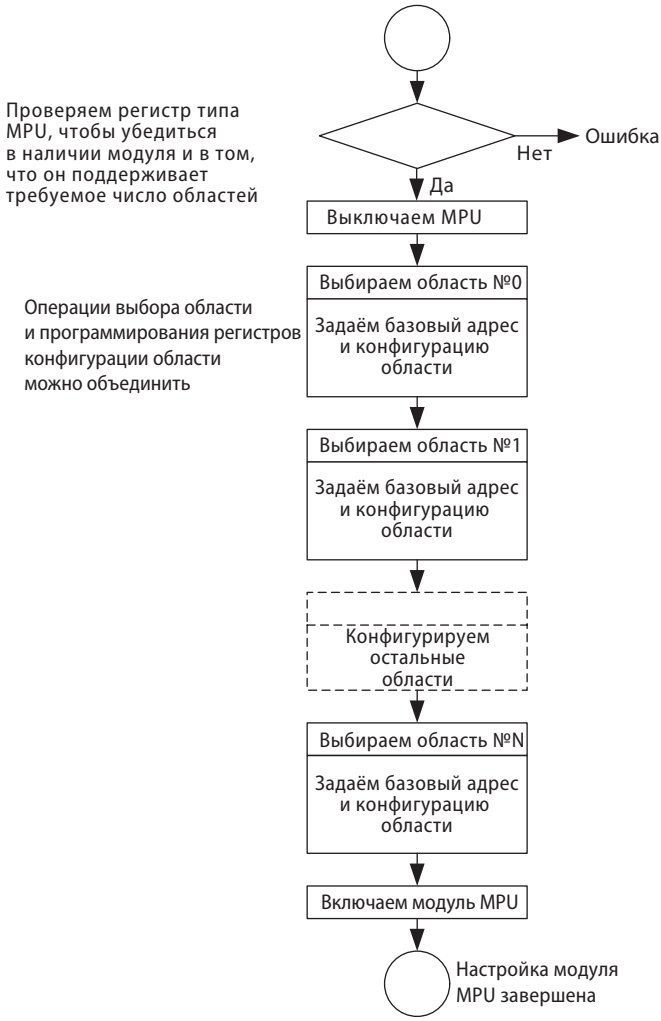


Рис. 13.3. *Пример алгоритма настройки модуля MPU.*

Если таблица векторов перемещена в ОЗУ, то перед включением модуля MPU не забудьте поместить в таблицу адрес обработчика исключения MemManage Fault и разрешить данное исключение, используя регистр SHCSR. Указанные меры необходимы для запуска этого обработчика при нарушении правил, заданных модулем MPU.

Предположим, что нам требуется всего четыре области памяти. В этом случае код инициализации модуля MPU (без функций контроля областей и их разрешения) может выглядеть следующим образом:

```
MPU->RNR  = 0;           // Регистр номера области MPU
                         // Выбираем область №0
MPU->RBAR = 0x00000000;  // Регистр базового адреса области MPU
                         // Базовый адрес = 0x00000000
MPU->RASR = 0x0307002F;  // Регистр атрибутов и размера области
                         // R/W, TEX=0, S=1, C=1, B=1, 16 Мбайт, разрешена
MPU->RNR  = 1;           // Выбираем область №1
MPU->RBAR = 0x20000000;  // Базовый адрес = 0x20000000
MPU->RASR = 0x03070033;  // R/W, TEX=0, S=1, C=1, B=1, 64 Мбайт, разрешена
MPU->RNR  = 2;           // Выбираем область №2
MPU->RBAR = 0x40000000;  // Базовый адрес = 0x40000000
MPU->RASR = 0x03050033;  // R/W, TEX=0, S=1, C=0, B=1, 64 Мбайт, разрешена
MPU->RNR  = 3;           // Выбираем область №3
MPU->RBAR = 0xA0000000;  // Базовый адрес = 0xA0000000
MPU->RASR = 0x01040027;  // Привилегированное R/W, TEX=0, S=1, C=0, B=0,
                         // 1 Мбайт, разрешена
MPU->CTRL = 1;           // Регистр управления MPU – включаем модуль MPU
```

То же на языке ассемблера:

```
LDR  R0,=0xE000ED98  ; Регистр номера области
MOV  R1,#0           ; Выбираем область №0
STR  R1, [R0]
LDR  R1,=0x00000000  ; Базовый адрес = 0x00000000
STR  R1, [R0, #4]    ; Регистр базового адреса области
LDR  R1,=0x0307002F  ; R/W, TEX=0, S=1, C=1, B=1, 16 Мбайт, разрешена
STR  R1, [R0, #8]    ; Регистр атрибутов и размера области
MOV  R1,#1           ; Выбираем область №1
STR  R1, [R0]
LDR  R1,=0x20000000  ; Базовый адрес = 0x20000000
STR  R1, [R0, #4]    ; Регистр базового адреса области
LDR  R1,=0x03070033  ; R/W, TEX=0, S=1, C=1, B=1, 64 Мбайт, разрешена
STR  R1, [R0, #8]    ; Регистр атрибутов и размера области
MOV  R1,#2           ; Выбираем область №2
STR  R1, [R0]
LDR  R1,=0x40000000  ; Базовый адрес = 0x40000000
STR  R1, [R0, #4]    ; Регистр базового адреса области
LDR  R1,=0x03050033  ; R/W, TEX=0, S=1, C=0, B=1, 64 Мбайт, разрешена
STR  R1, [R0, #8]    ; Регистр атрибутов и размера области
MOV  R1,#3           ; Выбираем область №3
STR  R1, [R0]
LDR  R1,=0xA0000000  ; Базовый адрес = 0xA0000000
STR  R1, [R0, #4]    ; Регистр базового адреса области
LDR  R1,=0x01040027  ; Привилегированное R/W, TEX=0, S=1, C=0, B=0,
                     ; 1 Мбайт, разрешена
STR  R1, [R0, #8]    ; Регистр атрибутов и размера области
MOV  R1,#1           ; Включаем модуль MPU
STR  R1, [R0,#-4]    ; Регистр управления MPU (0xE000ED98-4=0xE000ED94)
```

В результате будут определены четыре области MPU:

- *Код* — 0x00000000…0x00FFFFFF (16 Мбайт), полный доступ, кэшируемая память.
- *Данные* — 0x20000000…0x23FFFFFF (64 Мбайт), полный доступ, кэшируемая память.
- *Периферия* — 0x40000000…0x43FFFFFF (64 Мбайт), полный доступ, разделяемый ресурс.
- *Внешние устройства* — 0xA0000000…0xA00FFFFF (1 Мбайт), привилегированный доступ, строго упорядоченная память, неисполняемая.

Приведённый выше код можно сократить, объединив операции выбора области и записи в регистр базового адреса:

```
MPU->RBAR = 0x00000010; // Регистр базового адреса области MPU
                        // Базовый адрес = 0x00000000, корректен, область №0
MPU->RASR = 0x0307002F; // Регистр атрибутов и размера области
                        // R/W, TEX=0, S=1, C=1, B=1, 16 Мбайт, разрешена
MPU->RBAR = 0x20000011; // Базовый адрес = 0x20000000, корректен, область №1
MPU->RASR = 0x03070033; // R/W, TEX=0, S=1, C=1, B=1, 64 Мбайт, разрешена
MPU->RBAR = 0x40000012; // Базовый адрес = 0x40000000, корректен, область №2
MPU->RASR = 0x03050033; // R/W, TEX=0, S=1, C=0, B=1, 64 Мбайт, разрешена
MPU->RBAR = 0xA0000013; // Базовый адрес = 0xA0000000, корректен, область №3
MPU->RASR = 0x01040027; // Привилегированное R/W, TEX=0, S=1, C=0, B=0,
                        // 1 Мбайт, разрешена
MPU->CTRL = 1;          // Регистр управления MPU - включаем модуль MPU
```

Или, на языке ассемблера:

```
LDR  R0,=0xE000ED9C  ; Регистр базового адреса области
LDR  R1,=0x00000010  ; Базовый адрес = 0x00000000, корректен, область №0
STR  R1, [R0, #0]    ; Регистр базового адреса области MPU
LDR  R1,=0x0307002F  ; R/W, TEX=0, S=1, C=1, B=1, 16 Мбайт, разрешена
STR  R1, [R0, #4]    ; Регистр атрибутов и размера области MPU
LDR  R1,=0x20000011  ; Базовый адрес = 0x20000000, корректен, область №1
STR  R1, [R0, #0]    ; Регистр базового адреса области MPU
LDR  R1,=0x03070033  ; R/W, TEX=0, S=1, C=1, B=1, 64 Мбайт, разрешена
STR  R1, [R0, #4]    ; Регистр атрибутов и размера области MPU
LDR  R1,=0x40000012  ; Базовый адрес = 0x40000000, корректен, область №2
STR  R1, [R0, #0]    ; Регистр базового адреса области MPU
LDR  R1,=0x03050033  ; R/W, TEX=0, S=1, C=0, B=1, 64 Мбайт, разрешена
STR  R1, [R0, #4]    ; Регистр атрибутов и размера области MPU
LDR  R1,=0xA0000013  ; Базовый адрес = 0xA0000000, корректен, область №3
STR  R1, [R0, #0]    ; Регистр базового адреса области MPU
LDR  R1,=0x01040027  ; R/W, TEX=0, S=1, C=0, B=0, 1 Мбайт, разрешена
STR  R1, [R0, #4]    ; Регистр атрибутов и размера области MPU
MOV  R1,#1           ; Включаем модуль MPU
STR  R1, [R0,#-8]    ; Регистр управления MPU (0xE000ED9C-8=0xE000ED94)
```

Размер кода несколько уменьшился. Однако нет предела совершенству! Для получения более быстрого и компактного кода можно воспользоваться альтернативными адресами (псевдонимами) регистров MPU (см. **Табл. Г.34** в Приложении Г), которые позволяют обращаться к регистрам MPU_RBAR и MPU_RASR. Эти псевдонимы образуют непрерывную область размером 8 слов, что даёт возможность использовать команды групповой загрузки/сохранения (`LDM` и `STM`):

```
    LDR   R0,=0xE000ED9C       ; Регистр базового адреса области
    LDR   R1,=MPUconfig        ; Таблица предустановленных параметров модуля MPU
    LDMIA R1!, {R2, R3, R4, R5}; Читаем 4 слова из таблицы
    STMIA R0!, {R2, R3, R4, R5}; Пишем 4 слова в MPU
    LDMIA R1!, {R2, R3, R4, R5}; Читаем следующие 4 слова из таблицы
    STMIA R0!, {R2, R3, R4, R5}; Пишем следующие 4 слова в MPU
    B     MPUconfigEnd
    ALIGN 4   ; Эта директива необходима для корректного размещения
              ; последующей таблицы (с выравниванием на границу слова),
MPUconfig     ; чтобы мы могли использовать команду умножения
    DCD   0x00000010     ; Базовый адрес = 0x00000000, корректен, область №0
    DCD   0x0307002F     ; R/W, TEX=0, S=1, C=1, B=1, 16 Мбайт, разрешена
    DCD   0x20000011     ; Базовый адрес = 0x08000000, корректен, область №1
    DCD   0x03070033     ; R/W, TEX=0, S=1, C=1, B=1, 64 Мбайт, разрешена
    DCD   0x40000012     ; Базовый адрес = 0x40000000, корректен, область №2
    DCD   0x03050033     ; R/W, TEX=0, S=1, C=0, B=1, 64 Мбайт, разрешена
    DCD   0xA0000013     ; Базовый адрес = 0xA0000000, корректен, область №3
    DCD   0x01040027     ; R/W, TEX=0, S=1, C=0, B=0, 1 Мбайт, разрешена
MPUconfigEnd
    LDR   R0,=0xE000ED94 ; Регистр управления MPU
    MOV   R1,#1          ; Включаем модуль MPU
    STR   R1, [R0]
```

Разумеется, приведённое выше решение пригодно только в том случае, если вся требуемая информация известна заранее. В противном случае, придётся применять более универсальные методы. Один из таких методов заключается в использовании отдельной подпрограммы (`MpuRegionSetup`), настраивающей область MPU в соответствии с входными параметрами:

```
void MpuRegionSetup(unsigned int addr, unsigned int region,
  unsigned int size, unsigned int ap, unsigned int MemAttrib,
  unsigned int srd, unsigned int XN, unsigned int enable)
{ // Процедура настройки области MPU
  MPU->RBAR = (addr & 0xFFFFFFE0) | (region & 0xF) | 0x10;
  MPU->RASR = ((XN & 0x1)<<28) | ((ap & 0x7)<<24) |
              ((MemAttrib & 0x3F)<<16) | ((srd&0xFF)<<8) |
              ((size & 0x1F)<<1)| (enable & 0x1);
  return;
}
void MpuRegionDisable(unsigned int region)
{ // Процедура запрещения неиспользуемой области MPU
  MPU->RBAR = (region & 0xF) | 0x10;
  MPU->RASR = 0; // Запрещаем
  return;
}
void MpuSetup(void)
{ // Конфигурирование модуля MPU
  MPU->CTRL = 0; // Сначала отключаем MPU
  MpuRegionSetup(0x00000000, 0, 0x17, 3, 7, 0, 0, 1); // Область 0, 16 Мбайт
  MpuRegionSetup(0x20000000, 1, 0x19, 3, 7, 0, 0, 1); // Область 1, 64 Мбайт
  MpuRegionSetup(0x40000000, 2, 0x19, 3, 5, 0, 0, 1); // Область 2, 64 Мбайт
  MpuRegionSetup(0xA0000000, 3, 0x13, 1, 4, 0, 0, 1); // Область 3, 1 Мбайт
```

```
  MpuRegionDisable(4); // Запрещаем неиспользуемую область 4
  MpuRegionDisable(5); // Запрещаем неиспользуемую область 5
  MpuRegionDisable(6); // Запрещаем неиспользуемую область 6
  MpuRegionDisable(7); // Запрещаем неиспользуемую область 7
  MPU->CTRL = 1; // Включаем модуль MPU
  return;
}
```

В примере была использована подпрограмма, запрещающая обращение к неиспользуемым областям. Это необходимо делать в том случае, если вы не знаете, как указанные области были сконфигурированы раньше. Если неиспользуемые области ранее были разрешены, то их необходимо запретить, чтобы они не влияли на новую конфигурацию модуля MPU.

Разумеется, процедуры настройки модуля MPU можно реализовать и на ассемблере:

```
MpuSetup ; Процедура конфигурирования MPU, в которой несколько раз вызывается
         ; процедура настройки области
    PUSH  {R0-R6, LR}
    LDR   R0,=0xE000ED94     ; Регистр управления MPU
    MOV   R1,#0
    STR   R1,[R0]            ; Отключаем модуль MPU
    ; --- Область №0 ---
    LDR   R0,=0x00000000     ; Область 0: базовый адрес = 0x00000000
    MOV   R1,#0x0            ; Область 0: номер области = 0
    MOV   R2,#0x17           ; Область 0: размер = 0x17 (16 Мбайт)
    MOV   R3,#0x3            ; Область 0: AP = 0x3 (полный доступ)
    MOV   R4,#0x7            ; Область 0: атрибуты = 0x7
    MOV   R5,#0x0            ; Область 0: запрет подобластей = 0
    MOV   R6,#0x1            ; Область 0: {XN, Enable} = 0,1
    BL    MpuRegionSetup
    ; --- Область №1 ---
    LDR   R0,=0x20000000     ; Область 1: базовый адрес = 0x20000000
    MOV   R1,#0x1            ; Область 1: номер области = 1
    MOV   R2,#0x19           ; Область 1: размер = 0x19 (64 Мбайт)
    MOV   R3,#0x3            ; Область 1: AP = 0x3 (полный доступ)
    MOV   R4,#0x7            ; Область 1: атрибуты = 0x7
    MOV   R5,#0x0            ; Область 1: запрет подобластей = 0
    MOV   R6,#0x1            ; Область 1: {XN, Enable} = 0,1
    BL    MpuRegionSetup
    ...                      ; Настраиваем области №2 и №3
    ; --- Области №4-№7 запрещены ---
    MOV   R0,#4
    BL    MpuRegionDisable
    MOV   R0,#5
    BL    MpuRegionDisable
    MOV   R0,#6
    BL    MpuRegionDisable
    MOV   R0,#7
    BL    MpuRegionDisable
    LDR   R0,=0xE000ED94     ; Регистр управления MPU
    MOV   R1,#1
```

```
    STR   R1,[R0]            ; Включаем модуль MPU
    POP   {R0-R6, PC}        ; Возвращаемся
MpuRegionSetup
    ; Подпрограмма настройка области MPU
    ; Вход R0 : базовый адрес
    ;      R1 : номер области
    ;      R2 : размер
    ;      R3 : AP (права доступа)
    ;      R4 : атрибуты памяти ({TEX[2:0], S, C, B})
    ;      R5 : запрет подобластей
    ;      R6 : {XN, Enable}
    PUSH  {R0-R1, LR}
    BIC   R0, R0, #0x1F      ; Очищаем неиспользуемые биты адреса
    BFI   R0, R1, #0, #4     ; Помещаем номер области в R0[3:0]
    ORR   R0, R0, #0x10      ; Устанавливаем бит корректности
    LDR   R1,=0xE000ED9C     ; Регистр базового адреса области MPU
    STR   R0,[R1]            ; Задаём базовый адрес
    AND   R0, R6, #0x01      ; Получаем бит Enable
    UBFX  R1, R6, #1, #1     ; Получаем бит XN
    BFI   R0, R1, #28, #1    ; Кладём XN в R0[28]
    BFI   R0, R2, #1 , #5    ; Загружаем размер области (R2[4:0])
                             ; в R0[5:1]
    BFI   R0, R3, #24, #3    ; Загружаем права доступа (R3[2:0]) в R0[26:24]
    BFI   R0, R4, #16, #6    ; Загружаем атрибуты памяти (R4[5:0])
                             ; в R0[21:16]
    BFI   R0, R5, #8, #8     ; Загружаем биты запрета подобластей (SRD)
                             ; в R0[15:8]
    LDR   R1,=0xE000EDA0     ; Регистр размера и атрибутов области MPU
    STR   R0,[R1]            ; Инициализируем регистр
    POP   {R0-R1, PC}        ; Возвращаемся
MpuRegionDisable
    ; Подпрограмма для запрещения неиспользуемой области
    ; Вход R0 : номер области
    PUSH  {R1, LR}
    AND   R0, R0, #0xF       ; Очищаем неиспользуемые биты адреса
    ORR   R0, R0, #0x10      ; Устанавливаем бит корректности
    LDR   R1,=0xE000ED9C     ; Регистр базового адреса области MPU
    STR   R0,[R1]
    MOV   R0, #0
    LDR   R1,=0xE000EDA0     ; Регистр размера и атрибутов области MPU
    STR   R0,[R1]            ; Очищаем регистр (область запрещена)
    POP   {R1, PC}           ; Возвращаемся
```

В данном примере демонстрируется использование команды вставки битового поля (`BFI`). Эта команда позволяет значительно упростить операции объединения битовых полей.

13.4. Типичный процесс настройки модуля MPU

В большинстве приложений модуль MPU используется для того, чтобы предотвратить доступ из пользовательских программ к областям кода и данных при-

вилегированных процессов. Обычно эти функции реализуются встраиваемой ОС. Между переключениями контекста ОС изменяет конфигурацию MPU так, чтобы пользовательские приложения могли обращаться к своему коду и данным, а также к другим требуемым ресурсам. При написании процедуры конфигурирования модуля MPU необходимо предусмотреть следующие области:

1. Область кода:
   а) привилегированный код, включая начальную таблицу векторов;
   б) пользовательский код.
2. Область СОЗУ:
   а) привилегированные данные, включая основной стек;
   б) пользовательские данные, включая стек процесса;
   в) привилегированная область доступа к битам;
   г) пользовательская область доступа к битам.
3. Периферия:
   а) привилегированная периферия;
   б) пользовательская периферия;
   в) привилегированная область доступа к битам в диапазоне адресов периферийных устройств;
   г) пользовательская область доступа к битам в диапазоне адресов периферийных устройств.

В приведённом выше списке мы обозначили 10 областей — это больше, чем поддерживается модулем MPU процессора Cortex-M3. Однако привилегированные области можно задать, воспользовавшись «фоновой» областью (PRIVDEFENA = 1). Тогда нам необходимо будет определить только пять пользовательских областей MPU и ещё три останутся в запасе. При необходимости незадействованные области MPU можно использовать для организации дополнительных областей во внешней памяти для защиты данных, предназначенных только для чтения, или же для полной блокировки определённого участка памяти. Для уменьшения требуемого числа областей MPU некоторые из них можно объединить друг с другом.

13.4.1. Пример использования запрета подобластей

В ряде случаев нам может потребоваться, чтобы какие-то периферийные устройства были доступны для пользовательских программ, а какие-то — только для привилегированного доступа. В результате адресное пространство памяти периферийных устройств, доступных для пользователя, оказывается фрагментировано. Это можно реализовать одним из следующих способов:

- определить несколько пользовательских областей;
- определить привилегированные области внутри области пользовательской периферии;
- запретить подобласти в пользовательской области.

В первых двух случаях очень легко израсходовать все доступные области MPU. Используя же возможности запрета подобластей, мы сможем задать права доступа к отдельным блокам регистров периферийных устройств, не расходуя на это дополнительные области. Применение данного метода показано на **Рис. 13.4**.

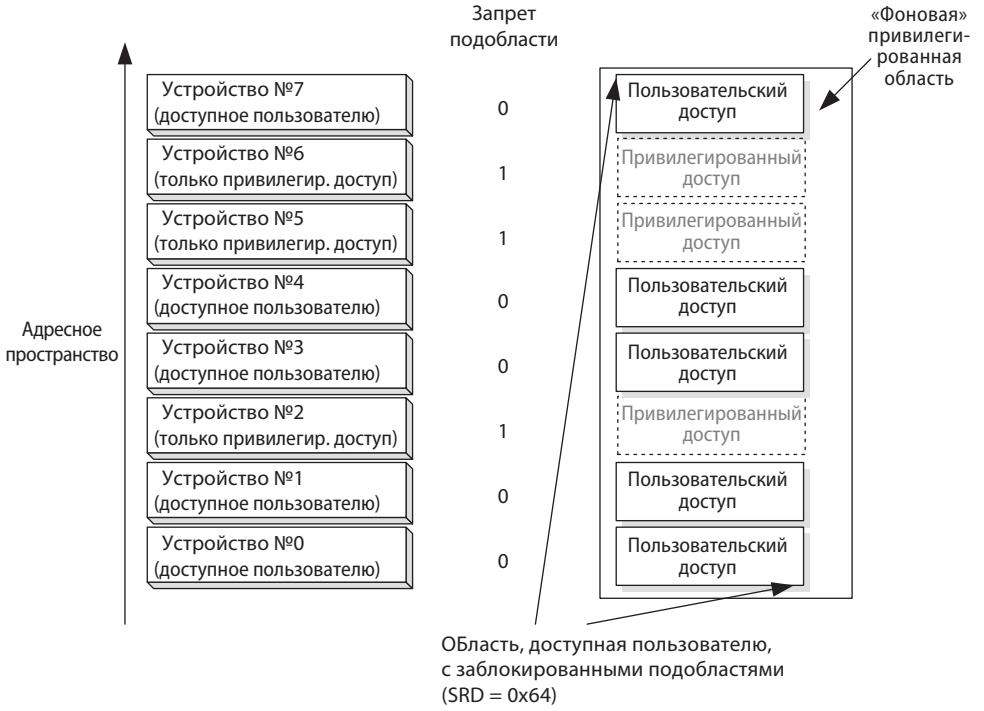


Рис. 13.4. *Использование запрета подобластей для управления правами доступа к разделяемым периферийным устройствам.*

Этот способ работает и с областями памяти. Однако появление блоков, требующих прав доступа, отличных от прав соседних блоков, всё же более вероятно в области периферийных устройств.

Предположим, что в нашей программе используются области памяти, описанные в **Табл. 13.11**.

Таблица 13.11. **Параметры областей памяти из приведённого ниже примера**

Адрес	Описание	Размер	Тип	Атрибуты памяти (C, B, A, S, XN)	Область MPU
0x00000000… 0x00003FFF	Привилегированный код	16 Кбайт	Только чтение	C, –, A, –, –	«Фоновая»
0x00004000… 0x00007FFF	Пользовательский код	16 Кбайт	Только чтение	C, –, A, –, –	Область №0
0x20000000… 0x20000FFF	Пользовательские данные	4 Кбайт	Полный доступ	C, B, A, –, –	Область №1
0x20001000… 0x20001FFF	Привилегированные данные	4 Кбайт	Привилегированный доступ	C, B, A, –, –	«Фоновая»
0x22000000… 0x2201FFFF	Пользовательская область доступа к битам	128 Кбайт	Полный доступ	C, B, A, –, –	Область №2

Таблица 13.11. **Параметры областей памяти из приведённого ниже примера** *(продолжение)*

Адрес	Описание	Размер	Тип	Атрибуты памяти (C, B, A, S, XN)	Область MPU
0x22020000… 0x2203FFFF	Привилегированная область доступа к битам	128 Кбайт	Полный доступ	C, B, A, –, –	«Фоновая»
0x40000000… 0x400FFFFF	Пользовательская периферия	1 Мбайт	Полный доступ	–, B, –, –, XN	Область №3
0x40040000… 0x4005FFFF	Привилегированная периферия в области пользовательских периферийных устройств	128 Кбайт	Привилегированный доступ	–, B, –, –, XN	Закрытая подобласть области №3
0x42000000… 0x43FFFFFF	Область доступа к битам пользовательской периферии	32 Мбайт	Полный доступ	–, B, –, –, XN	Область №4
0x42800000… 0x42BFFFFF	Область доступа к битам привилегированной периферии в пользовательской области памяти	4 Мбайт	Привилегированный доступ	–, B, –, –, XN	Закрытая подобласть области №4
0x60000000… 0x60FFFFFF	Внешнее ОЗУ	16 Мбайт	Полный доступ	C, B, A, –, –	Область №5
0xE0000000… 0xF00FFFFF	Контроллер NVIC, компоненты отладки и шина собственных периферийных устройств	1 Мбайт	Привилегированный доступ	–, –, –, –, XN	«Фоновая»

Примечание. Символ «A» в столбце атрибутов памяти определяет размещение кэша.

После определения требуемых областей мы можем написать процедуру настройки модуля MPU. Для облегчения понимания и последующих изменений кода воспользуемся ранее написанной функцией:

```
void MpuSetup(void)
{ // Настройка модуля MPU
  MPU->CTRL = 0; // Сначала отключаем MPU
  // Параметры: Адрес, Область, Размер, AP, Атрибуты, SRD, XN, Разрешение
  MpuRegionSetup(0x00004000, 0, 0x0D, 3, 0x2, 0, 0, 1); // Область №0
  // 0x00004000-0x00007FFF: пользовательская программа, 16 Кбайт, полный доступ,
  // атрибуты = 0x2 (TEX=0, S=0, C=1, B=0), запрет подобластей = 0, XN=0
  MpuRegionSetup(0x20000000, 1, 0x0B, 3, 0xB, 0, 0, 1); // Область №1
  // 0x20000000-0x20000FFF: пользовательские данные, 4 Кбайт, полный доступ,
  // атрибуты = 0xB (TEX=1, S=0, C=1, B=1), запрет подобластей = 0, XN=0
  MpuRegionSetup(0x22000000, 2, 0x10, 3, 0xB, 0, 0, 1); // Область №2
  // 0x22000000-0x2201FFFF: пользовательская область доступа к битам, 128 Кбайт,
```

```
  // полный доступ, атрибуты = 0xB (TEX=1, S=0, C=1, B=1),
  // запрет подобластей = 0, XN=0
  MpuRegionSetup(0x40000000, 3, 0x13, 3, 0x1,0x64,0,1); // Область №3
  // 0x40000000-0x400FFFFF: пользовательская периферия, 1 Мбайт, полный доступ,
  // атрибуты = 0x1 (TEX=0, S=0, C=0, B=1), запрет подобластей = 0x64, XN=0
  // Примечание: запрет подобластей = 0x64 согласно Рис. 13.4
  MpuRegionSetup(0x42000000, 4, 0x18, 3, 0x1,0x64,0,1); // Область №4
  // 0x42000000-0x43FFFFFF: пользовательская область доступа к битам для
  // периферии, 32 Мбайт, полный доступ, атрибуты = 0x1 (TEX=0, S=0, C=0, B=1),
  // запрет подобластей = 0x64, XN=0
  // Примечание: запрет подобластей = 0x64 согласно Рис. 13.4
  MpuRegionSetup(0x60000000, 5, 0x17, 3, 0x3, 0, 0, 1); // Область №5
  // 0x60000000-0x60FFFFFF: внешнее ОЗУ, 16 Мбайт, полный доступ,
  // атрибуты = 0x3 (TEX=0, S=0, C=1, B=1), запрет подобластей = 0, XN=0
  MpuRegionDisable(6); // Запрещаем неиспользуемую область №6
  MpuRegionDisable(7); // Запрещаем неиспользуемую область №7
  MPU->CTRL = 5;       // Включаем MPU, одновременно разрешая привилегированные
                       // обращения к «фоновой» области

return;
}
```

ГЛАВА 14
ПРОЧИЕ ВОЗМОЖНОСТИ ПРОЦЕССОРА CORTEX-M3

14.1. Системный таймер SYSTICK

Таймер SYSTICK, входящий в состав контроллера прерываний NVIC, вкратце был рассмотрен в Главе 8. Как мы с вами уже знаем, этот таймер представляет собой 24-битный вычитающий счётчик. При достижении счётчиком нулевого значения в него загружается значение перезагрузки из регистра SYST_RVR. При этом счёт прекращается только при сбросе бита разрешения ENABLE регистра управления и состояния SYST_CSR (**Рис. 14.1**).

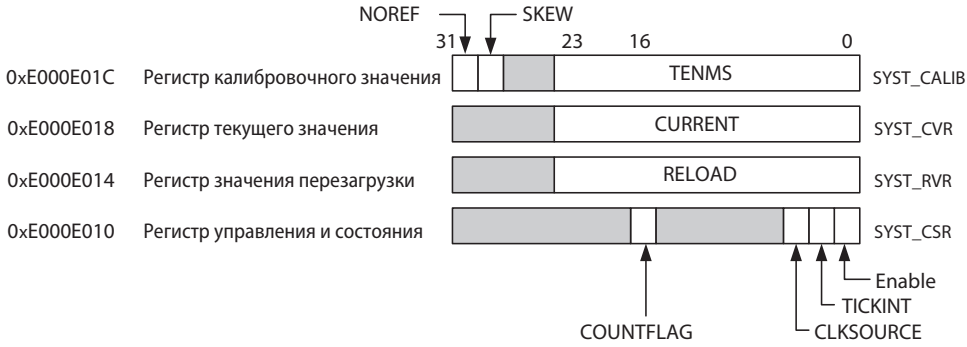


Рис. 14.1. *Регистры таймера SYSTICK в контроллере NVIC.*

В процессоре Cortex-M3 предусмотрено два источника тактового сигнала таймера SYSTICK. Одним из них является внутренний генератор процессора (независимый от системного тактового сигнала HCLK, так что формируемый им сигнал не пропадает при отключении системного тактового сигнала). В качестве другого источника может выступать внешний источник тактового сигнала. Частота этого сигнала должна быть, по меньшей мере, в два раза меньше частоты сигнала внутреннего генератора, поскольку последний применяется для синхронизации внешнего сигнала. Возможность использования внешнего тактового сигнала определяется разработчиком микросхемы и в некоторых моделях может отсутствовать. Чтобы определить наличие такой возможности, необходимо проверить состояние бита 31 регистра калибровочного значения таймера

SYST_CALIB. Разработчик микросхемы должен подать на этот вывод регистра соответствующий уровень.

При изменении значения таймера SYSTICK с 0 на 1 устанавливается флаг COUNTFLAG регистра SYST_CSR. Этот флаг сбрасывается при выполнении одной из следующих операций:

- чтение процессором регистра SYST_CSR;
- очистка счётчика SYSTICK посредством записи любого значения в регистр текущего значения таймера SYST_CVR.

Счётчик таймера SYSTICK может использоваться для периодической генерации соответствующего исключения. Обычно это требуется операционным системам для реализации функций управления задачами и ресурсами. Чтобы разрешить генерацию исключения SYSTICK, необходимо установить бит TICKINT регистра SYST_CSR. Кроме того, если таблица векторов была перемещена в ОЗУ, то в неё необходимо будет поместить адрес обработчика исключения SYSTICK. Например:

```
*((volatile unsigned int *)(SCB->VTOR+(15<<2))) = (unsigned int) SysTick_Handler;
```

На языке ассемблера эту операцию можно реализовать следующим образом:

```
; Настройка обработчика исключения SYSTICK (необходима только в том случае,
; если таблица векторов размещена в ОЗУ)
MOV R0, #0xF             ; Исключение №15
LDR R1, =SysTick_handler ; Адрес обработчика исключения
LDR R2, =0xE000ED08      ; Регистр смещения таблицы векторов
LDR R2, [R2]
STR R1, [R2, R0, LSL #2] ; Пишем вектор по адресу VectTblOffset+ExcpType*4
```

При наличии CMSIS-совместимого драйвера устройства для конфигурирования таймера SYSTICK можно использовать функцию `SysTickConfig`. Подробная информация об этой функции приведена в Приложении Ж. Также к регистрам таймера можно обращаться напрямую как к полям структуры:

- `SysTick->CTRL` (регистр управления и состояния);
- `SysTick->LOAD` (регистр значения перезагрузки);
- `SysTick->VAL` (регистр текущего значения);
- `SysTick->CALIB` (регистр калибровочного значения).

Например, чтобы настроить таймер для генерации исключения SYSTICK каждые 1024 такта процессора, можно использовать следующий код:

```
SysTick->LOAD = 1023; // Считать от 1023 до 0
SysTick->VAL = 0;     // Сбрасываем текущее значение в 0
SysTick->CTRL = 0x7;  // Разрешаем работу таймера и генерацию исключения SYSTICK
                      // Для тактирования используется тактовый сигнал процессора
```

Аналогичный код, написанный на ассемблере, будет выглядеть следующим образом:

```
; Разрешаем работу таймера SYSTICK и генерацию соответствующего исключения
LDR R0, =0xE000E010 ; Регистр управления и состояния SYSTICK
MOV R1, #0
STR R1, [R0]        ; Останавливаем счётчик во избежание случайной
                    ; генерации исключения
LDR R1, =1023       ; Генерируем исключение каждые 1024 такта (для получения
```

```
                    ; такой периодичности счётчик декрементируется с 1023 до 0,
                    ; поэтому значение перезагрузки равно 1023)
STR R1, [R0,#4]     ; Загружаем значение перезагрузки в соответствующий регистр
STR R1, [R0,#8]     ; Загружаем любое значение в регистр текущего значения
                    ; для очистки последнего и сброса бита COUNTFLAG
MOV R1, #0x7        ; Тактирование – системный тактовый сигнал, прерывание -
                    ; разрешено, работа таймера - разрешена
STR R1, [R0]        ; Запускаем счётчик
```

В таймере SYSTICK предусмотрен удобный доступ к калибровочной информации. Процессор Cortex-M3 имеет 24-битный вход, на который разработчик микросхемы может выставить значение, соответствующее 10-мс интервалу. Чтобы получить это значение, необходимо прочитать регистр калибровочного значения SYST_CALIB. Однако функция калибровки может быть не реализована — чтобы убедиться в возможности её использования, обратитесь к документации на используемое устройство.

Счётчик SYSTICK также можно задействовать в качестве сигнального таймера, запускающего некоторую задачу по истечении заданного времени. Например, если нам необходимо запустить задачу через 300 тактов, то мы можем описать её в виде обработчика исключения SYSTICK и запрограммировать таймер следующим образом:

```
volatile int SysTickFired;  // Глобальный программный флаг, индицирующий
                            // запуск процедуры SysTickAlarm
...
// Опционально: инициализируем обработчик SYSTICK, это необходимо только
// при размещении таблицы векторов в ОЗУ
*((volatile unsigned int *)(SCB->VTOR+(15<<2))) = (unsigned int) SysTickAlarm;
SysTick->CTRL = 0x0;        // Запрещаем SysTick
SysTick->LOAD = (300-12);   // Задаём значение перезагрузки минус 12
                            // (учитываем задержку обработки исключения)
SysTick->VAL = 0;           // Сбрасываем текущее значение счётчика
SysTickFired = 0;           // Сбрасываем программный флаг
SysTick->CTRL = 0x7;        // Разрешаем работу таймера, исключение SYSTICK
                            // и тактирование от системного тактового сигнала
while (SysTickFired == 0); // Ждём установки флага обработчиком SYSTICK
```

Обработчик исключения может быть таким:

```
void SysTickAlarm(void) // Обработчик исключения SYSTICK
{
  SysTick->CTRL = 0x0;                  // Запрещаем SysTick
                                        // Выполняем требуемые операции
  SCB->ICSR = SCB->ICSR & (0xFDFFFFFF); // Очищаем бит отложенного прерывания
                                        // для исключения SYSTICK, если оно было
                                        // отложено повторно
  SysTickFired++;                       // Изменяем программный флаг, сообщая
                                        // основной программе о выполнении
                                        // обработчика SYSTICK
  return;
}
```

Таймер запускается с нулевым значением счётчика, поскольку тот сбрасывается вручную в основной программе. Сразу же после этого в счётчик перегружается число 288 (300 – 12). Мы уменьшаем требуемое число тактов на 12 для того, чтобы учесть время задержки обработки прерывания. Однако если на момент достижения счётчиком нулевого значения будет обрабатываться другое исключение с таким же или более высоким приоритетом, то генерация исключения SYSTICK может немного задержаться.

Заметьте, что корректировать значение перезагрузки необходимо только при использовании таймера SYSTICK в качестве таймера с однократным запуском. В периодическом режиме значение перезагрузки должно быть равно числу тактов, уменьшенному на единицу.

Поскольку счётчик SYSTICK не останавливается автоматически, мы должны останавливать его в обработчике исключения (процедура `SysTickAlarm`). Более того, существует вероятность повторного откладывания исключения SYSTICK при задержке его обработки из-за более приоритетных исключений. Поэтому если исключение SYSTICK используется для однократного выполнения некоей задачи, то в обработчике также необходимо сбрасывать признак отложенности исключения.

В самом конце обработчика исключения SYSTICK мы изменяем значение переменной-флага `SysTickFired`, чтобы проинформировать основную программу о выполнении требуемой задачи.

14.2. Управление электропитанием

14.2.1. Спящие режимы

Одним из инструментов управления электропитанием в процессоре Cortex-M3 является наличие спящих режимов. При переводе системы в спящий режим системный тактовый сигнал может останавливаться, однако независимый тактовый генератор должен продолжать работать, чтобы обеспечить вывод процессора из спящего режима при возникновении прерывания. Поддерживаются следующие спящие режимы:

- Sleep — в этом режиме активируется сигнал SLEEPING процессора;
- Deep Sleep — в этом режиме активируется сигнал SLEEPDEEP процессора.

Для выбора конкретного режима пониженного энергопотребления в регистре управления SCR контроллера NVIC предусмотрено битовое поле SLEEPDEEP (**Табл. 14.1**). Использование сигналов SLEEPING и SLEEPDEEP зависит от конкретной реализации микроконтроллера. Возможны даже такие реализации, в которых активация обоих сигналов будет приводить к одним и тем же результатам.

Перевод в спящий режим осуществляется командами `WFI` (ожидание прерывания в спящем режиме) или `WFE` (ожидание события в спящем режиме). Таким событием может стать прерывание, ранее возникшее прерывание или же сигнал внешнего события, поданный в виде импульса на вход RXEV процессора. В самом процессоре предусмотрено «защёлкивание» событий, что позволяет уже прошедшему событию выводить процессор из спящего режима (**Рис. 14.2**).

Таблица 14.1. **Регистр управления системой SCR (0xE000ED10)**

Биты	Обозначение	Тип	Значение после сброса	Описание
4	SEVONPEND	R/W	0	Генерация события при откладывании прерывания; вывод из спящего режима, вызванного командой `WFE`, производится при откладывании нового прерывания, даже если уровень приоритета этого прерывания выше текущего уровня
3	Зарезервировано	—	—	—
2	SLEEPDEEP	R/W	0	Разрешает формирование сигнала SLEEPDEEP при переходе в спящий режим
1	SLEEPONEXIT	R/W	0	Включение функции Sleep-On-Exit
0	Зарезервировано	—	—	—

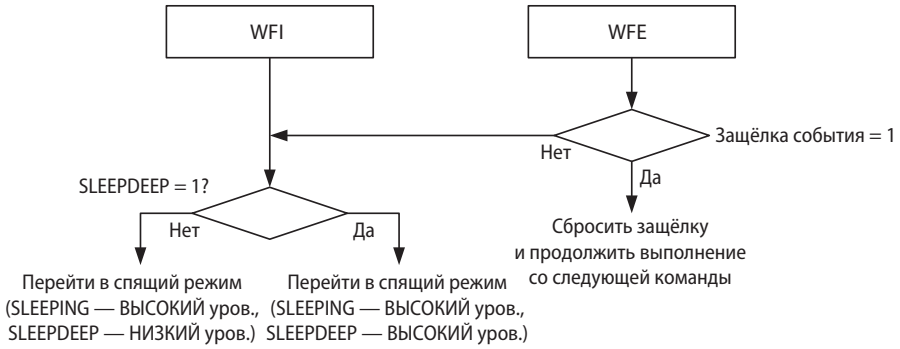


Рис. 14.2. *Переключение в режим пониженного энергопотребления.*

При наличии CMSIS-совместимого драйвера для вызова команд `WFI` и `WFE` можно задействовать встроенные функции `__WFI()` и `__WFE()` соответственно. Для обращения к регистру управления системой можно использовать выражение `SCB->SCR`.

Что в действительности происходит при переходе процессора в спящий режим, зависит исключительно от реализации микросхемы. Как правило, при этом прекращается генерация некоторых тактовых сигналов, что приводит к уменьшению потребления. По желанию разработчика микросхемы для ещё большего сокращения потребления может также отключаться часть узлов микросхемы. Возможен даже вариант, при котором отключается весь процессор и прекращается генерация всех тактовых сигналов. Единственным способом вывода системы из спящего режима в последнем случае является сброс устройства.

Чтобы прерывание могло вывести процессор из спящего режима, вызванного командой `WFI`, оно должно иметь приоритет, который выше текущего уровня приоритета (при наличии активного прерывания) и выше заданного регистром BASEPRI или регистрами маскирования (PRIMASK и FAULTMASK). Если прерывание не может быть обработано из-за низкого приоритета, то оно не сможет вывести процессор из спящего режима.

Со спящим режимом, вызванным командой WFE, ситуация несколько другая. Если прерывание, возникшее во время спящего режима, имеет приоритет ниже или равный заданному в регистрах маскирования или BASEPRI и при этом установлен бит SEVONPEND, то такое прерывание всё же выведет процессор из спящего режима. Условия вывода процессора Cortex-M3 из режимов пониженного энергопотребления приведены в **Табл. 14.2**.

***Таблица 14.2.* Вывод из режимов пониженного потребления**

Поведение команды	Вывод из спящего режима	Запуск обработчика IRQ
WFI		
IRQ с BASEPRI		
Приоритет IRQ > BASEPRIv	Да	Да
Приоритет IRQ =< BASEPRI	Нет	Нет
IRQ с BASEPRI и PRIMASK		
Приоритет IRQ > BASEPRI	Да	Нет
Приоритет IRQ =< BASEPRI	Нет	Нет
WFE		
IRQ с BASEPRI, SEVONPEND = 0		
Приоритет IRQ > BASEPRI	Да	Да
Приоритет IRQ =< BASEPRI	Нет	Нет
IRQ с BASEPRI, SEVONPEND = 1		
Приоритет IRQ > BASEPRI	Да	Да
Приоритет IRQ =< BASEPRI	Да	Нет
IRQ с BASEPRI и PRIMASK, SEVONPEND = 0		
Приоритет IRQ > BASEPRI	Нет	Нет
Приоритет IRQ =< BASEPRI	Нет	Нет
IRQ с BASEPRI и PRIMASK, SEVONPEND = 1		
Приоритет IRQ > BASEPRI	Да	Нет
Приоритет IRQ =< BASEPRI	Да	Нет

14.2.2. Функция Sleep-On-Exit

Ещё одной особенностью процессора Cortex-M3 является возможность автоматического возврата в спящий режим при выходе из процедуры обработки прерывания. Это позволяет нам постоянно удерживать ядро в спящем режиме, за исключением моментов обработки требуемого прерывания. Для использования указанной функции необходимо установить бит SLEEPONEXIT регистра SCR (**Рис. 14.3**).

Обратите внимание, что если функция Sleep-On-Exit включена, то процессор может перейти в спящий режим при возвращении в режим потока из любого исключения, даже если команды WFE/WFI не выполнялись. Чтобы процессор переходил в спящий режим строго в требуемые моменты времени, устанавливайте бит SLEEPONEXIT непосредственно перед переводом системы в спящий режим.

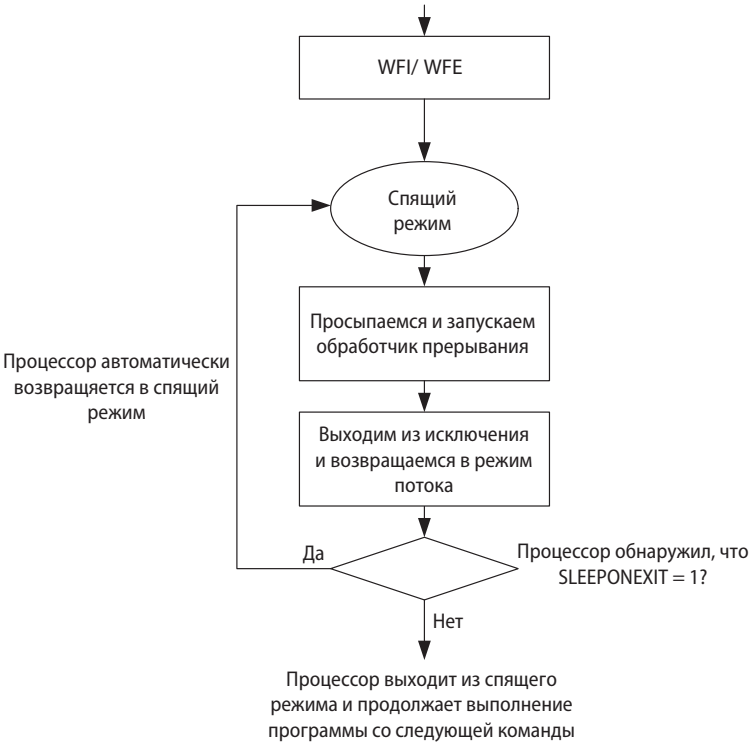


Рис. 14.3. *Пример использования функции Sleep-On-Exit.*

14.2.3. Контроллер WIC

Начиная со 2-й ревизии в процессоре Cortex-M3 появился новый опциональный компонент, позволяющий уменьшить энергопотребления системы. Этот компонент называется *контроллером вывода процессора из спящего режима по прерыванию* или, сокращённо, контроллером WIC (Wakeup Interrupt Controller). Данный контроллер подключается к существующему контроллеру NVIC и используется для формирования запроса на пробуждение процессора при возникновении прерывания.

С точки зрения программы, поведение команд `WFE` и `WFI` не изменяется. В контроллере WIC нет регистров, требующих программирования, поскольку всю необходимую информацию он получает от контроллера NVIC. Наличие контроллера WIC позволяет отключить все тактовые сигналы, поступающие на процессор. При появлении запроса прерывания контроллер WIC может послать запрос на пробуждение системному контроллеру или модулю управления электропитанием PMU (Power Management Unit), расположенным на кристалле, чтобы возобновить генерацию тактовых сигналов процессора (**Рис. 14.4**).

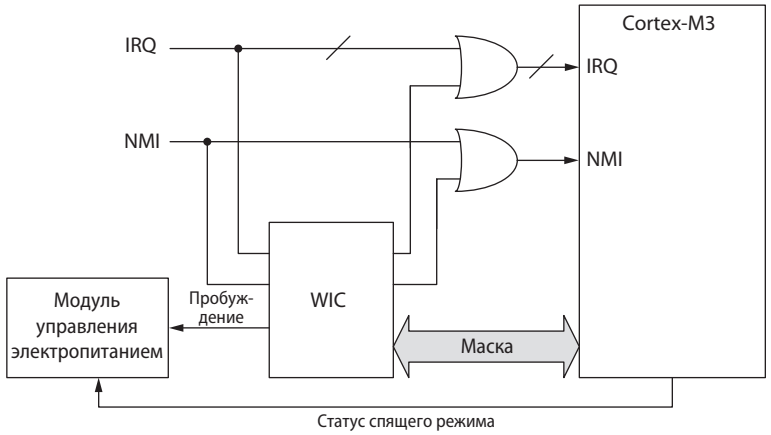


Рис. 14.4. *Обнаружение прерываний контроллером WIC при выключенных тактовых сигналах процессора.*

Наличие контроллера WIC также предоставляет новые возможности в части уменьшения энергопотребления устройства при нахождении в спящем режиме. Использование новейших технологий проектирования цифровых устройств позволяет нам отключать большинство узлов процессора Cortex-M3, оставляя включёнными только некоторую группу логических вентилей, хранящих текущее состояние логики. Данная технология называется State Retention Power Gating (SRPG). При совместном использовании технологии SRPG и контроллера WIC большинство узлов процессора в режиме Deep Sleep может быть отключено (**Рис. 14.5**). При нахождении процессора в этом режиме контроллер WIC продолжает работать и при возникновении прерывания генерирует запрос на включение системы и восстановление её состояния. Максимальная задержка обработки прерывания в данном случае зависит от времени, необходимого для перевода системы в рабочее состояние. В большинстве случаев эта задержка составляет от 20 до 30 тактов. В режиме обычного сна (бит SLEEPDEEP регистра SCR сброшен в 0) питание процессора не отключается.

Новая функция отключения питания является опцией и может отсутствовать в некоторых микроконтроллерах. Она требует наличия в кристалле модуля PMU, который разрабатывается изготовителем микросхемы. Этот модуль управляет циклами включения/выключения питания и может потребовать дополнительного программирования перед использованием. Для получения дополнительной информации следует обратиться к документации на микроконтроллер.

Обратите внимание, что при использовании функции отключения питания системный таймер SYSTICK останавливается на время нахождения процессора в режиме Deep Sleep. Кроме того, данную функцию нельзя использовать при подключённом отладчике, поскольку тот должен периодически обращаться к регистрам отладки для контроля состояния процессора.

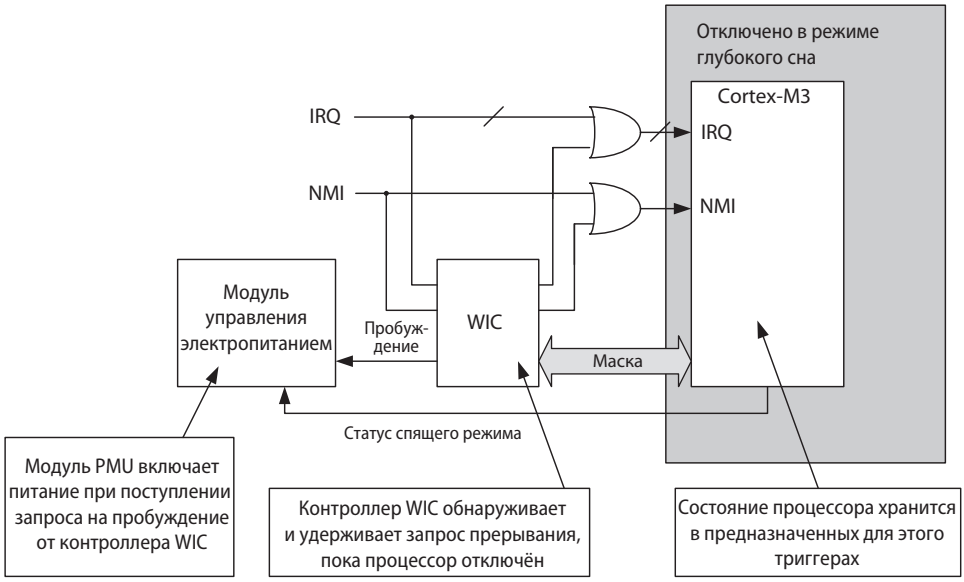


Рис. 14.5. *Обнаружение прерываний контроллером WIC при использовании процессором режима SRPG.*

14.3. Межпроцессорный обмен

В процессоре Cortex-M3 реализован очень простой интерфейс межпроцессорного обмена для передачи событий между процессорами. В процессоре имеется выходной сигнал для передачи событий (TXEV) и один входной сигнал для приёма событий (RXEV). В двухпроцессорных системах соответствующие выводы процессоров могут быть соединены так, как показано на **Рис. 14.6**.

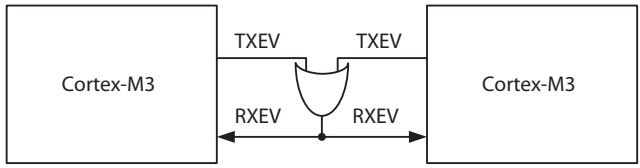


Рис. 14.6. *Обмен событиями в двухпроцессорной системе.*

Как уже было отмечено в предыдущем разделе 14.2 «Управление электропитанием», процессор может перейти в спящий режим при выполнении команды `WFE` и вернуться в активное состояние при получении внешнего события. С помощью команды `SEV` один из процессоров может вывести из спящего режима другой, при этом оба процессора начнут выполнять задачу одновременно (**Рис. 14.7**).

Пользователи CMSIS-совместимого драйвера устройства для вызова команды `SEV` могут задействовать встроенную функцию `__SEV()`. Используя эту функцию, мы можем запустить задачу на выполнение одновременно на обоих процессорах (возможно, с небольшим временным смещением, зависящим от реализации микросхемы и кода, осуществляющего контроль состояния задачи). Число

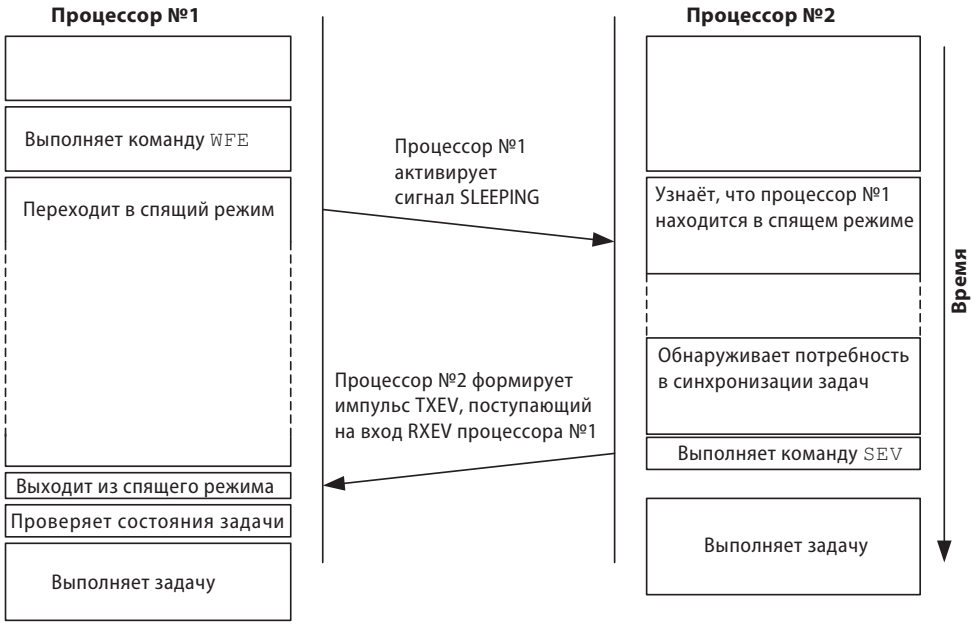


Рис. 14.7. *Использование сигналов событий для синхронизации задач.*

процессоров может быть любым, необходимо только, чтобы один из процессоров выступал в роли ведущего, формируя импульс события для остальных.

Необходимо отметить, что причиной выхода процессора из спящего режима могут стать и другие события, например прерывания или события отладки. Поэтому, прежде чем приступать к выполнению требуемой задачи, необходимо убедиться, что пробуждение процессора произошло в результате синхронизации задач. В большинстве многозадачных систем для корректной синхронизации задач требуется более сложная система межпроцессорных сообщений, наподобие очереди сообщений (mailbox), используемой во встраиваемых ОС.

Также необходимо отметить, что выполнение команды `WFE` не всегда приводит к переводу процессора в спящий режим. Поэтому данная команда обычно помещается внутрь цикла (для уменьшения потребления системы) и сопровождается кодом, который проверяет необходимость выполнения требуемой задачи, как показано на **Рис. 14.8**.

При выполнении команды `WFE` сначала проверяется состояние локальной защёлки события. Если защёлка не установлена, то процессор переходит в спящий режим. Если же защёлка установлена, то она сбрасывается, и процессор продолжает выполнение команд, не переходя в спящий режим. Локальная защёлка события может быть установлена предыдущими исключениями, а также командой `SEV`. Поэтому если вы выполните команду `WFE` сразу после команды `SEV`, то процессор не перейдёт в спящий режим, а просто продолжит выполнение программы со следующей команды (при этом защёлка будет сброшена командой `WFE`).

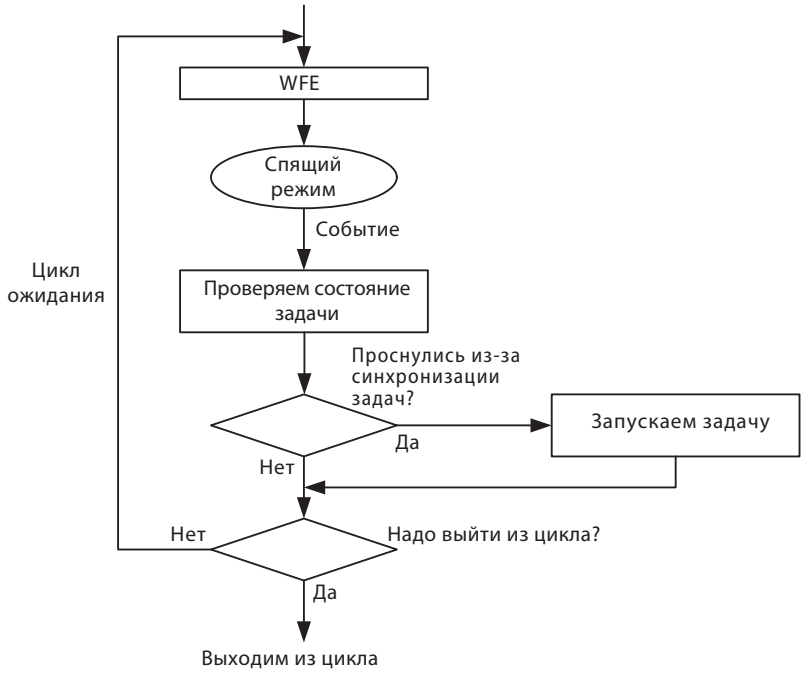


Рис. 14.8. *Пример использования команды* `WFE`.

Команду `WFE` можно применять для реализации семафора в многопроцессорной системе. В стандартных ситуациях, скажем при реализации семафора взаимного исключения (мьютекса), для управления доступом к разделяемой памяти или периферии обычно используют системный монитор монопольного доступа совместно с командами монопольного доступа. При этом процесс, нуждающийся в определённом ресурсе, должен вызвать специальную функцию для захвата «блокировки»:

```
void get_lock(volatile int * Lock_Variable)
{ // __LDREXW и __STREXW - встроенные функции CMSIS
  // CMSIS-совместимой библиотеки драйвера устройства
  int status = 0;
  do {
    while ( __LDREXW(&Lock_Variable) != 0); // Ждём снятия блокировки
    status = __STREXW(1, &Lock_Variable);   // Пытаемся установить флаг
                                            // блокировки, используя
                                            // команду STREX
  } while (status != 0);                    // Повторяем до победного конца
  __DMB();                                  // Барьер памяти данных
  return;
}
```

Та же процедура на ассемблере:

```
get_lock  ; Ассемблерная функция для установки флага блокировки
  LDR   r0, =Lock_Variable
  MOVS  r2, #1             ;
get_lock_loop
```

```
  LDREX r1, [r0]
  CMP   r1, #0
  BNE   get_lock_loop      ; Ресурс заблокирован, проверяем снова
  STREX r1, r2, [r0]       ; Пытаемся установить флаг Lock_Variable,
                           ; используя STREX
  CMP   r1, #0             ; Проверяем статус выполнения STREX
  BNE   get_lock_loop      ; Неудача, повторяем попытку
  DMB                      ; Барьер памяти данных
  BX    LR                 ; Возвращаемся
```

С другой стороны, процесс, использующий ресурс, должен разблокировать его, если тот больше не требуется:

```
void free_lock(volatile int * Lock_Variable)
{
  __DMB();            // Барьер памяти данных
  Lock_Variable = 0;  // Разблокируем
  return;
}
```

Эту же процедуру можно написать и на ассемблере:

```
free_lock ; Ассемблерная процедура для разблокирования ресурса
  LDR  r0, =Lock_Variable
  MOVS r1, #0
  DMB                     ; Барьер памяти данных
  STR  r1, [r0]           ; Сбрасываем флаг блокировки
  BX   LR                 ; Возвращаемся
```

Подобная взаимоблокировка ресурса может привести к излишнему потреблению во время простоя процессора. Использование команды `WFE` позволит нам уменьшить потребление процессора за счёт вывода его из спящего режима только при освобождении требуемого ресурса.

```
void get_lock_with_WFE(volatile int * Lock_Variable)
{
  int status = 0;
  do {
    while ( __LDREXW(&Lock_Variable) != 0){  // Проверяем состояние флага
                                             // блокировки
      __WFE();}                              // Если ресурс заблокирован, то
                                             // «засыпаем» до появления события
    status = __STREXW(1, &Lock_Variable);    // Пытаемся установить флаг
                                             // блокировки,
                                             // используя команду STREX
  } while (status != 0);                     // Повторяем до победного конца
  __DMB();                                   // Барьер памяти данных
  return;
}
```

В функции снятия блокировки используется команда `SEV`, выводящая из спящего режима другие процессоры, ожидающие освобождения ресурса.

```
void free_lock(volatile int * Lock_Variable)
{
  __DMB();           // Барьер памяти данных
  Lock_Variable = 0; // Снимаем блокировку
```

```
  __DSB();           // Обеспечиваем завершение операции записи
  __SEV();           // Генерируем событие для пробуждения остальных процессоров
  return;
}
```

Эти же процедуры могут быть написаны на ассемблере:

```
get_lock_with_WFE ; Ассемблерная функция для установки флага блокировки
  LDR   r0, =Lock_Variable
  MOVS  r2, #1          ;
get_lock_loop
  LDREX r1,[r0]
  CBNZ  r1, lock_is_set ; Если ресурс заблокирован - засыпаем
  STREX r1, r2, [r0]    ; Пытаемся установить флаг Lock_Variable, используя STREX
  CMP   r1, #0          ; Проверяем статус выполнения STREX
  BNE   get_lock_loop   ; Неудача, повторяем попытку
  DMB                   ; Барьер памяти данных
  BX    LR              ; Возвращаемся
lock_is_set
  WFE                   ; Ждём события
  B     get_lock_loop   ; Просыпаемся, повторяем попытку
free_lock_with_SEV ; Ассемблерная функция для сброса флага блокировки
  LDR  r0, =Lock_Variable
  MOVS r1, #0
  DMB                    ; Барьер памяти данных
  STR  r1, [r0]          ; Сбрасываем флаг блокировки
  DSB                    ; Обеспечиваем завершение операции записи
  SEV                    ; Генерируем событие для пробуждения остальных
                         ; процессоров
  BX   LR                ; Возвращаемся
```

Использование механизма передачи событий совместно с кодом реализации семафора позволяет уменьшить потребление процессора при взаимоблокировке ресурса. Аналогичные решения могут применяться для передачи сообщений и синхронизации задач.

В большинстве устройств с ядром Cortex-M3 используется всего один процессор, вход RXEV которого либо подтягивается к НИЗКОМУ уровню, либо подключается к периферийным устройствам, способным генерировать события.

14.4. Управление сбросом

В процессоре Cortex-M3 реализовано два механизма сброса системы. Первый из них основан на использовании бита SYSRESETREQ регистра AIRCR контроллера NVIC. При установке упомянутого бита процессор Cortex-M3 формирует запрос сброса для генератора сброса системы. Поскольку данный блок не входит в состав процессора Cortex-M3, конкретная реализация этого механизма зависит от разработчика микроконтроллера. Поэтому необходимо внимательно изучить документацию на применяемое устройство, чтобы определить, какие именно узлы микроконтроллера сбрасываются при установке указанного бита.

При наличии CMSIS-совместимого драйвера для активации рассмотренного механизма сброса системы можно использовать команду `NVIC _ SystemReset()`.

Описание данной функции приведено в Приложении Ж. Те же, кто не хочет использовать стандарт CMSIS, могут поступить следующим образом:

```
*((volatile unsigned int *)(0xE000ED0C)) = 0x05FA0004;
// Устанавливаем бит SYSRESETREQ (05FA - ключ доступа)
while(1); // Бесконечный цикл, гарантирующий отсутствие выполняемых команд
          // после запуска цикла сброса
```

Программисты, пишущие на ассемблере, для формирования запроса на сброс системы могут использовать следующий код:

```
  LDR R0,=0xE000ED0C ; Адрес регистра AIRCR контроллера NVIC
  LDR R1,=0x05FA0004 ; Устанавливаем бит SYSRESETREQ (05FA - ключ доступа)
  STR R1,[R0]
deadloop
  B   deadloop       ; Бесконечный цикл, гарантирующий отсутствие выполняемых
                     ; команд после запуска цикла сброса
```

Второй механизм управления сбросом основан на использовании бита VECTRESET того же регистра AIRCR (бит [0]). Установка этого бита в 1 вызывает сброс процессора, за исключением компонентов отладки. Сброс узлов, внешних по отношению к процессору, не производится. Так, если в SoC имеется модуль UART, то при установке бита VECTRESET этот модуль, равно как и остальные внешние периферийные устройства, не будет сброшен. Данный механизм предназначен, главным образом, для целей отладки, однако может использоваться и в том случае, если программе необходимо сбросить только процессор, не затрагивая остальные элементы системы.

На языке Си сброс процессора можно реализовать следующим образом:

```
*((volatile unsigned int *)(0xE000ED0C)) = 0x05FA0001;
// Устанавливаем бит VECTRESET (05FA - ключ доступа)
while(1); // Бесконечный цикл, гарантирующий отсутствие выполняемых команд
          // после запуска цикла сброса
```

Эту же операцию можно реализовать и на ассемблере:

```
  LDR R0,=0xE000ED0C ; Адрес регистра AIRCR контроллера NVIC
  LDR R1,=0x05FA0001 ; Устанавливаем бит VECTRESET (05FA - ключ доступа)
  STR R1,[R0]
deadloop
  B   deadloop       ; Бесконечный цикл, гарантирующий отсутствие выполняемых
                     ; команд после запуска цикла сброса
```

В общем случае, для выполнения программного сброса следует использовать бит SYSRESETREQ, а не VECTRESET, что позволит одновременно сбросить большинство компонентов системы. В зависимости от реализации микросхемы при этом могут сбрасываться все или только некоторые периферийные устройства, ядро и схема тактирования, включая блок ФАПЧ (Phase-Locked Loop — PLL). Более подробную информацию можно получить из документации на микросхему.

Следует обратить внимание на то, что между установкой бита SYSRESETREQ и реальным сбросом системы проходит некоторое время, в течение которого процессор сохраняет возможность реакции на прерывания. Если для вас это нежелательно, то перед генерацией запроса на сброс необходимо установить регистр маскирования прерываний (PRIMASK или FAULTMASK).

ГЛАВА 15
АРХИТЕКТУРА СИСТЕМЫ ОТЛАДКИ

15.1. Общие сведения о возможностях отладки

Процессор Cortex-M3 предоставляет пользователю непревзойденные возможности для отладки кода. По своему характеру все функции отладки можно разделить на две группы:

1. Инвазивные (с вмешательством в процесс выполнения программы):
   - останов и пошаговое выполнение программы;
   - аппаратные точки останова;
   - команда точки останова;
   - точки наблюдения при обращении к конкретному адресу, диапазону адресов или значению;
   - доступ к содержимому регистров (как на чтение, так и на запись);
   - исключение Debug monitor;
   - ПЗУ-отладка с использованием функции Flash Patch.
2. Неинвазивные:
   - обращения к памяти (содержимое памяти доступно даже во время работы ядра);
   - трассировка команд (посредством опционального модуля ETM);
   - трассировка данных;
   - программная трассировка (посредством модуля ITM);
   - профилирование (посредством модуля DWT).

Ряд компонентов отладки встроены непосредственно в процессор Cortex-M3. Система отладки базируется на архитектуре отладки CoreSight, которая предоставляет унифицированные решения для управления процессом отладки, сбора трассировочной информации и определения конфигурации системы отладки.

15.2. Обзор архитектуры CoreSight

Архитектура отладки CoreSight охватывает самые разные аспекты отладки, включая протокол интерфейса отладки, протокол шины отладки, управление компонентами отладки, средства обеспечения безопасности, интерфейс трассировки данных и т.п. Для ознакомления с данной архитектурой рекомендую обра-

титься к руководству [3]. Компоненты отладки, входящие в состав процессора Cortex-M3, также описаны в соответствующих разделах руководства [1]. В подавляющем большинстве случаев эти компоненты используются исключительно отладочными средствами, но никак не прикладной программой. Однако для лучшего понимания работы системы отладки будет нелишне вкратце ознакомиться с возможностями этих компонентов.

15.2.1. Отладочный интерфейс процессора

В отличие от традиционных процессоров ARM7/ARM9 система отладки процессора Cortex-M3 базируется на архитектуре отладки CoreSight. Традиционно в процессорах ARM для обращения к регистрам процессора и управления интерфейсом памяти использовался интерфейс JTAG (Joint Test Action Group — объединённая рабочая группа по автоматизации тестирования). В процессоре Cortex-M3 управление логикой отладки осуществляется по специальному интерфейсу, называемому портом доступа к средствам отладки (DAP), который практически идентичен интерфейсу шины APB архитектуры AMBA. Для управления интерфейсом DAP служит другой компонент, преобразующий протоколы обмена JTAG или Serial-Wire (SW) в протокол интерфейса шины DAP.

Поскольку внутренняя шина средств отладки во многом идентична шине APB, к ней можно легко подключать различные компоненты, получая в результате хорошо масштабируемую систему отладки. Кроме того, разделение интерфейса отладки и аппаратных средств управления отладкой обеспечивает полную прозрачность типа интерфейса, реализованного на кристалле. Это позволяет выполнять одни и те же задачи отладки вне зависимости от типа используемого интерфейса.

Управление имеющимися функциями отладки процессора Cortex-M3 осуществляется контроллером прерываний NVIC и рядом других компонентов, таких как модули FPB, DWT и ITM. В контроллере NVIC имеется несколько регистров для управления основными отладочными операциями, например остановом и пошаговым выполнением программы, тогда как другие блоки поддерживают более продвинутые возможности, такие как точки наблюдения, точки останова и вывод отладочных сообщений.

15.2.2. Интерфейс хоста отладки

Технология CoreSight поддерживает разные типы интерфейсов для соединения хоста отладки и системы на кристалле. Традиционно для этой цели использовался интерфейс JTAG. Теперь же, благодаря тому, что отладочный интерфейс процессора был заменён на шинный интерфейс общего назначения, мы можем, подключая различные модули между хостом отладки и интерфейсом отладки процессора, работать с самыми разными кристаллами, имеющими различные интерфейсы хоста отладки, не внося изменений в интерфейс отладки процессора.

На сегодняшний день устройства на базе Cortex-M3 поддерживают два различных интерфейса хоста отладки. Первый из них — это уже известный нам интерфейс JTAG, а второй — новый протокол обмена данными, получивший название *Serial-Wire* (или, сокращённо, SW). В интерфейсе SW число сигнальных ли-

ний уменьшено до двух. Компания ARM предлагает несколько модулей портов отладки (Debug Port — DP), поддерживающих различные протоколы обмена данными. С одной стороны модуль DP подключается к аппаратному отладчику, а с другой — к шинному интерфейсу DAP.

Почему именно Serial-Wire?

Процессор Cortex-M3 рассчитан на применение в бюджетных микроконтроллерах, среди которых нередко встречаются устройства с очень малым числом выводов. Так, некоторые микроконтроллеры начального уровня выпускаются в корпусах, имеющих всего 28 выводов. Несмотря на большую популярность протокола JTAG, выделение 4 выводов для целей отладки оказывается неприемлемым для 28-выводных устройств. Поэтому протокол Serial-Wire является весьма привлекательным решением, поскольку позволяет уменьшить число выводов, используемых для подключения отладчика, до двух.

15.2.3. Модули DP, AP и DAP

Соединение внешнего аппаратного отладчика с интерфейсом отладки процессора Cortex-M3 показано на **Рис. 15.1**.

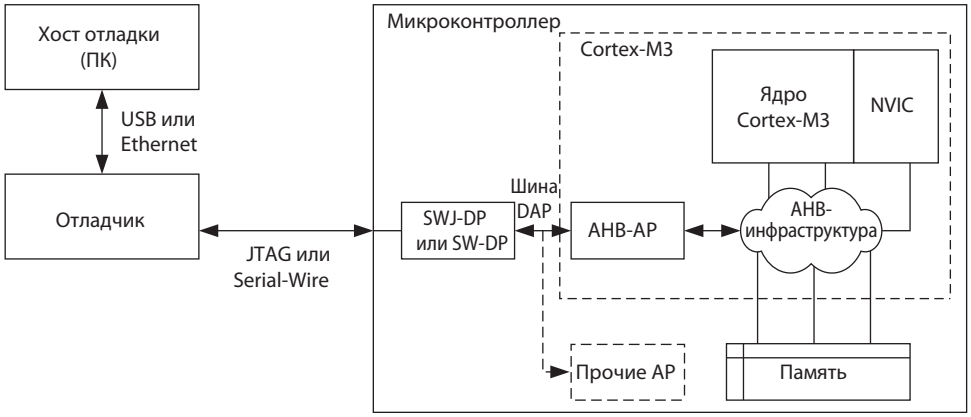


Рис. 15.1. *Подключение хоста отладки к процессору Cortex-M3.*

Интерфейсный модуль DP, в качестве которого обычно используется порт отладки Serial-Wire/JTAG (SWJ-DP) либо порт отладки Serial-Wire (SW-DP), преобразует сигналы отладчика в транзакции 32-битной шины отладки общего назначения (шина DAP на рисунке). Модуль SWJ-DP поддерживает протоколы JTAG и Serial-Wire, а модуль SW-DP — только Serial-Wire. В числе компонентов линейки CoreSight, предлагаемых компанией ARM, также имеется модуль JTAG-DP, поддерживающий только протокол JTAG. Разработчики микросхем могут выбрать любой из этих трёх модулей в соответствии со своими предпочтениями. Адреса на шине DAP являются 32-битными, при этом старшие 8 бит шины адреса используются для выбора конкретного устройства. Соответственно, к шине DAP может быть подключено до 256 устройств. В самом процессоре задействован всего один адрес, поэтому при необходимости вы можете подключить к шине DAP ещё 255 модулей порта доступа (AP). То есть теоретически вы можете реали-

зовать на одном кристалле сотни процессоров, использующих единственное подключение по интерфейсу JTAG или Serial-Wire.

Для подключения к шине DAP в процессоре Cortex-M3 используется порт доступа шины AHB (AHB-AP). Это устройство выполняет функции моста, преобразуя команды отладчика в транзакции на шине AHB, которые затем передаются во внутреннюю систему шин процессора. Такое решение обеспечивает доступ ко всему адресному пространству процессора, в том числе и к регистрам управления отладкой контроллера NVIC.

Среди продукции линейки CoreSight имеются модули портов доступа различных типов, в том числе порта доступа шины AHB (APB-AP) и порта доступа интерфейса JTAG (JTAG-AP). Модуль APB-AP может использоваться для формирования транзакций на шине AHB, а модуль JTAG-AP — для управления традиционными интерфейсами на базе протокола JTAG, такими как интерфейс отладки процессора ARM7.

15.2.4. Интерфейс трассировки

Другие компоненты архитектуры CoreSight относятся к трассировке. В процессоре Cortex-M3 могут присутствовать следующие источники трассировочной информации:

- трассировка команд — осуществляется модулем ETM;
- трассировка данных — осуществляется модулем DWT;
- отладочные сообщения — генерируются модулем ITM, обеспечивающим вывод сообщений стандартными средствами языка Си, такими как функция `printf()`, в графическом интерфейсе отладчика.

В процессе трассировки её результаты в виде пакетов данных выставляются источниками трассировки, такими как модуль ETM, на шину данных трассировки ATB (Advanced Trace Bus). Если в системе на кристалле имеется несколько источников трассировки (скажем, в случае многопроцессорной системы), то потоки данных с шин ATB могут быть объединены посредством специального модуля (в архитектуре CoreSight такой модуль называется ATB funnel). Сформированный поток данных поступает в модуль интерфейса порта трассировки (TPIU), который пересылает его внешнему трассировщику. После того как эти данные попадут в хост отладки (например, ПК), их можно будет разделить обратно на несколько потоков.

Несмотря на то что в самом процессоре Cortex-M3 имеется несколько источников трассировки, встроенные в него компоненты отладки спроектированы таким образом, что позволяют объединять трассировочные данные без использования дополнительных модулей ATB funnel. Выходной интерфейс системы трассировки может напрямую подключаться к модулю TPIU специального исполнения, разработанного для Cortex-M3. Далее трассировочные данные считываются внешним трассировщиком и пересылаются хосту (как правило, ПК) для анализа.

15.2.5. Характеристики архитектуры CoreSight

Решения на основе архитектуры CoreSight имеют следующие преимущества:

- Содержимое памяти и регистров периферийных устройств может контролироваться даже во время работы процессора.
- Для управления отладочными интерфейсами нескольких процессоров достаточно одного экземпляра отладчика. Например, при использовании интерфейса JTAG требуется только один TAP-контроллер, даже при наличии на кристалле нескольких процессоров.
- Внутренние интерфейсы отладки имеют простую шинную структуру, что обеспечивает их масштабируемость и простоту реализации дополнительных логических узлов для тестирования других блоков кристалла или SoC.
- Для захвата нескольких потоков данных трассировки достаточно одного внешнего трассировщика; в хосте отладки эти данные разделяются обратно на несколько потоков.

Система отладки, используемая в процессоре Cortex-M3, несколько отличается от стандартной реализации архитектуры CoreSight:

- Компоненты трассировки разработаны специально для процессора Cortex-M3. Некоторые из ATB-интерфейсов являются 8-битными, тогда как в архитектуре CoreSight везде используются 32-битные шины.
- Система отладки процессора Cortex-M3 не поддерживает технологию TrustZone[1)].
- Компоненты отладки располагаются в системной области памяти, тогда как в системах со стандартной архитектурой CoreSight для управления компонентами отладки используется отдельная шина со своим адресным пространством. Пример системы, в которой применяется традиционная реализация архитектуры CoreSight, приведён на **Рис. 15.2**.

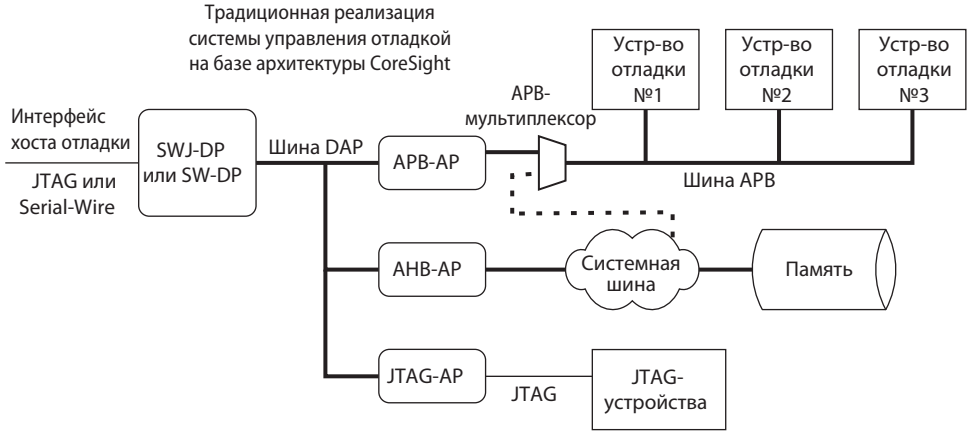


Рис. 15.2. *Традиционная система отладки на базе технологии CoreSight.*

[1)] Технология TrustZone, разработанная компанией ARM, реализует различные механизмы защиты для встраиваемых устройств.

В процессоре Cortex-M3 все компоненты отладки располагаются в общем адресном пространстве (**Рис. 15.3**).

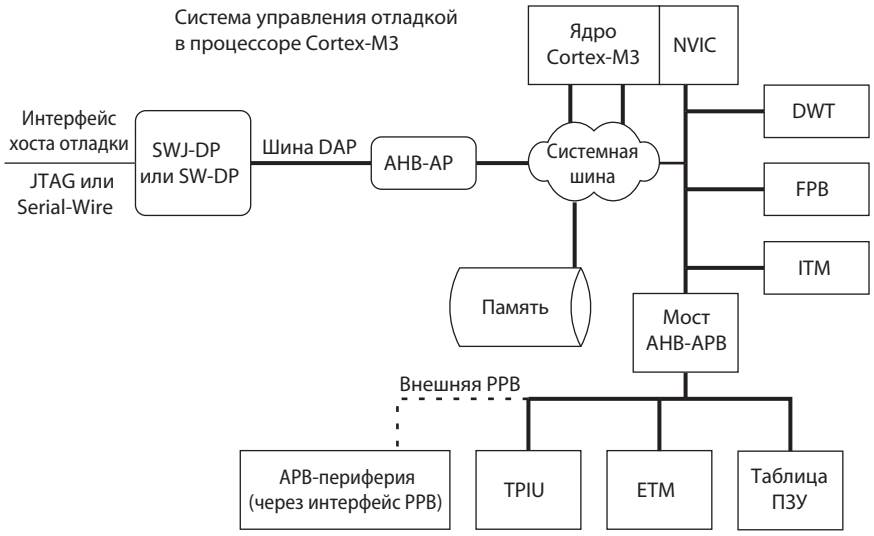


Рис. 15.3. *Система отладки процессора Cortex-M3.*

Дополнительную информацию об архитектуре CoreSight можно найти в руководстве [3].

Несмотря на то что компоненты отладки в процессоре Cortex-M3 организованы несколько иначе, нежели в обычных системах, задействующих архитектуру CoreSight, коммуникационные интерфейсы и протоколы, используемые в процессоре, полностью совместимы с данной архитектурой. Это позволяет напрямую подключаться к другим системам, использующим архитектуру CoreSight. В частности, в процессоре Cortex-M3 могут применяться такие компоненты, как модуль TPIU, модули портов отладки (DP) и блоки системы трассировки, позволяя включать его в многоядерную систему отладки.

15.3. Режимы отладки

В процессоре Cortex-M3 предусмотрено два режима выполнения отладочных операций. Первый из них — режим останова, при котором процессор полностью прекращает выполнение программы. Второй — режим исключения Debug monitor, при котором отладочные операции выполняются в обработчике данного исключения. В этом режиме могут возникать исключения с более высоким приоритетом. Исключение Debug monitor (исключение №12) имеет программируемый приоритет. Указанное исключение может быть вызвано как различными событиями отладки, так и ручной установкой бита признака отложенного прерывания. Характерные особенности обоих режимов:

1. Режим останова:
   - исполнение команд останавливается;
   - системный таймер SYSTICK останавливается;
   - поддерживаются пошаговые операции;
   - при пошаговом выполнении программы возможно возникновение и обработка прерываний. Внешние прерывания могут маскироваться.
2. Режим монитора отладки:
   - процессор выполняет обработчик исключения Debug monitor (исключение №12);
   - счётчик таймера SYSTICK продолжает работать;
   - новые прерывания могут как прерывать, так и не прерывать выполнение обработчика монитора отладки в зависимости от приоритета исключения Debug monitor и приоритета нового прерывания;
   - если событие отладки возникнет во время обработки прерывания с более высоким приоритетом, то оно будет пропущено;
   - поддерживаются пошаговые операции;
   - содержимое памяти (в частности, стека) может изменяться обработчиком исключения Debug monitor при сохранении контекста и во время выполнения обработчика.

Потребность в мониторе отладки обусловлена существованием таких систем, в которых останов процессора для выполнения отладочных операций невозможен в принципе. Например, в системе управления автомобильным двигателем или контроллере жёсткого диска процессор во время отладки должен продолжать обслуживать запросы прерываний, чтобы обеспечить безопасное выполнение требуемых операций и предотвратить повреждение отлаживаемого устройства. Используя монитор, отладчик может останавливать и отлаживать процессы уровня потока и обработчики низкоприоритетных прерываний, не влияя при этом на работу обработчиков системных исключений и прерываний с более высокими приоритетами.

Для перехода в режим останова необходимо установить бит C_DEBUGEN регистра DHCSR контроллера NVIC в 1. Программирование этого бита может быть выполнено только по интерфейсу DAP; соответственно, вы не сможете остановить процессор при отсутствии отладчика. После установки бита C_DEBUGEN ядро процессора может быть остановлено установкой бита C_HALT того же регистра. Данный бит может быть установлен как отладчиком, так и самой отлаживаемой программой.

В регистре DHCSR есть биты, назначение которых зависит от выполняемой операции. Так, при операциях записи в биты [31:16] регистра заносится ключ доступа. При чтении же регистра указанный ключ совсем не нужен, поэтому в старшем полуслове считанного значения содержатся биты состояния режима отладки (**Табл. 15.1**).

Таблица 15.1. **Регистр управления и состояния отладки в режиме останова DHCSR (0xE000EDF0)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:16	KEY	W	—	Ключ доступа к функциям отладки; при записи в регистр эти биты должны содержать значение 0xA05F, в противном случае операция записи не будет выполнена
25	S_RESET_ST	R	—	Ядро было сброшено или находится в состоянии сброса. Бит сбрасывается при чтении
24	S_RETIRE_ST	R	—	Команда завершена с момента последнего чтения регистра. Бит сбрасывается при чтении
19	S_LOCKUP	R	—	Если бит установлен в 1, то ядро находится в состоянии блокировки
18	S_SLEEP	R	—	Если бит установлен в 1, то ядро находится в спящем режиме
17	S_HALT	R	—	Если бит установлен в 1, то ядро остановлено
16	S_REGRDY	R	—	Операция чтения/записи регистра завершена
15:6	Зарезервировано	—	—	Зарезервировано
5	C_SNAPSTALL	R/W	0*	Прерывание «зависших» операций обращения к памяти
4	Зарезервировано	—	—	Зарезервировано
3	C_MASKINTS	R/W	0*	Маскирование прерываний в спящем режиме; бит может быть изменён только при остановленном процессоре
2	C_STEP	R/W	0*	Выполнение одной команды (пошаговая работа процессора); действителен только при установленном бите C_DEBUGEN
1	C_HALT	R/W	0*	Останов процессорного ядра; действителен только при установленном бите C_DEBUGEN
0	C_DEBUGEN	R/W	0*	Разрешение отладки в режиме останова

*Управляющие биты в регистре DHCSR сбрасываются только при сбросе по включению питания. При сбросе системы (например, посредством регистра AIRCR контроллера NVIC) эти биты не сбрасываются.

Необходимо отметить, что регистр DHCSR используется исключительно отладчиком. Прикладные программы не должны изменять содержимое этого регистра во избежание конфликтов с отладочными средствами.

- Управляющие биты регистра DHCSR сбрасываются только при сбросе по включению питания. При сбросе системы (например, посредством регистра AIRCR контроллера NVIC) эти биты не сбрасываются. Для отладки в режиме монитора используется другой регистр контроллера NVIC, а именно регистр DEMCR (**Табл. 15.2**). Помимо битов управления монитором отладки, регистр DEMCR содержит бит разрешения системы трассировки TRCENA и ряд битов управления перехватом векторов прерываний VC_xxx (последние могут использоваться только при отладке в режиме останова). Если на момент возникновения какого-либо отказа (или сброса ядра) был установлен соответствую-

щий бит VC_xxx, то формируется запрос на останов, в результате которого процессор будет остановлен сразу же после завершения текущей команды.

- Управляющие биты TRCENA и VC_xxx регистра DEMCR сбрасываются при сбросе по включению питания. При сбросе системы эти биты не сбрасываются. В то же время биты управления отладкой в режиме монитора сбрасываются в обоих случаях.

Таблица 15.2. **Регистр управления исключением и монитором отладки DEMCR (0xE000EDFC)**

Биты	Обозначение	Тип	Значение после сброса	Описание
24	TRCENA	R/W	0*	Разрешение работы системы трассировки; для использования модулей DWT, ETM, ITM и TPIU должен быть установлен в 1
23:20	Зарезервировано	—	—	Зарезервировано
19	MON_REQ	R/W	0	Указывает на то, что вызов монитора отладки произошёл в результате ручной установки бита отложенности прерывания, а не в результате события аппаратного отладчика
18	MON_STEP	R/W	0	Выполнение одной команды (пошаговая работа процессора); действителен только при установленном бите MON_EN
17	MON_PEND	R/W	0	Генерация запроса исключения Debug monitor; процессор приступит к обработке исключения согласно его приоритету
16	MON_EN	R/W	0	Разрешение исключения Debug monitor
15:11	Зарезервировано	—	—	Зарезервировано
10	VC_HARDERR	R/W	0*	Перехват тяжёлых отказов
9	VC_INTERR	R/W	0*	Перехват ошибок, возникающих при входе в обработчик исключения и при выходе из него
8	VC_BUSERR	R/W	0*	Перехват отказов шины
7	VC_STATERR	R/W	0*	Перехват отказов программы, вызванных некорректной информацией о состоянии
6	VC_CHKERR	R/W	0*	Перехват отказов программы, вызванных невыровненным доступом либо делением на ноль
5	VC_NOCPERR	R/W	0*	Перехват отказов программы, вызванных попыткой обращения к сопроцессору
4	VC_MMERR	R/W	0*	Перехват отказов системы управления памятью
3:1	Зарезервировано	—	—	Зарезервировано
0	VC_CORERESET	R/W	0*	Перехват сброса ядра

*Управляющие биты в регистре DEMCR сбрасываются только при сбросе по включению питания. При сбросе системы (например, посредством регистра AIRCR контроллера NVIC) эти биты не сбрасываются.

15.4. События отладки

Процессор Cortex-M3 может перейти в режим отладки (всё равно, в какой именно) по разным причинам. На **Рис. 15.4** показаны события, которые могут вызвать переход процессора в режим останова.

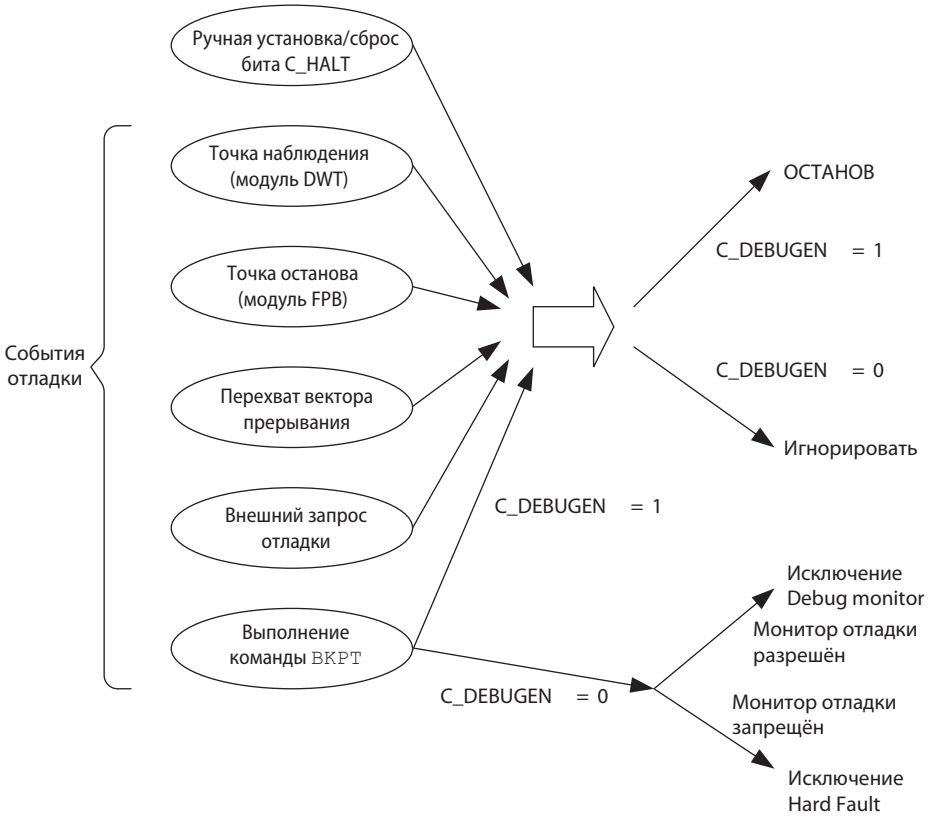


Рис. 15.4. *События отладки в режиме останова.*

Внешний запрос отладки формируется путём активации входа EDBGREQ процессора Cortex-M3. Куда именно будет подключён этот вход, зависит от разработчика конкретного микроконтроллера или SoC. Вполне возможны решения, в которых этот вход будет жёстко подтянут к НИЗКОМУ уровню, т.е. в принципе не может быть активирован. Однако чаще всего он подключается так, чтобы активироваться при формировании отладочных событий дополнительными компонентами отладки (разработчики микросхем могут размещать дополнительные компоненты в своих SoC). В многопроцессорной системе этот сигнал может активироваться при генерации отладочных событий другим процессором.

После завершения отладки нормальное выполнение программы может быть возобновлено посредством сброса бита C_HALT. Аналогичным образом, при отладке в режиме исключения Debug Monitor для вызова монитора отладки могут использоваться различные события (**Рис. 15.5**).

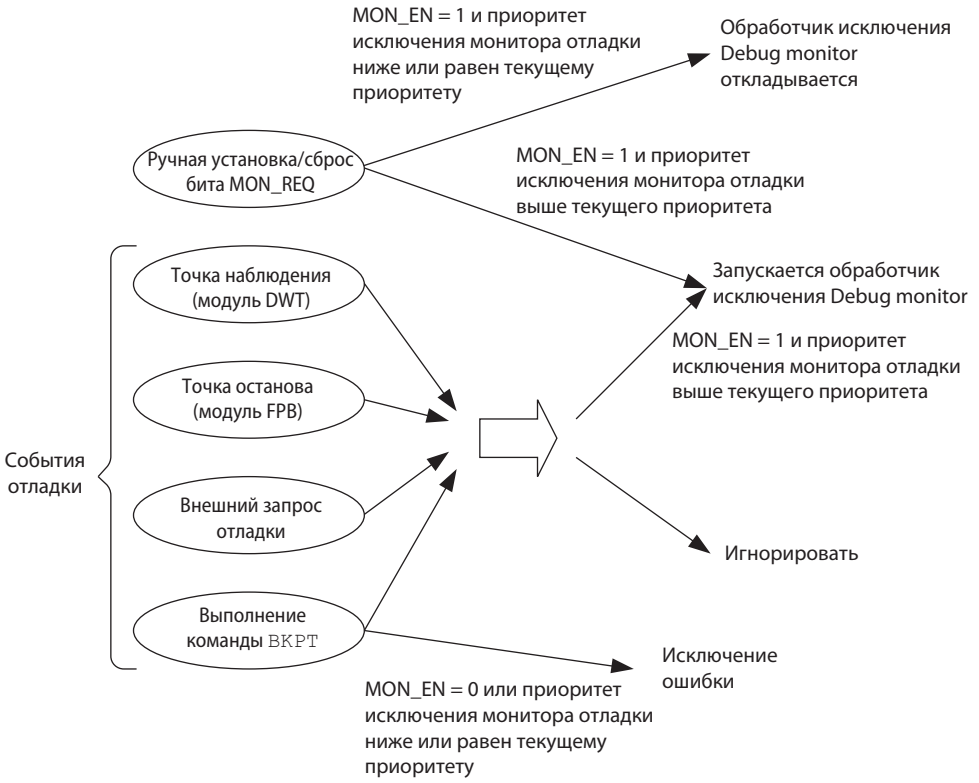


Рис. 15.5. *Отладочные события для отладки в режиме монитора.*

Процесс отладки с использованием монитора несколько отличается от отладки в режиме останова. Это связано с тем, что исключение Debug monitor является всего лишь одним из массы других исключений, и на его обработку влияет текущий приоритет процессора, если последний выполняет обработчик другого исключения.

После завершения отладки нормальное выполнение программы возобновляется при выходе из обработчика исключения.

15.5. Точки останова в процессоре Cortex-M3

Одним из наиболее широко используемых средств отладки в большинстве микроконтроллеров являются точки останова. В процессоре Cortex-M3 поддерживаются два механизма точек останова:

- команда точки останова `BKPT`;
- останов с использованием компараторов адреса модуля FPB.

Команда вставки точки останова (`BKPT #immed8`) является 16-битной командой Thumb, машинный код которой имеет вид 0xBExx. Младшие 8 бит кода определяются значением константы, расположенной после мнемоники команды. При выполнении этой команды генерируется событие отладки, которое может быть использовано для останова ядра процессора (если установлен бит C_DBGEN)

или для активации исключения монитора отладки, если последнее разрешено. Поскольку данное исключение является обычным исключением с изменяемым приоритетом, то монитор отладки может вызываться либо из основной программы, работающей в режиме потока, либо из обработчика исключения с меньшим приоритетом. Как следствие, если отладка ведётся с использованием монитора, то команда `BRPT` не должна вызываться из обработчиков исключений, таких как немаскируемое прерывание или Hard Fault, а сам монитор отладки в этом случае может быть запущен только после завершения обработчика исключения.

При возврате из обработчика исключения Debug monitor процессор возвращается по адресу команды `BKPT`, а не по следующему адресу, как при возврате из прочих обработчиков. Это связано с тем, что при нормальном использовании команды `BKPT` она подменяет собой обычную команду. Соответственно, после достижения процессором точки останова и выполнения требуемых отладочных операций команда `BKPT` заменяется на исходную команду (остальная память программ не затрагивается).

Если попытаться выполнить команду `BKPT` при сброшенных битах C_DEBUGEN и MON_EN, то будет сгенерировано исключение Hard Fault. При этом будут установлены бит DEBUGEVT регистра HFSR и бит BKPT регистра DFSR.

Модуль FPB может генерировать события точек останова даже в том случае, если модификация памяти программ невозможна. Правда, в этом случае его возможности оказываются ограничены шестью точками останова в области кода и двумя — в области констант. Более подробно модуль FPB будет рассмотрен в следующей главе.

15.6. Получение доступа к содержимому регистров при отладке

В контроллере NVIC предусмотрено два регистра для поддержки функций отладки. Это регистр выбора регистра ядра DCRSR (**Табл. 15.3**) и регистр содержимого регистра ядра DCRDR (**Табл. 15.4**). Указанные регистры позволяют отладчику обращаться к регистрам процессора. Учтите, что пересылка содержимого регистров может осуществляться только при остановленном процессоре.

Чтобы прочитать содержимое какого-либо регистра процессора с помощью этих регистров, необходимо выполнить следующие операции:

1. Убедиться, что процессор находится в состоянии останова.
2. Записать в регистр DCRSR значение со сброшенным битом [16] (операция чтения).
3. Дождаться установки бита S_REGRDY регистра DHCSR (0xE000EDF0).
4. Прочитать регистр DCRDR для получения содержимого регистра.

Для записи в регистр необходимо выполнить аналогичные действия:

1. Убедиться, что процессор находится в состоянии останова.
2. Загрузить значение в регистр DCRDR.
3. Записать в DCRSR значение с установленным битом 16 (операция записи).
4. Дождаться установки бита S_REGRDY регистра DHCSR (0xE000EDF0).

Таблица 15.3. **Регистр DCRSR (0xE000EDF4)**

Биты	Обозначение	Тип	Значение после сброса	Описание
16	REGWnR	W	—	Направление передачи данных: Запись = 1, Чтение = 0
15:5	Зарезервировано	—	—	—
4:0	REGSEL	W	—	Регистр, к которому производится обращение: 00000 = R0 00001 = R1 . . . 01111 = R15 10000 = xPSR/флаги 10001 = Основной указатель стека (MSP) 10010 = Указатель стека процесса (PSP) 10100 = Регистры специального назначения: [31:24] CONTROL [23:16] FAULTMASK [15:8] BASEPRI [7:0] PRIMASK Остальные значения зарезервированы

Таблица 15.4. **Регистр DCRDR (0xE000EDF8)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:0	Data	R/W	—	Регистр данных для хранения результата чтения регистра или значения, записываемого в регистр

Регистры DCRDR и DCRSR позволяют пересылать содержимое регистров только при отладке в режиме останова. При отладке с использованием монитора содержимое одних регистров можно прочитать из стека, а других — непосредственно в обработчике исключения Debug monitor.

Регистр DCRDR также может применяться для реализации полухостинга при наличии соответствующих библиотек функций и поддержки со стороны отладчика. Например, при использовании в приложении функции `printf` собственно вывод текста, как правило, осуществляется многократными вызовами функции `putc` (вывод символа). Реализация данной функции может предусматривать сохранение выводимого символа и текущего состояния в регистре DCRDR с последующей активацией режима отладки. Отладчик сможет обнаружить останов ядра и считать этот символ для его вывода на экран. Однако это решение требует останова процессора, тогда как полухостинг средствами модуля ITM не накладывает такого ограничения.

15.7. Прочие отладочные возможности ядра

В контроллере NVIC имеются и другие возможности, используемые при отладке:

- *Внешний сигнал запроса отладки.* Контроллер NVIC имеет вход внешнего сигнала запроса отладки, который позволяет переводить процессор Cortex-M3

в режим отладки при наступлении внешнего события. Данная возможность актуальна в основном для многопроцессорных систем. В простых микроконтроллерах этот вход обычно подтягивается к НИЗКОМУ уровню.

- *Регистр DFSR*. Поскольку процессор Cortex-M3 поддерживает различные события отладки, в нём предусмотрен специальный регистр DFSR, с помощью которого отладчик может определить произошедшее событие.
- *Управление сбросом*. Во время отладки процессорное ядро можно перезапустить с помощью бита VECTRESET или SYSRESETREQ регистра AIRCR контроллера NVIC (0xE000ED0C). Данный регистр управления сбросом позволяет сбрасывать процессор, не затрагивая при этом компоненты отладки, имеющиеся в системе.
- *Маскирование прерываний*. Указанная возможность очень полезна при пошаговой отладке. В частности, если вам необходимо отладить программу и при этом вы не хотите попадать в процедуры обработки прерываний, то вы можете замаскировать запросы прерываний. Для этого достаточно установить бит C_MASKINTS регистра DHCSR (0xE000EDF0).
- *Прерывание «зависших» пересылок*. Если пересылка по шине «зависла» на длительное время, то её можно прервать установкой бита C_SNAPSTALL регистра DHCSR (0xE000EDF0). Данная возможность может использоваться только при отладке в режиме останова.

ГЛАВА 16
КОМПОНЕНТЫ ОТЛАДКИ

16.1. Общие сведения

В составе процессора Cortex-M3 имеются различные компоненты, обеспечивающие реализацию таких функций, как точки останова, точки наблюдения, коррекция флэш-памяти (Flash Patch) и трассировка. Если вы занимаетесь разработкой прикладных программ, то вам скорее всего никогда не потребуется детальная информация об этих компонентах, поскольку они обычно используются исключительно отладочными средствами. В данной главе приводятся основные сведения обо всех компонентах отладки процессора. Если же вам необходима более подробная информация об указанных компонентах, обратитесь к руководству [1].

Программирование всех компонентов поддержки трассировки, так же как и модуля FPB, осуществляется по шине собственных периферийных устройств (PPB) процессора. В подавляющем большинстве случаев программирование этих компонентов производится только средствами отладки. Прикладным программам не рекомендуется обращаться к компонентам отладки (за исключением регистров портов стимулов модуля ITM), поскольку это может помешать работе отладчика.

16.1.1. Система трассировки в процессоре Cortex-M3

Система трассировки процессора Cortex-M3 базируется на архитектуре CoreSight. Результаты трассировки генерируются в виде пакетов данных различного размера (имеющих разное число байтов). Компоненты трассировки пересылают эти пакеты по шине ATB в модуль TPIU, который форматирует их в соответствии с требуемым протоколом интерфейса трассировки. После этого данные считываются внешним устройством, таким как анализатор порта трассировки TPA (**Рис. 16.1**).

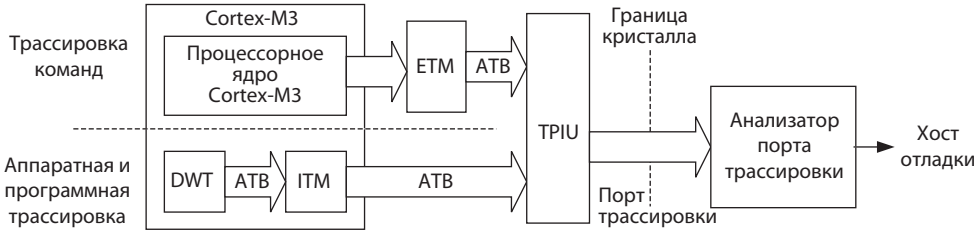


Рис. 16.1. *Система трассировки процессора Cortex-M3.*

В обычных процессорах Cortex-M3 может присутствовать до трёх источников трассировочных данных — это модули ETM, ITM и DWT. Заметьте, модуль ITM является опциональным, поэтому некоторые устройства с процессором Cortex-M3 не поддерживают трассировку команд. Во время работы каждому источнику трассировки присваивается уникальный 7-битный идентификатор (ATID), который пересылается вместе с пакетом данных по шине ATB. Этот идентификатор позволяет хосту отладки разделить полученные пакеты на отдельные потоки данных трассировки.

В отличие от многих других стандартных компонентов CoreSight, компоненты отладки процессора Cortex-M3 могут сами объединять потоки ATB, тогда как в обычных системах CoreSight для объединения пакетов используется отдельный блок, называемый *ATB funnel*.

Перед использованием системы трассировки необходимо установить бит разрешения трассировки TRCENA регистра DEMCR (см. **Табл. 15.2** или **Табл. Г.38** в Приложении Г). В противном случае, система трассировки будет отключена. Во время нормальной работы, не требующей трассировки, сброс бита TRCENA позволяет отключить часть логических узлов процессора и уменьшить его потребление.

16.2. Компоненты трассировки: модуль DWT

Модуль DWT имеет следующие возможности:

1. Содержит в своём составе четыре компаратора, каждый из которых может быть сконфигурирован для реализации:
   а) аппаратной точки наблюдения (генерирует событие точки наблюдения, переводящее процессор в один из режимов отладки);
   б) триггера ETM (в результате модуль ETM вставляет в поток трассировки команд триггер-пакет);
   в) триггера события выборки счётчика команд;
   г) триггера события выборки адреса.
   Первый компаратор также может использоваться для сравнения с содержимым счётчика тактов (CYCCNT) вместо сравнения со значением адреса.
2. Имеет счётчики для подсчёта:
   а) числа тактов (CYCCNT);
   б) числа «свёрнутых» команд;
   в) числа элементов операций загрузки/сохранения;
   г) числа тактов при нахождении в спящем режиме;
   д) числа тактов, затраченных на выполнение команды;
   е) числа тактов, связанных с накладными расходами на исключение.
3. Позволяет считывать значения PC с заданной периодичностью.
4. Обеспечивает трассировку событий, связанных с исключениями.

При использовании компаратора в качестве аппаратной точки наблюдения или триггера ETM он позволяет сравнивать либо адреса данных, либо значения счётчика команд. При работе в другом качестве компаратор сравнивает адреса.

Каждый компаратор имеет три регистра:

- COMP (регистр сравнения);
- MASK (регистр маски);
- FUNCTION (регистр управления).

Регистр COMP представляет собой 32-битный регистр, содержимое которого сравнивается с адресом данных (или значением счётчика команд или значением CYCCNT). Регистр MASK позволяет задавать, какие биты адреса будут игнорироваться во время сравнения (**Табл. 16.1**).

Таблица 16.1. **Декодирование содержимого регистров маски модуля DWT**

MASK	Игнорируемые биты
0	Сравниваются все биты
1	Игнорируется бит [0]
2	Игнорируются биты [1:0]
3	Игнорируются биты [2:0]
…	…
15	Игнорируются биты [14:0]

Используя регистр маски, можно отслеживать обращения к данным в диапазоне адресов до 32 Кбайт. Однако из-за ограниченного объёма FIFO-буфера в модулях DWT и ITM отследить большое число пересылок просто нереально, поскольку это приведёт к переполнению буфера трассировки и утере полученных данных.

Регистр FUNCTION определяет режим работы компаратора. Чтобы исключить некорректное поведение компаратора, перед программированием регистра FUNCTION в регистры MASK и COMP необходимо загрузить требуемые значения. Если нужно изменить функцию компаратора, то его сначала следует отключить, очистив регистр FUNCTION, затем соответствующим образом перепрограммировать регистры MASK и COMP, после чего снова включить.

Остальные счётчики модуля DWT обычно используются для профилирования кода приложения. Их можно запрограммировать на генерацию событий (в виде пакетов данных трассировки) при переполнении. В качестве типичного применения можно указать использование регистра CYCNT для подсчёта числа тактов, затраченных на выполнение конкретной задачи.

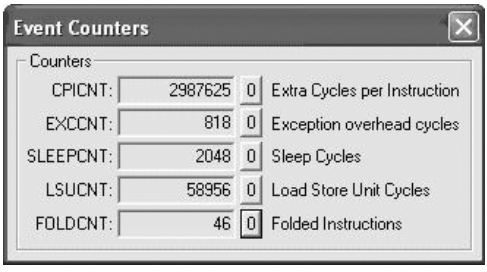


Рис. 16.2. *Статистика выполнения программы, собранная с использованием счётчиков модуля DWT (ИСР µVision компании Keil).*

Например, в ИСР µVision компании Keil эти счётчики могут использоваться для генерации статистической информации (**Рис. 16.2**). Данные счётчики запускают генерацию пакетов событий, которые собираются отладчиком с выхода модуля Serial-Wire Viewer (SWV).

16.3. Компоненты трассировки: модуль ITM

Модуль ITM имеет следующие возможности (**Рис. 16.3**):

- программа может напрямую записывать сообщения в порты стимулов модуля ITM и выводить их в виде данных трассировки;
- модуль DWT может генерировать пакеты трассировки и выводить их через модуль ITM;
- модуль ITM может генерировать пакеты временных меток, вставляемые в поток данных трассировки, что помогает отладчику вести хронометраж событий.

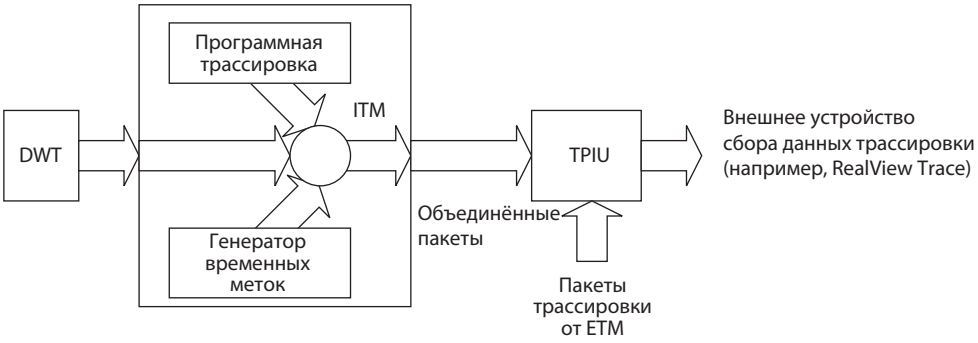


Рис. 16.3. *Объединение пакетов трассировки в модулях ITM и TPIU.*

Для вывода данных модуль ITM использует порт трассировки, поэтому если в микроконтроллере или SoC отсутствует модуль TPIU, то результаты трассировки никак не получится передать за пределы процессора. Из этого следует, что прежде чем задействовать модуль ITM, необходимо убедиться в том, что микроконтроллер или SoC имеют требуемую функциональность. В крайнем случае, если указанные возможности недоступны, вы можете выводить отладочные сообщения, используя полухостинг на основе регистра отладки контроллера NVIC (этот метод поддерживается отладчиком RealView Debugger компании ARM) или же модуль UART.

Чтобы задействовать модуль ITM, необходимо бит TRCENA регистра DEMCR установить в 1. В противном случае, модуль ITM будет отключён, а его регистры недоступны.

Кроме того, в модуле ITM имеется специальный регистр блокировки. Перед программированием модуля в этот регистр необходимо загрузить значение 0xC5ACCE55 (ключ доступа CoreSight), иначе все операции записи в регистры модуля ITM будут игнорироваться.

И наконец, в самом модуле ITM имеется ещё один регистр управления, отвечающий за разрешение/запрещение отдельных функций модуля. В этом регистре

также содержится поле ATID, в котором хранится значение идентификатора модуля ITM для шины ATB. Данный идентификатор должен отличаться от идентификаторов остальных источников трассировочных данных, чтобы хост отладки, получающий пакеты данных трассировки, мог выделить пакеты от модуля ITM среди других пакетов.

16.3.1. Программная трассировка с использованием модуля ITM

Одной из основных функций модуля ITM является поддержка вывода отладочных сообщений (скажем, посредством функции `printf`). В составе модуля ITM имеется 32 порта стимулов, в которые различные процессы могут выводить свою информацию. Последующее отделение сообщений различных процессов друг от друга осуществляется хостом отладки. Каждый порт может быть разрешён/запрещён посредством регистра разрешения трассировки ITM_TER, а также может быть запрограммирован (группами по восемь регистров) на предоставление пользовательским процессам доступа на запись.

В отличие от вывода текста посредством модуля UART, использование ITM не вносит большой задержки в работу программы. Внутри модуля ITM имеется FIFO-буфер, так что операции записи выводимых сообщений буферизуются. Соответственно, прежде чем писать в буфер, необходимо убедиться в том, что в нём есть место.

Выводимые сообщения могут быть считаны через интерфейс порта трассировки или интерфейс SWV модуля TPIU. При этом вам не потребуется удалять из финальной версии программы код, осуществляющий вывод отладочных сообщений, поскольку при сброшенном бите TRCENA модуль TPIU будет неактивным, и сообщения в любом случае выводиться не будут. Соответственно, вы можете включить вывод отладочных сообщений в «живой» системе и, используя регистр ITM_TER, выбрать интересующие вас порты.

К примеру, в ИСР μVision компании Keil имеется специальный модуль для сбора и отображения выводимой информации, который называется ITM Viewer (**Рис. 16.4**).

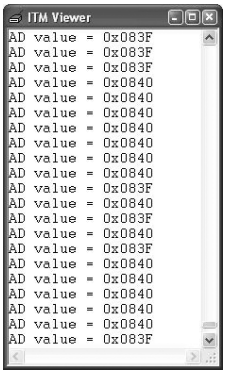


Рис. 16.4. *Окно «ITM Viewer» — вывод сообщений с выхода модуля ITM.*

16.3.2. Аппаратная трассировка с использованием модулей ITM и DWT

Модуль ITM задействуется при выводе пакетов данных аппаратной трассировки. Собственно пакеты генерируются модулем DWT, а модуль ITM выступает в качестве устройства объединения этих пакетов. Для использования трассировки средствами модуля DWT необходимо установить бит DWTEN регистра управления модуля ITM; остальные настройки модуля DWT выполняются через его регистры.

16.3.3. Временные отметки модуля ITM

При поступлении в FIFO-буфер модуля ITM очередного пакета трассировки модуль может вставлять в поток данных трассировки пакет временной отметки. Указанный пакет генерируется также при переполнении счётчика временных отметок.

Каждый пакет временной отметки содержит время, прошедшее с момента предыдущего события. Используя эти пакеты, трассировщик может определить время генерации каждого пакета и, таким образом, реконструировать процесс возникновения отладочных событий.

Объединив возможности трассировки модулей DWT и ITM, мы можем получить много полезной информации. Так, из содержимого окна трассировки исключений ИСР μVision мы можем узнать, какие исключения были обработаны и сколько ушло времени на каждое из них (**Рис. 16.5**).

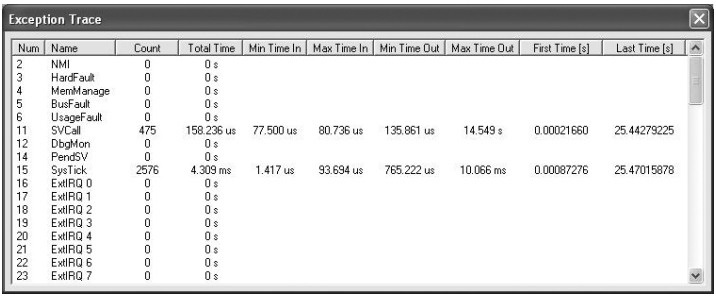


Рис. 16.5. *Окно «Exception Trace» ИСР μVision.*

16.4. Компоненты трассировки: модуль ETM

Модуль ETM используется для трассировки команд. Данный модуль является опциональным и может отсутствовать в отдельных устройствах на базе процессора Cortex-M3. Если модуль ETM включён, то в процессе трассировки он генерирует пакеты трассировки команд. В модуле предусмотрен FIFO-буфер, облегчающий захват потока трассировочных данных внешним устройством.

Для уменьшения объёма данных, генерируемых модулем ETM, он сообщает не обо всех командах, выполненных процессором. Как правило, модуль выводит информацию о ходе выполнения программы и значения адресов только в определённые моменты времени (например, при выполнении переходов). Поскольку хост отладки имеет копию двоичного образа программы, то этих данных ему бу-

дет достаточно для того, чтобы восстановить последовательность команд, выполненных процессором.

Модуль ETM также взаимодействует с другими компонентами отладки, такими как модуль DWT. В частности, компаратор модуля DWT может использоваться для генерации запускающих событий в модуле ETM или для управления пуском/остановом трассировки.

В отличие от модуля ETM традиционных процессоров ARM, модуль ETM, реализованный в процессоре Cortex-M3, не имеет собственных компараторов адреса, так как операции сравнения для модуля ETM может выполнять модуль DWT. Более того, поскольку трассировка данных также осуществляется модулем DWT, то реализация модуля ETM процессора Cortex-M3 довольно сильно отличается от традиционных решений для других процессорных ядер ARM.

Чтобы задействовать модуль ETM в процессоре Cortex-M3, необходимо выполнить следующие операции (этим занимается отладчик):

1. Установить бит TRCENA регистра DEMCR (см. **Табл. 15.2** или **Табл. Г.38**).
2. Разблокировать модуль ETM, чтобы получить возможность записи в его регистры. Для этого необходимо в регистр ETM_LOCK_ACCESS записать значение 0xC5ACCE55.
3. Занести в регистр идентификатора ATID уникальное значение, которое позволит хосту отладки отделить пакеты модуля ETM от пакетов из других источников трассировки.
4. Подать на вход разрешения неинвазивной отладки (NIDEN) модуля ETM сигнал ВЫСОКОГО уровня. Реализация этого сигнала зависит от конкретного устройства. Чтобы получить более подробную информацию, обратитесь к документации на используемый микроконтроллер.
5. Запрограммировать регистры управления модуля ETM на генерацию трассировочной информации.

16.5. Компоненты трассировки: модуль TPIU

Модуль TPIU используется для вывода пакетов трассировки, формируемых модулями ITM, DWT и ETM, во внешнее устройство (например, анализатор порта трассировки). Модуль TPIU процессора Cortex-M3 поддерживает два режима вывода информации:

- синхронный режим, использующий параллельную шину данных разрядностью до 4 бит;
- SWV-режим, использующий однопроводной интерфейс SWV[1)].

Реальная разрядность порта вывода данных в синхронном режиме может быть разной. Она зависит от типа корпуса микросхемы, а также от числа сигнальных контактов, которые можно будет задействовать для вывода данных трассировки. Максимальную разрядность порта трассировки, поддерживаемую конкретной микросхемой, можно определить из регистра TPIU_SSPSR модуля TPIU. Скорость вывода данных также программируется.

[1)] Доступен только в устройствах с процессором Cortex-M3 ревизии 1 и выше.

В режиме SWV применяется однобитный последовательный протокол, требующий всего один сигнальный провод. При этом также уменьшается пропускная способность канала передачи данных трассировки. При объединении SWV и протокола отладки Serial-Wire в качестве выхода SWV может задействоваться вывод TDO интерфейса JTAG (**Рис. 16.6**). Это позволяет использовать для сбора отладочной информации в режиме SWV стандартный JTAG-кабель (см., например, отладчик ULINK 2 компании Keil).

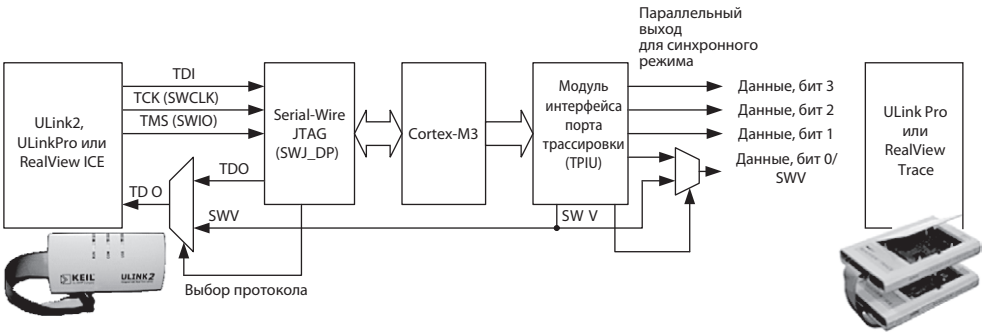


Рис. 16.6. *Использование выхода SWV.*

Также SWV может передавать данные на вывод, используемый в синхронном режиме в качестве одной из линий параллельного порта. Для сбора трассировочной информации, независимо от режима работы модуля TPIU, можно задействовать внешний анализатор порта трассировки, например RealView Trace компании ARM.

Если необходимо выполнить трассировку команд (с помощью модуля ETM), то желательно использовать синхронный режим, поскольку он обеспечивает более высокую пропускную способность. Для простой трассировки данных и трассировки событий (например, исключений), как правило, достаточно режима SWV, достоинством которого является меньшее число задействованных выводов.

Для использования модуля TPIU бит TRCENA регистра DEMCR следует установить в 1. Также необходимо запрограммировать соответствующим образом регистр выбора протокола (режима) и регистры управления разрядностью порта трассировки (эти операции выполняет ПО трассировки).

16.6. Модуль FPB

Модуль FPB выполняет следующие функции:

- установка аппаратных точек останова (генерирует событие точки останова для перевода процессора в один из режимов отладки);
- подмена команды или константы в области кода значением из области статического ОЗУ.

16.6.1. Точка останова

Назначение точек останова, в общем-то, понять не сложно. Во время отладки вы можете разместить одну или несколько точек останова по адресам программы

или констант. При выполнении команды или при чтении значения, расположенного по такому адресу, генерируется событие отладки, которое приводит к останову программы (для отладки в режиме останова) или генерации исключения Debug monitor (для отладки в режиме монитора). После этого вы можете проконтролировать содержимое регистров, памяти, начать пошаговую отладку программы и т.д.

16.6.2. Функция Flash Patch

Функция Flash Patch позволяет использовать программируемое ЗУ небольшого объёма для наложения «заплаток» на память программ, доступную только для чтения. Использование в качестве памяти программ масочного или однократно программируемого ПЗУ может значительно снизить стоимость продукции при массовом производстве. Однако в этом случае замена устройств при обнаружении ошибки в программе может дорого обойтись производителю. Если же добавить в устройство перепрограммируемую память (флэш или EEPROM) небольшого объёма, то можно будет ставить «заплатки» на программу, зашитую в устройстве. Для микроконтроллеров, использующих для хранения программы флэш-память, функция Flash Patch не актуальна, поскольку флэш-память допускает стирание и повторное программирование.

16.6.3. Компараторы

Модуль FPB содержит в общей сложности восемь компараторов:

- шесть компараторов команд;
- два компаратора констант.

Любой компаратор в каждый момент времени может использоваться для реализации либо точки останова, либо функции Flash Patch, но не обеих функций одновременно.

В модуле FPB имеется регистр управления функцией Flash Patch, который содержит бит, разрешающий работу модуля. Кроме того, для каждого компаратора предусмотрены отдельные биты разрешения, которые располагаются в регистрах управления компараторами. Для включения компаратора оба бита разрешения (модуля и компаратора) необходимо установить в 1.

Компараторы могут быть запрограммированы на то, чтобы переназначать адреса из области кода в область статического ОЗУ. Для использования данной функции в регистр REMAP необходимо занести начальный адрес переназначаемой области памяти. В трёх старших битах этого регистра (биты [31:29]) жёстко зашито значение b001, что ограничивает базовый адрес диапазоном 0x20000000…0x3FFFFF80, который всегда находится в области ОЗУ.

При совпадении адреса команды или константы с адресом, заданным в компараторе, производится чтение из таблицы, на которую указывает регистр REMAP (**Рис. 16.7**).

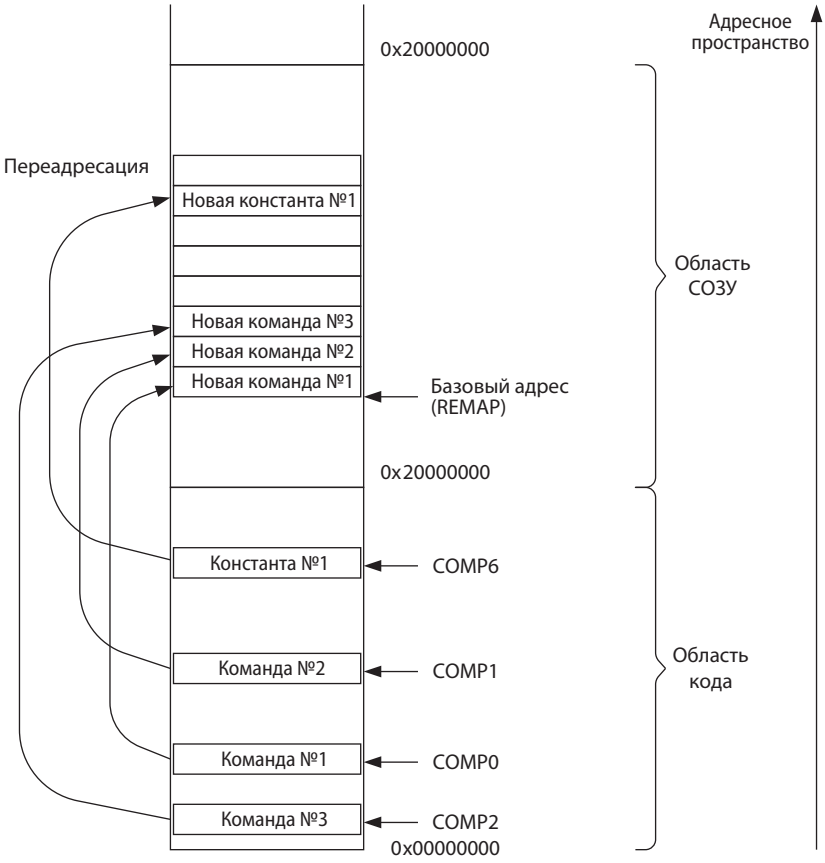


Рис. 16.7. *Функция Flash Patch: переназначение операций чтения команд и констант.*

Используя функцию переадресации, можно реализовать набор проверок типа «если — то», в которых оригинальные команда или константа будут подменяться другими (даже в том случае, если код программы расположен в ПЗУ или флэш-памяти). В частности, такое решение можно использовать для запуска программы или процедуры из ОЗУ, размещая такую «заплатку» в области кода, которая осуществляла бы переход к программе или процедуре тестирования. Это позволяет отлаживать устройства, в которых программа хранится в ПЗУ.

Второй функцией компараторов адреса команд является генерация точек останова с запуском одного из двух режимов отладки.

Загрузка константы — как?

При программировании на ассемблере нам очень часто приходится загружать непосредственные значения (константы) в регистры. Если значение константы велико, то для выполнения этой операции требуется несколько команд. Рассмотрим в качестве примера следующий фрагмент кода:

```
LDR   R0, =0xE000E400  ; Базовый адрес блока регистров приоритета
                       ; внешних прерываний
```

Поскольку ни в одной команде нет места для хранения 32-битного значения, нам придётся разместить константу в другой области памяти (обычно, сразу после кода программы), а затем прочитать её значение в регистр, воспользовавшись командой загрузки с адресацией относительно PC. То есть после дизассемблирования двоичного кода приведённого выше фрагмента мы получим что-то вроде:

```
LDR R0, [PC, #<immed_8>*4]
; immed_8 = (адрес константы - PC)/4
...
; Пул констант
...
DCD 0xE000E400
...
```

Или, при использовании набора команд Thumb-2:

```
LDR.W R0, [PC, #+/- <offset_12>]
; offset_12 = адрес константы - PC
...
; Пул констант
...
DCD 0xE000E400
...
```

Поскольку в программах, как правило, используется более одной константы, ассемблер или компилятор генерируют целый блок констант, называемый обычно *пулом констант* (literal pool).

В процессоре Cortex-M3 операции загрузки констант представляют собой операции чтения, осуществляемые по шине данных (D-Code или системной шине, в зависимости от положения константы в памяти).

16.7. Порт доступа шины AHB

Порт доступа усовершенствованной высокопроизводительной шины (AHB-AP) представляет собой мост между модулем интерфейса отладки (SWJ-DP или SW-DP) и системой памяти процессора Cortex-M3 (**Рис. 16.8**). В большинстве операций пересылки данных между хостом отладки и системой с процессором Cortex-M3 используются следующие регистры модуля AHB-AP:

- регистр слова состояния и управления (CSW);
- регистр адреса пересылки (TAR);
- регистр чтения/записи данных (DRW).

Регистр CSW позволяет управлять направлением передачи данных (чтение/запись), размером пересылки, типом пересылки и т.п. Регистр TAR используется для указания адреса пересылки, а регистр DRW предназначен для выполнения собственно операции пересылки данных (пересылка начинается при обращении к этому регистру).

Содержимое регистра данных DRW в точности соответствует содержимому шины. При выполнении двухбайтных и однобайтных пересылок требуемые данные будут сдвинуты в корректный байтовый тракт программным обеспечением отладчика.

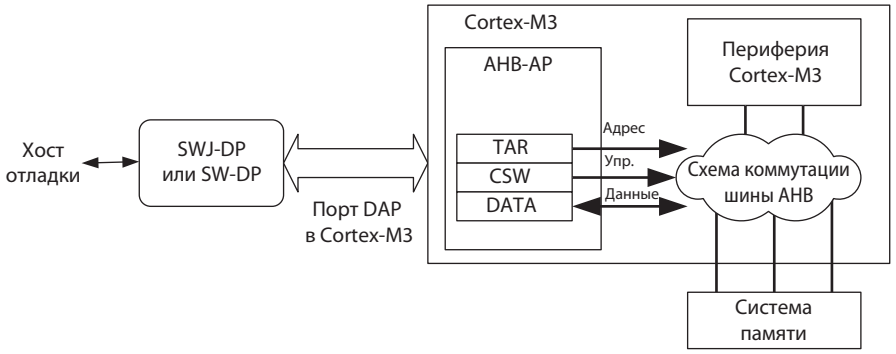


Рис. 16.8. *Подключение порта AHB-AP в процессоре Cortex-M3.*

Например, если вы хотите переслать полуслово по адресу 0x1002, то вы должны поместить данные в биты [31:16] регистра DRW. Модуль AHB-AP может формировать невыровненные пересылки, однако результат операции при этом не выравнивается. Так что программа отладчика должна будет либо сдвинуть данные самостоятельно, либо разбить одно обращение к невыровненным данным на несколько операций.

Остальные регистры AHB-AP обеспечивают дополнительные функциональные возможности. В частности, модуль имеет 4 регистра, объединённых в банк, которые в сочетании с функцией автоматического инкрементирования адреса ускоряют обращения к памяти в некотором диапазоне адресов и выполнение последовательных пересылок. В модуле AHB-AP также имеется специальный регистр, содержащий адрес таблицы ПЗУ.

В регистре CSW находится бит Master Type. Этот бит обычно устанавливается в 1, говоря о том, что пересылка, полученная устройством от моста AHB-AP, была запущена отладчиком. Однако отладчик может «симулировать» ядро процессора, сбросив указанный бит. В этом случае устройство на шине AHB будет функционировать так, как если бы к нему обращался процессор. Данная возможность облегчает тестирование периферийных устройств с FIFO-буфером, которые в присутствии отладчика могут работать иначе, нежели без него.

16.8. Таблица ПЗУ

Таблица ПЗУ используется для того, чтобы отладчик мог автоматически обнаружить компоненты отладки, имеющиеся в устройстве с процессором Cortex-M3. Как уже неоднократно говорилось, данный процессор является первым процессором компании ARM, имеющим архитектуру ARM v7-M. Он содержит предопределённую карту памяти и ряд компонентов отладки. Однако в будущих устройствах семейства Cortex-M расположение компонентов отладки в адресном пространстве может измениться. Это же может произойти в том случае, если разработчики микросхемы модифицируют компоненты отладки, используемые по умолчанию. Для того чтобы отладочные средства могли обнаружить компоненты, имеющиеся в системе отладки, и предназначена таблица ПЗУ, которая содержит информацию об адресах контроллера NVIC и модулей отладки.

Таблица ПЗУ располагается по адресу 0xE00FF000. Содержимое данной таблицы позволяет вычислить расположение в памяти системных компонентов и компонентов отладки. После этого отладчик может проверить регистры идентификатора обнаруженных компонентов и определить, какие именно компоненты имеются в системе.

Для процессора Cortex-M3 первый элемент таблицы ПЗУ (0xE00FF000) должен содержать смещение для регистров контроллера NVIC. По умолчанию значение первого элемента таблицы равно 0xFFF0F003; установленные биты [1:0] свидетельствуют о том, что устройство существует и что в таблице имеются и другие элементы. Адрес блока регистров контроллера NVIC может быть вычислен как 0xE00FF000 + 0xFFF0F000 = 0xE000E000.

Содержимое таблицы ПЗУ процессора Cortex-M3, принятое по умолчанию, приведено в **Табл. 16.2**. Однако в связи с тем, что разработчики микроконтроллеров могут добавлять, исключать или замещать определённые опциональные компоненты отладки другими CoreSight-совместимыми компонентами, содержимое таблицы применяемого вами микроконтроллера может оказаться другим.

***Таблица 16.2.* Содержимое таблицы ПЗУ Cortex-M3, принятое по умолчанию**

Адрес	Значение	Обозначение	Описание
0xE00FF000	0xFFF0F003	NVIC	Базовый адрес контроллера NVIC (0xE000E000)
0xE00FF004	0xFFF02003	DWT	Базовый адрес модуля DWT (0xE0001000)
0xE00FF008	0xFFF03003	FPB	Базовый адрес модуля FPB (0xE0002000)
0xE00FF00C	0xFFF01003	ITM	Базовый адрес модуля ITM (0xE0000000)
0xE00FF010	0xFFF41003/ 0xFFF41002	TPIU	Базовый адрес модуля TPIU (0xE0040000)
0xE00FF014	0xFFF42003/ 0xFFF42002	ETM	Базовый адрес модуля ETM (0xE0041000)
0xE00FF018	0	End	Маркёр конца таблицы
0xE00FFFCC	0x1	MEMTYPE	Признак того, что остальные устройства расположены в этом же адресном пространстве
0xE00FFFD0	0 / 0x04	PID4	Область идентификаторов периферии; зарезервировано
0xE00FFFD4	0 / 0x00	PID5	Область идентификаторов периферии; зарезервировано
0xE00FFFD8	0 / 0x00	PID6	Область идентификаторов периферии; зарезервировано
0xE00FFFDC	0 / 0x00	PID7	Область идентификаторов периферии; зарезервировано
0xE00FFFE0	0 / 0xC3	PID0	Область идентификаторов периферии; зарезервировано
0xE00FFFE4	0 / 0xB4	PID1	Область идентификаторов периферии; зарезервировано
0xE00FFFE8	0 / 0x0B	PID2	Область идентификаторов периферии; зарезервировано
0xE00FFFEC	0 / 0x00	PID3	Область идентификаторов периферии; зарезервировано
0xE00FFFF0	0 / 0x0D	CID0	Область идентификаторов компонентов; зарезервировано
0xE00FFFF4	0 / 0x10	CID1	Область идентификаторов компонентов; зарезервировано
0xE00FFFF8	0 / 0x05	CID2	Область идентификаторов компонентов; зарезервировано
0xE00FFFFC	0 / 0xB1	CID3	Область идентификаторов компонентов; зарезервировано

Младшие два бита значения каждого элемента используются для указания наличия компонента в системе. В любом устройстве обязательно должны присутствовать модули NVIC, DWT и FPB, поэтому два младших бита значений таблицы, соответствующих этим компонентам, всегда будут установлены в 1. В то же время модули TPIU и ETM могут быть заменены производителем на другие компоненты отладки семейства CoreSight.

В старших битах значений таблицы содержатся величины смещений компонентов относительно адреса начала таблицы ПЗУ. Например:

```
Адрес NVIC = 0xE00FF000 + 0xFFF0F000 = 0xE000E000 (усечено до 32 бит)
```

Средства отладки, поддерживающие технологию CoreSight, должны уметь определять адреса компонентов отладки из таблицы ПЗУ. В некоторых устройствах с ядром Cortex-M3 могут присутствовать другие компоненты, что приводит к появлению новых элементов в таблице ПЗУ. Используя содержимое таблицы, отладчик может вычислить базовые адреса всех компонентов отладки. После этого, читая содержимое регистров идентификаторов указанных компонентов, отладчик может определить, какие именно компоненты присутствуют в системе (**Рис. 16.9**).

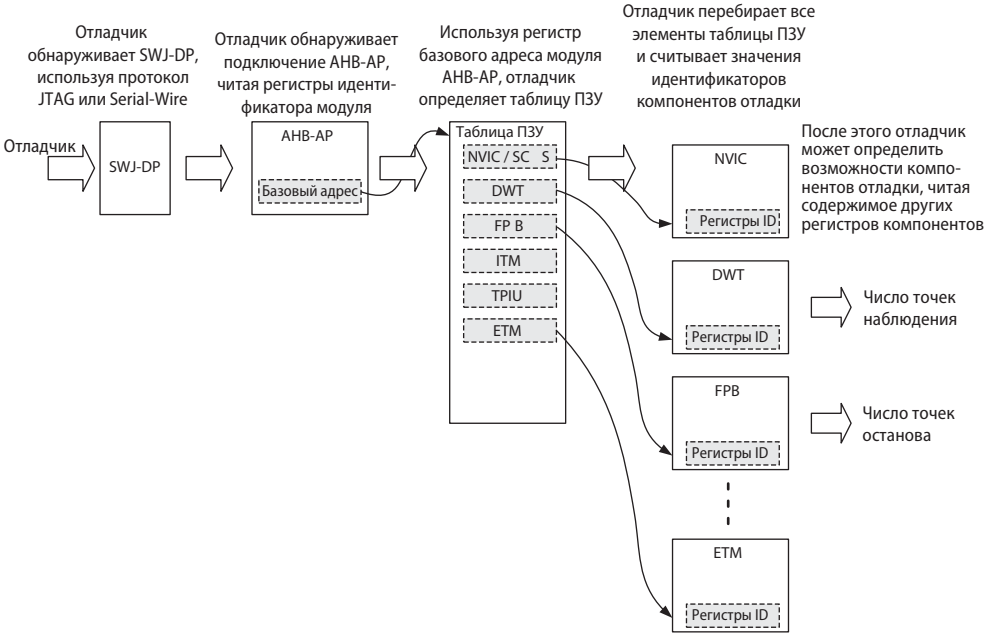


Рис. 16.9. *Автоматическое обнаружение компонентов в архитектуре CoreSight.*

ГЛАВА 17
ПРИСТУПАЯ К РАБОТЕ С ПРОЦЕССОРОМ CORTEX-M3

17.1. Выбор устройства с ядром Cortex-M3

Устройства, построенные на базе процессора Cortex-M3, отличаются друг от друга не только объёмами памяти, имеющимися периферийными устройствами и быстродействием, но и множеством других параметров. Процессор Cortex-M3, предлагаемый компанией ARM, имеет ряд конфигурируемых параметров:

- число внешних прерываний;
- число уровней приоритета прерываний (разрядность регистров уровня приоритета);
- наличие или отсутствие модуля защиты памяти MPU;
- наличие или отсутствие модуля трассировки ETM;
- используемый интерфейс отладки (Serial-Wire, JTAG или оба).

Очевидно, что выбор конкретного устройства в большинстве проектов определяется функциональными возможностями и параметрами того или иного микроконтроллера. Например:

- *Периферия.* Для многих приложений наличие того или иного периферийного устройства является основным критерием. Большее число периферийных устройств можно считать достоинством, однако это также влияет на потребление микроконтроллера и его стоимость.
- *Память.* Микроконтроллеры с ядром Cortex-M3 могут иметь флэш-память объёмом от нескольких килобайт до нескольких мегабайт. Также немаловажен объём внутреннего ОЗУ устройства. Как правило, эти факторы напрямую влияют на стоимость микроконтроллера.
- *Тактовая частота.* Процессор Cortex-M3 компании ARM может легко работать на частоте 100 МГц, даже при его изготовлении по 0.18-мкм технологии. Однако производители устройств на базе данного процессора могут указывать меньшие значения максимальной тактовой частоты из-за ограниченного быстродействия используемой памяти.
- *Посадочное место.* Микроконтроллеры с ядром Cortex-M3 могут выпускаться в самых разных корпусах — это зависит исключительно от политики производителя. Многие устройства имеют корпуса с малым числом выводов, что делает их идеальным выбором при создании недорогих изделий.

На сегодняшний день многие производители микроконтроллеров уже активно продвигают свои решения на базе процессора Cortex-M3, и ещё ряд производителей намереваются выпустить такие микроконтроллеры в ближайшее время. Перечислим наиболее заметных игроков на этом рынке.

Texas Instruments (бывшая Luminary Micro). В семейство Stellaris микроконтроллеров с ядром Cortex-M3 входит более 100 различных устройств, включая устройства с контроллерами Ethernet 10/100 MAC- и PHY-уровней, интерфейсами USB, CAN, SPI, I^2C, I^2S и т.п.

STMicroelectronics. Компания выпускает три линейки микроконтроллеров с ядром Cortex-M3:

- STM32 (Connectivity Line) — в рамках этой линейки выпускаются устройства, имеющие максимальную функциональность и поддержку интерфейсов USB On-The-Go (USB OTG), Ethernet, а также интерфейсов карт памяти.
- STM32L (Low Power) — микроконтроллеры данной линейки предназначены для приложений со сверхнизким потреблением и имеют встроенную поддержку интерфейса ЖКИ.
- STM32 (Value Line) — микроконтроллеры для бюджетных приложений.

Toshiba. Микроконтроллеры семейства TX03 предназначены для использования в самых разных областях, в том числе в промышленном оборудовании, автомобильных устройствах и бытовой аппаратуре. Эти микроконтроллеры поддерживают различные интерфейсы, включая USB, CAN, Ethernet; ряд моделей имеют в своём составе аналоговую периферию.

Atmel. Микроконтроллеры семейства SAM3U используют двухбанковую флэш-память, оснащены высокоскоростными интерфейсами High Speed USB, SPI, SDIO, SSC, имеют интерфейс карт памяти, а также АЦП.

Energy Micro. Микроконтроллеры семейства EFM®32 являются чрезвычайно энергоэффективными устройствами с инновационной периферией, которая может реагировать на внешние воздействия без участия ЦПУ. Помимо всего прочего, эти микроконтроллеры имеют интерфейс ЖКИ, модули АЦП/ЦАП, а также ряд специальных возможностей наподобие поддержки стандарта шифрования AES.

NXP. Компания выпускает две линейки микроконтроллеров с ядром Cortex-M3 — LPC1700 и LPC1300. Первая из них рассчитана на создание высокопроизводительных приложений и обеспечивает поддержку высокоскоростных коммуникационных интерфейсов, различных методов управления двигателями, а также стандартных промышленных интерфейсов, таких как USB, CAN, I^2S. Линейка LPC1300, наоборот, нацелена на устройства с малым потреблением, а также на устройства обработки смешанных сигналов.

17.2. Средства разработки

Для работы с микроконтроллерами на базе процессора Cortex-M3 вам потребуется определённый инструментарий. Обычно в него входят следующие компоненты:

- *Компилятор и/или ассемблер* — программное обеспечение для компиляции ваших программ, написанных на языке Си или на ассемблере. Практически все компиляторы поставляются вместе с ассемблером.
- *Симулятор* — программное обеспечение, симулирующее исполнение команд; применяется для отладки программы на первых этапах разработки приложения. Этот компонент необязателен.
- *Внутрисхемный эмулятор (ICE) или отладчик* — устройство, позволяющее подключать хост отладки, в качестве которого обычно выступает персональный компьютер, к разрабатываемому устройству. Может использоваться интерфейс JTAG или Serial-Wire.
- *Отладочная плата* — печатная плата с установленным на ней микроконтроллером.
- *Трассировщик* — опциональный программно-аппаратный комплекс для трассировки команд или считывания выходных данных модулей DWT и ITM и преобразования их в удобочитаемую форму. Иногда функции трассировщика встраиваются во внутрисхемный эмулятор.
- *Встраиваемая операционная система* — операционная система (ОС), работающая на микроконтроллере. Этот компонент необязателен — существует множество приложений, не требующих наличия ОС.

17.2.1. Си-компиляторы и отладчики

В настоящее время на рынке представлено довольно много Си-компиляторов и средств разработки, имеющих поддержку микроконтроллеров с процессором Cortex-M3 (**Табл. 17.1**).

Таблица 17.1. **Некоторые средства разработки с поддержкой процессора Cortex-M3**

Компания	Продукция*
ARM (www.arm.com)	Пакет разработки RealView Development Suite (RVDS) и внутрисхемный эмулятор RealView-ICE (RVI). Обратите внимание, что старые продукты, такие как ARM Development Suite (ADS) и Software Development Toolkit (SDT), не поддерживают процессор Cortex-M3
Keil, an ARM company (www.keil.com)	Пакет разработки Microcontroller Development Kit (MDK-ARM). Внутрисхемные отладчики и трассировщик семейства ULINK™
CodeSourcery (www.codesourcery.com)	Пакет разработки SourceryG++, основанный на инструментарии GNU Toolchain (www.codesourcery.com/gnu_toolchains/arm/)
Rowley Associates (www.rowley.co.uk)	Среда разработки CrossWorks for ARM, основанная на инструментарии GNU Toolchain (www.rowley.co.uk/arm/index.htm)
IAR Systems (www.iar.com)	Среда разработки IAR Embedded Workbench for ARM версии 4.40. Также предлагается стартовый комплект, в который входит отладочная плата с микроконтроллером LM3S102 (Texas Instruments) и внутрисхемный отладчик J-Link
Lauterbach (www.lauterbach.com)	Программно-аппаратные средства для отладки и трассировки

Таблица 17.1. **Некоторые средства разработки с поддержкой процессора Cortex-M3** *(продолжение)*

Компания	Продукция*
Segger (www.segger.com)	Внутрисхемный отладчик J-Link и трассировщик J-Trace
Signum (www.signum.com)	Внутрисхемные отладчики JTAGJet и JTAGJet-Trace для процессора Cortex-M3
Code Red (www.code-red-tech.com)	Интегрированная среда разработки Red Suite™ 2 на базе Eclipse, использующая компилятор GCC и отладчик Red Probe с поддержкой протоколов JTAG и SW
National Instrument (www.ni.com)	Модуль LabVIEW Embedded Module for ARM Microcontroller
Raisonance (www.raisonance.com)	Пакет программ RKit-ARM на базе компилятора GCC. Также имеется отладчик (RLink) и стартовый комплект (STM32 Primer)
GNU GCC (www.gcc.gnu.org)	Последние версии компилятора GCC поддерживают процессор Cortex-M3
*Названия продуктов являются зарегистрированными товарными знаками соответствующих компаний, указанных в левом столбце таблицы.	

Бесплатный компилятор Си/Си++ из набора GCC можно загрузить с сайта проекта GNU, а также с сайтов ряда компаний, например CodeSourcery и Raisonance. Некоторые коммерческие пакеты, такие как пакет MDK-ARM компании Keil, имеют полнофункциональные пробные версии.

17.2.2. Поддержка встраиваемых ОС

Многие приложения требуют операционной системы для поддержки многопоточности и управления ресурсами. В настоящее время на рынке предлагается множество встраиваемых ОС, некоторые из которых уже портированы на процессор Cortex-M3 (**Табл. 17.2**).

Таблица 17.2. **Встраиваемые ОС, портированные на процессор Cortex-M3**

Компания	Продукция*
FreeRTOS (www.freertos.org)	FreeRTOS
Express Logic (www.rtos.com)	ThreadX(TM) RTOS
Micrium (www.micrium.com)	µC/OS-II
Mentor Graphics (www.mentor.com)	Nucleus/Nucleus Plus
Pumpkin Inc. (www.pumpkininc.com)	Salvo RTOS
CMX Systems (www.cmx.com)	CMX-RTX
Keil (www.keil.com)	ARM RTX
Segger (www.segger.com)	emboss
IAR Systems (www.iar.com)	IAR PowerPac for ARM
eCosCentric (www.ecoscentric.com, www.ecos.sourceware.org)	eCos
Interniche Technologies Inc. (www.nichetask.com)	NicheTask
Green Hills Software (www.ghs.com)	µVelOSity

Таблица 17.2. **Встраиваемые ОС, портированные на процессор Cortex-M3** *(продолжение)*

Компания	Продукция*
Open source (www.linux-arm.org/LinuxKernel/LinuxM3)	μCLinux
Quadros System (www.quadros.com)	RTXC
ENEA (www.enea.com)	OSE Epsilon RTOS
Raisonance (www.stm32circle.com/projects/circleos.php)	CircleOS
SCIOPTA (www.sciopta.com)	SCIOPTA RTOS
Micro Digital (www.smxrtos.com)	SMX RTOS
*Названия продукции являются зарегистрированными товарными знаками соответствующих компаний, указанных в левом столбце таблицы.	

17.3. Различия между процессорами Cortex-M3 ревизий 0 и 1

В самых первых устройствах использовались процессоры ревизии 0. Изделия на базе процессора Cortex-M3 ревизии 1 появились на рынке в первом квартале 2006 года и на момент опубликования первого издания книги эта ревизия ядра использовалась во всех новых микроконтроллерах с процессором Cortex-M3. Ревизия 2 процессора появилась в 2008 году, а устройства с данным ядром начали выпускаться в 2009 году. Желательно точно знать номер ревизии процессора, реализованной в используемом вами микроконтроллере, поскольку эти ревизии существенно отличаются друг от друга.

В ревизии 1 процессора была немного изменена модель программирования и добавлены некоторые возможности:

- Начиная с ревизии 1, появилась возможность выравнивания стекового фрейма обработчика исключения на границу двойного слова. Для этого в регистр CCR контроллера NVIC был добавлен бит STKALIGN.
- В редакции r1p1 ревизии появился новый регистр состояния AUXFAULT (опциональный).
- В модуль DWT были добавлены новые возможности, включая контроль совпадения значений данных.
- Поскольку сменился номер ревизии, то было изменено и значение регистра идентификатора процессора.

Изменений, незаметных конечным пользователям, было намного больше:

- Области памяти кода были жёстко назначены следующие атрибуты: кэшируемая, выделяемая, небуферируемая и неразделяемая. Это нововведение затронуло интерфейсы шин AHB I-Code и D-Code, но не интерфейс системной шины. Изменения коснулись лишь возможности кэширования и буферизации памяти, расположенной вне процессора (например, кэш-памяти 2-го уровня или контроллеров памяти с кэшем). Функционирование внутреннего буфера записи процессора не изменилось, поэтому в большинстве микроконтроллеров данное новшество ни на что не повлияло.

- Появилась возможность мультиплексирования шин I-Code и D-Code. В этом режиме указанные шины можно было объединять с помощью простого мультиплексора шин (в предыдущем решении использовался компонент BusMatrix из набора ADK). Это позволило уменьшить общее число логических элементов.
- Был добавлен новый выходной порт для подключения к модулю AHB Trace Macrocell (HTM, компонент отладки архитектуры CoreSight от ARM).
- Появилась возможность обращения к компонентам отладки и регистрам управления отладкой даже во время системного сброса; эти регистры стали недоступными только во время сброса по включению питания.
- В модуль TPIU была добавлена поддержка режима работы Serial-Wire Viewer (SWV). Это позволило использовать для сбора трассировочной информации недорогое оборудование.
- Бит C_MASKINTS регистра DHCSR контроллера NVIC получил возможность влиять на содержимое поля VECTPENDING регистра ICSR. Теперь, если при установке бита C_MASKINTS маскировалось отложенное прерывание, то поле VECTPENDING обнуляется.
- Модуль интерфейса отладки JTAG-DP был заменён на модуль SWJ-DP (см. подраздел 17.3.1). Однако разработчики микроконтроллеров могли продолжать использовать модуль JTAG-DP, поскольку он остался в линейке продукции архитектуры CoreSight.

Поскольку ревизия 0 процессора не поддерживала выравнивание стека на границу двойного слова при обработке исключений, то в некоторых средах разработки, в частности в пакете RVDS компании ARM и пакете MDK-ARM компании Keil, были предусмотрены специальные опции для программного выравнивания стека, что обеспечивало совместимость разрабатываемого приложения с интерфейсом EABI. Это могло оказаться важным при необходимости работы с другими EABI-совместимыми средствами разработки.

Для определения номера ревизии процессора, реализованного в микроконтроллере или SoC, можно воспользоваться регистром идентификатора ЦПУ CPUID контроллера NVIC. Зависимость содержимого полей указанного регистра от номера ревизии и номера редакции ядра приведена в **Табл. 17.3**.

***Таблица 17.3.* Регистр идентификатора ЦПУ CPUID (0xE000ED00)**

	Implementer [31:24]	**Variant [23:20]**	**Constant [19:16]**	**Part No [15:4]**	**Revision [3:0]**
Ревизия 0 (r0p0)	0x41	0x0	0xF	0xC23	0x0
Ревизия 1 (r1p0)	0x41	0x0	0xF	0xC23	0x1
Ревизия 1 (r1p1)	0x41	0x1	0xF	0xC23	0x1
Ревизия 2 (r2p0)	0x41	0x2	0xF	0xC23	0x0

Все компоненты отладки процессора Cortex-M3 тоже имеют регистры идентификатора, содержимое которых может быть различным для ревизий 0 и 1 процессора.

17.3.1. Ревизия 1 — замена модуля JTAG-DP на SWJ-DP

На смену модулю JTAG-DP, использовавшемуся в ранних устройствах на базе процессора Cortex-M3, пришёл модуль SWJ-DP. Этот модуль обладает функциональностью модулей SW-DP и JTAG-DP и может автоматически определять используемый протокол (**Рис. 17.1**). При наличии данного компонента устройства с процессором Cortex-M3 могут поддерживать отладку как по интерфейсу Serial-Wire, так и по интерфейсу JTAG.

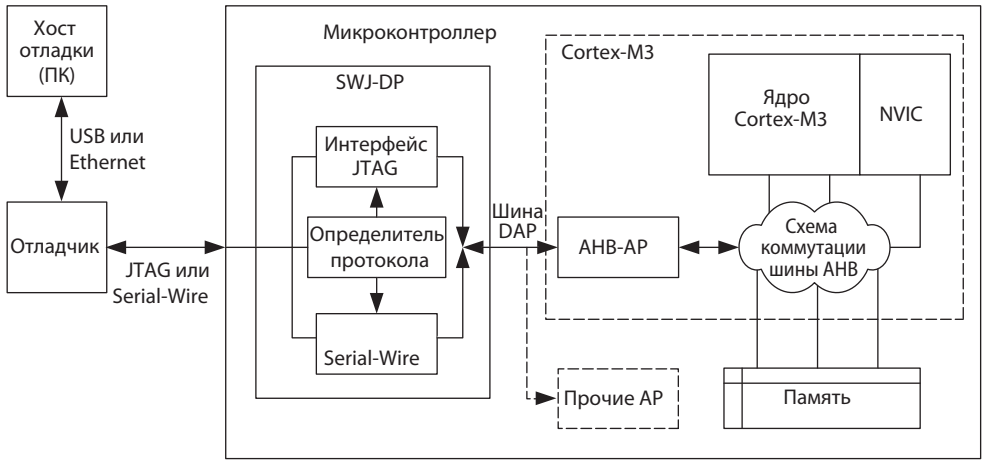


Рис. 17.1. *Модуль SWJ-DP — JTAG-DP и SW_DP «в одном флаконе».*

17.4. Различия между процессорами Cortex-M3 ревизий 1 и 2

В середине 2008 года компания ARM представила новую ревизию процессора Cortex-M3 — ревизию 2 (r2p0). Устройства, использующие данную ревизию процессора, начали появляться на рынке в 2009 году. В ревизии 2 процессора появились новые возможности, бóльшая часть которых нацелена на уменьшение потребления и увеличение гибкости процесса отладки. Изменения модели программирования, заметные конечному пользователю, описаны в следующих подразделах.

17.4.1. Выравнивание стека на границу двойного слова по умолчанию

По умолчанию в новой ревизии включено выравнивание стека на границу двойного слова при сохранении контекста (хотя изготовители микросхем могут и не использовать эту возможность). Такое поведение процессора позволяет уменьшить объём стартового кода для большинства приложений, написанных на Си (устраняется необходимость установки бита STKALIGN в регистре CCR).

17.4.2. Дополнительный регистр управления

В контроллере NVIC появился дополнительный регистр управления ACR, позволяющий более точно управлять функционированием процессора. Например, для целей отладки можно отключить буферы записи процессора, в результате чего отказы шины станут синхронными с командами обращения к памяти (точными). Это позволит легко определять команду, вызвавшую отказ, по значению адреса возврата, сохранённому в стеке. Формат регистра ACR приведён в **Табл. 17.4**.

Таблица 17.4. **Дополнительный регистр управления ACR (0xE000E008)**

Биты	Обозначение	Тип	Значение после сброса	Описание
2	DISFOLD	R/W	0	Запрещает «схлопывание» команды `IT` (предотвращает перекрытие исполнения команды `IT` со следующей командой)
1	DISDEFWBUF	R/W	0	Запрещает использование буфера записи для карты памяти, используемой по умолчанию (не влияет на обращения к областям памяти, определённым в модуле MPU)
0	DISMCYCINT	R/W	0	Запрещает прерывание команд, выполняющихся за несколько тактов, таких как команды групповой загрузки/сохранения (`LDM`/`STM`) и команды 64-битного умножения и деления

17.4.3. Новое значение регистров идентификации

Были изменены различные регистры идентификации контроллера NVIC и компонентов отладки. Так, значение регистра CPUID контроллера NVIC стало равным 0x412FC230 (см. **Табл. 17.3**).

17.4.4. Возможности отладки

Для облегчения процесса отладки в ревизии 2 процессора появились следующие улучшения:

- При трассировке данных, запускаемой точкой наблюдения в модуле DWT, теперь можно выбирать для трассировки отдельно пересылки записи и отдельно пересылки чтения, что позволяет уменьшить поток трассировочной информации.
- Бóльшая гибкость в реализации функций отладки. Например, можно уменьшить число поддерживаемых точек останова и точек наблюдения для уменьшения размеров кристалла, снизив тем самым энергопотребление устройства.
- Улучшенная поддержка отладки многопроцессорных устройств. Был добавлен новый интерфейс, позволяющий одновременно перезапускать и использовать в пошаговом режиме несколько процессоров (совершенно прозрачно для программиста).

17.4.5. Особенности режима пониженного энергопотребления

Также были улучшены имеющиеся возможности режима пониженного энергопотребления (**Рис. 17.2**). В ревизии 2 появилась возможность задержки момента выхода микроконтроллера из спящего режима. Это позволяет отключать большее число узлов кристалла; выполнение программы может возобновляться системой управления электропитанием при готовности устройства. Такой алгоритм работы необходим для микроконтроллеров, в которых имеются узлы, отключающиеся при переходе в спящий режим, поскольку для стабилизации напряжения питания после его восстановления может потребоваться некоторое время.

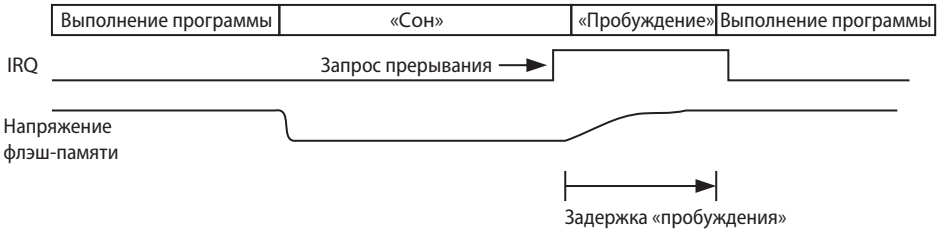


Рис. 17.2. *Задержанный выход из спящего режима (ревизия 2 процессора).*

Также в процессор были внедрены новые технологии, позволяющие ещё больше снизить потребление микроконтроллера. Предыдущие ревизии процессора Cortex-M3 требовали наличия независимого тактового сигнала для выхода из спящего режима по прерыванию.

Для устранения данного недостатка к процессору был подключён простой контроллер прерываний. Этот контроллер (WIC) дублирует функции маскирования прерываний контроллера NVIC при нахождении процессора в режиме Deep Sleep и извещает систему управления электропитанием о необходимости выхода из спящего режима. Наличие указанного контроллера позволяет полностью отключить все тактовые сигналы, поступающие в процессор (**Рис. 17.3**).

Помимо останова всех тактовых сигналов, данное решение позволяет отключать бóльшую часть узлов процессора с сохранением его состояния в специальных логических ячейках. При возникновении прерывания контроллер WIC формирует для модуля PMU запрос на включение системы. После подачи питания на процессор его предыдущее состояние восстанавливается, и он приступает к обработке прерывания.

Эти возможности позволяют уменьшить энергопотребление устройства в спящем режиме. Однако их реализация зависит от техпроцесса, по которому было изготовлено устройство. То есть описанные возможности могут быть доступны не во всех изделиях, несмотря на использование процессора ревизии 2.

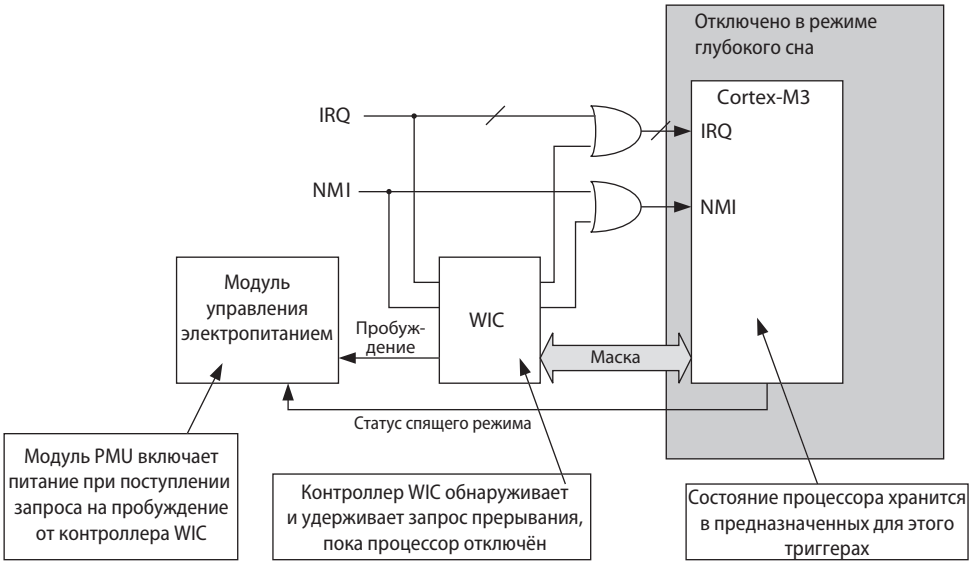


Рис. 17.3. *Контроллер WIC (ревизия 2 процессора).*

17.5. Чем же хороша ревизия 2 процессора Cortex-M3?

И всё же, как влияют все эти новые возможности на процесс разработки встраиваемых устройств?

Во-первых, они позволяют уменьшить потребление встраиваемых устройств и, соответственно, увеличить время работы от батарей. Когда контроллер WIC находится в режиме Deep Sleep, активной должна оставаться лишь малая часть узлов схемы. Кроме того, при реализации микроконтроллеров с чрезвычайно низким энергопотреблением разработчики могут уменьшить площадь кристалла за счёт сокращения числа доступных точек останова и точек наблюдения.

Во-вторых, новые возможности обеспечивают бóльшую гибкость при отладке и устранении неполадок. Помимо более развитых функций трассировки данных, которые могут использоваться отладчиком, в нашем распоряжении оказывается дополнительный регистр управления. С помощью указанного регистра мы можем отключать буферизацию операций записи, получая в итоге возможность отслеживать ошибочные команды или же запрещать прерывания на время исполнения команд, выполняющихся за несколько тактов, с тем чтобы все команды групповой загрузки/сохранения завершались перед обработкой исключения. Последнее весьма облегчает анализ содержимого памяти. Для многопроцессорных систем ревизия 2 процессора даёт возможность одновременно перезапускать и отлаживать в пошаговом режиме несколько ядер.

Кроме того, при разработке ревизии 2 процессора были проведены определённые работы по оптимизации ядра, которые обеспечили более высокую производительность и лучшие интерфейсные возможности. Это позволяет производите-

лям микроконтроллеров создавать более быстрые устройства на базе процессора Cortex-M3, имеющие более богатые функциональные возможности. Тем не менее, программисты встраиваемых систем должны обязательно учитывать следующее:

1. *Выравнивание стекового фрейма исключения на границу двойного слова.* В новой ревизии процессора стековые фреймы исключений по умолчанию выравниваются на границу двойного слова. Это может привести к некорректной работе программ, написанных на ассемблере для процессоров ревизий 0 или 1 и использующих стек для передачи параметров в обработчики исключений. Поэтому обработчики исключений должны сначала определять, было ли осуществлено выравнивание стека, проверяя бит 9 сохранённого в стеке регистра PSR. Затем они могут определить адреса данных, помещённых в стек до возникновения исключения. Или же можно просто сбросить в программе бит STKALIGN в 0 и получить в результате такое же поведение стека, как и в предыдущих ревизиях процессора. На работу приложений, удовлетворяющих требованиям стандарта EABI, т.е. написанным на языке Си и скомпилированным EABI-совместимым компилятором, это нововведение не влияет.
2. *Возможность останова таймера SYSTICK в режиме Deep Sleep.* Если микроконтроллер с процессором Cortex-M3 допускает отключение различных узлов в спящем режиме или же обеспечивает останов всех тактовых сигналов ядра при переводе его в режим Deep Sleep, то существует вероятность того, что таймер SYSTICK тоже не будет работать в этом спящем режиме. В таком случае встраиваемые приложения, использующие ОС, потребуют наличия внешнего (относительно ядра процессора) таймера.
3. *Функции отладки и отключения питания.* Новые возможности отключения узлов процессора блокируются при подключении процессора к отладчику. Это связано с тем, что во время отладочной сессии отладчик должен иметь возможность обращаться к отладочным регистрам процессора. При этом ядро должно сохранять возможность останова или перехода в спящий режим, однако не должно активировать цикл отключения питания даже в том случае, если данная функция разрешена. Для проверки работы функции отключения питания отлаживаемое устройство необходимо отключить от отладчика.

17.6. Различия между процессорами Cortex-M3 и Cortex-M0

Наверняка кто-то из вас уже слышал о процессоре Cortex-M0. Модель программирования этого процессора очень похожа на модель Cortex-M3. Однако он занимает меньшую площадь на кристалле, поддерживает меньшее число команд и имеет фон-неймановскую архитектуру. Процессор Cortex-M0 был разработан специально для использования в устройствах со сверхмалым потреблением, где число логических вентилей является критическим фактором. В минимальной конфигурации процессор Cortex-M3 содержит всего 12 000 логических вентилей,

что меньше, чем в большинстве 16-битных процессоров и некоторых топовых 8-битных процессоров. Несмотря на это, процессор Cortex-M0 обеспечивает производительность 0.9 DMIPS/МГц, что в два раза превышает производительность большинства 16-битных микроконтроллеров и почти в 10 раз современных 8-битных микроконтроллеров. Это позволяет говорить о процессоре Cortex-M3 как о самом энергоэффективном процессоре для микроконтроллеров общего применения.

17.6.1. Модель программирования

Между моделями программирования процессоров Cortex-M3 и Cortex-M0 существует ряд существенных отличий (**Рис. 17.4**). Так, непривилегированный уровень доступа имеется только в процессоре Cortex-M3. Также в процессоре Cortex-M0 отсутствуют регистры специального назначения FAULTMASK и BASEPRI.

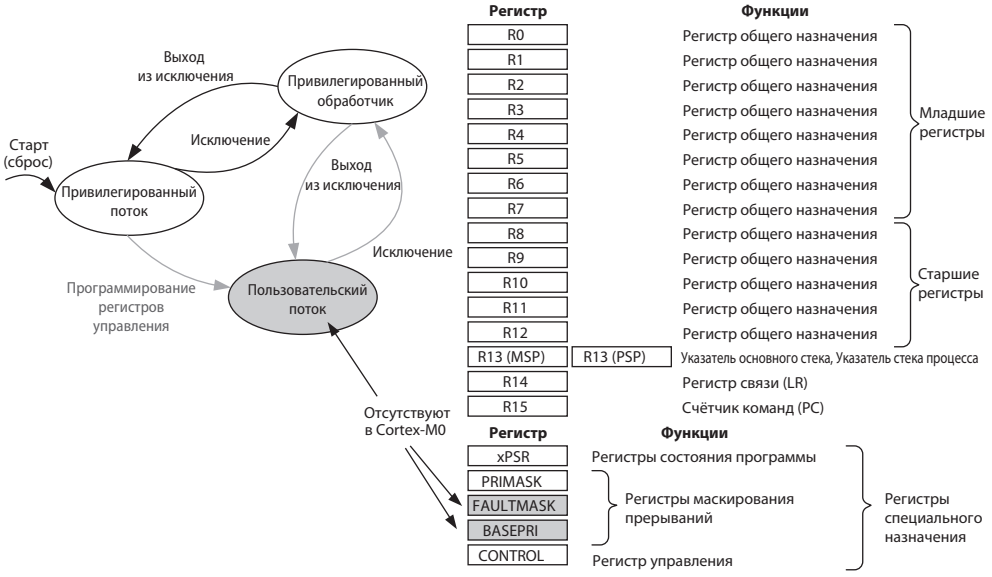


Рис. 17.4. *Различия в моделях программирования Cortex-M3 и Cortex-M0.*

Имеются некоторые различия и в формате регистров xPSR — в процессоре Cortex-M0 отсутствуют бит Q в регистре APSR и битовое поле ICI/IT в регистре EPSR. Это связано с тем, что процессор Cortex-M0 не поддерживает `IT`-блоки, а при прерывании команды групповой загрузки/сохранения её выполнение прекращается и начинается заново после завершения обработчика прерывания.

17.6.2. Исключения и контроллер NVIC

Обработка исключений в процессоре Cortex-M0 осуществляется так же, как и в процессоре Cortex-M3. Каждому прерыванию и исключению соответствует свой вектор, а поддержка вложенных исключений осуществляется контроллером NVIC автоматически. Некоторые из системных исключений процессора

Cortex-M3, а именно исключения Bus Fault, Usage Fault, MemManage Fault и Debug monitor в процессоре Cortex-M0 не реализованы. При возникновении системной ошибки в процессоре Cortex-M0 всегда запускается обработчик исключения Hard Fault. Соответственно, в процессоре Cortex-M0 отсутствует и ряд регистров состояния ошибок, имеющихся в Cortex-M3.

Регистры приоритета процессора Cortex-M0 имеют всего два бита. Соответственно, для прерываний и системных исключений с программируемым приоритетом доступно всего четыре уровня приоритета. Данный процессор не поддерживает динамическое изменение приоритетов, поэтому назначение приоритетов прерываний и исключений обычно производится в самом начале программы и после этого уже не изменяется. Модель программирования контроллера NVIC процессора Cortex-M0 во многом схожа с моделью в процессоре Cortex-M3. Основное отличие состоит в том, что регистры контроллера допускают только пословное обращение. Поэтому при необходимости изменить приоритет какого-либо прерывания придётся считать всё содержимое регистра, скорректировать уровень приоритета данного прерывания и записать полученное значение обратно. Также в контроллере NVIC процессора Cortex-M0 отсутствуют следующие регистры:

- *Регистр смещения таблицы векторов.* В процессоре Cortex-M0 положение таблицы векторов фиксировано. Однако микроконтроллеры могут воспользоваться функцией отображения памяти для изменения векторов исключений в процессе работы программы.
- *Регистр программного запуска прерывания.* Для программной генерации исключений используются регистры установки признака отложенного прерывания.
- *Регистр состояния активного прерывания.*
- *Регистр типа контроллера прерываний.*

17.6.3. Набор команд

Процессор Cortex-M0 базируется на архитектуре ARMv6-M. Он поддерживает 16-битные команды Thumb® и ряд 32-битных команд Thumb-2 (переход со ссылкой (`BL`), барьер синхронизации команд (`ISB`), барьер синхронизации данных (`DSB`), барьер памяти данных (`DMB`), `MRS` и `MSR`). Некоторые команды, имеющиеся в процессоре Cortex-M3, процессором Cortex-M0 не поддерживаются. Например:

- команда `IT`;
- команды сравнения и перехода (`CBZ` и `CBNZ`);
- команды умножения с накоплением (`MLA`, `MLS`, `SMLAL` и `UMLAL`), а также команды длинного умножения с 64-битным результатом (`UMULL` и `SMULL`);
- команды аппаратного деления (`UDIV`, `SDIV`) и насыщения (`SSAT`, `USAT`);
- команды табличных переходов (`TBH` и `TBB`);
- команды монопольного доступа;
- команды работы с битовыми полями (`UBFX`, `SBFX`, `BFI` и `BFC`);
- ряд команд обработки данных (`CLZ`, `RRX` и `RBIT`);
- команды загрузки и сохранения, использующие такие режимы адресации или комбинации регистров, которые допускаются только в 32-битных командах;
- команды загрузки/сохранения на непривилегированном уровне (`LDRT` и `STRT`).

17.6.4. Особенности системы памяти

Как и в процессоре Cortex-M3, карта памяти процессора Cortex-M0 разделена на несколько областей (CODE, SRAM, периферия и т.д.). Однако некоторые возможности системы памяти процессора Cortex-M3 в процессоре Cortex-M0 отсутствуют, а именно:

- области с побитовой адресацией;
- обращения к невыровненным данным;
- модуль защиты памяти;
- исключительный доступ.

17.6.5. Возможности отладки

Процессор Cortex-M0 не содержит никаких модулей поддержки трассировки (ETM или ITM). В сравнении с Cortex-M3 он поддерживает меньшее число точек останова и точек наблюдения (**Табл. 17.5**).

Таблица 17.5. **Сравнение возможностей отладки**

	Cortex-M0	**Cortex-M3**
Точки останова	до 4	до 8
Точки наблюдения	до 2	до 4

В большинстве случаев микроконтроллеры с процессором Cortex-M0 поддерживают только один протокол отладки (Serial-Wire или JTAG), в то время как устройства с процессором Cortex-M3 обычно поддерживают оба указанных протокола и способны динамически переключаться между ними.

17.6.6. Совместимость

Процессор Cortex-M0 обратно совместим с процессором Cortex-M3. Программы, скомпилированные для процессора Cortex-M0, могут выполняться процессором Cortex-M3. Однако программы, скомпилированные для Cortex-M3, не могут использоваться с процессором Cortex-M0 (**Рис. 17.5**).

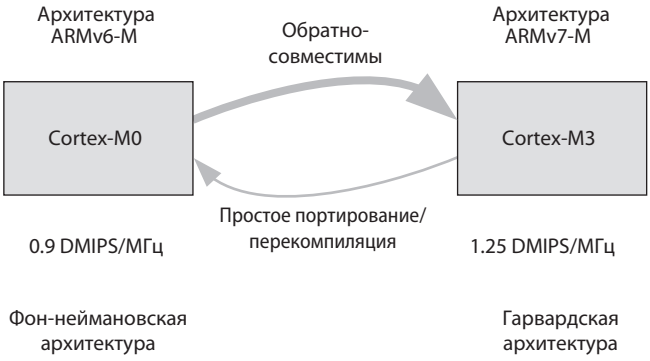


Рис. 17.5. *Совместимость процессоров Cortex-M3 и Cortex-M0.*

Из-за схожести обоих процессоров большинство программ можно написать так, что они смогут выполняться на любом из процессоров (это обеспечивается совместимостью карт памяти и периферийных устройств).

Если приложение должно выполняться как на микроконтроллерах с ядром Cortex-M3, так и на микроконтроллерах с ядром Cortex-M0, необходимо обратить внимание на ряд моментов:

1. Обращения к регистрам контроллера NVIC должны быть исключительно 32-битными или осуществляться с использованием соответствующих функций CMSIS-совместимой библиотеки.
2. Данные должны быть выровнены. Для отслеживания невыровненых пересылок в процессоре Cortex-M3 можно установить бит UNALIGN_TRP регистра CCR. В процессоре Cortex-M0 любая попытка обращения к невыровненным данным вызовет генерацию исключения Hard Fault.
3. Нельзя использовать метод bit-band в связи с тем, что в процессоре Cortex-M0 отсутствуют области с побитовой адресацией. В качестве альтернативного решения можно применить условную компиляцию кода, чтобы при компиляции программы для процессора Cortex-M0 все операции побитового доступа заменялись бы эквивалентными программными реализациями.

Совместимость процессоров Cortex-M3 и Cortex-M0 предоставляет много преимуществ разработчикам встраиваемых устройств. Помимо простоты переноса программного кода, она также позволяет отлаживать приложения для процессора Cortex-M0 на платформе Cortex-M3, которая располагает более широкими отладочными возможностями, в частности поддерживает трассировку команд и событий. Чтобы поведение Cortex-M3 стало ещё больше похожим на поведение Cortex-M0, мы можем, используя дополнительный регистр управления, запретить буферизацию записи. Необходимо только учитывать, что время исполнения некоторых команд в процессорах Cortex-M3 и Cortex-M0 может различаться.

ГЛАВА 18
ПЕРЕНОС ПРИЛОЖЕНИЙ С ПРОЦЕССОРА ARM7 НА ПРОЦЕССОР CORTEX-M3

18.1. Общие сведения

Для большинства разработчиков перенос существующего программного кода с одной платформы на другую является типовой задачей. А в связи с появлением на рынке устройств с ядром Cortex-M3 многие разработчики столкнулись с необходимостью переноса кода, написанного для процессора ARM7TDMI (далее в этой главе — ARM7), на процессор Cortex-M3. В данной главе обсуждаются некоторые вопросы, касающиеся переноса приложений с процессора ARM7 на процессор Cortex-M3.

При переносе кода с процессора ARM7 на процессор Cortex-M3 необходимо принимать во внимание следующее:

- особенности системы;
- исходные файлы на языке ассемблера;
- исходные файлы на языке Си;
- оптимизацию.

В общем случае, наибольших изменений требует код низкого уровня (драйверы аппаратуры, процедуры управления задачами и обработчики исключений). В то же время для переноса кода собственно приложения, как правило, достаточно внесения незначительных изменений и простой перекомпиляции.

18.2. Особенности системы

Системы, построенные на процессоре ARM7, имеют ряд отличий от систем на базе процессора Cortex-M3. В частности, у них различаются карты памяти, системы прерываний, модули MPU, управление системой и режимы работы.

18.2.1. Карта памяти

Прежде всего, внесение изменений в исходный код программы при её переносе с одного микроконтроллера на другой обусловлено различием их карт памяти. В процессоре ARM7 память и периферийные устройства могут располагаться практически по любым адресам, тогда как в процессоре Cortex-M3 используется предопределённая карта памяти. Различия в адресах обычно разрешаются на

этапе компиляции и компоновки. Перенос кода, связанного с каким-либо периферийным устройством, потребует гораздо бóльших усилий, поскольку модели программирования данного устройства могут оказаться совершенно разными. В этом случае драйвер устройства, скорее всего, придётся писать заново.

Многие микроконтроллеры с процессором ARM7 поддерживают *ремаппинг памяти*, что позволяет после запуска программы отобразить таблицу векторов в область статического ОЗУ. В процессоре Cortex-M3 таблица векторов может быть перемещена в ОЗУ с помощью одного из регистров контроллера NVIC, т.е. подобная переадресация уже не требуется. Соответственно, в большинстве устройств с процессором Cortex-M3 возможность переназначения памяти может отсутствовать.

Формат с обратным порядком байтов, используемый в процессоре ARM7, отличается от аналогичного формата в Cortex-M3. Чтобы привести программу в соответствие с новой системой хранения многобайтных значений, достаточно перекомпилировать её исходные файлы (при наличии в программе жёстко заданных таблиц соответствия может потребоваться их предварительное преобразование).

В процессоре ARM720T и некоторых более поздних процессорах, таких как ARM9, предусмотрена возможность размещения таблицы векторов по адресу 0xFFFF0000. Эта возможность предназначена для поддержки ОС Windows CE и в процессоре Cortex-M3 отсутствует.

18.2.2. Прерывания

Второй причиной необходимости внесения изменений являются различия между используемыми контроллерами прерываний. Придётся переписать код, отвечающий за управление контроллером, в частности разрешающий/запрещающий прерывания. Кроме того, необходимо будет добавить в этот код строки, отвечающие за конфигурирование уровней приоритета и адресов векторов для различных прерываний.

Также изменяется способ возврата из прерываний. В ассемблерных программах это потребует модификации кода возврата из прерывания, а при программировании на языке Си может потребоваться указание дополнительных директив компилятора.

Операции изменения регистра текущего состояния программы (CPSR), используемые в процессоре ARM7 для разрешения и запрещения прерываний, необходимо будет заменить на операции изменения регистров маскирования прерываний. Кроме того, процессор ARM7TDMI поддерживает повторное разрешение прерывания в момент возврата из него за счёт восстановления содержимого регистра CPSR из регистра SPSR. В процессоре же Cortex-M3 используется другой механизм: если прерывание было запрещено установкой регистра PRIMASK во время выполнения обработчика, то этот регистр должен быть очищен вручную перед выходом из обработчика. В противном случае, прерывания останутся запрещёнными.

В процессоре Cortex-M3 определённые регистры автоматически сохраняются в стеке при входе в прерывание и восстанавливаются из стека при выходе из пре-

рывания. Наличие этого механизма позволяет уменьшить объём программных манипуляций со стеком или вообще их исключить. Несколько особняком стоит быстрое прерывание (FIQ), для которого в традиционных процессорах ARM предусмотрены отдельные регистры R8…R11, что позволяет обработчику FIQ использовать данные регистры, не заботясь о сохранении их содержимого в стеке. Поскольку в процессоре Cortex-M3 регистры R8…R11 не сохраняются автоматически, то при переносе обработчика FIQ необходимо либо изменить регистры, используемые обработчиком, либо сохранять эти регистры в стеке самостоятельно.

Программный код, предназначенный для поддержки вложенных прерываний, можно спокойно убрать. Контроллер NVIC процессора Cortex-M3 имеет встроенную поддержку вложенных прерываний.

Системные ошибки в рассматриваемых процессорах тоже обрабатываются немного по-разному. Так, в процессоре Cortex-M3 предусмотрены различные регистры состояния отказов, позволяющие локализовать источник такого отказа. Также в процессоре Cortex-M3 определены новые типы отказов, в частности ошибки сохранения/восстановления из стека, отказы системы управления памятью и тяжёлые отказы. Поэтому обработчики отказов придётся переписывать.

18.2.3. Модуль MPU

С точки зрения программиста, модуль MPU является всего лишь дополнительным блоком системы, который должен быть сконфигурирован перед использованием. Микроконтроллеры с процессорами ARM7TDMI/ARM7TDMI-S не имеют такого модуля, поэтому перенос программы на процессор Cortex-M3 не вызовет никаких затруднений с этой стороны. А вот в устройствах с процессором ARM720T имеется модуль управления памятью (Memory Management Unit — MMU), функциональное назначение которого отличается от назначения модуля MPU процессора Cortex-M3. Если в приложении используется модуль MMU (для реализации системы виртуальной памяти), то такое приложение не может быть перенесено на процессор Cortex-M3.

18.2.4. Управление системой

Также при переносе программ необходимо обратить внимание на различные подходы к управлению системой в этих двух процессорах. Так, процессор Cortex-M3 имеет специальные команды для перехода в спящий режим. Кроме того, системный контроллер процессора Cortex-M3 не имеет ничего общего с контроллером, используемым в устройствах с процессором ARM7. Поэтому код, отвечающий за управление системой, придётся писать заново.

18.2.5. Режимы работы

В процессоре ARM7 имеется семь режимов работы, большинство которых было заменено в процессоре Cortex-M3 на различные исключения (**Табл. 18.1**).

Таблица 18.1. **Режимы работы и исключения процессоров ARM7 и Cortex-M3**

Режимы и исключения ARM7	Режимы и исключения Cortex-M3
Supervisor (режим по умолчанию)	Привилегированный уровень, режим потока
Supervisor (программное прерывание)	Привилегированный уровень, вызов супервизора (SVCall)
Быстрое прерывание (FIQ)	Привилегированный уровень, прерывание
Запрос прерывания (IRQ)	Привилегированный уровень, прерывание
Abort (prefetch)	Привилегированный уровень, исключение Bus Fault
Abort (data)	Привилегированный уровень, исключение Bus Fault
Undefined	Привилегированный уровень, исключение Usage Fault
System	Привилегированный уровень, режим потока
User	Пользовательский уровень (непривилегированный), режим потока

Быстрое прерывание FIQ, имеющееся в процессоре ARM7, может быть заменено на обычное прерывание (IRQ), поскольку процессор Cortex-M3 позволяет любому прерыванию назначить наивысший приоритет. Соответственно, такое прерывание будет способно вытеснять другие исключения, аналогично прерыванию FIQ в процессоре ARM7. Однако из-за того, что специальные регистры FIQ в процессоре ARM7 отличаются от регистров, сохраняемых в стеке процессором Cortex-M3, вам придётся либо сменить регистры, используемые в обработчике FIQ, либо сохранять эти регистры в стеке самостоятельно.

FIQ и немаскируемое прерывание

Многие разработчики могут ошибочно решить, что прямой заменой быстрого прерывания (FIQ) процессора ARM7 служит немаскируемое прерывание (NMI) процессора Cortex-M3. Действительно, в определённых приложениях такая замена вполне возможна. Однако некоторые отличия NMI от FIQ требуют особого внимания при переносе приложений с заменой прерывания FIQ на немаскируемое прерывание.

Во-первых, немаскируемое прерывание не может быть запрещено, тогда как прерывание FIQ в процессоре ARM7 можно запретить установкой бита F регистра CPSR. Соответственно, в устройствах с процессором Cortex-M3 обработчик NMI может запуститься уже на этапе начальной загрузки программы, тогда как в процессоре ARM7 прерывание FIQ после сброса запрещено.

Во-вторых, в обработчике NMI вы не можете использовать команду `SVC`, тогда как в процессоре ARM7 из обработчика FIQ можно спокойно вызывать команду запуска программного прерывания (`SWI`). Кроме того, при выполнении обработчика FIQ процессором ARM7 допускается возникновение и других исключений (кроме IRQ, поскольку бит I регистра CPSR автоматически устанавливается при обслуживании быстрого прерывания). В то же время в процессоре Cortex-M3 любое исключение отказа, возникшее при выполнении обработчика NMI, может вызвать блокировку процессора.

18.3. Файлы с исходным текстом на ассемблере

Процесс переноса ассемблерных файлов зависит от используемого набора команд — ARM или Thumb.

18.3.1. Режим Thumb

Если в файле содержится код, предназначенный для работы в режиме Thumb, то всё замечательно. В большинстве случаев такой файл можно без каких-либо проблем использовать в программе для нового процессора. Необходимо только проверить его на наличие команд Thumb, не поддерживаемых процессором Cortex-M3, а именно:

- любых команд, пытающихся переключить процессор в режим ARM;
- команды `SWI`, которая была заменена на команду `SVC` (не забудьте скорректировать код, используемый для передачи параметров и возвращаемого значения).

И напоследок, убедитесь в том, что при обращении к стеку в программе используется только модель полного спадающего стека. Эта рекомендация объясняется тем, что в процессоре ARM7TDMI может быть реализована другая стековая модель (скажем, полный вырастающий стек), хотя это и довольно экзотическая ситуация.

18.3.2. Состояние ARM

Если же в файле содержится код ARM, то задача усложняется. Выделим основные моменты:

- *Таблица векторов.* В процессоре ARM7 таблица векторов начинается с адреса 0x00 и содержит команды перехода. В процессоре Cortex-M3 таблица содержит начальное значение указателя стека, адрес вектора сброса, вслед за которыми располагаются адреса обработчиков исключений. Из-за этих различий таблицу векторов придётся полностью переписать.
- *Инициализация регистров.* В процессоре ARM7 зачастую требуется инициализировать разные регистры для каждого из режимов. Так, в процессоре ARM7 используются банковые указатели стека (R13), регистр связи (R14) и регистры SPSR. Поскольку процессор Cortex-M3 имеет другую программную модель, код инициализации регистров необходимо будет изменить. На самом деле, он станет гораздо проще, поскольку в нём уже не будет последовательного переключения процессора между различными режимами.
- *Переключение режимов работы и состояний.* Поскольку режим работы процессора Cortex-M3 определяется иначе, нежели в процессоре ARM7, то код, используемый для переключения режимов, необходимо будет удалить. Это же касается и кода, применяемого для переключения между состояниями ARM и Thumb.
- *Разрешение и запрещение прерываний.* В процессоре ARM7 разрешение/запрещение прерываний осуществляется сбросом/установкой бита I регистра CPSR. В процессоре Cortex-M3 для этих целей используются регистры маскирования прерываний PRIMASK и FAULTMASK. К тому же, в регистре xPSR процессора Cortex-M3 отсутствует бит F, поскольку данный процессор не имеет входа FIQ.

- *Обращения к сопроцессору.* В процессоре Cortex-M3 отсутствует поддержка сопроцессора, поэтому подобные операции не могут быть перенесены на новую платформу.
- *Обработчики прерываний и возврат из прерываний.* В процессоре ARM7 первая команда обработчика прерываний располагается в таблице векторов (как правило, это команда перехода на реальный обработчик). В процессоре Cortex-M3 данный этап уже не требуется. Возврат из прерываний в процессоре ARM7 осуществляется ручной корректировкой значения счётчика команд. В процессоре Cortex-M3 скорректированное значение счётчика команд сохраняется в стеке, а процесс возврата из прерывания запускается загрузкой в счётчик команд значения EXC_RETURN. Для возврата из прерываний в процессоре Cortex-M3 нельзя использовать такие команды, как `MOVS` и `SUBS`. Всё это требует корректировки обработчиков прерываний, а также кода, осуществляющего возврат из прерываний.
- *Код поддержки вложенных прерываний.* Если в процессоре ARM7 требовалось использование вложенных прерываний, то обработчик IRQ должен был переключить процессор в режим System, после чего повторно разрешить прерывания. В процессоре Cortex-M3 ничего этого не требуется.
- *Обработчик FIQ.* При переносе обработчика быстрого прерывания может потребоваться добавить в него команды, сохраняющие содержимое регистров R8…R11 в стеке. В процессоре ARM7 эти регистры являются банковыми, что позволяет обработчику FIQ не заботиться об их сохранении. В то же время в процессоре Cortex-M3 автоматически сохраняются только регистры R0…R3 и R12, а вот регистры R8…R11 — нет.
- *Обработчик* `SWI`. Команда `SWI` заменяется на `SVC`. Однако при портировании обработчика `SWI` необходимо будет переписать код, выполняющий извлечение параметра команды `SWI`. Адрес команды `SVC`, вызвавшей прерывание, можно определить по значению PC, сохранённому в стеке (в процессоре ARM7 значение счётчика команд считывается из регистра связи).
- *Команда* `SWP`. В процессоре Cortex-M3 такая команда отсутствует. Если в исходной программе данная команда использовалась для реализации семафоров, то её необходимо заменить на команды монопольного доступа. Естественно, при этом придётся переписать код поддержки семафоров. Если же команда использовалась исключительно для пересылки данных, то её можно будет заменить несколькими командами обращения к памяти.
- *Обращения к регистрам CPSR и SPSR.* Регистр CPSR процессора ARM7 в процессоре Cortex-M3 был заменён несколькими регистрами состояния программы xPSR, а регистр SPSR — ликвидирован. Если в программе производится чтение состояния флагов процессора, то такую команду следует заменить на команду чтения регистра APSR. Если обработчику исключения необходимо узнать содержимое регистра PSR, которое было до возникновения исключения, то он может прочитать его из стека, поскольку содержимое регистров xPSR автоматически сохраняется в стеке при входе в обработчик. То есть регистр SPSR в процессоре Cortex-M3 попросту не нужен.

- *Условное выполнение.* В процессоре ARM7 условное выполнение поддерживается многими командами ARM, в то время как в машинном коде большинства команд Thumb-2 места для поля условия не предусмотрено. При переносе подобного кода на процессор Cortex-M3 ассемблер может автоматически заменить эти условно выполняемые команды на конструкции с использованием `IT`-блоков. Или же мы можем вручную вставить в программу команды `IT` и команды условных переходов для получения условно выполняемого кода. Единственное неудобство, связанное с заменой условно выполняемых команд на `IT`-блоки, состоит в возможном увеличении размера кода. А это, в свою очередь, может вызвать другие проблемы (например, адреса, используемые в некоторых операциях загрузки/сохранения, могут оказаться за границами диапазона, поддерживаемого командой).
- *Использование значения PC при вычислениях в программе.* При выполнении кода ARM процессором ARM7 значение, возвращаемое при чтении PC, равно адресу команды плюс 8. Это связано с наличием трёхступенчатого конвейера — при чтении счётчика команд на этапе исполнения он уже дважды успевает инкрементироваться (каждый раз на 4 байта). Поскольку процессор Cortex-M3 поддерживает только код Thumb, то разница между считанным и реальным значением PC будет равна четырём.
- *Использование значения регистра R13.* В процессоре ARM7 указатель стека R13 содержит 32-битные значения; в процессоре Cortex-M3 младшие 2 бита указателя стека всегда сброшены в 0. Соответственно, в тех редких случаях, когда регистр R13 используется в качестве регистра данных, код программы необходимо будет модифицировать, чтобы исключить потерю младших битов значений.

Прочий код ARM можно попытаться скомпилировать как код Thumb/Thumb-2, после чего просмотреть сообщения ассемблера и определить дальнейшие корректировки. Например, некоторые команды обращения к памяти с пред- и постиндексацией, используемые в процессоре ARM7, не поддерживаются процессором Cortex-M3 и должны быть заменены на последовательности команд. В других местах программы возможно наличие длинных переходов или больших констант, которые не могут быть скомпилированы в коде Thumb, и поэтому данные команды должны быть вручную заменены на команды Thumb-2.

18.4. Файлы с исходным текстом на Си

Перенос программ, написанных на языке Си, выполняется гораздо проще, чем перенос ассемблерных программ. В большинстве случаев такую программу достаточно просто перекомпилировать для процессора Cortex-M3. Тем не менее, и здесь может возникнуть необходимость ручной корректировки кода:

- *Inline-ассемблер.* В некоторых программах могут присутствовать вставки с использованием inline-ассемблера, которые необходимо будет изменить. Такой код легко можно обнаружить по наличию ключевого слова `__asm`. При использовании пакета RVDS с компилятором RVCT версии 3.0 и выше inline-ассемблер необходимо заменить на встроенный ассемблер.

- *Обработчики прерываний.* В Си-программах для процессора ARM7 при объявлении обработчиков прерываний используется ключевое слово `__irq`. В процессоре Cortex-M3 сохранение регистров и возврат из прерывания осуществляются иначе, нежели в процессоре ARM7, поэтому ключевое слово `__irq` может быть придётся убрать из программы (это зависит от используемого средства разработки). В частности, в средствах разработки компании ARM (в том числе, в пакете RVDS и компиляторе RVCT) ключевое слово `__irq` поддерживает процессор Cortex-M3. Более того, это ключевое слово даже рекомендуется использовать, чтобы улучшить читаемость кода!

Прагмы Си-компилятора ARM наподобие `#pragma arm` и `#pragma thumb` необходимо будет удалить.

18.5. Скомпилированные объектные файлы

В составе большинства компиляторов Си имеются библиотеки различных функций и различные варианты стартового кода, распространяемые в виде предварительно скомпилированных объектных файлов. Некоторые из этих файлов (скажем, стартовый код для традиционных процессоров ARM) не могут использоваться с процессором Cortex-M3 из-за различий в рабочих режимах и состояниях, поддерживаемых процессорами. При наличии исходного кода объектного файла последний можно будет перекомпилировать с помощью набора команд Thumb-2. Для получения дополнительной информации обратитесь к документации на используемое средство разработки.

18.6. Оптимизация

После того как вам удастся заставить свою программу работать на процессоре Cortex-M3, можно попытаться её улучшить, чтобы увеличить скорость работы и уменьшить потребность в памяти. Следует обратить внимание на следующие моменты:

- *Использование команд Thumb-2.* Если 16-битная команда Thumb пересылает содержимое одного регистра в другой с последующим выполнением какой-либо операции над переданным значением, то данную последовательность операций в большинстве случаев можно заменить одной командой Thumb-2. При этом может также уменьшиться число тактов, необходимых для выполнения операции.
- *Использование метода bit-band.* Если регистр периферийного устройства расположен в области памяти с побитовой адресацией, то обращения к битам управления можно значительно упростить, используя область доступа к битам.
- *Умножение и деление.* Процедуры, требующие операций деления, такие как процедуры преобразования значений в десятичные для отображения их на дисплее, могут быть модифицированы с помощью команд деления Cortex-M3. Для выполнения умножения можно использовать различные варианты команд аппаратного умножения.

- *Непосредственные значения.* Некоторые непосредственные значения, которые невозможно было разместить в коде команды Thumb, можно легко закодировать в командах Thumb-2.
- *Ветвления.* Невозможность выполнения длинных переходов с помощью команд Thumb приводила к необходимости использования нескольких команд перехода. Такие команды можно заменить одной командой Thumb-2.
- *Двоичные данные.* Для экономии памяти несколько булевых переменных (принимающих только значения 0 или 1) можно упаковать в один байт/полуслово/слово, расположенное в бит-адресуемой области. Обращаться к этим переменным можно посредством области доступа к битам.
- *Работа с битовыми полями.* В процессоре Cortex-M3 имеется несколько команд для работы с битовыми полями: извлечение беззнакового битового поля (`UBFX`), извлечение битового поля со знаком (`SBFX`), вставка битового поля (`BFI`), очистка битового поля (`BFC`) и перестановка битов (`RBIT`). Использование этих команд может упростить реализацию таких операций, как обращения к периферийным устройствам, формирование пакетов данных или обмен данными по последовательному каналу.
- *IT-блоки.* Некоторые короткие переходы можно заменить блоком команды `IT`. Это позволит нам избежать напрасной траты тактов на очистку конвейера при выполнении перехода.
- *Переключение состояния ARM/Thumb.* В определённых ситуациях разработчикам приходилось разбивать исходный код приложения на несколько файлов, после чего компилировать одни из этих файлов с использованием набора ARM, а другие — с использованием набора Thumb. Обычно это делалось для увеличения плотности кода в тех случаях, когда производительность была некритична. Поскольку процессор Cortex-M3 поддерживает только набор команд Thumb-2, такое разбиение кода больше не требуется. Это позволяет исключить накладные расходы на переключение состояний, что ведёт к уменьшению размеров кода и, возможно, уменьшению числа файлов проекта.

ГЛАВА 19
РАЗРАБОТКА ПРИЛОЖЕНИЙ ДЛЯ CORTEX-M3 С ИСПОЛЬЗОВАНИЕМ GNU

19.1. Общие сведения

Многие разработчики при создании приложений для микроконтроллеров с процессорами ARM используют открытый инструментарий GNU, на котором основан ряд средств разработки для ARM. Набор инструментов GNU, поддерживающий процессор Cortex-M3, доступен как в виде исходных кодов для последующей компиляции GNU-компилятором gcc, так и в виде скомпилированных и готовых к использованию исполняемых файлов, которые распространяются различными компаниями-разработчиками программных средств.

Одной из таких компаний, предлагающей средства разработки для процессоров Cortex-M3, основанные на инструментарии GNU, является компания CodeSourcery. Пакет программ Sourcery G++, разработанный компанией, распространяется в трёх вариантах:

1. *Sourcery G++ Lite*: — бесплатная версия, которую можно свободно загрузить с веб-сайта компании (*www.codesourcery.com*). Эта версия поддерживает работу только из командной строки и имеет ограниченные возможности отладки.
2. *Sourcery G++ Personal Edition* — наиболее популярная версия пакета, поскольку имеет небольшую стоимость при достаточно больших возможностях:

- Интегрированная среда разработки с интерфейсом на базе пакета Eclipse.
- Поддержка широкого спектра микроконтроллеров с процессорами ARM, включая поддержку CS3 для этих микроконтроллеров (сценарии компоновщика и конфигурации средств отладки).
- Поддержка отладочных плат различных производителей, в том числе плат компаний Texas Instrument с микроконтроллерами Stellaris и STMicroelectronics с микроконтроллерами STM32.
- Большая подборка примеров программ.
- Встроенная поддержка различных отладочных интерфейсов, в том числе:
  - ARMUSB (встроен в микроконтроллеры Stellaris);
  - Segger J-Link;
  - Keil ULINK2.

- Мастер «Board Builder Wizard», облегчающий использование неизвестных плат:
  - клонирование определений платы;
  - изменение распределения памяти;
  - изменение стартового кода;
  - конфигурирование отладочных средств.
- Поддержка импорта примеров из библиотеки StellarisWare.

3. *Sourcery G++ Professional Edition* — все возможности варианта Personal Edition плюс дополнительные библиотеки и неограниченная техническая поддержка.

Все примеры, представленные в этой главе, рассчитаны на инструменты из пакета Sourcery G++ Lite, работающие с командной строкой, поскольку вопросы, касающиеся использования данного режима, актуальны для большинства средств разработки, основанных на инструментарии GNU. Сразу хочу предупредить, что в данной главе содержатся только самые основные сведения по применению указанных инструментов. Подробную информацию по работе с инструментами GNU вы можете получить из документации, имеющейся в составе используемого пакета программ, а также из сети Интернет.

Синтаксис ассемблера GNU (as.exe) немного отличается от синтаксиса оригинального ассемблера ARM. Эти различия касаются объявлений, директив компилятора, комментариев и прочих элементов языка. Соответственно, чтобы использовать в среде GNU ассемблерные программы, написанные для пакета RVDS компании ARM, их необходимо будет откорректировать.

19.2. Приобретение инструментария GNU

Скомпилированный вариант инструментов GNU можно загрузить по адресу: *www.codesourcery/com/sgpp/lite/arm*. По этому адресу доступно несколько сборок. Для начала возьмём любую сборку с поддержкой интерфейса EABI и без какой-либо встраиваемой операционной системы. Данный инструментарий доступен для различных платформ, включая Windows и Linux. Примеры, представленные в этой главе, работают в любой из указанных версий.

19.3. Процесс разработки программы

В состав инструментария GNU входят компилятор, ассемблер, редактор связей и ряд дополнительных утилит. Эти инструменты позволяют создавать проекты, содержащие одновременно исходные тексты на языках Си и ассемблера (**Рис. 19.1**).

При использовании компилятора GCC компоновщик обычно запускается компилятором по завершении этапа компиляции, что гарантирует передачу компоновщику корректных библиотек и параметров. Если же компоновщик будет использоваться напрямую, то связывание объектных файлов может оказаться невозможным. Кроме того, без информации со стороны компилятора компоновщик может сформировать на выходе двоичный образ, не совместимый с интерфейсом EABI.

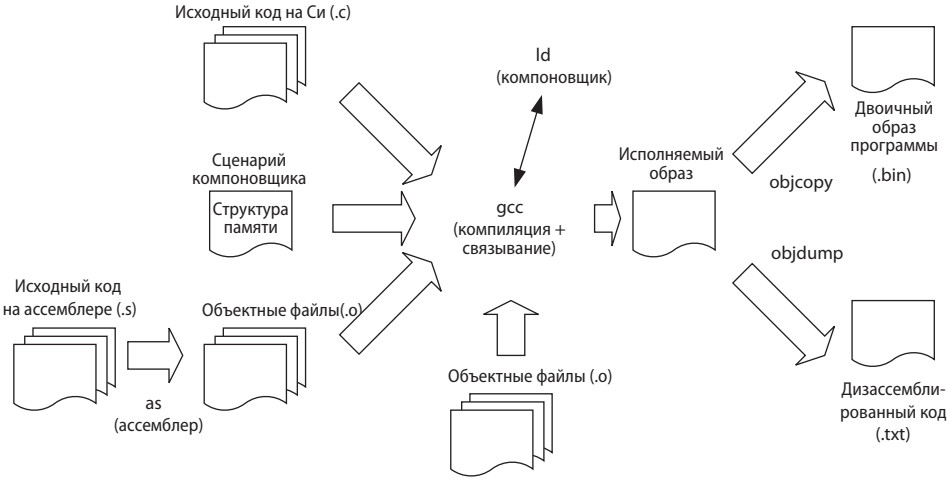


Рис. 19.1. *Разработка ПО с использованием пакета Sourcery G++.*

Существуют версии инструментария GNU для самых разных платформ (Symbian, Linux, EABI и т.д.). Имена программ обычно имеют префикс, определяемый целевой платформой, для которой предназначена данная версия. Например, при использовании EABI-совместимого[1)] инструментария компилятор Си (gcc) будет иметь имя `arm-xxxx-eabi-gcc`. В следующих примерах мы будем применять инструменты, поставляемые в составе пакета Sourcery G++. Названия этих программ указаны в **Табл. 19.1**.

Таблица 19.1. **Имена исполняемых файлов пакета Sourcery G++**

Назначение	Файл (EABI-вариант)
Ассемблер	`arm-none-eabi-as`
Компилятор с языка Си	`arm-none-eabi-gcc`
Компоновщик	`arm-none-eabi-ld`
Генератор двоичных образов	`arm-none-eabi-objcopy`
Дизассемблер	`arm-none-eabi-objdump`

Примечание. Заметьте, имена файлов отличаются от имён, используемых другими разработчиками программных средств.

Если исходный код вашего проекта написан целиком на ассемблере, то для связывания объектных файлов компоновщик можно вызвать напрямую (**Рис. 19.2**).

[1)] Стандарт EABI для архитектуры ARM — исполняемые файлы должны удовлетворять данной спецификации, чтобы их можно было использовать с различными наборами средств разработки.

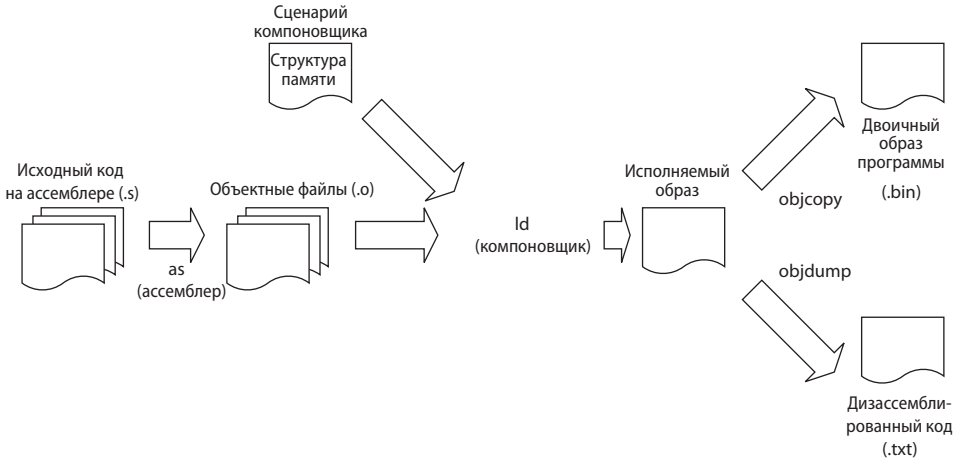


Рис. 19.2. *Разработка ПО на ассемблере.*

19.4. Примеры

Рассмотрим несколько примеров по использованию инструментария GNU.

19.4.1. Пример 1: первая программа

Для начала возьмём простую ассемблерную программу, уже рассмотренную нами в Главе 10, которая вычисляет сумму чисел от 1 до 10:

```
========== example1.s ==========
/* Определяем константы */
      .equ STACK_TOP, 0x20000800
      .text
      .syntax unified
      .thumb
      .global _start
      .type start, %function
_start:
      .word STACK_TOP, start
      /* Начало основной программы */
start:
      movs r0, #10
      movs r1, #0
      /* Вычисляем 10+9+8... +1 */
loop:
      adds r1, r0
      subs r0, #1
      bne  loop
      /* Результат в R1 */
deadloop:
      b    deadloop
.end
========== Конец файла ==========
```

- Директива `.word` в данном примере помогает нам задать начальное значение указателя стека (0x20000800) и вектор сброса (start).
- Директива `.text` указывает на то, что данная секция программы должна быть ассемблирована.
- Директива `.syntax unified` указывает на использование унифицированного синтаксиса языка ассемблера.
- Директива `.thumb` сообщает ассемблеру о том, что в программе используется набор команд Thumb®. Также для этого можно использовать традиционную директиву `.code16`.
- Директива `.global` позволяет сделать метку `_ start` видимой в других объектных файлах.
- `_ start` — метка, отмечающая начало программы.
- `start` — отдельная метка, адрес которой содержится в векторе сброса.
- `.type start, %function` — объявляет идентификатор `start` как функцию. Это необходимо сделать для всех векторов исключений. В противном случае, ассемблер сбросит младший значащий бит вектора в 0.
- Директива `.end` отмечает конец программы.

В отличие от ассемблера ARM, в ассемблере GNU метки должны завершаться символом двоеточия (:). Комментарии помещаются между символами «/*» и «*/», а директивы начинаются с символа точки (.).

Обратите внимание на то, что вектор сброса (start) объявлен как функция (`.type start, %function`), использующая код Thumb (`.thumb`). Так было сделано для того, чтобы установить младший бит вектора сброса в 1 (установленный младший бит говорит процессору о том, что выполнение кода начинается в состоянии Thumb). Если этого не сделать, то процессор попытается переключиться в режим ARM, что вызовет исключение Hard Fault. Для ассемблирования данного файла наберём в командной строке:

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example1.s -o example1.o
```

В результате будет создан объектный файл `example1.o`. Ключи `–mcpu` и `–mthumb` определяют используемый набор команд. Для компоновки проекта вызовем линкер `ld`:

```
$> arm-none-eabi-ld -Ttext 0x0 -o example1.out example1.o
```

Далее, сформируем двоичный образ, используя команду `objcopy`:

```
$> arm-none-eabi-objcopy -Obinary example1.out example1.bin
```

Можно проконтролировать получившийся код, дизассемблировав его с использованием программы `objdump`:

```
$> arm-none-eabi-objdump -S example1.out > example1.list
```

В результате выполнения последней программы мы получим файл листинга, содержащий что-то вроде:

```
example1.out: file format elf32-littlearm
Disassembly of section .text:
00000000 <_start>:
0:     20000800  .word 0x20000800
4:     00000009  .word 0x00000009
00000008 <start>:
```

```
8:     200a movs   r0, #10
a:     2100 movs   r1, #0
0000000c <loop>:
c:     1809 adds   r1, r1, r0
e:     3801 subs   r0, #1
10:    d1fc bne.n  c <loop>
00000012 <deadloop>:
12:    e7fe b.n    12 <deadloop>
```

19.4.2. Пример 2: связывание нескольких файлов

Как уже неоднократно говорилось, мы можем создать несколько объектных файлов, а затем связать их друг с другом. Давайте рассмотрим пример, состоящий из двух ассемблерных файлов `example2a.s` и `example2b.s`; первый из них содержит только таблицу векторов, а второй — программный код. Для передачи адресов между файлами используется директива `.global`:

```
========== example2a.s ==========
/* Определяем константы */
     .equ STACK_TOP, 0x20000800
     .syntax unified
     .global vectors_table
     .global start
     .global nmi_handler
     .thumb
vectors_table:
     .word STACK_TOP, start, nmi_handler, 0x00000000
     .end
========== Конец файла ==========
========== example2b.s ==========
/* Основная программа */
     .text
     .syntax unified
     .thumb
     .type start, %function
     .type nmi_handler, %function
     .global _start
     .global start
     .global nmi_handler
_start:
     /* Начало основной программы */
start:
     movs  r0, #10
     movs  r1, #0
     /* Вычисляем 10+9+8... +1 */
loop:
     adds  r1, r0
     subs  r0, #1
     bne   loop
     /* Результат в R1 */
deadloop:
     b     deadloop
```

```
     /* Пустой обработчик NMI для иллюстрации */
nmi_handler:
     bx    lr
     .end
========== Конец файла ==========
```

Для создания двоичного образа необходимо выполнить следующие операции:

1. Ассемблируем `example2a.s`:

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example2a.s -o example2a.o
```

2. Ассемблируем `example2b.s`:

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example2b.s -o example2b.o
```

3. Связываем объектные файлы в единый образ. Обратите внимание, что порядок указания объектных файлов в командной строке влияет на их расположение в исполняемом двоичном образе:

```
$> arm-none-eabi-ld -Ttext 0x0 -o example2.out example2a.o example2b.o
```

4. Формируем двоичный файл:

```
$> arm-none-eabi-objcopy -Obinary example2.out example2.bin
```

5. Как и в предыдущем примере, сгенерируем файл листинга, чтобы убедиться в корректности финального образа:

```
$> arm-none-eabi-objdump -S example2.out > example2.list
```

При большом числе файлов с исходным кодом процесс компиляции можно упростить, воспользовавшись UNIX-командой `makefile`. Отдельные пакеты разработки также могут иметь встроенные средства для облегчения процесса компиляции.

19.4.3. Пример 3: простая программа «Hello World»

Чтобы было не так скучно, давайте попробуем написать программу «Hello World». (**Примечание**. В следующем примере отсутствует код инициализации модуля UART; для запуска данного примера вы должны добавить собственный код инициализации UART. Пример инициализации модуля UART на Си приведён в Главе 20.)

```
========== example3a.s ==========
/* Определяем константы */
      .equ STACK_TOP, 0x20000800
      .syntax unified
      .thumb
      .global vectors_table
      .global _start
vectors_table:
      .word STACK_TOP, _start
      .end
========== Конец файла ==========
========== example3b.s ==========
.text
.syntax unified
.thumb
.global _start
```

```
.type _start, %function
_start:
      /* Начало основной программы */
      movs r0, #0
      movs r1, #0
      movs r2, #0
      movs r3, #0
      movs r4, #0
      movs r5, #0
      ldr r0,=hello
      bl puts
      movs r0, #0x4
      bl putc
deadloop:
      b deadloop
hello:
      .asciz «Hello\n»
      .align
puts: /* Подпрограмма передачи строки по UART */
      /* Вход: r0 = Адрес начала строки */
      /* Строка должна завершаться нулевым символом */
      push {r0, r1, lr}       /* Сохраняем регистры */
      mov r1, r0              /* Копируем адрес в R1, поскольку */
                              /* R0 будет использоваться в качестве */
                              /* входного параметра putc */
putsloop:
      ldrb.w r0,[r1],#1
      /* Читаем один символ и инкрементируем адрес */
      cbz r0, putsloopexit    /* Если символ - NULL, выходим */
      bl  putc
      b   putsloop
putsloopexit:
      pop {r0, r1, pc}        /* Возврат */
.equ UART0_DATA, 0x4000C000
.equ UART0_FLAG, 0x4000C018
putc: /* Подпрограмма для передачи символа через UART */
      /* Вход: R0 = передаваемый символ */
      push {r1, r2, r3, lr}   /* Сохраняем регистры */
      LDR  r1,=UART0_FLAG
putcwaitloop:
      ldr   r2,[r1]           /* Считываем флаги */
      tst.w r2, #0x20         /* Проверяем состояние буфера передачи */
      bne   putcwaitloop      /* Если полон, то ждём */
      ldr   r1,=UART0_DATA    /* Иначе загружаем символ в буфер передачи */
      str   r0, [r1]
      pop   {r1, r2, r3, pc}  /* Возврат */
     .end
========== Конец файла ==========
```

В данном примере для определения строки с завершающим нулем мы воспользовались директивой `.asciiz`. Эта директива эквивалентна директиве `.ascii`, определяющей строку, с последующей директивой `.byte`, определяющей

байт с нулевым значением. Для корректного выравнивания программного кода после определения строки мы разместили директиву .align.

Для компиляции программы, создания двоичного образа и дизассемблированного листинга можно использовать следующие команды:

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example3a.s -o example3a.o
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example3b.s -o example3b.o
$> arm-none-eabi-ld -Ttext 0x0 -o example3.out example3a.o example3b.o
$> arm-none-eabi-objcopy -Obinary example3.out example3.bin
$> arm-none-eabi-objdump -S example3.out > example3.list
```

19.4.4. Пример 4: данные в ОЗУ

Очень часто данные хранятся во встроенном ОЗУ микроконтроллера. Рассмотрим следующий пример:

```
========== example4.s ==========
    .equ STACK_TOP, 0x20000800
    .text
    .syntax unified
    .thumb
    .global _start
    .type start, %function
_start:
    .word STACK_TOP, start
    /* Начало основной программы */
start:
    movs r0, #10
    movs r1, #0
    /* Вычисляем 10+9+8... +1 */
loop:
    adds r1, r0
    subs r0, #1
    bne  loop
    /* Результат в R1 */
    ldr  r0,=Result
    str  r1,[r0]
deadloop:
    b    deadloop
    /* Данные расположены в секции LC (Local Common) */
    .lcomm Result 4 /* 4-байтное значение названо Result */
    .end
========== Конец файла ==========
```

Псевдокоманда .lcomm, использованная в примере, резервирует 4 байта в секции неинициализированных данных «bss» и помечает эту область меткой Result. Используя указанный идентификатор, программа может обращаться к данной области памяти.

Чтобы скомпоновать программу, мы должны сообщить компоновщику о местоположении ОЗУ. Для этого используется ключ –Tbss, размещающий секцию неинициализированных данных по заданному адресу:

```
$> arm-none-eabi-as -mcpu=cortex-m3 -mthumb example4.s -o example4.o
```

```
$> arm-none-eabi-ld -Ttext 0x0 -Tbss 0x20000000 -o example4.out example4.o
$> arm-none-eabi-objcopy -Obinary example4.out example4.bin
$> arm-none-eabi-objdump -S example4.out > example4.list
```

19.4.5. Пример 5: программа на Си

Одним из основных компонентов инструментария GNU является компилятор языка Си. В этом примере мы напишем программу целиком на Си. Помимо собственно программы, нам ещё потребуется файл сценария компоновщика для размещения сегментов по соответствующим адресам:

```
========== example5.c ==========
// Объявляем функции
void myputs(char *string1);
void myputc(char mychar);
int main(void);
void Reset_Handler(void);
void NMI_Handler(void);
void HardFault_Handler(void);
void UartInit(void);
// Объявляем _start() - в стартовом коде
extern void _start(void);
//---------------------------------
void Reset_Handler(void)
{
  // Вызываем обработчик сброса CS3
  _start();
}
//---------------------------------
// Пустой обработчик
void NMI_Handler(void)
{
  return;
}
//---------------------------------
// Пустой обработчик
void HardFault_Handler(void)
{
  return;
}
//---------------------------------
void UartInit(void)
{
  /* Здесь должен располагаться код инициализации UART */
  return;
}
//---------------------------------
// Начало основной программы
int main(void)
{
#define NVIC_CCR (*((volatile unsigned long *)(0xE000ED14)))
const char *helloworld = «Hello world\n»;
```

```
  NVIC_CCR = NVIC_CCR | 0x200; /* Устанавливаем STKALIGN в NVIC */
  UartInit();
  myputs(helloworld);
  while(1);
  return(0);
}
//---------------------------------
// Функция вывода строки
void myputs(char *string1)
{
  char mychar;
  int j;
  j=0;
  do {
    mychar = string1[j];
    if (mychar!=0) {
      myputc(mychar);
      j++;
    }
  } while (mychar != 0);
  return;
}
//---------------------------------
void myputc(char mychar)
{
#define UART0_DATA (*((volatile unsigned long *)(0x4000C000)))
#define UART0_FLAG (*((volatile unsigned long *)(0x4000C018)))
  // Ждём сброса флага занятости
  while ((UART0_FLAG & 0x20) != 0);
  // Выводим символ в UART
  UART0_DATA = mychar;
  return;
}
========== Конец файла ==========
```

Эта программа выводит строку «Hello world» через интерфейс UART. В зависимости от используемого модуля вы должны либо сами написать код его инициализации, либо воспользоваться функциями библиотеки, предоставляемой производителем микроконтроллера.

После выхода процессора из состояния сброса обработчик вызывает функцию `_start`, содержащую стартовый код. По завершении начальной инициализации стартовый код вызывает функцию `main()`. Для поддержки стартовой последовательности и таблицы векторов пакет Sourcery G++ использует инфраструктуру CS3 (CodeSourcery Common Start-up Code Sequence). В составе CS3 имеется предопределённая таблица векторов для процессора Cortex-M3, называемая `_cs3_interrupt_vector_micro`. Содержимое этой таблицы векторов приведено в **Табл. 19.2**.

Таблица 19.2. **Таблица векторов Cortex-M3 в CS3**

Номер	Имя вектора	Описание
0	__cs3_stack	Начальное значение указателя основного стека
1	__cs3_reset	Вектор сброса
2	__cs3_isr_nmi	Немаскируемое прерывание
3	__cs3_isr_hard_fault	Тяжёлый отказ
4	__cs3_isr_mpu_fault	Отказ системы управления памятью
5	__cs3_isr_bus_fault	Отказ шины
6	__cs3_isr_usage_fault	Отказ программы
7…10	__cs3_isr_reserved_7…10	Зарезервировано
11	__cs3_isr_svcall	Вызов супервизора
12	__cs3_isr_debug	Исключение монитора отладки
13	__cs3_isr_reserved_13	Зарезервировано
14	__cs3_isr_pendsv	PendSV
15	__cs3_isr_systick	Системный таймер
16…47	__cs3_isr_external_0… __cs3_isr_external_31	Внешнее прерывание

Соответствие между обработчиками исключений, используемыми в нашей программе, и данными именами векторов устанавливается сценарием компоновщика. В этом же файле описывается распределение памяти, включая расположение таблицы векторов. В версиях Personal и Professional пакета поставляются файлы сценариев для большинства выпускаемых микроконтроллеров с процессором Cortex-M3. При использовании версии Lite несколько общих файлов сценария можно найти в папке arm-none-eabi\lib. В данном примере мы воспользуемся сценарием, полученным модификацией общего файла сценария для процессоров Cortex-M (`generic-m.ld`). Содержимое этого модифицированного файла сценария приведено в Приложении Д.

Для запуска компилятора и компоновщика используется следующая командная строка:

```
$> arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb example5.c
   -T cortexm3.ld -o example5.o
```

Информация о карте памяти передаётся компоновщику на этапе компиляции.

Компилятор gcc запустит компоновщик автоматически, так что отдельно его вызывать не нужно. В заключение мы можем сгенерировать двоичный образ и дизассемблированный листинг:

```
$> arm-none-eabi-objcopy -Obinary example5.out example5.bin
$> arm-none-eabi-objdump -S example5.out > example5.list
```

Использование обработчика `Reset _ Handler` в данном примере необязательно. Вместо этого в сценарии компоновщика можно присвоить вектору `_ _ cs3 _ reset` значение `_ start` (адрес входа в стартовую процедуру).

19.4.6. Пример 6: перенаправление вывода в программе на Си

В предыдущем примере мы написали собственную функцию вывода текстовых строк, однако в большинстве случаев удобнее пользоваться функциями вывода из стандартной библиотеки Си. Так, для вывода текста вполне можно использовать стандартную функцию `printf`. Для этого нам необходимо написать функцию, которая перенаправляла бы результат работы `printf` в процедуру вывода по интерфейсу UART.

Реализация такой функции перенаправления демонстрируется в следующем примере:

```
========== example6.c ==========
#include<stdio.h>
// Объявляем функции
void myputc(char mychar);
int  main(void);
void Reset_Handler(void);
void NMI_Handler(void);
void HardFault_Handler(void);
void UartInit(void);
// Объявляем _start() - в стартовом коде
extern void _start(void);
//---------------------------------
void Reset_Handler(void)
{
  // Вызываем обработчик сброса CS3
  _start();
}
//---------------------------------
// Пустой обработчик
void NMI_Handler(void)
{
  return;
}
//---------------------------------
// Пустой обработчик
void HardFault_Handler(void)
{
  return;
}
//---------------------------------
void UartInit(void)
{
  /* Здесь должен располагаться код инициализации UART */
  return;
}
//---------------------------------
// Функция перенаправления вывода
int _write_r(void *reent, int fd, char *ptr, size_t len)
{
```

```
  size_t i;
  for (i=0; i<len; i++)
  {
    myputc(ptr[i]); // Вызываем нашу функцию вывода символа
  }
  return len;
}
//---------------------------------
// Начало основной программы
int main(void)
{
#define NVIC_CCR (*((volatile unsigned long *)(0xE000ED14)))
  NVIC_CCR = NVIC_CCR | 0x200; /* Устанавливаем STKALIGN в NVIC */
  UartInit();
  printf(«Hello world\n»);
  while(1);
  return(0);
}
//---------------------------------
// Функция для вывода символа
void myputc(char mychar)
{
#define UART0_DATA (*((volatile unsigned long *)(0x4000C000)))
#define UART0_FLAG (*((volatile unsigned long *)(0x4000C018)))
  // Ждём сброса флага занятости
  while ((UART0_FLAG & 0x20) != 0);
  // Выводим символ в UART
  UART0_DATA = mychar;
  return;
}
========== Конец файла ==========
```

Перенаправление осуществляется функцией `_write_r`, которая вызывает нашу процедуру вывода символа.

19.4.7. Пример 7: реализация собственной таблицы векторов

Если вы не используете пакет Sourcery G++, то вам может потребоваться самостоятельно описать таблицу векторов. Это можно сделать следующим образом:

```
// Определяем таблицу векторов
__attribute__ ((section(«vectors»)))
void (* const VectorArray[])(void) = {
  (void (*)(void))((unsigned long) MainStack + sizeof(MainStack)),
  Reset_Handler,
  NMI_Handler,
  HardFault_Handler
};
```

Стек можно описать в виде массива:

```
// Резервируем 64 слова памяти для основного стека
static unsigned long MainStack[64];
```

Поместить таблицу векторов в начало адресного пространства процессора можно с помощью сценария компоновщика. Например:

```
.text :
{
  CREATE_OBJECT_SYMBOLS
  __cs3_region_start_rom = .;
  *(.cs3.region-head.rom)
  __cs3_interrupt_vector = __cs3_interrupt_vector_micro;
  *(vectors) /* Таблица векторов */
```

Имя секции (`vectors`) должно соответствовать имени, использованному при описании таблицы векторов. В противном случае, таблица окажется расположенной в другом месте.

Этот метод может пригодиться даже при наличии инструментария Sourcery G++; если вам требуется более 32 векторов прерываний, то вы можете описать дополнительные векторы и поместить их сразу после таблицы векторов CS3.

19.5. Обращения к регистрам специального назначения

Инструментарий GNU поддерживает обращения к регистрам специального назначения. Названия этих регистров должны записываться строчными буквами. Например:

```
msr control, r1
mrs r1, control
msr apsr, R1
mrs r0, psr
```

19.6. Использование неподдерживаемых команд

Если вы применяете другой набор инструментов GNU ARM, то может получиться так, что используемый ассемблер не поддерживает необходимую вам команду. В этом случае команду можно будет вставить в программу в виде двоичного кода с помощью директивы `.word`. Например:

```
.equ DW_MSR_CONTROL_R0, 0x8814F380
       ...
       MOV R0, #0x1
.word DW_MSR_CONTROL_R0 /* Эта команда переводит процессор в пользовательский
                           режим */
       ...
```

19.7. Inline-ассемблер в компиляторе GCC

Как и Си-компилятор компании ARM, компилятор GCC тоже поддерживает inline-ассемблер. Правда, синтаксис несколько отличается:

```
__asm (« inst1 op1, op2... \n»
       « inst2 op1, op2... \n»
       ...
       « inst op1, op2... \n»
       : output_operands /* Опция */
       : input_operands /* Опция */
       : clobbered_register_list /* Опция */
       );
```

Например, простая процедура перевода процессора в спящий режим может выглядеть следующим образом:

```
void Sleep(void)
{ // Переход в спящий режим с помощью команды WFE
  __asm (
    «WFI\n»
    );
}
```

Если в ассемблерном коде должны использоваться входные и выходные переменные, то его можно записать следующим образом:

```
unsigned int DataIn, DataOut; /* Переменные для параметра и результата */
...
__asm («mov r0, %0\n»
       «mov r3, #5\n»
       «udiv r0, r0, r3\n»
       «mov %1, r0\n»
       :»=r» (DataOut) : «r» (DataIn) : «cc», «r3» );
```

В этом фрагменте входным параметром является переменная `DataIn` (параметр `%0`), а возвращаемым значением — переменная `DataOut` (параметр `%1`). Ассемблерный код модифицирует регистр R3 и изменяет флаги условия `cc`, в связи с чем они перечислены в списке затираемых регистров.

Для дальнейшего изучения inline-ассемблера рекомендую обратиться к документу «*GCC-Inline-Assembly-HOWTO*» (на англ. языке), свободно распространяемому в сети Интернет.

ГЛАВА 20
ИСПОЛЬЗОВАНИЕ ПАКЕТА REALVIEW MDK-ARM КОМПАНИИ KEIL

20.1. Общие сведения

В настоящее время на рынке представлено множество программных средств разработки для устройств с процессором Cortex-M3. Одним из наиболее популярных пакетов является пакет RealView Microcontroller Development Kit (RealView MDK-ARM), предлагаемый компанией Keil. В составе этого пакета имеются следующие компоненты:

- Интегрированная среда разработки (ИСР) µVision.
- Отладчик.
- Симулятор.
- Инструментарий RealView Compilation Tools компании ARM:
  - компилятор C/C++;
  - ассемблер;
  - компоновщик и различные утилиты.
- Операционная система реального времени RTX Real-Time Kernel.
- Стартовые коды для различных микроконтроллеров.
- ПО для программирования флэш-памяти микроконтроллеров.
- Примеры программ.

Для изучения процессора Cortex-M3 с помощью пакета RealView MDK-ARM наличие реальной платы с микроконтроллером на базе Cortex-M3 необязательно. В составе ИСР µVision имеется симулятор, позволяющий тестировать простые программы без отладочной платы.

Бесплатный компакт-диск с пробной версией пакета можно заказать на сайте компании (*www.keil.com*). Эта же версия пакета поставляется в составе некоторых отладочных комплектов от разных производителей микроконтроллеров.

20.2. Приступая к работе в ИСР µVision

Вместе с пакетом MDK-ARM поставляется множество примеров, в том числе и для микроконтроллеров с ядром Cortex-M3 и различных отладочных плат, предлагаемых на рынке. Помимо этого, с сайта производителя микроконтроллера можно загрузить библиотеки драйверов устройств, в которых тоже имеются

примеры программ. При разработке своего приложения вы можете взять за основу подходящий учебный проект или же создать свой проект с нуля. Данному вопросу и посвящена настоящая глава. Все примеры, приведённые в этой главе, были написаны в среде MDK-ARM версии 3.80 для микроконтроллера LM3S811 компании Texas Instruments.

После инсталляции пакета MDK-ARM вы сможете запустить ИСР μVision из меню «Программы». При первом запуске μVision будет автоматически создан пустой проект для традиционного процессора ARM. Мы можем закрыть этот проект и создать свой, выбрав пункт **New Project** в меню **Project** (**Рис. 20.1**).

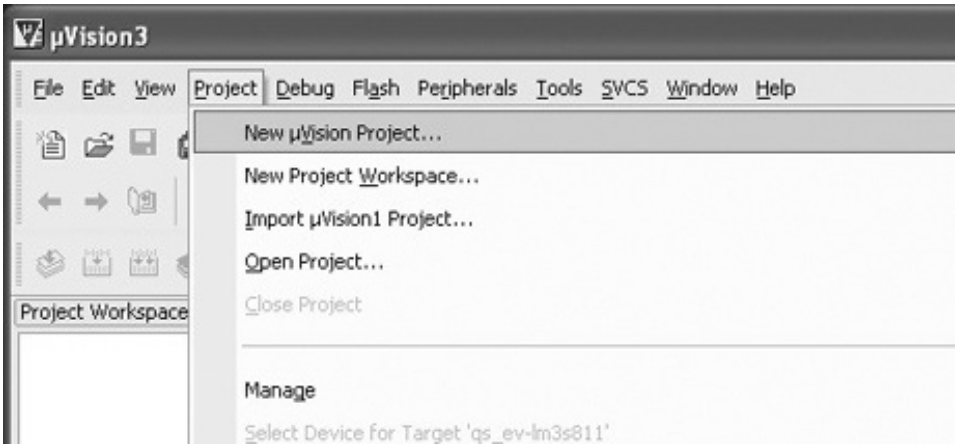


Рис. 20.1. *Создание нового проекта.*

Создадим для нашего проекта новую папку HelloWorld (**Рис. 20.2**). После этого нам нужно будет выбрать целевое устройство. Выберем микроконтроллер LM3S811 (**Рис. 20.3**).

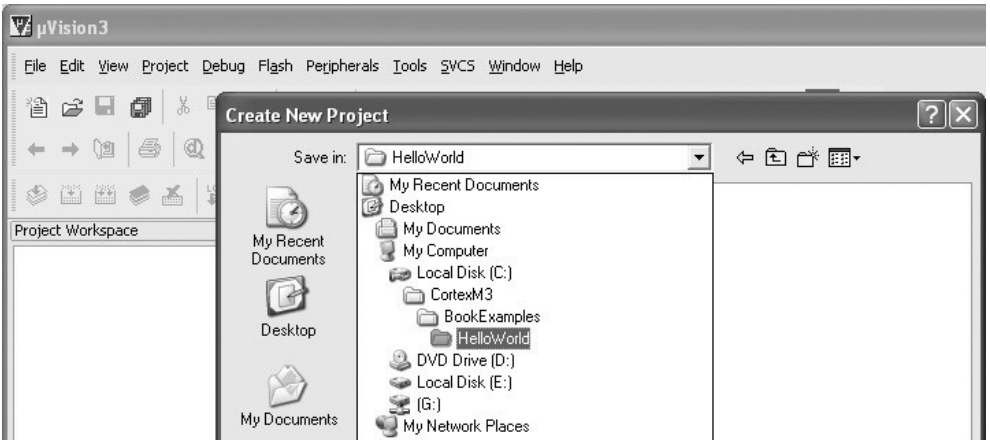


Рис. 20.2. *Выбор папки для размещения проекта.*

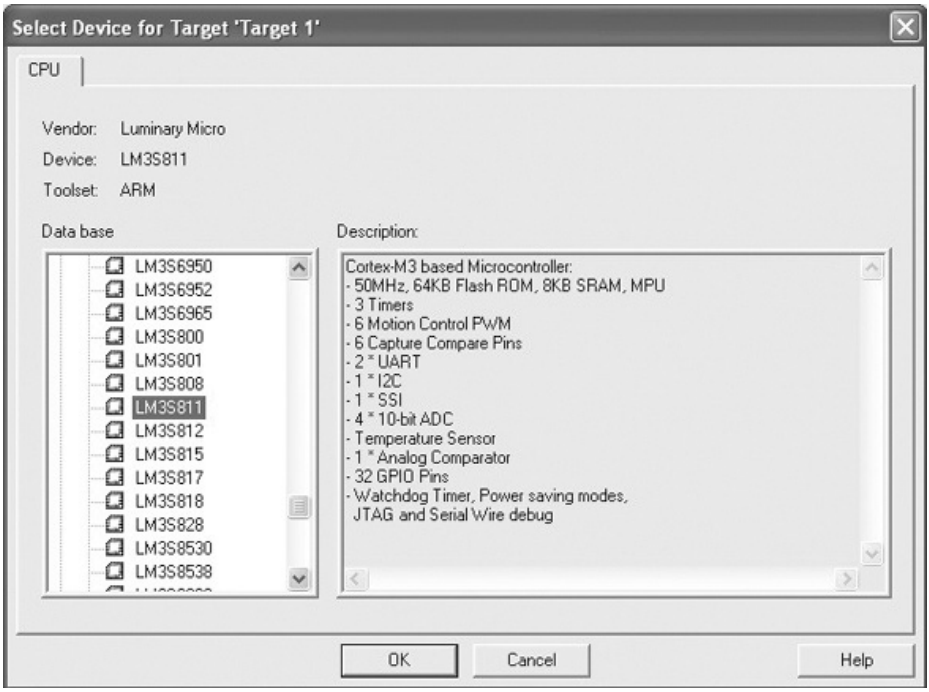


Рис. 20.3. *Выбор целевого микроконтроллера.*

После выбора микроконтроллера появится диалоговое окно с вопросом об использовании стартового кода, принятого по умолчанию. Нажмите на кнопку **Yes** (**Рис. 20.4**).

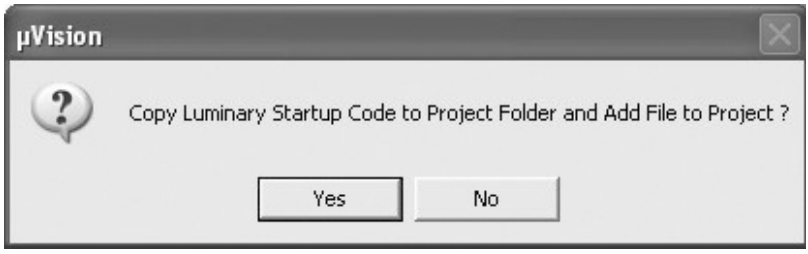


Рис. 20.4. *Выбор стандартного стартового кода.*

В результате проделанных манипуляций будет создан проект HelloWorld, содержащий всего один файл Startup.s (**Рис. 20.5**). Теперь нам нужно создать новый файл, который будет содержать тело основной программы (**Рис. 20.6**). Сохраним этот текстовый файл под именем Hello.c (**Рис. 20.7**). Теперь мы можем добавить созданный файл в проект через контекстное меню, появляющееся при щелчке правой кнопкой мыши на группе Source Group 1 в окне проекта (**Рис. 20.8**).

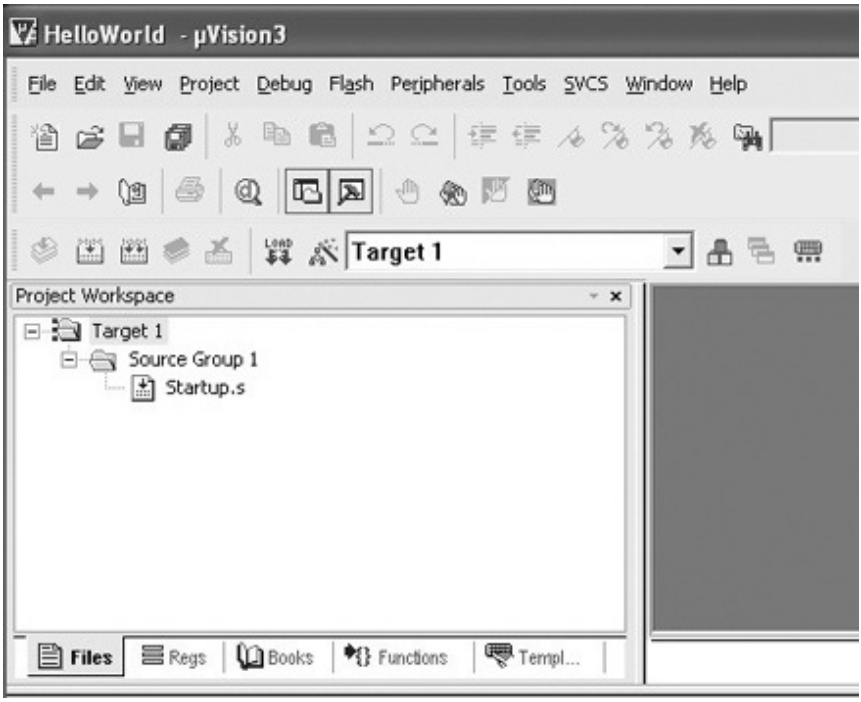


Рис. 20.5. *Созданный проект.*

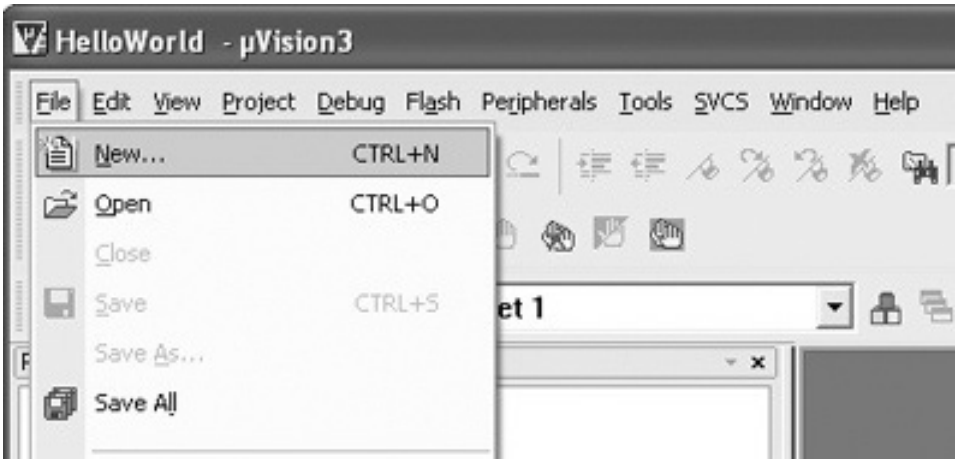


Рис. 20.6. *Создание нового файла с исходным кодом.*

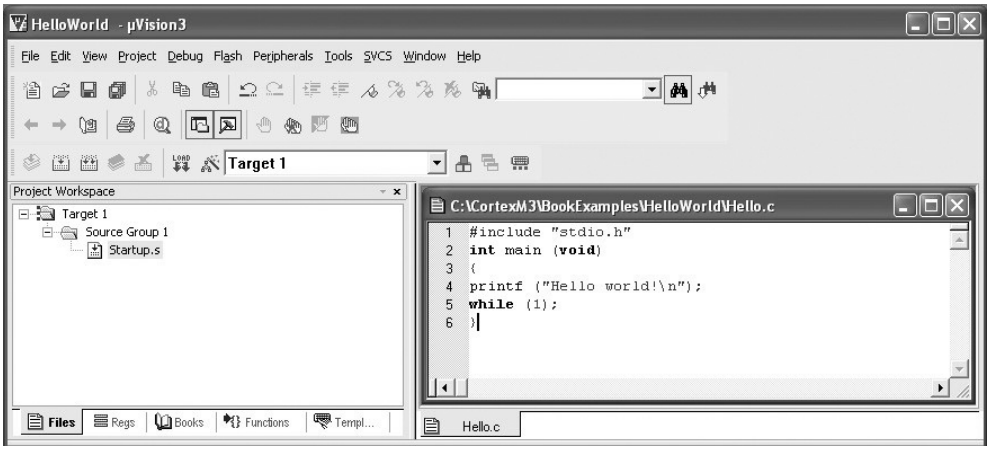


Рис. 20.7. *Программа «HelloWold».*

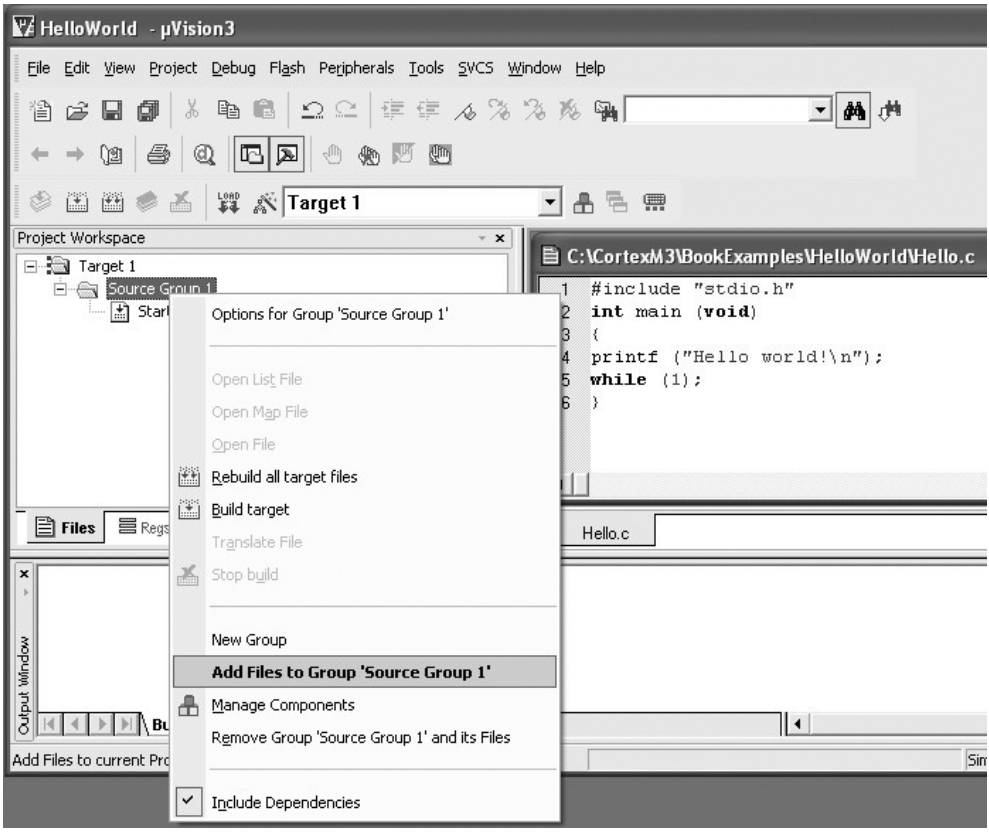


Рис. 20.8. *Добавление файла в проект.*

Выберем созданный нами файл Hello.c, после чего закроем окно **Add File**. Теперь наш проект содержит два файла (**Рис. 20.9**).

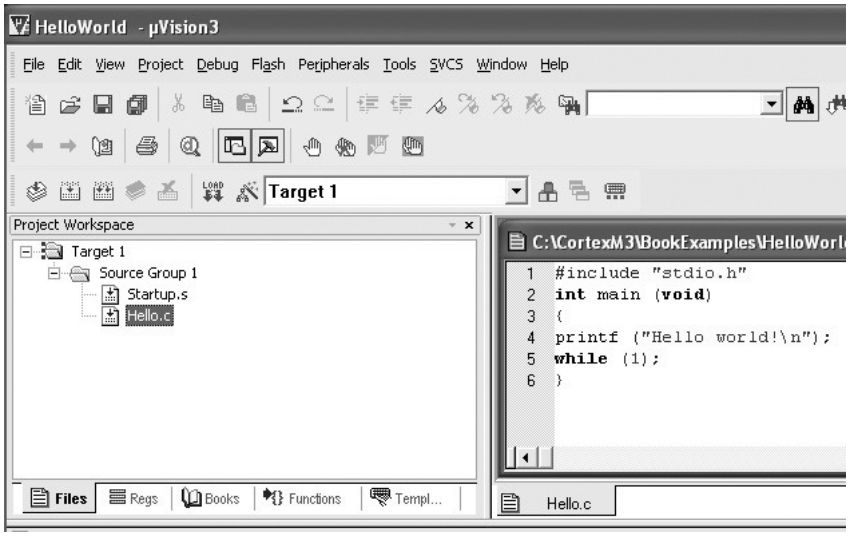


Рис. 20.9. *Окно проекта после добавления файла Hello.c.*

Для задания настроек проекта следует щёлкнуть правой кнопкой мыши на имени проекта Target 1 в окне **Project Workspace** и выбрать пункт **Options for Target 'Target 1'**. На вкладке **Target** появившегося диалогового окна мы увидим, что распределение памяти уже задано средой автоматически (**Рис. 20.10**). В этом же окне мы можем задать множество других параметров проекта, расположенных на соответствующих вкладках.

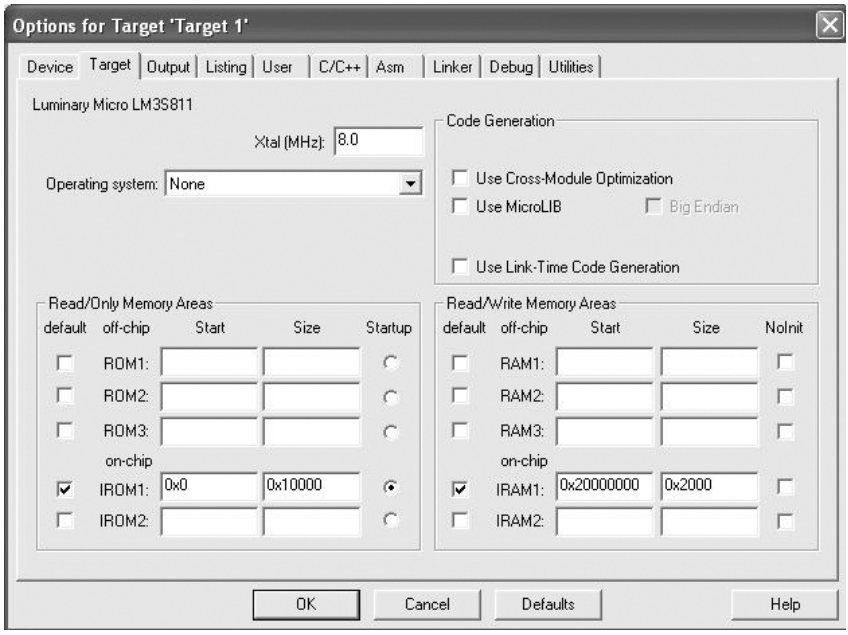


Рис. 20.10. *Диалоговое окно задания опций проекта.*

Переименование цели и групп файлов

Названия цели Target 1 и группы Source Group 1, используемые по умолчанию, можно заменить на более осмысленные. Для этого достаточно щёлкнуть на названии в окне **Project Workspace** и ввести другое имя.

Теперь мы можем приступить к компиляции программы. Для этого нужно нажать на соответствующую кнопку панели инструментов (**Рис. 20.11**) или же выбрать пункт **Build target** из контекстного меню, появляющегося при щелчке правой кнопкой мыши на имени проекта в окне **Project Workspace**.

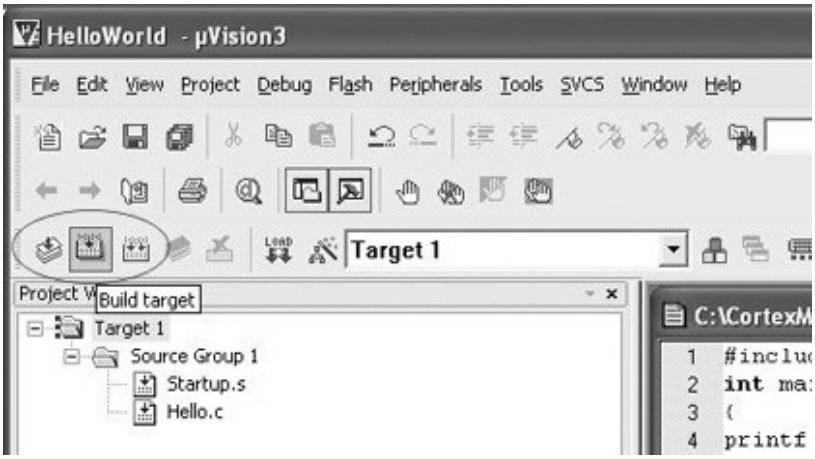


Рис. 20.11. *Запуск компиляции.*

Сообщение компилятора об успешном завершении процесса компиляции будет выведено в окне **Output Window** (**Рис. 20.12**).

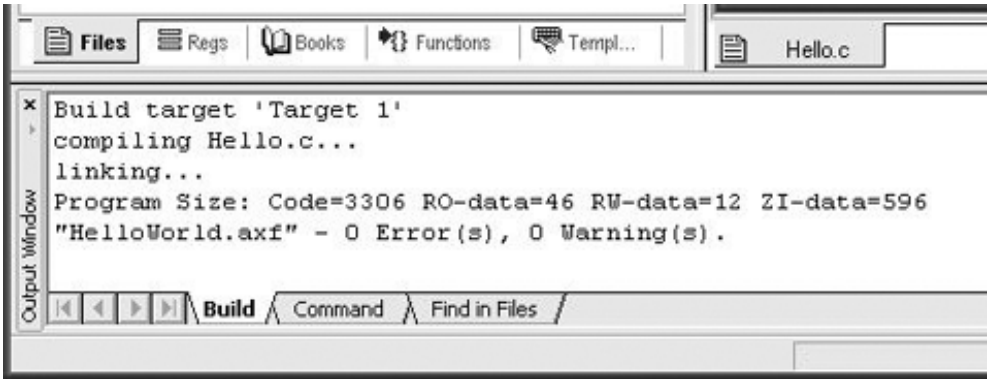


Рис. 20.12. *Вывод результата компиляции.*

20.3. Вывод сообщения «Hello World» по интерфейсу UART

В нашей программе используется функция `printf` из стандартной библиотеки языка Си. Эта библиотека не располагает никакой информацией об используемом аппаратном обеспечении. Поэтому для вывода текстовых сообщений посредством реальных устройств, таких как модуль UART микроконтроллера, нам придётся написать соответствующие процедуры.

Как уже упоминалось ранее в книге, реализация вывода информации на реальные аппаратные средства часто обозначается термином «перенаправление». Помимо собственно вывода текста, программный код поддержки перенаправления также может содержать функции для обработки ошибок и аварийного завершения программы. В данном примере мы коснёмся только функций вывода текстовой информации.

Следующая программа выводит сообщение «Hello World» в модуль UART0 микроконтроллера LM3S811. Эта программа рассчитана на соответствующую отладочную плату производства Texas Instruments. Источником тактового сигнала служит имеющийся на плате кварцевый резонатор частотой 6 МГц. Встроенный модуль PLL микроконтроллера после некоторой настройки формирует из этого сигнала системный тактовый сигнал частотой 50 МГц. Скорость передачи UART задаётся равной 115 200 бит/с; для приёма информации от UART используется программа HyperTerminal, работающая на ПК под управлением ОС Windows.

Чтобы перенаправить поток данных, формируемый функцией `printf`, нам необходимо реализовать свою функцию `fputc`. Ниже приведена возможная реализация этой функции, использующая для вывода символов в модуль UART функцию `sendchar`.

Файл hello.c

```
#include «stdio.h»
#pragma import(__use_no_semihosting_swi)
struct __FILE { int handle; };
FILE __stdout;
#define CR    0x0D // Возврат каретки
#define LF    0x0A // Перевод строки
void Uart0Init(void);
void SetClockFreq(void);
int sendchar(int ch);
// При использовании тактового сигнала 6 МГц закомментируйте следующую строку
#define CLOCK50MHZ
// Адреса регистров
#define SYSCTRL_RCC     *((volatile unsigned long *)(0x400FE060))
#define SYSCTRL_RIS     *((volatile unsigned long *)(0x400FE050))
#define SYSCTRL_RCGC1   *((volatile unsigned long *)(0x400FE104))
#define SYSCTRL_RCGC2   *((volatile unsigned long *)(0x400FE108))
#define GPIOPA_AFSEL    *((volatile unsigned long *)(0x40004420))
#define UART0_DATA      *((volatile unsigned long *)(0x4000C000))
#define UART0_FLAG      *((volatile unsigned long *)(0x4000C018))
#define UART0_IBRD      *((volatile unsigned long *)(0x4000C024))
```

```
#define UART0_FBRD      *((volatile unsigned long *)(0x4000C028))
#define UART0_LCRH      *((volatile unsigned long *)(0x4000C02C))
#define UART0_CTRL      *((volatile unsigned long *)(0x4000C030))
#define UART0_RIS       *((volatile unsigned long *)(0x4000C03C))
#define NVIC_CCR        *((volatile unsigned long *)(0xE000ED14))
int main (void)
{ // Простой код для вывода сообщения «Hello World»
  NVIC_CCR = NVIC_CCR | 0x200; // Устанавливаем STKALIGN
  SetClockFreq();              // Задаём частоту тактового сигнала (50 МГц/6 МГц)
  Uart0Init();                 // Инициализируем UART0
  printf («Hello world!\n»);
  while (1);
}
void SetClockFreq(void)
{
#ifdef CLOCK50MHZ
  // Устанавливаем бит BYPASS, сбрасываем биты USRSYSDIV и SYSDIV
  SYSCTRL_RCC = (SYSCTRL_RCC & 0xF83FFFFF) | 0x800 ;
  // Сбрасываем биты OSCSRC, PWRDN и OEN
  SYSCTRL_RCC = (SYSCTRL_RCC & 0xFFFFCFCF);
  // Изменяем SYSDIV, устанавливаем USRSYSDIV и частоту кварца
  SYSCTRL_RCC = (SYSCTRL_RCC & 0xF87FFC3F) | 0x01C002C0;
  while ((SYSCTRL_RIS & 0x40)==0); // Ждём установки бита PLLLRIS
  // Сбрасываем бит BYPASS
  SYSCTRL_RCC = (SYSCTRL_RCC & 0xFFFFF7FF) ;
#else
  // Устанавливаем бит BYPASS, сбрасываем биты USRSYSDIV и SYSDIV
  SYSCTRL_RCC = (SYSCTRL_RCC & 0xF83FFFFF) | 0x800 ;
#endif
  return;
}
void Uart0Init(void)
{
  SYSCTRL_RCGC1 = SYSCTRL_RCGC1 | 0x0003; // Разрешаем тактирование UART0 и UART1
  SYSCTRL_RCGC2 = SYSCTRL_RCGC2 | 0x0001; // Разрешаем тактирование PORTA
  UART0_CTRL = 0;      // Запрещаем UART
#ifdef CLOCK50MHZ
  UART0_IBRD = 27;     // Задаём скорость передачи для тактовой частоты 50 МГц
  UART0_FBRD = 9;
#else
  UART0_IBRD = 3;      // Задаём скорость передачи для тактовой частоты 6 МГц
  UART0_FBRD = 17;
#endif
  UART0_LCRH = 0x60;  // 8 бит, без контроля чётности
  UART0_CTRL = 0x301; // Разрешаем приём, передачу и включаем модуль UART
  GPIOPA_AFSEL = GPIOPA_AFSEL | 0x3; // Модуль UART0 использует выводы GPIO
  return;
}
/* Вывод символа в UART0 (используется функцией printf) */
int sendchar (int ch) {
  if (ch == '\n') {
```

```
  while ((UART0_FLAG & 0x8)); // Если UART занят - ждём
  UART0_DATA = CR;            // Выводим дополнительный символ CR для корректного
  }                           // отображения на экране
  while ((UART0_FLAG & 0x8)); // Если модуль занят - ждём
  return (UART0_DATA = ch);   // Выводим байт
}
/* Перенаправление текстового вывода */
int fputc(int ch, FILE *f) {
  return (sendchar(ch));
}
void _sys_exit(int return_code) {
  /* Заглушка */
  label: goto label; /* Бесконечный цикл */
}
```

Подпрограмма `SetClockFreq` конфигурирует модуль PLL для формирования системного тактового сигнала частотой 50 МГц (процесс настройки зависит от используемого устройства). Если константа `CLOCK50MHZ` не определена, то эта подпрограмма устанавливает тактовую частоту, равную 6 МГц.

Для инициализации модуля UART0 используется подпрограмма `Uart0Init`. Настройка модуля заключается в выборе скорости передачи (115 200 бит/с), разрядности слова (8 бит) и числа стоп-битов (1 стоп-бит), а также отключении контроля чётности. Эта же подпрограмма переключает выводы порта A, используемые модулем UART0, в режим альтернативных функций. Прежде чем можно будет обращаться к модулям GPIO и UART, необходимо включить тактирование данных модулей. Это осуществляется записью соответствующих значений в регистры SYSCTRL_RCGC1 и SYSCTRL_RCGC2.

Перенаправление осуществляется функцией `fputc` (это имя зарезервировано для функции, осуществляющей вывод символов). Данная функция вызывает функцию `sendchar` для вывода символа в модуль UART. При обнаружении символа новой строки указанная функция выводит дополнительный символ возврата каретки. Это необходимо для корректного отображения текста в окне HyperTerminal (в противном случае, текст новой строки перезаписывал бы предыдущую строку).

После внесения изменений в файл Hello.c перекомпилируем программу.

20.4. Тестирование программы

Если у вас уже есть оценочная плата LM3S811, то вы можете проверить работу программы, загрузив её во флэш-память микроконтроллера и убедившись в наличии сообщения «Hello World» в окне HyperTerminal. Для загрузки и проверки программы необходимо выполнить следующие действия (предполагается, что все драйверы, поставлявшиеся вместе с оценочной платой, установлены).

Прежде всего, необходимо настроить опции загрузки во флэш-память, выбрав пункт **Configure Flash Tools** из меню **Flash** (**Рис. 20.13**). В появившемся диалоговом окне выберите оценочную плату Luminary Eval Board (**Рис. 20.14**). Как видно из рисунка, μVision поддерживает самые разные средства отладки.

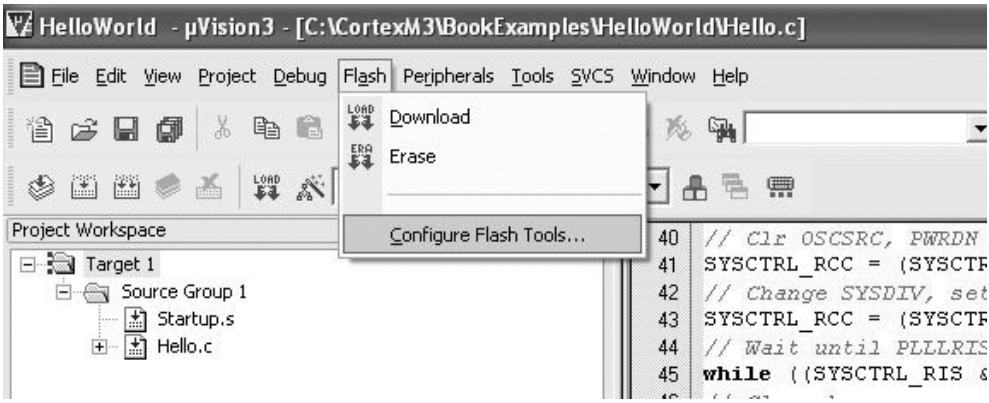


Рис. 20.13. *Вызов окна настройки программатора.*

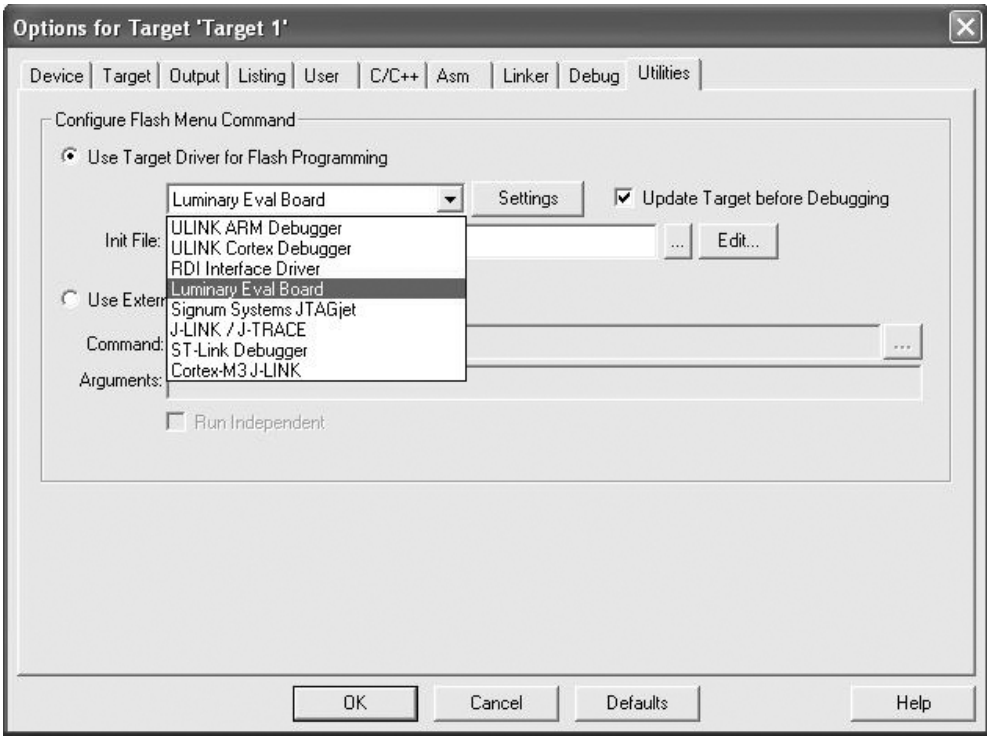


Рис. 20.14. *Выбор драйвера программатора.*

После выбора оценочной платы необходимо щёлкнуть по кнопке **Setting**, чтобы убедиться в корректности используемых настроек (**Рис. 20.15**).

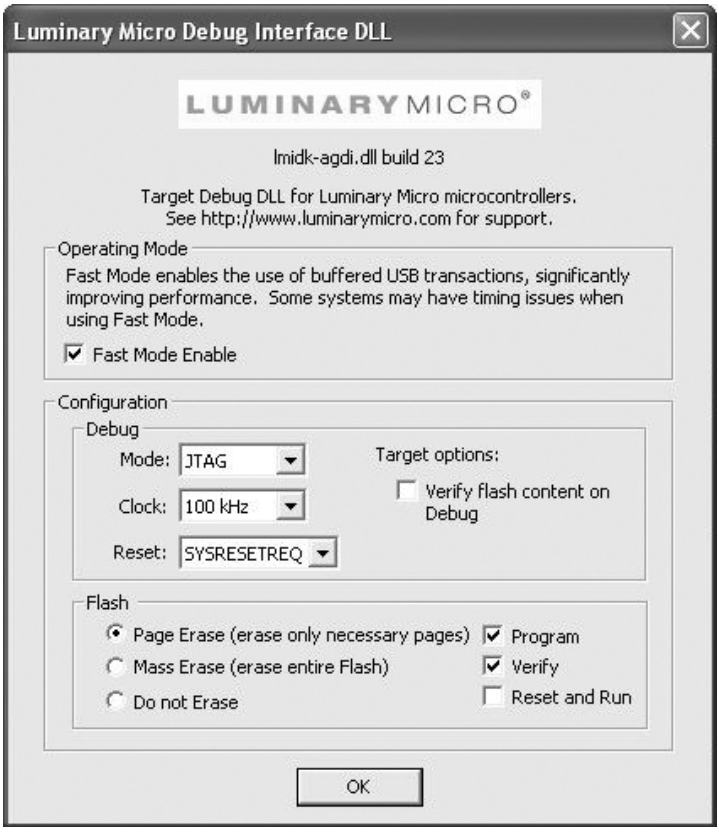


Рис. 20.15. *Окно выбора параметров программирования флэш-памяти.*

Теперь мы можем загрузить нашу программу в микроконтроллер, выбрав в меню **Flash** пункт **Download**. После завершения программирования в окне **Output Window** появится сообщение, показанное на **Рис. 20.16**. Если между платой и HyperTerminal уже была установлена связь, то перед программированием флэш-памяти может потребоваться закрыть программу, отключить USB-кабель от компьютера и повторно подключить его.

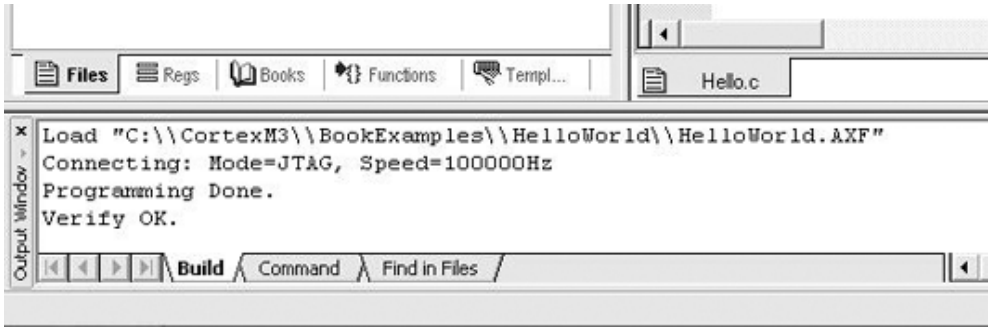


Рис. 20.16. *Вывод сообщения об удачном программировании.*

После завершения программирования вы можете запустить HyperTerminal и, подключив плату к виртуальному COM-порту (по USB), получить текстовое сообщение, переданное программой, выполняющейся в микроконтроллере (**Рис. 20.17**).

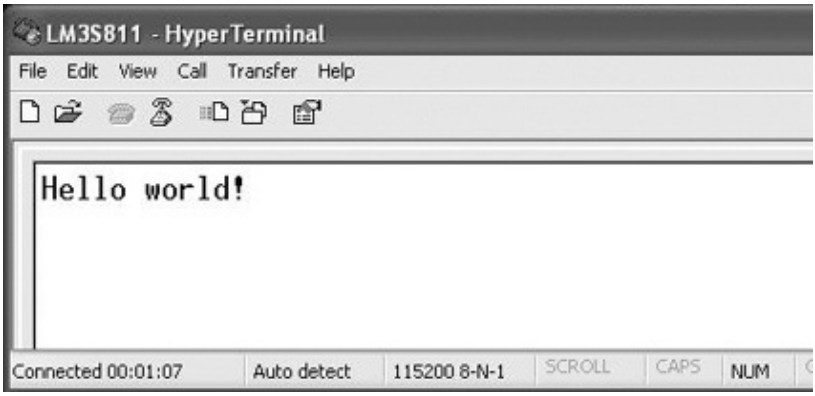


Рис. 20.17. *Сообщение в окне программы HyperTerminal.*

20.5. Использование отладчика

Отладчик в ИСР μVision поддерживает множество различных аппаратных средств отладки, в том числе и изделия линейки ULINK (ULINK-2, ULINK-PRO и ULINK-ME) компании Keil (**Рис. 20.18**).

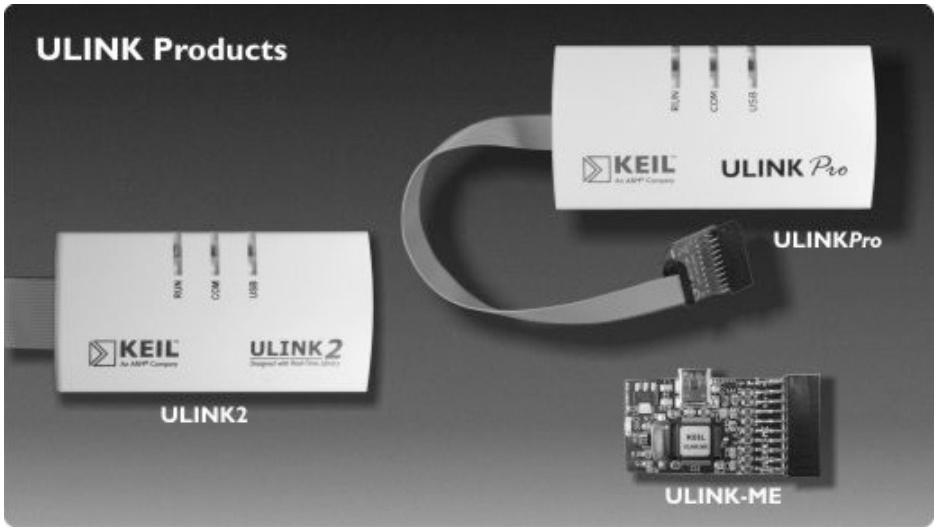


Рис. 20.18. *Отладочные средства линейки ULINK.*

Среда μVision поддерживает следующие внутрисхемные отладчики:

- JTAGJet и JTAGJet-Trace (компания Signum);
- J-Link и J-Trace (компания Segger);

- оценочные платы Stellaris (компания Texas Instrument, бывшая Luminary Micro);
- ST-Link (компания STMicroelectronics).

Для отладки нашего примера мы будем использовать отладчик ИСР µVision в связке с аппаратным отладчиком, входящим в состав оценочной платы Texas Instruments. Для изменения параметров отладки необходимо щёлкнуть правой кнопкой мыши на имени проекта Target1 в окне **Project Workspace** и выбрать из контекстного меню пункт **Options for Target 'Target 1'**. Перейдём на вкладку **Debug** и выберем отладчик Luminary Eval Board (**Рис. 20.19**).

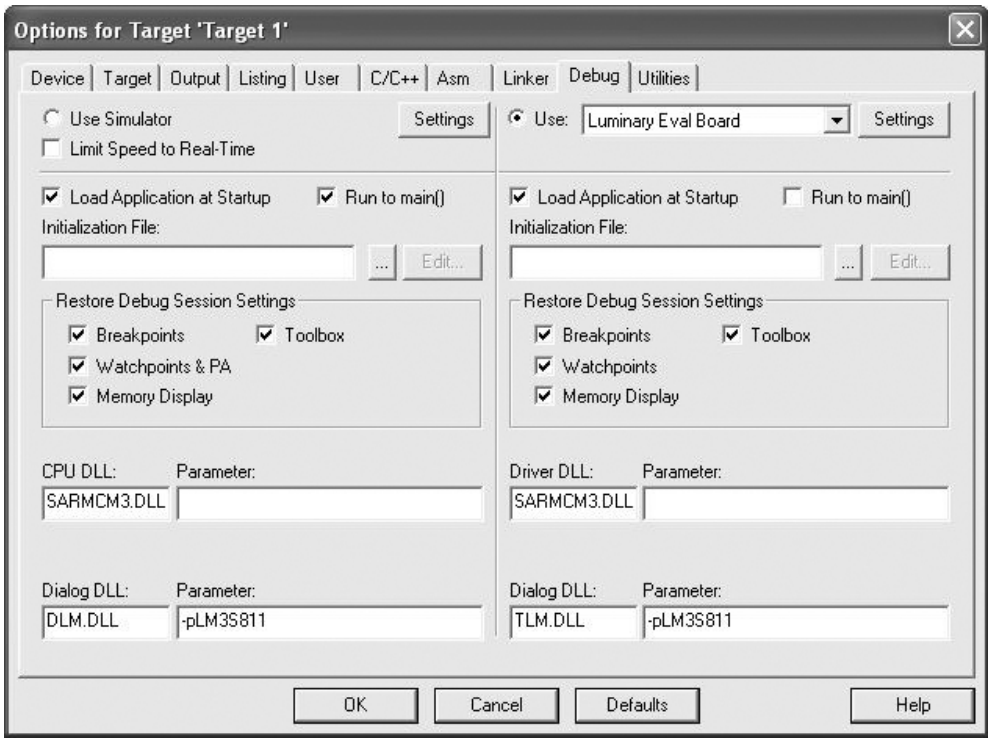


Рис. 20.19. *Выбор отладчика.*

Если после выбора отладчика нажать на кнопку **Settings**, то можно увидеть диалоговое окно настроек параметров загрузчика флэш-памяти, показанное на **Рис. 20.15**. В данном окне вы можете выбрать используемый протокол отладки, тактовую частоту канала обмена и ряд других параметров. При использовании отладчика ULINK-2/ULINK-Pro этих опций будет гораздо больше (**Рис. 20.20**).

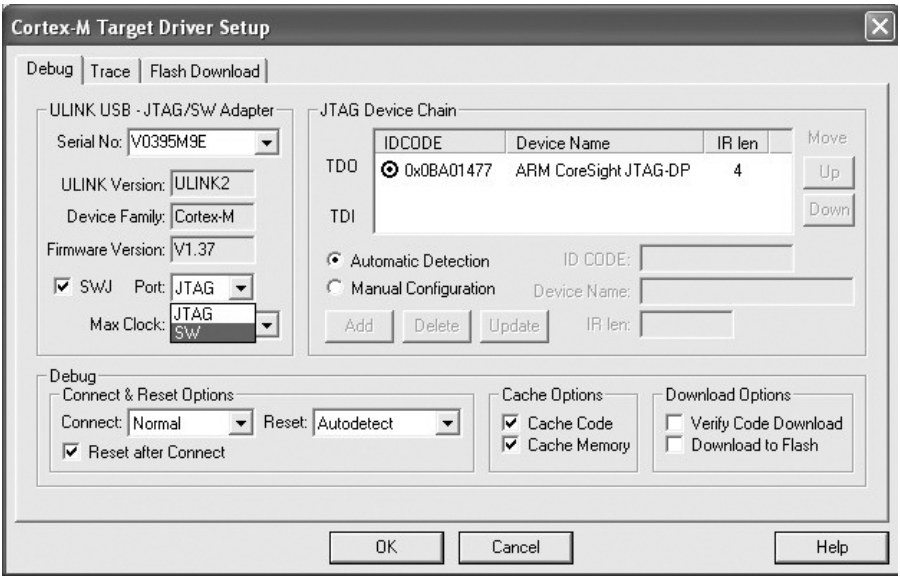


Рис. 20.20. *Опции отладчика ULINK-2.*

Теперь мы можем начать сессию отладки, выбрав в меню **Debug** пункт **Start/Stop Debug Session** (**Рис. 20.21**).

Примечание. Если используется виртуальный COM-порт на базе микросхемы FTDI и плата уже осуществляет обмен данными с программой HyperTerminal, то перед запуском сессии отладки нужно будет закрыть программу, отключить USB-кабель от ПК и снова подключить его.

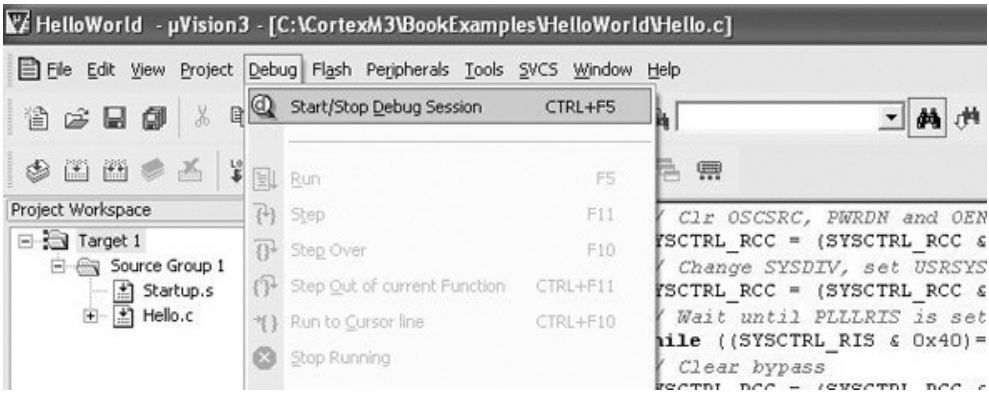


Рис. 20.21. *Запуск сессии отладки.*

После запуска отладчика в ИСР появится окно, отображающее содержимое регистров процессора. Открыв окно с дизассемблированным кодом, вы также можете увидеть адрес текущей команды. Из **Рис. 20.22** видно, что выполнение программы остановилось на метке `Reset _ Handler`, соответствующей первой команде исполняемого кода.

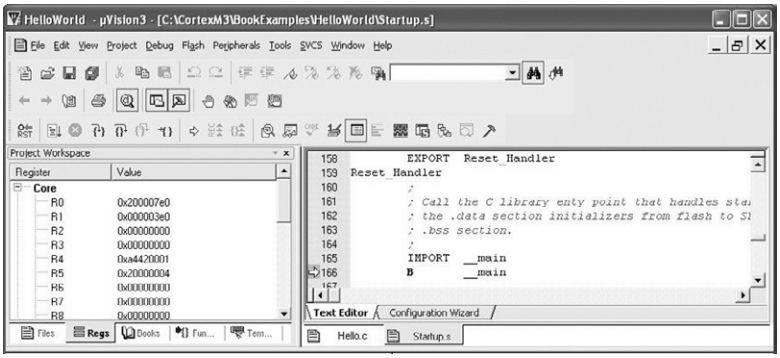


Рис. 20.22. *Среда отладки в ИСР µVision.*

Для установки точки останова в начале функции `main` можно щёлкнуть правой кнопкой мыши на 1-й строке кода основной программы и выбрать в контекстном меню пункт **Insert/Remove Breakpoint** (**Рис. 20.23**).

Примечание. Чтобы выполнение программы всегда останавливалось в начале функции `main`, можно также задействовать опцию отладчика `Run to main()`.

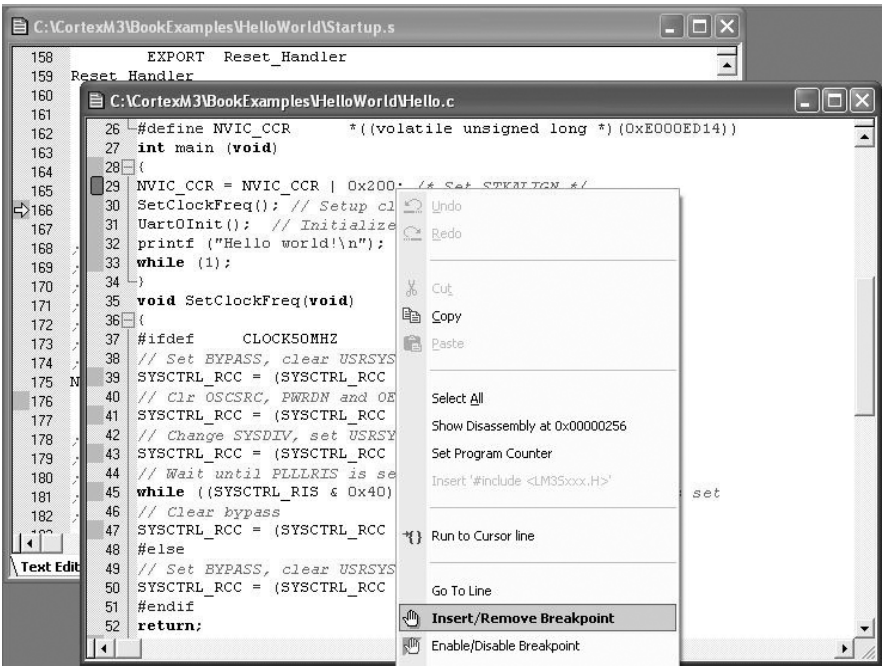


Рис. 20.23. *Установка/снятие точек останова.*

После установки точки останова слева от строки кода появится красный кружок (см. строку 29 на **Рис. 20.23**). Теперь мы можем запустить программу на выполнение, нажав кнопку **Run** на панели инструментов (**Рис. 20.24**).

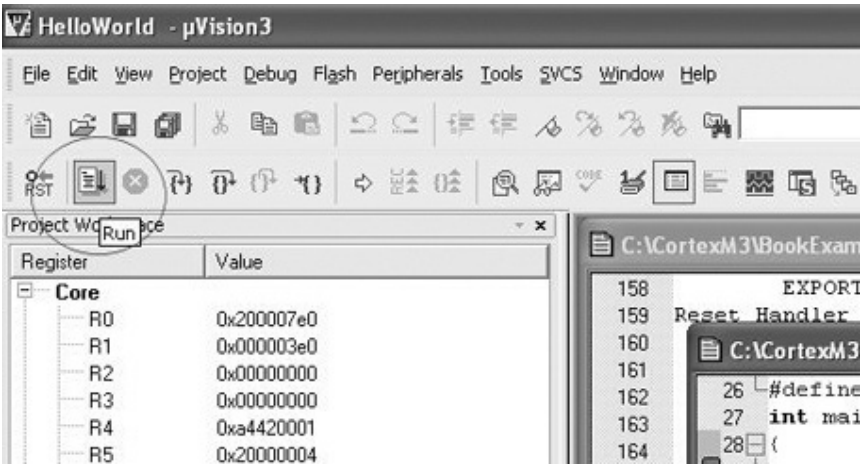


Рис. 20.24. *Запуск программы на выполнение.*

Программа начнёт выполняться и остановится при достижении функции `main` (**Рис. 20.25**). После этого мы сможем приступить к пошаговой отладке своего приложения (используя соответствующие кнопки панели инструментов), контролируя работу программы по содержимому регистров.

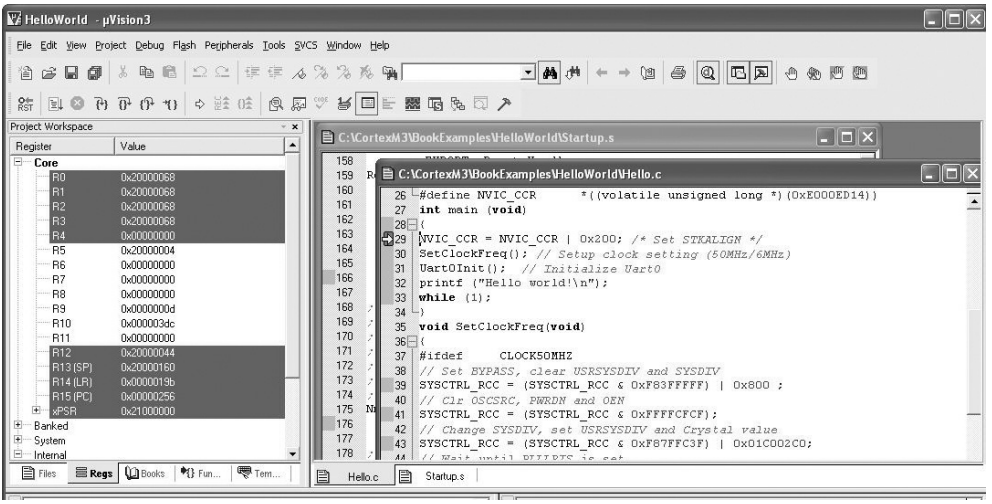


Рис. 20.25. *Прекращение выполнения программы в точке останова.*

20.6. Симулятор

В составе ИСР µVision имеется симулятор команд, который тоже можно использовать для отладки приложений. Работа в симуляторе во многом идентична работе с отладчиком и служит прекрасным подспорьем в изучении процессора Cortex-M3. Чтобы задействовать симулятор, выберите опцию отладки **Use Simulator** (**Рис. 20.26**). Сразу хочу предупредить, что симулятор может оказаться

неспособным симулировать работу определённых периферийных модулей ряда микроконтроллеров. Поэтому код, работающий с интерфейсом UART, может симулироваться некорректно.

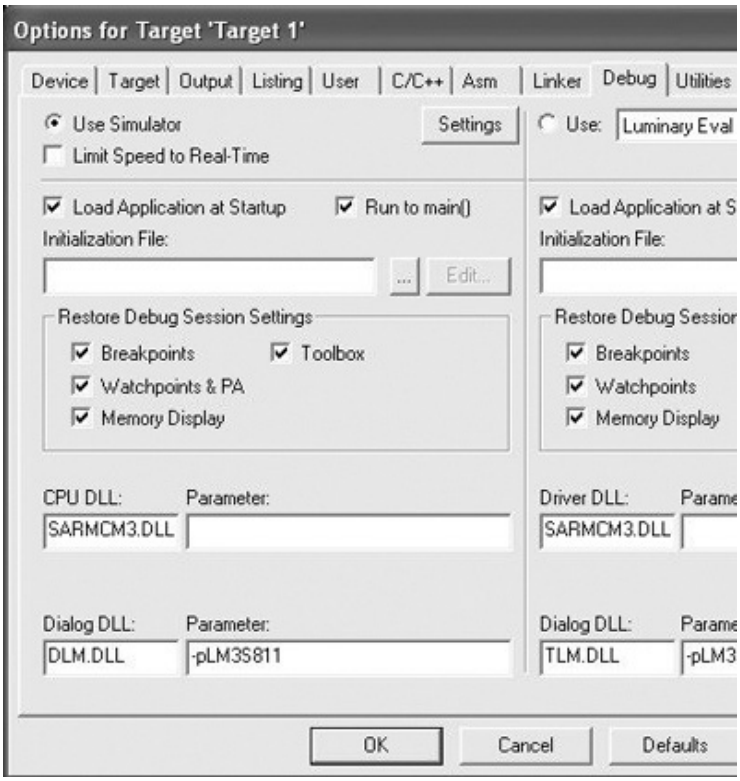


Рис. 20.26. *Выбор симулятора в качестве отладчика.*

Симулятор µVision имеет полную поддержку на уровне устройства различных микроконтроллеров с ядром Cortex-M3. В тех редких случаях, когда симуляция всего устройства оказывается недоступной, для отладки программы с использованием симулятора может потребоваться корректировка настроек памяти. Это можно сделать, выбрав в меню **Debug** пункт **Memory Map** (**Рис. 20.27**).

Например, вам может потребоваться включить в карту памяти диапазон адресов модуля UART (**Рис. 20.28**). Если этого не сделать, то при попытке обращения к данному модулю вы получите сообщение об ошибке. Однако в большинстве случаев все необходимые настройки карты памяти задаются автоматически при выборе целевого микроконтроллера и вручную обычно ничего делать не требуется.

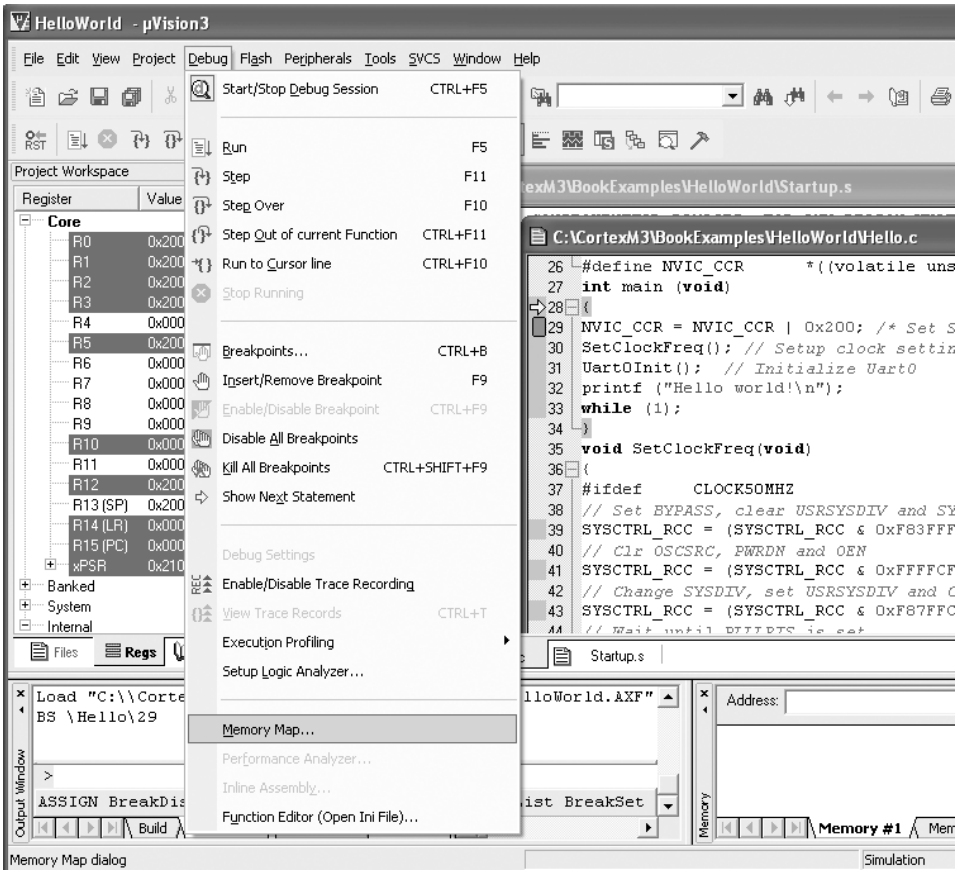


Рис. 20.27. *Вызов окна параметров карты памяти.*

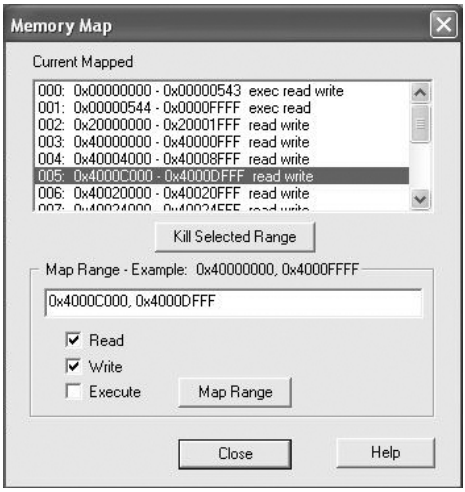


Рис. 20.28. *Настройка карты памяти для симулятора.*

20.7. Модификация таблицы векторов

В предыдущем примере мы использовали таблицу векторов, определённую в файле со стартовым кодом `startup.s`. Этот файл содержит таблицу векторов, а также заранее определённые обработчики сброса, немаскируемого прерывания, исключения Hard Fault и обычного прерывания. Указанные обработчики необходимо переписать в соответствии с требованиями приложения. Например, если в приложении используется прерывание от периферийного устройства, то необходимо изменить таблицу векторов таким образом, чтобы при возникновении прерывания запускалась бы ваша процедура обработки прерывания.

Используемые по умолчанию обработчики исключений написаны на ассемблере и находятся в файле `startup.s`. Однако ничто не мешает реализовать их на языке Си или же на ассемблере, но в другом файле. При этом в файле `startup.s` необходимо будет использовать директиву ассемблера `IMPORT` для указания того, что метка начала обработчика исключения определена в другом файле.

К примеру, если мы хотим добавить обработчики исключений SYSTICK и модуля USART, то мы можем изменить файл `startup.s` следующим образом (**Рис. 20.29**):

- закомментировать соответствующие строки таблицы векторов;
- вставить в программу директивы `IMPORT` для двух векторов исключений, обработчики которых будут описаны на языке Си. Это необходимо, если обработчики находятся в другом Си- или ассемблерном файле;
- добавить в таблицу вектора исключений, используя директиву `DCD`.

```
097         DCD     0                       ; Reserved
098         DCD     IntDefaultHandler       ; SVCall Handler
099         DCD     IntDefaultHandler       ; Debug Monitor Handler
100         DCD     0                       ; Reserved
101         DCD     IntDefaultHandler       ; PendSV Handler
102         ;DCD     IntDefaultHandler       ; SysTick Handler
103         IMPORT  SysTickHandler
104         DCD     SysTickHandler
105         DCD     IntDefaultHandler       ; GPIO Port A
106         DCD     IntDefaultHandler       ; GPIO Port B
107         DCD     IntDefaultHandler       ; GPIO Port C
108         DCD     IntDefaultHandler       ; GPIO Port D
109         DCD     IntDefaultHandler       ; GPIO Port E
110         ;DCD     IntDefaultHandler       ; UART0
111         IMPORT  Uart0Handler
112         DCD     Uart0Handler
113         DCD     IntDefaultHandler       ; UART1
114         DCD     IntDefaultHandler       ; SSI
115         DCD     IntDefaultHandler       ; I2C
116         DCD     IntDefaultHandler       ; PWM Fault
```

Рис. 20.29. *Вставка векторов прерываний в ассемблерный стартовый код.*

При использовании CMSIS-совместимых библиотек драйверов устройств такая корректировка файла `startup.s` не требуется, если, конечно же, имена обработчиков исключений соответствуют именам функций, указанных в стартовом коде CMSIS. Применение стартового кода CMSIS будет продемонстрировано в следующем примере.

20.8. Прерывания и стандарт CMSIS

В данном примере, реализующем секундомер, используются исключение SYSTICK и прерывание модуля UART0. Этот секундомер имеет три состояния, показанные на **Рис. 20.30**.

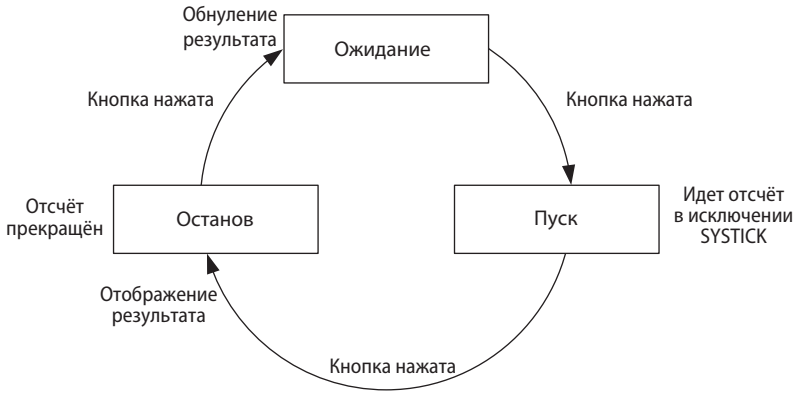


Рис. 20.30. *Конечный автомат секундомера.*

Управление секундомером осуществляется с персонального компьютера по интерфейсу UART. Чтобы упростить код программы, зафиксируем тактовую частоту на уровне 50 МГц.

Измерение временных интервалов осуществляется таймером SYSTICK, который прерывает выполнение программы каждые 100 мс. Для тактирования таймера используется тактовый сигнал ядра частотой 50 МГц. При каждом запуске обработчика исключения SYSTICK инкрементируется переменная `TickCounter` (если секундомер запущен).

Поскольку передача текстовой информации по интерфейсу UART является относительно медленным процессом, то управление секундомером производится непосредственно в обработчике исключения, а отображение текста и значений секундомера осуществляется в основной программе (уровень потока). Для управления пуском, остановом и сбросом секундомера используется простой конечный автомат. Обработка состояний этого автомата производится в обработчике прерывания UART, который запускается по каждому приёму символа.

Давайте создадим новый проект и назовём его StopWatch. Добавим в него файл `stopwatch.c`, а также несколько других файлов поддержки стандарта CMSIS.

Заменим файл `startup.s` файлом `startup_rvmdk.s` и поместим в файл `stopwatch.c` директиву вставки файла `lm3s_cmsis.h`, как показано на **Рис. 20.31**. Также добавим в проект файлы `system_lm3s.c` и `core_lm3s.c`. Первый из них содержит функцию `SystemInit()`, используемую в программе. Файл `core_lm3s.c` содержит встроенные функции. Несмотря на то что мы не применяем ни одной из этих функций, мы всё же включили данный файл в проект для полноты картины.

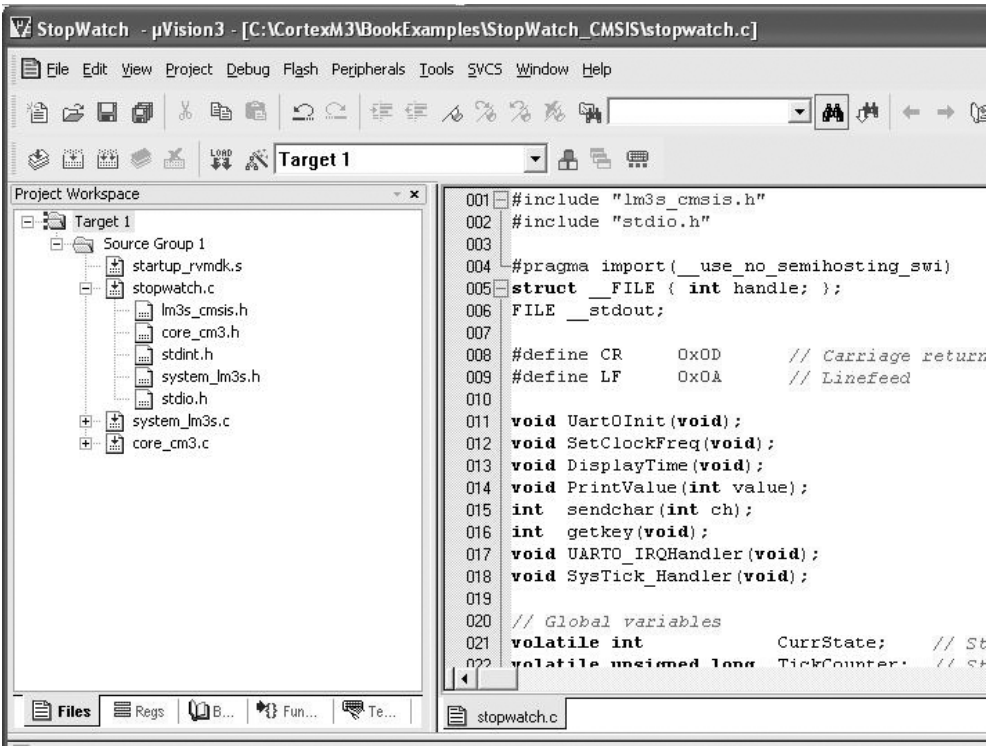


Рис. 20.31. *Окно проекта «StopWatch», использующего CMSIS.*

Имена некоторых файлов специфичны для производителя микроконтроллеров (например, `startup _ rvmdk.s` и `lm3s _ cmsis.h`). Эти файлы, используемые в проекте, можно взять из CMSIS-совместимой библиотеки драйвера устройства или непосредственно из пакета файлов поддержки стандарта CMSIS, которые можно загрузить с сайта *www.onarm.com*.

Использование стандарта CMSIS упрощает программу `stopwatch.c`, поскольку определения регистров периферийных устройств и некоторые системные функции содержатся в файлах CMSIS.

```
stopwatch.c
--------------
#include «lm3s_cmsis.h» // Включаемый файл поддержки CMSIS
#include «stdio.h»      // Для функции printf
#pragma import(__use_no_semihosting_swi)
struct __FILE { int handle; };
FILE __stdout;
// Специальные символы для функции вывода текста
#define CR 0x0D    // Возврат каретки
#define LF 0x0A    // Перевод строки
// Объявления функций
void Uart0Init(void);            // Инициализация UART0
void SetClockFreq(void);         // Установка тактовой частоты 50 МГц
void DisplayTime(void);          // Отображение времени
```

```
int sendchar(int ch);             // Вывод символа в UART
int getkey(void);                 // Чтение символа из UART
void UART0_IRQHandler(void);      // Обработчик прерывания UART0
void SysTick_Handler(void);       // Обработчик исключения SysTick
// Глобальные переменные
volatile int CurrState;               // Состояние конечного автомата
volatile unsigned long TickCounter;  // Счётчик секундомера
volatile int KeyReceived;             // Признак нажатия клавиши
volatile int userinput;               // Значение нажатой клавиши
// Состояния конечного автомата
#define IDLE_STATE 0
#define RUN_STATE  1
#define STOP_STATE 2
int main (void)
{
  int CurrStateLocal; // Локальная копия текущего состояния

  SystemInit();     // Инициализация системы – часть стандарта CMSIS
                    // Не требуется при использовании CMSIS v1.30 или выше,
                    // поскольку в этих версиях вызывается из стартового кода
  SetClockFreq();   // Настройка тактовой частоты (50 МГц)
  // Инициализируем глобальные переменные
  CurrState = 0;
  KeyReceived = 0;
  // Инициализируем аппаратуру
  SCB->CCR = SCB->CCR | 0x200; // Устанавливаем STKALIGN
  Uart0Init();                 // Инициализируем Uart0
  SysTick_Config(499999);      // Инициализируем SysTick (функция CMSIS)
  printf («Секундомер\n»);
  while (1) {
    CurrStateLocal = CurrState; // Делаем локальную копию, поскольку
    // это значение может быть в любой момент изменено обработчиком UART.
    switch (CurrStateLocal) {
      case (IDLE_STATE):
        printf («\nНажмите любую клавишу для пуска\n»);
        break;
      case (RUN_STATE):
        printf («\nНажмите любую клавишу для останова\n»);
        break;
      case (STOP_STATE):
        printf («\nНажмите любую клавишу для сброса\n»);
        break;
      default:
        CurrState = IDLE_STATE;
        break;
    } // Конец switch()
    while (KeyReceived == 0) {
      if (CurrStateLocal==RUN_STATE){
        DisplayTime();
      }
    }; // Ждём действий пользователя
```

```
    if (CurrStateLocal==STOP_STATE) {
      TickCounter=0;
      DisplayTime(); // Выводим нулевое значение
    }
    else if (CurrStateLocal==RUN_STATE) {
      DisplayTime(); // Выводим результат
    }
    if (KeyReceived!=0) KeyReceived=0;
  }; // Конец while()
} // Конец функции main()
// Конфигурирование тактового сигнала процессора и UART
void SetClockFreq(void)
{
  // Устанавливаем BYPASS, сбрасываем USRSYSDIV и SYSDIV
  SYSCTL->RCC = (SYSCTL->RCC & 0xF83FFFFF) | 0x800 ;
  // Сбрасываем OSCSRC, PWRDN и OEN
  SYSCTL->RCC = (SYSCTL->RCC & 0xFFFFCFCF);
  // Меняем SYSDIV, задаём значения USRSYSDIV и Crystal
  SYSCTL->RCC = (SYSCTL->RCC & 0xF87FFC3F) | 0x01C002C0;
  while ((SYSCTL->RIS & 0x40)==0); // Ждём установки PLLLRIS
  // Сбрасываем BYPASS
  SYSCTL->RCC = (SYSCTL->RCC & 0xFFFFF7FF);
  return;
}
// Инициализация UART0
void Uart0Init(void)
{ // Тактовый сигнал для работы UART
  SYSCTL->RCGC1 = SYSCTL->RCGC1 | 0x0003; // Включаем тактирование UART0 и UART1
  SYSCTL->RCGC2 = SYSCTL->RCGC2 | 0x0001; // Включаем тактирование PORTA
  // Настраиваем UART
  UART0->CTL = 0;                    // Отключаем UART
  UART0->IBRD = 27;                  // Программируем скорость передачи
  UART0->FBRD = 9;                   // для тактовой частоты 50 МГц
  UART0->LCRH = 0x60;                // 8 бит, без контроля чётности
  UART0->CTL = 0x301;                // Разрешаем передачу и приём, включаем UART
  UART0->IM = 0x10;                  // Разрешаем прерывание UART по приёму
  GPIOA->AFSEL = GPIOA->AFSEL | 0x3; // Назначаем выводы GPIO модулю UART0
  NVIC_EnableIRQ(UART0_IRQn);        // Разрешаем прерывание UART в NVIC
                                     // (функция CMSIS)
  return;
}
// Обработчик исключения SYSTICK
void SysTick_Handler(void) // Имя функции соответствует определённому
                           // в стандарте CMSIS
{
  if (CurrState==RUN_STATE) {
    TickCounter++;
  }
  return;
}
// Обработчик прерывания приёмника UART0
```

```
void UART0_IRQHandler(void) // Имя функции объявлено в стартовом коде CMSIS
{
  userinput = getkey();
  // Ставим флаг приёма символа
  KeyReceived++;
  // Сбрасываем запрос прерывания UART
  UART0->ICR = 0x10;
  // Обрабатываем состояния конечного автомата
  switch (CurrState) {
    case (IDLE_STATE):
      CurrState = RUN_STATE;
      break;
    case (RUN_STATE):
      CurrState = STOP_STATE;
      break;
    case (STOP_STATE):
      CurrState = IDLE_STATE;
      break;
    default:
      CurrState = IDLE_STATE;
      break;
  } // Конец switch()
  return;
}
// Выводит значение счётчика времени
void DisplayTime(void)
{
  unsigned long TickCounterCopy;
  unsigned long TmpValue;

  sendchar(CR);
  TickCounterCopy = TickCounter; // Делаем локальную копию, поскольку значение
      // может быть в любой момент изменено в обработчике SYSTICK
  TmpValue = TickCounterCopy / 6000; // Минуты
  printf («%d», TmpValue);
  TickCounterCopy = TickCounterCopy – (TmpValue * 6000);
  TmpValue = TickCounterCopy / 100; // Секунды
  sendchar(':');
  printf («%d», TmpValue);
  TmpValue = TickCounterCopy – (TmpValue * 100);
  sendchar(':');
  printf («%d», TmpValue); // Сотые доли секунды
  sendchar(' ');
  sendchar(' ');
  return;
}
// Выводит символ в UART0 (используется функцией printf)
int sendchar (int ch)
{
  if (ch == '\n'){
    while ((UART0->FR & 0x8)); // Если UART занят - ждём
```

```
    UART0->DR = CR;            // Выводим дополнительный символ CR для
                               // корректного отображения в окне HyperTerminal
  }
  while ((UART0->FR & 0x8));   // Если UART занят - ждём
  return (UART0->DR = ch);     // Выводим символ
}
// Получение команд пользователя
int getkey (void)
{ // Читаем символ из последовательного порта
  while (UART0->FR & 0x10); // Если FIFO приёмника пуст - ждём
  return (UART0->DR);
}
// Перенаправляет текстовый вывод
int fputc(int ch, FILE *f)
{
  return (sendchar(ch));
}
void _sys_exit(int return_code)
{
  // Заглушка
  label: goto label; // Бесконечный цикл
}
```

По сравнению с предыдущим примером «Hello World» процедура инициализации модуля UART немного изменилась — было добавлено разрешение прерывания по приёму символа. Обратите внимание, что запрос прерывания UART нужно разрешать как в регистре маски прерывания модуля UART, так и в контроллере NVIC. Для разрешения исключения SYSTICK необходимо запрограммировать только регистр управления и состояния SYSTICK.

Также в программе появилось несколько новых подпрограмм, включая обработчики UART и SYSTICK, функцию вывода данных и процедуру инициализации SYSTICK. В зависимости от реализации периферийного устройства от обработчика может потребоваться ручной сброс запроса прерывания/исключения. В обработчике UART это осуществляется записью в регистр сброса прерывания (UART0–>ICR).

После того как программа будет скомпилирована и загружена в микроконтроллер, мы сможем убедиться в её работоспособности, подключив плату к ПК с запущенным приложением HyperTerminal. Результат работы программы представлен на **Рис. 20.32**.

Рис. 20.32. *Результат работы программы StopWatch.*

20.9. Перевод существующих приложений на стандарт CMSIS

Можно довольно легко переделать любое приложение для процессора Cortex-M3 так, чтобы оно соответствовало стандарту CMSIS. Для этого необходимо сделать следующее:

- Заменить стандартный стартовый код на стартовый код CMSIS, предназначенный для целевого микроконтроллера.
- Модифицировать проект, включив в него файлы CMSIS.
- Модифицировать программу, включив в неё заголовочный файл CMSIS.
- Заменить определения регистров соответствующими определениями стандарта CMSIS.
- Заменить существующие функции работы с периферией процессора на функции CMSIS.
- Привести имена обработчиков исключений в соответствии с именами, определёнными в стартовом коде CMSIS.
- По мере возможности заменить процедуры инициализации периферийных устройств функциями CMSIS-совместимой библиотеки драйвера устройства.

Использование стандарта CMSIS позволит сделать приложение более переносимым (об этом уже говорилось в Главе 10).

ГЛАВА 21
ПРОГРАММИРОВАНИЕ CORTEX-M3 В LabVIEW

21.1. Общие сведения

Приложения для устройств на базе процессора Cortex-M3 можно создавать не только с помощью различных языков программирования, таких как ассемблер или Си. Одной из возможных альтернатив является использование графической среды разработки LabVIEW, предлагаемой компанией National Instruments. Программы, созданные в этой среде, можно запускать как на персональном компьютере, так и на микроконтроллерах ARM, включая устройства с ядром Cortex-M3.

21.2. Знакомство с LabVIEW

Компания National Instruments предлагает несколько версий среды LabVIEW, включая версии для персональных компьютеров, работающих под управлением Windows или Linux, а также версии для встраиваемых платформ, которые поддерживают самые разные встраиваемые процессоры, включая Cortex-M3 и ARM7TDMI.

Графический язык программирования среды LabVIEW поддерживает все возможности, присущие обычным языкам программирования, такие как циклы, условные операторы и использование различных типов данных. Основным отличием работы в среде LabVIEW является то, что программы создаются в виде блок-схем. К примеру, простой цикл для вычисления суммы чисел от 1 до 10 может быть представлен блоком `For Loop` (**Рис. 21.1**).

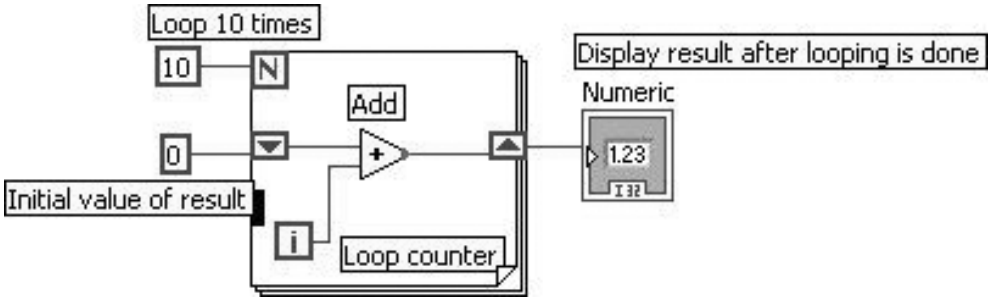


Рис. 21.1. *Простой цикл* `FOR` *для сложения чисел от 1 до 10.*

Пересылка данных изображается линиями связи (различным типам данных соответствуют разные стили линии), а переменные представляются иконками. К примеру, иконка U32 соответствует беззнаковому 32-битному целому, а иконка I32 — 32-битному целому со знаком.

Аналогом функций и подпрограмм традиционных языков программирования в среде LabVIEW являются модули, называемые *виртуальными инструментами* (VI). Передача данных в VI и вывод данных из них осуществляется через точки соединения, к которым VI более высокого уровня могут подключать входные и выходные переменные. Для примера в правой части **Рис. 21.2** показана блок-схема VI, который принимает четыре 32-битных значения и возвращает наибольшее из них. В левой части рисунка этот виртуальный инструмент используется для определения наибольшего 32-битного целого значения от четырёх источников: трёх элементов управления типа «ползунок» и одного генератора случайных чисел.

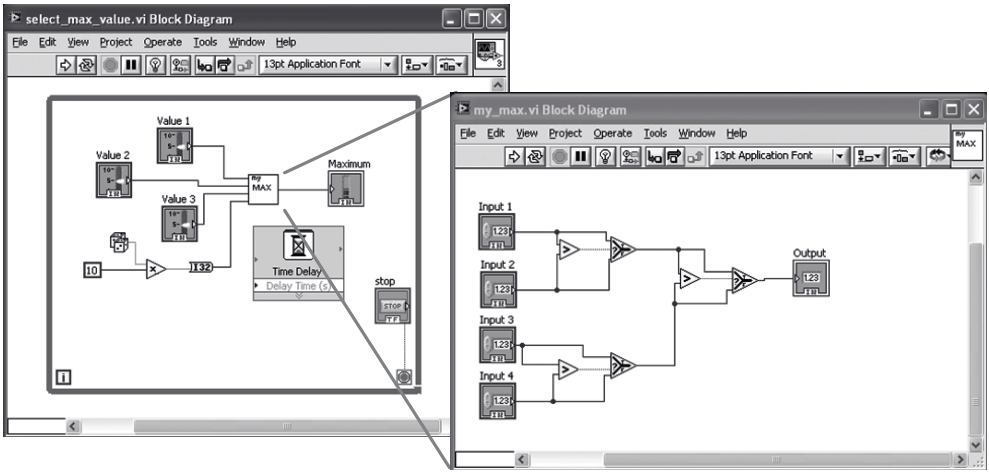


Рис. 21.2. *Блоки виртуальных инструментов VI и под-VI: несколько уровней иерархии.*

Среда программирования LabVIEW содержит большое число готовых к использованию компонентов, которые весьма облегчают разработку программы. Среди них имеются компоненты, реализующие различные функции обработки данных (такие как функция вычисления абсолютного значения `abs`), функции обработки сигналов (фильтры и элементы спектрального анализа), а также большое число компонентов для организации пользовательского интерфейса.

21.2.1. Типичные области применения

Попробуем оценить преимущества, которые нам даёт программирование в среде LabVIEW, по сравнению с традиционным программированием.

Простота использования — среда программирования LabVIEW значительно облегчает разработку сложных приложений, избавляя от необходимости досконально изучать аппаратное и программное обеспечение. Благодаря этому пользователи с самым разным уровнем подготовки, от студентов до специалистов, могут разрабатывать свои приложения, не тратя время на изучение архитектуры

процессора, а полностью сосредоточившись на разработке алгоритмов и совершенствовании функциональных возможностей устройства.

Наличие библиотеки компонентов — огромное число программных компонентов, предоставляемых средой LabVIEW, также облегчает разработку сложных приложений в сжатые сроки. Библиотека компонентов содержит сотни математических функций и функций обработки сигналов, ускоряющих разработку алгоритмов. Приложения, выполняющиеся на микроконтроллерах ARM, могут даже подключаться к графическому интерфейсу пользователя, запущенному на ПК, что облегчает управление встраиваемой системой и наблюдение за результатами её работы.

Поддержка многопоточности — графическая среда программирования по своей природе параллельна. Она позволяет запускать множество потоков в одно и то же время. В традиционных же системах программирования неопытному разработчику потребуется довольно много времени, прежде чем он сможет разобраться во встраиваемой системе реального времени (RTOS) на уровне, достаточном для создания многопоточного приложения.

Совместимость с тестовым оборудованием и оборудованием сбора данных — среда программирования LabVIEW предоставляет простые в использовании интерфейсные компоненты для связи с оборудованием сбора данных, а также поддерживает большое число интерфейсных плат, которые могут применяться как для управления промышленным оборудованием, так и для учебных целей.

Среда LabVIEW очень популярна в университетах, лабораториях и различных НИИ. Она широко используется в системах сбора данных, для автоматизации тестовых систем, для разработки и моделирования алгоритмов, для управления промышленным оборудованием, а также для создания прототипов встраиваемых систем.

21.2.2. Что нам нужно, чтобы использовать LabVIEW и ARM

Чтобы создавать программы для микроконтроллера на базе процессора Cortex-M3, нам прежде всего необходим модуль «LabVIEW Emmbedded Module for ARM Microcontrollers», в состав которого входит уже знакомый нам пакет MDK-ARM компании Keil. Для начала вы можете приобрести недорогой отладочный комплект (**Рис. 21.3**), в который входит печатная плата с установленным на ней микроконтроллером ARM (Cortex-M3/ARM7), отладчик ULINK2 от Keil, а также всё необходимое для разработки ПО (пробная версия пакета MDK-ARM имеется на прилагаемом компакт-диске). Более подробно об этом комплекте можно узнать, обратившись на сайт компании National Instruments (*www.ni.com/arm*).

Вы можете использовать среду LabVIEW для написания программ под любые микроконтроллеры с ядрами Cortex-M3 и ARM7. Однако в LabVIEW отсутствуют драйверы периферийных устройств для некоторых микроконтроллеров. В этом случае вам придётся написать интерфейсный код на Си и использовать его в проекте LabVIEW.

Рис. 21.3. *Отладочный комплект LabVIEW для процессора Cortex-M3.*

21.3. Процесс разработки

Процесс разработки приложения в среде LabVIEW обычно состоит из следующих этапов (**Рис. 21.4**):

- *Создание проекта и виртуального инструмента*. Для создания ARM-проекта можно воспользоваться мастером. На этом этапе производится настройка целевой платформы и драйвера устройства. После создания проекта создаётся VI (обычно пустой).
- *Определение входов и выходов*. Это могут быть как аппаратные интерфейсы микроконтроллера, так и пользовательские интерфейсы, запущенные на ПК, который подключён к системе. В случае аппаратных интерфейсов вам необходимо будет определить их входы и выходы в качестве базовых средств ввода/вывода, прежде чем вы сможете использовать их в своём виртуальном инструменте.
- *Создание приложения* с использованием графического программирования.
- *Компиляция проекта*.
- *Симуляция виртуального инструмента*. Созданный исполняемый образ можно передать в среду MDK-ARM для проверки работоспособности приложения.
- *Загрузка в микроконтроллер и тестирование.* По умолчанию программа загружается в микроконтроллер по завершении компиляции. Вы можете использовать интерфейс LabVIEW для приостановки, останова или пошагового выполнения программы. Вы также можете контролировать значения переменных, щёлкая мышкой на соединениях во время исполнения программы.

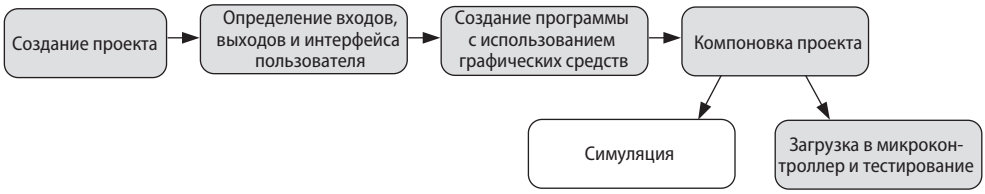


Рис. 21.4. *Пример процесса разработки.*

Каждый VI имеет два представления:

- вид передней панели — содержит GUI виртуального инструмента;
- вид блок-схемы — служит рабочей областью для графического программирования.

Когда вы создаёте VI, он по умолчанию имеет пустую переднюю панель. Вы можете разместить на ней элементы управления и индикации, чтобы определить входы и выходы системы. Например, для предыдущего примера нахождения наибольшего целого из трёх источников и случайного значения передняя панель может выглядеть так, как показано на **Рис. 21.5**.

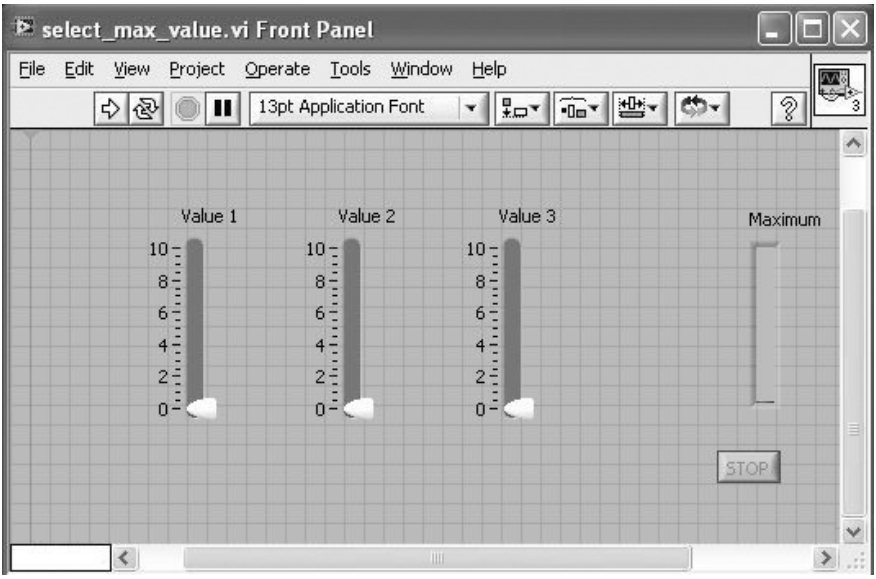


Рис. 21.5. *Простая передняя панель для VI, показанного на* **Рис. 21.2**.

В вашем распоряжении имеется большое число библиотечных элементов управления и индикации. После добавления требуемых элементов на переднюю панель они появляются на блок-схеме. Далее вы можете изменить их параметры (например, используемые типы данных) и создать свою графическую программу, объединив эти элементы с различными функциями LabVIEW.

21.4. Пример использования среды LabVIEW

21.4.1. Создание проекта

В данном подразделе мы создадим простое приложение, которое будет считывать аналоговый сигнал и отображать его в виде диаграммы на OLED-дисплее. Воспользуемся мастером для создания нового проекта: выберем на стартовом экране среды пункт **more**, а затем **ARM project**, как показано на **Рис. 21.6**. Теперь мы сможем создать пустой инструмент или воспользоваться существующим. В данном случае создадим новый VI. После этого выберем в качестве целевого устройства отладочную плату LM3S8962 (**Рис. 21.7**).

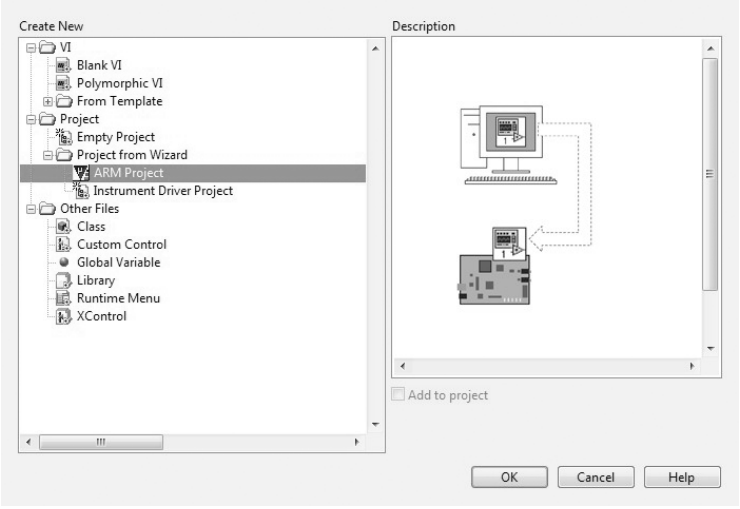


Рис. 21.6. *Создание нового проекта.*

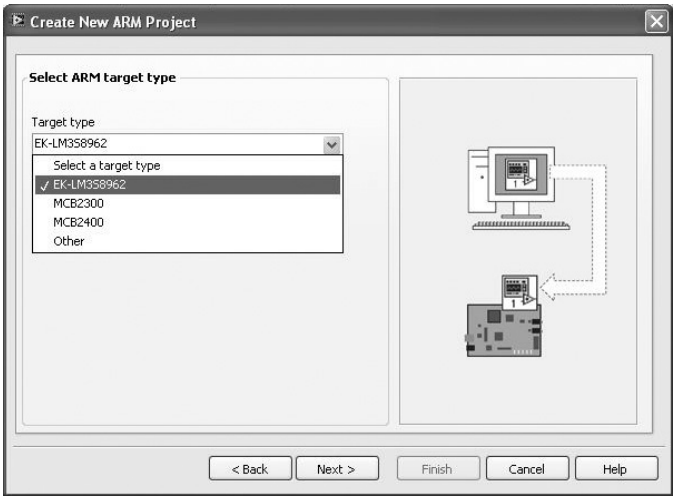


Рис. 21.7. *Выбор целевого устройства.*

В результате указанных операций мастер проектов создает спецификацию, которая позволит запускать этот проект в симуляторе. Нам она не потребуется, поскольку мы работаем с реальной платой. В конце своей работы мастер сохраняет созданный проект и пустой VI.

21.4.2. Определение входов и выходов

Следующий этап заключается в определении входов и выходов нашего приложения. В данном случае нам необходимо определить только вход ADC0 (OLED-дисплей управляется библиотечными компонентами, поэтому нам не нужно описывать его на данном этапе). Для выполнения указанной операции щёлкните правой кнопкой мыши на цели в окне проекта и выберите в контекстном меню пункт **New -> Elemental I/O** (**Рис. 21.8**). После этого в окне **New Elemental I/O** определите требуемый вход, как показано на **Рис. 21.9**.

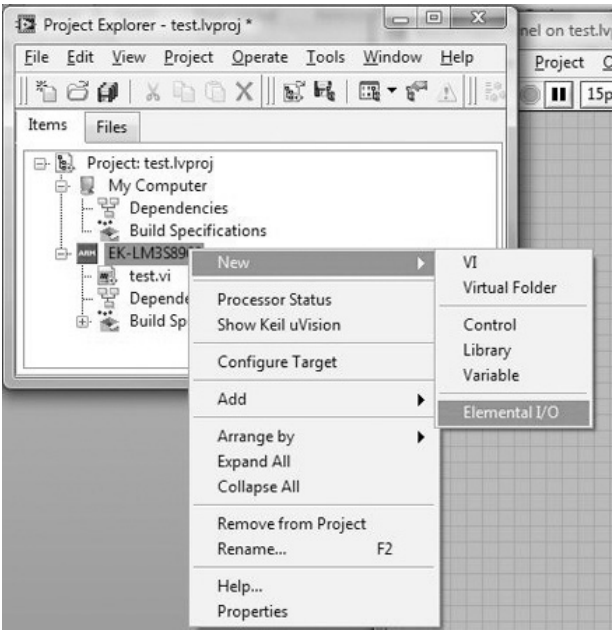


Рис. 21.8. *Определение базового элемента ввода/вывода.*

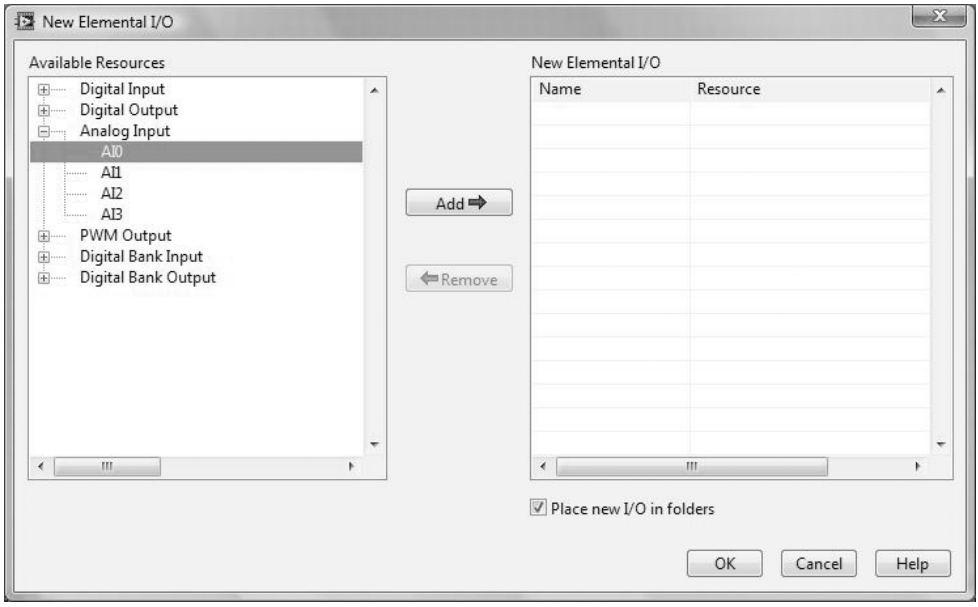


Рис. 21.9. *Добавление базового элемента ввода/вывода в проект.*

В этом окне имеются и другие элементы ввода/вывода, такие как кнопки, светодиоды, входы/выходы общего назначения (GPIO) и блоки широтно-импульсного модулятора (PWM). Все они могут значительно упростить процесс разработки вашего проекта.

21.4.3. Создание программы

Теперь мы можем приступить к описанию кода приложения в виде блок-схемы. Среда LabVIEW имеет очень много возможностей, которые просто физически невозможно рассмотреть в одной главе. К счастью, вместе со средой поставляется подробная документация, а каждый элемент графического программирования имеет контекстно-зависимую подсказку. Чтобы просмотреть доступные компоненты, можно щёлкнуть правой кнопкой мыши в области блок-схемы (**Рис. 21.10**). Из рисунка видно, в частности, что в категории ARM имеется несколько элементов управления OLED-дисплеями.

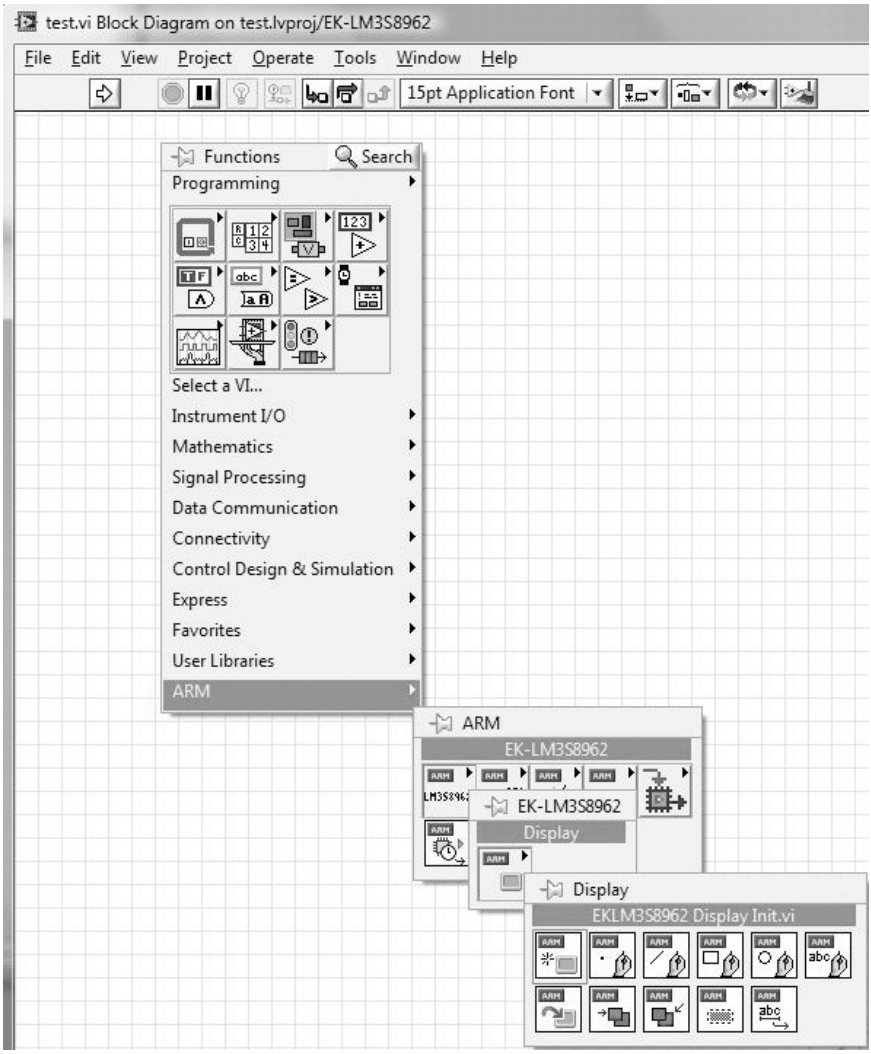


Рис. 21.10. *Доступ к функциям LabVIEW.*

Помимо элементов управления OLED-дисплеями, в этой категории также можно найти базовые элементы ввода/вывода, модули интерфейсов CAN, I^2C, SPI, а также функции управления прерываниями.

Программный код нашего приложения состоит из двух частей. В левой части блок-схемы будет расположен код, выполняющий следующие операции:

- инициализация OLED-дисплея;
- вывод на экран приветствия;
- формирование небольшой задержки;
- очистка экрана;
- вывод в верхнюю часть экрана надписи «Analog Input 0».

Большинство виртуальных инструментов, используемых в этой части программы, предназначены для оценочной платы LM3S8962 и содержатся в модуле «LabVIEW Emmbedded Module for ARM Microcontrollers» (**Рис. 21.11**).

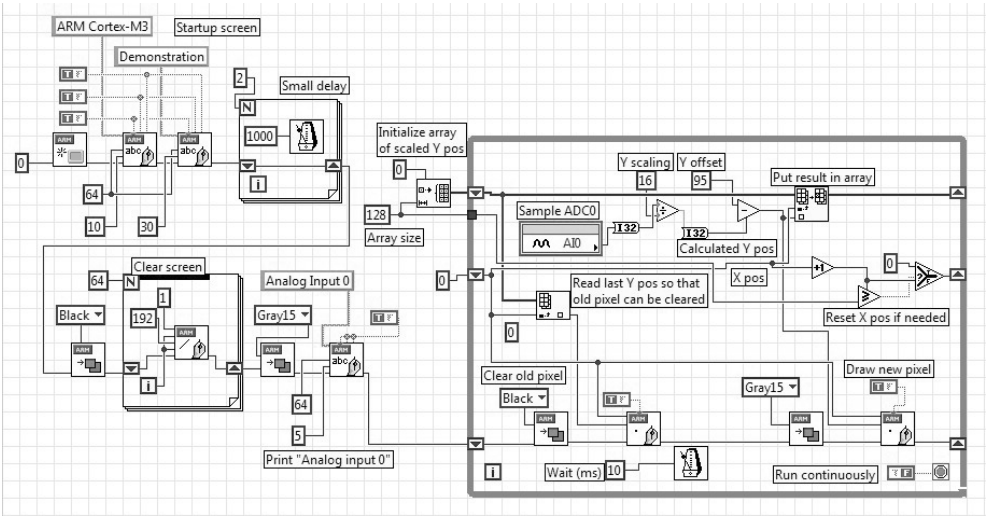


Рис. 21.11. *Блок-схема программы осциллографа.*

В правой части блок-схемы мы расположим блок цикла `While Loop`, в котором отсчёты берутся с входа ADC0 микроконтроллера и отображаются на экране в виде точки. При достижении координатой X правого края OLED-дисплея она сбрасывается в 0.

Вычисленная позиция Y также сохраняется в массиве. Этот массив содержит 128 элементов и используется для удаления старого изображения перед рисованием новой точки.

21.4.4. Компиляция программы и тестирование приложения

После завершения разработки программы нужно скомпилировать программу и проверить её работоспособность. Сначала щёлкните левой кнопкой мыши на кнопке с изображением стрелки. Если программа содержит ошибки, то изображение сменится на разломанную стрелку, показывая неготовность программы. Щёлкнув на кнопке ещё раз, вы получите отчёт об ошибках, обнаруженных в программе.

После успешного завершения процесса компиляции программа автоматически загружается в микроконтроллер и начинает выполняться. В нашем случае контроллер выводит осциллограмму для аналогового входа, как и было задумано (**Рис. 21.12**).

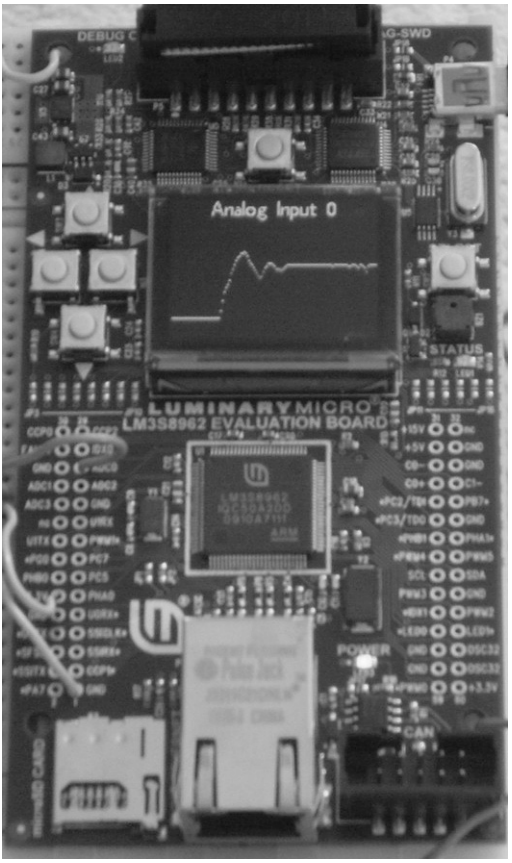


Рис. 21.12. *Приложение LabVIEW, выполняющееся на отладочной плате.*

21.5. Как это работает

При компиляции программы среда LabVIEW генерирует Си-код для созданного виртуального инструмента. Полученный код можно впоследствии скомпилировать, используя пакет MDK-ARM (**Рис. 21.13**). Для реализации параллельной работы виртуальных инструментов полученный код использует библиотеку реального времени RealView Real-Time Library (RL-ARM, *www.keil.com/arm/rl-arm/*), которая работает совместно с операционной системой RTX Real Time Kernel (*www.keil.com/arm/rl-arm/kernrl.asp*) компании Keil.

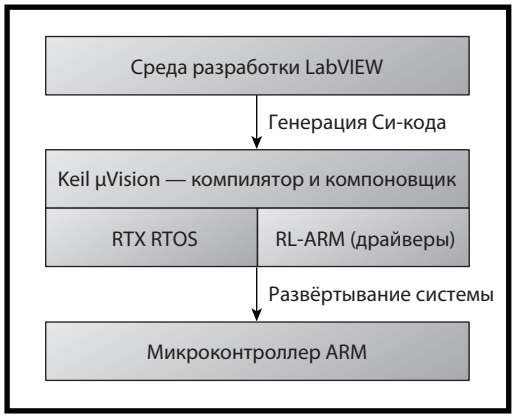


Рис. 21.13. *Разработка приложения с использованием LabVIEW и ИСР µVISION.*

Помимо многопоточности, использование библиотеки RL-ARM обеспечивает доступ к множеству драйверов, включая управление таймерами, стек протокола TCP/IP и стек протокола CAN.

21.6. Дополнительные возможности LabVIEW

- *Полухостинг.* Среда LabVIEW поддерживает создание графических интерфейсов пользователя, запускаемых на хосте отладки (ПК). Эта возможность просто незаменима на этапе макетирования системы, поскольку разработка графического интерфейса, работающего на микроконтроллере, может оказаться весьма сложной задачей. Библиотека LabVIEW содержит различные элементы пользовательского интерфейса, такие как переключатели, ползунки и диаграммы, позволяющие создавать профессиональные интерфейсы всего несколькими щелчками мыши. В рабочем режиме этот интерфейс (GUI) выполняется на ПК, а данные передаются на работающий микроконтроллер через отладчик с использованием различных протоколов (JTAG, TSP/IP или Serial). Это позволяет управлять устройством и получать результаты его работы в реальном времени.
- *Интеграция кода на Си.* Среда LabVIEW допускает вставку Си-кода в виртуальные инструменты. Это часто требуется при создании новых драйверов устройств и выполнения задач обработки данных, которые не могут быть реализованы с помощью имеющихся компонентов LabVIEW.
- *Отладка.* Среда LabVIEW позволяет отлаживать приложения. Во время симуляции или тестирования проекта вы можете использовать среду разработки VI в качестве отладчика, контролируя значения переменных при останове. В окне представления VI в виде блок-схемы предусмотрены кнопки паузы, пошагового выполнения, а также запуска профилирования значений (**Рис. 21.14**).

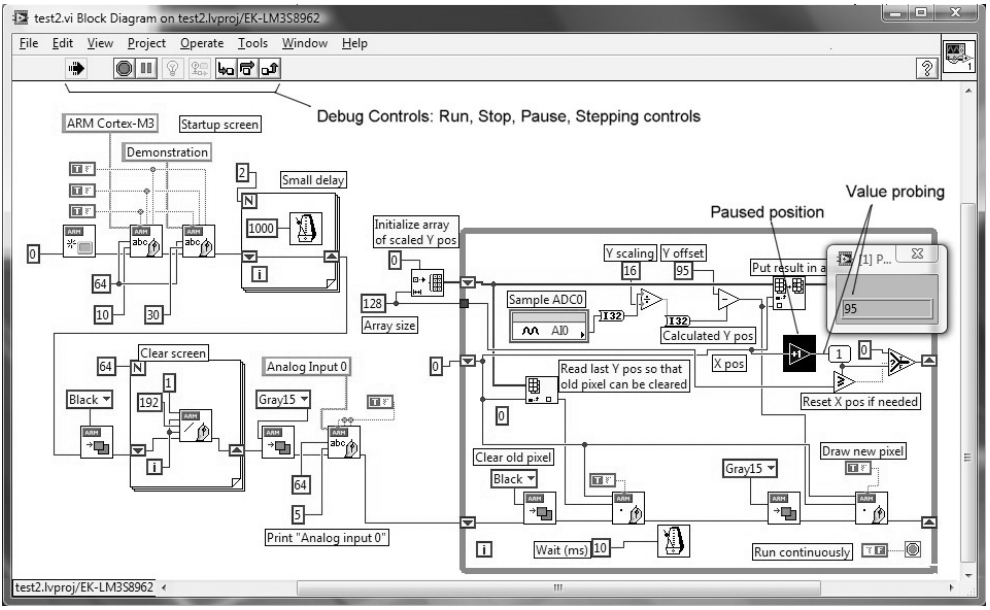


Рис. 21.14. *Отладка по блок-схеме VI.*

Кроме того, поскольку в составе ИСР µVision компании Keil имеется свой полнофункциональный отладчик, вы можете отлаживать код приложения в нём (**Рис. 21.15**).

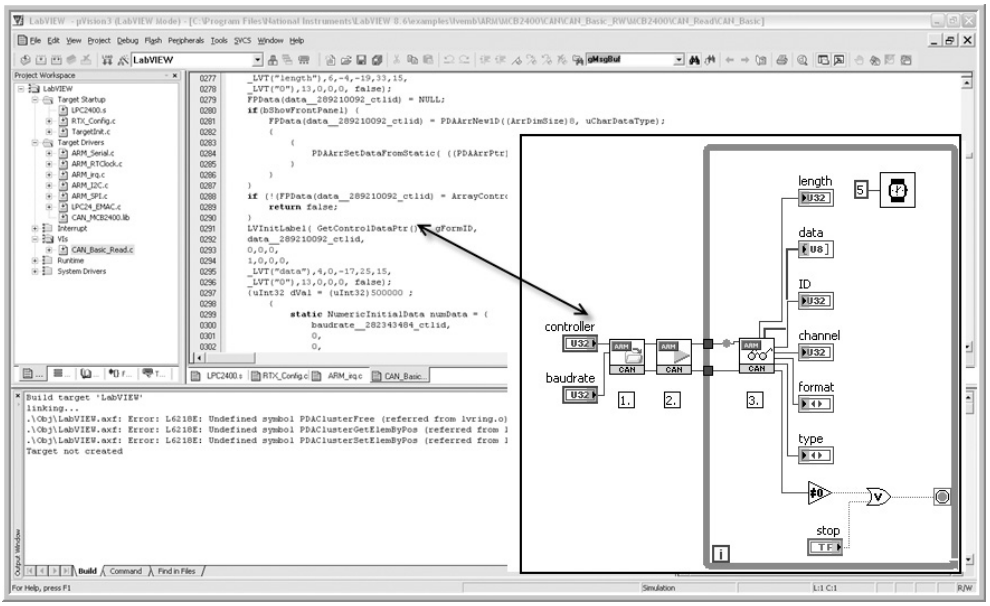


Рис. 21.15. *Отладка кода, сгенерированного LabVIEW, в ИСР µVision.*

21.7. Перенос проекта на другие процессоры ARM

Если вам недостаточно просто запустить приложение на отладочной плате, то вы можете портировать его на другие микроконтроллеры с процессором Cortex-M3 или на микроконтроллеры с другим процессором компании ARM, поддерживаемые ядром реального времени RTX Real-Time Kernel. Как правило, перенос приложения состоит из следующих операций:

- *Портирование ядра RTX Real-Time Kernel.* При использовании устройств с процессором Cortex-M3 данный этап не требуется, поскольку процессор имеет всё необходимое для поддержки RTX Real-Time Kernel. Если же вы собираетесь задействовать микроконтроллер с ядром ARM7, то вам придётся выполнить данную операцию. Хотя может получиться так, что порт ядра RTX Real-Time Kernel для вашего микроконтроллера уже имеется в составе пакета MDK-ARM. Чтобы убедиться в этом, следует просмотреть содержимое папки \Keil\ARM\Startup. Если вы найдёте в данной папке файл `RTX _ Conf*.c` для вашего микроконтроллера, значит, ядро ОС уже портировано на него.
- *Создание цели в LabVIEW и регистрация инструментария Keil.* Этот этап можно выполнить, создав папку цели вручную.
- *Интеграция отладочного модуля Real-Time Agent Module.* Вам может потребоваться добавить параметры конфигурации данного модуля в файл `RTX _ config.c`.
- *Разработка драйверов периферии и устройств ввода/вывода.* На этом этапе можно использовать редактор **Element I/O Device Editor** среды LabVIEW.

Более подробно процесс переноса приложений описан в руководстве «*LabVIEW Embedded for ARM Porting Guide*», которое можно загрузить с сайта компании National Instruments (*http://zone.ni.com/devzone/cda/tut/p/id/6994*).

ПРИЛОЖЕНИЕ A
НАБОР КОМАНД CORTEX-M3. СПРАВОЧНЫЙ МАТЕРИАЛ

В этом приложении приведено описание набора команд Cortex™-M3, взятое из руководства пользователя Cortex-M3 (данный материал воспроизводится с разрешения компании ARM). Приложение состоит из нескольких разделов. В двух первых разделах содержится информация общего характера:

- краткое описание набора команд (раздел A.1);
- соглашения, используемые при описании команд (раздел A.2).

Каждый из последующих разделов посвящён одной функциональной группе команд. Вместе эти разделы содержат описание всех команд, поддерживаемых процессором Cortex-M3:

- команды доступа к памяти (раздел A.3);
- общие команды обработки данных (раздел A.4);
- команды умножения и деления (раздел A.5);
- команды насыщения (раздел A.6);
- команды работы с битовыми полями (раздел A.7);
- команды управления и ветвления (раздел A.8);
- прочие команды (раздел A.9).

A.1. Краткое описание набора команд

В процессоре реализован один из вариантов набора команд Thumb®. Все поддерживаемые команды перечислены в **Табл. A.1**.

Таблица A.1. **Команды Cortex-M3**

Мнемоника	Операнды	Краткое описание	Флаги	Подраздел
ADC, ADCS	{Rd,} Rn, Op2	Сложение с переносом	N, Z, C, V	A.4.1
ADD, ADDS	{Rd,} Rn, Op2	Сложение	N, Z, C, V	A.4.1
ADD, ADDW	{Rd,} Rn, #imm12	Сложение	N, Z, C, V	A.4.1
ADR	Rd, label	Загрузка относительного адреса	—	A.3.1
AND, ANDS	{Rd,} Rn, Op2	Логическое И	N, Z, C	A.4.2
ASR, ASRS	Rd, Rm, <Rs|#n>	Арифметический сдвиг вправо	N, Z, C	A.4.3
B	Label	Переход	—	A.8.1
BFC	Rd, #lsb, #width	Очистка битового поля	—	A.7.1
BFI	Rd, Rn, #lsb, #width	Вставка битового поля	—	A.7.1

Таблица А.1. **Команды Cortex-M3** *(продолжение)*

Мнемоника	Операнды	Краткое описание	Флаги	Подраздел
BIC, BICS	{Rd,} Rn, Op2	Очистка битов	N, Z, C	A.4.2
BKPT	#imm	Точка останова	—	A.9.1
BL	Label	Переход со ссылкой	—	A.8.1
BLX	Rm	Косвенный переход со ссылкой	—	A.8.1
BX	Rm	Косвенный переход	—	A.8.1
CBNZ	Rn, label	Сравнение и переход, если результат не равен нулю	—	A.8.2
CBZ	Rn, label	Сравнение и переход, если результат равен нулю	—	A.8.2
CLREX	—	Сброс монопольного доступа	—	A.3.9
CLZ	Rd, Rm	Подсчёт ведущих нулевых битов	—	A.4.4
CMN	Rn, Op2	Сравнение с отрицательным операндом	N, Z, C, V	A.4.5
CMP	Rn, Op2	Сравнение	N, Z, C, V	A.4.5
CPSID	iflags	Изменение состояния процессора, запрещение прерываний	—	A.9.2
CPSIE	iflags	Изменение состояния процессора, разрешение прерываний	—	A.9.2
DMB	—	Барьер памяти данных	—	A.9.3
DSB	—	Барьер синхронизации данных	—	A.9.4
EOR, EORS	{Rd,} Rn, Op2	Исключающее ИЛИ	N, Z, C	A.4.2
ISB	—	Барьер синхронизации команд	—	A.9.5
IT	—	Условный блок IF-THEN	—	A.8.3
LDM	Rn{!}, reglist	Загрузка нескольких регистров с постинкрементом	—	A.3.6
LDMDB, LDMEA	Rn{!}, reglist	Загрузка нескольких регистров с преддекрементом	—	A.3.6
LDMFD, LDMIA	Rn{!}, reglist	Загрузка нескольких регистров с постинкрементом	—	A.3.6
LDR	Rt, [Rn, #offset]	Загрузка слова в регистр	—	A.3.2
LDRB, LDRBT	Rt, [Rn, #offset]	Загрузка байта в регистр	—	A.3.2
LDRD	Rt, Rt2, [Rn, #offset]	Загрузка двух слов в регистры	—	A.3.2
LDREX	Rt, [Rn, #offset]	Монопольная загрузка в регистр	—	A.3.8
LDREXB	Rt, [Rn]	Монопольная загрузка байта в регистр	—	A.3.8
LDREXH	Rt, [Rn]	Монопольная загрузка полуслова в регистр	—	A.3.8
LDRH, LDRHT	Rt, [Rn, #offset]	Загрузка полуслова в регистр	—	A.3.2
LDRSB, LDRSBT	Rt, [Rn, #offset]	Загрузка байта со знаком в регистр	—	A.3.2
LDRSH, LDRSHT	Rt, [Rn, #offset]	Загрузка полуслова со знаком в регистр	—	A.3.2
LDRT	Rt, [Rn, #offset]	Загрузка слова в регистр	—	A.3.2

***Таблица A.1.* Команды Cortex-M3** *(продолжение)*

Мнемоника	Операнды	Краткое описание	Флаги	Подраздел
LSL, LSLS	Rd, Rm, <Rs|#n>	Логический сдвиг влево	N, Z, C	A.4.3
LSR, LSRS	Rd, Rm, <Rs|#n>	Логический сдвиг вправо	N, Z, C	A.4.3
MLA	Rd, Rn, Rm, Ra	Умножение со сложением, 32-битный результат	—	A.5.1
MLS	Rd, Rn, Rm, Ra	Умножение с вычитанием, 32-битный результат	—	A.5.1
MOV, MOVS	Rd, Op2	Пересылка	N, Z, C	A.4.6
MOVT	Rd, #imm16	Пересылка старшего полуслова	—	A.4.7
MOVW, MOV	Rd, #imm16	Пересылка 16-битной константы	N, Z, C	A.4.6
MRS	Rd, spec_reg	Пересылка из PCH в POH	—	A.9.6
MSR	spec_reg, Rm	Пересылка из POH в PCH	N, Z, C, V	A.9.7
MUL, MULS	Rd, Rn, Rm	Умножение, 32-битный результат	N, Z	A.5.1
MVN, MVNS	Rd, Op2	Пересылка с инверсией	N, Z, C	A.4.6
NOP	–	Нет операции	—	A.9.8
ORN, ORNS	{Rd,} Rn, Op2	Логическое ИЛИ с инверсией	N, Z, C	A.4.2
ORR, ORRS	{Rd,} Rn, Op2	Логическое ИЛИ	N, Z, C	A.4.2
POP	reglist	Извлечение регистра из стека	—	A.3.7
PUSH	reglist	Загрузка регистра в стек	—	A.3.7
RBIT	Rd, Rn	Перестановка битов	—	A.4.8
REV	Rd, Rn	Перестановка байтов слова	—	A.4.8
REV16	Rd, Rn	Перестановка байтов в каждом полуслове	—	A.4.8
REVSH	Rd, Rn	Перестановка байтов в младшем полуслове и расширение знака	—	A.4.8
ROR, RORS	Rd, Rm, <Rs|#n>	Циклический сдвиг вправо	N, Z, C	A.4.3
RRX, RRXS	Rd, Rm	Расширенный циклический сдвиг вправо	N, Z, C	A.4.3
RSB, RSBS	{Rd,} Rn, Op2	Обратное вычитание	N, Z, C, V	A.4.1
SBC, SBCS	{Rd,} Rn, Op2	Вычитание с переносом	N, Z, C, V	A.4.1
SBFX	Rd, Rn, #lsb, #width	Извлечение битового поля со знаком	—	A.7.2
SDIV	{Rd,} Rn, Rm	Знаковое деление	—	A.5.3
SEV	–	Генерация события	—	A.9.9
SMLAL	RdLo, RdHi, Rn, Rm	Знаковое умножение со сложением ($32 \times 32 + 64$), 64-битный результат	—	A.5.2
SMULL	RdLo, RdHi, Rn, Rm	Знаковое умножение (32×32), 64-битный результат	—	A.5.2
SSAT	Rd, #n, Rm {,shift #s}	Знаковое насыщение	Q	A.6.1
STM	Rn{!}, reglist	Сохранение нескольких регистров с постинкрементом	—	A.3.6
STMDB, STMEA	Rn{!}, reglist	Сохранение нескольких регистров с преддекрементом	—	A.3.6

Таблица А.1. **Команды Cortex-M3** *(продолжение)*

Мнемоника	Операнды	Краткое описание	Флаги	Подраздел
STMFD, STMIA	Rn{!}, reglist	Сохранение нескольких регистров с постинкрементом	—	A.3.6
STR	Rt, [Rn, #offset]	Сохранение слова из регистра	—	A.3.2
STRB, STRBT	Rt, [Rn, #offset]	Сохранение байта из регистра	—	A.3.2
STRD	Rt, Rt2, [Rn, #offset]	Сохранение двух слов из регистров	—	A.3.2
STREX	Rd, Rt, [Rn, #offset]	Монопольное сохранение регистра	—	A.3.8
STREXB	Rd, Rt, [Rn]	Монопольное сохранение байта из регистра	—	A.3.8
STREXH	Rd, Rt, [Rn]	Монопольное сохранение полуслова из регистра	—	A.3.8
STRH, STRHT	Rt, [Rn, #offset]	Сохранение полуслова из регистра	—	A.3.2
STRT	Rt, [Rn, #offset]	Сохранение слова из регистра	—	A.3.2
SUB, SUBS	{Rd,} Rn, Op2	Вычитание	N, Z, C, V	A.4.1
SUB, SUBW	{Rd,} Rn, #imm12	Вычитание	N, Z, C, V	A.4.1
SVC	#imm	Вызов супервизора	—	A.9.10
SXTB	Rd, Rm {,ROR #n}	Расширение знака байта	—	A.7.3
SXTH	Rd, Rm {,ROR #n}	Расширение знака полуслова	—	A.7.3
TBB	[Rn, Rm]	Табличный переход с однобайтными смещениями	—	A.8.4
TBH	[Rn, Rm, LSL #1]	Табличный переход с двухбайтными смещениями	—	A.8.4
TEQ	Rn, Op2	Проверка на равенство	N, Z, C	A.4.9
TST	Rn, Op2	Проверка битов	N, Z, C	A.4.9
UBFX	Rd, Rn, #lsb, #width	Извлечение беззнакового битового поля	—	A.7.2
UDIV	{Rd,} Rn, Rm	Беззнаковое деление	—	A.5.3
UMLAL	RdLo, RdHi, Rn, Rm	Беззнаковое умножение со сложением (32 × 32 + 64), 64-битный результат	—	A.5.2
UMULL	RdLo, RdHi, Rn, Rm	Беззнаковое умножение (32 × 32), 64-битный результат	—	A.5.2
USAT	Rd, #n, Rm {,shift #s}	Беззнаковое насыщение	Q	A.6.1
UXTB	Rd, Rm {,ROR #n}	Дополнение нулями байта	—	A.7.3
UXTH	Rd, Rm {,ROR #n}	Дополнение нулями полуслова	—	A.7.3
WFE	–	Ожидание события	—	A.9.11
WFI	–	Ожидание прерывания	—	A.9.12

Примечания:

1. В угловых скобках указаны альтернативные формы операнда; в фигурных скобках указаны необязательные операнды.
2. В столбце «Операнды» таблицы приведены не все возможные варианты.
3. Операнд `Op2` — «гибкий» второй операнд, который может быть либо регистром, либо константой.
4. С большинством команд может использоваться необязательный суффикс условия выполнения.

Для получения более подробной информации обращайтесь непосредственно к описаниям команд.

A.2. Соглашения, используемые при описании команд

В этом разделе содержится более подробная информация об использовании команд:

- операнды (подраздел A.2.1);
- ограничения на использование регистров PC или SP (подраздел A.2.2);
- «гибкий» второй операнд (подраздел A.2.3);
- операции сдвига (подраздел A.2.4);
- выравнивание адреса (подраздел A.2.5);
- адресация относительно PC (подраздел A.2.6);
- условное выполнение (подраздел A.2.7);
- выбор разрядности команды (подраздел A.2.8).

A.2.1. Операнды

Операндом команды может быть регистр, константа или же другой параметр, специфичный для конкретной команды. Команда выполняет требуемые действия над операндами и во многих случаях сохраняет результат в регистр-приёмник. При наличии в команде такого регистра он обычно указывается перед операндами.

В некоторых командах операнды обладают определённой гибкостью в том смысле, что они могут быть как регистрами, так и константами (см. подраздел A.2.3 «Гибкий» второй операнд»).

A.2.2. Ограничения на использование регистров PC или SP

Многие команды имеют ограничения на использование счётчика команд (PC) или указателя стека (SP) в качестве операндов или регистра-приёмника. Это будет указано при описании конкретных команд.

Примечание

Нулевой бит (бит [0]) любого адреса, который загружается в PC командами `BX`, `BLX`, `LDM`, `LDR` или `POP`, должен быть установлен в 1 для корректной работы процессора, поскольку этот бит определяет требуемый набор команд, а процессор Cortex-M3 поддерживает только команды Thumb.

A.2.3. «Гибкий» второй операнд

Многие команды обработки данных имеют «гибкий» второй операнд. В описании синтаксиса таких команд этот операнд обозначается как *`Operand2`*.

Второй операнд *`Operand2`* может быть:

- константой;
- регистром с необязательным сдвигом (стр. 380).

Константа

Константа, используемая в качестве `Operand2`, записывается в виде:

```
#constant
```

где *constant* может быть:

- любой константой, которая может быть получена сдвигом 8-битного значения влево на произвольное число битов в пределах 32-битного слова;
- любой константой вида `0x00XY00XY`;
- любой константой вида `0xXY00XY00`;
- любой константой вида `0xXYXYXYXY`.

Примечание

В вышеприведённом описании символы `X` и `Y` обозначают шестнадцатеричные числа.

В некоторых командах *constant* может принимать значения из более широкого диапазона. Это будет указано особо в описании таких команд.

Когда константа используется в качестве второго операнда с командами `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` или `TST`, то в случае, если значение константы больше 255 и оно может быть получено сдвигом 8-битного числа, во флаг переноса заносится значение 31-го бита константы. Если вторым операндом является любая другая константа, то указанные команды не влияют на состояние флага переноса.

Подстановка команд

При указании константы с недопустимым значением используемый вами ассемблер может заменить команду на эквивалентную. Например, ассемблер может подставить вместо команды `CMP Rd, #0xFFFFFFFE` эквивалентную команду `CMN` *Rd*`, #0x02`.

Регистр с необязательным сдвигом

В этом случае второй операнд записывается в виде:

```
Rm {, shift}
```

где

Rm — регистр, содержащий значение для второго операнда.

shift — необязательная операция сдвига, выполняемая над содержимым регистра *Rm*. Для указания этой операции может использоваться одна из следующих мнемоник:

`ASR #`*n*	Арифметический сдвиг вправо на *n* битов, $1 \le n \le 32$
`LSL #`*n*	Логический сдвиг влево на *n* битов, $1 \le n \le 31$
`LSR #`*n*	Логический сдвиг вправо на *n* битов, $1 \le n \le 32$
`ROR #`*n*	Циклический сдвиг вправо на *n* битов, $1 \le n \le 31$
`RRX`	Расширенный циклический сдвиг вправо на 1 бит
—	Если мнемоника отсутствует, то сдвиг не выполняется, эквивалентно `LSL #0`

Если операция сдвига отсутствует или задана операция `LSL #0`, то команда использует значение, находящееся в регистре *Rm*.

При указании операции сдвига она выполняется над содержимым регистра *Rm*, а итоговое 32-битное значение используется в команде. Содержимое регистра *Rm* при этом не изменяется. Некоторые команды при использовании регистра со сдвигом также изменяют состояние флага переноса. Более подробно операции сдвига и их влияние на флаг переноса рассматриваются в подразделе А.2.4 «Операции сдвига».

А.2.4. Операции сдвига

Операции сдвига регистра выполняют сдвиг содержимого этого регистра вправо или влево на заданное число битов (*величина сдвига*). Сдвиг регистра может осуществляться:

- непосредственно командами `ASR`, `LSR`, `LSL`, `ROR` и `RRX` с записью результата в регистр-приёмник;
- при вычислении *Operand2* командами, использующими в качестве второго операнда регистр со сдвигом (см. подраздел А.2.3 «Гибкий» второй операнд»). Результат операции сдвига используется командой.

Допустимые значения величины сдвига зависят как от типа сдвига, так и от команды (см. описание конкретных команд или подраздел А.2.3 «Гибкий» второй операнд»). Если величина сдвига равна нулю, то сдвиг не осуществляется. Операции сдвига регистра изменяют состояние флага переноса, за исключением тех случаев, когда величина сдвига равна нулю. Далее приводятся описания различных операций сдвига и указывается влияние этих операций на флаг переноса. В данных описаниях *Rm* обозначает регистр, содержащий сдвигаемое число, а *n* — величину сдвига.

Команда ASR

Арифметический сдвиг вправо на *n* битов перемещает *32 – n* старших битов регистра *Rm* на *n* позиций вправо, в *32 – n* младших битов результата. Исходный бит [31] регистра копируется в *n* старших битов результата (**Рис. А.1**).

Рис. А.1. *ASR #3.*

Операцию `ASR #n` можно использовать для деления содержимого регистра *Rm* на 2^n с округлением результата к ближайшему меньшему целому.

В случае команды `ASRS` или при использовании команды `ASR #n` во втором операнде команд `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` или `TST` во флаг переноса загружается последний выдвинутый бит регистра *Rm*, бит [*n*–1].

Примечание

- Если *n* ≥ 32, то во все биты результата помещается значение бита [31] регистра *Rm*.
- Если *n* ≥ 32 и операция влияет на флаг переноса, то в этот флаг загружается значение бита [31] регистра *Rm*.

Команда LSR

Логический сдвиг вправо на *n* битов перемещает *32 – n* старших битов регистра *Rm* на *n* позиций вправо, в *32 – n* младших битов результата. Старшие *n* битов результата обнуляются (**Рис. А.2**).

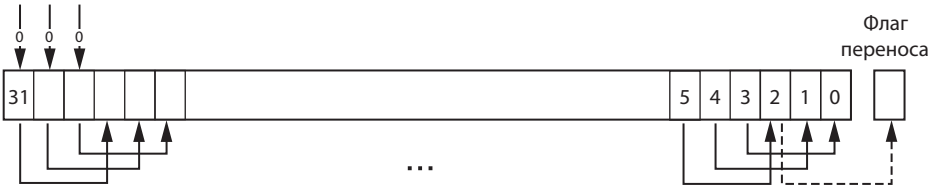


Рис. А.2. *LSR #3.*

Операцию `LSR #n` можно использовать для деления содержимого регистра *Rm* на 2^n, если содержимое регистра рассматривается как целое число без знака.

В случае команды `LSRS` или при использовании команды `LSR #n` во втором операнде команд `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` или `TST` во флаг переноса загружается последний выдвинутый бит регистра *Rm*, бит [*n*–1].

Примечание

- Если *n* ≥ 32, то все биты результата сбрасываются в 0.
- Если *n* ≥ 33 и операция влияет на флаг переноса, то он сбрасывается в 0.

Команда LSL

Логический сдвиг влево на *n* битов перемещает *32 – n* младших битов регистра *Rm* на *n* позиций влево, в *32 – n* старших битов результата. Младшие *n* битов результата обнуляются (**Рис. А.3**).

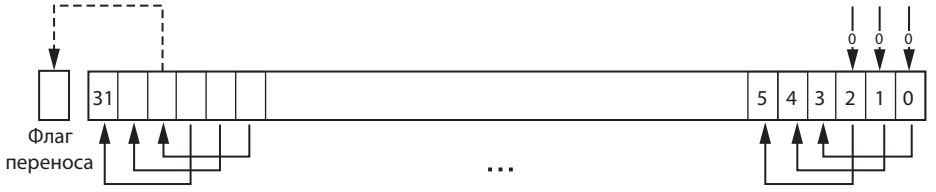


Рис. А.3. *LSL #3.*

Операцию `LSL #n` можно использовать для умножения содержимого регистра *Rm* на 2^n, если содержимое регистра рассматривается как целое число без знака или как целое число со знаком, представленное в дополнительном коде. При переполнении никаких предупреждений не выдаётся.

В случае команды `LSLS` или при использовании команды `LSL #n` во втором операнде команд `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` или `TST` во флаг переноса загружается последний выдвинутый бит регистра *Rm*, бит [32–*n*]. Эта операция не влияет на состояние флага переноса, если *n* = 0.

Примечание

- Если *n* ≥ 32, то все биты результата сбрасываются в 0.
- Если *n* ≥ 33 и операция влияет на флаг переноса, то он сбрасывается в 0.

Команда ROR

Циклический сдвиг вправо на *n* битов перемещает *32 – n* старших битов регистра *Rm* на *n* позиций вправо, в *32 – n* младших битов результата. Одновременно *n* младших битов регистра копируются в *n* старших битов результата (**Рис. А.4**).

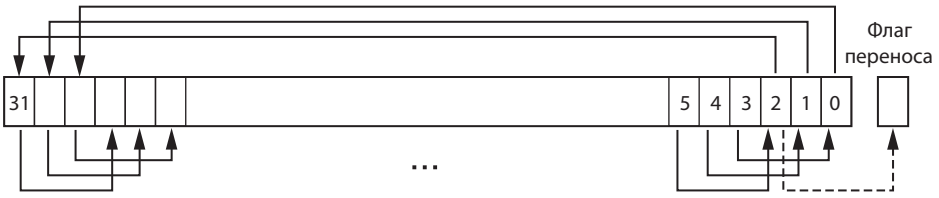


Рис. А.4. *ROR #3.*

В случае команды `RORS` или при использовании команды `ROR #n` во втором операнде команд `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` или `TST` во флаг переноса загружается последний перемещённый бит регистра *Rm*, бит [*n*–1].

Примечание

- Если *n* = 32, то результат равен содержимому регистра *Rm*. При этом если операция влияет на флаг переноса, то в него загружается значение бита [31] регистра *Rm*.
- Операция `ROR` при величине сдвига *n* > 32 эквивалентна операции `ROR` с величиной сдвига, равной *n – 32*.

Команда RRX

Расширенный циклический сдвиг вправо перемещает все биты регистра *Rm* на одну позицию вправо и копирует содержимое флага переноса в бит [31] результата (**Рис. А.5**).

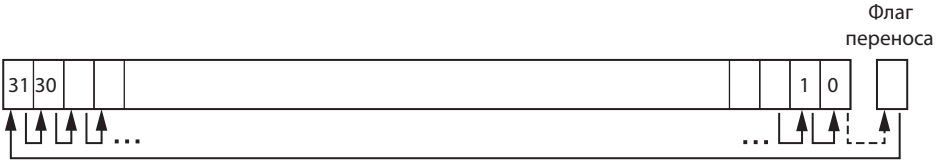


Рис. А.5. *RRX.*

В случае команды `RRXS` или при использовании команды `RRX` во втором операнде команд `MOVS`, `MVNS`, `ANDS`, `ORRS`, `ORNS`, `EORS`, `BICS`, `TEQ` или `TST` во флаг переноса загружается бит [0] регистра *Rm*.

А.2.5. Выравнивание адресов

Под выровненным доступом понимают операции, в которых чтение и запись слов, двойных слов и более длинных последовательностей слов осуществляется по адресам, выровненным на границу слова, или же операции, при которых для обращения к полуслову используется адрес, выровненный на границу полуслова. Обращения к байтам выровнены по определению.

Процессор Cortex-M3 поддерживает обращение к невыровненным данным только для следующих команд:

- `LDR`, `LDRT`;
- `LDRH`, `LDRHT`;
- `LDRSH`, `LDRSHT`;
- `STR`, `STRT`;
- `STRH`, `STRHT`.

Все остальные команды загрузки и сохранения при обращении по невыровненному адресу генерируют исключение Usage Fault, поэтому они должны использоваться только с выровненными адресами.

Обращения к невыровненным данным, как правило, осуществляются медленнее, чем к выровненным. Кроме того, некоторые области адресного пространства могут не поддерживать обращение по невыровненным адресам. В связи с этим компания ARM рекомендует всегда использовать выровненные данные. Для перехвата непреднамеренных обращений по невыровненным адресам используется бит UNALIGN_TRP регистра CCR, разрешающий формирование исключения Usage Fault для всех обращений по невыровненным адресам.

А.2.6. Адресация относительно PC

Выражение, определяемое относительно значения счётчика команд PC, или, иначе, *метка*, является идентификатором, представляющим адрес команды или константы. В командах это выражение хранится в виде смещения относительно значения счётчика команд. Ассемблер автоматически вычисляет требуемое смещение, исходя из адреса метки и адреса текущей команды. Если величина смещения получается слишком большой, то ассемблер выдаёт сообщение об ошибке.

Примечание

- Для команд `B`, `BL`, `CBNZ` и `CBZ` значением PC является адрес команды плюс 4 байта.
- Для большинства прочих команд, использующих метки, значением PC является адрес команды плюс 4 байта, при этом бит [1] результата сбрасывается в 0 для выравнивания адреса по границе слова.
- В используемом вами ассемблере для выражений, определяемых относительно PC, могут использоваться другие формы записи, скажем метка плюс/минус число или запись вида `[PC, #number]`.

А.2.7. Условное выполнение

Большинство команд обработки данных могут опционально изменять флаги условий регистра состояния прикладной программы APSR в соответствии с результатом операции. Одни команды влияют на все флаги, другие — только на некоторые из них. Если команда не влияет на флаг, то его состояние остаётся неизменным. Чтобы узнать, на какие флаги влияет та или иная команда, обратитесь к её описанию.

Команда может быть выполнена условно в соответствии со значением флагов, изменённых другой командой, либо сразу же после команды, изменившей флаги, либо после нескольких промежуточных команд, не влияющих на состояние флагов условий.

Условное выполнение реализуется командами условного перехода либо добавлением к мнемонике обычных команд суффиксов, определяющих условие выполнения команды. Все эти суффиксы перечислены в **Табл. А.2**. Код условия, определяемый суффиксом, позволяет процессору проверять соответствие состояния флагов заданному условию. Если код условия условно выполняемой команды не соответствует состоянию флагов, то команда:

- не выполняется;
- ничего не сохраняет в регистре-приёмнике;
- не воздействует ни на какие флаги;
- не генерирует никаких исключительных ситуаций.

Таблица А.2. **Суффиксы условия выполнения**

Суффикс	Флаги	Значение
`EQ`	Z = 1	Равно
`NE`	Z = 0	Не равно
`CS` или `HS`	C = 1	Выше или равно (беззнаковое «≥»)
`CC` или `LO`	C = 0	Ниже (беззнаковое «<»)
`MI`	N = 1	Отрицательный результат
`PL`	N = 0	Положительный результат либо ноль
`VS`	V = 1	Переполнение
`VC`	V = 0	Нет переполнения
`HI`	C = 1 и Z = 0	Выше (беззнаковое «>»)
`LS`	C = 0 или Z = 1	Ниже или равно (беззнаковое «≤»)
`GE`	N = V	Больше или равно (знаковое «≥»)
`LT`	N ≠ V	Меньше (знаковое «<»)
`GT`	Z = 0 и N = V	Больше (знаковое «>»)
`LE`	Z = 1 и N ≠ V	Меньше или равно (знаковое «≤»)
`AL`	Безразлично	Всегда; действие по умолчанию при отсутствии суффикса

Условно выполняемые команды, за исключением команд условных переходов, должны располагаться внутри блока команд `IF-THEN`. Дополнительная информация о команде `IT` и ограничения на её использование приведены в описании этой команды (см. подраздел А.8.3).

Если это предусмотрено разработчиком ассемблера, то последний может автоматически вставлять команду `IT` при обнаружении условно выполняемых команд вне `IT`-блока.

Для сравнения содержимого регистра с нулем и перехода согласно результату используйте команды `CBZ` и `CBNZ`.

Далее в этом подразделе описываются:

- Флаги условия
- Суффиксы условия выполнения

Флаги условия

В регистре APSR содержатся следующие флаги условия:

N Устанавливается в 1, если результат операции был отрицателен, иначе сбрасывается в 0.

Z Устанавливается в 1, если результат операции был равен нулю, иначе сбрасывается в 0.

C Устанавливается в 1, если в результате операции произошёл перенос, иначе сбрасывается в 0.

V Устанавливается в 1, если в результате операции произошло переполнение, иначе сбрасывается в 0.

Перенос возникает:

- если результат сложения больше или равен 2^{32};
- если результат вычитания положителен или равен нулю;
- в результате работы внутренней схемы циклического сдвига при выполнении команд пересылки данных или команд логических операций.

Переполнение возникает, когда знак результата, содержащийся в бите [31], отличается от знака, который получился бы при бесконечно большой точности, например:

- если при сложении двух отрицательных чисел получается положительное число;
- если при сложении двух положительных чисел получается отрицательное число;
- если при вычитании положительного числа из отрицательного получается положительное число;
- если при вычитании отрицательного числа из положительного получается отрицательное число.

Операции сравнения полностью эквивалентны командам вычитания (в случае `CMP`) или сложения (в случае `CMN`), за исключением того, что результат операции не сохраняется. Для получения более подробной информации обратитесь к описанию команд.

Примечание

Большинство команд изменяют флаги состояния только в том случае, если в мнемонике команды указан суффикс `S` (см. описание команд).

Суффиксы условия выполнения

Команды, допускающие условное выполнение, имеют необязательное поле кода условия, определяемого двухсимвольным суффиксом {*cond*}. Условное выполнение требует предварительного выполнения команды `IT`. Команда, содержащая код условия, выполняется только в том случае, если состояние флагов условий регистра APSR соответствует заданному условию. Все допустимые суффиксы условия выполнения приведены в **Табл. А.2**.

Для уменьшения числа команд переходов в коде программы можно использовать условное выполнение команды `IT`. Также в **Табл. А.2** показана взаимосвязь между суффиксами условия выполнения и флагами N, Z, C и V.

В Примере А.1 показано использование условно выполняемой команды для вычисления абсолютного значения числа `R0 = ABS(R1)`.

В Примере А.2 показано использование условно выполняемых команд для изменения регистра R4 в том случае, если значение R0 больше R1 и R2 больше R3.

Пример А.1. Абсолютное значение

```
MOVS    R0, R1        ; R0 = R1, изменяет состояние флагов
IT      MI            ; IT – пропускает следующую команду, если значение >=0
RSBMI   R0, R1, #0    ; Если значение < 0, то R0 = -R1
```

Пример А.2. Сравнение и изменение значения

```
CMP     R0, R1    ; Сравниваем R0 и R1
ITT     GT        ; IT – пропускаем две следующие команды, если условие GT
                  ; не выполняется
CMPGT   R2, R3    ; Если «больше», сравниваем R2 и R3, устанавливаем флаги
MOVGT   R4, R5    ; Если всё ещё «больше», копируем R4 = R5
```

А.2.8. Выбор разрядности команды

Многие команды могут формировать как 16-, так и 32-битный машинный код в зависимости от используемых операндов и регистра-приёмника. Для некоторых из этих команд можно явно задать разрядность с помощью специального суффикса. Суффикс `.W` указывает на необходимость генерации 32-битного кода, а суффикс `.N` — 16-битного кода.

Если ассемблер не сможет сформировать машинный код заданной разрядности, то он выдаст сообщение об ошибке.

Примечание

Иногда явное указание суффикса `.W` необходимо; например, если операнд является меткой или константой, как в случае команд перехода. Эта необходимость обусловлена тем, что ассемблер может оказаться не в состоянии автоматически сгенерировать машинный код требуемой разрядности.

При использовании суффикса, определяющего разрядность команды, он помещается непосредственно после мнемоники команды и суффикса условия выполнения, при наличии последнего. В Примере А.3 показана запись команды с использованием суффикса разрядности.

Пример А.3. Явное указание разрядности команды

```
BCS.W   label       ; Генерирует 32-битный код даже для короткого перехода
ADDS.W  R0, R0, R1  ; Генерирует 32-битный код, даже если та же самая
                    ; операция может быть выполнена 16-битной командой
```

А.3. Команды доступа к памяти

Все поддерживаемые команды доступа к памяти перечислены в **Табл. А.3**.

Таблица А.3. **Команды доступа к памяти**

Мнемоника	Краткое описание	Подраздел
ADR	Генерация относительного адреса	А.3.1. ADR
CLREX	Сброс монопольного доступа	А.3.9. CLREX
LDM{mode}	Загрузка нескольких регистров	А.3.6. LDM и STM
LDR{type}	Загрузка регистра с использованием непосредственного смещения	А.3.2. LDR и STR, непосредственное смещение
LDR{type}	Загрузка регистра с использованием смещения в регистре	А.3.3. LDR и STR, регистровое смещение
LDR{type}T	Загрузка регистра в непривилегированном режиме	А.3.4. LDR и STR, непривилегированный доступ
LDR	Загрузка регистра с использованием относительного адреса	А.3.5. LDR, относительная адресация
LDREX{type}	Монопольная загрузка регистра	А.3.8. LDREX и STREX
POP	Извлечение регистра из стека	А.3.7. PUSH и POP
PUSH	Загрузка регистра в стек	А.3.7. PUSH и POP
STM{mode}	Сохранение нескольких регистров	А.3.6. LDM и STM
STR{type}	Сохранение регистра с использованием непосредственного смещения	А.3.2. LDR и STR, непосредственное смещение
STR{type}	Сохранение регистра с использованием смещения в регистре	А.3.3. LDR и STR, регистровое смещение
STR{type}T	Сохранение регистра в непривилегированном режиме	А.3.4. LDR и STR, непривилегированный доступ
STREX{type}	Монопольное сохранение регистра	А.3.8. LDREX и STREX

А.3.1. ADR

Генерация относительного адреса.

Синтаксис

```
ADR{cond} Rd, label
```

где

cond — необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».

Rd — регистр-приёмник.

label выражение, определяемое относительно PC; см. подраздел А.2.6 «Адресация относительно PC».

Операция

Команда ADR формирует адрес, прибавляя непосредственное значение к значению счётчика команд PC, и сохраняет результат в регистре-приёмнике.

Эта команда упрощает генерацию переместимого кода, поскольку адрес в данном случае задаётся относительно значения PC. Если команда ADR предназначена для формирования адреса, используемого командами BX или BLX, то программист должен гарантировать, что бит [0] полученного адреса будет установлен в 1.

Значение *label* должно находиться в пределах от –4095 до +4095 относительно адреса в счётчике команд.

Примечание

Для использования всего диапазона значений смещения или для формирования адресов, не выровненных на границу слова, может потребоваться использование суффикса .W (см. подраздел А.2.8 «Выбор разрядности команды»).

Ограничения

Rd не должен быть SP либо PC.

Флаги условия

Команда не влияет на состояние флагов.

Пример

```
ADR R1, TextMessage  ; Заносит в R1 адрес ячейки памяти, обозначенной
                     ; меткой TextMessage
```

A.3.2. LDR и STR, непосредственное смещение

Загрузка и сохранение регистров с непосредственным смещением и пред- или постиндексацией.

Синтаксис

```
op{type}{cond} Rt, [Rn {, #offset}]   ; Непосредственное смещение
op{type}{cond} Rt, [Rn, #offset]!     ; Прединдексация
op{type}{cond} Rt, [Rn], #offset      ; Постиндексация
opD{cond} Rt, Rt2, [Rn {, #offset}]   ; Непосредственное смещение, два слова
opD{cond} Rt, Rt2, [Rn, #offset]!     ; Прединдексация, два слова
opD{cond} Rt, Rt2, [Rn], #offset      ; Постиндексация, два слова
```

где

op операция:

- LDR — Загрузка регистра
- STR — Сохранение регистра

type	тип операнда:	
	B	Беззнаковый байт, при загрузке дополняется нулями до 32-битного значения
	SB	Знаковый байт, расширяется до 32-битного значения (только LDR)
	H	Беззнаковое полуслово, при загрузке дополняется нулями до 32-битного значения
	SH	Знаковое полуслово, расширяется до 32-битного значения (только LDR)
	—	Не указан, тип операнда — слово
cond	необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».	
Rt	загружаемый или сохраняемый регистр.	
Rn	регистр, содержащий базовый адрес.	
offset	смещение относительно *Rn*. Если смещение не указано, то адрес равен содержимому *Rn*.	
Rt2	дополнительный регистр для загрузки или сохранения двойных слов.	

Операция

Команды LDR загружают один или два регистра данными из памяти. Команды STR сохраняют содержимое одного или двух регистров в памяти.

Команды загрузки и сохранения с непосредственным смещением могут использовать следующие режимы адресации.

Относительная адресация

Величина смещения прибавляется или вычитается из адреса, находящегося в регистре *Rn*. Результат используется в качестве адреса, по которому производится обращение к памяти. Регистр *Rn* не изменяется. Синтаксис языка ассемблера для этого режима адресации имеет вид:

```
[Rn, #offset]
```

Адресация с прединдексированием

Величина смещения прибавляется или вычитается из адреса, находящегося в регистре *Rn*. Результат используется в качестве адреса, по которому производится обращение к памяти, и сохраняется в регистре *Rn*. Синтаксис языка ассемблера для этого режима имеет вид:

```
[Rn, #offset]!
```

Адресация с постиндексированием

Значение, находящееся в регистре *Rn*, используется в качестве адреса, по которому производится обращение к памяти. После выполнения операции величина смещения прибавляется или вычитается из этого значения и результат сохраняется в регистре *Rn*. Синтаксис языка ассемблера для данного режима имеет вид:

```
[Rn], #offset
```

Загружаемое или сохраняемое значение может иметь разрядность байт, полуслово, слово или двойное слово. В командах загрузки байты и полуслова могут быть как знаковыми, так и беззнаковыми (см. подраздел A.2.5 «Выравнивание адреса»).

Допустимые значения смещения для всех режимов адресации приведены в **Табл. A.4**.

***Таблица A.4.* Допустимые значения смещения**

Тип операнда	Непосредственное смещение	Прединдексирование	Постиндексирование
Слово, полуслово (со знаком и без знака), байт (со знаком и без знака)	–255…4095	–255…255	–255…255
Двойное слово	Кратное 4 в диапазоне –1020…1020	Кратное 4 в диапазоне –1020…1020	Кратное 4 в диапазоне –1020…1020

Ограничения

Для команд загрузки:

- *Rt* может быть SP или PC только в командах загрузки слова.
- *Rt* должен отличаться от *Rt2*.
- *Rn* должен отличаться от *Rt* и *Rt2* в режимах с пред- и постиндексированием.

При использовании PC в качестве *Rt* в команде загрузки двойного слова:

- бит [0] загружаемого значения должен быть установлен в 1 для корректного выполнения команды;
- переход производится по адресу, получаемому сбросом бита [0] загруженного значения в 0;
- если команда условно выполняемая, то она должна быть последней командой в IT-блоке.

Для команд сохранения:

- *Rt* может быть SP только в командах сохранения двойного слова.
- *Rt* не должен быть PC.
- *Rn* не должен быть PC.
- *Rn* должен отличаться от *Rt* и *Rt2* в режимах с пред- и постиндексированием.

Флаги условий

Команда не влияет на состояние флагов.

Примеры

```
LDR    R8, [R10]             ; Загружает R8 из памяти по адресу, находящемуся
                             ; в R10
LDRNE  R2, [R5, #960]!       ; Загружает (условно) R2 из слова, расположенного со
                             ; смещением +960 байт относительно адреса,
                             ; находящегося в R5, и увеличивает R5 на 960
STR    R2, [R9,#const-struc] ; const-struc – это выражение, результатом которого
```

```
                              ; является константа из диапазона 0...4095
STRH   R3, [R4], #4           ; Сохраняет R3 как полуслово по адресу, находящемуся
                              ; в R4, затем увеличивает R4 на 4
LDRD   R8, R9, [R3, #0x20]    ; Загружает R8 и R9 из слов, расположенных
                              ; со смещениями +32 байта и
                              ; +36 байт соответственно
                              ; относительно адреса, находящегося в R3
STRD   R0, R1, [R8], #-16     ; Сохраняет R0 по адресу, находящемуся в R8,
                              ; а R1 – в слове памяти по адресу,
                              ; который на 4 байта больше адреса, находящегося
                              ; в R8, после чего уменьшает R8 на 16
```

A.3.3. LDR и STR, регистровое смещение

Загрузка и сохранение регистров с регистровым смещением.

Синтаксис

```
op{type}{cond} Rt, [Rn, Rm {, LSL #n}]
```

где

op — операция:

LDR	Загрузка регистра
STR	Сохранение регистра

type — тип операнда:

B	Беззнаковый байт, при загрузке дополняется нулями до 32-битного значения
SB	Знаковый байт, расширяется до 32-битного значения (только LDR)
H	Беззнаковое полуслово, при загрузке дополняется нулями до 32-битного значения
SH	Знаковое полуслово, расширяется до 32-битного значения (только LDR)
—	Не указан, тип операнда — слово

cond — необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».

Rt — загружаемый или сохраняемый регистр.

Rn — регистр, содержащий базовый адрес.

Rm — регистр, содержащий смещение.

LSL *#n* — необязательный сдвиг, $0 \le n \le 3$.

Операция

Команды LDR загружают один регистр данными из памяти.

Команды STR сохраняют содержимое одного регистра в памяти.

Адрес памяти для загрузки или сохранения данных задаётся относительно базового адреса, находящегося в регистре *Rn*. Величина смещения определяется

содержимым регистра *Rm*, которое может быть сдвинуто на 0…3 бита влево посредством команды LSL.

Загружаемое или сохраняемое значение может иметь разрядность байт, полуслово или слово. В командах загрузки байты и полуслова могут быть как знаковыми, так и беззнаковыми (см. подраздел A.2.5 «Выравнивание адреса»).

Ограничения

- *Rn* не должен быть PC.
- *Rm* не должен быть SP или PC.
- *Rt* может быть SP или PC только в командах загрузки/сохранения слова.
- *Rt* может быть PC только в командах загрузки слова.

При использовании PC в качестве *Rt* в команде загрузки слова:

- бит [0] загружаемого значения должен быть установлен в 1 для корректного выполнения команды; переход осуществляется по этому адресу, выровненному на границу полуслова;
- если команда условно выполняемая, то она должна быть последней командой в IT-блоке.

Флаги условий

Команда не влияет на состояние флагов.

Примеры

```
STR    R0, [R5, R1]          ; Сохраняет содержимое R0 по адресу, равному R5 + R1
LDRSB  R0, [R5, R1, LSL #1]  ; Загружает однобайтное значение с адреса,
                             ; равного R5 + 2*R1, выполняет расширение знака
                             ; до размера слова и помещает результат в R0
STR    R0, [R1, R2, LSL #2]  ; Сохраняет R0 по адресу, равному R1 + 4*R2
```

A.3.4. LDR и STR, непривилегированный доступ

Загрузка и сохранение регистров на непривилегированном уровне доступа.

Синтаксис

```
op{type}T{cond} Rt, [Rn {, #offset}]  ; Непосредственное смещение
```

где

op операция:

LDR	Загрузка регистра
STR	Сохранение регистра

type тип операнда:

B	Беззнаковый байт, при загрузке дополняется нулями до 32-битного значения
SB	Знаковый байт, расширяется до 32-битного значения (только LDR)

	`H`	Беззнаковое полуслово, при загрузке дополняется нулями до 32-битного значения
	`SH`	Знаковое полуслово, расширяется до 32-битного значения (только `LDR`)
	—	Не указан, тип операнда — слово
`cond`	необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».	
`Rt`	загружаемый или сохраняемый регистр.	
`Rn`	регистр, содержащий базовый адрес.	
`offset`	смещение относительно *`Rn`* (от 0 до 255). Если смещение не указано, то адрес равен содержимому *`Rn`*.	

Операция

Эти команды выполняют те же операции, что и команды доступа к памяти с непосредственным смещением (см. подраздел А.3.2 «LDR и STR, непосредственное смещение»). Отличие заключается в том, что эти команды осуществляют обращение к памяти исключительно на непривилегированном уровне, даже в коде, выполняющемся в привилегированном режиме.

При использовании в коде, выполняющемся в непривилегированном режиме, эти команды полностью идентичны обычным командам доступа к памяти с непосредственным смещением.

Ограничения

- *`Rn`* не должен быть PC.
- *`Rt`* не должен быть SP или PC.

Флаги условий

Команда не влияет на состояние флагов.

Примеры

```
STRBTEQ R4, [R7]            ; Сохраняет (условно) младший значащий байт
                            ; содержимого R4 по адресу, содержащемуся в R7,
                            ; на непривилегированном уровне
LDRHT   R2, [R2, #8]        ; Загружает в R2 полуслово с адреса, равного R8 + 8,
                            ; на непривилегированном уровне
```

A.3.5. LDR, относительная адресация

Загрузка регистра из памяти.

Синтаксис

```
LDR{type}{cond} Rt, label
LDRD{cond} Rt, Rt2, label             ; Загрузка двух слов
```

где

type	тип операнда:	
	B	Беззнаковый байт, при загрузке дополняется нулями до 32-битного значения
	SB	Знаковый байт, расширяется до 32-битного значения (только LDR)
	H	Беззнаковое полуслово, при загрузке дополняется нулями до 32-битного значения
	SH	Знаковое полуслово, расширяется до 32-битного значения (только LDR)
	—	Не указан, тип операнда — слово
cond	необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».	
Rt	загружаемый или сохраняемый регистр.	
Rt2	дополнительный регистр для загрузки двойного слова.	
label	выражение, определяемое относительно PC; см. подраздел A.2.6 «Адресация относительно PC».	

Операция

Команда LDR загружает в регистр данные из памяти. Адрес ячейки памяти определяется меткой (смещением относительно PC).

Загружаемое значение может иметь разрядность байт, полуслово или слово. Байты и полуслова могут быть как знаковыми, так и беззнаковыми (см. подраздел A.2.5 «Выравнивание адреса»).

Значение *label* должно находиться в допустимом для данной команды диапазоне. Допустимые величины смещения *label* относительно PC указаны в **Табл. A.5**.

***Таблица A.5.* Допустимые значения смещения**

Тип операнда	Диапазон значений смещения
Слово, полуслово (со знаком и без знака), байт (со знаком и без знака)	–4095…+4095
Двойное слово	–1020…+1020

Примечание

Для использования всего диапазона значений смещения может потребоваться использование суффикса .W (см. подраздел A.2.8 «Выбор разрядности команды»).

Ограничения

- *Rt* может быть SP или PC только в командах загрузки слова.
- *Rt2* не должен быть SP или PC.
- *Rt* должен отличаться от *Rt2*.

При использовании PC в качестве *Rt* в команде загрузки двойного слова:
- бит [0] загружаемого значения должен быть установлен в 1 для корректного выполнения команды; переход осуществляется по этому адресу, выровненному на границу полуслова;
- если команда условно выполняемая, то она должна быть последней командой в IT-блоке.

Флаги условий

Команда не влияет на состояние флагов.

Примеры

```
LDR    R0, LookUpTable       ; Загружает в R0 слово, расположенное по адресу,
                             ; помеченному меткой LookUpTable
LDRSB  R7, localdata         ; Считывает байт с адреса, помеченного меткой
                             ; localdata, выполняет расширение знака до размера
                             ; слова и помещает результат в R7
```

A.3.6. LDM и STM

Загрузка и сохранение нескольких регистров.

Синтаксис

op{*addr_mode*}{*cond*} *Rn*{!}, *reglist*

где

op — операция:
- LDM — Загрузка нескольких регистров
- STM — Сохранение нескольких регистров

addr _ mode — режим адресации:
- IA — Инкрементирование адреса после каждого обращения к памяти (режим по умолчанию)
- DB — Декрементирование адреса перед каждым обращением к памяти

cond — необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».

Rn — регистр, содержащий базовый адрес.

! — необязательный суффикс обратной записи. При наличии в мнемонике команды символа «!» последний адрес, по которому производилось чтение или запись данных, сохраняется в регистре *Rn*.

reglist — список из одного или нескольких регистров, загружаемых из памяти или сохраняемых в памяти, заключённый в фигурные скобки. Может включать в себя диапазоны регистров. При наличии в списке более одного регистра или диапазона регистров элементы списка разделяются запятыми (см. пункт «Примеры» данного подраздела).

- LDM и LDMFD являются синонимами LDMIA. Мнемоника LDMFD указывает на использование данной команды для извлечения данных из «полного» убывающего стека.
- LDMEA является синонимом LDMDB и указывает на использование данной команды для извлечения данных из «пустого» возрастающего стека.
- STM и STMEA являются синонимами STMIA. Мнемоника STMEA указывает на использование данной команды для помещения данных в «пустой» возрастающий стек.
- STMFD является синонимом STMDB и указывает на использование данной команды для помещения данных в «полный» убывающий стек.

Операция

Команды LDR загружают в регистры, указанные в *reglist*, слова из памяти, начиная с адреса, указанного в *Rn*. Команды STM сохраняют содержимое регистров, указанных в *reglist*, в памяти, начиная с адреса, указанного в *Rn*.

Для команд LDM, LDMIA, LDMFD, STM, STMIA и STMEA адреса ячеек памяти, по которым производятся обращения, располагаются в диапазоне от *Rn* до *Rn* + 4 * (*n* – 1) с интервалом в 4 байта, где *n* — число регистров в *reglist*. Обращения к памяти производятся в порядке возрастания номеров регистров, при этом регистру с наименьшим номером соответствует самый младший адрес, а регистру с наибольшим номером — самый старший адрес. При наличии в мнемонике команды суффикса, разрешающего обратную запись, значение *Rn* + 4 * (*n*–1) сохраняется в регистре *Rn*.

Для команд LDMDB, LDMEA, STMDB и STMFD адреса ячеек памяти, по которым производятся обращения, располагаются в диапазоне от *Rn* до *Rn* – 4 * (*n* – 1) с интервалом в 4 байта, где *n* — число регистров в *reglist*. Обращения к памяти производятся в порядке убывания номеров регистров, при этом регистру с наибольшим номером соответствует самый старший адрес, а регистру с наименьшим номером — самый младший адрес. При наличии в мнемонике команды суффикса, разрешающего обратную запись, значение *Rn* – 4 * (*n* – 1) сохраняется в регистре *Rn*.

Подобные мнемоники могут быть использованы для записи команд PUSH и POP; см. подраздел A.3.7 «PUSH и POP».

Ограничения

- *Rn* не должен быть PC.
- *reglist* не должен содержать SP.
- *Rt* должен отличаться от *Rt2*.
- В любой команде STM *reglist* не должен содержать PC.
- В любой команде LDM *reglist* не должен содержать PC, если в нём содержится LR.
- *reglist* не должен содержать *Rn*, если в команде указан суффикс обратной записи.

При наличии PC в *reglist* команды LDM:

- бит [0] загружаемого значения должен быть установлен в 1 для корректного выполнения команды; переход осуществляется по этому адресу, выровненному на границу полуслова;
- если команда условно выполняемая, то она должна быть последней командой в IT-блоке.

Флаги условий

Команда не влияет на состояние флагов.

Примеры

```
LDM    R8, {R0,R2,R9}        ; LDMIA - синоним для LDM
STMDB  R1!, {R3-R6,R11,R12}
```

Примеры неправильной записи

```
STM    R5!, {R5,R4,R9}       ; Значение, сохраняемое в R5, непредсказуемо
LDM    R2, {}                ; В списке должен находиться хотя бы один регистр
```

A.3.7. PUSH и POP

Загрузка регистров в стек и извлечение регистров из стека («полный» убывающий стек).

Синтаксис

```
PUSH{cond} reglist
POP{cond} reglist
```

где

cond — необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».

reglist — непустой список регистров, заключённый в фигурные скобки. Может включать в себя диапазоны регистров. При наличии в списке более одного регистра или диапазона регистров элементы списка разделяются запятыми.

Команды PUSH и POP являются синонимами для команд STMDB и LDM (или LDMIA), использующих в качестве регистра базового адреса регистр SP и сохраняющих в этом регистре последний адрес. В таких случаях предпочтительно использовать мнемоники PUSH и POP.

Операция

Команда PUSH сохраняет содержимое регистров в стеке, при этом регистр с наименьшим номером сохраняется в слове памяти с самым младшим адресом, а регистр с наибольшим номером — в слове с самым старшим адресом.

Команда POP загружает регистры из стека, при этом регистр с наименьшим номером загружается из слова памяти с самым младшим адресом, а регистр с наибольшим номером — из слова с самым старшим адресом.

Старшим адресом для команды PUSH является содержимое регистра SP, уменьшенное на 4, а младшим адресом для команды POP — содержимое регистра SP, т.е. эти команды реализуют «полный» убывающий стек. После завершения операции команда PUSH изменяет регистр SP таким образом, чтобы тот указывал на слово памяти, в котором был сохранён регистр с наименьшим номером. Команда POP после завершения операции изменяет регистр SP таким образом, чтобы тот указывал на слово памяти, расположенное выше того слова, из которого был считан регистр с наибольшим номером.

Если в списке *reglist* команды POP присутствует регистр PC, то переход по соответствующему адресу будет произведён после завершения команды POP. При этом бит [0] считанного из стека значения заносится в бит T регистра APSR. Для корректной работы процессора указанный бит должен быть установлен в 1.

Для получения дополнительной информации см. подраздел A.3.6 «LDM и STM».

Ограничения

- *reglist* не должен содержать SP.
- *reglist* команды PUSH не должен содержать PC.
- *reglist* команды POP не должен содержать PC, если в нём содержится LR.

При наличии PC в *reglist* команды POP:

- бит [0] загружаемого значения должен быть установлен в 1 для корректного выполнения команды; переход осуществляется по этому адресу, выровненному на границу полуслова;
- если команда условно выполняемая, то она должна быть последней командой в IT-блоке.

A.3.8. LDREX и STREX

Загрузка и сохранение регистров в режиме монопольного доступа.

Синтаксис

```
LDREX{cond} Rt, [Rn {, #offset}]
STREX{cond} Rd, Rt, [Rn {, #offset}]
LDREXB{cond} Rt, [Rn]
STREXB{cond} Rd, Rt, [Rn]
LDREXH{cond} Rt, [Rn]
STREXH{cond} Rd, Rt, [Rn]
```

где

cond	необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».
Rd	регистр-приёмник для сохранения возвращённого кода состояния.
Rt	загружаемый или сохраняемый регистр.

Rn	регистр, содержащий базовый адрес.
offset	смещение относительно *Rn*. Если смещение не указано, то адрес равен содержимому *Rn*.

Операция

Команды LDREX, LDREXB и LDREXH загружают из памяти в регистр слово, байт и полуслово соответственно. Команды STREX, STREXB и STREXH пытаются сохранить в памяти слово, байт и полуслово соответственно. Адрес, используемый в любой команде монопольного сохранения данных, должен быть идентичен адресу в последней выполненной команде монопольной загрузки. Значение, сохраняемое в памяти командой монопольного сохранения, должно иметь такую же разрядность, что и значение, считанное последней командой монопольной загрузки. То есть для осуществления синхронизации команда монопольной загрузки должна всегда использоваться в паре с аналогичной командой монопольного сохранения.

При сохранении данных команда монопольного сохранения заносит 0 в регистр-приёмник. Если попытка сохранения была неудачной, то в регистр-приёмник заносится 1. Нулевое значение, возвращённое командой сохранения в регистре-приёмнике, гарантирует, что ни один процесс в системе не сможет получить доступ к использованной ячейке памяти между командами монопольной загрузки и монопольного сохранения.

Из соображений быстродействия число команд, располагающихся между командой монопольной загрузки и командой монопольного сохранения, должно быть сведено к минимуму.

Примечание

Результат выполнения команды монопольного сохранения при обращении по адресу, отличному от использованного в предшествующей команде монопольной загрузки, будет непредсказуемым.

Ограничения

- Нельзя использовать PC.
- *Rd* не должен быть SP.
- *Rt* не должен быть SP.
- В командах STREX регистр *Rd* должен отличаться от *Rt* и от *Rn*.
- Величина *offset* должна находиться в диапазоне 0…1020 и быть кратной 4.

Флаги условий

Команда не влияет на состояние флагов.

Примеры

```
    MOV     R1, #0x1            ; Инициализируем значение для признака блокировки
try
    LDREX   R0, [LockAddr]      ; Читаем признак блокировки
    CMP     R0, #0              ; Блокировка снята?
    ITT     EQ                  ; IT-блок для команд STREXEQ и CMPEQ
```

```
STREXEQ R0, R1, [LockAddr] ; Пробуем поставить блокировку
CMPEQ   R0, #0             ; Успешно?
BNE     try                ; Нет – пробуем снова
....                       ; Да – поставили блокировку
```

A.3.9. CLREX

Сброс монопольного доступа.

Синтаксис

```
CLREX{cond}
```

где

cond необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».

Операция

После команды `CLREX` любая последующая команда `STREX`, `STREXB` и `STREXH` заносит 1 в регистр-приёмник и, соответственно, не осуществляет запись в память. Принудительное блокирование монопольной записи в память может потребоваться в обработчике исключительной ситуации в том случае, если таковая возникнет во время операции синхронизации между командой монопольной загрузки и соответствующей командой монопольного сохранения.

Флаги условий

Команда не влияет на состояние флагов.

Примеры

```
CLREX
```

A.4. Общие команды обработки данных

Все команды обработки данных перечислены в **Табл. A.6**.

Таблица A.6. **Команды обработки данных**

Мнемоника	Краткое описание	Подраздел
ADC	Сложение с переносом	A.4.1. ADD, ADC, SUB, SBC и RSB
ADD	Сложение	A.4.1. ADD, ADC, SUB, SBC и RSB
ADDW	Сложение	A.4.1. ADD, ADC, SUB, SBC и RSB
AND	Логическое И	A.4.2. AND, ORR, EOR, BIC и ORN
ASR	Арифметический сдвиг вправо	A.4.3. ASR, LSL, LSR, ROR и RRX
BIC	Очистка битов	A.4.2. AND, ORR, EOR, BIC и ORN
CLZ	Подсчёт ведущих нулевых битов	A.4.4. CLZ
CMN	Сравнение с отрицательным операндом	A.4.5. CMP и CMN
CMP	Сравнение	A.4.5. CMP и CMN
EOR	Исключающее ИЛИ	A.4.2. AND, ORR, EOR, BIC и ORN

Таблица A.6. **Команды обработки данных** *(продолжение)*

Мнемоника	Краткое описание	Подраздел
LSL	Логический сдвиг влево	A.4.3. ASR, LSL, LSR, ROR и RRX
LSR	Логический сдвиг вправо	A.4.3. ASR, LSL, LSR, ROR и RRX
MOV	Пересылка	A.4.6. MOV и MVN
MOVT	Пересылка старшего полуслова	A.4.7. MOVT
MOVW	Пересылка 16-битной константы	A.4.6. MOV и MVN
MVN	Пересылка с инверсией	A.4.6. MOV и MVN
ORN	Логическое ИЛИ с инверсией	A.4.2. AND, ORR, EOR, BIC и ORN
ORR	Логическое ИЛИ	A.4.2. AND, ORR, EOR, BIC и ORN
RBIT	Перестановка битов	A.4.8. REV, REV16, REVSH и RBIT
REV	Перестановка байтов слова	A.4.8. REV, REV16, REVSH и RBIT
REV16	Перестановка байтов в каждом полуслове	A.4.8. REV, REV16, REVSH и RBIT
REVSH	Перестановка байтов в младшем полуслове и расширение знака	A.4.8. REV, REV16, REVSH и RBIT
ROR	Циклический сдвиг вправо	A.4.3. ASR, LSL, LSR, ROR и RRX
RRX	Расширенный циклический сдвиг вправо	A.4.3. ASR, LSL, LSR, ROR и RRX
RSB	Обратное вычитание	A.4.1. ADD, ADC, SUB, SBC и RSB
SBC	Вычитание с переносом	A.4.1. ADD, ADC, SUB, SBC и RSB
SUB	Вычитание	A.4.1. ADD, ADC, SUB, SBC и RSB
SUBW	Вычитание	A.4.1. ADD, ADC, SUB, SBC и RSB
TEQ	Проверка на равенство	A.4.9. TST и TEQ
TST	Проверка битов	A.4.9. TST и TEQ

A.4.1. ADD, ADC, SUB, SBC и RSB

Сложение, сложение с переносом, вычитание, вычитание с переносом и обратное вычитание.

Синтаксис

```
op{S}{cond} {Rd,} Rn, Operand2
op{cond} {Rd,} Rn, #imm12          ; Только команды ADD и SUB
```

где

op — операция:

- ADD — Сложение
- ADC — Сложение с переносом
- SUB — Вычитание
- SBC — Вычитание с переносом
- RSB — Обратное вычитание

S — необязательный суффикс. При наличии в мнемонике команды суффикса S состояние флагов условий изменяется в соответствии

	с результатом операции; см. подраздел A.2.7 «Условное выполнение».
cond	необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».
Rd	регистр-приёмник. Если *Rd* отсутствует, то регистром-приёмником является *Rn*.
Rn	регистр, содержащий первый операнд.
Operand2	«гибкий» второй операнд; для получения дополнительной информации обратитесь к подразделу A.2.3 «Гибкий» второй операнд».
imm12	любое число в диапазоне 0…4095.

Операция

Команда ADD складывает *Operand2* или *imm12* с содержимым *Rn*. Команда ADC складывает *Operand2* с содержимым *Rn* и прибавляет к результату значение флага переноса.

Команда SUB вычитает *Operand2* или *imm12* из содержимого *Rn*. Команда SBC вычитает *Operand2* из содержимого *Rn* и, если флаг переноса сброшен, уменьшает результат на единицу.

Команда RSB вычитает значение, находящееся в *Rn*, из *Operand2*. Достоинством этой команды является большое число опций, поддерживаемых операндом *Operand2*.

Команды ADC и SBC используются для реализации функций многословной арифметики; см. пункт «Примеры многословных арифметических операций» далее в этом подразделе.

Примечание

Мнемоники ADDW и SUBW эквивалентны мнемоникам ADD и SUB соответственно при использовании в качестве операнда константы *imm12*.

Ограничения

- *Operand2* не должен быть SP или PC.
- *Rd* может быть SP только в командах ADD и SUB со следующими ограничениями:
  - *Rn* также должен быть SP;
  - любые операции сдвига в *Operand2* должны быть ограничены сдвигом влево не более чем на 3 бита.
- *Rn* может быть SP только в командах ADD и SUB.
- *Rd* может быть PC только в команде ADD{*cond*} PC, PC, Rm, причём:
  - использование суффикса S не допускается;
  - *Rm* не должен быть PC или SP;
  - если команда условно выполняемая, то она должна быть последней командой в IT-блоке.

- *Rn* может быть PC не только в команде ADD{*cond*} PC, PC, Rm, но и в других вариантах команд ADD и SUB со следующими ограничениями:
  - использование суффикса S не допускается;
  - второй операнд должен быть константой из диапазона 0…4095.

Примечание

- При использовании PC в операциях сложения и вычитания биты [1:0] регистра PC сбрасываются в 00 перед выполнением операции, выравнивая тем самым базовый адрес на границу полуслова.
- При необходимости генерации адреса команды вы должны скорректировать значение константы в соответствии со значением PC. Специалисты компании ARM рекомендуют вместо команды ADD или SUB с *Rn*, равным PC, использовать команду ADR, поскольку для последней ассемблер автоматически вычисляет правильное значение константы.

При использовании PC в качестве *Rd* в команде ADD{*cond*} PC, PC, Rm:

- бит [0] значения, загружаемого в PC, игнорируется;
- переход производится по адресу, получаемому сбросом бита [0] загружаемого значения в 0.

Флаги условия

Если суффикс S указан, то эти команды изменяют состояние флагов N, Z, C и V в соответствии с результатом операции.

Примеры

```
ADD     R2, R1, R3
SUBS    R8, R6, #240    ; Устанавливает флаги в соответствии с результатом
RSB     R4, R4, #1280   ; Вычитает содержимое R4 из константы 1280
ADCHI   R11, R0, R3     ; Сложение выполняется, только если флаг C установлен,
                        ; а флаг Z – сброшен
```

Примеры многословных арифметических операций

В Примере А.4 показаны две команды, которые выполняют сложение 64-битного целого, находящегося в регистрах R2 и R3, с другим 64-битным целым, находящимся в регистрах R0 и R1, и помещают результат в регистры R4 и R5.

Многословные значения не обязательно должны храниться в последовательно расположенных регистрах. В Примере А.5 приводится фрагмент кода, который вычитает одно 96-битное целое, хранящееся в регистрах R9, R1 и R11, из другого, хранящегося в регистрах R6, R2 и R8. Результат вычитания сохраняется в регистрах R6, R9 и R2.

Пример А.4. Сложение 64-битных чисел

```
ADDS    R4, R0, R2      ; Складываем младшие слова
ADC     R5, R1, R3      ; Складываем старшие слова, учитывая перенос
```

Пример A.5. Вычитание 96-битных чисел

```
SUBS    R6, R6, R9      ; Вычитаем младшие слова
SBCS    R9, R2, R1      ; Вычитаем средние слова, учитывая перенос
SBC     R2, R8, R11     ; Вычитаем старшие слова, учитывая перенос
```

A.4.2. AND, ORR, EOR, BIC и ORN

Логическое И, Логическое ИЛИ, Исключающее ИЛИ, очистка битов и Логическое ИЛИ с инверсией.

Синтаксис

```
op{S}{cond} {Rd,} Rn, Operand2
```

где

op операция:

AND	Логическое И
ORR	Логическое ИЛИ
EOR	Исключающее ИЛИ
BIC	Логическое И с инверсией или, иначе, очистка битов
ORN	Логическое ИЛИ с инверсией

S необязательный суффикс. При наличии в мнемонике команды суффикса S состояние флагов условий изменяется в соответствии с результатом операции; см. подраздел A.2.7 «Условное выполнение».

cond необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».

Rd регистр-приёмник. Если *Rd* отсутствует, то регистром-приёмником является *Rn*.

Rn регистр, содержащий первый операнд.

Operand2 «гибкий» второй операнд; для получения дополнительной информации, обратитесь к подразделу A.2.3 «Гибкий» второй операнд».

Операция

Команды AND, EOR и ORR выполняют побитовые операции «И», «ИЛИ» и «Исключающее ИЛИ» соответственно между содержимым *Rn* и *Operand2*.

Команда BIC выполняет побитовую операцию «И» между содержимым *Rn* и инвертированным значением *Operand2*.

Команда ORN выполняет побитовую операцию «ИЛИ» между содержимым *Rn* и инвертированным значением *Operand2*.

Ограничения

Нельзя использовать SP либо PC.

Флаги условия

Если суффикс `S` указан, то эти команды:

- изменяют флаги N и Z в соответствии с результатом операции;
- могут изменять состояние флага C при вычислении *`Operand2`*; см. подраздел А.2.3 «Гибкий» второй операнд»;
- не влияют на состояние флага V.

Примеры

```
AND     R9, R2, #0xFF00
ORREQ   R2, R0, R5
ANDS    R9, R8, #0x19
EORS    R7, R11, #0x18181818
BIC     R0, R1, #0xab
ORNS    R7, R11, R14, ROR #4
ORNS    R7, R11, R14, ASR #32
```

A.4.3. ASR, LSL, LSR, ROR и RRX

Арифметический сдвиг вправо, логический сдвиг влево, логический сдвиг вправо, циклический сдвиг вправо и расширенный циклический сдвиг вправо.

Синтаксис

`op``{S}{`*`cond`*`}` *`Rd`*`,` *`Rm`*`,` *`Rs`*

`op``{S}{`*`cond`*`}` *`Rd`*`,` *`Rm`*`,` *`#n`*

`RRX{S}{`*`cond`*`}` *`Rd`*`,` *`Rm`*

где

`op` — операция:

`ASR`	Арифметический сдвиг вправо
`LSL`	Логический сдвиг влево
`LSR`	Логический сдвиг вправо
`ROR`	Циклический сдвиг вправо

`S` — необязательный суффикс. При наличии в мнемонике команды суффикса `S` состояние флагов условий изменяется в соответствии с результатом операции; см. подраздел А.2.7 «Условное выполнение».

`cond` — необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».

`Rd` — регистр-приёмник.

`Rm` — регистр, содержащий сдвигаемое значение.

`Rs` — регистр, в котором хранится величина сдвига для содержимого *`Rm`*. Используется только младший байт регистра; соответственно, величина сдвига может быть в диапазоне 0…255.

n Величина сдвига. Диапазон допустимых значений зависит от команды:

ASR	от 1 до 32
LSL	от 0 до 31
LSR	от 1 до 32
ROR	от 1 до 31

Примечание

Вместо команды `LSLS Rd, Rm, #0` лучше использовать команду `MOVS Rd, Rm`.

Операция

Команды ASR, LSL, LSR и ROR сдвигают содержимое регистра *Rm* влево или вправо на число битов, определяемое константой *n* или регистром *Rs*. Команда RRX сдвигает содержимое регистра *Rm* на один бит вправо.

Все рассматриваемые команды сохраняют результат в регистре *Rd*, при этом содержимое регистра *Rm* не изменяется. Более подробно действия, выполняемые отдельными командами сдвига, описаны в подразделе A.2.4 «Операции сдвига».

Ограничения

Нельзя использовать SP либо PC.

Флаги условия

Если суффикс S указан, то:

- эти команды изменяют состояние флагов N и Z;
- во флаг C заносится значение последнего выдвинутого бита, если величина сдвига не равна нулю; см. подраздел A.2.4 «Операции сдвига».

Примеры

```
ASR     R7, R8, #9  ; Арифметический сдвиг вправо на 9 бит
LSLS    R1, R2, #3  ; Логический сдвиг влево на 3 бита с изменением флагов
LSR     R4, R5, #6  ; Логический сдвиг вправо на 6 бит
ROR     R4, R5, R6  ; Циклический сдвиг вправо на число битов, определяемое
                    ; младшим битом R6
RRX     R4, R5      ; Расширенный циклический сдвиг вправо
```

A.4.4. CLZ

Подсчёт ведущих нулевых битов.

Синтаксис

```
CLZ{cond} Rd, Rm
```

где

cond необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».

Rd регистр-приёмник.

Rm регистр операнда.

Операция

Команда CLZ определяет число ведущих нулевых битов в содержимом *Rm* и возвращает результат в *Rd*. Результат равен 32, если ни один бит регистра *Rm* не установлен, и 0, если установлен бит [31] регистра.

Ограничения

Нельзя использовать SP либо PC.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
CLZ     R4, R9
CLZNE   R2, R3
```

A.4.5. CMP и CMN

Сравнение и сравнение с отрицательным операндом.

Синтаксис

```
CMP{cond} Rn, Operand2
CMN{cond} Rn, Operand2
```

где

cond необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».

Rn регистр, содержащий первый операнд.

Operand2 «гибкий» второй операнд; для получения дополнительной информации обратитесь к подразделу А.2.3 «Гибкий» второй операнд».

Операция

Эти команды выполняют сравнение содержимого регистра *Rn* со значением *Operand2*. Они изменяют флаги условий в соответствии с результатом сравнения, но не сохраняют результат в регистре.

Команда CMP вычитает *Operand2* из содержимого регистра *Rm*, т.е. выполняет ту же операцию, что и команда SUBS, но без сохранения результата вычитания.

Команда CMN складывает *Operand2* с содержимым регистра *Rm*, т.е. выполняет ту же операцию, что и команда ADDS, но без сохранения результата сложения.

Ограничения

- Нельзя использовать PC.
- *Operand2* не должен быть SP.

Флаги условия

Эти команды изменяют состояние флагов N, Z, C и V в соответствии с результатом операции.

Примеры

```
CMP     R2, R9
CMN     R0, #6400
CMPGT   SP, R7, LSL #2
```

A.4.6. MOV и MVN

Пересылка и пересылка с инверсией.

Синтаксис

```
MOV{S}{cond} Rd, Operand2
MOV{cond} Rd, #imm16
MVN{S}{cond} Rd, Operand2
```

где

`S`	необязательный суффикс. При наличии в мнемонике команды суффикса `S` состояние флагов условий изменяется в соответствии с результатом операции; см. подраздел А.2.7 «Условное выполнение».
`cond`	необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».
`Rd`	регистр-приёмник. Если *`Rd`* отсутствует, то регистром-приёмником является *`Rn`*.
`Operand2`	«гибкий» второй операнд; для получения дополнительной информации обратитесь к подразделу А.2.3 «Гибкий» второй операнд».
`imm16`	любое число в диапазоне 0…65535.

Операция

Команда `MOV` копирует значение *`Operand2`* в регистр *`Rd`*. Если вторым операндом команды `MOV` является регистр со сдвигом, отличным от `LSL #0`, то вместо команд `MOV` предпочтительней использовать соответствующие команды сдвига:

- `ASR{S}{cond} Rd, Rm, #n` вместо `MOV{S}{cond} Rd, Rm, ASR #n`.
- `LSL{S}{cond} Rd, Rm, #n` вместо `MOV{S}{cond} Rd, Rm, LSL #n`, если $n \neq 0$.
- `LSR{S}{cond} Rd, Rm, #n` вместо `MOV{S}{cond} Rd, Rm, LSR #n`.
- `ROR{S}{cond} Rd, Rm, #n` вместо `MOV{S}{cond} Rd, Rm, ROR #n`.
- `RRX{S}{cond} Rd, Rm` вместо `MOV{S}{cond} Rd, Rm, RRX`.

Помимо этого, команда `MOV` поддерживает дополнительные формы записи *`Operand2`*, что позволяет использовать её вместо команд сдвига:

- `MOV{S}{cond} Rd, Rm, ASR Rs` эквивалентна `ASR{S}{cond} Rd, Rm, Rs`.
- `MOV{S}{cond} Rd, Rm, LSL Rs` эквивалентна `LSL{S}{cond} Rd, Rm, Rs`.

- `MOV{S}{cond} Rd, Rm, LSR Rs` эквивалентна `LSR{S}{cond} Rd, Rm, Rs`.
- `MOV{S}{cond} Rd, Rm, ROR Rs` эквивалентна `ROR{S}{cond} Rd, Rm, Rs`.

См. подраздел А.4.3 «ASR, LSL, LSR, ROR и RRX».

Команда `MVN` берёт значение *`Operand2`*, выполняет над ним побитовую операцию «НЕ» и сохраняет результат в регистре *`Rd`*.

Примечание

Команда `MOVW` выполняет ту же операцию, что и команда `MOV`, однако в качестве операнда может использовать только константу *`imm16`*.

Ограничения

- Нельзя использовать PC.
- *`Operand2`* не должен быть SP.

При использовании PC в качестве *`Rd`* в команде `MOV`:

- бит [0] значения, загружаемого в PC, игнорируется;
- переход производится по адресу, получаемому сбросом бита [0] загружаемого значения в 0.

Примечание

Несмотря на то что команду `MOV` можно использовать в качестве команды перехода, компания ARM настоятельно рекомендует использовать для этих целей команды `BL` и `BLX`, чтобы обеспечить переносимость кода между различными наборами команд ARM.

Флаги условия

Если суффикс `S` указан, то эти команды:

- изменяют флаги N и Z в соответствии с результатом операции;
- могут изменять состояние флага C при вычислении *`Operand2`*; см. подраздел А.2.3 «Гибкий» второй операнд».
- не влияют на состояние флага V.

Примеры

```
MOVS    R11, #0x000B    ; Записывает число 0x000B в R11, флаги изменяются
MOV     R1, #0xFA05     ; Записывает число 0xFA05 в R1, флаги не изменяются
MOVS    R10, R12        ; Копирует содержимое R12 в R10, флаги изменяются
MOV     R3, #23         ; Записывает число 23 в R3
MOV     R8, SP          ; Сохраняет значение указателя стека в R8
MVNS    R2, #0xF        ; Записывает число 0xFFFFFFF0 (инвертированное число 0xF)
                        ; в R2, флаги изменяются
```

A.4.7. MOVT

Пересылка старшего полуслова.

Синтаксис

```
MOVT{cond} Rd, #imm16
```

где

cond — необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».

Rd — регистр-приёмник. Если *Rd* отсутствует, то регистром-приёмником является *Rn*.

Operand2 — «гибкий» второй операнд; для получения дополнительной информации, обратитесь к подразделу A.2.3 «Гибкий» второй операнд».

imm16 — любое число в диапазоне 0…65535.

Операция

Команда MOVT загружает 16-битную константу *imm16* в старшее полуслово (биты [31:16]) регистра-приёмника *Rd*. Состояние битов *Rd* [15:0] при этом не изменяется.

Пара команд MOV и MOVT позволяет загружать в регистры произвольные 32-битные константы.

Ограничения

Rd не должен быть SP или PC.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
MOVT    R3, #0xF123     ; Загружает 0xF123 в старшее полуслово R3, младшее
                        ; полуслово и регистр APSR не изменяются
```

A.4.8. REV, REV16, REVSH и RBIT

Перестановка байтов и битов.

Синтаксис

```
op{cond} Rd, Rn
```

где

op — операция:

- REV — Перестановка байтов слова
- REV16 — Перестановка байтов каждого полуслова в отдельности
- REVSH — Перестановка байтов младшего полуслова и расширение знака до 32 бит
- RBIT — Перестановка битов 32-битного слова в обратном порядке

cond — необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».

Rd регистр-приёмник. Если *Rd* отсутствует, то регистром-приёмником является *Rn*.

Rn регистр, содержащий операнд.

Операция

Эти команды предназначены для изменения порядка байтов при хранении данных:

REV преобразует 32-битное число с обратным порядком байтов (big-endian) в число с прямым порядком байтов (little-endian) или наоборот, число с прямым порядком байтов в число с обратным порядком.

REV16 преобразует 16-битное число с обратным порядком байтов (big-endian) в число с прямым порядком байтов (little-endian) или наоборот, число с прямым порядком байтов в число с обратным порядком.

REVSH преобразует:

16-битное число со знаком с обратным порядком байтов в 32-битное число со знаком с прямым порядком байтов;

16-битное число со знаком с прямым порядком байтов в 32-битное число со знаком с обратным порядком байтов.

Ограничения

Нельзя использовать SP либо PC.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
REV     R3, R7      ; Меняет порядок байтов содержимого R7 и сохраняет
                    ; результат в R3
REV16   R0, R0      ; Меняет порядок байтов каждого полуслова R0
REVSH   R0, R5      ; Меняет порядок байтов 16-битного числа со знаком
REVHS   R3, R7      ; Переставляет байты при условии «выше или равно»
RBIT    R7, R8      ; Изменяет порядок битов содержимого R8 и сохраняет
                    ; результат в R7
```

A.4.9. TST и TEQ

Проверка битов и проверка на равенство.

Синтаксис

```
TST{cond} Rn, Operand2
TEQ{cond} Rn, Operand2
```

где

cond	необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».
Rn	регистр, содержащий первый операнд.
Operand2	«гибкий» второй операнд; для получения дополнительной информации обратитесь к подразделу A.2.3 «Гибкий» второй операнд».

Операция

Эти команды выполняют сравнение содержимого регистра *Rn* со значением *Operand2*. Они изменяют флаги условий в соответствии с результатом сравнения, но не сохраняют результат в регистре.

Команда TST выполняет побитовую операцию «Логическое И» между содержимым *Rn* и значением *Operand2*. То есть эта команда аналогична команде ANDS, за исключением того, что результат операции не сохраняется.

Для проверки состояния отдельного бита *Rn* используют команду TST, второй операнд которой является константой; в этой константе проверяемый бит установлен в 1, а остальные биты сброшены в 0.

Команда TEQ выполняет побитовую операцию «Исключающее ИЛИ» между содержимым *Rn* и значением *Operand2*. То есть эта команда аналогична команде EORS, за исключением того, что результат операции не сохраняется.

Команда TEQ используется для проверки равенства двух значений, не изменяя при этом флагов C или V. Кроме того, команду TEQ можно использовать для проверки знака числа. После сравнения флаг N будет равен результату операции «Исключающее ИЛИ» между знаковыми битами обоих операндов.

Ограничения

Нельзя использовать SP либо PC.

Флаги условия

Команды TST и TEQ:

- изменяют флаги N и Z в соответствии с результатом операции;
- могут изменять состояние флага C при вычислении *Operand2*; см. подраздел A.2.3 «Гибкий» второй операнд»;
- не влияют на состояние флага V.

Примеры

```
TST     R0, #0x3F8  ; Выполняет побитовое «И» между R0 и 0x3F8,
                    ; регистр APSR изменяется, но результат не сохраняется
TEQEQ   R10, R9     ; Проверяет (условно) равенство содержимого R10 и R9,
                    ; регистр APSR изменяется, но результат не сохраняется
```

A.5. Команды умножения и деления

Команды умножения и деления перечислены в **Табл. А.7**.

Таблица А.7. **Команды умножения и деления**

Мнемоника	Краткое описание	Подраздел
`MLA`	Умножение со сложением, 32-битный результат	А.5.1. MUL, MLA и MLS
`MLS`	Умножение с вычитанием, 32-битный результат	А.5.1. MUL, MLA и MLS
`MUL`	Умножение, 32-битный результат	А.5.1. MUL, MLA и MLS
`SDIV`	Знаковое деление	А.5.3. SDIV и UDIV
`SMLAL`	Знаковое умножение со сложением (32 × 32 + 64), 64-битный результат	А.5.2. UMULL, UMLAL, SMULL и SMLAL
`SMULL`	Знаковое умножение (32 × 32), 64-битный результат	А.5.2. UMULL, UMLAL, SMULL и SMLAL
`UDIV`	Беззнаковое деление	А.5.3. SDIV и UDIV
`UMLAL`	Беззнаковое умножение со сложением (32 × 32 + 64), 64-битный результат	А.5.2. UMULL, UMLAL, SMULL и SMLAL
`UMULL`	Беззнаковое умножение (32 × 32), 64-битный результат	А.5.2. UMULL, UMLAL, SMULL и SMLAL

A.5.1. MUL, MLA и MLS

Умножение, умножение со сложением и умножение с вычитанием (32-битные операнды, 32-битный результат).

Синтаксис

```
MUL{S}{cond} Rd, Rn, Rm   ; Умножение
MLA{cond} Rd, Rn, Rm, Ra  ; Умножение со сложением
MLS{cond} Rd, Rn, Rm, Ra  ; Умножение с вычитанием
```

где

`S` — необязательный суффикс. При наличии в мнемонике команды суффикса `S` состояние флагов условий изменяется в соответствии с результатом операции; см. подраздел А.2.7 «Условное выполнение».

`cond` — необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».

`Rd` — регистр-приёмник. Если *`Rd`* отсутствует, то регистром-приёмником является *`Rn`*.

`Rn, Rm` — регистры, содержащие перемножаемые значения.

`Ra` — регистр, содержащий прибавляемое или вычитаемое значение.

Операция

Команда `MUL` перемножает содержимое *`Rn`* и *`Rm`* и помещает 32 младших значащих бита результата в регистр *`Rd`*.

Команда `MLA` перемножает содержимое *`Rn`* и *`Rm`*, прибавляет содержимое *`Ra`* и помещает 32 младших значащих бита результата в регистр *`Rd`*.

Команда `MLS` перемножает содержимое *`Rn`* и *`Rm`*, вычитает произведение из содержимого *`Ra`* и помещает 32 младших значащих бита результата в регистр *`Rd`*.

Результат выполнения этих команд не зависит от того, имели операнды знак или нет.

Ограничения

- Нельзя использовать SP либо PC.

При использовании суффикса `S` с командой `MUL`:

- *`Rd`*, *`Rn`* и *`Rm`* должны быть в диапазоне R0…R7;
- *`Rd`* должен быть тем же регистром, что и *`Rm`*;
- использование суффикса *`cond`* не допускается.

Флаги условия

Если суффикс `S` указан, то команда `MUL`:

- изменяет флаги N и Z в соответствии с результатом операции;
- не влияет на состояние флагов C и V.

Примеры

```
MUL     R10, R2, R5     ; Умножение, R10 = R2 x R5
MLA     R10, R2, R1, R5 ; Умножение со сложением, R10 = (R2 x R1) + R5
MULS    R0, R2, R2      ; Умножение с изменением флагов, R0 = R2 x R2
MULLT   R2, R3, R2      ; Умножение (условное), R2 = R3 x R2
MLS     R4, R5, R6, R7  ; Умножение с вычитанием, R4 = R7 - (R5 x R6)
```

A.5.2. UMULL, UMLAL, SMULL и SMLAL

Знаковое и беззнаковое умножение с опциональным сложением (32-битные операнды, 64-битный результат).

Синтаксис

```
op{cond} RdLo, RdHi, Rn, Rm
```

где

`op` операция:

`UMULL`	Беззнаковое длинное умножение
`UMLAL`	Беззнаковое длинное умножение со сложением
`SMULL`	Знаковое длинное умножение
`SMLAL`	Знаковое длинное умножение со сложением

`cond` необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».

`RdHi, RdLo` регистры-приёмники. Для команд `UMLAL` и `SMLAL` в них же хранится значение, прибавляемое к произведению.

`Rn, Rm` регистры, содержащие перемножаемые значения.

Операция

Команда `UMULL` интерпретирует содержимое регистров *Rn* и *Rm* как целые числа без знака. Команда перемножает эти числа, помещая 32 младших значащих бита результата в регистр *RdLo*, а 32 старших значащих бита результата — в *RdHi*.

Команда `UMLAL` интерпретирует содержимое регистров *Rn* и *Rm* как целые числа без знака. Команда перемножает эти числа, прибавляет к 64-битному произведению 64-битное число, находящееся в регистрах *RdHi* и *RdLo*, и заносит результат обратно в регистры *RdHi* и *RdLo*.

Команда `SMULL` интерпретирует содержимое регистров *Rn* и *Rm* как целые числа со знаком, представленные в дополнительном коде. Команда перемножает эти числа, помещая 32 младших значащих бита результата в регистр *RdLo*, а 32 старших значащих бита результата — в *RdHi*.

Команда `SMLAL` интерпретирует содержимое регистров *Rn* и *Rm* как целые числа со знаком, представленные в дополнительном коде. Команда перемножает эти числа, прибавляет к 64-битному произведению 64-битное число, находящееся в регистрах *RdHi* и *RdLo*, и заносит результат обратно в регистры *RdHi* и *RdLo*.

Ограничения

- Нельзя использовать SP либо PC.
- *RdHi* и *RdLo* должны быть разными регистрами.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
UMULL   R0, R4, R5, R6  ; Без знака (R4, R0) = R5 x R6
SMLAL   R4, R5, R3, R8  ; Со знаком (R5, R4) = (R5, R4) + R3 x R8
```

A.5.3. SDIV и UDIV

Знаковое и беззнаковое деление.

Синтаксис

```
SDIV{cond} {Rd,} Rn, Rm
UDIV{cond} {Rd,} Rn, Rm
```

где

cond	необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».
Rd	регистр-приёмник. Если *Rd* отсутствует, то регистром-приёмником является *Rn*.
Rn	регистр, содержащий делимое.
Rm	регистр, содержащий делитель.

Операция

Команда SDIV выполняет целочисленное деление (со знаком) числа, находящегося в *Rn*, на число, находящееся в *Rm*, и сохраняет результат в регистре *Rd*. Команда UDIV выполняет целочисленное деление (беззнаковое) числа, находящегося в *Rn*, на число, находящееся в *Rm*, и сохраняет результат в регистре *Rd*.

Если число, находящееся в *Rm*, не делится нацело на число, находящееся в *Rd*, то результат округляется к нулю (дробная часть отбрасывается).

Ограничения

Нельзя использовать SP либо PC.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
SDIV    R0, R2, R4      ; Деление чисел со знаком, R0 = R2/R4
UDIV    R8, R8, R1      ; Деление чисел без знака, R8 = R8/R1
```

A.6. Команды насыщения

В этом разделе описываются команды насыщения (фиксации крайних значений) SSAT и USAT.

A.6.1. SSAT и USAT

Знаковое и беззнаковое насыщение с заданной разрядностью и опциональным сдвигом перед насыщением.

Синтаксис

```
op{cond} Rd, #n, Rm{, shift #s}
```

где

op — операция:
- SSAT — Насыщение значения со знаком в пределах знакового диапазона
- USAT — Насыщение беззнакового значения в пределах беззнакового диапазона

cond — необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».

Rd — регистр-приёмник.

n — определяет диапазон насыщения:
- $1 \le n \le 32$ для команды SSAT
- $0 \le n \le 31$ для команды USAT

Rm — регистр, содержащий исходное значение.

shift #s необязательный сдвиг, применяемый к содержимому *Rm* перед насыщением. Допускаются следующие команды сдвига:

ASR #*s*, где *s* должно находиться в диапазоне 1…31

LSL #*s*, где *s* должно находиться в диапазоне 0…31

Операция

Эти команды выполняют насыщение до знакового или беззнакового *n*-битного значения.

Команда SSAT выполняет заданную операцию сдвига, после чего фиксирует полученное значение в пределах знакового диапазона $-2^{n-1} \le x \le 2^{n-1} - 1$. Команда USAT выполняет заданную операцию сдвига, после чего фиксирует полученное значение в пределах беззнакового диапазона $0 \le x \le 2^n - 1$.

В случае команды SSAT это означает:

- Если фиксируемое значение меньше -2^{n-1}, то результат будет равен -2^{n-1}.
- Если фиксируемое значение больше $2^{n-1} - 1$, то результат будет равен $2^{n-1} - 1$.
- В противном случае, результат будет равен фиксируемому значению.

В случае команды USAT это означает:

- Если фиксируемое значение меньше 0, то результат будет равен 0.
- Если фиксируемое значение больше $2^n - 1$, то результат будет равен $2^n - 1$.
- В противном случае, результат будет равен фиксируемому значению.

Если результат операции отличается от фиксируемого значения, значит, возникло *насыщение*. В случае насыщения обе команды устанавливают флаг Q регистра APSR в 1. В противном случае, состояние флага Q не изменяется. Для сброса флага Q нужно использовать команду MSR; см. подраздел A.9.7 «MSR». Для чтения состояния флага Q можно использовать команду MRS; см. подраздел A.9.6 «MRS».

Ограничения

- Нельзя использовать SP либо PC.

Флаги условия

Эти команды не влияют на состояние флагов.

В случае насыщения данные команды устанавливают флаг Q в 1.

Примеры

```
SSAT    R7, #16, R7, LSL #4 ; Выполняет логический сдвиг влево
                            ; на 4 бита содержимого R7,
                            ; после чего фиксирует его в пределах
                            ; 16-битного знакового диапазона
                            ; и заносит результат обратно в R7
USATNE  R0, #7, R5          ; Фиксирует (условно) содержимое R5 в пределах
                            ; 7-битного беззнакового диапазона
                            ; и сохраняет его в R0
```

A.7. Команды работы с битовыми полями

Команды, оперирующие наборами смежных битов регистров (битовыми полями), перечислены в **Табл. A.8**.

***Таблица A.8.* Команды работы с битовыми полями**

Мнемоника	Краткое описание	Подраздел
BFC	Очистка битового поля	A.7.1. BFC и BFI
BFI	Вставка битового поля	A.7.1. BFC и BFI
SBFX	Извлечение битового поля со знаком	A.7.2. SBFX и UBFX
SXTB	Расширение знака байта	A.7.3. SXT и UXT
SXTH	Расширение знака полуслова	A.7.3. SXT и UXT
UBFX	Извлечение беззнакового битового поля	A.7.2. SBFX и UBFX
UXTB	Дополнение нулями байта	A.7.3. SXT и UXT
UXTH	Дополнение нулями полуслова	A.7.3. SXT и UXT

A.7.1. BFC и BFI

Очистка битового поля и вставка битового поля.

Синтаксис

```
BFC{cond} Rd, #lsb, #width
BFI{cond} Rd, Rn, #lsb, #width
```

где

cond — необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».

Rd — регистр-приёмник.

Rn — регистр-источник.

lsb — позиция младшего значащего бита поля. Должна находиться в диапазоне 0…31.

width — размер битового поля. Должен находиться в диапазоне 1…32 – *lsb*.

Операция

Команда BFC очищает битовое поле в регистре. Она сбрасывает *width* битов регистра *Rd*, начиная с младшего бита с номером *lsb*. Остальные биты регистра *Rd* остаются неизменными.

Команда BFI копирует битовое поле из одного регистра в другой. Эта команда замещает *width* битов в регистре *Rd*, начиная с младшего бита с номером *lsb*, содержимым *width* битов регистра *Rm*, начиная с бита [0]. Остальные биты регистра *Rd* остаются неизменными.

Ограничения

Нельзя использовать SP либо PC.

Флаги условия

Эти команды не влияют на состояние флагов.

Примеры

```
BFC     R4, #8, #12     ; Сбрасываем биты с 8-го по 19-й (12 бит) регистра R4
BFI     R9, R2, #8, #12 ; Замещаем биты с 8-го по 19-й (12 бит) регистра R9
                        ; значениями битов с 0-го по 11-й регистра R2
```

A.7.2. SBFX и UBFX

Извлечение битового поля со знаком и извлечение битового поля без знака.

Синтаксис

```
SBFX{cond} Rd, Rn, #lsb, #width
UBFX{cond} Rd, Rn, #lsb, #width
```

где

cond	необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».
Rd	регистр-приёмник.
Rn	регистр-источник.
lsb	позиция младшего значащего бита поля. Должна находиться в диапазоне 0…31.
width	размер битового поля. Должен находиться в диапазоне 1…32–*lsb*.

Операция

Команда SBFX извлекает битовое поле из регистра *Rn*, расширяет его до 32-битного значения и сохраняет результат в регистре *Rd*.

Команда UBFX извлекает битовое поле из регистра *Rn*, дополняет его нулями до 32-битного значения и сохраняет результат в регистре *Rd*.

Ограничения

Нельзя использовать SP либо PC.

Флаги условия

Эти команды не влияют на состояние флагов.

Примеры

```
SBFX    R0, R1, #20, #4  ; Берём биты с 20-го по 23-й (4 бита) из R1, расширяем
                         ; до 32 бит и сохраняем результат в R0
UBFX    R8, R11, #9, #10 ; Берём биты с 9-го по 18-й (10 бит) из R11, дополняем
                         ; нулями до 32 бит и сохраняем результат в R8
```

A.7.3. SXT и UXT

Расширение знака и дополнение нулями.

Синтаксис

```
SXT extend{cond} Rd, Rm {, ROR #n}
UXT extend{cond} Rd, Rm {, ROR #n}
```

где

extend — разрядность операнда:
- `B` — преобразование 8-битного значения в 32-битное
- `H` — преобразование 16-битного значения в 32-битное

cond — необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».

Rd — регистр-приёмник.

Rm — регистр, содержащий преобразуемое значение.

ROR #n — предварительный сдвиг операнда, допустимы следующие мнемоники:
- `ROR #8` — значение из *Rm* циклически сдвигается на 8 бит вправо
- `ROR #16` — значение из *Rm* циклически сдвигается на 16 бит вправо
- `ROR #24` — значение из *Rm* циклически сдвигается на 24 бит вправо

Если *ROR #n* не указано, то сдвиг операнда не производится.

Операция

Эти команды выполняют следующие операции:

1. Циклически сдвигают содержимое *Rm* на 0, 8, 16 или 24 бит вправо.
2. Извлекают биты из полученного значения:
   а) `SXTB` извлекает биты [7:0] и расширяет знак до 32 бит;
   б) `UXTB` извлекает биты [7:0] и дополняет нулями до 32 бит;
   в) `SXTH` извлекает биты [15:0] и расширяет знак до 32 бит;
   г) `UXTH` извлекает биты [15:0] и дополняет нулями до 32 бит.

Ограничения

Нельзя использовать SP либо PC.

Флаги условия

Эти команды не влияют на состояние флагов.

Примеры

```
SXTH    R4, R6, ROR #16 ; Циклически сдвигает R6 вправо на 16 бит, затем
                        ; берёт младшее полуслово, расширяет его знак
                        ; до 32 бит и сохраняет результат в R4
UXTB    R3, R10         ; Берёт младший байт содержимого R10, дополняет нулями
                        ; до 32 бит и сохраняет результат в R3
```

A.8. Команды ветвления и управления

Команды, оперирующие наборами смежных битов регистров (битовыми полями), перечислены в **Табл. А.9**.

Таблица А.9. **Команды ветвления**

Мнемоника	Краткое описание	Подраздел
B	Переход	A.8.1. B, BL, BX и BLX
BL	Переход со ссылкой	A.8.1. B, BL, BX и BLX
BLX	Косвенный переход со ссылкой	A.8.1. B, BL, BX и BLX
BX	Косвенный переход	A.8.1. B, BL, BX и BLX
CBNZ	Сравнение и переход, если не ноль	A.8.2. CBZ и CBNZ
CBZ	Сравнение и переход, если ноль	A.8.2. CBZ и CBNZ
IT	Блок IF-THEN	A.8.3. IT
TBB	Табличный переход с однобайтными смещениями	A.8.4. TBB и TBH
TBH	Табличный переход с двухбайтными смещениями	A.8.4. TBB и TBH

A.8.1. B, BL, BX и BLX

Команды перехода.

Синтаксис

```
B{cond} label
BL{cond} label
BX{cond} Rm
BLX{cond} Rm
```

где

B	переход (непосредственное смещение).
BL	переход со ссылкой (непосредственное смещение).
BX	косвенный переход (смещение в регистре).
BLX	косвенный переход со ссылкой (смещение в регистре).
cond	необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».
label	выражение, определяемое относительно PC; см. подраздел А.2.6 «Адресация относительно PC».
Rm	регистр, содержащий адрес перехода. Бит [0] регистра *Rm* должен быть установлен в 1, однако адрес перехода получается сбросом этого бита в 0.

Операция

Все указанные команды выполняют переход к метке *label* или по адресу, находящемуся в регистре *Rm*. Кроме того:

- команды BL и BLX заносят адрес следующей команды в регистр LR (регистр связи, =R14);
- команды BX и BLX генерируют исключение Usage Fault, если бит [0] регистра *Rm* равен 0.

Команда B{cond} label является единственной условно выполняемой командой, которая может располагаться как внутри, так и снаружи IT-блока. Все остальные команды перехода при условном выполнении должны располагаться внутри IT-блока, а при безусловном выполнении — снаружи IT-блока. Для получения дополнительной информации обратитесь к подразделу A.8.3 «IT».

Допустимые диапазоны для различных команд перехода указаны в **Табл. A.10**.

Таблица A.10. **Диапазоны переходов**

Команда	Диапазон переходов
B *label*	–16…+16 Мбайт
B*cond label* (вне IT-блока)	–1…+1 Мбайт
B*cond label* (внутри IT-блока)	–16…+16 Мбайт
BL{*cond*} *label*	–16…+16 Мбайт
BX{*cond*} *Rm*	Любое значение в регистре
BLX{*cond*} *Rm*	Любое значение в регистре

Примечание

Для использования всего диапазона переходов может потребоваться использование суффикса .W (см. подраздел A.2.8 «Выбор разрядности команды»).

Ограничения

- Нельзя использовать PC в команде BLX.
- Для команд BX и BLX бит [0] регистра *Rm* должен быть установлен в 1, однако переход будет осуществляться по адресу, полученному сбросом этого бита в 0.
- Если любая из этих команд используется внутри IT-блока, то она должна быть последней командой в блоке.

Примечание

Команда B*cond* — это единственная условно выполняемая команда, которая может вызываться вне IT-блока. Тем не менее, внутри IT-блока она имеет больший диапазон переходов.

Флаги условия

Эти команды не влияют на состояние флагов.

Примеры

```
B       loopA   ; Переход к метке loopA
BLE     ng      ; Условный переход к метке ng
```

```
B.W     target  ; Переход к метке target в пределах 16 Мбайт
BEQ     target  ; Условный переход к метке target
BEQ.W   target  ; Условный переход к метке target в пределах 1 Мбайт
BL      funC    ; Переход со ссылкой (вызов) к функции funC, адрес возврата
                ; запоминается в LR
BX      LR      ; Возврат из вызванной функции
BXNE    R0      ; Условный переход по адресу, находящемуся в R0
BLX     R0      ; Переход со ссылкой (вызов) по адресу, находящемуся в R0
```

A.8.2. CBZ и CBNZ

Сравнение и переход, если ноль/если не ноль.

Синтаксис

`CBZ` *`Rn, label`*

`CBNZ` *`Rn, label`*

где

`Rm` регистр, содержащий операнд.

`label` метка перехода.

Операция

Команды `CBZ` и `CBNZ` используются для того, чтобы избежать изменения флагов условия, а также для уменьшения объёма кода.

Команда `CBZ Rn, label` не изменяет флагов условия, а в остальном эквивалентна группе команд:

```
CMP Rn, #0
BEQ label
```

Команда `CBNZ Rn, label` не изменяет флагов условия, а в остальном эквивалентна группе команд:

```
CMP Rn, #0
BNE label
```

Ограничения

- *Rn* должен быть в диапазоне R0…R7.
- Адрес назначения должен находиться в диапазоне от 4 до 130 байт после команды.
- Эти команды не должны использоваться внутри `IT`-блока.

Флаги условия

Эти команды не влияют на состояние флагов.

Примеры

```
CBZ     R5, target  ; Переход вперёд, если R5 равно 0
CBNZ    R0, target  ; Переход вперёд, если R0 не равно 0
```

A.8.3. IT

Условный блок IF–THEN.

Синтаксис

```
IT{x{y{z}}} cond
```

где

x условие выполнения 2-й команды блока.

y условие выполнения 3-й команды блока.

z условие выполнения 4-й команды блока.

cond условие выполнения 1-й команды блока.

Для задания условия выполнения 2-й, 3-й и 4-й команды IT-блока может использоваться одна из следующих мнемоник:

T «то», применяет к команде условие *cond*.

E «иначе», применяет к команде условие, обратное *cond*.

Примечание

В качестве условия *cond* допускается использование мнемоники AL (условие «всегда»). В этом случае все команды в IT-блоке должны быть безусловно выполняемыми, а в качестве каждого из условий *x*, *y* и *z* должны быть указаны мнемоники T (или не указано ничего), но не E.

Операция

Команда IT переводит до 4 последующих команд в режим условного выполнения. Условия выполнения могут быть одинаковыми для всех команд или же некоторые из условий могут быть логической инверсией других. Условно выполняемые команды, расположенные после команды IT, образуют *IT-блок*.

Во всех командах IT-блока, включая любые команды перехода, должен быть указан суффикс условия выполнения {*cond*}.

Примечание

Используемый вами ассемблер может автоматически вставлять требуемые команды IT при обнаружении в коде условно выполняемых команд. Это можно уточнить, обратившись к документации на ассемблер.

Команда BKPT в IT-блоке выполняется всегда, независимо от истинности условия её выполнения.

Исключения могут возникать как между командой IT и соответствующим IT-блоком, так и внутри самого блока. В этом случае процессор переходит к соответствующему обработчику исключения, сохраняя информацию, необходимую для возврата, в регистре связи и регистре PSR, который затем помещается в стек.

Для возврата из обработчика исключения можно использовать любую предназначенную для этого команду, в таком случае выполнение команд IT-блока будет корректно возобновлено. Это единственная ситуация, при которой команде,

модифицирующей содержимое PC, разрешается переход на команду внутри IT-блока.

Ограничения

Следующие команды не могут присутствовать в IT-блоке:

- IT;
- CBZ и CBNZ;
- CPSID и CPSIE.

Прочие ограничения, касающиеся использования IT-блоков:

- Команды перехода и прочие команды, изменяющие PC, должны либо располагаться вне IT-блока, либо быть последней командой блока. Речь идёт о следующих командах:
  - ADD PC, PC, Rm;
  - MOV PC, Rm;
  - B, BL, BX, BLX;
  - любые команды LDM, LDR или POP, осуществляющие запись в PC;
  - TBB и TBH.
- Не допускается переход внутрь IT-блока, кроме как при возврате из обработчика исключения.
- Все условно выполняемые команды, за исключением команды B cond, должны находиться внутри IT-блока. Команда B cond может использоваться как вне IT-блока, так и внутри него, однако в последнем случае она имеет больший диапазон переходов.
- У любой команды IT-блока должен быть указан суффикс условия выполнения, который может либо совпадать, либо быть логической инверсией условия выполнения остальных команд блока.

Примечание

Используемый вами ассемблер может накладывать дополнительные ограничения на использование IT-блока, скажем, запрещать размещение внутри него директив ассемблера.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
ITTE    NE          ; Следующие 3 команды - условно выполняемые
ANDNE   R0, R0, R1  ; ANDNE не изменяет флаги условия
ADDSNE  R2, R2, #1  ; ADDSNE изменяет флаги условия
MOVEQ   R2, R3      ; Условное копирование
CMP     R0, #9      ; Преобразуем шестнадцатеричное число (0-15) в регистре R0
                    ; в ASCII-символ ('0'-'9', 'A'-'F')
ITE     GT          ; Следующие 2 команды - условно выполняемые
ADDGT   R1, R0, #55 ; Преобразуем 0xA -> 'A'
ADDLE   R1, R0, #48 ; Преобразуем 0x0 -> '0'
IT      GT          ; IT-блок с единственной условно выполняемой командой
```

```
ADDGT   R1, R1, #1  ; Условно инкрементируем R1
ITTEE   EQ          ; Следующие 4 команды – условно выполняемые
MOVEQ   R0, R1      ; Условное копирование
ADDEQ   R2, R2, #10 ; Условное сложение
ANDNE   R3, R3, #1  ; Условное «Логическое И»
BNE.W   dloop       ; Команда перехода может быть только последней командой
                    ; IT-блока
IT      NE          ; Следующая команда – условно выполняемая
ADD     R0, R0, R1  ; Синтаксическая ошибка: не указан код условия выполнения
```

A.8.4. TBB и TBH

Табличный переход с однобайтными/двухбайтными смещениями.

Синтаксис

```
TBB [Rn, Rm]
TBH [Rn, Rm, LSL #1]
```

где

Rn — регистр, содержащий адрес таблицы переходов. Если в качестве *Rn* используется PC, то адресом таблицы является адрес первого байта после команды TBB или TBH.

Rm — индексный регистр. Этот регистр содержит индекс элемента таблицы. В таблицах с двухбайтными смещениями корректное значение смещения получается удвоением содержимого *Rm* (операция LSL #1).

Операция

Эти команды выполняют переход в прямом направлении, используя таблицу однобайтных (TBB) или двухбайтных (TBH) смещений относительно PC. Регистр *Rn* является указателем на таблицу, а *Rm* — индексом элемента таблицы. В команде TBB величина смещения получается удвоением беззнакового однобайтного значения, извлечённого из таблицы. Аналогично, в команде TBH величина смещения получается удвоением беззнакового 2-байтного значения, извлечённого из таблицы. Переход осуществляется по адресу, отстоящему на полученную величину от адреса байта, следующего за командой TBB или TBH.

Ограничения

- *Rn* не должен быть SP.
- *Rm* не должен быть SP либо PC.
- Если любая из этих команд используется внутри IT-блока, то она должна быть последней командой в блоке.

Флаги условия

Эти команды не влияют на состояние флагов.

Примеры

```
    ADR.W   R0, BranchTable_Byte
    TBB     [R0, R1]                ; R1 - индекс, R0 – базовый адрес
                                    ; таблицы переходов
Case1
                                    ; Далее располагаются соответствующие команды
Case2
                                    ; Далее располагаются соответствующие команды
Case3
                                    ; Далее располагаются соответствующие команды
BranchTable_Byte
    DCB 0                           ; Вычисление смещения для варианта Case1
    DCB ((Case2-Case1)/2)           ; Вычисление смещения для варианта Case2
    DCB ((Case3-Case1)/2)           ; Вычисление смещения для варианта Case3

    TBH [PC, R1, LSL #1]            ; R1 - индекс, PC используется в качестве
                                    ; базового адреса таблицы переходов
BranchTable_H
    DCI ((CaseA - BranchTable_H)/2) ; Вычисление смещения для варианта CaseA
    DCI ((CaseB - BranchTable_H)/2) ; Вычисление смещения для варианта CaseB
    DCI ((CaseC - BranchTable_H)/2) ; Вычисление смещения для варианта CaseC
CaseA
                                    ; Далее располагаются соответствующие команды
CaseB
                                    ; Далее располагаются соответствующие команды
CaseC
                                    ; Далее располагаются соответствующие команды
```

A.9. Прочие команды

Все остальные команды, поддерживаемые процессором Cortex-M3, перечислены в **Табл. A.11**.

Таблица A.11. **Прочие команды**

Мнемоника	Краткое описание	Подраздел
BKPT	Точка останова	A.9.1. BKPT
CPSID	Изменение состояния процессора, запрещение прерываний	A.9.2. CPS
CPSIE	Изменение состояния процессора, разрешение прерываний	A.9.2. CPS
DMB	Барьер памяти данных	A.9.3. DMB
DSB	Барьер синхронизации данных	A.9.4. DSB
ISB	Барьер синхронизации команд	A.9.5. ISB
MRS	Пересылка из регистра специальных функций в регистр	A.9.6. MRS
MSR	Пересылка из регистра в регистр специальных функций	A.9.7. MSR
NOP	Нет операции	A.9.8. NOP
SEV	Послать событие	A.9.9. SEV
SVC	Вызов супервизора	A.9.10. SVC
WFE	Ожидание события	A.9.11. WFE
WFI	Ожидание прерывания	A.9.12. WFI

A.9.1. BKPT

Точка останова.

Синтаксис

```
BKPT #imm
```

где

imm выражение, результатом которого является целое число от 0 до 255 (8-битное значение).

Операция

Команда BKPT переводит процессор в режим отладки. Отладочные средства могут использовать это для считывания информации о состоянии системы при достижении команды, расположенной по заданному адресу.

Значение *imm* процессором игнорируется. При необходимости отладчик может использовать это значение для сохранения дополнительной информации о точке останова. Компания ARM не рекомендует использовать с командой BKPT значение 0xAB для любых целей, за исключением поддержки полухостинга. Команда BKPT может располагаться внутри IT-блока, однако выполняться она будет всегда, независимо от условия, заданного в команде IT.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
BKPT    #0x3    ; Точка останова с параметром, равным 0x3 (отладчик может
                ; извлечь эту константу, определив её положение по значению PC)
```

Примечание

Компания ARM не рекомендует использовать с командой BKPT значение 0xAB для любых целей, за исключением поддержки полухостинга.

A.9.2. CPS

Изменение состояния процессора.

Синтаксис

```
CPSeffect iflags
```

где

effect действие, производимое над регистром специального назначения:

- IE — очистка регистра
- ID — установка регистра

iflags один или несколько флагов:

- i — установка или очистка регистра PRIMASK
- f — установка или очистка регистра FAULTMASK

Операция

Команда CPS изменяет содержимое регистров PRIMASK и FAULTMASK.

Ограничения

- Команда CPS должна вызываться только в привилегированном режиме. В непривилегированном режиме она не выполняет никаких действий.
- Команда CPS не может выполняться условно и, соответственно, не должна использоваться внутри IT-блока.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
CPSID   i  ; Запрещает прерывания и конфигурируемые обработчики отказов
           ; (установка PRIMASK)
CPSID   f  ; Запрещает прерывания и все обработчики отказов (установка FAULTMASK)
CPSIE   i  ; Разрешает прерывания и конфигурируемые обработчики отказов
           ; (очистка PRIMASK)
CPSIE   f  ; Разрешает прерывания и все обработчики отказов (очистка FAULTMASK)
```

A.9.3. DMB

Барьер памяти данных.

Синтаксис

```
DMB{cond}
```

где

cond — необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».

Операция

Команда DMB используется в качестве барьера памяти данных. Она гарантирует, что все обращения к памяти, явно указанные до вызова команды DMB, будут выполнены до того, как начнут выполняться явные обращения к памяти, появляющиеся после команды DMB. Команда DMB не влияет на порядок или выполнение команд, не обращающихся к памяти.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
DMB     ; Барьер памяти данных
```

A.9.4. DSB

Барьер синхронизации данных.

Синтаксис

```
DSB{cond}
```

где

cond необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».

Операция

Команда `DSB` используется в качестве специального барьера для синхронизации памяти данных. Команды, располагающиеся после `DSB`, не будут выполняться до завершения команды `DSB`. Команда `DSB` завершается после того, как будут выполнены все явные обращения к памяти, запущенные перед командой.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
DSB     ; Барьер синхронизации данных
```

A.9.5. ISB

Барьер синхронизации команд.

Синтаксис

```
ISB{cond}
```

где

cond необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».

Операция

Команда `ISB` используется в качестве барьера для синхронизации команд. Она очищает конвейер процессора, в результате чего все команды, следующие за `ISB`, повторно извлекаются из кэша или памяти после завершения команды `ISB`.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
ISB     ; Барьер синхронизации команд
```

A.9.6. MRS

Пересылает содержимое регистра специального назначения в регистр общего назначения.

Синтаксис

```
MRS{cond} Rd, spec_reg
```

где

cond — необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».

Rd — регистр-приёмник.

spec_reg — любой из следующих регистров: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI_MAX, FAULTMASK или CONTROL.

Операция

Команда MRS используется совместно с командой MSR для выполнения операции вида «чтение—модификация—запись» для изменения содержимого регистров xPSR, скажем для сброса флага Q.

При переключении между двумя процессами необходимо сохранять состояние заменяемого процесса, включая содержимое соответствующих регистров xPSR. Аналогично, необходимо восстанавливать состояние запускаемого процесса. Для выполнения этих операций используют команды MRS (сохранение состояния) и MSR (восстановление состояния).

Примечание

Мнемоника BASEPRI_MAX эквивалентна мнемонике BASEPRI при использовании с командой MRS; см. подраздел А.9.7 «MSR».

Ограничения

Rd не может быть SP либо PC.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
MRS  R0, PRIMASK   ; Сохраняет содержимое PRIMASK в R0
```

A.9.7. MSR

Пересылает содержимое регистра общего назначения в регистр специального назначения.

Синтаксис

```
MSR{cond} spec_reg, Rd
```

где

cond — необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».

Rn — регистр-источник.

spec _ reg любой из следующих регистров: APSR, IPSR, EPSR, IEPSR, IAPSR, EAPSR, PSR, MSP, PSP, PRIMASK, BASEPRI, BASEPRI _ MAX, FAULTMASK или CONTROL.

Операция

Доступность регистров специального назначения определяется режимом работы процессора. В непривилегированном режиме доступен только регистр APSR. В привилегированном режиме доступны все регистры специального назначения.

В непривилегированном режиме запись в нереализованные биты или биты, содержащие состояние выполнения программы, регистра PSR игнорируются.

Примечание

При указании мнемоники BASEPRI _ MAX команда выполняет запись в регистр BASEPRI только в случае, если:

- *Rn* содержит ненулевое значение и текущее значение BASEPRI равно 0;
- *Rn* содержит ненулевое значение, меньшее текущего значения BASEPRI.

См. подраздел A.9.6 «MRS».

Ограничения

Rd не может быть SP либо PC.

Флаги условия

Эта команда изменяет состояние флагов, явно указанных в *Rn*.

Примеры

```
MRS  CONTROL, R1   ; Загружает содержимое R1 в регистр CONTROL
```

A.9.8. NOP

Нет операции.

Синтаксис

```
NOP{cond}
```

где

cond необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».

Операция

Команда NOP не выполняет никаких действий. Команда NOP не всегда требует времени для выполнения — процессор может убрать её из конвейера раньше, чем она достигнет стадии исполнения.

Используйте команду NOP для заполнения памяти программ, например для размещения последующей команды по 64-битной границе.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
NOP   ; Нет операции
```

A.9.9. SEV

Генерация события.

Синтаксис

```
SEV{cond}
```

где

cond — необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».

Операция

Команда `SEV` генерирует событие для передачи его всем процессорам многопроцессорной системы. Она также устанавливает в 1 локальную защёлку события. Более подробно использование команды `SEV` описано в разделе 14.3 «Многопроцессорный обмен».

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
SEV  ; Послать событие
```

A.9.10. SVC

Вызов супервизора.

Синтаксис

```
SVC{cond} #imm
```

где

cond — необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».

imm — выражение, результатом которого является целое число от 0 до 255 (8-битное значение).

Операция

Команда `SVC` генерирует исключение SVCall.

Значение *imm* процессором игнорируется. При необходимости, обработчик исключения может использовать это значение для определения запрашиваемой службы.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
SVC #0x32  ; Вызов супервизора (обработчик SVC может считать значение,
           ; переданное в команде, определив адрес команды по сохранённому
           ; в стеке значению PC)
```

A.9.11. WFE

Ожидание события.

Синтаксис

```
WFE{cond}
```

где

cond необязательный суффикс условия выполнения; см. подраздел A.2.7 «Условное выполнение».

Операция

Если локальная защёлка события сброшена, то команда `WFE` приостанавливает работу процессора до наступления любого из указанных событий:

- возникновение исключения, не маскированного регистром маскирования исключений или текущим уровнем приоритета;
- перевод исключения в состояние ожидания, если установлен бит `SEVONPEND` регистра SCR;
- запрос на перевод в режим отладки, если она разрешена.

Событие, сгенерированное периферией или другим процессором многопроцессорной системы с использованием команды `SEV`.

Если локальная защёлка события установлена в 1, то команда `WFE` очищает её и завершается. Для получения информации об использовании команды `WFE` и регистра события обратитесь к подразделу 14.2.1 «Спящие режимы».

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
WFE   ; Ожидание события
```

A.9.12. WFI

Ожидание прерывания.

Синтаксис

```
WFI{cond}
```

где

cond необязательный суффикс условия выполнения; см. подраздел А.2.7 «Условное выполнение».

Операция

Команда `WFI` приостанавливает работу процессора до наступления любого из указанных событий:

- исключительной ситуации;
- запроса на переход в режим отладки, независимо от того, разрешена она или нет.

Флаги условия

Эта команда не влияет на состояние флагов.

Примеры

```
WFI   ; Ожидание прерывания
```

ПРИЛОЖЕНИЕ Б
16-БИТНЫЕ КОМАНДЫ THUMB И ВЕРСИИ АРХИТЕКТУРЫ ARM

Поддержка большинства 16-битных команд Thumb® реализована в архитектуре v4T (ARM7TDMI). Однако некоторые команды появились только в последующих версиях (v5, v6 и v7) архитектуры. Эти команды указаны в **Табл. Б.1**.

Таблица Б.1. **Поддержка 16-битных команд в различных версиях архитектуры ARM**

Команда	v4T	v5	v6	Cortex™-M3 (v7-M)
BKPT	Нет	Да	Да	Да
BLX	Нет	Да	Да	Только BLX <reg>
CBZ, CBNZ	Нет	Нет	Нет	Да
CPS	Нет	Нет	Да	CPSIE <i/f>, CPSID <i/f>
CPY	Нет	Нет	Да	Да
NOP	Нет	Нет	Нет	Да
IT	Нет	Нет	Нет	Да
REV (различные формы)	Нет	Нет	Да	REV, REV16, REVSH
SEV	Нет	Нет	Нет	Да
SETEND	Нет	Нет	Да	Нет
SWI	Да	Да	Да	Изменена на SVC
SXTB, SXTH	Нет	Нет	Да	Да
UXTB, UXTH	Нет	Нет	Да	Да
WFE, WFI	Нет	Нет	Нет	Да

ПРИЛОЖЕНИЕ B
ИСКЛЮЧЕНИЯ ПРОЦЕССОРА CORTEX-M3
КРАТКАЯ СПРАВКА

Исключения, поддерживаемые процессором Cortex-M3, перечислены в **Табл. B.1**. В этой же таблице указаны регистры, используемые для задания уровня приоритета прерываний и для разрешения данных прерываний.

B.1. Типы исключений и их разрешение

***Таблица B.1.* Исключения процессора Cortex-M3**

Тип исключения	Наименование	Приоритет (адрес регистра уровня приоритета)	Разрешение
1	Reset	–3	Всегда разрешено
2	NMI	–2	Всегда разрешено
3	Hard Fault	–1	Всегда разрешено
4	MemManage Fault	Программируемый (0xE000ED18)	Регистр SHCSR модуля NVIC (0xE000ED24), бит [16]
5	Bus Fault	Программируемый (0xE000ED19)	Регистр SHCSR модуля NVIC (0xE000ED24), бит [17]
6	Usage Fault	Программируемый (0xE000ED1A)	Регистр SHCSR модуля NVIC (0xE000ED24), бит [18]
7…10	—	—	—
11	SVCall	Программируемый (0xE000ED1F)	Всегда разрешено
12	Debug monitor	Программируемый (0xE000ED20)	Регистр DEMCR модуля NVIC (0xE000EDFC), бит [16]
13	—	—	—
14	PendSV	Программируемый (0xE000ED22)	Всегда разрешено
15	SYSTICK	Программируемый (0xE000ED23)	Регистр SYST_CSR модуля SYSTICK (0xE000E010), бит [1]
16…255	IRQ	Программируемый (0xE000E400…0xE000E7EC)	Регистры NVIC_ISERx модуля NVIC (0xE000E100…0xE000E17C)

B.2. Содержимое стека после сохранения в нём контекста

Таблица B.2. **Стековый фрейм исключения**

Адрес	Данные	Порядок загрузки в стек
OLD_SP (N+32) →	(Ранее загруженные данные)	—
(N+28)	PSR	2
(N+24)	PC	1
(N+20)	LR	8
(N+16)	R12	7
(N+12)	R3	6
(N+8)	R2	5
(N+4)	R1	4
Новый SP (N) →	R0	3

Примечание. Если используется выравнивание стека по границе двойного слова, а указатель стека на момент возникновения исключения не был выровнен, то вершина стекового фрейма может располагаться по адресу ([OLD_SP–4] AND 0xFFFFFFF8) с соответствующим сдвигом остального содержимого таблицы на одно слово вниз.

ПРИЛОЖЕНИЕ Г
РЕГИСТРЫ КОНТРОЛЛЕРА NVIC И БЛОКА УПРАВЛЕНИЯ СИСТЕМОЙ
КРАТКАЯ СПРАВКА

В следующих таблицах (**Табл. Г.1**…**Г.40**) содержится описание битов различных регистров контроллера векторных прерываний NVIC, включая регистры, связанные с системным таймером SYSTICK.

Таблица Г.1. **Регистр типа контроллера прерываний ICTR (0xE000E004)**

Биты	Обозначение	Тип	Значение после сброса	Описание
4:0	INTLINESNUM	R	—	Количество входов прерываний с шагом 32: 0 = 1…32; 1 = 33…64; …

Таблица Г.2. **Дополнительный регистр управления ACR (0xE000E008)**

Биты	Обозначение	Тип	Значение после сброса	Описание
2	DISFOLD	R/W	0	Если этот бит установлен, запрещается перекрытие цикла исполнения `IT`-блока другой командой. Такое перекрытие (называемое *IT-folding*) является одним из методов оптимизации, позволяющим ускорить выполнение условно выполняемых команд
1	DISDEFWBUF	R/W	0	Если этот бит установлен, запрещается использование внутренних буферов записи процессора. В результате команда, расположенная за командой сохранения данных, не будет запущена до завершения операции сохранения. Этот бит не влияет на буферы записи, расположенные вне процессора (например, в мосте между шинами)
0	DISMCYCINT	R/W	0	Если этот бит установлен, запрещается прерывание команд, выполняющихся за несколько тактов

Примечание. Регистр ACR появился во 2-й ревизии процессора Cortex-M3.

Таблица Г.3. **Регистр управления и состояния системного таймера SYST_CSR (0xE000E010)**

Биты	Обозначение	Тип	Значение после сброса	Описание
16	COUNTFLAG	R	0	Читается как 1, если с момента последнего чтения данного регистра таймер досчитал до 0. Этот бит сбрасывается в 0 автоматически после чтения или при обнулении текущего значения таймера
2	CLKSOURCE	R/W	0	0 — внешний тактовый сигнал (STCLK); 1 — тактовый сигнал процессора
1	TICKINT	R/W	0	1 — разрешить генерацию прерывания SYSTICK при достижении таймером нулевого значения; 0 — не генерировать прерывание
0	ENABLE	R/W	0	Разрешение таймера SYSTICK

Таблица Г.4. **Регистр значения перезагрузки системного таймера SYST_RVR (0xE000E014)**

Биты	Обозначение	Тип	Значение после сброса	Описание
23:0	RELOAD	R/W	0	Значение, загружаемое в таймер при достижении им нулевого значения

Таблица Г.5. **Регистр текущего значения системного таймера SYST_CVR (0xE000E018)**

Биты	Обозначение	Тип	Значение после сброса	Описание
23:0	CURRENT	R/Wc	0	При чтении возвращается текущее значение счётчика. При записи в регистр счётчик обнуляется; очистка счётчика также сбрасывает флаг COUNTFLAG регистра SYST_CSR (см. **Табл. Г.3**)

Таблица Г.6. **Регистр калибровочного значения системного таймера SYST_CALIB (0xE000E01C)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31	NOREF	R	—	1 — нет внешнего опорного сигнала (сигнал STCLK недоступен); 0 — внешний тактовый сигнал доступен
30	SKEW	R	—	1 — калибровочное значение не соответствует 10-мс интервалу; 0 — калибровочное значение точное
23:0	TENMS	R/W	0	Калибровочное значение для периода 10 мс. Разработчик SoC должен подать это значение (в двоичном коде) на соответствующие входы процессора. Если при чтении данного поля возвращается 0, значит, калибровочное значение недоступно

Таблица Г.7. **Регистры разрешения внешних прерываний NVIC_ISERx (0xE000E100…0xE000E11C)**

Адрес	Обозначение	Тип	Значение после сброса	Описание
0xE000E100	NVIC_ISER0	R/W	0	Разрешает внешние прерывания с номерами от 0 до 31: бит [0] — прерывание №0; бит [1] — прерывание №1; … бит [31] — прерывание №31
0xE000E104	NVIC_ISER1	R/W	0	Разрешает внешние прерывания с номерами от 32 до 63
…	…	…	…	…

Таблица Г.8. **Регистры отмены разрешения внешних прерываний NVIC_ICERx (0xE000E180…0xE000E19C)**

Адрес	Обозначение	Тип	Значение после сброса	Описание
0xE000E180	NVIC_ICER0	R/W	0	Отменяет разрешение внешних прерываний с номерами от 0 до 31: бит [0] — прерывание №0; бит [1] — прерывание №1; … бит [31] — прерывание №31
0xE000E184	NVIC_ICER1	R/W	0	Отменяет разрешение внешних прерываний с номерами от 32 до 63
…	…	…	…	…

Таблица Г.9. **Регистры установки признака отложенного прерывания NVIC_ISPRx (0xE000E200…0xE000E21C)**

Адрес	Обозначение	Тип	Значение после сброса	Описание
0xE000E200	NVIC_ISPR0	R/W	0	Устанавливает признак отложенного прерывания для внешних прерываний с номерами от 0 до 31: бит [0] — прерывание №0; бит [1] — прерывание №1; … бит [31] — прерывание №31
0xE000E204	NVIC_ISPR1	R/W	0	Устанавливает признак отложенного прерывания для внешних прерываний с номерами от 32 до 63
…	…	…	…	…

Таблица Г.10. **Регистры сброса признака отложенного прерывания NVIC_ICPRx (0xE000E280…0xE000E29C)**

Адрес	Обозначение	Тип	Значение после сброса	Описание
0xE000E280	NVIC_ICPR0	R/W	0	Сбрасывает признак отложенного прерывания для внешних прерываний с номерами от 0 до 31: бит [0] — прерывание №0; бит [1] — прерывание №1; … бит [31] — прерывание №31
0xE000E284	NVIC_ICPR1	R/W	0	Сбрасывает признак отложенного прерывания для внешних прерываний с номерами от 32 до 63
…	…	…	…	…

Таблица Г.11. **Регистры активности внешних прерываний NVIC_IABRx (0xE000E300…0xE000E31C)**

Адрес	Обозначение	Тип	Значение после сброса	Описание
0xE000E200	NVIC_IABR0	R	0	Признак активности внешних прерываний с номерами от 0 до 31: бит [0] — прерывание №0; бит [1] — прерывание №1; … бит [31] — прерывание №31
0xE000E204	NVIC_IABR1	R	0	Признак активности внешних прерываний с номерами от 32 до 63
…	…	…	…	…

Таблица Г.12. **Регистры уровня приоритета внешних прерываний PRI_x (0xE000E400…0xE000E4EF; адресация в таблице — побайтовая)**

Адрес	Обозначение	Тип	Значение после сброса	Описание
0xE000E400	PRI_0	R/W	0	Уровень приоритета внешнего прерывания №0
0xE000E401	PRI_1	R/W	0	Уровень приоритета внешнего прерывания №1
…	…	…	…	…
0xE000E41F	PRI_31	R/W	0	Уровень приоритета внешнего прерывания №31
…	…	…	…	…

Таблица Г.13. **Регистр идентификатора ЦПУ CPUID (0xE000ED00)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:24	IMPLEMENTER	R	0x41	Код производителя; для компании ARM = 0x41
23:20	VARIANT	R	0x0/0x1/0x2	Номер версии, определяемый реализацией
19:16	Constant	R	0xF	Константа
15:4	PARTNO	R	0xC23	Номер изделия
3:0	REVISION	R	0x0/0x1	Ревизия

Таблица Г.14. **Регистр управления и состояния прерываний ICSR (0xE000ED04)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31	NMIPENDSET	R/W	0	Немаскируемое прерывание отложено
28	PENDSVSET	R/W	0	Запись 1 в этот бит переводит исключение PendSV в состояние ожидания. При чтении возвращается текущее состояние
27	PENDSVCLR	W	0	Запись 1 в этот бит очищает признак отложенного прерывания для исключения PendSV
26	PENDSTSET	R/W	0	Запись 1 в этот бит переводит исключение SYSTICK в состояние ожидания. При чтении возвращается текущее состояние
25	PENDSTCLR	W	0	Запись 1 в этот бит очищает признак отложенного прерывания для исключения SYSTICK
23	ISRPREEMPT	R	0	Отложенное прерывание станет активным на следующем шаге (используется для отладки)
22	ISRPENDING	R	0	Имеется отложенное внешнее прерывание
21:12	VECTPENDING	R	0	Номер отложенного исключения с наибольшим приоритетом
11	RETTOBASE	R	0	Устанавливается в 1 при выполнении процессором обработчика исключения; обеспечивает переход процессора в режим потока после возврата из прерывания при отсутствии других отложенных исключений
8:0	VECTACTIVE	R	0	Номер обрабатываемого в данный момент исключения

Таблица Г.15. **Регистр смещения таблицы векторов VTOR (0xE000ED08)**

Биты	Обозначение	Тип	Значение после сброса	Описание
29	TBLBASE	R/W	—	Таблица расположена в области кода (0) или в области ОЗУ (1)
28:7	TBLOFF	R/W	—	Смещение таблицы относительно начала области кода или области ОЗУ

Таблица Г.16. **Регистр управления прерываниями и сбросом AIRCR (0xE000ED0C)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:16	VECTKEY	R/W	—	Ключ доступа; при записи в регистр в данном поле должно быть записано значение 0x05FA, иначе запись будет проигнорирована. При чтении регистра в этих битах возвращается значение 0xFA50
15	ENDIANNESS	R	—	Отображает формат хранения данных: 1 — обратный порядок байтов (BE-8), 0 — прямой порядок байтов; этот бит может быть изменён только в момент сброса
10:8	PRIGROUP	R/W	0	Группирование приоритетов
2	SYSRESETREQ	W	—	Формирует запрос генерации сигнала сброса
1	VECTCLRACTIVE	W	—	Очищает всю активную информацию о состоянии исключений; обычно используется при отладке или операционной системой для восстановления после системной ошибки (сброс является более безопасным методом)
0	VECTRESET	W	—	Сбрасывает процессор Cortex-M3 (за исключением компонентов отладки), однако не сбрасывает блоки, внешние по отношению к процессору

Таблица Г.17. **Регистр управления системой SCR (0xE000ED10)**

Биты	Обозначение	Тип	Значение после сброса	Описание
4	SEVONPEND	R/W	0	Формирование события при откладывании. Разрешает пробуждение процессора из спящего режима командой `WFE` при появлении нового отложенного прерывания, независимо от приоритета этого прерывания
3	Зарезервировано	—	—	—
2	SLEEPDEEP	R/W	0	Разрешает формирование сигнала SLEEPDEEP при переходе в спящий режим
1	SLEEPONEXIT	R/W	0	Включает функцию Sleep-on-exit
0	Зарезервировано	—	—	—

Таблица Г.18. **Регистр управления конфигурацией CCR (0xE000ED14)**

Биты	Обозначение	Тип	Значение после сброса	Описание
9	STKALIGN	R/W	0 или 1	Включить выравнивание на границу двойного слова при сохранении данных в стеке во время исключения. В ревизии 1 процессора этот бит по умолчанию сброшен в 0, а в ревизии 2 — установлен в 1. В ревизии 0 процессора такая возможность отсутствует
8	BFHFNMIGN	R/W	0	Игнорировать отказы шины при обработке исключения Hard Fault и немаскируемого прерывания
7:5	Зарезервировано	—	—	Зарезервировано
4	DIV_0_TRP	R/W	0	Перехватывать деление на ноль
3	UNALIGN_TRP	R/W	0	Перехватывать обращения к невыровненным данным
2	Зарезервировано	—	—	Зарезервировано
1	USERSETMPEND	R/W	0	Разрешить пользовательскому коду выполнять запись в регистр программной генерации прерывания STIR
0	NONBASETHRDENA	R/W	0	Разрешить обработчикам исключений возвращаться в состояние потока на любом уровне вложенности

Таблица Г.19. **Регистры уровня приоритета системных исключений PRI_x (0xE000ED18…0xE000ED23; адресация в таблице — побайтовая)**

Адрес	Обозначение	Тип	Значение после сброса	Описание
0xE000ED18	PRI_4	R/W	0	Уровень приоритета исключения MemManage Fault
0xE000ED19	PRI_5	R/W	0	Уровень приоритета исключения Bus Fault
0xE000ED1A	PRI_6	R/W	0	Уровень приоритета исключения Usage Fault
0xE000ED1B	—	—	—	—
0xE000ED1C	—	—	—	—
0xE000ED1D	—	—	—	—
0xE000ED1E	—	—	—	—
0xE000ED1F	PRI_11	R/W	0	Уровень приоритета исключения SVCall
0xE000ED20	PRI_12	R/W	0	Уровень приоритета исключения монитора отладки
0xE000ED21	—	—	—	—
0xE000ED22	PRI_14	R/W	0	Уровень приоритета исключения PendSV
0xE000ED23	PRI_15	R/W	0	Уровень приоритета исключения SYSTICK

Таблица Г.20. **Регистр управления и состояния обработчиков системных исключений SHCSR (0xE000ED24)**

Биты	Обозначение	Тип	Значение после сброса	Описание
18	USGFAULTENA	R/W	0	Разрешает исключение Usage Fault
17	BUSFAULTENA	R/W	0	Разрешает исключение Bus Fault
16	MEMFAULTENA	R/W	0	Разрешает исключение MemManage Fault
15	SVCALLPENDED	R/W	0	Исключение SVCall отложено; выполнение обработчика исключения SVCall было прервано исключением с более высоким приоритетом
14	BUSFAULTPENDED	R/W	0	Исключение Bus Fault отложено; выполнение обработчика исключения Bus Fault было прервано исключением с более высоким приоритетом
13	MEMFAULTPENDED	R/W	0	Исключение MemManage Fault отложено; выполнение обработчика исключения MemManage Fault было прервано исключением с более высоким приоритетом
12	USGFAULTPENDED	R/W	0	Исключение Usage Fault отложено; выполнение обработчика исключения Usage Fault было прервано исключением с более высоким приоритетом*
11	SYSTICKACT	R/W	0	Читается как 1, если исключение SYSTICK активно
10	PENDSVACT	R/W	0	Читается как 1, если исключение PendSV активно
8	MONITORACT	R/W	0	Читается как 1, если исключение монитора отладки активно
7	SVCALLACT	R/W	0	Читается как 1, если исключение SVCall активно
3	USGFAULTACT	R/W	0	Читается как 1, если исключение Usage Fault активно
1	BUSFAULTACT	R/W	0	Читается как 1, если исключение Bus Fault активно
0	MEMFAULTACT	R/W	0	Читается как 1, если исключение MemManage Fault активно
*Бит 12 (USGFAULTPENDED) в ревизии 0 процессора Cortex-M3 отсутствует.				

Таблица Г.21. **Регистр состояния отказа системы управления памятью MMFSR (0xE000ED28; размер — 1 байт)**

Биты	Обозначение	Тип	Значение после сброса	Описание
7	MMARVALID	—	0	Индицирует корректность содержимого регистра MMAR
6:5	—	—	—	—
4	MSTKERR	R/Wc	0	Ошибка загрузки в стек
3	MUNSTKERR	R/Wc	0	Ошибка извлечения из стека
2	—	—	—	—
1	DACCVIOL	R/Wc	0	Нарушение доступа к данным
0	IACCVIOL	R/Wc	0	Нарушение доступа к команде

Таблица Г.22. **Регистр состояния отказа шины BFSR (0xE000ED29; размер — 1 байт)**

Биты	Обозначение	Тип	Значение после сброса	Описание
7	BFARVALID	—	0	Индицирует корректность содержимого регистра BFAR
6:5	—	—	—	—
4	STKERR	R/Wc	0	Ошибка загрузки в стек
3	UNSTKERR	R/Wc	0	Ошибка извлечения из стека
2	IMPRECISERR	R/Wc	0	«Неточное» нарушение доступа к данным
1	PRECISERR	R/Wc	0	«Точное» нарушение доступа к данным
0	IBUSERR	R/Wc	0	Нарушение доступа к команде

Таблица Г.23. **Регистр состояния отказа программы UFSR (0xE000ED2A; размер — 1 полуслово)**

Биты	Обозначение	Тип	Значение после сброса	Описание
9	DIVBYZERO	R/Wc	0	Попытка выполнения операции деления на ноль (может быть установлен только при установленном бите DIV_0_TRP регистра CCR)
8	UNALIGNED	R/Wc	0	Ошибка невыровненного доступа (может быть установлен только при установленном бите UNALIGN_TRP регистра CCR)
7:4	—	—	—	—
3	NOCP	R/Wc	0	Попытка выполнения команды сопроцессора
2	INVPC	R/Wc	0	Попытка выхода из обработчика исключения с некорректным значением EXC_RETURN
1	INVSTATE	R/Wc	0	Попытка переключения в некорректное состояние (например, в состояние ARM)
0	UNDEFINSTR	R/Wc	0	Попытка выполнения неопределённой команды

Таблица Г.24. **Регистр состояния тяжёлого отказа HFSR (0xE000ED2C)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31	DEBUGEVT	R/Wc	0	Исключение Hard Fault было вызвано событием отладки
30	FORCED	R/Wc	0	Исключение Hard Fault было вызвано отказом шины, отказом системы управления памятью или отказом программы
29:2	—	—	—	—
1	VECTBL	R/Wc	0	Исключение Hard Fault было вызвано ошибкой выборки вектора
0	—	—	—	—

Таблица Г.25. **Регистр состояния отказа отладки DFSR (0xE000ED30)**

Биты	Обозначение	Тип	Значение после сброса	Описание
4	EXTERNAL	R/Wc	0	Активирован сигнал EDBGRQ
3	VCATCH	R/Wc	0	Произведена выборка вектора
2	DWTTRAP	R/Wc	0	Произошло событие совпадения DWT
1	BKPT	R/Wc	0	Выполнена команда `BKPT`
0	HALTED	R/Wc	0	Останов запрошен контроллером NVIC

Таблица Г.26. **Регистр адреса отказа системы управления памятью MMFAR (0xE000ED34)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:0	ADDRESS	R	—	Адрес памяти, обращение к которому вызвало отказ системы управления памятью

Таблица Г.27. **Регистр адреса отказа шины BFAR (0xE000ED38)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:0	ADDRESS	R	—	Адрес памяти, обращение к которому вызвало отказ шины

Таблица Г.28. **Дополнительный регистр состояния отказа AFSR (0xE000ED3C)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:0	Определяется производителем	R/Wc	0	Определяется производителем

Таблица Г.29. **Регистр типа модуля MPU MPU_TYPE (0xE000ED90)**

Биты	Обозначение	Тип	Значение после сброса	Описание
23:16	IREGION	R	—	Число областей команд, поддерживаемых данным MPU; поскольку в архитектуре ARMv7-M используется унифицированный модуль MPU, это поле всегда содержит 0
15:8	DREGION	R	—	Число областей, поддерживаемых данным MPU
0	SEPARATE	R	—	Этот бит всегда равен 0, поскольку используется унифицированный модуль MPU

Таблица Г.30. **Регистр управления MPU MPU_CTRL (0xE000ED94)**

Биты	Обозначение	Тип	Значение после сброса	Описание
2	PRIVDEFENA	R/W	0	Разрешение доступа к карте памяти в привилегированном режиме
1	HFNMIENA	R/W	0	Установленный бит разрешает работу модуля MPU в обработчиках NMI и исключения Hard Fault. Если бит сброшен, то при выполнении этих обработчиков модуль MPU не используется (блокируется)
0	ENABLE	R/W	0	Установка этого бита в 1 разрешает работу модуля MPU

Таблица Г.31. **Регистр номера региона MPU MPU_RNR (0xE000ED98)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:0	REGION	R/W	0	Определяет номер программируемой области

Таблица Г.32. **Регистр базового адреса области MPU MPU_RBAR (0xE000ED9C)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:N	ADDR	R/W	0	Базовый адрес области
4	VALID	R/W	0	Если бит установлен в 1, то номер области MPU, программируемой на данном этапе, будет определяться полем REGION этого регистра. Если бит сброшен в 0, то используется область, заданная регистром MPU_RNR
3:0	REGION	R/W	0	Если бит VALID установлен в 1, то это поле используется вместо регистра номера области MPU_RNR; в противном случае, содержимое данного поля игнорируется

Таблица Г.33. **Регистр атрибутов и размера области MPU MPU_RASR (0xE000EDA0)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:29	Зарезервировано	—	—	Зарезервировано
28	XN	R/W	0	Исполнение команд запрещено
27	Зарезервировано	—	—	Зарезервировано
26:24	AP	R/W	000	Поле прав доступа к данным
23:22	Зарезервировано	—	—	Зарезервировано
21:19	TEX	R/W	000	Поле расширения типа
18	S	R/W	—	Разделяемая
17	C	R/W	—	Кэшируемая
16	B	R/W	—	Буферируемая
15:8	SRD	R/W	0x00	Запрет подобластей
7:6	Зарезервировано	—	—	Зарезервировано
5:1	SIZE	R/W	—	Размер области MPU
0	ENABLE	R/W	0	Разрешение области

Таблица Г.34. **Регистры псевдонимов регистров MPU_RBAR/ MPU_RASR (0xE000EDA4…0xE000EDB8)**

Адрес	Регистр	Описание
0xE000EDA4	Псевдоним D9C	Псевдоним 1 регистра MPU_RBAR
0xE000EDA8	Псевдоним DA0	Псевдоним 1 регистра MPU_RASR
0xE000EDAC	Псевдоним D9C	Псевдоним 2 регистра MPU_RBAR
0xE000EDB0	Псевдоним DA0	Псевдоним 2 регистра MPU_RASR
0xE000EDB4	Псевдоним D9C	Псевдоним 3 регистра MPU_RBAR
0xE000EDB8	Псевдоним DA0	Псевдоним 3 регистра MPU_RASR

Таблица Г.35. **Регистр управления остановом и состояния отладчика DHCSR (0xE000EDF0)**

Биты	Обозначение	Тип	Значение после сброса	Описание
31:16	KEY	W	—	Ключ доступа к функциям отладки; при записи в регистр в этом поле должно быть записано значение 0xA05F, иначе запись будет проигнорирована
25	S_RESET_ST	R	—	Ядро было сброшено или находится в состоянии сброса. Бит сбрасывается при чтении
24	S_RETIRE_ST	R	—	Команда завершена с момента последнего чтения регистра. Бит сбрасывается при чтении
19	S_LOCKUP	R	—	Если бит установлен в 1, то ядро находится в состоянии блокировки
18	S_SLEEP	R	—	Если бит установлен в 1, то ядро находится в спящем режиме
17	S_HALT	R	—	Если бит установлен в 1, то ядро остановлено

Таблица Г.35. **Регистр управления остановом и состояния отладчика DHCSR (0xE000EDF0)** *(продолжение)*

Биты	Обозначение	Тип	Значение после сброса	Описание
16	S_REGRDY	R	—	Операция чтения/записи регистра завершена
15:6	Зарезервировано	—	—	Зарезервировано
5	C_SNAPSTALL	R/W	—	Прерывание «зависших» операций обращения к памяти
4	Зарезервировано	—	—	Зарезервировано
3	C_MASKINTS	R/W	—	Маскирование прерываний в спящем режиме; бит может быть изменён только при остановленном процессоре
2	C_STEP	R/W	—	Выполнение одной команды (пошаговая работа процессора); действителен только при установленном бите C_DEBUGEN
1	C_HALT	R/W	—	Останов процессорного ядра; действителен только при установленном бите C_DEBUGEN
0	C_DEBUGEN	R/W	—	Разрешение отладки в режиме останова

Таблица Г.36. **Регистр выбора регистра ядра DCRSR (0xE000EDF4)**

Биты	Обозначение	Тип	Значение после сброса	Описание
16	REGWnR	W	—	Направление передачи данных: Запись = 1, Чтение = 0
15:5	Зарезервировано	—	—	—
4:0	REGSEL	W	—	Регистр, к которому производится обращение: 00001 = R1; … 01111 = R15; 10000 = xPSR/Флаги; 10001 = MSP; 10010 = PSP; 10100 = Регистры специального назначения: [31:24] — CONTROL, [23:16] — FAULTMASK, [15:8] — BASEPRI, [7:0] — PRIMASK. Остальные значения зарезервированы

Таблица Г.37. **Регистр содержимого регистра ядра DCRDR (0xE000EDF8)**

Биты	Обозначение	Тип	Значение после сброса	Описание
32:0	Data	R/W	—	Регистр данных для хранения результата чтения регистра или значения, записываемого в регистр

Таблица Г.38. **Регистр управления исключением и монитором отладки DEMCR (0xE000EDFC)**

Биты	Обозначение	Тип	Значение после сброса	Описание
24	TRCENA	R/W	0	Разрешение работы системы трассировки; должен быть установлен в 1 для использования модулей DWT, ETM, ITM и TPIU
23:20	Зарезервировано	—	—	Зарезервировано
19	MON_REQ	R/W	0	Указывает на то, что вызов монитора отладки произошёл в результате ручной установки бита признака отложенного прерывания, а не в результате события аппаратного отладчика
18	MON_STEP	R/W	0	Выполнение одной команды (пошаговая работа процессора); действителен только при установленном бите MON_EN
17	MON_PEND	R/W	0	Генерация запроса исключения Debug monitor; процессор приступит к обработке исключения согласно его приоритету
16	MON_EN	R/W	0	Разрешение исключения Debug monitor
15:11	Зарезервировано	—	—	Зарезервировано
10	VC_HARDERR	R/W	0	Перехват тяжёлых отказов
9	VC_INTERR	R/W	0	Перехват отказов, возникающих при входе в обработчик исключения и при выходе из него
8	VC_BUSERR	R/W	0	Перехват отказов шины
7	VC_STATERR	R/W	0	Перехват отказов программы, вызванных некорректной информацией о состоянии
6	VC_CHKERR	R/W	0	Перехват отказов программы, вызванных обращением к невыровненным данным либо делением на ноль
5	VC_NOCPERR	R/W	0	Перехват отказов программы, вызванных попыткой обращения к сопроцессору
				Перехват ошибок, возникающих при входе в обработчик исключения и при выходе из него
4	VC_MMERR	R/W	0	Перехват отказов системы управления памятью
3:1	Зарезервировано	—	—	Зарезервировано
0	VC_CORERESE	R/W	0	Перехват сброса ядра

Таблица Г.39. **Регистр программной генерации прерывания STIR (0xE000EF00)**

Биты	Обозначение	Тип	Значение после сброса	Описание
8:0	INTID	W	—	При записи в это поле какого-либо числа устанавливается бит признака отложенного прерывания для прерывания с соответствующим номером

Таблица Г.40. **Регистры идентификатора периферии контроллера NVIC (0xE000EFD0…0xE0EFFC)**

Адрес	Обозначение	Тип	Значение после сброса	Описание
0xE000EFD0	PERIPHID4	R	0x04	Идентификатор периферийного устройства
0xE000EFD4	PERIPHID5	R	0x00	Идентификатор периферийного устройства
0xE000EFD8	PERIPHID6	R	0x00	Идентификатор периферийного устройства
0xE000EFDC	PERIPHID7	R	0x00	Идентификатор периферийного устройства
0xE000EFE0	PERIPHID0	R	0x00	Идентификатор периферийного устройства
0xE000EFE4	PERIPHID1	R	0xB0	Идентификатор периферийного устройства
0xE000EFE8	PERIPHID2	R	0x0B/0x1B/0x2B*	Идентификатор периферийного устройства
0xE000EFEC	PERIPHID3	R	0x00	Идентификатор периферийного устройства
0xE000EFF0	PCELLID0	R	0x0D	Идентификатор компонента
0xE000EFF4	PCELLID1	R	0xE0	Идентификатор компонента
0xE000EFF8	PCELLID2	R	0x05	Идентификатор компонента
0xE000EFFC	PCELLID0	R	0xB1	Идентификатор компонента
*0x0B для ревизии 0 процессора Cortex-M3, 0x1B — для ревизии 1 и 0x2B — для ревизии 2.				

ПРИЛОЖЕНИЕ Д
РУКОВОДСТВО ПО ЛОКАЛИЗАЦИИ ОШИБОК В ПРОГРАММАХ ДЛЯ CORTEX-M3

Д.1. Общие сведения

Среди различных задач, возникающих при использовании процессора Cortex-M3, чуть ли не на первом месте стоит задача локализации ошибок в случае некорректной работы программы. К счастью, в процессоре Cortex-M3 предусмотрено несколько регистров состояния отказов, облегчающих выполнение указанной задачи (**Табл. Д.1**). Причём, к трём из данных регистров (MMFSR, BFSR и UFSR) можно обращаться одновременно с помощью команд пересылки 32-битных значений. В таком случае используется специальный комбинированный регистр состояния отказов CFSR (**Рис. Д.1**).

Таблица Д.1. **Регистры состояния отказов процессора Cortex-M3**

Адрес	Регистр	Полное название	Размер
0xE000ED28	MMFSR	Регистр состояния отказа системы управления памятью	Байт
0xE000ED29	BFSR	Регистр состояния отказа шины	Байт
0xE000ED2A	UFSR	Регистр состояния отказа программы	Полуслово
0xE000ED2C	HFSR	Регистр состояния тяжёлого отказа	Слово
0xE000ED30	DFSR	Регистр состояния отказа отладки	Слово
0xE000ED3C	AFSR	Дополнительный регистр состояния отказа	Слово

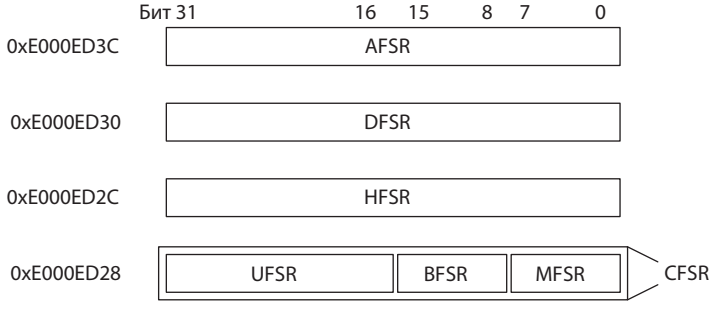


Рис. Д.1. *Обращение к регистрам состояния отказов.*

При наличии CMSIS-совместимой библиотеки к этим регистрам можно обращаться как к элементам структуры:

- `SCB -> CFSR` — регистр состояния конфигурируемых отказов;
- `SCB -> HFSR` — регистр состояния тяжёлого отказа;
- `SCB -> DFSR` — регистр состояния отказа отладки;
- `SCB -> AFSR` — дополнительный регистр состояния отказа.

Ещё одним источником информации является значение счётчика команд, сохраняемое в стеке. При входе в обработчик исключения отказа содержимое счётчика располагается по адресу [SP+24]. Поскольку в процессоре Cortex-M3 имеется два указателя стека, то перед извлечением данного значения из стека обработчик должен определить используемый указатель.

В случае отказов шины или системы управления памятью вы можете даже определить адрес, обращение к которому вызвало ошибку. Для отказов системы управления памятью этот адрес содержится в регистре MMFAR, а для отказов шины — в регистре BFAR. Причём, содержимое указанных регистров можно считать достоверным только в том случае, если установлен бит MMARVALID (регистр MMFSR) или BFARVALID (регистр BFSR) соответственно. Регистры MMFAR и BFAR физически являются одним и тем же регистром, поэтому в каждый момент времени только один из них может содержать корректное значение (**Табл. Д.2**).

Таблица Д.2. **Регистры адреса отказа процессора Cortex-M3**

Адрес	Регистр	Полное название	Размер
0xE000ED34	MMFAR	Регистр адреса отказа системы управления памятью	Слово
0xE000ED38	BFAR	Регистр адреса отказа шины	Слово

При использовании CMSIS-совместимых драйверов устройств к регистрам адресов отказов можно обращаться аналогично регистрам состояния отказов:

- `SCB -> MMFAR` — регистр адреса отказа системы управления памятью;
- `SCB -> BFAR` — регистр адреса отказа шины.

Последним источником информации о причинах ошибки может стать регистр связи (LR). Если отказ был вызван некорректным значением EXC_RETURN, то при входе в обработчик отказа содержимое данного регистра будет таким же, как и на момент возникновения ошибки. Обнаружив некорректность содержимого LR, программист может использовать данную информацию для определения причин записи в этот регистр неверного значения возврата.

Д.2. Обработчики исключений отказов

В большинстве случаев обработчики отказов, используемые при разработке устройства, довольно сильно отличаются от тех же обработчиков, присутствующих в уже работающей системе. На этапе создания программы данные обработчики нужны, главным образом, для того, чтобы извещать программиста о возникших проблемах, тогда как в работающей системе обработчики отказов должны, в первую очередь, заниматься восстановлением работоспособности этой системы. В данном подразделе мы коснёмся только вопроса формирования сообще-

ний об ошибках, поскольку действия, необходимые для восстановления системы, очень сильно зависят от назначения и параметров конкретного устройства.

Если программа достаточно сложная, то в самом обработчике отказа никаких сообщений обычно не выводится. Вместо этого содержимое соответствующих регистров сохраняется в памяти, а детальный отчёт о возникшей проблеме формируется уже в обработчике исключения SVCall. Такое решение позволяет предотвратить возникновение отказов, которые могут вызвать блокировку процессора, при отображении или выводе служебной информации. В простых приложениях такие предосторожности, как правило, излишни, и выводом информации может заниматься сам обработчик отказа.

Д.2.1. Вывод содержимого регистров состояния отказов

Прежде всего, обработчик отказа должен передать программисту значение соответствующего регистра состояния отказа. Речь идёт о следующих регистрах:

- UFSR;
- BFSR;
- MMFSR;
- HFSR;
- DFSR;
- AFSR (опция).

Д.2.2. Вывод содержимого PC и прочих регистров, сохранённых в стеке

Чтобы извлечь из стека значение PC, в обработчике отказа может использоваться следующий код:

```
TST   LR, #0x4    ; Проверяем 2-й бит значения EXC_RETURN в регистре LR
ITTEE EQ          ; Если ноль (равно), то
MRSEQ R0, MSP     ; использовался основной стек, помещаем MSP в R0
LDREQ R0,[R0,#24] ; Извлекаем старое значение PC из стека
MRSNE R0, PSP     ; Иначе, использовался стек процесса, помещаем PSP в R0
LDRNE R0,[R0,#24] ; Извлекаем старое значение PC из стека
```

И всё же большинство проектов для процессора Cortex-M3 создаётся на языке Си. Однако в программах на Си определить местоположение стекового фрейма для последующего обращения к нему бывает весьма непросто, что связано с невозможностью получить стандартными средствами языка значение указателя стека. Для передачи содержимого стекового фрейма в обработчик отказа, написанный на языке Си, необходимо использовать своеобразную ассемблерную «надстройку». Эта «надстройка» будет извлекать значение указателя стека и передавать его в функцию, формирующую сообщения об ошибке (**Рис. Д.2**). Данный механизм идентичен описанному в Главе 12. В следующем примере мы продемонстрируем использование встроенного ассемблера, поддерживаемого пакетами RVDS и MDK-ARM.

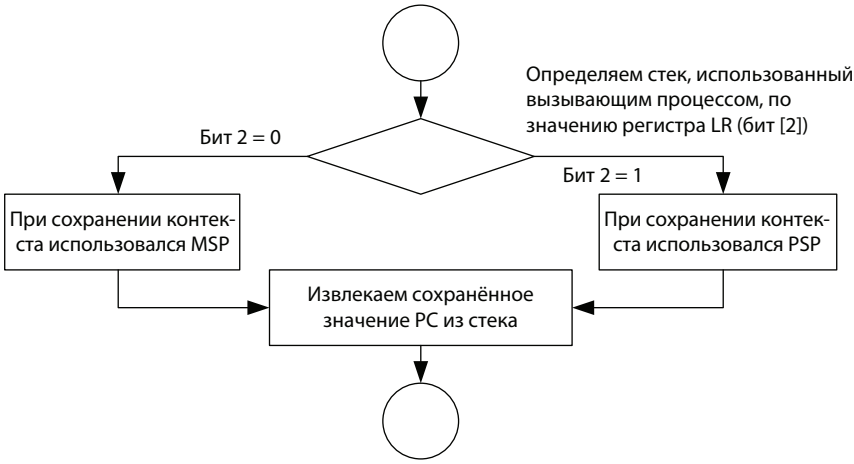


Рис. Д.2. *Извлечение из стека исходного значения PC.*

Сначала напишем ассемблерную «надстройку». Адрес этой «надстройки» должен располагаться в таблице векторов на месте вектора исключения Hard Fault. Данная «надстройка» копирует значение используемого указателя стека в регистр R0, после чего вызывает собственно обработчик, написанный на Си.

```
// Ассемблерная «надстройка» для обработчика тяжёлого отказа
// Определяет адрес стекового фрейма и передаёт его
// как указатель в обработчик исключения
__asm void hard_fault_handler_asm(void)
{
  TST   LR, #4
  ITE   EQ
  MRSEQ R0, MSP
  MRSNE R0, PSP
  B     __cpp(hard_fault_handler_c)
}
```

Вторую часть обработчика напишем на Си. Используя переданный указатель, мы можем считать содержимое регистров, сохранённых в стеке:

```
// Обработчик исключения Hard Fault, принимающий
// в качестве входного параметра адрес стекового фрейма
void hard_fault_handler_c(unsigned int * hardfault_args)
{
  unsigned int stacked_r0;
  unsigned int stacked_r1;
  unsigned int stacked_r2;
  unsigned int stacked_r3;
  unsigned int stacked_r12;
  unsigned int stacked_lr;
  unsigned int stacked_pc;
  unsigned int stacked_psr;

  stacked_r0  = ((unsigned long) hardfault_args[0]);
  stacked_r1  = ((unsigned long) hardfault_args[1]);
```

```
  stacked_r2  = ((unsigned long) hardfault_args[2]);
  stacked_r3  = ((unsigned long) hardfault_args[3]);

  stacked_r12 = ((unsigned long) hardfault_args[4]);
  stacked_lr  = ((unsigned long) hardfault_args[5]);
  stacked_pc  = ((unsigned long) hardfault_args[6]);
  stacked_psr = ((unsigned long) hardfault_args[7]);

  printf («[Обработчик Hard Fault]\n»);
  printf («R0 = %x\n», stacked_r0);
  printf («R1 = %x\n», stacked_r1);
  printf («R2 = %x\n», stacked_r2);
  printf («R3 = %x\n», stacked_r3);
  printf («R12 = %x\n», stacked_r12);
  printf («LR = %x\n», stacked_lr);
  printf («PC = %x\n», stacked_pc);
  printf («PSR = %x\n», stacked_psr);
  printf («BFAR = %x\n», (*((volatile unsigned long *)(0xE000ED38))));
  printf («CFSR = %x\n», (*((volatile unsigned long *)(0xE000ED28))));
  printf («HFSR = %x\n», (*((volatile unsigned long *)(0xE000ED2C))));
  printf («DFSR = %x\n», (*((volatile unsigned long *)(0xE000ED30))));
  printf («AFSR = %x\n», (*((volatile unsigned long *)(0xE000ED3C))));

  exit(0); // Завершаем работу

  return;
}
```

Хочу обратить ваше внимание на то, что данный обработчик будет работать некорректно, если указатель стека указывает на неверную область памяти (например, из-за переполнения стека). Это касается любого кода, написанного на Си, поскольку практически любая функция, написанная на Си, использует стек.

Чтобы облегчить себе работу по поиску ошибок, можно сгенерировать файл листинга с дизассемблированным кодом.

Д.2.3. Чтение регистра адреса отказа

После сброса бита MMARVALID или BFARVALID содержимое регистра адреса отказа может оказаться стёртым. Поэтому обращаться к этому регистру необходимо следующим образом:

1. Прочитать регистр BFAR/MMAR.
2. Проверить бит BFARVALID/MMARVALID. Если он сброшен, то содержимое регистра BFAR/MMAR следует считать недействительным.
3. Сбросить бит BFARVALID/MMARVALID.

Бит корректности содержимого регистра адреса проверяется уже после чтения самого регистра — это сделано для того, чтобы избежать вытеснения текущего обработчика более высокоприоритетным между чтением бита корректности и собственно регистра адреса. В противном случае, возможен следующий сценарий развития событий:

1. Читаем бит BFARVALID/MMARVALID.

2. Бит корректности установлен, приступаем к чтению регистра BFAR/MMFAR.
3. Исключение с более высоким приоритетом прерывает выполнение текущего обработчика, в результате чего возникает новый отказ и запускается соответствующий обработчик отказа.
4. Обработчик отказа с более высоким приоритетом сбрасывает бит BFARVALID/MMARVALID, вызывая очистку регистра BFAR/MMFAR.
5. После возврата в исходный обработчик отказа читается регистр BFAR/MMFAR. Но поскольку его содержимое уже некорректно, мы получаем неверное значение адреса отказа.

Таким образом, чтобы гарантировать корректность содержимого регистра адреса отказа, бит корректности необходимо читать только после чтения регистра адреса.

Д.2.4. Сброс битов состояния отказов

После формирования сообщения об отказе соответствующий бит регистра состояния отказа необходимо сбрасывать, чтобы не ввести в заблуждение обработчик исключения при последующем его запуске. Если бит корректности адреса не сбросить, то при возникновении очередного отказа содержимое регистра адреса отказа изменено не будет.

Д.2.5. Прочие вопросы

После входа в обработчик отказа часто возникает необходимость сохранения содержимого регистра связи. Однако если отказ был вызван ошибкой стека, то помещение регистра LR в стек может лишь усугубить ситуацию. Как мы знаем, регистры R0…R3 и R12 к этому моменту уже сохранены в стеке, поэтому можем спокойно скопировать содержимое LR в любой из указанных регистров.

Д.3. Определение причины отказа

На основе полученной информации мы можем выяснить причину некорректной работы программы. Наиболее распространённые причины возникновения отказов приведены в **Табл. Д.3…Д.7**.

Таблица Д.3. **Отказы системы управления памятью (регистр MMFSR)**

Бит	Возможные причины отказа
MMARVALID (бит 7)	Показывает, что в регистре MMFAR содержится корректное значение адреса, обращение к которому вызвало отказ
MSTKERR (бит 4)	Ошибка произошла во время сохранения контекста в стеке (при входе в исключение). 1. Указатель стека повреждён. 2. Размер стека слишком велик, в результате чего указатель стека попал в область, не определённую или запрещённую конфигурацией модуля MPU.
MUNSTKERR (бит 3)	Ошибка произошла во время восстановления контекста из стека (при выходе из исключения). 1. Указатель стека был повреждён в обработчике исключения. 2. Конфигурация модуля MPU была изменена в обработчике исключения.

Таблица Д.3. **Отказы системы управления памятью (регистр MMFSR)** *(продолжение)*

Бит	Возможные причины отказа
DACCVIOL (бит 1)	Нарушение прав доступа к памяти, заданных конфигурацией модуля MPU. Например, попытка пользовательского приложения обратиться к области, доступной только для привилегированного доступа
IACCVIOL (бит 0)	1. Нарушение прав доступа к памяти, определённых конфигурацией модуля MPU. Например, попытка пользовательского приложения обратиться к области, доступной только в привилегированном режиме. Для отыскания проблемного кода можно воспользоваться значением PC, сохранённым в стеке. 2. Переход в область памяти, не поддерживающую выборку команд. 3. Неверный код возврата из исключения. 4. Некорректное значение элемента таблицы векторов. Как пример, загрузка в память исполняемого образа для традиционного процессора ARM или же возникновение исключения до инициализации таблицы векторов. 5. Повреждение значения PC, находящегося в стеке, во время обработки исключения.

Таблица Д.4. **Отказы шины (регистр BFSR)**

Бит	Возможные причины отказа
BFARRVALID (бит 7)	Показывает, что в регистре BFAR содержится корректное значение адреса, обращение к которому вызвало отказ
STKERR (бит 4)	Ошибка произошла во время сохранения контекста в стеке (при входе в исключение). 1. Указатель стека повреждён. 2. Размер стека слишком велик, в результате чего указатель стека попал в область, не определённую или запрещённую конфигурацией модуля MPU. 3. Используется неинициализированный указатель PSP.
UNSTKERR (бит 3)	Ошибка произошла во время восстановления контекста из стека (при выходе из исключения). Если при сохранении контекста ошибки не было, то причиной отказа, скорее всего, является повреждение указателя стека во время выполнения обработчика исключения
IMPRECISERR (бит 2)	Ошибка на шине при обращении к данным. Ошибка может быть вызвана: 1. Обращением к неинициализированному устройству. 2. Обращением в пользовательском режиме к устройству, допускающему обращение только в привилегированном режиме. 3. Выполнением пересылки с размером, не поддерживаемым устройством.
PRECISERR (бит 1)	Ошибка на шине при обращении к данным. Адрес, обращение к которому вызвало отказ, может содержаться в регистре BFAR. Ошибка может быть вызвана: 1. Обращением к неинициализированному устройству. 2. Обращением в пользовательском режиме к устройству, допускающему обращение только в привилегированном режиме. 3. Выполнением пересылки с размером, не поддерживаемым устройством.

Таблица Д.4. **Отказы шины (регистр BFSR)** *(продолжение)*

Бит	Возможные причины отказа
IBUSERR (бит 0)	1. Переход в недопустимую область памяти; например, вызванный неверным значением указателя на функцию. 2. Некорректное значение кода возврата из исключения; например, при порче сохранённого в стеке значения EXC_RETURN, в результате чего возврат из исключения ошибочно интерпретируется как операция перехода. 3. Некорректное значение элемента таблицы векторов. Как пример, загрузка в память исполняемого образа для традиционного процессора ARM или же возникновение исключения до инициализации таблицы векторов. 4. Повреждение значения PC, находящегося в стеке, во время выполнения подпрограммы. 5. Обращение к регистрам контроллера NVIC или SCB в пользовательском режиме (непривилегированное).

Таблица Д.5. **Отказы программы (регистр UFSR)**

Бит	Возможные причины отказа
DIVBYZERO (бит 9)	Произошло деление на ноль при установленном бите DIV_0_TRP. Местоположение команды, вызвавшей отказ, можно определить по значению PC, сохранённому в стеке
UNALIGNED (бит 8)	Была попытка невыровненного доступа при установленном бите UNALIGN_TRP. Местоположение команды, вызвавшей отказ, можно определить по значению PC, сохранённому в стеке
NOCP (бит 3)	Попытка выполнения команды сопроцессора. Местоположение команды, вызвавшей отказ, можно определить по значению PC, сохранённому в стеке
INVPC (бит 2)	1. Некорректное значение EXC_RETURN во время возврата из исключения. Например: а) возврат в режим потока при EXC_RETURN = 0xFFFFFFF1; б) возврат в режим обработчика при EXC_RETURN = 0xFFFFFFF9. При этом текущее содержимое регистра LR соответствует тому, которое было на момент возврата из исключения, вызвавшего отказ. 2. Неверное значение признака активности исключения. Например: а) возврат из исключения при сброшенном бите признака активности исключения. Может быть вызвано записью в бит VECTCLTRACTIVE регистра AIRCR или сбросом бита признака активности в регистре SHCSR контроллера NVIC; б) возврат из исключения в режим потока при наличии одного и более активных исключений. 3. Повреждение стека, вызвавшее загрузку неверного значения в регистр IPSR. В данном случае содержимое PC, находящееся в стеке, будет соответствовать месту, в котором было прервано выполнение основной программы. Для определения причины отказа лучше всего использовать функцию трассировки исключений модуля DWT. 4. Некорректное для текущей команды значение бита ICI/IT. Данная ошибка может возникнуть в том случае, если в обработчике прерывания, прервавшего выполнение операции групповой загрузки/сохранения, было изменено значение PC, находящееся в стеке. Это приведёт к тому, что после возврата из прерывания бит ICI окажется установленным при выполнении команды, не использующей биты ICI. То же может произойти из-за повреждения содержимого регистра PSR, находящегося в стеке.

***Таблица Д.5.* Отказы программы (регистр UFSR)** *(продолжение)*

Бит	Возможные причины отказа
INVSTATE (бит 1)	1. Загрузка в счётчик команд адреса перехода со сброшенным младшим битом. Адрес перехода можно определить по содержимому PC, находящемуся в стеке. 2. Младший бит адреса вектора в таблице векторов сброшен в 0. Точку входа в обработчик исключения можно определить по содержимому PC, находящемуся в стеке. 3. Повреждение PSR, находящегося в стеке, во время обработки исключения и, как результат, попытка переключения ядра процессора в состояние ARM при возврате в прерванную программу.
UNDEFINSTR (бит 0)	1. Попытка выполнения команды, не поддерживаемой процессором Cortex-M3. 2. Некорректное/повреждённое содержимое ячейки памяти. 3. Загрузка на этапе компоновки объектного кода ARM. Проверьте настройки компилятора. 4. Проблема с выравниванием команд. Так, если при работе с инструментарием GNU после применения директивы `.ascii` не указать директиву `.align`, то последующие команды могут оказаться невыровненными (будут расположены по нечётным адресам).

***Таблица Д.6.* Тяжёлые отказы (регистр HFSR)**

Бит	Возможные причины отказа
DEBUGEVF (бит 31)	Отказ был вызван событием отладки: 1. События точки останова/наблюдения. 2. Выполнение команды `BKPT` при запрещённом исключении монитора отладки (MON_EN = 0) и запрещённой отладке в режиме останова (C_DEBUGEN = 0). Некоторые компиляторы Си могут по умолчанию вставлять в программу код поддержки полухостинга, использующий команды `BKPT`.
FORCED (бит 30)	1. Попытка выполнить команду `SVC`/`BKPT` из обработчика SVCall/Debug monitor или из обработчика другого исключения с таким же или более высоким приоритетом. 2. Отказ произошёл из-за того, что соответствующий обработчик запрещён или не может быть запущен по причине маскирования исключения или же выполнения обработчика другого исключения с таким же или более высоким приоритетом.
VECTBL (бит 1)	Ошибка выборки вектора. Возможные причины: 1. Отказ шины во время выборки вектора. 2. Некорректные настройки смещения таблицы векторов.

***Таблица Д.7.* Отказы отладки (регистр DFSR)**

Бит	Возможные причины отказа
EXTERNAL (бит 4)	Активация сигнала EDBGRQ
VCATCH (бит 3)	Произошло событие захвата вектора прерывания
DWTTRAP (бит 2)	Произошло событие точки наблюдения модуля DWT
BRPT (бит 1)	1. Была выполнена команда `BKPT`. 2. Модуль FPB генерировал событие точки останова. В некоторых случаях команды `BKPT` вставляются в итоговый код стартовой процедурой при инициализации отладки по методу полухостинга. В рабочем коде таких команд быть не должно. Пожалуйста, обратитесь к документации на используемый вами компилятор
HALTED (бит 0)	Запрос останова в контроллере NVIC

Д.4. Прочие возможные проблемы

Прочие проблемы, наиболее часто возникающие при разработке программ для процессоров с ядром Cortex-M3, указаны в **Табл. Д.8**.

***Таблица Д.8.* Прочие возможные проблемы**

Ситуация	Возможные причины
Программа не запускается	Возможно, неправильно сконфигурирована таблица векторов: 1. Таблица расположена по недопустимому адресу. 2. Младшие биты векторов (включая вектор исключения Hard Fault) не установлены в 1. 3. В таблице используются команды перехода (как в таблице векторов классических процессоров ARM). Сгенерируйте листинг с дизассемблированным кодом и удостоверьтесь, что таблица векторов настроена правильно
Программа «падает» после исполнения нескольких команд	Такое поведение программы может быть вызвано некорректными установками формата хранения данных, некорректной инициализацией указателя стека (проверьте таблицу векторов) или же использованием объектной библиотеки для традиционного процессора ARM (код ARM вместо кода Thumb). Функции данных библиотек могут вызываться из стартового кода. Пожалуйста, проверьте настройки компилятора и компоновщика и убедитесь, что используются библиотечные файлы Thumb и Thumb-2
Процессор не переходит в спящий режим после выполнения команды `WFE`	Выполнение команды `WFE` не всегда вызывает переход процессора в спящий режим. Если внутренний регистр события был установлен до вызова команды `WFE`, то она просто очистит этот регистр и выполнится как `NOP`. Поэтому команду `WFE` рекомендуется вызывать в цикле
Выполнение программы неожиданно останавливается	Если включена функция Sleep-On-Exit, то при возврате из обработчика исключения в режим потока процессор будет переходить в спящий режим даже при отсутствии команд `WFE`/`WFI`
Неожиданное поведение SEVONPEND	Бит SEVONPEND регистра SCR разрешает запрещённым прерываниям выводить процессор из спящего режима, вызванного командой `WFE`, но не `WFI`. Событие вывода из спящего режима генерируется только в момент перевода прерывания в состояние ожидания. Если на момент исполнения команды `WFE` бит признака отложенного прерывания был уже установлен, то появление нового запроса прерывания не вызовет генерации указанного события и, соответственно, не выведет процессор из спящего режима
Уровень приоритета прерывания работает не так, как ожидалось	В отличие от многих других процессоров, нулевое значение в процессоре Cortex-M3 соответствует максимально возможному уровню приоритета исключений с программируемым приоритетом. Чем больше значение уровня приоритета, тем ниже собственно приоритет. При программировании регистров уровня приоритета убедитесь, что вы заносите значения в реализованные (старшие) биты регистров. Большинство микроконтроллеров с ядром Cortex-M3 имеют 3-битные (8 уровней) или 4-битные (16 уровней) регистры уровня приоритета. При этом используются старшие биты регистров. Поэтому если вы попытаетесь записать в регистр значения вида 0x03, 0x07 и т.п., в нём окажется ноль

***Таблица Д.8.* Прочие возможные проблемы** *(продолжение)*

Ситуация	Возможные причины
Выполнение команды `SVC` вызывает генерацию исключения отказа	Процессор Cortex-M3 не поддерживает рекурсивную обработку исключений — исключение не может быть обработано до тех пор, пока его приоритет не окажется выше текущего приоритета. Как следствие, вы не можете использовать команду `SVC` в обработчиках исключений SVCall, Hard Fault и NMI, а также в обработчиках других исключений, приоритет которых выше или равен приоритету исключения SVCall
Параметры, передаваемые в исключение SVCall, искажаются	При передаче параметров в обработчики исключений, скажем SVCall, значения данных параметров (регистры R0…R3) необходимо извлекать из стекового фрейма, а не из регистрового банка. Это связано с тем, что до входа в обработчик SVCall может возникнуть другое исключение, которое будет обработано первым. Поскольку обработчики исключений могут изменять регистры R0…R3 и R12 (стандарт AAPCS не требует от Си-подпрограмм сохранять содержимое этих регистров), то при входе в обработчик SVCall содержимое указанных регистров следует считать неопределённым. Соответственно, для корректной передачи параметров необходимо использовать стек. Это потребует написания на ассемблере некоторой «надстройки», которая будет определять значение корректного указателя стека и передавать его в качестве указателя в обработчик, написанный на Си. Пример подобного кода можно найти в Главе 11 настоящей книги, а также в руководстве по применению AN179, выпущенному компанией ARM. Аналогичные меры предосторожности следует принимать при передаче данных из обработчика исключения в прерванную программу. Обработчик должен сохранить передаваемые данные в стековом фрейме. В противном случае, содержимое регистров будут перезаписано при восстановлении контекста
Исключение SYSTICK возникает при сброшенном бите TICKINT	Бит TICKINT регистра управления и состояния таймера SYSTICK используется для разрешения и запрещения исключения SYSTICK. Однако если это исключение уже находится в состоянии ожидания, то сброс бита TICKINT не предотвратит генерацию исключения. Чтобы гарантированно заблокировать генерацию исключения SYSTICK, необходимо сбросить не только бит TICKINT, но и соответствующий бит признака отложенного прерывания в регистре ICSR контроллера NVIC
Интерфейс JTAG заблокирован	Во многих микроконтроллерах на базе процессора Cortex-M3 интерфейс JTAG выведен на контакты, используемые портами ввода/вывода. Если сразу же после запуска программы эти выводы конфигурируются в качестве линий ввода/вывода, то вы не сможете войти в режим отладки или стереть флэш-память микроконтроллера

***Таблица Д.8.* Прочие возможные проблемы** *(продолжение)*

Ситуация	Возможные причины
Возникают неожиданные прерывания	Некоторые микроконтроллеры имеют буфер записи в мосте шины периферийных устройств. Поэтому возможна следующая ситуация: обработчик исключения сбрасывает прерывание, выполняя запись в устройство, производится выход из обработчика, после чего этот обработчик запускается повторно из-за того, что периферийное устройство не успело снять запрос прерывания. Данную проблему можно решить следующим образом: 1. Перед выходом из исключения выполнить в процедуре обработки прерывания пустую операцию записи в периферийное устройство. Однако при этом может увеличиться время выполнения обработчика. 2. Сбрасывать прерывание в начале процедуры обработки прерывания, чтобы к моменту выхода из обработчика оно гарантированно было бы сброшено. Однако данный способ не сработает, если длительность обработчика окажется меньше задержки, обусловленной буферированием записи (это можно определить только опытным путём).
Проблемы при использовании обычных прерываний в качестве программных	Некоторые программисты пытаются использовать незадействованные типы исключений, имеющиеся в контроллере NVIC, в качестве программных прерываний. Однако исключения внешних прерываний, так же как исключение PendSV, являются «неточными». То есть никто не гарантирует, что обработчик такого исключения будет запущен сразу же после перевода этого исключения в состояние ожидания. Чтобы удостовериться в запуске обработчика, можно воспользоваться методом циклического опроса

Неожиданное переключение в состояние ARM при выполнении команд `BLX` и `BX`	Если вы используете ассемблер GNU и вызываете функцию, описанную в другом файле, то следует убедиться, что данный идентификатор действительно является именем функции, т.е. объявлен как:
	`.text`
	`.global имя_функции`
	`.type имя_функции, %function`
	В противном случае, вызов функции или переход по указанной метке может привести к случайному переключению в состояние ARM.
	Кроме того, при вызове компоновщика необходимо указать ключи `-mcortex-m3 -mthumb`, иначе компоновщик может взять неверную версию Си-библиотеки

Неожиданное запрещение прерываний	В процессорах Cortex-M3 и ARM7TDMI по-разному реализован возврат из исключений. В процессоре ARM7TDMI, если прерывание было запрещено в обработчике, то при возврате из исключения оно снова разрешается за счёт восстановления содержимого регистра CPSR (бит I). В случае же процессора Cortex-M3, если вы запретите прерывания, установив регистр PRIMASK (используя команду `CPSID I` или функцию `__disable_irq()`), то вам необходимо будет повторно разрешить эти прерывания до выхода из обработчика. В противном случае, после возврата из исключения регистр PRIMASK останется установленным, что вызовет блокировку всех прерываний

***Таблица Д.8.* Прочие возможные проблемы** *(продолжение)*

Ситуация	Возможные причины
Неожиданные обращения по невыровненным адресам	Процессор Cortex-M3 поддерживает невыровненные пересылки при выполнении операций загрузки/сохранения одного регистра. Компиляторы языка Си, как правило, генерируют невыровненные пересылки только при обращении к элементам упакованных структур или при ручном манипулировании указателями. Поэтому данная проблема большей частью касается программирования на ассемблере, когда пользователь может случайно сформировать невыровненную пересылку, даже не заметив этого. Рассмотрим, к примеру, чтение 32-битного регистра периферийного устройства. При использовании других процессоров ARM младшие два бита адреса будут проигнорированы (поскольку мост AHB-APB принудительно сбрасывает эти биты в 0). Если тот же код будет выполняться процессором Cortex-M3, то невыровненная пересылка будет разбита на несколько выровненных, а результат операции может оказаться совершенно другим. Этот же вопрос может оказаться актуальным при переносе ПО с процессора Cortex-M3 на другой процессор ARM, не поддерживающий невыровненные пересылки. Обнаружить использование программой невыровненных пересылок очень просто. Для этого достаточно установить бит UNALIGN_TRP регистра CCR контроллера NVIC, после чего любая невыровненная пересылка будет вызывать исключение Usage Fault, что позволит отследить и убрать из программы все нежелательные обращения по невыровненным адресам

ПРИЛОЖЕНИЕ E
ПРИМЕР СЦЕНАРИЯ КОМПОНОВЩИКА ДЛЯ ПАКЕТА SOURCERY G++ КОМПАНИИ CODESOURCERY

E.1. Сценарий компоновщика для Cortex-M3

Приведённый ниже сценарий компоновщика[1)], предназначенный для процессора Cortex™-M3, написан на основе типового сценария, который поставляется с пакетом разработки Sourcery G++ Lite (файл generic-m.ld). Данный сценарий предполагает использование стартового кода и таблицы векторов CS3 (CodeSourcery Common Start-up Code Sequence), т.е. рассчитан на конкретный инструментарий. При работе в других средах разработки, основанных на Си-компиляторе GNU, следует обращаться к документации и примерам, входящим в состав используемого пакета.

```
/* Linker script for generic-m
*
* Version:Sourcery G++ Lite 2009q1-161
* BugURL:https://support.codesourcery.com/GNUToolchain/
*
* Copyright 2007, 2008 CodeSourcery, Inc.
*
* The authors hereby grant permission to use, copy, modify, distribute,
* and license this software and its documentation for any purpose, provided
* that existing copyright notices are retained in all copies and that this
* notice is included verbatim in any distributions. No written agreement,
* license, or royalty fee is required for any of the authorized uses.
* Modifications to this software may be copyrighted by their authors
* and need not follow the licensing terms described here, provided that
* the new terms are clearly indicated on the first page of each file where
* they apply.
* */
OUTPUT_FORMAT («elf32-littlearm», «elf32-bigarm», «elf32-littlearm»)
ENTRY(_start)
SEARCH_DIR(.)
GROUP(-lgcc -lc -lcs3 -lcs3unhosted -lcs3micro)
MEMORY
{
/* ROM is a readable (r), executable region (x)          */
```

[1)] Файл cortexm3.ld используется в Примере 5 и Примере 6 из Главы 19.

```
rom (rx) : ORIGIN = 0, LENGTH = 32k

/* RAM is a readable (r), writable (w) and              */
/* executable region (x)                                */
ram (rwx) : ORIGIN = 0x20000000, LENGTH = 16k
}
/* These force the linker to search for particular symbols from
* the start of the link process and thus ensure the user's
* overrides are picked up
*/
EXTERN(__cs3_reset_generic_m)
INCLUDE micro-names.inc
EXTERN(__cs3_interrupt_vector_micro)
EXTERN(__cs3_start_c main __cs3_stack __cs3_heap_end)

PROVIDE(__cs3_heap_start = _end);
PROVIDE(__cs3_heap_end = __cs3_region_start_ram + __cs3_region_size_ram);
PROVIDE(__cs3_region_num = (__cs3_regions_end - __cs3_regions) / 20);
PROVIDE(__cs3_stack = __cs3_region_start_ram + __cs3_region_size_ram);

SECTIONS
{
  .text :
  {
    CREATE_OBJECT_SYMBOLS
    __cs3_region_start_rom = .;
    *(.cs3.region-head.rom)
    ASSERT (. == __cs3_region_start_rom, «.cs3.region-head.rom not permitted»);

    /* Vector table */
    __cs3_interrupt_vector = __cs3_interrupt_vector_micro;
    *(.cs3.interrupt_vector) /* vector table */
    /* Make sure we pulled in an interrupt vector. */
    ASSERT (. != __cs3_interrupt_vector_micro, «No interrupt vector»);

    /* Map CS3 vector symbols to handler names in C */
    __cs3_reset = Reset_Handler;
    __cs3_isr_nmi = NMI_Handler;
    __cs3_isr_hard_fault = HardFault_Handler;

    *(.text .text.* .gnu.linkonce.t.*)
    *(.plt)
    *(.gnu.warning)

    *(.glue_7t) *(.glue_7) *(.vfp11_veneer)
    *(.ARM.extab* .gnu.linkonce.armextab.*)
    *(.gcc_except_table)
  } >rom
  .eh_frame_hdr : ALIGN (4)
  {
    KEEP (*(.eh_frame_hdr))
```

```
} >rom
.eh_frame : ALIGN (4)
{
  KEEP (*(.eh_frame))
} >rom
/* .ARM.exidx is sorted, so has to go in its own output section. */
__exidx_start = .;
.ARM.exidx :
{
  *(.ARM.exidx* .gnu.linkonce.armexidx.*)
} >rom
__exidx_end = .;
.rodata : ALIGN (4)
{
  *(.rodata .rodata.* .gnu.linkonce.r.*)

  . = ALIGN(4);
  KEEP(*(.init))

  . = ALIGN(4);
  __preinit_array_start = .;
  KEEP (*(.preinit_array))
  __preinit_array_end = .;

  . = ALIGN(4);
  __init_array_start = .;
  KEEP (*(SORT(.init_array.*)))
  KEEP (*(.init_array))
  __init_array_end = .;

  . = ALIGN(4);
  KEEP(*(.fini))

  . = ALIGN(4);
  __fini_array_start = .;
  KEEP (*(.fini_array))
  KEEP (*(SORT(.fini_array.*)))
  __fini_array_end = .;

  . = ALIGN(0x4);
  KEEP (*crtbegin.o(.ctors))
  KEEP (*(EXCLUDE_FILE (*crtend.o) .ctors))
  KEEP (*(SORT(.ctors.*)))
  KEEP (*crtend.o(.ctors))

  . = ALIGN(0x4);
  KEEP (*crtbegin.o(.dtors))
  KEEP (*(EXCLUDE_FILE (*crtend.o) .dtors))
  KEEP (*(SORT(.dtors.*)))
  KEEP (*crtend.o(.dtors))
```

```
  /* Add debug information
  . = ALIGN(4);
  __my_debug_regions = .;
  LONG (__cs3_heap_start)
  LONG (__cs3_heap_end)
  LONG (__cs3_stack) */

  . = ALIGN(4);
  __cs3_regions = .;
  LONG (0)
  LONG (__cs3_region_init_ram)
  LONG (__cs3_region_start_ram)
  LONG (__cs3_region_init_size_ram)
  LONG (__cs3_region_zero_size_ram)
  __cs3_regions_end = .;

  . = ALIGN (8);
  *(.rom)
  *(.rom.b)
  _etext = .;
} >rom

.data : ALIGN (8)
{
  __cs3_region_start_ram = .;
  _data = .;
  *(.cs3.region-head.ram)
  KEEP(*(.jcr))
  *(.got.plt) *(.got)
  *(.shdata)
  *(.data .data.* .gnu.linkonce.d.*)
  . = ALIGN (8);
  *(.ram)
  _edata = .;
} >ram AT>rom
.bss :
{
  _bss = .;
  *(.shbss)
  *(.bss .bss.* .gnu.linkonce.b.*)
  *(COMMON)
  . = ALIGN (8);
  *(.ram.b)
  _ebss = .;
  _end = .;
  __end = .;
} >ram AT>rom
__cs3_region_init_ram = LOADADDR (.data);
__cs3_region_init_size_ram = _edata - ADDR (.data);
__cs3_region_zero_size_ram = _end - _edata;
__cs3_region_size_ram = LENGTH(ram);
```

```
  .stab            0 (NOLOAD) : { *(.stab) }
  .stabstr         0 (NOLOAD) : { *(.stabstr) }
  /* DWARF debug sections.
  * Symbols in the DWARF debugging sections are relative to the beginning
  * of the section so we begin them at 0. */
  /* DWARF 1 */
  .debug           0 : { *(.debug) }
  .line            0 : { *(.line) }
  /* GNU DWARF 1 extensions */
  .debug_srcinfo   0 : { *(.debug_srcinfo) }
  .debug_sfnames   0 : { *(.debug_sfnames) }
  /* DWARF 1.1 and DWARF 2 */
  .debug_aranges   0 : { *(.debug_aranges) }
  .debug_pubnames  0 : { *(.debug_pubnames) }
  /* DWARF 2 */
  .debug_info      0 : { *(.debug_info .gnu.linkonce.wi.*) }
  .debug_abbrev    0 : { *(.debug_abbrev) }
  .debug_line      0 : { *(.debug_line) }
  .debug_frame     0 : { *(.debug_frame) }
  .debug_str       0 : { *(.debug_str) }
  .debug_loc       0 : { *(.debug_loc) }
  .debug_macinfo   0 : { *(.debug_macinfo) }
  /* DWARF 2.1 */
  .debug_ranges    0 : { *(.debug_ranges) }
  /* SGI/MIPS DWARF 2 extensions */
  .debug_weaknames    0 : { *(.debug_weaknames) }
  .debug_funcnames    0 : { *(.debug_funcnames) }
  .debug_typenames    0 : { *(.debug_typenames) }
  .debug_varnames     0 : { *(.debug_varnames) }
  .note.gnu.arm.ident 0 : { KEEP (*(.note.gnu.arm.ident)) }
  .ARM.attributes     0 : { KEEP (*(.ARM.attributes)) }
  /DISCARD/ : { *(.note.GNU-stack)
}
```

ПРИЛОЖЕНИЕ Ж
ФУНКЦИИ ДОСТУПА К ЯДРУ СТАНДАРТА CMSIS

Стандарт программного интерфейса микроконтроллеров с ядром Cortex™ (CMSIS) определяет следующие базовые функции:

- функции доступа к периферии ядра;
- встроенные (intrinsic) функции.

В данном приложении приводятся основные сведения об этих функциях. Некоторые из функций CMSIS используют стандартные типы данных, определённые в файле `stdint.h` (**Табл. Ж.1**).

Таблица Ж.1. **Стандартные типы данных, используемые в функциях CMSIS**

Тип	Описание
uint32_t	32-битное целое без знака
uint16_t	16-битное целое без знака
uint8_t	8-битное целое без знака

Ж.1. Нумерация исключений и прерываний

Ряд функций CMSIS, работающих с прерываниями, в качестве входного параметра используют порядковые номера прерываний. Причём, эти номера отличаются от номеров регистров состояния прерываний IPSRx. В стандарте CMSIS прерывания периферийных устройств нумеруются, начиная с нуля, а для обозначения системных исключений используются отрицательные значения. (**Табл. Ж.2**).

Таблица Ж.2. **Нумерация исключений и прерываний**

Номер прерывания в CMSIS	Номер исключения в процессоре (IPSR)	Исключение	Имя исключения (элемент перечисления IRQn_Type)	Имя обработчика исключения
—	—	Сброс	—	Reset_Handler
–14	2	NMI	NonMaskableInt_IRQn	NMI_Handler
–13	3	Hard Fault	—	HardFault_Handler
–12	4	MemManage Fault	MemoryManagement_IRQn	MemManage_Handler

Таблица Ж.2. **Нумерация исключений и прерываний** *(продолжение)*

Номер прерывания в CMSIS	Номер исключения в процессоре (IPSR)	Исключение	Имя исключения (элемент перечисления IRQn_Type)	Имя обработчика исключения
–11	5	Bus Fault	BusFault_IRQn	BusFault_Handler
–10	6	Usage Fault	UsageFault_IRQn	UsageFault_Handler
–5	11	SVCall	SVCall_IRQn	SVC_Handler
–4	12	Debug monitor	DebugMonitor_IRQn	DebugMon_Handler
–2	14	PendSV	PendSV_IRQn	PendSV_Handler
–1	15	SysTick	SysTick_IRQn	SysTick_Handler
0	16	Прерывание 0 периферийного устройства	<Зависит от микроконтроллера>	<Зависит от микроконтроллера>
1	17	Прерывание 1 периферийного устройства	<Зависит от микроконтроллера>	<Зависит от микроконтроллера>
2	18	Прерывание 2 периферийного устройства	<Зависит от микроконтроллера>	<Зависит от микроконтроллера>
…	…	…	<Зависит от микроконтроллера>	<Зависит от микроконтроллера>

Ж.2. Функции доступа к контроллеру NVIC

Следующие функции позволяют использовать различные возможности контроллера NVIC.

Имя функции	**void NVIC_SetPriorityGrouping(uint32_t PriorityGroup)**
Описание	Изменяет настройки группирования приоритета в контроллере прерываний NVIC (функция недоступна в процессорах Cortex-M0/M1)
Параметры	PriorityGroup — значение поля группирования приоритетов (PRIGROUP)
Возвращаемое значение	Нет

Имя функции	**uint32_t NVIC_GetPriorityGrouping(void)**
Описание	Читает настройки группирования приоритета из контроллера прерываний NVIC (функция недоступна в процессорах Cortex-M0/M1)
Параметры	Нет
Возвращаемое значение	Содержимое поля группирования приоритетов (PRIGROUP)

Имя функции	**void NVIC_EnableIRQ(IRQn_Type IRQn)**
Описание	Разрешает прерывание в контроллере NVIC
Параметры	IRQn_Type IRQn — положительный номер прерывания. Системные исключения (отрицательные значения) не поддерживаются
Возвращаемое значение	Нет

Имя функции	**void NVIC_DisableIRQ(IRQn_Type IRQn)**
Описание	Запрещает прерывание в контроллере NVIC
Параметры	IRQn_Type IRQn — положительный номер прерывания. Системные исключения (отрицательные значения) не поддерживаются
Возвращаемое значение	Нет

Имя функции	**uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)**
Описание	Считывает бит отложенности внешнего прерывания
Параметры	IRQn_Type IRQn — номер прерывания. Системные исключения (отрицательные значения) не поддерживаются
Возвращаемое значение	1 — если прерывание отложено, 0 — в противном случае

Имя функции	**void NVIC_SetPendingIRQ(IRQn_Type IRQn)**
Описание	Устанавливает бит отложенности внешнего прерывания
Параметры	IRQn_Type IRQn — номер прерывания. Системные исключения (отрицательные значения) не поддерживаются
Возвращаемое значение	Нет

Имя функции	**void NVIC_ClearPendingIRQ(IRQn_Type IRQn)**
Описание	Сбрасывает бит отложенности внешнего прерывания
Параметры	IRQn_Type IRQn — номер прерывания. Системные исключения (отрицательные значения) не поддерживаются
Возвращаемое значение	Нет

Имя функции	**uint32_t NVIC_GetActive(IRQn_Type IRQn)**
Описание	Считывает бит активности внешнего прерывания (функция недоступна в процессорах Cortex-M0/M1)
Параметры	IRQn_Type IRQn — номер прерывания. Системные исключения (отрицательные значения) не поддерживаются
Возвращаемое значение	1 — если прерывание активно, 0 — в противном случае

Имя функции	**void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)**
Описание	Устанавливает приоритет прерывания или системного исключения с программируемым уровнем приоритета
Параметры	IRQn_Type IRQn — номер прерывания. uint32_t priority — приоритет прерывания. Функция автоматически сдвигает полученное значение влево для его размещения в реализованных битах регистра приоритета
Возвращаемое значение	Нет

Имя функции	**uint32_t NVIC_GetPriority(IRQn_Type IRQn)**
Описание	Считывает приоритет прерывания или системного исключения с программируемым уровнем приоритета
Параметры	IRQn_Type IRQn — номер прерывания
Возвращаемое значение	Приоритет прерывания. Функция автоматически сдвигает считанное значение вправо, чтобы убрать нереализованные биты регистра приоритета

Имя функции	**uint32_t NVIC_EncodePriority (uint32_t PriorityGroup, uint32_t PreemptPriority, uint32_t SubPriority)**
Описание	Формирует значение приоритета прерывания из значений параметра группирования приоритетов, приоритета группы и субприоритета. В случае конфликта между параметром группирования приоритетов и доступными битами приоритета (__NVIC_PRIO_BITS) используется минимально возможное значение параметра группирования приоритетов (функция недоступна в процессорах Cortex-M0/M1)
Параметры	uint32_t PriorityGroup — параметр группирования приоритетов; uint32_t PreemptPriority — приоритет группы (начиная с 0); uint32_t SubPriority — субприоритет (начиная с 0)
Возвращаемое значение	Приоритет прерывания

Имя функции	**void NVIC_DecodePriority (uint32_t Priority, uint32_t PriorityGroup, uint32_t* pPreemptPriority, uint32_t* pSubPriority)**
Описание	Декодирует приоритет прерывания, извлекая из его значения приоритет группы и субприоритет с учётом заданного значения параметра группирования приоритетов. В случае конфликта между параметром группирования приоритета и доступными битами приоритета (__NVIC_PRIO_BITS) используется минимально возможное значение параметра группирования приоритетов (функция недоступна в процессорах Cortex-M0/M1)
Параметры	uint32_t Priority — приоритет прерывания; uint32_t PriorityGroup — параметр группирования приоритетов; uint32_t PreemptPriority — приоритет группы (начиная с 0); uint32_t SubPriority — субприоритет (начиная с 0)
Возвращаемое значение	Нет

Ж.3. Функции для управления системой и системным таймером

Следующие функции предназначены для конфигурирования системы.

Имя функции	**void SystemInit (void)**
Описание	Инициализирует систему
Параметры	Нет
Возвращаемое значение	Нет

Имя функции	**void SystemCoreClockUpdate(void)**
Описание	Обновляет значение переменной `SystemCoreClock`. Эта функция должна вызываться после каждого изменения тактовой частоты процессора. Данная функция появилась в версии 1.30 стандарта CMSIS. В более ранних версиях стандарта указанная функция отсутствовала, а для хранения значения тактовой частоты использовалась глобальная переменная `SystemFrequency`
Параметры	Нет
Возвращаемое значение	Нет

Имя функции	**void NVIC_SystemReset(void)**
Описание	Инициирует запрос сброса системы
Параметры	Нет
Возвращаемое значение	Нет

Имя функции	**uint32_t SysTick_Config(uint32_t ticks)**
Описание	Инициализирует и запускает таймер SYSTICK, а также разрешает соответствующее исключение. Функция программирует системный таймер таким образом, чтобы он генерировал исключение SYSTICK каждые «ticks» тактов тактового сигнала ядра
Параметры	uint32_t ticks — число тактов между двумя последовательными прерываниями
Возвращаемое значение	Нет

Ж.4. Функции доступа к регистрам ядра

Следующие функции предназначены для доступа к регистрам специальных функций процессора.

Имя функции	Описание
`uint32_t __get_MSP(void)`	Читает значение MSP
`void __set_MSP(uint32_t topOfMainStack)`	Изменяет значение MSP
`uint32_t __get_PSP(void)`	Читает значение PSP
`void __set_PSP(uint32_t topOfProcStack)`	Изменяет значение PSP
`uint32_t __get_BASEPRI(void)`	Читает значение BASEPRI
`void __set_BASEPRI(uint32_t basePri)`	Изменяет значение BASEPRI
`uint32_t __get_PRIMASK(void)`	Читает значение PRIMASK
`void __set_PRIMASK(uint32_t priMask)`	Изменяет значение PRIMASK
`uint32_t __get_FAULTMASK(void)`	Читает значение FAULTMASK
`void __set_FAULTMASK(uint32_t faultMask)`	Изменяет значение FAULTMASK
`uint32_t __get_CONTROL(void)`	Читает значение CONTROL
`void __set_CONTROL(uint32_t control)`	Изменяет значение CONTROL

Ж.5. Встроенные функции CMSIS

Стандартом CMSIS предусмотрен ряд встроенных функций, предназначенных для вызова команд процессора, которые не могут быть сгенерированы стандартным конструкциями языка Си. Функции `__enable_fault_irq` и `__disable_fault_irq` недоступны для процессоров Cortex-M0/M1.

Функции для вызова команд управления системой

Имя функции	Команда	Описание
`void __WFI(void)`	`WFI`	Ожидание прерывания (в спящем режиме)
`void __WFE(void)`	`WFE`	Ожидание события (в спящем режиме)
`void __SEV(void)`	`SEV`	Генерация события
`void __enable_irq(void)`	`CPSIE i`	Разрешение прерываний (очистка PRIMASK)
`void __disable_irq(void)`	`CPSID i`	Запрещение прерываний (установка PRIMASK)
`void __enable_fault_irq(void)`	`CPSIE f`	Разрешение прерываний (очистка FAULTMASK)
`void __disable_fault_irq(void)`	`CPSID f`	Запрещение прерываний (установка FAULTMASK)
`void __NOP(void)`	`NOP`	Нет операции
`void __ISB(void)`	`ISB`	Барьер синхронизации команд
`void __DSB(void)`	`DSB`	Барьер синхронизации данных
`void __DMB(void)`	`DMB`	Барьер памяти данных

В следующей таблице приведены функции для вызова команд монопольного доступа к данным. Эти функции недоступны в процессорах Cortex-M0/M1.

Имя функции	Команда	Описание
`uint8_t __LDREXB(uint8_t *addr)`	`LDREXB`	Монопольная загрузка байта в регистр
`uint16_t __LDREXH(uint16_t *addr)`	`LDREXH`	Монопольная загрузка полуслова в регистр
`uint32_t __LDREXW(uint32_t *addr)`	`LDREX`	Монопольная загрузка слова в регистр
`uint32_t __STREXB(uint8_t value, uint8_t *addr)`	`STREXB`	Монопольное сохранение байта. Возвращает статус операции записи (успех = 0, отказ = 1)
`uint32_t __STREXH(uint16_t value, uint8_t *addr)`	`STREXH`	Монопольное сохранение полуслова. Возвращает статус операции записи (успех = 0, отказ = 1)
`uint32_t __STREXW(uint32_t value, uint8_t *addr)`	`STREX`	Монопольное сохранение слова. Возвращает статус операции записи (успех = 0, отказ = 1)
`void __CLREX(void)`	`CLREX`	Сброс признака монопольного доступа, установленного операцией монопольного чтения

В следующей таблице приведены функции для вызова специфических команд обработки данных. Функция `__RBIT` недоступна в процессорах Cortex-M0/M1.

Имя функции	Команда	Описание
`uint32_t __REV(uint32_t value)`	`REV`	Изменение порядка байтов в слове
`uint32_t __REV16(uint32_t value)`	`REV16`	Изменение порядка байтов в каждом полуслове
`uint32_t __REVSH(uint32_t value)`	`REVSH`	Изменение порядка байтов в младшем полуслове и расширение результата до 32 бит
`uint32_t __RBIT(uint32_t value)`	`RBIT`	Изменение порядка битов в слове

Ж.6. Функции вывода отладочных сообщений

В стандарте CMSIS также определена одна функция, позволяющая выводить отладочные сообщения с использованием модуля ITM.

Имя функции	**uint32_t ITM_putchar(uint32_t chr)**
Описание	Выводит символ через 0-й канал вывода модуля ITM. При отсутствии подключённого отладчика функция сразу же завершается. Если отладчик подключён и трассировка разрешена, то функция выводит символ в интерфейс ITM и ожидает его передачи
Параметры	uint32_t chr — передаваемый символ
Возвращаемое значение	Переданный символ `chr`

ПРИЛОЖЕНИЕ 3
СОЕДИНИТЕЛИ ДЛЯ ПОДКЛЮЧЕНИЯ ОТЛАДОЧНЫХ СРЕДСТВ

3.1. Общие сведения

В этом приложении рассматриваются несколько разновидностей соединителей, наиболее часто применяющихся для подключения отладочных средств. Большинство средств разработки для процессоров ARM используют, как минимум, одну из предложенных цоколёвок. При проектировании печатной платы рекомендуется использовать стандартную разводку сигналов отладчика, чтобы избежать проблем с его подключением.

3.2. Универсальный 20-контактный разъём

На новых печатных платах с микроконтроллерами ARM устанавливаются 20-контактные разъёмы с шагом выводов 0.05 дюйма (Samtec FTSH-120), используемые как для отладки, так и для трассировки.

Примечание. Сигналы, наименования которых на последующих рисунках выделены серым цветом, не реализованы в ядре Cortex™-M3.

Универсальный 20-контактный разъём поддерживает протоколы отладки JTAG и Serial-Wire (см. **Рис. 3.1** и **3.2**). При использовании протокола Serial-Wire линия TDO может задействоваться в качестве выхода модуля Serial-Wire Viewer (SWV) для вывода трассировочной информации. На этот разъём также выведен 4-битный порт трассировки, используемый в тех случаях, когда для передачи

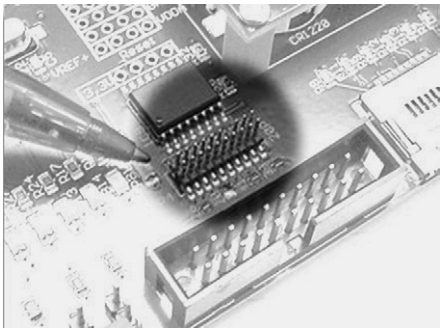


Рис. 3.1. *20-контактный универсальный разъём.*

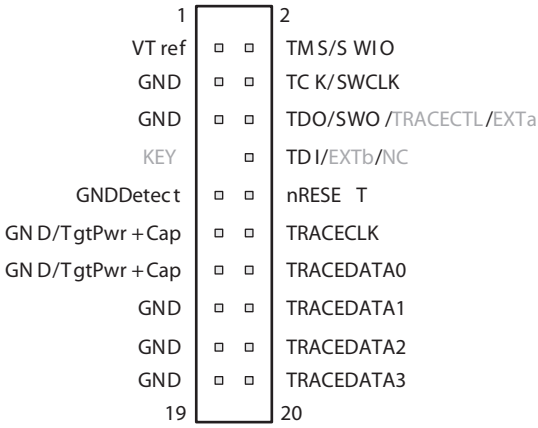


Рис. 3.2. *Цоколёвка 20-контактного универсального разъёма.*

трассировочных данных требуется канал с повышенной пропускной способностью (например, при включении модуля трассировки ETM).

Соединитель FTSH-120 имеет меньшие размеры, чем традиционные соединители IDC, и рекомендуется для применения в новых разработках. Данный соединитель, в частности, установлен на демонстрационной плате MCBSTM32E компании Keil.

3.3. 10-контактный разъём

Если в устройстве нет модуля ETM, то для подключения отладчика можно использовать 10-контактный разъём с шагом 0.05 дюйма, который имеет ещё меньшие размеры. Как и 20-контактный универсальный разъём, данный соединитель поддерживает протоколы JTAG и Serial-Wire (**Рис. 3.3** и **3.4**).

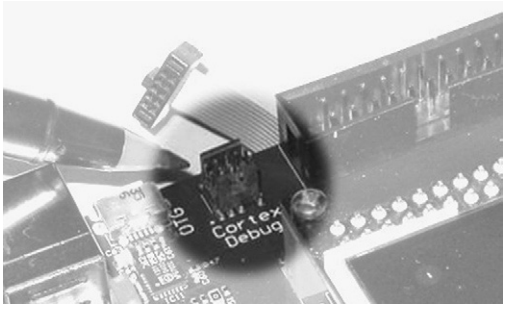


Рис. 3.3. *10-контактный разъём.*

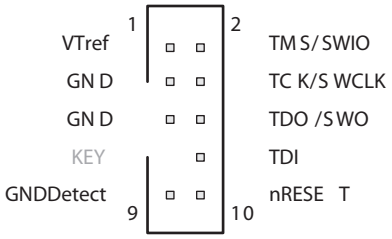


Рис. 3.4. *Цоколёвка 10-контактного разъёма.*

3.4. Традиционный 20-контактный разъём IDC

Стандартным разъёмом для подключения отладочных средств на отладочных платах компании ARM является 20-контактный разъём IDC (**Рис. 3.5**). Этот разъ-

ём поддерживает отладку по протоколам JTAG и Serial-Wire (SWIO и SWCLK), а также вывод данных модуля SWV. Вывод nICEDETECT позволяет отлаживаемому микроконтроллеру определять подключение отладчика. При отсутствии отладчика этот вывод «подтянут» к питанию, а при подключении отладчика данный вывод «подтягивается» к земле. Эта возможность используется в некоторых отладочных платах, поддерживающих несколько конфигураций JTAG. Сигнал nSRST является опциональным. Поскольку для сброса системы с процессором Cortex-M3 отладчик может задействовать контроллер NVIC, в микроконтроллерах этот сигнал, как правило, не реализуют.

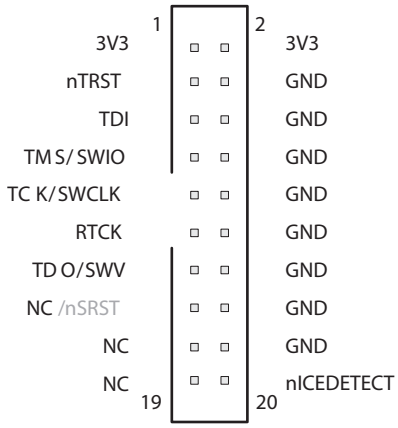


Рис. 3.5. *Цоколёвка 20-контактного разъёма IDC.*

3.5. Традиционный 32-контактный разъём MICTOR

При необходимости реализации порта трассировки в некоторых системах на базе процессоров ARM используется разъём MICTOR (например, для трассировки команд посредством модуля ETM; см. **Рис. 3.6**). Этот разъём также поддерживает протоколы отладки JTAG/SWD. Параллельно разъёму MICTOR может быть подключён 20-контактный разъём IDC (одновременно используется только один из них).

Как правило, микроконтроллеры с процессором Cortex-M3 имеют шину трассировки разрядностью всего 4 бита, поэтому большинство выводов разъёма, предназначенных для передачи трассировочной информации, не задействуются. Разъём MICTOR используется, главным образом, со старшими процессорами семейства Cortex (Cortex-A8/A9, Cortex-R4), а также в многопроцессорных системах, требующих порт трассировки большей разрядности. В этом случае также задействуются и ряд других контактов разъёма. Для систем на базе процессора Cortex-M3 рекомендуется использовать универсальный 20-контактный разъём, позволяющий выполнять отладку и трассировку посредством модуля ETM.

1	NC	NC	2
3	NC	NC	4
5	GN D	TRACECLK	6
7	Pulldown	Pulldow n	8
9	NC/nSRST	Pulldow n	10
11	TD O/SW V	Pullup (Vref)	12
13	RTCK	VSuppl y	14
15	TC K/SWCLK	0/TRACEDATA[7]	16
17	TM S/ SWIO	0/TRACEDATA[6]	18
19	TD I	0/TRACEDATA[5]	20
21	nTRST	0/TRACEDATA[4]	22
23	0/TRACEDATA[15]	TRACEDATA[3]	24
25	0/TRACEDATA[14]	TRACEDATA[2]	26
27	0/TRACEDATA[13]	TRACEDATA[1]	28
29	0/TRACEDATA[12]	0	30
31	0/TRACEDATA[11]	0	32
33	0/TRACEDATA[10]	1	34
35	0/TRACEDATA[9]	0/TRACECTRL	36
37	0/TRACEDATA[8]	TRACEDATA[0]	38

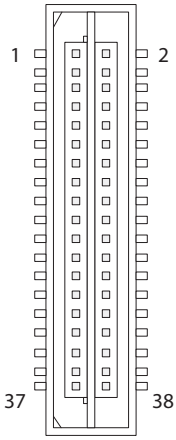


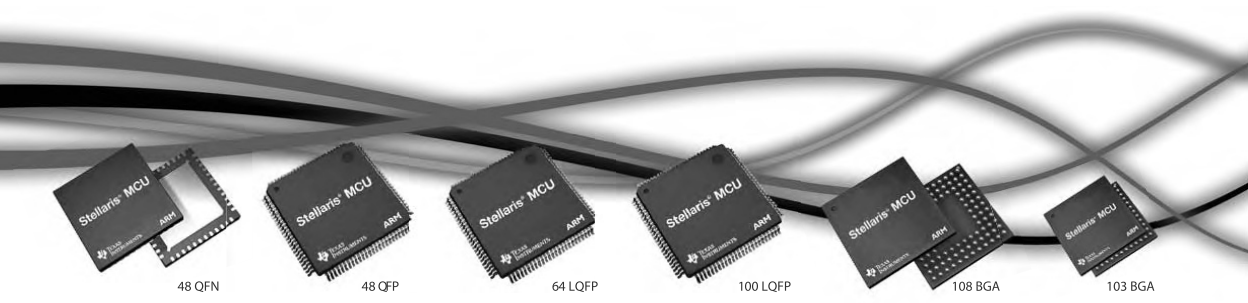
Рис. 3.6. *Цоколёвка 38-контактного разъёма MICTOR.*

ПРИЛОЖЕНИЕ И

СЕМЕЙСТВО МИКРОКОНТРОЛЛЕРОВ STELLARIS®

Материал предоставлен
компанией «КОМПЭЛ»
и российским представительством
компании «Texas Instruments»

Texas Instruments

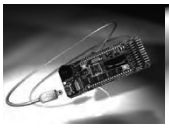


48 QFN 48 QFP 64 LQFP 100 LQFP 108 BGA 103 BGA

И.1. Приступая к работе

Разработка продукции

Компания Texas Instruments оказывает разнообразную поддержку разработчикам, благодаря которой вы сможете легко и быстро выводить свою продукцию на рынок. Компактные, универсальные, с широкими коммуникационными возможностями — именно так можно охарактеризовать предлагаемые оценочные наборы, которые являются недорогим и эффективным решением, позволяющим ознакомиться с нашими микроконтроллерами и быстро приступить к реализации своего проекта (**www.ti.com/stellaris_evkits**).



Оценочный набор Stellaris LM3S811

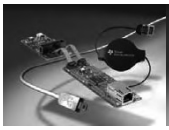
Оценочный набор Stellaris LM3S1968

Оценочный набор Stellaris LM3S2965

Оценочный набор Stellaris LM3S3748

Оценочный набор Stellaris LM3S6965

Оценочный набор Stellaris LM3S8962



Оценочный набор Stellaris LM3S9B92

Оценочный набор Stellaris LM3S9B90

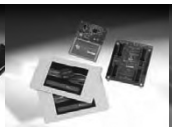
Набор для разработки Stellaris LM3S9B96

Stellaris DK-LM3S9B96-FPGA

Оценочный набор Stellaris EVALBOT + книга «Micrium µC/OS-III»

Оценочный набор Stellaris LM3S9B92 EVALBOT

Набор для разработки Stellaris Simplicity



Набор для разработки Stellaris RSID

Набор для разработки Stellaris ZigBee

Оценочный набор Stellaris® LM4F232

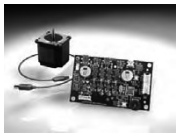
Наши наборы референс-дизайнов позволяют ускорить создание продукции, поскольку содержат готовое аппаратное и программное обеспечение, а также всю необходимую документацию, включая файлы проекта (**www.ti.com/stellaris_rdkits**).

Набор референс-дизайна интеллектуального дисплейного модуля Stellaris — одноплатный компьютер

Набор референс-дизайна интеллектуального дисплейного модуля Stellaris

Набор референс-дизайна интелектуального дисплейного модуля Stellaris с 3.5 дюймовым экраном

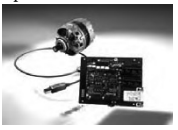
Набор референс-дизайна Stellaris для преобразования последовательных протоколов в Ethernet



Набор референс-дизайна Stellaris для управления шаговым двигателем

Набор референс-дизайна Stellaris для управления бесщёточным двигателем постоянного тока

Набор референс-дизайна Stellaris для управления щёточным двигателем постоянного тока с интерфейсом CAN



Набор референс-дизайна Stellaris для управления асинхронным двигателем переменного тока

И.2. Семейство микроконтроллеров Stellaris®

Stellaris представляет собой передовое семейство надёжных микроконтроллеров для систем реального времени, в основе которых лежат процессоры новой серии Cortex-M™ компании ARM®. Удостоенные многочисленных наград 32-битные микроконтроллеры Stellaris — это современные многофункциональные системы на кристалле, позволяющие реализовывать многозадачные приложения реального времени. Сложные приложения, которые были не по зубам микроконтроллерам предыдущих поколений, теперь можно легко реализовать с помощью мощных, экономически выгодных и простых в программировании микроконтроллеров семейства Stellaris.

Семейство микроконтроллеров Stellaris предназначено для применения в критичных к стоимости приложениях, требующих значительной вычислительной мощности и широких коммуникационных возможностей. К таким приложениям относятся системы энергосбережения, системы управления электроприводом, системы мониторинга (дистанционное управление, пожарная сигнализация, системы безопасности и т.д.), системы отопления, вентиляции, кондиционирования воздуха и управления зданием, системы контроля и преобразования энергии, сетевое оборудование и коммутаторы, системы промышленной автоматики, электронные кассовые терминалы, испытательное и измерительное оборудование, медицинские приборы и игровые устройства.

Семейство Stellaris — это микроконтроллеры будущего!

Почему выбрана ARM-архитектура?

- При начальной цене в 1 доллар за микроконтроллер на базе технологии ARM семейство Stellaris обеспечивает унификацию, которая исключает в дальнейшем модернизацию архитектуры или обновление программных средств.
- «Экосистема» программных инструментов и решений от ARM является крупнейшей в мире: каждый год на рынок встраиваемых систем поставляется более 5 млрд ARM-процессоров.
- Архитектура ARM Cortex обеспечивает разработчикам доступ к совместимому по набору команд семейству микроконтроллеров с ценой от 1 доллара и тактовой частотой до 1 ГГц.

Почему выбрано ядро Cortex-M3?

Cortex-M3 — версия процессорного ядра с набором команд ARMv7, которая имеет следующие особенности:

- набор команд оптимизирован для однотактного обращения к флэш-памяти;
- детерминированная и быстрая обработка прерываний: время реакции на прерывание равно 12 тактам или 6 тактам при вложенных прерываниях;
- три дежурных режима со стробированием тактового сигнала для снижения энергопотребления;
- однотактное умножение и деление на аппаратном уровне;
- производительность процессора 1.25 DMIPS/МГц (выше, чем у ARM7 и ARM9);
- дополнительные возможности отладки, включая установку точек останова и флэш-коррекцию программ;.

Преимущества по сравнению с приложениями для ARM7:

- требуется примерно половина объёма флэш-памяти (кодового пространства);
- в 2…4 раза быстрее в приложениях микроконтроллерного управления;
- не требуется кода ассемблера.

Почему следует выбирать семейство Stellaris?

Разработанное для важных микроконтроллерных приложений, семейство Stellaris позволяет войти в наиболее быстро растущую в отрасли «экосистему» микроконтроллеров с совместимым кодом стоимостью от 1 доллара и тактовой частотой до 1 ГГц.

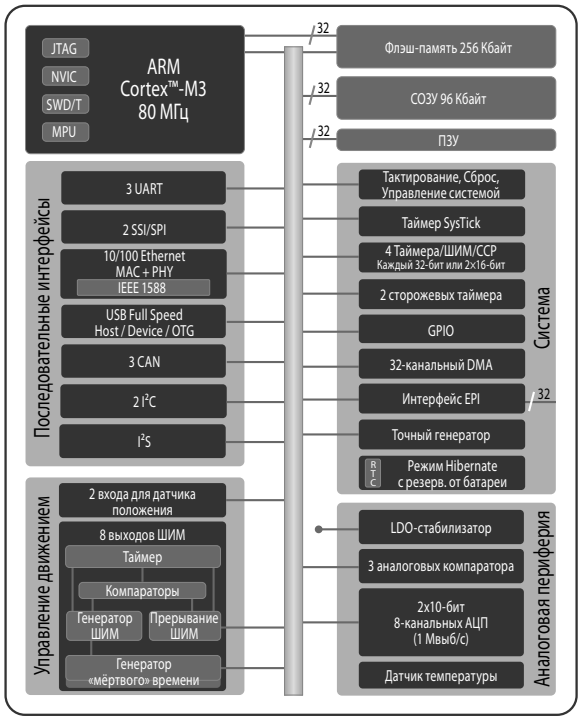
- Простота разработки программ с помощью бесплатного набора библиотек StellarisWare.
- Встроенные аналоговые компараторы и АЦП предоставляют возможности для аппаратной и программной оптимизации параметров системы.
- Улучшенные коммуникационные возможности, включая контроллеры 10/100 Ethernet MAC/PHY, USB и USB OTG, CAN, а также расширенные интерфейсы периферии.
- Оптимизированная по производительности архитектура с быстрыми внутренними шинами и быстродействующей флэш-памятью.
- Порты ввода/вывода общего назначения микроконтроллера способны генерировать прерывания, совместимы с 5 В и имеют программируемую нагрузочную способность и управляемую скорость нарастания выходного напряжения.

Почему следует выбирать решения Texas Instruments?

Разработчики встраиваемых систем, выбравшие решения компании Texas Instruments на базе процессоров ARM, получают:

- Огромное множество разнообразных микроконтроллеров, совместимых по набору команд — от дешёвых моделей стоимостью от 1 доллара до высокопроизводительных с тактовой частотой более 1 ГГц, входящих в самые разные семейства, начиная с микроконтроллеров семейства Stellaris и заканчивая микропроцессорами семейства Sitara — только Texas Instruments способна предложить такой широкий ассортимент изделий, совместимых по набору команд.
- Высокую степень интеграции и лёгкость разработки всех узлов будущего устройства благодаря использованию решений Texas Instruments в области аналоговой техники, управления электропитанием и обработки смешанных сигналов.
- Непревзойдённую поддержку по продажам в любой стране мира и поддержку технических специалистов Texas Instruments и наших дистрибьюторов.
- Доступ к недорогим и специализированным наборам для проектирования и программному обеспечению, способствующим более быстрому выводу изделий на рынок.
- Бесплатный набор библиотек StellarisWare.

Компания Texas Instruments поставила в общей сложности свыше 6 млрд микроконтроллеров с ядрами ARM, что составляет почти 25% от продаж ARM-микроконтроллеров по всему миру. Доверьте ваше будущее лидеру!



Обобщённая структура микроконтроллеров семейства Stellaris

ARM POWERED®

Cortex™
Intelligent Processors by ARM

И.3. Программное обеспечение StellarisWare®

StellarisWare® упрощает разработку программного обеспечения

При использовании микроконтроллеров Stellaris все части программы, включая обработчики прерываний и стартовый код, можно писать на языках высокого уровня Си/Си++. Чтобы ещё больше упростить процесс создания программы, мы предлагаем вам пакет StellarisWare®, содержащий примеры кода и бесплатные библиотеки для поддержки приложений:

- Stellaris Peripheral Driver Library — библиотека драйверов периферии, содержащая функции управления и инициализации периферийных устройств микроконтроллеров Stellaris;
- Stellaris USB Library — библиотека USB, позволяющая реализовывать приложения с функциями USB Device, USB Host или USB On-The-Go (OTG);
- Stellaris Graphic Library — графическая библиотека;
- Stellaris Boot Loader — загрузчик для программирования в условиях эксплуатации;
- Stellaris In-System Programming — библиотека функций для внутрисхемного программирования;
- Stellaris Utilities — библиотека оптимизированных часто используемых функций, таких как функции контроля CRC и вычисления таблиц AES;
- Stellaris IEC 60730 Library — библиотека функций для поддержки требований стандарта IEC 60730 Class B;
- Stellaris IQMath Library — библиотека математических функций, позволяющая ускорить выполнение операций с плавающей точкой на процессорах с фиксированной точкой;
- Stellaris Wireless Library — библиотека функций для работы с беспроводными решениями Texas Instruments;
- Stellaris Open Source Support — библиотека поддержки открытых реализаций Ethernet и RTOS;
- Stellaris Code Examples — разнообразные примеры программного кода.

Чтобы узнать о последних изменениях в составе пакета StellarisWare, воспользуйтесь ссылкой **www.ti.com/stellaris-ware-software**.

Во многих микроконтроллерах семейства Stellaris функции пакета StellarisWare «зашиты» в ПЗУ. Это позволяет использовать данные библиотеки для быстрой разработки эффективных и функциональных приложений в тех случаях, когда весь объём встроенной флэш-памяти занят пользовательской программой.

Набор библиотек StellarisWare имеет следующие особенности и преимущества:

- бесплатен для использования с микроконтроллерами Stellaris;
- упрощает и ускоряет разработку приложений — может использоваться как для разработки, так и в качестве примеров;
- позволяет создавать полнофункциональный код, который легко поддерживать;
- написан целиком на языке Си, за исключением тех участков кода, где это оказалось совершенно невозможно. Но даже написанный на Си, этот набор библиотек достаточно эффективно использует ресурсы памяти и процессора благодаря компактности набора команд Thumb2, поддерживаемого ядром Cortex-M3;
- полностью использует все возможности ядра Cortex-M3 по обслуживанию прерываний и не требует никаких специфических средств компилятора или ассемблерных вставок;
- может быть скомпилирован как с включением кода для контроля ошибок (на этапе разработки приложения), так и без такового (при создании финальной версии программы);
- доступен как в виде объектных файлов, так и в исходных кодах, т.е. вы можете использовать библиотеку «как есть» или же адаптировать её под свои нужды;
- компилируется в пакетах ARM/Keil, IAR, Code Composer Studio, Code Red, Code Sourcery, а также средств разработки GNU.

Последнюю версию пакета StellarisWare всегда можно найти по адресу **www.ti.com/stellarisware**.

Библиотека драйверов периферии Stellaris Peripheral Driver Library

Библиотека драйверов периферии Stellaris Peripheral Driver Library — это бесплатный набор программ для управления периферией микроконтроллеров семейства Stellaris на базе ядра ARM Cortex-M3. Существенно превосходящая по своим возможностям инструменты конфигурирования периферии на базе GUI-интерфейса библиотека Stellaris Peripheral Driver Library выполняет как инициализацию, так и управление периферией в режиме опроса или по прерываниям.

Библиотека Stellaris Peripheral Driver Library обеспечивает поддержку двух моделей программирования: прямой доступ к регистрам и программный драйвер. Каждую модель программирования можно использовать по желанию разработчика независимо или совместно, в зависимости от требований приложения или программной среды. Модель прямого доступа к регистрам включает файлы заголовка для каждого микроконтроллера семейства Stellaris и обеспечивает, в общем случае, более компактный и эффективный код в среде разработки, которая знакома большинству разработчиков встраиваемого программного обеспечения, ранее работавших с 8- и 16-битными микроконтроллерами. Модель программирования с использованием драйверов позволяет инженерам-программистам не вникать в особенности работы аппаратных средств, таких как функционирование каждого регистра, битовые поля, их взаимодействие и учёт последовательности работы периферии, что, как правило, сокращает время разработки приложения.

Графическая библиотека Stellaris Graphics Library

Графическая библиотека Stellaris Graphics Library — это бесплатный набор базовых функций и специальных графических элементов для создания графических пользовательских интерфейсов устройств на базе микроконтроллеров Stellaris, имеющих графический дисплей. Библиотека графики состоит из трёх основных уровней функциональности: уровень драйвера дисплея, который используется в приложении; уровень графических примитивов для построения точек, линий, прямоугольников, окружностей, шрифтов, растровых изображений и текста как в активном буфере дисплея, так и в неотображаемом буфере для предотвращения эффекта мерцания, а также уровень специальных графических элементов, которые обеспечивают отображение компонентов пользовательского интерфейса на дисплее (кнопки, ползунки, поля списка и др.) и их взаимодействие с пользователем в конкретном приложении.

Библиотека USB-устройств Stellaris USB Library

Все микроконтроллеры Stellaris с USB-интерфейсами проходят тест на функционирование в режиме USB Device и USB Embedded Host. Библиотека USB-устройств Stellaris USB Library — это бесплатный набор типов данных и функций для создания приложений с функциональностью USB Device, Host или On-the-Go (OTG) для систем на базе микроконтроллеров Stellaris. Предлагается несколько программных интерфейсов: от уровня простого управления USB-контроллером до интерфейсов высокого уровня, которые осуществляют API-поддержку специальных устройств.

Примеры USB Device	Примеры USB Host	Примеры USB OTG
HID-клавиатура HID-мышь CDC Serial Накопитель данных Generic Bulk Аудиоустройство Обновление ПО устройств Осциллограф	Накопитель данных HID-клавиатура HID-мышь	Протокол SRP (протокол запроса сессии) Протокол HNP (протокол обмена с хостом)
Предоставляется INF-файл Windows для поддерживаемых классов USB (в предварительно скомпилированной библиотеке DLL, что экономит время разработки).		

Библиотека Stellaris IEC 60730 Library

Стандарт IEC 60730 Class B охватывает бóльшую часть бытовых электроприборов, таких как стиральные/сушильные машины, холодильники, морозильники, электроплиты. Библиотека Stellaris IEC 60730 Library позволяет создавать встраиваемые приложения, удовлетворяющие требованиям по безопасности стандарта IEC 60730 Class B. Функции этой библиотеки используются совместно с такими аппаратными узлами микроконтроллеров семейства Stellaris, как сторожевые таймеры и прецизионный генератор. Библиотека поддерживает требования стандарта IEC 60730 по однократному (в момент запуска) и периодическому тестированию программы. Поскольку библиотека Stellaris IEC 60730 Library обеспечивает фундаментальную верификацию основных операций микроконтроллера, она часто используется в приложениях для заводского тестирования устройств, не попадающих под требования стандарта IEC 60730.

Библиотека Stellaris IQMath Library

Библиотека Stellaris IQMath Library, предлагаемая компанией Texas Instruments, представляет собой набор высоко оптимизированных математических функций для программистов Си/Си++, позволяющий с наименьшими затратами конвертировать алгоритмы, в которых требуются операции с плавающей точкой, в код, использующий арифметику с фиксированной точкой. Эти подпрограммы обычно применяются в приложениях реального времени с интенсивными вычислениями, где критична как скорость выполнения программы, так и высокая точность. Указанные функции выполняются гораздо быстрее своих аналогов, написанных на стандартном Си. Библиотека Stellaris IQMath Library также позволяет преодолеть ограничения, присущие арифметике с фиксированной точкой, путём задания программируемого динамического диапазона и разрешения.

Библиотеки Stellaris Wireless Library

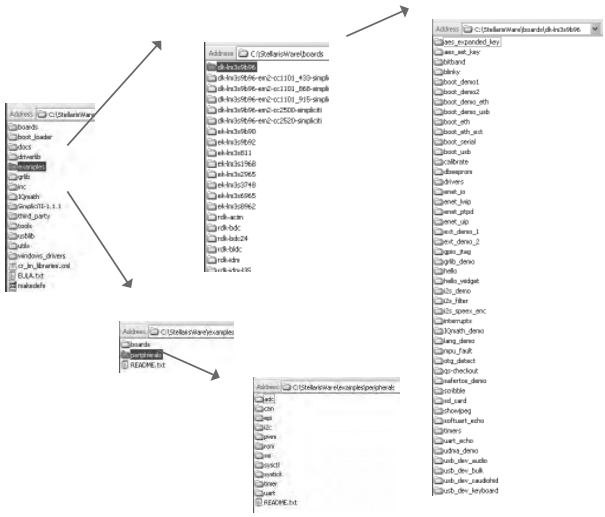
Благодаря своей высокой производительности и коммуникационным возможностям микроконтроллеры семейства Stellaris с ядром ARM Cortex-M3, предлагаемые компанией Texas Instruments, являются идеальным выбором для использования в самых разных беспроводных приложениях. Микроконтроллеры Stellaris и беспроводные решения от Texas Instruments привносят интеллектуальность и расширенную функциональность в такие приложения, как измерения, домашняя автоматизация и безопасность. Пакет StellarisWare облегчает вывод беспроводных приложений на рынок, предоставляя функции для использования протоколов RFID, Low-Power RF, ZigBee и Bluetooth.

Поддержка стандарта CMSIS

Помимо разнообразных библиотек, входящих в состав пакета StellarisWare, компания Texas Instruments также обеспечивает поддержку стандарта CMSIS, стандартизованного уровня аппаратных абстракций для процессоров семейства Cortex-M. Стандарт CMSIS описывает простые программные интерфейсы с процессором, адресованные изготовителям микросхем и поставщикам промежуточного ПО, которые позволяют упростить повторное использование программного кода и ускорить вывод новых устройств на рынок.

Примеры кода для микроконтроллеров Stellaris

Все наборы для проектирования и оценочные наборы Stellaris поставляются вместе с обширным набором приложений, которые представляют собой примеры использования микроконтроллеров Stellaris и набора библиотек StellarisWare. Каждый набор содержит приложение для быстрого старта, которое специально предназначено для демонстрации функций, заложенных в оценочную плату. Поскольку приложение для быстрого старта одновременно использует большую часть периферии на плате, наборы также поставляются вместе с примерами для работы с периферией. Эти приложения представляют собой примеры автономного кода для всей периферии, которая поддерживается в наборе. Приложения для быстрого старта и примеры для работы с периферией содержат исходный код и файлы проекта для поддержки пользователя при разработке системы. Для всех проектов примеров прилагается техническая документация, в которой описывается функциональность каждого приложения.



Поддержка внутрисистемного программирования Stellaris

Микроконтроллеры Stellaris поддерживают различные механизмы внутрисистемного программирования.

Все микроконтроллеры семейства Stellaris поставляются с загрузчиком, «зашитом» во встроенном ПЗУ, либо записанным во флэш-памяти. Это обеспечивает максимальную гибкость при программировании готовой продукции. Также мы бесплатно предлагаем исходный код программы-загрузчика, который позволит обновлять ваши устройства в условиях эксплуатации.

Системный загрузчик Stellaris в ПЗУ

Большинство микроконтроллеров Stellaris имеют системный загрузчик, расположенный во встроенном ПЗУ.

Этот загрузчик позволяет программировать флэш-память подобных микроконтроллеров (как на производстве, так и в условиях эксплуатации) с использованием интерфейса UART, I^2C или SSI. Благодаря широким возможностям выбора интерфейсов, а также возможности передачи сигналов при обновлении программ в системе, пользователи получают максимальную гибкость с помощью системного загрузчика Stellaris.

Системный загрузчик Stellaris во флэш-памяти

Для приложений, требующих программирования в условиях эксплуатации, мы также предлагаем исходный код бесплатного системного загрузчика Stellaris, который можно поместить в приложении в начале флэш-памяти. Благодаря широким возможностям выбора интерфейсов, включающих UART, I^2C, SSI, USB Host, USB Device и Ethernet, а также возможности передачи сигналов при обновлении программ в системе, пользователи получают максимальную гибкость с помощью системного загрузчика Stellaris. Библиотека драйверов периферии Stellaris Peripheral Driver Library включает исходный код и содержит дополнительную информацию, касающуюся системного загрузчика Stellaris, в том числе примеры приложений, в которых используется данный загрузчик.

- Бесплатная лицензия и бесплатное использование (для пользователей микроконтроллеров Stellaris).
- Компактный код, который можно разместить в начале флэш-памяти, чтобы использовать его как загрузчик приложения.
- Загрузчик также используется для обновления программного обеспечения приложений, работающих на микроконтроллерах Stellaris.
- Поддерживаемые интерфейсы: UART (по умолчанию), I^2C, SSI, USB Host (накопитель данных), USB Device (DFU) и Ethernet (протокол BOOTP).

Последовательный загрузчик флэш-памяти Stellaris

Младшие представители семейства Stellaris поставляются с предустановленным во флэш-память бесплатным последовательным загрузчиком. Данный загрузчик может использоваться совместно с утилитой LMFlash, стандартным JTAG-отладчиком или же промышленным программатором для загрузки кода приложения во флэш-память микроконтроллера на производстве.

Последовательный загрузчик — это компактное приложение, которое позволяет программировать флэш-память без использования интерфейса отладчика или программатора. Мы поставляем бесплатную утилиту для программирования флэш-памяти под названием LMFlash, которая может запускаться из командной строки или же посредством графического интерфейса. Утилита обеспечивает полное использование всех команд последовательного загрузчика. Для пользователей, создающих собственный программатор флэш-памяти, мы также поставляем простую утилиту загрузки по интерфейсу UART, которая обеспечивает полное использование всех команд последовательного загрузчика. В руководстве по применению AN01242 приводится исходный код и информация о последовательном загрузчике и утилите загрузки sflash.exe.

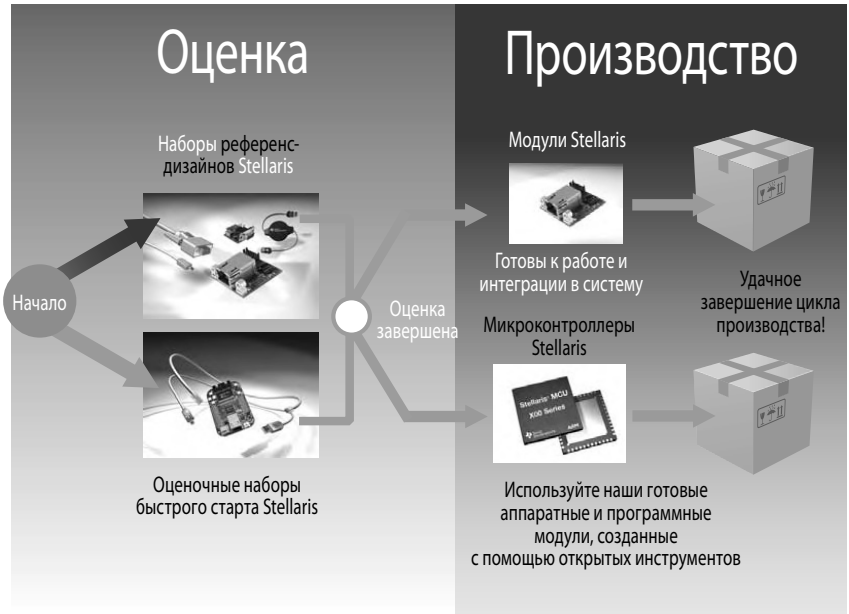
- Предустанавливается во флэш-память всех микроконтроллеров Stellaris, не имеющих системного загрузчика в ПЗУ.
- Представляет собой компактную программу, позволяющую программировать флэш-память без использования отладчика.
- Поддерживает интерфейсы UART и SSI.
- Имеется бесплатная утилита LMFlash, которая обеспечивает выполнение всех команд, поддерживаемых последовательным загрузчиком.

И.4. Наборы референс-дизайнов и модули

Гибкость при изготовлении продукции

Мы обеспечиваем гибкость выбора стратегии вывода продукции на рынок для целого спектра приложений. Инженеры могут начать оценку продукции и её разработку с помощью оценочного набора Stellaris Evaluation Kit (**www.ti.com/stellaris_evkits**) или готового набора референс-дизайнов Stellaris Reference Design Kit (**www.ti.com/stellaris_rdkits**). Разработчики могут использовать стандартные, готовые к работе модули Stellaris Modules (**www.ti.com/stellaris_modules**) или включить во встраиваемое приложение бесплатное эталонное аппаратное и программное обеспечение, созданное с помощью открытых инструментов разработки.

Наши модули позволяют ускорить вывод продукции на рынок благодаря готовым устройствам в удобном форм-факторе и применению эффективного программного обеспечения. С помощью набора референс-дизайна и модулей, обеспечивающих открытые средства проектирования, можно получить предварительно собранные блоки или создать свои собственные модули. Пакеты по проектированию печатных плат для каждого модуля, которые можно получить по адресу **www.ti.com/stellaris_modules**, предоставляют пользователю электрические схемы, перечни комплектующих (BOM) и Gerber-файлы.



Модуль Stellaris для преобразования последовательных протоколов в Ethernet

Интеллектуальный дисплейный модуль Stellaris с питанием через Ethernet (Power-over-Ethernet)

Интеллектуальный дисплейный модуль Stellaris с интерфейсом Ethernet

Интеллектуальный дисплейный модуль Stellaris с 3.5-дюймовым экраном

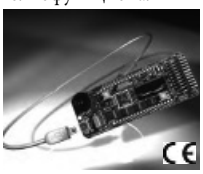
Модуль Stellaris для управления щёточным двигателем постоянного тока

Модуль Stellaris для управления бесщёточным двигателем постоянного тока

Модуль Stellaris для управления шаговым двигателем

Модуль Stellaris для управления асинхронным двигателем переменного тока

И.5. Микроконтроллеры реального времени



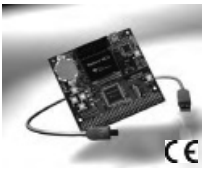
Мы предлагаем 30 недорогих полнофункциональных микроконтроллеров Stellaris на базе ARM Cortex-M3 с малым числом выводов в двух вариантах корпусов: в 48-выводном корпусе LFQP и в новом компактном 48-выводном корпусе QFN (**Табл. И.1**) . Серии LM3S100 и LM3S300 прекрасно подходят для базовых встраиваемых приложений и модернизации 8- и 16-битных систем. Серии LM3S600 и LM3S800 оптимизированы для встраиваемых приложений, в которых требуется реализация алгоритмов управления повышенной сложности. Все микроконтроллеры Stellaris обеспечивают высокую производительность и высокую степень интеграции, которые предпочтительны для сравнительно недорогих приложений, где требуется реализация сложных алгоритмов управления. К этим приложениям относятся, например, системы управления приводом, медицинские приборы, системы отопления, вентиляции, кондиционирования воздуха и управления зданием, системы промышленной автоматики, электронные системы на транспорте, кассовые терминалы и игровые устройства.

Оценочный набор Stellaris LM3S811

- Оценочная плата с 50-МГц микроконтроллером LM3S811.
- OLED-дисплей с разрешением 96×16 точек.
- Программируемая пользователем кнопка и светодиод.
- Удобная кнопка сброса (RESET) и светодиодный индикатор питания.
- Потенциометр с дисковым переключателем на входе встроенного в кристалл АЦП.
- Последовательный интерфейс внутрисхемной отладки через USB.
- USB-кабель.
- 20-выводной JTAG/SWD-кабель.
- Компакт-диск, содержащий:
  - оценочную версию сред разработки;
  - полную документацию;
  - руководство по быстрому старту и исходный код;
  - набор библиотек StellarisWare, включая библиотеку драйверов периферии и примеры исходного кода.
- Примеры приложений, демонстрирующие использование различных операционных систем реального времени, можно загрузить на **www.ti.com/stellaris_lm3s811**.

Информация для заказа оценочных наборов

Код	Описание
EKK-LM3S811	Оценочный набор Stellaris LM3S811 для Keil RealView MDK-ARM (ограничение размера кода — 32 Кбайт)
EKI-LM3S811	Оценочный набор Stellaris LM3S811 для IAR Systems Embedded Workbench (ограничение размера кода — 32 Кбайт)
EKC-LM3S811	Оценочный набор Stellaris LM3S811 для CodeSourcery G++ (30-дневное ограничение)
EKT-LM3S811	Оценочный набор Stellaris LM3S811 для Code Red Technologies Red Suite (ограничение размера кода — 32 Кбайт)
EKS-LM3S811	Оценочный набор Stellaris LM3S811 для Code Composer Studio™ от компании Texas Instruments (полная версия с привязкой к плате)



Микроконтроллеры Stellaris на базе ARM Cortex-M3 серии LM3S1000 сочетают в себе расширенные порты ввода/вывода общего назначения и увеличенный объём встроенной памяти, а также обладают низким энергопотреблением, оптимизированным для устройств с батарейным питанием. Микроконтроллеры серии LM3S1000 (**Табл. И.2**) предлагаются в 64-выводных корпусах LQFP, 100-выводных корпусах LQFP или 108-выводных корпусах BGA и обеспечивают высокую производительность и высокую степень интеграции, которые хорошо подходят для недорогих приложений, реализующих сложные алгоритмы управления. К этим приложениям относятся, например, системы управления приводом, медицинские приборы, системы отопления, вентиляции, кондиционирования воздуха и управления зданием, системы промышленной автоматики, электронные системы на транспорте, кассовые терминалы и игровые устройства.

Оценочный набор Stellaris LM3S1968

- Оценочная плата LM3S1968 с приложением для быстрого начала разработки:
  - микроконтроллер Stellaris LM3S1968 — 256 Кбайт флэш-памяти, 64 Кбайт SRAM, 8-канальный АЦП и до 52 портов ввода/вывода общего назначения;
  - все порты ввода/вывода LM3S1968 выведены на маркированные внешние контактные площадки;
  - поддержка энергосберегающего спящего режима (hibernate);
  - простой запуск: USB-кабель обеспечивает последовательную связь, отладку и питание;
  - графический OLED-дисплей с разрешением 128×64 точек и 16 градациями серого;
  - пользовательский светодиод, переключатели и кнопки, электромагнитный динамик;
  - стандартный 20-контактный JTAG-разъём для отладки от ARM.
- USB- и JTAG-кабели.
- Компакт-диск, содержащий:
  - оценочную версию сред разработки, полную документацию, руководство по быстрому старту и исходный код;
  - набор библиотек StellarisWare, включая библиотеку драйверов периферии и примеры исходного кода.

Информация для заказа оценочных наборов

Код	Описание
EKK-LM3S1968	Оценочный набор Stellaris LM3S1968 для Keil RealView MDK-ARM (ограничение размера кода — 32 Кбайт)
EKI-LM3S1968	Оценочный набор Stellaris LM3S1968 для IAR Systems Embedded Workbench (ограничение размера кода — 32 Кбайт)
EKC-LM3S1968	Оценочный набор Stellaris LM3S1968 для CodeSourcery G++ (30-дневное ограничение)
EKT-LM3S1968	Оценочный набор Stellaris LM3S1968 для Code Red Technologies Red Suite (90-дневное ограничение)
EKS-LM3S1968	Оценочный набор Stellaris LM3S1968 для Code Composer Studio™ от компании Texas Instruments (полная версия с привязкой к плате)

Таблица И.1. **Микроконтроллеры серий LM3S100/300/600/800**

Характеристики			LM3S101	LM3S102	LM3S300	LM3S301	LM3S308	LM3S310	LM3S315	LM3S316	LM3S317	LM3S328	LM3S600	LM3S601	LM3S608
Память	Флэш-память, Кбайт		8	8	16	16	16	16	16	16	16	16	32	32	32
	ОЗУ, Кбайт		2	2	4	2	4	4	4	4	4	4	8	8	8
	Библиотеки в ПЗУ		—	—	—	—	—	—	—	—	—	—	—	—	—
	DMA		—	—	—	—	—	—	—	—	—	—	—	—	—
	SAFERTOS™		—	—	—	—	—	—	—	—	—	—	—	—	—
Ядро	Макс. такт. частота, МГц		20	20	25	20	25	25	25	25	25	25	50	50	50
	Встроенный прецизионный генератор		—	—	—	—	—	—	—	—	—	—	—	—	—
	MPU		—	—	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Таймеры	SysTick (24-битный)		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Таймеры общего назначения		2	2	3	2	3	3	3	3	3	3	3	3	3
	Часы реального времени		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Сторожевой таймер		1	1	1	1	1	1	1	1	1	1	1	1	1
	Управление приводом	Модуль ШИМ	—	—	—	2	—	6	2	4	6	—	—	6	—
		Вход защиты ШИМ	—	—	—	1	—	1	1	1	1	—	—	1	—
		Генератор «мёртвого» времени	—	—	—	✓	—	✓	✓	✓	✓	—	—	✓	—
		Модуль CCP	1	2	6	2	6	6	6	6	6	6	6	6	6
		QEI-каналы	—	—	—	—	—	—	—	—	—	—	—	1	—
Интерфейс внешней периферии			—	—	—	—	—	—	—	—	—	—	—	—	—
Последовательные интерфейсы	Ethernet	10/100 MAC+PHY	—	—	—	—	—	—	—	—	—	—	—	—	—
		IEE 1588	—	—	—	—	—	—	—	—	—	—	—	—	—
	CAN MAC		—	—	—	—	—	—	—	—	—	—	—	—	—
	USB D, H или O		—	—	—	—	—	—	—	—	—	—	—	—	—
	UART		1	1	2	1	2	2	2	2	1	2	2	2	2
	I^2C		—	1	1	—	1	—	—	1	—	1	1	1	1
	SSI/SPI		1	1	1	1	1	1	1	1	1	1	1	1	1
	I^2S		—	—	—	—	—	—	—	—	—	—	—	—	—
Аналоговая периферия	АЦП (10- бит)	Модули АЦП	—	—	—	1	1	—	1	1	1	1	—	—	1
		Каналы АЦП	—	—	—	3	8	—	4	4	6	8	—	—	8
		Быстродействие АЦП (выб/с)	—	—	—	250K	500K	—	250K	250K	250K	500K	—	—	500K
	Встроенный датчик температуры		—	—	—	✓	✓	—	✓	✓	✓	✓	—	—	✓
	Аналоговые/цифровые компараторы		2/—	1/—	3/—	2/—	1/—	3/—	1/—	1/—	1/—	—/—	3/—	3/—	1/—
Порты ввода/вывода общ. назнач. (совместимы с 5 В)			2…18	0…18	8…36	12…33	5…28	3…36	7…32	3…32	3…30	7...28	8...36	0...36	5...28
Режим энергосбережения с резерв. от батареи			—	—	—	—	—	—	—	—	—	—	—	—	—
LDO-стабилизатор напряжения			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Рабочая температура			I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E
Тип корпуса			28 SOIC 48QFP 48QFN	28 SOIC 48QFP 48QFN	48 QFP 48 QFN	48 QFP 48 QFN	48 QFP 48 QFN	48 QFP 48 QFN	48 QFP 48 QFN	48 QFP 48 QFN	48 QFP 48 QFN	48 QFP 48 QFN	48 QFP 48 QFN	48 QFP 48 QFN	48 QFP 48 QFN
Серийное производство (P), опытные образцы (S)			P	P	P	P	P	P	P	P	P	P	P	P	P

LM3S610	**LM3S611**	**LM3S612**	**LM3S613**	**LM3S615**	**LM3S617**	**LM3S618**	**LM3S628**	**LM3S800**	**LM3S801**	**LM3S808**	**LM3S811**	**LM3S812**	**LM3S815**	**LM3S817**	**LM3S818**	**LM3S828**
32	32	32	32	32	32	32	32	64	64	64	64	64	64	64	64	64
8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
6	6	2	4	6	6	6	—	—	6	—	6	2	6	6	6	—
1	1	1	1	1	1	1	—	—	1	—	1	1	1	1	1	—
✓	✓	✓	✓	✓	✓	✓	—	—	✓	—	✓	✓	✓	✓	✓	—
6	6	6	6	6	6	4	4	6	6	6	6	6	6	6	4	6
—	—	—	—	—	—	1	—	—	1	—	—	—	—	—	1	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
1	1	1	1	1	—	—	1	1	1	1	1	1	1	—	—	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
1	1	1	1	1	1	1	1	—	—	1	1	1	1	1	1	1
2	4	2	4	2	6	6	8	—	—	8	4	2	2	6	6	8
500K	500K	500K	500K	500K	500K	500K	1M	—	—	500K	500K	250K	500K	1M	1M	1M
✓	✓	✓	✓	✓	✓	✓	✓	—	—	✓	✓	✓	✓	✓	✓	✓
—/—	—/—	1/—	1/—	3/—	1/—	1/—	—/—	3/—	3/—	1/—	1/—	1/—	3/—	1/—	1/—	—/—
6...34	4...32	7...34	3...32	0...34	1...30	0...30	9...28	8...36	0...36	5...28	1...32	7...34	0...34	1...30	0...30	7...28
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E
48 QFP 48 QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN	48QFP 48QFN
P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P

Таблица И.2. **Микроконтроллеры серии LM3S1000**

Характеристика			LM3S1110	LM3S1133	LM3S1138	LM3S1150	LM3S1162	LM3S1165	LM3S1166	LM3S1332	LM3S1435	LM3S1439	LM3S1512	LM3S1538	LM3S1601	LM3S1607	LM3S1608	LM3S1620	LM3S1625
Память	Флэш-память, Кбайт		64	64	64	64	64	64	64	96	96	96	96	96	128	128	128	128	128
	ОЗУ, Кбайт		16	16	16	16	16	16	16	16	32	32	64	64	32	32	32	32	32
	Библиотеки в ПЗУ		—	—	—	—	—	—	—	—	—	—	—	—	—	✓	—	—	✓
	DMA		—	—	—	—	—	—	—	—	—	—	—	—	—	✓	—	—	✓
	SAFERTOS™		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Ядро	Макс. такт. частота, МГц		25	50	50	50	50	50	50	50	50	50	25	50	50	50	50	25	50
	Встроенный прецизионный генератор		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	MPU		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Таймеры	SysTick (24-битный)		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Таймеры общ. назначения		3	4	4	4	4	4	4	4	3	3	4	4	4	4	4	3	4
	Часы реального времени		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Сторожевой таймер		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	Управление приводом	Модуль ШИМ	—	2	—	6	6	6	6	—	2	6	—	—	—	—	—	6	4
		Вход защиты ШИМ	—	1	—	1	1	1	1	—	1	1	—	—	—	—	—	1	1
		Генератор «мёртвого» времени	—	✓	—	✓	✓	✓	✓	—	✓	✓	—	—	—	—	—	✓	✓
		Модуль CCP	2	8	6	6	6	8	8	8	4	6	8	8	8	6	8	4	4
		QEI-каналы	—	—	—	1	—	—	—	—	—	1	1	1	—	—	—	1	—
Интерфейс внешней периферии			—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Последовательные интерфейсы	Ethernet	10/100 MAC+PHY	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
		IEE 1588	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	CAN MAC		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	USB D, H или O		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	UART		2	3	3	3	3	3	3	2	2	2	3	3	3	3	2	2	1
	I^2C		—	1	2	1	1	1	1	—	1	1	2	2	2	2	2	1	2
	SSI/SPI		1	2	2	2	2	2	2	1	1	2	2	2	2	1	2	2	1
	I^2S		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Аналоговая периферия	АЦП (10-бит)	Модули АЦП	—	1	1	—	1	1	1	1	1	1	1	1	—	1	1	—	1
		Каналы АЦП	—	2	8	—	2	4	4	3	2	4	2	8	—	8	8	—	6
		Быстродействие АЦП (выб/с)	—	250K	1M	—	500K	500K	500K	250K	500K	500K	250K	500K	—	500K	500K	—	500K
	Встроенный датчик температуры		—	✓	✓	—	✓	✓	✓	✓	✓	✓	✓	✓	—	✓	✓	—	✓
	Аналоговые/цифровые компараторы		2/—	1/—	3/—	3/—	3/—	1/—	1/—	3/—	1/—	1/—	3/—	—/—	2/—	—/—	2/—	3/—	1/—
Порты ввода/вывода общ. назнач. (совместимы с 5 В)			20 … 41	9 … 44	9 … 46	7 … 52	4 … 46	4 … 43	4 … 43	29 … 57	21 … 46	14 … 52	15 … 58	9 … 43	23 … 60	0 … 33	17 … 52	11 … 52	0 … 33
Режим энергосбережения с резерв. от батареи			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	—
LDO-стабилизатор напряжения			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Рабочая температура			I/E	I/E	I/E	I/E	I/E	I/E	I	I/E	I/E	I/E	I/E	I/E	I/E	I	I/E	I/E	I
Тип корпуса			100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	64 LQFP	100 LQFP 108 BGA	100 LQFP 108 BGA	64 LQFP
Серийное производство (P), опытные образцы (S)			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

LM3S1626	LM3S1627	LM3S1635	LM3S1636	LM3S1637	LM3S1751	LM3S1776	LM3S1811	LM3S1816	LM3S1850	LM3S1911	LM3S1918	LM3S1937	LM3S1958	LM3S1960	LM3S168	LM3S1969	LM3SIJ11	LM3SIJ16	LM3SIN11	LM3SIN16	LM3SIW16	LM3SIZ16
128	128	128	128	128	128	128	256	256	256	256	256	256	256	256	256	256	128	128	64	64	32	16
32	32	32	32	32	64	64	32	32	32	64	64	64	64	64	64	64	20	20	12	12	8	6
✓	✓	—	—	—	—	✓	✓	✓	—	—	—	—	—	—	—	—	✓	✓	✓	✓	✓	✓
✓	✓	—	—	—	—	✓	✓	✓	—	—	—	—	—	—	—	—	✓	✓	✓	✓	✓	✓
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50	50
—	—	—	—	—	—	—	1	1	—	—	—	—	—	—	—	—	1	1	1	1	1	1
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
4	4	4	4	4	3	3	4	4	3	4	4	3	4	4	4	4	3	3	3	3	3	3
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1	1	1	1	1	1	1	2	2	1	1	1	1	1	1	1	1	2	2	2	2	2	2
4	6	6	6	6	4	8	—	—	6	—	—	6	—	6	6	6	—	—	—	—	—	—
1	1	1	1	1	1	3	—	—	1	—	—	1	—	1	1	1	—	—	—	—	—	—
✓	✓	✓	✓	✓	✓	✓	—	—	✓	—	—	✓	—	✓	✓	✓	—	—	—	—	—	—
4	4	8	8	6	6	2	8	8	6	8	8	4	8	8	4	4	6	6	6	6	6	6
1	1	—	—	1	—	—	—	—	1	—	—	—	—	2	2	2	—	—	—	—	—	—
—	—	—	—	—	—	—	✓	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
2	2	3	3	3	3	1	3	3	2	3	2	2	3	3	3	3	3	3	3	3	3	3
1	1	2	2	1	1	1	2	2	1	2	2	1	2	2	2	2	2	2	2	2	2	2
1	1	2	2	1	2	1	2	2	1	2	2	1	2	2	2	2	2	2	2	2	2	2
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
1	1	1	1	1	1	1	1	1	—	—	1	1	1	—	1	1	1	1	1	1	1	1
6	4	4	4	4	4	6	8	8	—	—	8	4	8	—	8	8	8	8	8	8	8	8
500K	500K	500K	500K	1M	500K	1M	1M	1M	—	—	500K	1M	1M	—	1M	1M	1M	1M	1M	1M	1M	1M
✓	✓	✓	✓	✓	✓	✓	✓	✓	—	—	✓	✓	✓	—	✓	✓	✓	✓	✓	✓	✓	✓
—/—	—/—	2/—	2/—	1/—	1/—	—/—	2/8	2/8	3/—	2/—	2/—	1/—	—/—	3/—	3/—	3/—	2/8	2/8	2/8	2/8	2/8	2/8
0…33	0…33	12…56	12…56	7…43	21…56	1…33	0…67	0…33	17…56	23…60	17…52	27…56	21…52	7…60	5…52	5…52	0…67	0…33	0…67	0…33	0…33	0…33
—	—	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
I	I	I/E	I	I/E	I/E	I	I	I	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I	I	I	I	I	I	I
64 LQFP	64 LQFP	100 LQFP 108 BGA	100 LQFP	100 LQFP 108 BGA	100 LQFP 108 BGA	64 LQFP	100 LQFP	64 LQFP	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP	100 LQFP	64 LQFP	100 LQFP	64 LQFP	64 LQFP	64 LQFP
✓	P	P	P	P	P	P	S	S	P	P	P	P	P	P	P	P	S	S	S	S	S	S

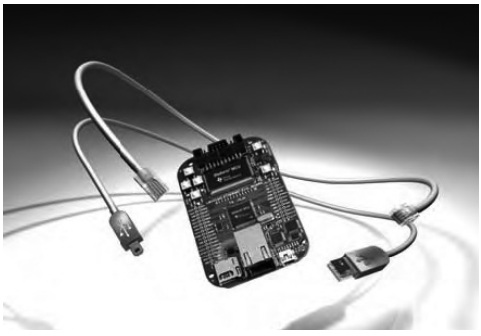
[a] ШИМ-управление приводом выполняется с помощью специальных аппаратных средств (выводы PWM) или с помощью функций управления приводом, имеющихся у таймеров общего назначения (выводы CCP). Подробности см. в технической документации. [b] Минимум — число выводов, предназначенных для портов ввода/вывода общего назначения; дополнительные выводы доступны, если не используется какая-либо периферия. Подробности см. в технической документации. [c] Промышленный диапазон (I): –40…+85°C, расширенный диапазон (E): –40…+105°C. [d] 108-выводной корпус BGA и 64-выводной корпус LQFP предлагаются только для промышленного диапазона рабочих температур.

И.6. Микроконтроллеры с интерфейсом Ethernet

Микроконтроллеры Stellaris на базе ARM Cortex-M3 серии LM3S6000 сочетают в себе возможность работы в режиме реального времени, расширенные порты ввода/вывода общего назначения и увеличенный объём встроенной памяти, а также обладают низким энергопотреблением, оптимизированным для приложений с резервным питанием от батареи. Микроконтроллеры серии LM3S6000 являются первыми в мире микроконтроллерами с полностью интегрированным 10/100-Мбит/с Ethernet-решением, совместимым с ARM-архитектурой. Микроконтроллеры серии LM3S6000 (**Табл. И.3**) содержат не только Ethernet MAC-контроллер (MAC), но и трансивер (PHY). Кроме того, отдельные микроконтроллеры серии LM3S6000 обеспечивают аппаратную поддержку протокола Precision Time Protocol согласно IEEE 1588.

Оценочный набор Stellaris LM3S6965 Ethernet

Оценочные наборы Stellaris LM3S6965 обеспечивают компактную и универсальную платформу для оценки микроконтроллеров Stellaris на базе ARM Cortex-M3 с интерфейсом Ethernet. Набор содержит два примера приложений, демонстрирующих реализацию встраиваемого веб-сервера. Готовое к работе приложение для быстрого старта включает в себя встраиваемый веб-сервер, использующий Ethernet-стек Open Source lwIP. Набор также содержит веб-сервер с ОС реального времени FreeRTOS.org™ и Ethernet-стек Open Source uIP. Каждая плата имеет внутрисхемный интерфейс отладки In-Circuit Debug Interface (ICDI), который обеспечивает возможность аппаратной отладки не только для размещённого на плате микроконтроллера Stellaris, но и для любой платы на базе контроллеров Cortex M3/M4 от Texas Instruments. Оценочный набор содержит все кабели, программное обеспечение и документацию, необходимые для быстрого проектирования и запуска приложения для микроконтроллеров Stellaris. Кроме того, примеры приложений, демонстрирующие использование различных ОС реального времени и коммерческих Ethernet-стеков, можно загрузить на **www.ti.com/stellaris_lm3s6965**.



Особенности оценочного набора Stellaris LM3S6965 Ethernet

- Оценочная плата LM3S6965.
- Микроконтроллер Stellaris LM3S6965 с интегрированным 10/100-Мбит/с Ethernet-контроллером (MAC+PHY).
- Простая установка: USB-кабель обеспечивает последовательную связь, отладку и питание.
- Графический OLED-дисплей с разрешением 128×64 точек и 16 градациями серого.
- Пользовательский светодиод, переключатели и кнопки.
- Электромагнитный динамик.
- Все порты ввода/вывода LM3S6965 выведены на маркированные внешние контактные площадки.
- Стандартный 20-контактный JTAG-разъём для отладки от ARM.
- Разъём карты MicroSD.
- Убирающийся Ethernet-кабель, USB-кабель и JTAG-кабель.
- Приложение для быстрого старта работает как с Ethernet, так и без него (непосредственная связь с компьютером).
- Компакт-диск, содержащий:
  - оценочную версию сред программирования;
  - руководство по быстрому старту и исходный код;
  - полную документацию;
  - набор библиотек StellarisWare, включая библиотеку драйверов периферии и пример исходного кода.

Информация для заказа оценочных наборов

Код	Описание
EKK-LM3S6965	Оценочный набор Stellaris LM3S6965 Ethernet для Keil RealView MDK-ARM (ограничение размера кода — 32 Кбайт)
EKI-LM3S6965	Оценочный набор Stellaris LM3S6965 Ethernet для IAR Systems Embedded Workbench (ограничение размера кода — 32 Кбайт)
EKC-LM3S6965	Оценочный набор Stellaris LM3S6965 Ethernet для CodeSourcery G++ (30-дневное ограничение)
EKT-LM3S6965	Оценочный набор Stellaris LM3S6965 Ethernet для Code Red Technologies Red Suite (90-дневное ограничение)
EKS-LM3S6965	Оценочный набор Stellaris LM3S6965 для Code Composer Studio™ от компании Texas Instruments (полная версия с привязкой к плате)

Таблица И.3. **Микроконтроллеры серии LM3S6000**

Характеристика			LM3S6100	LM3S6110	LM3S6420	LM3S6422	LM3S6432	LM3S6537	LM3S6610	LM3S6611	LM3S6618	LM3S6633	LM3S6637	LM3S6730	LM3S6753	LM3S6911	LM3S6918	LM3S6938	LM3S6950	LM3S6952	LM3S6965
Память	Флэш-память, Кбайт		64	64	96	96	96	96	128	128	128	128	128	128	128	256	256	256	256	256	256
	ОЗУ, Кбайт		16	16	32	32	32	64	32	32	32	32	32	64	64	64	64	64	64	64	64
	Библиотеки в ПЗУ		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	DMA		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	SAFERTOS™		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Ядро	Макс. такт. частота, МГц		25	25	25	25	50	50	25	50	50	50	50	50	50	50	50	50	50	50	50
	Встроенный прецизионный генератор		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	MPU		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Таймеры	SysTick (24-битный)		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Таймеры общего назначения		3	3	3	3	3	4	4	4	4	3	4	3	4	4	4	4	4	3	4
	Часы реального времени		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Сторожевой таймер		1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	Управление приводом	Модуль ШИМ	—	2	—	—	2	6	4	—	—	—	—	—	6	—	—	—	6	4	6
		Вход защиты ШИМ	—	1	—	—	1	1	1	—	—	—	—	—	1	—	—	—	1	1	1
		Генератор «мёртвого» времени	—	✓	—	—	✓	✓	✓	—	—	—	—	—	✓	—	—	—	✓	✓	✓
		Модуль CCP	4	4	4	4	4	6	6	6	6	6	6	4	4	6	6	6	6	4	4
		QEI-каналы	—	—	—	—	—	—	1	—	—	—	—	—	1	—	—	—	1	1	2
Интерфейс внешней периферии					—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Последовательные интерфейсы	Ethernet	10/100 MAC+PHY	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		IEE 1588	—	—	—	—	—	✓	—	—	—	—	—	—	✓	—	—	—	✓	—	—
	CAN MAC		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	USB D, H или O		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
	UART		1	1	1	1	2	2	3	3	2	2	2	1	2	3	2	3	3	3	3
	I^2C		—	—	—	—	1	1	1	2	2	1	1	—	1	2	2	1	1	1	2
	SSI/SPI		1	1	1	1	1	1	1	2	2	1	1	1	1	2	2	1	2	1	1
	I^2S		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
Аналоговая периферия	АЦП (10- бит)	Модули АЦП	—	—	—	1	1	1	—	—	1	1	1	—	1	—	1	1	—	1	1
		Каналы АЦП	—	—	—	2	3	4	—	—	8	3	4	—	4	—	8	8	—	3	4
		Быстродействие АЦП (выб/с)	—	—	—	250K	250K	500K	—	—	500K	500K	1M	—	500K	—	500K	1M	—	500K	1M
	Встроенный датчик температуры		—	—	—	✓	✓	✓	—	—	✓	✓	✓	—	✓	—	—	✓	—	✓	✓
	Аналоговые/цифровые компараторы		1/—	3/—	2/—	2/—	2/—	2/—	3/—	2/—	2/—	1/—	3/—	2/—	2/—	2/—	2/—	3/—	3/—	3/—	2/—
Порты ввода/вывода общ. назнач. (совместимы с 5 В)			10 … 30	8 … 35	23 … 46	12 … 34	14 … 43	6 … 41	5 … 46	10 … 46	5 … 38	15 … 41	11 … 41	23 … 46	5 … 41	10 … 46	5 … 38	7 … 38	1 … 46	6 … 43	0 … 42
Режим энергосбережения с резерв. от батареи			—	—	—	—	—	✓	✓	✓	✓	✓	✓	—	✓	✓	✓	✓	✓	✓	✓
LDO-стабилизатор напряжения			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Рабочая температура			I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E
Тип корпуса			100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA	100LQFP 108BGA
Серийное производство (P), опытные образцы (S)			P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P	P

[a] ШИМ-управление приводом выполняется с помощью специальных аппаратных средств (выводы PWM) или с помощью функций управления приводом, имеющихся у таймеров общего назначения (выводы CCP). Подробности см. в технической документации. [b] Минимум — число выводов, предназначенных для портов ввода/вывода общего назначения; дополнительные выводы доступны, если не используется какая-либо периферия. Подробности см. в технической документации. [c] Промышленный диапазон (I): –40…+85°C, расширенный диапазон (E): –40…+105°C. [d] 108-выводной корпус BGA предлагается только для промышленного диапазона рабочих температур.

И.7. Микроконтроллеры с интерфейсом USB

Оценочный набор Stellaris LM3S3748 USB Host/Device

Оценочный набор Stellaris LM3S3748 USB Host/Device

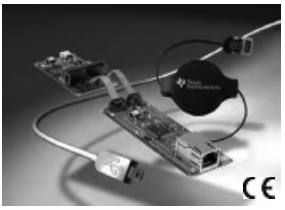
Оценочная плата Stellaris LM3S3748 демонстрирует ключевые особенности микроконтроллера LM3S3748, включая полноскоростной (12 Мбит/с) контроллер USB 2.0 Host/Device, АЦП и последовательные интерфейсы. В режиме USB Device компактный переключатель позволяет выбрать питание через шину или автономное питание. Готовое к работе приложение для быстрого старта использует четыре сигнала АЦП, образующие два дифференциальных канала, для реализации программного осциллографа с частотой выборки 1 Мвыб/с с ЖК-дисплеем, демонстрируя высокоскоростную систему сбора и обработки данных с усовершенствованным пользовательским интерфейсом, который разработан с помощью графической библиотеки StellarisWare Graphics Library. Приложение для быстрого старта использует USB-библиотеку StellarisWare USB Library для работы в режимах USB Host и USB Device, записывает растровые изображения сигналов, отображаемых на дисплее, и CSV-данные во встроенный USB-накопитель и имеет возможность соединения с компьютером для дистанционного отображения данных. Плата LM3S3748 также содержит внутрисхемный интерфейс отладки In-Circuit Debug Interface (ICDI), который обеспечивает возможность аппаратной отладки не только для установленного на плате микроконтроллера Stellaris, но и для любой платы на базе микроконтроллера Stellaris. В режиме отладочного интерфейса встроенный в плату микроконтроллер блокируется, что позволяет программировать или осуществлять отладку внешней платы. Примеры приложений, демонстрирующие использование различных ОС реального времени и коммерческих коммуникационных стеков, можно загрузить на **www.ti.com/stellaris_lm3s3748**.

- 50-МГц микроконтроллер Stellaris LM3S3748 со встроенной 128-Кбайт флэш-памятью и 64-Кбайт SRAM.
- Приложение для быстрого старта, реализующее 2-канальный осциллограф.
- Поддержка питания через шину или автономного питания через USB.
- Цветной графический ЖК-дисплей с разрешением 128×128 точек.
- Пользовательский светодиод и навигационный переключатель.
- 8-Ом электромагнитный динамик с усилителем.
- Разъём карты MicroSD.
- Стандартный 20-контактный JTAG-разъём для отладки от ARM и JTAG/SWD-кабель.
- Порты ввода/вывода LM3S3748 выведены на маркированные внешние контактные площадки.
- USB-кабели и измерительные щупы осциллографа, необходимые для работы приложения быстрого старта.
- USB-накопитель.
- Компакт-диск, содержащий:
  - оценочную версию программных инструментов, полную документацию, руководство по быстрому старту и исходный код;
  - набор библиотек StellarisWare, включая библиотеку драйверов периферии и пример исходного кода.

Информация для заказа оценочных наборов

Код	Описание
EKK-LM3S3748	Оценочный набор Stellaris LM3S3748 USB Host/Device для Keil RealView MDK-ARM (ограничение размера кода — 32 Кбайт)
EKI-LM3S3748	Оценочный набор Stellaris LM3S3748 для IAR Systems Embedded Workbench (ограничение размера кода — 32 Кбайт)
EKC-LM3S3748	Оценочный набор Stellaris LM3S3748 для CodeSourcery G++ (30-дневное ограничение)
EKT-LM3S3748	Оценочный набор Stellaris LM3S3748 для Code Red Technologies Red Suite (90-дневное ограничение)
EKS-LM3S3748	Оценочный набор Stellaris LM3S3748 для Code Composer Studio™ от компании Texas Instruments (полная версия с привязкой к плате)

Оценочный набор Stellaris LM3S9B92 Ethernet+CAN



Оценочный набор Stellaris LM3S9B92 Ethernet+CAN

Имея в своём составе две платы, на одной из которых находится микроконтроллер LM3S9B92 с интерфейсами Ethernet+USB-OTG+CAN, а на другой реализован интерфейс внутрисхемной отладки BD-ICDI, оценочный набор Stellaris LM3S9B92 предоставляет недорогую, компактную и универсальную платформу для оценки микроконтроллеров Stellaris на базе ARM Cortex-M3 с интерфейсами Ethernet, USB и CAN. В оценочной плате используется микроконтроллер LM3S9B92, который позволяет реализовывать сложные алгоритмы управления и имеет 8 ШИМ-выходов для обеспечения управления приводом и снабжения энергией и 2 модуля квадратурного энкодера (QEI) для подключения импульсных датчиков положения. Кроме того, микроконтроллер LM3S9B92 способен работать с внешним 16-МГц кварцевым генератором, который обеспечивает основную тактовую частоту, используемую для непосредственного тактирования ARM-ядра или внутренней схемы ФАПЧ для увеличения частоты тактового сигнала ядра до 80 МГц. Для тактирования Ethernet-контроллера используется 25-МГц кварцевый генератор. Микроконтроллер LM3S9B92 также имеет встроенный LDO-стабилизатор напряжения, который обеспечивает питание внутренних цепей.

Информация для заказа оценочных наборов

Код	Описание
EKK-LM3S9B92	Недорогой оценочный набор Stellaris LM3S9B92 для Keil RealView MDK-ARM (ограничение размера кода — 32 Кбайт)
EKI-LM3S9B92	Недорогой оценочный набор Stellaris LM3S9B92 для IAR Systems Embedded Workbench (ограничение размера кода — 32 Кбайт)
EKC-LM3S9B92	Недорогой оценочный набор Stellaris LM3S9B92 для CodeSourcery G++ (30-дневное ограничение)
EKT-LM3S9B92	Недорогой оценочный набор Stellaris LM3S9B92 для Code Red Technologies Red Suite (90-дневное ограничение)
EKS-LM3S9B92	Оценочный набор Stellaris LM3S9B92 для Code Composer Studio™ от компании Texas Instruments (полная версия с привязкой к плате)

Таблица И.4. **Микроконтроллеры с интерфейсом USB**

Характеристика			LM3S6511	LM3S3739	LM3S3748	LM3S3749	LM3S3826	LM3S3J26	LM3S3N26	LM3S3W26	LM3S3Z26
Память	Флэш-память, Кбайт		128	128	128	128	256	128	64	32	16
	ОЗУ, Кбайт		32	64	64	64	32	20	12	8	6
	Библиотеки в ПЗУ		✓	✓	✓	✓	✓	✓	✓	✓	✓
	DMA		✓	✓	✓	✓	✓	✓	✓	✓	✓
	SAFERTOS™		—	—	—	—	—	—	—	—	—
Ядро	Макс. такт. частота, МГц		50	50	50	50	50	50	50	50	50
	Встроенный прецизионный генератор		—	—	—	—	✓	✓	✓	✓	✓
	MPU		✓	✓	✓	✓	✓	✓	✓	✓	✓
Таймеры	SysTick (24-битный)		✓	✓	✓	✓	✓	✓	✓	✓	✓
	Таймеры общего назначения		4	4	4	4	3	3	3	3	3
	Часы реального времени		✓	✓	✓	✓	✓	✓	✓	✓	✓
	Сторожевой таймер		1	1	1	1	2	2	2	2	2
	Управление приводом	Модуль ШИМ	—	—	8	8	—	—	—	—	—
		Вход защиты ШИМ	—	—	4	4	—	—	—	—	—
		Генератор «мёртвого» времени	—	—	✓	✓	—	—	—	—	—
		Модуль CCP	8	8	8	7	6	6	6	6	6
		QEI-каналы	—	—	1	1	—	—	—	—	—
Интерфейс внешней периферии			—	—	—	—	—	—	—	—	—
Последовательные интерфейсы	Ethernet	10/100 MAC+PHY	—	—	—	—	—	—	—	—	—
		IEE 1588	—	—	—	—	—	—	—	—	—
	CAN MAC		—	—	—	—	—	—	—	—	—
	USB D, H или O		O	H	H	H	D	D	D	D	D
	UART		1	3	2	3	3	3	3	3	3
	I²C		1	2	2	2	2	2	2	2	2
	SSI/SPI		1	2	2	2	2	2	2	2	2
	I²S		—	—	—	—	—	—	—	—	—
Аналоговая периферия	АЦП (10- бит)	Модули АЦП	1	1	1	1	1	1	1	1	1
		Каналы АЦП	4	8	8	8	8	8	8	8	8
		Быстродействие АЦП (выб/с)	500K	500K	1M	1M	1M	1M	1M	1M	1M
	Встроенный датчик температуры		✓	✓	✓	✓	✓	✓	✓	✓	✓
	Аналоговые/цифровые компараторы		2/—	2/—	2/—	2/—	2/8	2/8	2/8	2/8	2/8
Порты ввода/вывода общ. назнач. (совместимы с 5 В)			0…33	14…61	3…61	0…61	0…33	0…33	0…33	0…33	0…33
Режим энергосбережения с резерв. от батареи			✓	✓	✓	✓	✓	✓	✓	✓	✓
LDO-стабилизатор напряжения			✓	✓	✓	✓	✓	✓	✓	✓	✓
Рабочая температура			I	I	I	I	I	I	I	I	I
Тип корпуса			64LQFP	100LQFP	100LQFP	100LQFP	64LQFP	64LQFP	64LQFP	64LQFP	64LQFP
Серийное производство (P), опытные образцы (S)			P	P	P	P	S	S	S	S	S

[a] ШИМ-управление приводом выполняется с помощью специальных аппаратных средств (выводы PWM) или с помощью функций управления приводом, имеющихся у таймеров общего назначения (выводы CCP). Подробности см. в технической документации. [b] Минимум — число выводов, предназначенных для портов ввода/вывода общего назначения; дополнительные выводы доступны, если не используется какая-либо периферия. Подробности см. в технической документации. [c] Промышленный диапазон (I): –40…+85°C, расширенный диапазон (E): –40…+105°C.

И.8. Микроконтроллеры с интерфейсами USB и CAN

Оценочный набор Stellaris LM3S3748 USB Host/Device

Оценочный набор Stellaris LM3S3748 USB Host/Device

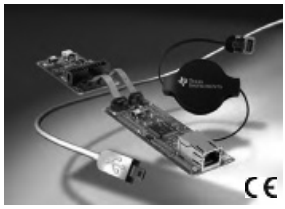
Оценочная плата Stellaris LM3S3748 демонстрирует ключевые особенности микроконтроллера LM3S3748, включая полноскоростной (12 Мбит/с) контроллер USB 2.0 Host/Device, АЦП и последовательные интерфейсы. В режиме USB Device компактный переключатель позволяет выбрать питание через шину или автономное питание. Готовое к работе приложение для быстрого старта использует четыре сигнала АЦП, образующие два дифференциальных канала, для реализации программного осциллографа с частотой выборки 1 Мвыб/с с ЖК-дисплеем, демонстрируя высокоскоростную систему сбора и обработки данных с усовершенствованным пользовательским интерфейсом, который разработан с помощью графической библиотеки StellarisWare Graphics Library. Приложение для быстрого старта использует USB-библиотеку StellarisWare USB Library для работы в режимах USB Host и USB Device, записывает растровые изображения сигналов, отображаемых на дисплее, и CSV-данные во встроенный USB-накопитель и имеет возможность соединения с компьютером для дистанционного отображения данных. Плата LM3S3748 также содержит внутрисхемный интерфейс отладки In-Circuit Debug Interface (ICDI), который обеспечивает возможность аппаратной отладки не только для установленного на плате микроконтроллера Stellaris, но и для любой платы на базе микроконтроллера Stellaris. В режиме отладочного интерфейса встроенный в плату микроконтроллер блокируется, что позволяет программировать или осуществлять отладку внешней платы. Примеры приложений, демонстрирующие использование различных ОС реального времени и коммерческих коммуникационных стеков, можно загрузить на **www.ti.com/stellaris_lm3s3748**.

- 50-МГц микроконтроллер Stellaris LM3S3748 со встроенной 128-Кбайт флэш-памятью и 64-Кбайт SRAM.
- Приложение для быстрого старта, реализующее 2-канальный осциллограф.
- Поддержка питания через шину или автономного питания через USB.
- Цветной графический ЖК-дисплей с разрешением 128×128 точек.
- Пользовательский светодиод и навигационный переключатель.
- 8-Ом электромагнитный динамик с усилителем.
- Разъём карты MicroSD.
- Стандартный 20-контактный JTAG-разъём для отладки от ARM и JTAG/SWD-кабель.
- Порты ввода/вывода LM3S3748 выведены на маркированные внешние контактные площадки.
- USB-кабели и измерительные щупы осциллографа, необходимые для работы приложения быстрого старта.
- USB-накопитель.
- Компакт-диск, содержащий:
  - оценочную версию сред программирования, полную документацию, руководство по быстрому старту и исходный код;
  - набор библиотек StellarisWare, включая библиотеку драйверов периферии и пример исходного кода.

Информация для заказа оценочных наборов

Код	Описание
EKK-LM3S3748	Оценочный набор Stellaris LM3S3748 USB Host/Device для Keil RealView MDK-ARM (ограничение размера кода — 32 Кбайт)
EKI-LM3S3748	Оценочный набор Stellaris LM3S3748 для IAR Systems Embedded Workbench (ограничение размера кода — 32 Кбайт)
EKC-LM3S3748	Оценочный набор Stellaris LM3S3748 для CodeSourcery G++ (30-дневное ограничение)
EKT-LM3S3748	Оценочный набор Stellaris LM3S3748 для Code Red Technologies Red Suite (90-дневное ограничение)
EKS-LM3S3748	Оценочный набор Stellaris LM3S3748 для Code Composer Studio™ от компании Texas Instruments (полная версия с привязкой к плате)

Оценочный набор Stellaris LM3S9B92 Ethernet+CAN



Оценочный набор Stellaris LM3S9B92 Ethernet+CAN

Имея в своём составе две платы, на одной из которых находится микроконтроллер LM3S9B92 с интерфейсами Ethernet+USB-OTG+CAN, а на другой реализован интерфейс внутрисхемной отладки BD-ICDI, оценочный набор Stellaris LM3S9B92 предоставляет недорогую, компактную и универсальную платформу для оценки микроконтроллеров Stellaris на базе ARM Cortex-M3 с интерфейсами Ethernet, USB и CAN. В оценочной плате используется микроконтроллер LM3S9B92, который позволяет реализовывать сложные алгоритмы управления и имеет 8 ШИМ-выходов для обеспечения управления приводом и снабжения энергией и 2 модуля квадратурного энкодера (QEI) для подключения импульсных датчиков положения. Кроме того, микроконтроллер LM3S9B92 способен работать с внешним 16-МГц кварцевым генератором, который обеспечивает основную тактовую частоту, используемую для непосредственного тактирования ARM-ядра или внутренней схемы ФАПЧ для увеличения частоты тактового сигнала ядра до 80 МГц. Для тактирования Ethernet-контроллера используется 25-МГц кварцевый генератор. Микроконтроллер LM3S9B92 также имеет встроенный LDO-стабилизатор напряжения, который обеспечивает питание внутренних цепей.

Информация для заказа оценочных наборов

Код	Описание
EKK-LM3S9B92	Недорогой оценочный набор Stellaris LM3S9B92 для Keil RealView MDK-ARM (ограничение размера кода — 32 Кбайт)
EKI-LM3S9B92	Недорогой оценочный набор Stellaris LM3S9B92 для IAR Systems Embedded Workbench (ограничение размера кода — 32 Кбайт)
EKC-LM3S9B92	Недорогой оценочный набор Stellaris LM3S9B92 для CodeSourcery G++ (30-дневное ограничение)
EKT-LM3S9B92	Недорогой оценочный набор Stellaris LM3S9B92 для Code Red Technologies Red Suite (90-дневное ограничение)
EKS-LM3S9B92	Оценочный набор Stellaris LM3S9B92 для Code Composer Studio™ от компании Texas Instruments (полная версия с привязкой к плате)

Таблица И.5. **Микроконтроллеры с интерфейсами USB и CAN**

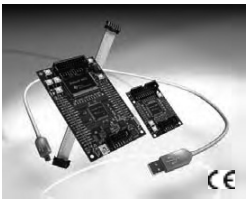
Характеристика			LM3S5632	LM3S5651	LM3S5652	LM3S5656	LM3S5662	LM3S5732	LM3S5737	LM3S5739	LM3S5747	LM3S5749	LM3S5752	LM3S5762	LM3S5791
Память	Флэш-память, Кбайт		128	128	128	128	128	128	128	128	128	128	128	128	128
	ОЗУ, Кбайт		32	32	32	32	32	64	64	64	64	64	64	64	64
	Библиотеки в ПЗУ		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	DMA		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	SAFERTOS™		—	—	—	—	—	—	—	—	—	—	—	—	—
Ядро	Макс. такт. част., МГц		50	80	50	80	50	50	50	50	50	50	50	50	80
	Встроенный прецизионный генератор		—	✓	—	✓	—	—	—	—	—	—	—	—	✓
	MPU		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Таймеры	SysTick (24-битный)		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Таймеры общего назначения		3	4	3	4	3	3	3	4	3	4	3	3	4
	Часы реал. времени		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Сторожевой таймер		1	2	1	2	1	1	1	1	1	1	1	1	2
	Управление приводом	Модуль ШИМ	—	6	—	6	6	—	—	—	6	8	—	6	8
		Вход защиты ШИМ	—	4	—	4	1	—	—	—	1	4	–	1	4
		Генератор «мёртвого» времени	—	✓	—	✓	✓	—	—	—	✓	✓	–	✓	✓
		Модуль CCP	5	8	6	8	5	5	3	8	2	5	6	5	8
		QEI-каналы	—	2	—	1	—	—	—	—	—	1	—	—	2
Интерфейс внешней периферии			—	—	—	—	—	—	—	—	—	—	—	—	✓
Последовательные интерфейсы	Ethernet	10/100 MAC+PHY	—	—	—	—	—	—	—	—	—	—	—	—	—
		IEE 1588	—	—	—	—	—	—	—	—	—	—	—	—	—
	CAN MAC		1	2	1	1	1	1	1	1	1	2	1	1	2
	USB D, H или O		H	O	O	O	O	H	H	H	H	H	O	O	O
	UART		2	3	1	3	1	2	1	3	1	2	1	1	3
	I²C		2	2	1	2	–	2	2	2	1	2	1	–	2
	SSI/SPI		1	2	1	2	1	1	2	2	1	2	1	1	2
	I²S		—	✓	—	—	—	—	—	—	—	—	—	—	✓
Аналоговая периферия	АЦП (10- бит)	Модули АЦП	1	2	1	2	1	1	1	1	1	1	1	1	2
		Каналы АЦП	6	16	6	8	4	6	8	8	8	8	6	4	16
		Быстродействие АЦП (выб/с)	500K	1M	500K	1M	500K	500K	500K	500K	500K	1M	500K	500K	1M
	Встроенный датчик температуры		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Аналоговые/цифровые компараторы		—/ —	2/ 16	1/ —	2/ 16	—/ —	—/ —	—/ —	2/ —	—/ —	2/ —	1/ —	—/ —	3/ 16
Порты ввода/вывода общ. назнач. (совместимы с 5 В)			1…33	0…67	0…33	0…33	0…33	1…33	27…61	12…61	27…61	0…61	0…33	0…33	0…72
Режим энергосбережения с резерв. от батареи			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	—
LDO-стабилизатор напряжения			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Рабочая температура			I	I	I	I	I	I	I	I	I	I	I	I	I
Тип корпуса			64-LQFP	100-LQFP	64-LQFP	64-LQFP	64-LQFP	64-LQFP	100-LQFP	100-LQFP	100-LQFP	100-LQFP	64-LQFP	64-LQFP	100-LQFP
Серийное производство (P), опытные образцы (S)			P	S	P	S	P	P	P	P	P	P	P	P	S

Таблица И.5. **Микроконтроллеры с интерфейсами USB и CAN** *(продолжение)*

Характеристика			LM3S5951	LM3S5956	LM3S5B91	LM3S5K31	LM3S5K36	LM3S5P31	LM3S5P36	LM3S5P51	LM3S5P56	LM3S5R31	LM3S5R36	LM3S5T36	LM3S5Y36
Память	Флэш-память, Кбайт		256	256	256	128	128	64	64	64	64	256	256	32	16
	ОЗУ, Кбайт		64	64	96	24	24	24	24	24	24	48	48	12	8
	Библиотеки в ПЗУ		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	DMA		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	SAFERTOS™		—	—	—	—	—	—	—	—	—	—	—	—	—
Ядро	Макс. такт. част., МГц		80	80	80	80	80	80	80	80	80	80	80	80	80
	Встроенный прецизионный генератор		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	MPU		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Таймеры	SysTick (24-битный)		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Таймеры общего назначения		4	4	4	3	3	3	3	4	4	4	4	3	3
	Часы реал. времени		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Сторожевой таймер		2	2	2	2	2	2	2	2	2	2	2	2	2
	Управление приводом	Модуль ШИМ	6	6	8	6	6	6	6	6	6	8	8	6	6
		Вход защиты ШИМ	4	4	4	4	4	4	4	4	4	4	4	4	4
		Генератор «мёртвого» времени	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		Модуль CCP	8	8	8	6	6	6	6	8	8	8	8	6	6
		QEI-каналы	2	1	2	2	1	2	1	2	1	2	1	1	1
Интерфейс внешней периферии			—	—	✓	—	—	—	—	—	—	✓	—	—	—
Последовательные интерфейсы	Ethernet	10/100 MAC+PHY	—	—	—	—	—	—	—	—	—	—	—	—	—
		IEE 1588	—	—	—	—	—	—	—	—	—	—	—	—	—
	CAN MAC		2	1	2	1	1	1	1	2	1	1	1	1	1
	USB D, H или O		O	O	O	D	D	D	D	O	O	D	D	D	D
	UART		3	3	3	3	3	3	3	3	3	3	3	3	3
	I^2C		2	2	2	2	2	2	2	2	2	2	2	2	2
	SSI/SPI		2	2	2	2	2	2	2	2	2	2	2	2	2
	I^2S		✓	—	✓	—	—	—	—	✓	—	✓	—	—	—
Аналоговая периферия	АЦП (10-бит)	Модули АЦП	2	2	2	2	2	2	2	2	2	2	2	2	2
		Каналы АЦП	16	8	16	16	8	16	8	16	8	16	8	8	8
		Быстродействие АЦП (выб/с)	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M	1M
	Встроенный датчик температуры		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Аналоговые/цифровые компараторы		2/ 16	2/ 16	3/ 16	2/ 16	2/ 16	2/ 16	2/ 16	2/ 16	2/ 16	2/ 16	2/ 16	2/ 16	2/ 16
Порты ввода/вывода общ. назнач. (совместимы с 5 В)			0…67	0…33	0…72	0…67	0…33	0…67	0…33	0…67	0…33	0…67	0…33	0…33	0…33
Режим энергосбережения с резерв. от батареи			✓	✓	—	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LDO-стабилизатор напряжения			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Рабочая температура			I	I	I	I	I	I	I	I	I	I	I	I	I
Тип корпуса			100-LQFP	64-LQFP	100-LQFP	100-LQFP	64-LQFP	100-LQFP	64-LQFP	100-LQFP	64-LQFP	100-LQFP	64-LQFP	64-LQFP	64-LQFP
Серийное производство (P), опытные образцы (S)			S	S	S	S	S	S	S	S	S	S	S	S	S

[a] ШИМ-управление приводом выполняется с помощью специальных аппаратных средств (выводы PWM) или с помощью функций управления приводом, имеющихся у таймеров общего назначения (выводы CCP). Подробности см. в технической документации. [b] Минимум — число выводов, предназначенных для портов ввода/вывода общего назначения; дополнительные выводы доступны, если не используется какая-либо периферия. Подробности см. в технической документации. [c] Промышленный диапазон (I): –40…+85°C, расширенный диапазон (E): –40…+105°C.

И.9. Микроконтроллеры с интерфейсом CAN



Микроконтроллеры Stellaris на базе ARM Cortex-M3 серии LM3S2000 сочетают в себе возможность работы в промышленных сетях, расширенные порты ввода/вывода общего назначения и увеличенный объём встроенной памяти, а также обладают низким энергопотреблением, оптимизированным для приложений с резервным питанием от батареи. Микроконтроллеры серии LM3S2000 (**Табл. И.6**), разработанные для приложений с интерфейсом CAN, работают по технологии Bosch CAN 2.0 A/B, которая является «золотым» стандартом для промышленных сетей, предназначенных для связи на короткие расстояния.

Оценочный набор Stellaris LM3S2965 CAN

Оценочный набор Stellaris LM3S2965 обеспечивает компактную и универсальную платформу оценки микроконтроллеров Stellaris на базе ARM Cortex-M3. С помощью двух плат, на одной из которых установлен микроконтроллер LM3S2965 с интерфейсом CAN, а на другой — микроконтроллер LM3S2110 с интерфейсом CAN, оценочный набор позволяет подготовить CAN-сеть к работе. Приложение для быстрого старта демонстрирует передачу и приём CAN-пакетов между двумя оценочными платами. Плата LM3S2965 также имеет внутрисхемный интерфейс отладки ICDI, который позволяет осуществлять аппаратную отладку не только для установленного на плате микроконтроллера Stellaris, но и для любой платы на базе микроконтроллера Stellaris. Оценочный набор содержит все кабели, программное обеспечение и документацию, необходимые для быстрой разработки и запуска приложений для микроконтроллеров Stellaris. Кроме того, примеры приложений, демонстрирующие использование различных ОС реального времени и коммерческих CAN-стеков, можно загрузить на **www.ti.com/stellaris_lm3s2965**.

Особенности оценочного набора Stellaris LM3S2965 CAN

- Полностью готовая к работе сеть CAN с приложением для быстрого старта, которое обеспечивает CAN-сеть и CAN-трафик.
- Оценочная плата LM3S2965 CAN и отдельная плата LM3S2110 CAN Device.
- Микроконтроллеры Stellaris LM3S2965 и LM3S2110, каждый из которых содержит MAC-контроллер CAN.
- Простая установка: USB-кабель обеспечивает последовательную связь, отладку и питание.
- Графический OLED-дисплей с разрешением 128×64 точек и 16 градациями серого.
- Пользовательский светодиод, переключатели и кнопки.
- Электромагнитный динамик.
- Все порты ввода/вывода LM3S2965 и LM3S2110 выведены на маркированные внешние контактные площадки.
- Стандартный 20-контактный JTAG-разъём для отладки от ARM.
- Плоский CAN-кабель, USB-кабель и JTAG-кабель.
- Компакт-диск, содержащий:
  - оценочную версию сред разработки;
  - руководство по быстрому старту и исходный код;
  - полную документацию.

Информация для заказа оценочных наборов

Код	Описание
EKK-LM3S2965	Оценочный набор Stellaris LM3S2965 CAN для Keil RealView MDK-ARM (ограничение размера кода — 32 Кбайт)
EKI-LM3S2965	Оценочный набор Stellaris LM3S2965 CAN для IAR Systems Embedded Workbench (ограничение размера кода — 32 Кбайт)
EKC-LM3S2965	Оценочный набор Stellaris LM3S2965 CAN для CodeSourcery G++ (30-дневное ограничение)
EKT-LM3S2965	Оценочный набор Stellaris LM3S2965 CAN для Code Red Technologies Red Suite (90-дневное ограничение)
EKS-LM3S2965	Оценочный набор Stellaris LM3S2965 для Code Composer Studio™ от компании Texas Instruments (полная версия с привязкой к плате)

Таблица И.6. **Микроконтроллеры серии LM3S2000**

Характеристики			LM3S2110	LM3S2139	LM3S2276	LM3S2410	LM3S2412	LM3S2432	LM3S2533	LM3S2601	LM3S2608	LM3S2616
Память		Флэш-память, Кбайт	64	64	64	96	96	96	96	128	128	128
		ОЗУ, Кбайт	16	16	32	32	32	32	64	32	32	16
		Библиотеки в ПЗУ	—	—	1	—	—	—	—	—	—	✓
		DMA	—	—	1	—	—	—	—	—	—	✓
		SAFERTOS™	—	—	—	—	—	—	—	—	—	—
Ядро		Макс. такт. част., МГц	25	25	50	25	25	50	50	50	50	50
		Встроенный прецизионный генератор	—	—	—	—	—	—	—	—	—	—
		MPU	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Таймеры		SysTick (24-битный)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		Таймеры общего назначения	3	3	3	3	3	3	4	4	4	4
		Часы реального времени	✓	✓	✓	✓	✓	✓	✓	✓	✓	—
		Сторожевой таймер	1	1	1	1	1	1	1	1	1	1
	Управление приводом	Модуль ШИМ	2	—	8	—	2	2	6	—	—	6
		Вход защиты ШИМ	1	—	3	—	1	1	1	—	—	1
		Генератор «мёртвого» времени	✓	—	✓	—	✓	✓	✓	—	—	✓
		Модуль CCP	4	6	1	4	4	4	6	8	8	–
		QEI-каналы	—	—	—	—	—	—	—	—	—	1
Интерфейс внешней периферии			—	—	—	—	—	—	—	—	—	—
Последовательные интерфейсы	Ethernet	10/100 MAC+PHY	—	—	—	—	—	—	—	—	—	—
		IEE 1588	—	—	—	—	—	—	—	—	—	—
	CAN MAC		1	1	1	1	1	1	1	1	1	1
	USB D, H или O		—	—	—	—	—	—	—	—	—	—
	UART		1	2	1	1	2	2	2	3	2	1
	I^2C		1	1	1	—	1	1	1	2	2	1
	SSI/SPI		1	1	1	1	1	1	1	2	2	—
	I^2S		—	—	—	—	—	—	—	—	—	—
Аналоговая периферия	АЦП (10- бит)	Модули АЦП	—	1	1	—	1	1	1	—	1	1
		Каналы АЦП	—	4	6	—	3	3	3	—	8	6
		Быстродействие АЦП (выб/с)	—	250K	1M	—	250K	250K	250K	—	500K	1M
	Встроенный датчик температуры		—	✓	✓	—	✓	✓	✓	—	✓	✓
	Аналоговые/цифровые компараторы		3/ —	3/ —	—/ —	2/ —	2/ —	2/ —	3/ —	2/ —	2/ —	2/ —
Порты ввода/вывода общ. назнач. (совместимы с 5 В)			11…40	26…56	0…33	37…60	20…49	5…34	11…48	21…60	15…52	1…33
Режим энергосбережения с резерв. от батареи			—	—	✓	—	—	—	✓	✓	✓	✓
LDO-стабилизатор напряжения			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Рабочая температура			I/E	I/E	I	I/E	I/E	I/E	I/E	I/E	I/E	I
Тип корпуса			100LQFP, 108BGA	100LQFP, 108BGA	64LQFP	100LQFP, 108BGA	100LQFP, 108BGA	100LQFP, 108BGA	100LQFP, 108BGA	100LQFP, 108BGA	100LQFP, 108BGA	64LQFP
Серийное производство (P), опытные образцы (S)			P	P	P	P	P	P	P	P	P	P

LM3S2620	**LM3S2637**	**LM3S2651**	**LM3S2671**	**LM3S2678**	**LM3S2730**	**LM3S2739**	**LM3S2776**	**LM3S2793**	**LM3S2911**	**LM3S2918**	**LM3S2919**	**LM3S2939**	**LM3S2948**	**LM3S2950**	**LM3S2965**	**LM3S2B93**
128	128	128	128	128	128	128	128	128	256	256	256	256	256	256	256	256
32	32	32	32	32	64	64	64	64	64	64	64	64	64	64	64	96
—	—	—	✓	✓	—	—	✓	✓	—	—	—	—	—	—	—	✓
—	—	—	✓	✓	—	—	✓	✓	—	—	—	—	—	—	—	✓
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
25	50	50	50	50	50	50	50	80	50	50	50	50	50	50	50	80
—	—	—	—	—	—	—	—	✓	—	—	—	—	—	—	—	✓
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
4	4	4	4	4	3	3	3	4	4	4	4	3	4	4	4	4
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
1	1	1	1	1	1	1	1	2	1	1	1	1	1	1	1	2
4	—	4	2	4	—	6	8	8	—	—	—	4	—	6	6	8
1	—	1	1	2	—	1	3	4	—	—	—	1	—	1	1	4
✓	—	✓	✓	✓	—	✓	✓	✓	—	—	—	✓	—	✓	✓	✓
6	6	6	2	2	4	6	1	8	8	8	8	4	8	6	6	8
1	—	—	—	1	—	1	—	2	—	—	—	1	—	1	2	2
—	—	—	—	—	—	—	—	✓	—	—	—	—	—	—	—	✓
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
2	1	1	1	1	1	1	1	2	1	1	1	1	2	2	2	2
—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—
1	2	3	1	1	1	2	1	3	3	2	2	3	3	3	3	3
1	1	1	1	—	—	1	1	2	2	2	2	1	1	1	2	2
1	1	2	1	1	1	1	1	2	2	2	2	1	2	2	2	2
—	—	—	—	—	—	—	—	✓	—	—	—	—	—	—	—	✓
—	1	1	1	1	—	1	1	2	—	1	1	1	1	—	1	2
—	4	4	4	8	—	4	6	16	—	8	8	3	8	—	4	16
—	500K	500K	500K	500K	—	500K	1M	1M	—	500K	500K	500K	1M	—	1M	1M
—	✓	✓	✓	✓	—	✓	✓	✓	—	—	✓	✓	✓	—	✓	✓
3/ —	3/ —	1/ —	3/ —	—/ —	2/ —	1/ —	—/ —	3/ 16	2/ —	2/ —	2/ —	3/ —	3/ —	3/ —	3/ —	3/ 16
12…52	15…46	16…53	3…33	1…33	37…60	20…56	0…33	0…67	21…60	15…52	15…52	18…57	12…52	10…60	3…56	0…67
✓	✓	✓	—	—	—	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
I/E	I/E	I/E	I	I	I/E	I/E	I	I	I/E	I/E	I	I/E	I/E	I/E	I/E	I
100LQFP, 108BGA	100LQFP, 108BGA	100LQFP, 108BGA	64LQFP	64LQFP	100LQFP, 108BGA	100LQFP, 108BGA	64LQFP	100LQFP	100LQFP, 108BGA	100LQFP, 108BGA	100LQFP	100LQFP, 108BGA	100LQFP, 108BGA	100LQFP, 108BGA	100LQFP, 108BGA	100LQFP
P	P	P	P	P	P	P	P	S	P	P	P	P	P	P	P	S

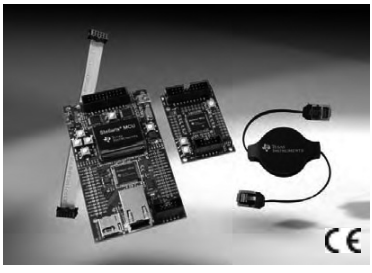
[a] ШИМ-управление приводом выполняется с помощью специальных аппаратных средств (выводы PWM) или с помощью функций управления приводом, имеющихся у таймеров общего назначения (выводы CCP). Подробности см. в технической документации. [b] Минимум — число выводов, предназначенных для портов ввода/вывода общего назначения; дополнительные выводы доступны, если не используется какая-либо периферия. Подробности см. в технической документации. [c] Промышленный диапазон (I): –40…+85°C, расширенный диапазон (E): –40…+105°C. [d] 108-выводной корпус BGA и 64-выводной корпус LQFP предлагаются только для промышленного диапазона рабочих температур.

И.10. Микроконтроллеры с интерфейсами Ethernet и CAN

Микроконтроллеры Stellaris на базе ARM Cortex-M3 серии LM3S8000 сочетают в себе возможность работы в промышленных сетях, расширенные порты ввода/вывода общего назначения и увеличенный объём встроенной памяти, а также обладают низким энергопотреблением, оптимизированным для приложений с резервным питанием от батареи. Микроконтроллеры серии LM3S8000 являются первыми в мире микроконтроллерами, сочетающими полностью интегрированное 10/100-Мбит/с Ethernet-решение и сетевую технологию Bosch CAN с ARM-архитектурой. Микроконтроллеры серии LM3S8000 (**Табл. И.7**) содержат до трёх контроллеров CAN 2.0 A/B, Ethernet MAC-контроллер (MAC) и трансивер (PHY). Кроме того, отдельные микроконтроллеры серии LM3S8000 обеспечивают аппаратную поддержку протокола Precision Time Protocol согласно IEEE 1588.

Оценочный набор Stellaris LM3S8962 Ethernet+CAN

Оценочный набор Stellaris LM3S8962 обеспечивает компактную и универсальную платформу для оценки микроконтроллеров Stellaris на базе ARM Cortex-M3 с интерфейсами Ethernet и CAN. С помощью двух плат, на одной из которых содержится микроконтроллер LM3S8962 с интерфейсами Ethernet и CAN, а на другой — микроконтроллер LM3S2110 с интерфейсом CAN, оценочный набор позволяет подготовить сеть CAN к работе. Набор содержит также два примера демонстрационного приложения встраиваемого веб-сервера. Готовое к работе приложение для быстрого старта включает в себя встраиваемый веб-сервер, использующий Ethernet-стек Open Source lwIP, и демонстрирует передачу и приём CAN-пакетов между двумя оценочными платами. Набор также содержит веб-сервер с ОС реального времени FreeRTOS.org™ и Ethernet-стек Open Source uIP. Плата LM3S8962 также имеет внутрисхемный интерфейс отладки ICDI, который позволяет осуществлять аппаратную отладку не только для установленного на плате микроконтроллера Stellaris, но и для любой платы на базе микроконтроллера Stellaris. Оценочный набор содержит все кабели, программное обеспечение и документацию, необходимые для быстрой разработки и запуска приложений для микроконтроллеров Stellaris. Кроме того, примеры приложений, демонстрирующие использование различных ОС реального времени и коммерческих стеков Ethernet и CAN, можно загрузить на **www.ti.com/stellaris_lm3s8962**.



Особенности оценочного набора Stellaris LM3S8962

- Полностью готовая к работе сеть Ethernet+CAN с приложением для быстрого старта, которое одновременно обеспечивает трафик Ethernet- и CAN-сети.
- Оценочная плата LM3S8962 Ethernet+CAN и отдельная плата LM3S2110 CAN Device.
- Микроконтроллер Stellaris LM3S8962 с полностью интегрированным 10/100-Мбит/с Ethernet (MAC+PHY) и CAN MAC.
- Микроконтроллер Stellaris LM3S2110 с полностью интегрированным CAN MAC.
- Простая установка: USB-кабель обеспечивает последовательную связь, отладку и питание.
- Графический OLED-дисплей с разрешением 128×64 точек и 16 градациями серого.
- Пользовательский светодиод, переключатели и кнопки.
- Электромагнитный динамик.
- Все порты ввода/вывода LM3S8962 и LM3S2110 выведены на маркированные внешние контактные площадки.
- Стандартный 20-контактный JTAG-разъём для отладки от ARM.
- Убирающийся Ethernet-кабель, плоский CAN-кабель, USB-кабель и JTAG-кабель.
- Компакт-диск, содержащий:
  - оценочную версию сред программирования;
  - полную документацию;
  - руководство по быстрому старту и исходный код;
  - набор библиотек StellarisWare, включая библиотеку драйверов периферии и пример исходного кода.

Информация для заказа оценочных наборов

Код	Описание
EKK-LM3S8962	Оценочный набор Stellaris LM3S8962 для Keil RealView MDK-ARM (ограничение размера кода — 32 Кбайт)
EKI-LM3S8962	Оценочный набор Stellaris LM3S8962 для IAR Systems Embedded Workbench (ограничение размера кода — 32 Кбайт)
EKC-LM3S8962	Оценочный набор Stellaris LM3S8962 для CodeSourcery G++ (30-дневное ограничение)
EKT-LM3S8962	Оценочный набор Stellaris LM3S8962 для Code Red Technologies Red Suite (90-дневное ограничение)
EKS-LM3S8962	Оценочный набор Stellaris LM3S8962 для Code Composer Studio™ от компании Texas Instruments (полная версия с привязкой к плате)

Таблица И.7. **Микроконтроллеры серии LM3S8000**

Характеристики			LM3S8530	LM3S8538	LM3S8630	LM3S8730	LM3S8733	LM3S8738	LM3S8930	LM3S8933	LM3S8938	LM3S8962	LM3S8970	LM3S8971
Память	Флэш-память, Кбайт		96	96	128	128	128	128	256	256	256	256	256	256
	ОЗУ, Кбайт		64	64	32	64	64	64	64	64	64	64	64	64
	Библиотеки в ПЗУ		—	—	—	—	—	—	—	—	—	—	—	—
	DMA		—	—	—	—	—	—	—	—	—	—	—	—
	SAFERTOS™		—	—	—	—	—	—	—	—	—	—	—	—
Ядро	Макс. такт. частота, МГц		50	50	50	50	50	50	50	50	50	50	50	50
	Встроенный прецизионный генератор		—	—	—	—	—	—	—	—	—	—	—	—
	MPU		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Таймеры	SysTick (24-битный)		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Таймеры общего назначения		4	4	4	4	4	4	4	4	4	4	4	4
	Часы реального времени		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	Сторожевой таймер		1	1	1	1	1	1	1	1	1	1	1	1
	Управление приводом	Модуль ШИМ	—	—	—	—	—	—	—	—	—	6	—	6
		Вход защиты ШИМ	—	—	—	—	—	—	—	—	—	1	—	1
		Генератор «мёртвого» времени	—	—	—	—	—	—	—	—	—	✓	—	✓
		Модуль CCP	2	4	2	2	4	6	2	4	6	2	2	6
		QEI-каналы	2	—	—	—	—	—	—	—	—	2	–	1
Интерфейс внешней периферии			—	—	—	—	—	—	—	—	—	—	—	—
Последовательные интерфейсы	Ethernet	10/100 MAC+PHY	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
		IEE 1588	—	✓	—	✓	—	—	—	✓	✓	✓	✓	–
	CAN MAC		3	1	1	1	1	1	2	1	1	1	3	1
	USB D, H или O		—	—	—	—	—	—	—	—	—	—	—	—
	UART		1	2	2	2	2	3	1	2	3	2	2	1
	I^2C		1	1	1	1	1	1	1	1	2	1	1	—
	SSI/SPI		2	1	1	1	1	2	1	1	1	1	2	1
	I^2S		—	—	—	—	—	—	—	—	—	—	—	—
Аналоговая периферия	АЦП (10- бит)	Модули АЦП	—	1	—	—	1	1	—	1	1	1	—	1
		Каналы АЦП	—	8	—	—	4	8	—	4	8	4	—	8
		Быстродействие АЦП (выб/с)	—	1M	—	—	500K	500K	—	1M	1M	500K	—	1M
	Встроенный датчик температуры		—	✓	—	—	✓	✓	—	✓	✓	✓	—	✓
	Аналоговые/цифровые компараторы		—/—	3/—	—/—	—/—	3/—	1/—	—/—	3/—	3/—	1/—	—/—	1/—
Порты ввода/вывода общ. назнач. (совместимы с 5 В)			8...35	7...36	10...31	11...32	5...35	4...38	13...34	6...36	3...38	5...42	17...46	4…38
Режим энергосбережения с резерв. от батареи			—	—	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
LDO-стабилизатор напряжения			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Рабочая температура			I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E	I/E
Тип корпуса			100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA	100 LQFP 108 BGA
Серийное производство (P), опытные образцы (S)			P	P	P	P	P	P	P	P	P	P	P	P

[a] ШИМ-управление приводом выполняется с помощью специальных аппаратных средств (выводы PWM) или с помощью функций управления приводом, имеющихся у таймеров общего назначения (выводы CCP). Подробности см. в технической документации. [b] Минимум — число выводов, предназначенных для портов ввода/вывода общего назначения; дополнительные выводы доступны, если не используется какая-либо периферия. Подробности см. в технической документации. [c] Промышленный диапазон (I): –40…+85°C, расширенный диапазон (E): –40…+105°C. [d] 108-выводной корпус BGA предлагается только для промышленного диапазона рабочих температур.

И.11. Микроконтроллеры с интерфейсами Ethernet, USB и CAN

Микроконтроллеры Stellaris на базе ARM Cortex-M3 серии LM3S9000 обеспечивают высокую производительность и сочетают в себе возможность работы в промышленных сетях и расширенные возможности интерфейсов периферии, а также обладают низким энергопотреблением, оптимизированным для приложений с резервным питанием от батареи. Микроконтроллеры серии LM3S9000 являются первыми в мире микроконтроллерами, сочетающими полностью интегрированное 10/100-Мбит/с Ethernet-решение, модуль USB On-the-Go и сетевую технологию Bosch CAN, совместимую с ARM-архитектурой. Микроконтроллеры серии LM3S9000 (**Табл. И.8**) содержат до двух контроллеров CAN 2.0 A/B, Ethernet MAC-контроллер (MAC) и трансивер (PHY), а также полноскоростной модуль USB с функциональностью OTG или Host/Device. Все микроконтроллеры серии LM3S9000 содержат два отдельных блока АЦП, набор библиотек StellarisWare™, записанный в ПЗУ, включая библиотеку драйверов периферии и системный загрузчик, криптографические AES-таблицы, а также схему обнаружения ошибок CRC. Кроме того, отдельные микроконтроллеры серии LM3S9000 также содержат в ПЗУ ядро SafeRTOS™ и обеспечивают аппаратную поддержку протокола Precision Time Protocol согласно IEEE 1588.

Микроконтроллеры серии LM3S9000 также содержат встроенный 16-МГц генератор повышенной точности с возможностью программной настройки и второй сторожевой таймер с независимым тактовым генератором. Отдельные микроконтроллеры серии LM3S9000 также снабжены гибким интерфейсом внешней периферии (EPI), который представляет собой специально выделенную параллельную шину (до 32 бит) для внешней периферии, которая поддерживает SDRAM, SRAM/флэш-память и связь Machine-to-Machine (M2M) (со скоростью до 150 Мбайт/с).

Оценочный набор Stellaris LM3S9B92 Ethernet+USB+CAN

С помощью двух плат, на одной из которых содержится микроконтроллер LM3S9B92 с интерфейсами Ethernet, USB-OTG и CAN, а другая реализует интерфейс внутрисхемной отладки, оценочный набор Stellaris LM3S9B92 обеспечивает недорогую, компактную и универсальную платформу для оценки микроконтроллеров Stellaris на базе ARM Cortex-M3 с интерфейсами Ethernet, USB и CAN. В оценочной плате используется микроконтроллер LM3S9B92, который позволяет реализовывать сложные алгоритмы управления и имеет 8 ШИМ-выходов для обеспечения управления приводом и источниками питания и 2 модуля квадратурного энкодера (QEI) для подключения импульсных датчиков положения. Микроконтроллер LM3S9B92 также способен работать с внешним 16-МГц кварцевым генератором, обеспечивающим основную тактовую частоту, которая используется для непосредственного тактирования ARM-ядра или внутренней схемы ФАПЧ для увеличения частоты тактового сигнала ядра до 80 МГц. Для тактирования Ethernet-контроллера используется 25-МГц кварцевый генератор. Микроконтроллер LM3S9B92 также имеет встроенный LDO-стабилизатор напряжения, который обеспечивает питание внутренних цепей.

Информация для заказа

Код	Описание
EKK-LM3S9B92	Недорогой оценочный набор Stellaris LM3S9B92 для Keil RealView MDK-ARM (ограничение размера кода — 32 Кбайт)
EKI-LM3S9B92	Недорогой оценочный набор Stellaris LM3S9B92 для IAR Systems Embedded Workbench (ограничение размера кода — 32 Кбайт)
EKC-LM3S9B92	Недорогой оценочный набор Stellaris LM3S9B92 для CodeSourcery G++ (30-дневное ограничение)
EKT-LM3S9B92	Недорогой оценочный набор Stellaris LM3S9B92 для Code Red Technologies Red Suite (90-дневное ограничение)
EKS-LM3S9B92	Оценочный набор Stellaris LM3S9B92 для Code Composer Studio™ от компании Texas Instruments (полная версия с привязкой к плате)

Оценочный набор Stellaris LM3S9B90 Ethernet+USB-OTG+CAN

С помощью двух плат, на одной из которых содержится микроконтроллер LM3S9B90 с интерфейсами Ethernet, USB-OTG и CAN, а другая реализует интерфейс внутрисхемной отладки, оценочный набор Stellaris LM3S9B90 обеспечивает недорогую, компактную и универсальную платформу для оценки микроконтроллеров Stellaris на базе ARM Cortex-M3 с интерфейсами Ethernet, USB и CAN. В оценочной плате используется микроконтроллер LM3S9B90, который имеет модуль Hibernation для эффективного снижения энергопотребления устройства во время увеличенного периода бездействия. Микроконтроллер LM3S9B90 также способен работать с внешним 16-МГц кварцевым генератором, обеспечивающим основную тактовую частоту, которая используется для непосредственного тактирования ARM-ядра или внутренней схемы ФАПЧ для увеличения частоты тактового сигнала ядра до 80 МГц. Для тактирования Ethernet-контроллера используется 25-МГц кварцевый генератор, а для часов реального времени — 4.194304-МГц кварцевый генератор. Микроконтроллер LM3S9B90 также имеет встроенный LDO-стабилизатор напряжения, который обеспечивает питание внутренних цепей.

Информация для заказа

Код	Описание
EKK-LM3S9B90	Недорогой оценочный набор Stellaris LM3S9B90 для Keil RealView MDK-ARM (ограничение размера кода — 32 Кбайт)
EKI-LM3S9B90	Недорогой оценочный набор Stellaris LM3S9B90 для IAR Systems Embedded Workbench (ограничение размера кода — 32 Кбайт)
EKC-LM3S9B90	Недорогой оценочный набор Stellaris LM3S9B90 для CodeSourcery G++ (30-дневное ограничение)
EKT-LM3S9B90	Недорогой оценочный набор Stellaris LM3S9B90 для Code Red Technologies Red Suite (90-дневное ограничение)
EKS-LM3S9B90	Оценочный набор Stellaris LM3S9B90 для Code Composer Studio™ от компании Texas Instruments (полная версия с привязкой к плате)

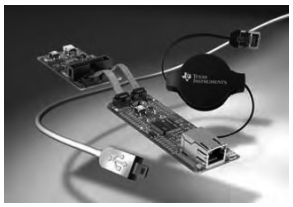
Особенности оценочного набора

- Высокопроизводительный микроконтроллер Stellaris с встроенной памятью большого объёма;
  - 32-битное ядро ARM Cortex-M3;
  - 256-Кбайт основная флэш-память, 96-Кбайт SRAM;
  - набор библиотек StellarisWare, записанный в ПЗУ.
- 10/100-Мбит/с Ethernet-порт с двумя индикаторными светодиодами.
- Полноскоростной порт USB 2.0 OTG.
- Виртуальный последовательный коммуникационный порт.
- Контактные площадки увеличенного размера для портов ввода/вывода общего назначения.

Содержимое оценочного набора

- Оценочная плата (EVB).
- Плата интерфейса внутрисхемной отладки BD-ICDI.
- Кабели:
  - USB-кабель;
  - 10-контактный плоский JTAG-кабель;
  - 8-контактный плоский кабель для питания/UART.
- Компакт-диск, содержащий:
  - полный исходный код, электрическую схему и Гербер-файлы для производства печатной платы;
  - набор библиотек StellarisWare, включая библиотеку драйверов периферии и пример исходного кода;
  - набор оценочных средств разработки программного обеспечения для Stellaris.

Микроконтроллер Stellaris LM3S9B96 с SAFERTOS в ПЗУ



Оценочный набор Stellaris LM3S9B90 Ethernet+USB-OTG+CAN

Оценочный набор Stellaris LM3S9B92 Ethernet+USB-OTG+CAN

Таблица И.8. **Микроконтроллеры серии LM3S9000**

Характеристики			LM3S9790	LM3S9792	LM3S9997	LM3S9B90	LM3S9B92	LM3S9B95	LM3S9B96	LM3S9L97
Память	Флэш-память, Кбайт		128	128	256	256	256	256	256	128
	ОЗУ, Кбайт		64	64	64	96	96	96	96	48
	Библиотеки в ПЗУ		✓	✓	✓	✓	✓	✓	✓	✓
	DMA		✓	✓	✓	✓	✓	✓	✓	✓
	SAFERTOS™		—	—	—	—	—	—	1	—
Ядро	Макс. такт. част., МГц		80	80	80	80	80	80 100	80	80
	Встроенный прецизионный генератор		✓	✓	✓	✓	✓	✓	✓	✓
	MPU		✓	✓	✓	✓	✓	✓	✓	✓
Таймеры	SysTick (24-битный)		✓	✓	✓	✓	✓	✓	✓	✓
	Таймеры общего назначения		4	4	4	4	4	4	4	4
	Часы реального времени		✓	✓	✓	✓	✓	✓	✓	✓
	Сторожевой таймер		2	2	2	2	2	2	2	2
	Управление приводом	Модуль ШИМ	—	8	6	—	8	8	8	6
		Вход защиты ШИМ	—	4	4	—	4	4	4	4
		Генератор «мёртвого» времени	—	✓	✓	—	✓	✓	✓	✓
		Модуль CCP	8	8	8	8	8	8	8	8
		QEI-каналы	—	2	2	—	2	2	2	2
Интерфейс внешней периферии			✓	✓	—	✓	✓	✓	✓	—
Последовательные интерфейсы	Ethernet	10/100 MAC+PHY	✓	✓	✓	✓	✓	✓	✓	✓
		IEEE 1588	—	—	✓	—	—	✓	✓	✓
	CAN MAC		2	2	2	2	2	2	2	2
	USB D, H или O		O	O	O	O	O	O	O	O
	UART		3	3	3	3	3	3	3	3
	I^2C		2	2	2	2	2	2	2	2
	SSI/SPI		2	2	2	2	2	2	2	2
	I^2S		✓	✓	✓	✓	✓	✓	✓	✓
Аналоговая периферия	АЦП (10-бит)	Модули АЦП	2	2	2	2	2	2	2	2
		Каналы АЦП	16	16	16	16	16	16	16	16
		Быстродействие АЦП (выб/с)	1M	1M	1M	1M	1M	1M	1M	1M
	Встроенный датчик температуры		✓	✓	✓	✓	✓	✓	✓	✓
	Аналоговые/цифровые компараторы		3/16	3/16	2/16	3/16	3/16	3/16	3/16	2/16
Порты ввода/вывода общ. назнач. (совместимы с 5 В)			0 … 60	0 … 65	0 … 60	0 … 60	0 … 65	0 … 65	0 … 65	0 … 60
Режим энергосбережения с резерв. от батареи			✓	—	✓	✓	—	—	—	✓
LDO-стабилизатор напряжения			✓	✓	✓	✓	✓	✓	✓	✓
Рабочая температура			I	I	I	I	I	I	I	I
Тип корпуса			100LQFP	100LQFP	100LQFP	100LQFP	100LQFP	100LQFP	100LQFP	100LQFP
Серийное производство (P), опытные образцы (S)			S	S	S	S	S	S	S	S

[a] ШИМ-управление приводом выполняется с помощью специальных аппаратных средств (выводы PWM) или с помощью функций управления приводом, имеющихся у таймеров общего назначения (выводы CCP). Подробности см. в технической документации. [b] Минимум — число выводов, предназначенных для портов ввода/вывода общего назначения; дополнительные выводы доступны, если не используется какая-либо периферия. Подробности см. в технической документации. [c] Промышленный диапазон (I): –40…+85°C, расширенный диапазон (E): –40…+105°C.

И.12. Набор для проектирования на базе микроконтроллера Stellaris LM3S9B96

Набор для проектирования на базе микроконтроллера Stellaris LM3S9B96 (DK-LM3S9B96) — это полнофункциональный набор для проектирования устройств на базе микроконтроллеров серии LM3S9000. Плата для проектирования LM3S9B96 содержит максимально возможный набор периферии для демонстрации возможностей микроконтроллеров и обеспечивает максимальную гибкость за счёт применения конфигурируемых выводов для всех портов ввода/вывода. Плата для проектирования LM3S9B96 предоставляет платформу для оценки приложений с повышенными требованиями к размеру памяти, а также приложений, которые используют новые функциональные возможности микроконтроллеров, например I²S-аудио, расширенный интерфейс периферии (EPI) и обмен данными по интерфейсам Ethernet, USB OTG и CAN. Целевые области применения включают сетевые системы, графические пользовательские интерфейсы (GUI) и приложения, использующие человеко-машинный интерфейс (HMI). Плата для проектирования LM3S9B96 является также полезным инструментом для разработки систем с использованием таких средств, как .NET Micro Framework от Microsoft и Embedded LabView от National Instruments.

Готовое к работе приложение для быстрого старта способно выбирать устройства периферии платы из набора для проектирования DK-LM3S9B96 с помощью демонстрационного меню на сенсорном экране. Возможны различные режимы работы, в том числе демонстрация поддержки USB-мыши; TFTP-сервер для файловой системы, который осуществляет доступ к 1-Мбайт последовательной флэш-памяти; веб-сервер, использующий lwIP TCP/IP-стек; доступ к карте MicroSD; устройство просмотра изображений JPEG; последовательную командную строку и аудиоплеер. Плата для проектирования содержит встроенный внутрисхемный последовательный интерфейс отладки (ICDI), который поддерживает как JTAG-, так и SWD-отладку. Стандартный 20-контактный отладочный разъём от ARM поддерживает несколько способов отладки. Набор также содержит много примеров приложений и полный исходный код.

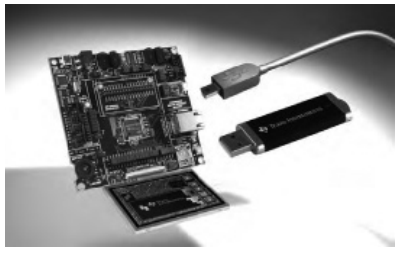
Особенности набора для проектирования Stellaris LM3S9B96

- 3.5-дюймовый цветной графический ЖК-дисплей:
  - TFT ЖК-модуль с разрешением 320×240 точек;
  - резистивный сенсорный интерфейс.
- 80-МГц микроконтроллер LM3S9B96 с 256-Кбайт флэш-памятью, 96-Кбайт SRAM и интегрированными интерфейсами Ethernet MAC+PHY, USB OTG и CAN.
- 8-Мбайт SDRAM (опционально подключаемая по интерфейсу EPI-плата).
- Конфигурируемая плата для сигналов интерфейса внешней периферии (EPI).
- 1-Мбайт последовательная флэш-память.
- Прецизионный источник опорного напряжения 3 В.
- Операционная система SafeRTOS™ в ПЗУ микроконтроллера.
- Стерео I²S-аудиокодек:
  - линейный выход;
  - выход на головные телефоны;
  - вход микрофона;
  - линейный вход.
- CAN-интерфейс.
- Разъём 10/100 BaseT Ethernet.
- Разъём USB OTG:
  - режимы Device, Host и OTG.
- Пользовательский светодиод и кнопка.
- Дисковый потенциометр.
- Разъём карты MicroSD.
- Стандартный 10-контактный JTAG-разъём для отладки от ARM.
- Встроенный интерфейс внутрисхемной отладки (ICDI).
- Виртуальный последовательный порт USB.
- Перемычки для удобного распределения ресурсов портов ввода/вывода.
- Поддержка набора библиотек StellarisWare, включая графическую библиотеку и библиотеку драйверов периферии.

Содержимое набора для проектирования Stellaris LM3S9B96

Набор для проектирования Stellaris DK-LM3S9B96 обеспечивает инженеров средствами, необходимыми для проектирования прототипа готового к работе встраиваемого приложения и включает в себя следующие компоненты:

- плату для проектирования Stellaris® LM3S9B96 с EPI-платой 8-Мбайт SDRAM и конфигурируемой платой для сигналов интерфейса внешней периферии (EPI);
- USB-кабель типа Mini-B (длиной около 1 м) для отладочных функций;
- USB-кабель с вилкой Micro-A и розеткой Std-A (для соединения с флэш-накопителем);
- USB-кабель с вилкой Std-A и вилкой Micro-B (для подсоединения к компьютеру);
- флэш-накопитель USB (128 Мбайт);
- 20-проводной приёмный кабель;
- Ethernet-кабель;
- карта MicroSD;
- 5-В источник питания с международным адаптером;
- компакт-диск с инструментами разработки, документацией и примерами исходного кода:
  - включает оценочные версии инструментов разработки от Keil, IAR, Code Red Technologies и Code Sourcery.



Набор для проектирования Stellaris LM3S9B96

Информация для заказа

Код	Описание
DK-LM3S9B96	Набор для разработки, включая оценочные инструменты от Keil, IAR, Code Red Technologies и CodeSourcery

И.13. Платы расширения набора для проектирования Stellaris LM3S9B96

Плата расширения памяти Stellaris Flash & SRAM Memory Expansion Board

Плата Stellaris Flash & SRAM Memory Expansion Board (DK-LM3S9B96-FS8) компании Texas Instruments представляет собой опциональную плату расширения для отладочной платы Stellaris DK-LM3S9B96. Данная плата расширения подключается напрямую к порту интерфейса внешней периферии (EPI) микроконтроллера Stellaris LM3S9B96, предоставляя в распоряжение пользователя внешнюю флэш-память, внешнее ОЗУ и интерфейс контроллера ЖК-дисплея, а также демонстрируя лёгкость одновременного подключения к микроконтроллеру микросхем внешней памяти и различной периферии посредством гибкого интерфейса EPI, сконфигурированного в режиме Host Bus 8 с мультиплексированием шин адреса/данных. Отладочная плата DK-LM3S9B96 (поставляется отдельно) предоставляет законченную платформу для ознакомления с возможностями микроконтроллеров Stellaris серии LM3S9000. Данная плата имеет богатый набор коммуникационных интерфейсов, таких как 10/100 Ethernet, CAN, Full Speed USB On-The-Go и I²S. Приложение для быстрого старта, демонстрирующее возможности платы расширения, выводит на 3.5-дюймовый цветной VGA-дисплей платы DK-LM3S9B96 JPEG-изображения из ОЗУ и флэш-памяти платы расширения.

Особенности платы Stellaris Flash & SRAM Memory Expansion Board

- 8 Мбайт флэш-памяти.
- 1 Мбайт статического ОЗУ.
- Интерфейс ЖК-дисплея, отображённый на адресное пространство памяти.
- Микросхема EEPROM с интерфейсом I²C объёмом 1 Кбит для хранения конфигурационных параметров.
- Индикатор наличия питания.

Комплект поставки платы Stellaris Flash & SRAM Memory Expansion Board

Вместе с набором для разработки DK-LM3S9B96 плата расширения Stellaris Flash & SRAM Memory Expansion Board представляет собой инструментальное средство, необходимое для разработки и макетирования встраиваемых приложений, в которое входит:

- плата расширения памяти Stellaris DK-LM3S9B96-FS8;
- пластиковые монтажные стойки.

Информация для заказа

Код	Описание
DK-LM3S9B96-FS8	Плата расширения Stellaris Flash & SRAM Memory Expansion Board

Плата расширения Stellaris FPGA Expansion Board

Плата Stellaris FPGA Expansion Board (DK-LM3S9B96-FPGA) компании Texas Instruments представляет собой опциональную плату расширения для отладочной платы Stellaris DK-LM3S9B961. Данная плата расширения подключается напрямую к порту интерфейса внешней периферии (EPI) платы Stellaris DK-LM3S9B96, демонстрируя возможности микроконтроллера семейства Stellaris по организации машинно-машинного (M2M) параллельного интерфейса с высокой пропускной способностью. Подключив данную плату расширения к отладочной плате DK-LM3S9B96, пользователи смогут воспроизводить на большом 3.5-дюймовом ЖК-дисплее отладочной платы видеоизображение, формируемое платой расширения, а также управлять функциями платы расширения. Пользователи могут исследовать различные возможности порта EPI, загружая собственные прошивки в ПЛИС Xilinx Spartan 3E платы расширения. Отладочная плата DK-LM3S9B96 (поставляется отдельно) предоставляет законченную платформу для ознакомления с возможностями микроконтроллеров Stellaris серии LM3S9000. Данная плата имеет богатый набор коммуникационных интерфейсов, таких как 10/100 Ethernet, CAN, Full Speed USB On-The-Go и I²S.

Особенности платы Stellaris FPGA Expansion Board

- ПЛИС Xilinx Spartan 3E объёмом 100 тыс. вентилей.
- Модуль цветной КМОП VGA (640×480) видеокамеры с матрицей 1/13 дюйма.
- 1 Мбайт асинхронного статического ОЗУ с временем доступа 10 нс для организации видеобуфера.
- Микросхема EEPROM с интерфейсом I²C объёмом 1 Кбит для хранения конфигурационных параметров.
- Стандартные разъёмы 1×6 и 2×5 для программирования ПЛИС.
- 8 тестовых контактов ПЛИС предоставляют пользователю 5 линий ввода и 3 линии ввода/вывода.
- Интерфейс EPI работает в режиме GPM D16-A12 на частоте 50 МГц.
- ПО обработки видео имеет следующие возможности:
  - вывод цветного видеоизображения с разрешением QVGA (полноэкранный режим ЖК-дисплея платы DK-LM3S9B96);
  - вывод графического экранного меню поверх видеоизображения;
  - графический пользовательский интерфейс на базе виджетов, использующий возможности сенсорного экрана;
  - пауза/возобновление воспроизведения с сохранением захваченного кадра в формате BMP на SD-карту;
  - управление яркостью, насыщенностью и контрастностью изображения;
  - зеркальное отображение и переворот изображения.

Комплект поставки платы Stellaris FPGA Expansion Board

Вместе с набором для разработки DK-LM3S9B96 плата расширения Stellaris FPGA Expansion Board представляет собой инструментальное средство, необходимое для разработки и макетирования встраиваемых приложений, в которое входит:

- плата расширения Stellaris FPGA Expansion Board с ПЛИС Spartan компании Xilinx;
- цветной VGA-видеодатчик компании Omnivision.

Информация для заказа

Код	Описание
DK-LM3S9B96-FPGA	Плата расширения Stellaris FPGA Expansion Board

И.14. Плата расширения Stellaris LM3S9B96

Плата расширения Stellaris LM3S9B96 EM2 Expansion Board

Плата Stellaris LM3S9B96 EM2 Expansion Board (DK-LM3S9B96-EM2) компании Texas Instruments представляет собой опциональную плату расширения для отладочной платы Stellaris DK-LM3S9B96. Данная плата расширения подключается напрямую к порту интерфейса внешней периферии (EPI) платы Stellaris DK-LM3S9B96. Плата расширения EM2 служит переходником между соединителем интерфейса EPI отладочной платы и соединителем оценочного радиочастотного модуля (РЧ). Данная плата расширения позволяет на основе набора Stellaris DK-LM3S9B96 разрабатывать беспроводные приложения с использованием различных оценочных РЧ модулей.

Отладочная плата DK-LM3S9B96 (поставляется отдельно) предоставляет законченную платформу для ознакомления с возможностями микроконтроллеров Stellaris серии LM3S9000 и обеспечивает богатый набор коммуникационных интерфейсов, таких как 10/100 Ethernet, CAN, Full Speed USB On-The-Go и I^2S.

Особенности платы Stellaris LM3S9B96 EM2 Expansion Board

- Два набора соединителей для подключения до двух оценочных РЧ модулей.
- Микросхема EEPROM с интерфейсом I^2C объёмом 1 Кбит для хранения конфигурационных параметров.
- Разъёмы цифрового и аналогового аудиоинтерфейсов.
- Разъём порта SDIO MOD1.
- Тактовый генератор частотой 32 кГц, подключённый к основному разъему EM2 платы расширения.

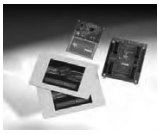
Плата расширения Stellaris LM3S9B96 EM2 Expansion Board

Информация для заказа

Код	Описание
DK-LM3S9B96-EM2	Плата расширения Stellaris LM3S9B96 EM2 Expansion Board

Дополнительные наборы для проектирования беспроводных приложений

Чтобы дать вам возможность разрабатывать и макетировать беспроводные встраиваемые приложения, компания Texas Instruments также создала ряд беспроводных наборов для проектирования. Любой из этих наборов, соединённый с другими компонентами посредством платы расширения Stellaris LM3S9B96 EM2 Expansion Board, позволяет реализовать рабочий прототип беспроводной сети. Эти наборы для проектирования ориентированы на маломощные беспроводные сети и используют единый протокол обмена.



Набор Stellaris® 2.4 GHz SimpliciTI Wireless Kit (содержит оценочный модуль CC2500 и плату расширения DK-LM3S9B96-EM2)

Набор Stellaris® 13.56 MHz RFID Wireless Kit (содержит оценочный модуль TRF7960TB и плату расширения DK-LM3S9B96-EM2)

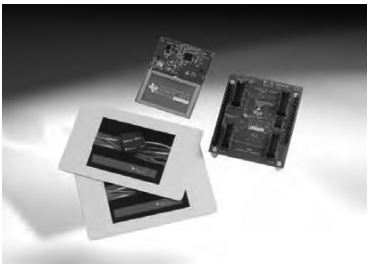
Набор Stellaris® 2.4 GHz ZigBee Wireless Kit (содержит оценочный модуль CC2520 и плату расширения DK-LM3S9B96-EM2)

И.15. Беспроводные наборы Stellaris

Беспроводные наборы Stellaris ускоряют вывод продукции на рынок

Объединяя производительные и простые в применении микроконтроллеры семейства Stellaris на базе ядра ARM Cortex-M3 с ведущими беспроводными решениями, беспроводные наборы Stellaris позволяют разработчикам внедрять в свои устройства поддержку различных беспроводных технологий (RFID, Low-Power RF и ZigBee). Каждый из этих наборов, будучи подключённым к отладочной плате DK-LM3S9B96, содержит все программные и аппаратные средства, необходимые для того, чтобы быстро приступить к разработке, а приложение быстрого старта, входящее в состав каждого набора, позволяет разработчикам менее чем за 10 мин создать работающую беспроводную сеть.

Содержимое набора Stellaris 13.56 MHz RFID Wireless Kit (DK-EM2-7960R)



- Плата расширения Stellaris DK-LM3S9B96-EM2 Expansion Board.
- Модуль RFID-считывателя TRF7960TB.
- Две бесконтактные смарт-карты стандарта ISO/IEC 14443A (MIFARE®-1K).
- Дополнительные RFID-метки, поддерживаемые TRF7960TB и плата расширения Expansion Board.
- Разнообразные примеры ПО, в том числе программы для чтения RFID-карт MIFARE 1K и MIFARE 4K, для одновременной работы с двумя картами, для управления считывателем через последовательный порт с использованием командной строки.
- Скоро появится набор библотек, документация и примеры приложений стандарта ISO/IEC 14443A с поддержкой дополнительных протоколов ISO/IEC.

Информация для заказа

Код	Описание
DK-EM2-7960R	Беспроводной набор Stellaris 13.56 MHz RFID Wireless Kit

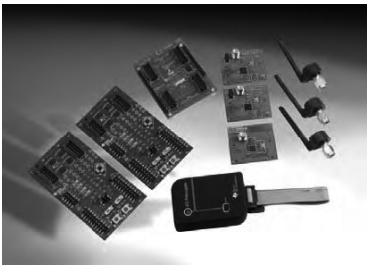
Содержимое набора Stellaris 2.4 GHz SimpliciTI™ Wireless Kit (DK-EM2-2500S)

- Плата расширения Stellaris DK-LM3S9B96-EM2 Expansion Board.
- Оценочный беспроводной модуль CC2500EM.
- Набор eZ430-RF2500.
- Поддержка кросс-платформенного ПО, включая доступ к прошивкам для маломощных РЧ решений, предназначенных для диапазона менее 1 и до 2.4 ГГц.
- Набор библиотек, документация и пример приложения SimpliciTI™, поддерживающего простую сеть с топологией «звезда», а также соединения типа «точка—точка».

Информация для заказа

Код	Описание
DK-EM2-2500S	Беспроводный набор Stellaris 2.4 GHz SimpliciTI™ Wireless Kit

Содержимое набора Stellaris ZigBee Networking Kit (DK-EM2-2520Z)



- Плата расширения Stellaris DK-LM3S9B96-EM2 Expansion Board.
- Оценочный беспроводной модуль CC2520EM.
- Две платы для датчиков с батарейным питанием.
- Два беспроводных оценочных модуля CC2530EM для установки на платы датчиков.
- CC-DEBUGGER для опционального перепрограммирования модулей CC2530EM.
- Законченное решение для построения беспроводной системы стандарта ZigBee с использованием ПО Z-Stack™ 2.4.
- Набор библиотек, документация и пример приложения-координатора, поддерживающего два конечных устройства, объединённых в сеть с топологией «звезда», которые передают информацию о температуре и напряжении.

Информация для заказа

Код	Описание
DK-EM2-2520Z	Беспроводной набор Stellaris ZigBee Networking Kit

И.16. Роботизированные оценочные наборы Stellaris®

Оценочная плата Stellaris Robotic Evaluation Board

Оценочная плата Stellaris Robotic Evaluation Board (EVALBOT) представляет собой роботизированную платформу на базе микроконтроллера Stellaris LM3S9B92. На этой плате также установлены различные аналоговые компоненты Texas Instruments для управления приводом, питанием и осуществления коммуникационных функций. Чтобы начать использовать плату EVALBOAT, достаточно всего нескольких минут, требуемых для сборки набора (все необходимые инструменты входят в состав набора). При автономной работе питание платы осуществляется от трёх элементов AA (входят в набор). Плата автоматически переключается на питание от USB при подключении к ПК в качестве USB-устройства или при отладке. Тестовые точки на плате позволяют наблюдать все основные сигналы. Два 20-контактных соединителя позволяют добавить функции беспроводной связи, используя стандартные оценочные беспроводные модули TI (платы EM). Дополнительные сигналы микроконтроллера выведены на контактные площадки рядом с микроконтроллером. Для отладки и программирования флэш-памяти используется встроенный интерфейс внутрисхемной отладки (IDI), которому для соединения с ПК требуется только USB-кабель (входит в набор).

Особенности платы

- Роботизированная оценочная плата.
- Механические компоненты, устанавливаемые пользователем.
- Микроконтроллер Stellaris LM3S9B92 с 256 Кбайт флэш-памяти, 96 Кбайт ОЗУ, USB OTG, Ethernet MAC+PHY и I^2S.
- Разъём для карты памяти MicroSD.
- Аудиокодек формата I^2S с динамиком.
- Разъёмы USB Host и USB Device.
- Ethernet-разъём RJ45.
- Яркий OLED-дисплей с голубой подсветкой размером 96×16 точек.
- Встроенный интерфейс внутрисхемной отладки (ICDI).
- Питание от батареек (3 элемента AA) или от шины USB.
- Функции для роботизации:
  - два электромотора постоянного тока для передвижения платы;
  - оптические датчики определяют вращение колёс с точностью 45°;
  - датчик обнаружения удара о препятствие.

Оценочная плата Stellaris Robotic Evaluation Board в комплекте с книгой «µC/OS-III: The Real-Time Kernel» (EKB-UCOS3-BNDL)

В этот набор входят роботизированная оценочная плата Stellaris Robotic Evaluation Board и книга «µC/OS-III: The Real-Time Kernel» Ж. Лабросса. В первой части книги рассматриваются принципы работы ядра реального времени на примере ОС µC/OS-III разработки компании Micrium. Во второй части приводятся различные учебные проекты на базе рассмотренной ОС, использующие богатые возможности оценочной платы EM-EVALBOT.

Среди учебных проектов имеются:

- *Вывод информации на экран платы EM-EVALBOT.* Данная программа циклически выводит на OLED-дисплей различные текстовые строки и предназначена для ознакомления с базовой структурой ОС µC/OS-III при её использовании на плате EM-EVALBOT.
- *Использование аудиоподсистемы платы EM-EVALBOT.* Данная программа воспроизводит wav-файлы, записанные во внутренней флэш-памяти микроконтроллера LM3S9B92.
- *Простое управление платой EM-EVALBOT.* Данная программа позволяет независимо управлять обоими двигателями при помощи пользовательских кнопок, останавливать двигатели при срабатывании датчика столкновения и отслеживать посредством µC/Probe состояние робота: состояние и скорость вращения двигателей, состояние датчика столкновения и задействованные ресурсы ЦПУ.

Книга «µC/OS-III: The Real-Time Kernel» (EKB-UCOS3-BOOK)

В данной книге рассматривается работа ядра реального времени на примере ОС µC/OS-III компании Micrium и платы Stellaris EVM-EVALBOT. Помимо ознакомления с основными концепциями и принципами работы ядер реального времени, в книге также содержатся примеры совместного использования µC/OS-III и оценочной платы EVM-EVALBOT. Эта книга предназначена для профессиональных программистов встраиваемых систем, консультантов, радиолюбителей и студентов, которые хотят разобраться, каким образом ядро реального времени используется с современными многофункциональными микроконтроллерами.

Информация для заказа

Код	Описание
EKB-UCOS3-BNDL	Оценочная плата Stellaris® Robotic Evaluation Board в комплекте с книгой «µC/OS-III: The Real-Time Kernel» (автор Ж. Лабросс)
EKB-UCOS3-EVM	Оценочная плата Stellaris® Robotic Evaluation Board для использования с ОС µC/OS-III компании Micrium
EKB-UCOS3-BOOK	Книга «µC/OS-III: The Real-Time Kernel» (автор Ж. Лабросс)

И.17. Набор референс-дизайна интеллектуального дисплейного модуля Stellaris® — одноплатный компьютер

Набор референс-дизайна интеллектуального дисплейного модуля Stellaris — одноплатный компьютер (IDM-SBC) представляет собой законченный пользовательский интерфейс на базе сенсорной панели с разрешением QVGA для систем управления, автоматики, а также контрольно-измерительного оборудования и является первым эталонным проектом с мощным микроконтроллером LM3S9B92. Набор IDM-SBC содержит USB- и Ethernet-модули, SDRAM объёмом 8 Мбайт, последовательную флэш-память объёмом 1 Мбайт, встроенную флэш-память объёмом 256 Кбайт и SRAM объёмом 96 Кбайт. Он позволяет упростить разработку программного обеспечения для набора образцов проектирования с помощью набора библиотек StellarisWare с графической библиотекой и инструментами разработки устройств на базе ARM-процессоров от партнёров ARM. SDRAM объёмом 8 Мбайт подключается к микроконтроллеру LM3S9B92 с помощью шины External Peripheral Interface (EPI).

Модули Stellaris IDM — это первые дисплейные модули, в которых используются высокоэффективные и надёжные микроконтроллеры на базе процессора ARM Cortex-M3. Эти модули применяются в контроллерах доступа, системах безопасности, интеллектуальной бытовой технике, тонких клиентах и в приложениях промышленной автоматики.

Особенности интеллектуального дисплейного модуля Stellaris — одноплатного компьютера

- Яркий ЖК-дисплей QVGA с сенсорным экраном:
  - 262 тыс. цветов, 3.5-дюймовый дисплей QVGA с разрешением 320×240 точек;
  - подсветка на белом светодиоде с резистивной сенсорной панелью.
- Опции последовательных интерфейсов:
  - USB 2.0 Host;
  - 10/100-Мбит/с Ethernet MAC и PHY.
- 1-Мбит/с интерфейс CAN.
- I^2C-интерфейс для внешней периферии и датчики.
- Последовательный порт UART с сигналами TTL-уровня.
- Высокопроизводительный 80-МГц микроконтроллер LM3S9B92:
  - 32-битное ядро ARM Cortex™-M3;
  - 256-Кбайт однотактная флэш-память, 96-Кбайт однотактная SRAM.
- Универсальная память на печатной плате:
  - 8-Мбайт SDRAM, подсоединяемая по шине EPI;
  - 1-Мбайт последовательная флэш-память, подсоединяемая по шине SPI;
  - разъём карты MicroSD;
  - разъём USB Host для подсоединения внешних накопителей.
- Источник питания с широким диапазоном входного напряжения 12…40 В (DC) с вспомогательным 5-В выходом.
- I^2S-монокодек для высококачественного аудио с 0.8-Вт усилителем и внешним 8-Ом динамиком.
- Блок винтовых зажимов для подсоединения питания, а также интерфейсов I^2C и CAN.
- Компактная печатная плата размером 2×3 дюйма.
- Лёгкая адаптация к приложению:
  - включает все исходные коды, примеры приложений и файлы проекта;
  - возможно проектирование на базе инструментов от Keil, IAR, Code Sourcery и Code Red Technologies (с помощью оценочного набора Stellaris или отладчика системы на базе процессоров ARM Cortex-M3);
  - поддержка набора библиотек StellarisWare, включая графическую библиотеку и библиотеку драйверов периферии;
  - поставляется вместе с запрограммированным в условиях производства игровым демонстрационным приложением для быстрого старта;
  - системный Ethernet-загрузчик для обновления набора библиотек.

Содержимое набора референс-дизайна интеллектуального дисплейного модуля Stellaris — одноплатного компьютера

Интеллектуальный дисплейный модуль Stellaris — одноплатный компьютер (IDM-SBC) предлагается как полный набор референс-дизайна (RDK-IDMSBC) и содержит все необходимые компоненты для быстрой оценки IDM-SBC, включая:

- плату Stellaris® IDM-SBC;
- отладочный адаптер MDL-ADA2 из 10-контактного разъёма в 20-контактный разъём;
- USB флэш-накопитель (128 Мбайт);
- 5-В источник питания с международным адаптером;
- Ethernet-кабель;
- 8-Ом динамик;
- компакт-диск с инструментами разработки, документацией и исходным кодом, включая руководство по быстрому старту, руководство пользователя, справочное руководство по программному обеспечению, документацию на плату, спецификацию, электрическую схему и Гербер-файлы.

Интеллектуальный дисплейный модуль Stellaris — одноплатный компьютер

Информация для заказа

Код	Описание
RDK-IDM-SBC	Набор референс-дизайна интеллектуального дисплейного модуля Stellaris — одноплатного компьютера (IDM-SBC)

И.18. Набор референс-дизайна интеллектуального дисплейного модуля Stellaris®

Набор референс-дизайна интеллектуального дисплейного модуля Stellaris (RDK-IDM) представляет собой законченное решение для проектирования пользовательских графических интерфейсов на базе сенсорной панели с подключением по Ethernet. Набор содержит все необходимые аппаратные и программные компоненты для разработки и интеграции модуля интеллектуального дисплея в таких приложениях, как системы промышленного управления и автоматики, а также контрольно-измерительное оборудование. Используя технологию электропитания Power-over-Ethernet (PoE) или источник питания постоянного тока, набор для проектирования на базе модуля интеллектуального дисплея позволяет создать простое решение на базе интеллектуального терминала, который может задействовать один Ethernet-кабель типа CAT5 для питания и обмена данными по сети. Набор также имеет последовательный интерфейс для простой реализации человеко-машинного интерфейса на базе сенсорного дисплея для встраиваемых управляющих устройств. Набор RDK-IDM — первый проект на базе модуля дисплея, в котором применяется высокоэффективный и надёжный микроконтроллер на базе процессора ARM Cortex-M3, что позволяет использовать этот модуль в контроллерах доступа, системах безопасности, интеллектуальной бытовой технике и в приложениях промышленной автоматики.

Особенности набора

Основу набора референс-дизайна интеллектуального дисплейного модуля Stellaris составляет высокоинтегрированный 32-битный микроконтроллер LM3S6918 на базе процессора ARM Cortex-M3 с модулем 10/100-Мбит/с Ethernet MAC и PHY. ARM-архитектура обеспечивает доступ к самой обширной в мире «экосистеме» инструментов разработки, приложений, методов обучения и поддержки, операционных систем и программных стеков протоколов. Разработка специализированного программного обеспечения для RDK-IDM облегчается благодаря обширной графической библиотеке Stellaris Graphics Library и инструментам проектирования ARM от проверенных партнёров ARM по программному обеспечению.

Особенности набора RDK-IDM:

- яркий QVGA ЖК-дисплей с сенсорной поверхностью:
  - 16-бит цвет, 2.8-дюймовый дисплей QVGA с разрешением 240×320 точек;
  - подсветка на белом светодиоде с резистивной сенсорной панелью.
- Ethernet и последовательные интерфейсы:
  - 10/100-Мбит/с Ethernet с автоматическим распознаванием MDI/MDIX и светодиодным индикатором трафик/канал;
  - разъём обеспечивает сигналы TXD и RXD;
  - сигналы уровня RS-232.
- высокопроизводительный 50-МГц микроконтроллер LM3S6918 со встроенной 256-Кбайт флэш-памятью и встроенной 64-Кбайт SRAM;
- гибкие интерфейсы и блок выводов:
  - разъём карты MicroSD;
  - релейный выход;
  - четыре входа для АЦП на блоке выводов;
- гибкие возможности подключения питания:
  - технология Power-over-Ethernet (по IEEE 802.3af);
  - гнездо для питания от источника 24 В (DC), выводы для подсоединения 5-В питания (DC);
- лёгкая адаптация к приложению:
  - включает все исходные коды и файлы проекта;
  - включает полные примеры приложений;
  - проектирование на базе инструментов от Keil, IAR, Code Sourcery и Code Red Technologies;
  - поддержка графической библиотеки Stellaris Graphics Library и библиотеки драйверов периферии Stellaris Peripheral Driver Library.

Содержимое набора

Интеллектуальный дисплейный модуль Stellaris представляет собой набор референс-дизайна и набор для проектирования (RDK-IDM), а также отдельный готовый к работе модуль (модуль MDL-IDM с технологией Power-over-Ethernet или модуль MDL-IDM28 с Ethernet-интерфейсом). Референс-дизайн и набор для проектирования поставляется со всеми средствами, необходимыми для быстрой оценки и простой адаптации модуля интеллектуального дисплея к вашему приложению. Он содержит следующие компоненты:

- интеллектуальный дисплейный модуль Stellaris (модуль MDL-IDM на металлических стойках);
- 80-МГц микроконтроллер LM3S9B92 с 256-Кбайт флэш-памятью, 96-Кбайт SRAM, интегрированным модулем Ethernet MAC+PHY и интерфейсами USB OTG и CAN;
- 24-В источник питания с международным адаптером;
- убираемый Ethernet-кабель;
- адаптер отладки;
- компакт-диск с руководством по быстрому старту, руководством пользователя, справочным руководством по программному обеспечению, исходным кодом, спецификацией, электрической схемой и Гербер-файлами.

Набор референс-дизайна интеллектуального дисплейного модуля Stellaris

Интеллектуальный дисплейный модуль Stellaris

Информация для заказа

Код	Описание
RDK-IDM	Набор референс-дизайна интеллектуального дисплейного модуля Stellaris с интерфейсом Ethernet
MDL-IDM	Интеллектуальный дисплейный модуль Stellaris с технологией Power-over-Ethernet, индивидуальная упаковка
MDL-IDM-B	Интеллектуальный дисплейный модуль Stellaris с технологией Power-over-Ethernet, групповая упаковка
MDL-IDM28	Интеллектуальный дисплейный модуль Stellaris с интерфейсом Ethernet, индивидуальная упаковка
MDL-IDM28-B	Интеллектуальный дисплейный модуль Stellaris с интерфейсом Ethernet, групповая упаковка

И.19. Набор референс-дизайна интеллектуального дисплейного модуля Stellaris® с 3.5-дюймовым экраном

Интеллектуальный дисплейный модуль с 3.5-дюймовым экраном (MDL-IDM-L35) представляет собой законченный графический интерфейс на базе сенсорного QVGA-дисплея для таких приложений, как системы промышленного управления и автоматики, а также контрольно-измерительное оборудование. Модуль MDL-IDM-L35 обеспечивает возможность обмена данными по последовательному, цифровому и аналоговому интерфейсу для простой реализации человеко-машинного интерфейса на базе сенсорной панели во встраиваемых устройствах управления. Разработка программного обеспечения для RDK-IDM-L35 упрощается благодаря обширной графической библиотеке и инструментам разработки ARM-процессоров от партнёров ARM. Модули IDM — это первые дисплейные модули, в которых применяется высокоэффективный и надёжный микроконтроллер на базе процессора ARM Cortex-M3, позволяющий использовать их в контроллерах доступа, системах безопасности, интеллектуальной бытовой технике и в устройствах промышленной автоматики.

Особенности набора

MDL-IDM-L35 представляет собой программируемый модуль со следующими особенностями:

- яркий QVGA ЖК-дисплей с сенсорной поверхностью:
  - 262 тыс. цветов, 3.5-дюймовый дисплей QVGA с разрешением 320×240 точек;
  - подсветка на белом светодиоде с резистивной сенсорной панелью;
- последовательные интерфейсы:
  - последовательный порт RS-232 с сигналами уровня RS-232;
  - последовательный порт UART с TTL-уровнями сигналов;
- высокопроизводительный микроконтроллер LM3S1958 и большой объём памяти:
  - 50-МГц 32-битный процессор ARM Cortex-M3;
  - 256-Кбайт основная флэш-память, 64-Кбайт SRAM;
- разъём карты MicroSD;
- напряжение питания 5 В, DC/DC-преобразователь, который формирует напряжение 3.3 В для питания платы;
- лёгкая адаптация к приложению:
  - включает полный исходный код, примеры приложений и файлы проекта;
  - проектирование с помощью средств разработки IDM-L35 от Keil, IAR, Code Sourcery и Code Red Technologies (с помощью оценочного набора Stellaris или отладчика ARM Cortex-M3);
  - поддержка графической библиотеки Stellaris Graphics Library и библиотеки драйверов периферии Stellaris Peripheral Driver Library.

Интеллектуальный дисплейный модуль с 3.5 дюймовым экраном

Содержимое набора

Интеллектуальный дисплейный модуль Stellaris представляет собой набор референс-дизайна и набор для проектирования (RDK-IDM-L35), а также отдельный готовый к работе модуль (MDL-IDM-L35).

Референс-дизайн и набор для проектирования поставляются со всем необходимым для быстрой оценки и простой адаптации интеллектуального дисплейного модуля к вашему приложению и содержат следующие компоненты:

- интеллектуальный дисплейный модуль Stellaris с 3.5-дюймовой сенсорной панелью QVGA (MDL-IDM-L35) на металлических стойках;
- последовательный кабель USB-TTL для питания платы и соединения с портом UART0 микроконтроллера LM3S1958;
- адаптер JTAG-отладки для соединения 10-контактного разъёма с мелким шагом с 20-контактным стандартным разъёмом;
- 24-В источник питания с международным адаптером;
- компакт-диск с руководством по быстрому старту, руководством пользователя, справочным руководством по программному обеспечению, исходным кодом, спецификацией, электрической схемой и Гербер-файлами.

Интеллектуальный дисплейный модуль с 3.5-дюймовым экраном

Информация для заказа

Код	Описание
RDK-IDM-L35	Набор референс-дизайна интеллектуального дисплейного модуля Stellaris с 3.5-дюймовым экраном
MDL-IDM-L35	Интеллектуальный дисплейный модуль Stellaris с 3.5-дюймовым экраном, индивидуальная упаковка
MDL-IDM-L35-B	Интеллектуальный дисплейный модуль Stellaris с 3.5-дюймовым экраном, групповая упаковка

И.20. Набор референс-дизайна Stellaris® для преобразования последовательных протоколов в Ethernet

Набор референс-дизайна для преобразования последовательных протоколов в Ethernet (RDK-S2E) представляет собой законченное и готовое к реализации решение, которое обеспечивает обмен данными по Ethernet для любых устройств с последовательным интерфейсом. Набор содержит все необходимые аппаратные и программные компоненты для разработки и интеграции вашего проекта по преобразованию последовательных протоколов в Ethernet в промышленных приложениях. Типичным применением RDK-S2E является расширение возможностей действующих систем, которые содержат последовательный порт для конфигурирования или управления. Кроме того, новейшие компьютеры, особенно портативные компьютеры, не всегда имеют последовательные порты, а связь ограничена длиной кабеля (обычно не более 10 м). Реализация решения Stellaris по преобразованию последовательных протоколов в Ethernet в устройствах с последовательными интерфейсами даёт множество преимуществ, включая отсутствие необходимости внесения больших изменений в топологию платы или программное обеспечение; возможность использования сетей, отличных от Ethernet; компактный форм-фактор и отсутствие ограничений на максимальную длину кабеля при последовательном соединении.

Особенности набора

Набор RDK-S2E — первое решение для преобразования последовательных протоколов в Ethernet, в котором используется высокоэффективный и надёжный микроконтроллер на базе процессора ARM Cortex-M3. Основу проекта преобразования последовательных протоколов в Ethernet составляет высокоинтегрированный 50-МГц 32-битный микроконтроллер LM3S6432 с процессорным ядром ARM Cortex-M3 и однотактной встроенной флэш-памятью большой ёмкости и SRAM, которых достаточно для эффективной обработки сетевого трафика. Для максимальной экономии площади микроконтроллер Stellaris предлагается в компактном корпусе BGA и содержит интегрированный модуль 10/100-Мбит/с Ethernet MAC и PHY. ARM-архитектура обеспечивает доступ к самой широкой в мире «экосистеме» инструментов разработки, приложений, методов обучения и поддержки, операционных систем и программных стеков протоколов.

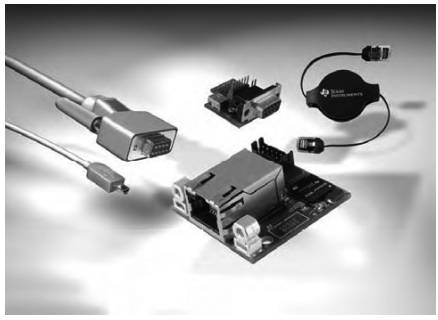
Особенности модуля RDK-S2E:

- микроконтроллер Stellaris LM3S6432 на базе процессора ARM Cortex-M3 в компактном корпусе BGA размером 10×10 мм, что позволяет уменьшить площадь платы;
- 10/100-Мбит/с порт Ethernet:
  - автоматическая коррекция MDI/MDIX;
  - светодиодный индикатор трафик/канал;
- 2 порта UART с контролем потока RTS/CTS:
  - UART0 имеет уровни RS-232, приёмопередатчик работает на скорости до 250 Кбит/с;
  - UART1 имеет уровни CMOS/TTL, способен работать со скоростью 1.5 Мбит/с;
- программное обеспечение:
  - IP-конфигурирование со статическим IP-адресом или DHCP;
  - Telnet-сервер для доступа к последовательному порту;
  - веб-сервер для конфигурирования модуля;
  - ответчик на UDP-запросы для обнаружения устройства;
  - Telnet-клиент для расширителя последовательного порта на базе Ethernet;
  - SSH-сервер для безопасного соединения;
- модуль поддерживает напряжение питания 5 и 3.3 В;
- множество вариантов монтажа, включая поставляемый по запросу монтажный кронштейн;
- порт JTAG для программирования в условиях производства.

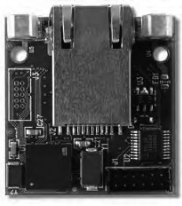
Содержимое набора

Модуль Stellaris для преобразования последовательных протоколов в Ethernet представляет собой набор референс-дизайна (RDK-S2E) плюс отдельный готовый к работе модуль (MDL-S2E). Набор референс-дизайна поставляется со всем необходимым для быстрой оценки и простой адаптации модуля MDL-S2E к вашему приложению и содержит следующие компоненты:

- модуль Stellaris для преобразования последовательных протоколов в Ethernet (MDL-S2E);
- плату адаптера RS-232;
- убираемый Ethernet-кабель;
- последовательный кабель DB9;
- USB-кабель;
- компакт-диск с руководством по быстрому старту, руководством пользователя, справочным руководством по программному обеспечению, исходным кодом, спецификацией, электрической схемой и Гербер-файлами.



Набор референс-дизайна Stellaris для преобразования последовательных протоколов в Ethernet



Модуль Stellaris для преобразования последовательных протоколов в Ethernet

Информация для заказа

Код	Описание
RDK-S2E	Набор референс-дизайна Stellaris для преобразования последовательных протоколов в Ethernet (RDK)
MDL-S2E	Модуль Stellaris для преобразования последовательных протоколов в Ethernet, индивидуальная упаковка
MDL-S2E-B	Модуль Stellaris для преобразования последовательных протоколов в Ethernet, групповая упаковка

И.21. Набор референс-дизайна Stellaris® для управления шаговым двигателем

Набор референс-дизайна Stellaris для управления шаговым двигателем (RDK-Stepper) содержит все необходимые аппаратные и программные компоненты для проектирования, модернизации и интеграции современных устройств с шаговым двигателем. Набор RDK-Stepper, благодаря вычислительной мощности и гибкости микроконтроллеров Stellaris и использованию MOSFET и драйверов MOSFET компании Fairchild Semiconductor, позволяет создавать усовершенствованные системы управления шаговым двигателем, оптимизированные по характеристикам, стоимости и гибкости. Шаговые двигатели особенно подходят для применения в 2- и 3-осевых ЧПУ типа CNC, сортировочном и калибровочном оборудовании, специализированных принтерах и сканерах, а также в устройствах промышленной автоматики.

Архитектура программного обеспечения набора может быть использована в широком спектре приложений от микромощных устройств до сильноточного оборудования. Такая мощная и гибкая архитектура программного обеспечения, построенная на базе подпрограмм обработки прерываний с приоритетом, работает эффективно в фоновом режиме, оставляя значительный резерв ресурсов для системного приложения или сетевых заданий.

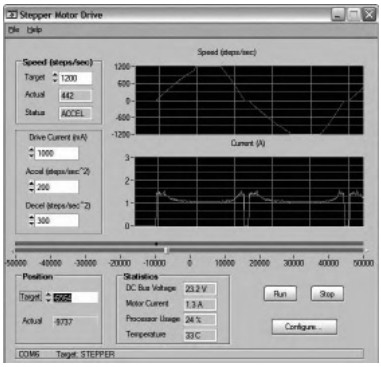
Особенности набора

Набор RDK-Stepper содержит полнофункциональный микроконтроллер LM3S617, предназначенный для устройств управления приводом, силовой каскад, выполненный на основе микросхемы драйвера FAN73832 и MOSFET FDMS3672 компании Fairchild Semiconductor, шаговый двигатель NEMA23, управляющую программу с графическим интерфейсом для Windows, а также сопутствующие кабели, исходный код и документацию. Набор референс-дизайна для управления шаговым двигателем позволяет использовать встроенные функции микроконтроллера Stellaris и вычислительную мощность процессорного ядра ARM Cortex-M3 для реализации импульсного (чопперного) управления без необходимости применять внешний контроллер шагового двигателя или компаратор. Графическая управляющая программа позволяет экспериментировать с различными параметрами привода и фиксировать характеристики двигателя.

Особенности набора референс-дизайна RDK-Stepper:

- усовершенствованное импульсное (чопперное) управление биполярными шаговыми двигателями;
- программное импульсное (чопперное) управление шаговыми двигателями с высоким крутящим моментом при высокой скорости шага;
- режимы быстрого и медленного спада;
- режимы полного шага, полушага, микрошага и волны;
- высокая скорость шага (до 10 000 шагов/с);
- программируемый ток удержания;
- виртуальный COM-порт, интегрированный в USB;
- поддержка внешнего отладчика через стандартный 20-контактный разъём от ARM;
- простое подсоединение питания и двигателя с помощью подключаемой контактной колодки;
- системный загрузчик для обновления микропрограммного обеспечения через последовательный порт.

Набор референс-дизайна Stellaris для управления шаговым двигателем



Скриншот окна программы управления из набора референс-дизайна RDK-Stepper для Windows

Содержимое набора

Набор референс-дизайна RDK-Stepper поставляется со всеми необходимыми компонентами для оценки системы управления биполярным шаговым двигателем, включая:

- главную плату управления;
- шаговый двигатель NEMA23;
- 24-В источник питания с международным адаптером;
- USB-кабель;
- программу управления с графическим интерфейсом для Windows на компакт-диске;
- компакт-диск с руководством по быстрому старту, руководством пользователя, справочным руководством по программному обеспечению, исходным кодом, спецификацией, электрической схемой и Гербер-файлами.

Информация для заказа

Код	Описание
RDK-Stepper	Набор референс-дизайна Stellaris для управления шаговым двигателем
MDL-Stepper	Плата управления шаговым двигателем, индивидуальная упаковка
MDL-Stepper-B	Плата управления шаговым двигателем, групповая упаковка

И.22. Набор референс-дизайна Stellaris® для управления бесщёточным двигателем постоянного тока

Набор референс-дизайна Stellaris для управления бесщёточным двигателем постоянного тока (RDK-BLDC) с интерфейсами Ethernet и CAN содержит все необходимые аппаратные и программные компоненты для проектирования, модернизации и интеграции приложения на базе бесщёточного двигателя постоянного тока в промышленных сетях. Набор RDK-BLDC позволяет использовать вычислительную мощность и гибкость микроконтроллеров Stellaris и силовые модули компании Fairchild Semiconductor для реализации сложных четырёхквадрантных алгоритмов управления 3-фазными бесщёточными двигателями постоянного тока, рассчитанными на напряжение до 36 В. Бесщёточные двигатели постоянного тока особенно хорошо подходят для применения в промышленной автоматике, робототехнике, в электрических инвалидных колясках и движущихся механизмах, насосах и вентиляционных системах, а также в компактных бытовых приборах.

Особенности набора

Набор RDK-BLDC содержит полнофункциональный микроконтроллер Stellaris LM3S8971 с интерфейсами Ethernet и CAN, 3-фазный бесщёточный двигатель постоянного тока, программу управления с графическим интерфейсом для Windows, а также сопутствующие кабели, исходный код и документацию. Набор RDK-BLDC использует встроенные функции управления приводом и коммуникационные возможности микроконтроллера Stellaris LM3S8971, а также вычислительную мощность процессорного ядра ARM Cortex-M3 для того, чтобы обеспечить оптимальные алгоритмы управления широким спектром бесщёточных двигателей постоянного тока в разнообразных приложениях. Программа управления с графическим интерфейсом позволяет подбирать различные параметры управления и изучать их влияние на характеристики двигателя.

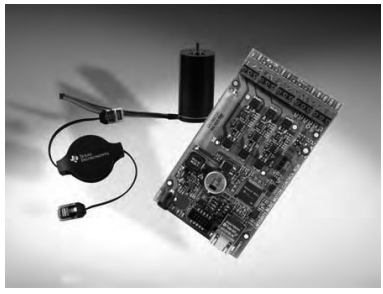
Особенности набора RDK-BLDC:

- коммуникационные интерфейсы 10/100-Мбит/с Ethernet и CAN;
- усовершенствованные алгоритмы управления 3-фазными бесщёточными двигателями постоянного тока;
- четырёхквадрантные алгоритмы работы для точного управления двигателем;
- режимы работы: с датчиком Холла, квадратурный и без датчика;
- управление 3-фазными бесщёточными двигателями постоянного тока, рассчитанными на напряжение до 36 В и мощность до 500 Вт;
- набор легко адаптируется к приложению за счёт исходного кода и файлов проекта;
- легко расширяемое программное обеспечение по управлению приводом с контролем прерываний;
- резерв ресурсов в 30 MIPS для системного программного обеспечения;
- собранная на плате схема торможения;
- вход инкрементного импульсного датчика положения;
- кнопочный переключатель тестовых режимов;
- светодиодные индикаторы для отображения режимов питания, функционирования и сбоя;
- поставляемый на заказ управляемый вентилятор для принудительного воздушного охлаждения;
- винтовые зажимы для всех кабелей питания и сигналов;
- порт JTAG/SWD для отладки программ.

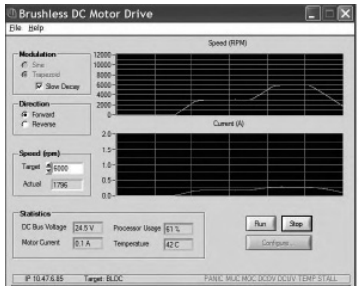
Содержимое набора

Набор RDK-BLDC поставляется со всеми необходимыми средствами для оценки систем управления бесщёточными двигателями постоянного тока, включая:

- главную плату управления;
- 3-фазный бесщёточный двигатель постоянного тока;
- 24-В источник питания;
- убираемый Ethernet-кабель;
- адаптер отладки;
- программу управления с графическим интерфейсом для Windows на компакт-диске;
- компакт-диск с руководством по быстрому старту, руководством пользователя, справочным руководством по программному обеспечению, исходным кодом, спецификацией, электрической схемой и Гербер-файлами.



Набор референс-дизайна Stellaris для управления бесщёточным двигателем постоянного тока



Вид программы управления с графическим интерфейсом для Windows из набора эталонного проекта RDK-BLDC

Информация для заказа

Код	Описание
RDK-BLDC	Набор референс-дизайна Stellaris для управления бесщёточным двигателем постоянного тока
MDL-BLDC	Плата управления бесщёточным двигателем постоянного тока, индивидуальная упаковка
MDL-BLDC -B	Плата управления бесщёточным двигателем постоянного тока, групповая упаковка

И.23. Набор референс-дизайна Stellaris® для управления щёточным двигателем постоянного тока с интерфейсом CAN (RDK-BDC24)

Набор референс-дизайна Stellaris для управления щёточным двигателем постоянного тока (RDK-BDC24) с интерфейсом CAN позволяет управлять скоростью вращения 12-В и 24-В щёточных двигателей постоянного тока с потреблением до 40 А и содержит новый управляющий вход, основанный на интерфейсе RS-232, который может использоваться в качестве преобразователя последовательного интерфейса в интерфейс CAN. Также модуль RDK-BDC24 обеспечивает богатый набор интерфейсов датчиков, имеет широкие коммуникационные и управляющие возможности. В частности, в модуле предусмотрены входы аналогового и импульсного датчика положения, реализованы высокопроизводительные интерфейсы CAN и порт RS-232. Модуль использует высокооптимизированное программное обеспечение и содержит мощный 32-битный микроконтроллер Stellaris LM3S2616, который обеспечивает плавную и бесшумную работу двигателя постоянного тока в широком диапазоне скоростей.

На плате модуля MDL-BDC24 присутствуют такие высококачественные аналоговые компоненты от Texas Instruments, как приёмопередатчик шины CAN SN65HVD1050, приёмопередатчик интерфейса RS-232 MAX3221, DC/DC-преобразователь TPS54040, стабилизатор напряжения TPS73633 и усилитель для резистивного датчика тока INA193. Этот модуль рассчитан на использование в самых разных потребительских и промышленных устройствах, включая системы промышленной автоматики, мобильные роботы, бытовую технику, насосные и вентиляционные системы, а также электрические инвалидные коляски и движущиеся механизмы.

Особенности набора

Набор MDL-BDC поставляется как готовый к работе программно адаптируемый модуль, обладающий следующими особенностями:

- бесшумное управление щёточными двигателями постоянного тока (15 кГц ШИМ);
- три опции регулирования скорости вращения:
  - стандартный в отрасли интерфейс R-C сервопривода (ШИМ);
  - интерфейс CAN;
  - последовательный интерфейс RS-232;
- интерфейс CAN или RS-232 для управления с обратной связью, управления скоростью, положением или токового управления;
- обмен данными через интерфейс CAN:
  - возможность полного конфигурирования опций модуля;
  - мониторинг тока, напряжения, скорости и других параметров в режиме реального времени;
  - загрузка микропрограммного обеспечения через интерфейсы RS-232/CAN;
- последовательный обмен данными через интерфейс RS-232:
  - подключение устройств с интерфейсом RS-232 к шине CAN;
  - прямое подключение к последовательному порту ПК или порту cRIO от National Instruments;
- входы для концевых выключателей прямого/обратного хода;
- возможности микропрограммного обеспечения:
  - полностью конфигурируемые параметры обратной связи;
  - мониторинг сигналов от всех датчиков в режиме реального времени, включая потребляемый двигателем ток, положение ротора, скорость и состояние концевых выключателей;
- светодиодный индикатор для отображения режимов работы, направления вращения и сбоев;
- переключатель режимов работы двигателя — торможение/движение по инерции;
- вход импульсного датчика положения (QEI) и аналоговый вход;
- цветные винтовые зажимы для подключения питания;
- лёгкая адаптация к приложению с помощью инструментов Keil, IAR, Code Sourcery, Code Red Technologies или Texas Instruments (используя оценочный набор Stellaris или отладчик ARM Cortex-M3).

Содержимое набора

Набор Stellaris MDL-BDC24 поставляется не только в виде отдельного готового для производства модуля, но и как законченный набор референс-дизайна (RDK-BDC24), созданный с помощью открытых инструментов. Этот набор имеет все необходимые компоненты для быстрой оценки и простой адаптации модуля MDL-BDC24 для вашего приложения и включает в себя следующие элементы:

- модуль управления двигателем MDL-BDC24;
- щёточный двигатель постоянного тока Mabuchi RS-555PH-3255 (5000 об/мин, 12 В, 3 А);
- источник питания с универсальным входом;
- преобразователь последовательного интерфейса в интерфейс CAN (DB9 — RJ12);
- два модульных кабеля 6P-6C (длиной 1 и 7 футов);
- терминатор шины CAN (120 Ом);
- адаптер JTAG-отладки для соединения 10-контактного разъёма с мелким шагом с 20-контактным стандартным разъёмом;
- компакт-диск с документацией, утилитой обновления микропрограммного обеспечения LM Flash, графической средой BDC-COMM для управления и отслеживания состояния двигателя, исходным кодом, спецификацией, электрической схемой и Гербер-файлами.

Набор референс-дизайна Stellaris для управления щёточным двигателем постоянного тока с интерфейсом CAN

Информация для заказа

Код	Описание
MDL-BDC24	Модуль Stellaris для управления щёточным двигателем постоянного тока с интерфейсом CAN (RDK-BDC24), индивидуальная упаковка
MDL-BDC24-B	Модуль Stellaris для управления щёточным двигателем постоянного тока с интерфейсом CAN (RDK-BDC24), групповая упаковка
RDK-BDC24	Набор референс-дизайна Stellaris для управления щёточным двигателем постоянного тока с интерфейсом CAN (включает модуль MDL-BDC24)

И.24. Набор референс-дизайна Stellaris® для управления асинхронным электродвигателем переменного тока

Набор референс-дизайна Stellaris для управления асинхронным электродвигателем переменного тока (RDK-ACIM) содержит все необходимые аппаратные и программные компоненты для разработки, модернизации и интеграции современных приложений на базе асинхронных двигателей переменного тока. Набор RDK-ACIM позволяет использовать вычислительную мощность и гибкость микроконтроллеров Stellaris и силовые модули компании Fairchild Semiconductor для создания усовершенствованных систем на базе электродвигателя переменного тока с регулируемой скоростью вращения, которые обеспечивают высокую производительность, низкую стоимость и гибкость. Асинхронные двигатели переменного тока особенно хорошо подходят для применения в бытовой технике (холодильниках, посудомоечных машинах, стиральных машинах и сушильных аппаратах), системах управления отоплением, вентиляцией и кондиционированием воздуха для жилых помещений и небольших предприятий, а также в приводах промышленных 3-фазных электродвигателей.

Архитектура программного обеспечения набора непосредственно масштабируема для применения в приложениях мощностью менее 1 л.с. до сотен кВт. Такая мощная и гибкая архитектура программного обеспечения, построенная на базе подпрограмм обработки прерываний с приоритетом, работает эффективно в фоновом режиме, оставляя существенный запас ресурсов для системного приложения или сетевых заданий.

Особенности набора

Набор RDK-ACIM содержит полнофункциональный микроконтроллер Stellaris LM3S818 для устройств управления приводом, силовой модуль FSBS10CH60 от Fairchild Semiconductor, электродвигатель переменного тока Selni для 3-фазного оборудования, управляющую программу с графическим интерфейсом для Windows, а также сопутствующие кабели, исходный код программ, электрическую схему, спецификацию и документацию. Набор RDK-ACIM использует встроенные функции и вычислительную мощность микроконтроллера Stellaris для реализации современных энергоэффективных алгоритмов управления, включая пространственно-векторную модуляцию (SVM). Управляющая программа с графическим интерфейсом позволяет экспериментировать с различными параметрами привода и фиксировать характеристики двигателя.

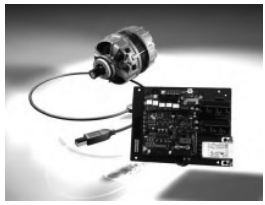
Особенности набора RDK-ACIM:

- усовершенствованные алгоритмы управления 3-фазными и однофазными асинхронными двигателями переменного тока;
- активная схема торможения;
- активная схема управления бросками тока;
- возможность управления внешним каскадом коррекции коэффициента мощности;
- возможность изменений линейного фильтра, конденсаторов шины питания и JTAG-интерфейса;
- содержит код для основных алгоритмов управления, в том числе для пространственно-векторной модуляции и контроля синусоиды;
- точное измерение тока;
- несколько изолированных входов управления, в том числе:
  - виртуальный COM-порт на базе интегрированного порта USB;
  - программа под Windows для конфигурирования, управления и мониторинга;
  - последовательный порт с логическими уровнями сигналов;
  - потенциометр регулировки скорости и переключатель режимов;
  - мониторинг скорости вращения и положения через вход импульсного датчика положения/тахометра;
  - электрически изолированный JTAG-порт для отладки программного обеспечения;
  - системный загрузчик для обновления микропрограммного обеспечения через последовательный порт.

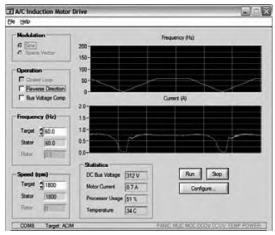
Содержимое набора

Набор RDK-ACIM поставляется со всеми необходимыми компонентами для оценки системы управления на базе асинхронного электродвигателя переменного тока, включая:

- главную управляющую плату с установленным радиатором;
- электродвигатель переменного тока для 3-фазного домашнего оборудования (0…20 000 об/мин);
- кабели питания;
- USB-кабель;
- программу управления с графическим интерфейсом для Windows на компакт-диске;
- компакт-диск с руководством по быстрому старту, руководством пользователя, справочным руководством по программному обеспечению, исходным кодом, спецификацией, электрической схемой и Гербер-файлами;
- системный загрузчик для обновления микропрограммного обеспечения через последовательный порт.



Набор референс-дизайна Stellaris для управления асинхронным электродвигателем переменного тока



Вид программы управления с графическим интерфейсом для Windows из набора референс-дизайна RDK-ACIM

Информация для заказа

Код	Описание
RDK-ACIM	Набор референс-дизайна Stellaris для управления асинхронным электродвигателем переменного тока
MDL-ACIM	Плата управления асинхронным электродвигателем переменного тока, индивидуальная упаковка
MDL-ACIM-B	Плата управления асинхронным электродвигателем переменного тока, групповая упаковка

И.25. Общие сведения о встраиваемых процессорах TI на базе технологии ARM

Компания TI предлагает широкий спектр продукции на базе технологии ARM, которая предназначена для разнообразных приложений и обеспечивает оптимальную производительность, энергопотребление и стоимость систем. Эти процессоры на базе технологии ARM представлены различными линейками продукции компании TI (см. **Табл. И.9**). Более подробная информация приведена на сайте **www.ti.com/arm**.

Применение устройств компании TI на базе технологии ARM

ARM-процессоры предлагают широкий выбор функций и производительности, что позволяет создавать решения, которые наиболее точно соответствуют предъявляемым к ним требованиям.

Области применения для этих устройств:

- системы обработки данных:
  - точки продаж предприятий розничной торговли (POS-терминалы);
  - портативные компьютеры;
- проводная связь:
  - сетевые системы;
  - широковещательное оборудование;
- потребительская электроника:
  - портативные аудио/видеоплееры;
  - цифровые абонентские ТВ приставки;
  - цифровые фотоаппараты;
  - сетевые устройства;
  - системы управления отоплением, вентиляцией и кондиционированием воздуха;
  - игровые устройства;
  - планшетные компьютеры;
- автомобильная электроника:
  - информационно-развлекательные устройства;
  - системы безопасности и управления;
  - тормозная система и рулевое управление;
  - гибридные автомобили и электромобили;
- промышленные системы:
  - медицинское оборудование;
  - автоматика и приводы;
  - измерительные системы;
  - источники питания;
  - дистанционный контроль;
  - управление системами зданий;
  - системы промышленной автоматики;
  - испытательное и измерительное оборудование;
  - человеко-машинный интерфейс (HMI).

Таблица И.9. **Продукция компании TI на базе технологии ARM***

Процессор TI	ЦПУ	МГц	Операционная система	Основная периферия
Sitara™ AM1705	ARM9™	450	Linux, Windows CE, RTOS	I²C, SPI, UART, USB, MMC/SDIO, EMAC
Sitara AM1707	ARM9	450	Linux, Windows CE, RTOS	I²C, SPI, UART, USB, MMC/SDIO, EMAC
Sitara AM1802	ARM9	300	Linux, Windows CE, RTOS	I²C, SPI, UART, USB, MMC/SDIO, EMAC
Sitara AM1806	ARM9	450	Linux, Windows CE, RTOS	I²C, SPI, UART, USB, MMC/SDIO
Sitara AM1808	ARM9	450	Linux, Windows CE, RTOS	I²C, SPI, UART, USB, MMC/SDIO, EMAC, SATA
Sitara AM1810	ARM	375	Linux, Windows CE, RTOS	I²C, SPI, UART, USB, MMC/SDIO, EMAC, SATA, PROFIBUS
Sitara AM3505	ARM Cortex™-A8	600	Linux, Windows CE, RTOS, Android	I²C, SPI, UART, USB, MMC/SDIO, EMAC, CAN
Sitara AM3517	ARM Cortex-A8	600	Linux, Windows CE, RTOS, Android	I²C, SPI, UART, USB, MMC/SDIO, EMAC, CAN
Sitara AM3703	ARM Cortex-A8	1000	Linux, Windows CE, RTOS, Android	I²C, SPI, UART, USB, MMC/SDIO
Sitara AM3715	ARM Cortex-A8	1000	Linux, Windows CE, RTOS, Android	I²C, SPI, UART, USB, MMC/SDIO
Sitara AM3892	ARM Cortex-A8	1500	Linux, Windows CE, RTOS	PCIe, SATA, EMAC, UART, USB
Sitara AM3894	ARM Cortex-A8	1500	Linux, Windows CE, RTOS	PCIe, SATA, Gigabit EMAC, UART, USB
OMAP-L137	ARM926 + C674x DSP	300	Linux, Windows CE, VxWorks	MMC/SD, SDRAM/NAND, EMAC, UART, USB 2.0 HS OTG, USB 1.1
OMAP-L138	ARM926 + C674x DSP	300	Linux, Windows CE, VxWorks	mDDR/DDR2, SDRAM/NAND, SATA, uPP, EMAC, USB 2.0 HS OTG, USB 1.1
TMS320C6A8168	C674x+ ARM Cortex-A8	1500	Linux, Windows CE, RTOS	DDR2/DDR3, SRAM/Pseudo SRAM/NAND, NOR Flash, SD, SATA, uPP, EMAC, USB 2.0 HS
TMS320C6A8167	C674x+ ARM Cortex-A8	1500	Linux, Windows CE, RTOS	DDR2/DDR3, SRAM/Pseudo SRAM/NAND, NOR Flash, SD, SATA, uPP, EMAC, USB 2.0 HS
TMS320DM355	ARM926	135, 216, 270	Linux	mDDR/DDR2, USB 2.0 H/OTG
TMS320DM335	ARM926	135, 216	Linux	mDDR/DDR2, USB 2.0 H/OTG
TMS320DM357	ARM926	270	Linux	EMAC, DDR2, JTAG, USB 2.0 OTG
TMS320DM365	ARM926	216, 270, 300	Linux	EMAC, mDDR/DDR2, HPI, голосовой кодек, USB 2.0 H/OTG
TMS320DM6467	ARM926 + C64x DSP	594/729, 297/365	Linux, Windows CE	EMAC, DDR2, USB 2.0, HPI, PCI, ATA
TMS320DM6446	ARM926 + C64x DSP	300/600	Linux, Windows CE	EMAC, DDR2, USB 2.0, HPI, ATA, интерфейс карт памяти
TMS320DM6443	ARM926 + C64x DSP	300/600	Linux, Windows CE	EMAC, DDR2, USB 2.0, HPI, ATA, интерфейс карт памяти
TMS320DM6441	ARM926 + C64x DSP	256/512	Linux, Windows CE	EMAC, DDR2, USB 2.0, HPI, ATA, интерфейс карт памяти
TMS570LS2x	ARM Cortex-R4F	160	AUTOSAR, различные встраиваемые RTOS	FlexRay, CAN, буферированный АЦП, буферированный SPI, SCI/ LIN, Timer Coprocessor, функции самотестирования и контроля ECC для критичных к безопасности приложений
Stellaris® LM3Sx00s	ARM Cortex-M3	20…50	Различные встраиваемые RTOS	(МК) АЦП, SSI/SPI, UART, I²C, модуль управления двигателями
Stellaris LM3S1000s	ARM Cortex-M3	25…80	Различные встраиваемые RTOS	(МК) АЦП, SSI/SPI, UART, I²C, модуль управления двигателями, режим Hibernate
Stellaris LM3S2000s	ARM Cortex-M3	25…80	Различные встраиваемые RTOS	(МК) CAN, АЦП, SSI/SPI, UART, I²C, модуль управления двигателями, режим Hibernate
Stellaris LM3S3000s	ARM Cortex-M3	50	Различные встраиваемые RTOS	(МК) USB 2.0 FS D/H/OTG, АЦП, SSI/SPI, UART, I²C, модуль управления двигателями, режим Hibernate, StellarisWare® в ПЗУ
Stellaris LM3S5000s	ARM Cortex-M3	50…80	Различные встраиваемые RTOS	(МК) USB 2.0 FS D/H/OTG, CAN, АЦП, SSI/SPI, UART, I²C, модуль управления двигателями, режим Hibernate, StellarisWare® в ПЗУ
Stellaris LM3S6000s	ARM Cortex-M3	25…50	Различные встраиваемые RTOS	(МК) 10/100 Ethernet MAC+PHY, АЦП, SSI/SPI, UART, I²C, модуль управления двигателями, режим Hibernate
Stellaris LM3S8000s	ARM Cortex-M3	50	Различные встраиваемые RTOS	(МК) 10/100 Ethernet MAC+PHY, CAN, АЦП, SSI/SPI, UART, I²C, модуль управления двигателями, режим Hibernate
Stellaris LM3S9000s	ARM Cortex-M3	80…100	Различные встраиваемые RTOS	(МК) 10/100 Ethernet MAC+PHY, USB 2.0 FS D/H/OTG, CAN, АЦП, SSI/SPI, UART, I²C, модуль управления двигателями, режим Hibernate, StellarisWare® в ПЗУ

И.26. Высокопроизводительные контроллеры Stellaris ARM Cortex-M4F от Texas Instruments

Компания Texas Instruments представила новое поколение микроконтроллеров Stellaris на базе ARM Cortex-M4 — LM4Fx. Микроконтроллеры выполнены по 65-нм технологии, что позволяет достичь хорошего соотношения между такими параметрами, как высокая производительность, малое энергопотребление и цена.

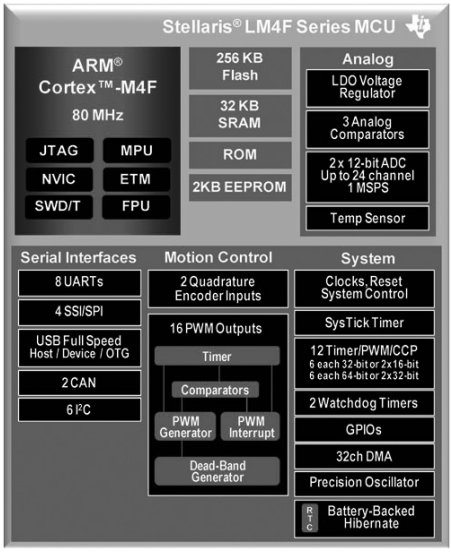
Семейство Stellaris ARM Cortex-M4, получившее название Stellaris LM4F, по сути является расширением популярного семейства ARM Cortex-M3, включающее дополнительные DSP-подобные инструкции, а также модуль операций с плавающей точкой: процессор ARM Cortex-M4 поддерживает широкий набор одноцикловых команд умножения с накоплением (MAC), команды централизованного управления потоком данных (SIMD) и арифметические команды «с насыщением», а также имеет модуль обработки операций с плавающей точкой (FPU) с одинарной точностью.

Основными областями применения семейства LM4F являются микроконтроллеры общего назначения, USB-контроллеры, контроллеры для промышленной автоматики и управления двигателями.

Для ускорения процесса разработки программного обеспечения Texas Instruments предоставляет бесплатное программное обеспечение StellarisWare®, поддерживающее все контроллеры семейства LM4F. StellarisWare представляет собой комплект API, который был специально разработан для контроллеров Stellaris, чтобы минимизировать стоимость разработки программного обеспечения и сократить время выхода на рынок. Большинство из API-интерфейсов встроено в ROM. Также стоит отметить, что всё программное обеспечение написано на языке C, что облегчает процесс корректировки и написания собственного программного кода.

Основные характеристики первого поколения LM4F

- Ядро ARM Cortex-M4F с модулем обработки операций с плавающей точкой (FPU) с одинарной точностью.
- До 256 Кбайт встроенной флэш-памяти и 32 Кбайт ОЗУ.
- 2 Кбайт встроенного EEPROM.
- До двух 12-бит АЦП с 24 входными каналами.
- До двух CAN-контроллеров.
- Три аналоговых компаратора.
- Опциональный интерфейс USB 2.0 с поддержкой режимов Device/Host/OTG.
- Расширенные функции управления двигателями при помощи использования до 16 ШИМ-контроллеров и двух интерфейсов квадратурного энкодера.
- Большой выбор последовательных интерфейсов:
  - до 8 модулей UART;
  - до 6 каналов I2C;
  - до 4 каналов SPI/SSI.
- Режимы пониженного энергопотребления, в том числе «спящий режим».
- Корпуса 64-LQFP, 100-LQFP и 144-LQFP.



Основные преимущества

- Новый 12-бит АЦП с частотой выборки 1 Мвыб./с позволяет достичь точности ±1 бит во всём диапазоне температур.
- Ядро ARM Cortex-M4F со встроенным FPU ускоряет выполнение математических операций и упрощает процесс обработки цифровых сигналов.
- Первый микроконтроллер ARM Cortex-M, выполненный в 65-нм технологическом процессе, обеспечивает хорошее соотношение между высокой производительностью и малым энергопотреблением:
  - время пробуждения менее 500 мкс;
  - ток в рабочем режиме менее 370 мкА/МГц;
  - ток энергопотребления в ждущем режиме — 1.6 мкА.
- Минимальное количество циклов перезаписи флэш-памяти — 100 000 циклов.
- Широкий диапазон совместимой памяти и вариантов корпусов позволяют сделать оптимальный выбор устройства.
- Обширный выбор периферии на кристалле открывает возможности применения в различных приложениях, включая 1D-сканеры, микропринтеры, цифровое питание, управление двигателями, управление светодиодным табло, устройства промышленной автоматики.

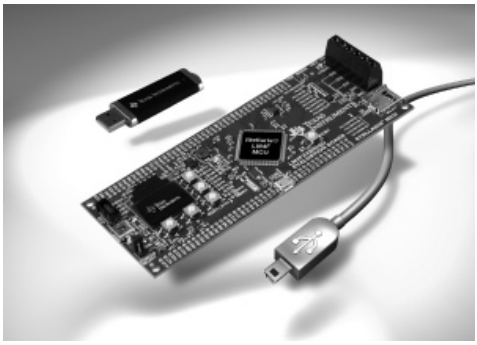
Быстрый старт с микроконтроллером Stellaris ARM Cortex-M4F

Texas Instruments представляет линейку новых оценочных наборов для микроконтроллеров серии LM4F.

Оценочный набор Stellaris® LM4F232 USB+CAN — это компактный и универсальный набор для демонстрации возможностей микроконтроллера Stellaris LM4F232, основанного на ядре ARM® Cortex™-M4F.

Оценочный набор позволяет показать доступные в микроконтроллере LM4F232 функции, а именно: контроллер USB 2.0 в режимах OTG/Host/Device, CAN-контроллер, аналоговые функции и возможности низкого энергопотребления.

В комплект также входит множество примеров в исходном коде для быстрого начала программирования в Си-коде.



Оценочный набор включает в себя следующие компоненты:

- микроконтроллер Stellaris LM4F232H5QD с 256-Кбайт внутренней флэш-памятью в корпусе 144-LQFP;
- 99 × 64 цветной OLED-дисплей, обеспечивающий функции интерфейса;
- разъём USB Micro-AB для макетирования USB-применений;
- слот для карт памяти microSD;
- 5-мм винтовые клеммы для подключения внешних датчиков и других аналоговых входов;
- прецизионное опорное напряжение для точного аналого-цифрового преобразования;
- датчик температуры для измерения температуры;
- 3-осевой акселерометр для детектирования положения;
- контрольные точки для всех линий ввода/вывода, что позволяет легко анализировать сигналы при отладке;
- пять пользовательских/навигационных кнопок (в том числе выбор/пробуждение) для ввода данных пользователем;
- один пользовательский светодиод;
- 10-контактный JTAG, обеспечивающий стандартный интерфейс для отладки.

Состав набора

Отладочный набор включает всё необходимое для разработки и выполнения приложений на микроконтроллере Stellaris:

- оценочную плату Stellaris EK-LM4F232;
- внутрисхемный отладочный интерфейс, реализованный на плате;
- набор кабелей:
  - кабель USB Mini-B для отладки;
  - USB переходной кабель Micro-A-в-Std-A;
  - кабель USB Micro-B-USB-A;
  - USB флэш-накопитель;
- литиевую батарею 3-V CR2032;
- CD-диск, содержащий:
  - полную документацию;
  - StellarisWare®-библиотеку периферийных драйверов и примеры исходного кода;
- пакет разработчика Stellaris Firmware с примерами исходных кодов;
- приложения для быстрого старта с исходными кодами;
- Windows-приложения для быстрого старта;
- ознакомительную версию среды разработки — одну из следующего списка:
  - Keil™ RealView® Microcontroller Development Kit (MDK-ARM);
  - IAR Embedded Workbench® development tools;
  - CodeSourcery tools development tools;
  - Code Red Technologies Tools;
  - Texas Instruments' Code Composer Studio™ IDE.

Информация для заказа

Код	Описание
EKK-LM4F232	Оценочный набор Stellaris LM4F232 для Keil RealView MDK-ARM (с ограничением программного кода 32 Кбайт)
EKI-LM4F232	Оценочный набор Stellaris LM4F232 для IAR Systems Embedded Workbench® (с ограничением программного кода 32 Кбайт)
EKC-LM4F232	Оценочный набор Stellaris LM4F232 для CodeSourcery tools (с ограничением по времени использования — 30 дней)
EKT-LM4F232	Оценочный набор Stellaris LM4F232 для Code Red Technologies Tools (с ограничением по времени использования — 90 дней)
EKS-LM4F232	Оценочный набор Stellaris LM4F232 для Code Composer Studio™ IDE (код привязан к плате)

СПИСОК ЛИТЕРАТУРЫ

1. **Cortex-M3 Technical Reference Manual (TRM)**:
   может быть загружен с сайта компании ARM по адресу
   http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337g/index.html.

2. **ARMv7-M Architecture Application Level Reference Manual**:
   может быть загружен с сайта компании ARM по адресу
   www.arm.com/products/CPUs/ARM_Cortex-M3_v7.html.

3. **CoreSight Technology System Design Guide**:
   может быть загружен с сайта компании ARM по адресу
   http://infocenter.arm.com/help/topic/com.arm.doc.dgi0012b/index.html.

4. **AMBA Specification**:
   может быть загружен с сайта компании ARM по адресу
   www.arm.com/products/solutions/AMBA_Spec.html.

5. **AAPCS Procedure Call Standard for the ARM Architecture**:
   может быть загружен с сайта компании ARM по адресу
   http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042c/index.html.

6. **RVCT 4.0 Compilation Tools Compiler User Guide**:
   может быть загружен с сайта компании ARM по адресу
   http://infocenter.arm.com/help/topic/com.arm.doc.dui0205i/index.html.

7. **ARM Application Note 179: Cortex-M3 Embedded Software Development**:
   может быть загружен с сайта компании ARM по адресу
   http://infocenter.arm.com/help/topic/com.arm.doc.dai0179b/index.html.

8. **RVCT 4.0 Compilation Tools Compiler Reference Guide**:
   может быть загружен с сайта компании ARM по адресу
   http://infocenter.arm.com/help/topic/com.arm.doc.dui0348b/index.html.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Н

О

П

Р

С

Т

У

Ц

Ш

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: **115487, г. Москва, 2-й Нагатинский пр-д, д. 6А**.

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 интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. +7 (499) 782-38-89.

Электронный адрес: **books@alians-kniga.ru**.

Джозеф Ю

Ядро Cortex-M3 компании ARM
Полное руководство

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com

Ответственный редактор *Т. Е. Брод*
Верстальщик *А. Ю. Анненков*

Формат 70x100/16. Бумага офсетная.
Гарнитура «Minion Pro». Печать офсетная.
Объем 34,5 п. л. Усл. п. л. 44,7
Тираж 100 экз.

Издательство ДМК Пресс
Веб-сайт издательства: www.дмк.рф