Интегрированная среда разработки и отладки программ MC Studio

Руководство программиста

Листов 235

Порядок использования данного документа

Настоящая документация охраняется действующим законодательством Российской Федерации об авторском праве и смежных правах, в частности, законом Российской Федерации «Об авторском праве и смежных правах». ГУП НПЦ «ЭЛВИС» является единственным правообладателем исключительных авторских прав на настоящую документацию.

Настоящую документацию, не иначе как по предварительному согласию ГУП НПЦ «ЭЛВИС», запрещается:

- воспроизводить, т.е. изготавливать один или более экземпляров настоящей документации, ее части, в любой форме, любым способом;
- сдавать в прокат;
- публично показывать, исполнять или сообщать для всеобщего сведения,
- переводить;
- переделывать или другим образом перерабатывать (дорабатывать).

ГУП НПЦ «ЭЛВИС» оставляет за собой право в любой момент вносить изменения (дополнения) в настоящую документацию без предварительного уведомления о таком изменении (дополнении).

ГУП НПЦ «ЭЛВИС» не несет ответственности за вред, причиненный при использовании настоящей документации.

Передача настоящей документации не означает передачи каких-либо авторских прав ГУП НПЦ «ЭЛВИС» на нее.

Возникновение каких-либо прав на материальный носитель, на котором передается настоящая документация, не влечет передачи каких-либо авторских прав на данную документацию.

Все указанные в настоящей документации товарные знаки принадлежат их владельцам.

Оглавление

1. Вступление

Проект программы в среде **MultiCore Studio** состоит из модулей RISC и DSP, содержащих файлы с текстом программы на языках *C* (только RISC) и *Ассемблера*. Проект может не содержать модулей DSP, но как минимум один модуль RISC должен присутствовать.

В данной книге подробно рассматривается процесс написания программ в среде **MultiCore Studio**. В книгу вошли следующие главы:

- Программирование под RISC - здесь рассматривается написание программы для ядра RISC на языке *Ассемблера* или *C*, а также взаимосвязь RISC с ядром DSP;
- Программирование под DSP - глава описывает процесс создания программы для ядра DSP на языке *Ассемблера* ***Elas***;
- Порты ввода-вывода - в этой главе рассматривается архитектура портов ввода-вывода ИМС "*МУЛЬТИКОР*", а также способы работы с ними;
- DMA - данная глава описывает способы использования каналов **DMA** для обмена данными;
- Системный управляющий сопроцессор - здесь рассматриваются регистры сопроцессора **CP0**;
- Обработка исключений - в этой главе описываются все возможные в ИМС "*МУЛЬТИКОР*" исключения, расположение их векторов и способы их обработки;
- Обработка прерываний - данная глава включает в себя описание всех возможных прерываний в ИМС "*МУЛЬТИКОР*";
- Таймеры - здесь рассматриваются таймеры ИМС "*МУЛЬТИКОР*", а также способы работы с ними;
- Создание симулятора внешнего устройства - в данной главе рассматривается процесс написания библиотеки DLL симулятора внешнего устройства, подключаемого к симулятору ИМС "*МУЛЬТИКОР*".

В конце книги приведены:

- Примеры проектов для среды **MC Studio**;
- Пример симулятора внешнего устройства.

2. Программирование под RISC

2.1. Введение

Ядро RISC процессора MultiCore при работе программы является ведущим (*master*). Ядру доступны все ресурсы процессора, в то время как ядру DSP доступны только его внутренние ресурсы. В данной главе речь пойдет о написании программ для ядра RISC. Глава включает в себя:

- Карту памяти процессора MultiCore;
- Описание регистров ядра RISC;
- Описание программы для ядра RISC, написанной на языке C;
- Описание программы для ядра RISC, написанной на языке Ассемблера;
- Способы задания символических имен;
- Способы записи выражений (непосредственных операндов);
- Размещение в памяти секций текста и данных RISC;
- Способы задания макроопределений в программе RISC;
- Краткий обзор системы команд RISC-ядра;
- Способы взаимодействия ядра RISC с ядром DSP;
- Способы кэширования команд;
- Методы разработки проектов с оверлейными структурами.

2.2. Память MultiCore

Карта физической памяти MultiCore-12 и MultiCore-24 приведена в таблице 1.1. Коды адреса и данных указаны в шестнадцатеричной системе счисления. Объемы областей памяти указаны с учетом ее дальнейшего расширения.

Таблица 1.1. Карта физической памяти MultiCore.

Диапазон физических адресов	Диапазон виртуальных адресов	Название области	Объем области, Мбайт
FFFF_FFFF 2000_0000	-	Внешняя память	3584
1FFF_FFFF 1C00_0000	BFFF_FFFF BC00_0000	Внешняя память (ПЗУ)	64
1BFF_FFFF 1800_0000	BBFF_FFFF B800_0000	Внутренняя память	64
17FF_FFFF 0000_0000	B7FF_FFFF A000_0000	Внешняя память	384

Вся внешняя память доступна через порт **MPORT**.

Для **CPU** все адресное пространство памяти является 32-разрядным. Память **MEM**, а также внешняя память, могут адресоваться с точностью до байта.

При DMA-обменах вся память является словной (32 разряда) для MultiCore-12. Для MultiCore-24 DMA-обмены могут быть как 32-разрядными, так и 64-разрядными.

Карта внутренней памяти MC-12 приведена в таблице 1.2.1, карта внутренней памяти MC-24 - в таблице 1.2.2:

Таблица 1.2.1. Карта внутренней памяти MultiCore-12.

Диапазон адресов	Название области	Объем области, Кбайт
1BFF_FFFF 1880_0000	Резерв	56000
187F_FFFF 1840_0000	Память и регистры DSP-ядра	4096
183F_FFFF 1830_0000	Резерв	1024
182F_FFFF 182F_0000	Регистры CPU	64
182E_FFFF 1801_0000	Резерв	3000
1800_FFFF 1800_0000	Память CRAM	64

Таблица 1.2.2. Карта внутренней памяти MultiCore-24.

Диапазон адресов	Название области	Объем области, Кбайт
1BFF_FFFF 1880_0000	Резерв	56000
187F_FFFF 1840_0000	Память и регистры DSP-ядра	4096
183F_FFFF 1830_0000	Резерв	1024
182F_FFFF 182F_0000	Регистры CPU	64
182E_FFFF 1800_8000	Резерв	3000
1800_7FFF 1800_0000	Память CRAM	32

Адреса регистров RISC-ядра приведены на странице "Регистры RISC".

Примечание: При выделении места под данные, занимающие меньше 32 бит (типы **short**, **char**), память будет выравниваться по словам.

Пример:

```
char    cA1;
char    cA2;
char    cA3;
short   sA4;
short   sA5;
char    cA6;
```

В данном примере, место выделяется под пять переменных типа **char** (размером в 1 байт) и две переменных типа **short** (размером в 2 байта). Всего места должно быть выделено `1*5 + 2*2 = 9` байт. Тем не менее, при выделении места под переменные, для `cA1`, `cA2` и `cA3` будет выделено три байта, но, так как затем следует переменная `sA4` размером в 2 байта, а в текущем слове свободен только 1 байт, переменная `sA4` попадет в следующее слово. Старший байт предыдущего слова при этом будет пропущен для выравнивания данных по словам, то есть потерян. Чтобы избежать подобных потерь, необходимо самостоятельно выравнивать данные по словам:

```
char    cA1;
char    cA2;
char    cA3;
char    cA6;
short   sA4;
short   sA5;
```

Также для выравнивания возможно использование директив **.align**.

Здесь в первое слово попадут переменные `cA1`, `cA2`, `cA3` и `cA6`, а переменные `sA4` и `sA5` будут размещены во втором слове. Таким образом, данные выровнены по словам и потери байтов памяти не происходит.

2.3. Регистры RISC

На данной странице приводится перечень всех программно доступных регистров RISC-ядра. Для каждого регистра приведено краткое описание, а также виртуальный адрес. При обращении к регистрам в программе RISC следует использовать именно их виртуальные адреса в памяти. Регистры ядра RISC приводятся в таблицах 1.3 - 1.9:

Таблица 1.3. Регистры DMA последовательных портов.

Условное обозначение регистра	Название регистра	Адрес регистра
Регистры DMA		
CSR_SpTx0 CP_SpTx0 IR_SpTx0	Регистр управления и состояния канала SpTx0 Регистр указателя цепочки канала SpTx0 Индексный регистр памяти канала SpTx0	B82F_0000 B82F_0008 B82F_000C
OR_SpTx0 Y_SpTx0	Регистр смещения памяти канала SpTx0 Регистр параметров направления Y при двухмерной адресации памяти канала SpTx0	B82F_0010 B82F_0014
CSR_SpRx0 CP_SpRx0 IR_SpRx0	Регистр управления и состояния канала SpRx0 Регистр указателя цепочки канала SpRx0 Индексный регистр памяти канала SpRx0	B82F_0100 B82F_0108 B82F_010C
OR_SpRx0 Y_SpRx0	Регистр смещения памяти канала SpRx0 Регистр параметров направления Y при двухмерной адресации памяти канала SpRx0	B82F_0110 B82F_0114
CSR_SpTx1 CP_SpTx1 IR_SpTx1	Регистр управления и состояния канала SpTx1 Регистр указателя цепочки канала SpTx1 Индексный регистр памяти канала SpTx1	B82F_0200 B82F_0208 B82F_020C
OR_SpTx1 Y_SpTx1	Регистр смещения памяти канала SpTx1 Регистр параметров направления Y при двухмерной адресации памяти канала SpTx1	B82F_0210 B82F_0214
CSR_SpRx1 CP_SpRx1 IR_SpRx1	Регистр управления и состояния канала SpRx1 Регистр указателя цепочки канала SpRx1 Индексный регистр памяти канала SpRx1	B82F_0300 B82F_0308 B82F_030C
OR_SpRx1 Y_SpRx1	Регистр смещения памяти канала SpRx1 Регистр параметров направления Y при двухмерной адресации памяти канала SpRx1	B82F_0310 B82F_0314

Таблица 1.4. Регистры DMA линковых портов.

Условное обозначение регистра	Название регистра	Адрес регистра
Регистры DMA		
CSR_LpCh0 CP_LpCh0 IR_LpCh0	Регистр управления и состояния канала LpCh0 Регистр указателя цепочки канала LpCh0 Индексный регистр памяти канала LpCh0	B82F_0400 B82F_0408 B82F_040C
OR_LpCh0 Y_LpCh0	Регистр смещения памяти канала LpCh0 Регистр параметров направления Y при двухмерной адресации памяти канала LpCh0	B82F_0410 B82F_0414
CSR_LpCh1 CP_LpCh1 IR_LpCh1	Регистр управления и состояния канала LpCh1 Регистр указателя цепочки канала LpCh1 Индексный регистр памяти канала LpCh1	B82F_0500 B82F_0508 B82F_050C
OR_LpCh1 Y_LpCh1	Регистр смещения памяти канала LpCh1 Регистр параметров направления Y при двухмерной адресации памяти канала LpCh1	B82F_0510 B82F_0514
CSR_LpCh2 CP_LpCh2 IR_LpCh2	Регистр управления и состояния канала LpCh2 Регистр указателя цепочки канала LpCh2 Индексный регистр памяти канала LpCh2	B82F_0600 B82F_0608 B82F_060C
OR_LpCh2 Y_LpCh2	Регистр смещения памяти канала LpCh2 Регистр параметров направления Y при двухмерной адресации памяти канала LpCh2	B82F_0610 B82F_0614
CSR_LpCh3 CP_LpCh3 IR_LpCh3	Регистр управления и состояния канала LpCh3 Регистр указателя цепочки канала LpCh3 Индексный регистр памяти канала LpCh3	B82F_0700 B82F_0708 B82F_070C
OR_LpCh3 Y_LpCh3	Регистр смещения памяти канала LpCh3 Регистр параметров направления Y при двухмерной адресации памяти канала LpCh3	B82F_0710 B82F_0714

Таблица 1.5. Регистры DMA каналов MemCh.

Условное обозначение регистра	Название регистра	Адрес регистра
	Регистры DMA	
CSR_MemCh0 IOR_MemCh0	Регистр управления и состояния канала MemCh0 Регистр индекса и смещения внутренней памяти канала MemCh0	B82F_0800 B82F_0804
CP_MemCh0 IR_MemCh0	Регистр указателя цепочки канала MemCh0 Индексный регистр внешней памяти канала Memh0	B82F_0808 B82F_080C
OR_MemCh0 Y_MemCh0 Run0	Регистр смещения внешней памяти канала MemCh0 Регистр параметров направления Y при двухмерной адресации внешней памяти канала MemCh0 Псевдорегистр управления состоянием бита RUN регистра CSR_MemCh0	B82F_0810 B82F_0814 B82F_0818
CSR_MemCh1 IOR_MemCh1	Регистр управления и состояния канала MemCh1 Регистр индекса и смещения внутренней памяти канала MemCh1	B82F_0900 B82F_0904
CP_MemCh1 IR_MemCh1	Регистр указателя цепочки канала MemCh1 Индексный регистр внешней памяти канала MemCh1	B82F_0908 B82F_090C
OR_MemCh1 Y_MemCh1 Run1	Регистр смещения внешней памяти канала MemCh1 Регистр параметров направления Y при двухмерной адресации внешней памяти канала MemCh1 Псевдорегистр управления состоянием бита RUN регистра CSR_MemCh1	B82F_0910 B82F_0914 B82F_0918
CSR_MemCh2 IOR_MemCh2	Регистр управления и состояния канала MemCh2 Регистр индекса и смещения внутренней памяти канала MemCh2	B82F_0A00 B82F_0A04
CP_MemCh2 IR_MemCh2	Регистр указателя цепочки канала MemCh2 Индексный регистр внешней памяти канала Memh2	B82F_0A08 B82F_0A0C
OR_MemCh2 Y_MemCh2 Run2	Регистр смещения внешней памяти канала MemCh2 Регистр параметров направления Y при двухмерной адресации внешней памяти канала MemCh2 Псевдорегистр управления состоянием бита RUN регистра CSR_MemCh2	B82F_0A10 B82F_0A14 B82F_0A18
CSR_MemCh3 IOR_MemCh3	Регистр управления и состояния канала MemCh3 Регистр индекса и смещения внутренней памяти канала MemCh3	B82F_0B00 B82F_0B04
CP_MemCh3 IR_MemCh3	Регистр указателя цепочки канала MemCh3 Индексный регистр внешней памяти канала Memh3	B82F_0B08 B82F_0B0C
OR_MemCh3 Y_MemCh3 Run3	Регистр смещения внешней памяти канала MemCh3 Регистр параметров направления Y при двухмерной адресации внешней памяти канала MemCh3 Псевдорегистр управления состоянием бита RUN регистра CSR_MemCh3	B82F_0B10 B82F_0B14 B82F_0B18

Таблица 1.6. Регистры линковых портов.

Условное обозначение регистра	Название регистра	Адрес регистра
Регистры линковых портов		
LTx0	Буфер передачи порта LPORT0	B82F_7000
LRx0	Буфер приема порта LPORT0	B82F_7000
LCSR0	Регистр управления и состояния порта LPORT0	B82F_7004
LDIR0	Регистр управления порта ввода-вывода LPORT0	B82F_7008
LDR0	Регистр данных порта ввода-вывода LPORT0	B82F_700C
LTx1	Буфер передачи порта LPORT1	B82F_8000
LRx1	Буфер приема порта LPORT1	B82F_8000
LCSR1	Регистр управления и состояния порта LPORT1	B82F_8004
LDIR1	Регистр управления порта ввода-вывода LPORT1	B82F_8008
LDR1	Регистр данных порта ввода-вывода LPORT1	B82F_800C
LTx2	Буфер передачи порта LPORT2	B82F_9000
LRx2	Буфер приема порта LPORT2	B82F_9000
LCSR2	Регистр управления и состояния порта LPORT2	B82F_9004
LDIR2	Регистр управления порта ввода-вывода LPORT2	B82F_9008
LDR2	Регистр данных порта ввода-вывода LPORT2	B82F_900C
LTx3	Буфер передачи порта LPORT3	B82F_A000
LRx3	Буфер приема порта LPORT3	B82F_A000
LCSR3	Регистр управления и состояния порта LPORT3	B82F_A004
LDIR3	Регистр управления порта ввода-вывода LPORT3	B82F_A008
LDR3	Регистр данных порта ввода-вывода LPORT3	B82F_A00C

Таблица 1.7. Регистры последовательных портов.

Условное обозначение регистра	Название регистра	Адрес регистра
Регистры портов обмена последовательным кодом		
STx0	Буфер передачи данных порта SPOTR0	B82F_5000
Rx0	Буфер приема данных SPOTR0	B82F_5000
STCTL0	Регистр управления передачей данных SPOTR0	B82F_5004
SRCTL0	Регистр управления приемом данных SPOTR0	B82F_5008
TDIV0	Регистр коэффициентов деления при передаче данных SPOTR0	B82F_500C
RDIV0	Регистр коэффициентов деления при приеме данных SPOTR0	B82F_5010
MTCS0	Выбор канала передачи данным в многоканальном режиме SPOTR0	B82F_5014
MRCS0	Выбор канала приема данным в многоканальном режиме SPOTR0	B82F_5018
KEYWD0	Регистр кода сравнения SPOTR0	B82F_501C
KEYMASK0	Регистр маски сравнения SPOTR0	B82F_5020
MRCE0	Выбор канала для сравнения принимаемых данных SPOTR0	B82F_5024
STx1	Буфер передачи данных порта SPOTR1	B82F_6000
SRx1	Буфер приема данных SPOTR1	B82F_6000
STCTL1	Регистр управления передачей данных SPOTR1	B82F_6004
SRCTL1	Регистр управления приемом данных SPOTR1	B82F_6008
TDIV1	Регистр коэффициентов деления при передаче данных SPOTR1	B82F_600C
RDIV1	Регистр коэффициентов деления при приеме данных SPOTR1	B82F_6010
MTCS1	Выбор канала передачи данным в многоканальном режиме SPOTR1	B82F_6014
MRCS1	Выбор канала приема данным в многоканальном режиме SPOTR1	B82F_6018
KEYWD1	Регистр кода сравнения SPOTR1	B82F_601C
KEYMASK1	Регистр маски сравнения SPOTR1	B82F_6020
MRCE1	Выбор канала для сравнения принимаемых данных SPOTR1	B82F_6024

Таблица 1.8. Регистры порта UART.

Условное обозначение регистра	Название регистра	Адрес регистра
Регистры UART		
RBR	Приемный буферный регистр	182F_3000
THR	Передающий буферный регистр	182F_3000
IER	Регистр разрешения прерываний	182F_3004
IIR	Регистр идентификации прерывания	182F_3008
FCR	Регистр управления FIFO	182F_3008
LCR	Регистр управления линией	182F_300C
MCR	Регистр управления модемом	182F_3010
LSR	Регистр состояния линии	182F_3014
MSR	Регистр состояния модемом	182F_3018
SPR	Регистр Scratch Pad	182F_301C
DLL	Регистр делителя младший	182F_3000
DLM	Регистр делителя старший	182F_3004
SCLR	Регистр предделителя (scaler)	182F_3014

Таблица 1.9. Регистры таймеров, порта внешней памяти и системные регистры.

Условное обозначение регистра	Название регистра	Адрес регистра
Регистры интервального таймера (IT)		
ITCSR	Регистр управления	182F_D000
ITPERIOD	Регистр периода работы таймера	182F_D004
ITCOUNT	Регистр счетчика	182F_D008
ITSCALE	Регистр предделителя	182F_D00C
Регистры WDT		
WTCSR	Регистр управления	182F_D010
WTPERIOD	Регистр периода работы таймера	182F_D014
WTCOUNT	Регистр счетчика	182F_D018
WTSCALE	Регистр предделителя	182F_D01C
Регистры RTT		
RTCSR	Регистр управления	182F_D020
RTPERIOD	Регистр периода работы таймера	182F_D024
RTCOUNT	Регистр счетчика	182F_D028
Регистры порта внешней памяти		
CSCON0	Регистр конфигурации 0.	182F_1000
CSCON1	Регистр конфигурации 1.	182F_1004
CSCON2	Регистр конфигурации 2.	182F_1008
CSCON3	Регистр конфигурации 3.	182F_100C
CSCON4	Регистр конфигурации 4.	182F_1010
SDRCON	Регистр конфигурации памяти SDRAM	182F_1014
CKE_CTR	Регистр управления состоянием вывода CKE микросхемы	182F_1018
Системные регистры		
MASKR	Регистр маски	182F_4000
QSTR	Регистр заявок	182F_4004
CSR	Регистр управления	182F_4008

Примечание: Для удобства работы с регистрами MultiCore-12 и MultiCore-24 существуют заголовочные файлы *memory_12, memory_12_asm.h*, *memory_24.h* и *memory_24_asm.h*, позволяющие обращаться к регистрам MultiCore-12 и MultiCore-24 по их именам.

Файлы *memory_12_asm.h/memory_24_asm.h* следует использовать при написании программы RISC на языке Ассемблера. В этих файлах задан символ `CPU_BASE`, определяющий базовый адрес регистров RISC в памяти. Для каждого регистра символ в *memory_12_asm.h/memory_24_asm.h* определяется как смещение от этого базового адреса.

Пример:

```
lui $30,CPU_BASE
li  $2,1
sw  $2,DLL($30)
sw  $0,LCR($30)
```

В этом примере данные загружаются из регистров общего назначения в регистры **DLL** и **LCR** порта **UART**. Символы `DLL` и `LCR` определены в файлах *memory_12_asm.h/memory_24_asm.h* как смещения от базового адреса `CPU_BASE`, предварительно загруженного в регистр `$30`. Описание регистров общего назначения приводится ниже.

Файлы *memory_12.h/memory_24.h* предназначены для использования в программе RISC, написанной на языке *C*. **Пример**:

```
int i=0xFFFFFFFF;
MASKR=i;
```

В данном примере в регистр **MASKR** помещается значение переменной `i`, то есть `0xFFFFFFFF`.

Для подключения файлов *memory_12.h/memory_24.h* и *memory_12_asm.h/memory_24_asm.h* к программе необходимо использовать директивы `#include` и **`.include`** соответственно.

Кроме приведенных в таблицах регистров, в ядре RISC присутствуют регистры общего назначения (**GPR** - ***G**eneral **P**urpose **R**egisters*). Перечень этих регистров приведен в таблице 1.10:

Таблица 1.10. Регистры общего назначения.

Обозначение	Название	Обозначение	Название
$0	zero	$16	s0
$1	at	$17	s1
$2	v0	$18	s2
$3	v1	$19	s3
$4	a0	$20	s4
$5	a1	$21	s5
$6	a2	$22	s6
$7	a3	$23	s7
$8	t0	$24	t8

$9	t1	$25	t9
$10	t2	$26	k0
$11	t3	$27	k1
$12	t4	$28	gp
$13	t5	$29	sp
$14	t6	$30	s8
$15	t7	$31	ra

Обращения к регистрам общего назначения в программе RISC производится по указанным в таблице **обозначениям**. Например, инструкция `lui $30,0xb844` загружает в старшую часть регистра **s8** непосредственный операнд `0xB844`.

Примечание 1: Регистр **zero** доступен только по чтению. В этом регистре всегда находится нуль. Регистр **zero** используется, например, в операциях сравнения с нулем.

Примечание 2: Регистр **ra** используется для хранения адреса возврата при использовании переходов с возвратом. Если в программе используются подобные переходы, не следует вручную менять содержимое этого регистра.

Среди регистров общего назначения присутствуют также регистры **HI** и **LO**, предназначенные для хранения результатов операций умножения и деления. Доступ к регистрам осуществляется только операциями **MTHI**, **MTLO**, **MFHI**, **MFLO**. Подробнее об этом см. книгу "**RISC Instructions Set**".

В таблице 1.11 приведен список регистров системного управляющего сопроцессора **CP0**:

Таблица 1.11. Регистры системного управляющего сопроцессора.

INDEX	Индекс матрицы TLB	$0
RANDOM	Случайным образом сгенерированный индекс для матрицы TLB	$1
ENTRYL o0	Младшая часть строки TLB для виртуальных страниц с четными номерами	$2
ENTRYL o1	Младшая часть строки TLB для виртуальных страниц с нечетными номерами	$3
CONTEXT	Указатель на PTE	$4
PageMask	Управление размером страницы во вхождениях TLB	$5
WIRED	Управление количеством закрепленных "привязанных" строк TLB	$6
BadVAddr	Содержит адрес, вызвавший последнее связанное с адресацией исключение	$8
COUNT	Счетчик процессорных циклов	$9
ENTRYHI	Старшая часть строки TLB	$10
COMPARE	Управление прерыванием от таймера	$11
SR	Регистр состояния	$12
CAUSE	Код причины последнего исключения	$13
EPC	Виртуальный адрес команды (состояние PC), вызвавшей исключение	$14
PRID	Идентификатор версии процессора	$15
CONFIG	Конфигурационный регистр	$16
CONFIG1	Конфигурационный регистр 1	$16.sel
LLAddr	Содержит физический адрес, по которому выполняется последняя команда LL	$17
ErrorEPC	Содержит состояние PC в момент возникновения последнего исключения по ошибке	$30

Регистры **CP0** доступны в программе RISC посредством операций **`mtc0`** и **`mfc0`**. Эти операции описаны в книге "**RISC Instructions Set**".

Регистры DSP-ядра также доступны из программы RISC. Способы доступа к этим регистрам приводятся на странице "Доступ к регистрам DSP-ядра".

2.4. Программа на языке С

Так как ядро RISC процессора MultiCore является ведущим, основной задачей программы RISC является управление. То есть программа RISC осуществляет обработку исключений и прерываний, обмен данными с портами, загрузку в регистры и память ядра DSP необходимых данных, а также запуск программы DSP на исполнение.

Программа для ядра RISC может быть написана как на языке *C*, так и на языке *Ассемблера*. Язык *C*, используемый для написания программы RISC соответствует стандарту *ANSI C*.

Чтобы использовать в проекте программы RISC, написанные на языке *C*, необходимо при создании проекта указать используемый набор инструментов (**Tools Set**) равный *\Tools4\bin\m12nlib.ini* или *\Tools4\bin\m24nlib.ini*. Это требуется для подключения библиотеки *Lib*, необходимой для правильной работы программ, написанных на языке *C*.

При написании программы на языке *C* рекомендуется использовать следующие файлы:

- *memory_12.h/memory_24.h* - этот заголовочный файл содержит описание указателей на регистры процессора MultiCore-12, а также на начало памяти **PRAM** и **XRAM** ядра DSP. Используя этот заголовочный файл, программист получает возможность обращаться к регистрам MultiCore-12 по именам, определенным в *memory_12.h/memory_24.h* (например, `PC` - регистр программного счетчика **PC** (DSP)). Если же файл *memory_12.h/memory_24.h* не подключен, обращения к регистрам возможны только по их адресам.
- *OVR.c* и *OVR.h* - эти два файла необходимы для работы с проектами, имеющими динамически подгружаемые секции программы DSP. Подробнее об этом см. главу "Проекты с оверлейными структурами".

Рассмотрим пример простой программы, написанной для ядра RISC на языке *C*:

```
#include "memory_12.h"

extern int Start_op;
extern int InA;
extern int InB;
extern int OutX;

void exit();
void Initialize();

main()
{
```

```
    int       InputA=10;
    int       InputB=100;
    int       OutputX;
    Initialize();
    InA=InputA;
    InB=InputB;
    DCSR=0x4000;
    while ((~(QSTR))&(1<<31));
    OutputX=OutX;
    exit();
}

void exit()
{
     while (1);
}

void Initialize()
{
     PC=((unsigned int)&Start_op - (unsigned int)&PRAM)>>2;
     A0=((unsigned int)&InA - (unsigned int)&XRAM)>>2;
     A1=((unsigned int)&InB - (unsigned int)&XRAM)>>2;
     A2=((unsigned int)&OutX - (unsigned int)&XRAM)>>2;
}
```

Прежде всего, директивой `#include` подключается файл *memory_12.h*.

Далее при помощи конструкции **`extern int`** `имя_символа` объявляется несколько внешних символов. Это могут быть символы, определенные в других частях программы RISC, или же символы программы DSP. Подробнее о символах RISC и DSP см. соответственно страницы "Символы RISC" и "Символы DSP".

В данном примере внешние символы используются для установки адреса первой исполняемой инструкции программы DSP, а также для обмена данными с памятью DSP-ядра как показано ниже.

Затем в программе объявляются два заголовка функций - **`void`** `exit();` и **`void`** `Initialize();`. Описание этих функций приведено после описания функции `main()`.

Функция `main()` является функцией старта программы. После исполнения функций инициализации библиотеки *Lib*, программа перейдет к исполнению именно этой функции. В теле функции `main()` выполняются те или иные инструкции *C*, а также осуществляются вызовы подпрограмм (в данном случае это функции `Initialize()` и `exit()`). В начале функции объявляются три переменных типа **`int`**: `InputA=10`, `InputB=100` и `OutputX`. Эти переменные будут использоваться для обмена данными с ядром DSP. Затем осуществляется вызов функции `Initialize()`.

Функция `Initialize()` задает начальные значения регистрам ядра DSP, а именно регистру программного счетчика **PC** и трем регистрам устройства генерации адреса **AGU**: `A0`, `A1`, `A2`. Значение **PC** задается как адрес внешнего символа `Start_op` минус адрес начала программной памяти DSP (символ `PRAM`, определенный в файле *memory_12.h*).

Значения адресных регистров `A0`, `A1`, `A2` задаются как разность адреса внешнего символа и символа `XRAM` - начала памяти данных. Таким образом, программа RISC установила в **PC** адрес первой исполняемой инструкции, а в адресных регистрах **A0**, **A1**, **A2** - адреса трех ячеек памяти секции данных DSP.

После вызова функции `Initialize()` осуществляется обмен данными с ядром DSP, запуск программы DSP и ожидание ее останова. Подробнее об этом см. главу "Взаимодействие с ядром DSP".

По окончании программы RISC осуществляется вызов функции `exit()`, содержащей бесконечный цикл.

Примечание 1: если в программе RISC на языке *C* необходимо вставить операции *Ассемблера* для RISC, следует использовать следующую конструкцию: `asm("операция");`. Например, `asm("nop");` ассемблируется как операция **NOP** *Ассемблера* для RISC.

Примечание 2: при вычислении адресов, помещаемых в регистры **PC**, **A0**, **A1** и **A2**, каждый адрес делится на 4 (сдвигается на 2 бита вправо). Это необходимо для перехода от байтовой адресации в RISC к словной адресации в DSP.

2.5. Программа на языке Ассемблера

Так как ядро RISC процессора MultiCore является ведущим, основной задачей программы RISC является управление. То есть программа RISC осуществляет обработку исключений и прерываний, обмен данными с портами, загрузку в регистры и память ядра DSP необходимых данных, а также запуск программы DSP на исполнение.

Программа для ядра RISC может быть написана как на языке *C*, так и на языке *Ассемблера*.

Программа на языке Ассемблера состоит из последовательности исходных операторов - по одному оператору на строку кода. Оператор программы Ассемблера может включать в себя до трех полей: поле метки, поле операции и поле операндов. Поля отделяются друг от друга некоторым числом пробелов или символов табуляции. При этом строчные и прописные буквы считаются эквивалентными при записи мнемоник команд, директив, кодов условий и имен регистров, но отличаются при записи меток и литерных констант.

Поле метки

Поле символического имени (метки) всегда должно быть первым полем оператора программы на языке Ассемблера. Если метка не задана, первым полем будет поле операции. В таком случае, для повышения читабельности текста программы, рекомендуется ставить перед полем операции несколько пробелов или символов табуляции.

Принципы ввода символических имен (меток) в поле метки изложены на странице "Символы RISC".

Поле операции

Поле операции обязательно должно отделяться от поля метки как минимум одним пробелом или символом табуляции.

В поле операции возможно использовать:

- мнемоническое имя команды (см. книгу "**RISC Instructions Set**");
- директиву Ассемблера для RISC (см. книгу "**RISC Tools. User's Guide**", главу "**Директивы ассемблера**");
- вызов макроопределения.

Поле операндов

Поле операндов также отделяется от поля метки пробелом и представляет собой набор аргументов для операции. Аргументы должны отделяться друг от друга запятыми. В качестве аргументов могут быть использованы регистры и выражения. При обращении к регистру по номеру используется следующая нотация: `$номер_регистра`.

Выражения в поле операндов вводятся в соответствии с правилами, определенными на странице "Выражения".

Файл с программой для RISC состоит из одной и более секций. Это могут быть секции текста и данных. Начало секции текста определяется директивой **`.text`**, а секции данных - директивой **`.data`**. Если в файле необходимо определить несколько различных секций текста или данных, необходимо указывать в начале каждой секции ее имя и заканчивать каждую секцию директивой **`.end`**. Подробнее данные директивы рассмотрены в главе "**Директивы ассемблера**"книги "**RISC Tools. User's Guide**". Размещение в памяти секций программы RISC-ядра рассматривается на странице "Секции RISC".

Для задания точки входа программы RISC используется директива **`.ent`** `имя_символа`. При этом символ, указанный в директиве, должен быть задан в контексте секции текста, содержащей директиву.

Для подключения к программе RISC дополнительных файлов, задающих, например, необходимые для работы символы, следует использовать директиву **`.include`** `"имя_файла"`. Рекомендуется подключать к программе RISC, написанной на языке Ассемблера, файл *memory_12_asm.h/memory_24_asm.h*, описывающий регистры процессора MultiCore и позволяющий использовать определенные в этом файле символы для обращения к регистрам.

Для записи комментариев в программе RISC, написанной на языке *Ассемблера*, следует использовать символ **#**. Для записи комментариев в несколько строк также применима нотация *C* - размещение комментариев между символов **`/*`** и **`*/`**.

Для использования в программе на языке Ассемблера ASCII-символов следует использовать константы в виде `'символ`. Данная комбинация заменяется кодом символа и может быть использована в выражениях. Например, `'#`.

При необходимости сформировать в памяти строку ее следует записывать в кавычках `""`. Например, `"abc"` При этом используется нотация языка *C* для вставки спецсимволов:

- `\\` - вставка символа "\";
- `\b` - возврат на позицию (код `010`);
- `\f` - перевод страницы (код `014`);
- `\n` - перевод строки (код `012`);
- `\r` - перевод каретки (код `015`);
- `\t` - табуляция (код `011`);
- `\цифра цифра цифра` - задание числа в восьмеричной системе;
- `\x цифра цифра` - задание числа двумя шестнадцатеричными цифрами;
- `\"` - вставка кавычки.

Если необходимо записать определенный набор операторов несколько раз и, возможно, с разными параметрами, - для этого следует использовать директивы цикла **`.irp`**, **`.irpc`** и **`.rept`**. Эти директивы описаны в главе "**Директивы ассемблера**" книги "**RISC Tools. User's Guide**".

При помощи директив условного ассемблирования в тексте программы RISC возможно задавать блоки кода, ассемблируемые компилятором только если выполняются те или иные условия. Директивы условного ассемблирования описаны в главе "**Условное ассемблирование**" книги "**RISC Tools. User's Guide**".

При помощи директивы **`.macro`** в программе RISC на языке *Ассемблера* возможно определять макросы. Подробнее об этом см. страницу "Макроопределения".

Краткий обзор системы инструкций *Ассемблера* для RISC приводится на странице "Система команд".

Пример программы для RISC-ядра, написанной на языке Ассемблера, рассматривается в главе "Примеры создания проектов".

2.6. Символы RISC

Символические имена (**метки**) в программе RISC, написанной на языке Ассемблера применимы в инструкциях RISC для программных переходов, ветвлений, вызовов подпрограмм и записи/чтения данных из памяти.

Для записи символического имени можно использовать любую комбинацию букв (латинского алфавита), цифр и символов подчеркивания (_). Первым символом имени должна быть буква. Прописные и строчные буквы символического имени считаются различными, то есть метки `ABCDEF` и `Abcdef` являются разными.

При **задании символического имени** (сопоставлении его с инструкцией или ячейкой памяти), после имени символа следует ставить знак двоеточия (**:**). Символическое имя в программе RISC может быть задано как с первой позиции строки, так и после нескольких пробелов.

Пример 1:

```
My_Instruction:
  MOVE $4,$0
```

Задается символическое имя `My_Instruction`, сопоставляемое с адресом инструкции **`MOVE`** `$4,$0` в памяти.

Пример 2:

```
  My_Data:
.word 0
```

Задается символическое имя `My_Data`, сопоставляемое с адресом 32-разрядного слова, зарезервированного в памяти директивой **`.word`** `0`.

При **обращении к символу** (в инструкциях переходов, ветвлений и т.д.) следует указывать имя символа без двоеточия в конце.

Пример 3:

```
bne $6,$8,My_Instruction
```

Выполняется программный переход (в случае неравенства содержимого регистров `$6` и `$8`) по адресу, сопоставленному с символическим именем `My_Instruction`. При обращении к символу двоеточие не ставится.

Если символ, к которому происходит обращение, не определен в контексте данной секции, компилятор по умолчанию считает его внешним. В том случае, когда необходимо сделать символ доступным в других секциях RISC, его следует объявить глобальным посредством директивы **`.global`** `имя_символа`, если программа написана на языке *Ассемблера*. Если же программа написана на языке *C*, символ глобальным объявлять не нужно.

Недопустимо использовать глобальные символы в качестве аргументов выражений, заданных не в той секции, где определен символ. Это ограничение установлено в связи с тем, что результаты выражений подсчитываются на этапе компиляции секции, а адреса глобальных символов становятся доступны только после компоновки.

Примечание: при определении символического имени, указывающего на конец секции текста или данных, рекомендуется задавать символ перед последней инструкцией (данными) секции. При этом, инструкция (данные) должны быть однословными, так как символу будет соответствовать адрес первого слова инструкции (данных).

Если же символ задан после всех инструкций (данных), ему будет соответствовать адрес, следующий за последним адресом секции. С этого же адреса может компоноваться следующая секция. В таком случае метка ее начала будет недоступна отладчику. Это не повлияет на работу программы, но информация об исполняемой в данный момент секции в процессе отладки может быть некорректна.

Примечание 2: в программе RISC, написанной на языке *C*, имена функций и переменных также являются символами.

2.7. Выражения

В программе RISC допустимо вводить в качестве операндов инструкций или директив **выражения**.

Выражение определяет адрес или численное значение и состоит из аргументов и операторов.

Аргументы выражений

Аргументами выражений могут быть символические имена, числа, а также другие выражения, заключенные в скобки или использующие операторы префикса.

Операторы выражений

Операторами выражений являются арифметические функции, такие как `%` или `+`. В тексте выражения допустимо отделять аргументы от операторов при помощи пробелов. Различают два типа операторов - операторы префикса и операторы инфикса.

Операторы префикса имеют один аргумент, который следует в тексте выражения сразу за оператором. В выражениях программы RISC-ядра допустимы следующие операторы префикса:

- `-` Отрицание в дополнительном коде;
- `~` Побитовое НЕ.

Операторы инфикса располагаются между двумя аргументами. Допустимо использовать следующие операторы инфикса:

1. Операторы высокого приоритета

- `*` Умножение;
- `/` Деление. Если аргументами деления являются целые числа, дробная часть будет отброшена;
- `%` Остаток от деления;
- `<` `<<` Сдвиг влево. Идентичен оператору '`<<`' языка С;
- `>` `>>` Сдвиг вправо. Идентичен оператору '`>>`' языка С.

2. Операторы среднего приоритета

- `|` Побитовое ИЛИ;
- `&` Побитовое И;
- `^` Побитовое исключающее ИЛИ;
- `!` Побитовое ИЛИ-НЕ.

3. Операторы низкого приоритета

- `+` Сложение;
- `-` Вычитание.

Примечание: операторы с одинаковым приоритетом вычисляются компилятором слева направо.

Пустое выражение не имеет значения - это либо пробел, либо нуль. В любом месте текста программы, где требуется ввести абсолютное выражение, допускается пропустить выражение (например, в директивах выделения памяти, таких как **`.word`**). Компилятор подставит вместо пропущенного выражения `0`.

Значение выражения вычисляется на этапе компиляции и подставляется в программу, это может быть число или смещение. Если компилятор не будет обладать всей информацией, необходимой для вычисления выражений, компиляция будет прервана и появится сообщение об ошибке. Выражение не может быть вычислено компилятором в следующих случаях:

- Выражение не абсолютно, то есть в выражении присутствует неопределенный аргумент. Неопределенными аргументами являются символические имена, заданные вне секции, содержащей выражение.
- В выражении присутствуют недопустимые (неподдерживаемые) операторы.

2.8. Секции RISC

При компоновке, программа RISC разделяется на секции текста и данных. Секции текста содержат код программы, в то время как секции данных включают все переменные и константы, определенные в программе *C*, а также все ячейки памяти, зарезервированные в программе Ассемблера посредством директив выделения памяти (таких как **`.space`**, **`.word`** и т.д.).

Настройка размещения секций RISC в памяти MultiCore осуществляется посредством диалога настроек проекта (пункт *Settings* меню **Project**). В данном диалоге для каждой секции следует вводить имя, адрес внутри модуля (**Unit VMA**) и адрес фактического размещения секции в памяти. Для секций текста и данных программы RISC адреса **Unit VMA** и **VMA** должны быть равны. При настройке адресов размещения важно учитывать, что секции не должны перекрываться. Если порядок и место размещения секций в памяти RISC не существенны, рекомендуется задавать **`Unit VMA`**=**`VMA`**=`0xBFC00000` для всех секций RISC. При таких настройках секции будут скомпонованы в памяти RISC последовательно одна за другой.

Также следует учитывать, что секции текста (данных), находящиеся в одном модуле и имеющие одно и то же имя, будут скомпонованы *в одну секцию* текста (данных).

Размещение в памяти секций ядра DSP рассматривается на странице "Секции DSP". Здесь же отметим, что ядро RISC не отличает секции текста и данных DSP - для RISC все эти секции являются секциями данных.

Карта памяти процессора MultiCore приведена на странице "Память MultiCore".

2.9. Макроопределения

Ассемблер для RISC-ядра позволяет программисту определять в тексте программы **макросы** (макроопределения). Макросом называется блок кода, расположенный между директивами **`.macro`** и **`.endm`**, имеющий имя, а также ноль и более параметров. После определения макроса, программист может использовать лишь его имя и набор параметров, вместо того, чтобы вставлять в текст один и тот же код несколько раз. Во время сборки проекта компилятор, обнаружив вызов макроса, автоматически подставит

вместо него блок кода, определяющий макрос. Если при вызове указаны какие-либо параметры, они также будут подставлены в код, если нет - будут подставлены значения параметров по умолчанию.

Для задания макроопределения используется следующий синтаксис:

- Заголовок макроопределения (директива **.macro** имя_макроса);
- Тело макроопределения - блок кода программы;
- Директива **.endm**.

Задание имени макроопределения и параметров может быть выполнено одним из следующих способов:

- Имя_макроопределения **.macro** параметры
- **.macro** Имя_макроопределения параметры
- **.macro** Имя_макроопределения(параметры)

При этом параметры макроса отделяются друг от друга запятыми.

Пример:

```
.macro The_Sum A,B,C=2 ;C=A+B
ADD $\C,$\A,$\B
.endm
```

В приведенном примере в теле макроса в регистр $\C помещается сумма содержимого регистров $\A и $\B. При этом, параметр C имеет значение по умолчанию, равное 2. Если параметр C при использовании макроса не задан (пропущен), компилятор автоматически подставит 2.

Для использования заданного макроса в программе используется запись имя_макроса параметры. Как и при задании макроса, параметры должны быть разделены запятыми. Например, заданный выше макрос используется при помощи записи The_Sum 2,3,4. При этом при компиляции в текст программы вместо имени макроса и параметров будет вставлен оператор ADD $4,$2,$3. Если необходимо пропустить параметр, его значение заменяется пробелом. При этом, запятые, отделяющие пробел от остальных параметров, должны присутствовать.

Как видно из вышеприведенного примера, для выполнения подстановки параметров в теле макроса следует использовать обращение следующего вида: \имя_параметра. Возможна также ссылка на параметр в форме &имя_параметра.

Если в теле макроопределения необходима безусловная вставка некоторого текста, который содержит обрабатываемые символы, то их можно защитить от обработки при помощи заключения в конструкцию следующего вида: \(текст).

Для преждевременного выхода из макроопределения следует использовать директиву **.exitm**. Для удаления - директиву **.purgem**.

Все использованные здесь директивы описаны в главе "**Директивы Ассемблера**" книги "**RISC Tools. User's Guide**".

2.10. Система команд

Для написания программы RISC на языке *Ассемблера* используется набор команд, приведенный в таблицах 1.12-1.20:

Таблица 1.12. Команды чтения/записи.

Условное обозначение	Описание
LB	Load Byte
LBU	Load Byte Unsigned
LH	Load Halfword
LHU	Load Halfword Unsigned
LW	Load Word
LWL	Load Word Left
LWR	Load Word Right
LL	Load Linked Word
LUI	Load Upper Immediate
SB	Store Byte
SH	Store Halfword
SW	Store Word
SWL	Store Word Left
SWR	Store Word Right
SC	Store Conditional Word

Таблица 1.13. Команды ALU с непосредственным операндом.

Условное обозначение	Описание
ADDI	ADD Immediate
ADDIU	ADD Immediate Unsigned
SLTI	Set and Less Then Immediate
SLTIU	Set on Less Than Unsigned Immediate
ANDI	AND Immediate
ORI	OR Immediate
XORI	Exclusive OR Immediate
LUI	Load Upper Immediate

Таблица 1.14. Команды ALU с тремя операндами.

Условное обозначение	Описание
ADD	Add
ADDU	Add Unsigned
CLO	Count Leading Ones
CLZ	Count Leading Zeroes
SUB	Subtract
SUBU	Subtract Unsigned
SLT	Set and Less Than
SLTU	Set on Less Than Unsigned
AND	And
OR	Or
XOR	Exclusive OR
NOR	Nor

Таблица 1.15. Команды сдвига.

Условное обозначение	Описание
SLL	Shift Left Logical
SRL	Shift Right Logical
SRA	Shift Right Arithmetical
SLLV	Shift Left Logical Variable
SRLV	Shift Right Logical Variable
SRAV	Shift Right Arithmetical Variable

Таблица 1.16. Команды умножения и деления.

Условное обозначение	Описание
MUL	Multiply with register write
MULT	Multiply
MULTU	Multiply Unsigned
MADD	Multiply-Add
MADDU	Multiply-Add Unsigned
MSUB	Multiply-Subtract
MSUBU	Multiply-Subtract Unsigned
DIV	Divide
DIVU	Divide Unsigned
MFHI	Move From HI
MFLO	Move From LO
MTHI	Move To HI
MTLO	Move To LO

Таблица 1.17. Команды ветвления.

Условное обозначение	Описание
BEQ	Branch on Equal
BEQL	Branch on Equal Likely
BNE	Branch on Not Equal
BNEL	Branch on Not Equal Likely
BLEZ	Branch on Less than or Equal Zero
BLEZL	Branch on Less than or Equal Zero Likely
BGTZ	Branch on Greater Than Zero
BGTZL	Branch on Greater Than Zero Likely
BLTZ	Branch on Less Than Zero
BLTZL	Branch on Less Than Zero Likely
BGEZ	Branch on Greater Than or Equal Zero
BGEZL	Branch on Greater Than or Equal Zero Likely
BLTZAL	Branch on Less Than Zero And Link
BLTZALL	Branch on Less Than Zero And Link Likely
BGEZAL	Branch on Greater Than or Equal Zero And Link
BGEZALL	Branch on Greater Than or Equal Zero And Link Likely

Таблица 1.18. Команды безусловного перехода.

Условное обозначение	Описание
J	Jump
JAL	Jump and Link
JR	Jump Register
JALR	Jump and Link Register

Таблица 1.19. Специальные команды.

Условное обозначение	Описание
SYSCALL	System Call
BREAK	Breakpoint
ERET	Return from Exception
NOP	No Operation
SYNC	Synchronize Shared Memory
TEQ	Trap if Equal
TEQI	Trap if Equal Immediate
TGE	Trap if Greater Then or Equal
TGEI	Trap if Greater Then or Equal Immediate
TGEIU	Trap if Greater Then or Equal Immediate Unsigned
TGEU	Trap if Greater Then or Equal Unsigned
TLT	Trap if Less Than
TLTI	Trap if Less Than Immediate
TLTIU	Trap if Less Than Immediate Unsigned
TLTU	Trap if Less Than Unsigned
TNE	Trap if Not Equal
TNEI	Trap if Not Equal Immediate
WAIT	Wait for Interrupts

Таблица 1.20. Команды управления сопроцессором CP0.

Условное обозначение	Описание
MTC0	Move To CP0
MFC0	Move From CP0
MOVN	Move Conditional on Not Zero
MOVZ	Move Conditional on Zero
TLBR	Read indexed TLB entry
TLBWI	Write indexed TLB entry
TLBWR	Write Random TLB entry
TLBP	Probe TLB for matching entry

Подробное описание приведенной здесь системы команд рассматривается в книге "**RISC Instructions Set**".

2.11. Взаимодействие с ядром DSP

2.11.1. Введение

В данной главе рассматривается взаимодействие RISC и DSP-ядер процессора MultiCore. При взаимодействии, ядро RISC является ведущим (*master*), а ядро DSP - ведомым (*slave*), то есть управление работой ядра DSP осуществляется из ядра RISC. Далее в этой главе рассматриваются:

- способы доступа к регистрам DSP-ядра;
- методы обмена данными с внутренней памятью DSP-ядра;
- способы запуска программы DSP на исполнение из ядра RISC;
- способы останова программы DSP из ядра RISC;
- ожидание ядром RISC останова программы DSP.

2.11.2. Доступ к регистрам DSP-ядра

Доступ к регистрам DSP-ядра из программы RISC осуществляется одним из следующих способов:

- по виртуальным адресам;
- по символам, определенным в файлах *memory_12_asm.h/memory_24_asm.h* (в программе, написанной на языке *Ассемблера*);
- по указателям, определенным в файлах *memory_12.h/memory_24.h* (в программе, написанной на языке *C*).

Рассмотрим все эти способы.

Виртуальные адреса

Как и регистры ядра RISC, регистры DSP имеют виртуальные адреса в памяти MultiCore. В таблицах 1.21 и 1.22 приводятся адреса для всех программно доступных регистров ядра DSP.

Таблица 1.21. Общие регистры DSP-ядра.

Условное обозначение	Разрядность	Название регистра	Виртуальный адрес регистра
		Общие регистры DSP-ядра	
		PCU	
DCSR	16	Регистр режима работы	0×B848_0100
SR	16	Регистр состояния	0×B848_0104
IDR	16	Регистр-идентификатор	0×B848_0108
PC	16	Программный счетчик	0×B848_0120
SS	16	Стек программного счетчика	0×B848_0124
LA	16	Регистр адреса цикла	0×B848_0128
CSL	16	Стек адреса цикла	0×B848_012C
LC	16	Счетчик цикла	0×B848_0130
CSH	16	Стек счетчика цикла	0×B848_0134
SP	16	Регистр указателя стека	0×B848_0138
SAR	16	Регистр указателя останова	0×B848_013C
CNTR	16	Счетчик исполнения команд	0×B848_0140
		AGU	
A0	16	Регистр адреса A0	0×B848_0080
A1	16	Регистр адреса A1	0×B848_0084
A2	16	Регистр адреса A2	0×B848_0088
A3	16	Регистр адреса A3	0×B848_008C
A4	16	Регистр адреса A4	0×B848_0090
A5	16	Регистр адреса A5	0×B848_0094
A6	16	Регистр адреса A6	0×B848_0098
A7	16	Регистр адреса A7	0×B848_009C
I0	16	Регистр индекса I0	0×B848_00A0
I1	16	Регистр индекса I1	0×B848_00A4
I2	16	Регистр индекса I2	0×B848_00A8
I3	16	Регистр индекса I3	0×B848_00AC
I4	16	Регистр индекса I4	0×B848_00B0
I5	16	Регистр индекса I5	0×B848_00B4
I6	16	Регистр индекса I6	0×B848_00B8
I7	16	Регистр индекса I7	0×B848_00BC
M0	16	Регистр модификатора M0	0×B848_00C0
M1	16	Регистр модификатора M1	0×B848_00C4
M2	16	Регистр модификатора M2	0×B848_00C8
M3	16	Регистр модификатора M3	0×B848_00CC
M4	16	Регистр модификатора M4	0×B848_00D0
M5	16	Регистр модификатора M5	0×B848_00D4
M6	16	Регистр модификатора M6	0×B848_00D8
M7	16	Регистр модификатора M7	0×B848_00DC

Таблица 1.22. Секционные регистры.

		Секционные регистры	
		Адресные генераторы YRAM	
		AGU-Y0	
AT(0)	16	Регистр адреса AT (0-я секция)	0×B848_00E0
IT(0)	16	Регистр индекса IT (0-я секция)	0×B848_00E4
MT(0)	16	Регистр модификатора MT (0-я секция)	0×B848_00E8
DT(0)	16	Регистр модификатора DT (0-я секция)	0×B848_00EC
		AGU-Y1	
AT(1)	16	Регистр адреса AT (1-я секция)	0×B848_00F0
IT(1)	16	Регистр индекса IT (1-я секция)	0×B848_00F4
MT(1)	16	Регистр модификатора MT (1-я секция)	0×B848_00F8
DT(1)	16	Регистр модификатора DT (1-я секция)	0×B848_00FC
		Регистры данных	
		RF0	
R0.L(0)	32	Регистр данных R0.L (0-я секция)	0×B848_0000
R2.L(0)	32	Регистр данных R2.L (0-я секция)	0×B848_0004
R4.L(0)	32	Регистр данных R4.L (0-я секция)	0×B848_0008
R6.L(0)	32	Регистр данных R6.L (0-я секция)	0×B848_000C
R8.L(0)	32	Регистр данных R8.L (0-я секция)	0×B848_0010
R10.L(0)	32	Регистр данных R10.L (0-я секция)	0×B848_0014
R12.L(0)	32	Регистр данных R12.L (0-я секция)	0×B848_0018
R14.L(0)	32	Регистр данных R14.L (0-я секция)	0×B848_001C
R16.L(0)	32	Регистр данных R16.L (0-я секция)	0×B848_0020
R18.L(0)	32	Регистр данных R18.L (0-я секция)	0×B848_0024
R20.L(0)	32	Регистр данных R20.L (0-я секция)	0×B848_0028
R22.L(0)	32	Регистр данных R22.L (0-я секция)	0×B848_002C
R24.L(0)	32	Регистр данных R24.L (0-я секция)	0×B848_0030
R26.L(0)	32	Регистр данных R26.L (0-я секция)	0×B848_0034
R28.L(0)	32	Регистр данных R28.L (0-я секция)	0×B848_0038
R30.L(0)	32	Регистр данных R30.L (0-я секция)	0×B848_003C
		RF1	
R0.L(1)	32	Регистр данных R0.L (1-я секция)	0×B848_0040
R2.L(1)	32	Регистр данных R2.L (1-я секция)	0×B848_0044
R4.L(1)	32	Регистр данных R4.L (1-я секция)	0×B848_0048
R6.L(1)	32	Регистр данных R6.L (1-я секция)	0×B848_004C
R8.L(1)	32	Регистр данных R8.L (1-я секция)	0×B848_0050
R10.L(1)	32	Регистр данных R10.L (1-я секция)	0×B848_0054
R12.L(1)	32	Регистр данных R12.L (1-я секция)	0×B848_0058
R14.L(1)	32	Регистр данных R14.L (1-я секция)	0×B848_005C
R16.L(1)	32	Регистр данных R16.L (1-я секция)	0×B848_0060
R18.L(1)	32	Регистр данных R18.L (1-я секция)	0×B848_0064
R20.L(1)	32	Регистр данных R20.L (1-я секция)	0×B848_0068
R22.L(1)	32	Регистр данных R22.L (1-я секция)	0×B848_006C
R24.L(1)	32	Регистр данных R24.L (1-я секция)	0×B848_0070
R26.L(1)	32	Регистр данных R26.L (1-я секция)	0×B848_0074
R28.L(1)	32	Регистр данных R28.L (1-я секция)	0×B848_0078
R30.L(1)	32	Регистр данных R30.L (1-я секция)	0×B848_007C

		Секционные регистры состояния	
CCR (0)	16	Регистр кодов условий (0-я секция)	0×B848_0160
PDNR (0)	16	Регистр параметра денормализации (0-я секция)	0×B848_0164
AC0 (0)	32	Регистр-аккумулятор 0 (0-я секция)	0×B848_0168
AC1 (0)	32	Регистр-аккумулятор 1 (0-я секция)	0×B848_016C
CCR (1)	16	Регистр кодов условий (1-я секция)	0×B848_0170
PDNR (1)	16	Регистр параметра денормализации (1-я секция)	0×B848_0174
AC0 (1)	32	Регистр-аккумулятор 0 (1-я секция)	0×B848_0178
AC1 (1)	32	Регистр-аккумулятор 1 (1-я секция)	0×B848_017C

Пример:

```
li      $2,0xffffffff
sw      $2,0xb8480000
```

В данном примере в регистр `v0` загружается число `0xFFFFFFFF`, а затем это число сохраняется в регистре **R0** ядра DSP (адрес `0xB8480000`).

Примечание 1: операция **`li`** (*Load Immediate*) не входит в систему команд ядра RISC, а является макросом, применяемым для загрузки непосредственных операндов в регистры общего назначения. При загрузке непосредственного операнда размером более 16 бит, операция будет разбита на команды **`lui`** и **`ori`**, заполняющие регистр общего назначения непосредственным операндом в два такта.

Примечание 2: в операции **`sw`** непосредственно указан виртуальный адрес регистра **R0**. Тем не менее, при компиляции эта команда заменяется двумя - загрузкой базового адреса (`0xb8400000`) в регистр общего назначения `at` и загрузкой необходимых данных по адресу, указанному в виде `0(at)`, то есть по смещению относительно базового адреса.

Примечание 3: в DSP-ядре *Elcore-14* не существует секции 1. Эта секция существует только в модификации *Elcore-24*.

Примечание 4: Все перечисленные регистры доступны как по записи, так и по чтению. Исключением является младший байт регистра **SR**, а также регистр **IDR** - они доступны только по чтению.

Примечание 5: Обращение к содержимому любого из регистров влечет приостановку программного конвейера за исключением регистров **DCSR**, **SR**, **IDR**, **SAR**, **CNTR**.

Файлы memory_12_asm.h/memory_24_asm.h

При написании программы RISC на языке *Ассемблера* для обращения к регистрам удобнее, вместо прямого задания их виртуальных адресов, использовать имена регистров. Для этого существует заголовочный файл *memory_12_asm.h/memory_24_asm.h*. В этих файлах определяются символы, соответствующие адресам регистров RISC и DSP. Символы определяются как смещения от базовых адресов. Для обращения к регистрам DSP используется базовый адрес, соответствующий символу `DSP_BASE`. Таким образом, для загрузки в регистр **R0** содержимого регистра **v0** можно записать:

```
lui   $30,DSP_BASE
sw    $2,R0($30)
```

Особенно удобно использовать этот способ в том случае, когда необходимо последовательно проинициализировать несколько регистров DSP-ядра (или считать данные из них). Тогда в один из регистров общего назначения (например, `$30`) нужно поместить базовый адрес `DSP_BASE`, а затем в инструкциях **`sw`** (или иных инструкциях загрузки/записи) указывать имя регистра в качестве смещения.

Пример:

```
lui   $30,DSP_BASE
sw    $2,R0($30)
sw    $2,R2($30)
sw    $2,R4($30)
sw    $2,R6($30)
sw    $2,R8($30)
```

Здесь содержимое регистра **v0** последовательно загружается в регистры **R0**-**R8** ядра DSP. Как видно из примера, базовый адрес загружается лишь один раз. Если же заменить конструкцию вида `смещение(базовый адрес)` на явно указанный виртуальный адрес, каждая из инструкций **`sw`** будет заменена на двумя командами - загрузкой базового адреса в регистр **at** и загрузкой слова по адресу вида `смещение(базовый адрес)`. То есть, пользуясь файлами *memory_12_asm.h/memory_24_asm.h* (или загружая базовые адреса самостоятельно), можно сократить операцию записи/чтения регистра DSP до одного такта.

Примечание: не допускается использовать в качестве базового адреса непосредственный операнд.

Файлы memory_12.h/memory_24.h

Если программа RISC написана на языке *C*, обращения к регистрам RISC и DSP по именам возможны благодаря символам, определенным в файлах *memory_12.h/memory_24.h*. Например, для записи `0xFFFFFFFF` в регистр **R0** ядра DSP можно использовать операцию `R0=0xFFFFFFFF;`, то есть содержимому ячейки памяти, определенной символом `R0`, присваивается значение `0xFFFFFFFF`.

Все приведенные здесь инструкции описаны в книге "**RISC Instructions Set**".

2.11.3. Обмен данными с памятью DSP

Обмен данными между ядрами RISC и DSP осуществляется в одностороннем порядке. То есть ядро RISC может осуществлять запись/чтение данных всей памяти процессора MultiCore, включая внутреннюю память DSP, в то время как ядру DSP доступна только его внутренняя память.

Таким образом, программа RISC может осуществлять запись данных в **PRAM**, **XRAM** и **YRAM** ядра DSP, а также в доступные по записи регистры DSP. Данными могут быть значения, необходимые для работы программы DSP, или блоки кода программы DSP.

При обменах следует учитывать, что вся память MultiCore для ядра RISC представляется как байтовая.

Для обмена данными с памятью ядра DSP в программе RISC возможно использовать:

- **прямую адресацию** - адрес нужной области памяти ядра DSP указывается программистом явно.
- **глобальные символы** - вместо адреса используется символ, указывающий на область памяти. Символ должен быть объявлен в программе DSP как глобальный, а в программе RISC - как внешний.

При объявлении внешних символов в программе RISC необходимо соблюдать соответствие типов данных RISC и DSP. То есть, если в секции DSP данные заданы, например, директивой **`.word`**, то в программе RISC внешний символ должен быть типа **`int`**. В таблице 1.23 приведены различные способы задания данных в DSP и соответствующие им типы данных *ANSI C*, которые и следует использовать в программе RISC.

Таблица 1.23. Типы данных в DSP.

Стандарт ANSI C	Директива ассемблера DSP	Размер в байтах	Пример описания С.	Пример описания ассемблер DSP
shot, unsigned short	.dw	2 байта	shot var=245;	.dw 245
int, unsigned int	.dl	4 байта	int var=245;	.dl 245
float	.real	4 байта	float var=2.5;	.real 2.5
int, unsigned int	.dl	4 байта	int var=0x80018000;	.dl [-32767,0x8000]
int, unsigned int	.dl	4 байта	int var=0x01001108;	.dl [[@1,0],[@17,0x8]]
short	.fr	2 байта	shot var=0xe000;	.fr -0.25
int	.frl	4 байта	int var=7ffffffe;	.frl 0.99999999

Прямая адресация

Данный метод предполагает явное указание программистом адреса требуемой области памяти (например, `0xB8440000`). Адрес представляет собой 32-разрядное число. Следует учитывать, что для ядра RISC все адресное пространство адресуется с точностью до байта. То есть, чтобы обратиться, например, к ячейке **XRAM** с адресом внутри DSP-ядра равным `0x100`, необходимо использовать адрес `0xB8400400`. Здесь `0x00000400=0x00000100<<2`, а `0xB8400000` - виртуальный адрес начала **XRAM**.

Таким образом, необходимый адрес в области памяти DSP-ядра для программы RISC состоит из адреса начала этой области памяти и адреса требуемой ячейки внутри области. При этом:

- Адрес начала программной памяти **PRAM** равен `0xB8440000`;
- Адрес начала памяти данных **XRAM** равен `0xB8400000`;
- Адрес начала памяти данных **YRAM** равен адресу последней ячейки **XRAM**, увеличенному на единицу.

Подробнее об адресации внутренней памяти DSP-ядра см. страницу "Внутренняя память DSP-ядра".

Глобальные символы

Данный метод использует символические имена для адресации ячеек памяти. Чтобы сопоставить ячейке памяти DSP-ядра символ и использовать его в программе RISC, необходимо:

1. В файле, содержащем программу DSP, определить этот символ в соответствии с правилами задания символических имен в программе DSP. Символ будет сопоставлен со следующей за ним строкой кода, содержащей инструкцию или данные.
2. В этом же файле указать, что заданный символ является глобальным. Задание глобального символа осуществляется посредством ввода директивы **.global** с именем символа, отделенным от директивы пробелом.
3. В файле с программой RISC объявить символ внешним. Это осуществляется при помощи конструкции **extern** тип_символа имя_символа для программы С или при помощи директивы **.extern** имя_символа для программы на языке Ассемблера для RISC.

Пример 1:
Программа DSP:

```
 .text
 .global Start_DSP
     ...
Start_DSP: ABS R0,R0
     ...
```

Программа RISC:

```
#include "memory_12.h"
     ...
extern int Start_DSP;
     ...
main()
{
     ...
  PC=((unsigned int)&Start_DSP - (unsigned int)&PRAM)>>2;
     ...
}
```

В приведенном примере задается глобальный символ Start_DSP (для наглядности он выделен красным шрифтом). В процессе компоновки, символу будет придано значение ячейки программной памяти **PRAM**, в которой находится инструкция **ABS** R0,R0. В программе RISC символ объявлен как внешний и используется для помещения адреса инструкции **ABS** R0,R0 в программный счетчик **PC**. То есть при старте DSP-ядра, программа будет исполняться именно с этой инструкции.

Примечание: В **PC** помещается разность адресов символов Start_DSP и PRAM. Вычисление разности необходимо для преобразования адреса RISC символа Start_DSP в адрес **PRAM**. Символ PRAM - первая ячейка программной памяти. Он определен в файлах заголовков *memory_12.h/memory_24.h*.

Примечание 2: при вычислении адреса, помещаемого в регистр **PC**, адрес делится на 4 (сдвигается на 2 бита вправо). Это необходимо для перехода от байтовой адресации в RISC к словной адресации в DSP.

Пример 2:
Программа DSP:

```
    ...
 .global My_Symbol
    ...
 .data
    ...
My_Symbol: .word 0
    ...
 .end
```

Программа RISC:

```
    ...
extern int My_Symbol;
    ...
main()
{
    ...
 int A;
    ...
 My_Symbol=18;
 A=My_Symbol;
    ...
}
```

В данном примере задается глобальный символ `My_Symbol`, сопоставленный с ячейкой памяти данных DSP-ядра. Так как данные в ячейке памяти, соответствующей символу, заданы директивой **`.word`**, их размерность равна 32-м битам, следовательно, тип внешнего символа должен быть **`int`**. В программе RISC задается переменная `A` типа **`int`**. Затем, в ячейку, сопоставленную с символом `My_Symbol`, записывается число `18`, после чего значение ячейки считывается в переменную `A`. Это пример обмена данными между ядром RISC и памятью данных ядра DSP.

Обмен данными с регистрами DSP описан на странице "Доступ к регистрам DSP-ядра".

2.11.4. Запуск программы DSP

Так как ядро DSP в процессоре MultiCore является ведомым, для запуска программы DSP-ядра на исполнение необходимо отдавать команду старта извне. А именно - из программы RISC-ядра, являющегося ведущим.

Запуск программы DSP осуществляется посредством записи **1** в бит **RUN** регистра **DCSR** (**`DCSR`**`[14]`). После этого программа DSP будет исполняться до останова. Первой исполняемой инструкцией будет инструкция с адресом, загруженным в регистр программного счетчика **PC** ядра DSP.

Также DSP-ядро может перейти в состояние **RUN** по сигналам от каналов **DMA MemCh**.

Пример 1: `DCSR |= 0x4000;` - инструкция записывает **1** в 14й бит **DCSR** по *ИЛИ*. То есть, значения остальных битов регистра не изменяются (так как `0x4000 = 0100000000000000`).

Пример 2:

```
lui   $30,DSP_BASE
lhu   $3,DCSR($30)
ori   $3,0x4000
sh    $3,DCSR($30)
```

В данном примере сначала в старшие 16 бит регистра `$30` помещается базовый адрес `DSP_BASE`. Это адрес начала регистровой памяти DSP, определенный в файлах *memory_12_asm.h/memory_24_asm.h*. Для адресов регистров DSP-ядра *memory_12_asm.h/memory_24_asm.h* определяют символы как смещения от базового адреса `DSP_BASE`. Затем в регистр `$3` загружается текущее значение регистра **DCSR** (16 бит). В регистр `$3` помещается значение `(0x4000 or DCSR)`. Таким образом, бит **RUN** устанавливается в **DCSR** равным единице по *ИЛИ*, то есть остальные биты регистра не изменяют значений. Последний оператор сохраняет результат в регистр **DCSR**. Программа DSP-ядра запущена на исполнение.

Примечание: так как регистр **DCSR** - 16-разрядный, то для загрузки/записи его значений используются команды **lhu** и **sh** - *Load Halfword Unsigned* и *Store Halfword*.

Подробное описание использованных здесь команд приведено в книге "**RISC Instructions Set**".

2.11.5. Останов программы DSP

После запуска программа DSP исполняется до останова. Останов программы DSP происходит в следующих случаях:

- В бит **RUN** регистра **DCSR** (**DCSR**`[14]`) принудительно помещается нуль. Данный способ применим, если необходимо остановить программу DSP из ядра RISC;
- В ядре DSP исполняется инструкция **STOP**;
- Программный счетчик **PC** достигает адреса останова (установленного в регистре **SAR**);
- Ядро DSP выполняет заданное число инструкций. Число инструкций задается в регистре **CNTR**. Если **CNTR**`=0`, число исполняемых инструкций не ограничено.

После останова, ядро DSP переходит в состояние **STOP**. Запуск DSP-ядра описан на странице "Запуск программы DSP".

Пример 1: `DCSR &= 0xBFFF;` - инструкция записывает **0** в 14й бит **DCSR** по *И*. То есть, значения остальных битов не изменяются (так как `0xBFFF = 1011111111111111`).

Пример 2:

```
lui   $30,DSP_BASE
lhu   $3,DCSR($30)
andi  $3,0xBFFF
sh    $3,DCSR($30)
```

Первые две инструкции загружают текущее значение регистра **DCSR** в регистр `$3`. Затем, в регистр `$3` помещается результат операции `(0xBFFF and $3)`. Таким образом, бит **RUN** устанавливается в **DCSR** равным нулю по *И*, то есть остальные биты регистра не изменяют значений. Последний оператор сохраняет результат в регистр **DCSR**. Программа DSP-ядра остановлена.

Примечание: так как регистр **DCSR** - 16-разрядный, то для загрузки/записи его значений используются команды **lhu** и **sh** - *Load Halfword Unsigned* и *Store Halfword*.

Способы обращения к регистрам DSP-ядра описаны на странице "Доступ к регистрам DSP-ядра".

Подробное описание использованных здесь команд приведено в книге "**RISC Instructions Set**".

2.11.6. Ожидание останова программы DSP

В том случае, если ядру RISC для дальнейшей работы необходимо дождаться результатов исполнения программы DSP, программа RISC должна ожидать перехода ядра DSP в состояние **STOP**.

При останове DSP-ядра в регистре запроса прерывания **QSTR** бит **SBS** (**QSTR**[31]) установится в **1**. Следовательно, ожидание останова DSP-ядра происходит до тех пор, пока **SBS** не равен 1.

Пример 1: **while** ((~(QSTR))&(1<<31)); - инструкция исполняет пустой цикл до тех пор, пока 31й бит регистра QSTR не станет равным 1.

Пример 2:

```
        lui   $30,CPU_BASE
Wait_for_DSPStop:
        lw    $2,QSTR($30)
        li    $3,1<<31
        and   $2,$3
        bne   $3,$2,Wait_for_DSPStop
        nop
```

В данном примере сначала в старшие 16 бит регистра `$30` помещается базовый адрес `CPU_BASE`. Это адрес начала регистровой памяти RISC, определенный в файлах *memory_12_asm.h/memory_24_asm.h*. Для адресов регистров RISC-ядра *memory_12_asm.h/memory_24_asm.h* определяют символы как смещения от базового адреса `CPU_BASE`. Затем в регистр `$2` загружается текущее значение регистра **QSTR** (16 бит). В регистр `$3` помещается значение `0x80000000` (`1<<31`). После этого в регистр `$2` помещается результат операции (`QSTR` **and** `0x80000000`), а затем происходит сравнение этого результата с `0x80000000`. Если значения равны, значит в 31м бите **QSTR** установлена единица и DSP-ядро остановлено, иначе происходит переход к символу `Wait_for_DSPStop`, то есть проверка осуществляется заново.

При написании программ, задействующих оба ядра ИМС "*МУЛЬТИКОР*", приведенный выше способ ожидания не всегда является эффективным. Кроме циклической проверки регистра **QSTR** можно использовать обработчик исключения по прерыванию. Кроме того, ядро DSP способно формировать прерывание в RISC без остановки исполнения собственной программы. Прерывания, формируемые ядром DSP в RISC, описаны на странице "Прерывания от DSP-ядра".

Описание регистров ядра RISC приведено на странице "Регистры RISC".

Подробное описание использованных здесь команд приведено в книге "**RISC Instructions Set**".

2.12. Кэш

2.12.1. Организация кэш

Кэш команд предназначается для увеличения быстродействия системы, так как представляет собой память, отвечающую на внутренние запросы чтения быстрее, чем при выполнении цикла чтения оперативной памяти по шине.

В ИМС "*МУЛЬТИКОР*" реализован виртуально индексируемый и контролируемый по физическому тэгу **кэш команд** типа *direct mapped*. Это позволяет осуществлять доступ к кэш параллельно с преобразованием виртуального адреса в физический. Объем кэш составляет 16 Кбайт.

Пополнения кэш выполняются посредством четырехсловного буфера, в который поступают данные, полученные из памяти во время передачи 4-х ступенчатой пачкой (**Burst**). Критическое пропущенное слово всегда возвращается первым. До получения критического слова кэш блокируется, но во время активности на шине остальных 3-х ступеней **Burst** конвейер может продвигаться дальше.

Кэш команд состоит из двух массивов – массива тэгов и массива данных. Кэш индексируется виртуально, поскольку для выбора соответствующей строки в обоих массивах используется виртуальный адрес. Контроль осуществляется по физическому тэгу, так как массив тэгов содержит физический, а не виртуальный адрес.

На рисунке 1.1 представлен формат каждой строки массивов тэгов и данных. Тэговая строка содержит 22 старших бита физического адреса (биты `[31:10]`) и бит валидности.

Строка данных содержит четыре 32-х разрядных слова – всего 16 байт.

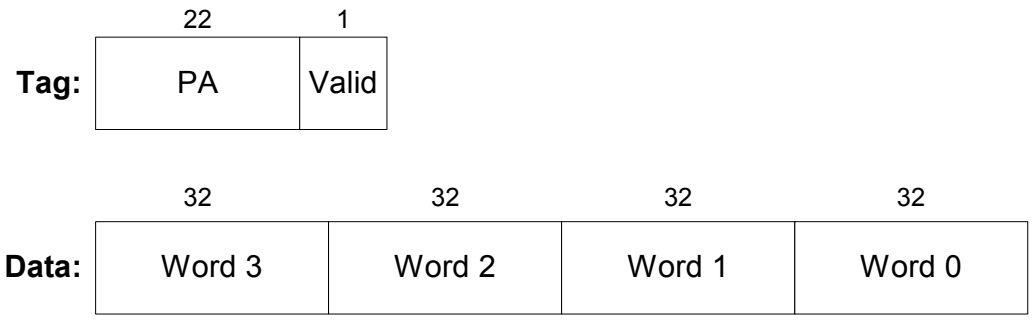


Рисунок 1.1. Формат строки массивов тэгов и данных.

Область памяти может быть кэшируемой, либо некэшируемой. Кэшированием управляет регистр Config (**CP0**).

Описание способов кэширования команд приводится на странице "Кэширование".

2.12.2. Кэширование

Для кэширования команд программы RISC необходимо осуществить следующие действия:

- В регистре Config (**CP0**) установить поле K0 в значение 0 - кэшируемая, некогерентная область;
- Из виртуального адреса начала кэшируемой области вычесть значение 0x20000000;
- Выполнить переход по полученному адресу (командой **jr**).

После осуществления указанных действий, команды программы будут кэшироваться до тех пор, пока не будет осуществлен возврат в область адресов 0xBFC*****.

Структура кэш в ИМС "*МУЛЬТИКОР*" описана на странице "Организация кэш".
Пример кэширования команд приведен на странице "Пример программы с использованием кэширования".

2.12.3. Пример программы с использованием кэширования

Рассмотрим текст, осуществляющий кэширование команд RISC-програмы. В данном примере кэшируется область кода, исполняющая цикл с большим числом итераций. Проект примера состоит из файла *main.s*:

```
#файл main.s
 .text

Start_programm:
                        MFC0 $17,$16,0          # 1
                        ORI  $18,$0,7           # 2
                        NOT  $18,$18            # 3
                        AND  $17,$17,$18        # 4
                        MTC0 $17,$16,0          # 5 - запись 0b000 в младшие три
бита Config0

                        LA   $17, Caching_code  #загрузка адреса метки
Caching_code
                        OR   $18,$0,$0          #обнуление GPR#18
                        LUI  $18, 0x2000        #загрузка 0x2000 в старшую часть
GPR#18
                        XOR  $17, $17, $18      #GPR#17=&CAin xor 0x20000000
                        JALR $17                #переход по GPR#17 (в начало
кэшируемой области)
                        NOP                     #delay slot

                        #бесконечный цикл (сюда программа возвращается из кэшируемой
области)
            Infinite_cycle:
                              NOP
                              J   Infinite_cycle
                              NOP

Caching_code:                                   #начало кэшируемой области
```

```
                    OR    $3,$0,$0              #обнуление GPR#3
                    LI    $4,100000             #число итераций -> GPR#4
          Cycle:    ADDI  $3,$3,1               #GPR#3++
                    NOP                         #
                    NOP                         #тело цикла
                    NOP                         #
                    BNE   $3,$4,Cycle           #если счетчик не достиг
максимального значения, переход к Cycle
                    NOP                         #delay slot

                    JR    $31                   #возврат из кэшируемой области
                    NOP                         #delay slot
                                                #конец кэшируемой области
```

Приведенная программа выполняет следующие действия:

- В первых пяти строках обнуляется поле K0 (`Config[2:0]`, **CP0**). Нулевое значение этого поля соответствует режиму "кэшируемая, некогерентная область";
- Затем, адрес кэшируемой области преобразуется к виду `0x9FC00044`;
- После этого происходит переход по вновь полученному адресу с сохранением адреса возврата;
- Далее исполняется кэшируемый цикл в 100000 итераций. По окончании цикла содержимое регистра **GPR#3** должно равняться `0x186A0`;
- После исполнения цикла программа возвращается из кэшируемой области и исполняет бесконечный цикл.

2.13. Проекты с оверлейными структурами

2.13.1. Введение

В связи с тем, что ядро DSP процессора MultiCore обладает доступом только к своей внутренней памяти, случается, что перед программистом встает проблема нехватки памяти для размещения текста или данных программы DSP. В таком случае необходимо разбивать программу на части и динамически подгружать необходимый текст (данные) из памяти RISC по мере исполнения программы.

Структура программы с динамически подгружаемыми частями программы называется **оверлейной структурой**. Части программы, динамически подгружаемые в память DSP друг поверх друга, в дальнейшем будем называть **оверлейными секциями**.

Так как ядру DSP недоступна память RISC-ядра, загрузкой оверлейных секций должна заниматься программа RISC. В данной главе речь пойдет о принципах и способах построения программ с оверлейными структурами для процессора MultiCore.

В главе рассматриваются:

- компоновка оверлейных секций программы DSP в памяти RISC;
- загрузка оверлейной секции в программную память или память данных ядра DSP;
- использование таблицы загруженных секций OVRtable.

Пример готового проекта с оверлейными структурами рассматривается в разделе "Примеры создания проектов".

2.13.2. Компоновка оверлейных секций

Оверлейные секции при сборке проекта размещаются в памяти RISC и подгружаются в память DSP-ядра по мере необходимости в процессе исполнения программы. Настройки размещения всех секций проекта **MultiCore Studio** осуществляются в диалоге настроек проекта. Для каждой секции текста или данных проекта в этом диалоге необходимо ввести адрес размещения секции внутри модуля (**Unit VMA**) и адрес фактического размещения секции в памяти RISC (**VMA**).

Рассмотрим настройки размещения оверлейных секций:

- В **Unit VMA** для оверлейной секции необходимо указывать внутренний адрес памяти DSP, по которому эта секция будет в дальнейшем загружена. При загрузке оверлейной секции в память DSP из программы RISC следует указывать именно этот адрес. Если же секция будет загружена по иному адресу, ни одно из символических имен не будет доступно. Например, если оверлейная секция должна загружаться в PRAM с нулевого адреса, то **`Unit VMA`**`=0`, а при загрузке следует указывать адрес `0xB8440000`. Если же оверлейная секция должна загружаться в память данных **XRAM**, например, по адресу `0x40`, то **`Unit VMA`**`=0x40`, а при загрузке из программы RISC следует указывать адрес `0xB8400100`. Это связано с тем, что вся память RISC адресуется с точностью до байта, а память DSP - словная. Подробнее о приведении адреса памяти DSP к виду адреса RISC см. страницу "Обмен данными с памятью DSP".
- Адрес размещения оверлейной секции в памяти RISC (**VMA**) устанавливается в соответствии с общими настройками проекта. При этом, если указать для всех оверлейных секций **`VMA`**`=0`, они скомпонуются в памяти RISC последовательно после секций RISC-программы.

2.13.3. Загрузка оверлейной секции

Загрузка оверлейной секции осуществляется программой RISC. При этом секция копируется из памяти RISC-ядра в программную память или память данных ядра DSP. Загрузка должна производиться *строго* по адресу, указанному в диалоге настроек проекта в поле **Unit VMA**, представленному в виде адреса RISC. Подробнее об этом см. страницу "Компоновка оверлейных секций".

Для корректной работы отладчика при работе с оверлейными секциями необходимо пользоваться таблицей загруженных секций. В противном случае в процессе отладки будет выводиться неверная информация об исполняемых в данных момент инструкциях программы DSP.

Пример:
Программа RISC:

```
...
extern int Start_Overlay_Section;
extern int End_Overlay_Section;
...
```

```
main()
{
  ...
 unsigned* pSRC;
 unsigned* pDST;
 int wSIZE=(&End_Overlay_Section-&Start_Overlay_Section+1);

 pSRC=&Start_Overlay_Section;
 pDST=(unsigned *)0xb8400000;
 while (wSIZE--) *pDST++=*pSRC++;
  ...
}
```

Программа DSP:

```
 .text
  ...
  .global Start_Overlay_Section
  .global End_Overlay_Section
  ...
 .data
Start_Overlay_Section:
           .word 0x0ABC1234
           .real -0.64596409
End_Overlay_Section:
           .word 5
  ...
 .end
```

В данном примере иллюстрируется загрузка в **XRAM** ядра DSP секции данных, скомпонованной в памяти RISC. Копирование производится с адреса метки `Start_Overlay_Section` по адресу `0xb8400000` (виртуальный адрес начала **XRAM**). Размер копируемой секции вычисляется как разность адресов меток `End_Overlay_Section` и `Start_Overlay_Section`, увеличенная на единицу. Таким образом, в **XRAM** попадает три слова (числа `0x0abc1234`, `-0.64596409` и `5`).

Способы задания символических имен (меток) в программе DSP описаны на странице "Символы DSP".

2.13.4. Таблица загруженных секций

Таблица загруженных секций **OVRtable** размещается в памяти RISC и содержит информацию обо всех загруженных в данный момент в память DSP секциях. Каждая строка таблицы включает в себя:

- адрес размещения секции в памяти RISC;
- адрес размещения секции в памяти DSP;
- размер загруженной секции.

  Размер таблицы равен 32 строкам.

Таблица загруженных секций необходима отладчику для отладки оверлейных проектов. Поэтому, если в Вашем проекте есть динамически загружаемые секции, для

корректного отображения исполняющихся в данный момент блоков кода программы следует подключить и использовать таблицу **OVRtable**. Для этого необходимо:

1. Подключить к проекту файл заголовка *OVR.h* в тексте программы RISC директивой `#include`;
2. Включить в RISC-модуль проекта файл *OVR.c*;
3. Для загрузки секций использовать функцию `OVRLoad`, определенную в этом файле, либо заполнять таблицу **OVRtable** самостоятельно (в соответствии с приведенным здесь форматом таблицы).

Файлы *OVR.h* и *OVR.c* находятся в директории проекта *Overlay*: *\MCStudio\Samples\Overlay*.

Таблица будет сформирована автоматически, загрузка секций и заполнение строк таблицы осуществляется функцией `OVRLoad`. Рассмотрим файл *OVR.h*:

```
#ifndef OVR
#define OVR

#define max_Num 32

typedef struct TOVRtable{
     unsigned SRC;
     unsigned DST;
     unsigned wSIZE;
}TOVRtable;

int OVRLoad(unsigned SRC, unsigned DST , unsigned wSIZE);

extern TOVRtable OVRtable[];

#endif
```

Функция **int** `OVRLoad (`**unsigned** `SRC,` **unsigned** `DST,` **unsigned** `wSIZE)` копирует секцию размера `wSIZE` (размер указан в словах), начиная с адреса `SRC` по адресу `DST`, и заносит информацию о загруженной секции в таблицу **OVRtable**. Функция возвращает номер строки **OVRtable**, содержащей информацию о загруженной секции, или **-1**, если секцию не удалось загрузить. Если копируемая оверлейная секция перекрывает в памяти DSP другие секции, информация о них будет удалена из таблицы.

Рассмотрим файл *OVR.c*:

```
#include "OVR.h"

TOVRtable OVRtable[max_Num];

int OVRLoad(unsigned SRC, unsigned DST , unsigned wSIZE){
 int num, new_num = -1, rep_num = -1;//new-ближайший свободный, rep-замененный
 unsigned* pSRC = (unsigned *)SRC;
 unsigned* pDST = (unsigned *)DST;

 if(wSIZE)
 {
  //Поиск места в таблице секций и стирание секций наложения
  for(num = 0; num < max_Num; num ++){
    if(!OVRtable[num].SRC){
```

```
        if(new_num < 0) new_num = num;
    }else{
        if(
           ((DST >= OVRtable[num].DST) && (DST < (OVRtable[num].DST +
           (OVRtable[num].wSIZE<<2))))
           ||
           (((DST + wSIZE) >= OVRtable[num].DST) && ((DST + (wSIZE<<2)) <
           (OVRtable[num].DST + (OVRtable[num].wSIZE<<2))))
           ||
           ((DST < OVRtable[num].DST) && ((DST + wSIZE)>(OVRtable[num].DST +
           (OVRtable[num].wSIZE<<2))))
         )
        {
          //Замена существующей секции
                OVRtable[num].SRC = 0;
                if(rep_num < 0) rep_num = num;
        }
       }
   }
    if(rep_num >= 0) new_num = rep_num;//Номер секции в таблице
    //Заносим данные в таблицу
    OVRtable[new_num].SRC = SRC;
    OVRtable[new_num].DST = DST;
    OVRtable[new_num].wSIZE = wSIZE;
    //Копируем блок во внутреннюю память
    while(wSIZE --) *pDST++ = *pSRC++;
 }
 return new_num;
}
```

Функция `OVRLoad` подходит для заполнения таблицы загруженных секций в том случае, если программа RISC написана на языке *C*. В случае, если программа написана на языке Ассемблера, таблицу `OVRtable` следует заполнять самостоятельно. Для этого необходимо:

1. Определить символ **OVRtable** в программе RISC и выделить под него 96 слов памяти. Например:
   ```
   OVRtable:
           .space 96*4,0
   ```
   Директива **`.space`** `96*4,0` заполняет нулями `96*4` байт памяти.
2. Заполнять память, выделенную под таблицу **OVRtable** вручную в соответствии с загружаемыми секциями. При этом поля таблицы заполняются в порядке, определенном при описании таблицы в файле *OVR.h*.

3. Программирование под DSP

3.1. Введение

Ядро DSP процессора MultiCore предназначено для цифровой обработки сигналов. Ядро является ведомым (*slave*), поэтому программе DSP для запуска необходима команда программы RISC. Ядру DSP доступна только внутренняя память, поэтому при нехватке этой памяти необходимо подгружать в память DSP требуемые блоки кода или данных из программы RISC.

В данной главе рассматриваются принципы написания программы для ядра DSP. Глава включает в себя:

- Карту внутренней памяти DSP-ядра;
- Описание регистров DSP-ядра;
- Описание программы DSP, написанной на языке Ассемблера;
- Краткое описание форматов инструкций DSP-ядра;
- Методы запуска и останова программы;
- Способы задания символических имен в программе DSP;
- Способы записи выражений (непосредственных операндов);
- Размещение в памяти секций текста и данных DSP;
- Способы адресации в программе DSP;
- Применение условно исполняемых инструкций;
- Способы задания макроопределений;
- Программные переходы и ветвления;
- Подпрограммы;
- Организация циклов;
- Параллельно исполняемые инструкции;
- Краткий обзор системы команд DSP-ядра.

3.2. Внутренняя память DSP

Память DSP–ядра *ELcore_x4* включает в себя два независимых пространства - память данных и память программ.

32-разрядная память данных состоит из двух областей – **X**-памяти и **Y**-памяти. Чтение обеих областей памяти данных может происходить одновременно при помощи адресных генераторов - соответственно **AGU** и **AGU-Y**. Запись производится при помощи генератора **AGU** только в одну из областей. Для **SIMD**-модификации DSP–ядра *ELcore_24* весь объем памяти данных **XRAM**, **YRAM** распределяется поровну между секционными модулями памяти.

Организация программной памяти

Память программ **PRAM** имеет 64-разрядную организацию, позволяющую осуществлять хранение и выборку в течение одного такта как 32-разрядных, так и 64-разрядных инструкций. Обе модификации DSP-ядра - *Elcore_14* и *Elcore_24* – имеют одинаковую память **PRAM объемом 4К 32-разрядных (или 2К 64-разрядных) слов**.

Память **PRAM** адресуется программным адресным генератором, входящим в состав устройства программного управления.

При последовательном ходе программы адрес программной памяти определяется состоянием программного счетчика **PC**, при программных переходах адрес определяется инструкцией перехода.

Организация памяти данных

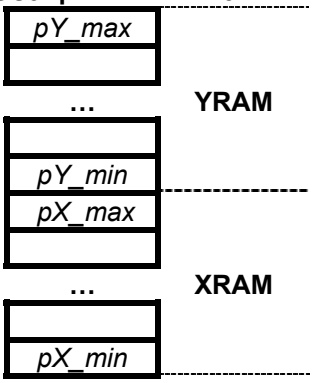
Общее пространство памяти данных DSP-ядра состоит из двух областей: **X**- и **Y**-памяти (**XRAM**, **YRAM**), имеющих 32-разрядную организацию.

Память **XRAM** и память **YRAM** имеют следующий объем:

- **ELcore_14 XRAM – 24К 32-разрядных слов;**
  YRAM - 12К 32-разрядных слов;
- **ELcore_24 XRAM – 32К 32-разрядных слов;**
  YRAM - 8К 32-разрядных слов;

При этом адресация памяти **XRAM** начинается с нулевого адреса, а памяти **YRAM** - с адреса, следующего за последним адресом **XRAM** в соответствии с таблицей 2.1, где *pX_min, pX_max* – соответственно минимальный и максимальный адрес **X**-памяти *pY_min, pY_max* – соответственно минимальный и максимальный адрес **Y**-памяти.

Таблица 2.1. Адресация X и Y-памяти.



В таблице 2.2 приводятся граничные адреса **X**- и **Y**-памяти для *ELcore_14* и *ELcore_24* (адреса даются в шестнадцатеричной системе счисления).

Таблица 2.2. Граничные адреса X и Y-памяти.

	pX_min	pX_max	pY_min	pY_max
ELcore_14	0x0000	0x5FFF	0x6000	0x8FFF
ELcore_24	0x0000	0x7FFF	0x8000	0x9FFF

Генерация адресов для **X**- и **Y**-памяти данных при внутренних обменах DSP-ядра осуществляется 16-разрядными адресными генераторами - **AGU** и **AGU-Y**.

*Адресный генератор **AGU** является общим для всего DSP-ядра* и производит адресацию всех сегментов **X**- и **Y**-памяти данных DSP-ядра.

В каждой секции DSP-ядра имеется отдельное устройство для генерации адресов **Y**-памяти - ***AGU-Y0*** и ***AGU-Y1***. Устройство **AGU-Y** адресует только **Y**-память и только по чтению.

Особенности адресации памяти данных в режимах SCALAR и ***SIMD.***

Адреса, вырабатываемые генераторами **AGU**, **AGU-Y0** и **AGU-Y1**, будем обозначать соответственно **XAB**, **YAB0** и **YAB1** (так же, как и соответствующие им адресные шины).

В режиме SCALAR указатели памяти, то есть адреса ячеек **X**- и **Y**-памяти, к которым происходят обращения, совпадают с вырабатываемыми адресами:

```
pX=XAB,        pX_min J  XAB J   pY_max;
pY=YAB0,       pY_min J  YAB0 J   pY_max;
```

Примечание. Одновременное обращение к **Y**-памяти со стороны обоих генераторов, **AGU** и **AGU-Y**, запрещено. При таком одновременном обращении к **Y**-памяти приоритет имеет генератор **AGU**. Данные, считанные в этом случае генератором **AGU-Y**, будут неправильными.

В режиме SIMD для DSP–ядра *ELcore_24* весь объем памяти данных **XRAM**, **YRAM** распределяется поровну между секциями. При этом все ячейки с четными адресами принадлежат к одной секции, все ячейки с нечетными адресами - к другой.

В режиме **SIMD** указатели памяти для каждой из секций определяются формулами:

```
pX0=2*XAB+(SW) ,  pX_min J  XAB J   pY_max/2;
pX1=2*XAB+(!SW),  pX_min J  XAB J   pY_max/2;
pY0=2*YAB0,     pY_min/2 J  YAB0 J   pY_max/2;
pY1=2*YAB1+1,     pY_min/2 J  YAB1 J   pY_max/2;
```

Управляющий бит SW (8-й разряд регистра **SR**) позволяет производить перекрестный обмен между секциями.

Примечание. При `pX_min J XAB J pX_max/2` со стороны генератора **AGU** происходит обращение к **X**-памяти, при `pY_min/2 J XAB J pY_max/2` - к **Y**-памяти.

3.3. Регистры DSP

В данном разделе рассматривается описание регистров DSP-ядра и доступа к ним.

По своему назначению все регистры делятся на регистры данных, объединенные в регистровые файлы, и регистры управления (все остальные).

Кроме **RF** и 32-разрядных регистров-аккумуляторов **AC0** и **AC1**, все регистры – 16-разрядные. Все регистры доступны как по чтению, так и по записи, за исключением регистра **IDR** и младшего байта регистра **SR**, доступных только по чтению.

Регистры ALU

Каждая вычислительная секция **ALU** содержит регистровый файл **RF** – реконфигурируемый массив (*16x32* или *32x16*) регистров данных, регистр параметра денормализации **PDNR**, регистр кодов условий (регистр признаков) **CCR**, два *32*-разрядных регистра-аккумулятора **AC0**, **AC1**.

Регистровый файл

Исходные данные и результаты всех операций **ALU** хранятся в регистровом файле (**RF**), который представляет собой реконфигурируемый массив регистров данных (32 регистра по 16-разрядов либо 16 регистров по 32 разряда).

16-разрядные регистры данных могут иметь номера с R0 по R31, а 32-разрядные регистры – только четные номера с R0 по R30. При 32-разрядных операциях четные и нечетные регистры объединяются попарно и образуют *16* 32-разрядных регистров, причем младшие 16 бит представлены в регистрах с четными номерами, старшие 16 бит - в регистрах с нечетными номерами. Мнемонически, для отличия четных 16-разрядных регистров от 32-разрядных, к наименованию последних добавляется через точку суффикс L (**long**), например: `R0.L`. Для команд ядра DSP регистры **RF** являются операндами.

Регистры-аккумуляторы AC0, AC1

Регистры-аккумуляторы **AC0**, **AC1** являются специализированными 32-разрядными регистрами данных, предназначенными для накопления результата в операциях умножения с накоплением (**`MAC`**, **`MAC2`**, **`MACL`**, **`MACX`**, **`SAC2`**). В операциях **`MAC`**, **`MACL`** регистры **AC0**, **AC1** объединяются в один 64-разрядный регистр для получения 64-разрядного результата.

Начальное состояние **`AC0`**=**`AC1`**=`0x00000000`.

Регистр PDNR

Регистр **PDNR** - секционный регистр управления, предназначенный для измерения параметра денормализации (*PDN*) и управления режимом блочной экспоненты и режимом масштабирования (*Scaling*). Таблица 2.3 отображает формат регистра **PDNR**.

Назначение разрядов регистра **PDNR** приведено ниже.

Таблица 2.3. Формат регистра PDNR.

15	14:10	9:8	7	6	5	4:0
Esc	--	SC	Epdn	--	F(X/L)	Cpdn

- **Cpdn** – текущий код PDN:
- **F (X/L) –** формат анализируемой информации в **PDN** (0 – 32 бит, 1 – 32 бит комплексная);
- **Epdn** – программный признак разрешения детектирования и изменения **PDN** (Epdn: 0 – нет разрешения, 1 – разрешение);
- **SC** – величина масштабирования результата в **AU**;

- **Esc** – признак разрешения масштабирования результата в **AU** (0 – нет разрешения, 1 – разрешение).

Начальное состояние регистра **PDNR**=0x0000.

Регистр CCR

Регистр **CCR** - секционный регистр, предназначенный для хранения признаков результатов вычислительных операций. Формат регистра **CCR** приведен в таблице 2.4.

Назначение разрядов регистра **CCR** приведено ниже.

Таблица 2.4. Формат регистра CCR.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Evm	Um	Nm	Zm	Vm	Cm	RND	S	t	E	Ev	U	N	Z	V	C

- **C** – признак переноса, сформированного в результате выполнения операции (0 – нет переноса, 1 – есть перенос);
- **V** – признак переполнения результата (0 – нет переполнения, 1 – есть переполнение);
- **Z** – признак нулевого результата (0 – результат не нулевой, 1 – результат нулевой);
- **N** – знак результата (0 – знак положительный, 1 – знак отрицательный);
- **U** – признак ненормализованного результата (0 – нормализованный результат, 1 – ненормализованный результат);
- **Ev** – запомненный ранее возникший признак переполнения результата (0 – не было переполнения, 1 – было переполнение);
- **E** – экспоненциальный признак (формируется командой **CMPE**);
- **t** – признак истинности условия после исполнения условной команды (t=0 – безусловная команда либо условие ложно; t=1 – условие истинно);
- **S** – бит включения режима насыщения результата (0 – отключение режима насыщения, 1 – включение режима насыщения);
- **RND** – бит управления режимом округления результата (0 – CR **(Convergent Rounding)**, 1 – TCR **(Two's-Complement Rounding)**);
- **Cm** – признак переноса сформированного в результате выполнения микрооперации в MS

(0 – нет переноса, 1 – есть перенос);

- **Vm** – признак переполнения результата в MS (0 – нет переполнения, 1 – есть переполнение);
- **Zm** – наличие нулевого результата в MS (0 – результат не нулевой, 1 – результат нулевой);
- **Nm** – значение знака результата в MS (0 – знак положительный, 1 – знак отрицательный);
- **Um** – признак ненормализованного результата в MS (0 – нормализованный результат, 1 – ненормализованный результат);
- **Evm** – запомненный ранее возникший признак переполнения результата в MS (0 – не было переполнения, 1 – было переполнение);

Начальное состояние регистра **CCR**=0x0000.

Регистры AGU

Генератор адреса для памяти **X** содержит восемь наборов по три регистра: регистр адреса **A***n*, регистр смещения **I***n*, регистр модификатора **M***n* (*n=0-7*).

Каждый генератор адреса для памяти **Y** содержит набор из четырех регистров: регистра адреса **AT**, регистров смещения **IT** и **DT**, регистра модификатора **MT**.

Регистры PCU

Устройство программного управления **PCU** включает в себя набор управляющих регистров и стеков:

- Регистр управления и состояния **DCSR**;
- Программный счетчик **PC**;
- Регистр состояния **SR**;
- Регистр-идентификатор **IDR**;
- Регистр адреса окончания цикла **LA**;
- Регистр счетчика циклов **LC**;
- Системный стек **SS**;
- Стеки циклов **CSL**, **CSH**;
- Регистр указателей стека **SP**;
- Регистр адреса останова **SAR**;
- Счетчик команд **CNTR**.

Регистр управления и состояния (**DCSR**) содержит разряды управления, определяющие состояние и режим работы DSP-ядра, а также прерывания, формируемые DSP-ядром для обработки в RISC-ядре.

Назначение разрядов регистра **DCSR** указано в таблице 2.5.

Таблица 2.5. Регистр DCSR.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RST	RUN	-	DBG	-	-	-	-	DE3	DE2	DE1	DE0	STP	BRK	SE	PI

RST – программный RESET;
RUN - состояние исполнения программы;
DBG – режим отладки.

DE3- запуск DMA (3 канал);
DE2- запуск DMA (2 канал);
DE1- запуск DMA (1 канал);
DE0- запуск DMA (0 канал);
STP – прерывание по останову STOP;
BRK –прерывание по останову BREAK;
SE – прерывание по ошибке стека SE;
PI – программное прерывание PI.

Начальное состояние **DCSR**=0x0000.

Программный счетчик PC

Регистр программного счетчика **PC** предназначен для хранения 16-разрядного адреса инструкции в программной памяти. Инкрементированное значение **PC** заносится в системный стек при инициализации нового программного цикла **DO**, **DOFOR** и при входе в подпрограмму.

Начальное состояние **PC**=0x0000.

Регистр состояния SR

Регистр состояния **SR** содержит параметры управления и состояния DSP-ядра. Разряды [7:0] регистра **SR** доступны только по чтению, остальные - по чтению и записи.

Назначение разрядов регистра **SR** указано в таблице 2.6.

Таблица 2.6. Регистр SR.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SI	SRSI		BC	YM	-	-	-	t	E	Ev	U	N	Z	V	C

SI – признак режима SIMD;
SRSI – способ формирования интегральных признаков в режиме SIMD;
BC - признак режима "BroadCasting", т.е. одновременной загрузки памяти данных всех секций DSP-ядра;
YM – режим адресации памяти YRAM;
C – перенос;
V – признак переполнения;
Z - признак нулевого результата;
N - признак отрицательного результата;
U - признак ненормализованного результата;
Ev- флаг переполнения (с сохранением);
E – экспоненциальный признак;
t – признак истинности последнего условия.

Разряды `[7:0]` регистра **SR** содержат интегральные признаки предыдущей арифметической операции.

Разряд `11` регистра **SR** (бит `YM`) предназначен для выбора режима адресации генератора **AGU-Y**.

Остальные разряды регистра **SR** предназначены для работы в режиме SIMD в многосекционных модификациях DSP-ядра.

При начальной установке все разряды регистра SR обнуляются.

Таблица 2.7. Определение CCP.

SRSI [14:13]	Алгоритм определения CCR
00	Использование CCR0 нелевой секции
01	Объединение секционнах CCR0,1,2,3 по «И»
10	Объединение секционнах CCR0,1,2,3 по «ИЛИ»
11	Резерв

Начальное состояние **SR**=`0x0000`.

Регистр-идентификатор IDR

Регистр-идентификатор **IDR** содержит код версии DSP-ядра согласно таблице 2.8. Регистр доступен только по чтению.

Таблица 2.8. Регистр IDR.

IDR[15:0]	Модификация DSP-ядра
0x0003	DSP-ядро ELcore_14
0x0013	DSP-ядро ELcore_24
Другие коды	Другие модификации DSP-ядра

Регистр адреса окончания цикла LA

Регистр адреса цикла **LA** содержит адрес последней инструкции в программном цикле **DO**, **DOFOR**. Этот адрес заносится в стек **SS** по команде **DO**, **DOFOR** и извлекается обратно по окончании вложенного цикла либо по команде **ENDDO**.

Начальное состояние **LA**=0x0000.

Регистр счетчика циклов LC

Регистр счетчика циклов содержит:

1) Текущее значение 14-разрядного счетчика программных циклов **Nc** – разряды 0-13 регистра **LC**;
2) **LF** – Флаг цикла DO – разряд 14 регистра **LC**;
3) **FV** - Флаг цикла DOFOR – разряд 15 регистра **LC**.

Формат регистра LC приведен в таблице 2.9.

Таблица 2.9. Формат регистра LC.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
FV	LF	Nc													

Значение счетчика программных циклов **Nc** определяет количество повторений программного цикла DO, в пределах от 1 до (2^{14} – 1). Этот регистр заносится в верхнюю (старшую) половину стека циклов **CSL** по команде **DO** (образуется вложенный программный цикл) и извлекается обратно по окончании вложенного цикла либо по команде **ENDDO**.

Начальное состояние **LC**=0x0000.

Стеки SS, CSL, CSH

Устройство программного управления содержит системный стек **SS** и стеки циклов **CSL**, **CSH**. Системный стек **SS** имеет объем *15* 16-разрядных слов и используется для автоматического сохранения содержимого регистра программного счетчика **PC** при входе в подпрограмму или в цикл **DO**/**DOFOR**. Стеки циклов имеют объем по 7x16 бит и предназначены для хранения соответственно длины цикла и адреса последней инструкции цикла (**LC** и **LA**). Стеки участвуют в обменах как 16-разрядные регистры управления – **SS**, **CSL** и **CSH**.

Регистр указателей стека SP

Регистр указателей стека **SP** содержит указатели на последнее записанное в стеки **SS**, **CSH** слово.

Назначение разрядов регистра **SP** указано в таблице 2.10.

Таблица 2.10. Регистр SP.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
-	-	-	UFC	CSE	CP[2:0]			-	-	UFS	SSE	SP[3:0]			
CP[2:0] – указатель стека циклов; CSE – флаг ошибки стека циклов; UFC – флаг переполнения стека циклов.								SP[3:0] – указатель системного стека; SSE – флаг ошибки системного стека; UFS – флаг переполнения системного стека.							

Младший байт регистра **SP** содержит указатель и флаги системного стека; старший байт - указатель и флаги стека циклов.

Начальное состояние **SP**=0x0000.

Регистр адреса останова SAR

Регистр адреса останова **SAR** является специализированным 16-разрядным регистром, использующимся при отладке DSP-ядра. Регистр **SAR** определяет точку останова (*Breakpoint*) - адрес инструкции, непосредственно перед исполнением которой должен произойти останов DSP-ядра. Перед исполнением инструкции с указанным адресом DSP-ядро переходит в состояние останова (**RUN**=0) и флаг прерывания **BRK** устанавливается в «1».

Начальное состояние **SAR**=0xFFFF.

Счетчик команд CNTR

Счетчик команд **CNTR** - специализированный 16-разрядный регистр, предназначенный для отладки DSP-ядра. Регистр **CNTR** задает пошаговый режим исполнения программ в соответствии с таблицей 2.11.

Таблица 2.11. Регистр CNTR.

CNTR	Режим исполнения программ
0x0000	Нормальный режим исполнения программ. Число исполняемых программных команд не ограничено.
N>0	Пошаговый режим исполнения программ. После исполнения N инструкций DSP-ядро переходит в состояние останова (RUN=0) и флаг прерывания BRK устанавливается в «1»

Начальное состояние **CNTR**=0x0000.

Доступ ко всем регистрам DSP-ядра из программы DSP осуществляется по имени регистра.

Примеры:

```
MOVE    CCR,R0
ADDL    R2,R0
MOVE    0x0010,A2
MOVE    R0,(A2)
MOVE    R2,CCR
```

В данном примере содержимое регистра **CCR** помещается в **R0**, затем к **R0** прибавляется содержимое регистра **R2**. После этого в адресный регистр **A2** помещается адрес 0x0010, содержимое регистра **R0** записывается по адресу **A2**, а содержимое регистра **R2** помещается в регистр **CCR**. Этот пример иллюстрирует обращения к регистрам DSP-ядра.

Примечание: запись имени адресного регистра в скобках (например, (A2)) в команде пересылки означает, что пересылка будет осуществлена не в сам адресный регистр, а в ячейку памяти DSP-ядра, адрес которой содержится в адресном регистре. Подробнее об этом см. страницу "Способы адресации".

3.4. Программа на языке Ассемблера

Текст программы DSP, написанной на языке *Ассемблера*, состоит из последовательности исходных инструкций.
В теле одной инструкции может выполняться несколько команд (операций). При этом, каждая инструкция должна быть расположена на отдельной строке.

Таким образом, в рамках данного файла справки приняты следующие терминологические определения:

Инструкция – набор команд (операций), выполняющихся одновременно.

Команда (операция) – часть инструкции, определяющая действие того или иного операционного устройства DSP-ядра.

Инструкции программы DSP размещаются в программной памяти **PRAM** DSP-ядра последовательно в порядке возрастания адреса. Каждая инструкция занимает в **PRAM** одно или два 32-х разрядных слова. Адрес чтения из программной памяти следующей команды формируется с помощью регистра программного счетчика **PC**, автоматически инкрементирующегося (на 1 или 2) при последовательном ходе программы.

Файл с программой для DSP состоит из одной и более секций. Это могут быть секции текста и данных. Начало секции текста определяется директивой **`.text`**, а секции данных - директивой **`.data`**. Если в файле необходимо определить несколько различных секций текста или данных, необходимо указывать в начале каждой секции ее имя и заканчивать каждую секцию директивой **`.end`**. Подробнее данные директивы рассмотрены в главе "**Директивы ассемблера**"книги "**DSP Tools. User's Guide**". Размещение в памяти секций программы DSP-ядра рассматривается на странице "Секции DSP".

Секция текста программы DSP содержит набор инструкций. Инструкция может содержать следующие поля:

1. **Поле метки** - поле задания символического имени. Это поле всегда идет первым в инструкции программы DSP. Подробнее о задании символических имен в программе DSP см. страницу "Символы DSP";
2. **Поле первой операции** - в данном поле может содержаться команда DSP-ядра в мнемонической записи, директива Ассемблера или вызов макроопределения;
3. **Поле операндов первой операции** - данное поле содержит набор аргументов, необходимых для первой операции. Аргументами могут являться регистры **RF**, регистры **AGU**, управляющие регистры и выражения;
4. **Поле второй операции** - поле содержит вторую команду DSP-ядра и задается в соответствии с форматами **8** программы DSP. Допустимо исполнять в одной инструкции две команды DSP в том случае, если они относятся к разным группам команд - **OP1** и **OP2**. Подробнее об этом см. страницу "Параллельные операции";
5. **Поле операндов второй операции** содержит аргументы для второй команды DSP. Этими аргументами могут быть только регистры **RF**;
6. **Поле первой пересылки** - здесь определяется первая пересылка данных. Источником в операции пересылки могут быть регистры данных (**R** - 16-разрядные или **R.L** - 32-разрядные), регистры управления **RC**, память **XRAM** или **YRAM**, и данные `#5`, `#16` или `#32`. Получателем могут являться регистры данных, регистры

управления **RC**, память **XRAM** или **YRAM**. При этом для адресации памяти данных **XRAM** или **YRAM** используются регистры **AGU**. Подробнее об этом см. страницу "Способы адресации";

7. **Поле второй пересылки** допустимо только для форматов **8a** и **8b**. В данном поле осуществляется только пересылка данных из памяти **YRAM** в регистр **R0**.
8. **Поле комментария** - здесь задаются комментарии к тексту программы. Комментарии следует начинать с символа `;`. Для многострочных комментариев применима нотация *C* - скобки **`/*`** и **`*/`**.

Минимально инструкция программы DSP содержит поле первой операции и поле операндов первой операции. Например **`ADD`** `R2,R0`.

Пример инструкции программы DSP:

```
Instr:  FMPY R2,R12,R12  FADD R4,R10,R10 (A0)+,R8
```

Данная инструкция содержит поле метки (определяется символическое имя `Instr`), операцию **`FMPY`**, операцию **`FADD`** и пересылку одного слова из памяти данных в регистр `R8` с пост-увеличением содержимого адресного регистра `A0` на единицу.

Краткое описание форматов инструкций DSP-ядра приведено здесь.

Примечание: некоторые операции DSP, такие как, например, операции пересылки, требуют при задании операндов - регистров **RF** - указать, используется ли регистровая пара размерностью 32 бита, или же один 16-битный регистр. Для явного указания используется суффикс **`.L`**. Пример:

```
MOVE 2,R0
MOVE 2,R0.L
```

Здесь, в первой и во второй инструкции, в регистр `R0` пересылается число `2`. Но при этом в первом случае суффикс **`.L`** не указан, поэтому число `2` будет размещено в 16-битном регистре `R0`, а старшие 16 бит (регистр `R1`) останутся неизменны. Во втором случае суффикс **`.L`** указан, поэтому число `2` будет размещено в 32-битной регистровой паре `(R1,R0)`, то есть в 32-битном регистре `R0` будет расположено число `0x00000002`.

В секции данных программы DSP осуществляется задание констант, а также выделение места в памяти данных DSP-ядра для хранения результатов программы. Для выделения места в памяти применяются директивы Ассемблера DSP, такие как **`.float`**, **`.real`**, **`.word`**, **`.space`**, **`.skip`** и другие. Все они описаны в главе "**Директивы ассемблера**"книги "**DSP Tools. User's Guide**". При использовании директив, задающих данные размером меньше 32 бит, следует помнить, что адресовать память DSP можно только словно, то есть по 32 бита.

Пример секции данных программы DSP:

```
 .data
InputA:    .word 0xffffffff
InputB:    .real -0.3425247e-5
TheArray:  .space 20*4,0x01
 .end
```

В данном примере в секции данных сначала директивой **`.word`** в памяти размещается число `0xffffffff`. Это число будет доступно в памяти по адресу,

определенному символом `InputA`. Затем, задается символ `InputB`, по нему будет доступно размещенное в следующем слове памяти число `-0.3425247e-5`. После этого директивой **`.space`** задается массив из 20 слов. Так как директива **`.space`** выделяет место под байты, для задания 20 слов используется запись `20*4`, то есть 80 байт. Каждый байт заполняется значением `0x01`.

Примеры программ DSP-ядра, написанных на языке Ассемблера, рассматриваются в главе "Примеры создания проектов".

Описание набора команд ядра DSP приведено в книге "**DSP Instructions Set**".

3.5. Форматы инструкций Ассемблера

Ассемблер DSP-ядра поддерживает форматы инструкций, приведенные в таблицах 2.11 и 2.12:

Таблица 2.11. Инструкции с распараллеливанием операций.

Команды с распараллеливанием операций					
№	условие	Блок MS (умнож, сдвиги)	Блок ALU (+/-/логика)	Пересылки память-регистр Передачи регистр-регистр	Длина, 32-р слов
1	-	OP #5/S1,S2,D	OP[s] S1,S2,D	XRAM YRAM	2
2	-	OP #5/S1,S2,D	OP[s] S1,S2,D	R. L/R YRAM	2
3	-	OP #5/S1,S2,D	OP[s] S1,S2,D	RC -	2
4	CC	OP #5/S1,S2,D	OP[s] S1,S2,D	R. L	2
5	CC	OP S,D		#32/#16 -> R.L/R/RC -	2
6	-	OP #5/S,D		XRAM -	1/2
7	-	OP #5/S,D		R.L/R/RC -	1

Ограничения:

1. Для формата 1: пересылки **YRAM** не совместимы с операцией **`ADDSUB`**.
2. Для формата 1: запись в **XRAM** не совместима с операцией **`ASRLE`**.
3. Для формата 5: нельзя осуществлять пересылки с **RC**-регистрами **PDNR**, **CCR** (секционными).
4. Для формата 4: требуется явно указывать `R.L`.

Таблица 2.12. Инструкции без распараллеливания операций.

№	Условие		Длина, 32-р слов
1	CC	OP #5/S1,S2,D	1
2	CC	OP #32/ #16/S1,S2,D	2/1/1
3	CC	MOVE XRAM	1/2
4	CC	MOVE #32 -> R.L	2
5	-	MOVE #16 -> R/RC	1
6	CC	MOVE R.L/R/RC	1

Ограничения:

Для формата 3: для адресации со смещением `(A+dspl)` инструкция будет двухсловной.

Пояснения к форматам инструкций:

1. Для любого формата инструкция исполняется за 1 такт.
2. `S1`, `S2` – регистры-источники, `D` – регистр-приемник регистрового файла.
3. Регистровый файл содержит 32 16-разрядных регистров с номерами 0:31 или 16 32-разрядных регистров с четными номерами 0,2..30.
4. `R.L` – 32-разрядный регистр регистрового файла, `R` – 16-разрядный регистр регистрового файла, `RC` – 16-разрядный управляющий регистр.
5. **`#n`** – n-разрядный непосредственный операнд, **`#5`** – непосредственный параметр сдвига для операций сдвига.
6. `cc` – условие выполнения инструкции. Подробнее условно исполняемые инструкции рассматриваются здесь.
7. `[s]` – опция: признак масштабирования результата арифметической операции в **ALU**. При масштабировании результат может быть сдвинут вправо на 0/1/2 бита.

Подробно каждый из форматов инструкции DSP-ядра рассматривается в книге "**DSP Instructions Set**".

3.6. Запуск и останов программы

Ядро DSP процессора MultiCore является ведомым, поэтому для запуска программы DSP необходима команда программы RISC. Чтобы запустить программу DSP на исполнение, необходимо установить бит **RUN** регистра **DCSR** (**`DCSR`**`[14]`) в единицу. Команды программы RISC, осуществляющие запуск программы DSP описаны на странице "Запуск программы DSP".

Ядро DSP также может перейти в состояние **RUN** по сигналам от каналов **DMA MemCh**.

Тем не менее, останов программы ядро DSP может осуществить самостоятельно.

Программа DSP останавливается в следующих случаях:

- В ядре DSP исполняется инструкция **STOP**;
- Программный счетчик **PC** достигает адреса останова (установленного в регистре **SAR**);
- Ядро DSP выполняет заданное число инструкций. Число инструкций задается в регистре **CNTR**. Если **`CNTR`**`=0`, число исполняемых инструкций не ограничено.

Программа DSP также переходит в состояние останова если в бит **RUN** регистра **DCSR** программой RISC был принудительно записан нуль.

3.7. Символы DSP

Символические имена (**метки**) в программе DSP применимы в инструкциях DSP для программных переходов, ветвлений, вызовов подпрограмм и записи/чтения данных из памяти.

Для записи символического имени можно использовать любую комбинацию букв (латинского алфавита), цифр и символов подчеркивания (_). Первым символом имени должна быть буква. Прописные и строчные буквы символического имени считаются различными, то есть метки `ABCDEF` и `Abcdef` являются разными.

Если при **задании символического имени** (сопоставлении его с инструкцией или ячейкой памяти), имя символа введено не с начала строки (перед именем символа встречаются пробелы), после имени символа следует ставить знак двоеточия (**:**). Если же символ задан с начала строки, двоеточие ставить необязательно. Тем не менее, для повышения читабельности текста программы, рекомендуется всегда ставить двоеточие при задании символического имени.

При **обращении к символу** (в инструкциях переходов, ветвлений и т.д.) следует указывать имя символа без двоеточия в конце.

Пример 1:

```
My_Instruction
  MOVE R0,R2.L
```

Задается символическое имя `My_Instruction`, сопоставляемое (при сборке проекта) с адресом инструкции **`MOVE`** `R0,R2.L` в памяти. Символ задается с начала строки, поэтому двоеточие не поставлено.

Пример 2:

```
My_Data:
  .word 0
```

Задается символическое имя `My_Data`, сопоставляемое с адресом 32-разрядного слова, зарезервированного в памяти директивой **`.word`** `0`. Символ задан не с начала строки, поэтому после имени символа поставлено двоеточие.

Пример 3:

```
J My_Instruction
```

Выполняется программный переход по адресу, сопоставленному с символическим именем `My_Instruction`. При обращении к символу двоеточие не ставится.

Если необходимо сделать тот или иной символ доступным в других секциях RISC и DSP, его следует объявить глобальным посредством директивы **`.global`** `имя_символа`.

Недопустимо использовать глобальные символы в качестве аргументов выражений, заданных не в той секции, где определен символ.

Примечание: при определении символического имени, указывающего на конец секции текста или данных, рекомендуется задавать символ перед последней инструкцией

(данными) секции. При этом, инструкция (данные) должны быть однословными, так как символу будет соответствовать адрес первого слова инструкции (данных).

Если же символ задан после всех инструкций (данных), ему будет соответствовать адрес, следующий за последним адресом секции. С этого же адреса может компоноваться следующая секция. В таком случае метка ее начала будет недоступна отладчику. Это не повлияет на работу программы, но информация об исполняемой в данный момент секции в процессе отладки может быть некорректна.

Примечание 2: если в программе DSP необходимо обратиться к внешнему символу, определенному в другой секции текста (данных) DSP, объявлять символ внешним (директивой **`.extern`**) необязательно. Компилятор Ассемблера DSP считает все символы, не определенные в контексте данной секции, внешними. Тем не менее, в секции, где символ определен, необходимо объявить его глобальным как показано выше.

3.8. Выражения

В программе DSP-ядра допустимо в качестве операндов команд или директив Ассемблера вводить **выражения**. **Выражением** является вычисляемая в процессе компиляции и компоновки величина. При этом вычисленное значение выражения должно соответствовать требованиям формата команды или директивы. Тем не менее, промежуточные результаты могут выходить за эти рамки.

Выражения, которые могут быть вычислены на этапе компиляции, явным образом вставляются в код программы. Если же на этапе компиляции для вычисления выражения недостаточно данных (за счет наличия символических имен), вычисление будет перенесено на этап компоновки. При этом, недопустимо использовать в выражении символические имена, определенные в разных областях памяти.

Числовые константы, которые встречаются непосредственно в виде операндов или являются частью выражений следует записывать следующим образом:

- для целочисленных констант можно использовать различные системы счисления, при этом возможны следующие способы указания системы счисления:
- `0` с одним из символов `oOqQ` указывает на восьмеричную систему счисления (например, `0q27723577`);
- `0` с символом `hHxX` указывает на шестнадцатеричную систему счисления (например, `0xFFFF`);
- `0` с символом `B` указывает на двоичную систему счисления (например, `0B100011011101`);
- по умолчанию целые числа, которые начинаются с нуля, тоже относятся к восьмеричным;
- остальные числа по умолчанию считаются десятичными.
- вещественная константа имеет следующий формат: *+/- целая_часть [.дробная_часть] [E/e] [+/-] экспонента*, причем либо разделитель *дробной части* "**.**", либо *экспонента* должны присутствовать.

При написании целочисленных выражений допустимо использовать следующие **операторы**:

- **(выражение)** - объединение элементов выражения;
- **||**, **&&** - логические операции *ИЛИ* и *И*;
- **==**, **<>**, **<**, **<=**, **>=** , **>** - операции сравнения;
- **+**, **-** - операции сложения и вычитания;
- **&**, **^**, **~**, **|**, **!** - побитовые операции *И*, *ИСКЛЮЧАЮЩЕЕ ИЛИ*, *НЕ*, *ИЛИ* и логическое *НЕ*;
- *****, */*, **%**, **<<**, **>>** - операции умножения, деления, получения остатка, сдвига вправо и влево;
- **унарный -** и **+**.

Приоритет выполнения операций соответствует вышеприведенному порядку. Например, оператор "**!**" имеет более высокий приоритет, чем операция сравнения.

Ассемблер не включает выражений с участием значений с плавающей точкой, за исключением специальных DSP-констант, приведенных ниже.

DSP-ядро поддерживает ряд специальных форматов с фиксированной точкой. Кроме того, предполагается использование форматов с программной плавающей точкой. Для того чтобы обеспечить возможность использования их в ассемблерном коде, введена специальная операция "**[ ]**". Аргументы для этой операции должны быть определены. Не допускается использование неопределенных имен. Имеются следующие форматы записи операции:

- **[целое выражение]** - эквивалентно записи *(выражение)*;
- **[старшее полуслово, младшее полуслово]** - позволяет задать 32-битное слово из двух половинок, в качестве полуслова можно использовать либо целочисленное выражение, либо вещественную константу в диапазоне *(-1.,1.);*
- **[константа с плавающей точкой]** - биты *мантиссы* (*32*) для константы в форме программной плавающей точки;
- **[^константа с плавающей точкой]** - биты *экспоненты* (*16*) для константы в форме программной плавающей точки;
- **[@старший байт, младший байт]** - задание *16*-битной константы из двух *8*-битных половин.

Таблица 2.13 иллюстрирует форматы данных DSP-ядра.

Таблица 2.13. Форматы данных DSP-ядра.

№	Описание формата	Непосредственный операнд	Константа в памяти DSP
1	Целый 16-разрядный	*# -32767*	*.dw –32767*
2	Целый 32-разрядный	*# -32768*32767*	*.dl –0xFFFFFFFF*
3	Целый комплексный (16 разр., 16 разр.)	*# [-32767,-0x8000]*	*.dw –32767, -0x8000*
4	Дробный 16-разрядный	*# -0.875*	*.fr –1.0*

5	Дробный 32-разрядный	*# -0.875*	*.frl –0.875*
6	Дробный комплексный (16 разр., 16 разр.)	*# [-0.5,-0.375]*	*.single –0.5, -0.375*
7	Программная плавающая точка	*-31.25e-1*	*.double –31.25e-1*
8	Плавающая точка (32 бита)	*#2.5*	*.real -3.7e6*

Пояснения:

1. Значение числа в дробном 16-разрядном формате равно:

$B_{14}*2^{-1} + B_{13}*2^{-2} + \ldots + B_0*2^{-14}$.

Код – дополнительный, бит B_0 – младший. Выражение для 32–разрядного формата аналогично.

2. Значение числа с плавающей точкой равно:

$2^{\pm E} * (\pm F)$, где:

E – экспонента в 16-разрядном дополнительном коде, | E | <16384,

F – мантисса в 32-разрядном дополнительном коде.

Примеры использования:

```
1 0000 00000000     addl   #0.5,R2,R0
1      40000000
2 0002 07FCE184     add    #-0.25,R0
6 0008 00000000     .double 0.5
6      40000000
```

Макрос для присвоения значения константы с плавающей программной точкой 3-м регистрам.

```
9                   .macro  movfi,v,a,b
10                  move   #[^\v],\a
11                  move   #[\v],\b.L
12                  .endm
```

Использование макроса:

```
13                movfi 0.25,r1,r2
13 0010 00028700    > move #[^0.25],r1
13      0000FFFF
13 0012 00058700    > move #[0.25],r2.L
13      40000000
14 0014 20004000    .fr    0.5,0.25,-0.5,-0.25
14   14 E000C000
23 0021 40000000    .frl   0.5, 0.25, 0.125
23      20000000
23      10000000
```

Для отличия *32*-битных значений с плавающей точкой от значений с фиксированной точкой в командах следует использовать псевдокоманды **`.ffloat`** и **`.ffix`**. Эти псевдокоманды рассмотрены в главе "**Директивы Ассемблера**" книги "**DSP Tools**". По умолчанию работает режим с фиксированной точкой.

3.9. Секции DSP

При компоновке, программа DSP разделяется на секции текста и данных. Секции текста программы DSP содержат исполняемый код программы, в то время как секции данных содержат все наборы необходимых для работы программы данных, заданных программистом при написании программы. Правила написания секций текста и данных программы DSP приведены на странице "Программа на языке Ассемблера".

Настройка размещения секций DSP в памяти MultiCore осуществляется посредством диалога настроек проекта (пункт *Settings* меню **Project**). В данном диалоге для каждой секции следует вводить имя, адрес внутри модуля (**Unit VMA**) и адрес фактического размещения секции в памяти (**VMA**). При этом, **Unit VMA** - есть адрес, по которому секция будет загружена во внутреннюю память DSP-ядра. То есть, если, например, секция текста будет исполняться с начала программной памяти **PRAM**, то для нее `Unit VMA=0x00000`. В качестве адреса **VMA** для секций DSP следует указывать адрес, по которому секция будет располагаться в памяти MultiCore. То есть, если секцию необходимо сразу загрузить во внутреннюю память DSP, ее адрес **VMA** будет вычисляться по формулам:

- `VMA=0xB8400000 + (Unit VMA << 2)` для секции данных и
- `VMA=0xB8440000 + (Unit VMA << 2)` для секции текста.

Здесь `0xB8400000` - виртуальный адрес начала памяти данных **XRAM** со стороны RISC, а `0xB8440000` - виртуальный адрес начала программной памяти **PRAM** со стороны RISC.

Если же указанный адрес **VMA** не попадает в область внутренней памяти DSP-ядра, перед использованием секцию необходимо будет загрузить из программы RISC. Подробнее об этом см. главу "Проекты с оверлейными структурами".

Также следует учитывать, что секции текста (данных), находящиеся в одном модуле и имеющие одно и то же имя, будут скомпонованы *в одну секцию* текста (данных).

3.10. Способы адресации

В ядре DSP процессора MultiCore применяются следующие режимы адресации:

- **Прямая адресация** - используется при пересылках данных между регистрами данных или управления DSP-ядра;
- **Абсолютная адресация программной памяти** и **адресация программной памяти относительно программного счетчика** - используется при организации программных переходов и циклов;
- **Косвенная адресация** - используется при обменах с памятью данных.

Все виды адресации ядра DSP приведены в таблице 2.14. В правом столбце приводится синтаксис использования того или иного вида адресации в программе DSP на языке *Ассемблера*.

Таблица 2.14. Виды адресации.

Виды адресации	Использование регистров AGU			Тип ссылки					Ассемблерный синтаксис
	An (AT)	In (IT,DT)	Mn (MT)	C	R	P	X	Y	
Прямая регистровая адресация									
Регистр данных или управления	-	-	-	√	√				<имя регистра>
Косвенная регистровая адресация									
Отсутствие модификации адреса (XRAM)	+	-	-				√		(An)
Отсутствие модификации адреса (YRAM)	+	-	-					√	(AT)
Пост - инкремент на 1	+	-	+				√		(An) +
Пост - инкремент на In	+	+	+				√		(An) + In
Пост - инкремент на IT	+	+	+					√	(AT) + IT
Пост - инкремент на DT	+	+	+					√	(AT) + DT
Пост - декремент на 1	+	-	+				√		(An) -
Пост - декремент на In	+	+	+				√		(An) - In
Адресация со смещением на In (XRAM)	+	+	+				√		(An + In)
Адресация со смещением на IT (YRAM)	+	+	+					√	(AT + IT)
Непосредственное смещение	+	-	+				√		(displ)
Абсолютная адресация программной памяти									
Абсолютная прямая адресация	-	-	-			√			#I16
Абсолютная косвенная адресация	+	-	-			√			(An)
Адресация программной памяти относительно программного счетчика (PC)									
Относительная прямая адресация	-	-	-			√			PC + #I16
Относительная косвенная адресация	+	-	-			√			PC + An

Обозначения:

C - ссылка на регистр управления RC;

R - ссылка на регистр данных R;

P - ссылка на память программ PRAM;

X - ссылка на память данных XRAM;

Y - ссылка на память данных YRAM;

Прямая адресация

Прямая регистровая адресация определяет, что операндом является один или более регистров данных или управления (включая регистры адресного генератора).

Операндом может быть один, два или три регистра, как это определяется соответствующей командой. Используемая при этом в команде ссылка называется регистровой ссылкой.

Пример: **MOVE** R7,CCR

R7 – регистровая ссылка на регистр данных R7 (ссылка типа **RF**);

CCR – регистровая ссылка на регистр управления **CCR** (ссылка типа **RC**).

Виды адресации программной памяти

При формировании адреса программной памяти может использоваться абсолютная и относительная, прямая и косвенная адресация.

Абсолютная адресация программной памяти применяется в операциях программных переходов и циклов, использующих абсолютный адрес перехода – **J**, **JD**, **JS**, **DO**, **DO_R**. Относительная адресация памяти программ применяется в операциях переходов и циклов, формирующих адрес перехода относительно программного счетчика **PC** – **B**, **BD**, **BS**, **DOR**, **DOR_R**.

И абсолютная, и относительная адресация может быть либо прямой, когда адрес перехода задается непосредственным операндом, либо косвенной когда адрес перехода содержится в адресном регистре.

Примеры:

1) **J** #Symbol - абсолютная прямая адресация (в инструкции метка кодируется как непосредственное значение);
2) **BS** A7 – относительная косвенная адресация (относительная – потому, что команда **BS** подразумевает адресацию относительно **PC**, косвенная – потому, что относительный адрес перехода задан в регистре A7).

Косвенная адресация памяти данных

Косвенную адресацию памяти данных (**XRAM**, **YRAM**) обеспечивают адресные генераторы **AGU** и **AGU-Y**.

При косвенной адресации для указания на ячейку памяти используется адресный регистр **An**, а в общем случае – группа регистров **An**, **In**, **Mn**, позволяющих по определенным правилам вычислить значение указателя. Используемые режимы генерации адреса приводятся ниже.

- **Отсутствие модификации адреса (An)** - адрес операнда содержится в адресном регистре. При выполнении команды значение адреса не изменяется. *Пример*: **MOVE** R2,(A0) - по адресу, содержащемуся в адресном регистре **A0**, пересылается содержимое регистра **R2**.
- **Пост – инкремент на 1** - адрес операнда содержится в адресном регистре **An**. После использования адреса его значение увеличивается на 1 и сохраняется в том же адресном регистре. Тип используемой арифметики определяется соответствующим регистром модификатора. Регистр смещения не используется. *Пример*: **MOVE** (A0)+,R2 - с адреса, содержащегося в адресном регистре **A0**, считывается 32-

разрядное слово и помещается в регистр **R2**, после чего адрес в **A0** увеличивается на единицу.

- **Пост – инкремент на In** - Адрес операнда содержится в адресном регистре **An**. После использования адреса его значение увеличивается на величину смещения, содержащуюся в регистре **In**, и сохраняется в том же адресном регистре **An**. Тип используемой арифметики определяется соответствующим регистром модификатора **Mn**. Содержимое регистра смещения не изменяется. *Пример*: **MOVE** (A0)+I0,R0 - содержимое ячейки памяти с адресом **A0** помещается в регистр **R0**. Новый адрес **A0** вычисляется в зависимости от содержимого регистра **M0** как показано далее.
- **Пост – декремент на 1** - адрес операнда содержится в адресном регистре **An**. После использования адреса его значение уменьшается на 1 и сохраняется в том же адресном регистре. Тип используемой арифметики определяется соответствующим регистром модификатора. Регистр смещения не используется. *Пример*: **MOVE** (A0)-,R0 - с адреса, содержащегося в адресном регистре **A0**, считывается 32-разрядное слово и помещается в регистр **R2**, после чего адрес в **A0** уменьшается на единицу.
- **Пост – декремент на In** - адрес операнда содержится в адресном регистре **An**. После использования адреса его значение уменьшается на величину смещения, содержащуюся в регистре **In**, и сохраняется в том же адресном регистре **An**. Тип используемой арифметики определяется соответствующим регистром модификатора **Mn**. Содержимое регистра смещения не изменяется. *Пример*: **MOVE** (A0)-I0,R0 - содержимое ячейки памяти с адресом **A0** помещается в регистр **R0**. Новый адрес **A0** вычисляется в зависимости от содержимого регистра **M0** как показано далее.
- **Адресация со смещением на In** - адресом операнда является сумма значений, хранящихся в адресном регистре **An** и в регистре смещения **In**. Тип используемой арифметики определяется соответствующим регистром модификатора **Mn**. Содержимое регистра адреса **An** и регистра смещения **In** остается неизменным. *Пример*: **MOVE** (A0+I0),R0 - содержимое ячейки памяти с адресом **A0+I0** помещается в регистр **R0**. Сумма вычисляется с учетом установленной в **M0** арифметики как показано далее. Содержимое **A0** и **I0** не изменяется.
- **Непосредственное смещение (An + displ)** - адресом операнда является сумма значений, хранящихся в адресном регистре **An** и непосредственного смещения, содержащегося в поле команды. Тип используемой арифметики определяется соответствующим регистром модификатора **Mn**. Содержимое регистра адреса **An** остается неизменным. Регистр смещения **In** не используется. *Пример*: **MOVE** (A0+10),R0 - содержимое ячейки памяти с адресом **A0+10** помещается в регистр **R0**. Сумма вычисляется с учетом установленной в **M0** арифметики как показано далее. Содержимое **A0** не изменяется.

Примечание: так как DSP-ядро *Elcore_x4* имеет словную адресацию, все операции пересылки данных в память (**XRAM**,**YRAM**) DSP-ядра (и из памяти) используют 32-битные данные для пересылки.

Типы адресной арифметики

Адресный генератор поддерживает четыре типа адресной арифметики:

- линейная,
- модульная,

- модульная с кратным обращением,
- арифметика с обратным переносом.

Предоставляемые возможности достаточны для организации в памяти структур данных типа очередей (*FIFO*), линий задержки, циклических буферов, стеков, буферов с обратным порядком адресации для реализации *БПФ*.

Работа с данными при этом сводится в большей степени к манипуляциям с адресами, чем к пересылкам больших блоков данных.

Тип используемой адресной арифметики определяется значением, хранящимся в регистре модификатора. Для модульной арифметики содержимое регистров модификаторов определяет также модуль. Каждый адресный регистр имеет один связанный с ним регистр модификатора.

Значения модификатора **Mn** и соответствующие им типы адресной арифметики указаны в таблице 2.15:

Таблица 2.15. Модульная адресация.

Модификатор Mn	Адресная арифметика
0x0000	Арифметика с обратным переносом
0x0001	Модуль 2
0x0002	Модуль 3
…	…
0x7FFE	Модуль 32767 (2^{15}-1)
0x7FFF	Модуль 32768 (2^{15})
0x8001	Модуль 2 с кратным обращением
0x8003	Модуль 4 с кратным обращением
0x8007	Модуль 8 с кратным обращением
….	…
0x9FFF	Модуль 2^{13} с кратным обращением
0xBFFF	Модуль 2^{14} с кратным обращением
0xFFFF	Линейная арифметика (модуль 2^{16})
Остальные комбинации - резерв	

Линейная адресная арифметика (**Mn**=0xFFFF)

Модификация адреса выполняется с использованием обычной 16-разрядной линейной (по модулю 65536) арифметики. Для вычисления адреса могут использоваться 16-разрядное смещение, **In**, +1 или −1. Диапазон значений может рассматриваться как знаковый (от −32768 до +32767) либо как беззнаковый (от 0 до 65535), так как адресное **ALU** работает в обоих случаях одинаково.

Адресная арифметика с обратным переносом (**Mn**=0x0000)

Этот вариант адресной арифметики выбирается посредством установки регистра модификатора в 0. Модификация адреса в этом случае выполняется аппаратно с распространением переноса в обратном направлении – от старших разрядов к младшим.

Операция модификации адреса с обратным переносом эквивалентна последовательному выполнению следующих процедур:

- Изменению на обратный порядка следования разрядов в регистрах адреса и смещения (при этом старший бит становится младшим и т.д.);
- Модификации адреса посредством нормальной операции сложения;
- Возвращению первоначального порядка следования разрядов адреса.

В случае, когда величина смещения составляет $2^{(k-1)}$ (целая степень двойки), такая модификация адреса эквивалентна:

- Обращению порядка следования k младших разрядов **An**;
- Увеличению на 1;
- Возвращению исходного порядка следования k младших разрядов **An**.

Рассматриваемый режим адресной арифметики удобен при реализации алгоритма быстрого преобразования Фурье (*БПФ*).

Модульная адресная арифметика (**Mn**=M-1)

Модификация адреса выполняется по модулю M, где M - целое число в пределах от 2 до 32768. Арифметика по модулю M вынуждает значение адреса оставаться в пределах диапазона значений, отличающихся друг от друга не более чем на M-1.

Величина M-1 хранится в регистре модификатора адреса. Нижняя граница диапазона (базовый адрес) должна иметь нули в младших k разрядах, где $2^k \geq M$. Верхняя граница диапазона определяется как сумма нижней границы и модуля минус единица (базовый_адрес+M-1).

base_addr={**An**[15:k],{k{0}}};

base_addr ≤ XAB ≤ base_addr+M-1;

Нижняя и верхняя границы диапазона определяются значением **An**. При этом необязательно устанавливать **An** равным базовому адресу. Достаточно того, чтобы величина **An** находилась в пределах требуемого диапазона.

Если при вычислении адреса в этом режиме используется смещение **In**, его величина не должна превышать M. Выходной адрес XAB для этого случая определяется формулой:

$XAB = base\_addr + (An[k-1:0] \pm In)_{modM}$;

Рассматриваемый тип адресной арифметики удобен при организации циклических буферов для реализации на их основе структур данных типа очередей (*FIFO*), линий задержки и т.п.

Кратная модификация адреса по модулю

Этот тип адресной арифметики выбирается посредством установки в «1» 15-го разряда регистра модификатора **Mn**, как это показано в таблице выше.

Модификация адреса выполняется по модулю M, где M - степень двойки в пределах от 2^1 до 2^{14}. Арифметика по модулю M вынуждает значение адреса оставаться в пределах диапазона значений, отличающихся друг от друга не более чем на M-1.

Величина M-1 хранится в младших 15-ти разрядах регистра модификатора адреса **Mn**. Нижняя граница диапазона (базовый адрес) должна иметь нули в младших k разрядах, где $2^k \geq M$. Верхняя граница диапазона определяется как сумма нижней границы и модуля минус единица (базовый_адрес+M-1).

Выходной адрес XAB и границы диапазона определяются по тем же формулам, что и при обычной модульной арифметике:

```
XAB=base_addr+(An[k-1:0]±In)modM;
base_addr={An[15:k],{k{0}}};
base_addr ≤ XAB ≤ base_addr+M-1;
```

Отличие состоит в том, что для данного типа адресной арифметики величина смещения **In** может быть произвольной.

3.11. Условно исполняемые инструкции

Условно исполняемые инструкции есть инструкции, исполняемые ядром DSP только в том случае, если указанное условие истинно. Мнемоника нужного условия отделяется от имени операции точкой. Например, `ADDL.eq R10,R2,R12`. В данном случае, указано условие **eq**.

Для проверки условий в DSP-ядре существует регистр **CCR** (*Condition Code Register*). В этот регистр помещаются признаки результата предыдущей операции, исходя из которых и устанавливается истинность или ложность того или иного условия. Описание регистра **CCR** приведено на странице "Регистры DSP".

Условия, используемые в программе DSP, приведены в таблице 2.16:

Таблица 2.16. Условия.

№ кода	Код условия	Условие	Мнемоника
0	0000	C=0	**.cc** (Carry Clear) / .**hs** (Higher or Same)
1	0001	C=1	**.cs** (Carry Set) / .**lo** (LOwer)
2	0010	Z=0	**.ne** (Not Equal to zero)
3	0011	Z=1	**.eq** (EQual to zero)
4	0100	N=0	**.pl** (Plus)
5	0101	N=1	**.mi** (MInus)
6	0110	N^V=0	**.ge** (GrEater than or equal)
7	0111	N^V=1	**.lt** (Less Than)
8	1000	Z|(N^V)=0	**.gt** (Greater Than)
9	1001	Z|(N^V)=1	**.le** (Less than or Equal)
10	1010	U=0	**.nr** (NoRmalized)
11	1011	U&(~V)=1	**.un** (UnNormalized)
12	1100	V=1	**.vs** (oVerflow Set)
13	1101	V=0	**.vc** (oVerflow Clear)
14	1110	t=1	**.t** (признак истинности предыдущего условия)
15	1111	-	**.al** (Always)

Примечание: В поле **Условие** для каждого модификатора условия приводится состояние того или иного бита (битов) регистра **CCR**, необходимое для истинности условия.

Пример:

```
CMPL          R0,R2
ADDL.eq   0x10,R2,R2
SUBL.t    0x10,R0,R0
J.t       The_Symbol
```

В данном примере осуществляется сравнение содержимого 32-разрядных регистров `R0` и `R2`. В том случае, если значения в регистрах равны (выполняется условие `.eq`), к содержимому регистра `R2` прибавляется число `0x10`, а из содержимого регистра `R0` вычитается число `0x10`, после чего осуществляется переход по некоторой метке `The_Symbol`. При этом, в операциях **`ADDL`** и **`J`** использован модификатор условия `.t`, то есть эти операции выполняются в том случае, если выполнена предыдущая условная операция. Таким образом, если выполняется операция **`ADDL.`**`eq` (то есть, `R2=R0`), то за ней последовательно выполняются операции **`SUBL.`**`t` и **`J.`**`t`. Если же значения регистров `R0` и `R2` не равны, ни одна из следующих за сравнением операций выполнена не будет.

Примечание: термин "*условно исполняемая инструкция*" означает, что условие распространяется на все операции *OP1*, *OP2* и операции пересылки, указанные в инструкции. То есть, в случае истинности условия параллельно исполняются все операции данной инструкции. В случае же, если условие ложно, ни одна из операций не будет исполнена.

Описание набора команд ядра DSP, а также описание процесса формирования признаков результата операции приводится в книге "**DSP Instructions Set**".

3.12. Макроопределения

Ассемблер для DSP-ядра позволяет программисту определять в тексте программы **макросы** (макроопределения). Макросом называется блок кода, расположенный между директивами **`.macro`** и **`.endm`**, имеющий имя, а также ноль и более параметров. После определения макроса, программист может использовать лишь его имя и набор параметров, вместо того, чтобы вставлять в текст один и тот же код несколько раз. Во время сборки проекта компилятор, обнаружив вызов макроса, автоматически подставит вместо него блок кода, определяющий макрос. Если при вызове указаны какие-либо параметры, они также будут подставлены в код, если нет - будут подставлены значения параметров по умолчанию.

Для задания макроопределения используется следующий синтаксис:

- Заголовок макроопределения (директива **`.macro`** `имя_макроса`);
- Тело макроопределения - блок кода программы;
- Директива **`.endm`**.

Задание имени макроопределения и параметров может быть выполнено одним из следующих способов:

- `Имя_макроопределения` **`.macro`** `параметры`
- **`.macro`** `Имя_макроопределения параметры`
- **`.macro`** `Имя_макроопределения (параметры)`

При этом параметры макроса отделяются друг от друга запятыми.

Параметры могут отделяться друг от друга запятой или пробелами и включать задание значения по умолчанию:

```
4               .macro abc x,y=2 z=7
5               .dl    \x
6               .dl    \z-\y
7               .endm
```

При вызове **макроопределения** можно не указывать все параметры, при этом пропущенные параметры будут либо "пустыми", либо иметь соответствующее значение по умолчанию.

Как видно из вышеприведенного примера, для выполнения подстановки параметров следует использовать обращение следующего вида: *\имя_параметра*. Возможна также ссылка на параметр в форме *&имя_параметра*.

При вызове **макроопределения** возможно указание модификатора, для ссылки на модификатор в теле **макроопределения** следует использовать `\0`, при этом точка не входит в значение параметра:

```
.macro test
    b.\0  next
.endm
test.eq
```

Если в теле **макроопределения** необходима безусловная вставка некоторого текста, которые содержит обрабатываемые символы, то их можно защитить от обработки при помощи заключения в конструкцию следующего вида: *\(текст)*.

Для преждевременного выхода из **макроопределения** следует использовать директиву **`.exitm`**. Для удаления **макроопределения** - директиву **`.purgem`**.

Перед стартом процесса вставки макроопределения осуществляется преобразование параметров. В частности, отбрасываются окружающие кавычки.

3.13. Программные переходы и ветвления

Для организации переходов и ветвлений в программе DSP используются операции **`J`** (*Jump*), **`JD`** (*Jump Delay*), **`B`** (*Branch*) и **`BD`** (*Branch Delay*).

Для указания адресов переходов/ветвлений допустимо использовать:

- символы;
- выражения;
- адресные регистры.

При этом, для переходов (*Jump*) новое значение программного счетчика **PC** будет равно адресу, указанному в команде, а для ветвлений (*Branch*) новое значение **PC** вычисляется как смещение от предыдущего значения (**PC**=**PC** + *offset*).

Переходы и ветвления в программе DSP могут быть как условными, так и безусловными. Для задания условного перехода/ветвления необходимо использовать модификаторы условий. Подробнее об этом см. страницу "Условно исполняемые инструкции".

Пример 1:

```
J.eq        The_Symbol
```

В данном примере осуществляется переход по метке `The_Symbol`. Переход осуществляется в том случае, если выполняется условие `.eq`.

Пример 2:

```
JD.ne  0x1<<2
NOP
```

Здесь осуществляется переход с задержкой по адресу, равному значению выражения `0x1<<2`, то есть по адресу `0x100`. Переход осуществляется в том случае, если выполняется условие `.ne`. В слоте задержки команды **JD** (*delay slot*) расположена операция **NOP**.

Пример 3:

```
B.pl        5
```

В этом примере программа осуществляет ветвление по адресу **PC**+5, то есть смещение на 5 слов относительно программного счетчика. Ветвление осуществляется в том случае, если выполняется условие `.pl`.

Пример 4:

```
BD      The_Symbol
NOP
```

В данном случае осуществляется ветвление по метке `The_Symbol`. При сборке программы DSP автоматически вычисляется смещение от строки **BD** `The_Symbol` до адреса метки `The_Symbol`. В операцию будет подставлено именно это смещение. Так как модификаторов условий не указано, ветвление является безусловным, то есть выполняется всегда. В слоте задержки команды **BD** (*delay slot*) расположена операция **NOP**.

Примечание: операции **JS** и **BS** рассматриваются на странице "Подпрограммы".

Описание набора команд ядра DSP приведено в книге "**DSP Instructions Set**".

3.14. Подпрограммы

Для вызовов подпрограмм в программе DSP используются операции **JS** (*Jump to Subroutine*) и **BS** (*Branch to Subroutine*).

При переходе на подпрограмму, адрес подпрограммы заносится в программный счетчик **PC**, адрес следующей за вызовом подпрограммы инструкции заносится в системный стек, а указатель системного стека инкрементируется. В конце подпрограммы

должна стоять инструкция **RTS** (*ReTurn from Subroutine*). По этой инструкции из системного стека в **PC** помещается адрес возврата из подпрограммы, а указатель системного стека уменьшается на единицу.

Таким образом, инструкций **JR**/**BS** и **RTS** достаточно для входа в подпрограмму и выхода из нее.

Адрес подпрограммы, указываемый в операциях **JR**/**BS** формируется так же, как и для программных переходов/ветвлений.

Пример:

```
CMPL  0,R2
JS.ne My_Subroutine   ;или BS.ne   My_Subroutine
MOVE  R2,(A0)

...

My_Subroutine:
  ADDL  R0,R2
  ADDL  0x10,R2,R2
  LSRL  2,R2,R2
  RTS
```

В данном примере осуществляется сравнение содержимого регистра R2 с нулем. Затем, если содержимое R2 не равно нулю, осуществляется переход на подпрограмму My_Subroutine, выполняющую некоторые вычисления. В конце подпрограммы расположена инструкция **RTS**. По этой инструкции происходит возврат из подпрограммы на строку **MOVE** R2,(A0).

Описание набора команд ядра DSP приведено в книге "**DSP Instructions Set**".

3.15. Организация циклов

Для организации циклов в программе DSP существуют операции **DO**, **DOFOR** и **ENDDO**. Операция **DO** организует цикл с заданным числом повторений. Операция **DOFOR** предназначена для организации бесконечных циклов. Операция **ENDDO** осуществляет преждевременный выход из цикла.

При оформлении цикла необходимо задать число итераций (для цикла **DO**) и адрес последней инструкции цикла.

При входе в цикл в DSP-ядре сохраняются адреса первой и последней команды цикла, а также число итераций (или флаг бесконечного цикла в случае операции **DOFOR**). Число повторений задается либо непосредственным операндом, либо 16-разрядным регистром регистрового файла **RF**. Адрес последней команды цикла задается либо меткой, либо непосредственным операндом. При этом, если перед заданием адреса стоит символ **#**, то адрес считается абсолютным. В противном случае адрес вычисляется относительно

текущего значения программного счетчика **PC**. При указании смещения относительно **PC** следует учитывать число слов (1 или 2), используемое для кодирования каждой инструкции цикла включая саму операцию **DO**.

При выходе из цикла по завершении необходимого числа итераций или по вызову операции **ENDDO**, стеки DSP-ядра возвращают сохраненные в них значения, счетчики стеков декрементируются, а в **PC** помещается адрес операции, следующей за последней операцией цикла.

Пример 1:

```
  DO  5,End_of_Cycle
   ADDL   R0,R2
   INCL   R0,R0
End_of_Cycle:
   MOVE   R2,(A0)+
```

В данном примере выполняется цикл из 5 повторений операций **ADDL**, **INCL** и **MOVE**. Число повторений задано непосредственным операндом, адрес последней инструкции - меткой End_of_Cycle.

Пример 2:

```
  MOVE   5,R6
  DO     R6,3
         ADDL   R0,R2
         INCL   R0,R0
End_of_Cycle:
         MOVE   R2,(A0)+
```

Пример 2 демонстрирует такой же цикл, как и *пример 1*, но число повторений задано в регистре R6, а адрес последней инструкции указан как смещение от **PC** на 3 слова.

Пример 3:

```
  DOFOR  End_of_Cycle
      ADDL   R0,R2
      INCL   R0,R0
      CMPL   0xFF,R2
      J.ge   Leave_Cycle
End_of_Cycle:
      MOVE   R2,(A0)+
      NOP
   ...
Leave_Cycle:
      ENDDO
```

В данном примере показано, как при помощи операции **DOFOR** оформить цикл с постусловием. Перед последней командой цикла (**MOVE**) содержимое регистра R2 сравнивается с непосредственным операндом 0xFF. В том случае, если содержимое регистра R2 больше или равно числу 0xFF, происходит переход по метке Leave_Cycle, по которой расположена операция **ENDDO**. После выполнения **ENDDO** цикл заканчивается и в **PC** устанавливается адрес следующей за циклом операции (в данном случае это **NOP**).

При организации циклов в программе DSP необходимо учитывать следующие ограничения:

- Заданное количество повторений цикла **DO** должно находиться в пределах *от 1 до 16383*, то есть не превышать размера регистра **LC**. Следует отдельно отметить, что указанное число итераций не следует указывать равным нулю. Поэтому, если в программе число итераций цикла **DO** переменное и задается регистром, следует проверять значение этого регистра перед входом в цикл. В противном случае программа может работать неверно;
- Количество вложенных циклов **DO**, **DOFOR** не должно превышать семи (ограничение связано с глубиной стека циклов);
- Количество вложенных друг в друга циклов **DO**, **DOFOR** и подпрограмм **BS**, **JS** не должно превышать пятнадцать (ограничение связано с глубиной системного стека);
- Цикл **DO**, **DOFOR** не может оканчиваться на инструкцию программного управления (**DO**, **DOFOR**, **B**, **J**, **BD**, **JD**, **BS**, **JS**);
- Цикл **DO**, **DOFOR** не может оканчиваться на ту же инструкцию, что и вложенный в него цикл.

Регистры и стеки DSP-ядра рассматриваются на странице "Регистры DSP".
Описание набора команд ядра DSP приведено в книге "**DSP Instructions Set**".

3.16. Параллельные операции

В программе для ядра DSP возможно параллельное (одновременное) выполнение двух вычислительных операций и двух операций пересылок в рамках одной инструкции.

Для того, чтобы две вычислительные операции могли выполняться параллельно, они должны выполняться на разных операционных устройствах. Например, операции **AND** и **OR** не могут выполняться параллельно, так как обе они выполняются на логическом устройстве **LU**.

Параллельно же могут выполняться операции, одна из которых исполняется при помощи арифметико-логического устройства **ALU** (*OP1*), а другая - при помощи умножителя-сдвигателя **MS** (*OP2*).

Кроме параллельных вычислительных операций, в инструкцию также могут входить (в зависимости от формата) до двух операций пересылки. Сводная таблица вариантов параллельно исполняемых вычислительных операций и операций пересылок приведена на странице "Форматы инструкций Ассемблера".

Пример:

```
ASRL   14,R8,R8   ADDSUBL  R14,R10,R14  (A2),R16
```

В данном примере параллельно исполняются *арифметический сдвиг*, операция "*сложение-вычитание*", а также чтение из *памяти данных*.

Подробное описание форматов инструкций ядра DSP, а также список поддерживаемых форматов для каждой инструкции приведены в книге "**DSP Instructions Set**".

3.17. Система команд

Команды (операции) DSP-ядра по своему действию делятся на три больших группы:

1) **Вычислительные команды**
   Каждая из команд данного типа производит некоторое действие над данными, хранящимися в регистровом файле (**RF**) DSP-ядра, и полученные результаты помещаются также в регистры **RF**. Кроме того, формируется набор признаков результата, который помещается в регистр **CCR**.
2) **Команды пересылок**
   При помощи команд пересылок производится обмен данными между регистрами, регистрами и памятью данных, либо загрузка непосредственных данных в регистры.
3) **Команды программного управления**
   Команды программного управления производят изменения в последовательности исполнения инструкций DSP-ядра. С их помощью организуются программные переходы, циклы и т. д.

Рассмотрим все эти три группы команд подробнее:

Вычислительные команды делятся по характеру исполняемой операции и по форматам обрабатываемых данных на более мелкие группы команд, приведенные в таблицах 2.17-2.25.

При определении мнемоники команд приняты следующие соглашения:

- команды, работающие в длинном формате, имеют на конце суффикс "**L**"(Long);
- команды, работающие в комплексных форматах, имеют на конце суффикс "**X**"(compleX);
- команды, работающие в формате с плавающей точкой, имеют префикс "**F**"(Float).

Имеются некоторые исключения из приведенных выше правил, в частности, для команд, работающих одновременно с различными форматами данных.

Таблица 2.17. Команды сложения/вычитания в форматах с фиксированной точкой.

Мнемоники команд	Содержание команды (формат данных)
	Сложение
ADC	Сложение с переносом (Short)
ADCL	Сложение с переносом (Long)
ADC16L	Сложение смешанное
ADD	Сложение (Short)
ADDL	Сложение (Long)
ADDLR	Сложение (Long) с округлением
ADDLRTR	Сложение (Long) с округлением и преобразованием формата (в Short)
ADDX	Сложение комплексное (X16)
AD1	Сложение и инкремент (Short)
	Вычитание
SBC	Вычитание с переносом (Short)
SBCL	Вычитание с переносом (Long)
SUB	Вычитание (Short)
SUBL	Вычитание (Long)
SUBLR	Вычитание (Long) с округлением
SUBLRTR	Вычитание (Long) с округлением и преобразованием формата (в Short)
SUBX	Вычитание комплексное (X16)
	Сложение-вычитание
ADDSUB	Сложение-вычитание (Short)
ADDSUBL	Сложение-вычитание (Long)
ADDSUBX	Сложение-вычитание (X16)
ASH	Сложение и вычитание двух пар чисел (Short)
SAH	Сложение и вычитание двух пар чисел (Short)
	Инкремент/декремент
DEC	Декремент (Short)
DECL	Декремент (Long)
INC	Инкремент (Short)
INCL	Инкремент (Long)

Таблица 2.18. Команды умножения в форматах с фиксированной точкой.

Мнемоники команд	Содержание команды (формат данных)
	Умножение
MPF	Умножение дробное со знаком (Short)
MPF2	Парное умножение дробное со знаком (Short)
MPF2S	Парное умножение дробное со знаком (Short), с перестановкой сомножителей
MPSS	Умножение целое со знаком (Short)
MPUU	Умножение целое без знака (Short)
MPX	Умножение дробное комплексное (X8), второй операнд - комплексно-сопряженный
MPYL	Умножение целое со знаком (Long)
	Умножение с накоплением (MAC)
MAC	Умножение целое со знаком (Short) и накопление (в формате Double)
MACL	Умножение целое со знаком (Long) и накопление (в формате Double)
MACX	Умножение дробное комплексно-сопряженное (X8) и целочисленное (X16)
MAC2	Парное умножение целое (Short) и накопление двух результатов (в формате Long)
SAC2	Парное накопление (в формате Long) со знаком

Таблица 2.19. Команды сдвига в форматах с фиксированной точкой.

Мнемоники команд	Содержание команды (формат данных)
	Арифметический сдвиг
ASL	Арифметический сдвиг влево (Short)
ASLL	Арифметический сдвиг влево (Long)
ASLX	Арифметический сдвиг влево (X16)
ASR	Арифметический сдвиг вправо (Short)
ASRL	Арифметический сдвиг вправо (Long)
ASRX	Арифметический сдвиг вправо (X16)
	Логический сдвиг
LSL	Логический сдвиг влево (Short)
LSLL	Логический сдвиг влево (Long)
LSLX	Логический сдвиг влево (X16)
LSR	Логический сдвиг вправо (Short)
LSRL	Логический сдвиг вправо (Long)
LSRX	Логический сдвиг вправо (X16)
	Циклический сдвиг на один разряд
ROL	Циклический сдвиг на один разряд влево (Short)
ROLL	Циклический сдвиг на один разряд влево (Long)
ROR	Циклический сдвиг на один разряд вправо (Short)
RORL	Циклический сдвиг на один разряд вправо (Long)

Таблица 2.20. Другие арифметические команды в форматах с фиксированной точкой.

Мнемоники команд	Содержание команды (формат данных)
	Обнуление регистра
CLR	Обнуление (очистка) регистра (Short)
CLRL	Обнуление (очистка) регистра (Long)
	Транзит
TR	Транзит (Short)
TRL	Транзит (Long)
	Изменение знака
NEG	Изменение знака (Short)
NEGL	Изменение знака (Long)
	Абсолютное значение
ABS	Абсолютное значение (Short)
ABSL	Абсолютное значение (Long)
	Сравнение
CMP	Сравнение (Short)
CMPL	Сравнение (Long)
CMPM	Сравнение модулей (Short)
CMPML	Сравнение модулей (Long)
CS2	Парная операция выбора большего из двух чисел (Short) с фиксацией бита выбора
	Максимум-минимум
MAX	Выбор большего числа (Short)
MAXL	Выбор большего числа (Long)
MIN	Выбор меньшего числа (Short)
MINL	Выбор меньшего числа (Long)
MAXM	Выбор числа с большим модулем (Short)
MAXML	Выбор числа с большим модулем (Long)
MINM	Выбор числа с меньшим модулем (Short)
MINML	Выбор числа с меньшим модулем (Long)
	Определение признаков операнда
TST	Определение признаков операнда (Short)
TSTL	Определение признаков операнда (Long)
TSTX	Определение признаков операнда (X16)

Таблица 2.21. Округление, преобразование форматов, упаковка/распаковка.

Мнемоники команд	Содержание команды (формат данных)
	Округление
RNDL	Округление
	Преобразование формата
FTR	Преобразование формата
FTRL	Преобразование формата
FTRFL	Преобразование формата
	Упаковка/распаковка
PACK	Упаковка (Short)
PACKL	Упаковка (Long)
DISPX	Распаковка (целочисленная) X8 в X16
DISPFX	Распаковка (дробная) X8 в X16

Таблица 2.22. Логические команды, операции с битами и битовыми полями.

Мнемоники команд	Содержание команды (формат данных)
	Логические команды
AND	Логическое умножение (Short)
ANDL	Логическое умножение (Long)
ANDC	Логическое умножение с инверсией (Short)
ANDCL	Логическое умножение с инверсией (Long)
ANDI	Инверсия логического умножения (Short)
EOR	Логическое исключающее ИЛИ (Short)
EORL	Логическое исключающее ИЛИ (Long)
NOT	Логическое отрицание (Short)
NOTL	Логическое отрицание (Long)
OR	Логическое сложение (Short)
ORC	Логическое сложение с инверсией (Short)
ORL	Логическое сложение (Long)
ORCL	Логическое сложение с инверсией (Long)
ORI	Инверсия логического сложения (Short)
	Определение параметра денормализации
PDN	Определение параметра денормализации (Short)
PDNL	Определение параметра денормализации (Long)
PDNX	Определение параметра денормализации (X16)
	Операции с битами и битовыми полями
BTST	Проверка разряда (Short)
BTSTL	Проверка разряда (Long)
MSKG	Формирование маски (Short)
MSKGL	Формирование маски (Long)
INSL	Вставка разрядов (Long)
SWL	Перестановка (Long)
	Сложение бит
SMB	Сложение бит (Short)
SMBL	Сложение бит (Long)

Таблица 2.23. Команды для обработки данных в формате с плавающей точкой 24E8.

Мнемоники команд	Содержание команды (формат данных)
FADD	Сложение
FSUB	Вычитание
FAS	Сложение и вычитание
FTST	Определение признаков операнда
FMPY	Умножение
FINT	Округление к ближайшему целому (четному), запись в формате плавающей точки 24E8
FLOOR	Округление к ближайшему целому снизу, запись в формате плавающей точки 24E8
CVFI	Преобразование формата: формат 24E8 в 32-разрядное целое со знаком в дополнительном коде
CVIF	Преобразование формата: 32-разрядное целое в дополнительном коде со знаком в формат 24E8
FIN	Первая итерация обратной величины
FINR	Первая итерация обратной величины от квадратного корня

Таблица 2.24. Команды для обработки данных в формате с плавающей точкой 32E16.

Мнемоники команд	Содержание команды (формат данных)
CMPE	Сравнение экспонент
ASRLE	Правый сдвиг мантиссы в соответствии с битом E
PDNLE	Измерение параметра денормализации мантиссы
PDNE	Измерение параметра денормализации мантиссы
CVEF	Преобразование формата: формат 32E16 в 24E8
CVFE	Преобразование формата: формат 24E8 в 32E16

Команды пересылок

При помощи **команд пересылок** производится обмен данными между регистрами, регистрами и памятью данных, либо загрузка непосредственных данных в регистры.

Для всех видов команд пересылок используется одна и та же мнемоническая запись – **MOVE**, однако форматы и коды инструкций зависят от типа пересылки и ее параметров.

Команды программного управления производят изменения в последовательности исполнения инструкций DSP-ядра. С их помощью организуются программные переходы, циклы, вход в подпрограмму и выход из нее, останов DSP-ядра.

Команды программных переходов **B**, **BD**, **BS**, **J**, **JD**, **JS** являются условными, остальные команды - безусловные.

Таблица 2.25. Команды программного управления.

Мнемоники команд	Содержание команды
DO	Оператор цикла
DOFOR	Оператор бесконечного цикла
ENDDO	Окончание цикла
B	Ветвление программы
BD	Ветвление программы (отложенное)
BS	Вызов подпрограммы
J	Программный переход
JD	Программный переход (отложенный)
JS	Вызов подпрограммы
RTS	Возврат из подпрограммы
NOP	Пустая операция
STOP	Останов

Подробнее команды DSP-ядра и форматы обрабатываемых ими данных рассматриваются в книге "**DSP Instructions Set**".

4. Порты ввода-вывода

4.1. Введение

В данной главе рассматриваются следующие порты ввода-вывода ИМС "*МУЛЬТИКОР*":

- Универсальный асинхронный порт (UART);
- Порты обмена последовательным кодом (SPort);
- Линковые порты (LPort).

4.2. Универсальный асинхронный порт UART

4.2.1. Введение

В данном разделе рассматриваются характеристики универсального асинхронного порта **UART**, а также способы работы с ним. Глава включает в себя:

- Общие характеристики порта **UART**;
- Описание регистров **UART**;
- Описание программируемого генератора скорости обмена;
- Метод работы с FIFO по прерыванию;
- Метод работы с FIFO по опросу;
- Способы программирования порта UART;
- Пример программы, использующей порт UART.

4.2.2. Общие характеристики

Универсальный асинхронный порт **UART** имеет следующие характеристики:

- по архитектуре совместим с *UART 16550*;
- частота приема и передачи данных – от 50 до 1 M baud;
- **FIFO** для приема и передачи данных имеют объем по 16 байт;
- полностью программируемые параметры последовательного интерфейса: длина символа от 5 до 8 бит; генерация и обнаружение бита четности; генерация стопового бита длиной 1, 1/2 или 2 бита;
- диагностический режим внутренней петли;
- эмуляция символьных ошибок;
- функция управления модемом (CTS, RTS, DSR, DTR, RI, DCD).

Структурная схема порта **UART** приведена на рисунке 3.1:

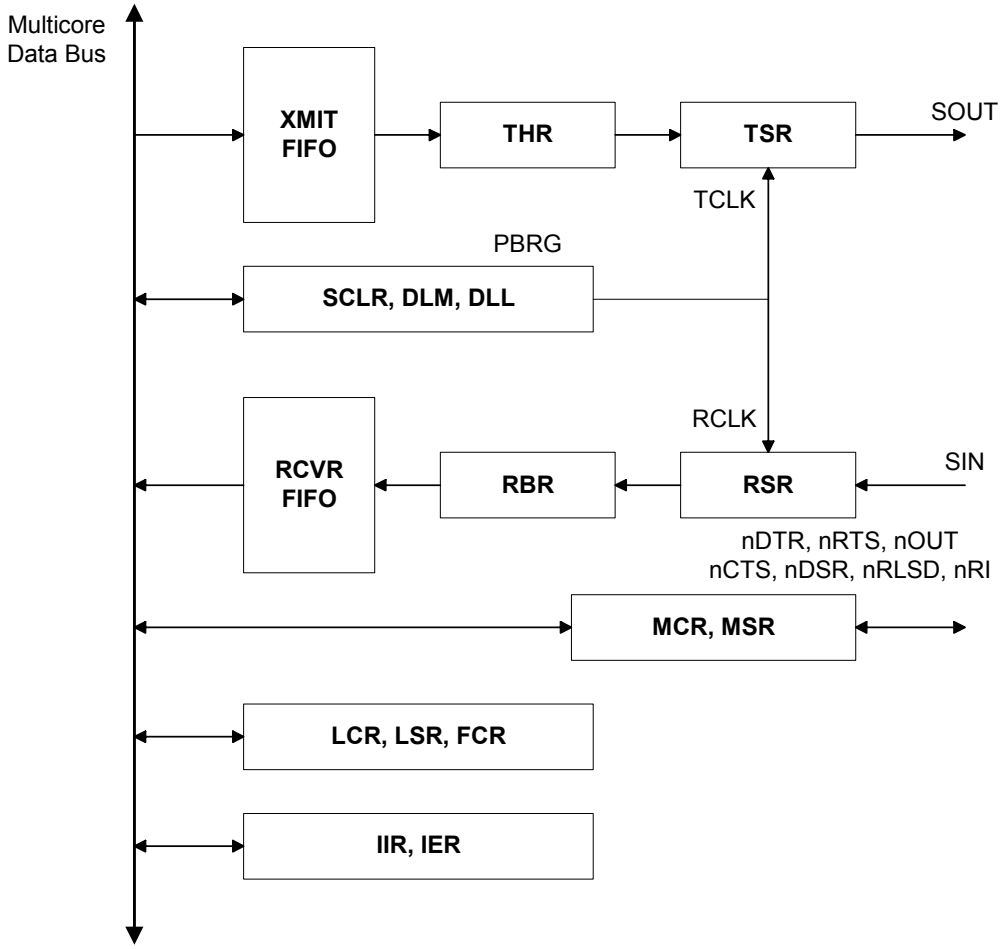


Рисунок 3.1. Структурная схема порта UART.

Передаваемые данные записываются в регистр **THR**, а затем аппаратно переписываются в передающий сдвигающий регистр (**TSR**), если он пуст. После этого в регистр **THR** может быть записаны следующие данные.

После приема данных в приемный сдвигающий регистр (**RSR**) данные переписываются в регистр **RBR**.

Описание регистров универсального асинхронного порта приведено в разделе "Регистры UART".

4.2.3. Регистры UART

В таблице 3.1 приведен перечень регистров универсального асинхронного порта **UART**:

Таблица 3.1. Регистры UART.

Условное обозначение регистра	Название регистра	Смещение	Доступ (R - чтение, W - запись)
RBR	Приемный буферный регистр	0 (DLAB=0)	R
THR	Передающий буферный регистр	0 (DLAB=0)	W
IER	Регистр разрешения прерываний	1 (DLAB=0)	R/W
IIR	Регистр идентификации прерывания	2	R
FCR	Регистр управления **FIFO**	2	W
LCR	Регистр управления линией	3	R/W
MCR	Регистр управления модемом	4	R/W
LSR	Регистр состояния линии	5	R
MSR	Регистр состояния модема	6	R/W
SPR	Регистр *Scratch Pad*	7	R/W
DLL	Регистр делителя младший	0 (DLAB=1)	R/W
DLM	Регистр делителя старший	1 (DLAB=1)	R/W
SCLR	Регистр предделителя (*scaler*)	5	W

Регистр LCR

Формат регистра управления линией **LCR** приведен в таблице 3.2:

Таблица 3.2. Формат регистра LCR.

Номер бита	Условное обозначение	Назначение
1:0	WLS (Word Length Select)	Количество бит данных в передаваемом символе: 00 -5 бит, 01 -6 бит, 10 -7 бит, 11 -8 бит.
2	STB (Number Stop Bits)	Количество стоп-бит: 0 - 1 стоп-бит, 1 - 2 стоп-бита (для 5-битного символа стоп-бит имеет длину 1,5 бита). Приемник анализирует только первый стоп бит.
3	PEN (Parity Enable)	Разрешение генерации (передатчик) или проверки (приемник) контрольного бита: 1 – контрольный бит (паритет или постоянный) разрешен, 0 – запрещен.
4	EPS (Even Parity Select)	Выбор типа контроля (при PEN=1): 0 – нечетность, 1 – четность.
5	STP (Stick Parity)	Принудительное формирование бита паритета: 0 – контрольный бит генерируется в соответствии с паритетом выводимого символа, 1 – постоянное значение контрольного бита: при EPS=1 - нулевое, при EPS=0 – единичное.
6	SBC (Set Break Control)	Формирование обрыва линии: 0 – нормальная работа; 1 – на выходе SOUT устанавливается низкий уровень (Spacing level). Это влияет только на выход SOUT, а не на логику передачи символа.
7	DLAB (Divisor Latch Access bit)	Управление доступом к регистрам: 0 – разрешен доступ к регистрам RBR, THR, IER; 1 – разрешен доступ к регистрам DLL, DLM

Исходное состояние регистра **LCR** – нули.

Бит SBC используется как признак «Внимание» для приемного терминала, подключенного к выходу **UART**. Для того чтобы не был передан ошибочный символ, при использовании бита SBC, необходимо выполнять следующую последовательность действий:

- Загрузить в регистр **THR** все нули по признаку `THRE=1`;
- Установить `SBC=1` по следующему `THRE=1`;
- Дождаться `TEMT=1`.

Для восстановления нормальной передачи необходимо установить `SBC=0`.

Регистр FCR

Формат регистра управления **FIFO** приведен в таблице 3.3:

Таблица 3.3. Формат регистра FCR.

Номер бита	Условное обозначение	Назначение
0	FEWO (FIFO Enable)	Разрешение работы XMIT и RCVR FIFO: 0 – символьный режим; 1 – режим FIFO. При изменении состояния этого бита, данные из FIFO, не удаляются. Запись в биты RFR, TFR, RFTL выполняется, если FEWO=1.
1	RFR (Receiver FIFO Reset)	Установка RCVR FIFO в исходное состояние. Регистр RSR не обнуляется. После записи 1 в этот бит он автоматически сбрасывается.
2	TFR (Transmitter FIFO Reset)	Установка XMIT FIFO в исходное состояние. Регистр TSR не обнуляется. После записи 1 в этот бит он автоматически сбрасывается.
5:3	-	Резерв
7:6	RFTL (RCVR FIFO Trigger Level)	Порог заполнения RCVR FIFO (в байтах), при котором формируется прерывание: 00 – 1; 01 – 4; 10 – 8; 11 – 14.

Исходное состояние регистра **FCR** - нули.

Регистр LSR

Формат регистра управления линией приведен в таблице 3.4:

Таблица 3.4. Формат регистра LSR.

Номер бита	Условное обозначение	Назначение
0	RDR (Receiver Data Ready)	Готовность данных. Устанавливается после приема символа данных и передачи его в регистр RBR или FIFO. Сбрасывается после чтения регистра RBR (в символьном режиме) или чтения всего содержимого RCVR FIFO (в режиме FIFO)
1	OE (Overrun Error)	Ошибка переполнения. Устанавливается, если содержимое регистра RBR не было прочитано, в сдвигающий регистр принят следующий символ и начат прием очередного символа. При этом новый символ записывается в сдвигающий регистр вместо старого. В режиме FIFO устанавливается, если после перехода порогового (trigger) уровня FIFO заполнено до конца, во входной сдвигающий регистр полностью принят следующий символ и начат прием очередного символа. При этом в FIFO ничего не передается. Бит сбрасывается при чтении содержимого регистра LSR.
2	PE (Parity Error)	Ошибка контрольного бита (паритета или фиксированного). В режиме FIFO этот бит показывает на ошибку в символе, находящемся наверху FIFO. Бит сбрасывается при чтении содержимого регистра LSR.

3	FE (Framing Error)	Ошибка кадра. Устанавливается, если стоп-бит равен нулю (Spacing level). В режиме FIFO этот бит показывает на ошибку в символе, находящемся наверху FIFO. После этой ошибки UART пересинхронизируется. Бит сбрасывается при чтении содержимого регистра LSR.
4	BI (Break Interrupt)	Обрыв линии. Устанавливается, если вход приема данных находится в состоянии 0 (Spacing level) не менее чем время передачи всего символа. В режиме FIFO этот бит показывает на ошибку в символе, находящемся наверху FIFO. При возникновении этой ситуации, в FIFO загружается только один нулевой символ. Прием следующих символов разрешается после того, как вход приема данных перейдет в единичное состояние (Marking state) и будет принят действительный стартовый бит. Бит сбрасывается при чтении содержимого регистра LSR.
5	THRE (Transmitter Holding Register Empty)	Передающий буферный регистр пуст. Показывает, что UART готов принять следующий символ для передачи. Устанавливается, когда содержимое регистра THR передается в передающий сдвигающий регистр. Одновременно с этим генерируется прерывание THREI, если оно разрешено. Бит сбрасывается при записи символа в регистр THR. В режиме FIFO этот бит устанавливается, когда XMIT FIFO пусто, и сбрасывается, если в XMIT FIFO записывается хотя бы один символ.
6	TEMT (Transmitter Empty)	Передатчик пуст. Устанавливается, если регистры THR и TSR пусты. Имеет нулевое состояние, если хотя бы один из регистров THR и TSR не пуст. В режиме FIFO этот бит устанавливается, если нет символов ни в XMIT FIFO, ни в регистре TSR.
7	EIRF (Error in RCVR FIFO)	Наличие хотя бы одного признака ошибки в FIFO. В символьном режиме этот бит всегда равен нулю. Бит сбрасывается при чтении содержимого регистра LSR, если в FIFO нет больше признаков ошибок.

Исходное состояние битов THRE, TEMT – 1, остальных – 0.

Установка битов OE, PE, FE, BI приводит к формированию прерывания по состоянию входа приема данных (*Receiver Line Status Interrupt*), если это прерывание разрешено.

Регистр IER

Формат регистра разрешения прерываний приведен в таблице 3.5:

Таблица 3.5. Формат регистра IER.

Номер бита	Условное обозначение	Назначение
0	ERBI	Разрешение прерывания по наличию принятых данных (RDAI), а также по таймауту (CTI)
1	ETBEI	Разрешение прерывания по отсутствию данных в регистре THR (THREI)
2	ERLSI	Разрешение прерывания по статусу приема данных (RLSI)
3	EMSI	Разрешение прерывания по статусу модема (MSI)
7:4	-	Резерв

Исходное состояние регистра **IER** - нули.

Регистр IIR
Формат регистра идентификации прерывания приведен в таблице 3.6:

Таблица 3.6. Формат регистра IIR.

Номер бита	Условное обозначение	Назначение
0	IP (Interrupt Pending)	Признак наличия прерывания: 0 – есть прерывание; 1 – нет прерывания.
3:1	IID[2:0]	Код идентификации прерывания в соответствии с таблицей 2
5:4	-	Резерв
7:6	FE	Признак разрешения работы RCVR и XMIT FIFO

Исходное состояние бита IP - 1, остальных битов - 0.
В таблице 3.7 приводится возможные значения поля IID:

Таблица 3.7. Значения поля IID.

Код Поля ID[2:0]	Уровень Приоритета (1 – наивысший)	Тип Прерывания	Причина прерывания	Условие Сброса Прерывания
011	1	Статус приема данных (RLSI – Receiver Line Status Interrupt)	OE - Overrun Error; PE - Parity Error; FE - Framing Error; BI - Break Interrupt.	Чтение содержимого регистра LSR. Чтение из FIFO символа, по которому сформировано это прерывание. Обнуление FIFO.
010	2	Наличие принятых данных (RDAI – Received Data Available Interrupt)	Наличие данных в регистре RBR или достижение заданного порога FIFO	Чтение содержимого регистра RBR. Считывание данных из FIFO до уровня ниже порогового.
110	2	Таймаут (CTI – Character Timeout Interrupt)	С момента приема последнего символа в RCVR FIFO прошло время, равное длительности передачи 4-х символов и не было ни чтения FIFO, ни приема очередного символа.	Чтение содержимого регистра RBR. Прием очередного символа. Сброс FIFO.
001	3	Регистр THR пуст (THREI – Transmitter Holding Register Empty Interrupt)	Регистр THR пуст	Запись символа в регистр THR
000	4	Статус модема (MSI – Modem Status Interrupt)	Изменение состояния сигналов на входах порта nCTS, nDSR, nRI, nDCD	Чтение содержимого регистра MSR.

Регистр MCR

Формат регистра управления модемом приведен в таблице 3.8:

Таблица 3.8. Формат регистра MCR.

Номер бита	Условное обозначение	Назначение
0	DTR	Управление выходом nDTR: 0 – на выходе высокий уровень; 1 – на выходе низкий уровень.
1	RTS	Управление выходом nRTS: 0 – на выходе высокий уровень; 1 – на выходе низкий уровень.
2	Out 1	Управление выходом OUT1: 0 – на выходе высокий уровень; 1 - на выходе низкий уровень.
3	Out 2	Управление выходом OUT1: 0 – на выходе высокий уровень; 1 - на выходе низкий уровень.
4	LOOP	Режим петли. Используется для тестирования UART. При установке этого бита в 1 выполняется следующее: На выходе SOUT UART устанавливается высокий уровень; Вход SIN UART отключается от внешнего вывода; Выход регистра TSR подключается к входу регистра RSR; На выходах nDTR, nRTS, nOUT1, nOUT2 устанавливаются высокие уровни; Входы nCTS, nDSR, nDCD, nRI UART отключаются от внешних выводов; Выходы разрядов DTR, RTS, Out 1, Out 2 регистра MCR подключаются к входам разрядов DSR, CTS, RI, DCD регистра MSR соответственно. В режиме петли передаваемые данные немедленно принимаются. В режиме петли все прерывания формируются как обычно.
7:5	-	Резерв

Исходное состояние регистра **MCR** - нули.

Регистр MSR

Формат регистра состояния модема приведен в таблице 3.9:

Таблица 3.9. Формат регистра MSR.

Номер бита	Условное обозначение	Назначение
0	DCTS	Признаки любого изменения состояния входного сигнала CTS. Бит устанавливается в единичное состояние, если сигнал CTS изменил свое состояние после последнего считывания содержимого регистра MSR. Одновременно с этим формируется прерывание MSI, если оно разрешено. Бит сбрасывается при чтении содержимого регистра MSR.
1	DDSR	Признаки любого изменения состояния входного сигнала DSR. Бит устанавливается в единичное состояние, если сигнал DSR изменил свое состояние после последнего считывания содержимого регистра MSR. Одновременно с этим формируется прерывание MSI, если оно разрешено. Бит сбрасывается при чтении содержимого регистра MSR.
2	TERI	Признаки перехода входного сигнала RI с низкого уровня на высокий уровень. Бит устанавливается в единичное состояние, если сигнал RI изменил свое состояние после последнего считывания содержимого регистра MSR. Одновременно с этим формируется прерывание MSI, если оно разрешено. Бит сбрасывается при чтении содержимого регистра MSR.
3	DRLSD	Признаки любого изменения состояния входного сигнала RLSD. Бит устанавливается в единичное состояние, если сигнал RLSD изменил свое состояние после последнего считывания содержимого регистра MSR. Одновременно с этим формируется прерывание MSI, если оно разрешено. Бит сбрасывается при чтении содержимого регистра MSR.
4	CTS	Состояние сигнала на входе nCTS: 0 – на входе высокий уровень; 1 – на входе низкий уровень.
5	DSR	Состояние сигнала на входе nDSR: 0 – на входе высокий уровень; 1 – на входе низкий уровень.
6	RI	Состояние сигнала на входе nRI: 0 – на входе высокий уровень; 1 – на входе низкий уровень.
7	RLSD	Состояние сигнала на входе n RLSD: 0 – на входе высокий уровень; 1 – на входе низкий уровень.

Исходное состояние битов `3:0` регистра **MSR** – нули. Биты `7:4` следуют за инверсией состояния соответствующих входных сигналов.

4.2.4. Программируемый генератор скорости обмена

В универсальном асинхронном порте **UART** есть программируемый генератор скорости обмена данными (**PBRG** – *Programmable Baud Rate Generator*). Он состоит из 8-разрядного предделителя и 16-разрядного основного делителя частоты. На вход предделителя поступает тактовая частота, на которой работает шина данных **UART** (**CLK**). Выходная частота предделителя поступает на вход основного делителя. Выходная частота генератора **PBRG** в 16 раз больше частоты обмена последовательными данными.

Коэффициент деления предделителя задается 8-разрядным регистром **SCLR** таким образом, чтобы частота на выходе предделителя соответствовала одной из трех стандартных частот (см. таблицу 1, таблицу 2, таблицу 3). Значение частоты на выходе предделителя равно **CLK**/(**SCLR** + 1). Коэффициент деления основного делителя задается 16-разрядным регистром, который является конкатенацией регистров **DLM** и **DLL**. Для получения одной из стандартных частот передачи значение этого коэффициента выбирается из таблиц 3.10, 3.11 и 3.12.

Таблица 3.10. Коэффициенты деления (1).

Требуемая скорость обмена (бод)	Делитель для получения частоты 16 * бод	Ошибка в процентах Разница между требуемой и действительной скоростью
50	2304	-
75	1536	-
110	1047	0.026
134.5	857	0.058
150	768	-
300	384	-
600	192	-
1200	96	-
1800	64	-
2000	58	0.690
2400	48	-
3600	32	-
4800	24	-
7200	16	-
9600	12	-
19200	6	-
38400	3	-
56000	2	2.860

Таблица 3.11. Коэффициенты деления (2).

Требуемая скорость обмена (бод)	Делитель для получения частоты 16 * бод	Ошибка в процентах Разница между требуемой и действительной скоростью
50	3840	-
75	2560	-
110	1745	0.026
134.5	1428	0.034
150	1280	-
300	640	-
600	320	-
1200	160	-
1800	107	0.312
2000	96	-
2400	80	-
3600	53	0.628
4800	40	-
7200	27	1.230
9600	20	-
19200	10	-
38400	5	-
56000	3	14.285

Таблица 3.12. Коэффициенты деления (3).

Требуемая скорость обмена (бод)	Делитель для получения частоты 16 * бод	Ошибка в процентах Разница между требуемой и действительной скоростью
50	10000	-
75	6667	0.005
110	4545	0.010
134.5	3717	0.013
150	3333	0.010
300	1667	0.020
600	833	0.040
1200	417	0.080
1800	277	0.080
2000	250	-
2400	208	1.160
3600	139	0.080
4800	104	1.160
7200	69	0.644
9600	52	1.160
19200	26	1.160
38400	13	1.160
56000	9	0.790
128000	4	2.344
256000	2	2.344

Период частот передачи и приема (**TCLK** и **RCLK**) **UART** вычисляется по формуле: **CLK**/(**SCLR** + 1) /((конкатенация содержимого регистров **DLM** и **DLL**) *16). Минимальная величина, которая может быть записана в регистры {**DLM,DLL**}, равна 1.

Исходное состояние регистров **DLL**, **DLM**, **SCLR** – нули.

4.2.5. Работа с FIFO по прерыванию

Если установлен режим работы с **FIFO** (EFWO=1 в регистре FCR) и разрешены прерывания по приему (бит ERI=1 в регистре IER), то в процессе приема:

- формируется прерывание, если число символов в **RCVR FIFO** достигло запрограммируего порога. Это прерывание сбрасывается, если при чтении из **FIFO** число символов оставшихся в нем, станет меньше запрограммированного порога;
- одновременно с этим в регистре IIR устанавливается индикатор наличия принятых данных **RDAI**. Индикатор обнуляется, при чтении из **FIFO** до снижения запрограммированного порога;
- может возникнуть прерывание по статусу приема данных (**RLSI**), приориет которого выше, чем **RDA**.
- бит RDR в регистре LSR устанавливается в момент передачи символа из регистра **RSR** в **RCVR FIFO**. Этот бит обнуляется при считывании из **FIFO** всех символов данных.

Если установлен режим работы с **FIFO** (EFWO=1 в регистре FCR) и разрешены прерывания по приему (ERI=1 в регистре IER), то генерируется прерывание по таймауту, если с момента приема последнего символа в **RCVR FIFO** прошло время, равное длтельности передачи 4-х символов и за это время не было:

- ни чтения **RCVR FIFO**;
- ни приема в **RCVR FIFO** очередного символа.

При 12-битном символе и скорости передачи 300 бод, прерывание по этой причине возникнет через 160 мс.

При возникновении прерывания по таймауту оно обнуляется при считывании символа из **RCVR FIFO**. При этом обнуляется и таймер, генерирующий данное прерывание. Если прерывание по тайауту не возникло, то таймер таймаута обнуляется при приеме нового символа или при считывании символа из **RCVR FIFO**.

Если установлен режим работы с **FIFO** (EFWO=1 в регистре FCR) и разрешены прерывания по передаче данных (бит ETI=1 в регистре IER), то генерируется прерывание по передаче следующим образом:

- формируется прерывание **THREI**, если **XMIT FIFO** пусто. Это прерывание обнуляется, как только выполняется запись символа в регистр **THR** (при приеме данного прерывания в **XMIT FIFO** можно записать от 1 до 16 символов) или считывается содержимое регистра IIR;
- индикатор TEMT в регистре LSR установится в единичное состояние через время равное длительности одного символа минус последний стоп бит, после установки

THRE=1. Первое прерывание по передаче (если оно разрешено) формируется немедленно после установки FEWO=1.

4.2.6. Работа с FIFO по опросу

Если установлен режим работы с **FIFO** (EFWO=1 в регистре FCR) и запрещены прерывания, то обмен данными выполняется по опросу, а управление **FIFO** приема и передачи (**RCVR**, **XMIT**) выполняется раздельно.

В этом режиме опрос состояния **RCVR** и **XMIT FIFO** осуществляется программно, посредством считывания содержимого регистра LSR:

- бит RDR=1, пока есть данные в **RCVR FIFO**;
- биты OE, PE, FE, BI указывают на ошибки. Эти ошибки обрабатываются так же, как и при работе по прерыванию;
- бит THRE=1, если **XMIT FIFO** пусто;
- бит TEMT=1, если в **XMIT FIFO** и **TSR** нет данных.

При работе по опросу нет индикации таймаута и факта достижения порога **RCVR FIFO**. Однако оба **RCVR** и **XMIT FIFO** могут хранить символы данных.

4.2.7. Программирование порта UART

Чтобы запрограммировать универсальный асинхронный порт **UART** на передачу/прием данных необходимо:

- Установить делители частоты **DLL** и **DLM**;
- Установить в регистре управления линией LCR количество бит в передаваемом символе и прочие параметры приема/передачи;
- Выбрать режим приема/передачи - посимвольный, **FIFO** по прерыванию, **FIFO** по опросу (для отладки - режим петли);
- В случае работы с **FIFO** по прерыванию, описать соответствующие обработчики исключений по прерыванию;
- При работе в посимвольном режиме или **FIFO** по опросу - опрашивать значения полей регистра LSR;
- В любом из перечисленных режимов передача осуществляется записью символа в регистр **THR**, а прием - чтением символа из регистра **RBR**.

Пример программы, осуществляющей передачу данных через порт **UART** в режиме работы с **FIFO** по опросу, рассматривается здесь.

4.2.8. Пример программы, использующей порт UART

Рассмотрим текст программы, использующей порт **UART** для передачи некоторого массива данных. Программа написана на языке *C* и включает в себя файл *main.c*, а также заголовочный файл *memory_12.h*, описывающий адресное пространство ИМС *MultiCore-12*.

Файл *main.c* содержит, собственно, код программы, использующей порт **UART**:

```
#include "memory_12.h"

main()
{
      int i;

      char Input[40];

      for (i=0; i<40; i++)
      {
          Input[i]=(char)i;
      }

      LCR=0x80;                              //разрешение доступа к DLL, DLM
      DLL=0x1;                               //установка делителя
      LCR=0x3;                               //длина передаваемого слова - 8 бит
      FCR=0x1;                               //включение режима FIFO

      /*
         циклическая передача сорока значений из массива Input[40]
         в режиме FIFO по опросу (прерывания выключены)
      */
      for (i=0; i<40; i+=4)
      {
          THR=Input[i];                      //передача очередных значений
          THR=Input[i+1];
          THR=Input[i+2];
          THR=Input[i+3];
          while (!(LSR & 0x20));             //ожидание установки бита THRE
      }

      while (1);
}
```

Приведенная программа осуществляет следующие действия:

- Устанавливает длину передаваемого слова равной 8 битам;
- Включает режим **FIFO**;
- Передает 40 байт данных (числа от 0 до 39).

Передача данных происходит циклически в режиме **FIFO**. При этом каждую итерацию в **XMIT FIFO** заносится 4 символа (в данном случае, 4 байта). Затем программа ожидает передачи этих значений (установки бита THRE регистра LSR), после чего переходит к передаче следующих четырех значений.

4.3. Порты обмена последовательным кодом (SPort)

4.3.1. Введение

В данной главе рассматриваются порты обмена последовательным кодом (**SPort**), их характеристики, а также способы работы с ними. Глава включает в себя:

- Общие сведения о портах обмена последовательным кодом;
- Регистры портов **SPort**;
- Описание одноканального режима работы;
- Описание режима петли;
- Описание многоканального режима работы;
- DMA и прерывания последовательных портов;
- Программирование последовательных портов;
- Пример программы, использующей порт SPort.

4.3.2. Общие сведения

Синхронный порт обмена последовательным кодом (**SPORT**) обеспечивает интерфейс ввода-вывода с широким набором периферийных устройств. Благодаря большому набору режимов тактовой и кадровой синхронизации этот порт обеспечивает реализацию многих коммуникационных протоколов и простое аппаратное сопряжение со стандартными конверторами и кодеками.
Порт имеет следующие основные характеристики:

- обеспечивает независимые функции передачи и приема данных;
- передает слова данных длиной от 3 до 32 бит, младшим или старшим битом в перед;
- используются двойная буферизация передаваемых данных и тройная буферизация принимаемых данных;
- частота последовательной передачи и приема и кадровая синхронизация может генерироваться самостоятельно или приниматься от внешних источников;
- выполняет однословный обмен данными с внутренней памятью по прерываниям под управлением **CPU**;
- выполняет обмен блоками данных при помощи **DMA**;
- имеет многоканальный режим работы для интерфейсов с временным разделением (**TMD**).

На рисунке 3.2 приведена структурная схема порта обмена последовательным кодом:

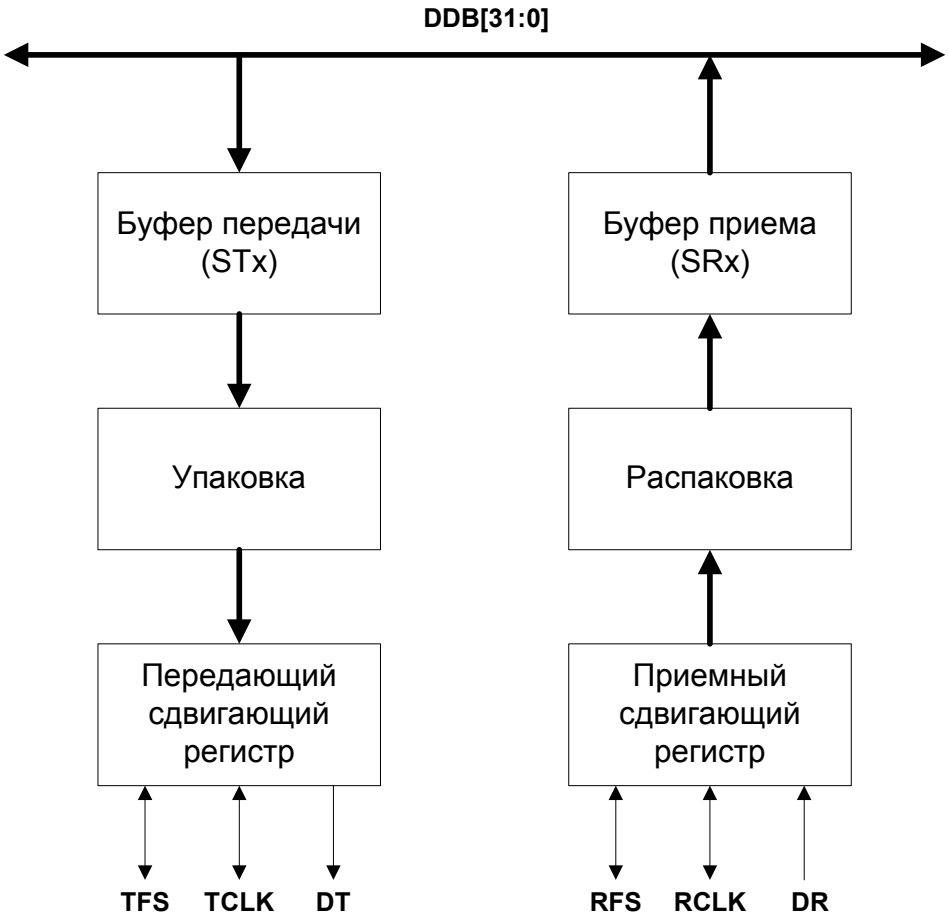


Рисунок 3.2. Структурная схема порта SPort.

Порт обмена последовательным кодом состоит из передающей и приемной частей.

Данные для передачи записываются в буфер STx. Затем данные автоматически переписываются в передающий сдвигающий регистр и выдвигаются на выходной вывод DT порта синхронно с тактовой частотой **TCLK**. Если используется кадровая синхронизация, то сигнал **TFS** индицирует начало передачи последовательного кода. Вывод DT находится в активном состоянии, если порт активизирован для передачи данных (бит `TEN=1` в регистре STCTL), или во время активного временного слота в многоканальном режиме.

При приеме данные вдвигаются в порт с вывода DR синхронно с частотой **RCLK**. Если используется кадровая синхронизация, то сигнал **RFS** сигнализирует о начале слова. Когда все слово вдвинуто, оно автоматически переписывается в буфер SRx.

4.3.3. Регистры SPort

В данном разделе рассматриваются регистры порта **SPort**. Перечень регистров приведен в таблице 3.13:

Таблица 3.13. Регистры SPort.

Условное обозначение регистра	Название регистра
STx	Буфер передачи данных
SRx	Буфер приема данных
STCTL	Регистр управления передачей данных
SRCTL	Регистр управления приемом данных
TDIV	Регистр коэффициентов деления при передаче данных
RDIV	Регистр коэффициентов деления при приеме данных
MTCS	Выбор канала передачи данных в многоканальном режиме
MRTS	Выбор канала приема данных в многоканальном режиме
KEYWD	Регистр кода сравнения
KEYMASK	Регистр маски сравнения
MRCE	Выбор канала для сравнения принимаемых данных

STx

Буфер передачи **STx** является буфером **FIFO** на два 32-разрядных слова: выходной регистр данных и выходной сдвигающий регистр. Два 32-разрядных слова могут быть сразу записаны в буфер **STx**, если он был до этого пуст.
Буфер передачи **STx** генерирует прерывание (бит **SPortT** в регистре QSTR) при следующих условиях:

- хотя бы один из битов TEN (`STCTL[0]`) или MCE (`SRCTL[23]`) имеют единичное состояние;
- выходной регистр данных пуст. Данный регистр пуст после начального включения или после передачи его содержимого в выходной сдвигающий регистр;
- соответствующий канал **DMA** не активизирован;
- данное прерывание не замаскировано.

Данное прерывание формируется в момент активизации последовательного порта на передачу при пустом буфере **STx**, или в момент переписи содержимого выходного регистра данных в выходной сдвигающий регистр. Прерывание, генерируемое буфером передачи, сигнализирует о том, что буфер **STx** готов принять следующее слово. Прерывание от буфера передачи сбрасывается в момент записи в него слова данных.

Бит состояния TUVF в регистре STCTL устанавливается, если сформирован сигнал кадровой синхронизации, а в буфер **Tx** не загружены новые данные. Этот бит может быть обнулен только посредством деактивизации данного порта (`TEN=0`). В многоканальном режиме бит TUVF всегда равен нулю.

SRx

Буфер приема **SRx** является буфером **FIFO** на три 32-разрядных слова: два входных регистра данных и входной сдвигающий регистр. Два принятых 32-разрядных слова могут храниться в буфере **SRx**, пока вдвигается третье слово. Третье слово затирает второе, если оно не было считано из буфера **SRx** (**CPU** или **DMA**). Если это произойдет, устанавливается бит состояния ROVF в регистре SRCTL. Этот бит может быть обнулен только посредством деактивизации данного порта. Почти три полных слова могут быть приняты до того, как бит ROVF может быть установлен. Бит ROVF используется в одноканальном и многоканальном режимах.

В момент окончания приема слова данных в буфер **SRx** генерируется прерывание, если оно разрешено, и соответствующий канал **DMA** не активизирован. Данное прерывание сбрасывается после чтения слова данных из буфера **SRx**.

STCTL

Формат регистра **STCTL** приведен в таблице 3.14:

Таблица 3.14. Формат регистра STCTL.

Номер разряда	Условное обозначение	Назначение
0	TEN	Разрешение передачи: 0 - передача запрещена; 1 - передача разрешена.
2:1	-	Резерв
3	TENDN	Выбор порядка передаваемых данных (*endian*): 0 - передача осуществляется старшими разрядами вперед (*little endian*); 1 - передача осуществляется младшими разрядами вперед (*big endian*).
8:4	TLEN	Длина передаваемого слова. Это поле определяет длину слова в битах (на единицу больше, чем код TLEN). Длина слова может быть от 3 битов (TLEN=2) до 32 битов (TLEN=31).
9	TPACK	Разрешение распаковки передаваемых данных: 0 - распаковка запрещена; 1 - разрешена распаковка 32-битного слова перед его передачей в буфере **Tx** в два слова, разрядность которых - 16 бит или меньше. Распаковка выполняется, если длина передаваемых слов данных меньше или равна 16 битам (определяется полем TLEN).
10	TICLK	Разрешение выдачи внутренней частоты передачи на вывод **TCLK**: 0 - вывод **TCLK** является входом; 1 - вывод **TCLK** является выходом и на него передается частота, период которой определяется полем TDIV[15:0].

11	-	Резерв
12	TCKRE	Выбор фронта частоты **TCLK**, по которому осуществляется опрос состояния передаваемых данных и импульса кадровой синхронизации: 0 - по отрицательному фронту; 1 - по положительному фронту.
13	TFSR	Требование кадровой синхронизации при передаче каждого слова: 0 - кадровая синхронизация требуется только при передаче первого слова; 1 - кадровая синхронизация требуется при передаче каждого слова.
14	ITFS	Разрешение выдачи внутреннего сигнала кадровой синхронизации **TFS**: 0 - вывод **TFS** является входом; 1 - вывод **TFS** является выходом и на него подается внутренний сигнал кадровой синхронизации, период которого определяется полем TDIV[31:16].
15	DITFS	Разрешение выдачи внутреннего кадрового синхроимпульса вне зависимости от наличия данных в буфере STx: 0 - запрещение; 1 - разрешение.
16	LTFS	Выбор активного уровня импульса кадровой синхронизации при передаче данных: 0 - импульс кадровой синхронизации имеет активный высокий уровень; 1 - импульс кадровой синхронизации имеет активный низкий уровень.
17	TLAFS	Выбор режима кадровой синхронизации при передаче данных: 0 - режим ранней кадровой синхронизации; 1 - режим поздней кадровой синхронизации.
19:18	-	Резерв
23:20	MFD	Выбор задержки начала передачи данных от импульса кадровой синхронизации при многоканальном режиме работы. При MFD=0 **TFS** и первый передаваемый бит совпадают. Максимальное значение поля - 15.
28:24	CHNL	Номер текущего канала при многоканальном режиме работы. Это поле содержит инкрементирующие счетчик по модулю **NCH**. Поле доступно только по чтению.

29	TUVF	Признак недозагрузки буфера STx. В многоканальном режиме не устанавливается (всегда ноль). Поле доступно только по чтению. Обнуляется только при TEN=0.
31:30	TXS	Состояние буфера STx: 00 - буфер пуст; 10 - буфер частично полон; 11 - буфер полон.

В многоканальном режиме работы биты TEN, TFSR, ITFS, TLAFS, DITFS должны иметь нулевое состояние.

Перед записью в регистр **STCTL** нового значения, его необходимо предварительно обнулить.

Исходное состояние регистра **STCTL** - все нули. При TEN=0, биты CHNL, TUVF обнуляются.

Признак TUVF устанавливается в одноканальном режиме работы, если сформирован сигнал **TFS** (самим портом или внешним источником), а буфер STx пуст. Если установлен режим генерации внутреннего **TFS** (ITFS=1), то, при DITFS=0, **TFS** формируется только в том случае, если буфер STx не пуст. То есть формирование **TFS** синхронизируется посредством записи данных в буфер STx. При DITFS=1 **TFS** формируется вне зависимости от наличия данных в буфере STx.

SRCTL

Формат регистра **SRCTL** приведен в таблице 3.15:

Таблица 3.15. Формат регистра SRCTL.

Номер разряда	Условное обозначение	Назначение
0	REN	Разрешение приема данных: 0 - прием запрещен; 1 - прием разрешен.
1	-	Резерв
2	DTYPE	Тип данных. Если длина принимаемых слов меньше 32 бит, то значащие биты размещаются в младших разрядах буфера **Rx**, а состояние старших разрядов определяется битом DTYPE следующим образом: 0 - старшие разряды имеют нулевое состояние (расширение нулями); 1 - старшие разряды имеют состояние старшего бита принятого слова (расширение знаком).

3	RENDN	Выбор порядка приема битов данных (*endian*): 0 - прием осуществляется старшими разрядами вперед (*little endian*); 1 - прием осуществляется младшими разрядами вперед (*big endian*).
8:4	RLEN	Длина принимаемого слова. Это поле определяет длину слова в битах (на единицу больше, чем код RLEN). Длина слова может быть от трех бит (RLEN=2) до 32 бит (RLEN=31).
9	RPACK	Разрешение упаковки принимаемых данных: 0 - запрещена упаковка; 1 - разрешена упаковка каждой пары принимаемых слов данных, длина которых меньше или равна 16 бит. Пара упаковывается в 32-битное слово перед записью в буфер SRx.
10	RICLK	Разрешение выдачи внутренней частоты передачи на вывод **RCLK**: 0 - вывод **RCLK** является входом; 1 - вывод **RCLK** является выходом, и на него подается частота, период которой определяется полем RDIV[15:0].
11	-	Резерв
12	RCKRE	Выбор фронта частоты **RCLK**, по которому осуществляется опрос состояния передаваемых данных и импульса кадровой синхронизации: 0 - по отрицательному фронту; 1 - по положительному фронту.
13	RFSR	Требование кадровой синхронизации при приеме каждого слова: 0 - кадровая синхронизация требуется только при приеме первого слова; 1 - кадровая синхронизация требуется при приеме каждого слова.
14	IRFS	Разрешение выдачи внутреннего сигнала кадровой синхронизации **RFS**: 0 - вывод **RFS** является входом; 1 - вывод **RFS** является выходом, и на него выдается сигнал кадровой синхронизации, период которого определяется полем RDIV[31:16].
15	IMODE	Разрешение сравнения кода принятых данных в многоканальном режиме работы порта: 0 - запрещение сравнения; 1 - разрешение сравнения.

16	LRFS	Выбор активного уровня импульса кадровой синхронизации при приеме данных: 0 - импульс кадровой синхронизации имеет активный высокий уровень; 1 - импульс кадровой синхронизации имеет активный низкий уровень.
17	RLAFS	Выбор режима кадровой синхронизации при приеме данных: 0 - режим ранней кадровой синхронизации; 1 - режим поздней кадровой синхронизации.
19:18	-	Резерв
20	IMAT	Выбор режима сравнения принятых данных в многоканальном режиме работы порта: 0 - принятые данные записываются в буфер **Rx**, если сравнение произошло не успешно (т. е. сравниваемые данные не совпали); 1 - принятые данные записываются в буфер **Rx**, если сравнение прошло успешно.
21	-	Резерв
22	SPL	Разрешение замыкания внутренней петли данных: 0 - обычный режим работы; 1 - сигналы приемной частоты порта **DR**, **RCLK**, **RFS** внутренне объединяются с сигналами передающей частоты порта **DT**, **TCLK**, **TFS**, которые становятся выходами.
23	MCE	Разрешение многоканального режима работы: 0 - режим запрещен; 1 - режим разрешен.
28:24	NCH	Число временных каналов при многоканальном режиме работы порта. Число каналов равно коду в этом поле, увеличенному на единицу. Число каналов может быть от 1 (NCH=0) до 32 (NCH=31).
29	ROVF	Признак переполнения буфера **Rx**. Доступен только по чтению. Обнуляется при REN=0.
31:30	RXS	Состояние буфера SRx: 00 - буфер пуст; 10 - буфер частично полон; 11 - буфер полон. Поле доступно только по чтению.

При многоканальном режиме работы биты SPL, REN, RFSR, RLAFS должны иметь нулевое состояние.

Перед записью в регистр **SRCTL** нового значения, его необходимо предварительно обнулить.

Исходное состояние регистра **SRCTL** - нули.

TDIV

Формат регистра коэффициентов деления при передаче данных (**TDIV**) приведен в таблице 3.16:

Таблица 3.16. Формат регистра TDIV.

Номер разряда	Условное обозначение	Назначение
15:0	**TCLKDIV**	Определяет период частоты TCLK.
31:16	**TFSDIV**	Определяет период частоты формирования кадрового синхроимпульса TFS.

Период частоты **TCLK** вычисляется по формуле:

`период частоты CLK * 2((содержимое поля TCLKDIV) +1)`

При выборе данной частоты необходимо учитывать системные ограничения.

Период формирования кадрового синхроимпульса вычисляется по формуле:

`период частоты TCLK * ((содержимое поля TFSDID) +1)`

При `TFSDIV=0` кадровый синхроимпульс постоянно активен. Величина TFSDIV не должна быть меньше, чем длина слова минус 1.

Если порт **SPort** не используется, то делитель TFSDIV может быть использован как делитель внешней частоты, или для генерации периодических импульсов или прерывания. Для выполнения этих функций **SPort** должен быть активизирован.

RDIV

Формат регистра коэффициентов деления при приеме данных (**RDIV**) приведен в таблице 3.17:

Таблица 3.17. Формат регистра RDIV.

Номер разряда	Условное обозначение	Назначение
15:0	**RCLKDIV**	Определяет период частоты RCLK.
31:16	**RFSDIV**	Определяет период частоты формирования кадрового синхроимпульса RFS.

Период частоты **RCLK** вычисляется по формуле:

`период частоты CLK * 2((содержимое поля RCLKDIV) +1)`

При выборе данной частоты необходимо учитывать системные ограничения.

Период формирования кадрового синхроимпульса вычисляется по формуле:

`период частоты RCLK * ((содержимое поля RFSDID) +1)`

При RFSDIV=0 кадровый синхроимпульс постоянно активен. Величина RFSDIV не должна быть меньше, чем длина слова минус 1.

Если порт **SPort** не используется, то делитель RFSDIV может быть использован как делитель внешней частоты, или для генерации периодических импульсов или прерывания. Для выполнения этих функций **SPort** должен быть активизирован.

Регистры выбора канала в многоканальном режиме

Все регистры выбора каналов являются 32-разрядными, каждый бит соответствует своему каналу. Исходное состояние регистров - нули. В таблице 3.18 приведен перечень регистров выбора канала в многоканальном режиме:

Таблица 3.18. Регистры выбора канала

Условное обозначение регистра	Название регистра
MTCS	Выбор канала для передачи данных
MRCS	Выбор канала для приема данных
MRCE	Выбор канала для сравнения принимаемых данных

При единичном состоянии бита в регистре **MTCS** последовательному порту разрешается передавать слово в соответствующем временном канале. При нулевом состоянии бита в регистре **MTCS** последовательному порту запрещается передавать слово в соответствующем временном канале. В этом временном канале вывод **DT** находится в третьем состоянии. В регистре **MTCS** может быть установлено любое число единиц.

При единичном состоянии бита в регистре **MRCS** последовательному порту разрешается принимать слово в соответствующем временном канале. Принятое слово загружается в буфер **Rx**. При нулевом состоянии бита в регистре **MRCS** последовательному порту запрещается принимать слово в соответствующем временном канале. То есть слово игнорируется. В регистре **MRCS** может быть установлено любое число единиц.

Работа регистра **MRCE** разрешается, если разрешено сравнение принимаемых слов данных в соответствии с содержимым регистров KEYWD и KEYMASK, то есть бит IMODE в регистре SRCTL имеет единичное состояние. При единичном состоянии бита в регистре **MRCE** последовательному порту разрешается сравнивать принимаемое слово в соответствующем разрешенном временном канале. Принятое слово загружается в буфер **Rx**. При нулевом состоянии бита в регистре **MRCE** последовательный порт в соответствующем временном интервале принимает все слова данных. То есть сравнения не производятся. В регистре **MRCE** может быть установлено любое число единиц.

Регистры сравнения принимаемых данных в многоканальном режиме

Перечень регистров сравнения принимаемых данных в многоканальном режиме приведен в таблице 3.19:

Таблица 3.19. Регистры сравнения принимаемых данных.

Условное обозначение регистра	Название регистра
KEYWD	Регистр сравнения
KEYMASK	Регистр маски

Регистры являются 32-разрядными. Исходное состояние регистров неопределено.

Регистр **KEYWD** содержит образец для сравнения с принятым словом данных.

Регистр **KEYMASK** указывает, сравнение каких бит в принятом слове разрешено. При нулевом состоянии бита в регистре **KEYMASK** разрешается сравнение соответствующего бита в принятом слове данных и регистре **KEYWD**. При единичном состоянии бита в регистре **KEYMASK** запрещается (маскируется) сравнение соответствующего бита в принятом слове данных и регистре **KEYWD**, то есть состояние бита не анализируется.

4.3.4. Одноканальный режим работы

В одноканальном режиме работы передающая и приемная части последовательного порта работают раздельно и независимо. Режимы передачи и приема слов данных могут быть различны.

Для синхронизации передачи данных формируются кадровые синхроимпульсы **TFS**. При `TFSR=1` (кадрированные данные) каждое слово сопровождается кадровым синхроимпульсом. При `TFSR=0` (некадрированные данные) кадровый синхроимпульс используется для инициализации всего процесса передачи данных и формируется только один раз перед передачей первого бита информации. В этом случае, данные по каналу связи идут одним потоком.

На рисунке 3.3 приведены временные диаграммы передачи кадрированных и некадрированных данных:

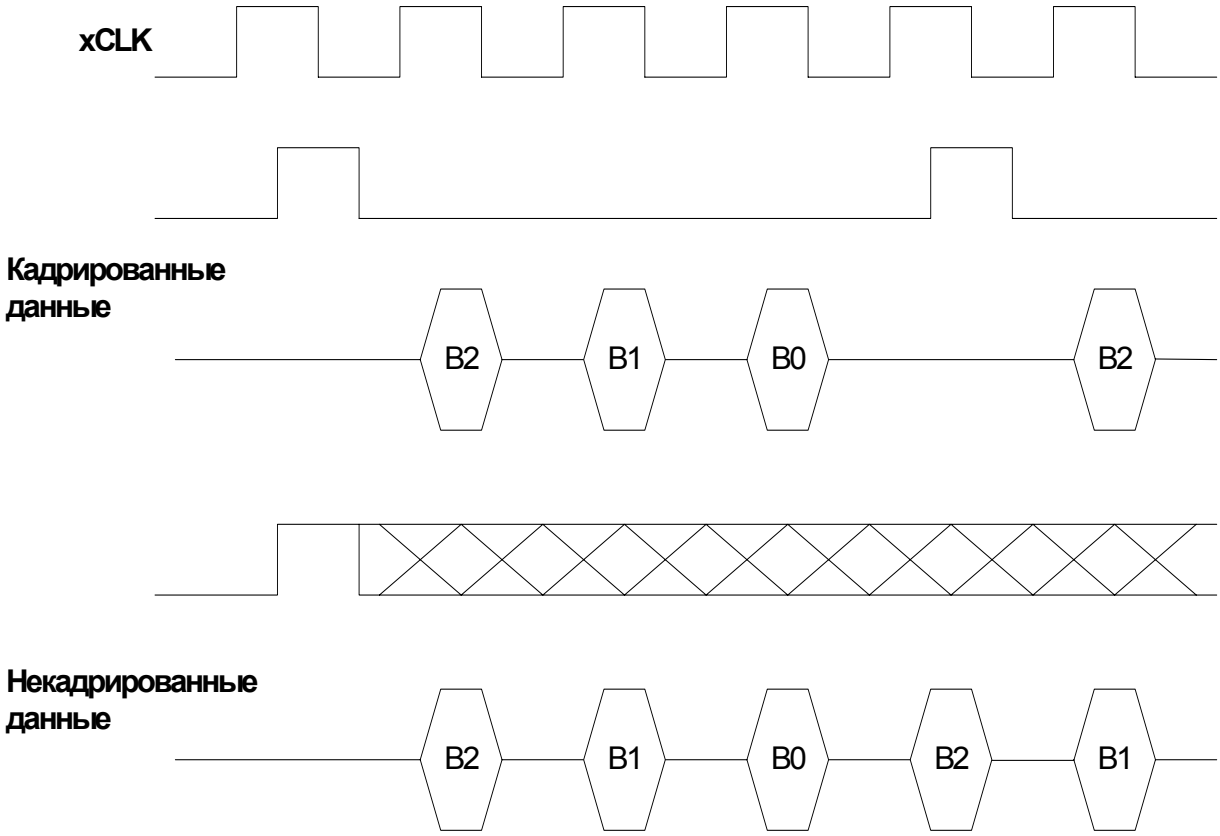


Рисунок 3.3. Передача кадрированных и некадрированных данных.

Аналогично, для синхронизации приема данных формируются кадровые синхроимпульсы **RFS**. При RFSR=1 каждое слово сопровождается кадровым синхроимпульсом. При RFSR=0 кадровый синхроимпульс используется для инициализации всего процесса приема данных и формируется только один раз перед приемом первого бита информации. В этом случае, данные по каналу связи идут одним потоком.

Кадровые синхроимпульсы **TFS** и **RFS** могут формироваться самим портом или поступать от внешнего источника.

При работе последовательного порта может использоваться ранняя или поздняя кадровая синхронизация. Временные диаграммы ранней и поздней кадровой синхронизации приведены на рисунке 3.4.:

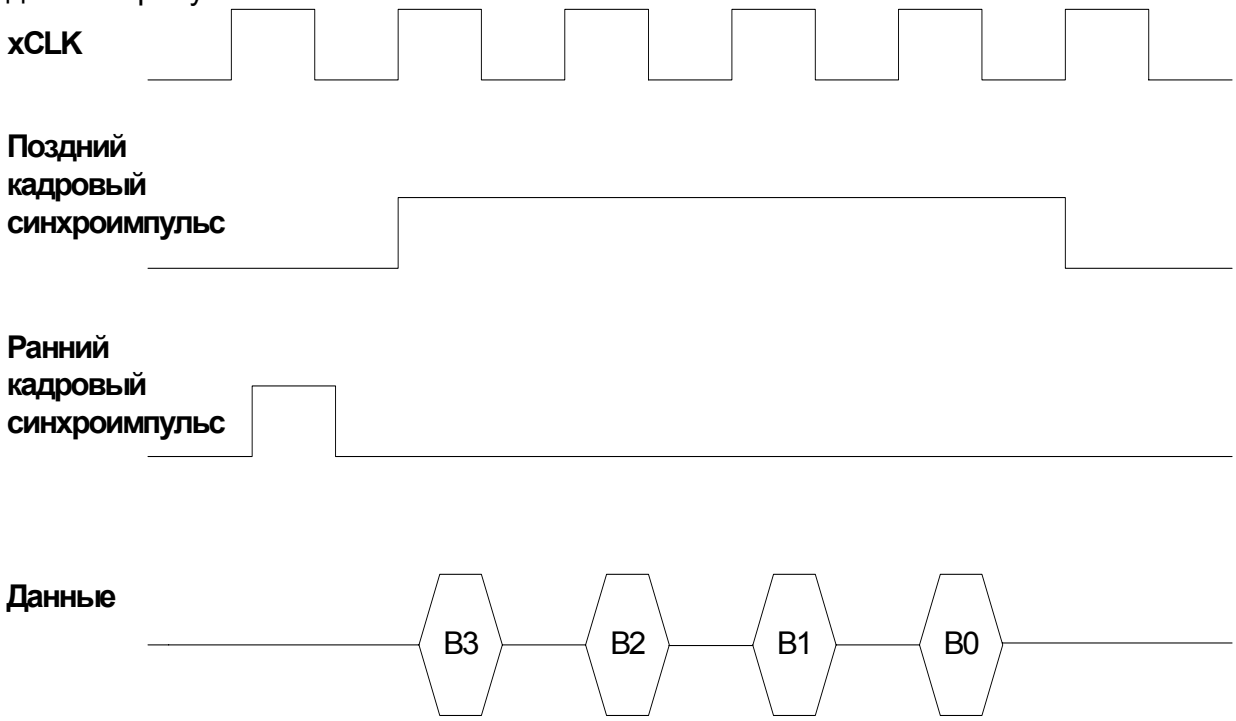


Рисунок 3.4. Ранняя и поздняя кадровая синхронизация.

Для настройки передающей части порта в одноканальном режиме необходимо в регистре STCTL выбрать необходимые параметры передачи и установить в единичное состояние бит TEN.

Для обеспечения приема данных в одноканальном режиме необходимо выбрать параметры приема и установить в единичное состояние бит REN (регистр SRCTL).

4.3.5. Режим петли

Режим петли используется для тестирования работы последовательного порта.

В этом режиме сигналы приемной части порта **DR**, **RCLK**, **RFS** внутренне соединяются с сигналами передающей части порта **DT**, **TCLK**, **TFS**. При этом выводы **DT**, **TCLK**, **TFS** переходят в активное состояние.

В режиме петли должны быть разрешены режимы генерации внутренней частоты передачи и внутреннего кадрового синхроимпульса передачи.

Проверка многоканального режима работы в режиме петли не обеспечивается.

Для включения последовательного порта в режим петли необходимо:

- в регистрах STCTL и SRCTL установить параметры передачи: биты TENDN, TLEN, TFSR, RENDN, RLEN, RFSR, TCKRE. Эти параметры должны быть одинаковы для передающей и приемной частей порта;
- в регистре SRCTL установить в единичное состояние биты REN, SPL.
- в регистре STCTL установить в единичное состояние биты TICLK, ITFS, TEN, а биты IRFS, RICLK – в нулевое состояние.
  Сначала определяется состояние регистра SRCTL, а затем – регистра STCTL.

4.3.6. Многоканальный режим работы

Последовательный порт обеспечивает многоканальный режим работы, который позволяет обмениваться данными в системах с временным мультиплексированием (**TDM** - *time-division-multiplexed*). В многоканальной системе каждое слово данных передается в своем временном канале (слоте). Многоканальный режим работы включается при `MCE=1`.

В многоканальной системе данные передаются кадрами. Кадр содержит число слов, равное числу временных каналов. Признаком начала каждого кадра передачи данных является сигнал кадровой синхронизации **RFS**. Этот сигнал используется для синхронизации каналов и рестарта всех последовательных портов. **RFS** может генерироваться одним из последовательных портов многоканальной системы, или формироваться внешним источником кадровой синхронизации.

В многоканальном режиме приемная и передающая части последовательного порта работают одновременно и используют общее оборудование.

В многоканальном режиме сигнал **TFS** является признаком того, что данный последовательный порт находится в режиме передачи информации и вывод **DT** имеет активное состояние.

Последовательный порт автоматически выбирает временной канал. Имеется 32 канала для передачи или приема данных. Другими словами, последовательный порт в каждом временном канале может выполнять следующие действия:

- передавать данные;
- принимать данные;
- передавать и принимать данные;
- не принимать и не передавать данные.

В многоканальном режиме работы:

- вывод **DT** переводится в активное состояние (из высокоомного) только в разрешенном временном канале;

- вывод **TCLK** является входом и должен быть соединен с соответствующим выводом **RCLK**;
- вывод **TFS** обычно остается не подсоединенным;
- выводы **RFS** всех портов многоканальной системы объединяются.

На рисунке 3.5 приведена временная диаграмма приема и передачи данных в многоканальном режиме. В данном примере порт выполняет прием данных во временном канале 0 и передает данные во временные каналы 1 и 2:

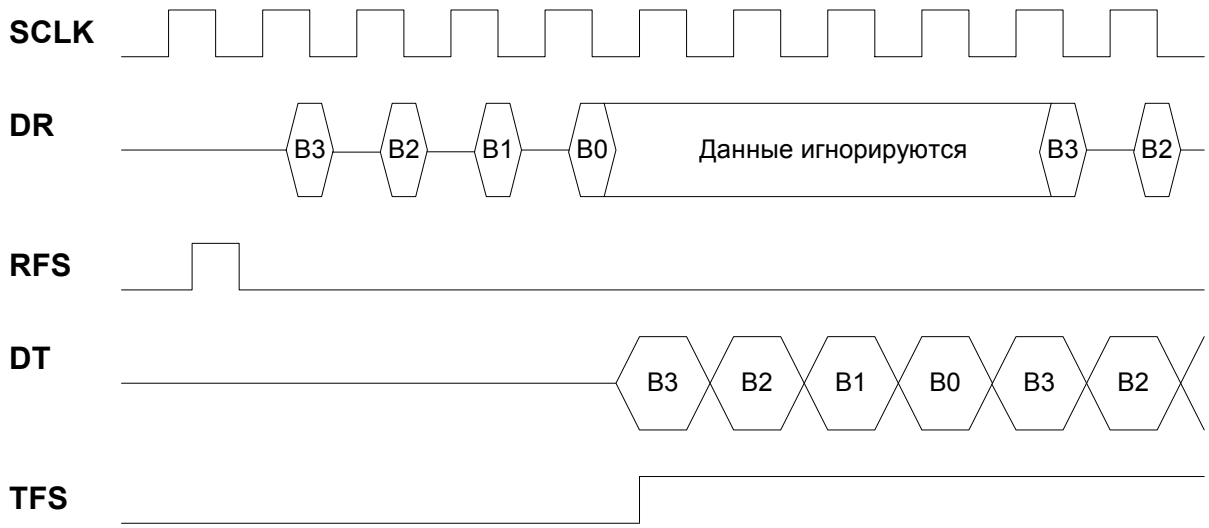


Рисунок 3.5. Прием и передача данных в многоканальном режиме.

Для обеспечения работы данного последовательного порта в многоканальном режиме необходимо:

- в поле NCH регистра SRCTL установить число каналов, которое используется в данной системе;
- в поле MFD регистра STCTL установить величину задержки между импульсом кадровой синхронизации и началом передачи первого бита данных. Задержка измеряется в периодах частоты передачи данных. При `MFD=0` кадровый синхроимпульс по времени совпадает с первым битом. Максимальная величина MFD равна 15. Программирование этой задержки позволяет работать по разным протоколам передачи данных. При работе на максимальной частоте передачи данных (`CLK/2`) в MFD должен быть установлен код не менее 1;
- в регистре MTCS установить в единичное состояние биты временных каналов, в которых требуется передавать данные;
- в регистре MRCS установить в единичное состояние биты временных каналов, в которых требуется принимать данные;
- в регистре SRCTL определить состояние бит IMODE и IMAT, то есть установить режим сравнения принимаемых данных (при необходимости);

- в регистрах STCTL и SRCTL установить параметры передачи и приема слов (биты TENDN, TLEN, RENDN, RLEN, TCKRE, RCKRE, LTFS, LRFS). Следует отметить, что для последовательного порта параметры передачи и приема в многоканальном режиме должны быть одинаковы;
- в регистры KEYWD, KEYMASK, MRCE записать необходимые коды, если данные необходимо принимать в режиме сравнения;
- в регистре SRCTL установить в единичное состояние бит IRFS, если данный последовательный порт должен формировать кадровый синхроимпульс **RFS**;
- биты TEN, TFSR, ITFS, TLAFS, DITFS, REN, RFSR, RLAFS в регистрах STCTL и SRCTL должны иметь нулевое состояние;
- в регистре SRCTL установить в единичное состояние бит MCE.

Номер временного канала, который в данный момент времени активен, содержится в доступном только по чтению поле CHNL регистра STCTL. Это поле содержит инкрементирующий счетчик по модулю NCH.

Если в многоканальном режиме для данного порта наступил активный временной канал для передачи, то она выполнится вне зависимости от наличия необходимых данных в буфере STx. Признак недозагрузки буфера STx (TUVF) в многоканальном режиме не устанавливается.

В многоканальном режиме признак переполнения буфера SRx (ROVF) функционирует.

В многоканальном режиме работы прием данных можно выполнять со сравнением, используя регистры KEYWD, KEYMASK и MRCE. При этом, каждое принятое слово данных сравнивается с содержимым регистра KEYWD с использованием маски в регистре KEYMASK. Режим сравнения определяется состоянием бит IMODE и IMAT в регистре SRCTL. Если сравнение произошло неуспешно, то принятое слово данных в буфер SRx не записывается при бите IMAT, установленном в единицу. Если бит IMAT установлен в ноль и сравниваемые данные не совпали (сравнение произошло неуспешно), то принятое слово данных в буфер SRx записывается.

4.3.7. DMA и прерывания SPort

С последовательным портом могут быть связаны два канала **DMA**:
SportTxCh – передача данных в последовательный канал;
SportRxCh – прием данных из последовательного канала.
Эти каналы, а также способы работы с ними, рассматриваются в разделе "DMA последовательных портов".

Прерывания от последовательного порта
Последовательный порт формирует прерывания по приему и передаче данных.
Если соответствующий канал **DMA** активизирован, то прерывания формируются по завершению передачи или приема всего блока данных.
Если соответствующий канал **DMA** не активизирован, то прерывания формируются по завершению передачи или приема каждого слова данных.

4.3.8. Программирование последовательных портов

Чтобы запрограммировать порты обмена последовательным кодом на прием/передачу данных, необходимо:

- Установить значения регистров коэффициентов деления TDIV/RDIV;
- Выбрать требуемый режим работы (одноканальный/петля/многоканальный) и настроить регистры порта в соответствии с этим режимом;
- Определить механизм приема/передачи данных (прямая запись/чтение или посредством каналов **DMA**);
- Определить способ обслуживания окончания передачи/приема (по опросу, по прерываниям);
- При работе по опросу - опрашивать состояние полей TXS и RXS регистров STCTL и SRCTL соответственно;
- При работе по прерываниям - описать обработчики исключений по возможным прерываниям.

Описание работы с портами **SPort** посредством каналов **DMA** приводится на странице "DMA SPort".

Пример программы, работающей с портом **SPort0** по опросу приведен на странице "Пример программы, использующей последовательные порты".

4.3.9. Пример программы, использующей порт SPort

Рассмотрим пример программы, передающей и принимающей данные через порт **SPort0**. Программа состоит из файла *main.c* и заголовочного файла *memory_12.h*, описывающего адресное пространство ИМС *MultiCore-12*. Кроме того, для создания режима петли используется заглушка порта **SPort0** на себя. То есть, все данные, попадающие в буфер STx, передаются в принимающий буфер SRx.

Файл *main.c* содержит код программы, использующей **SPort**:

```
#include "memory_12.h"

main()
{
      int OutputArray[256];
      int InputArray[256];
      int i;

      for (i=0;i<256;i++)
          OutputArray[i]=i;

      //настройка SPort
      TDIV0=0x1f0001;
      RDIV0=0x1f0001;
      SRCTL0=0x1f1;
      STCTL0=0x45f1;

      for (i=255;i>-1;i--)
      {
          STx0=OutputArray[i];          //передача очередного значения
```

```
        while (!(SRCTL0 & 1<<31));     //ожидание получения новых данных
        InputArray[255-i]=SRx0;        //чтение полученного значения
    }

    SRCTL0 &= 0xfffffffe;              //сброс бита REN
    STCTL0 &= 0xfffffffe;              //сброс бита TEN

    while (1);
}
```

Данная программа осуществляет следующие действия:

- Заполняет массив `OutputArray[256]` числами от 0 до 256;
- Настраивает порт **SPort** на передачу и прием данных с длиной слова в 32 бита (`TLEN=RLEN=31`);
- Осуществляет циклическую передачу значений массива `OutputArray[256]` от последнего элемента к первому.

Передача осуществляется записью очередного значения в порт STx. После этого программа ожидает установки поля RXS регистра SRCTL в значение `RXS=0x2`. Это значение указывает, что переданное слово появилось в буфере SRx. Затем, слово, переданное в SRx, считывается в массив `InputArray[256]` (при этом RXS сбрасывается).

По завершении цикла биты REN и TEN регистров SRCTL и STCTL сбрасываются.

В результате работы программы массив `InputArray[256]` будет содержать значения массива `OutputArray[256]`, записанные в обратном порядке.

4.4. Линковые порты (LPort)

4.4.1. Введение

В данной главе рассматриваются линковые порты, их характеристики, а также способы работы с ними. Глава включает в себя:

- Основные характеристики линковых портов;
- Описание регистров линковых портов;
- DMA и прерывания от линковых портов;
- Способы программирования линковых портов;
- Пример программы, использующей линковый порт.

4.4.2. Основные характеристики

Линковый порт (**LPort**) имеет следующие основные характеристики:

- частота передачи данных – **CLK**`/4`, **CLK**`/2` (**CLK** – тактовая частота *MultiCore-12*);
- использована двойная буферизация передаваемых и принимаемых данных;

- выполняет однословный обмен данными по прерываниям под управлением RISC-ядра;
- выполняет обмен блоками данных при помощи **DMA**;
- по внешнему интерфейсу линковый порт совместим с ADSP-21160.

Структурная схема линкового порта приведена на рисунке 3.6:

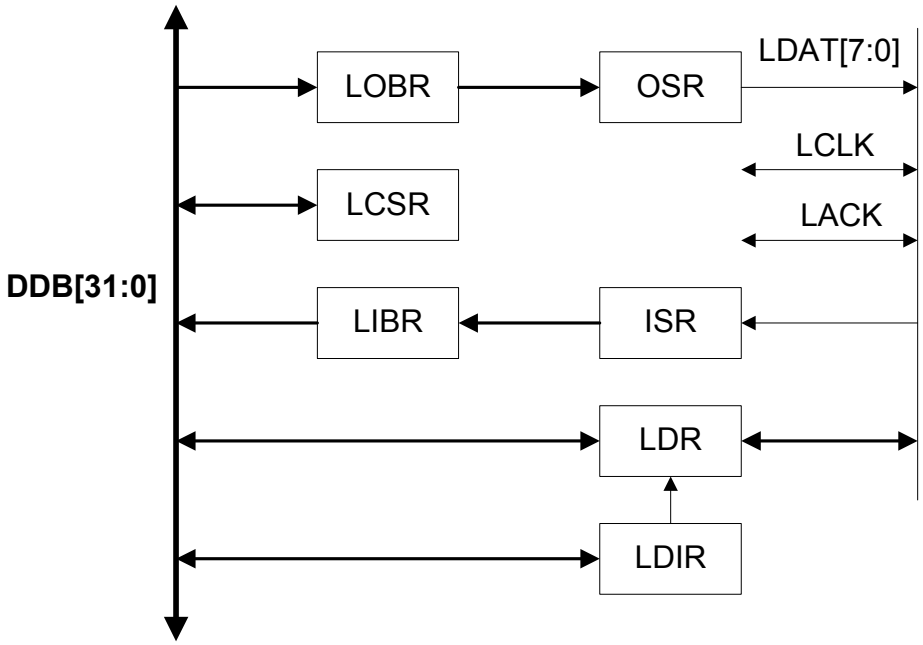


Рисунок 3.6. Структурная схема порта LPort.

Передаваемые 32-разрядные данные записываются в выходной буферный регистр (**OBR**), а затем аппаратно переписываются в передающий сдвигающий регистр (**OSR**), если он пуст. После этого, в выходной буферный регистр могут быть записаны очередные данные. Из передающего сдвигающего регистра данные выдаются во внешнюю шину данных тетрадами или байтами.

Из внешней шины данные поступают в приемный сдвигающий регистр (**ISR**) тетрадами или байтами. После набора 32-рязрядного слова, он переписывается во входной буферный регистр (**IBR**).

Данные передаются, начиная со старшей тетрады или старшего байта.

Если **LPort** неактивизирован (LEN=0), внешние лини **LDAT**[7:0], **LCLK**, **LACK** можно использовать как 10-разрядный двухнаправленный порт ввода-вывода.

В таблице 3.20 приведены выводы порта **LPort**:

Таблица 3.20. Выводы порта LPort.

Название вывода	Тип вывода	Описание
LDAT[3:0]/[7:0]	IO	Внешняя шина данных. Данные по этой шине передаются по положительному вронту сигнале LCLK
LCLK	O	Частота передачи данных
LACK	IO	Подтверждение приема

4.4.3. Регистры LPort

В данном разделе рассматриваются регистры линковых портов LPort. Перечень регистров приведен в таблице 3.21:

Таблица 3.21. Регистры линковых портов.

Условное обозначение регистра	Название регистра
LTx	Буфер передачи данных
LRx	Буфер приема данных
LCSR	Регистр управления и состояния
LDIR	Регистр управления направлением выводов порта ввода-вывода
LDR	Регистр данных порта ввода-вывода

LTx

Буфер передачи **LTx** является буфером FIFO на два 32-разрядных слова и состоит из выходного буферного регистра и передающего сдвигающего регистра. Два 32-разрядных слова могут быть сразу записаны в буфер **LTx**, если он был до этого пуст.

Буфер передачи **LTx** генерирует прерывание (бит LportTx в регистре QSTR) при следующих условиях:

- бит `LTRAN=1`;
- выходной регистр данных пуст;
- соответствующий канал **DMA** не активизирован;
- данное прерывание не замаскировано.

Данное прерывание формируется в момент активизации линкового порта на передачу при пустом буфере **LTx**, или в момент переписи содержимого выходного регистра данных в выходной сдвигающий регистр. Прерывание, генерируемое буфером передачи, сигнализирует о том, что буфер **LTx** готов принять следующее слово. Прерывание от буфера передачи сбрасывается в момент записи в него данных.

Загрузка данных в порт возможна только при активизации порта на передачу.

LRx

Буфер приема **LRx** является буфером FIFO на два 32-разрядных слова и состоит из входного регистр данных и входного буферного регистра. Одно принятое 32-разрядное слово может храниться в буфере **LRx**, пока вдвигается второе слово.

В момент окончания приема в буфер **LRx** 32-разрядного слова данных, генерируется прерывание, если оно разрешено, а соответствующий канал **DMA** не активизирован. Данное прерывание сбрасывается при чтении данных из буфера приема.

Считывание данных из буфера приема возможно только при активизации порта на прием.

LCSR

В таблице 3.21 приведен формат регистра управления и состояния **LCSR**:

Таблица 3.21. Формат регистра LCSR.

Номер разряда	Условное обозначение	Назначение
0	LEN	Разрешение работы порта: 0 – все выводы порта находятся в высокоимпедансном состоянии; 1 – порт работает в соответствии с состоянием бита LTRAN.
1	LTRAN	Режим работы порта: 0 – приемник; 1 – передатчик.
2	LCLK	Управление частотой работы порта: 0 – CLK/4; 1 – CLK/2.
4:3	LSTAT	Состояние буферов Tx или Rx: 00 – буфер пуст; 10 – буфер содержит одно слово данных; 11 – буфер полон.
5	LRERR	Ошибка приема данных: 0 – приняты все биты данных; 1 – приняты не все биты данных.
6	LDW	Разрядность внешней шины данных: 0 - 4-разряда (32-разрядное слово передается за 8 посылок); 1 - 8-разряда (32-разрядное слово передается за 4 посылки).
7	SRQ_TX	Признак запроса обслуживания на передачу данных
8	SRQ_RX	Признак запроса обслуживания на прием данных
31:9	-	Резерв

Исходное состояние регистра **LCSR** – нули. Биты LEN, LTRAN, LCLK доступны по записи и чтению, а LSTAT и LRERR – только по чтению.

Биты LSTAT и LRERR сбрасываются при LEN=0.

LDR и LDIR

10-разрядный регистр данных порта ввода-вывода (**LDR**) предназначен для реализации гибкого интерфейса с внешними устройствами. Внешние выводы порта ввода-вывода совмещены с внешними выводами линкового порта.

Соответствие разрядов регистра **LDR** и внешних линий линкового порта приведено в таблице 3.22:

Таблица 3.22. Разряды LDR.

Номер разряда Регистра LDR	Внешние выводы LPORT
0	**LCLK**
1	**LACK**
9:2	**LDAT[7:0]**

Настройка направления выводов порта ввода-вывода осуществляется программно при помощи 10-разрядного регистра **LDIR**. Если разряд этого регистра имеет нулевое состояние, то соответствующий разряд порта ввода-вывода является входом и наоборот. Линии порта ввода-вывода могут быть выходами, если LEN=0.

Исходное состояние регистров **LDR**, **LDIR** – нули.

4.4.4. DMA и прерывания LPort

С каждым линковым портом связан канал **DMA LportCh**. Направление передачи **DMA** определяется битом LTRAN.

Каналы **DMA** линковых портов описаны в разделе "DMA LPort".

Линковый порт формирует прерывания по приему и передаче данных.

Если соответствующий канал **DMA** разрешен, то прерывания формируются по завершению передачи или приема всего блока данных.

Если соответствующий канал **DMA** запрещен, то прерывания формируются по завершению передачи или приема каждого 32-разрядного слова данных.

Если линковый порт не активизирован (LEN=0), он формирует прерывание по запросу обслуживания, если:

- на внешней шине выставлены данные на прием (активное состояние сигнала **LCLK**);
- из внешней шины поступил запрос на выдачу данных (активное состояние сигнала **LACK**).

Данное прерывание сбрасывается после установки LEN=1.

4.4.5. Программирование линковых портов

Чтобы запрограммировать линковый порт **LPort** на прием или передачу данных необходимо:

- В регистре LCSR установить бит LTRAN в единицу, если порт будет работать на передачу, или в ноль, если порт будет принимать данные;
- Установить нужные значения полей LCLK и LDW в регистре LCSR;
- Разрешить работу порта битом LEN регистра LCSR;

- При работе с портом по прерываниям - написать соответствующие обработчики исключений по прерыванию;
- При работе с портом по опросу - опрашивать состояние поля LSTAT регистра LCSR;
- При работе по **DMA** - настроить соответствующий канал **DMA** для чтения данных из порта (записи данных в порт);
- Если порт функционирует без участия **DMA** - передавать данные следует посредством записи очередного слова данных в буфер LTx, а принимать - чтением из буфера LRx.

Каналы **DMA** линковых портов рассматриваются в разделе "DMA LPort".

Прерывания в ИМС "*МУЛЬТИКОР*" рассматриваются в разделе "Обработка прерываний".

Пример программы, использующей порт **LPort** приведен здесь.

4.4.6. Пример программы, использующей линковый порт

Рассмотрим пример программы, передающей данные через порт **LPort0** и принимающей их через порт **LPort1**. Программа состоит из файла *main.c* и заголовочного файла *memory_12.h*, описывающего адресное пространство ИМС *MultiCore-12*. Кроме того, для создания режима петли используется заглушка порта **LPort0** на порт **LPort1**. То есть, все данные, попадающие в буфер LTx порта **LPort0**, передаются в принимающий буфер LRx порта **LPort1**.

Файл *main.c* содержит код программы, использующей порты **LPort**:

```
#include "memory_12.h"

main ()
{
      int InputArray[256];
      int OutputArray[256];
      int i;

      for (i=0;i<256;i++)
          OutputArray[i]=i;

      //настройка портов LPort0(передача), LPort1(прием)
      LCSR0=0x47;
      LCSR1=0x45;

      //передача и прием данных
      for (i=255;i>-1;i--)
      {
          LTx0=OutputArray[i];          //передача очередного значения
          while (!(LCSR1 & 1<<4));      //ожидание получения новых данных
          InputArray[255-i]=LRx1;       //чтение полученного значения
      }

      LCSR0=0;
      LCSR1=0;

      while (1);
}
```

Данная программа осуществляет следующие действия:

- Заполняет массив `OutputArray[256]` числами от 0 до 256;
- Настраивает порт **LPort0** на передачу, а **LPort1** - на прием данных. Передача и прием осуществляются на частоте **CLK**/2 по 8 разрядов за посылку;
- Осуществляет циклическую передачу значений массива `OutputArray[256]` от последнего элемента к первому.

Передача осуществляется записью очередного значения в порт LTx. После этого программа ожидает установки поля LSTAT регистра LCSR1 в значение `LSTAT=0x2`. Это значение указывает, что переданное слово появилось в буфере LRx. Затем, слово, переданное в LRx, считывается в массив `InputArray[256]` (при этом LSTAT сбрасывается).

По завершении цикла регистры LCSR0 и LCSR1 обнуляются.

В результате работы программы массив `InputArray[256]` будет содержать значения массива `OutputArray[256]`, записанные в обратном порядке.

5. DMA

5.1. Введение

В данной главе рассматриваются способы работы с каналами контроллера прямого доступа в память (**DMA**). Каналы **DMA** используются для передачи больших объемов данных, минуя ЦПУ (RISC-ядро), то есть обращаясь в память без использования операций пересылки.

Глава включает в себя:

- Общие положения по каналам **DMA**;
- Описание каналов DMA для обмена между внутренней и внешней памятью;
- Описание каналов DMA последовательных портов;
- Описание каналов DMA линковых портов;
- Способы самоинициализации каналов.

Также глава содержит примеры текста программ на языке *C* с использованием каналов **DMA**.

5.2. Общие положения

5.2.1. Типы каналов DMA

Контроллер **DMA** имеет 12 каналов следующих типов:

- Каналы обмена данными между последовательными портами и внутренней (MEM, PMEM, XMEM) или внешней памятью;
- Каналы обмена данными между линковыми портами и внутренней (MEM, PMEM, XMEM) или внешней памятью;
- Каналы обмена данными между внутренней (MEM, PMEM, XMEM) и внешней памятью.

В таблице 4.1 приведен перечень каналов **DMA** для *МУЛЬТИКОР-12* и *МУЛЬТИКОР-24*.

Таблица 4.1. Перечень каналов DMA.

Условное обозначение канала	Назначение канала	Приоритет каналов DMA и CPU
SportRxCh0	Прием данных из буфера SRx порта SPORT0 во внутреннюю или внешнюю память	0
SportRxCh1	Прием данных из буфера SRx порта SPORT1 во внутреннюю или внешнюю память	1
SportTxCh0	Передача данных из внутренней или внешней памяти в буфер STx порта SPORT0	2
SportTxCh1	Передача данных из внутренней или внешней памяти в буфер STx порта SPORT1	3
CPU	-	4
LportCh3 – LportCh0	Обмен данными между буферами данных линковых портов и памятью (внешней или внутренней)	8-5
MemCh3 – MemCh0	Обмен данными между внешней памятью и внутренней памятью.	12-9 (изменяется циклически)

Каждый канал **MemCh[3-0]** имеет внешний сигнал запроса передачи (DMAR[3-0] соответственно), позволяющий организовывать эффективный обмен данными с внешними устройствами, например с *FIFO*.

Если при работе **DMA** изменяется программный код в памяти, то когерентность кэш программ ЦПУ (*ICACHE*) аппаратно не обеспечивается. В этом случае для обеспечения когерентности используется бит **FLUSH** в регистре **CSR**.

5.2.2. Приоритет каналов

ЦПУ по шине **RDB** без конфликтов с **DMA** обменивается с памятью MEM, с системными регистрами **CSR**, **MASKR**, **QSTR**, и с регистрами таймеров **IT**, **WDT**, **RTT**. Кроме того, ЦПУ без конфликтов обменивается с регистрами **MPORT** и внешней памятью (через шину **EDB**), если нет **DMA**-обменов с внешней памятью.

При передаче данных каналы **DMA** конфликтуют между собой всегда. Каналы **DMA** конфликтуют с ЦПУ, если:

- ЦПУ и **DMA** одновременно запрашивают шину **DDB**;
- ЦПУ и **DMA** одновременно запрашивают внешнюю память по шине **EDB**.

Приоритет каналов **DMA** указан в правой колонке таблицы типов каналов (0 – наивысший приоритет). Если несколько каналов одновременно запрашивают шину **DDB**, то ее занимает канал, приоритет которого самый высокий.

Взаимный приоритет каналов **MemCh** изменяется циклически следующим образом. Исходное распределение приоритетов между каналами **MemCh** (в порядке их убывания): **MemCh0**, **MemCh1**, **MemCh2**, **MemCh3**. Далее, после каждой **DMA**-передачи распределение приоритетов изменяется циклическим сдвигом влево, таким образом, что приоритет канала, который выполнил передачу, становится самым низким. Например, если после исходного состояния передал канал **MemCh0**, то приоритеты распределятся следующим образом: **MemCh1**, **MemCh2**, **MemCh3**, **MemCh0**. Далее, если передал канал **MemCh3**, то приоритеты распределятся следующим образом: **MemCh0**, **MemCh1**, **MemCh2**, **MemCh3** и т.д.

5.2.3. Темп передачи данных

DMA-передача одного 32-разрядного слова данных между внутренней памятью и **SPORT** или **LPORT** выполняется за время TCLK (период частоты *CLK*).

Время передачи одного 32-разрядного слова данных между внешней памятью и **SPORT** или **LPORT** или внутренней памятью, равно:

- для асинхронной внешней памяти – 2*TCLK + TCLK*N, где N – число тактов ожидания (код в поле **WS** регистров **CSCON**, увеличенный на 1).
- для синхронной внешней памяти - TCLK.

Каналы последовательных и линковых портов за один цикл занятия шины **DDB** передают одно слово данных. После передачи этого слова шина **DDB** данным каналом освобождается.

Каналы **MemCh** за один цикл занятия шины **DDB** передают пачку данных. Размер пачки задается полем **WN** в регистре **CSR** соответствующего канала **DMA** и определяется системными требованиями по передаче данных. Если после передачи пачки данных нет запросов от других каналов **DMA** или ЦПУ, то данный канал без перерыва начинает передавать следующую пачку данных и т.д.

ЦПУ за один цикл занятия шины **DDB** выполняет одну операцию *Load/Store* (после этого шина освобождается).

За один цикл занятия шины **EDB**, ЦПУ выполняет одну из следующих совокупностей операций (после этого шина освобождается):

- обмен одним словом данных по команде *Load/Store*;
- выборка команды из внешней памяти;
- выборка команды из внешней памяти и обмен одним словом данных по команде *Load/Store*;
- записывает в **ICACHE** 4 команды, если адрес команды *CACHED*, а ее нет в **ICACHE** (ситуация MISS);
- записывает в **ICACHE** 4 команды, если адрес команды *CACHED*, а ее нет в **ICACHE** (ситуация MISS) и обмен одним словом данных по команде *Load/Store*.

5.2.4. Регистры DMA

Для управления работой каждого канала **DMA** имеются следующие регистры:

- регистр управления и состояния (**CSR**);
- набор регистров индекса (адрес памяти) и смещения (**IOR**, **IR**, **OR**, **Y**);
- регистр начального адреса блока параметров **DMA** передачи (**CP**).

Все каналы для передачи двухмерных массивов имеют регистр **Y**, в котором хранится смещение и размер направления **Y**. Разные типы каналов содержат разный набор регистров.

Исходное состояние регистров **CRS**: разряды `15:0` – нули, а состояние разрядов `31:16` не определено. Исходное состояние остальных регистров не определено.

Индекс содержит адрес 32-разрядного слова в памяти (младшие два разряда индекса должны быть равны нулю). Смещение, умноженное на 4, прибавляется к индексу после

передачи каждого слова данных. Если выполняется обмен данными с *SDRAM*, то смещение прибавляется после передачи каждой пачки 32-разрядных слов, которая передается в режиме "*Burst*". То есть, при обмене данными с *SDRAM*, величина смещения в регистре **OR** должна быть не меньше, чем размер пачки, указанный в поле **WN** регистра **CSR** (**WN**=0, **OR**≥1; **WN**=1, **OR**≥2 и т.д.).

5.2.5. Прерывания DMA

Канал **DMA** формирует прерывание (если установлен соответствующий бит в регистре **MASKR** и бит **IM**[7] в регистре **STATUS** RISC-ядра):

- при единичном состоянии бита **DONE**;
- при единичном состоянии битов **END** и **IM**.

Обнуление битов **DONE** и **END** (и снятие соответствующего прерывания) выполняется посредством чтения содержимого регистра **CSR**. Обнуление бита **DONE** также может быть выполнено посредством записи в него нуля.

5.3. Каналы обмена между внутренней и внешней памятью

5.3.1. Формат регистров MemCh

4 канала **MemCh** обеспечивают обмен данными между внутренней памятью *МУЛЬТИКОР* (MEM, PMEM, XMEM) и внешней памятью.

Формат регистров состояния и управления этих каналов приведен в таблице 4.2:

Таблица 4.2. Формат регистров состояния и управления каналов MemCh.

Номер разряда	Условное обозначение	Назначение
0	RUN	Состояние работы канала DMA: 0 – состояние останова; 1 – состояние обмена данными.
1	DIR	Направление обмена данными: 0 – внутренняя память => внешняя память; 1 – внутренняя память <= внешняя память.
5:2	WN	Число слов данных (пачка), которое передается за одно предоставление прямого доступа: 0 – 1 слово, F – 16 слов. Посредством этого параметра можно плавно изменять приоритет каналов DMA относительно RISC и относительно друг друга.
6	-	Резерв
7	START_DSP	Разрешение запуска работы DSP-ядра (перевод из состояния STOP в состояние RUN) после завершения передачи блока данных: 0 – запуск запрещен; 1 – запуск разрешен.
8	MODE	Режим модификации адреса внутренней памяти: 0 – линейный режим; 1 – режим с реверсивным переносом.
9	2D	Режим модификации адреса внешней памяти: 0 – одномерный режим; 1 – двухмерный режим.
10	MASK	Маска внешнего запроса прямого доступа nDMAR: 0 – запрос запрещен; 1 – запрос разрешен. Если разряд равен нулю, то канал работает только под управлением бита RUN. Если разряд равен 1, то для инициализации канала необходимо также наличие запроса nDMAR (низкий уровень).
11	FLYBY	Признак выполнения обмена между внешней памятью и внешним устройством.
12	CHEN	Признак разрешения самоинициализации (выполнения цепочки DMA передач)
13	IM	Маска прерывания при окончании передачи блока данных: 0 – прерывание запрещено; 1 – прерывание разрешено.
14	END	Признак окончания передачи блока данных
15	DONE	Признак завершения передачи цепочки блоков данных. Аппаратно устанавливается в 1 после завершения передачи цепочки блоков данных (при CHEN=0), при этом бит RUN сбрасывается. Доступен по записи и чтению. Состояние данного бита дублируется в соответствующий бит регистра QSTR
31:16	WCX	Счетчик слов при одномерной адресации. Счетчик числа слов в строке при двухмерной адресации.

Состоянием разряда 0 регистра **CSR** можно управлять, используя адрес псевдорегистра **Run**. При этом остальные разряды этого регистра не изменяются. Эта процедура может быть использована для временной приостановки канала **DMA**.

Примечание: в ИМС МС-24 6й разряд регистра **CSR** (поле EN64) управляет переключением разрядности **DMA**. Если `EN64=0`, обмены **DMA** 32-разрядные, если `EN64=1` - 64-разрядные.

Для задания адресов обмена данными каналы **MemCh** содержат три регистра:
- 32-разрядный регистр индекса и смещения адреса внутренней памяти **IOR**;
- 32-разрядный индексный регистр внешней памяти **IR**;
- 16-разрядный регистр смещения внешней памяти **OR**.

Формат регистра индекса и смещения **IOR_MEM** приведен в таблице 4.3:

Таблица 4.3. Формат регистра IOR_MEM.

Номер разряда	Условное обозначение	Назначение
23:0	ADDR	Адрес внутренней памяти
31:24	OFFSET	Смещение адреса внутренней памяти в 32-разрядных словах после передачи слова данных

Смещение, задаваемое полем **OFFSET**, имеет диапазон от –128 до +127.

При инверсном режиме модификации адреса внутренней памяти, смещение, задаваемое полем **OFFSET**, имеет диапазон от 0 до 255.

Поле **ADDR** в регистре **IOR_MEM** указывает адрес внутренней памяти относительно базового адреса `0x18000000`.

16-разрядный регистр **OR** содержит код смещения внешней памяти в 32-разрядных словах. Он используется всегда. При адресации в двухмерном режиме он указывает смещение в направлении X. Смещение рассматривается как число со знаком в диапазоне от –32768 до +32767.

При работе каналов **MemCh** внешняя память может адресоваться в двухмерном режиме аналогично каналам последовательных портов.

Пример запуска канала DMA на передачу из внутренней памяти во внешнюю находится здесь.

5.3.2. Пример использования

Рассмотрим пример программы, настраивающей канал прямого доступа к памяти **MemCh**. Программа написана на языке *C*:

```
//файл main.c
#include "memory_12.h"
...
extern int OutputArray;

int ExternalArray[256];

main()
{
...
  //блок настройки DMA
  CSR_MemCh0=0x1000000;
  IOR_MemCh0=(1<<24)|((unsigned int)&OutputArray-(unsigned int)0xb8000000);
  OR_MemCh0=0x1;
  IR_MemCh0=((unsigned int)&ExternalArray-0xA0000000)&0xFFFFFFFC;
  Run0=1;
...
}

;файл dsp_data.s
...
   .global OutputArray
...
 .data
...
OutputArray:  .space 256*4,0
...
 .end
```

В файле программы DSP-ядра *dsp_data.s* задается глобальный массив `OutputArray` из 256 слов. В программе RISC массив объявляется как внешний. Приведенная программа осуществляет пересылку содержимого массива `OutputArray` (находящегося во внутренней памяти ядра DSP) в массив `ExternalArray`, расположенный во внешней памяти. Для пересылки используется канал **MemCh0**.

Рассмотрим отдельно каждую строчку блока настройки **DMA**.

Прежде всего, в регистр **CSR** используемого канала пересылается значение `0x1000000`. Первые 16 бит, установленные в 0, означают, что пересылка будет происходить из внутренней памяти во внешнюю память, за одно предоставление доступа будет передаваться одно слово, программа ядра DSP после окончания передачи данных запускаться не будет и т.д. Подробнее о формате регистра **CSR** каналов **MemCh** см. страницу "Формат регистров каналов обмена между внутренней и внешней памятью". Старшие 16 бит установлены в `0x100`, то есть передаваться должны 256 слов.

Далее, происходит настройка регистра **IOR**. Смещение (старшие 8 бит) равно единице, то есть после передачи каждого слова смещение по внутренней памяти будет происходить на 32 разряда. Младшие 24 бита принимают значение `((unsigned int)&OutputArray-(unsigned int)0xb8000000)`, то есть из адреса массива `OutputArray` вычитается значение `0xb8000000`. Это происходит потому, что адреса внутренней памяти (MEM, PMEM, XMEM) в каналах **DMA** следует задавать относительно адреса `0xb8000000`.

После этого в регистр **OR** записывается единица. Таким образом, смещение по внешней памяти также будет происходить на 32 разряда.

Затем, в регистр IR помещается значение `((unsigned int)&ExternalArray-0xA0000000)&0xFFFFFFFC`. Это значение физического адреса массива `ExternalArray` (напомним, что в **DMA** используются физические адреса) с нулями в двух последних битах. Дело в том, что адреса внешней памяти при обмене по каналам **DMA** должны быть 32-разрядными, то есть младшие два бита обязательно обнулять. Поэтому, при обменах с внешней памятью следует обращать внимание на выбранный для обмена стартовый адрес, так как если два последних бита адреса не равны нулю, то это необходимо учесть при выборе числа передаваемых слов чтобы передать массив полностью.

В заключение блока, псевдорегистр **Run** устанавливается в единицу, запуская канал на передачу. Отметим, что использовать псевдорегистр **Run** необязательно, вместо этого можно задать значение регистру **CSR** после остальных регистров канала. То есть, убрав строчку `CSR_MemCh0=0x1000000`, можно вместо строки `Run0=1` записать `CSR_MemCh0=0x1000001`.

Таким образом, канал **MemCh0** запущен на передачу 256 слов из внутренней во внешнюю память. По окончании передачи, бит **Run** регистра **CSR** установится в ноль.

Примечание: настройка регистров каналов **DMA** допускается только из программы ядра RISC. Из ядра DSP возможен только запуск заранее настроенного канала. Пример такого запуска рассматривается на странице "Пример самоинициализации DMA".

Примечание 2: в данном примере использован файл *memory_12.h* для обращения к регистрам канала **DMA** по заданным в этом файле символам. Подробнее о заданных условных именах регистров в файле *memory_12.h* см. страницы "Регистры RISC" и "Доступ к регистрам DSP-ядра".

5.4. DMA последовательных портов

5.4.1. Формат регистров SpT, SpR

Для обслуживания последовательных портов имеется 4 канала **DMA**: **SportTxCh0**, **SportRxCh0**, **SportTxCh1**, **SportRxCh1** (отдельно на передачу и прием соответственно).

Формат регистров управления и состояния **CSR_SpRx0**, **CSR_SpTx0**, **CSR_SpTx1**, **CSR_SpRx1** каналов **DMA** последовательных портов приведен в таблице 4.4:

Таблица 4.4. Формат регистров управления и состояния последовательных портов

Номер разряда	Условное Обозначение	Назначение
0	RUN	Состояние работы канала DMA: 0 – состояние останова; 1 – состояние обмена данными.
1-8	-	Резерв
9	2D	Режим модификации адреса памяти: 0 – одномерный режим; 1 – двухмерный режим.
11,10	-	Резерв
12	CHEN	Признак разрешения самоинициализации (выполнения цепочки DMA передач)
13	IM	Маска прерывания при окончании передачи блока данных: 0 – прерывание запрещено; 1 – прерывание разрешено.
14	END	Признак окончания передачи блока данных
15	DONE	Признак завершения передачи цепочки блоков данных. Аппаратно устанавливается в 1 после завершения передачи данных (при CHEN=0), при этом бит RUN сбрасывается. Доступен по записи и чтению. Состояние данного бита дублируется в соответствующий бит регистра QSTR
31:16	WCX	Счетчик слов при одномерной адресации. Счетчик числа слов в строке при двухмерной адресации.

Для задания адреса памяти (внутренней или внешней) каналы **DMA** последовательных портов содержат два регистра:

- 32-разрядный индексный регистр памяти **IR**;
- 16-разрядный регистр смещения памяти **OR**.

16-разрядный регистр **OR** содержит код смещения памяти в 32-разрядных словах. Он используется всегда. При адресации в двухмерном режиме он указывает смещение в направлении X. Смещение рассматривается как число со знаком в диапазоне от –32768 до +32767.

При работе каналов последовательных портов память (внутренняя или внешняя) может адресоваться в двухмерном режиме. Для этого имеется 32-разрядный регистр **Y**, формат которого приведен в таблице 4.5:

Таблица 4.5. Формат регистра Y.

Номер разряда	Условное Обозначение	Назначение
15:0	OY	Смещение (приращение) адреса памяти в 32-разрядных словах по направлению Y. Используется только при двухмерной адресации.
31:16	WCY	Число строк по Y направлению. Используется только при двухмерной адресации.

При двухмерном режиме адресации поле **WC** регистра **CSR** содержит число слов в строке (направление X), а регистр **Y** (поле **YWC**) содержит число строк (направление Y). Двухмерная адресация выполняется следующим образом:

- Содержимое счетчика **WC** сохраняется в буферном регистре;
- 1 цикл. Индексный регистр внешней памяти модифицируется с использованием смещения **OR_MEM**. Счетчик **WC** декрементируется. Если он равен 0, то переход ко второму циклу.
- 2 цикл. Состояние счетчика **WC** восстанавливается из буферного регистра. Индексный регистр внешней памяти модифицируется с использованием смещения **YO**. Счетчик **YWC** декрементируется. Если он не равен 0, то переход к первому циклу. Если он равен нулю, то работа канала завершается.

5.4.2. Пример использования

Рассмотрим пример программы, написанной на языке *C* и настраивающей канал **DMA** для обмена данными с портом **Sport0**:

```
#include "memory_12.h"

void exit();

int InputArray[256];
int OutputArray[256];

main()
{
      int i;
      for (i=0; i<256; i++)
          OutputArray[i]=i;

      TDIV0=0x1f0001;
      RDIV0=0x1f0001;
      STCTL0=0x65f1;
      SRCTL0=0x21f1;

      IR_SpTx0=((unsigned int)&OutputArray[255]-0xA0000000);
      OR_SpTx0=-1;
      IR_SpRx0=((unsigned int)&InputArray-0xA0000000);
      OR_SpRx0=1;
      CSR_SpTx0=0x1000001;
```

```
        CSR_SpRx0=0x1000001;

        exit();
}

void exit()
{
     while(1);
}
```

Приведенная программа осуществляет пересылку 256 элементов массива `OutputArray` в массив `InputArray` в обратном порядке. Для передачи и приема элементов используется порт **Sport0**. Для симуляции процесса передачи-приема данных используется устройство заглушки портов ввода-вывода, установленное на порт **Sport0**. Суть устройства в том, что все данные, попадающие в буфер передачи порта, отправляются в буфер приема. Подключение устройства заглушки рассматривается в книге "**MC Studio User's Guide**".

Пересылка данных из внешней памяти в буфер передачи осуществляется каналом **DMA SpTx0**, а пересылка из буфера приема во внешнюю память - каналом **SpRx0**.

Для осуществления пересылки необходимо настроить порт и каналы **DMA**. В первую очередь, настраиваются регистры коэффициентов деления частоты для передачи и приема (**TDIV0**, **RDIV0**). Затем устанавливаются значения регистров управления передачей и приемом (**STCTL0**, **SRCTL0**), причем нулевой бит, установленный в единицу означает разрешение передачи и приема соответственно. Подробнее порт **Sport** рассматривается в разделе "Порты ввода-вывода".

После настройки порта, осуществляется настройка каналов **DMA** на обмен данными с портом. Канал пересылки данных в буфер передачи настраивается на пересылку 256 слов, начиная с последнего элемента массива `OutputArray` со смещением -1 (то есть к нулевому элементу). Канал пересылки данных из буфера приема во внешнюю память настраивается на пересылку 256 элементов в массив `InputArray`. Напомним, что при настройке каналов **DMA** необходимо указывать физические адрес, поэтому из виртуальных адресов вычитается значение `0xA0000000`. Описание каналов **DMA** последовательных портов приведено на странице "Формат регистров SpT, SpR".

В завершении программы выполняется бесконечный цикл, вставленный для ожидания результатов пересылки.

Примечание: настройка регистров каналов **DMA** допускается только из программы ядра RISC. Из ядра DSP возможен только запуск заранее настроенного канала. Пример такого запуска рассматривается на странице "Пример самоинициализации DMA".

Примечание 2: в данном примере использован файл *memory_12.h* для обращения к регистрам канала **DMA** по заданным в этом файле символам. Подробнее о заданных условных именах регистров в файле *memory_12.h* см. страницы "Регистры RISC" и "Доступ к регистрам DSP-ядра".

5.5. DMA линковых портов

5.5.1. Формат регистров LpCh

Для обслуживания линковых портов имеется 4 канала **DMA**: **LportCh0**, **LportCh1**, **LportCh2**, **LportCh3**.

Формат регистров управления и состояния **CSR_Lp0**, **CSR_Lp1**, **CSR_Lp2**, **CSR_Lp3** каналов **DMA** линковых портов приведен в таблице 4.6:

Таблица 4.6. Формат регистров управления и состояния линковых портов

Номер разряда	Условное Обозначение	Назначение
0	RUN	Состояние работы канала DMA: 0 – состояние останова; 1 – состояние обмена данными.
8:1	-	Резерв
9	2D	Режим модификации адреса памяти: 0 – одномерный режим; 1 – двухмерный режим.
11:10	-	Резерв
12	CHEN	Признак разрешения самоинициализации (выполнения цепочки DMA передач)
13	IM	Маска прерывания при окончании передачи блока данных: 0 – прерывание запрещено; 1 – прерывание разрешено.
14	END	Признак окончания передачи блока данных
15	DONE	Признак завершения передачи цепочки блоков данных. Аппаратно устанавливается в 1 после завершения передачи данных (при CHEN=0), при этом бит RUN сбрасывается. Доступен по записи и чтению. Состояние данного бита дублируется в соответствующий бит регистра QSTR
31:16	WCX	Счетчик слов при одномерной адресации. Счетчик числа слов в строке при двухмерной адресации.

Для задания адреса памяти (внутренней или внешней) каналы **DMA** линковых портов содержат два регистра:

- 32-разрядный индексный регистр памяти **IR**;
- 16-разрядный регистр смещения памяти **OR**.

16-разрядный регистр **OR_MEM** содержит код смещения памяти в 32-разрядных словах. Он используется всегда. При адресации в двухмерном режиме он указывает смещение в направлении X. Смещение рассматривается как число со знаком в диапазоне от –32768 до +32767.

При работе каналов **LportCh** внешняя память может адресоваться в двухмерном режиме аналогично каналам последовательных портов.

5.5.2. Пример использования

Рассмотрим пример программы, написанной на языке *C* и настраивающей каналы **DMA** на обмен данными с портами **Lport0** и **Lport1**.

```
#include "memory_12.h"

void exit();

int InputArray[256];
int OutputArray[256];

main()
{
      int i;
      for (i=0; i<256; i++)
          OutputArray[i]=i;

      LDIR0=0x3ff;
      LDIR1=0x300;

      LCSR0=0x47;
      LCSR1=0x45;

      IR_LpCh0=((unsigned int)&OutputArray[255]-0xA0000000);
      OR_LpCh0=-1;
      IR_LpCh1=((unsigned int)&InputArray-0xA0000000);
      OR_LpCh1=1;

      CSR_LpCh0=0x1000001;
      CSR_LpCh1=0x1000001;

      exit();
}

void exit()
{
     while(1);
}
```

Данная программа идентична программе примера работы **DMA** с портом **Sport**, приведенном на странице "Пример использования DMA последовательных портов". Разница заключается в использованных портах, а также - в том, что в данной программе для приема и передачи использованы два порта.

5.6. Самоинициализация DMA

Все каналы **DMA** могут выполнять процедуру самоинициализации (выполнение цепочки передач **DMA**).

Для выполнения самоинициализации в каналах имеется 16-разрядный регистр **CP**, в котором хранится начальный адрес блока параметров очередного **DMA**-обмена. Эти параметры при самоинициализации аппаратно загружаются в соответствующие регистры канала **DMA**. Процедура этой загрузки ничем не отличается от обычного **DMA**-обмена. Блок параметров может размещаться только во внутренней памяти **MEM**.

Блоки параметров, размещаемых в памяти, имеют следующую структуру (в порядке возрастания адресов):

- каналы последовательных портов и линковых портов – **IR**, **OR**, **Y**, **CP**, **CSR**;
- каналы **MemCh** – **IOR**, **IR**, **OR**, **Y**, **CP**, **CSR**.

Параметры, соответствующие 16-разрядным регистрам, размещаются в младших разрядах памяти. В слове памяти, соответствующем регистру **CSR** должно быть: **RUN**=1, **DONE**=0. Если необходимо продолжить цепочку команд, то необходимо указать **CHEN**=1.

Для запуска работы канала **DMA** в режиме с самоинициализацией необходимо в регистр **CP** записать адрес первого блока параметров **DMA**-передачи. При этом 31-й разряд записываемых данных должен содержать 1 (признак пуска самоинициализации). В результате этого, соответствующий канал загрузит в свои регистры параметры **DMA**-передачи и начнет обмен данными. После окончания передачи, бит **END** в регистре **CSR** блока данных устанавливается в единичное состояние. Также, если бит **IM**=1, выдается прерывание. После этого канал проверяет состояние бита **CHEN**. Если он равен 1, то будет загружен следующий блок параметров **DMA**-передачи и т.д. В противном случае цепочка **DMA**-обменов закончится и в регистре **CSR** бит **DONE** установится в единичное состояние.

При необходимости каналы **DMA** могут инициализироваться программно. Для этого RISC должен загрузить все необходимые регистры индекса и смещения, а затем регистр **CSR**. При загрузке регистра **CSR** бит **RUN** необходимо установить в единичное состояние. Следует отметить, что бит **RUN** может быть использован для приостановки канала **DMA**. Для этого в любой момент времени в него необходимо записать 0. Примеры программной инициализации **DMA**-каналов рассматриваются в следующих разделах:

- Пример использования DMA последовательных портов;
- Пример использования DMA линковых портов;
- Пример использования DMA MemCh.

Программу настройки самоинициализации **DMA** можно найти в разделе "Пример самоинициализации DMA".

5.7. Пример самоинициализации DMA

Рассмотрим пример программы, настраивающей каналы **DMA MemCh0** и **SportTxCh0** на самоинициализацию. Данная программа осуществляет пересылку четырех блоков данных из внутренней памяти (**XRAM**) во внешнюю при помощи канала **MemCh0**, а затем - вывод этих блоков посредством порта **Sport0**. Пересылка данных в буфер передачи порта осуществляется посредством канала **DMA SportTxCh0**. Для приема данных из порта используется устройство *ReadDevice32* библиотеки **Devcore**, позволяющее выводить в файл данные, поступающие из порта. Подробнее библиотека **Devcore** рассматривается в книге "**MC Studio. User's guide**".

Прежде всего, для удобства работы с программой, в файле *main.h* задаются два типа структур:

```
//файл main.h
typedef struct {
        int _IOR;
        int _IR;
        int _OR;
        int _Y;
        int _CP;
        int _CSR;
} TMemChSelfInitTable;

typedef struct {
        int _IR;
        int _OR;
        int _Y;
        int _CP;
        int _CSR;
} TPortChSelfInitTable;
```

Эти типы используются для заполнения блоков самоинициализации каналов **MemCh** и каналов **DMA** параллельных и последовательных портов.

Теперь рассмотрим текст программы для RISC:

```
//файл main.c
#include "memory_12.h"
#include "main.h"

/*
  физический адрес размещения первого блока
  самоинициализации MemCh0 в памяти MEM
*/
#define MChInitMEM 0x18000000

/*
  физический адрес размещения первого блока
  самоинициализации SportTxCh0 в памяти MEM
*/
```

```
#define PChInitMEM 0x18000060

TMemChSelfInitTable  MChInitTable;
TPortChSelfInitTable PChInitTable;

extern int Output1[256];
extern int Output2[256];
extern int Output3[256];
extern int Output4[256];

int Input1[256];
int Input2[256];
int Input3[256];
int Input4[256];

void CopySelfInitTable(void *ATable, void *VMA, bool IsMemCh);
void exit();

main()
{
	int BranchesCounter=0;

	//заполнение первого блока самоинициализации MemCh0
	MChInitTable._IOR=(1<<24)|((unsigned int)&Output2-(unsigned int)0xb8000000);
	MChInitTable._IR=((unsigned int)&Input1-(unsigned int)0xA0000000)&0xFFFFFFFC;
	MChInitTable._OR=1;
	MChInitTable._Y=0;
	MChInitTable._CP=(MChInitMEM+(6<<2))|0x80000000;
	MChInitTable._CSR=0x1001080;

	//копирование первого блока MemCh0 в MEM
	CopySelfInitTable(&MChInitTable,&MEM,1);

	//старт самоинициализации MemCh0
	CP_MemCh0=MChInitMEM|0x80000000;

	//инициализация буфера передачи порта Sport0
	TDIV0=0x1f0001;
	STCTL0=0x65f1;

	//запуск DSP
	SR=0;
	SAR=0xFFFF;
	PC=0;
	DCSR=0x4000;

	//заполнение первого блока самоинициализации SportTxCh0
	PChInitTable._IR=((unsigned int)&Input1-(unsigned int)0xA0000000);
	PChInitTable._OR=1;
	PChInitTable._Y=0;
	PChInitTable._CP=(PChInitMEM+(5<<2))|0x80000000;
	PChInitTable._CSR=0x1001000;

	//копирование первого блока SportTxCh0 в MEM
	CopySelfInitTable(&PChInitTable, (void *)((unsigned int)&MEM+(24<<2)),0);
```

```
//старт самоинициализации SportTxCh0
CP_SpTx0=(PChInitMEM)|0x80000000;

//заполнение второго блока самоинициализации MemCh0
MChInitTable._IOR=(1<<24)|((unsigned int)&Output3-(unsigned int)0xb8000000);
MChInitTable._IR=((unsigned int)&Input2-(unsigned int)0xA0000000)&0xFFFFFFFC;
MChInitTable._OR=1;
MChInitTable._Y=0;
MChInitTable._CP=(MChInitMEM+(12<<2))|0x80000000;
MChInitTable._CSR=0x1001080;

//копирование второго блока MemCh0 в MEM
CopySelfInitTable(&MChInitTable,(void *)((unsigned int)&MEM+(6<<2)),1);

//заполнение второго блока самоинициализации SportTxCh0
PChInitTable._IR=((unsigned int)&Input2-(unsigned int)0xA0000000);
PChInitTable._OR=1;
PChInitTable._Y=0;
PChInitTable._CP=(PChInitMEM+(10<<2))|0x80000000;
PChInitTable._CSR=0x1001000;

//копирование второго блока SportTxCh0 в MEM
CopySelfInitTable(&PChInitTable, (void *)((unsigned int)&MEM+(29<<2)),0);

//заполнение третьего блока самоинициализации MemCh0
MChInitTable._IOR=(1<<24)|((unsigned int)&Output4-(unsigned int)0xb8000000);
MChInitTable._IR=((unsigned int)&Input3-(unsigned int)0xA0000000)&0xFFFFFFFC;
MChInitTable._OR=1;
MChInitTable._Y=0;
MChInitTable._CP=(MChInitMEM+(18<<2))|0x80000000;
MChInitTable._CSR=0x1001080;

//копирование третьего блока MemCh0 в MEM
CopySelfInitTable(&MChInitTable,(void *)((unsigned int)&MEM+(12<<2)),1);

//заполнение третьего блока самоинициализации SportTxCh0
PChInitTable._IR=((unsigned int)&Input3-(unsigned int)0xA0000000);
PChInitTable._OR=1;
PChInitTable._Y=0;
PChInitTable._CP=(PChInitMEM+(15<<2))|0x80000000;
PChInitTable._CSR=0x1001000;

//копирование третьего блока SportTxCh0 в MEM
CopySelfInitTable(&PChInitTable, (void *)((unsigned int)&MEM+(34<<2)),0);

//заполнение четвертого блока самоинициализации MemCh0
MChInitTable._IOR=(1<<24)|((unsigned int)&Output1-(unsigned int)0xb8000000);
MChInitTable._IR=((unsigned int)&Input4-(unsigned int)0xA0000000)&0xFFFFFFFC;
MChInitTable._OR=1;
MChInitTable._Y=0;
MChInitTable._CP=0;
MChInitTable._CSR=0x1000000;

//копирование четвертого блока MemCh0 в MEM
CopySelfInitTable(&MChInitTable,(void *)((unsigned int)&MEM+(18<<2)),1);
```

```
        //заполнение четвертого блока самоинициализации SportTxCh0
        PChInitTable._IR=((unsigned int)&Input4-(unsigned int)0xA0000000);
        PChInitTable._OR=1;
        PChInitTable._Y=0;
        PChInitTable._CP=0;
        PChInitTable._CSR=0x1000000;

        //копирование четвертого блока SportTxCh0 в MEM
        CopySelfInitTable(&PChInitTable, (void *)((unsigned int)&MEM+(39<<2)),0);

        while ((BranchCounter!=4)||(CSR_SpTx0&1))
        {
              //если программа DSP остановлена и канал передачи не занят,
осуществляется вывод очередного блока
             if (((QSTR)&(1<<31))&&((~CSR_SpTx0)&1)&&(BranchesCounter<4))
             {
                CSR_SpTx0 |= 1;
                BranchesCounter++;
             }
        };
        exit();
}

/*
  функция копирует структуру типа TPortChSelfInitTable (IsMemCh=0) или
TMemChSelfInitTable (IsMemCh=1)
  по виртуальному адресу VMA (подразумевается адрес в памяти MEM)
*/
void CopySelfInitTable(void *ATable, void *VMA, bool IsMemCh)
{
      unsigned int *source;
      unsigned int *dest;
      int i;

      source=ATable;
      dest=VMA;

      if (IsMemCh)
         i=0;
      else
         i=1;

      for(; i<6;i++)
            *dest++=*source++;
}

void exit()
{
      while(1);
}
```

Последовательность действий, выполняемых данной программой, такова:

- Сначала, программа заполняет и копирует в память **MEM** первый блок самоинициализации канала **MemCh0**, после чего запускает самоинициализацию (без запуска самого канала **DMA**). Для копирования таблицы самоинициализации (на этом

этапе и далее) в память **MEM** используется функция **CopySelfInitTable**. Также программа осуществляет настройку порта **Sport0** для осуществления последующего вывода данных.

- Затем, программа RISC запускает на исполнение программу DSP, осуществляющую генерацию очередного блока данных и запуск канала **DMA** для передачи этого блока во внешнюю память. После запуска канала, программа DSP останавливается до завершения передачи сгенерированного блока. Повторный запуск DSP осуществляется автоматически каналом **MemCh0**.
- После запуска программы DSP, программа RISC заполняет и копирует в память **MEM** блоки самоинициализации канала **SportTxCh0**, а также оставшиеся блоки самоинициализации канала **MemCh0**.
- После копирования всех блоков, программа входит в цикл ожидания, продолжающийся до тех пор, пока не будет завершен вывод четвертого блока.

Массивы `Input1`, `Input2`, `Input3` и `Input4` используются для приема данных из внешних массивов `Output1`, `Output2`, `Output3` и `Output4` по каналу **MemCh0**.

Рассмотрим программу DSP:

```
//файл calc.s
 .text
  .global Output1
  .global Output2
  .global Output3
  .global Output4
DSP_prepare_to_run:
                    CLRL    R0                  Output1,A0
                    DO 256, ptr_EndDo
                            MOVE R0,(A0)+
                    ptr_EndDo:
                            INCL R0,R0

                    MOVE    Output1,A0
                    MOVE    Output2,A2
                    MOVE    0x3ff,M0
                    MOVE    0x3ff,M2

DSP_creating_output:

                     MOVE    coeffs,A1

                     DO 256, co_EndDo
                                     MOVE    R4,(A2)+
                                     MOVE    (A1)+,R2
                     co_EndDo:
                                     MPYL    R0,R2,R4        (A0)+,R0

                     MOVE    0x10,DCSR
                     J       DSP_creating_output

 .data
Output1: .space 256*4,0
Output2: .space 256*4,0
Output3: .space 256*4,0
```

```
Output4: .space 256*4,0

coeffs:
          .word  6,0,-7,51,100,54,-129,515,1196,499,-1379,1783,8909,1489,-9834,18747
          .word  6,0,-8,50,86,55,-145,519,1106,487,-1469,1909,8447,1322,-10293,18713
          .word  5,0,-9,49,73,56,-161,521,1015,473,-1559,2022,7986,1140,-10751,18657
          .word  4,0,-10,47,61,56,-177,521,926,455,-1647,2123,7527,944,-11205,18578
          .word  4,0,-11,45,49,57,-194,518,837,434,-1733,2209,7072,733,-11654,18476
          .word  4,0,-12,44,38,57,-211,514,750,410,-1817,2284,6620,509,-12097,18353
          .word  3,0,-13,42,27,56,-229,507,665,383,-1899,2347,6173,270,-12534,18208
          .word  3,0,-14,40,18,56,-247,500,582,353,-1977,2398,5732,17,-12963,18042
          .word  2,0,-15,38,8,55,-266,490,501,320,-2052,2437,5297,-249,-13383,17855
          .word  2,0,-17,36,0,53,-284,479,422,282,-2122,2465,4869,-530,-13794,17647
          .word  2,0,-18,34,-7,51,-302,467,346,242,-2188,2483,4449,-825,-14194,17419
          .word  1,0,-19,32,-14,50,-320,454,273,198,-2249,2491,4038,-1133,-14583,17172
          .word  1,0,-21,31,-20,47,-339,439,203,151,-2304,2489,3636,-1454,-14959,16907
          .word  1,0,-22,29,-26,44,-357,424,136,100,-2354,2478,3244,-1788,-15322,16623
          .word  1,-1,-24,27,-31,40,-374,408,71,46,-2396,2459,2863,-2135,-15671,16322
          .word  -1,-25,36,-391,-11,-2431,-2493,-16004,16004,2493,2431,11,391,-36,25,1
 .end
```

Приведенная программа выполняет следующие действия:

- Заполнение первого блока числами от 0 до 255;
- Генерация очередного блока посредством умножения элементов предыдущего блока на некоторые коэффициенты;
- Запуск канала **DMA** для вывода сгенерированного блока.

При запуске канала **DMA** (`MOVE 0x10,DCSR`) бит **Run** устанавливается в нуль, то есть происходит останов программы. Запуск программы DSP осуществляется каналом **MemCh0** по окончании пересылки каждого блока.

Примечание: в данном примере использован файл *memory_12.h* для обращения к регистрам канала **DMA** по заданным в этом файле символам. Подробнее о заданных условных именах регистров в файле *memory_12.h* см. страницы "Регистры RISC" и "Доступ к регистрам DSP-ядра".

6. Системный управляющий сопроцессор

6.1. Системный управляющий сопроцессор: Введение

Системный Управляющий Сопроцессор (**CP0**) обеспечивает регистровый интерфейс с процессорным ядром MIPS32 и поддерживает управление памятью, преобразование адреса, обработку исключений и другие привилегированные операции. Каждому регистру **CP0** соответствует определяющий его уникальный номер; этот номер называется *номером регистра*. Например, регистру **PageMask** соответствует 5-й номер регистра.

Обмен данными между **CPU** и **CP0** осуществляется посредством команд **`mtc0`** (*Move To CP0*) и **`mfc0`** (*Move From CP0*) с указанием номера регистра.

После записи нового значения в один из регистров **CP0**, его обновление происходит не сразу, а по прошествии периода от 0 и более команд. Этот период называется *периодом особой ситуации*.

В данной главе детально рассматривается описание регистров CP0.

6.2. Регистры CP0

В таблице 5.1 приведен перечень регистров **CP0** в порядке возрастания номера регистра.

Таблица 5.1. Регистры CP0.

Номер регистра	Название регистра	Функция
0	Index	Индекс матрицы TLB (режим TLB)
1	Random	Случайным образом сгенерированный индекс для буфера TLB (режим TLB)
2	EntryLo0	Младшая часть строки TLB для виртуальных страниц с четными номерами (режим TLB)
3	EntryLo1	Младшая часть строки TLB для виртуальных страниц с нечетными номерами (режим TLB)
4	Context	Указатель на строку в таблице страниц памяти (режим TLB)
5	PageMask	Управление переменным размером страниц строк TLB (режим TLB)
6	Wired	Управление количеством закрепленных "привязанных" строк TLB (режим TLB)
7	Reserved	Резерв
8	BadVAddr	Содержит адрес, вызвавший последнее связанное с адресацией исключение
9	Count	Счетчик процессорных циклов
10	EntryHi	Старшая часть строки TLB (режим TLB)

11	Compare	Управление прерыванием таймера
12	Status	Состояние и управление процессором
13	Cause	Причина последнего исключения
14	EPC	Значение счетчика команд во время последнего исключения
15	PRId	Идентификация и ревизия процессора
16	Config/Config1	Конфигурационный регистр
17	LLAddr	Загрузка адреса сопряжения
18-19	Не реализованы	
20-22	Reserved	Резерв
23-24	Не реализованы	
25-27	Reserved	Резерв
28-29	Не реализованы	
30	ErrorEPC	Значение счетчика команд при последней ошибке
31	Не реализован	

Регистры **CP0** обеспечивают интерфейс между системой команд (**ISA**) и архитектурой процессора. Каждый регистр, описанный в данном разделе, представлен своим порядковым номером и значением поля *select*.

Все поля описанных регистров характеризуются свойствами записи/чтения, а также значением после аппаратного сброса. Свойства записи/чтения охарактеризованы в таблице 5.2.

Таблица 5.2. Свойства записи/чтения

Свойства записи/чтения	Аппаратная интерпретация	Программная интерпретация
R/W	Поле, в котором все биты программно и аппаратно доступны по записи и чтению. Аппаратное обновление этого поля доступно для программы при чтении программой. Программное обновление этого поля доступно для процессора при чтении процессором. Если значение поля после сброса не определено, программа или процессор должны проинициализировать это поле, чтобы первое чтение возвратило предсказуемое значение.	
R	Поле, значение которого постоянно или обновляется только процессором. Значение поля после начальной установки восстанавливается также при включении питания. Если значение поля не определено после начальной установки, процессор обновляет его только при условиях, определенных при описании поля.	Поле, для которого значение, записанное программой, процессором игнорируется. Программное прочтение этого поля возвращает последнее обновленное процессором значение. Если значение поля не определено после начальной установки, программное прочтение этого поля возвратит непредсказуемое значение кроме тех случаев, когда произошло обновление процессором значения этого поля по возникновению условий, определенных в описании поля условий.
0	Поле, значение которого процессором не обновляется и всегда равно нулю.	Программное чтение всегда возвращает нуль.

6.3. Регистр Index

Регистр **Index** является 32-х разрядным регистром, доступным для чтения и записи. Он содержит индекс доступа к **TLB** для команд **`tlbp`**, **`tlbr`** и **`tlbwi`**. Ширина поля индекса зависит от количества строк **TLB** и равна 4.

Функционирование процессора *не определено*, если в регистр **Index** записано значение большее или равное количеству строк **TLB**.

В таблице 5.3 приводится формат регистра **Index**:

Таблица 5.3. Формат регистра Index.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	**Биты**			
P	31	Неудачная проба. Устанавливается в 1, если предыдущей командой TLBProbe (TLBP) не было найдено соответствия в TLB.	R	Не определено
0	30:4	При чтении возвращается нуль	0	0
Index	3:0	Индекс строки TLB, к которой относятся команды TLBRead и TLBWrite	R/W	Не определено

6.4. Регистр Random

Регистр **Random** доступен только для чтения. Его значение используется как индекс **TLB** для команды **`tlbwr`**. Ширина поля *Random* определяется таким же образом, как для регистра Index.

Значение этого регистра изменяется между верхней и нижней границами следующим образом:

- Нижняя граница определяется количеством строк **TLB** , зарезервированных для использования операционной системой (содержимое регистра Wired). Строка, чей индекс равен значению **Wired**, является первой из доступных для записи командой *TLB Write Random* (**`tlbwr`**).
- Верхняя граница равна общему количеству строк **TLB** минус 1.

Регистр **Random** уменьшается на 1 при продвижении конвейера RISC, возвращаясь к максимальному значению по достижению величины, равной значению регистра Wired.

Процессор инициализирует регистр **Random** значением, равным верхней границе по возникновению исключения Reset и по записи в регистр Wired.

В таблице 5.4 приводится формат регистра **Random**:

Таблица 5.4. Формат регистра Random.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	**Биты**			
0	31:4	При чтении возвращается нуль	0	0
Random	3:0	Случайный индекс строки TLB	R	TLB Entries - 1

6.5. Регистры EntryLo0, EntryLo1

Пара регистров **EntryLo** действует как интерфейс между **TLB** и командами **`tlbr`**, **`tlbwi`**, **`tlbwr`**.

В режиме **TLB EntryLo0** содержит строки для четных страниц **TLB**, а **EntryLo1** – для нечетных страниц.

После ошибки адресации и возникновения исключений TLB refill, TLB invalid и TLB modified, содержимое регистров **EntryLo0** и **EntryLo1** не определено.

В таблице 5.5 отображен формат регистров **EntryLo0** и **EntryLo1**:

Таблица 5.5. Формат регистров EntryLo0 и EntryLo1.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
R	31:30	Резервные. При чтении возвращается нуль	R	0
0	29:26	При чтении возвращается нуль	R	0
PFN	25:6	Номер страничного кадра. Соответствует битам 31:12 физического адреса.	R/W	Не определено
0	5:4	Не используются. При чтении возвращается нуль	R	0
C	5:3	Атрибут когерентности страницы. См. табл.2.18.	R/W	Не определено
D	2	"Dirty" – бит, разрешающий запись. Указывает на то, что в страницу была сделана запись, и/или страница открыта для записи. Если этот бит равен 1, разрешается сохранение в этой странице. Если он равен 0, сохранение в этой странице вызывает исключение TLB Modified.	R/W	Не определено
V	1	Бит валидности. Указывает, на то, что строка TLB и, соответственно, отображение виртуальной страницы, является действительным. Если этот бит равен 1, доступ к странице разрешается. Если этот бит равен 0, доступ к странице вызывает исключение TLB Invalid.	R/W	Не определено
G	0	Бит глобальности. При записи в TLB битом G в строке TLB становится логическое "И" битов G EntryLo0 и EntryLo1. Если бит G строки TLB равен 1, результат сравнения полей ASID игнорируется при поиске по TLB. При чтении строки TLB биты G EntryLo0 и EntryLo1 отражают состояние бита G TLB.	R/W	Не определено

6.6. Регистр Context

Регистр **Context** доступен для чтения и записи. В нем содержится указатель на строку в матрице **PTE** (*page table entry*). Эта матрица является структурой данных операционной системы, в которой содержатся преобразования виртуального адреса в физический. При возникновении промаха **TLB**, операционная система загружает в **TLB** недостающее преобразование из матрицы **PTE**. Регистр **Context** дублирует часть информации, содержащейся в регистре BadVAddr, но организован таким образом, что операционная система может прямо ссылаться к 8-байтной матрице **PTE** в памяти.

При возникновении исключения **TLB** (TLB Refill, TLB Invalid, или TLB Modified) биты $VA_{31:13}$ виртуального адреса записываются в поле BadVPN2 регистра **Context**. Поле PTEBase записывается и используется операционной системой.
После возникновения исключения ошибки адресации значение поля BadVPN2 регистра **Context** не определено.

Таблица 5.6 отображает формат регистра **Context**:

Таблица 5.6. Формат регистра Context.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
PTEBase	31:23	Это поле используется операционной системой и обычно содержит значение, позволяющее операционной системе использовать регистр Context в качестве указателя на текущую матрицу PTE в памяти.	R/W	Не определено
BadVPN2	22:4	Это поле заполняется процессором при промахе TLB. Оно содержит биты $VA_{31:13}$ пропущенного виртуального адреса	R	Не определено
0	3:0	При чтении возвращается нуль	0	0

6.7. Регистр PageMask

Регистр **PageMask** доступен для чтения и записи, и используется для чтения **TLB** и записи в **TLB**. Он содержит маску сравнения, которая устанавливает переменную размера страниц для каждой строки **TLB**, как показано в таблице 5.7. Если значение регистра отлично от значений, приведенных в таблице, поведение процессора при поиске по **TLB** не определено.

Таблица 5.7. Размер страницы для строки TLB.

Размер страницы	Бит											
	24	**23**	**22**	**21**	**20**	**19**	**18**	**17**	**16**	**15**	**14**	**13**
4 КБайт	0	**0**	0	0	0	0	0	0	0	0	0	0
16 Кбайт	0	0	0	0	0	0	0	0	0	0	1	1
64 Кбайт	0	0	0	0	0	0	0	0	1	1	1	1
256 Кбайт	0	0	0	0	0	0	1	1	1	1	1	1
1 Мбайт	0	0	0	0	1	1	1	1	1	1	1	1
4 Мбайт	0	0	1	1	1	1	1	1	1	1	1	1
16 Мбайт	1	1	1	1	1	1	1	1	1	1	1	1

В таблице 5.8 приводится формат регистра **PageMask**.

Таблица 5.8. Формат регистра PageMask.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	**Биты**			
Mask	24:13	Бит маски, содержащий "1", указывает на то, что соответствующий бит виртуального адреса не должен принимать участие при поиске соответствия по TLB	R/W	Не определено
0	31:25, 12:0	При чтении возвращается нуль	0	0

6.8. Регистр Wired

Регистр **Wired** доступен для чтения и записи. Этот регистр определяет границу между случайными и "привязанными" строками **TLB**, как показано на рисунке 5.1. Ширина поля **Wired** определяется так же, как для регистра Index. "Привязанные" строки зафиксированы – они не являются удаляемыми и не могут быть перезаписаны командой **`tlbwr`**. Эти строки могут быть перезаписаны только командой **`tlbwi`**.

Регистр **Wired** устанавливается в нулевое состояние исключением по аппаратному сбросу (Reset). Запись в регистр **Wired** вызывает установку регистра Random в значение, равное его верхней границе.

Если значение, записанное в регистр **Wired**, больше или равно числу строк **TLB**, операция процессора не определена.

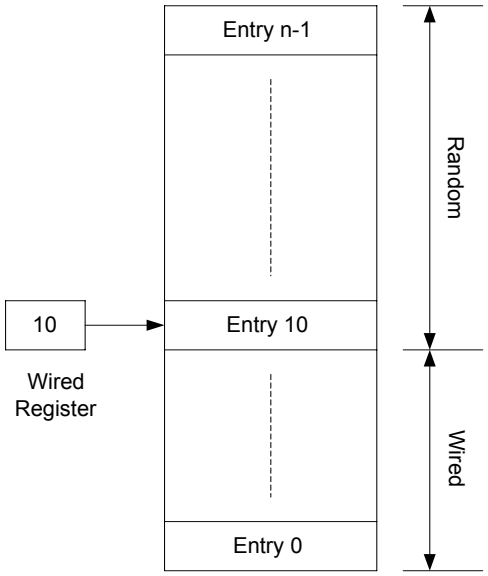


Рисунок 5.1. Граница между случайными и "привязанными" строками TLB.

В таблице 5.9 приведен формат регистра **Wired**:

Таблица 5.9. Формат регистра Wired.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
0	31:4	При чтении возвращается нуль	0	0
Wired	3:0	Граница между “привязанными” и случайными строками TLB.	R/W	0

6.9. Регистр BadVAddr

Регистр **BadVAddr** доступен только для чтения и содержит последний виртуальный адрес, вызвавший одно из следующих исключений:

- Ошибка адреса (**AdEL** или **AdES**)
- TLB Refill
- TLB Invalid
- TLB Modified

В таблице 5.10 приведен формат регистра **BadVAddr**:

Таблица 5.10. Формат регистра BadVAddr.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
BadVAddr	31:0	Виртуальный адрес, вызвавший исключение	R	Не определено

6.10. Регистр Count

Регистр **Count** действует как таймер, увеличивающий свое значение каждый такт.

Регистр может быть записан в функциональных или диагностических целях, включая установку или синхронизацию процессора.

Таблица 5.11 иллюстрирует формат регистра **Count**:

Таблица 5.11. Формат регистра Count.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
Count	31:0	Счетчик	R/W	Не определено

6.11. Регистр EntryHi

Регистр **EntryHi** содержит информацию соответствия виртуального адреса, использующуюся при чтении, записи и операциях доступа к **TLB**.

При возникновении исключений **TLB** (TLB Refill, TLB Invalid, или TLB Modified) биты $VA_{31:13}$ виртуального адреса записываются в поле VPN2 регистра **EntryHi**. В поле ASID, которое используется в процессе сравнения при поиске по **TLB**, программно записывается идентификатор текущего адресного пространства.

Поле VPN2 регистра **EntryHi** не определено после прерывания по ошибке адресации.

В таблице 5.12 рассматривается формат регистра **EntryHi**:

Таблица 5.12. Формат регистра EntryHi.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
VPN2	31:13	Разряды $VA_{31:0}$ виртуального адреса (виртуальный номер страницы, деленный на 2). Это поле записывается аппаратно при исключении TLB или при чтении TLB, и программно перед записью в TLB.	R/W	Не определено
0	12:8	При чтении возвращается нуль	0	0
ASID	7:0	Идентификатор адресного пространства. Это поле записывается аппаратно при чтении TLB, и программно при установке текущего значения ASID для записи в TLB и для сравнения при поиске по TLB с соответствующими полями ASID в строках TLB.	R/W	Не определено

6.12. Регистр Compare

Регистр **Compare** действует совместно с регистром Count с целью реализации функции таймера и прерывания по таймеру. Прерывание по таймеру является выходным сигналом процессора.

Когда значение регистра Count равняется значению регистра **Compare**, выход процессора *SI_TimerInt* устанавливается в 1. Этот выход остается равным 1, пока в регистр **Compare** не будет произведена запись. *SI_TimerInt* может быть заведен обратно в процессор на один из входов аппаратных прерываний. Обычно это делается мультиплексированием его с аппаратным прерыванием 5 для установки бита прерывания **IP**[7] в регистре Cause.

Для диагностических целей регистр **Compare** доступен для чтения и записи. Однако при нормальном функционировании регистр **Compare** используется только для записи. При записи значения в регистр в качестве побочного эффекта происходит очистка прерывания по таймеру.

Таблица 5.13 отображает формат регистра **Compare**:

Таблица 5.13. Формат регистра Compare.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
Compare	31:0	Период счета таймера	R/W	Не определено

6.13. Регистр Status

Регистр **Status** (**SR**) является регистром, доступным для чтения и записи. Он содержит поля рабочего режима, разрешения прерываний и диагностические состояния процессора. Для задания режимов функционирования процессора поля этого регистра объединяются следующим образом:

Разрешение прерываний: Прерывания разрешаются, когда истинны все следующие условия:

- IE=1
- EXL=0
- ERL=0

Если эти условия выполнены, прерывания разрешаются установкой битов IM.

Рабочие режимы: Процессор всегда находится в одном из двух режимов – **Kernel** или **User**. Режим задается установкой следующих битов регистра **Status**.

- Режим **User**: UM=1, EXL=0 и ERL=0
- Режим **Kernel**: UM=0, или EXL=1, или ERL=1

В таблице 5.14 приводится формат регистра **Status**:

Таблица 5.14. Формат регистра Status.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
CU3-CU0	31:28	Не используются	R/W	Не определено
RP	27	Разрешает режим пониженного потребления. Состояние RP бита выдается наружу как сигнал SI_RP	R/W	0
-	26:23	При чтении возвращается нуль	0	0
BEV	22	Управление размещением векторов исключения: 0: Нормальный 1: Начальная загрузка	R/W	1
TS	21	TLB-закрытие системы. Этот бит устанавливается, если при выполнении команд TLBWI или TLBWR образуется команда, которая приводит к условию закрытия, если оно разрешено. Программа может записывать в этот разряд только 0, чтобы очистить его, и не может вызвать переход этого бита из 0 в 1.	R/W	0
NMI	19	Указывает, что вход в вектор исключения начальной установки был осуществлен по причине возникновения NMI. 0: Не NMI (Аппаратный сброс) 1: NMI Программное обеспечение может записывать в этот бит только 0, чтобы очистить его, и не может записать 1.	R/W	1 для NMI, иначе 0
-	18:16	При чтении возвращается нуль	0	0
IM[7:0]	15:8	Маска прерываний: управление разрешением внешних, внутренних и программных прерываний. Прерывание принимается в случае, если установлен бит IE регистра Status и установлены соответствующие биты как в поле IM[7:0] регистра Status, так и в поле IP[7:0] регистра Cause. 0: Запрос на прерывание не разрешен. 1: Запрос на прерывание разрешен.	R/W	Не определено
-	7:5	При чтении возвращается нуль	0	0
UM	4	Указывает на то, что процессор работает в непривилегированном режиме (User): 0: Процессор работает в привилегированном режиме (Kernel) 1: Процессор работает в непривилегированном режиме (User) Замечание: процессор может также находиться в режиме Kernel, если установлены биты EXL или ERL. Это условие не влияет на состояние бита UM.	R/W	Не определено
-	3	При чтении возвращается нуль	0	0

ERL	2	Уровень ошибки. Устанавливается процессором при возникновении исключений Reset и NMI. 0: Нормальный уровень 1: Уровень ошибки Когда бит ERL установлен: Процессор находится в режиме Kernel. Прерывания запрещены. Команда ERET использует адрес возврата, содержащийся в ErrorEPC вместо EPC. kuseg используется как неотображаемая и некэшируемая область. Это позволяет иметь доступ к главной памяти при ошибках кэш. Поведение процессора не определено, если бит ERL установлен при выполнении кода из useg/kuseg.	R/W	1
EXL	1	Уровень Исключения. Устанавливается процессором при возникновении любого исключения, кроме Reset и NMI. 0: Нормальный уровень 1: Уровень исключения Когда бит EXL установлен: Процессор переходит в привилегированный режим (Kernel). Прерывания запрещены. Исключения TLB Refill используют общий вектор исключения вместо вектора TLB Refill. Если происходит другое исключение, EPC не модифицируется.	R/W	Не определено
IE	0	Разрешение Прерывания. 0: Отключает прерывания 1: Разрешает прерываниям	R/W	Не определено

6.14. Регистр Cause

Регистр **Cause**, в основном, описывает причину последнего исключения. Кроме того, поля регистра управляют запросами на программные прерывания и определяют вектор, которым обрабатываются прерывания. Все поля регистра **Cause**, за исключением IP[1:0], IV и WP, доступны только для чтения.

В таблице 5.15 приводится формат регистра **Cause**:

Таблица 5.15. Формат регистра Cause.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
BD	31	Указывает на то, что последнее исключение произошло в слоте задержки перехода: 0: Не в слоте задержки 1: В слоте задержки Замечание: бит BD не модифицируется на новом исключении, если установлен бит EXL.	R	Не определено
0	30:24	При чтении возвращается нуль	0	0
IV	23	Указывает, какой вектор используется для обслуживания исключений прерывания – общий или специальный вектор прерываний: 0: Используется общий вектор исключения (0x180) 1: Используется специальный вектор прерываний (0x200)	R/W	Не определено
0	22:16	При чтении возвращается нуль	0	0
IP[7:2]	15:10	Указывает, какие внешние прерывания установлены: 15: Аппаратное прерывание 5 или прерывание по таймеру 14: Аппаратное прерывание 4 13: Аппаратное прерывание 3 12: Аппаратное прерывание 2 11: Аппаратное прерывание 1 10: Аппаратное прерывание 0	R	Не определено
IP[1:0]	9:8	Управляет запросами программных прерываний (посредством записи «1» в данные разряды): 9: Запрос программного прерывания 1; 8: Запрос программного прерывания 0.	R/W	Не определено
0	7	Прерывание от встроенных средств отладки программ (OnCD).	R/W	0
Exc Code	6:2	Код исключения — см. Таблицу 2		
0	1:0	При чтении возвращается нуль	0	0

Таблица 5.16. Поле Exc Code.

Значение Exc Code	Мнемоника	Описание
0	Int	Прерывание
1	Mod	TLB-исключение модификации
2	TLBL	TLB-исключение (загрузка или вызов команды)
3	TLBS	TLB-исключение (сохранение)
4	AdEL	Прерывание по ошибке адресации (загрузка или вызов команды)
5	AdES	Прерывание по ошибке адресации (сохранение)
6-7		Не используются
8	Sys	Системное исключение
9	Bp	Исключение Breakpoint
10	RI	Исключение зарезервированной команды
11	CpU	Исключение недоступности сопроцессора
12	Ov	Исключение целочисленного переполнения
13	Tr	Исключение Trap
14-22		Зарезервированы
23		Не используется
24	MCheck	Аппаратный контроль
25-31		Зарезервированы

6.15. Регистр EPC

Программный счетчик исключения (**EPC**) является регистром, доступным для чтения и записи. **EPC** содержит адрес, начиная с которого возобновляется исполнение программы после завершения обработки исключения. Все биты регистра **EPC** значимы и должны перезаписываться.

Для синхронных (точных) исключений **EPC** содержит одно значение из следующих:

- Виртуальный адрес команды, которая была прямой причиной исключения;
- Виртуальный адрес команды перехода (*Branch* или *Jump*), непосредственно предшествующей исключению, если команда, вызвавшая исключение, находится в слоте задержки перехода и установлен бит BD в регистре Cause.

Если установлен бит EXL в регистре Status, процессор не записывает адрес в регистр **EPC** при возникновении новых исключений. Однако, новое значение можно записать в **EPC** командой `mtc0`.

В таблице 5.17 приведен формат регистра **EPC**:

Таблица 5.17. Формат регистра EPC.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
EPC	31:0	Программный счетчик исключения	R/W	Не определено

6.16. Регистр PRId

Регистр идентификации процессора (**PRId**) – это 32-х разрядный регистр, доступный только для чтения. Он содержит информацию, идентифицирующую изготовителя, опции изготовителя, идентификацию процессора, и его версию.

В таблице 5.18 приводится формат регистра **PRId**:

Таблица 5.18. Формат регистра PRId.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
R		При чтении возвращается нуль	R	0
Company ID	23:16	Идентификация компании, которая проектировала или изготовляла процессор.	R	1010
Processor ID	15:8	Идентификация типа процессора.	R	10010
Revision	7:0	Номер версии процессора. Позволяет программам различать разные версии одного типа процессора.	R	0

6.17. Регистр Config/Config1

Регистр **Config** определяет различную конфигурационную информацию, а также информацию о возможностях процессора. Большинство полей регистра **Config** инициализируется аппаратно при выполнении исключения Reset или имеет постоянное значение, и только поле K0 должно быть проинициализировано программно обработчиком исключения Reset.

Таблица 5.19 иллюстрирует формат регистра **Config**:

Таблица 5.19. Формат регистра Config.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	**Биты**			
M	31	Этот бит аппаратно устанавливается в высокий уровень, указывая на наличие регистра Config1	R	1
K23	30:28	Это поле управляет кэшируемостью адресных сегментов kseg2 и kseg3 в режиме FM. В режиме TLB не используется. См. табл.2.33.	FM:R/W	FM:010
			TLB:R	TLB:000
KU	27:25	Это поле управляет кэшируемостью адресных сегментов kuseg и useg в режиме FM. В режиме TLB не используется. См. табл.2.33.	FM:R/W	FM:010
			TLB:R	TLB:000
0	24:21	Не используются	0	0
MDU	20	Тип MDU: итеративный умножитель и делитель	R	1
R	19	При чтении возвращается нуль	0	0
MM	18:17	Режим No Merging для 32 bit collapsing write buffer	R	0
BM	16	Тип передачи Burst: последовательный	R	0
BE	15	Режим endian: Little endian	R	0
AT	14:13	Тип архитектуры, реализованной процессором: MIPS32.	R	0
AR	12:10	Номер версии: 1	R	0
MT	9:7	Тип MMU: 1: Стандартный TLB (FM = 0) 3: Фиксированное отображение (FM = 1) 0, 2, 4-7: зарезервированы	R	TLB: 01
				FM: 11
R	6:3	При чтении возвращается нуль	0	0
K0	2:0	Алгоритм когерентности для kseg0	R/W	010

В таблице 5.20 описываются атрибуты когерентности кэш:

Таблица 5.20. Атрибуты когерентности кэш.

Значение C[5:3]	
0, 1, 3*, 4, 5, 6	Кэшируемая, некогерентная область
2*, 7	Некэшируемая область
* - Архитектура MIPS32 предусматривает только эти два значения. Остальные значения не используются и отображаются в используемые значения. Например, 0, 1, 4, 5 и 6 отображается в 3, а 7 – в 2.	

Регистр Config1

Регистр **Config1** является дополнением к регистру **Config** и кодирует дополнительную информацию о возможностях процессора. Все поля регистра **Config1** доступны только для чтения. Формат регистра **Config1** приведен в таблице 5.21:

Таблица 5.21. Формат регистра Config1.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
R	31	При чтении возвращается нуль	0	0
Размер MMU	30:25	Это поле содержит количество строк TLB минус 1. В режиме TLB возвращается код 15 в десятичном формате, в режиме Fixed Mapping – 0.	R	001111 (FM =0) 000000 (FM =1)
IS	24:22	Количество наборов кэш команд: резервная опция	R	111
IL	21:19	Размер строки кэш команд: 16 байт	R	011
IA	18:16	Тип кэш команд: Direct mapped	R	0
DS	15:13	Нет кэш данных	R	0
DL	12:10	Нет кэш данных	R	0
DA	9:7	Нет кэш данных	R	0
R	6:5	При чтении возвращается нуль	0	0
PC	4	Нет регистра Performance Counter	R	0
WR	3	Нет регистра WATCH	R	0
CA	2	Не реализовано	R	0
EP	1	EJTAG не реализован	R	0
FP	0	Нет плавающей арифметики	R	0

6.18. Регистр LLAddr

Регистр **LLAddr** содержит физический адрес последней команды *Load Linked* (`11`). Этот регистр используется только для диагностических целей.

Формат регистра иллюстрирован таблицей 5.22:

Таблица 5.22. Формат регистра LLAddr.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
0	**31:28**	При чтении возвращается нуль	0	0
Paddr[31:4]	27:0	Физический адрес последней команды LL	R	Не определено

6.19. Регистр ErrorEPC

Доступный для чтения и записи, регистр **ErrorEPC** полностью подобен регистру EPC, но используется при возникновении исключений ошибок. Все биты регистра **ErrorEPC** значимы и должны перезаписываться. Регистр также используется для сохранения значения счетчика команд при возникновении исключений Reset и немаскируемого прерывания (**NMI**).

Регистр **ErrorEPC** содержит виртуальный адрес, начиная с которого может возобновиться исполнение программы после обработки ошибочной ситуации.

Этот адрес может быть:

- Виртуальным адресом команды, вызвавшей исключение;
- Виртуальным адресом команды перехода (*Branch* или *Jump*), непосредственно предшествующей исключению, если команда, вызвавшая ошибку, находится в слоте задержки перехода.

В отличие от регистра EPC, для регистра **ErrorEPC** не имеется соответствующего признака слота задержки перехода.

Формат регистра **ErrorEPC** приведен в таблице 5.23:

Таблица 5.23. Формат регистра ErrorEPC.

Поля		Описание	Чтение/ Запись	Начальное состояние
Имя	Биты			
ErrorEPC	31:0	Счетчик команд при исключении ошибки	R/W	Не определен

7. Обработка исключений

7.1. Введение

Исключения и **прерывания** представляют собой принудительную передачу управления задаче или процедуре. Такая задача или процедура называется обработчиком. Прерывания происходят в произвольные моменты времени выполнения программы в ответ на сигналы аппаратного обеспечения. Исключения происходят вследствие выполнения команд, приводящих к этим исключениям.

В терминах ИМС серии "*МУЛЬТИКОР*" исключения являются более общим понятием, то есть при возникновении прерывания следует обрабатывать исключение по прерыванию.

Ячейки памяти, содержащие обработчики исключений, называются векторами исключений.

Ядро RISC способно принимать исключения от ряда источников, в том числе промах буфера преобразования адресов (**TLB**), арифметические переполнение, прерывание ввода-вывода, и системные вызовы. Обнаружив одно из этих исключений, ЦПУ приостанавливает нормальную последовательность исполнения команд и процессор входит в режим *Kernel*.

При этом ядро отключает прерывания и вынуждает процессор запустить программу обработчика исключений, расположенную в фиксированных адресах памяти. Обработчик сохраняет контекст процессора – содержимое счетчика команд, текущий режим процессора и статус разрешения прерываний. Таким образом, контекст может быть восстановлен по завершению обработки исключения.

При возникновении исключения в регистр **Exception Program Counter** (**EPC**) загружается адрес, начиная с которого исполнение команд может возобновиться после завершения обработки исключения. В регистр **EPC** помещается адрес команды, вызвавшей исключение или, если команда находилась в слоте задержки перехода, адрес команды перехода, предшествующей слоту задержки. Чтобы различить эти ситуации, программное обеспечение должно проанализировать бит **BD** (*branch delay*) в регистре **Cause** сопроцессора **CP0**.

В данной главе рассматриваются все возникающие в ИМС "*МУЛЬТИКОР*" исключения, а также способы их обработки. Глава включает в себя:

- Описание условий возникновения исключений;
- Приоритеты исключений;
- Расположение векторов исключений;
- Обработку общих исключений;
- Описание видов исключений;
- Программное обслуживание исключений;
- Пример обработки исключений.

7.2. Условия возникновения исключений

Исключения обрабатываются на стадии *M* конвейера.

При обнаружении исключительной ситуации, команда, находящаяся на стадии *M*, и все команды, следующие за ней на конвейере, отменяются. Соответственно, все условия остановки конвейера, относящиеся к этой команде, а также условия последующих исключений, которые также могут относиться к ней, игнорируются.

Когда условие исключения обнаруживается на стадии *M*, процессор заполняет необходимые регистры **CP0** значениями, относящимися к состоянию исключения, изменяет счетчик команд (**PC**) на адрес соответствующего *вектора обработки исключения* и очищает признаки исключения, относящиеся к более ранним стадиям конвейера.

Такая реализация позволяет завершить исполнение команды, находящейся на стадии *W*, и запретить завершение последующих команд. Таким образом, значения, сохраненного в регистре **EPC** (в случае ошибок – в **Error PC**), достаточно для возобновления исполнения. Это также обеспечивает поступление исключений в соответствии с порядком исполнения команд – команда, вызывающая исключение, может быть уничтожена командой с более поздней стадии конвейера, также вызвавшей исключение.

7.3. Приоритеты исключений

В таблице 6.1 перечислены все возможные в ИМС *МС-12* исключения с их относительными приоритетами (от высшего к низшему). Некоторые из этих исключений могут случаться одновременно - в таком случае вызывается исключение с наивысшим приоритетом.

Таблица 6.1. Приоритеты исключений.

Исключение	Описание
Reset	Аппаратный сброс
NMI	Внешнее немаскируемое прерывание
TLB_Ri, TLB_Ii	Промах TLB при выборке команды, Попадание в запрещенную страницу TLB (V=0) при выборке команды
AdELi	Ошибка выравнивания адреса при выборке команды; Ссылка на адрес режима Kernel при работе в режиме User при выборке команды
MCheck Sys Bp CpU RI Ov Tr AdELd AdES	Запись в TLB, создающая конфликт с существующей строкой TLB Выполнение команды SYSCALL Выполнение команды BREAK Выполнение команды сопроцессора в режиме User Выполнение зарезервированной команды Переполнение в арифметической команде Выполнение trap (когда условие trap истинно) Ошибка выравнивания адреса при загрузке данных; Ссылка на адрес режима Kernel при работе в режиме User при загрузке данных Ошибка выравнивания адреса при сохранении данных; Попытка сохранения по адресу Kernel в режиме User
TLB_Rd, TLB_Id	Промах TLB при загрузке данных; Попадание в запрещенную страницу TLB (V=0) при загрузке данных
TLB_M	Сохранение в TLB-странице с D=0
Interrupt	Установка немаскируемых HW или SW - прерываний

7.4. Расположение векторов исключений

Векторы исключений аппаратного сброса (**Reset**) и внешнего немаскируемого прерывания (**NMI**) всегда находятся по адресу `0xBFC00000`. Адреса всех остальных исключений являются комбинациями векторных смещений и базового адреса. Базовый адрес для них определяется состоянием бита BEV регистра **Status** (**CP0**). При `BEV=0`, базовый адрес равен `0x80000000`. При `BEV=1` базовый адрес равен `0xBFC00200`. В таблице 6.2 приведены смещения от базового адреса как функции исключения:

Таблица 6.2. Смещения векторов исключений.

Исключение	Смещение вектора
TLB Refill, EXL = 0	0x000
Reset, NMI	0x000
Исключения общего характера (General Exeptions)	0x180
Interrupt, $Cause_{IV}$ = 1	0x200

В таблице 6.3 приводятся все возможные векторы исключений при различных значениях BEV:

Таблица 6.3. Векторы исключений.

Исключение	BEV	EXL	IV	Вектор
Reset, NMI	-	-	-	0xBFC0_0000
TLB Refill	0	0	-	0x8000_0000
TLB Refill	0	1	-	0x8000_0180
TLB Refill	1	0	-	0xBFC0_0200
TLB Refill	1	1	-	0xBFC0_0380
Interrupt	0	0	0	0x8000_0180
Interrupt	0	0	1	0x8000_0200
Interrupt	1	0	0	0xBFC0_0380
Interrupt	1	0	1	0xBFC0_0400
Остальные	0	-	-	0x8000_0180
Остальные	1	-	-	0xBFC0_0380

Подробное описание регистров **CP0** приводится в разделе "Системный управляющий сопроцессор".

7.5. Обработка общих исключений

Кроме исключений аппаратного сброса и немаскируемого прерывания, которые обслуживаются особым образом, обработка всех остальных исключений происходит в соответствии со следующим основным маршрутом:

- Если бит EXL регистра состояния (**Status**) очищен, в регистр EPC загружается значение **PC**, по которому выполнение программы будет перезапущено. Также, при необходимости, устанавливается бит BD (*Branch Delay*) в регистре причины (**Cause**).

Если в слоте задержки перехода не находится команды, бит BD будет очищен, а в EPC загружается текущее значение счетчика команд. Если же в слоте задержки перехода есть команда, бит BD устанавливается в единицу, а в EPC загружается значение `PC-4`. Если в регистре **Status** установлен бит EXL, в регистр EPC ничего не загружается, а бит BD в регистре **Cause** не модифицируется.

- В поля CE и ExcCode регистра **Cause** загружаются значения, соответствующие исключению.
- В регистре **Status** устанавливается бит EXL.
- Процессор стартует с вектора исключения.

Значение, загруженное в EPC, представляет собой адрес возврата из исключения и в обычной ситуации программе обработки исключения не требуется его модифицировать. Программе также не нужно просматривать бит BD в регистре **Cause**, если не возникает необходимости определить действительный адрес команды, вызвавшей исключение.

Схематично процесс обработки общих исключений можно записать следующим образом:

```
if StatusEXL == 0 then
   if InstructionInBranchDelaySlot then
      EPC <= PC - 4
      CauseBD <= 1
   else
      EPC <= PC
      CauseBD <= 0
   endif
   if (ExceptionType == TLBRefill) then
      vectorOffset <= 0x000
      elseif (ExceptionType == Interrupt) and (CauseIV == 1) then
             vectorOffset <= 0x200
      else
             vectorOffset <= 0x180
   endif
else
   vectorOffset <= 0x180
endif
CauseCE <= FaultingCoprocessorNumber
CauseExcCode <= ExceptionType
StatusEXL <= 1
if (StatusBEV == 1) then
   PC <= 0xBFC0_0200 + vectorOffset
else
   PC <= 0x8000_0000 + vectorOffset
endif
```

7.6. Виды исключений

7.6.1. Виды исключений

В данном разделе рассматриваются описания всех возможных в ИМС "*МУЛЬТИКОР*" исключений:

- Исключение по аппаратному сбросу;
- Исключение по немаскируемому прерыванию;
- Исключение по обновлению TLB;
- Исключение по некорректному использованию TLB;
- Исключение по ошибке адресации;
- Исключение по аппаратному контролю;
- Исключение по системному вызову;
- Исключение по команде BREAK;
- Исключение по зарезервированной команде;
- Исключение по недоступности сопроцессора;
- Исключение по целочисленному переполнению;
- Исключение по ловушке;
- Исключение по сохранению в запрещенной области;
- Исключение по прерыванию.

7.6.2. Исключение по аппаратному сбросу

Данное немаскируемое исключение происходит при установке сигнала аппаратного сброса (**Reset**). При возникновении исключения аппаратного сброса процессор полную начальную инициализацию - приводит автоматы к исходному состоянию. Процессор переводится в состояние, из которого он может начать запуск команд, находящихся в некэшируемой и неотображаемой области. После возникновения исключения аппаратного сброса состояние процессора не определено кроме следующего:

- Регистр **Random** устанавливается в значение, равное количеству строк `TLB - 1`.
- Регистр **Wired** устанавливается в 0.
- Регистр **Config** устанавливается в свое начальное состояние (*boot state*).
- Поля RP, BEV, TS, NMI и ERL регистра **Status** устанавливаются в заданные значения.
- В **PC** загружается значение `0xBFC00000`.

Вектор исключения:
Reset (`0xBFC00000`)

Операция:

```
Random <= TLBEntries - 1
Wired <= 0
Config <= ConfigurationState
Status_RP <= 0
```

```
      Status_BEV <= 1
      Status_TS <= 0
      Status_NMI <= 0
      Status_ERL <= 1
   PC <= 0xBFC00000
```

7.6.3. Исключение по немаскируемому прерыванию

Немаскируемое прерывание возникает по положительному фронту входного сигнала **NMI** или при срабатывании сторожевого таймера **WDT**. Исключение **NMI** происходит только в пределах границ команды, поэтому оно не вызывает сброса или другой переинициализации аппаратных средств. Состояние кэш, памяти, а также другие состояния процессора остаются неизменными. Значения регистров также сохраняются за исключением следующего:

- Поля BEV, TS, NMI и ERL регистра **Status** принимают заданные значения.
- В регистр **ErrorEPC** загружается значение `PC-4`, если если прерывание произошло на фоне команды в слоте задержки перехода. В противном случае в регистр **ErrorEPC** загружается значение **PC**.
- В **PC** загружается значение `0xBFC00000`.

Вектор исключения:
Reset (`0xBFC00000`)

Операция:

```
StatusBEV <= 1
StatusTS <= 0
StatusNMI <= 1

StatusERL <= 1
if InstructionInBranchDelaySlot then
   ErrorEPC <= PC - 4
else
   ErrorEPC <= PC
endif
PC <= 0xBFC0_0000
```

7.6.4. Исключение по обновлению TLB

Исключение **TLB Refill** происходит во время выборки команды или доступа к данным, если в **TLB** нет ни одной строки, соответствующей ссылке к отображенному адресному пространству, и бит EXL в регистре **Status** равен 0.

Значение поля ExcCode регистра Cause:
TLBL: Произошла ссылка по загрузке данных или выборке команды
TLBS: Произошла ссылка по сохранению данных

Таблица 6.4. Дополнительно сохраняемые состояния по исключению TLB Refill.

Состояние регистра	Значение
BadVAddr	Ошибочный адрес
Context	Поле BadVPN2 содержит $VA_{31:13}$ ошибочного адреса
EntryHi	Поле VPN2 содержит $VA_{31:13}$ ошибочного адреса; поле ASID содержит ASID отсутствующей ссылки

Вектор исключения:
Вектор **TLB Refill** (смещение `0x000`)

7.6.5. Исключение по некорректному использованию TLB

Исключение **TLB Invalid** происходит во время выборки команды или доступа к данным в одном из следующих случаев:

- В **TLB** нет ни одной строки, соответствующей ссылке к отображенному адресному пространству, и бит EXL в регистре **Status** равен 1.
- Строка **TLB** соответствует ссылке к отображенному адресу, но ее бит валидности выключен.

Значение поля ExcCode регистра Cause:
TLBL: Произошла ссылка по загрузке данных или выборке команды
TLBS: Произошла ссылка по сохранению данных

Таблица 6.5. Дополнительно сохраняемые состояния по исключению TLB Invalid.

Состояние регистра	Значение
BadVAddr	Ошибочный адрес
Context	Поле BadVPN2 содержит $VA_{31:13}$ ошибочного адреса
EntryHi	Поле VPN2 содержит $VA_{31:13}$ ошибочного адреса; поле ASID содержит ASID отсутствующей ссылки

Вектор исключения:
Общий вектор исключения (смещение `0x180`)

7.6.6. Исключение по ошибке адресации

Исключение по ошибке адресации во время доступа к команде или данным возникает при попытке выполнить одно из следующих действий:

- Выбрать команду, загрузить или сохранить слово данных, если они не выровнены в границах слова
- Загрузить или сохранить пол-слова, если оно не выровнено в границах пол-слова
- Обратиться по адресу пространства **Kernel** при работе в режиме **User**

Значение поля ExcCode регистра Cause:
ADEL: Произошла ссылка по загрузке данных или выборке команды
ADES: Произошла ссылка по сохранению данных

Таблица 6.6. Дополнительно сохраняемые состояния по исключению ошибки адресации.

Состояние регистра	Значение
BadVAddr	Ошибочный адрес

Вектор исключения:
Общий вектор исключения (смещение `0x180`)

7.6.7. Исключение по аппаратному контролю

Данное исключение возникает, если при выполнении команды записи в **TLB** (**`TLBWI`** или **`TLBWR`**) обнаруживается, что поле виртуального адреса записываемой строки соответствует такому же полю одной из строк, уже хранящихся в **TLB**.

При возникновении данной ситуации запись в **TLB** не выполняется и устанавливается бит TS в регистре **Status**. Этот бит является статусным и не влияет на функционирование процессорного ядра. Он сбрасывается программно после разрешения данной ситуации, осуществляемого очисткой конфликтных строк в **TLB**.

Значение поля ExcCode регистра Cause:
Mcheck

Дополнительно сохраняемые состояния:
Нет

Вектор исключения:
Общий вектор исключения (смещение `0x180`)

7.6.8. Исключение по системному вызову

Исключение **System Call** является одним из шести исключений исполнения. Эти исключения имеют одинаковый приоритет. Исключение **System Call** возникает при исполнении команды **`SYSCALL`**.

Значение поля ExcCode регистра Cause:
Sys

Дополнительно сохраняемые состояния:
Нет

Вектор исключения:
Общий вектор исключения (смещение `0x180`)

7.6.9. Исключение по команде BREAK

Исключение **Breakpoint** является одним из шести исключений исполнения. Эти исключения имеют одинаковый приоритет. Исключение **Breakpoint** возникает при исполнении команды **BREAK**.

Значение поля ExcCode регистра Cause:
Bp

Дополнительно сохраняемые состояния:
Нет

Вектор исключения:
Общий вектор исключения (смещение `0x180`)

7.6.10. Исключение по зарезервированной команде

Исключение по зарезервированной команде является одним из шести исключений исполнения. Эти исключения имеют одинаковый приоритет. Исключение зарезервированной команды вызывается при исполнении команды с неопределенным старшим кодом операции (*major opcode*) или полем функции.

Значение поля ExcCode регистра Cause:
RI

Дополнительно сохраняемые состояния:
Нет

Вектор исключения:
Общий вектор исключения (смещение `0x180`)

7.6.11. Исключение по недоступности сопроцессора

Исключение недоступности сопроцессора является одним из шести исключений исполнения. Эти исключения имеют одинаковый приоритет. Исключение недоступности сопроцессора вызывается при попытке исполнения команды сопроцессора **CP0** в режиме **User**, если сопроцессор не был заказан для использования.

Значение поля ExcCode регистра Cause:
CPU

Дополнительно сохраняемые состояния:
Нет

Вектор исключения:
Общий вектор исключения (смещение `0x180`)

7.6.12. Исключение по целочисленному переполнению

Исключение целочисленного переполнения является одним из шести исключений исполнения. Эти исключения имеют одинаковый приоритет. Исключение целочисленного переполнения вызывается, когда выбранные целочисленные команды приводят к переполнению в двоичном коде.

Значение поля ExcCode регистра Cause:
Ov

Дополнительно сохраняемые состояния:
Нет

Вектор исключения:
Общий вектор исключения (смещение `0x180`)

7.6.13. Исключение по ловушке

Исключение **Trap** является одним из шести исключений исполнения. Все такие исключения имеют одинаковый приоритет. Исключение **Trap** вызывается, если условие команды *trap* истинно (*TRUE*).

Значение поля ExcCode регистра Cause:
Tr

Дополнительно сохраняемые состояния:
Нет

Вектор исключения:
Общий вектор исключения (смещение `0x180`)

7.6.14. Исключение по сохранению в запрещенной области

Это исключение возникает при обращению по записи данных к отображенному адресу, если найденная строка **TLB** действительна, но страница запрещена для записи.

Значение поля ExcCode регистра Cause:
Mod

Таблица 6.7. Дополнительно сохраняемые состояния по исключению сохранения в запрещенной области.

Состояние регистра	Значение
BadVAddr	Ошибочный адрес
Context	Поле BadVPN2 содержит $VA_{31:13}$ ошибочного адреса
EntryHi	Поле VPN2 содержит $VA_{31:13}$ ошибочного адреса; поле ASID содержит ASID отсутствующей ссылки

Вектор исключения:
Общий вектор исключения (смещение `0x180`)

7.6.15. Исключение по прерыванию

Исключение прерывания возникает, когда сигнал одного или более разрешенных регистром **Status** прерываний устанавливается на входе процессора.

Значение поля ExcCode регистра Cause:
Mod

Таблица 6.8. Дополнительно сохраняемые состояния по исключению прерывания.

Состояние регистра	Значение
$Cause_{IP}$	Указывает код прерывания

Вектор исключения:
Общий вектор исключения (смещение `0x180`), если бит IV регистра **Cause** равен 0;
Вектор прерывания (смещение `0x200`), если бит IV регистра **Cause** равен 1.

7.7. Программное обслуживание исключений

Программное обслуживание исключений осуществляется посредством программ-обработчиков исключений. При возникновении исключения, программа переходит по адресу, содержащему вектор данного исключения.

Таким образом, если разместить по этому адресу обработчик исключения или команду перехода к нему, программа RISC-ядра будет выполнять код обработчика данного исключения.

Разместить обработчик исключения в памяти можно двумя способами:

1. Разместить обработчик (или переход к нему) по нужному адресу при сборке проекта. Для этого необходимо записать программу обработчика в отдельной секции и скомпоновать эту секцию по требуемому адресу. Такой способ удобен при написании программы RISC на языке Ассемблера. Кроме того, обработчик исключения по аппаратному сбросу (**Reset**) необходимо разместить по адресу `0xBFC00000` именно при сборке проекта.
2. Разместить переход к функции обработчика по нужному адресу во время исполнения программы RISC (обычно, во время исполнения обработчика исключения **Reset**). Такой подход удобен при использовании обработчиков, написанных на языке *C*, когда заранее неочевидны размеры той или иной секции программы.

При размещении программ-обработчиков следует помнить, что расположение векторов исключений зависит от состояния регистров **CP0**. Все возможные адреса векторов приведены на странице "Расположение векторов исключений".

В общем виде программа-обработчик исключения исполняет следующие функции:

- Получение информации о причине возникновения исключения или прерывания;
- Разрешение исключительной ситуации в соответствии с полученной информацией;
- Возврат из исключения (команда **eret**).

7.8. Пример обработки исключений

Рассмотрим способ написания обработчика исключений на примере обработки запроса на прерывание **QSTR**[31] - то есть прерывания от DSP. Приведенная ниже программа будет запускать ядро DSP до останова. При останове ядра DSP (команда **STOP**) будет возникать прерывание и вызываться обработчик исключения по этому прерыванию.

Прежде всего необходимо написать обработчик исключения по аппатному сбросу (Reset), так как это исключение возникает сразу при запуске программы. Таким обработчиком в данном примере будет программа файла *start.s*:

```
  .include "memory_12_asm.h"
  .set noreorder
  .text

Start_Programm:
               lui   $30,CPU_BASE    #разрешение прерывания от DSP (QSTR[31]) в MASKR
               lw    $3,MASKR($30)
               li    $4,0x80000000
               or    $2,$4,$3
               sw    $2,MASKR($30)
               li    $4,0x8001

               mtc0  $4,$12          #разрешение прерываний

               li    $3,0            #передача параметра "PC"
               jal   Run_DSP
               nop
               j     Infinite_Cycle
               nop
```

Программа файла *start.s* является обработчиком исключения Reset, поэтому при сборке проекта секция текста данного файла располагается по адресу 0xBFC00000. Данная программа осуществляет следующие действия:

- разрешает прерывания от DSP-ядра в регистре маскирования прерываний (**MASKR**[31]=1);
- разрешает прерывания в регистре **Status** (**CP0**);
- помещает 0 в регистр **$3**;
- переходит (с возвратом) к метке Run_DSP;
- переходит к метке Infinite_Cycle.

По меткам Run_DSP и Infinite_Cycle расположены программы, описанные в файле *main.s*. Программа файла main.s является основной программой RISC-ядра и располагается при сборке по адресу 0xBFC01000. Рассмотрим содержимое файла *main.s*:

```
  .include "memory_12_asm.h"
  .text
       .global Infinite_Cycle
       .global Run_DSP

Infinite_Cycle:                    #бесконечный цикл. используется для ожидания
прерывания "Останов DSP"
               nop
               j    Infinite_Cycle
               nop

Run_DSP:                           #запуск DSP
               lui  $30,DSP_BASE
               sh   $3,PC($30)     #в $3 - адрес старта программы
               lhu  $3,DCSR($30)
               ori  $3,0x4000
               sh   $3,DCSR($30)
               jr   $31            #возврат
               nop
```

Данная программа содержит две подпрограммы - Infinite_Cycle и Run_DSP. Run_DSP - функция, осуществляющая запуск программы DSP, начиная с адреса, переданного в регистре **$3**. После запуска DSP функция переходит к адресу, сохраненному в регистре возврата **$31**. Infinite_Cycle - программа, выполняющая бесконечный цикл. Данная программа используется для ожидания прерываний от DSP.

Всякий раз, когда возникает прерывание от DSP-ядра (в данном случае - по команде **STOP**), управление передается обработчику исключения по прерыванию. Код программы обработчика приведен в файле *interrupts.s*:

```
  .include "memory_12_asm.h"
  .text

Process_DSP_Stop:
                lui  $30,DSP_BASE
                lw   $3,R2($30)
                addi $3,0x1
                sw   $3,R2($30)
                li   $3,0
                li   $30,0xbfc01010
                li   $31,0x800001b4
                jr   $30
                nop
                nop
                eret
```

Обработчик исключения по прерыванию в данном примере осуществляет следующие действия:

- увеличивает счетчик обработанных прерываний (расположен в регистре **R2** ядра DSP);

- помещает 0 в регистр **$3**;
- помещает адрес команды eret в регистр возврата **$31**;
- вызывает функцию `Run_DSP`;
- возвращает управление подпрограмме `Infinite_Cycle` (по команде **eret**).

Так как при задании нового значения регистра **Status** (файл *start.s*) биты BEV, EXL и IV были установлены в ноль, вектор данного исключения следует разместить по адресу `0x80000180`. Ввиду незначительных размеров обработчика, код программы обработчика исключения по прерыванию был также размещен по этому адресу.

Таким образом, в этой программе перезапуск ядра DSP осуществляется обработчиком исключения по прерыванию.

Текст программы DSP в данном случае несущественен, главное условие - программа должна содержать команду **STOP**.

Примечание: так как вектор исключения по прерыванию в данном примере расположен по адресу `0x800001b4`, для вызова функции `Run_DSP` и для возврата из нее используется команда **jr** (*Jump Register*). Команды **j** и **jal** использовать нельзя, так как расстояние между обработчиком и функцией `Run_DSP` превышает 256 Мб.

Примечание 2: сброс прерывания **QSTR**`[31]` осуществляется автоматически по запуску DSP-ядра функцией `Run_DSP`.

Примечание 3: к каждому из файлов программы RISC директивой **.include** подключен файл *memory_12_asm.h*. Этот файл описывает адресное пространство *MultiCore-12*.

Все возможные адреса векторов исключений рассматриваются на странице "Расположение векторов исключений".

8. Обработка прерываний

8.1. Введение

Исключения и **прерывания** представляют собой принудительную передачу управления задаче или процедуре. Такая задача или процедура называется обработчиком. Прерывания происходят в произвольные моменты времени выполнения программы в ответ на сигналы аппаратного обеспечения. Исключения происходят вследствие выполнения команд, приводящих к этим исключениям.

В терминах ИМС серии "*МУЛЬТИКОР*" исключения являются более общим понятием, то есть при возникновении прерывания следует обрабатывать исключение по прерыванию.

В данной главе рассматривается:

- Условия возникновения прерываний;
- Регистры QSTR и MASKR;
- Регистры CP0, управляющие обслуживанием прерываний;
- Прерывания от DSP-ядра.

8.2. Условия возникновения прерываний

В ИМС серии "*МУЛЬТИКОР*" прерывания происходят при аппаратном (или программном) запросе на обслуживание того или иного события.

Возникновение HW/SW-прерывания влечет за собой возникновение исключения по прерыванию (*Interrupt Exception*), которое и следует обрабатывать (см. главу "Обработка исключений"). Обработка исключения по прерыванию происходит только если выполняются следующие условия:

- Биты EXL и ERL регистра **Status** (**CP0**) установлены в 0;
- если прерывание внутреннее, то запрос должен быть разрешен в регистре **MASKR**;
- запрос на прерывание должен быть разрешен в регистре **Status** (**CP0**).

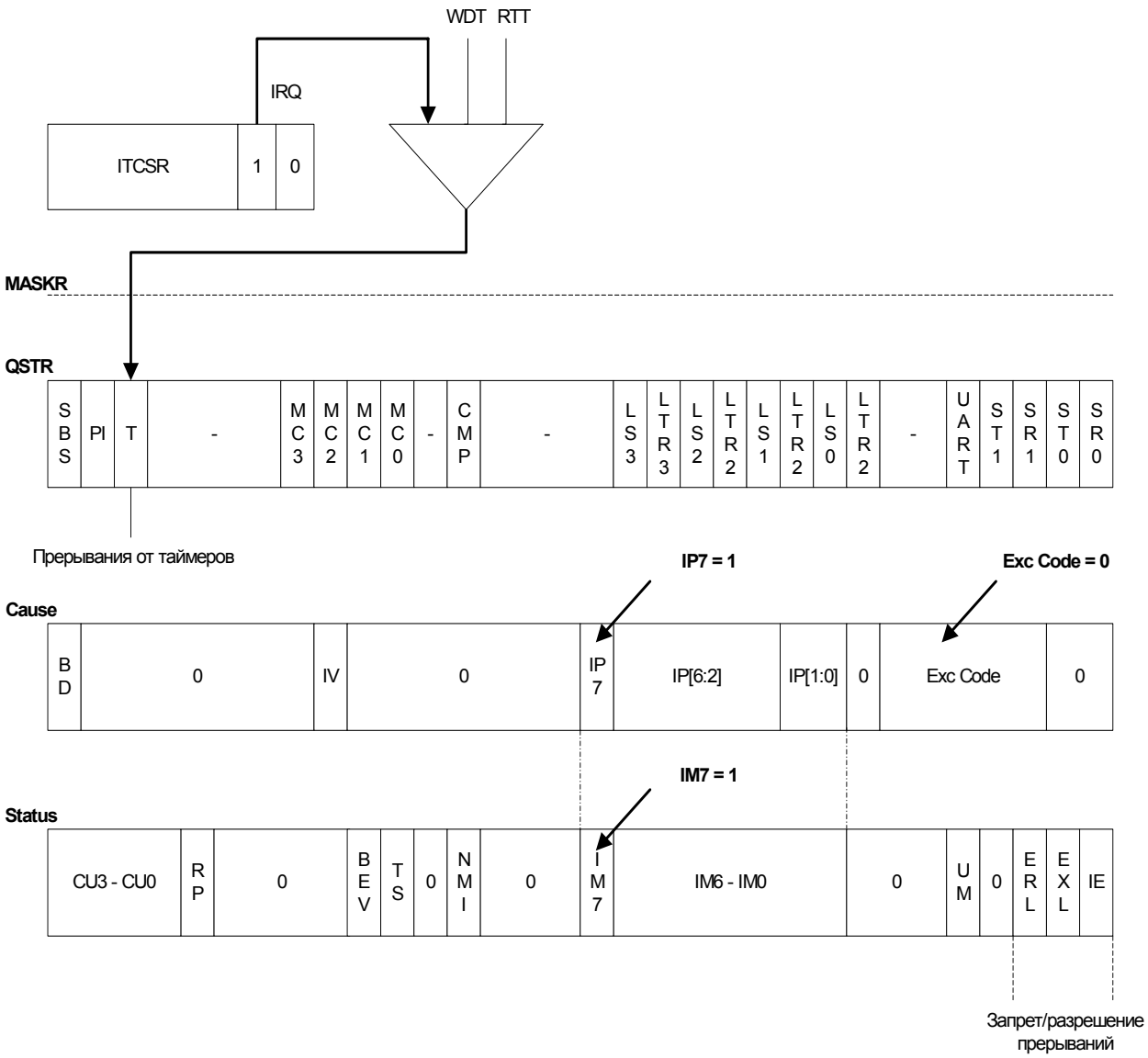
Регистры **CP0**, управляющие обслуживанием прерываний рассматриваются здесь.

Системные регистры, управляющие внутренними прерываниями рассматриваются на странице "Регистры QSTR и MASKR".

Ниже перечислены все возможные в ИМС "*МУЛЬТИКОР*" прерывания:

- Немаскируемое прерывание;
- Прерывания от таймеров;
- Прерывания от каналов DMA;
- Прерывания от портов ввода-вывода;
- Прерывания от ядра DSP;
- Внешние прерывания.

На рисунке демонстрируется состояние некоторых регистров процессора при возникновении прерывания от интервального таймера. Бит прерывания выставляется в 29-м бите регистра **QSTR** в том случае, если это прерывание разрешено регистром **MASKR**, а также разрешено 7-е аппаратное прерывание в регистре Status (`IM[7]`). Кроме того, поля ERL/EXL/IE должны быть установлены в состояние разрешения прерываний. При возникновении прерывания бит `IP[7]` регистра **Cause** будет равен единице, а поле Exc Code - нулю.



8.3. Регистры QSTR и MASKR

Регистры **QSTR** и **MASKR** являются системными регистрами и используются для управления обслуживанием внутренних прерываний.

В регистре **QSTR**, доступном только по чтению, устанавливаются запросы на то или иное внутреннее прерывание в соответствии с таблицей 7.1 (1 - есть запрос, 0 - нет запроса):

Таблица 7.1. Формат регистра QSTR.

Номер Разряда	Условное обозначение прерывания	Название прерывания
0	SRx0	Прерывание от порта SPORT0 при приеме данных или от канала DMA SportRxCh0
1	STx0	Прерывание от порта SPORT0 при выдаче данных или от канала DMA SportTxCh0
2	SRx1	Прерывание от порта SPORT1 при приеме данных или от канала DMA SportRxCh1
3	STx1	Прерывание от порта SPORT1 при выдаче данных или от канала DMA SportTxCh1
4	Uart	Прерывание от UART
6:5	-	Резерв
7	LTRx0	Прерывание от порта LPORT0 при обмене данными или от канала DMA LportCh0
8	LSrq0	Запрос обслуживания от порта LPORT0
9	LTRx1	Прерывание от порта LPORT1 при обмене данными или от канала DMA LportCh0
10	LSrq1	Запрос обслуживания от порта LPORT1
11	LTRx2	Прерывание от порта LPORT2 при обмене данными или от канала DMA LportCh0
12	LSrq2	Запрос обслуживания от порта LPORT2
13	LTRx3	Прерывание от порта LPORT3 при обмене данными или от канала DMA LportCh0
14	LSrq3	Запрос обслуживания от порта LPORT3
18:15	-	Резерв
19	Compare	Прерывание от таймера CPU
20	-	Резерв
21	MemCh0	Прерывание от канала DMA MemCh0
22	MemCh1	Прерывание от канала DMA MemCh1
23	MemCh2	Прерывание от канала DMA MemCh2
24	MemCh3	Прерывание от канала DMA MemCh3
28:25	-	Резерв
29	Timer	Прерывание от таймеров IT, WDT, RTT
30	PI	Программное прерывание от DSP-ядра.
31	SBS	Признаки: Переполнение стека DSP-ядра. Останова DSP-ядра в результате сравнения содержимого программного счетчика с адресом останова. Остановка DSP-ядра при завершении требуемого числа шагов при пошаговом исполнении программы. Выполнение DSP-ядром команды STOP.

В процессе обслуживания прерывания необходимо проанализировать состояние устройства для определения причины его возникновения. Сброс прерывания осуществляется в момент исключения причины возникновения данного прерывания. Например, прерывание от **LPORT** (при неактивизированном **DMA**) сбрасывается при записи данных в буфер **LTx** или при чтении данных из буфера **LRx**.

Все незамаскированные прерывания объединяются по «или» и поступают в разряд `IP[5]` регистра **Cause CP0**.

Исходное состояние регистра **QSTR** – нули.

Регистр **Cause** и другие регистры **CP0**, управляющие обслуживанием прерываний, рассматриваются здесь.

Каждое внутреннее прерывание маскируется при помощи 32-разрядного регистра маски **MASKR**, формат которого аналогичен формату регистра **QSTR**. Исходное состояние данного регистра – нули (все внутренние прерывания запрещены). Регистр доступен по записи и чтению.

8.4. Регистры CP0, управляющие обслуживанием прерываний

Системный Управляющий Сопроцессор (**CP0**) обеспечивает регистровый интерфейс с процессорным ядром *MIPS32* и поддерживает управление памятью, преобразование адреса, обработку исключений и другие привилегированные операции. Каждому регистру **CP0** соответствует определяющий его уникальный номер; этот номер называется *номером регистра*. Например, регистру **Status** соответствует 12-й номер регистра. Обмен данными между **CPU** и **CP0** осуществляется посредством команд **`mtc0`** и **`mfc0`** (подробнее см. книгу "**RISC Instructions Set**").

Здесь рассматриваются регистры **CP0**, управляющие обработкой прерываний - регистры **Status** и **Cause**.

Регистр Status (регистр 12 CP0)

Регистр **Status** (**SR**) является регистром, доступным для чтения и записи. Он содержит поля рабочего режима, разрешения прерываний и диагностические состояния процессора. Для задания режимов функционирования процессора поля этого регистра объединяются следующим образом:

Разрешение прерываний: Прерывания разрешаются, когда истинны все следующие условия:

- `IE=1`
- `EXL=0`
- `ERL=0`

Если эти условия выполнены, прерывания разрешаются установкой битов IM.

Рабочие режимы: Процессор всегда находится в одном из двух режимов – **Kernel** или **User**. Режим задается установкой следующих битов регистра **Status CP0**.

- Режим **User**: `UM=1`, `EXL=0`, и `ERL=0`
- Режим **Kernel**: `UM=0`, или `EXL=1`, или `ERL=1`

В таблице 7.2 приведен формат регистра **Status**:

Таблица 7.2. Формат регистра Status.

31 28	27	26 23	22	21	20	19	18 16	15 8	7 5	4	3	2	1	0
CU3-CU0	RP	0	BEV	TS	0	NMI	0	IM7-IM0	0	UM	0	ERL	EXL	IE

Для процесса обработки внутренних и внешних прерываний важны следующие поля данного регистра:

- BEV - данное поле управляет размещением векторов исключений (0 - нормальный вектор, 1 - начальная загрузка). Напомним, что возникновение прерывания влечет за собой возникновение исключения по прерыванию;
- NMI - если значение данного поля - единица, то произошло немаскируемое прерывание;
- IM[7:0] - в данном поле осуществляется установка разрешения/запрещения запроса на прерывание (0 - запрос запрещен, 1 - запрос разрешен). Прерывания принимаются только если установлены соответствующие им биты IM, а также бит IE;
- UM - указывает на режим работы процессора (0 - **Kernel**, 1 - **User**). Напомним, что процессор может работать в режиме **Kernel** даже если `UM=1`, но выполняется одно из указанных выше условий;
- ERL - Уровень ошибки. Установка в единицу данного поля переводит процессор в режим **Kernel** и запрещает прерывания.
- EXL - Уровень исключения. Установка в единицу данного поля переводит процессор в режим **Kernel** и запрещает прерывания;
- IE - Поле разрешения прерываний (0 - запрещает прерывания, 1 - разрешает прерывания).

Регистр Cause (регистр 13 CP0)

Основное назначение регистра **Cause** - описывать причину последнего возникшего исключения. Кроме того, поля регистра управляют запросами на программные прерывания и определяют вектор, которыми обрабатываются прерывания. Все поля регистра **Cause** являются доступными только по чтению, за исключением полей IP[1:0], IV и WP.

Формат регистра **Cause** приводится в таблице 7.3:

Таблица 7.3. Формат регистра Cause.

31	30 24	23	22 16	15 10	9 8	7	6 2	1 0
BD	0	IV	0	IP[7:2]	IP[1:0]	0	Exc Code	0

Для обработки прерываний имеет значение содержимое следующих полей:

- IV - указывает, какой вектор должен использоваться для обработки прерываний (0 - общий вектор, смещение `0x180`; 1 - специальный вектор, смещение `0x200`);
- IP[7:2] - указывает, какие внешние прерывания установлены:
  - 15 - Аппаратное прерывание 5 или прерывание по таймеру;
  - 14 - Аппаратное прерывание 4;
  - 13 - Аппаратное прерывание 3;
  - 12 - Аппаратное прерывание 2;
  - 11 - Аппаратное прерывание 1;

- 10 - Аппаратное прерывание 0;
- IP[1:0] - указывает, какие программные прерывания установлены: 9 - программное прерывание 1, 8 - программное прерывание 0;
- ExcCode - поле содержит код исключения. Для исключения по прерыванию ExcCode=0.

Способы обработки исключительных ситуаций (в том числе и исключений по прерыванию) рассматриваются в разделе "Обработка исключений".

Примечание: в некоторых случаях, обработку прерываний обслуживают не только регистры **Status** и **Cause**, но и другие. Подробнее эти регистры рассматриваются в разделе "Системный управляющий сопроцессор".

8.5. Прерывания от DSP-ядра

При взаимодействии процессорных ядер RISC и DSP, ядро DSP может формировать в RISC следующие запросы на прерывания:

- программное;
- по переполнению стека;
- при выполнении команды **STOP**;
- при достижении адреса останова в том случае, если программа исполнялась до адреса останова; при завершении требуемого числа шагов в случае пошагового исполнения программы.

Ядро RISC в DSP прерываний не формирует.

Для разрешения перечисленных прерываний от ядра DSP необходимо установить в единицу бит IM[7] регистра **Status** (**CP0**), а также 31 бит регистра **MASKR**. Кроме того, для разрешения программного прерывания необходимо установить в единицу бит **MASKR**[30].

Запрос на аппаратное прерывание формируется в 31-м бите регистра **QSTR**.

Запрос на программное прерывание от DSP устанавливается в поле **QSTR**[30]. Чтобы запросить прерывание из программы DSP, необходимо установить бит PI (**DCSR**[0]) в единицу. Например:

```
MOVE DCSR,R0
ORL  0x0001,R0,R0
MOVE R0,DCSR
```

Данный код записывает в **DCSR**[0] единицу по "*или*", то есть сохраняет значения остальных полей. Таким образом, при помощи программного прерывания от DSP можно сигнализировать о том или ином событии ядру RISC не останавливая программу DSP.

Подробнее взаимодействие RISC и DSP рассматривается в разделе "Взаимодействие с ядром DSP".

9. Таймеры

9.1. Введение

В данной главе рассматриваются таймеры ИМС "*МУЛЬТИКОР*", их характеристики, регистры, а также способы работы с ними. Глава включает в себя описания:

- Интервального таймера;
- Таймера реального времени;
- Сторожевого таймера.

9.2. Интервальный таймер

9.2.1. Введение

Интервальный таймер (**IT**) предназначен для выработки периодических прерываний на основе деления тактовой частоты **CPU**. Основные характеристики интервального таймера:

- Число разрядов основного делителя – 32;
- Число разрядов предделителя – 8;
- Программное управление стартом и остановкой таймера;
- Доступ ко всем регистрам обеспечивается в любой момент времени.

В данном разделе рассматривается:

- Структурная схема интервального таймера;
- Описание регистров;
- Способы программирования интервального таймера;
- Пример программы с использованием интервального таймера.

9.2.2. Структурная схема

На рисунке 8.1 приведена структурная схема интервального таймера:

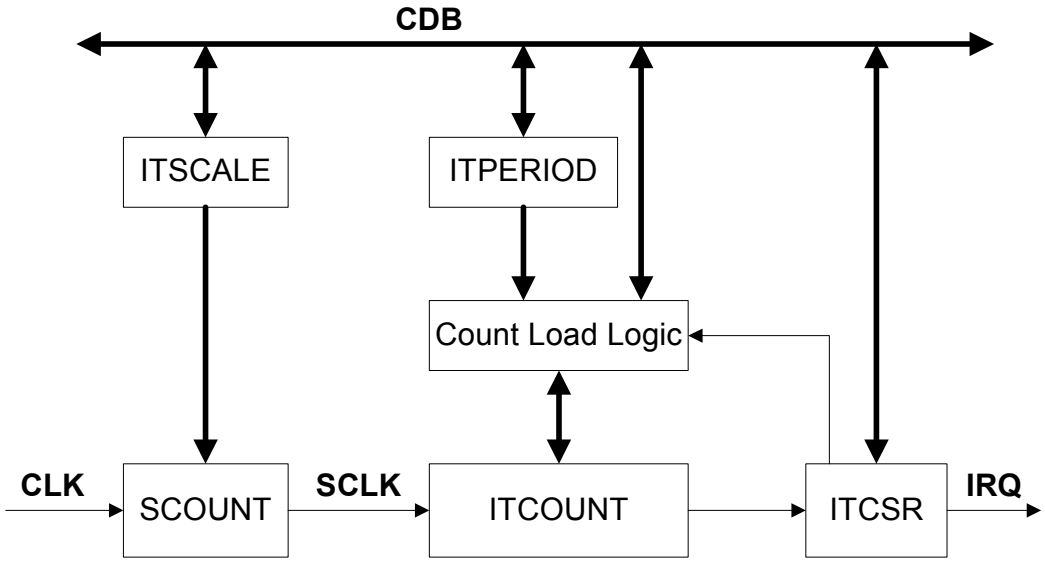


Рисунок 8.1. Структурная схема интервального таймера.

В состав интервального таймера входят следующие основные узлы:

- **ITCSR** - регистр управления и состояния;
- **ITCOUNT** - счетчик основного делителя;
- **ITPERIOD** - регистр периода основного делителя;
- **ITSCALE** - регистр предделителя;
- **SCOUNT** – счетчик предделителя;
- **Count Load Logic** - логика загрузки счетчика основного делителя.

На структурной схеме интервального таймера использованы следующие обозначения:

- **CDB** – шина данных **CPU**;
- **CLK** – тактовая частота работы **CPU**;
- **S_CLK** – выходная частота предделителя;
- **IRQ** – запрос на прерывание от интервального таймера.

9.2.3. Регистры

Перечень программно доступных регистров интервального таймера **IT** приведен в таблице 8.1:

Таблица 8.1. Программно доступные регистры интервального таймера.

Условное обозначение регистра	Название регистра	Тип доступа	Исходное состояние
ITCSR[2:0]	Регистр управления и состояния	W/R	0
ITPERIOD[31:0]	Регистр периода	W/R	FFFF_FFFF
ITCOUNT[31:0]	Регистр счетчика основного делителя частоты	W/R	0000_0000
ITSCALE[7:0]	Регистр предделителя частоты	W/R	0000

В таблице 8.2 приводится формат регистра **ITCSR**:

Таблица 8.2. Формат регистра ITCSR.

Номер разряда	Условное обозначение	Описание
0	EN	Разрешение работы таймера: 0 – запрещение работы (неактивное состояние таймера); 1 – разрешение работы (активное состояние таймера).
1	INT	Признак срабатывания таймера. Состояние данного разряда транслируется в бит Timer регистра QSTR (на входе этого регистра он объединяется по логическому "или" с одноименными разрядами регистров управления и состояния таймеров WDT и RTT). Сбрасывается при записи нуля в этот разряд.

8-разрядный регистр **ITSCALE** используется для задания коэффициента предделения тактовой частоты **CPU** (**CLK**), которая поступает на вход счетчика **SCOUNT**.

32-разрядные регистр **ITPERIOD** используется для задания периода работы основного делителя.

32-разрядный счетчик основного делителя частоты **ITCOUNT** работает в режиме декремента. На вход этого счетчика поступает частота (**S_CLK**) с выхода счетчика предделителя.

9.2.4. Программирование интервального таймера

Перед началом работы с интервальным таймером необходимо загрузить значение периода в регистр **ITPERIOD** и значение коэффициента предделения частоты в регистр **ITSCALE**.

Для активизации таймера необходимо в бит EN регистра **ITCSR** записать 1. В момент этой записи содержимое регистров **ITSCALE** и **ITPERIOD** переписывается в счетчики **SCOUNT** и **ITCOUNT** соответственно. После этого оба счетчика начинают работать в режиме декремента. При этом предделитель работает от частоты **CLK**, а счетчик **ITCOUNT** – от частоты **S_CLK**, формируемой предделителем.

Когда оба счетчика **SCOUNT** и **ITCOUNT** достигают нулевого состояния, в регистре **ITCSR** устанавливается бит INT и формируется запрос на прерывание QSTR[29] (бит TIMER), а содержимое регистров **ITSCALE** и **ITPERIOD** опять переписывается в счетчики **SCOUNT** и **ITCOUNT** соответственно. Далее таймер работает аналогичным образом.

Запрос на прерывание формируется каждые {(**itperiod** + 1)*(**itscale** + 1)} тактов работы **CPU**, где **itperiod** и **itscale** – содержимое регистров **ITPERIOD** и **ITSCALE** соответственно.

При необходимости, в любой момент времени в **ITCOUNT** и **ITPERIOD** можно произвести запись новых данных и тем самым изменить значение отрабатываемого временного интервала.

Обслуживание срабатывания интервального таймера осуществляется одним из следующих способов:

- Обслуживание по опросу: следует программно опрашивать состояние бита INT регистра **ITCSR**. При срабатывании интервального таймера бит INT устанавливается в единицу;
- Обслуживание по прерыванию: в том случае, если прерывание интервального таймера разрешено регистрами Status (**CP0**) и MASKR - следует использовать обработчик исключения по прерыванию.

Пример программы, использующей интервальный таймер (обслуживание по прерыванию), рассматривается здесь.

9.2.5. Пример программы с использованием интервального таймера

Рассмотрим пример программы, использующей интервальный таймер. Программа состоит из двух файлов - *main.c* и *ProcITimer.s*. Кроме того, к файлу *main.c* подключается заголовочный файл *memory_12.h*, описывающий адресное пространство *MultiCore-12*. Файл *main.c* содержит текст основной программы RISC, написанной на языке *C*:

```
#include "memory_12.h"

extern int i;

main()
{
      MASKR |= 1<<29;          //разрешение прерывания по таймеру

      asm("or   $4,$0,$0");    //обнуление GPR#4
      asm("li   $4,0x8001");   //0x8001 -> GPR#4
      asm("mtc0 $4,$12");      //GPR-4 -> Status (CP0) (разрешение прерываний
таймеров)
```

```
        ITPERIOD=0x1f;          //ITPERIOD=0x1f (период равен ITPERIOD+1=32)
        ITSCALE=0x1;            //предделитель=0x1 (общий период будет равен
32*(ITSCALE+1)=64)
        ITCSR=0x1;              //включение таймера

        while (1)
        {
              i++;
        }
}
```

Программа файла *main.c* осуществляет следующие действия:

- Разрешает прерывания таймеров в регистре MASKR (бит TIMER);
- Разрешает прерывания таймеров в регистре Status (**CP0**);
- Настраивает период и предделитель интервального таймера так, что запрос на прерывание генерируется каждые 64 такта работы **CPU**;
- Входит в бесконечный цикл, в котором инкрементирует значение внешней переменной `i`.

Каждые 64 такта интервальный таймер генерирует запрос на прерывание (`QSTR[29]`). Так как прерывание разрешено регистрами Status и MASKR, управление передается обработчику исключений по прерыванию, описанному в файле *ProcITimer.s*:

```
  .text
   .global i

TimerInterrupt:                     #обработчик исключения по прерыванию
               li   $3,0x1          #1 -> GPR#3
               lui  $30,0xb82f      #CPU_BASE -> GPR#30
               sw   $3,0xd000($30)  #1 -> ITCSR (INT=0, EN=1)
               la   $30,i           #&i -> GPR#30
               lw   $3,($30)        #i  -> GPR#3
               la   $30,0xB8400000  #&XRAM -> GPR#30
               sw   $3,($30)        #i -> XRAM
               eret                 #Возврат из исключения

  .data
i: .space 4,0
```

Секция текста обработчика `TimerInterrupt` расположена по адресу `0x80000180` (см. страницу "Расположение векторов исключений"). Секция данных (переменная `i`) скомпонована вслед за секциями файла *main.c*.

Обработчик исключения по прерыванию от интервального таймера получает управление каждые 64 такта **CPU** и осуществляет следующие действия:

- Сбрасывает прерывание интервального таймера (обнулением бита INT регистра **ITCSR**);
- Сохраняет текущее значение переменной `i` в **XRAM** DSP (по адресу `0xB8400000`);
- Возвращается в основную программу RISC по команде **`eret`**.

В результате работы программы по адресу `0xB8400000` каждые 64 такта будет записываться новое значение счетчика `i`.

9.3. Таймер реального времени

9.3.1. Введение

Таймер реального времени (**RTT**) предназначен для выработки периодических прерываний на основе деления внешней тактовой частоты **RTCXTI**. Основные характеристики таймера реального времени:

- Число разрядов делителя – 32;
- Программное управление стартом и остановкой таймера;
- Доступ ко всем регистрам обеспечивается в любой момент времени.

В данном разделе рассматривается:

- Структурная схема таймера реального времени;
- Описание регистров;
- Способы программирования таймера реального времени.

9.3.2. Структурная схема

На рисунке 8.2 приведена структурная схема таймера реального времени:

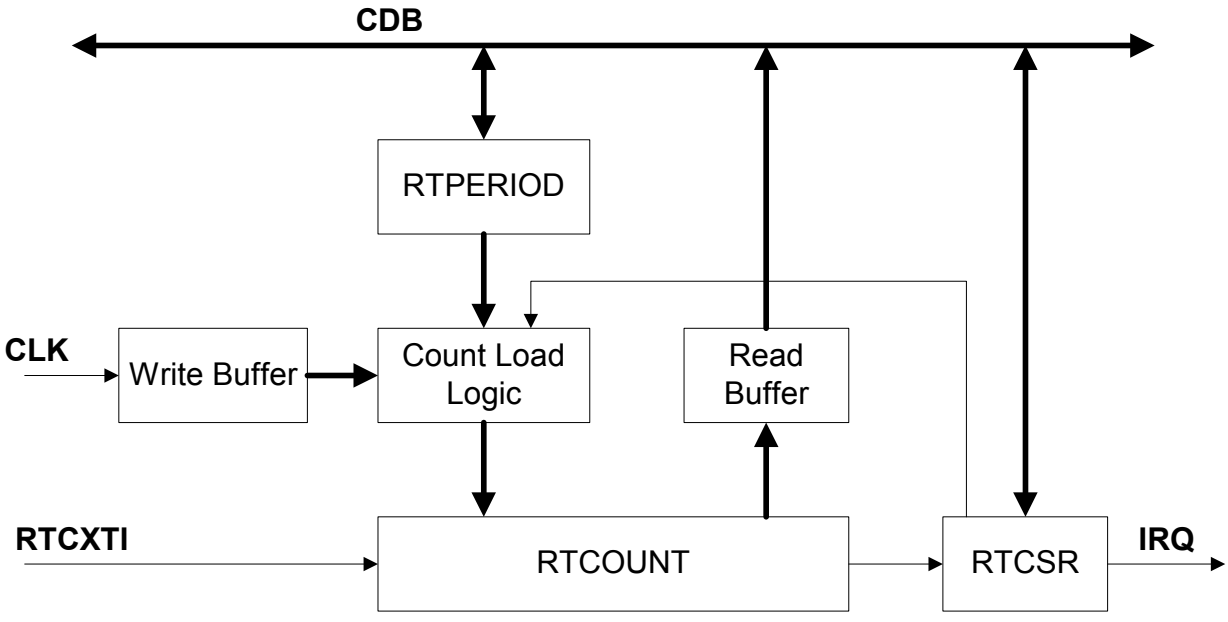


Рисунок 8.2. Структурная схема таймера реального времени.

В состав таймера реального времени входят следующие основные узлы:

- **RTCSR** - регистр управления и состояния;
- **RTCOUNT** - счетчик основного делителя;
- **RTPERIOD** - регистр периода основного делителя;
- **Count Load Logic** - логика загрузки счетчика основного делителя;
- **Write Buffer** – буфер записи;
- **Read Buffer** – буфер чтения.

На структурной схеме интервального таймера использованы следующие обозначения:

- **CDB** – шина данных **CPU**;
- **CLK** – тактовая частота работы **CPU**;
- **RTCXTI** – внешняя тактовая частота;
- **IRQ** – запрос на прерывание от таймера реального времени.

На вход таймера реального времени поступает внешняя тактовая частота **RTCXTI**. Для правильной работы **RTT** должно выполняться соотношение: $f_{RTCXTI} \leq \frac{f_{CLK}}{7}$, где f_{RTCXTI} и f_{CLK} - значения частот **RTCXTI** и **CLK** соответственно.
Как правило, **RTCXTI** имеет частоту 32,768 кГц.

9.3.3. Регистры

Перечень всех программно доступных регистров таймера реального времени приведен в таблице 8.3:

Таблица 8.3. Программно доступные регистры таймера реального времени.

Условное обозначение регистра	Название регистра	Тип доступа	Исходное состояние
RTCSR[1:0]	Регистр управления и состояния	W/R	0
RTPERIOD[31:0]	Регистр периода	W/R	0000_7FFF
RTCOUNT[31:0]	Регистр счетчика делителя	W/R	0000_0000

Формат регистра **RTCSR** приведен в таблице 8.4:

Таблица 8.4. Формат регистра RTCSR.

Номер разряда	Условное обозначение	Описание
0	EN	Разрешение работы таймера: 0 – запрещение работы (неактивное состояние таймера); 1 – разрешение работы (активное состояние таймера).
1	INT	Признак срабатывания таймера. Состояние данного разряда транслируется в бит Timer регистра QSTR (на входе этого регистра он объединяется по логическому «или» с одноименными разрядами регистров управления и состояния таймеров WDT и IT). Сбрасывается при записи нуля в этот разряд.

32-разрядный регистр **RTPERIOD** используется для задания периода работы таймера. Если **RTPERIOD**=0x00007FFF, а частота **RTCXTI**=32,768 кГц, то таймер реального времени формирует прерывание каждую секунду.

32-разрядный счетчик **RTCOUNT** работает в режиме декремента от частоты **RTCXTI**.

9.3.4. Программирование таймера реального времени

Перед началом работы с таймером необходимо загрузить данные в регистр **RTPERIOD**.

Для активизации таймера необходимо в бит EN регистра **RTCSR** записать 1. В момент этой записи содержимое регистра **RTPERIOD** переписывается в счетчик **RTCOUNT**, который начинает работать в режиме декремента. Когда счетчик **RTCOUNT** достигнет нулевого состояния, в регистре **RTCSR** устанавливается бит INT и формируется запрос на прерывание QSTR[29] (бит TIMER), а содержимое регистра **RTPERIOD** опять переписывается в счетчик **RTCOUNT**. Далее таймер работает аналогичным образом.

При необходимости, в любой момент времени в **RTPERIOD** и **RTCOUNT** можно произвести запись новых данных и тем самым изменить значение, отрабатываемого временного интервала.

Следует отметить, что при записи в **RTCOUNT**, обновление его содержимого происходит с задержкой, равной периоду **RTCXTI**.

9.4. Сторожевой таймер

9.4.1. Введение

Сторожевой таймер (**WDT**) предназначен для:

- вывода системы из "зависания", если программное обеспечение зациклилось и не формирует соответствующих управляющих воздействий;
- выработки прерываний на основе деления тактовой частоты **CPU**.

Основные характеристики таймера:

- число разрядов основного делителя – 32;
- число разрядов предделителя – 8;
- программное управление стартом и остановкой таймера;
- два режима работы: режим сторожевого таймера (**WDM**) и режим интервального таймера (**ITM**);
- два режима отработки временных интервалов: однократный и периодический;
- доступ ко всем регистрам обеспечивается в любой момент времени.

В данном разделе рассматривается:

- Структурная схема сторожевого таймера;
- Описание регистров;
- Способы программирования сторожевого таймера;
- Пример программы, использующей сторожевой таймер.

9.4.2. Структурная схема

Структурная схема сторожевого таймера приведена на рисунке 8.3:

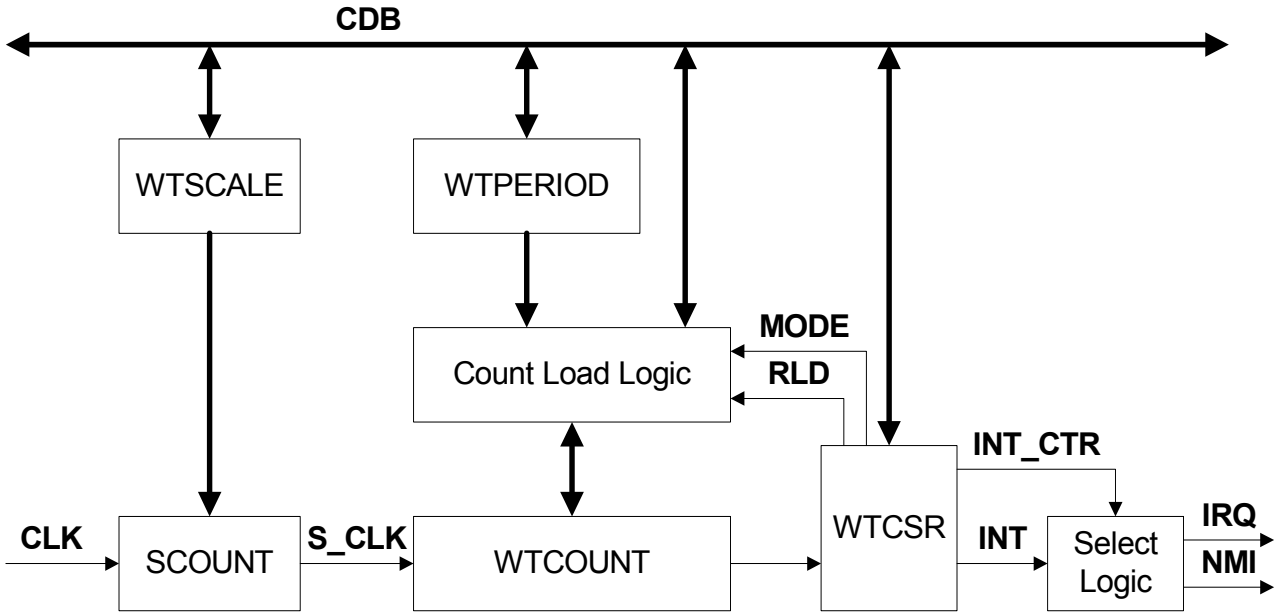


Рисунок 8.3. Структурная схема сторожевого таймера.

В состав сторожевого таймера входят следующие основные узлы:

- **WTCSR** - регистр управления и состояния;
- **WTCOUNT** - счетчик основного делителя;
- **WTPERIOD** - регистр периода основного делителя;
- **WTSCALE** - регистр предделителя;
- **SCOUNT** – счетчик предделителя;
- **Count Load Logic** - логика загрузки счетчика основного делителя.

На структурной схеме интервального таймера использованы следующие обозначения:

- **CDB** – шина данных **CPU**;
- **CLK** – тактовая частота работы **CPU**;
- **S_CLK** – выходная частота предделителя;
- **IRQ** – запрос на прерывание от интервального таймера;
- **NMI** – немаскируемое прерывание.

9.4.3. Регистры

Перечень программно доступных регистров сторожевого таймера приведен в таблице 8.5:

Таблица 8.5. Программно доступные регистры сторожевого таймера.

Условное обозначение регистра	Название регистра	Тип доступа	Исходное состояние
WTCSR[14:0]	Регистр управления и состояния	W/R	0000
WTPERIOD[31:0]	Регистр периода	W/R – в неактивном состоянии; R – в активном состоянии.	FFFF_FFFF
WTCOUNT[31:0]	Регистр счетчика основного делителя частоты	W/R – в неактивном состоянии; R – в активном состоянии.	0000_0000
WTSCALE[15:0]	Регистр предделителя частоты	W/R – в неактивном состоянии; R – в активном состоянии.	0000

8-разрядный регистр **WTSCALE** используется для задания коэффициента предделения тактовой частоты **CPU** (**CLK**), которая поступает на вход счетчика **SCOUNT**.

32-разрядный регистр **WTPERIOD** используется для задания периода работы основного делителя.

32-разрядный счетчик основного делителя частоты **WTCOUNT** работает в режиме декремента. На вход этого счетчика поступает частота **S_CLK** с выхода счетчика предделителя.

В таблице 8.6 приведен формат регистра **WTCSR**:

Таблица 8.6. Формат регистра WTCSR.

Номер разряда	Условное обозначение	Описание
7: 0	KEY	Поле для записи ключей. Запись в это поле последовательности кодов A0 (ключ KEY1) и F5 (ключ KEY2) приводит к переключению таймера из режима сторожевого таймера (WDM) в режим интервального таймера (ITM). Поле доступно по чтению и записи. Поле доступно по записи только в режиме WDM: когда EN=1 или когда таймер находится в состоянии Timeout. Сбрасывается в ноль при переводе таймера из режима ITM в режим WDM. Значение в исходном состоянии – 0.
8	EN	Разрешение работы таймера: 0 – запрещение работы (неактивное состояние таймера); 1 – разрешение работы (активное состояние таймера). Доступен по чтению и записи. Запись нуля в этот бит при работе таймера в режиме WDM не имеет эффекта. Значение в исходном состоянии – 0.
9	INT	Признак срабатывания таймера. В зависимости от содержимого поля INT_CTR состояние данного разряда транслируется или в бит Timer регистра QSTR (на входе этого регистра он объединяется по логическому «или» с одноименными разрядами регистров управления и состояния таймеров RTT и IT), или в немаскируемое прерывание (NMI). Сбрасывается при записи нуля в этот разряд, а также при переводе таймера из режима ITM в режим WDM. Доступен по чтению и записи в режиме ITM и только по чтению в режиме WDM. Значение в исходном состоянии – 0.
10	MODE	Режим работы таймера: 0 – режим сторожевого таймера (WDM); 1 – режим обычного таймера (ITM). Доступен по чтению и записи при EN=0 и только по чтению при EN=1. Значение в исходном состоянии – 0.
11	RLD	Бит управления перезагрузкой SCOUNT и WTCOUNT при работе в режиме ITM: 0 – таймер однократно отрабатывает временной интервал и останавливается; 1 – таймер отрабатывает заданный временной интервал периодически. После отработки очередного временного интервала содержимое WTSCALE и WTPERIOD загружается в SCOUNT и WTCOUNT соответственно. Доступен по чтению и записи при EN=0 и только по чтению при EN=1. Значение в исходном состоянии – 0.
13: 12	INT_CTR	Управления типом прерывания, которое формируется таймером WDT: 00, 11 – прерывание не формируется; 01 – обычное прерывание (QSTR[29]). Как правило, используется в режиме ITM; 10 – немаскируемое прерывание (NMI). Как правило, используется в режиме WDM. Поле доступно по чтению и записи при EN=0 и только по чтению при EN=1. Значение в исходном состоянии – 0.

9.4.4. Программирование сторожевого таймера

В исходном состоянии **WDT** находится в режиме сторожевого таймера. Для перевода его в режим интервального таймера необходимо записать 1 в бит MODE регистра **WTCSR**. Следует отметить, что смена режима работы таймера посредством записи в бит MODE возможна, только если таймер не активен (EN=0).

Перед началом работы с таймером **WDT** необходимо загрузить значение периода в регистр **WTPERIOD** и значение коэффициента предделения частоты в регистр **WTSCALE**.

Для активизации таймера необходимо в бит EN регистра **WTCSR** записать 1. В момент этой записи содержимое регистров **WTSCALE** и **WTPERIOD** переписывается в счетчики **SCOUNT** и **WTCOUNT** соответственно. После этого оба счетчика начинают работать в режиме декремента. При этом предделитель работает от частоты **CLK**, а счетчик **WTCOUNT** – от частоты **S_CLK**, формируемой предделителем.

После активизации таймера, **WTCOUNT**, **WTPERIOD**, **WTSCALE**, а также поля INT_CTR, MODE и RLD регистра **WTCSR**, становятся недоступными по записи.

Сторожевой таймер в режиме WDM необходимо периодически обслуживать. То есть, если он был активизирован в режиме WDM, то для того, чтобы не возникло состояния *Timeout* необходимо периодически выполнять следующую последовательность действий:

- переключить таймер из режима WDM в режим ITM посредством последовательной записи в поле KEY регистра **WTCSR** кодов 0xA0 и 0xF5;
- остановить таймер посредством записи 0 в бит EN регистра **WTCSR**;
- установить MODE=0.

В случае, если вслед за значением 0xA0 в поле KEY будет записано значение, не равное 0xF5, таймер перейдет в состояние *Timeout*.

Если после активизации таймера в режиме WDM, он не будет переведен в режим ITM, то, когда оба счетчика **SCOUNT** и **WTCOUNT** достигнут нулевого значения, таймер перейдет в состояние *Timeout*.

В состоянии *Timeout* таймер формирует признак INT и останавливается, а запись в какой-либо из его регистров блокируется. Для вывода **WDT** из состояния *Timeout* необходимо его переключить в режим ITM посредством последовательной записи в поле KEY регистра **WTCSR** кодов 0xA0 и 0xF5.

При переключении таймера из неактивного состояния в режиме ITM в режим WDM путем записи 0 в поле MODE регистра **WTCSR** происходит обнуление поля KEY.

При работе таймера в режиме ITM при RLD=0 он однократно отрабатывает заданный временной интервал, устанавливает INT=1 и останавливается (когда оба счетчика **SCOUNT** и **WTCOUNT** достигают нулевого состояния). Если RLD=1, то каждый раз после достижения счетчиками нулевого состояния и установки INT=1, происходит перезагрузка значений

периода и коэффициента предделения частоты. То есть, таймер отрабатывает заданный временной интервал периодически до тех пор, пока он не будет остановлен.

Запрос на прерывание формируется каждые `{(wtperiod + 1)*(wtscale + 1)}` тактов работы **CPU**, где **wtperiod** и **wtscale** – содержимое регистров **WTPERIOD** и **WTSCALE** соответственно.

Пример программы, использующей сторожевой таймер (в режиме WDT), приведен здесь.

9.4.5. Пример программы, использующей сторожевой таймер

Рассмотрим пример программы, использующей сторожевой таймер в режиме WDT. Программа написана на языке Ассемблера и состоит из трех файлов: *start.s*, *main.s* и *WDTfunc.s*. Кроме того, к каждому из файлов директивой **.include** подключен заголовочный файл *memory_12_asm.h*, описывающий адресное пространство *MultiCore-12*.

Файл *start.s* содержит обработчик исключений по аппаратному сбросу (Reset) и немаскируемому прерыванию (NMI). Обработчик расположен по адресу `0xBFC00000`, с которого и стартует программа. Рассмотрим текст файла *start.s*:

```
   .include "memory_12_asm.h"
   .text

 Reset:                          #Обработчик исключений Reset и NMI
       li    $4,1<<19
       mfc0  $3,$12              #чтение содержимого регистра Status
       and   $3,$4               #проверка бита NMI
       beq   $3,$4,NMI_proc      #если бит NMI=1, переход к обработчику NMI
       nop                       #delay slot

       li    $4,0xff11
       mtc0  $4,$12              #переход в режим user, разрешение всех прерываний

       j     Main                #переход к основной программе
       nop                       #delay slot

 NMI_proc:                       #Обработчик исключений NMI
       li    $4,0xa0             #KEY1 -> GPR#4
       lui   $30,CPU_BASE        #базовый адрес -> GPR#30
       sb    $4,WTCSR($30)       #KEY1 -> WTCSR[7:0] (KEY)
       li    $4,0xf5             #KEY2 -> GPR#4
       sb    $4,WTCSR($30)       #KEY2 -> WTCSR[7:0] (KEY)
       sw    $0,WTCSR($30)       #сброс таймера WDT

       li    $4,0xff11
       mtc0  $4,$12              #переход в режим user, разрешение всех прерываний

       j     Main                #переход к основной программе
       nop                       #delay slot
```

Программа стартует с метки `Reset` и осуществляет следующие действия:

- Проверяет бит NMI регистра Status (**CP0**);
- Если бит не установлен, программа переводит ядро в режим **User**, разрешает все прерывания и передает управление основной программе (по метке `Main`);
- Если бит установлен, значит обработчик получил управление по немаскируемому прерыванию. В таком случае происходит переход к обработчику **NMI** - `NMI_proc`.

Обработчик `NMI_proc` переводит сторожевой таймер в режим ITM (последовательной записью в поле KEY значений KEY1 и KEY2). Затем, таймер сбрасывается, ядро переводится в режим **User**, все прерывания разрешаются и осуществляется переход по метке `Main` к основной программе.

Текст основной программы содержится в файле *main.s*:

```
  .include "memory_12_asm.h"
  .text
   .global Main

Main:
     lui   $30,CPU_BASE
     li    $4,0x3ff
     sw    $4,WTPERIOD($30) #установка периода WDT в 1024 такта
     li    $4,0x2900
     sw    $4,WTCSR($30)    #запуск сторожевого таймера в режиме WDT

     or    $3,$0,$0
     li    $5,0xff
     li    $4,0xff


     #цикл с обслуживанием WDT
Cycle1:
     or    $2,$0,$0
           SubCycle:
                    addi   $2,0x1               #увеличение счетчика
                    bne    $2,$4,SubCycle       #вложенный цикл от 0 до 0xff
                    nop                         #delay slot
     addi  $3,0x1          #увеличение счетчика
     jal   WDTmaintain     #переход на обслуживание WDT
     nop                   #delay slot

     bne   $3,$5,Cycle1    #цикл от 0 до 0xffff
     nop                   #delay slot

     #цикл без обслуживания WDT
Cycle2:
     nop
     j     Cycle2
     nop
```

Основная программа осуществляет следующие действия:

- Настраивает сторожевой таймер на режим WDT с периодом в 1024 такта;
- Входит в цикл `Cycle1` (от `0` до `0xff`);

- В цикле `Cycle1` исполняется вложенный цикл `Cycle2` (от `0` до `0xff`), а затем - происходит переход на процедуру обслуживания сторожевого таймера `WDTmaintain`;
- После завершения цикла `Cycle1` программа входит в бесконечный цикл `Cycle2`.

Текст процедуры обслуживания сторожевого таймера `WDTmaintain` расположен в файле *WDTfunc.s*:

```
  .include "memory_12_asm.h"
  .text
   .global WDTmaintain

WDTmaintain:                    #процедура обслуживания WDT
      li    $6,0xa0             #KEY1 -> GPR#6
      lui   $30,CPU_BASE        #базовый адрес -> GPR#30
      sb    $6,WTCSR($30)       #KEY1 -> WTCSR[7:0] (KEY)
      li    $6,0xf5             #KEY2 -> GPR#6
      sb    $6,WTCSR($30)       #KEY2 -> WTCSR[7:0] (KEY)
      sw    $0,WTCSR($30)       #сброс таймера WDT
      li    $6,0x2900
      sw    $6,WTCSR($30)       #перезапуск WDT
      jr    $31                 #возврат из обслуживания WDT
      nop                       #delay slot
```

Процедура `WDTmaintain` осуществляет обслуживание сторожевого таймера так, чтобы не возникло состояния *Timeout*. Для этого таймер сначала переводится в режим ITM (записью KEY1 и KEY2 в поле KEY регистра **WTCSR**), затем останавливается, после чего перезапускается в режиме WDT. При этом отсчет заданного числа тактов начинается заново.

Использование процедуры `WDTmaintain` позволяет программе функционировать в нормальном режиме без возникновения состояния *Timeout*. Таким образом, цикл `Cycle1` успешно завершается.

Затем, программа входит в бесконечный цикл `Cycle2`, используемый для симуляции "зависания" системы. Этот цикл не применяет процедуру `WDTmaintain` для обслуживания сторожевого таймера. Поэтому, по истечении 1024 тактов таймером будет сгенерировано немаскируемое прерывание, выводящее программу из цикла `Cycle2`.

10. Создание симулятора внешнего устройства

В данной главе рассматривается процесс создания симулятора внешнего устройства, подключаемого к симулятору процессора MultiCore. Под внешним устройством подразумевается целевое устройство, которое должно функционировать во взаимодействии с процессором MultiCore. Создание и использование симулятора внешнего устройства существенно облегчает отладку этого взаимодействия.

Симулятор внешнего устройства выполняется в виде библиотеки DLL. Возможно описание нескольких устройств в одной библиотеке. Рекомендуется создавать DLL в среде *MS Visual C++*. Если же Вы используете иную среду разработки, необходимо обеспечить совместимость используемых типов данных и экспортируемых функций. Для этого обратитесь к документации по используемой среде.

В дальнейшем, речь пойдет о разработке внешнего устройства только в среде *MS Visual C++*.

Библиотека DLL должна содержать:

- класс внешнего устройства. Для создания этого класса используйте абстрактный класс IDevice;
- диалог настройки устройства;
- три экспортируемые функции (*GetDeviceCount*, *GetDeviceName* и *CreateDevice*).

10.1. Абстрактный класс IDevice

Библиотека внешнего устройства обязательно должна содержать заголовок абстрактного класса **IDevice**. Важно знать, что при взаимодействии симулятора MultiCore с Вашей DLL, симулятор вызывает именно методы класса **IDevice**. Поэтому, класс внешнего устройства следует создавать как наследный от **IDevice**.

Заголовок абстрактного класса **IDevice** должен выглядеть так:

```
class IDevice
{
public:
        virtual LPCSTR __stdcall GetDeviceName() = 0;
        virtual DWORD __stdcall GetDeviceCoreFileName(LPSTR lpFileName, DWORD nSize)
= 0;
        virtual VOID __stdcall Step() = 0;
        virtual HWND __stdcall Configure(HWND hwndParent) = 0;
        virtual BOOL __stdcall SendData(DWORD dwData) = 0;
        virtual BOOL __stdcall ConnectPort (IPort* lpPort) = 0;
        virtual IPort* __stdcall GetConnectedPort() = 0;
        virtual VOID __stdcall SetConfigurationFile(LPSTR szFileName) = 0;
        virtual VOID __stdcall Release() = 0;
};
```

Методы `ConnectPort` и `GetConnectedPort` ссылаются на класс IPort. Заголовок этого класса также должен быть включен в исходный код Вашей DLL.

10.2. Абстрактный класс IPort

Абстрактный класс **IPort** содержит правила описания всех портов симулятора. Для создания модели внешнего устройства не следует наследовать класс **IPort**, но, тем не менее, заголовок этого класса необходимо включить в DLL. **IPort** используется некоторыми функциями абстрактного класса IDevice, а также служит для передачи данных симулятору MultiCore.

Заголовок абстрактного класса **IPort** должен выглядеть так:

```
class IPort
{
public:
        virtual BOOL __stdcall SendData(DWORD dwData) = 0;
        virtual BOOL __stdcall ConnectDevice(IDevice* lpDevice) = 0;
        virtual IDevice* __stdcall GetConnectedDevice() = 0;
        virtual LPSTR __stdcall GetPortName() = 0;
};
```

Заголовков классов IDevice и **IPort** достаточно, чтобы описать класс модели внешнего устройства.

10.3. Создание класса устройства

В классе модели внешнего устройства описываются все необходимые функции устройства. Класс должен быть наследным от абстрактного класса IDevice, причем все виртуальные функции класса IDevice должны быть описаны. Если какая-либо функция не нужна для работы внешнего устройства, ее следует описать как "пустую".

В класс модели внешнего устройства также рекомендуется включить указатель на подсоединенный порт, имя устройства, идентификатор и адрес диалога настроек устройства.

Пример заголовка класса модели внешнего устройства:

```
class CDevice1: public IDevice
{
public:
     CDevice1();
     ~CDevice1();

     //описание функций класса IDevice
         LPCSTR __stdcall GetDeviceName() = 0;
         DWORD __stdcall GetDeviceCoreFileName(LPSTR lpFileName, DWORD nSize) = 0;
```

```
        VOID __stdcall Step() = 0;
        HWND __stdcall Configure(HWND hwndParent) = 0;
        BOOL __stdcall SendData(DWORD dwData) = 0;
        BOOL __stdcall ConnectPort (IPort* lpPort) = 0;
        IPort* __stdcall GetConnectedPort() = 0;
        VOID __stdcall SetConfigurationFile(LPSTR szFileName) = 0;
        VOID __stdcall Release() = 0;

    //дополнительные функции
    static LRESULT CALLBACK ConfigDialogProc(HWND hwndDlg, UINT uMsg, WPARAM wParam,
LPARAM lParam);
};
```

Следует учесть, что все виртуальные функции, описанные в классе **IDevice**, предназначены для выполнения следующих действий:

- `GetDeviceName` - возвращает имя внешнего устройства;
- `GetDeviceCoreFileName` - заполняет `lpFileName` полным путем к DLL устройства;
- `Step` - выполняет один шаг устройства;
- `Configure` - создает диалог настроек устройства и возвращает его дескриптор, `hwndParent` - дескриптор родительского окна;
- `SendData` - осуществляет прием данных внешним устройством. Не вызывайте эту функцию - ее вызывает порт при поступлении на него данных. `dwData` - принимаемые данные;
- `ConnectPort` - соединяет устройство с портом. `lpPort` указывает на подсоединяемый порт. Рекомендуется сохранять `lpPort` в классе устройства. Не вызывайте `ConnectPort` - она вызывается симулятором MultiCore;
- `GetConnectedPort` - возвращает указатель на подсоединенный порт. Указатель на подсоединенный порт необходим для передачи данных симулятору MultiCore;
- `SetConfigurationFile` - функция вызывается при создании диалога настроек устройства. Если необходимо загружать какие-либо настройки из файла или иного источника, опишите загрузку из файла `szFileName` в теле этой функции;
- `Release` - функция вызывается при отсоединении устройства. Используйте ее для освобождения выделенной памяти и удаления экземпляра класса модели устройства.

Дополнительная функция `ConfigDialogProc` предназначена для обработки сообщений диалогом настроек устройства.

10.4. Диалог настроек устройства

Библиотека модели внешнего устройства должна содержать диалог настроек устройства. Если для работы устройства не требуется настроек, следует включить в библиотеку пустой диалог. Правила оформления диалога настроек следующие:

- Диалог должен иметь свойства: `style=WS_CHILD`; `border=WS_NONE`;
- Функция обработки сообщений диалога должна: по сообщению `WM_INITDIALOG` загружать в диалог все настройки(если имеются), по сообщению `WM_DESTROY` - сохранять все настройки(если имеются);

- Диалог должен создаваться по вызову функции `Configure(HWND hwndParent)` класса устройства, причем аргумент `hwndParent` этой функции содержит дескриптор родительского окна.

Пример функции `Configure`:

```
HWND __stdcall CDevice1::Configure(HWND hwndParent)
{
  //hModuleInstance - дескриптор модуля(DLL). Следует сохранять дескриптор модуля,
например, в функции DLLMain.
  hwndDlgConfig=CreateDialog(hModuleInstance,MAKEINTRESOURCE(idDlgConfig),hwndParent,(
DLGPROC)ConfigDialogProc);
  return hwndDlgConfig;
};
```

10.5. Шаг устройства

Каждый шаг симулятора MultiCore вызывается функция класса внешнего устройства `Step()`. По вызову этой функции устройство может осуществлять какие-либо действия. Также, если длительность одного такта устройства в N раз больше длительности такта MultiCore, то по вызову `Step()` можно увеличивать счетчик тактов, а по достижении счетчиком числа N вызывать функцию, исполняющую необходимые операции устройства. Например:

```
...

int Counter=0;
int DeviceCounter=0;

...

VOID __stdcall CDevice1::Step()     //шаг MultiCore
{
  Counter++;
  if (Counter==100)
  {
     Counter=0;
     Operate();           //по достижении 100 шагов MultiCore выполнить шаг устройства
  };
};

...

VOID __stdcall CDevice1::Operate() //шаг устройства
{
   DeviceCounter=DeviceCounter+100;
};

...
```

10.6. Прием данных

Для приема данных от симулятора MultiCore существует функция класса IDevice::SendData(DWORD dwData). Аргумент dwData - принимаемые данные. В теле функции SendData следует осуществлять обработку принимаемых данных (вывод, сохранение и т.д.).

Пример работы функции SendData:

```
...

int Counter=0;
DWORD Values[10];

...

BOOL __stdcall CDevice1::SendData(DWORD dwData)
{
  Values[Counter]=dwData; //сохранение принимаемых данных в массиве
  if (Counter==9)
  {
     Counter=0;
  }
  else
  {
     Counter++;
  };
  return TRUE;
};

...
```

Не следует вызывать SendData самостоятельно - эта функция вызывается портом по поступлении данных.

10.7. Передача данных

Если необходимо передать данные симулятору MultiCore, следует вызвать функцию SendData(DWORD dwData) абстрактного класса IPort. Для этого нужно воспользоваться указателем на подсоединенный порт. Пример:

```
...

VOID __stdcall CDevice1::Operate()
{
  IPort* tempPort;
  tempPort=GetConnectedPort();
  DWORD dwData;
  BOOL  WasSent;
  dwData=100;
  WasSent=tempPort->SendData(dwData);
};

...
```

Следует учесть, что для правильной работы функции `GetConnectedPort` следует сохранять указатель на подсоединенный порт по вызову функции `ConnectPort(IPort* lpPort)`.

10.8. Экспортируемые функции

В завершение разработки модели внешнего устройства, библиотека DLL должна описать и экспортировать три функции:

- `DWORD __stdcall GetDeviceCount()` - возвращает количество устройств, описанных в этой DLL.
- `BOOL __stdcall GetDeviceName(LPSTR szBuf, DWORD iIndex)` - заполняет `szBuf` именем устройства под номером `iIndex` (начиная с нулевого номера). Следует помнить, что значение `szBuf`, возвращаемое этой функцией станет аргументом для функции `CreateDevice`;
- `LPVOID __stdcall CreateDevice(LPSTR szDeviceName)` - создает устройство с именем `szDevName` и возвращает указатель на это устройство.

Пример:

```
//файл Device1.cpp
#include "Device1.h"
...

DWORD __stdcall GetDeviceCount()
{
  return 1;   //описано всего одно устройство
};

BOOL __stdcall GetDeviceName(LPSTR szBuf, DWORD iIndex)
{
  if (iIndex==0)
  {
     strcpy(szBuf,"Device1");   //если iIndex=0, то szBuf заполняется именем
устройства
     return TRUE;          //и возвращается TRUE
  }
  else
  {
     return FALSE;         //иначе возвращается FALSE
  };
};

LPVOID __stdcall CreateDevice(LPSTR szDeviceName)
{
  if (szDeviceName)
  {
     return new CDevice1();     //если szDeviceName - ненулевая строка, возвращается
указатель на вновь созданное
                    //устройство
```

```
  }
  else
     return 0;              //иначе возвращается нуль
};

//файл exports.def

EXPORTS
  GetDeviceCount
  GetDeviceName
  CreateDevice
```

Теперь библиотека DLL готова и может быть подключена к симулятору процессора MultiCore.

В конце книги рассматривается иллюстрированное описание создания простого внешнего устройства. См. главу "Пример создания устройства".

11. Примеры создания проектов

В данной главе рассмотрены примеры создания трех проектов:
- проекта для ядра RISC;
- проекта для ядер RISC и DSP;
- проекта с оверлейными структурами.

11.1. Проект для ядра RISC

В этом разделе рассматривается создание простого проекта (*RISCprj*) для работы только с ядром RISC.

Пример включает в себя:
- модуль RISC с одним файлом на языке Ассемблера для RISC (см. "Состав проекта");
- описание программы на языке Ассемблера для RISC (см. "Программа");
- описание настройки размещения проекта в памяти (см. "Настройки проекта").

Проект *RISCprj* находится в директории *\MCStudio\Samples\OnlyRisc*.

11.1.1. Состав проекта

Проект *RISCprj* включает в себя один RISC-модуль с именем **RISC**. В модуле содержится один файл *test.s* с кодом программы, написанной на языке Ассемблера для RISC. Файл заголовка *memory_12_asm.h* описывает адресное пространство MultiCore-12. На рисунке 10.1 изображено окно проекта *RISCprj*.

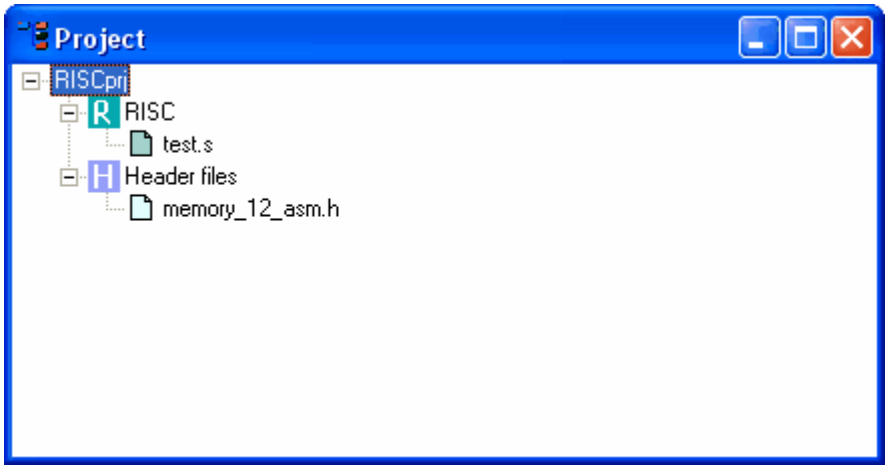


Рисунок 10.1. Окно проекта RISCprj.

Программа, содержащаяся в файле *test.s*, осуществляет вывод в порт UART. Она описана на странице "Программа".

11.1.2. Программа

В файле *test.s* проекта *RISCprj* содержится следующая программа:

```
//файл test.s
 .include "memory_12_asm.h"
 .set reorder
 .text
 .ent main

    main:
    lui $30,CPU_BASE
    li $4,1
    move $9,$0

    li $2,0x80
    sw $2,LCR($30)
    li $2,1
    sw $2,DLL($30)
    sw $0,LCR($30)

    li $3,10
    li $8,0x60
    la $4,output
    send_data:
         lb $5,($4)
         sb $5,THR($30)
         waiting:
                 lb $6,LSR($30)
                 andi $6,0x60
         bne $6,$8,waiting

         addi $4,4
         addi $3,-1
    bne $0,$3,send_data

    loop:
    nop
    nop
    nop
    j loop

 .end main

 .data
output:
 .word 0,1,2,3,4,5,6,7,8,9
```

Первая инструкция программы помещает адрес, соответствующий символу `CPU_BASE` в регистр `$30`. Символ `CPU_BASE`, заданный в файле *memory_12_asm.h*, определяет базовый адрес для описания регистров RISC-ядра. В файле *memory_12_asm.h* адреса всех регистров RISC определяются как смещения от базового адреса `CPU_BASE`.

Таким образом, например, запись `LCR($30)` означает смещение, определенное символом `LCR`, от базового адреса, находящегося в регистре `$30` (`CPU_BASE`). Эта запись соответствует обращению к регистру **LCR**.

Программа инициализирует порт UART, а затем последовательно передает в буфер выхода этого порта десять цифр, от нуля до девяти. Секция текста программы в данном примере начинается директивой **`.text`** и заканчивается директивой **`.end`** `main`. Затем следует секция данных, начинающаяся с директивы **`.data`**. Директива **`.ent`** указывает точку входа программы.

Размещение секций проекта *RISCprj* описано на странице "Настройки проекта".

Подробнее о директивах RISC-Ассемблера см. книгу "**RISC Tools. User's Guide**".
Подробнее об инструкциях Ассемблера для RISC см. книгу "**RISC Instructions Set**".

11.1.3. Настройки проекта

Диалог настроек проекта (рисунок 10.2) отображает размещение секций текста и данных каждого модуля проекта. В этом примере, проект содержит только один RISC-модуль с именем **RISC**.

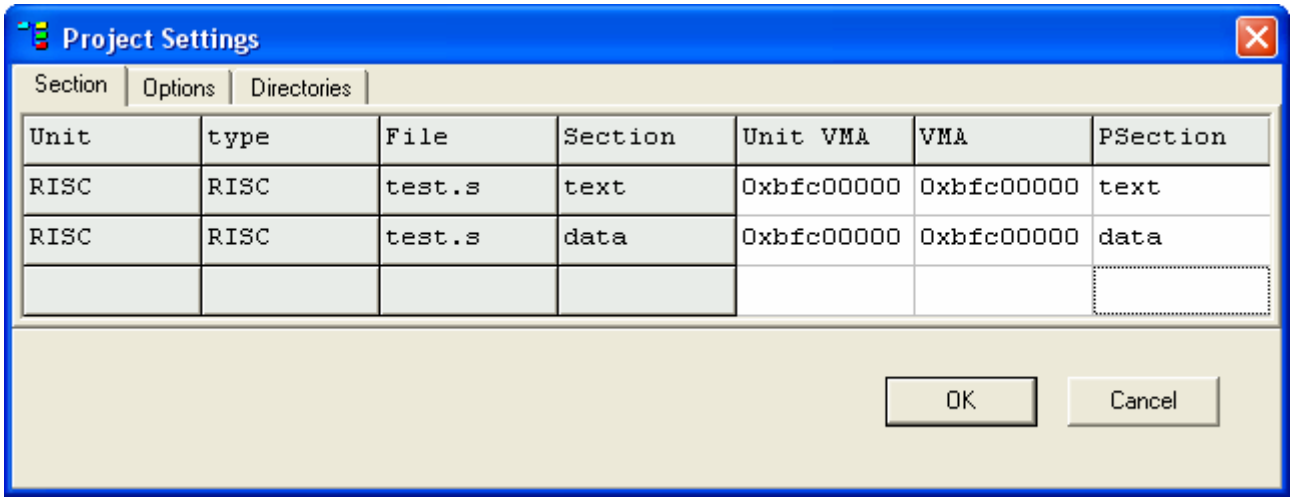


Рисунок 10.2. Диалог настроек проекта RISCprj.

В данном случае, секции текста и данных модуля **RISC** файла *test.s* размещаются в памяти RISC-ядра последовательно, начиная с адреса `0xBFC00000`. Размер системной области (**System Hall**) для данного проекта равен нулю. Поэтому, фактический адрес размещения секций будет равен указанному (`0xBFC00000`). В режиме отладки проекта размещение этих секций можно наблюдать в окне памяти (рисунок 10.3):

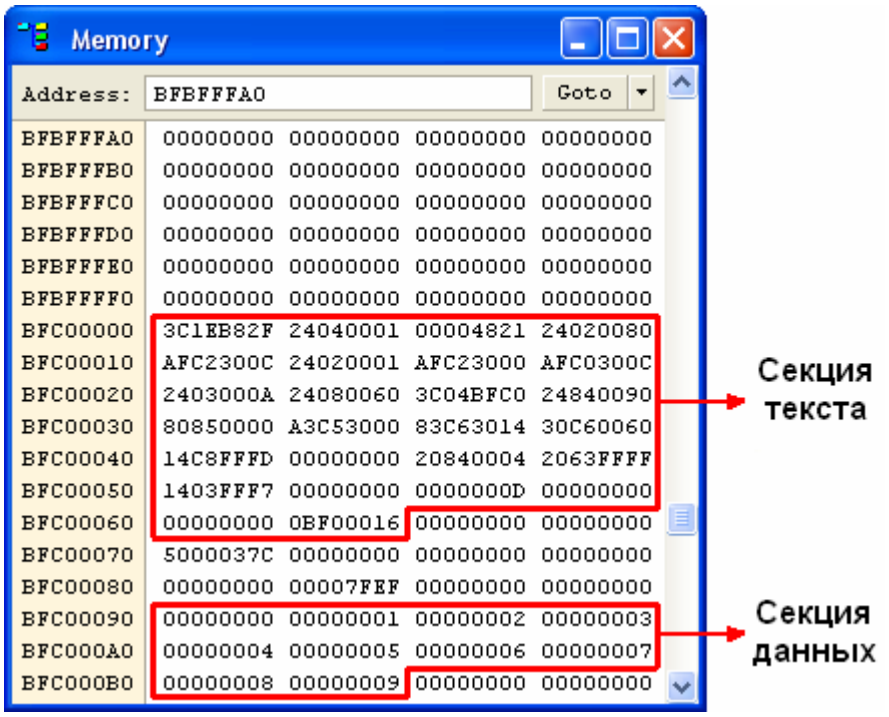


Рисунок 10.3. Размещение секций проекта RISCprj.

11.2. Проект для ядер RISC и DSP

Данный пример иллюстрирует создание проекта *RiscDspProject* для работы с ядрами RISC и DSP процессора MultiCore. Пример включает в себя:

- состав проекта;
- описание программы для RISC;
- описание программы для DSP;
- описание настроек размещения проекта в памяти.

Проект *RiscDspProject* находится в директории *\MCStudio\Samples\RiscDsp*.

11.2.1. Состав проекта

Проект *RiscDspProject* состоит из одного модуля RISC с именем **RISC** и одного модуля DSP с именем **DSP**. Файл *main.c* модуля **RISC** содержит программу для RISC-ядра, написанную на языке С. Файл *calc.s* модуля **DSP** содержит программу для DSP-ядра, написанную на языке DSP-Ассемблера. Файл заголовка *memory_12.h* описывает адресное пространство процессора MultiCore-12. На рисунке 10.4 изображено окно проекта *RiscDspProject*:

Рисунок 10.4. Окно проекта RiscDspProject.

11.2.2. Программа для RISC

Программа для RISC-ядра, содержащаяся в файле *main.c* в данном примере выполняет следующие действия:

- инициализация регистров управления DSP;
- передача двух чисел в память DSP-ядра;
- запуск программы DSP на исполнение;
- прием результата работы программы DSP из памяти.

Файл *main.c* выглядит так:

```
#include "memory_12.h"

extern int Start_DSP;
extern int InA;
extern int InB;
extern int OutC;

void exit();

main()
{
      int InputA=5;
      int InputB=2;
      int OutputC;

      DCSR   = 0;
      SR     = 0;
      SAR    = 0xFFFF;
      PC=((unsigned int)&Start_DSP - (unsigned int)&PRAM)>>2;
      A0=((unsigned int)&InA - (unsigned int)&XRAM)>>2;
      A1=((unsigned int)&InB - (unsigned int)&XRAM)>>2;
      A2=((unsigned int)&OutC - (unsigned int)&XRAM)>>2;

      InA=InputA;
```

```
        InB=InputB;

        DCSR = 0x4000;
        while((~(QSTR))&(1<<31));

        OutputC=OutC;
        exit();
};

void exit()
{
     while (1);
};
```

Переменные `Start_DSP`, `InA`, `InB` и `OutC` объявлены как внешние. Они описаны в программе DSP.

В функции `main()` объявляются переменные `InputA`, `InputB` и `OutputC`, предназначенные для приема/передачи данных ядру DSP.

Далее идет сброс регистров **DCSR** и **SR** и установка регистра останова **SAR** равным `0xFFFF`.

Счетчик команд **PC** устанавливается равным `&Start_DSP - &PRAM` (адрес переменной `Start_DSP` в памяти RISC минус адрес начала программной памяти DSP).

Адресные регистры **A0**, **A1** и **A2** устанавливаются соответственно равными `&InA - &XRAM`, `&InB - &XRAM` и `&OutC - &XRAM` (адрес каждой переменной в памяти RISC минус адрес начала памяти данных DSP).

После этого регистр **DCSR** устанавливается равным `0x4000`, это запускает DSP-программу на исполнение.

Ожидание завершения программы DSP осуществляется строкой `while((~(QSTR))&(1<<31))`.

Затем переменной `OutputC` присваивается результат работы DSP-ядра.

Символы `DCSR`, `QSTR`, `SR`, `SAR`, `PC`, `A0`, `A1`, `A2`, `PRAM` и `XRAM` описаны в заголовочном файле *memory_12.h*. Для того чтобы использовать их, нужно подключить *memory_12.h* директивой `#include`.

11.2.3. Программа для DSP

Программа для DSP-ядра, содержащаяся в файле *calc.s*, осуществляет следующие действия:

- запись двух чисел из памяти данных DSP в регистры общего назначения;
- вычисление суммы этих чисел;
- запись результата в память данных DSP.

Файл calc.s выглядит так:

```
.text
     .global Start_DSP
     .global InA
     .global InB
     .global OutC
```

```
Start_DSP:
          MOVE     (A0),R0
          MOVE     (A1),R2
          ADDL     R2,R0
          MOVE     R0,(A2)
          STOP

 .data
InA:  .dl 0
InB:  .dl 0
OutC: .dl 0
 .end
```

Секция текста модуля **DSP** начинается с директивы **.text** и заканчивается директивой **.data**, после которой следует секция данных.

Символ `Start_DSP` объявлен как глобальный. Он используется в программе RISC для размещения в счетчике команд **PC** адреса первой инструкции программы DSP.

Глобальные символы `InA`, `InB` и `OutC`, описанные в секции данных, позволяют программе RISC получить адреса размещения входных значений и результата в памяти данных DSP.

Далее следует, собственно, программа. Программа считывает из памяти по адресам, содержащимся в адресных регистрах **A0** и **A1**, в регистры **R0** и **R2** входные данные. Адреса входных данных и результата размещаются в адресных регистрах **A0**, **A1** и **A2** во время исполнения программы RISC.

После чтения входных данных из памяти, происходит сложение считанных чисел.

Затем, результат помещается в память по адресу, указанному в адресном регистре **A2**, после чего инструкция **STOP** устанавливает 31-й бит регистра **QSTR** в единицу. Таким образом, RISC-ядро информируется о завершении программы DSP и может считать результат из памяти.

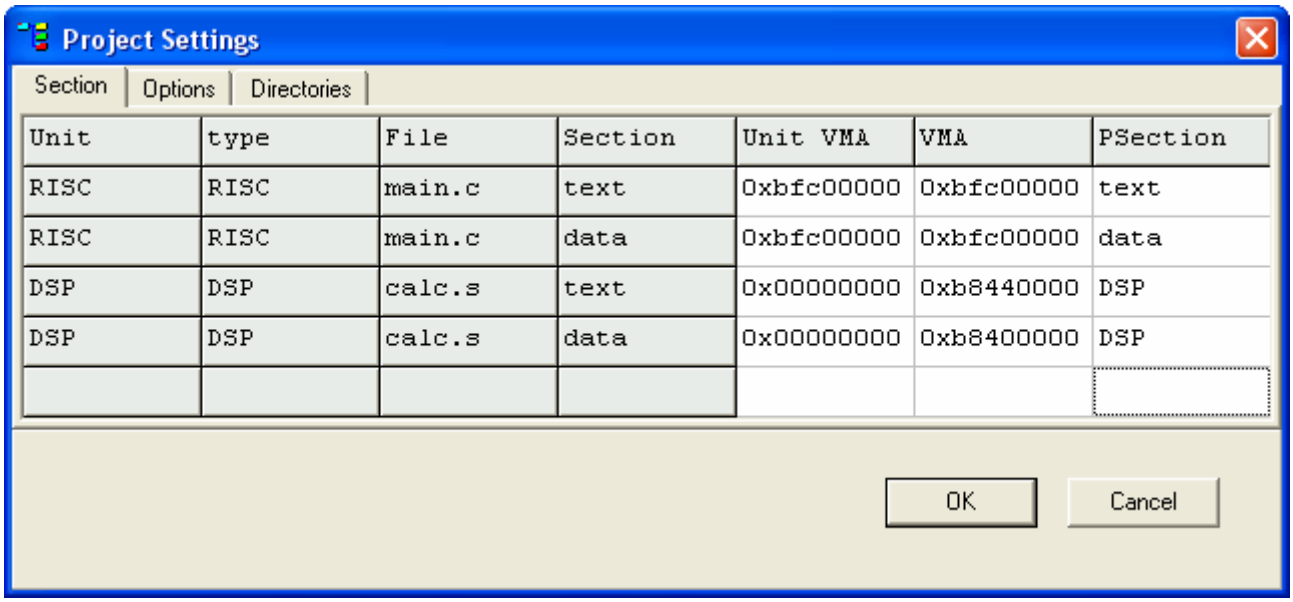
Размещение секций проекта *RiscDspProject* описано на странице "Настройки проекта".

Подробнее о директивах DSP-Ассемблера см. книгу "**Инструменты RISC**".

Подробнее об инструкциях Ассемблера для DSP см. книгу "**Инструкции DSP**".

11.2.4. Настройки проекта

Диалог настроек проекта (рисунок 10.5) *RiscDspProject* содержит информацию об адресах размещения секций текста и данных каждого модуля проекта. В данном случае, это модули **RISC** и **DSP**.



Unit	type	File	Section	Unit VMA	VMA	PSection
RISC	RISC	main.c	text	0xbfc00000	0xbfc00000	text
RISC	RISC	main.c	data	0xbfc00000	0xbfc00000	data
DSP	DSP	calc.s	text	0x00000000	0xb8440000	DSP
DSP	DSP	calc.s	data	0x00000000	0xb8400000	DSP

Рисунок 10.5. Диалог настроек проекта RiscDspProject.

Секции текста и данных модуля RISC размещаются по адресу `0xBFC00000`. Тем не менее, при компоновке секции RISC последовательно размещаются, начиная с адреса `0xBFC01000`. Это происходит потому, что адреса от `0xBFC00000` до `0xBFC01000` выделены под системную область (**System Hall**). Размер системной области можно переопределить в закладке **Options** диалога настроек проекта.

Секция текста модуля **DSP** размещена по адресу `0xB8440000`. Это виртуальный адрес начала программной памяти **PRAM** DSP. Секциям текста, размещенным при компоновке в программной памяти DSP, для исполнения достаточно указать в счетчике команд **PC** адрес первой исполняемой команды. Секциям текста, размещенным при компоновке вне **PRAM**, для исполнения необходима загрузка в программную память DSP. Адрес секции текста DSP внутри модуля (**Unit VMA**) равен `0x0000`. Это означает, что секция будет видима в **PRAM** с нулевого адреса.

Секция данных модуля **DSP** размещена по адресу `0xB8400000`. Это виртуальный адрес начала памяти данных DSP. Данные DSP, размещенные при компоновке в памяти данных, могут использоваться программой DSP сразу. Данные, размещенные вне памяти данных, перед использованием необходимо загрузить. Адрес секции данных DSP внутри модуля (**Unit VMA**) равен `0x0000`. Это означает, что секция будет видима в **XRAM** с нулевого адреса.

Подробнее о загрузке секций DSP память во время выполнения программы см. "Проект с оверлейными структурами".

На рисунках 10.6 и 10.7 приведено размещение секций текста и данных DSP в памяти:

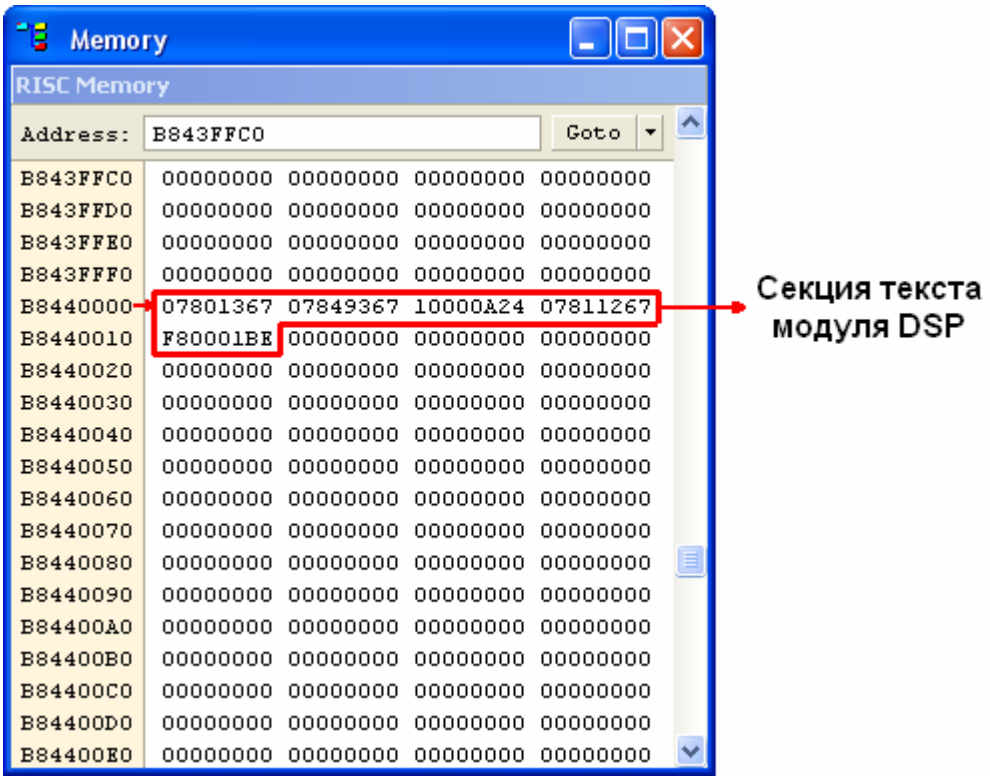


Рисунок 10.6. Размещение секции текста модуля DSP.

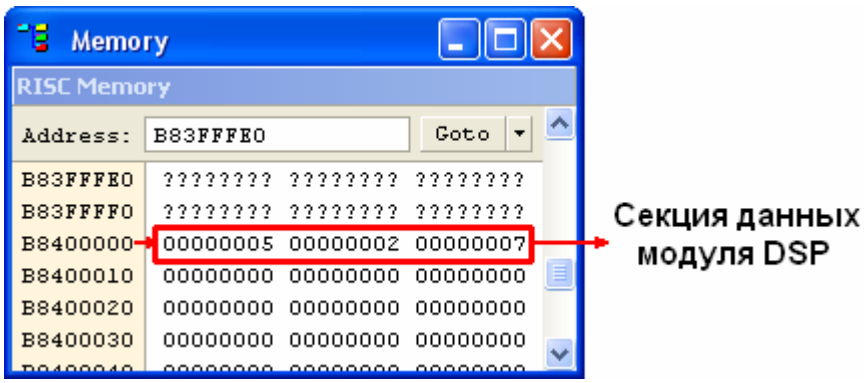
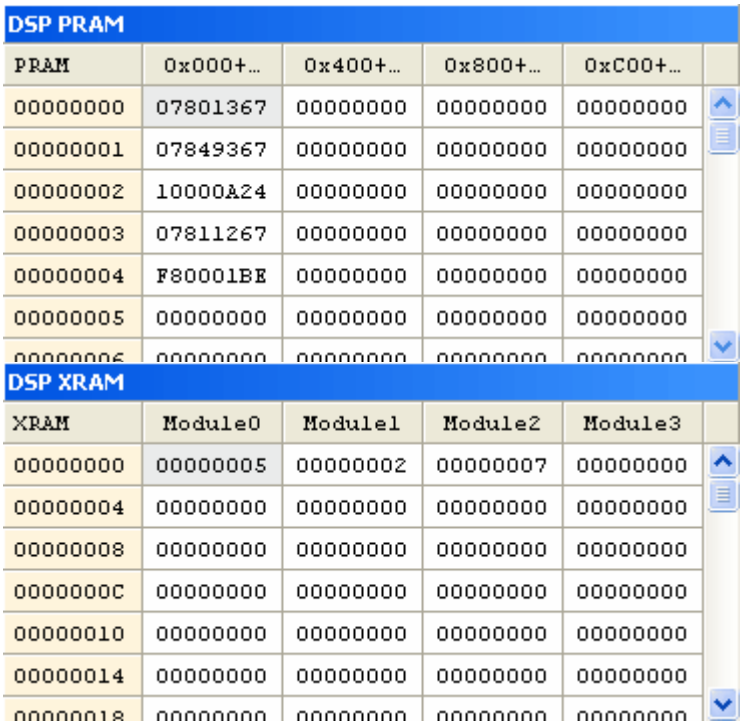


Рисунок 10.7. Размещение секции данных модуля DSP.

При входе в режим отладки программы видно, что секции текста и данных модуля **DSP** уже размещены в **PRAM** и **XRAM**, как показано на рисунке 10.8:



DSP PRAM

PRAM	0x000+…	0x400+…	0x800+…	0xC00+…
00000000	07801367	00000000	00000000	00000000
00000001	07849367	00000000	00000000	00000000
00000002	10000A24	00000000	00000000	00000000
00000003	07811267	00000000	00000000	00000000
00000004	F80001BE	00000000	00000000	00000000
00000005	00000000	00000000	00000000	00000000

DSP XRAM

XRAM	Module0	Module1	Module2	Module3
00000000	00000005	00000002	00000007	00000000
00000004	00000000	00000000	00000000	00000000
00000008	00000000	00000000	00000000	00000000
0000000C	00000000	00000000	00000000	00000000
00000010	00000000	00000000	00000000	00000000
00000014	00000000	00000000	00000000	00000000

Рисунок 10.8. Содержимое PRAM и XRAM в проекте RiscDspProject.

Примечание: число `0x00000007` размещается в памяти *в результате* выполнения программы DSP.

11.3. Проект с оверлейными структурами

Данный пример иллюстрирует создание проекта *Overlay* с оверлейными структурами. Принципы написания подобных проектов изложены в главе "Проекты с оверлейными структурами".

Пример включает в себя:

- состав проекта;
- описание программы для RISC-ядра;
- описание порядка исполнения проекта;
- описание процесса загрузки секций;
- описание настроек размещения проекта в памяти.

Проект *Overlay* находится в директории *\MCStudio\Samples\Overlay*.

11.3.1. Состав проекта

Проект *Overlay* состоит из одного RISC-модуля с именем **RISC** и одного DSP-модуля с именем **DSP**. Модуль **RISC** содержит файл *main.c*, в котором описана программа для RISC-ядра, а также файл *OVR.c*, описывающий таблицу загруженных секций и логику работы с ней.

Модуль **DSP** состоит из семи файлов, каждый из которых содержит оверлейные секции текста и данных. Иерархические деревья секций текста и данных изображены на рисунке 10.9:

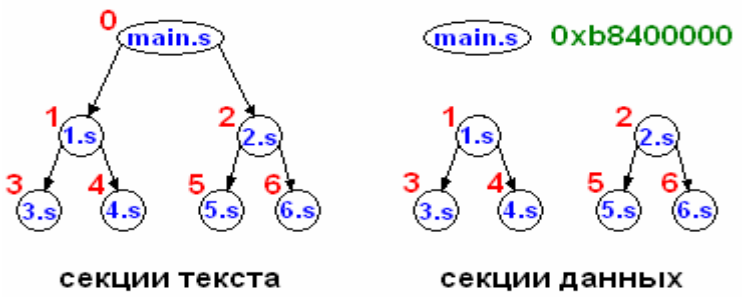


Рисунок 10.9. Иерархия оверлейных секций.

Иерархия секций отражает порядок их загрузки и исполнения. Синим цветом выделены имена файлов, содержащих оверлейные секции. Красные цифры слева от каждого узла иерархии - условные обозначения секций. Далее в данной главе для обозначения загружаемых секций будут использованы именно эти цифры. Зеленым цветом выделен адрес секции данных файла *main.s*. Эта секция загружается в **XRAM** ядра DSP на этапе компоновки.

Состав проекта *Overlay* отображен в окне проекта (рисунок 10.10):

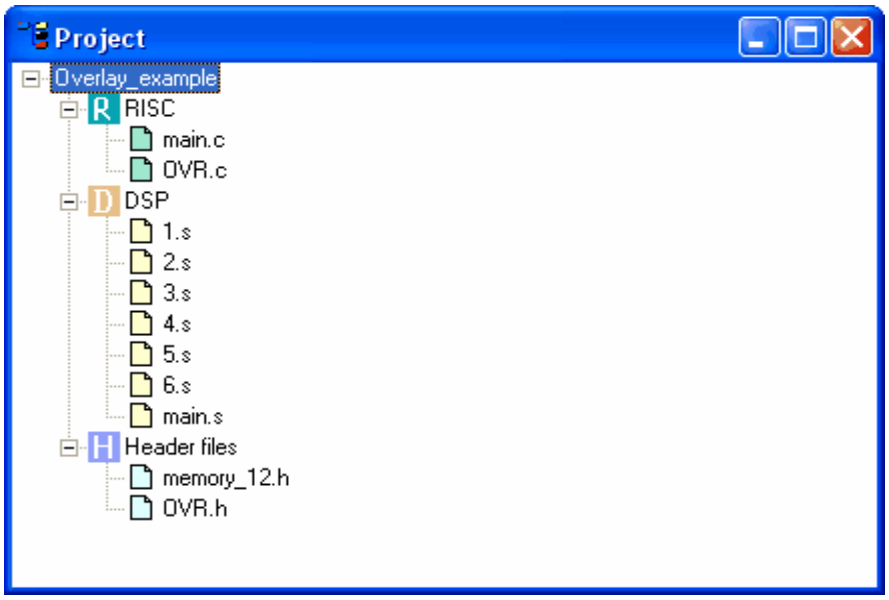


Рисунок 10.10. Состав проекта Overlay.

Заголовочный файл *memory_12.h* описывает адресное пространство MultiCore. Заголовочный файл *OVR.h* описывает структуру таблицы загруженных секций.

Программа для RISC-ядра описана на странице "Программа для RISC".

Порядок загрузки и исполнения секций DSP проекта *Overlay* описан на странице "Порядок исполнения".

Способ загрузки каждой оверлейной секции описан на странице "Загрузка секций".

Проект *Overlay* находится в директории *\MCStudio\Samples\Overlay*.

11.3.2. Программа для RISC

Программа для RISC проекта *Overlay* размещена в файле *main.c* модуля **RISC**:

```
#include "memory_12.h"
#include "OVR.h"

extern int Start_DSP;
extern int End_0;
extern int Start_1;
extern int End_1;
extern int Start_2;
extern int End_2;
extern int Start_3;
extern int End_3;
extern int Start_4;
extern int End_4;
extern int Start_5;
extern int End_5;
extern int Start_6;
extern int End_6;
extern int BranchNeeded;
extern int SectionNumber_1;
extern int SectionNumber_2;
extern int SectionNumber_3;
extern int SectionNumber_4;
extern int SectionNumber_5;
extern int SectionNumber_6;

void exit();
void LoadBranch(int BranchIndex);

int main()
{
     int Index;

     SR=0;
     DCSR=0;
     SAR=0xFFFF;
     int A;
     int B;
     OVRLoad((unsigned)&Start_DSP,(unsigned)0xb8440000,(&End_0-&Start_DSP));
     DCSR=0x4000;
     while ((~(QSTR))&(1<<31));
     Index=BranchNeeded;
      do
```

```
        {
            LoadBranch(Index);
            DCSR=0x4000;
            while ((~(QSTR))&(1<<31));
            Index=BranchNeeded;
        }
        while (Index!=5);

        exit();
        return 0;
};

void exit()
{
  while (1);
};

void LoadBranch(int BranchIndex)
{
      switch (BranchIndex)
      {
       case   1 : OVRLoad((unsigned)&Start_1,(unsigned)0xb8440100,(&End_1-&Start_1));
                  OVRLoad((unsigned)&SectionNumber_1,(unsigned)0xb8400080,1);
                  OVRLoad((unsigned)&Start_3,(unsigned)0xb8440200,(&End_3-&Start_3));
                  OVRLoad((unsigned)&SectionNumber_3,(unsigned)0xb8400100,1);
                  break;

       case   2 : OVRLoad((unsigned)&Start_4,(unsigned)0xb8440200,(&End_4-&Start_4));
                  OVRLoad((unsigned)&SectionNumber_4,(unsigned)0xb8400100,1);
                  break;

       case   3 : OVRLoad((unsigned)&Start_2,(unsigned)0xb8440100,(&End_2-&Start_2));
                  OVRLoad((unsigned)&SectionNumber_2,(unsigned)0xb8400080,1);
                  OVRLoad((unsigned)&Start_5,(unsigned)0xb8440200,(&End_5-&Start_5));
                  OVRLoad((unsigned)&SectionNumber_5,(unsigned)0xb8400100,1);
                  break;

       case   4 : OVRLoad((unsigned)&Start_6,(unsigned)0xb8440200,(&End_6-&Start_6));
                  OVRLoad((unsigned)&SectionNumber_6,(unsigned)0xb8400100,1);
                  break;
      };
};
```

Заголовочный файл *OVR.h* описывает таблицу загруженных секций, необходимую для отладчика.

Программа объявляет 21 внешний символ. Первые 14 символов - метки начала и конца секций кода, последние 7 - метки начала секций данных. Эти символы используются для получения адреса размещения секций в памяти RISC и вычисления размера секций. Символ `BranchNeeded` необходим для получения номера следующей ветви.

После объявления внешних символов программа инициализирует регистры DSP-ядра. Символы `SR`, `DCSR`, `QSTR` и `SAR`, описаны в файле *memory_12.h*. Для использования

этих символов необходимо подключить заголовочный файл *memory_12.h* директивой `#include`.

Затем, функция `OVRLoad` загружает в память DSP секцию 0 программы DSP. Строка `DCSR=0x4000` запускает программу DSP на исполнение. Строка `while((~(QSTR))&(1<<31))` указывает RISC-ядру ожидать команды **STOP** ядра DSP. После остановки DSP-ядра, в переменную `Index` считывается номер необходимой для загрузки ветви программы `BranchNeeded` (этот номер формируется при исполнении секции **0**).

После получения номера первой необходимой ветви исполняется цикл. Каждую итерацию цикла в память DSP загружается очередная ветвь программы. Номер нужной ветви формируется в ядре DSP. Затем загруженная ветвь запускается на исполнение, а ядро RISC ожидает команды **STOP**. Цикл исполняется до тех пор, пока номер ветви не станет равным **5** (несуществующая ветвь). Загрузку нужной ветви осуществляет функция `LoadBranch`.

Функция `OVRLoad` описана на странице "Таблица загруженных секций".

Полный текст всех DSP-файлов, а также порядок загрузки и исполнения секций приведены на странице "Порядок исполнения".

Функция `LoadBranch` и цикл загрузки ветвей описаны на странице "Загрузка секций".

Размещение секций проекта в памяти рассматривается на странице "Настройки проекта".

Проект *Overlay* находится в директории *\MCStudio\Samples\Overlay*.

11.3.3. Порядок исполнения

Здесь рассматривается содержимое всех файлов DSP и порядок загрузки секций в память DSP-ядра. Порядок загрузки и исполнения секций текста определяется из иерархического дерева (см. страницу "Состав проекта"): сначала загружается и исполняется ветвь **0-1-3**, затем **0-1-4**, потом **0-2-5** и **0-2-6**. Пронумеруем эти ветви как **1**, **2**, **3** и **4**. Загрузка каждой секции кода будет сопровождаться загрузкой секции данных соответствующего номера. То есть, при загрузке ветви **1**, будут исполняться секции кода **0**, **1**, **3** и будут загружены секции данных **1** и **3**. Секция данных **0** загружаться не будет, так как она размещается в **XRAM** DSP-ядра на этапе компоновки.

Первым в память DSP-ядра загружается секция текста **0** (файл *main.s*):

```
  .text
       .global Start_DSP
       .global Next_Branch
       .global End_0
       .global BranchNeeded

Start_DSP:
            MOVE BranchNeeded,A0
            MOVE SectionPassed,A1

Next_Branch:
```

```
            CLRL R2
            MOVE R2,(A1)+
            MOVE (A0),R0
            INC  R0,R0
            MOVE R0,(A0)
            STOP

            CMP  1,R0
            J.eq Start_1
            CMP  2,R0
            J.eq Start_1
            CMP  3,R0
            J.eq Start_2
            CMP  4,R0
            J.eq Start_2

            STOP
End_0:
        nop

  .data
BranchNeeded: .dl 0
SectionPassed: .skip 48

  .end
```

В файле main.s объявлены глобальные символы `Start_DSP`, `Next_Branch`, `End_0` и `BranchNeeded`. Символы `Start_DSP` и `End_0` необходимы для получения адреса секции в памяти RISC и для вычисления размера секции. Символ `Next_Branch` используется другими секциями для перехода к вычислению номера следующей ветви, как показано далее. Символ `BranchNeeded` используется RISC-ядром для получения от ядра DSP номера ветви.

Символы `BranchNeeded` и `SectionPassed` объявлены в секции данных **0**. Эта секция при компоновке загружается непосредственно в **XRAM** и, следовательно, не требует динамической загрузки. В области памяти, выделенной по метке `SectionPassed`, исполняемые секции будут размещать свои номера.

Первые две строки кода размещают в адресных регистрах **A0** и **A1** адреса памяти **XRAM**, соответствующие меткам `BranchNeeded` и `SectionPassed`.

Затем, по метке `Next_Branch`, номер исполняемой секции (**0**) помещается по адресу **A1**, при этом содержимое регистра **A1** инкрементируется. После этого из **XRAM** (по адресу **A0**) в регистр **R0** помещается номер предыдущей выполненной ветви. К этому номеру прибавляется единица. Таким образом, в регистре **R0** сформирован номер следующей ветви. Содержимое регистра **R0** возвращается в **XRAM** по адресу **A0**, после чего программа останавливается (счетчик команд устанавливается на следующую за **`STOP`** строку кода). После остановки, программа ядра RISC берет из памяти значение `BranchNeeded`, загружает необходимую ветвь и снова запускает программу DSP на исполнение.

Далее, содержимое **R0** сравнивается поочередно с номерами ветвей, и переходит (команда **`J.`**`eq`) к необходимой ветви. Если же содержимое регистра **R0** не равно ни одному из номеров ветвей, программа останавливается.

Первой исполняется ветвь **1** (секции **0**-**1**-**3**). При выполнении этой ветви секция **0** переходит (**Jump**) к секции **1** (файл *1.s*):

```
  .text
       .global Start_1
       .global End_1
       .global SectionNumber_1

 Start_1:
            MOVE SectionNumber_1,A2
            MOVE (A2),R2
            MOVE R2,(A1)+
            CMP  1,R0
            J.eq Start_3
            J    Start_4
 End_1:
         nop

  .data
 SectionNumber_1: .dl 0x00000001
  .end
```

В файле *1.s* объявлены три глобальных символа - `Start_1`, `End_1` и `SectionNumber_1`. Символы `Start_1` и `End_1` необходимы для получения адреса секции в памяти RISC и вычисления размеров этой секции. Также символ `Start_1` используется для перехода из секции **0** в секцию **1**.

Символ `SectionNumber_1` используется программой RISC для загрузки секции данных **1**.

В секции кода **1** по метке `Start_1` адрес памяти **XRAM**, соответствующий символу `SectionNumber_1`, помещается в регистр **A2**. Затем, содержимое памяти по адресу **A2** помещается в регистр **R2**. После этого содержимое регистра **R2** размещается в **XRAM** по адресу **A1**, при этом содержимое регистра **A1** инкрементируется. Таким образом, секция **1** (как и другие секции кода) размещает в **XRAM** свой номер.

После размещения номера секции в памяти, значение регистра **R0** сравнивается с единицей. Регистр **R0** содержит номер исполняемой в данный момент ветви (номер формируется при исполнении секции **0**). Если номер ветви равен единице, программа переходит к секции **3**, иначе - к секции **4**. Переход осуществляется командой **Jump**.

Рассмотрим исполнение секции **3** (файл *3.s*):

```
  .text
       .global Start_3
       .global End_3
       .global SectionNumber_3

 Start_3:
            MOVE SectionNumber_3,A2
            MOVE (A2),R2
            MOVE R2,(A1)+
            J    Next_Branch
 End_3:
         nop
  .data
 SectionNumber_3: .dl 0x00000003
  .end
```

К секции **3** программа переходит (**Jump**) от секции **1**.

Здесь объявлены глобальные символы `Start_3`, `End_3` и `SectionNumber_3`. Они используются аналогично символам секции **1**. Секция кода **3** размещает в памяти свой номер (`0x00000003`) после чего осуществляет безусловный переход (**Jump**) по метке `Next_Branch`. Символ `Next_Branch` расположен в секции текста **0**. По этому символу, программа вычисляет номер следующей исполняемой ветви и передает управление ядру RISC. Таким образом, переход к символу `Next_Branch` являет собой окончание исполнения очередной ветви программы DSP.

За ветвью **1** исполняется ветвь **2** (**0**-**1**-**4**). Секции **0** и **1** уже загружены, следовательно ядро RISC загружает в память DSP только секцию **4** (поверх секции **3**). После этого, на исполнение запускается секция **0**, от нее программа переходит к секции **1**, а от секции один - к секции **4** (файл *4.s*):

```
  .text
       .global Start_4
       .global End_4
       .global SectionNumber_4

Start_4:
           MOVE SectionNumber_4,A2
           MOVE (A2),R2
           MOVE R2,(A1)+
           J    Next_Branch
End_4:
        nop

  .data
SectionNumber_4: .dl 0x00000004
  .end
```

Секция **4** полностью аналогична секции **3**.

После ветви **2** следует исполнение ветви **3** (**0**-**2**-**5**). Секция **0** уже загружена, поэтому ядро RISC загружает в память DSP только секции **2** и **5**. После загрузки, на исполнение запускается секция **0**, от нее программа переходит к секции **2**, а от секции **2** - к секции **5**.

Секция **2** (файл *2.s*) полностью аналогична секции **1**:

```
  .text
       .global Start_2
       .global End_2
       .global SectionNumber_2

Start_2:
           MOVE SectionNumber_2,A2
           MOVE (A2),R2
           MOVE R2,(A1)+
           CMP  3,R0
           J.eq Start_5
           J    Start_6
End_2:
        nop

  .data
SectionNumber_2: .dl 0x00000002

  .end
```

Здесь, как и в секции **1**, в конце программы следует переход к секции **5** или **6**, в зависимости от номера исполняемой ветви (содержимое регистра **R0**).

Из секции **2** программа переходит в секцию **5** (файл *5.s*):

```
  .text
       .global Start_5
       .global End_5
       .global SectionNumber_5

Start_5:
            MOVE SectionNumber_5,A2
            MOVE (A2),R2
            MOVE R2,(A1)+
            J    Next_Branch
End_5:
          nop

  .data
SectionNumber_5: .dl 0x00000005
  .end
```

Секция **5** полностью аналогична секции **3**.

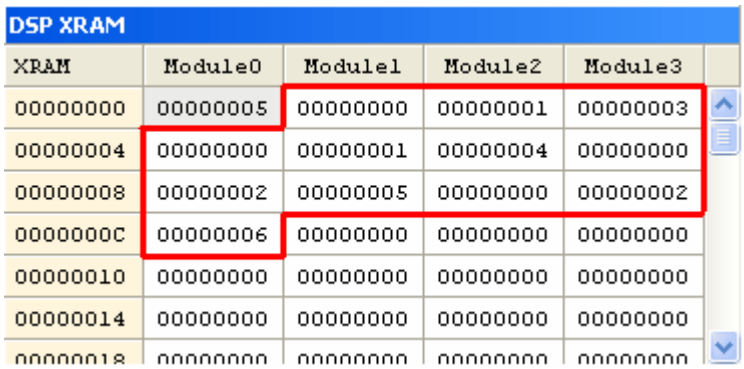
После исполнения ветви **3** следует исполнение ветви **4** (**0**-**2**-**6**). Так как секции **0** и **2** уже загружены, ядро RISC загружает только секцию **6**. После загрузки, на исполнение запускается секция **0**, затем программа переходит к секции **2**, а от секции **2** - к секции **6** (файл *6.s*):

```
  .text
        .global Start_6
        .global End_6
        .global SectionNumber_6

Start_6:
            MOVE SectionNumber_6,A2
            MOVE (A2),R2
            MOVE R2,(A1)+
            J    Next_Branch
End_6:
          nop

  .data
SectionNumber_6: .dl 0x00000006
  .end
```

Секция **6** полностью аналогична секции **3**.

После исполнения ветви **4**, номер следующей ветви устанавливается равным пяти. Это сигнализирует ядру RISC об окончании программы DSP.

В результате работы программы DSP в **XRAM** сформируются данные, приведенные на рисунке 10.11:



DSP XRAM

XRAM	Module0	Module1	Module2	Module3
00000000	00000005	00000000	00000001	00000003
00000004	00000000	00000001	00000004	00000000
00000008	00000002	00000005	00000000	00000002
0000000C	00000006	00000000	00000000	00000000
00000010	00000000	00000000	00000000	00000000
00000014	00000000	00000000	00000000	00000000
00000018	00000000	00000000	00000000	00000000

Рисунок 10.11. Содержимое XRAM в проекте Overlay.

Число `0x00000005` было сформировано для указания ядру RISC об окончании программы DSP. Красной линией выделен блок данных, заполненный номерами выполненных секций. Как видно из рисунка, секции выполнялись в порядке **0-1-3**, **0-1-4**, **0-2-5**, **0-2-6**, то есть в установленном нами порядке.

Следует помнить, что после метки конца каждой секции текста (здесь - `End_x`) обязательно должна следовать какая-либо операция. В данном примере, это операция **NOP**. Это необходимо потому, что секции размещаются в памяти RISC одна за другой, и если между меткой конца одной секции и меткой начала следующей не будет ни одной операции, эти метки будут иметь один адрес, и метка начала следующей секции будет потеряна, что приведет к некорректной работе симулятора.

Текст программы RISC-ядра приведен на странице "Программа для RISC".

Состав проекта и иерархическое дерево секций приведены на странице "Состав проекта".

Описание способа загрузки секций в память DSP приведено на странице "Загрузка секций".

Размещение секций проекта в памяти RISC приведено на странице "Настройки проекта".

Проект *Overlay* находится в директории *\MCStudio\Samples\Overlay*.

11.3.4. Загрузка секций

Загрузка секций текста и данных DSP происходит в программе RISC-ядра. Для этого используется функция `LoadBranch (int BranchIndex)`, загружающая в память DSP необходимую ветвь дерева секций:

```
void LoadBranch(int BranchIndex)
{
     switch (BranchIndex)
     {
      case   1 : OVRLoad((unsigned)&Start_1,(unsigned)0xb8440100,(&End_1-&Start_1));
                 OVRLoad((unsigned)&SectionNumber_1,(unsigned)0xb8400080,1);
```

```
                OVRLoad((unsigned)&Start_3,(unsigned)0xb8440200,(&End_3-&Start_3));
                OVRLoad((unsigned)&SectionNumber_3,(unsigned)0xb8400100,1);
                break;

        case    2 : OVRLoad((unsigned)&Start_4,(unsigned)0xb8440200,(&End_4-&Start_4));
                OVRLoad((unsigned)&SectionNumber_4,(unsigned)0xb8400100,1);
                break;

        case    3 : OVRLoad((unsigned)&Start_2,(unsigned)0xb8440100,(&End_2-&Start_2));
                OVRLoad((unsigned)&SectionNumber_2,(unsigned)0xb8400080,1);
                OVRLoad((unsigned)&Start_5,(unsigned)0xb8440200,(&End_5-&Start_5));
                OVRLoad((unsigned)&SectionNumber_5,(unsigned)0xb8400100,1);
                break;

        case    4 : OVRLoad((unsigned)&Start_6,(unsigned)0xb8440200,(&End_6-&Start_6));
                OVRLoad((unsigned)&SectionNumber_6,(unsigned)0xb8400100,1);
                break;
      };
  };
```

Функция в качестве входного параметра принимает номер загружаемой ветви. Далее, в зависимости от номера, загружаются те или иные секции кода и данных.

Для загрузки каждой секции используется функция **int** OVRLoad (**unsigned** SRC, **unsigned** DST, **unsigned** wSIZE). В качестве адреса размещения секции в памяти RISC (SRC) функции передается адрес символа Start_X для секций кода и SectionNumber_X для секций данных (X - номер загружаемой секции). В качестве адреса назначения (DST) передается адрес, попадающий в область памяти DSP. Следует помнить, что адреса загрузки секций в память DSP должны быть выбраны в соответствии с указанными в диалоге настроек проекта. При указании иных адресов, глобальные символы программы не будут доступны внутри DSP.

Подробнее о размещении секций проекта *Overlay* в памяти см. страницу "Настройка проекта".

Параметр wSIZE (размер загружаемой секции) для секций кода вычисляется из меток начала и конца секции:
wSIZE=End_X - Start_X. Обратите внимание, что при этом способе вычисления размера секций, операция **NOP**, расположенная после метки End_X, загружаться не будет.

Для секций данных размер указан явно (1 слово).

Функция OVRLoad кроме загрузки секций также заполняет таблицу загруженных секций. Поэтому, для корректной работы отладчика, загружать секции нужно именно при помощи этой функции.

Описание функции OVRLoad приведено на странице "Таблица загруженных секций".

Настройки размещения секций в памяти описаны на странице "Настройки проекта".

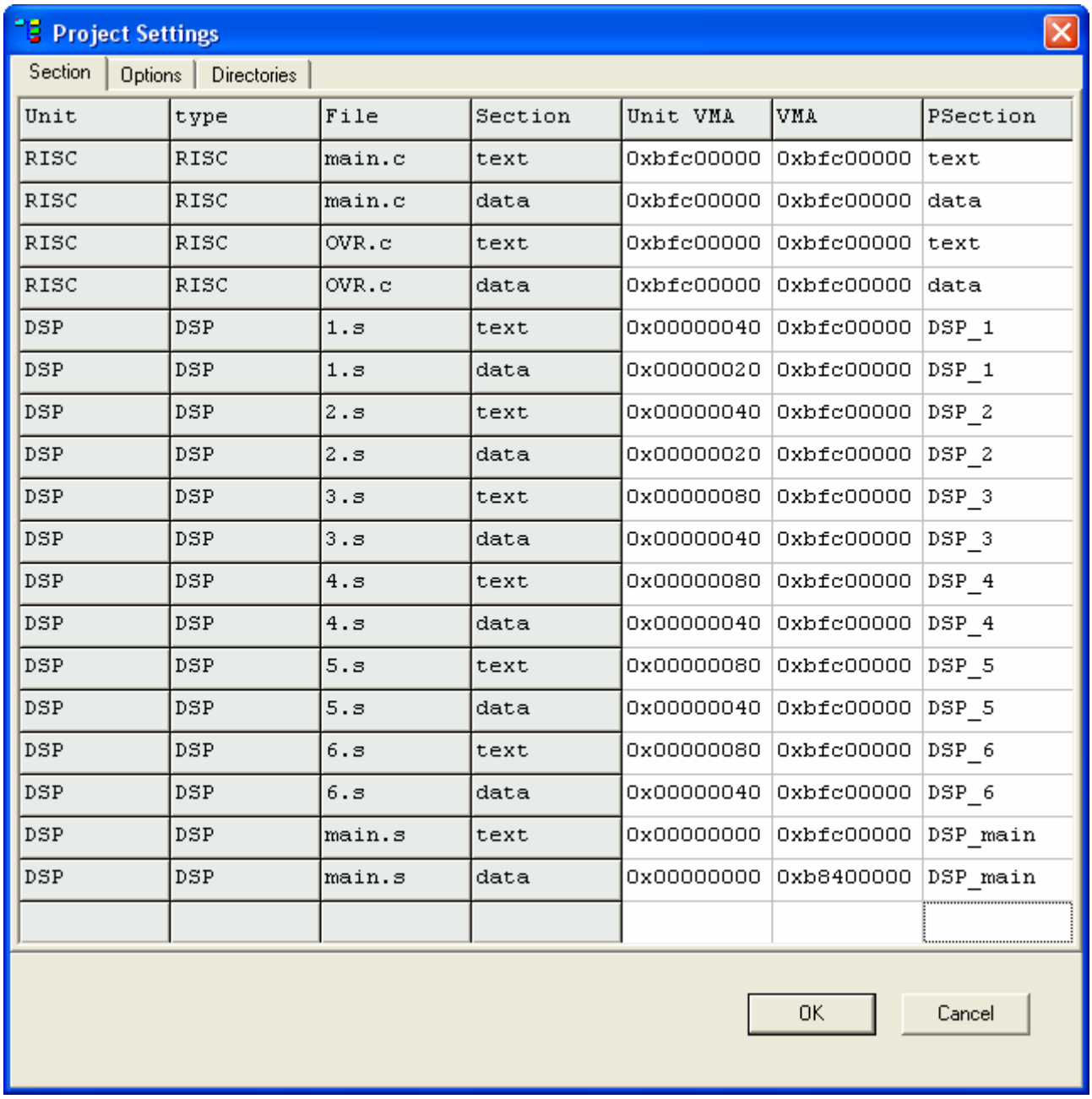
Порядок загрузки и исполнения секций DSP описан на странице "Порядок исполнения".

Иерархическое дерево секций приведено на странице "Состав проекта".

Проект *Overlay* находится в директории *\MCStudio\Samples\Overlay*.

11.3.5. Настройки проекта

Диалог настроек проекта *Overlay* (рисунок 10.12) содержит информацию о размещении в памяти секций текста и данных модулей RISC и DSP:



Unit	type	File	Section	Unit VMA	VMA	PSection
RISC	RISC	main.c	text	0xbfc00000	0xbfc00000	text
RISC	RISC	main.c	data	0xbfc00000	0xbfc00000	data
RISC	RISC	OVR.c	text	0xbfc00000	0xbfc00000	text
RISC	RISC	OVR.c	data	0xbfc00000	0xbfc00000	data
DSP	DSP	1.s	text	0x00000040	0xbfc00000	DSP_1
DSP	DSP	1.s	data	0x00000020	0xbfc00000	DSP_1
DSP	DSP	2.s	text	0x00000040	0xbfc00000	DSP_2
DSP	DSP	2.s	data	0x00000020	0xbfc00000	DSP_2
DSP	DSP	3.s	text	0x00000080	0xbfc00000	DSP_3
DSP	DSP	3.s	data	0x00000040	0xbfc00000	DSP_3
DSP	DSP	4.s	text	0x00000080	0xbfc00000	DSP_4
DSP	DSP	4.s	data	0x00000040	0xbfc00000	DSP_4
DSP	DSP	5.s	text	0x00000080	0xbfc00000	DSP_5
DSP	DSP	5.s	data	0x00000040	0xbfc00000	DSP_5
DSP	DSP	6.s	text	0x00000080	0xbfc00000	DSP_6
DSP	DSP	6.s	data	0x00000040	0xbfc00000	DSP_6
DSP	DSP	main.s	text	0x00000000	0xbfc00000	DSP_main
DSP	DSP	main.s	data	0x00000000	0xb8400000	DSP_main

Рисунок 10.12. Диалог настроект проекта Overlay.

Все секции одного типа, имеющие одинаковые имена, при компоновке будут собраны в одну секцию. Поэтому, если требуется сформировать в памяти отдельно несколько секций текста DSP (например, для их последующей динамической загрузки), их следует именовать по-разному. Поэтому, все секции DSP имеют разные имена, как показано на рисунке.

VMA-адреса размещения всех секций проекта (за исключением секции данных **0**) в памяти RISC равны `0xBFC00000`. Следовательно, все секции будут последовательно собраны в памяти RISC с адреса 0xBFC01000 (первая *1000* байт зарезервирована под системную область).

Для секций текста и данных модуля **`RISC Unit VMA`**=**`VMA`**, то есть секции RISC будут доступны по адресам, указанным в диалоге настройки.

Рассмотрим размещение в памяти секций модуля DSP:

Адреса секций текста **0**, **1**, **2**, **3**, **4**, **5** и **6** внутри модуля (**Unit VMA**) - `0x00000000`, `0x00000040`, `0x00000040`, `0x00000080`, `0x00000080`, `0x00000080` и `0x00000080`. Обратите внимание, что при загрузке секций в файле *main.c* используются не указанные здесь адреса, а `0xB8440000`, `0xB8440100`, `0xB8440100`, `0xB8440200`, `0xB8440200`, `0xB8440200` и `0xB8440200`. Это связано с тем, что в столбце **Unit VMA** для этих секций указывается предполагаемый адрес в **PRAM**, а **PRAM** имеет словную адресацию. Однако, в программе RISC для копирования секций используется байтовая адресация памяти RISC, поэтому все младшие части указанных там адресов равны адресам, указанным здесь, сдвинутым влево на два бита. Например, `0xB8440100=0xB8440000 + (0x00000040<<2)` и так далее. Старшая часть адреса (`B844`) - виртуальный адрес начала **PRAM**.

Секции текста, имеющие одинаковые адреса **Unit VMA**, являются оверлейными и при загрузке перекрывают друг друга.

Секции данных **1**, **2**, **3**, **4**, **5** и **6** имеют адреса внутри модуля (**Unit VMA**), соответственно равные `0x00000020`, `0x00000020`, `0x00000040`, `0x00000040`, `0x00000040` и `0x00000040`. Аналогично случаю с секциями текста, это предполагаемые адреса размещения в памяти данных DSP. В файле *main.c* для загрузки секций используются адреса со сдвинутой влево на два бита младшей частью. Старшая часть адреса, указанного для загрузки секций равна `B840`, то есть виртуальному адресу начала **XRAM**.

Секция данных **0** (файл *main.s*) загружается в память RISC по адресу **`VMA`**=`0xB8400000`. Это виртуальный адрес начала памяти данных DSP, при компоновке секция сразу попадет в **XRAM**. Адрес **`Unit VMA`**=`0x00000000`, то есть секция будет доступна в **XRAM** начиная с нулевого адреса.

Способ загрузки секций в память DSP рассмотрен на странице "Загрузка секций".

Проект *Overlay* находится в директории *\MCStudio\Samples\Overlay*.

Итак, при данной адресации секций в памяти, все глобальные символы текста и данных доступны внутри DSP и проект *Overlay* выполняется корректно.

12. Пример симулятора устройства

В данной главе рассматривается пример создания и подключения модели внешнего устройства. Пример включает в себя:

- Описание класса внешнего устройства CDeviceX;
- Диалог настроек устройства;
- Вывод принимаемых данных в консоль;
- Описание трех экспортируемых функций;
- Файл test.s с программой, написанной на языке Ассемблера для RISC, для передачи данных в порт UART.

12.1. CDeviceX

Результирующая библиотека *DeviceX.dll* содержит одно устройство: *DeviceX.Device1*. Рассмотрим класс, описывающий это устройство.

```
//файл DeviceX.h
#include "model.h"        //в файле model.h содержатся заголовки абстрактных классов
IDevice и IPort

class CDeviceX: public IDevice  //класс CDeviceX - наследный от абстрактного класса
IDevice
{
public:
  VOID __stdcall Release();
  CDeviceX();
  ~CDeviceX();
  LPCSTR __stdcall GetDeviceName();
  DWORD __stdcall GetDeviceCoreFileName (LPSTR szFilename, DWORD nSize);
  VOID __stdcall Step();
  HWND __stdcall Configure(HWND hwndParent);
  VOID __stdcall SetConfigurationFile(LPSTR szFileName);
  BOOL __stdcall SendData(DWORD dwData);
  BOOL __stdcall ConnectPort(IPort* lpPort);
  IPort* __stdcall GetConnectedPort();
  static LRESULT CALLBACK ConfigDialogProc(HWND hwndDlg, UINT uMsg, WPARAM wParam,
LPARAM lParam);

protected:
  IPort*   pConnectedPort;
  LPSTR    pDeviceName;
  INT      idDlgConfig;
  HWND     hwndDlgConfig;
};
```

Класс `CDeviceX` объявлен как наследный от класса IDevice и описывает все виртуальные функции этого абстрактного класса. Кроме этих функций, класс внешнего устройства содержит:

- конструктор и деструктор;
- функцию обработки сообщений диалогом настроек устройства;
- указатель на подсоединенный порт;
- указатель на строку с именем устройства;
- идентификационный номер и дескриптор диалога настроек устройства.

Рассмотрим описание функций класса `CDeviceX`.

```
//файл DeviceX.cpp

#include "stdafx.h"
#include "resource.h"
#include "DeviceX.h"

#pragma warning(disable:4786)

HINSTANCE  hModuleInstance;
HANDLE  hConsoleOutput;
COORD   coordCursorPos;

////////////////////////////////////////////////////////////////////////////
BOOL APIENTRY DLLMain ( HANDLE hModule,
                DWORD ul_reason_for_call,
             LPVOID lpReserved
           )
{
  hModuleInstance=(HINSTANCE)hModule;
  return TRUE;
};

////////////////////////////////////////////////////////////////////////////

CDeviceX::CDeviceX()
{
  pConnectedPort=0;
  pDeviceName=strdup("Device1");
  idDlgConfig=ConfigureDialog;
  hwndDlgConfig=0;
};

CDeviceX::~CDeviceX()
{
};

LPCSTR __stdcall CDeviceX::GetDeviceName()
{
  return pDeviceName;
};

DWORD __stdcall CDeviceX::GetDeviceCoreFileName(LPSTR szFileName,DWORD nSize)
{
  return GetModuleFileName(hModuleInstance,szFileName,nSize);
};

VOID __stdcall CDeviceX::Step()
```

```
{
};

BOOL __stdcall CDeviceX::ConnectPort(IPort* pPort)
{
  pConnectedPort=pPort;
  return TRUE;
};

IPort* __stdcall CDeviceX::GetConnectedPort()
{
  return pConnectedPort;
};

VOID __stdcall CDeviceX::Release()
{
  FreeConsole();
  delete this;
};

...
```

Функция `DLLMain` создается средой *Visual C++* при создании проекта DLL. В теле функции сохраняется дескриптор модуля устройства. Он будет необходим при создании диалога настроек.

В конструкторе `CDeviceX()` устанавливаются начальные значения для указателя на подсоединенный порт, имени устройства, идентификатора и дескриптора диалога настроек. Следует помнить, что имя устройства должно совпадать с именем, возвращаемым экспортируемой функцией `GetDeviceName`.

Деструктор `~CDeviceX()` и функция `Step()` не выполняют никаких действий в данном примере.

Функция `GetDeviceName` класса `CDeviceX` возвращает указатель на имя устройства. В данном случае, это имя - "*Device1*".

Функция `GetDeviceCoreName(LPSTR szFileName,DWORD nSize)` вызывает стандартную функцию `GetModuleFileName` для заполнения `szFileSize` полным путем к DLL устройства. Эта функция вызывается симулятором и должна быть описана.

В теле функции `ConnectPort(IPort* pPort)` сохраняется указатель на подсоединенный порт.

Функция `GetConnectedPort` возвращает указатель на подсоединенный порт, сохраненный в теле функции `ConnectPort`.

Функция `Release` удаляет из памяти консоль, созданную для вывода данных, и экземпляр класса `CDeviceX`.

Описание функции `SendData` изложено в главе "Прием и вывод данных".

В конце файла *DeviceX.cpp* расположены функции `GetDeviceCount`, `GetDeviceName` и `CreateDevice`. Они описаны в главе "Экспортируемые функции".

Создание и экспорт диалога настроек устройства *DeviceX.Device1* описаны в главе "Диалог настроек".

12.2. Диалог настроек

Устройство *DeviceX.Device1* не имеет никаких настроек. Тем не менее, для подключения устройства к симулятору MultiCore необходимо создать пустой диалог настроек. Диалог, используемый в данном примере, приведен на рисунке 11.1:

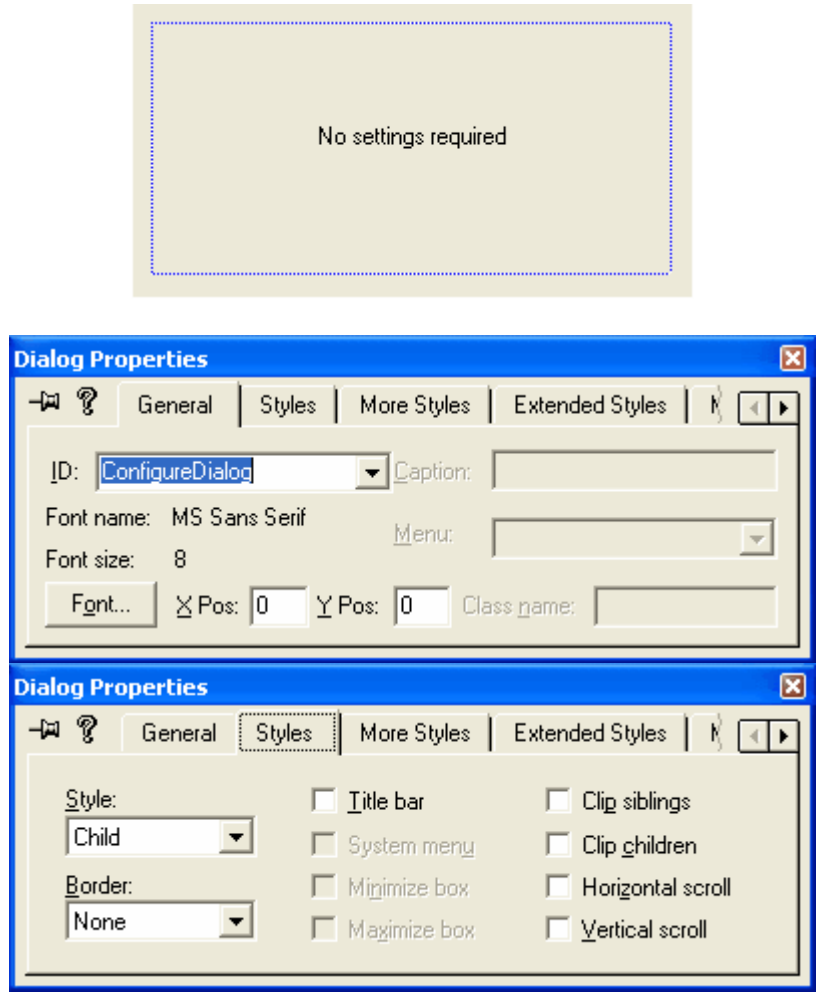


Рисунок 11.1. Диалог настроек внешнего устройства DeviceX.

Стиль диалога должен быть обязательно `WS_CHILD`, границы должны быть `WS_NONE`.

Идентификатор диалога (ConfigureDialog) сохраняется в теле конструктора класса CDeviceX.

Рассмотрим способ создания диалога:

```
//файл DeviceX.cpp

...

LRESULT CALLBACK CDeviceX::ConfigDialogProc(HWND hwndDlg, UINT uMsg, WPARAM wParam,
LPARAM lParam)
{
  switch (uMsg) {
  case WM_INITDIALOG:
     return TRUE;
  case WM_DESTROY:
     return FALSE;
  }

  return FALSE;
};

HWND __stdcall CDeviceX::Configure(HWND hwndParent)
{
  hwndDlgConfig=CreateDialog(hModuleInstance,MAKEINTRESOURCE(idDlgConfig),hwndParent,(
DLGPROC)ConfigDialogProc);
  return hwndDlgConfig;
};

VOID __stdcall CDeviceX::SetConfigurationFile(LPSTR szFileName)
{
};
```

Функция ConfigDialogProc описывает обработку сообщений диалогом настроек. Диалог настроек должен подгружать настройки по сообщению WM_INITDIALOG и сохранять их по сообщению WM_DESTROY. Так как в этом примере настройки не используются, никаких операций загрузки/сохранения в функции ConfigDialogProc нет.

Функция Configure(HWND hwndParent) создает диалог настроек устройства. Для этого вызывается стандартная функция CreateDialog. Первым параметром функции является дескриптор модуля устройства, он сохраняется в теле функции DLLMain. Вторым параметром указан сохраненный в конструкторе класса CDeviceX идентификатор диалога настроек. В качестве родительского окна указывается аргумент функции Configure - hwndParent. Функция ConfigDialogProc является обработчиком сообщений для создаваемого диалога.

Функция SetConfigurationFile используется для указания файла настроек. Так как в этом примере нет настроек, функция не выполняет никаких действий.

При подключении устройства *DeviceX.Device1* к симулятору MultiCore диалог настроек появится в диалоге Devices (рисунок 11.2):

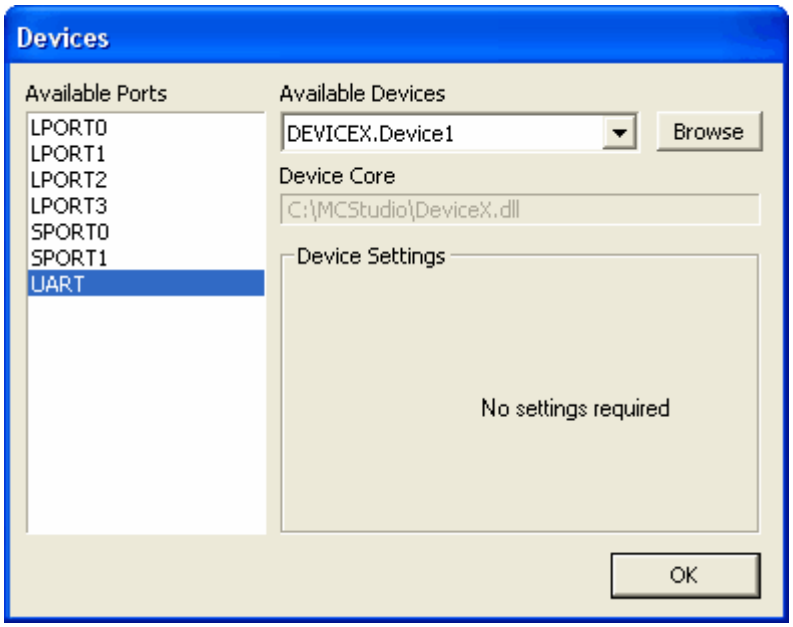


Рисунок 11.2. Диалог подключения внешнего устройства DeviceX.

12.3. Прием и вывод данных

Вывод принятых устройством *DeviceX.Device1* данных осуществляется в консоль. Консоль создается при вызове функции `CreateDevice`. Она описана в главе "Экспортируемые функции". Прием данных осуществляется функцией класса CDeviceX `SendData(DWORD dwData)`.

Рассмотрим описание этой функции:

```
//файл DeviceX.cpp

...

BOOL __stdcall CDeviceX::SendData(DWORD dwData)
{
  char strData[10] = {0,0,0,0,0,0,0,0,0,0};
  int iProcessingData;
  iProcessingData=(int)dwData;
  _itoa(iProcessingData,&strData[0],10);
  WriteConsole(hConsoleOutput,&strData[0],10,NULL,NULL);
  coordCursorPos.Y++;
  SetConsoleCursorPosition(hConsoleOutput,coordCursorPos);
  return TRUE;
};

...
```

Функция `SendData(DWORD dwData)` преобразует поступившие данные `dwData` в строку `strData`, затем выводит `strData` в консоль посредством стандартной функции

`WriteConsole`, после чего переводит курсор консоли на новую строку (стандартной функцией `SetConsoleCursorPosition`).

Внешний вид консоли вывода принятых данных приведен на рисунке 11.3:

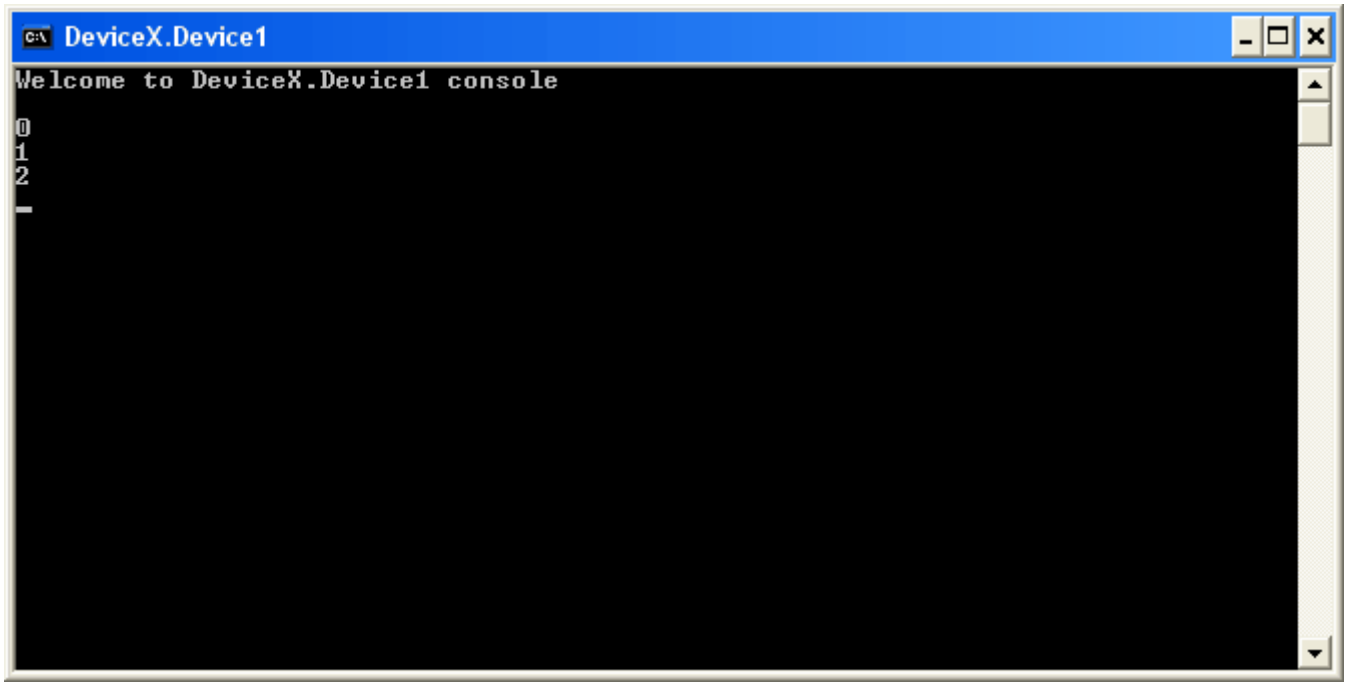


Рисунок 11.3. Консоль внешнего устройства DeviceX.

В этом примере данные пересылаются симулятором MultiCore через порт UART. Программа, осуществляющая пересылку данных, описана в главе "Передача данных в порт UART".

12.4. Экспортируемые функции

Для правильного соединения внешнего устройства с симулятором процессора MultiCore, библиотека *DeviceX.dll* экспортирует три функции:

```
//файл DeviceX.cpp

...

DWORD __stdcall GetDeviceCount()
{
  return 1;
};
```

Функция `GetDeviceCount` возвращает количество устройств, описанных в этой DLL. В данном случае описано всего одно устройство (*Device1*).

```
BOOL __stdcall GetDevName(LPSTR szBuf, DWORD iIndex)
{
  if (iIndex==0)
  {
     strcpy(szBuf,"Device1");
     return TRUE;
  }
  else
     return FALSE;
};
```

Функция `GetDevName(LPSTR szBuf, DWORD iIndex)` заполняет `szBuf` именем устройства *Device1*, если `iIndex` равен номеру устройства *Device1* (нулю).

```
LPVOID __stdcall CreateDevice(LPSTR szDeviceName)
{
  if (szDeviceName)
  {
     if (!AllocConsole())
     {
        FreeConsole();
        AllocConsole();
     };
     coordCursorPos.X=0;
     coordCursorPos.Y=0;
     hConsoleOutput=GetStdHandle(STD_OUTPUT_HANDLE);
     SetConsoleTitle("DeviceX.Device1");
     WriteConsole(hConsoleOutput,"Welcome to DeviceX.Device1 console",34,NULL,NULL);
     coordCursorPos.Y=coordCursorPos.Y=coordCursorPos.Y+2;
     SetConsoleCursorPosition(hConsoleOutput,coordCursorPos);
     return new CDeviceX();
  }
  else
     return 0;
};
```

Функция `CreateDevice(LPSTR szDeviceName)` создает устройство с именем `szDeviceName` (если такое устройство есть) и возвращает указатель на него. Кроме того, по вызову `CreateDevice` создается консоль для вывода принимаемых данных.

Экспорт трех этих функций описан в файле *exports.def*:

```
//файл exports.def
EXPORTS
  GetDeviceCount
  GetDeviceName=GetDevName
  CreateDevice
```

12.5. Передача данных в порт UART

Для передачи данных в порт UART используется программа, написанная на языке Ассемблера для RISC:

```
//файл test.s
.include "memory_12_asm.h"
.set reorder
.text
.ent main

    main:
    lui $30,CPU_BASE
    li $4,1
    move $9,$0

    li $2,0x80
    sw $2,LCR($30)
    li $2,1
    sw $2,DLL($30)
    sw $0,LCR($30)

    li $3,10
    li $8,0x60
    la $4,output
    send_data:
        lb $5,($4)
        sb $5,THR($30)
        waiting:
                lb $6,LSR($30)
                andi $6,0x60
        bne $6,$8,waiting

        addi $4,4
        addi $3,-1
    bne $0,$3,send_data

    loop:
    nop
    nop
    nop
    j loop

.end main

.data
output:
.word 0,1,2,3,4,5,6,7,8,9
```

Программа инициализирует порт UART для передачи данных, а затем последовательно передает в UART 10 цифр от нуля до девяти. Для корректной передачи каждой цифры используется цикл с постусловием, выполняющийся до тех пор, пока буфер Tx не пуст. После этого передается следующая цифра. По окончанию передачи программа входит в бесконечный цикл.

13. Предметный указатель