ВСТРОЕННЫЕ СИСТЕМЫ РЕАЛЬНОГО ВРЕМЕНИ

1. Введение

Развитие микропроцессорной техники имеет значение не только для создания высокопроизводительных вычислительных средств, но и в сфере технической инженерии, направленность которой столь разностороння, насколько разносторонней является человеческая деятельность. Диапазон применений микропроцессоров широк и в него попадают как многочисленные используемые в быту приборы, так и сложные системы, работающие в составе средств передачи и приема информации, в измерительной технике, в системах управления и слежения за различными по физическим свойствам объектами, в системах управления, в промышленном производстве и т.д. Перечень можно долго продолжать. Но смысл этого не в том, чтобы подчеркнуть разнообразие применений микропроцессорных систем, встраиваемых в тот или иной технический объект, устройство или прибор, а в том, чтобы найти в этом разнообразии то общее, те основные принципы, средства и методы, которые можно было бы положить в основу создания таких систем.

В соответствии с названием курса термин «встроенные системы» следует понимать как «*встроенные системы реального времени*». «Реальное время» для нас имеет принципиальное значение. Это, прежде всего,

- быстрая и адекватная реакция на события, которые происходят вне системы и на которые эта система обязана реагировать, и
- обработка поступающей в систему информации в реальном масштабе времени, т.е. с такой скоростью, которая, с одной стороны, позволяет получать информацию от внешних источников с допустимыми задержками, а с другой, – использовать эту информацию по назначению.

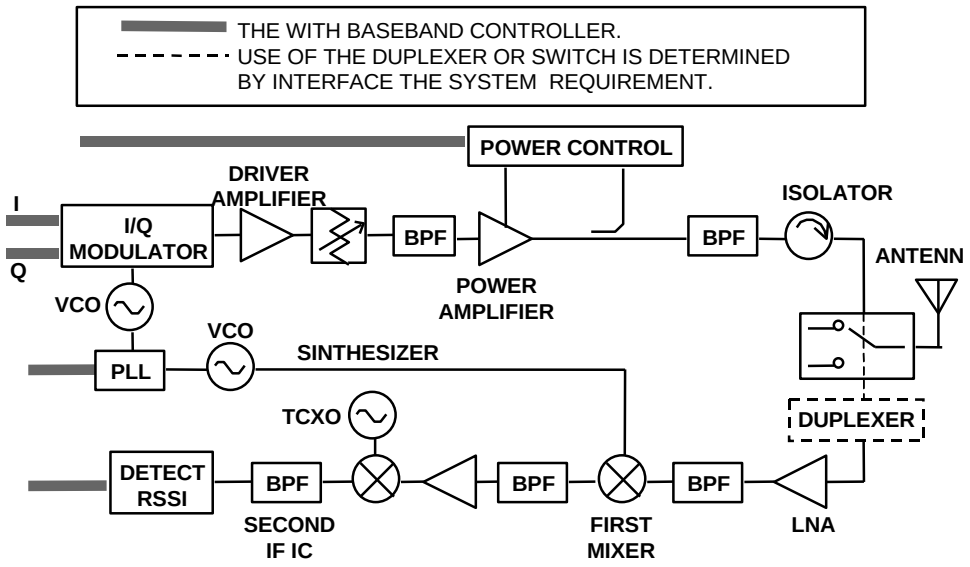
Выполняемые встроенной системой функции реализуются совокупностью аппаратных и программных средств. Аппаратные средства (*hardware*) – это микросистема, в составе которой, как минимум, имеются микропроцессор, память и устройства ввода/вывода. Сюда могут быть добавлены другие процессоры или сопроцессоры, а также специализированные под конкретные операции модули. К последним можно отнести спецвычислители, цифровые фильтры, средства аппаратной поддержки коммуникаций и т.д. – например, в виде серийных или заказных интегральных схем. Программные средства (*software*) – это операционная система, в среде которой «живут» программные модули – задачи, программный код которых, будучи переданным для исполнения процессору, реализует те функции встроенной системы, для выполнения которых эта система предназначена.

Операционные системы встроенных систем имеют свои особенности. Это отражается и в используемом для них названии – *операционные системы реального времени* – ОСРВ или *Real-Time Operating System* – RTOS. Главная особенность состоит в том, что RTOS – компактная многозадачная операцион-

ная система с достаточно развитыми средствами межзадачного взаимодействия и с быстрой и надежной реакцией на внешние события.

Рассмотрим несколько примеров, для которых встроенные системы актуальны. При этом коснемся лишь той их части, которая ближе к радиотехнике, к средствам передачи, приема и обработки информации.

Первый пример (рис. 1.1) показывает микроволновый приемопередатчик, в котором встроенная система (на рисунке не показана) формирует модулирующий сигнал, контролирует мощность передатчика, управляет частотами локальных гетеродинов и производит последетекторную обработку принимаемого сигнала.



RSSI – *Receiver Signal Strength Indicator*

Рис. 1.1

Второй пример по существу является продолжением предыдущего, поскольку в больших деталях показывает работу цифровых приёмников спутникового канала связи. На рис. 1.2 приведена структурная схемы приёмника, в составе которого имеются основной канал и канал слежения за спутником Земли (канал ССЗ).

Принимаемые сигналы промодулированы псевдослучайной последовательностью, поэтому первой функцией приёмника является операция свертки принимаемого сигнала с внутренними сигналами $П(t)$ и $П'(t)$, вырабатываемыми генератором псевдослучайных последовательностей (ГПСП), который синхронизируется от управляемого цифровым кодом $Z_{уСС3}$ синтезатора тактовой частоты СТЧ. Сигнал $Z_{уСС3}$ вырабатывается системой слежения за задержкой цифрового приемника.

Дискретизация сигнала в основном канале выполняется аналого-цифровым квадратурным преобразователем (АЦКП). Моменты выборки сигна-

ла синхронизированы с несущим колебанием и определяются синтезатором несущей частоты СНЧ, который является исполнительным элементом системы фазовой автоподстройки частоты (кольцо ФАП образуют АЦКП, программные компоненты процессора и СНЧ). Синтезатор несущей частоты управляется цифровым кодом $Z_{\text{уССН}}$.

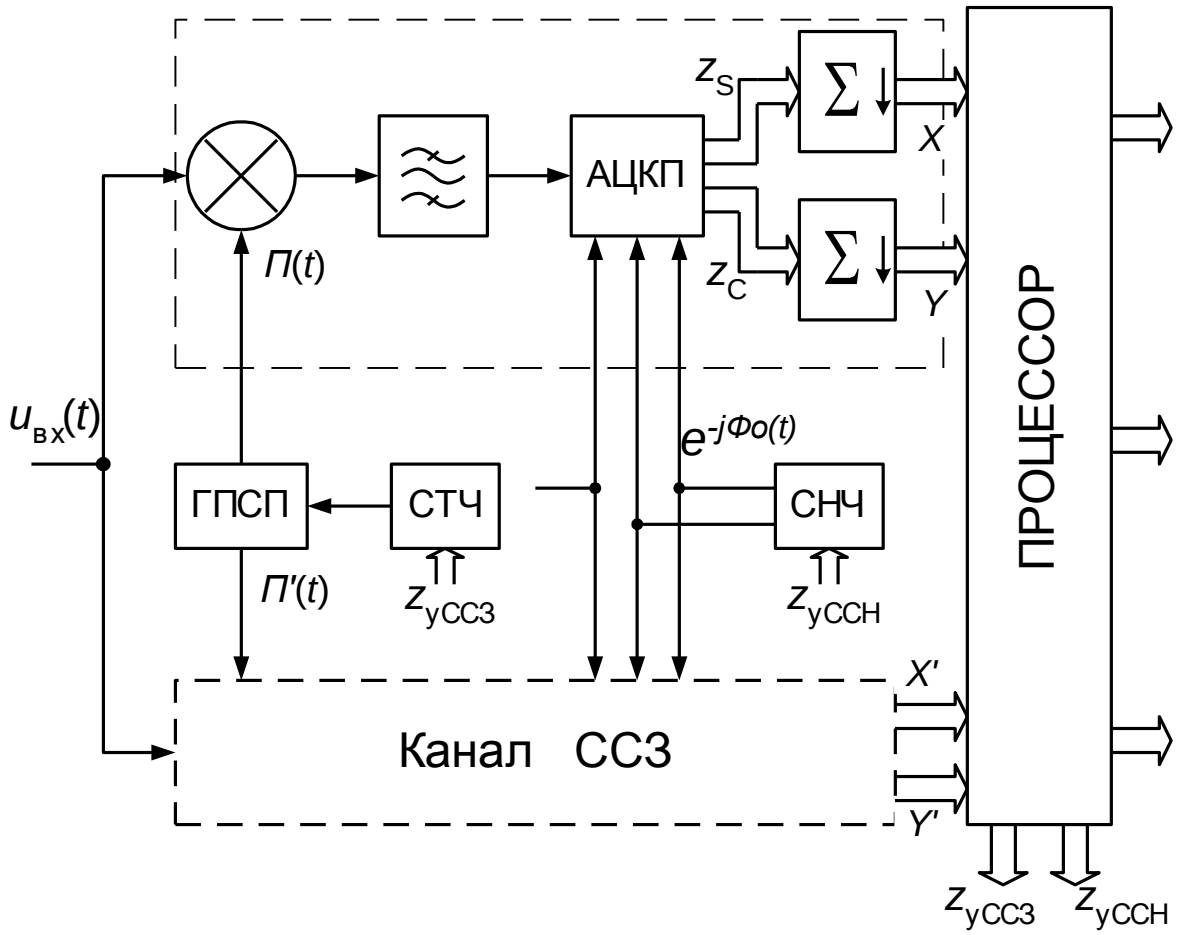


Рис. 1.2

Косинусная Z_C и синусная и Z_S компоненты комплексного цифрового сигнала с АЦКП поступают в накопители, играющие роль фильтров нижних частот. После накопителей темп выдачи последовательности числового кода становится приемлемым для обработки процессором.

Другой вариант цифрового приемника показан на рис. 1.3. От первого его отличает то, что аналого-цифровое преобразование происходи не на промежуточной, а на радио частоте. Вследствие этого операция свертки с ПСП происходит уже после аналого-цифрового преобразования и делается это в цифровой форме. Кроме того, в цифровой форме и аппаратными средствами выполняется операция комплексного перемножения и фильтрации. При этом алгоритм работы приёмника в основе своей остается прежним. Смещена, как видим, граница между аналоговой и цифровой частями приемника. Причём граница имеется и в цифровой части – это граница между аппаратной и программной цифровой обработкой.

Темп поступления цифровых данных с АЦКП во втором случае высок настолько, что не всегда есть возможность для его обработки с применением процессоров. Поэтому на этапе комплексного перемножения и накопления в сумматоре задачу цифровой обработки решается с помощью специализированных (заказных) интегральных микросхем или с помощью программируемой логики. В настоящее время такой путь решения задач цифровой обработки сигналов становится всё более актуальным, поскольку позволяет значительно облегчает разработку и создание систем реального времени.

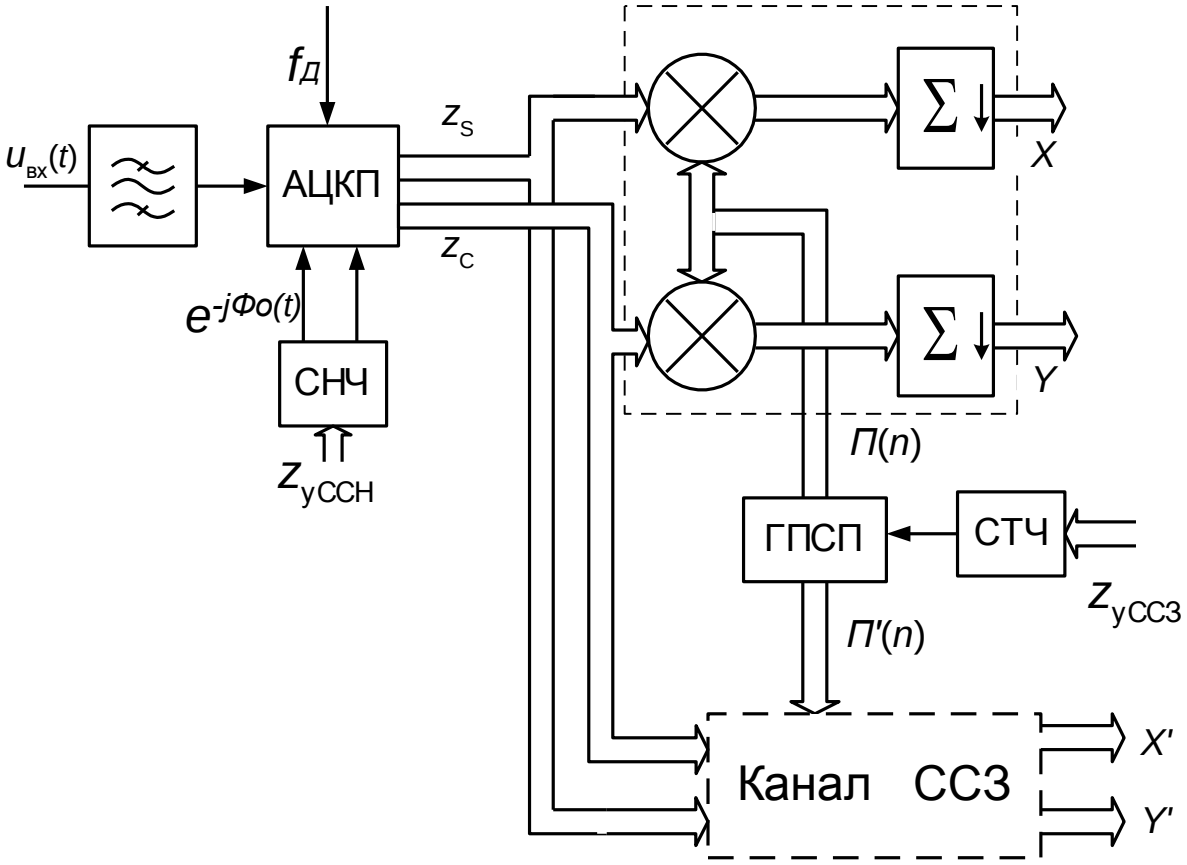


Рис. 1.3

Подытоживая сказанное, отметим, что круг решаемых с помощью встроенных систем достаточно широк. В данном курсе рассматриваются 1) программное обеспечение систем реального времени – структура и способы построения RTOS (ОСРВ) и 2) технические (аппаратные) средства встроенных систем. Такое сочетание подлежащих рассмотрению вопросов невозможно без компромиссных решений, касающихся выбора уровня строгости и степени детализации излагаемого материала. То, о чем придется вести речь не направлено на конкретные приложения, а содержит рекомендации и сведения, позволяющие глубже понять структуру встроенных систем реального времени, взаимодействие образующих эту структуру компонент и процессы в них происходящие.

2. Операционные системы для однокристальных микроконтроллеров

Вопрос о том, что является операционной системой (ОС), в достаточной мере абстрактен. Исходя из разных аспектов, можно давать совершенно разные ответы и с разной степенью детализации, приводя совершенно не похожие друг на друга примеры. Очевидно, что ОС для больших машин настолько отличается от ОС 8-разрядного процессора какой-нибудь встроенной (*embedded*) системы, что найти там общие черты – задача непростая (и, вдобавок, бессмысленная).

Для микроконтроллеров (МК) операционную систему можно охарактеризовать как совокупность программного обеспечения (ПО), дающего возможность разбить поток выполнения программы на несколько независимых, асинхронных по отношению друг к другу процессов и организовать взаимодействие между ними. Т.е. внимание обращено на базовые функции, оставляя в стороне такие вещи, присущие ОС для больших машин, как файловые системы (т.к. и файлов-то никаких, обычно, нет), драйверы устройств (которые вынесены на уровень пользовательского ПО) и проч.

Основной функцией ОС является поддержка параллельного асинхронного выполнения разных процессов (заданий) и взаимодействия между ними. Встаёт вопрос о планировании (*scheduling*) процессов – определения того, когда и какой процесс должен получить управление, когда отдать управление другому процессу. Эта задача возлагается (хотя и не полностью) на часть ядра ОС, называемой планировщиком (*scheduler*). Одна из основных задач встроенной системы состоит в обеспечении взаимодействия с устройствами ввода/вывода (УВВ). При этом с точки зрения реального времени важна скорость обмена данными с УВВ. Поэтому вначале рассмотрим способы взаимодействия процессора с УВВ.

Различают три основных способа обмена данными с УВВ:

- программный ввод/вывод;
- ввод/вывод по прерываниям;
- ввод/вывод в режиме прямого доступа к памяти.

2.1. Система с циклическим опросом устройств ввода/вывода (*Round-Robin Architecture*)

Прототипом систем с программным вводом/выводом является программная система с циклическим (*round-robin*) опросом, представляющая собой бесконечный цикл, в который встроены операции по обмену данными с УВВ (листинг 2.1).

Листинг 2.1

```
void main(void)
{
  while(TRUE)
     {
       if(!! I/O Device A needs service)
        {  !! Take care of I/O Device A
           !! Handle data to or from I/O Device A
        }
       if(!! I/O Device B needs service)
        {
           !! Take care of I/O Device B
           !! Handle data to or from I/O Device B
        }
       etc.
       etc.
       if(!! I/O Device Z needs service)
        {  !! Take care of I/O Device Z
           !! Handle data to or from I/O Device Z
        }
     }
}
```

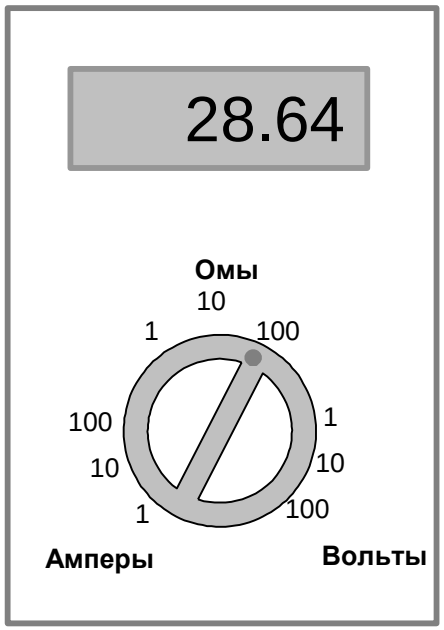


Рис. 2.1

Работу алгоритма *round-robin* поясняет программа (листинг 2.2) обслуживания цифрового мультиметра (рис. 2.1). В цикле проверяется состояние переключателя режима работы мультиметра. В зависимости от положения переключателя происходит ветвление на ту часть программы, которая реализует соответствующие выбранному режиму измерения, форматирует результат и отображает его на дисплее. *Здесь и далее словесному описанию необходимых и не подтверждёнными конкретными операциями действий предшествует сочетание символов* «*! !*».

Архитектура *round-robin* имеет то достоинство, что она достаточно проста. Но она не обладает возможностями, которые позволили бы решать с её помощью задачи реального времени в особенности применительно к системам, имеющим разветвленную и сложную конфигурацию и развитую функциональность.

Листинг 2.2

```
void vDigitMultiMeterMain(void)
{
 enum {OHMS_1,OHMS_10,...,VOLTS_100} eSwitchPosition;
  while(TRUE)
  {
   eSwitchPosition = !! Read the position of the switch;
   switch (eSwitchPosition)
```

```
    {
      case OHMS_1:
        !! Read hardware to measure ohms
        !! Format result
        break;
      case OHMS_10:
        !! Read hardware to measure ohms
        !! Format result
        break
...
...
      case VOLTS_100:
        !! Read hardware to measure volts
        !! Format result
        break
    }
     !! Write result to display
  }
```

2.2. Прерывания и внешние события

Те или иные действия используемых во встроенных системах центральных процессорных элементов (ЦПЭ, или CPU – *Central Processor Unit*) могут быть инициированы не только предписанными программой инструкциями, но и асинхронно устройствами ввода/вывода при их переходе в определенное состояние, например, в состояние готовности принимать/передавать данные от/к CPU. Переход УВВ в такое состояние носит событийный характер и фиксируется по сигналам запроса, посылаемым к CPU непосредственно или через соответствующий контроллер – котроллер прерываний. Смысл термина «прерывание» состоит в том, что CPU при получении сигнала запроса от УВВ прерывает исполнение текущей программы (точнее текущей на момент запроса инструкции) и переключается на соответствующую сигналу запроса процедуру обработки. Последнюю принято называть процедурой обработки прерывания (*Interrupt Handler* или *Interrupt Service Routine* – ISR). Важное значение имеет время реакции на запросы прерывания, что для систем реального времени является основной характеристикой.

Рассмотрим (1) действия процессора в ответ на регистрацию запроса прерывания (*Interrupt Request* – IRQ) и (2) особенности алгоритма обработки прерывания, отражаемые в ISR. Если источником запроса является, например, последовательный коммуникационный порт, то при поступлении от него сигнала IRQ, процессор должен ввести или вывести (в зависимости от режима работы) подготовленный для передачи символ и сохранить его в памяти или в регистре данных порта. По окончании передачи одного символа аналогичным образом происходит процесс передачи следующего символа и т.д. Подобные действия

повторяются и при обмене данными с другими УВВ, например, с сетевым устройством.

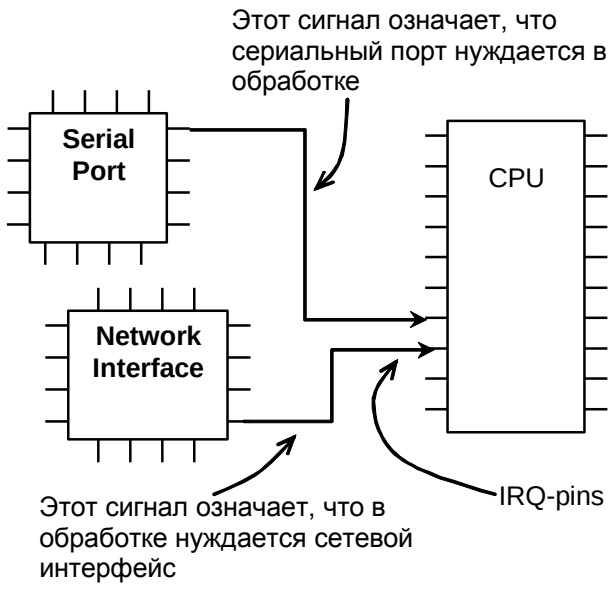


Рис. 2.2

На рис. 2.1 представлен случай, когда запросы прерывания поступают непосредственно на предназначенные для этого контакты интегральной схемы (ИС) CPU.

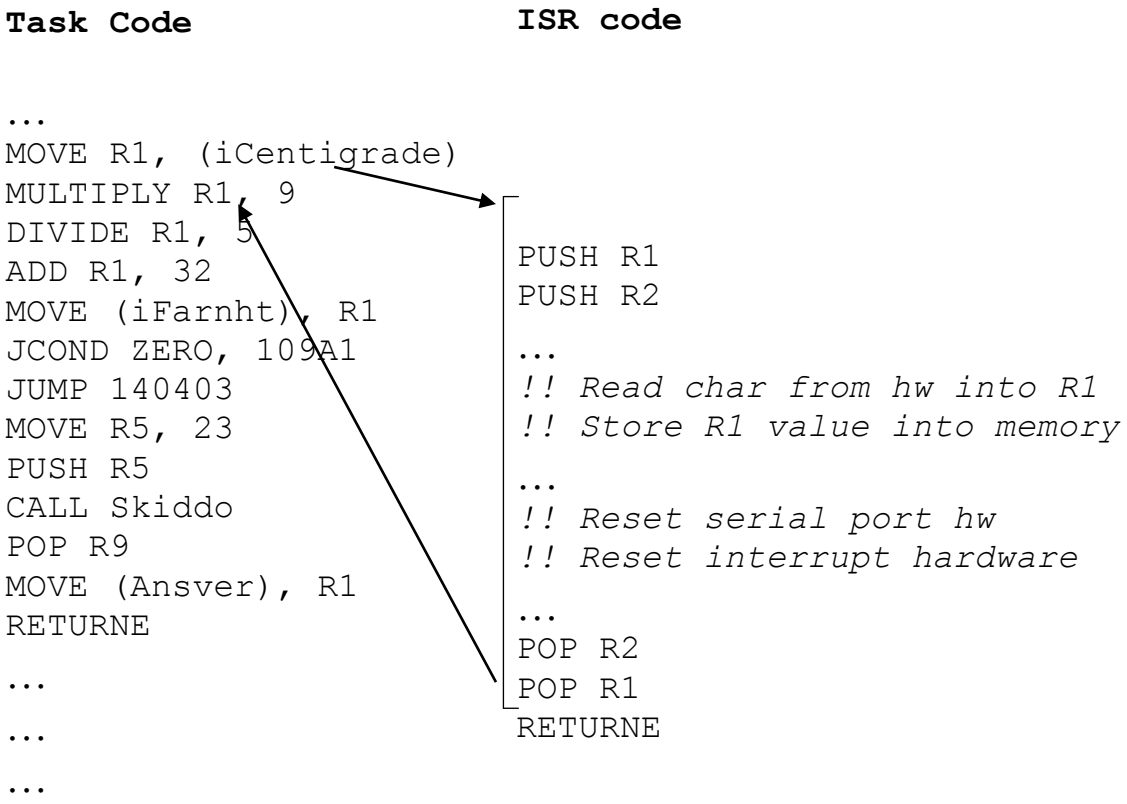
ISR является процедурой, вызов которой происходит названными выше аппаратными средствами. Процедура обработки прерывания строится в соответствии с принципами модульного программирования. Её отличительные особенности являются следствием того, что вызов ISR производится аппаратными средствами. Как и программные вызовы, вызов ISR сопровождается сохранением адреса возврата. Отличается вызов ISR тем, что при его реализации дополнительно в стековой области памяти должно быть сохранено слово состояния процессора (*Processor State Word* – PSW). Это связано с тем, что в PSW содержится флаг глобального запрещения/разрешения прерывания (например, *Interrupt Flag* – IF). Этот флаг, будучи установленным, разрешает восприятие IRQ запросов. При переходе на процедуру обработки прерывания в большинстве процессоров IF автоматически сбрасывается, запрещая тем самым восприятие других запросов прерывания и давая возможность непрерываемого выполнения вызванной по сигналу IRQ процедуры ISR. Поэтому после возврата из ISR флаг прерываний необходимо восстановить, чтобы обеспечить дальнейшее функционирование системы прерываний. Прерывания могут быть разрешены также и при выполнении ISR, в результате чего возможными становятся *вложенные* прерывания. Это делается с помощью соответствующей включённой в ISR команды.

При переходе на процедуру обработки прерывания адрес возврата процессор сохраняет автоматически. Некоторые процессоры автоматически сохраняют и PSW. Для этого чаще всего используется стек, расположенный в оперативной памяти (в оперативном запоминающем устройстве – в ОЗУ). Однако процессоры, назначением которых является обработка сигналов в реальном времени, могут иметь встроенные аппаратные средства поддержки системы прерываний, такие как аппаратный стек, альтернативные банки регистров и регистровые окна. Независимо от того, каков механизм сохранения адреса возврата и регистра флагов, реализующий его алгоритм строится по единой схеме.

Поскольку состояние процессора отражается не только в его PSW, но и в текущих значениях других регистров, то при возврате из процедуры обработки

прерывания должны быть восстановлены значения всех регистров CPU для того, чтобы можно было продолжить прерванную сигналом IRQ работу процессора. Для этого процедура обработки прерывания должна обеспечить сохранение тех регистров CPU, которые используются при работе ISR. При возврате из прерывания адрес возврата, а в некоторых процессорах и PSW восстанавливаются автоматически. По этой причине в отношении последних следует различать команды возврата из процедуры и возврата из процедуры обработки прерывания. Процесс сохранения/восстановления всех регистров процессора обозначают термином *переключение контекста – сохранение/восстановление контекста*.

Листинг 2.3



Сказанное выше иллюстрирует представленная в листинге 2.3 схема взаимодействия решаемой процессором задачи (`Task Code`) и процедуры обработки прерывания (`Interrupt Routine`). Для примера взята программа перевода вводимого значения температуры в градусах Цельсия (`iCentigrade`) в значение температуры по Фаренгейту, написанная с использованием «модельного» языка Ассемблера. Здесь запрос прерывания возникает при выполнении команды `MULTIPLY R1,9`. При этом на стадии выполнения ISR в стеке сначала сохраняются (команды `PUSH R1` и `PUSH R2`), а затем восстанавливаются (команды `POP R2` и `POP R1`) значения регистров `R1` и `R2`.

Некоторые общие вопросы

1) *Как процессор определяет адрес процедуры обработки прерывания?* Это зависит от процессора. Для некоторых из них ISR располагаются в памяти статически по заранее определенному адресу. Другие процессоры имеют более сложный механизм входа в ISR. Все точки входа – *векторы прерываний* – сводятся в единую таблицу – *таблицу прерываний*. Элементы этой таблицы обычно содержат команды безусловного перехода по стартовому адресу ISR. Существуют микропроцессоры, в которых применены оба эти механизма.

2) *Где в памяти расположена таблица прерываний?* Опять же это зависит от процессора. В одних таблица прерываний расположена, начиная с адреса 0x00000. В других микропроцессор имеет средства для указания исполняемой программе, где эта таблица находится.

3) *Возможна ли обработка прерывания во время исполнения инструкции?* Как правило – нет. В большинстве случаев процессор заканчивает исполнение текущей инструкции и только после этого реализует действия по обработке прерывания.

4) *Если два или более запросов на прерывание поступили одновременно, какое из них будет обработано первым?* То, которое имеет наибольший приоритет. В связи с этим возникает необходимость иметь достаточно развитую систему приоритетов, функции которой обычно возлагаются на специализированное устройство – контроллер прерываний. При этом распределением приоритетов можно управлять программными средствами. Программируя контроллер можно также запрещать (*маскировать*) запросы прерываний от отдельных устройств.

5) *Может ли запрос прерывания прерывать уже выполняемую ISR?* В большинстве процессоров – да. В некоторых процессорах это возможно по умолчанию. В большинстве процессоров при переходе на процедуру обработки прерывания автоматически запрещаются и необходимы программные действия, чтобы возможными стали вложенные прерывания. В некоторых случаях сигналы IRQ с более высоким приоритетом могут прервать обработку прерывания с более низким приоритетом. Если запрос IRQ поступает во время обработки прерывания с более высоким приоритетом, то низкоприоритетный запрос IRQ фиксируется, но действия по его обработке реализуются только по окончании обработки высокоприоритетного прерывания. В этой связи употребим термин «*отложенные прерывания*».

6) *Что бывает, когда сигналы IRQ поступают в то время, когда прерывания процессора запрещены?* В большинстве случаев такие запросы запоминаются контроллером прерываний, но их обработка *откладывается* до того времени, когда восприимчивость процессора к запросам на прерывание будет восстановлена.

7) *Возможны или невозможны прерывания после того, как была произведена начальная установка процессора?* Невозможны.

8) *Можно ли писать процедуры обработки прерываний на языке С?* Да, можно. Большинство компиляторов для встроенных систем имеют средства описания и распознавания ISR. Например, в виде

```
void interrupt vHandlerIRQ(void)
{
...
}.
```

При наличии в описании слова `interrupt` компилятор добавляет в процедуру (в данном случае в процедуру `vHandlerIRQ()`) код для сохранения и восстановления контекста, а также для возврата из прерывания с учетом специфики используемого процессора. Средствами языка С можно также инициализировать таблицу прерываний путем размещения в соответствующем элементе таблицы точки входа в ISR. Но нужно иметь в виду, что процедуры обработки прерываний, написанные на языке высокого уровня – ЯВУ (на языке С, в частности) характеризуются, как правило, замедленной реакцией на вызвавшее прерывание событие в сравнении с процедурами, написанными на Ассемблере. Однако использование ЯВУ имеет свои преимущества – оно аппаратно независимо, в то время как язык Ассемблера ориентирован на конкретную архитектуру CPU.

2.3. Архитектура Round-Robin с прерываниями

Программный полинг модно сочетать с обработкой устройств ввода/вывода по запросам прерывания. При этом процедуры обработки прерываний выполняют лишь наиболее ответственную часть работы – устанавливают флаг готовности. В последующем установленные флаги проверяются в программном цикле (листинг 2.4).

Листинг 2.4

```
BOOL fDeviceA = FALSE;
BOOL fDeviceB = FALSE;
...
BOOL fDeviceB = FALSE;
void iterrupt vHandleDeviceA(void)
{ !! Take care of I/O Device A
  fDeviceA = TRUE;
}
...

void iterrupt vHandleDeviceB(void)
{ !! Take care of I/O Device B
  fDeviceB = TRUE;
}
...
```

```
...
void iterrupt vHandleDeviceZ(void)
{ !! Take care of I/O Device Z
  fDeviceZ = TRUE;
}

void main(void)
{ WHILE(TRUE)
  {
    if(fDeviceA)
    { fDeviceA=FALSE;
      !! Handle data to or from I/O Devise A
    }
    if(fDeviceB)
    { fDeviceB=FALSE;
      !! Handle data to or from I/O Devise B
    }
    ...
    ...
    if(fDeviceZ)
    { fDeviceZ=FALSE;
      !! Handle data to or from I/O Devise Z
    }
  }
}
```

Round-Robin с прерываниями

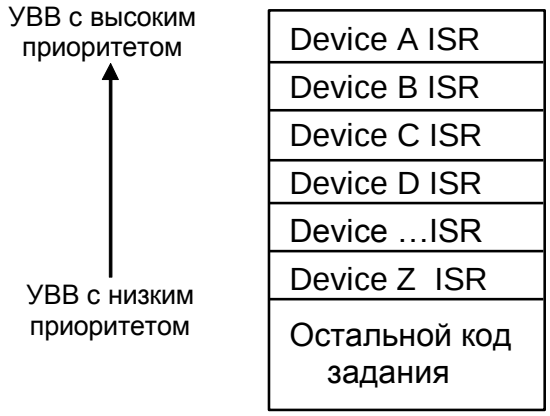


Рис. 2.3

Проблема приоритетов решается программно путём установки порядка опроса флагов готовности. Например, так, что для наиболее приоритетных УВВ (рис. 2.3) частота проверки флагов готовности выше, чем для устройств с меньшим приоритетом. При этом считается, что все процедуры, имеющие дело с обработкой прерываний, имеют приоритет более высокий, чем все иные процедуры программного кода. Кроме того, ни одна из ISR не может быть прервана ни одним другим запросом IRQ.

Архитектуре Round-Robin соответствует бесконечный цикл, на фоне которого в ответ на запросы выполняются процедуры обработки прерываний (рис. 2.4). На рисунке представлен тот случай, когда допустимы вложенные (*nested*) прерывания. Механизм вложенных прерываний предполагает расстановку их приоритетов аппаратными или программно-аппаратными средствами.

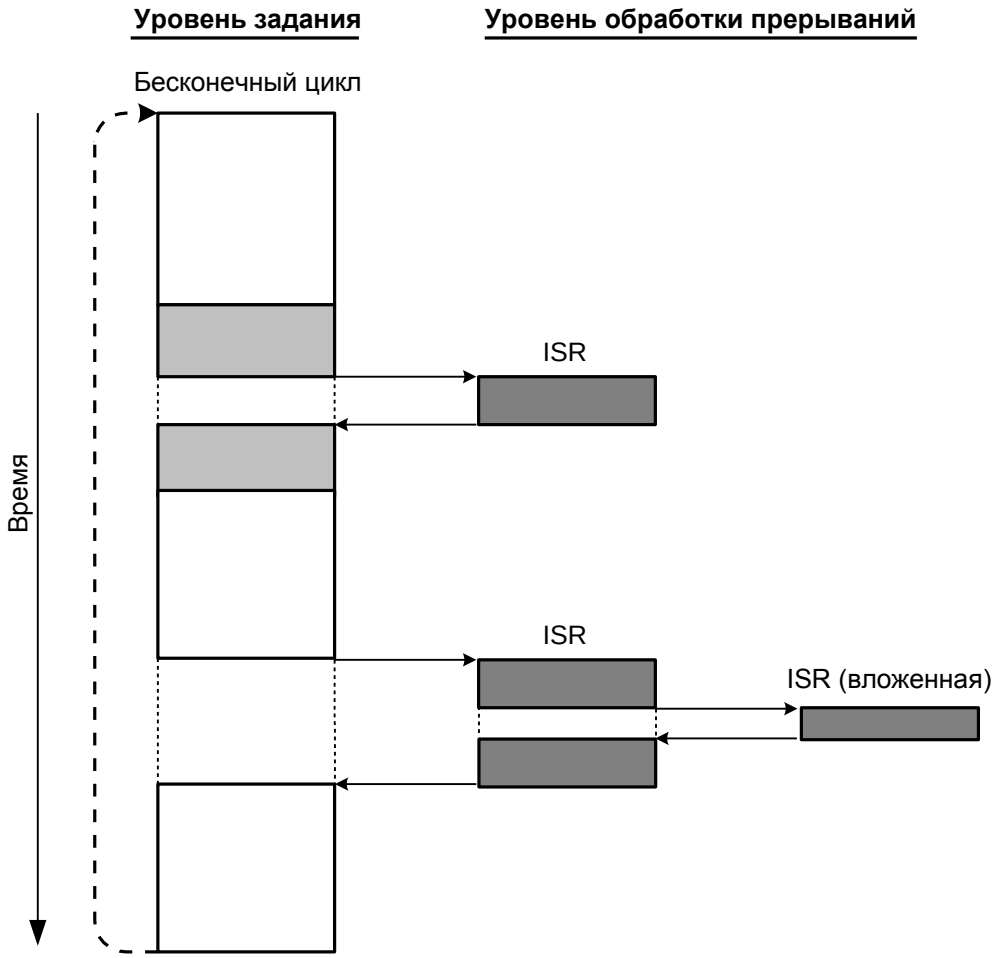


Рис. 2.4

Архитектура Round-Robin применима для многих систем. В качестве примера рассмотрим коммуникационное устройство сопряжения (мост, *bridge*). Будем считать, что мост имеет два порта, которые служат для передачи и приёма данных. При этом через один из портов передаются закодированные (шифрованные) данные. В задачу моста входит (в зависимости от направления передачи) их декодирование/кодирование и передача/приём через другой порт (рис. 2.5).

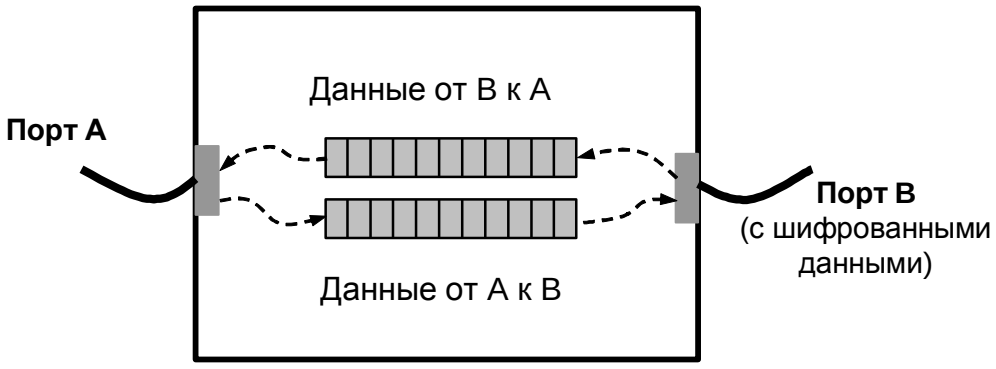


Рис. 2.5

- Всякий раз, когда символ получен по одному из коммуникационных портов, генерируется запрос прерывания и этот запрос должен быть обработан как можно быстрее, поскольку только после этого микропроцессор моста сможет прочитать следующий символ.
- МП выводит символы в коммуникационные порты с разделением времени. После того, как МП передал символ в коммуникационный порт, передатчик канала связи должен освободить порт для приёма следующего символа. Считаем, что аппаратура коммуникационного порта не вносит задержки в процесс передачи данных.
- Процедура обслуживания такова, что вводимые/выводимые символы сохраняются/берутся в/из соответствующей направлению передачи очереди. Поскольку передача данных идёт в двух направлениях, то таких очередей – две. Перед записью поступившего из порта символа в очередь производится проверка наличия в очереди свободной ячейки. С очередями работают как процедуры обработки прерываний, так и основная программа. Поэтому нужны средства корректного разделения этого совместно используемого ресурса.
- Процедуры кодирования и декодирования символов выполняются с разделением времени.

Структура программного кода, обеспечивающего функционирование моста, представлен в листинге 2.5.

Листинг 2.5

```
#define QUEUE_SIZE 100
typedef struct
{ char chQueue[QUEUE_SIZE];
  int iHeard;       /* Place to add next item */
  int iTail;        /* Place to read next item */
} QUEUE;
static QUEUE qDataFromLinkA;
static QUEUE qDataFromLinkB
static QUEUE qDataTonLinkA;
static QUEUE qDataToLinkB;

static BOOL fLinkAReadyToSend = TRUE;
static BOOL fLinkBReadyToSend = TRUE;

void iterrupt vGotCharacterOnLinkA(void)
{ char ch;
  ch = !! Read character from Communication Link A;
  vQueueAdd (&qDataFromLinkA, ch);
}

void iterrupt vGotCharacterOnLinkB(void)
{ char ch;
```

```
  ch = !! Read character from Communication Link B;
  vQueueAdd (&qDataFromLinkB, ch);
}

void iterrupt vSentCharacterOnLinkA(void)
{
  fLinkAReadyToSend = TRUE;
}

void iterrupt vSentCharacterOnLinkB(void)
{
  fLinkBReadyToSend = TRUE;
}

void main(void)
{
  char ch;

  /* Initialize the queue */
  vQueueInitialize (&qDataFromLinkA);
  vQueueInitialize (&qDataFromLinkB);
  vQueueInitialize (&qDataToLinkA);
  vQueueInitialize (&qDataToLinkB);

  /* Enable the interrupts */
  enable();
  while(TRUE)
  {
    vEncrypt();
    vDecrypt();
    if(fLinkAReadyToSend && QueueHasData(&qDataToLinkA))
    {  ch = chQueueGetData(&qDataToLinkA);
       disable();
       !! Send ch to Link A
       fLinkAReadyToSend = FALSE;
       enable();
    }
    if(fLinkBReadyToSend && QueueHasData(&qDataToLinkB))
    {  ch = chQueueGetData(&qDataToLinkB);
       disable();
       !! Send ch to Link B
       fLinkBReadyToSend = FALSE;
       enable();
    }
  }
}
```

```
void vEncrypt(void)
{
  char chClear;
  char chCryptic;

  /* While there are characters from port A ... */
  while(fQueueHasData(&qDataFromLinkA))
  {
    /*...encrypt them and put them on queue for port B */
    chClear = chQueueGetData(&qDataFromLinkA);
    chCryptic = !! Do encryption (this code is deep secret)
    vQueueAdd(&qDataToLinkB, chCryptic);
  }
}

void vDecrypt(void)
{
  char chClear;
  char chCryptic;

  /* While there are characters from port B ... */
  while(fQueueHasData(&qDataFromLinkB))
  {
    /*...decrypt them and put them on queue for port A */
    chCryptic = chQueueGetData(&qDataFromLinkB);
    chClear = !! Do decryption (no one understand this code)
    vQueueAdd(&qDataToLinkA, chClear);
  }
}
```

Микропроцессор переходит на выполнение процедур обработки прерываний `vGotCharacterOnLinkA()` и `vGotCharacterOnLinkB()` всякий раз при получении через каналы связи очередного символа. ISR читает код символа из соответствующего порта и помещает его в очередь `qDataFromLinkA` или `qDataFromLinkB`. Процедура `main()` вызывает `vEncrypt()` и `vDecrypt()`, каждая из которых читает содержимое своей очереди, кодирует или декодирует полученные данные. По заданной программе `main()` сканирует эти очереди, чтобы определить есть ли в них данные для пересылки. Очереди являются разделяемой областью памяти. Однако доступ к очередям со стороны разных процедур конфликтов не вызывает, поскольку для работы с очередями используются специально предназначенные для этого функции.

Две переменные `fLinkAReadyToSend` и `fLinkBReadyToSend` хранят состояние каналов *A* и *B* и по ним определяется готовность каналов передавать данные по линиям связи. Всякий раз, когда символ посылается в один из каналов, соответствующей переменной присваивается значение «`FALSE`», поскольку с этого момента времени используемое устройство становится занятым. После

того, как символ отправляется в линию связи, соответствующий канал связи генерирует сигнал запроса прерывания и процедура его обработки присваивает связанной с этим каналом переменной (`fLinkAReadyToSend` или `fLinkBReadyToSend`) значение «`TRUE`». Заметим, что всякий раз, когда `main`-программа производит запись в эти переменные или в порты каналов связи, необходимо запрещать прерывания, чтобы избежать возникновения разделения данных проблемы «*shared-data problem*».

2.4. Архитектура с управляемой очередью заданий (*Function-Queue-Scheduling Architecture*)

Более развитая архитектура, которой свойственны уже элементы планирования заданий, поясняет листинг 2.6. Это *Function-Queue-Scheduling Architecture* (FQS-архитектура). Характерным для неё является то, что `main`-программа для вызова той или иной функции обращается к очереди указателей на эти функции.

Листинг 2.6

```
!! Queue of function pointer

void interrupt vHandleDeviceA(void)
{   !! Take care of I/O Device A
    !! Put function_A on queue of function pointer
}

void interrupt vHandleDeviceB(void)
{   !! Take care of I/O Device B
    !! Put function_B on queue of function pointer
}

void main(void)
{ while(TRUE)
  { while(!! Queue of function pointers is empty )
        ;
        !! Call first function on queue
  }
}

void function_A(void)
{ !! Handle actions required by device A
}

void function_B(void)
{ !! Handle actions required by device B
}
```

Вызов процедур по их указателям должен осуществлять по некоторым заранее определенным правилам. В основу их может быть положена схема приоритетов. Например, задание, которое обеспечивает наиболее быструю реакцию, должно занимать верхнюю позицию в списке очередности на выполнение. Опасность возникает тогда, когда высоким приоритетом обладают задания с длительным временем выполнения. В таком случае для низкоприоритетных заданий вообще никогда может не дойти очередь на исполнение.

2.5. Архитектура систем реального времени (*Real-Time Operating System Architecture, RTOS-Architecture*)

Операционным системам реального времени (ОСРВ, *Real-Time Operation System* – RTOS обладающая одним важным свойством: время реакции на события в такой ОС детерминировано. Это даёт возможность оценить, через какое время с момента поступления события оно буде обработано. Конечно, такая оценка достаточно приблизительна, т.к. на момент возникновения события система может обрабатывать прерывание, и эту обработке не всегда можно прервать. Или процесс, который должен обработать событие, не имеет в данный момент контроля над процессором (например, имеет низкий приоритет и вытеснен другим более приоритетным процессом).

Рассмотрит особенности RTOS на примере двух заданий, обрабатывающих данные от двух устройств (`Device A` и `Device B`, листинг 2.7).

Листинг 2.7

```
void interrupt vHandleDeviceA(void)
{   !! Take care of I/O Device A
    !! Set signal X
}

void interrupt vHandleDeviceB(void)
{  !! Take care of I/O Device B
   !! Set signal Y
}

void Task1(void)
{ while(TRUE)
   {  !! Wait for Signal X
      !! Handle data to or from I/O Device A
   }
}

void Task2(void)
{ while(TRUE)
   {  !! Wait for Signal Y
```

```
        !! Handle data to or from I/O Device B
    }
}
…
…
```

Как и в рассмотренных ранее случаях, процедуры обработки прерываний выполняют наиболее ответственные функции по обмену данными с устройствами ввода/вывода. Кроме того, они устанавливают сигналы для оповещения программных заданий (`Task1` и `Task2`), которые заняты обработкой полученных с помощью ISR данных.

Отметим отличительные особенности RTOS.

- Сигналы в среде RTOS служат не только для обмена информацией между процедурами обработки прерываний и программными кодами задач, но используются также операционной системой ОС (код, относящийся к ОС, в листинге 2.7 не приведён) для управления заданиями. По этой причине для сигналов нельзя использовать переменные, разделяемые разными заданиями.
- Последовательность программных действий в среде RTOS не определяется пользовательскими приложениями (например, с помощью программных циклов). Планирование задач – это функция операционной системы. Подобная функция заложена, в частности, и в структуру, представленную в листинге 2.7. RTOS «знает» о всех задачах (процедурах), поскольку с их помощью выполняются возложенные на операционную систему функции. Для исполнения выбирается управляющий поток того задания или той процедуры, которые на текущий момент времени аиболее высоким приоритетом.
- ОС может временно остановить выполнение одного задания для того, чтобы дать возможность выполнения другому. Например, в том случае, когда выполняемое задание ожидает ввода/вывода или освобождения необходимого ему ресурса.

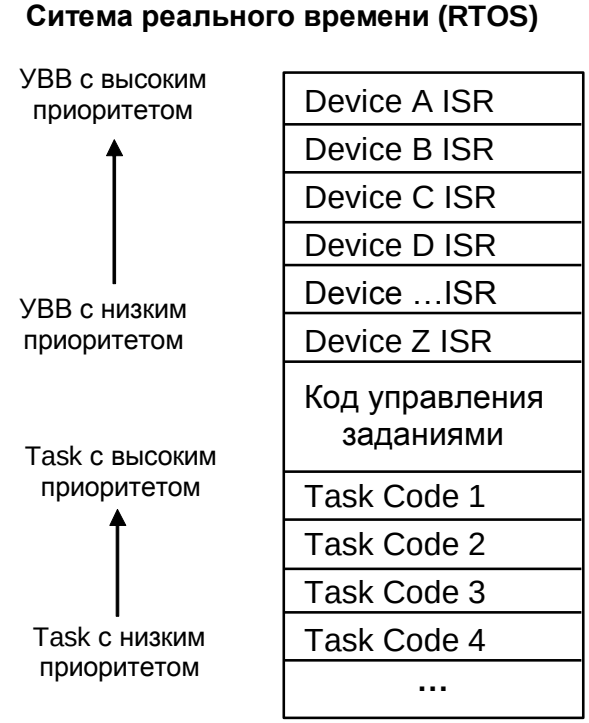


Рис. 2.6

Последнее из перечисленных отличий является наиболее существенным: системы, использующие RTOS-архитектуру способны управлять заданиями таким образом, чтобы реакция на события определялась в основном процедурами обработки прерываний. Поясним это, вновь обратившись к листингу 2.7. Пусть `Task1` имеет более высокий приоритет. Тогда при установке процедурой `vHandleDeviceA()` сигнала `X` RTOS должна запустить `Task1`. Если в это время выполнялась `Task2`, то она будет отложена, и управление будет передано `Task1`.

Из сказанного следует, что при создании задачи, выполняемой в среде RTOS, необходимо указывать приоритет, которым создаваемая задача будет обладать. Делается это в функции `main()` (листинг 2.8) с помощью вызовов соответствующей функции (в данном случае функции `StartTask()`). Кроме того, `main()` инициализирует структуры данных (функция `InitRTOS()`) и выполняет другие необходимые для запуска RTOS операции.

Листинг 2.8

```
void main(void)
{
  /* Initialize (but not start) the RTOS */
  InitRTOS();

  /* Tell the RTOS about our tasks */
  StartTask(vButtonTask, HIGH_PRIORITY);
  StartTask(vLevelTask, LOW_PRIORITY);

  /* Start the RTOS (This function never returns).*/
  StartRTOS();
}
```

Приоритеты назначаются не только заданиям, но и запросам IRQ. Пример распределения приоритетов в RTOS приведён на рис. 2.6. При этом приоритеты процедур обработки прерываний всегда выше приоритета любого из заданий.

2.6. Задача и состояние задачи

Задача (задание, *task*) – это основной строительный элемент RTOS. Чаще всего задания не сложны в написании: для большинства RTOS задачи – это подпрограммы, которые взаимодействуют с RTOS путём вызовов свойственных операционной системе функций. Последние обеспечивают соответствующие поступившим вызовам операции. При этом операционная система получает сведения о точке входа в программный код задания (процедуры), а также па-

раметры. Среди параметров – приоритет задания, расположение области памяти, выделенной под стек и другие.

Каждая задача может пребывать в следующих состояниях:

- **Является исполняемой** (*Running*). Это означает, что в данное время МП выполняет инструкции, принадлежащие заданию. Поскольку МП один (здесь не рассматривается многопроцессорная конфигурация), то исполняться может только одно задание.
- **Готова к исполнению** (*Ready*). Такая задача обозначена как исполняемая, но при условии доступности микропроцессора. В состоянии готовности одновременно могут находиться несколько задач.
- **Задача блокирована** (*Blocked*) - на данный момент времени выполнить ее нельзя независимо от того, доступен или не доступен процессор. В блокированное состояние задача переводится всегда, когда она ожидает некоторого внешнего события. Такое происходит, например, с задачей по обработке данных от сетевого интерфейса при появлении задержек в поступлении этих данных. В качестве другого примера можно взять обработку запросов клавиатуры: в паузах между нажатием клавиш задача, связанная с обработкой поступающей от клавиатуры информации, блокируется.

На рис. 2.7 показана схема переходов задания из одного состояния в другое.

Рис. 2.7

- Задание может быть блокировано вследствие принятия им самим решения о невозможности выполнения. Никакая другая задача в системе (и планировщик в том числе) не может заблокировать задачу. Задача может быть вновь переведена в состояние готовой к исполнению только после её блокирования.

- Когда задача блокирована, она не получает процессорного времени до тех пор, пока не будет выведена из этого состояния. Поэтому процедуры обработки прерываний, а также другие (связанные с блокированной) задачи должны иметь возможность посылать в случае необходимости блокированной задаче сигналы о её востребованности, то есть о необходимости перевода этой задачи в состояние исполняемой. Иначе заблокированная задача останется таковой навсегда.
- Переключение задачи из состояния готовности в состояние исполнения – это работа планировщика. Задача сама блокирует себя. Другие задачи и ISR могут перевести задачу из состояния *blocked* состояние *ready* . Только планировщик предоставляет задаче возможность исполнения, но при условии, что до этого она была переведена в состояние готовности и имеет наиболее высокий приоритет среди всех прочих готовых к исполнению.

В большинстве операционных систем реального времени состояние заданий определяется с более высокой степенью детализации, что обусловлено стремлением разработчиков к более точному их описанию. Например, состояние задачи, выведенной из режима исполнения, можно характеризовать широко используемыми терминами: задача приостановлена (*suspended*) или переведена в состояние сна, «повисшая» задача (*pended*), задача в состоянии ожидания (*waiting*), задача остановлена (*dormant*) и задержана (*delayed*). Перечисленные определения являются по сути оттенками понятия «блокированная задача». В силу этого состояние *blocked* является более обобщенным. Поэтому на данном отапе при описании RTOS будем пользоваться термином «блокированная (*blocked*) задача».

Каждая задача имеет свой собственный (*private*) **контекст**, который включает значения регистров процессора, программного счётчика и указателя стека (рис. 2.8). Однако есть и другие данные – глобальные, статические, инициализируемые и неинициализируемые и т.д., – которые разделяются разными задачами, исполняемыми в среде RTOS и её функциями. Обычно RTOS имеет свои собственные структуры данных, которые недоступны для некоторой части задач. На рис. 2.8 к глобальным данным имеют доступ все три представленные на нём задания `Task 1`, `Task 2` и `Task 3`, но им не доступны системные данные RTOS. Задачах в RTOS можно говорить как о процессах (*processes*) однако они скорее похожи на совокупность (малых) программных модулей (*threads*).

Совместное использование разными задачами данных (переменных), облегчает передачу данных от одного задания к другому. Разделяемые переменные объявляются как правило в одном программный модуле, они описываются как общедоступные (*public*), или как внешние. В качестве примера возьмём псевдокод программы контроля за расходом нефтепродуктов из резервуаров хранения – танков (листинг 2.9). Задание `vRespondToButton()` выводит на терминал результат обработки поступивших от контрольной аппаратуры данных с помощью задания `vCalculateTankLevels()`. Оба задания имеют доступ к общей области памяти – к структуре `tankdata[]`.

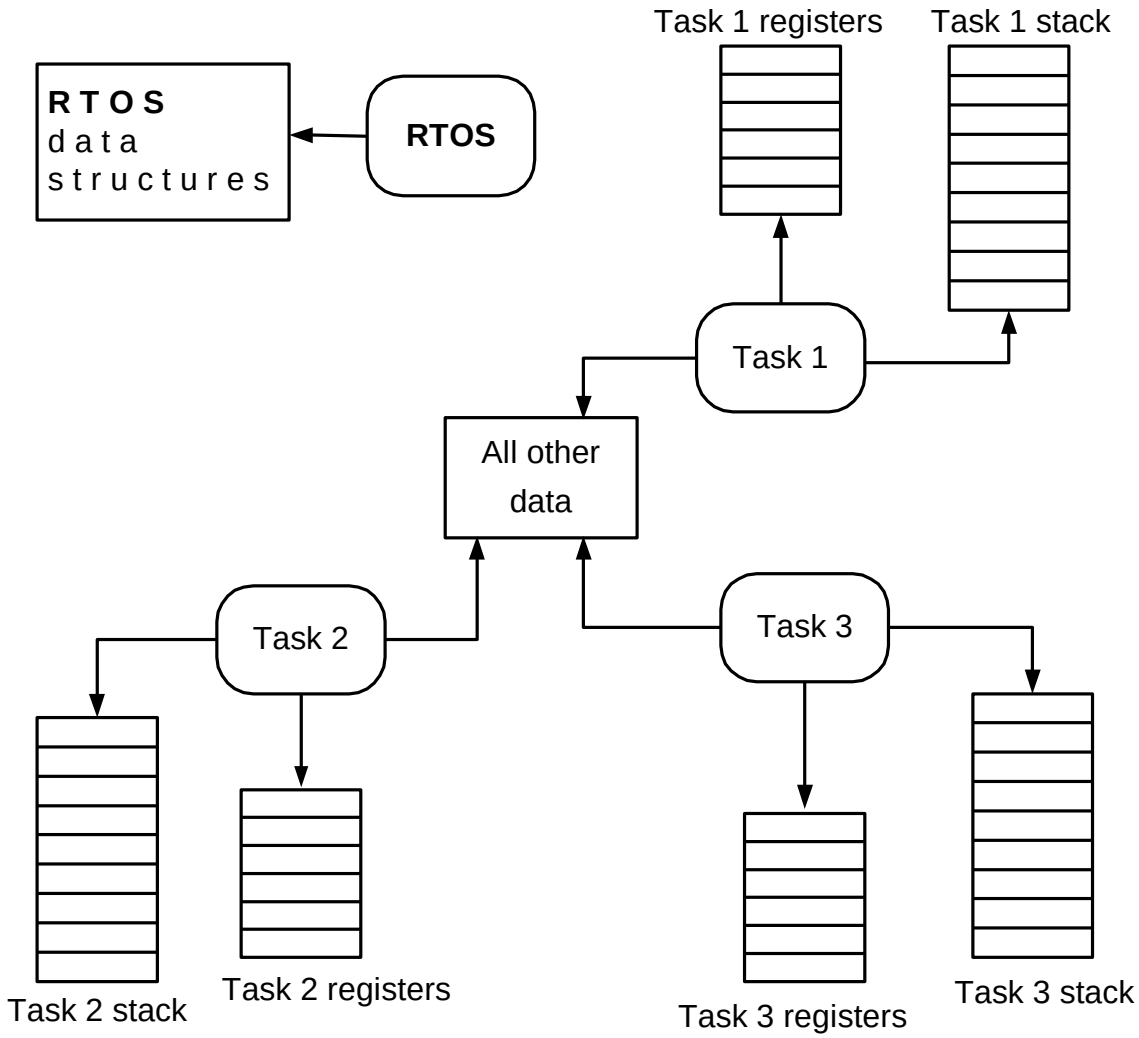


Рис. 2.8

Листинг 2.9

```
struct
{ long lTankLevel;
  long lTimeUpdated;
} tankdata[MAX_TANKS];

/* «Button Task» */
void vRespondToButton(void)       /* High priority */
{ int i;
  while(TRUE)
  { !! Block until user pushes a button
    i = !! ID of button pressed;

    printf("\nTIME: %08ld   LEVEL: %08ld",
             tankdata[i].lTimeUpdated,
             tankdata[i].lTankLevel);
  }
}
```

```
/* «Levels Task» */
void vCalculateTankLevels(void)       /* Low priority */
{ int i = 0;
  while(TRUE)
  { !! Real levels of floats in tank
    !! Do some interminable calculations
    !! Do yet more interminable calculations

    /* Store the result */
    tankdata[i].lTimeUpdated = !! Current time
       /* Between these two instruction is a
           bad place for a task switch */
    tankdata[i].lTankLevel = !! Result calculation

    !! Figure out which tank to do next
    i = !! something new
  }
}
```

Замечание. *Особенности использования термина «поток управления»*

Концепция процессов (заданий) используется очень давно. Хотя их можно рассматривать с точки зрения управляющих потоков, есть особенности в применении термина «поток» (*thread*). Он относится к управляющим потокам внутри одного процесса. С точки зрения взаимодействия процессов все потоки одного процесса имеют общие глобальные переменные (то есть поточной модели свойственно использование общей памяти). Однако потокам требуется синхронизация доступа к глобальным данным.

2.7. Планирование заданий

Информация, которую планировщик использует для предоставления заданию возможности управления процессором, содержится контрольном блоке (*Task Control Block* – TCB). Каждому заданию в среде RTOS соответствует свой TCB (рис. 2.9а), в котором сохраняется значение указателя стека SP и прописаны состояние, приоритет и связь задания со структурами данных, отражающими происходящие в системе события. Всё это учитывается в процессе переключения заданий. Блоки TCB образуют связанный список (рис. 2.9а), и в каждом из блоков указывается его связь с предыдущим и со следующим за ним по списку.

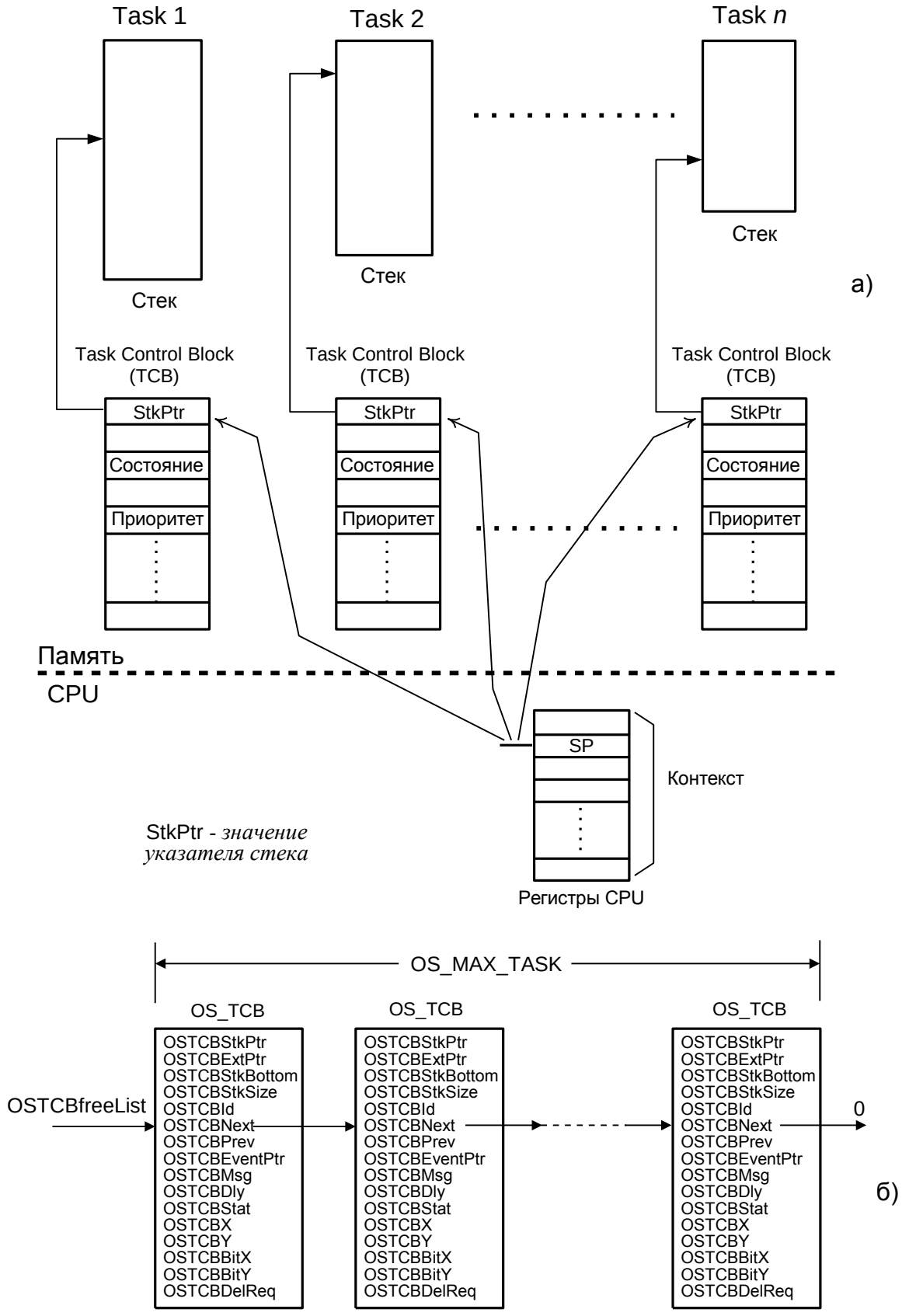


Рис. 2.9

Часть RTOS, называемая планировщиком (*scheduler*), отслеживает и сохраняет последовательность смены состояний каждой из работающих под

управлением операционной системы задач и решает, какой из них передать управление процессором. При этом используется система приоритетов. Каждой задаче, выполняемой в среде RTOS, назначается свой уровень приоритета. На текущий момент времени к исполнению принимается та задача, которая (1) не находится в числе блокированных и (2) имеет наибольший приоритет. Планировщик не «играет» с приоритетами и поэтому не может исправлять допущенные при разработке системы ошибки. Примером ошибочного решения может послужить, назначение высокого приоритета заданию, которое занимает процессор длительное время. Вероятнее всего, что такое задание не даст возможности запуска задания с более низким приоритетом, и планировщик такую ситуацию исправить не может.

Некоторые общие связанные с работой планировщика вопросы:

- *Каковы средства оповещения планировщика о блокировании и деблокировании задачи?* RTOS предоставляет ряд функций, с помощью которых планировщику сообщается о том, какие задачи ожидают событий и что это за события, а также то, какие события произошли.
- *Что происходит, если все задачи оказываются заблокированными?* Планировщик переходит на бесконечный цикл ожидания какого-либо события. Если ничего не случается, то это *«ваши проблемы»*. Поработайте над программным обеспечением.
- *Что будет в том случае, если в состоянии готовности находятся две задачи с равными приоритетами?* Это зависит от того, какая RTOS используется. В большинстве систем эта проблема решается запретом на использование равных приоритетов. Другие системы разрешают проблему путём разделения процессорного времени между равноприоритетными заданиями.
- *Если одна задача выполняется, а другая с более высоким приоритетом становится разблокированной, будет ли первая задача остановлена и переведена в состояние готовности с сохранением прав?* Для RTOS с прерываемым приоритетным обслуживанием (*preemptive* RTOS) низкоприоритетная задача останавливается всякий раз, когда деблокируется задача с более высоким приоритетом. RTOS с непрерывающимся приоритетным обслуживанием (*nonpreemptive* RTOS) низкоприоритетная задача лишается процессорного времени только в том случае, если она переводится в состояние блокированной. Характеристики *preemptive* RTOS и *nonpreemptive* RTOS имеют значительные отличия. В дальнейшем рассматривать будем только *preemptive* RTOS.

Существуют разные способы планирования заданий. Рассмотрим те из них, которые наиболее часто применяются во встроенных системах.

- Кооперативные (*cooperative*) **без вытеснения**, когда процессы выполняются последовательно, и для того, чтобы управление от одного процесса перешло к другому, нужно, чтобы текущий процесс сам отдал управление системе. Кооперативные планировщики также могут быть приоритетными или неприори-

тетными. Примером кооперативной ОС с приоритетным планированием является Salvo (www.pumpkininc.com). Кооперативное планирование без приоритетов организуется как бесконечный цикл в функции типа main, откуда по очереди вызываются другие функции, являющиеся «процессами». На рис. 2.10 представлен случай, когда управление заданию с высоким приоритетом передаётся только по завершении задания с низким приоритетом. При этом низкоприоритетное задание может приостановлено по запросу прерывания с вызовом ISR.

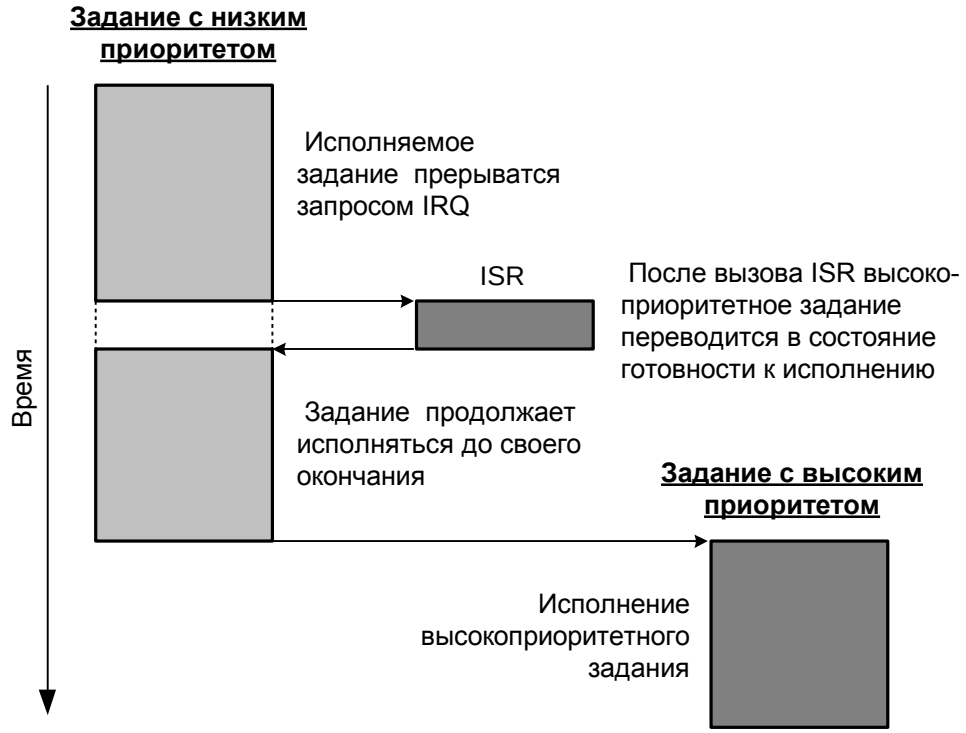


Рис. 2.10

– **С приоритетным вытеснением** (*preemptive*), когда при возникновении «работы» для более приоритетного процесса, он вытесняет менее приоритетный (т.е. более приоритетный при необходимости отбирает управление у менее приоритетного, рис. 2.11). Примером ОС с такими планировщиком являются, например, широко известная и популярная коммерческая ОС реального времени µC/OC-II (www.Micrium.com).
– **С вытеснением без приоритетов** (*round-robin* или «карусельного» типа), когда каждый процесс получает квант времени, по истечении которого управление у данного процесса отбирается операционной системой и передаётся следующему в очереди процессу.

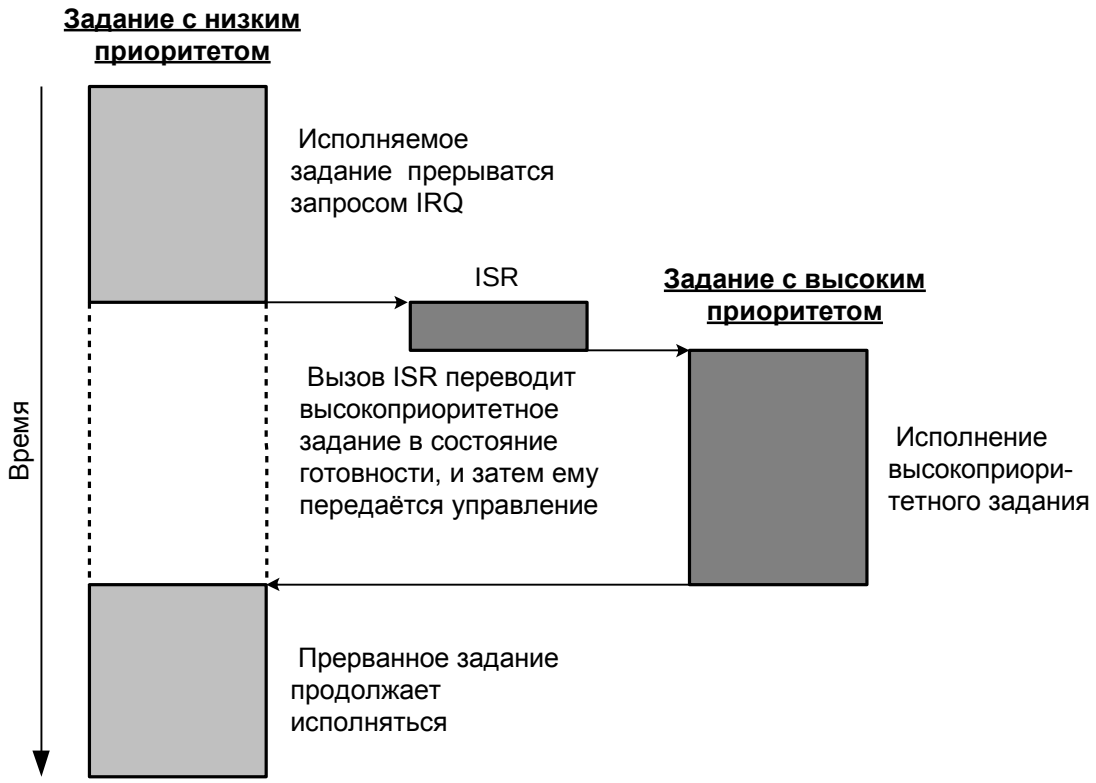


Рис. 2.11

Названы лишь некоторые (базовые) типы. Реально встречаются различные комбинации из упомянутых вариантов, что вносит значительное разнообразие в алгоритмы планирования процессов.

2.8. Структура операционной системы для микроконтроллеров

Разные системы реального времени наряду с различиями имеют много общих черт. Любая другая операционная система строится на основе ядра (*Kernel*) и процессов (заданий). Наряду с этим ОС включает средства межпроцессорного взаимодействия и менеджер памяти.

Ядро осуществляет:

- организацию процессов;
- планирование на уровне процессов и на уровне прерываний;
- поддержку межпроцессного взаимодействия;
- поддержку системного времени (системный таймер).

Исходный код состоит из двух основных частей: общей (*Common*) и платформеннозависимой (*Taget*).

Общая часть ОС включает объявления и определения функций ядра, процессов (функции создания процесса), системных сервисов.

Платформеннозависимая часть – объявления и определения, отвечающие за реализацию функций, присущих данной целевой платформе, а также расширения языка для текущего компилятора. Сюда относятся

- код переключения контекста, определение процесса `IdleProcess` на данной платформе,
- создание процесса и формирование структуры стека (`stack frame`),
- функции оформления критических секций,
- определение класса-«обёртки» (`wrapper`) критической секции,
- макросы, регулирующие поддержку аппаратного таймера данной платформы, используемого в качестве системного
- и другие платформеннозависимые компоненты.

Пример: ОС scm_RTOS (*Single-Chip Microcontroller Real-Time Operation System*) для однокристальных микроконтроллеров

Для разработки этой операционной системы требуется компилятор C++ или его укороченная версия EC++, ориентированная на scm_RTOS.

Конфигурирование системы производится путём определения макросов и указания их значений в специальном конфигурационном файле `scmRTOS_config.h`, который должен быть подключен к проекту.

Исходные тексты общей части содержатся в четырёх файлах:

- `OS_Kernel.cpp` – определения функций ядра;
- `OS_Services.cpp` – определения функций сервисов;
- `scmRTOS.h` – главный заголовочный файл, содержит основные объявления и определения;
- `scmRTOS_defs.h` – вспомогательные объявления и макросы.

Исходный код платформеннозависимой части находится в трёх файлах:

- `OS_Taget.h` – платформеннозависимые объявления и макросы;
- `OS_Taget_asm.h` – низкоуровневый код, функции переключения контекста;
- `OS_Taget_cpp.cpp` – определения функции-конструктора процесса `TProcess` и системного процесса `IdleProcess`.

Процессы (задания) в scm_RTOS

Процессы реализуют возможность создавать отдельный (асинхронный по отношению к остальным) поток управления. Каждый процесс для этого предоставляет функцию, которая должна содержать бесконечный цикл, являющийся главным циклом процесса. В листинге 2.11 представлено, как должна выглядеть функция процесса для операционной системы scm_RTOS.

Листинг 2.11

```
{1}  OS PROCESS void TSlon::Exec()
{2}  {
{3}   … // Declarations
{4}   … // Init Process's data
{5}     for(;;)
{6}      {
{7}       … // Process's main loop
{8}      }
```

```
{9}   }
```

При старте системы управление передаётся в функцию процесса, где на входе могут быт размещены объявления используемых данных {3} и код инициализации {4}, которыми следует главный цикл {5} – {8}. Пользовательский код должен быть написан так, чтобы исключить выход из функции процесса. Например, войдя в главный цикл, не покидать его (основной подход), либо, если нужно выйти из главного цикла, то попасть или в другой цикл (пусть даже пустой), или в бесконечную спячку, вызвав функцию специальную `Sleep()`. В коде процесса не должно также быть операторов возврата из функции – `return`.

Организация процессов в scm_RTOS

Функция организации процессов (листинг 2.12) сводится к регистрации созданных процессов. При этом в конструкторе каждого процесса вызывается функция ядра `RegisterProcess(TProcess*)`, которая проверяет, что процесс с таким приоритетом не зарегистрирован, и помещает значение указателя на процесс, переданного в качестве аргумента, в таблицу `ProcessTable` (аналог списка TCB, рис. 2.9) процессов системы. Местоположение этого указателя в таблице определяется в соответствии с приоритетом данного процесса, который является как бы индексом при обращении к таблице.

Листинг 2.12

```
{1}  bool OS::RegisterProcess(TRrocess* p)
{2}  {
{3}   TProcess* proc = ProcessTable[p->Priority];
{4}   if(!proc)
{5}    {
{6}     ProcessTable[p->Priority] = p;
{7}     SetProcessReady(p->Priority);
{8}     return true;
{9}    }
{10}  else
{11}   {
{12}    return false;
{13}   }
{14}  }
```

Межпроцессное взаимодействие в scm_RTOS

Так как процессы в системе выполняются параллельно и асинхронно по отношению друг к другу, то простое использование глобальных данных для обмена некорректно и опасно: во время обращения к тому или иному объекту (который может быть переменной встроенного типа, массивом, структурой, объектом класса и проч.) со стороны одного процесса может произойти прерывание его работы другим (более приоритетным) процессом, который также производит обращение к тому же объекту, и, в силу неатомарности операций обращения (чтение/запись), второй процесс может как нарушить правильность действий первого процесса, так и просто считать некорректные данные.

Для предотвращения таких ситуаций нужно принимать специальные меры: производить обращение внутри так называемых критических секций (*Critical Section*), когда прерывания запрещены, или использовать специальные средства межпроцессного взаимодействия. К таким средствам в scm_RTOS относятся:

- флаги событий (`EventFlag`);
- семафоры взаимоисключения (`Mutex`);
- каналы для передачи данных в виде очереди из байт (`Channel`);
- почтовые ящики и сообщения (`MailBox, Message`).

2.9. Межзадачное взаимодействие и синхронизация заданий

Программа в листинге 2.9 содержит ошибочные решения. Представленные в ней два задания разделяют данные, находящиеся в массиве структур `tankdata[]`. В соответствии с установленными приоритетами RTOS может остановить исполнение `vCalculateTankLevels()`, если это потребуется по запросу от клавиатуры пользовательского терминала, и перейти к исполнению задачи `vRespondToButton()`, которая обрабатывает поступающие от клавиатуры данные. При этом операционная система может остановить `vCalculateTankLevels()` во время пересылки ею данных в область памяти, где хранятся структуры `tankdata`. Поскольку эти пересылки не обозначены как неразделимые (*atomic*) операции, то в результате `vRespondToButton()` может получить лишь частично обновленные данные.

Есть много механизмов обеспечения бесконфликтного взаимодействия заданий. С их помощью разрешается проблема защиты разделяемых данных (*shared data problem*) и обеспечивается средства межзадачного взаимодействия (*intertask communication*).

Для получения доступа к разделяемому ресурсу наиболее широко используются:

1) запрещение прерываний (*disabling interrupts*) посредством низкоуровневых макросов `Disable interrupts` и `Reenable interrupts`

```
Disable interrupts;
     Access the resource (read/write from/to variables);
Reenable interrupts;
```

или путём вызова функций RTOS (например, `OS_ENTER_CRITICAL()` и `OS_EXIT_CRITICAL()`), обеспечивающих вход и выход из критической секции;

```
void Function (void)
{ OS_ENTER_CRITICAL();
  .
  /* You can access shared data in here */
  .
  OS_EXIT_CRITICAL();
 }
```

2) проверка и установка (*test-and-set*) условной переменной;

```
Disable interrupts;
if ('Access Variable' is 0) {
     Set variable to 1;
     Reenable interrupts;
     Access the resource;
     Disable interrupts;
     Set the 'Access Variable' back to 0;
```

```
        Reenable interrupts;
    } else {
        Reenable interrupts;
        /* You don't have access to the resource,
           try back later; */
    }
```

3) запрещение работы планировщика (*disabling scheduling*), например, с помощью функции RTOS `OSSchedLock()` в сочетании с функцией разрешения `OSSchedUnlock();`

```
void Function(void)
{ OSSchedLock();
  .
  . /* You can access shared data in here (interrupts
       are recognized) */
  .
  OSSchedUnlock();
}
```

4) использование семафоров (*semaphores*).

2.9.1. Семафоры

Наиболее часто используемым средством разделения ресурсов разными задачами в среде RTOS являются семафоры. Терминология в отношении семафоров не является строго установленной. Употребимы, например, парные сочетания: «*get*» – получить и «*give*» – передать, а также «*take–release*», «*wait–signal*», «*pend–post*», «*p – v*» и некоторые другие.

По-существу семафоры являются инструментом синхронизации (*synchronize*) активности взаимодействующих заданий. С помощью семафоров

- контролируется доступ к разделяемым ресурсам посредством взаимных исключений (*mutual exclusions – mutexes*),
- передаются сигналы (*signals*) о произошедших событиях (*events*).

Некоторые RTOS имеют не один, а несколько типов семафоров. Наиболее часто используются бинарные семафоры (*binary semaphores*). По отношению к ним употребим также термин «*mutexes*». Доступ к разделяемому ресурсу можно получить после проверки состояния семафора. Если ресурс свободен, то семафор «открыт». Получение семафора запрещает обращение к нему со стороны других заданий. Поэтому после завершения работы с разделяемым ресурсом семафор необходимо освободить. В представленном ниже тексте `OSSemPend()` является функцией получения (захвата) семафора, а функция `OSSemPost()` – функцией его освобождения.

```
OS_EVENT *SharedDataSem;
void Function(void)
```

```
{ INT8U err;
  OSSemPend(SharedDataSem, 0, &err);
  .
  . /* You can access shared data in here
  .    (interrupts are recognized) */
  .
  OSSemPost(SharedDataSem);
}
```

Механизм использования бинарных семафоров поясняет также рис. 2.12, где представлена ситуация, при которой доступ к разделяемому ресурсу – принтеру – требуется заданиям `Task1` и `Task2`. Прежде чем передать данные принтеру, каждое из заданий проверяет состояние семафора. Если семафор «открыт», то, получив его, задание меняет состояние семафора («закрывает» его), предотвращая тем самым возможность доступа к принтеру со стороны другого задания. После передачи данных принтеру, семафор должен быть освобождён, что делает принтер вновь доступным.

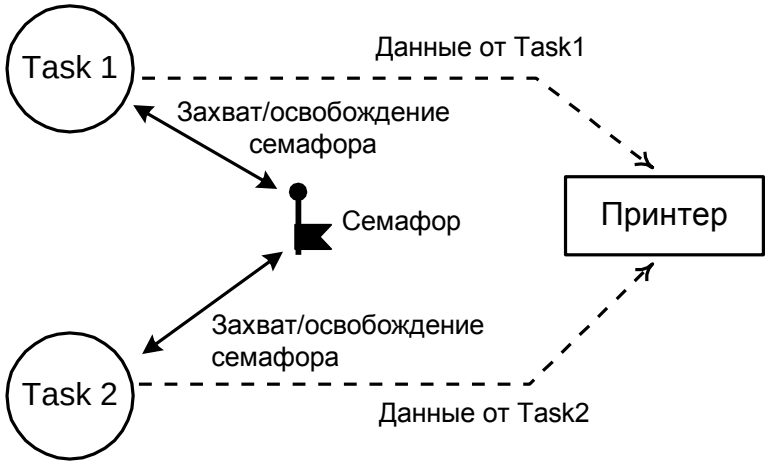


Рис. 2.12

Работа с семафорами выполняется парно вызываемыми функциями RTOS. Одна функция служит для определения возможности доступа к разделяемому ресурсу и для предотвращения одновременного использования этого ресурса другими задачами путём установки семафора в состояние «закрыт»; вторая – для установки семафора в состояние «открыт».

В листинге 2.13 представлена описанная ранее (листинг 2.9) программа контроля расхода топлива в резервуарах, но с добавлением в неё семафорных функций. Каждая из задач, прежде чем воспользоваться разделяемой областью памяти, с помощью вызова `TakeSemaphore()` проверяет семафор и, получив доступ к ресурсу, блокирует его использование другими задачами. Только после освобождения семафора вызовом `ReleaseSemaphore()` эти другие задачи могут получить семафор в своё распоряжение. В каждый текущий момент времени только одна задача может владеть семафором.

Листинг 2.13

```
struct
{ long lTankLevel;
  long lTimeUpdated;
} tankdata[MAX_TANKS];

/* «Button Task» */
void vRespondToButton(void)      /* High priority */
{ int i;
  while(TRUE)
  { !! Block until user pushes a button
    i = !! Get ID of button pressed;
    TakeSemaphore();
    printf("\nTIME: %08ld   LEVEL: %08ld",
        tankdata[i].lTimeUpdated,
        tankdata[i].lTankLevel);
    ReleaseSemaphore();
  }
}


/* «Level Task» */
void vCalculateTankLevels(void)      /* Low priority */
{ int i = 0;
  while(TRUE)
  { ...
    ...
    TakeSemaphore();
    !! Set tankdata[i].lTimeUpdated
    !! Set tankdata[i].lTankLevel
    ReleaseSemaphore();
    ...
    ...
  }
}
```

Функция `TakeSemaphore()` вызывается перед тем, как задаче `vCalculateTankLevels()` требуется обновление данных в структуре `tankdata`. Задание, которое интерпретирует команды с клавиатуры, доступа к структуре данных `tankdata` на стадии их обновления получить не сможет – при попытке обращения к этим данным семафор покажет, что запрашиваемая область памяти занята. Более детально перечисленная последовательность событий представлена на рис. 2.13.

1. При исполнении «`Level Task`» возникает запрос прерывания от клавиатуры, и это задание переводится в состояние «*готово к исполнению*».
2. После обработки запроса прерывания планировщик передаёт управление более высокоприоритетному заданию «`Button Task`».

3. Когда «Button Task» попытается получить семафор вызовом TakeSemaphore(), то обнаружится, что семафор занят, и задание «Button Task» блокируется.

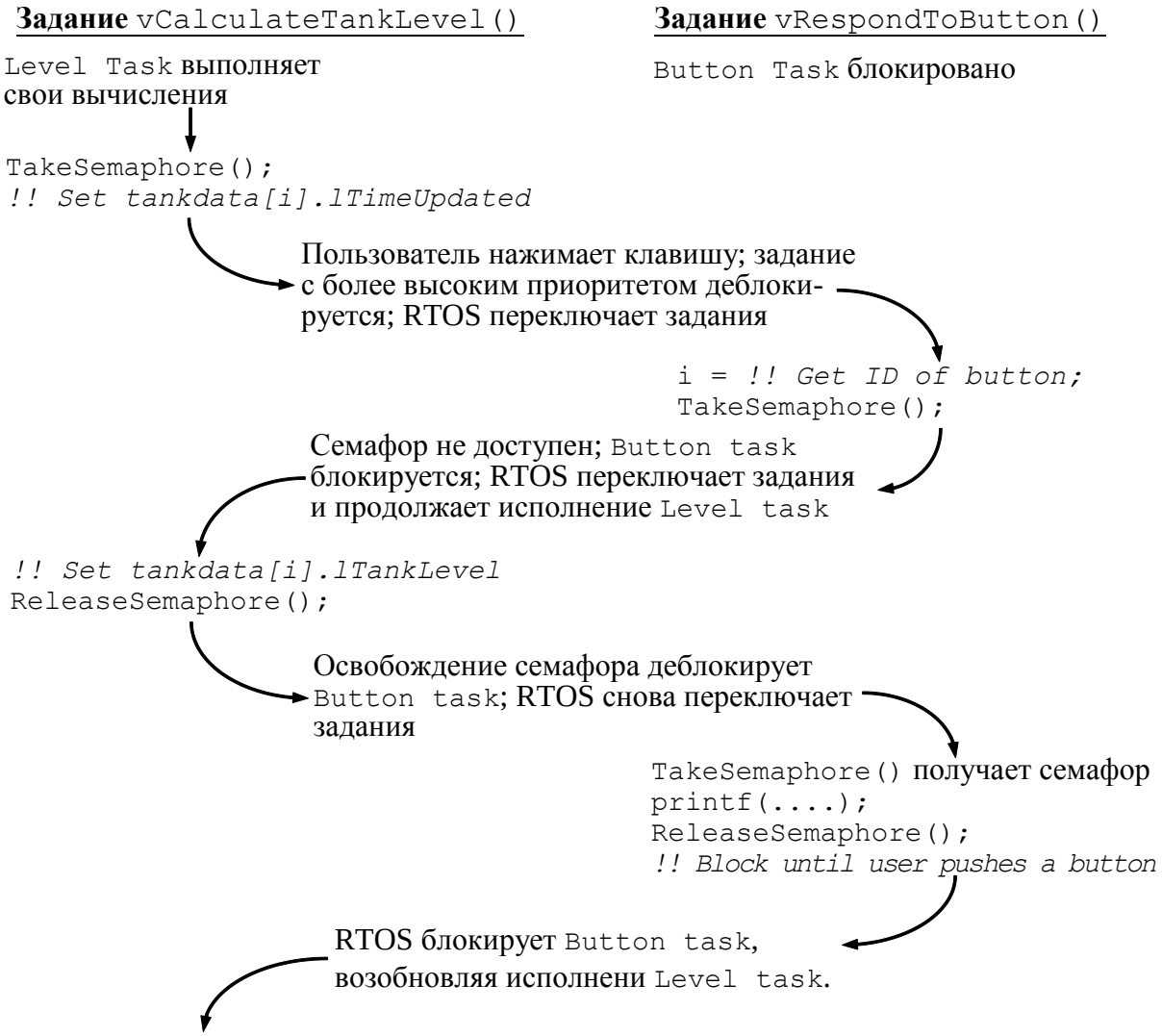


Рис. 2.13

4. RTOS просматривает список готовых к исполнению задач, убеждается, что «Level Task» находится в состоянии готовности к исполнению и даёт ей возможность закончить обновление данных в структуре tankdata.
5. Когда «Levels Task» освободит семафор вызовом ReleaseSemaphore(), «Button Task» будет разблокирована и её работа снова будет продолжена.

Листинг 2.14

```
#define TASK_PRIORITY_READ 11
#define TASK_PRIORITY_CONTROL 12
#define STK_SIZE 1024
static unsigned int ReadStk[STK_SIZE];
static unsigned int ControlStk[STK_SIZE];
static int iTemperatures[2];
```

```
OS_EVENT *p_semTemp;

void main(void)
{ /* Initialize (but do not start)the RTOS */
  OSInit();
  /* Tell the RTOS about our tasks */
  OSTaskCreate(vReadTemperanureTask,NULLP,
       (void*)&ReadStk[STK_SIZE],TASK_PRIORITY_READ);
  OSTaskCreate(vControlTask,NULLP,
       (void*)&ControlStk[STK_SIZE],TASK_PRIORITY_CONTROL);
  /* Start the RTOS. (This function never returns.) */
  OSStart();
}
void vReadTemperatureTask(void)
{ while(TRUE)
  { OSTimeDly(5); /* Delay about 1/4 second */
    OSSemPend(p_semTemp,WAIT_FOREVER);
    !! read in iTemperatures[0];
    !! read in iTemperatures[1];
   OSSemPost(p_semTemp);
  }
}

void vControlTask(void)
{ p_semTemp = OSSemCreate(1);
  while(TRUE)
  { OSSemPend(p_semTemp,WAIT_FOREVER);
    if(iTemperatures[0] != iTemperatures[1])
     !! Set off howling alarm;
    OSSemPost(p_semTemp);
   !! Do other useful work
  }
}
```

В листинге 2.14 приведена программа для мониторинга двух датчиков температуры. Используемые в этой программе имена функций и заданий начинаются с «OS». Это означает их принадлежность к соответствующей операционной системе.

Функция `OSSemPend()` получает открытый семафор и закрывает его; `OSSemPost()` освобождает семафор. `OSSemCreate()` инициализирует семафор и поэтому должна вызываться раньше, чем любая из функций `OSSemPost()` и `OSSemPend()`. Для описания семафора используется находящаяся под непосредственным управлением RTOS структура `OS_EVENT`. Передаваемый в `OSSemPend()` параметр `WAIT_FOREVER` служит для предотвращения повторного использования семафора. Обратившаяся к семафору процедура через `WAIT_FOREVER` сигнализирует о своей работе с блоком данных, доступ к кото-

рым регулируется семафором. `OSTimeDly(5)` в `vReadTemperatureTask()` вносит задержку на 0,25 с при входе в цикл `while`. Длительность задержки определяется по прерываниям от таймера, который периодически с интервалом 0,25 с выводит `vReadTemperatureTask()` из состояния сна, инициирует действия по вводу значений измеряемых температур и сохранению результатов измерений в памяти. Доступ к памяти регламентируется семафором.

Подобное листингу 2.14 использование семафоров не снимает проблему разделения данных. В чём причина?

Инициализация семафоров

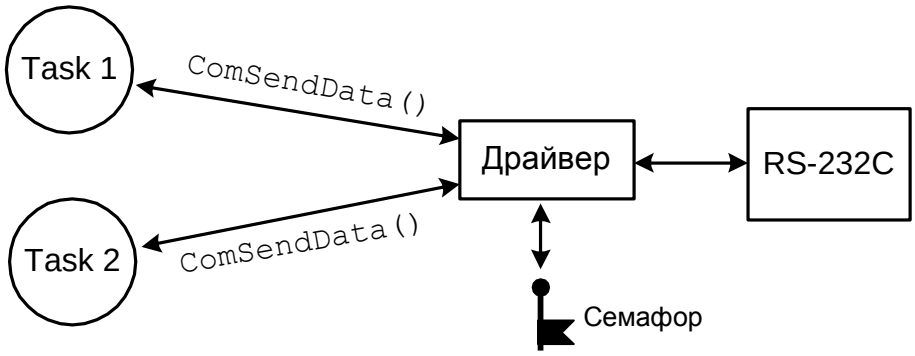
До использования семафор должен быть проинициализирован. Начальное значение семафора может лежать в пределах от 0 до 65535. Если семафор используется как бинарный (*mutex*), то он первоначально устанавливается в «закрытое» состояние, например, в состояние «1». В листинге 2.14 семафор инициализируется всякий раз при исполнении задания `vControlTask()` с помощью функции `OSSemCreate()`, и эта инициализация может произойти до освобождения семафора заданием `vReadTemperatureTask()`. Поэтому важно знать, как и когда вызов `OSSemCreate()` реализуется. Необходимо, в частности, учитывать то, что `vControlTask()` должна иметь достаточно времени для полного выполнения `OSSemCreate()`. Это время обеспечивается тем, что перед обращением к семафору задание `vReadTemperatureTask()` вызывает функцию `OSTimeDly(5)`. Однако, что будет происходить в действительности, зависит от того, какие приоритеты назначены задачам `vReadTemperatureTask()` и `vControlTask()`. Чтобы избежать проблем следует инициализировать семафоры (в данном случае вызовом `OSSemCreate()`) на стадии инициализации RTOS, например, перед функцией инициализации системы `OSInit()`.

Инкапсулированные семафоры

В рассмотренных выше случаях семафоры использует каждое задание. Удобнее бывает применять *инкапсулированные* (внедрённые – *encapsulated*) семафоры. В таком случае заданию самому не требуется получать доступ к семафору. Делаеть это можно с помощью вызова соответствующей функции. В рассмотренном ниже примере (рис. 2.14) к такой функции относится `CommSendData()` с тремя параметрами, последним из которых является `timeout`. Эта функция используется двумя заданиями, которые передают данные через *com*-порт с подтверждением состоявшейся передачи. Семафор инкапсулирован в драйвер устройства ввода/вывода.

Первое задание вызывает функцию `CommSendData()`, которая захватывает семафор для передачи данных и ждёт ответа от устройства. Если в это время другое задание через ту же функцию `CommSendData()` обратится к пор-

ту, то обнаружится, что порт занят, и это задание блокируется, пока семафор не будет освобождён после предшествующего вызова `CommSendData()`.




```
INT8U CommSendData (char *data, char *response,
                    INT16U timeout)
{ !! Захватить семафор порта RS-232C;
  !! Передать данные в порт;
  !! Дождаться подтверждения приёма данных
  !! (с установленным таймаутом);
  if (timedout) {
    Release semaphore;    !! Освободить семафор
    return (error_code); !! Сообщить об ошибке
  } else {
    Release semaphore;    !! Освободить семафор
    return (no_error);    !! Передача без ошибки
  }
}
```

Рис. 2.14

Семафоры счётного типа

Существуют семафоры счётного типа (*counting semaphore*). Используются они в том случае, когда ресурс требуется для нескольких заданий в одно и то же время. Например, счётные семафоры используются для управления доступом к пулу (набору) буферов (*buffer pool*).

Пусть в пул входят 10 буферов. Для получения буфера памяти задание обращается к менеджеру буферов, вызывая `BufReq()`. После использования буфера задание возвращает его менеджеру с помощью функции `BufRel()`. Менеджер должен удовлетворить первые 10 запросов. Если все семафоры использованы, то обратившееся к менеджеру задание блокируется, пока семафор не станет доступным. Прерывания запрещаются для обеспечения эксклюзивного доступа к связанному списку буферов. Когда задание закончило работу со своим буфером, оно вызывает `BufRel()` для возврата буфера менеджеру. Буфер помещается в связанный список ещё до того, как семафор будет освобождён. За

счёт инкапсулированного в `BufReq()` и в `BufRel()` интерфейса с менеджером буферов не требуется включение в задание деталей предоставления буфера.

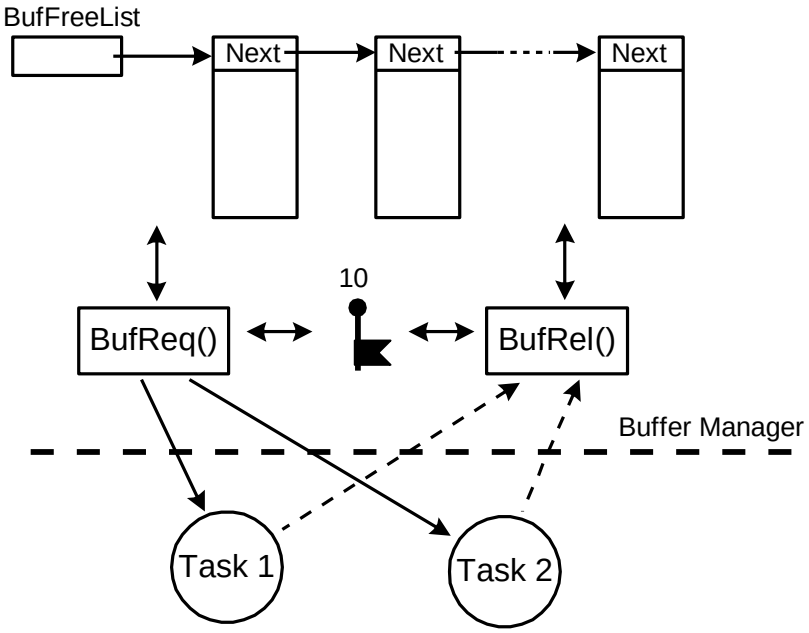


Рис. 2.15

Если семафор используется для доступа к N идентичным ресурсам, он инициализируется значением N.

Семафоры и реентерабельность

Бывает так, что несколько задач вызывают одну функцию с параметром, который модифицирует блок (иди ячейку) памяти, принадлежащей одной структуре или переменной. Такая функция как бы сама становится разделяемым ресурсом. Поэтому область памяти, к которой имеет отношение передаваемый параметр, должна быть защищена для обеспечения бесконфликтного взаимодействия заданий.

Пример программы с использованием семафора как средства защиты модифицируемой статической переменной `cErrors` приведён в листинге 2.15. Функция `vCountErrors()`, используемая заданиями `Taskl()` и `Task2()`, начинается с проверки и последующего закрытия семафора, а заканчивается его переводом в открытое состояние. Вследствие этого задание, вызвавшее `vCountErrors()` получает доступ к защищаемой семафорами области памяти. Какое бы задание не вызвало `vCountErrors()`, следующие за этим попытки воспользоваться семафором будут блокированы. Такую функцию называют *реентерабельной*. В терминах реентерабельности часть кода, использующая `cErrors`, становится атомарной, но не в том смысле, что её исполнение вообще не может быть прервано, а в том, что это исполнение не может быть прервано попыткой доступа к разделяемому ресурсу со стороны других заданий. Приме-

нительно к таким функциям в RTOS используется термин «*Nucleus*». С этим связано появление в именах функций, работающих с разделяемым ресурсом и в именах некоторых переменных, констант и функций префикса NU. В листинге 2.15 к ним относятся параметр `NU_SUSPEND` и функция `NU_Obtain_Semaphore()`, принимающая `NU_SUSPEND` в качестве параметра.

Листинг 2.15

```
void Taskl(void)
{ ...
  ...
  vCountErrors(9)
  ...
  ...
}
void Task2(void)
{ ...
  ...
  vCountErrors(11)
  ...
  ...
}

static int cErrors;
static NU_SEMAPHORE semErrors;
void vCountErrors(int cNewErrors)
{
  NU_Obtain_Semaphore(&semErrors,NU_SUSPEND);
  cErrors += cNewErrors;
  NU_Release_Semaphore(&semErrors);
}
```

Семафоры как способ передачи сигналов

Применение семафоров предоставляет простой способы связи одной задачи с другой, а также задачи с процедурой обработки прерываний. Примером может послужить взаимодействие задания по форматированию данных для вывода на печать и процедуры обработки прерываний от принтера (листинг 2.16). Форматированные данные сохраняются в фиксированной области памяти (в буфере). Сигнал о размещении данных для печати принтеру сообщается с помощью емафора.

Листинг 2.16

```
/* Place to construct report. */
static char a_chPrint[10][21];

/* Count of lines in report. */
static int iLinesTotal;
```

```
/* Count of lines printed so far. */
static int iLinesPrinted;

/* Semaphore to wait for report to finish. */
static OS_EVENT *semPrinter;

void vPrinterTask(void)
{
 BYTE byError;  /* Place for an error return. */
 int wMsg;
 /* Initialize the semaphore as already taken. */
 semPrinter = OSSemInit(0);
 while(TRUE)
  {
   /* Wait for a message telling what report to format. */
   wMsg=(int)OSQPend(QPrinterTask,WAIT_FOREVER,&byError);
   !! Format the report into a_chPrint
    iLinesTotal = !! Count of lines in the report

   /* Print the first line of the report. */
   iLinesPrinted = 0;
   vHardwarePrinterOutputLine(a_chPrint[iLinesPrinted++]);

   /* Wait for print job to finish. */
   OSSemPend (semPrinter, WAIT_FOREVER, &byError);
  }
}

void vPrinterInterrupt(void)
{ if(iLinesPrinted == iLinesTotal)
    /* The report is done. Release the semaphore. */
    OSSemPost(semPrinter);
  else
    /* Print the next line. */
    vHardwarePrinterOutputLine(a_chPrint[iLinesPrinted++]);
}
```

Предположим, что прерывания от принтера генерируются после вывода на печать одной целой форматированной строки, а ISR построчно берёт данные из буфера. В такой системе после поступления в буфер одной строки пересылка в него следующей задерживается до тех пор, пока не будет закончена печать предыдущей.

Такой способ передачи данных легко реализовать, если задача, заполнив буфер форматированной строкой, открывает семафор, сигнализируя о том, что строка помещена в буфер. При этом само задание блокируется (переводится в состояние ожидания семафора). Когда строка полностью отпечатана, ISR сиг-

налит об этом, переводя семафор в открытое состояние. Задача, определив в очередной раз, что семафор открыт, помещает в буфер новую строку.

Заметим, что семафор инициализируется в задании `vPrinterTask()`. Такой способ инициализации в данном случае позволителен. Когда задание `vPrinterTask()` форматирует первую строку и затем обращается к семафору, при второй попытке взятия семафора оно блокируется. Семафор открывает ISR и тем самым деблокирует задачу `vPrinterTask()`, которая только в этом случае приступает к своей работе.

2.9.2. Сервисы систем реального времени

Коммерчески доступные RTOS, как правило, обладают достаточно развитыми средствами поддержки многозадачности. К ним относятся, прежде всего, средства межзадачного взаимодействия, взаимодействия процедур обработки прерываний и RTOS, средства управления памятью и таймеры. Начнём со средств межзадачного взаимодействия.

Очереди сообщений, почта и нити *(Message Queues, Mailboxes, and Pipes)*

Задания в среде RTOS требуют координации их совместной работы и строятся, вообще говоря, с расчётом на имеющиеся у операционной системы средства для обмена данными. Мы уже обсуждали в связи с этим механизм семафоров. Теперь обсудим **очереди сообщений, почту и нити (каналы)**.

Для начала рассмотрим пример двух заданий `Task1()` и `Task2()`, приоритет которых высок настолько, что обеспечивает им право на первоочередное выполнение. Предположим, что время от времени эти задания обнаруживают ошибочные ситуации, о появлении которых они по сети оповещают некоторый процесс. Чтобы не вносить задержек в выполнение `Task1()` и `Task2()`, есть смысл ввести дополнительное задание `ErrorsTask()` с функцией восприятия информации от `Task1()` и `Task2()` и её перенаправления по назначению.

Подобную схему взаимодействия задач можно организовать через очередь сообщений (листинг 2.17). Когда одной из задач (`Task1()` или `Task2()`) необходимо запротоколировать сообщение об ошибке, вызывается `vLogError()` и с её помощью сообщение помещается в очередь, дело с которой имеет `ErrorsTask()`.

Листинг 2.17

```
/* RTOS queue function prototypes */
void AddToQueue(int iData);
void ReadFromQueue(int *p_iData);

void Task1(void)
```

```
{
  ⋮
  if(!! problem arises)
     vLogError(ERROR_TYPE_X);

  !! Other things that need to be done soon.
  ⋮
}

void Task2(void)
{
  ⋮
  if (!!problem arises)
     vLogError (ERROR_TYPE_Y);

  !! Other things that need to be done soon.
  ⋮
}

void vLogError(int iErrorType)
{
  AddToQueue(iErrorType);
}

static int cErrors;
void ErrorsTask(void)
{
  int iErrorType;

  while(FOREVER)
  {
    ReadFromQueue(&iErrorType);
    ++cErrors;
    !! Send cErrors and iErrorType out on network
  }
}
```

Функция `AddToQueue()` добавляет в очередь элемент, соответствующий параметру `iErrorType`, а функция `ReadFromQueue()` читает значение стоящего в начале очереди сообщения и возвращает это значение вызвавшему `ReadFromQueue()` заданию. Если очередь пуста, `ReadFromQueue()` блокирует это задание. RTOS гарантирует, что принадлежащие ей функции являются реентерабельными. Если RTOS переключается с задания `Task1()` на задание `Task2()` в то время, когда `Task1()` ещё выполняет `AddToQueue()`, то вследствие этого переключения `Task2()`также будет вызывать `AddToQueue()`. В этих условиях RTOS должна гарантировать выполнение обоих заданий. Каж-

дый раз, когда функция `ErrorsTask()` вызывает `ReadFromQueue()`, последняя выполниет действия по извлечению из очереди имеющейся в ней записи об ошибке. Если при работе `ReadFromQueue()` случится переключение с `ErrorsTask()` на `Task1()` или на `Task2()` и обратно, то `ReadFromQueue()` вследствие своей (как и `AddToQueue()`) рентабельности будет выполняться с разделением времени.

Некоторые особенности работы с очередями

При разработки систем реального времени необходимо учитывать следующие факторы:

- В большинстве RTOS очереди инициализируются с помощью специально предназначенных для этого функций. Как и в случае с семафорами инициализировать очереди нужно до их первого использования. В некоторых системах учитывается возможность управления очередями, поэтому попутно ведется распределение памяти.
- Большинство RTOS позволяют иметь несколько очередей. Поэтому при вызове работающих с очередями функций возникает необходимость в дополнительных параметрах, предназначенных для идентификации очереди. В разных RTOS делается это по-разному.
- Если есть попытка записи в полностью заполненную очередь, то операционная система должна либо послать уведомление об ошибке (так поступает большая часть RTOS), либо блокировать задание до той поры, пока другое работающее с той же очередью задание не высвободит место для новой записи (что свойственно меньшей части RTOS).
- Многие RTOS имеют функции для чтения данных из очереди. При попытке чтения пустой очереди, такие функции возвращают сообщение об ошибке. Эти же функции, как правило, поддерживают и механизм блокирования тех заданий, которые обратились к пустой очереди.
- Объём данных, которые можно поместить в очередь при однократном обращении к ней, может не совпадать с общим объёмом передаваемых данных. Многие RTOS не обладают необходимой на этот случай гибкостью. Для RTOS общепринято при обращении к очереди заносить в неё число байт и использовать указатель типа *void* на место их расположения в памяти.

В листинге 2.18 дан пример программы, выполняющей те же действия, что и программа в листинге 2.17, но с использованием реальных RTOS-функций, принадлежащих операционной системе µ*C/OS*.

Листинг 2.18

```
/* RTOS queue function prototypes */
OS_EVENT *OSQCreate(void **ppStart, BYTE bySize);
unsigned char OSQPost(OS_EVENT *pOse, void *pvMsg);
```

```
void *OSQPend(OS_EVENT *pOse,WORD wTimeout,BYTE *pByErr);

#define WAIT_FOREVER 0

/* Our message queue*/
static OS_EVENT *pOseQueue;

/* The data space for our queue. The RTOS will manage this.*/
#define SIZEOF_QUEUE 25
void *apvQueue[SIZEOF_QUEUE];

void main(void)
{
  :
  /* The queue gets initialized before the tasks are started */
  pOseQueue = OSQCreate(apvQueue, SIZEOF_QUEUE);
  :
  !! Start Task1
  !! Start Task2
  :
}

void Task1(void)
{
  :
  if(!!problem arises)
    vLogError(ERROR_TYPE_X);

  !! Other things that need to be done soon.
  :
}

void Task2(void)
{
  :
  if(!!problem arises)
     vLogError(ERROR_TYPE_Y);

  !! Other things that need to be done soon.
  :
}

void vLogError(int iErrorType)
{
  BYTE byReturn; /* Return code from writing to queue */
```

```
    /* Write to the queue. Cast the error type as a void pointer
       to keep the compiler happy. */
    byReturn = OSQPost(pOseQueue, (void *)iErrorType);
    if(byReturn != OS_NO_ERR)
       !!Handle the situation that arises when the queue is full
}

static int cErrors;

void ErrorsTask(void)
{
    int iErrorType;
    BYTE byErr;

    while(FOREVER)
    {
       /* Cast the value received from the queue back to an int.
          (Note that there is no possible error from this, so we
           ignore byErr.)*/
       iErrorType =
          (int)OSQPend(pOseQueue, WAIT_FOREVER, &byErr);

       ++cErrors;

       !! Send cErrors and iErrorType out on network
    }
}
```

Указатели и очереди

Листинг 2.18. поясняет широко используемый интерфейс, который позволяет использовать один указатель, описанный как указатель на *void*, при каждом обращении к очереди. Кроме того, он является иллюстрацией техники программирования, используемой при пересылке небольшого количества данных, резервируя память под эти данные с помощью массива указателей на *void*.

Идея, положенная в основу другого интерфейса, состоит в том, что одно задание может передать некоторое количество данных другому заданию, поместив их в буфер и снабдив указателем, размещённым в очереди. Эту технику иллюстрирует листинг 2.19. Задание `vReadTemperaturesTask()` вызывает библиотечную С-функцию `malloc` для размещения переменных, в которые помещаются значения двух измеряемых температур, и записывает указатель на этот буфер в очередь. `vMainTask()` извлекает указатель на буфер из очереди, сравнивает полученные значения температур и освобождает выделенную под буфер память.

Листинг 2.19

```
/* Queue function prototypes */
OS_EVENT *OSQCreate(void **ppStart, BYTE bySize);
unsigned char OSQPost(OS_EVENT *pOse, void *pvMsg);
void *OSQPend(OS_EVENT *pOse,WORD wTimeout, BYTE *pByErr);

#define WAIT_FOREVER 0

static OS_EVENT *pOseQueueTemp;

void vReadTemperaturesTask(void)
{
  int *pTemperatures;

  while(TRUE)
  {
    !! Wait until it's time to read the next temperature

    /* Get a new buffer for the new set of temperatures. */
    pTemperatures = (int *)malloc(2*sizeof *pTemperatures);

    pTemperatures[0] = !! read in value from hardware;
    pTemperatures[1] = !! read in value from hardware;

    /* Add a pointer to the new temperatures to the queue */
    OSQPost(pOseQueueTemp, (void *)pTemperatures);
  }
}

void vMainTask(void)
{
  int *pTemperatures
  BYTE byErr;

  while(TRUE)
  {
    pTemperatures =
       (int*) OSQPend(pOseQueueTemp, WAIT_FOREVER, &byErr);
    if(pTemperatures[0] != pTemperatures[1])
      !! Set off howling alarm;

    free(pTemperatures);
  }
}
```

Почтовые ящики (*Mailboxes*)

Почтовые ящики во многом напоминают очереди. Типовая RTOS имеет функции для создания почтовых ящиков, предназначенных для передачи сообщений сообщений. Почтовые ящики разных RTOS имеют отличия. Перечислим некоторые из них.

- Как правило, RTOS позволяют размещать в одном почтовом ящике несколько сообщений, и число этих сообщений в таких системах устанавливается при создании почтовых ящиков. В то же время есть системы, которые не допускают размещения в одном почтовом ящике на текущий момент времени более чем одного сообщения. Причём следующее сообщение в ящик не может быть помещено, если не прочитано предыдущее.
- Есть системы, которые не ограничивают число передаваемых через один почтовый ящик сообщений. Имеются ограничения только на общее число сообщений, которые содержатся во всех имеющиеся в системе почтовых ящиках.
- В некоторых RTOS предусмотрена система приоритетов для передаваемых по почте сообщений. Независимо от порядка их поступления сообщение с высоким приоритетом выдается раньше низкоприоритетного сообщения.

Например, в *MultiTask*! системе от U.S. Software Corporation каждое сообщение является указателем на «*void*». Все почтовые ящики в такой системе создаются при её конфигурации. В процессе конфигурации системы определяется и число почтовых ящиков. Для работы с почтой используются следующие три функции:

```
int sndmsg(unsigned int uMbId, void *p_vMsg,
     unsigned int uPriority);
void *rcvmsg(unsigned int uMbId, unsigned int uTimeout);
void *chkmsg(unsigned int uMbId);
```

Во всех трёх функциях параметр `uMbId` является идентификатором почтового ящика. Функция `sndmsg()` добавляет сообщение по указателю `p_vMsg` в очередь почтового ящика с идентификатором `uMbId`, присваивает сообщению приоритет `uPriority`. Если `uMbId` является недействительным или число сообщений во всех ящиках превысило допустимое значение, возвращается ошибка через переменную типа int. Функция `rcvmsg()` возвращает сообщение с наибольшим приоритетом из ящика, помеченного идентификатором `uMbId`; она же блокирует задание, которое обратилось к пустому ящику. Задача может использовать параметр `uTimeout` для установки времени ожидания сообщения. Функция `chkmsg()` возвращает первое поступившее в почтовый ящик сообщение и NULL, если ящик пуст (последнее означает, что NULL-сообщение не может быть действительным для рассматриваемой многозадачной системы).

Нити (Каналы, *Pipes*)

Нити также очень похожи на очереди. RTOS может создавать нити, записывая в них или читая из них. В деталях это средство межзадачного взаимодействия зависит от конкретной операционной системы. В частности:

- некоторые RTOS позволяют передавать с помощью нитей сообщения любой длины (в отличие от почтовых ящиков и очередей, для которых размер сообщений ограничен);
- нити в некоторых системах являются полностью байт-ориентированными. Если `Task_A()` записывает в нить 11 байт, а `Task_B()` добавляет в неё еще 19 байт, то при чтении 14-ти байт из нити `Task_C()` получит все 11 байт от первой задачи и только 3 байта от второй. Оставшиеся 16 байт теряются – их при необходимости может прочитать любая другая задача.
- Некоторые RTOS используют `fread()` и `fwrite()` из стандартной C библиотеки для чтения и записи в каналы (нити).

На каком из перечисленных выше средств коммуникаций остановить свой выбор зависит как от особенностей той или иной RTOS, так и от того, на какой аппаратной основе она базируется. Значение имеют функциональная гибкость, объём памяти, скорость вычислительных операций, время реакции на прерывания, время, в течение которого система запрещает прерывания. И это далеко не полный перечень того, что необходимо учитывать при создании и использовании систем реального времени.

Часто встречающиеся ошибки

Очереди, почтовые ящики и нити дают достаточно простые средства разделения данных между задачами в среде RTOS, однако при их использовании могут возникать неожиданные на первый взгляд проблемы. Перечислим некоторые из способов уберечься от ошибочных решений:

- Большинство RTOS при работе с очередями, почтовыми ящиками и нитями не регламентирует того, какие задачи могут передавать данные, а какие принимать. Поэтому в практике разработки встроенных систем корректное использование средств связи между задачами должно отслеживаться программистом.
- RTOS не контролирует задачи на предмет правильной интерпретации данных, принимаемых или передаваемых посредством очередей, почтовых ящиков и нитей. Если, например, одна задача передает в очередь целые, а другая из этой очереди принимает и интерпретирует их как указатели, то проблемы неизбежны. Конечно, можно полагаться на то, что такие ошибки «отловит» компилятор, но не всегда написанный вами код позволяет это сделать.
- Выход за пределы области памяти, которая выделена для очередей, почтовых ящиков и нитей –аварийная ситуация для RTOS. Одним из способов

избежать этого является резервирование большего объема памяти, чем реально того требуется для очередей, почтовых ящиков и нитей.

2.9.3. Функции таймера

Важным для встроенных систем является отслеживание текущего времени, возможность установки и измерения временных интервалов, таймаутов и т.п. Например, сетевое оборудование, приняв данные, ожидает результата проверки на соответствие своему адресу и в случае отсутствия такового возвращает эти данные в сеть. С помощью таймеров регулируются временные характеристики промышленных роботов, электрических приводов и т.д.

Примером простого сервиса, который предоставляет большинство RTOS, является задержка исполнения задания. Можно на заданное время приостановить выполнение задания и после этого вновь предоставить ему возможность выполнения. В листинге 2.20 приведён фрагмент программы телефонного вызова.

Листинг 2.20

```
/* Message queue for phone numbers to dial. */
extern MSG_Q_ID queuePhoneCall;

void vMakePhoneCallTask(void)
{
  #define MAX_PHONE_NUMBER  11

  char a_chPhoneNumber[MAX_PHONE_NUMBER];
      /* Buffer for null-terminated ASCII number */
  char *p_chPhoneNumber;
      /* Pointer into a_chPhoneNumber */
  ⋮
  while(TRUE)
  {
    msgQreceive(queuePhoneCall, a_chPhoneNumber,
       MAX_PHONE_NUMBER, WAIT_FOREVER);

    /* Dial each of the digits */
    p_chPhoneNumber = a_chPhoneNumber;
    while(*p_chPhoneNumber)
    {
        taskDelay(100);  /* 0.1 of a second silence */
        vDialingToneOn(*p_chPhoneNumber - '0');
        taskDelay(100);  /* 0.1 of a second with tone */
        vDialingToneOff();

        /* Go to the next digit in the phone number */
```

```
            ++p_chPhoneNumber;
        }
        ⋮
    }
}
```

Каждой цифре телефонного кода сопоставлен сигнал определенной частоты (тоновое кодирование). Функция `vMakePhoneCallTask()` принимает телефонный номер из очереди сообщений, а `msgQreceive()` копирует номер из очереди в `a_chPhoneNumber`. В цикле `while` вызывается `taskDelay()` в начале для того, чтобы создать паузу, а потом для генерации соответствующего набранной цифре тона. Функция `vDialingToneOn()` включает, а `vDialingToneOff()` выключает генератор тона. Функции `msgQreceive()` и `taskDelay()` относятся к числу тех, которые используются в операционной системе *VxWorks* (*Wind River System, Inc.*)

Часто возникающие вопросы

Можно ли проверить то, что `taskDelay()` *правильно отрабатывает заданное в передаваемом ей параметре число миллисекунд?* Взятая из *VxWorks* функция `taskDelay()` эквивалентна тем, что используются для внесения задержек в большинстве RTOS. В качестве параметра они получают число **импульсных системных сигналов** (тиков – *ticks*). Период следования этих сигналов всегда известен и определяется соответствующей аппаратурой.

Насколько точно `taskDelay()` *отрабатывает нужную задержку?* Точность близка к временному интервалу между соседними импульсами системного сигнала, который генерируется прерываниями от системного таймера. Прерывания следуют периодически с заданным интервалом времени.

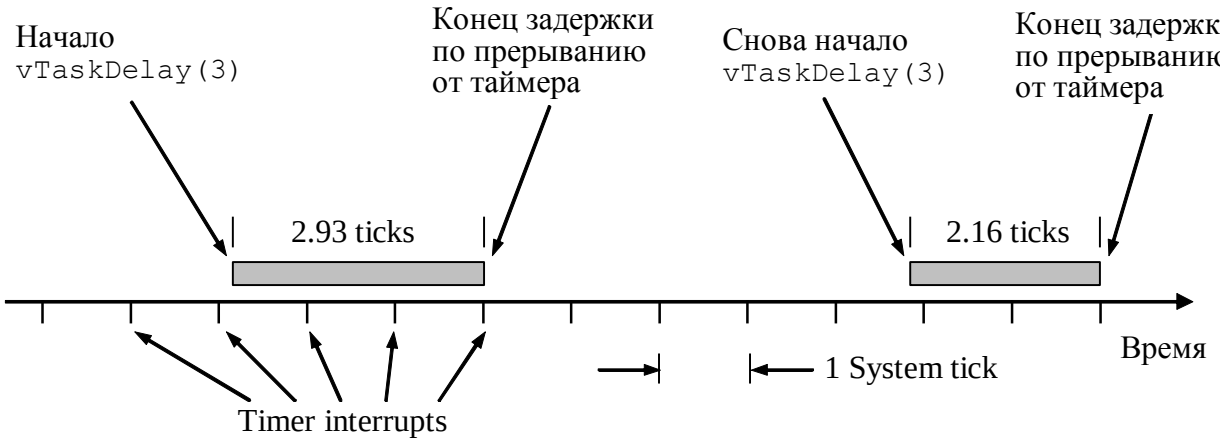


Рис. 2.16

На рис. 6.5 поясняется образование временной задержки при вызовах `taskDelay()` с параметром, равным 3. Первый вызов `taskDelay()` происходит сразу же после поступления запроса прерывания от таймера. Задача, вы-

звавшая `taskDelay()`, блокируется на время до наступления следующего (третьего по счёту) прерывания. Будет ли при этом задача переключена на исполнение, зависит от того, каким приоритетом она обладает, потому что на этот момент времени процессор может выполнять более высокоприоритетное задание. Следующий вызов `taskDelay(3)` происходит незадолго до очередного запроса прерывания таймера и поэтому временная задержка в итоге оказывается короче предыдущей.

Каким образом RTOS опознает характеристики (параметры) установленного в аппаратуре таймера? Как и все операционные системы RTOS на низком уровне аппаратно зависимы (зависят, в частности, от используемого CPU). Поэтому разработчики RTOS предоставляют программный продукт (*Board Support Package* – BSP), содержащий драйверы для поддержки взаимодействия с аппаратными компонентами, а также инструкции и правила для аппаратно ориентированного программирования.

Как обеспечить необходимую точность временных отсчетов? Один из путей – это сокращение временных интервалов между тиками системного таймера. Другой – применение дополнительного таймера с повышенной точностью отсчётов времени.

Примеры использования таймеров

Большинство RTOS имеют достаточно развитые средства для работы с системным таймером. Например, с их помощью можно задавать ограничения на время ожидания сообщения в очереди или в почтовом ящике, на время ожидания семафора и т.д.

Нередко функции RTOS могут вызываться через определяемые заранее отрезки времени. Такие функции могут вызываться либо из процедуры обработки прерываний таймера, либо из специального высокоприоритетного задания. Зависит это от того, как RTOS работает с таймером.

В листинге 2.21 представлена программа управления приёмопередатчиком, который время от времени то включается, то выключается. Процедура выключения проста – выключается источник питания. Процесс включения выполняется за несколько шагов. Вначале должен быть включён источник питания на базовом радиооборудовании. Через 12 миллисекунд устанавливается частота настройки. Затем через 3 миллисекунды система включает передатчик или приёмник, после чего устройство становится готовым к работе.

Листинг 2.21

```
/* Message queue for radio task. */
extern MSG_Q_ID queueRadio;

/* Timer for turning the radio on. */
static WDOG_ID wdRadio;
```

```
static int iFrequency;    /* Frequency to use. */

void vSetFrequency(int i);
void vTurnOnTxorRx(int i);

void vRadioControlTask(void)
{
  #define MAX_MSG 20
  char a_chMsg[MAX_MSG+1]; /* Message sent to this task. */

  enum
  {
    RADIO_OFF,
    RADIO_STARTING,
    RADIO_TX_ON,
    RADIO_RX_ON,
  } eRadioState;      /* State of the radio */
  eRadioState = RADIO_OFF;

  /* Create the radio timer */
  wdRadio = wdCreate();

  while(TRUE)
  {
    /* Find out what to do next */
    msgQReceive(queueRadio, a_chMsg, MAX_MSG, WAIT_FOREVER);

    /* The first character of the message tells this task what
       the message is. */
    switch(a_chMsg[0])
    {
      case 'T':
      case 'R':
        /* Someone wants to turn on the transmitter */
        if(eRadioState == RADIO_OFF)
         {
           !! Turn on power to the radio hardware.

           eRadioState = RADIO_STARTING;
           /* Get the frequency from the message */
           iFrequency = *(int *)a_chMsg[1];

           !! Store what needs doing when the radio is on.
           /* Make the next step 12 milliseconds from now. */
           wdStart (wdRadio, 12, vSetFrequency,
           (int) a_chMsg[0])
         }
        else
```

```
            { !! Handle error. Can't turn radio on if not off break;
              break;
            }
          case 'K':
             /* The radio is ready. */
            eRadioState = RADIO_TX_ON;
            !! Do whatever we want to do with the radio break;
            break;
          case 'L':
             /* The radio is ready. */
            eRadioState = RADIO_RX_ON;
            !! Do whatever we want to do with the radio break;
            break;
          case 'X':
             /* Someone wants to turn off the radio. */
            if (eRadioState == RADIO_TX_ON ||
                  eRadioState == RADIO_RX_ON)
             {
               !! Turn off power to the radio hardware.
               eRadioState = RADIO_OFF;
             }
            else
               !! Handle error. Can't turn radio off if not on break;
            break;
          .
          .
          .
          default:
            !! Deal with the error of a bad message break;
            break;

      }
    }
}

void vSetFrequency(int i)
{
  !! Set radio frequency to iFrequency;

  /* Turn on the transmitter 3 milliseconds from now. */
  wdStart(wdRadio, 3, vTurnOnTxorRx,i)
}

void vTurnOnTxorRx(int i)
{
  if(i == (int) 'T')
   {
    !! Turn on the transmitter

    /* Tell the task that the radio is ready to go. */
```

```
        msgQSend(queueRadio, "K",1,
            WAIT_FOREVER, MSG_PRI_NORMAL);
      }
     else
      {
        !! Turn on the receiver

        /* Tell the task that the radio is ready to go. */
        msgQSend(queueRadio, "L",1,
          WAIT_FOREVER, MSG_PRI_NORMAL);
      }
  }
```

Используемые в программе функции принадлежат конкретной OS – уже упоминавшейся системе *VxWorks*. Одна из этих функций – `wdStart()` – стартует таймер. Поясним ее работу. Функция `wdStart()` получает четыре параметра и используется для вызова через заданное в миллисекундах время (второй параметр) другой функции, имя и параметр которой содержат соответственно третий параметр и четвёртый параметры `wdStart()`. Задача `vRadioControlTask()` управляет приемопередатчиком, для чего использует символьные сообщения. При этом сообщения `'T'` и `'R'` сигнализируют о том, что какое-то из заданий потребовало включить передатчик или приёмник. Получив одно из них `vRadioControlTask()` включает источник питания и затем стартует таймер с помощью вызова `wdStart()` для запуска таймера. Когда таймер отсчитает 12 мс, RTOS вызывает `vSetFrequency()` с параметром , значение которого соответствует четвертому параметру `wdStart()`. Функция `vSetFrequency()` программирует частоту и вновь запускает таймер для отработки задержки перед вызовом `vTurnOnTxorRx()`. С помощью `vTurnOnTxorRx()` RTOS включает передатчик или приёмник (в зависимости от того, что требуется) и возвращает сообщение о том, что устройство готово к работе и может быть использовано.

2.9.4 События (*Events*)

Важной для любой RTOS является информация о происходящих в системе процессах. Действенным для получения этой информации является механизм *событий*. По существу события сродни флагам, которые одна задача может установить (сбросить) для того, чтобы другая задача могла предпринять соответствующие состоянию флагов действия.

Назовем основные особенности событий в RTOS.

- При ожидании одного события блокированными могут быть более чем одно задание. После регистрации события блокировка снимается со всех

ожидающих события заданий, но запускается то из них, которое обладает наибольшим приоритетом.

- Как правило, события в RTOS объединяются в **группы**. Задачи могут ожидать нескольких событий из числа тех, которые принадлежат одной группе.
- После фиксации события и деблокирования ожидавших его задач, различные RTOS по-разному сбрасывают флаг события. В некоторых RTOS флаг сбрасывается автоматически. В других предполагается, что делать это должно задание с помощью соответствующего программного кода. Не важно как, но *флаг должен быть сброшен*. В противном случае восприятие следующего события становится невозможной.

В качестве примера на использование событий возьмем сканер штрих-кода (листинг 2.22). Когда пользователь приводит сканер в действие (например, нажатием клавиши), то стартует задача, которая включает сканерный механизм и распознает принимаемый от него штрих-код. При старте сканера генерируется запрос прерывания. Процедура обработки прерывания устанавливает флаг, сигнализирующий о произошедшем событии. Флаг опознает ожидающая его задача и выполняет предписанные ей действия.

Есть отличие в трактовке понятия «событие» применительно к RTOS и при обычном его использовании в операционных системах, основной задачей которых является предоставление ресурсов. Используемые в листинге 2.22 функции взяты из арсенала системы AMX.

Листинг 2.22

```
/* Handle for the trigger group of events.*/
AMXID amxidTrigger;

/* Constants for use in the group.*/
#define TRIGGER_MASK   0×0001
#define TRIGGER_SET    0×0001
#define TRIGGER_RESET  0×0000
#define KEY_MASK       0×0002
#define KEY_SET        0×0002
#define KEY_RESET      0×0000

void main(void)
{
  ⋮
  /* Create an event group with
      the trigger and keyboard events reset. */

  ajevcre (&amxidTrigger, 0, ''EVTR'');
  ⋮
```

```
}

void interrupt vTriggerISR(void)
{
  /* The user pulled the trigger. Set the event. */
  ajevsig(amxidTrigger, TRIGGER_MASK, TRIGGER_SET);
}

void interrupt vKeyISR(void)
{
  /* The user pressed a key. Set the event. */
  ajevsig(amxidTrigger, KEY_MASK, KEY_SET);

  !! Figure out which key the user pressed and store that value
}

void vScanTask(void)
{
   ⋮
   while(TRUE)
   {
     /* Wait for user to pull the trigger. */
     ajevwat(amxidTrigger, TRIGGER_MASK, TRIGGER_SET
             WAIT_FOR ANY, WAIT_FOREVER);

     /* Reset the trigger event */
     ajevsig(amxidTrigger, TRIGGER_MASK, TRIGGER_RESET);

     !! Turn on the scanner hardware and look for a scan.
     ⋮
     !! When the scan has been found, turn off the scanner.
   }
}

void vRadioTask(void)
{
   ⋮
   while(TRUE)
   { /* Wait for user to pull the trigger or press a key. */
     ajevwat(amxidTrigger, TRIGGER_MASK | KEY_MASK,
          TRIGGER_SET | KEY_SET, WAIT_FOR ANY, WAIT_FOREVER);

     /* Reset the key event. (The trigger event will be reset
         by the vScanTask() ) */
     ajevsig(amxidTrigger, KEY_MASK, KEY_RESET);

     !! Turn on the radio.
```

```
        ⋮
        !! When data has been sent, turn off the radio.
    }
}
```

Функция

```
ajevcre(AMXID *p_amxidGroup, unsigned int uValueInit,
                        char *p_chTag)
```

создает группу из 16 событий, обработчиком которых является функция, на которую указывает `p_amxidGroup`. Передаваемое в функцию состояние флагов событий (сброшен или установлен) определяется параметром `uValueInit`. В AMX группа именуется четырьмя символами с указателем `p_chTag`. Это позволяет заданиям находить системные объекты по их имени в тех случаях, когда они не имеют доступа к обработчику событий.

Для сброса и установки флага события в группе, обозначенной идентификатором `amxidGroup`, используется функция

```
ajevsig(AMXID amxidGroup, unsigned int uMask,
            unsigned int uValueNew).
```

Параметр `uMask` отмечает флаги, которые должны быть установлены или сброшены и в дальнейшем оставаться неизменными; параметр `uValueNew` – это флаги, которым могут присваиваться новые значения в соответствии с произошедшими событиями.

Время ожидания одного или нескольких событий из группы `amxidGroup` устанавливается с помощью функции

```
ajevwat(AMXID amxidGroup, unsigned int uMask,
    unsigned int uValue,int iMatch, long lTimeout).
```

Параметр `uMask` обозначает события, которые задание ожидает, а `uValue` – какие из флагов должны быть сброшены, а какие установлены. Параметр `iMatch` определяет условие, при котором задание деблокируется (при наличии всех событий в группе или при наличии только одного из них). Наконец, `lTimeout` – это время ожидания отмеченных параметром `uMask` событий.

AMX имеет также функции, с помощью которых группы событий можно удалить, например, тогда, когда они длительное время не использовались. Есть функции определения текущего состояния флагов всех событий, а также тех флагов из группы событий, которые на текущий момент времени привели к деблокированию ожидающих этих событий задач.

2.9.5. Сопоставление средств межзадачных коммуникаций

Мы рассмотрели несколько методов обмена данными между разными задачами в среде RTOS: очереди, почту, семафоры и события. Попробуем сравнить или сопоставить эти методы.

- Использование семафоров – это наиболее простой и быстрый метод. Однако с его помощью невозможно передавать объемную информацию. Фактически они позволяют делать только однобитовые сообщения с целью уведомления об «открытии» (освобождении) семафора.
- Использование событий сложнее, но времени занимает не многим больше, чем семафоры. Выгоднее применять механизм событий потому, что задание может ожидать события в группе из нескольких, в то время как ожидаемым может быть только один семафор.
- Очереди более предпочтительны при передаче значительных объемов информации от одного задания к другому и даже в том случае, когда приходится ожидать доступа к очереди. Недостаток их в том, что (1) передача сообщения в очередь и его извлечение из неё требуют большего процессорного времени и (2) больше вероятность возникновения непредсказуемых проблем, связанных с разделением данных. Почтовые пересылки и нити с точки зрения перечисленного мало чем отличаются от очередей.

2.9.6. События и переключение заданий в операционной системе µC/OS-II

Операционная система µC/OS-II разрабатывалась с целью применения во встроенных системах на базе микроконтроллеров фирмы *Advanced RISC Machines* (ARM), *Ltd*. Межзадачное взаимодействие и синхронизация заданий в этой ОС основаны на происходящих в системе событиях. Однако в отличие от событий, описанных в разделе 2.9.4, к событиям в ОС µC/OS-II отнесены те, которые обусловлены межзадачным взаимодействием с применением разделяемых ресурсов. Такими событиями могут быть освобождение ресурса или доступность сервиса. В частности они связаны с обращениями заданий к семафорам, очередям, почтовым ящикам. ОС обнаруживает события по вызовам соответствующих функций, к которым относятся функции

- захвата и освобождения семафоров `OSSemPend()` и `OSSemPost()`,
- захвата и освобождения почтовых ящиков `OSMBoxPend()` и `OSMBoxPost()`,
- захвата и освобождения очередей `POSQend()` и `OSQPost()`.

Для регистрации событий используются сигналы, фиксируемые в контрольных блоках ECB (*Event Control Block*, листинг 2.23).

Листинг 2.23

```
typedef struct {
void *OSEventPtr;    /* Ptr to message or queue structure */
INT8U OSEventTbl[OS_EVENT_TBL_SIZE]; /* Wait list for event
                                        to occur */
INT16U OSEventCnt;   /* Count (when event is a semaphore) */
INT8U  OSEventType;  /* Event type */
INT8U  OSEventGrp;   /* Group for wait list */
} OS_EVENT;
```

Как и контрольные блоки заданий (TCB) ECB являются структурами – структурами (в ОС µC/OS-II структурами типа `OS_EVENT`), где содержится необходимая для обработки поступившего сигнала информация (рис. 2.17). Номер ECB в списке зависит от того, какое событие в нём регистрируется (обращение к семафору, почтовому ящику или к очереди), и соответствует типу и номеру того события, которое нужно заданию. Общее число ECB устанавливается посредством определения

`#define OS_MAX_EVENTS`

и задаётся на стадии конфигурации ОС при вызове функции **`OSInit()`**, которая создаёт список свободных ECBs. Когда семафоры, почтовые ящики и очереди создаются, соответствующий ECB исключается из списка свободных и инициализируется. ECBs не могут быть возвращены в список свободных, т.к. семафоры, почтовые ящики и очереди не могут быть удалены.

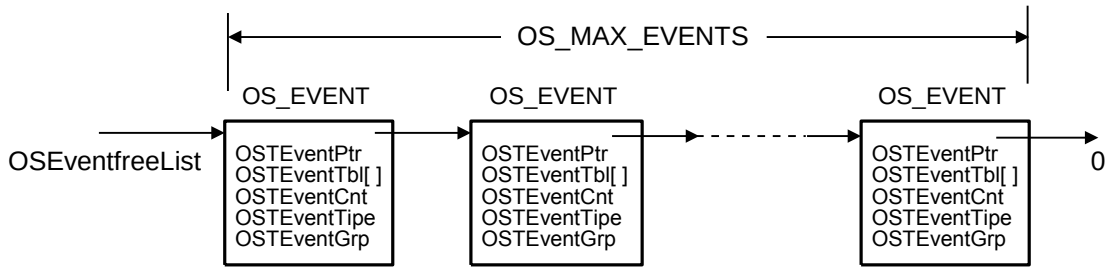


Рис. 2.17

Взаимодействие заданий между собой и с процедурами обработки прерываний показано на рис. 2.18.

Рис. 2.18

По отношению к ECBs применимы следующие операции:

- инициализация (функция **`OSEventWaitListInit()`**)
- перевод задания в состояние готовности (ф-я **`OSEventTaskRdy()`**),
- перевод задания в состояние ожидания события или истечения таймаута (функция **`OSEventTaskWait()`**),

– перевод задания в состояние готовности по истечении таймаута при ожидании события (функция **`OSEventTO()`**).

В µC/OS-II состояние задания меняется по схеме, представленной на рис. 2.19. Ранее (рис. 2.7) была рассмотрена схема с тремя состояниями. **Теперь к ним добавилось** состояние ожидания «WAITING» и состояние «DORMANT – *бездействующий, спящий*». Состояние WAITING соответствуют блокированному заданию на рис. 2.7. Переход в состояние DORMANT вообще исключает задание из числа работающих.

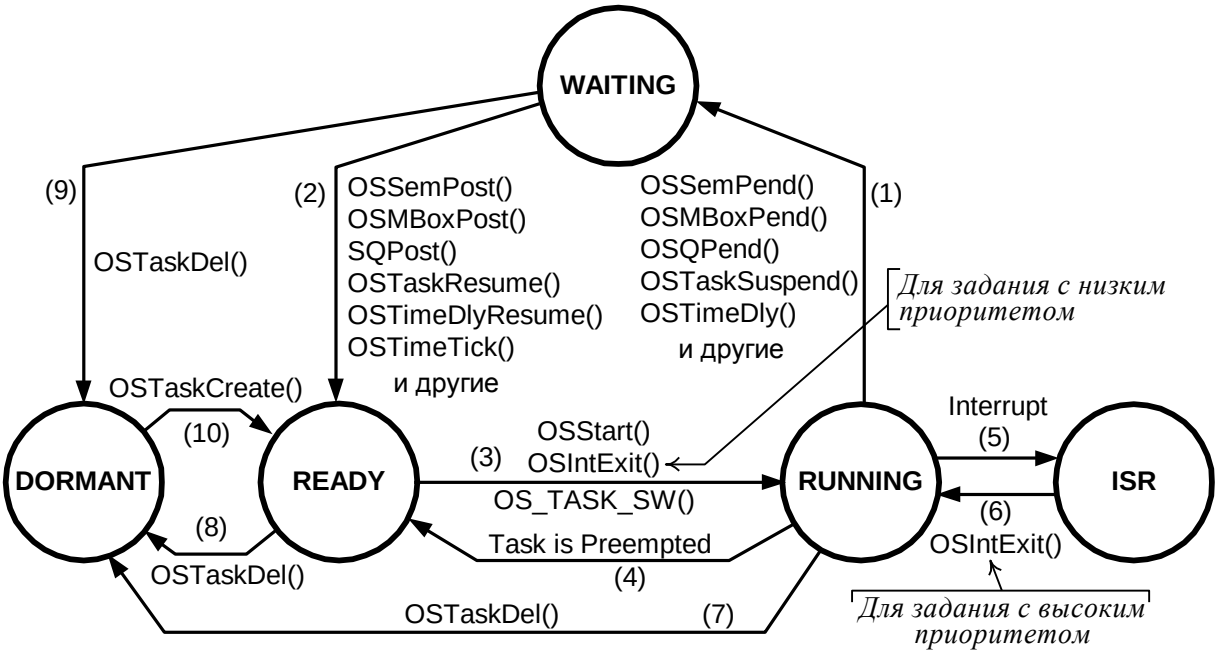


Рис. 2.19

Исполняемое задание (RUNNING) может быть остановлено (переведено в состояние WAITING), если для него нет доступа к необходимому ресурсу, а также при поступлении запросов прерывания и при передаче управления другому более высокоприоритетному заданию. Выполнение задания может быть прервано (путь 1),

– если при вызове функций `OSSemPend()`, `OSMBoxPend()` и `OSQPend()` была обнаружена недоступность семафора, почтового ящика или очереди,
– если задание блокировало само себя посредством вызова `OSTaskSuspend()`,
– если выполнение задания задерживается на заданное число системных тиков с помощью функции `OSTimeDly()`.

Из состояния ожидания задание может быть переведено в состояние готовности (READY – путь 2)

– при освобождении ресурса другими заданиями (функции `OSSemPost()`, `OSMBoxPost()` и `OSQPost()`),

– при вызове функции `OSTaskResume()`, возобновляющей выполнение ранее блокированного с помощью функции `OSTaskSuspend()` задания,
– по истечении таймаута, заданного функцией `OSTimeDly()`, с помощью вызова `OSTimeDlyResume()`,
– при посылке с помощью функции `OSTimeTick()` µC/OS-II сигнала об истечении заданного интервала времени, что позволяет ОС перевести задание в состояние готовности и вновь планировать его выполнение; функция `OSTimeTick()` вызывается процедурой обработки прерываний от системного таймера или высокоприоритетным заданием.

В состоянии готовности могу пребывать несколько заданий. Планировщик передаёт управление тому из них, которое обладает наиболее высоким приоритетом. Работа планировщика завершается вызовом функции переключения контекста `OS_Task_SW()` – путь 3. Если окажется, что через некоторое время при исполнении спланированного задания требуется передать управлению более приоритетному заданию, то текущее задание переводится в состояние READY (путь 4).

Важной задачей RTOS является обслуживание запросов прерываний IRQ. Приоритет процедур обработки запросов IRQ, всегда более высок, чем приоритет любого из заданий. Поэтому исполнение задания может быть прервано сигналом `Interrupt` (путь 5). После завершения работы ISR с помощью вызова `OSIntExit()` управление может быть вновь передано прерванному заданию (путь 6), если на момент окончания ISR это задание обладает наиболее высоким приоритетом и нет необходимости в перепланировании. Если после вызова ISR планировщик перевёл задание в состояние READY, то в состояние исполняемого оно переводится также в ответ на вызов `OSIntExit()` – путь 4.

Задание может быть удалено, если это потребовалось самому заданию при его исполнении (путь 7), а также другими заданиями и системой (пути 8 и 9). Во всех этих случаях вызывается функция `OSTaskDel()`.

Наконец, в состояние готовности задание переводится функцией `OSTaskCreate()`, а в состояние исполняемого – функцией `OSTaskStart()` на стадии инициализации ОС.

Пример действий ОС при обращении к разделяемому ресурсу представлен в Приложении 1, в котором показана последовательность вызовов функций

ожидания события `OSEventTaskWait()`,
планировщика `OSSched()`,
переключения контекста `OS_TASK_SW()`,
отработки таймаута `OSEventTO()` *и*
приведения задания в состояние готовности `OSEventTaskRdy()`

при работе с функциями `OSMBoxPend()` и `OSMboxPost()`.

2.10. Управление памятью

Средства управления памятью имеются в большинстве систем реального времени. Хотя некоторые из них подобны библиотечным С-функциям типа `malloc()` и `free()`, создатели систем реального времени предпочитают избегать использования последних, поскольку работают они достаточно медленно, а время исполнения непредсказуемо. Предпочтение отдается функциям, с помощью которых создаются и освобождаются фиксированные по размеру буферы.

В среде многозадачных RTOS возможно создание областей памяти (*set up pools*), каждая из которых состоит из нескольких буферов. Такие пулы памяти компонуются, как правило, из буферов фиксированного размера. В этом случае распределением памяти под буферы занимаются функции типа

```
void *getbuf(unsigned int uPoolId,unsigned int uTimeout);
void *regbuf(unsigned int uPoolId).
```

Обе функции используются для выделения памяти и каждая из них возвращает указатель на буфер. Если память под буфер не может быть выделена, то задание, вызвавшее `getbuf()`, блокируется. Функция `regbuf()` в этом случае возвращает NULL указатель. Место расположения буфера в памяти определяется по параметру `uPoolId`. На случай отсутствия свободной памяти в текущий момент времени, предусматривается ожидание выделения памяти в течение времени, которое задается параметром `uTimeout`. Поскольку все буферы по размерам одинаковы, то выделенная под буфер память определяется общим размером пула, которому данный буфер принадлежит. Поэтому при работе с функциями `getbuf()` и `regbuf()` считается, что размер буфера известен.

Выделенная под буфер память освобождается вызовом функции

```
void *relbuf(unsigned int uPoolId, void *p_vBuffer).
```

Заметим, что проверка соответствия указателя `p_vBuffer` и идентификатора пула `uPoolId` не выполняется. Если этого не делает и ваше задание, то последствия могут быть катастрофическими.

Для RTOS типична ситуация, когда не известно, где и как используется свободная память. В момент старта любая встроенная система не контролируется операционной системой. При старте RTOS не имеет средств для определения, какая память свободна и какую её часть будет занимать разработанное вами программное обеспечение. Многозадачная система управляет памятью (в том числе и памятью, выделяемой под буферы) только после её инициализации, но ей необходимо сообщить, где и какая память имеется. Это делается с помощью специальной функции, например, с помощью `init_mem_pool()`:

```
int init_mem_pool{
      unsigned int uPoolId,
      void     *p_vMemory,
      unsigned int uBufSize,
```

```
                    unsigned int uBufCount,
                    unsigned int uPoolType
                ).
```

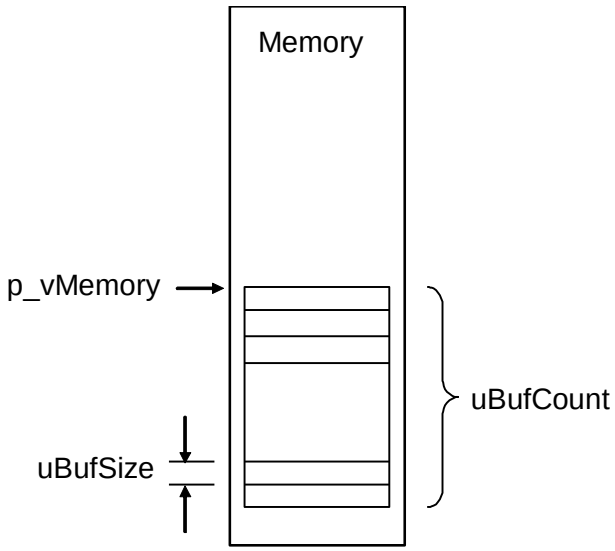


Рис. 2.20

Для идентификации пула используется `uPoolId`. Впоследствии при необходимости `uPoolId` будет использован функциями `getbuf()`, `regbuf()` и `relbuf()` так, как это описано раньше. Параметр `p_vMemory` служит для указания на отводимую под пул область памяти; `uBufSize` и `uBufCount` определяют размер каждого из буферов и их число. Параметр `uPoolType` указывает на то, будут ли буферы использоваться заданиями или процедурами обработки прерываний.

Управление памятью специфично для каждой системы реального времени, поэтому сказанное выше – это лишь попытка оттенить какие-то общие правила. В листинге 2.24 представлен пример работы с памятью двух заданий. Задание `vPrintFormatTask()` форматирует данные и результат сохраняет в буфере, сформированном как двумерный массив `a_lines[MAX_LINES][MAX_LINE_LENGTH]` с максимальной длиной строки `MAX_LINE_LENGTH` и общим количеством строк `MAX_LINES`. Задание `vPrintOutputTask()` извлекает данные из буфера и выводит их на печать. Для инициализации буфера используется функция `init_mem_pool()`, которой в качестве параметров помимо размерности и места расположения буфера в памяти передаются его идентификатор `LINE_POOL` и тип `TASK_POOL`.

Листинг 2.24

```
#define LINE_POOL           1
#define MAX_LINE_LENGTH     40
#define MAX_LINES           80

static char a_lines[MAX_LINES][MAX_LINE_LENGTH];

void main(void)
{
  ⋮
  init_mem_pool(LINE_POOL, a_lines,
         MAX_LINES, MAX_LINE_LENGTH, TASK_POOL);
}
void vPrintFormatTask(void)
```

```
{
  char *p_chLine;
  ⋮

/* Format lines and send them to the vPrintOutputTask() */

  p_chLine = getbuf(LINE_POOL, WAIT_FOREVER);
  sprintf(p_chLine, "INVENTORY REPORT");
  sndmsg(PRINT_MBOX, p_chLine, PRIORITY_NORMAL);
  p_chLine = getbuf(LINE_POOL, WAIT_FOREVER);
  sprintf(p_chLine, "DATA: %02/%02/%02",
           iMonth, iDay, iYear%100);
  sndmsg(PRINT_MBOX, p_chLine, PRIORITY_NORMAL);
  p_chLine = getbuf(LINE_POOL, WAIT_FOREVER);
  sprintf(p_chLine, "TIME: %02/%02", iHouer, iMinute);
  sndmsg(PRINT_MBOX, p_chLine, PRIORITY_NORMAL);
  ⋮
}

void vPrintOutputTask(void)
{
  char *p_chLine;
  while(TRUE)
    {
       /* Wait for a line to come in. */
       p_chLine = rcvmsg(PRINT_MBOX, WAIT_FOREVER);
       !! Do wait is needed to send the line to the printer

       /* Free the buffer back to the pool. */
       relbuf(LINE_POOL, p_chLine);
    }
}
```

Важно, чтобы вывод текущей информации на печать не задерживался процессом форматирования данных. Поэтому задание `vPrintFormatTask()` имеет более высокий приоритет по сравнению с заданием `vPrintOutputTask()`. Для занесения данных в пул буферов используется ОС сервис – функции `sprintf()` и `sndmsg()`.

Отформатированные и ожидающие вывода на печать строки данных хранятся в пуле буферов размером по 40 байт. Не всегда на печать выводятся все 40 символов, из-за чего отведенная под буферы память используется не полностью. Это «расплата» за относительно высокую скорость передачи данных через буферы фиксированного размера. Компромиссным решение в такой ситуации является использование нескольких пулов с различными размерами буферов и их разумное распределение по разным задачам.

2.11. Процедуры обработки прерываний и окружение RTOS

Отличительной особенностью процедур обработки прерываний является то, что их блокирование недопустимо. Избежать блокирования ISR можно путём их вызова и обмена данными с УВВ через посредство операционной системы. Поэтому процедуры обработки прерываний в большинстве RTOS должны строиться в соответствии со следующими двумя правилами.

Правило 1. ISR *не должны вызывать те функции* RTOS*, которые могут их блокировать.* Следовательно, процедуры обработки прерываний не могут пользоваться семафорами, обращаться к очередям или почтовым ящикам ожидать событий и т.д. Если ISR вызывает функцию RTOS и в результате этого вызова блокируется, то помимо процедуры обработки прерывания блокированной окажется и та задача, которая выполнялась на момент поступления запроса на прерывание и даже в том случае, если эта задача имеет высокий приоритет. Но всякая ISR должна предпринимать действия для сброса установленного обслуживаемым устройством запроса для того, чтобы создать условия для восприятия очередного запроса.

Правило 2. *ISR не должна вызывать те функции RTOS, которые могут вызвать переключение задач в той ситуации, когда RTOS не известно, что на стадии исполнения находится процедура обработки прерывания, а не какая-либо другая задача.* Это означает, что ISR не может передавать данные ожидающей их задаче по почте или через очередь, работать с семафорами и устанавливать флаги событий без уведомления об этом операционной системы. Если в процедуре обработки прерывания это правило нарушено, то RTOS может перенаправить управление от ISR (которую она принимает за обычное задание) к другой задаче и завершение ISR будет задержано на длительное время. При этом блокированными могут оказаться не только прерывания с более низким приоритетом, но, возможно, и все прерывания без исключения.

Для иллюстрации этих правил приведём несколько примеров.

Правило 1: *работаем без блокирующих функций*.

Листинг 2.25 возвращает нас к задаче мониторинга температуры, измеряемой двумя датчиками (листинг 2.14). На этот раз задача, выполняющая функцию контроля, и процедура обработки прерывания совместно используют массив `iTemperatures[2]`, в котором сохраняются значения измеренных температур. Разделение этого ресурса между ISR `vReadTemperanures()` и заданием `vTaskTestTemperature()` предлагается обеспечить с помощью семафора. Однако *работать такая программа не будет*, т.к. написана она с нарушением Правила 1.

Действительно, может случиться так, что исполнение `vTaskTestTemperature()` будет прервано процедурой обработки прерывания в то время, когда задание, получив семафор, закрыло его. ISR, в свою очередь,

вызовет `GetSemaphore()`, после чего RTOS обнаружит, что семафор уже получен и закрыт. Это вызовет остановку процедуры обработки прерывания вслед за остановкой `vTaskTestTemperature()`, которая была остановлена запросом на прерывание. В возникшей таким образом тупиковой ситуации RTOS также остановит свою работу.

Листинг 2.25

```
static int iTemperatures[2];

void interrupt vReadTemperanures(void)
{
  GetSemaphore(SEMAPHORE_TEMPERATURE); /***NOT_ALLOWED***/
  iTemperatures[0] = !! read in value from hardware;
  iTemperatures[1] = !! read in value from hardware;
  GiveSemaphore(SEMAPHORE_TEMPERATURE);
}

void vTaskTestTemperature(void)
{
  int iTemp0, iTemp1;

  while(TRUE)
  {
    GetSemaphore(SEMAPHORE_TEMPERATURE);
    iTemp0 = iTemperatures[0];
    iTemp1 = iTemperatures[1];
    GiveSemaphore(SEMAPHORE_TEMPERATURE);
    if(iTemp0 != iTemp1)
       !! Set off howling alarm;
  }
}
```

Примечание

Некоторые RTOS *могут вести себя по-другому. Например, при вызове процедурой обработки прерывания функции* `GetSemaphore()`, RTOS *отмечает, что* `vTaskTestTemperature()` *уже получила семафор и по этой причине продолжает её исполнение, приостанавливая* ISR. *Это исключает возникновение тупиковой ситуации, но возникает задержка в передаче семафора ISR и увеличивается время реакции на запрос прерывания.*

Проблемы не исключаются и в том случае, когда ISR выполняется на фоне какой-то другой задачи. Если `vTaskTestTemperature()` получила семафор, а после этого при исполнении другой задачи возник запрос на прерывание, то при обращении процедуры обработки прерывания к семафору она обнаружит, что семафор блокирован. Это приведет к лишению процессорного времени не только затребовавшей семафор ISR, но и всех процедур обработки прерываний с меньшим приоритетом и той задачи, исполнение которой было останов-

лено прерыванием. И так будет до тех пор, пока `vTaskTestTemperature()` не получит процессор, что позволит ей освободить семафор.

В арсенале некоторых RTOS имеются функции, использование которых позволяет избежать блокирования. Например, такими функциями являются те, которые возвращают состояние семафора. Применение неблокирующих функций поясняет листинг 2.26, в котором использована неблокирующая функция `sc_qpost()`, принадлежащая RTOS *VRTX* (*trademark of Microtec Research, Incorporated*). Необходимые для `vMainTask()` данные передаются с помощью очереди сообщений. Если при этом очередь оказывается заполненной, то `sc_qpost()` возвращает сообщение об ошибке. Недостатком представленной в листинге 2.26 программы является то, что при обращении к заполненной очереди возможен пропуск некоторых измеренных значений температур. Однако этот недостаток вообще свойственен механизму очередей.

Листинг 2.26

```
/* Queue for temperature. */
int iQueueTemp;

void interrupt vReadTemperanures(void)
{
  int aTemperatures[2];
  int iError;

  /* Get a new temperatures. */
  aTemperatures[0] = !! read in value from hardware;
  aTemperatures[1] = !! read in value from hardware;

  /* Add the temperatures to a queue. */
  sc_qpost(iQueueTemp,
     (char *)((aTemperatures[0]<<16 | aTemperatures[1]),
        &iError);
}

void vMainTask(void)
{
  long int lTemps; /* 32 bit: the same size as a pointer. */
  int aTemperatures[2];
  int iError;

  while(TRUE)
  {
    lTemps = (long) sc_pend(iQueueTemp, WAIT_FOREVER,
             sizeof(int), &iError);
    iTemperatures[0] = (int)(lTemps>>16);
    iTemperatures[1] = (int)(lTemps & 0x0000ffff);
    if(iTemperatures[0] != iTemperatures[1])
```

```
            !! Set off howling alarm;
        }
    }
```

Правило 2: ISR *не вызывает блокирующих функций без предварительного уведомления* RTOS *о своей работе.*

Пусть процедура обработки прерывания работает так, как это показывает рис. 2.21. С течением времени процессор переходит от одного задания к другому. При этом может случиться так, что ISR прервёт выполнение задания TaskLow с низким приоритетом и во время своего исполнения обратится к RTOS, чтобы послать почтовое сообщение. Причём сделает это в соответствии с Правилом 1 – не вызывая функций, способных блокировать ISR. По окончании обработки запроса прерывания RTOS переходит либо к задаче TaskLow, прерванной ISR, либо к задаче TaskHigh с более высоким приоритетом.

Рис. 2.21

Однако на самом деле может быть так, как показано на рис. 2.22. Если почтовый ящик на момент обращения к нему ISR был занят высокоприоритетным заданием, RTOS перейдет к исполнению задачи TaskHigh. Системе ничего не известно о том, что до переключения на TaskHigh выполнялась процедура обработки прерывания. Поэтому RTOS относит ISR к низкоприоритетному заданию и вместо того, чтобы продолжить процедуру обработки прерывания, переключается на задание TaskHigh, которое имеет наибольший приоритет среди ожидающих исполнения задач. Как следствие, процедура обработки прерывания не сможет быть завершена даже после выполнения TaskHigh.

В системах реального времени используются разные методы, позволяющие исключить возникновение подобных ситуаций. Один из них состоит в том, что RTOS перехватывает все запросы прерывания и только после их анализа вызывает нужную ISR (рис. 2.23). Если, например, при исполнении процедуры обработки прерывания потребуется обращение к почтовому ящику, то после окончания почтовых операций RTOS продолжит исполнение ISR независимо от

того, какое из заданий было разблокировано по окончании операций с почтовым ящиком. После возврата из прерывания управление снова переходит к RTOS и её планировщик определяет, какому из заданий передать процессор.

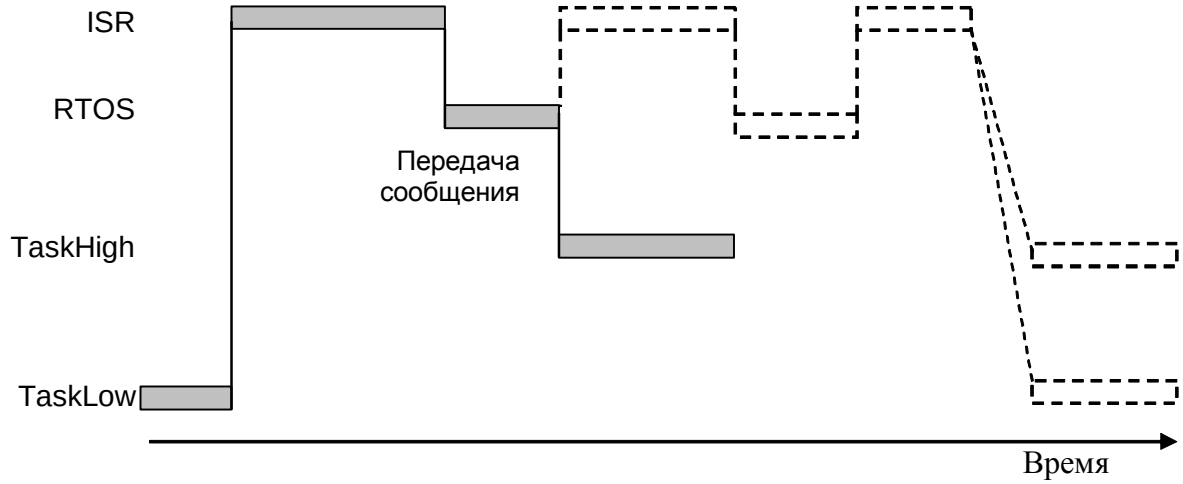


Рис. 2.22

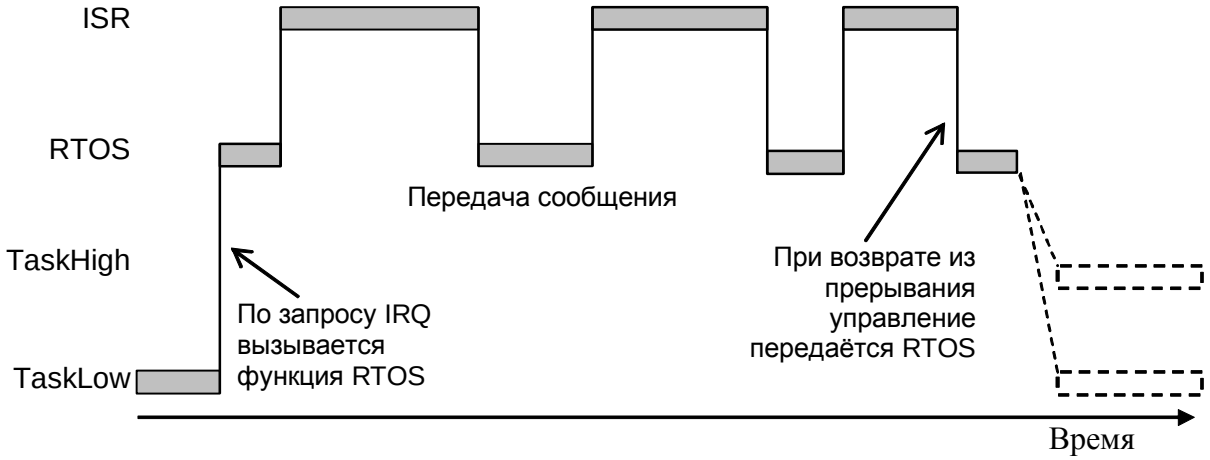


Рис. 2.23

процедуре обработки прерывания потребуется обращение к почтовому ящику, то после окончания почтовых операций RTOS продолжит исполнение ISR независимо от того, какое из заданий было разблокировано по окончании операций с почтовым ящиком. После возврата из прерывания управление снова переходит к RTOS и ее планировщик определяет, какому из заданий передать процессор.

RTOS, использующие этот метод, обладают набором средств, с помощью которых определяется стартовый адрес ISR и проверяется его соответствие типу запрашиваемого устройством прерывания.

Альтернативным методом является предоставление функций, которые могут быть вызваны из ISR для уведомления RTOS о том, какой из обработчи-

ков прерывания находится в стадии исполнения (рис. 2.24). После вызова такой функции RTOS сама способна отличить ISR от обычного задания и после операций обращения ISR к почтовому ящику или к другим средствам межзадачного взаимодействия возвращается к продолжению процедуры обработки прерывания, несмотря на готовность других задач к исполнению. ISR завершается вызовом специальной функции RTOS, которая, в свою очередь, вызывает планировщик для принятия решения о том, какой из задач предоставить процессор. Существенно то, что планировщик не вмешивается в ход выполнения ISR. И все-таки нужно отметить, что такой план хорош, если процедура обработки прерывания вызывает функции оповещения RTOS в подходящее для этого время.

Рис. 2.24

Есть ещё и третий механизм: RTOS предоставляет отдельный набор функций, работать с которыми могут только ISR. Например, наряду с `OsSemPost()` может существовать `OsISRSemPost()`, вызывать которую может только процедура обработки прерывания. `OsISRSemPost()` делает то же, что и `OsSemPost()`, но всегда обеспечивает возврат в ту ISR, которая её вызвала. Передача управления после завершения ISR происходит посредством планировщика.

Правило 2 и вложенные прерывания

В системе, допускающей вложенные прерывания, запрос прерывания с более высоким приоритетом может остановить обработку запроса с низким приоритетом. Если высокоприоритетная ISR делает вызовы функций RTOS, то низкоприоритетная ISR должна уведомлять RTOS о вызвавшем её низкоприоритетном запросе. Когда высокоприоритетная ISR закончила свою работу, планировщик может запустить некоторую другую задачу раньше, чем завершится обработка запроса прерывания с низким приоритетом (рис. 2.25). Из этого следует, что к планированию заданий можно приступать только после того, как все выполняемые на текущий момент времени ISR будут завершены.

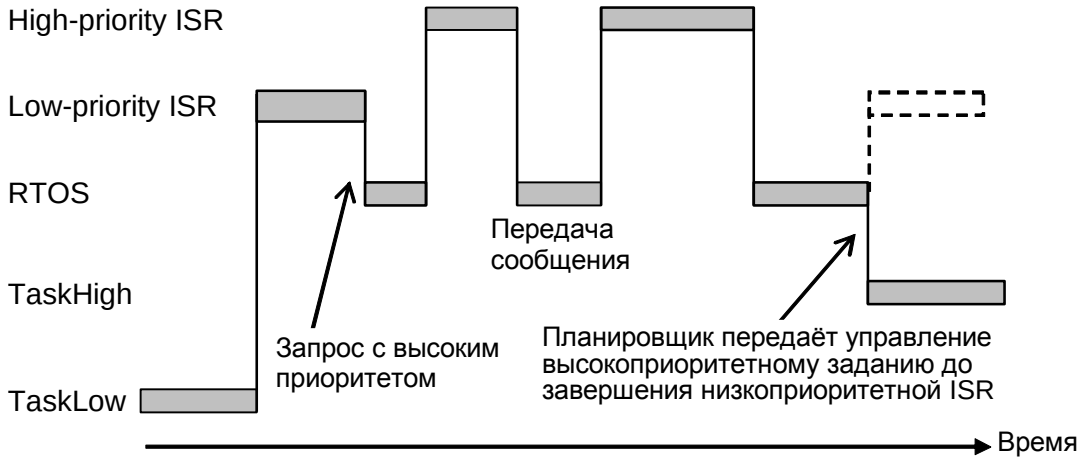


Рис. 2.25

Некоторые выводы

- Задача, выходя на связь с другой задачей, должна координировать свою активность и обеспечивать бесконфликтное разделение данных. Большинство RTOS предоставляют для этого очереди сообщений, почтовые ящики, нити (каналы). Специфика применения этих средств межзадачного взаимодействия зависит от используемой RTOS.
- Существует общее правило, согласно которому при передаче блоков данных через очередь, почтовый ящик или канал от одного задания к другому передаётся указатель на буфер очереди, почтового ящика или канала.
- Система реального времени имеет системный таймер. Периодические запросы прерывания от таймера позволяют RTOS отслеживать ход системного времени и производить временное регламентирование предоставляемых системой сервисных операций (функций). Функции таймера наиболее широко используются
  - для блокирования задачей самой себя по истечении заданного числа системных тиков,
  - для определения таймаутов при ожидании семафора, очереди и т.д.,
  - для планирования запуска некоторых функций с задержкой на заданное число системных тиков.
- События – это однобитовые флаги, используемые как сигналы, передаваемые от одной задачи к другой. События могут быть объединены в группы, а задания могут ожидать одного или некоторой комбинации событий в группе.
- Многие RTOS допускают применение стандартных функций `malloc()` и `free()` для управления памятью. Однако они работают медленно и время их исполнения непредсказуемо. Для задач реального времени более подходит управление памятью с использованием пулов буферов фиксированного размера.
- Процедуры обработки прерываний в среде RTOS должны удовлетворять двум условиям:

- ISR не должны вызывать функций, которые могут быть причиной их блокирования.
- Если использование таких функций необходимо, то ISR не должны вызывать их без уведомления RTOS о том, что они были вызваны на стадии выполнения ISR. В операционных системах реального времени применяются разные механизмы использования такой информации.

3. Операционная система для микроконтроллеров фирмы *Advanced RISC Machines*, *Ltd*

В данном разделе поясняется процесс конструирования RTOS для микроконтроллеров фирмы *Advanced RISC Machines* (ARM), *Ltd*. При генерации ОС используется программное обеспечение операционной системы µC/OS-II, а также низкоуровневые функции библиотеки µHAL (библиотеки *Micro*-HAL – ARM *Hardware Abstraction Layer*). Функции библиотеки µHAL предназначены для работы с аппаратной частью целевой системы, а также представляют сервис, позволяющий получать информацию о функционировании системы и упрощающий процесс разработки RTOS и её приложений.

3.1. Инициализация операционной системы

Рассмотрим простую ОС с двумя процессами `Task1` и `Task2`. Точка входа в операционную систему – это начало функции `main()` (листинг 3.1).

Листинг 3.1

```
/* Main function. */
int main(int argc, char **argv)
{
 char Id1 = '1';
 char Id2 = '2';
  ARMTargetInit();        /* do target (µHAL based ARM
                             System)initialization */
  OSInit();               /* needed by µC/OS */
  OSTimeSet(0);
  Sem1 = OSSemCreate(1); /* create the semaphores */
  Sem2 = OSSemCreate(1);
   /* create the tasks in µC/OS and assign priority to them */
  OSTaskCreate(Task1,(void *)&Id1,
                     (void *)&Stack1[STACKSIZE-1],1);
  OSTaskCreate(Task2,(void *)&Id2,
                     (void *)&Stack2[STACKSIZE-1],2);
  ARMTargetStart();       /* Start the µHAL based ARM system.
                             System running */
  OSStart();
   /* never reached */
}
```

Перед вызовом `main()` должна быть проинициализирована аппаратная составляющая целевой системы. Для этого (например, для установки карты памяти) используются соответствующие µHAL функции. После этого ОС инициализирует сама себя и приступает к работе.

При выполнение `main()` включают следующие действия:

1. Сначала из `main()` вызывается µHAL функция `ARMTargetInit()` (листинг 3.2 – находится в модуле `os_cpu_c.c`), которая производит необходимые установки. После этого `OSInit()` инициализирует разделы операционной системы.

Листинг 3.2

```
#define BUILD_DATE "Date: " __DATE__ "\n"
  /* Initialize an ARM Target board */
void ARMTargetInit(void)
{
   /* ---- Tell the world who we are ---- */
   uHALr_printf("uCOS-II Running on a");
   #if defined(EBSA285)
     uHALr_printf("n EBSA-285 (21285 evaluation board)\n");
   #elif defined(BRUTUS)
     uHALr_printf(" Brutus (SA-1100 verification plaform)\n");
   #else
     uHALr_printf("%s\n, ucosii_banner") ;
   #endif
   uHALr_printf(uHAL_VERSION_STRING);
   uHALr_printf("\n") ;
   uHALr_printf(BUILD_DATE);
   uHALr_printf("\n") ;
   #ifdef DEBUG
      uHALr_printf("Initialising target\n");
   #endif

   uHALr_ResetMMU();    /* disable the MMU */
   ARMDisableInt();     /* disable interrupts (IRQs) */

  /*---- soft vectors -------*/
   #ifdef DEBUG
      uHALr_printf("Setting up soft vectors\n");
   #endif

  /* Define pre & post-process routines for Interrupt */
   uHALir_DefineIRQ(IrqStart, IrqFinish, (PrVoid) 0);
   uHALr_InitInterrupts();

   #ifdef DEBUG
      uHALr_printf("Timer init\n");
   #endif

   uHALr_InitTimers();

   #ifdef DEBUG
      uHALr_printf("targetInit() complete\n");
   #endif
}              /* targetInit */
```

2. `main()` создаёт нужные потоки (в данном случае задания `Task1` и `Task2`) с заданными приоритетами. Каждый поток имеет свой стек, который организуется на работу с регистрами переднего плана. Первичный регистр PC (*Program Counter*) содержит адрес потока управления.
3. На завершающем этапе `main()` стартует операционную систему путём вызова µHAL-специфицированного кода `ARMTargetStart()` (листинг 3.3) и µC/OS-II функции `OSStart()`. `ARMTargetStart()` запускает системный таймер, который через каждую миллисекунду генерирует сигналы запросов прерывания. При поступлении запроса прерывания ОС проверяет, требуется или нет переключение контекста. `OSStart()` находит среди готовых к испол-

нению задание с наиболее высоким приоритетом (в данном случае `Task1`) и запускает его, загружая его регистры из стека задания.

Листинг 3.3

```
void ARMTargetStart(void)  /* start the ARM target running */
{  #ifdef DEBUG
       uHALr_printf("Starting target\n") ;
   #endif
   /* request the system timer */
   if(uHALr_RequestSystemTimer( (PrHandler)OSTimeTick,
                (const unsigned char *)"uCOS-II") <= 0)
      uHALr_printf("Timer/IRQ busy\n");
   uHALr_InstallSystemTimer();  /* Start system timer & enable
                                   the interrupt. */
}
```

3.2. Переключение контекста

Переключение заданий в µC/OS-II происходит тогда, когда удовлетворяются следующие условия:

– поток делает системный вызов (например, ожидает семафора или таймера);
– принимается прерывание от таймера и готово к исполнению задание с более высоким у текущего задания приоритетом.

Листинги 3.4 и 3.5 показывают процесс переключения двух заданий `Task1` и `Task2`, Планировщик работает с учётом доступности для заданий семафоров `Sem1` и `Sem2` и установленных таймаутов. При этом семафоры используются для передачи сигналов.

Листинг 3.3. Переключение контекста `Task1`

```
void Task1(void *i)
{  uint Reply;
   for (;;)
    {
      OSSemPend(Sem2, 0, &Reply); /* wait for the semaphore */
      uHALr_printf("1+");
      OSTimeDly(100);             /* wait a short while */
      uHALr_printf("1-");
      OSSemPost(Sem1);             /* signal the semaphore */
    }
}
```

Листинг 3.5. Переключение контекста `Task2`

```
void Task2(void *i)
{ uint Reply;
  for (;;)
    {
      OSSemPend(Sem1, 0, &Reply); /* wait for the semaphore */
      uHALr_printf("[");
      OSTimeDly(1000);            /* wait a short while */
```

```
            uHALr_printf("2]");
            OSSemPost(Sem2);              /* signal the semaphore */
        }
    }
```

Переключение контекстов `Task1` и `Task2` происходит следующим образом:

1. При первоначальном вызове `OSSemPend()` из `Task1` переключения контекста не происходит, но семафор `Sem2` получает значение «занят». `Task1` выводит символы `1+` перед тем, как произвести вызов `OSTimeDly()`.
2. `OSTimeDly()` приостанавливает выполнение `Task1` и µC/OS-II запускает `Task2`. Происходит переключение контекста, при котором контекст текущего потока (все его регистры, включая слово состояния CPRS) в принадлежащем потоку `Task1` стеке и восстанавливается контекст задания с наиболее высоким приоритетом, в данном случае задания `Task2`.
3. Заданию `Task2` при вызове `OSSemPend()` не нужно ждать семафора `Sem1` и оно выводит символ `[`. После этого вызывается `OSTimeDly()`, что приводит к приостановке выполнения `Task2` до истечения установленного для него таймаута.
4. С этого момента задание `Task1` ещё не может выполняться, т.к. остался неотработанным его таймаут – менее длительный, чем у `Task2`. До истечения этого таймаута текст, выведенный функцией `ARMTargetInit()` из `Task1` (листинг 3.6), остаётся без изменений. В режиме «*standalone*» для вывода используется последовательный порт, а при отладке (режим «*semihosted*») – консоль.

Листинг 3.6. Первоначальный вывод

```
uCOS-II Running on an Integrator board
uHAL v1.1:
Date: Aug 12 1999
1+[
```

5. Когда происходит запрос прерывания от таймера, включается в работу управляющая обработкой прерываний функция `uHALir_TrapIRQ()` из µHAL-набора (здесь не показана). Она сохраняет текущий набор регистров в стеке и проверяет, есть ли необходимость запуска процедуры, стартующей обработку прерывания. Для µC/OS-II такой процедурой является `IrqStart()` (листинг 3.7).

Листинг 3.7

```
extern int OSIntNesting;
/* This is what µCOS does at the start of an IRQ */
void IrqStart(void)
{
   OSIntNesting++;      /* increment nesting counter */
}
```

6. `IrqStart()` наращивает счётчик вложенных запросов прерывания `OSIntNesting`. Этот счётчик используется планировщиком µC/OS-II. Выполняются действия по запуску процедуры обработки прерывания. Завершает их вызов `OSTimeTick()`, обращенный к таймеру ОС. `OSTimeTick()` декрементирует таймер, принадлежащий задержанному по таймауту потоку, и тем самым приближает время его дальнейшего исполнения. В данном случае первым истекает время задержки процесса `Task1`, и он первый будет спланирован на исполнение. По завершении работы µHAL программы, управляющей обработкой прерываний, µC/OS-II проверяет, завершена ли процедура обработки текущего запроса. В листинге 3.8 показана процедура завершения обработки прерывания `IrqFinish()`.

Листинг 3.8

```
/* This is what uCOS does at the end of an IRQ */
extern int OSIntExit(void);
extern void IRQContext(void); /* post Dispatch IRQ
                                 processing */
PrVoid IrqFinish(void)
{
  /* call exit routine - return TRUE
     if a context switch is needed */
  if (OSIntExit() == TRUE)
        return (IRQContext);
  return ((PrVoid) 0);
}
```

7. `IrqFinish()` вызывает `OSIntExit()` с целью определения необходимости контекстного переключения. Если переключение контекста необходимо, `IrqFinish()` возвращает стартовый адрес функции `IRQContext()` (специфицированная в µC/OS-II функция контекстного переключения). Как правило, процедура обработки прерываний восстанавливает сохранённые в стеке регистры и производит возврат из прерывания. Поскольку `IrqFinish()` возвращает адрес, то с помощью этой функции вызывается µC/OS-II процедура переключения контекста с использованием сохранённых в стеке значений регистров.
8. Когда исполнение `Task1` продолжится, на вывод поступят очередные два символа `1-` и будет установлен сигнал-семафор `Sem1` (путём инкремента значения). Затем `Task1` переходит к ожиданию семафора `Sem2`, который на данное время находится в сброшенном процессом `Task1` состоянии. Процесс `Task1` блокируется.
9. Когда для исполнения выбирается `Task2` (после истечения установленной для этого процесса задержки), выводятся два символа `2]` и процессу `Task1` посылается установленный сигнал-семафор `Sem2`. Становится возможным выполнение `Task1`. Когда это случается, вступает в работу планировщик. Разобравшись в приоритетах планировщик блокирует `Task2` и передаёт управление процессу `Task1`. В дальнейшем эти операции повторяются циклически:

```
uHAL v1.1:
Date: Aug 12 1999
1+[1-2]1+[1-2]1+[1-2]1+[1-2]1+[1-2]1+[1-2]1+[1-
```

Дополнительная информация

Исходные программные тексты для RTOS *с тремя заданиями, использующими* «*почтовые ящики*»*, находятся в Приложении 2. Приложение 3 содержит тексты дополнительных* (*не включённых в Приложение 2*) *функций* µC/OS-II.

Замечание

Концепция процессов (*заданий*) *используется очень давно. Хотя их можно рассматривать с точки зрения управляющих потоков, есть особенности в применении термина* «*поток*» (thread)*. Он относится к управляющим потокам внутри одного процесса. С точки зрения взаимодействия процессов все потоки одного процесса имеют общие глобальные переменные* (*то есть поточной модели свойственно использование общей памяти*)*. Однако потокам требуется синхронизация доступа к глобальным данным.*

В файле «eCos_2_примера.doc» *приведены примеры синхронизации потоков в операционной системе* eCos (*Embedded Configurable Operating System*).

4. Микропроцессоры компании ARM Limited

Архитектуру ARM (*Advanced* RISC *Machine*) процессоров отличает то, что на её основе создаются как простые встроенные микросистемы с низким энергопотреблением, так и сложные многофункциональные аппаратно-программные комплексы, работающие под управлением таких операционных систем, как JavaOS, Linux, Microsoft WindowsCE и др.

В первом случае системы используют ресурсы микропроцессорного ядра, интегрированного в один кристалл вместе с разнообразными устройствами ввода/вывода (включая простейшие параллельные и последовательные цифровые порты и сложные, поддерживающие различные сетевые протоколы каналы связи). Сюда следует отнести, прежде всего, коммуникационные микроконтроллеры. Обычно их используют в тех приложениях, в которых конечный пользователь никогда не добавляет программное обеспечение к системе.

Возможности ARM-систем могут быть расширены добавлением внешних сопроцессоров, в частности, сопроцессоров с плавающей точкой и управляющего сопроцессора CP15. Последний предоставляет дополнительные средства для конфигурации и управления системой, включая средства защиты памяти, что важно для многозадачных приложений.

Во втором случае микропроцессор имеет кэшированное макроядро, способное выполнять операции с многоразрядными данными и работающее совместно с встроенной системой управления (встроенный сопроцессор CP15). Такое макроядро ориентировано на использование в «открытых» системах, для которых необходимы полное управление виртуальной памятью и развитая защита собственной памяти.

Всё это позволяет создавать масштабируемые системы с возможностью выбора необходимых для конкретных приложений микропроцессорных ресурсов. Этому служит то, что ARM-компоненты могут быть выполнены как на отдельных микросхемах (например, отдельно CPU с устройствами ввода/вывода, отдельно сопроцессор управления памятью и сопроцессор с плавающей точкой), так и быть интегрированными в одну микросхему. Кроме этого, ARM-процессоры, работающие обычно с 32-разрядными командами, могут управляться укороченными до 16-ти разрядов Thumb-инструкциями, что повышает компактность создаваемого для них программного кода.

ARM МП являются RISC-процессорами, поэтому

- операции в них выполняются только с данными, находящимися в регистрах,
- отдельно выполняются команды загрузки/сохранения регистров,
- режимы адресации используют только значения, взятые из регистров или из определённых в командах полей,
- все команды имеют фиксированный размер (32 бита при обычной работе и 16 бит при работе в Thumb-режиме).

В состав ARM-макроядра помимо собственно ядра процессора входят расширения отладки: контроллер TAP (*Test Access Port*) сканирования (*boundary scan*) и внутрисхемный эмулятор (*ICEBreaker*). Аппаратные расширения отладки облегчают разработку пользовательского прикладного программного обеспечения, операционных систем и самих аппаратных средств. Аппаратные расширения отладки позволяют останавливать ядро или при выборке заданной команды (в контрольной точке), или при обращении к данным (в точке просмотра), или асинхронно – по запросу отладки. В этих точках через JTAG последовательный интерфейс может быть исследовано внутреннее состояние ядра, находящегося в состоянии отладки, и внешние признаки состояния системы. По завершении процесса отладки состояния ядра и системы могут быть восстановлены, а выполнение программы продолжено.

Режим отладки устанавливается или запросом по одной из линий внешнего интерфейса отладки, или запросом от внутреннего функционального блока ICEBreaker, состоящего из двух модулей установки контрольных точек (*watchpoint*), работающих в реальном масштабе времени с регистрами состояния и управления ядра и обеспечивающих поддержку встроенной отладки ядра. ICEBreaker программируется в последовательном режиме с использованием контроллера TAP – средства управления работой цепочек сканирования (*Scan Chain* 0, 1 и 2) через последовательный интерфейс JTAG.

Некоторые ARM-микропроцессоры для расширения возможностей отладки имеют встроенный сопроцессор CP14.

Интерфейс отладки основан на архитектуре, описанной в стандарте IEEE Std. 1149.1-1990 «*Standard Test Access Port* (TAP) *and Boundary-Scan Architecture*».

4.1. Структура микропроцессоров с ядрами ARM7TDMI и ARM9TDMI

Изначально ARM-процессоры были рассчитаны на работу под управлением 32-разрядных инструкций. Со временем к ним были добавлены Thumb-инструкции, имеющие укороченный 16-разрядный формат. Технология Thumb была встроена в ядро ARM7 в 1995 году и развивалась в последующие годы. В настоящее время она встроена практически во все пользующиеся спросом ARM-процессоры.

Thumb-технология является одной из отличительных черт ARM-архитектуры. Однако наряду с этим ей свойственно многое из того, что характеризует тенденцию развития микропроцессорной техники. ARM-процессоры обладают встроенным полнофункциональным механизмом управления памятью (*Memory Management Unit* – MMU), имеют возможность использования виртуальной памяти и средств защиты памяти, эффективно реагируют на события и обладают быстрым механизмом контекстного переключения, что особен-

но важно при работе в многозадачной среде. Рассмотрим эти особенности, обратившись к некоторым конкретным архитектурным решениям.

Для начала рассмотрим ядро ARM7. Это ядро после внедрения в него средств поддержки Thumb-инструкций получило типовое обозначение ARM7TDMI и за достаточно короткое время было лицензировано большим количеством фирм-изготовителей всевозможного оборудования,

Процессор ARM7TDMI – 32-разрядный RISC процессор с 3-уровневым конвейером, сформированным вокруг банка из 37 32-разрядных регистров, в который входят 6 регистров состояния. Процессор оснащён встроенным умножителем 32×8 и 32-разрядным многорегистровым циклическим устройством сдвига (рис. 4.1). Пять независимых встроенных шин (PC шина, шина инкремента, шина ALU и A- и B-шины) обеспечивают, при выполнении команд высокую степень параллелизма. Как видно из рисунка, в блоке конвейера процессора имеется декомпрессор команд Thumb, преобразующий 16-разрядный формат к виду, приемлемому для работы вычислительных устройств ядра процессора.

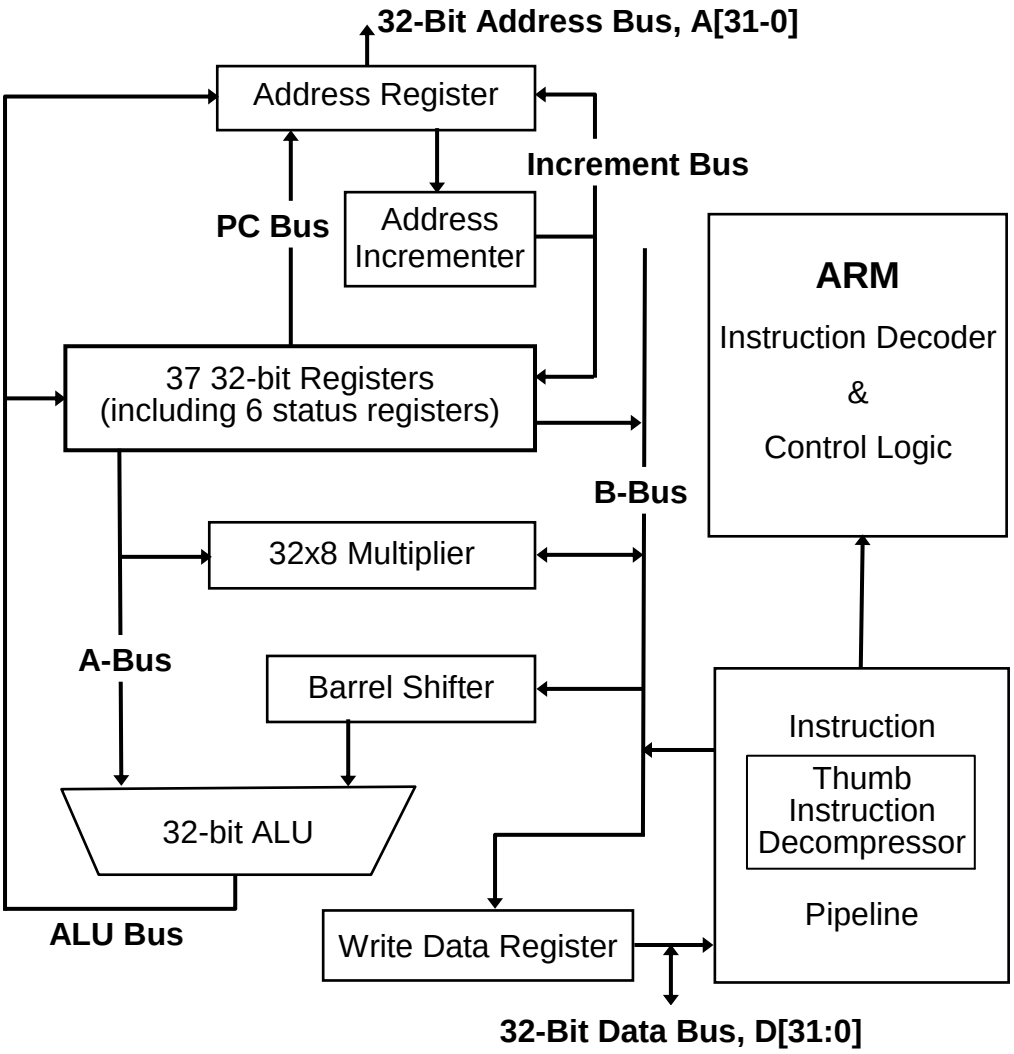


Рис. 4.1. Блок-схема ядра ARM7TDMI

Интерфейс с памятью у процессора ARM7TDMI организуется следующими основными элементами:

- 32-разрядной шиной адреса;
- 32-разрядной двунаправленной шиной передачи данных (*32-Bit Data Bus* – D[31:0], рис. 4.1). В процессорах разных версий шина D[31:0] может быть разделена на две отдельные однонаправленные шины данных DIN[31:0] и DOUT[31:0], через которые перемещаются команды и данные. Данные могут иметь формат слова, полуслова или байта;
- устройством управления, определяющим, например, форматом перемещаемых данных и направление их передачи и, кроме того, уровень приоритета.

На основе ядра ARM7TDMI были созданы макроядра ARM710T и ARM720T, ориентированное на персональные информационные устройства и Internet применения. Макроядра оснащены встроенным MMU, имеют возможность использования виртуальной памяти.

Возможности базирующихся на ARM7TDMI систем могут быть расширены за счёт добавления до 16 внешних сопроцессоров.

В семействе 32-разрядных ARM RISC процессоров ARM9TDMI сохранены основные ставшие промышленным стандартом преимущества ARM7TDMI. В настоящее время в семейство ARM9TDMI входит процессорное ядро ARM9TDMI и оснащённые кэш-памятью процессорные макроядра ARM920T и ARM940T.

Для всех процессоров семейства ARM9TDMI общим является:

- высокопроизводительный 32-разрядный ARM RISC механизм,
- гарвардская архитектура с раздельными шинами команд и данных,
- пятиуровневый конвейер операций,
- 16-разрядная Thumb система команд,
- встроенные возможности EmbeddedICE JTAG отладки программного обеспечения,
- совместимость с низковольтными CMOS технологиями,
- 100% совместимость двоичных кодов пользователя с ARM7TDMI,
- возможность интеграции класса «система-на-кристалле» со встроенным тестированием в процессе производства.

Процессор ARM920T (рис. 4.2) имеет два внутренних сопроцессора:

- CP14 – для программного доступа к используемым при отладке коммуникационным каналам;
- CP15 – для управления системой. CP15 имеет дополнительные регистры, используемые для конфигурации и управления кэш-памятью, устройствами управления памятью программы (*Instruction* MMU) и памятью данных (*Data* MMU), средствами защиты памяти, режимом синхронизации, способом представления данных в памяти и др. В частности, указанный на рис. 4.2 регистр C13 используется для хранения идентификатора ID теку-

щего (исполняемого) процесса. Значение идентификатора ID используется при выделении процессу памяти и при вычислении исполнительного адреса в ходе исполнения процесса.

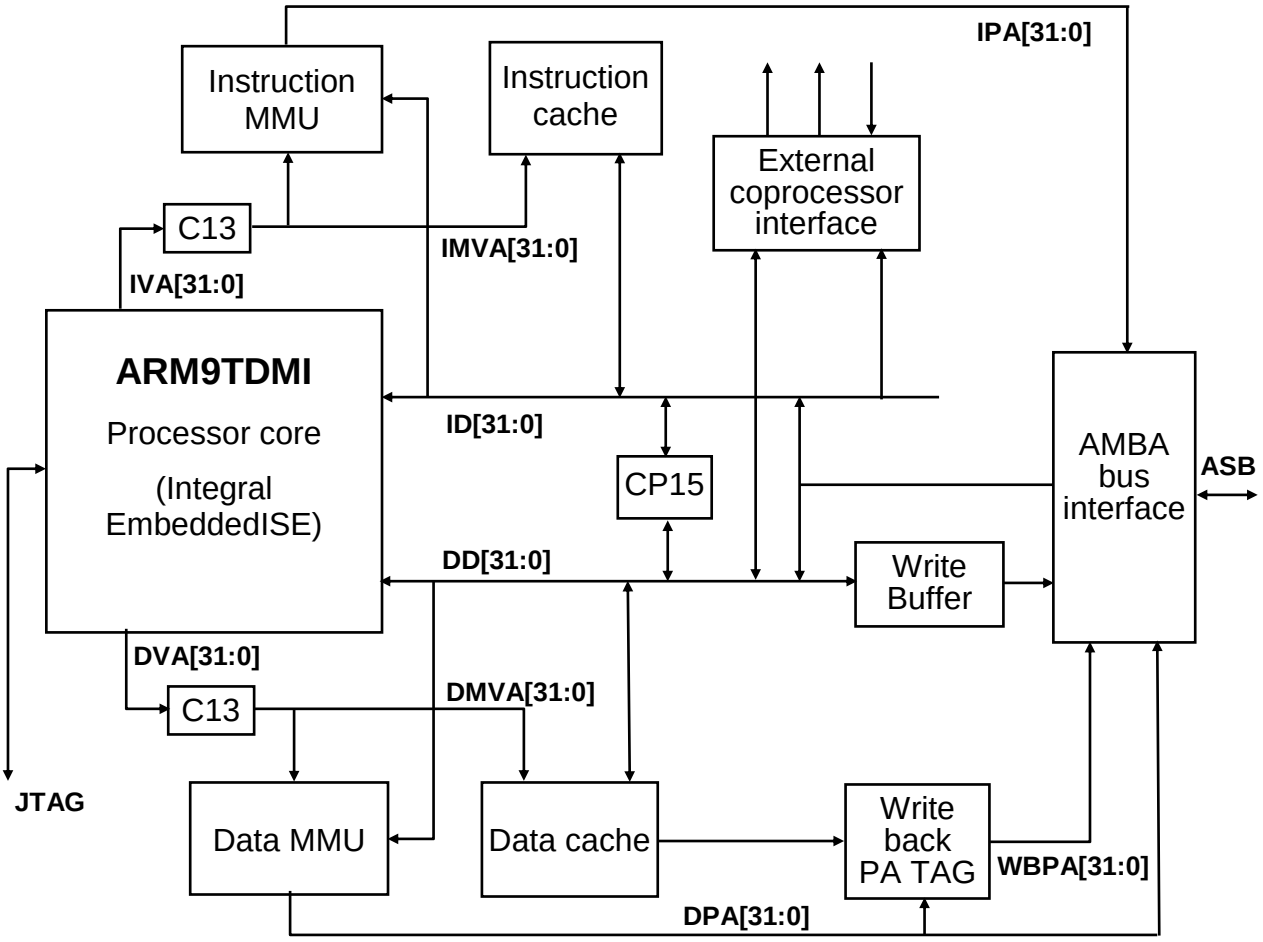


Рис. 4.2. Блок-схема макроядра ARM920T

Имеется интерфейс для подключения к системе дополнительных (кроме встроенных в микросхему МП) сопроцессоров, например, сопроцессоров с плавающей точкой.

Обращения к памяти могут быть кэшированы и буферизованы. Имеются кэш-память программы (*Instruction Cache*), кэш-память данных (*Data Cache*), буфер записи (*Write Buffer*) и память для хранения тегов физических адресов PA (*Write Back PA TAG*). Последняя используется для синхронизации кэш-памяти данных с целью обеспечения соответствия содержимого кэша данных и оперативной памяти. Используемый здесь термин «*Write Back*» характеризует способ обновления кэш-памяти[1]. Имеются средства конфигурации перечисленных выше устройств.

Внешние устройства подсоединяются к макроядру с помощью схемы согласования шин AMBA-ASB (*Advanced Microcontroller Bus Architecture* – AMBA, *Advanced System Bus* – ASB), которая является интерфейсом с внешней

[1] Работе с памятью в ARM системах посвящён разделах 4.3.

двунаправленной магистралью ASB, способной обеспечить работу нескольких ведущих.

В многозадачной среде исполняемые процессы различаются по идентификаторам ID, хранящимся в регистре C13 сопроцессора CP15. В соответствии с идентификатором каждому процессу выделяется отдельное адресное пространство. При исполнении программы ядро генерирует виртуальные адреса инструкций IVA[31:0] и виртуальные адреса данных DVA[31:0], которые преобразуются в модифицированные виртуальные адреса IVMA[31:0] и DVMA[31:0] соответственно. Модифицированные виртуальные адреса преобразуются в физические IPA[31:0] и DPA[31:0] устройствами управления адресацией команд и данных – *Instruction* MMU и *Data* MMU. При этом память разбивается на страницы и для каждой страницы устанавливаются размер, права доступа, возможность применения к ней кэширования и буферизации. Преобразование виртуального адреса в модифицированный виртуальный адрес выполняется с использованием идентификаторов процессов ID.

Ядро принимает инструкции по однонаправленной шине ID[31:0]. Для обмена данными используется двунаправленная шина DD[31:0].

Данные в память передаются через буфер записи. При этом, если адресуемая страница памяти является кэшируемой, то модифицируется содержимое кэш-памяти данных, а тэг модифицированной линии кэша заносится в память тэгов физических адресов. В дальнейшем интерфейс с внешней памятью переносит данные из буфера данных в оперативную память системы, используя физические адреса, взятые из памяти PA тэгов и передаваемые по шине WBPA[31:0] (*Write Back Physical Address* – WBPA).

На рис. 4.3 в расширенном виде представлена структура ядра одного из процессоров семейства ARM9x – ядро процессора ARM9E-S. Вычислительные устройства – умножитель (*Multiplier*), аккумулятор (ACC), сдвигатель (*Shifter*) и арифметическо-логическое устройство (ALU) – связаны с многопортовым регистровым блоком (*Register Bank*) через систему мультиплексоров. При этом счётчик команд рассматривается как один из регистров регистрового блока. Признаки производимых операций сохраняются в регистре состояния программы (*Program Status Register* – PSR).

Исполнительные адреса инструкций берутся из регистра адреса команд (*Instruction Address Register* – IAReg) и передаются по шине IA[31:1]. В транслируемых по шине IA[31:1] адресах отсутствует младший разряд. Это объясняется тем, что команды процессора имеют 16 или 32 разряда и инструкции в памяти выравниваются по словам, из-за чего для адресации инструкций младший (нулевой) бит не требуется. Нарушение такого выравнивания недопустимо. Источниками адреса команд являются счётчик команд (один из регистров регистрового блока), инкрементер адреса команды (*Incrementer*), адрес перехода/вызова (*Imm*) или возврата и вектор исключительных ситуаций (*Exception vectors*). Исключительные ситуации (или просто исключения) возникают на фоне внутренних и внешних событий. К внешним событиям относятся прерывания.

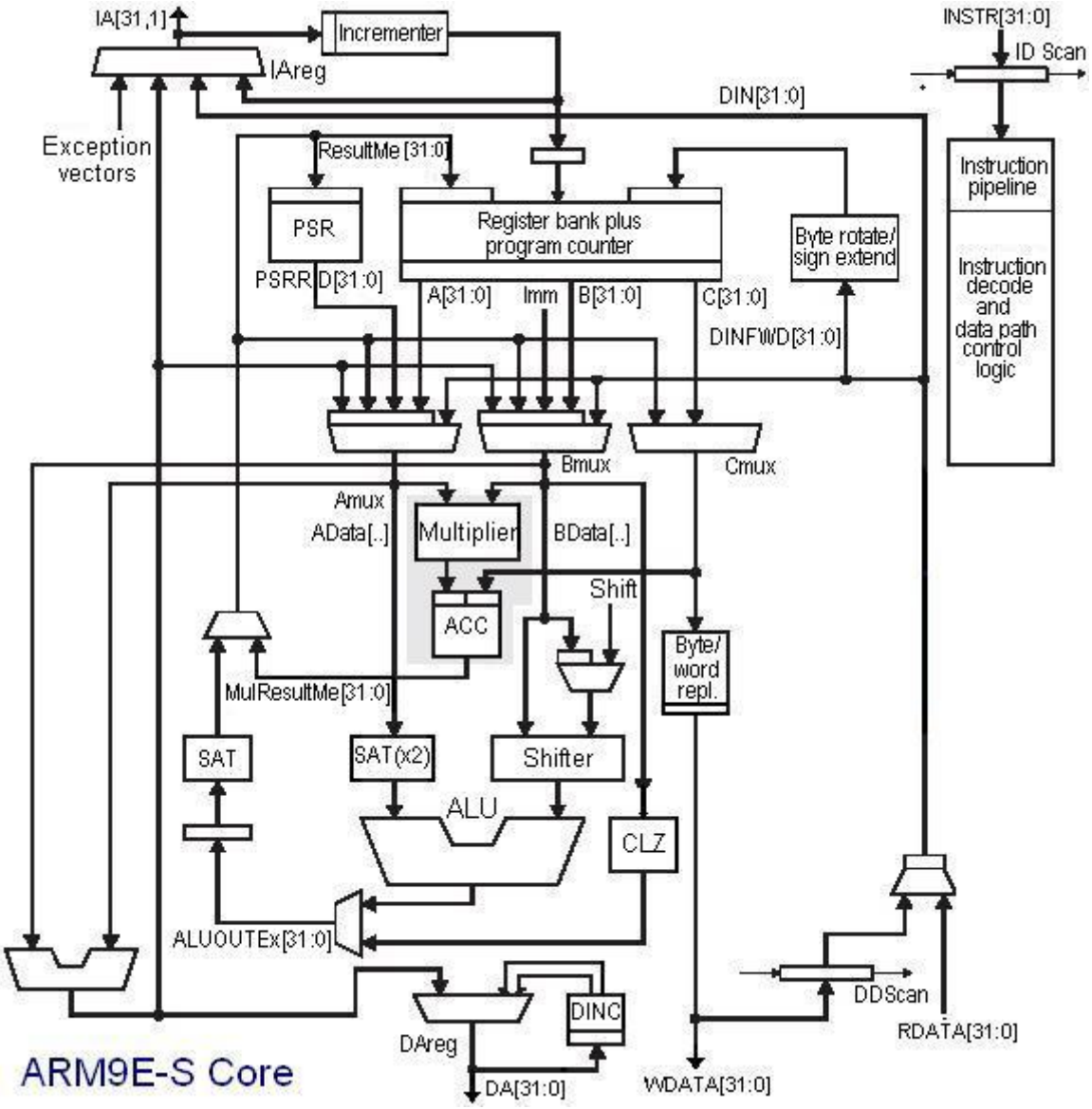


Рис. 4.3. Структурная схема ядра ARM9E-S

Адреса данных передаются по шине DA[31:0] из регистра адреса данных (*Data Address Register* – DAreg). При этом значения адресов берутся из поля команды при прямой адресации, из регистров регистрового блока, где были сохранены как результаты операций, реализующих режимы адресации, а также получаются путём простого наращивания с помощью инкрементера адреса данных (DINC).

Инструкции из памяти принимаются по шине INSTR[31:0] и проходят по конвейеру операций:

- выбор инструкций из памяти,
- декодирование инструкции и указанных в инструкции регистров,
- выполнение операций ALU, MAC и сдвигателя,
- загрузка (сохранение) регистров из(в) памяти(ь),

– сохранение результата операции в одном из регистров регистрового блока.

Для обмена данными с памятью в ядре имеются двунаправленная шина DIN[31:0] и две однонаправленные шины – шина RDATA[31:0] для приёма данных из памяти и шина WDATA[31:0] для передачи данных в память. Если вновь обратиться к структурной схеме на рис. 4.2, то в ней для передачи данных используется двунаправленная шина DD[31:0]. Можно сказать, что двунаправленная шина данных DD[31:0] образована однонаправленными шинами RDATA[31:0] и WDATA[31:0].

Шинам IA[31:1]. DA[31:0] и INSTR[31:0] на рис. 4.3 соответствуют шины IVA[31:0], DVA[31:0] и ID[31:0] соответственно.

4.2. Механизм трансляции адреса и кэш-память

Упомянутые выше механизмы виртуальной адресации и кэш-памяти требуют пояснений, поскольку с ними тесно связаны вопросы построения многозадачных (многопроцессных систем). Многозадачностью называют такой способ организации работы системы, при котором в её памяти одновременно содержатся программы и данные для выполнения нескольких процессов обработки информации (задач). При этом должна обеспечиваться взаимная защита программ и данных, относящихся к различным задачам, а также возможность перехода от выполнения одной задачи к другой (переключение задач). Память системы сегментируется и каждому отдельному процессу (задаче) выделяется свой сегмент с ограниченными правами доступа со стороны других процессов (задач). Для начала рассмотрим процесс трансляции адресов памяти с общих позиций.

В тех случаях, когда объём памяти системы недостаточен для того, чтобы соответствовать перекрываемому процессором адресному пространству, а генерируемые при исполнении программы адреса выходят за пределы адресного пространства физической памяти, формируемые ядром процессора адреса нельзя непосредственно использовать для адресации памяти. Возникает необходимость отображения генерируемых адресов на физическую память системы. Это является одной из задач, решаемой с помощью механизма виртуальной адресации.

В многозадачных системах с сегментированной памятью процессор формирует линейный (виртуальный) адрес, исходя из логического адреса, определяемого по содержимому специального регистра – селектора сегмента и относительному адресу. Линейный адрес находится как сумма базового адреса (адреса начала) сегмента и относительного адреса (смещения в сегменте). Схема формирования линейного адреса представлена на рис. 4.4.

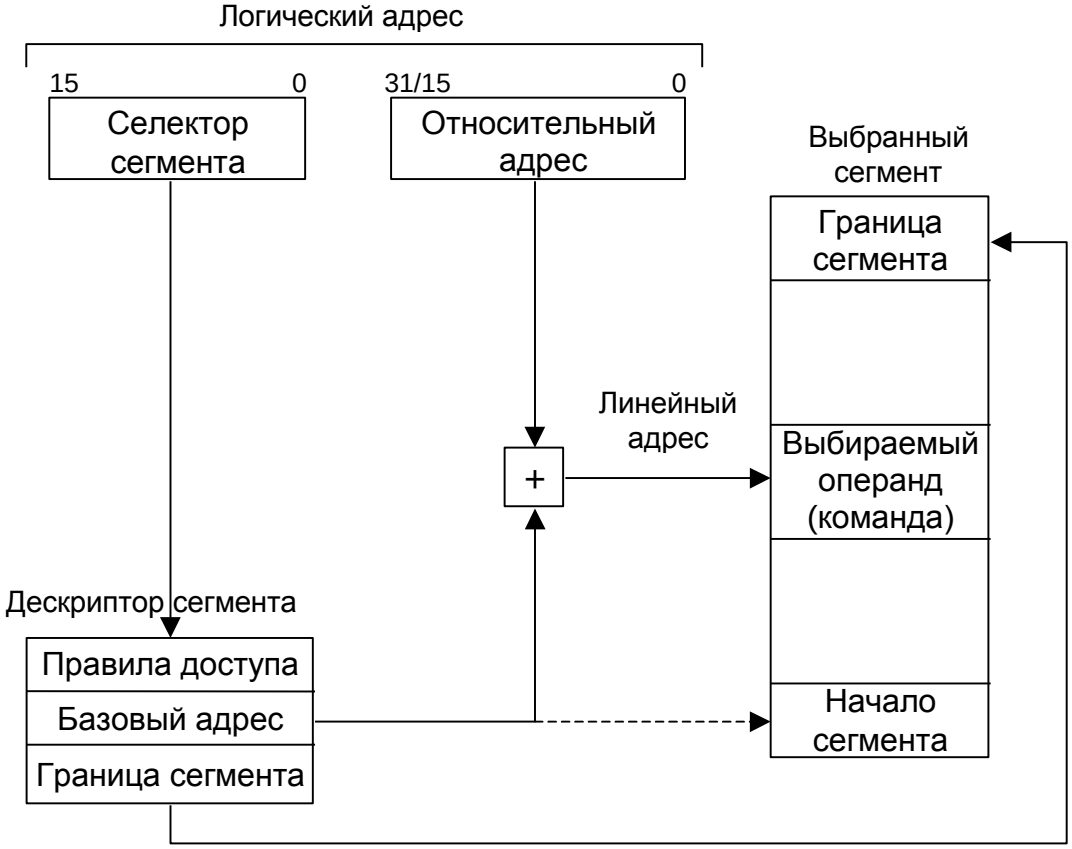


Рис. 6.4. Формирование линейного адреса

Современные процессоры имеют средства, обеспечивающие защиту от случайного (непредусмотренного решаемой задачей) обращения к сегментам и страницам памяти. Система защиты предусматривает различные виды контроля:

- контроль доступа к сегментам и страницам с помощью системы привилегий;
- контроль использования сегментов и страниц путём ограничений на разные виды обращения к ним: запрещение записи в сегменты данных (страницы), разрешение только для чтения, запрещение чтения сегментов (страниц) программ, разрешённых только для выполнения, запрещение обращения к незагруженным (отсутствующим) сегментам и страницам и ряд других;
- ограничение набора выполняемых команд в зависимости от уровня привилегий выполняемой программы (выделение привилегированных команд).

Страничная организация памяти

Как правило, память разбивается на страницы, размеры которых могут быть различными для разных процессоров или для одного и того же процессора, но при использовании различных режимов работы с памятью. Страницы могут быть размером от одного или нескольких килобайт (стандартным является размер в 4 Кбайта) до нескольких мегабайт (например, 4 Мбайта).

Линейный адрес разбивается на три поля (рис. 4.5): поле каталога разделов – директории (*Directory*), поле таблицы страниц (*Table*) и поле адреса в странице (*Offset*). Базовый адрес каталога разделов задаётся значением специального регистра процессора DB (*Directory Base*). Регистр программно доступен. Обычно его содержимое определяется действиями на стадии инициализации операционной системы.

По содержащимся в каталоге разделов указателям страниц находится базовый адрес нужной таблицы страниц. Из выбранной таблицы страниц берётся адрес страницы, в которой находится адресуемая ячейка памяти и по смещению *Offset* находится её адрес.

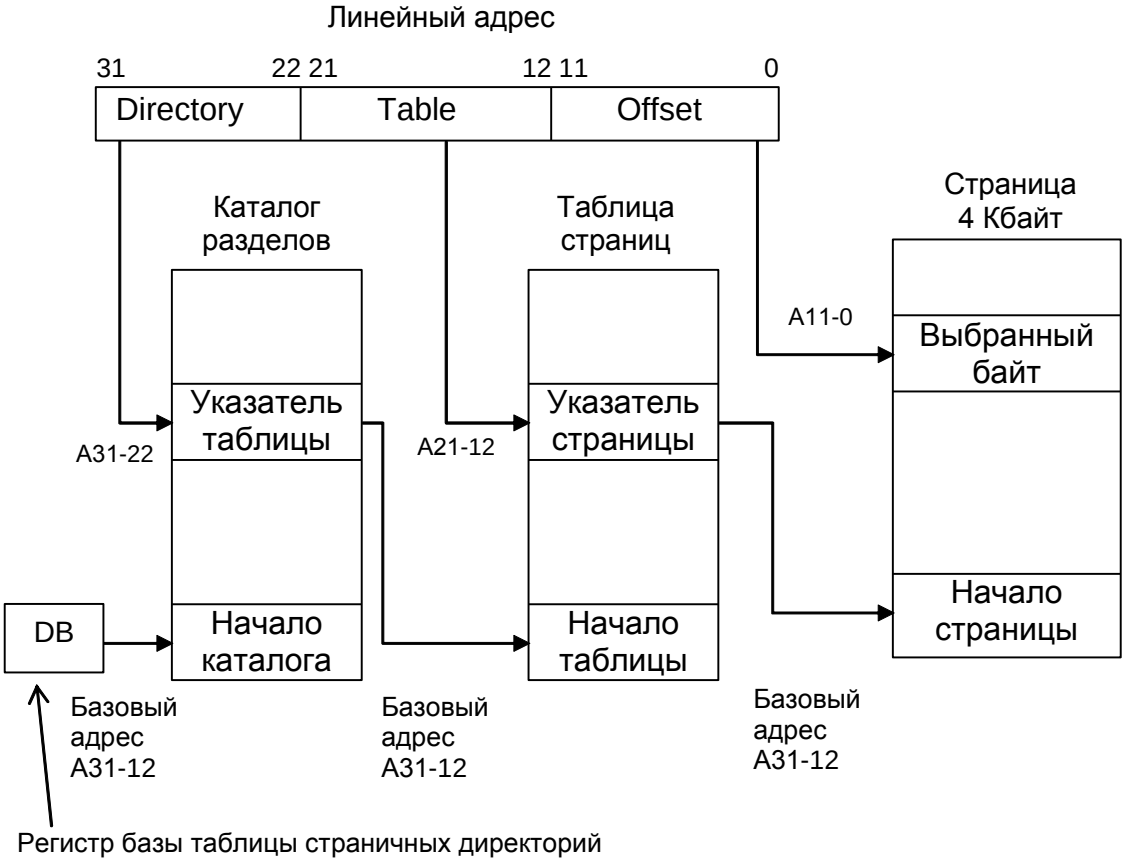


Рис. 4.5 Формирование 32-разрядного физического адреса при страничной организации памяти для страниц стандартного размера

Таким образом, трансляция адреса выполняется как некоторая последовательность действий, предполагающих многократное обращение к памяти, где хранятся каталог разделов и таблицы страниц. Существенное сокращение времени преобразования адресов достигается путём использования специализированной внутренней кэш-памяти, которая называется буфером ассоциативной трансляции (*Translation Look-Aside Buffer* – TLB). Эти буферы хранят физические адреса нескольких ранее выбранных страниц. При повторном обращении к этим страницам процедура трансляции не выполняется, а выбирается из TLB

ранее полученный адрес страницы. Нередко процессоры содержат два буфера TLB – один для адресов команд, другой для адресов данных.

Общие принципы организации кэш-памяти

В основу работы кэш-памяти положен принцип локальности программы или, как ещё говорят, гнездовой характер обращений, имея в виду, что адреса последовательных обращений к основной памяти (ОП) образуют компактную группу. При обращении к ОП в кэш-память копируются не отдельные данные, а блоки цифровой информации, включающие те данные, которые с большой степенью вероятности будут использованы в ЦП на последующих шагах работы

Понятие тега, индекса и блока

При использовании кэш-памяти память условно представляется в виде двумерного массива (рис. 4.6), состоящего из строк и столбцов. На пересечении строк и столбцов находятся блоки, состоящие из нескольких байтов с числом кратным 2 (2, 4, 8, 16, 32), разным для разных МП-систем. Адрес блока в строке называется *индексом* блока, а адрес строки, которой он принадлежит – *тегом*.

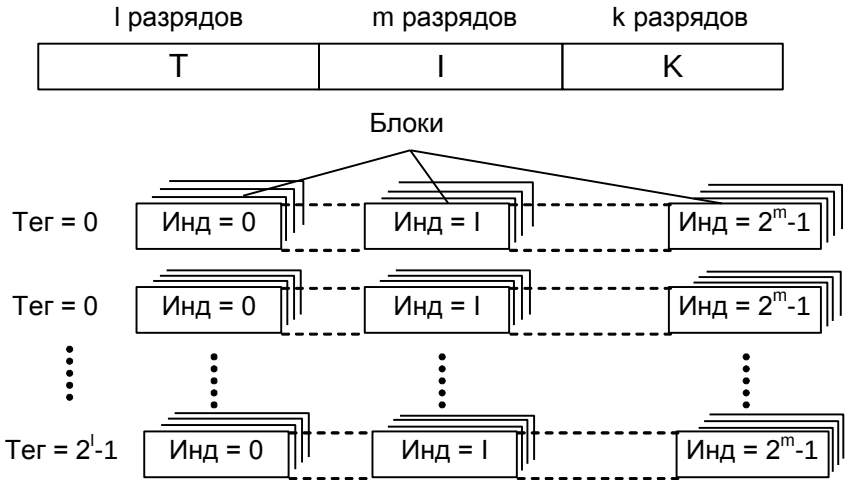


Рис. 4.6. Представление памяти при использовании механизма кэширования

Механизм кэш-памяти с прямым отображением

Во всех механизмах кэш-память представляет собой две сверхоперативные памяти (СОП): память отображения данных и память тегов. В случае памяти прямого отображения память отображения выглядит как строка двумерного массива с размером ячейки, равным размеру блока (рис. 4.7).

Память тегов имеет тот же объем, но с размером ячейки, определяемым размером тега.

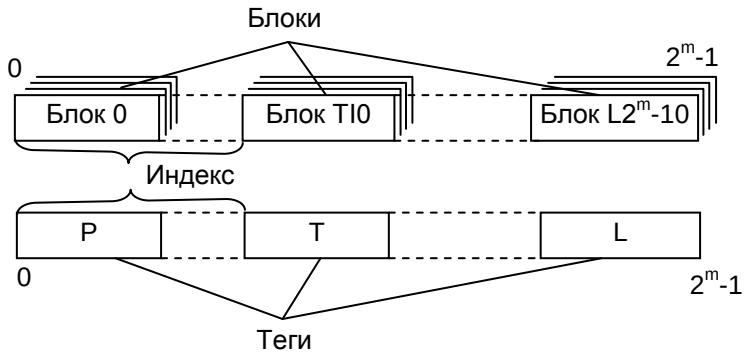


Рис. 4.7. Кэш-памяти с прямым отображением

Механизм кэш-памяти с ассоциативным отображением данных

В отличие от предыдущей кэш-память с ассоциативным отображением данных и СОП данных, и СОП тегов содержит по несколько «строк». Число строк может быть разным для разных систем, но обычно кратно 2. Известны кэш памяти на 2, 4, 8 строк. Однако число строк отображения данных и число строк тегов равны, так что каждой строке данных однозначно соответствует определенная строка тегов.

На рис. 4.8 приведён пример кэш-памяти с ассоциативным отображением данных. В примере скопированы блоки с адресами L,0,0; T2,I1,0; ...; P,I2,0; ...;T1,2^m-1,0; P,0,0; ...; T1,I1,0; ...; L,2^m-1,0.

Обращение к ОП по адресу T,I,K происходит следующим образом. Проверяется, нет ли блока TI в кэш-памяти. По ассоциативному признаку T адреса обращения осуществляется ассоциативный поиск в СОП тегов в столбце с индексом I из адреса обращения.

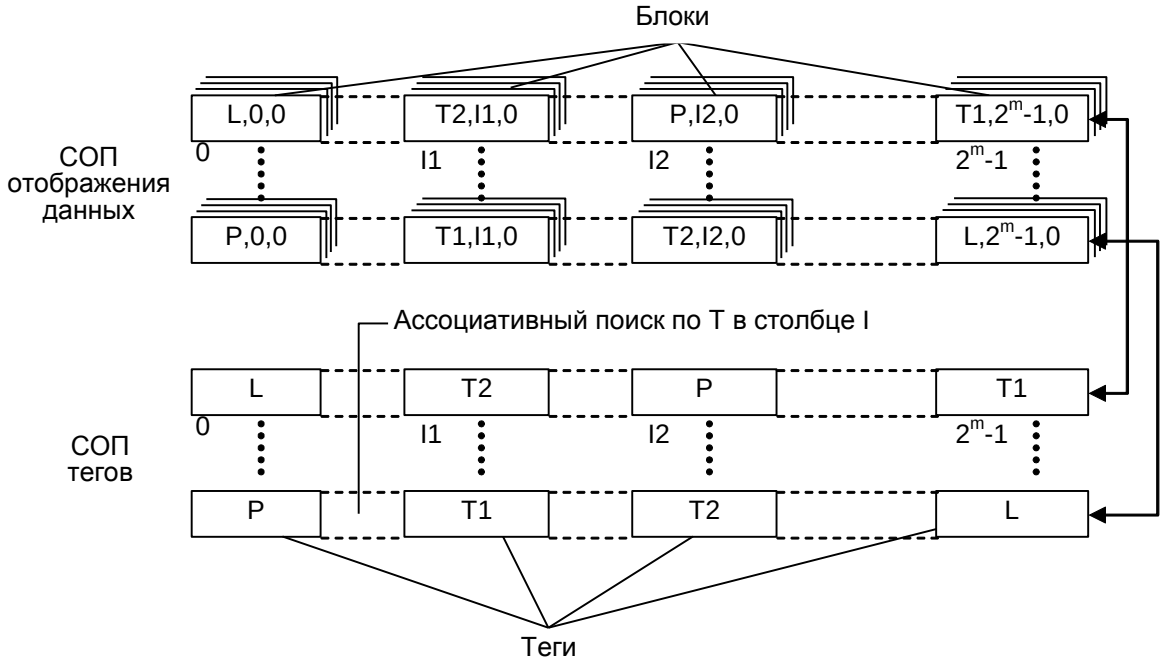


Рис. 4.8. Кэш-память с ассоциативным отображением

Если значение Т совпадает с одним из значений, записанных в память тегов, то фиксируется кэш-попадание и обращение реализуется через обращение к СОП данных по значению I адреса из соответствующей строки. В примере – из первой и последней строк в зависимости от того, в какой строке совпали теги.

Копирование блока в кэш-память осуществляется путём записи блока и тега, ему соответствующего, в любую строку; если таковой нет, то в соответствии с принятым алгоритмом обновления на место прежней копии записывается новый блок. Для управления перезаписью блоков используются различные алгоритмы, основанные на учёте статистики кэш-попаданий за некоторый интервал наблюдения. При этом обычно в ячейку СОП тегов вводятся дополнительные разряды, учитывающие частоту кэш-попаданий. Чем меньше кэш-попаданий, тем скорее последует перезапись блока.

При конструировании систем, как правило, применяется некоторая комбинация перечисленных методов для уверенности в достоверности данных.

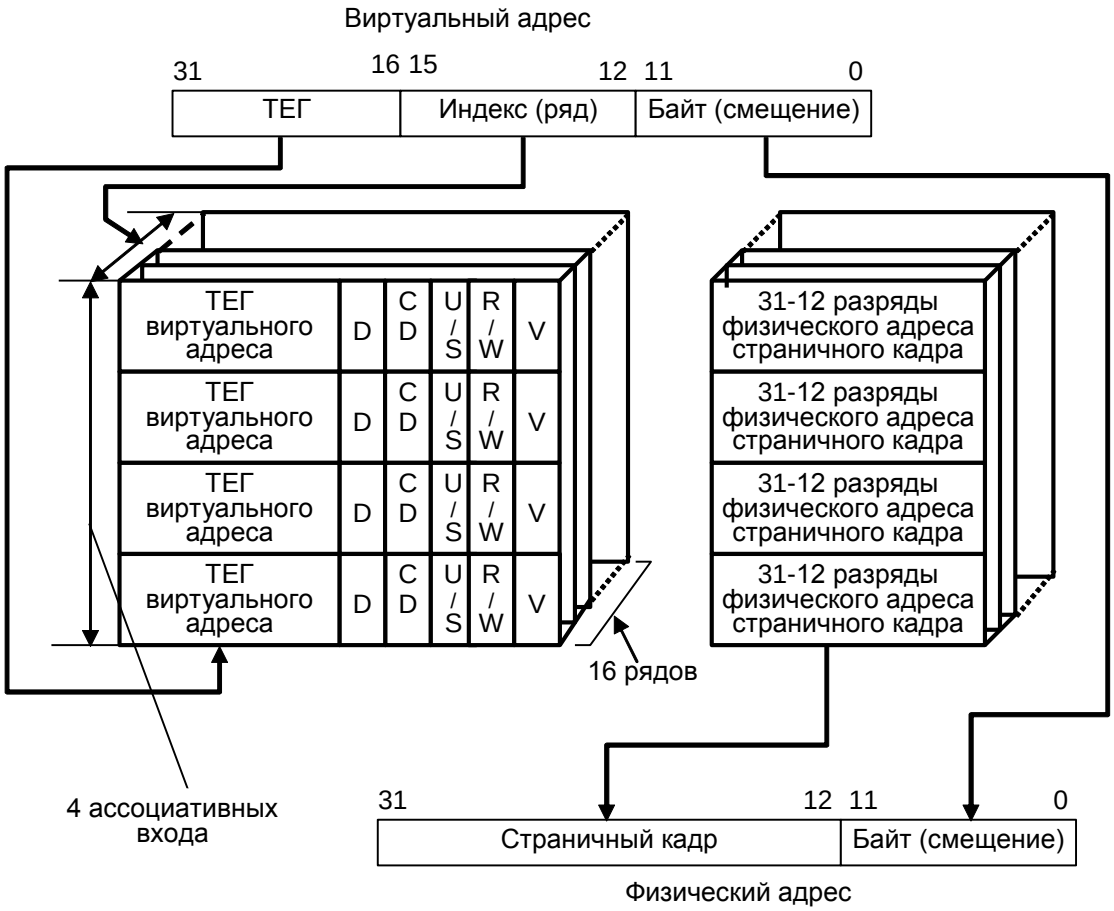


Рис. 4.9. Организация буфера TLB для 4-Кбайтных страниц (D – «грязь»; CD – кэш-копирование разрешено; WT – сквозная запись; U/S – режим пользователь/супервизор; R/W – разрешение записи; V – корректность)

Кэш память адресной трансляции

Современные МП поддерживают различные по размеру страницы и для каждого размера страниц используются отдельные кэш-буферы (ассоциативные буферы страничной трансляции – TLB).

В качестве примера на рис. 4.9 представлена структура TLB в МП INTEL для 4-Кбайтных страниц. Такой буфер имеет 64 записи с ассоциативным поиском по четырём входам.

Внутренние кэш-памяти команд и данных

Современные процессоры имеют отдельные кэш-памяти данных и команд. Использование раздельных кэш-памятей для данных и команд обеспечивает одновременные кэш-просмотры.

Каждая кэш-память имеет два ряда тегов (рис. 4.10): виртуальные теги для внутренних обращений МП и физические теги, используемые для обращений к основной памяти (внешних обращений через магистраль). На рис. 4.10 показано, как разряды виртуального и соответствующего ему физического адреса управляют обращениями к кэш-памяти. Наличие виртуального и физического тегов позволяет поставить один единственный физический адрес, полученный при трансляции через TLB-буфер, в соответствие двум или более виртуальным адресам.

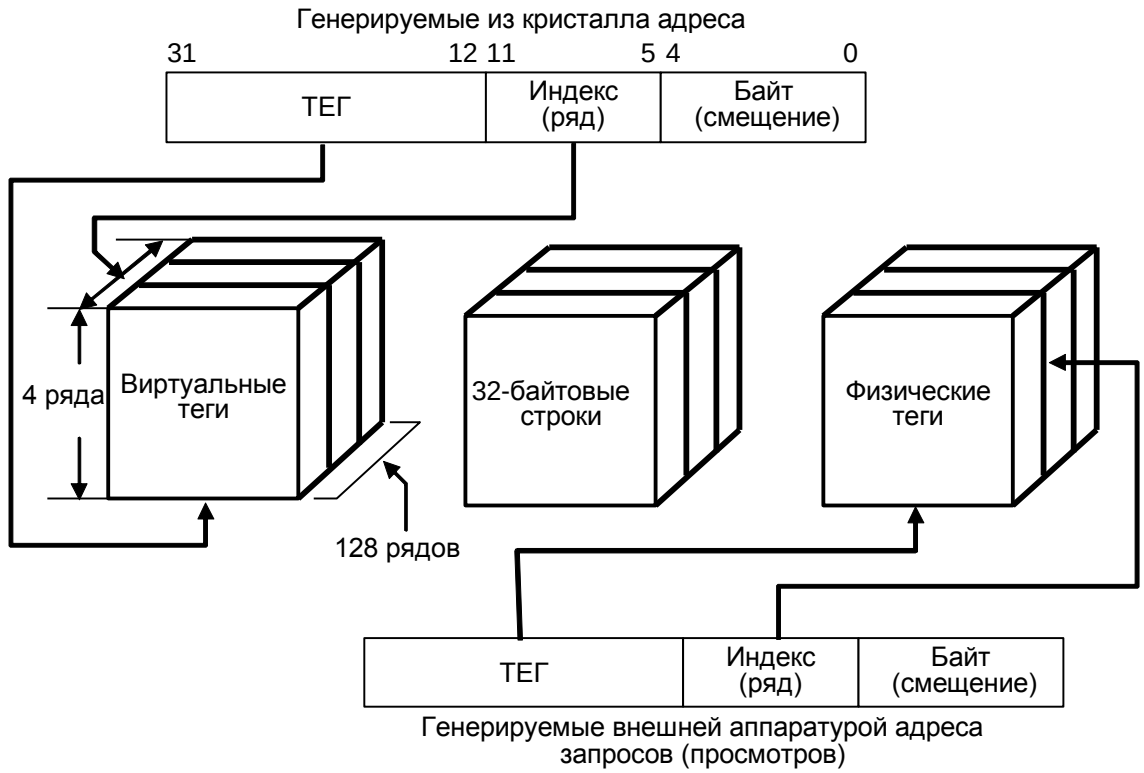


Рис. 4.10. Выдача адресов для кэш-просмотров

Кэш-промахи при записи в память не копируются в кэш-памяти, а данные, связанные с ними, помещаются в буфер записи (*Write Buffer*) и затем, когда внешняя магистраль доступна, отправляются в основную память.

При чтении, если обнаруживается физический адрес (кэш-попадание по физическому адресу), данное из магистрали игнорируется, а используется данное в ЧИПе, и поле виртуального тега замещается новым виртуальным тегом. При записях, если обнаруживается виртуальный адрес (кэш-попадание по виртуальному адресу), запись вводится на магистраль и обновляется память. Если физический адрес обнаруживается (кэш-попадание по физическому адресу), обновляется строка в кэш-памяти, а виртуальный тег замещается новым виртуальным тегом.

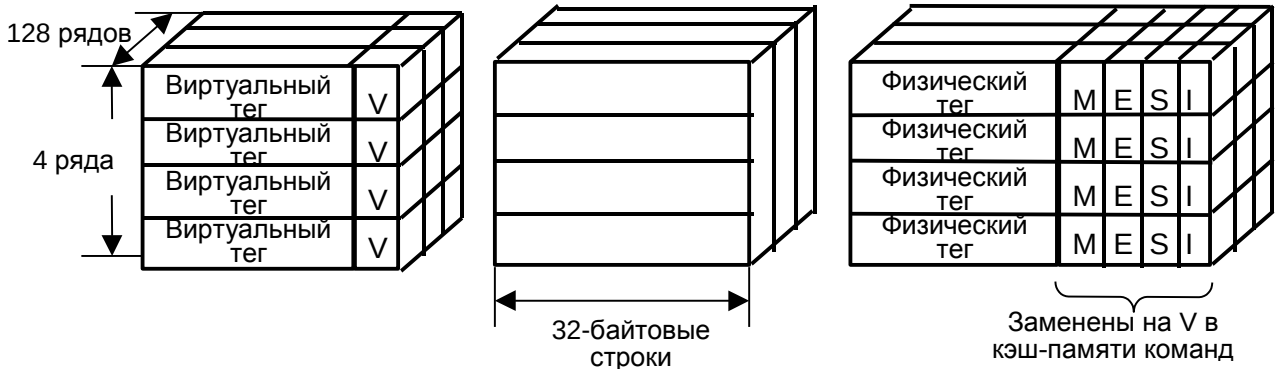


Рис. 4.11. Организация кэш-памяти данных (M - модифицированная; E – эксклюзивная; S – разделяемая; I – некорректная; V – корректность)

Рис. 4.11 иллюстрирует **организацию кэш-памяти данных**. Кэш-память данных использует четыре разряда статуса (M, S, E и I) для каждого физического тэга и один разряд статуса – разряд корректности V – для виртуального тэга.

В многопроцессорных системах важным моментом является обеспечение когерентности кэшей. Когерентность обеспечивается по MESI-алгоритму, который устанавливает правила сброса/установки битов состояния M (модифицированная), E (эксклюзивная), S (разделяемая) и I (некорректная) строки с запрашиваемым физическим тэгом.

- Бит M показывает, что ни один другой кэш не содержит копий этой строки. Строка изменена, но изменения ещё не попали в ОЗУ, т. е. значение ОЗУ не действительно. Сигнал о модификации рассылается другим процессорам, которые переведут свои строки в состояние I.
- Бит E устанавливается, когда никакой другой кэш копий такой строки не содержит. Содержимое строки совпадает с содержимым ОЗУ, т. е. обе существующие копии строго действительны. Процессор с единственной строкой в кэше отслеживает все обращения к ней.
- Бит S показывает, что один или несколько других кэшей содержат копии этой строки. Все копии, как в ОЗУ, так и в кэшах действительны. На шине выставляется сигнал об обнаружении разделяемости.

– Бит I сообщает, что строка ОЗУ содержит устаревшие данные. Обращение к этой строке со стороны «своего» процессора блокируется до тех пор, пока не будут модифицированы данные в ОЗУ.

Информация о состоянии M, S, E и I битов передаётся по специально выделенным для этой цели линиям – линиям, обеспечивающим когерентность кэшей.

Для минимизации трафика при обращениях ядра процессора к памяти обычно используется стратегия обратной записи (*write-back*). Стратегия обратной записи (также называемой обратным копированием или отложенной записью) уменьшает трафик внешней магистрали за счёт значительного сокращения многих ненужных записей в память. Записи в какую-либо строку кэш-памяти не немедленно перезаписываются в основную память, а накапливаются в кэш-памяти. Изменённая кэш-строка переписывается в основную память лишь тогда, когда её место в кэш-памяти нужно другому данному, когда модифицированное данное нужно другому процессору или когда выполняется процедура сброса кэш-памяти.

При стратегии сквозной записи запись, запрашивающая строку в кэш-памяти, вызывает обновление как кэш-памяти, так и основной памяти. Путём установки соответствующих разрядов в дескрипторах страниц можно выбрать стратегию сквозной или обратной записи для специфицированных областей основной памяти. Стратегия сквозной записи используется, например, для таких областей памяти, которые используются под очереди межпроцессорных сообщений.

Стратегия (*write-once*) – однократная запись – представляет собой сочетание сквозной и обратной записей. Сквозная запись применяется для первой записи кэш-строки, в то же время последующие записи в ту же строку идут по стратегии обратной записи. Однократная запись выгодна в мультипроцессорных системах для обеспечения кэш-согласования при минимально возможном трафике магистрали. Первая запись извещает другой процессор о факте изменения строки. Стратегия однократной записи используется также, если к МП подключена кэш-память второго уровня для согласования обоих уровней кэш-памяти.

Возможны варианты, когда внешняя система может динамически изменять стратегию обновления (обратная запись, сквозная запись, однократная запись) внутренней кэш-памяти данных для каждой внутренней кэш-строки.

Кэш-память команд организована подобно кэш-памяти данных, но в ней отсутствуют биты M, E, S, I. Вместо них присутствует только один бит V. Поддержание ситуации, когда одна физическая строка может отвечать разным виртуальным адресам для команд состоит не в том, чтобы просто изменять виртуальный тэг для некоторой строки, а скорее в выборе строки, если имеет место кэш-промах по значению виртуального тэга. Если физический адрес уже существует в кэш-памяти команд, соответствующая ему строка и тэги перезаписываются. Таким образом, даже несмотря на то, что к некоторой физической стро-

ке можно обращаться по разным виртуальным адресам, процессор никогда не вводит строку дважды в кэш-память команд.

4.3. Трансляция адреса и управление памятью в ARM процессорах

Высокие требования к системам реального времени и разнообразие их применений накладывают свой отпечаток и на систему памяти – от простой однородной памяти с единым адресным пространством до сложной, позволяющей оптимальным образом использовать ресурсы системы. Выше при рассмотрении архитектур процессоров мы обсуждали разные виды памяти: память программ, память данных и постоянную память, для каждой из которых может быть выделено своё адресное пространство. И это находится в русле тех требований, которые предъявляются к памяти встроенных систем:

- поддержка нескольких типов памяти,
- использование кэш-памяти (*cache*),
- использование буферов записи (*write buffers*) в ОЗУ,
- поддержка виртуальной памяти (*virtual memory*).

Портам ввода/вывода, отображённым на память, выделяется отдельная область адресного пространства. Специфика работы с этой областью обусловлена прежде всего тем, что порты ввода/вывода могут иметь значительное различие по времени доступа.

Большинство систем обладает средствами инициализации и управления памятью. К ним относятся

- разрешение/запрещение кэширования и буферизации основной памяти,
- поддержка виртуального адресного пространства и отображения виртуального адреса в физический,
- возможность работы с разными (по техническому исполнению и времени доступа) типами памяти,
- средства защиты специфицированных областей памяти и ограничение прав доступа к ним,
- обеспечение корректной работы с отображёнными на память портами ввода/вывода. (Это автоматически делается в случае с однородной памятью, но сложности возникают при работе с кэшированной и буферизованной памятью).

Прежде чем перейти к конкретным вопросам управления памятью, сделаем несколько общих замечаний (не считаясь с тем, что некоторые из замечаний сами нуждаются в пояснениях, так как затрагивают специфические вопросы управления памятью).

1) Память, выделенная под программный код и под данные, может быть кэширована и буферирована. При этом
   - основная память (RAM, ОЗУ) может быть и кэширована, и буферирована, в то время как
   - постоянная память (ROM, ПЗУ) может только кэшироваться, но не может быть буферированной, поскольку не доступна для записи.

2) Выполняемые процессором операции записи в ОЗУ могут производиться в то время, когда системная шина занята. В этом случае запись в основную память на некоторое время откладывается, а предназначенные для ОЗУ данные временно сохраняются в буферах записи. Назвать их можно также *буферами данных*, поскольку используются они исключительно для записи *данных* в память. Механизм буферов записи рассчитан на снижение числа пустых тактов ожидания процессором доступа к системной памяти. В некоторых процессорах записанные в буфер данные могут быть перераспределены. По этой причине запись в ОЗУ не всегда соответствует тому порядку, который предусмотрен последовательностью выполненных процессором команд. Это означает, что область памяти, предназначенная для портов ввода/вывода, не может быть буферированной.
3) В ARM процессорах при совместной работе с другими (если они есть) ведущими на системной магистрали нет средств обеспечения когерентности кэша и буферов записи. Проблемы когерентности решаются программным способом.

   Если память разделяется разными ведущими без применения контролирующего когерентность программного кода, то следует
   - запретить кэширование памяти при чтении, а
   - при записи запретить и кэширование, и буферирование.

   Если система поддерживает когерентность буферирования данных (с помощью соответствующего программного кода), то доступ к буферам записи процессора возможен также со стороны других ведущих системной магистрали и в режиме чтения, и в режиме записи. При этом
   - другие ведущие могут прочитать содержащиеся в буфере данные после того, как выполнена перезапись соответствующей области кэша (кэш с отложенной записью (*Write Back Cache*)) и содержимое буфера записано в основную память;
   - занесённые в буфер (а затем и переданные в ОЗУ) данные от других ведущих попадают в кэш данного процессора при обращении к нему в режиме чтения.

   Если в системе есть несколько ведущих на системной шине и они используют семафоры, то семафоры не могут быть кэшированными или буферизованными.
4) В случае применения кэш-памяти с отложенной записью обновлённые при записи в кэш данные должны быть перенесены в ОЗУ до того, как устройство управления памятью (MMU) модифицирует таблицу страниц, с помощью которой формируется физический адрес памяти.
5) Индексация линий кэш-памяти производится либо по виртуальному, либо по физическому адресу памяти. При этом физическая страница памяти должна отображаться только в одну страницу памяти виртуальной. Иначе результат такого отображения может быть непредсказуем. ARM-

процессоры не поддерживают когерентность при многократном отображении виртуальной памяти в одну страницу памяти физической.

6) ARM-архитектура допускает разделение кэш-памяти на память инструкций и память данных.

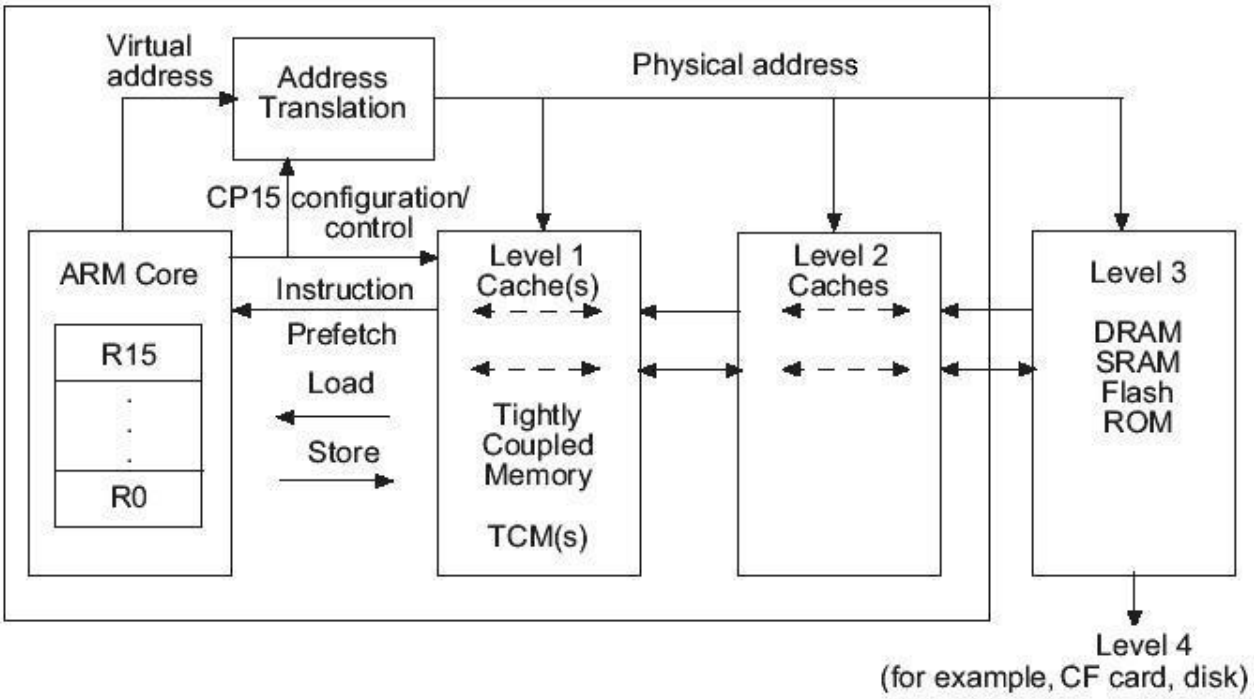


Рис. 4.12. Иерархия памяти в ARM системах

Память ARM систем в общем случае можно разделить на три (исключая регистры процессора) основные категории (рис. 4.12):

1. память первого уровня, составленная из кэш-памяти и сильно связанной с ядром внутренней памяти (*Tightly Coupled Memory* – TSM);
2. память второго уровня, являющаяся кэш-памятью;
3. память третьего уровня – основная память системы (постоянная и оперативная);
4. память четвёртого уровня – память на внешних носителях (например, дисковые накопители).

В ARM процессорах используются три способа адресации памяти:

- по виртуальному адресу (VA);
- по модифицированному виртуальному адресу (MVA);
- по физическому адресу (PA).

Механизм адресации по виртуальному адресу VA не имеет существенных отличий от рассмотренного выше преобразования виртуального адреса в физический.

Адресация с использованием модифицированного виртуального адреса во многом напоминает вычисление исполнительного адреса с применением сегментных регистров. Основное отличие, заключается в том, что процессу выделяется не сегмент произвольного размера, а отдельный раздел (блок) в вирту-

альном адресном пространстве. Вследствие этого виртуальное адресное пространство оказывается поделённым между существующими в системе процессами.

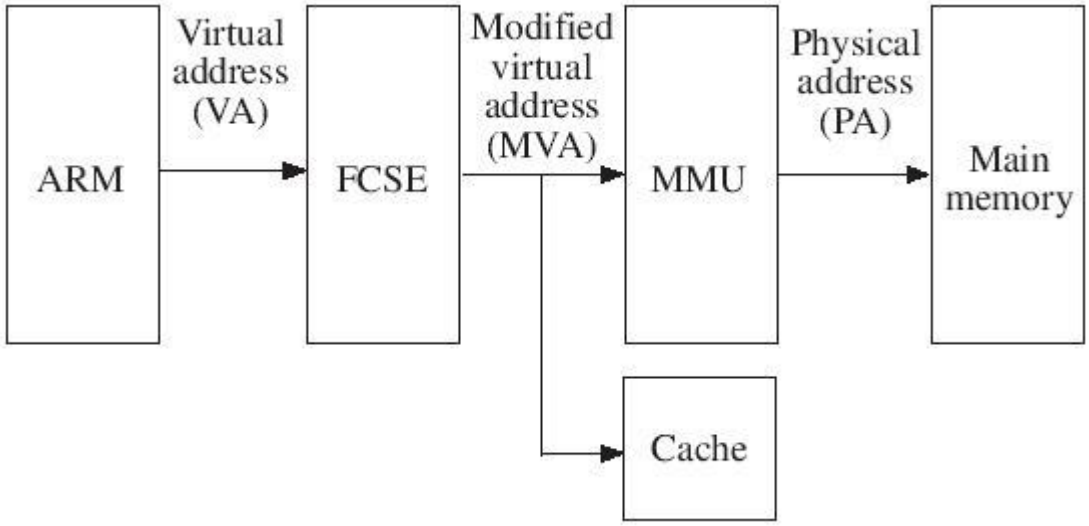


Рис. 4.13. Схема преобразования виртуального адреса в физический

Каждому процессу назначается идентификатор ID, значение которого определяют семь старших бит 32-разрядного MVA. В итоге система может поддерживать до 128 процессов, выделяя каждому исполняемому процессу блок виртуальной памяти объёмом в 32M. Модифицированный виртуальный адрес формируется сопроцессором CP15, который вычисляет его по схеме

$$MVA = VA + ID \times 32M,$$

и направляет в кэш-память и в устройства управления памятью (MMU). Таким образом, с помощью VA осуществляется адресация в пределах выделяемого исполняемому процессу 32M блока. Как следствие, разрядность виртуального адреса VA на семь бит меньше разрядности MVA.

Рассмотренный механизм трансляции адреса является расширением, обеспечивающим быстрое контекстное переключение (*Fast Context Switch Extension* – FCSE). Ввод в действие механизма FCSE допускает возможность перекрытия виртуальных адресных пространств, принадлежащих различным процессам. Такое перекрытие допустимо, поскольку действительные адреса никогда не перекрываются из-за различий вносимых идентификаторами процессов.

Когда виртуальные адресные пространства процессов перекрываются, то при их переключении обычно приходится изменять содержимое таблицы страниц, используемых для преобразования виртуального адреса в физический. Это вызывает также необходимость обновления содержимого буферов страничной трансляции TLB. В результате увеличивается время переключения процессов. Поскольку механизм FCSE даёт возможность использования различными процессами одних и тех же виртуальных адресов VA, то снижаются накладные расходы, связанные с обновлением хранящихся в TLB таблиц страниц.

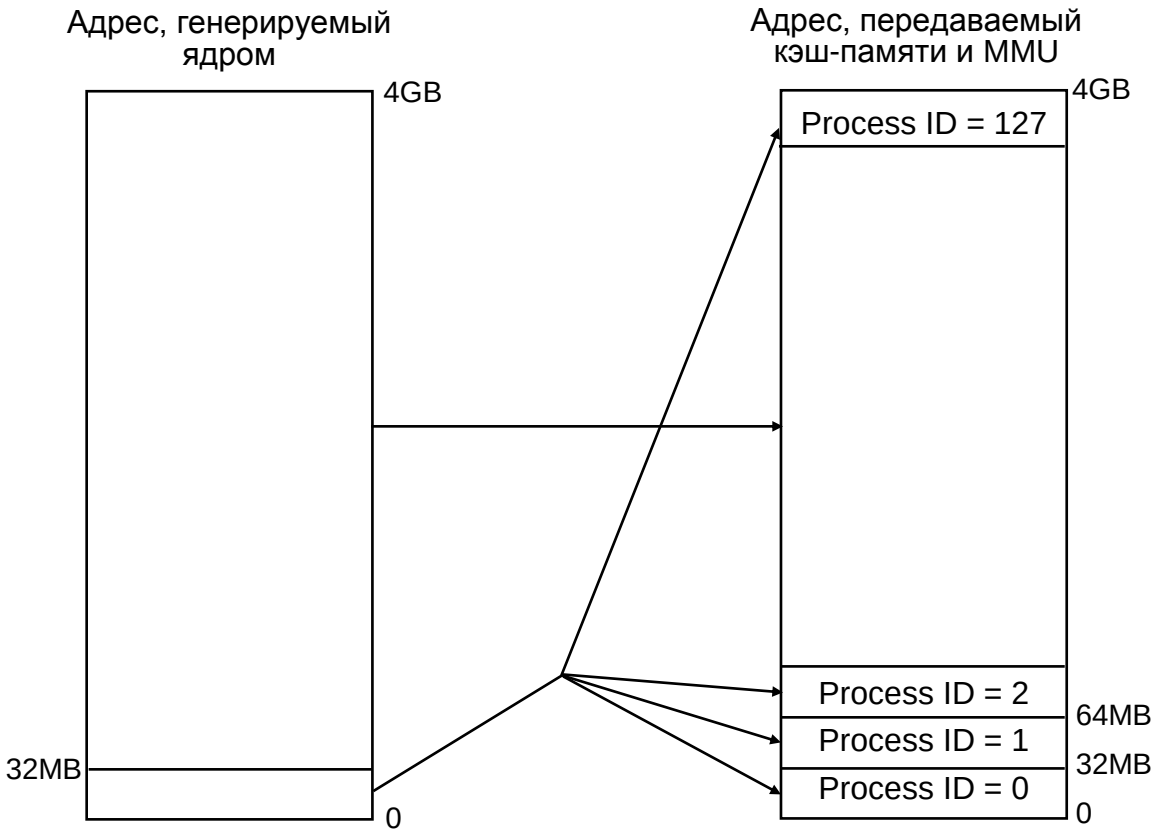


Рис. 4.14. Схема отображения адреса сопроцессором CP15 с использованием идентификатора процесса

Более подробно схема отображения генерируемого ядром адреса на виртуальное адресное пространство показана на рис. 4.14. Если поле виртуального (генерируемого ядром процессора) адреса VA[31:25] содержит только нулевые биты, то VA преобразуется в MVA и отображается на один из 32M блоков в зависимости от значения идентификатора процесса ID. В противном случае, т. е. при VA[31:25] ≠ 0b0000000, виртуальный адрес VA не модифицируется и выполняется обычная трансляция VA в физический адрес PA. Связь между VA и MVA можно выразить следующим образом:

```
if(VA[31:25] == 0b0000000) then
  MVA = VA | (ID << 25)
else
  MVA = VA
```

Если в микропроцессорной ARM системе используемое адресное пространство не превосходит объёма физической памяти, то нет необходимости в применении механизма виртуальной адресации. В этом случае процессор может быть настроен на работу в режиме адресации по физическому адресу.

Система управления памятью

Система управления памятью (*Memory Management Unit* – MMU) контролирует память, структурируя её в соответствии с требованиями, которые предъявляет встроенная система реального времени. Большая часть функций контро-

ля связана с создаваемыми в памяти адресными таблицами (TLB). Элементы этих таблиц определяют свойства соответствующих областей памяти с размерами от 1 Кбайта до 1 Мбайта. К числу этих свойств относятся:

Принадлежность виртуальному или физическому адресному пространству

В том случае, когда генерируемые процессором адреса являются виртуальными, MMU отображает виртуальный адрес в адрес физический, учитывая общий объем памяти системы. Преобразовать виртуальный адрес в физический можно по-разному. Например, во избежание конфликтного взаимодействия разных процессов, можно назначать этим процессам неперекрывающиеся области памяти. В случае, когда какое-либо приложение имеет разбросанную по адресному пространству карту памяти, можно его раздробленные части расположить в смежных областях.

Права доступа к области памяти

MMU имеет механизм ограничения прав доступа к объектам памяти, с помощью которого устанавливается, какие области памяти доступны для чтения, записи или чтения/записи. В тех случаях, когда исполняемая программа обращается к недоступной области памяти, фиксируется особая ситуация, на которую процессор реагирует, предпринимая соответствующие действия. Права доступа к защищаемым областям памяти зависят также от того, в каком режиме исполняется программа – в режиме пользователя или в привилегированном режиме.

Использование кэш-памяти и буферов записи

Для выделенной области памяти можно запретить или разрешить кэширование и буферизацию.

Процесс генерации исполнительного адреса всегда связан с просмотром адресной таблицы (*translation table walk*), что требует временных затрат, хотя делается это автоматически и с привлечением аппаратных средств (*translation table walk hardware*). Для того, чтобы свести к минимуму временные затраты на генерацию адресов, результат работы с адресной таблицей оформляется в виде одной или нескольких структур – буферов страничной трансляции TLB. Обычно в системах с однородной памятью используется один буфер TLB. Если память данных и память программ разделены, то для каждой из них имеется свой TLB. В таком случае и память программ, и память данных имеют свой кэш. Другими словами имеется по одному TLB на каждый кэш.

Порядок (последовательность) доступа к памяти

Когда процессор обращается к памяти, MMU прежде всего определяет наличие соответствующего виртуального адреса в TLB. В самом начале процесса трансляции адреса буфер TLB пуст и потому для отображения виртуального адреса в физический может быть использован только после заполнения. Если память данных и память программ разделены, для выборки инструкции

используется TLB инструкций, а для доступа к данным – TLB данных. Для извлечения информации из расположенной в основной памяти адресной таблицы используется специально предназначенное для этого устройство просмотра адресной таблицы (*translation table walk hardware*). После того, как обращение к таблице состоялось, соответствующая запись заносится либо в свободное место TLB, либо замещает одну из уже имеющихся в TLB записей.

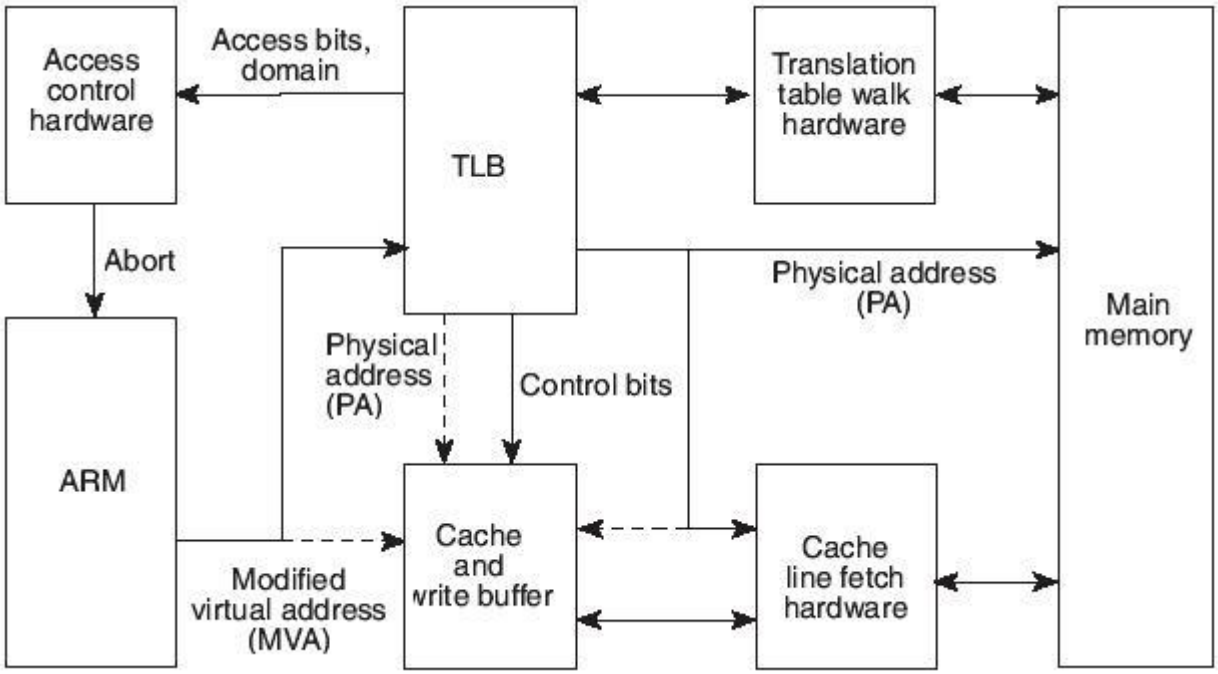


Рис. 4.15. Схема доступа к кэшированной памяти

Как только запись в TLB произведена, заключённая в ней информация используется следующим образом (рис. 4.15):

1. Для управления работой кэш-памяти и буферов записи используются биты C (*cachable*) и B (*bufferable*). Эти биты запрещают или разрешают кэширование и буферизацию соответствующей области памяти.
2. Память с целью контроля прав доступа делится на 16 доменов (D15-D0). Права доступа к доменам устанавливаются в предназначенном для этого 32-разрядном регистре управления доступом к доменам – *Domain Access Control Register*. Доступность текущего домена с идентификатором *domain* определяется по состоянию соответствующих бит в дескрипторах, используемых на различных уровнях трансляции адреса. Если произошло обращение к домену с нарушением прав доступа, процессору направляется сигнал об особой ситуации.
3. Если кэширование памяти отсутствует, то напрямую работает с основной памятью, используя генерируемые процессором физические адреса.

Управление памятью осуществляется посредством её разбиения на секции или страницы.

Секция (*section*) – это область памяти, составленная из блоков размером в 1 Мбайт.

Страницы (*pages*) могут быть трёх размеров:

Tiny pages – блок памяти размером в 1 Кбайт.

Small pages – блок памяти размером в 4 Кбайт.

Large pages – блок памяти размером в 64 Кбайт.

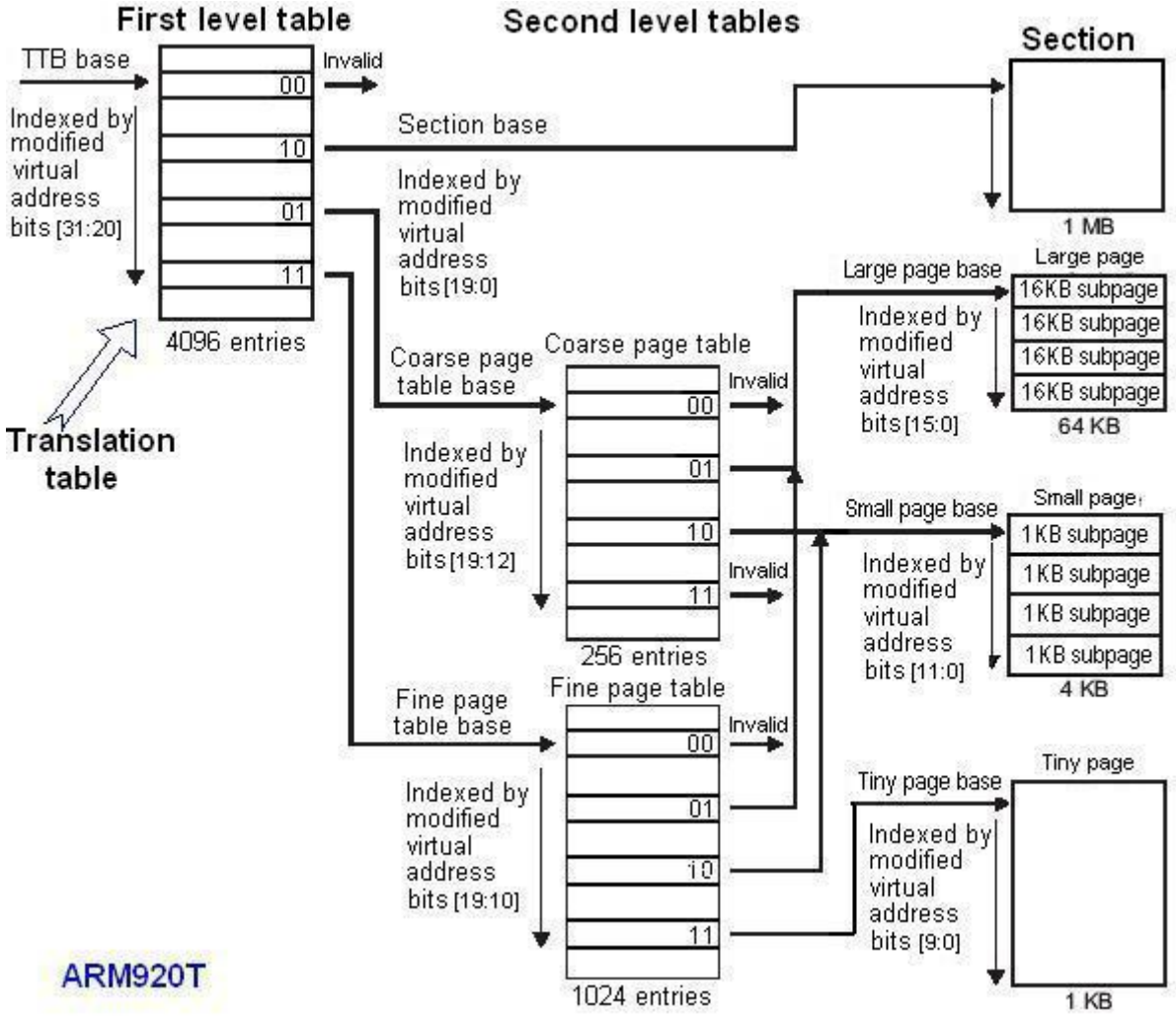


Рис. 4.16. Схема трансляции физического адреса

Адресная таблица расположена в основной памяти и имеет два уровня (рис. 4.16). При разбиении памяти на секции используется только один (*first-level-table*) уровень; при страничной адресации добавляется ещё один (*second-level-tables*) уровень:

First-level-table – содержит дескрипторы секций, а также указатели на дескрипторы страниц в таблице второго уровня;

Second-level-tables – таблица дескрипторов больших (*large*) и малых (*small*) страниц (*course page table*) и таблица больших, малых и «очень малых» (*tiny*) страниц (*fine page table*).

Что касается адресной таблицы первого уровня, то её базовый физический адрес (*Translation Table Base* – TTB) размещается в специальном регистре процессора. При этом значение имеют только старшие биты [31:14], а младшие [13:0] равны нулю.

Биты [31:14] регистра базы адресной таблицы объединяются со старшими битами [31:20] генерируемого процессором виртуального или модифицированного виртуального адреса, младшие [13:0] биты которого равны нулю (*Should-Be-Zero* – SBZ), и тем самым образуют 32-разрядный физический адрес дескриптора первого уровня (рис. 4.17).

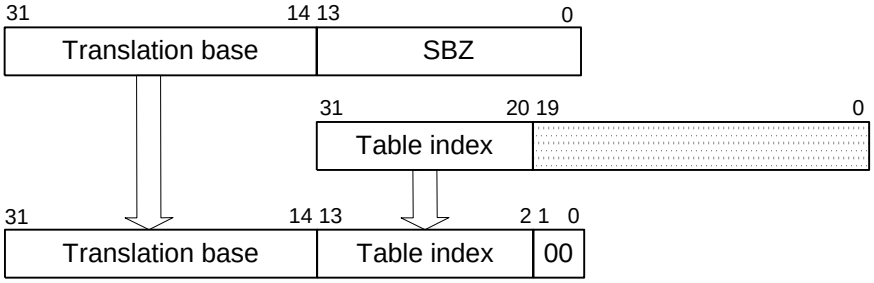


Рис. 4.17. Формирование физического адреса дескриптора первого уровня

Адресная таблица первого уровня составлена из 32-разрядных **дескрипторов первого уровня** (*first-level descriptor*). Есть четыре типа дескрипторов первого уровня. Различают их по комбинации младших бит [1:0]:

- Биты [1:0] == 0b00. Это означает, что виртуальный адрес не отображён на физическую память. При попытке обращения по этому адресу генерируется сигнал ошибки трансляции адреса «MMU *fault*», который воспринимается и обрабатывается CPU.
- Значения бит [1:0] == 0b10 соответствуют **дескрипторам секций**.
- Если бит [0] == 1, то при трансляции адреса помимо таблицы первого уровня используется адресная таблица второго уровня. При этом физический адрес дескрипторов второго уровня определяется по дескриптору первого уровня.
  - Если бит [1] == 0, то память дробится на крупноразмерные страницы (*coarse page*), к числу которых относятся *small pages* и *large pages*.
  - Если бит [1] == 1, память разбивается на страницы малого размера; такими страницами являются *tiny pages*.

  Следует заметить, что объём памяти, занимаемый таблицей для страниц большого размера, меньше 1 Кбайта, в то время как на одну таблицу страниц малого размера приходится 4 Кбайта.

Как уже отмечалось, всё виртуальное адресное пространство в ARM архитектуре разбито на 16 одинаковых по размеру доменов, каждый из которых состоит из определенного количества секций или страниц. Принадлежность секций (страниц) тому или иному домену устанавливается двумя битами в дескрипторах первого уровня. Доступ к каждому домену контролируется соответствующим 2-битовым полем в регистре управления доступом к доменам (*Do-*

main Access Control Register). Поддерживаются два вида доступа: со стороны клиента (*client*) и со стороны менеджера (*manager*). Первый вид доступа обеспечивает установленные пользовательские права доступа к секциям (страницам) домена, второй позволяет контролировать эти права и управлять ими.

Для отображения виртуального адреса в физический при разбиении памяти на секции достаточно дескрипторов, содержащихся в таблице первого уровня (рис. 4.17).

Для адресации страниц дополнительно используются **дескрипторы второго уровня**. **С**хема трансляция адреса дескрипторов страниц (дескрипторов второго уровня) не зависит от размера страниц. Однако некоторые особенности все-таки имеются и состоят они в том, что дескрипторы страниц в 1 Кбайт индексируются 10-разрядным кодом (биты [19:10]), взятым из соответствующего поля виртуального адреса, в то время как для адресации дескрипторов других страниц используется 8-ми разрядный код (биты [19:12]).

Формат дескрипторов второго уровня зависит от того, страницы какого размера он описывает. Имеются четыре типа дескрипторов второго уровня, принадлежность к каждому из которых устанавливается по младшим битам [1:0]:

1. Если биты [1:0] == 0b00, то виртуальный адрес не может быть отображен на физическую память и при попытке обращения по этому адресу генерируется сигнал ошибки трансляции адреса «MMU *fault*», который воспринимается и обрабатывается CPU.
2. Дескриптор, у которого биты [1:0] == 0b01, описывает страницы размером в 64 Кбайт.
3. Если биты [1:0] == 0b10, то такой дескриптор описывает страницы размером в 4 Кбайт.
4. Страницы размером в 1 Кбайт описывают дескрипторы, у которых биты [1:0] == 0b11.

Данные и инструкции в памяти выровнены по границам 32-разрядного слова, так как основной набор инструкций процессоров ARM имеет 32-разрядный формат. При этом минимальной единицей памяти остается 1 байт.

Особые ситуации при обращении к памяти

При трансляции исполнительного адреса в физическую память может случиться так, что область памяти, к которой происходит обращение, оказывается недоступной. (Считаем, что на системной шине только один ведущий – микропроцессор). Такие случаи фиксируются как особые (исключительные) ситуации. При их возникновении процессору направляются сигналы, получив которые CPU останавливает выполнение текущей программы, анализирует причины возникновения связанного с особой ситуацией сигнала и предпринимает действия для их устранения. Особые ситуации могут быть вызваны

1. ошибками, обнаруживаемыми системой управления памятью (MMU *faults*) и, в частности, тогда, когда нарушаются установленные на доступ к памяти ограничения, и
2. ошибками, возникающими во внешней системе памяти (*external aborts*), например, при несанкционированном доступе или при обращении по недействительному адресу.

Обобщенно MMU *faults* и *external aborts* для краткости обозначают одним термином «*aborts*».

Система управления памятью фиксирует четыре типа ошибок:

- ошибки, связанные с нарушением границ выравнивая в памяти по 32-разрядным словам (*alignment fault*),
- ошибки трансляции адреса (*translation fault*),
- ошибки доступа к доменам (*domain fault*) и
- ошибки, связанные с нарушением прав доступа к страницам или секциям (*permission fault*).

Система памяти сигнализирует о трёх типах особых ситуаций, связанных:

- с обращением к недействительной линии кэш-памяти (*line fetches*),
- с обращением к основной памяти (*memory accesses*) – память некэширована и небуферизована,
- с доступом к используемым при трансляции адреса таблицам (*translation table accesses*).

Логика, следуя которой MMU контролирует обращения к памяти, представлена на рис. 4.18. Из рисунка видно, что есть различия в последовательности и логике принимаемых решений при работе с памятью, разбитой на секции и с памятью, имеющей страничную организацию.

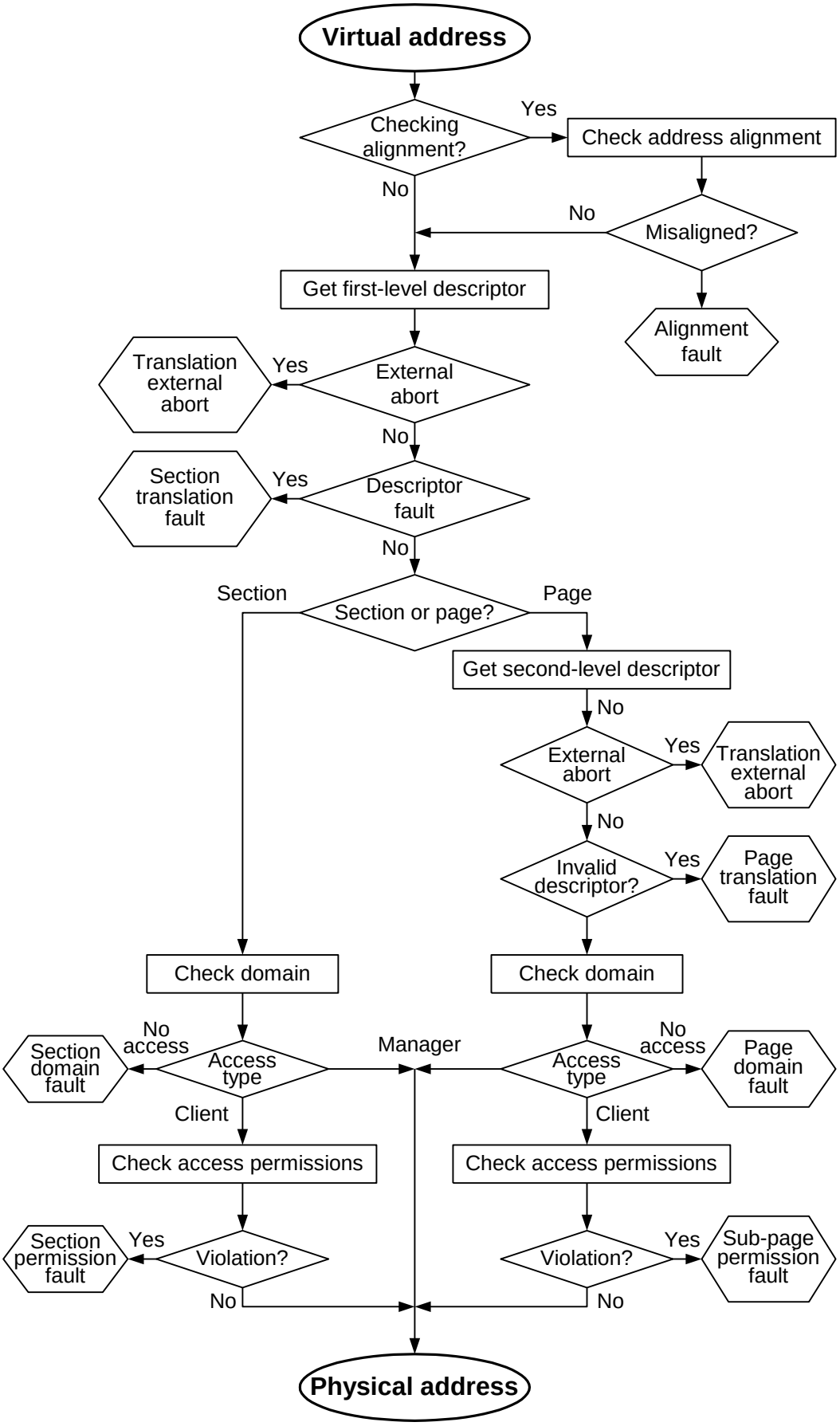


Рис. 4.18. Логика управления памятью

Семафоры

Для работы с семафорами используются специальные Swap (SWP) и Swap Byte (SWPB) инструкции, которые позволяют обращаться к памяти в едином неделимом цикле чтения-записи. Два нижеследующие примера демонстрируют особенности использования семафоров при доступе к разделяемым ресурсам памяти.

- Если ресурсы системной шины разделяют несколько ведущих (несколько процессоров), то использование Swap инструкций по отношению к семафорам позволяет реализовать бесконфликтное взаимодействие между различными ведущими.

  В этом случае семафоры должны размещаться в некэшируемой области памяти, использование Swap инструкций даёт возможность блокирования доступа к шине со стороны других использующих механизм семафоров ведущих во время выполнения операции чтение-запись.
- В многопотоковых (*multithreads*) однопроцессорных системах с помощью Swap инструкций контролируется взаимодействие между потоками (процессами).

  В этом случае нет необходимости в блокировании доступа к шине во время операции чтение-запись и семафоры могут быть расположены в кэшируемой области памяти. Использование Swap и Swap Byte инструкций позволяет сделать механизм семафоров более быстродействующим.

4.4. Микропроцессоры с ядром ARM10E

Продолжая начатый в разделе 4.1 обзор ARM микропроцессоров, рассмотрим ещё одну их разновидность, в основу которой положено целочисленное ядро ARM10E™. В качестве примера на рис. 4.19 представлена структурная схема МП ARM1020E.

В состав МП ARM1020E входят:

- целочисленное ядро, включающее
- блок загрузки/хранения (*Load/Store unit*),
- блок предварительной выборки инструкций (*Prefetch unit*),
- блок целочисленных вычислений (*Integer unit*);
- встроенные сопроцессоры C14 и C15 и интерфейс с внешними сопроцессорами;
- два устройства управления памятью (*Instruction* MMU и *Data* MMU);
- кэш-память данных (*Data cache*) и кэш-память команд (*Instruction cache*);
- буфер записи в память (*Write buffer*);
- схема согласования шин AMBA-AHB (*Advanced Microcontroller Bus Architecture* – AMBA, *Advanced High-performance Bus* – AHB), которая является интерфейсом с внешней магистралью AHB;

– встроенная система отладки на основе JTAG интерфейса (*Embedded*ICE-RT *logic for* JTAG-*based debug*) с TAP (*Test Access Port*) контроллером. Опционально процессор ARM1020E может быть связан с внешней схемой трассировки (*Embedded Trace Macrocell* – ETM) для отслеживания в реальном времени потоков инструкций и данных во встроенной системе.

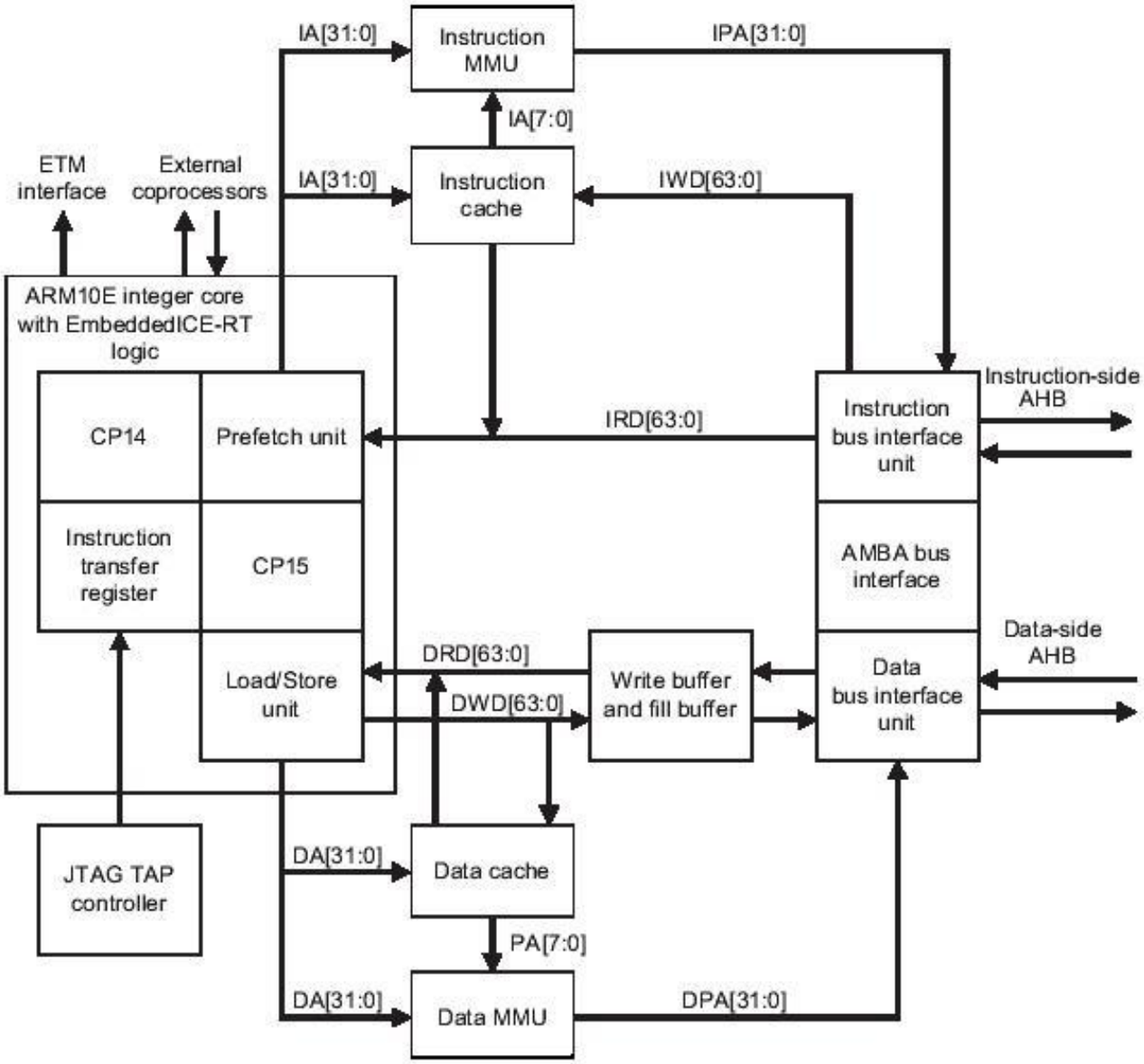


Рис. 4.19. Структурная схема микропроцессора ARM1020E

Каждая линия кэш-памяти данных имеет тэг виртуального и тэг физического адресов. Память тегов физических адресов (*write-back Physical Address* (PA) TAG RAM) используется для реализации режима обратной записи моди-

фицированной строки кэш-памяти данных в основную память с целью поддержки когерентности кэш-памяти данных и ОЗУ.

Особенностью процессора является 64-разрядный интерфейс между ядром кэш-памятью, буфером записи и AMBA-AHB интерфейсом. Ядро принимает инструкции по однонаправленной шине IRD[63:0] (*Instruction Read Data*). Для обмена данными используются шины DRD[63:0] (*Data Read Data*) и DWD[63:0] (*Data Write Data*).

Генерируемые ядром 32-разрядные виртуальные адреса инструкций и данных направляются соответственно в кэш-память команд и кэш-память данных, а также в устройства управления памятью MMU по шинам IA[31:0] и DA[31:0]. Физические адреса инструкций и данных передаются во внешнюю среду через AMBA-AHB интерфейс. Для этого используются шины IPA[31:0] (*Instruction Physical Address*) и DPA[31:0] (*Data Physical Address*). Обновление кэш-памяти команд выполнятся через шину IWD[31:0] (*Instruction Write Data*), а кэш-памяти данных – через шину DWD[63:0].

Целочисленное ядро МП ARM1020E

Ядра МП ARM1020E (структурная схема представлена на рис. 4.20) поддерживает конвейер операций и в состав процессора входят:

- **блок предварительной выборки** инструкций (*Prefetch unit*). Предназначением блока является выборка инструкций из кэш или основной памяти, а также предсказание ветвлений, что позволяет уменьшить число перезагрузок конвейера операций;
- **целочисленных блок** (*Integer unit*) – предназначен для декодирования и исполнения инструкций, поступающих из блока предварительной выборки. В составе блока имеются сдвигатель, арифметическо-логическое устройство (ALU) и умножитель. Целочисленный блок участвует в вычислении исполнительных адресов при выполнении операций загрузки/хранения. В нём находится программный автомат, управляющий последовательностью исполнения инструкций, включая поддержку циклических операций, изменение режима работы процессора, реакцию на исключительные ситуации и события при отладке системы;
- **блок загрузки/хранения** (*Load/Store unit* – LSU) – при 64-битовом выравнивании данных в памяти способен загружать (сохранять) два регистра (64 бита) за один цикл. При цепочечных передачах после первого обращения к памяти по адресу, вычисленному целочисленным блоком, последующие обращения к памяти могут быть выполнены в автономном режиме. Целочисленный блок свои операции по обработке данных может выполнять параллельно с блоком загрузки/хранения;
- **многопортовый регистровый блок** – связан с целочисленным блоком и блоком загрузки хранения через систему мультиплексоров.

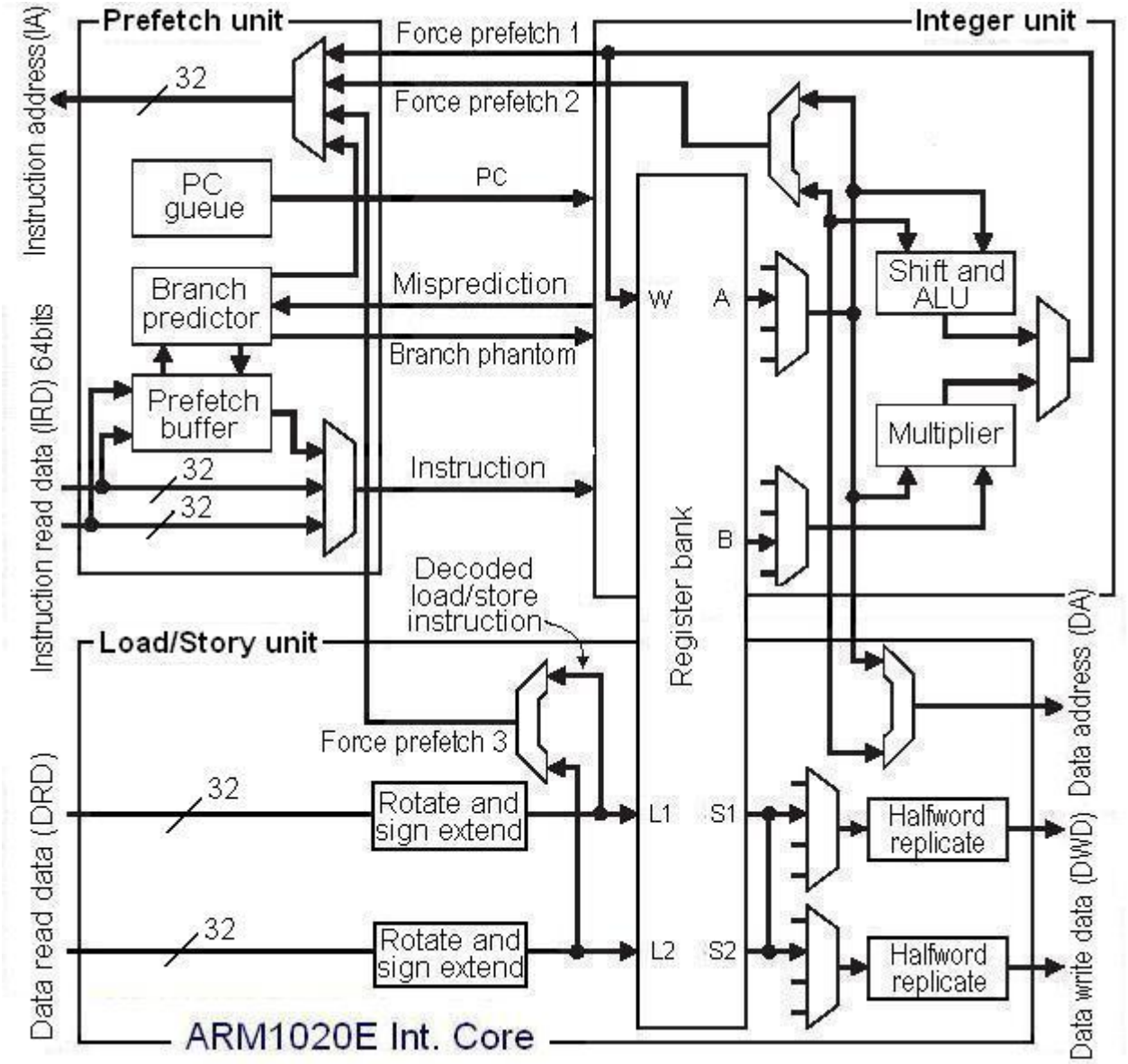


Рис. 4.20. Структурная схема ядра микропроцессора ARM1020E

В ядро ARM10 встроен конвейер операций, на шести уровнях которого выполняются следующие действия:

1. **выборка** (*Fetch*) инструкций из кэш-памяти команд и предсказание ветвления путём просмотра предварительно выбранных инструкций;
2. **предварительное декодирование** (декодирование предварительно выбранной инструкции – *Issue*);
3. **декодирование инструкции** (*Decode*), подлежащей исполнению, чтение регистров, используемых в операциях ALU, транспортировка сигналов внутренних блокировок и их разрешение;
4. **выполнение инструкции** (*Execute*) – вычисление исполнительного адреса, выполнение сдвиговых операций, операций ALU и начальных действий при умножении, установка флагов операций, проверка кода условия, определение выполнимости (*Misprediction*) предсказанного ветвления (*Branch phantom*) и сохранение дешифрированного регистра данных;

5. **доступ** к кэш-памяти данных (*Memory*) и заключительная стадия операции умножения;
6. **запись** (*Write*) в регистр результата операций целочисленного блока, загрузка регистра из памяти, изъятие не подлежащей выполнению предварительно выбранной инструкции (например, из-за ошибочного предсказания ветвления).

На стадии выборки используется FIFO (*First-In-First-Out buffer*) буфер для выбираемых команд, который может хранить до трёх инструкций. Используется он для того, чтобы во время исполнения текущей инструкции была возможность обнаружения находящеёся среди выбранных команд инструкции ветвления и тем самым возможность предсказания ветвления.

На стадиях предварительного декодирования и декодирования инструкции параллельно дешифрируются инструкции возможных ветвлений и инструкции, подлежащие исполнению.

На стадиях выполнения, доступа к памяти и записи в регистр (загрузки регистра) одновременно выполняются:

- предсказание ветвлений;
- операции ALU и умножителя;
- операции загрузки/сохранения, в том числе и многоцикловые операции, связанные с исполнением инструкций загрузки/сохранения нескольких регистров;
- многоцикловые операции записи в регистры сопроцессора CP15.

Шины DRD и DWD являясь 64-разрядными внутри ядра разбиваются на пары 32-разрядных шин, по которым производится обмен данными с регистровым блоком (порты L1, L2 и S1, S2), имеющим 32-разрядные регистры.

При выборке инструкций адрес предварительной выборки (*Force prefetch* 1-3) может быть результатом вычислений, производимых целочисленным блоком (*Force prefetch* 1, 2), или результатом декодирования команд загрузки/хранения, выполняемого блоком загрузки/хранения (*Force prefetch* 3). Последнее относится к случаю, когда при возврате из процедур извлекается содержимое стеков (в многопотоковой среде каждый процесс имеет свой стек), где было сохранено значение счётчика команд.

Конвейер, в котором производятся операции с данными и адресами, требуется механизм для сохранения адресов задержанных переходов. Частью этого механизма является очередь значений счётчика команд (PC *queue*).

4.5. Исключительные ситуации и регистры ARM процессоров

При работе системы возникают различные события, которые требуют соответствующей реакции со стороны процессора. События связаны как с процессами, происходящими внутри процессора, так и с тем, что происходит во внешней по отношению к нему среде. Во всей своей совокупности эти события

характеризуют как *исключительные ситуации* или просто *исключения*. К исключительным ситуациям относят и те, что связаны с запросами прерываний.

Регистровая модель процессора, казалось бы никак не связана с исключительными ситуациями. Однако это не совсем так. Некоторые адресные регистры и регистры данных имеют альтернативные версии. В этом случае говорят о первичных и вторичных регистрах. Наличие вторичных регистров позволяет при одном уровне вложения процедур не прибегать к сохранению первичных регистров в памяти, а воспользоваться вторичными, работа с которыми ничем не отличается от работы с первичными. Это имеет важное значение для системы прерываний, поскольку позволяет в реальном времени обслуживать запросы от отдельного выделенного устройства (например, от аналого-цифрового преобразователя).

Вторичные регистры имеются не у всех, а лишь у некоторых выделенных регистров регистрового блока. Число выделенных регистров зависит от режима работы процессора. Выделенных регистров нет в режиме обычного пользователя и имеются они только в привилегированных режимах. При этом число вторичных регистров зависит от уровня привилегий.

Обращение к ARM процессорам при рассмотрении связанных с исключительными ситуациями вопросов, обусловлено тем, что в свойственную этим процессорам систему обслуживания исключительных ситуаций и запросов прерываний заложен механизм, имеющий общее значение для микропроцессорных систем.

Система прерывания ARM процессоров

Систему управления прерываниями процессоров семейства ARM в значительной степени ориентирована на применение во встроенных системах. Примером будет 16/32-разрядный RISC-микроконтроллер K532C50100/5000A (прототипом является NetARM-II) фирмы Samsung, предназначенный для сетевых приложений.

Поскольку систему прерываний целесообразно изучать во взаимодействии CPU с внешними устройствами и памятью, то в рассмотрение возьмём конкретную микросистему – платформу для разработки программного обеспечения для микропроцессоров NetARM-II SNDS100 (*Samsung* NetARM *Development System*).

Карта памяти системы SNDS100 дана на рис. 4.21. Выделить нужно два типа памяти – постоянную (ПЗУ или ROM) и оперативную (ОЗУ или RAM). Постоянная память – это электрически перепрограммируемое запоминающее устройство (флэш-память), в которой хранится код инициализации системы, а также тот код, под управлением которого система работает (*boot code*). Кроме того, в карте памяти показано адресное пространство, принадлежащее портам ввода/вывода.

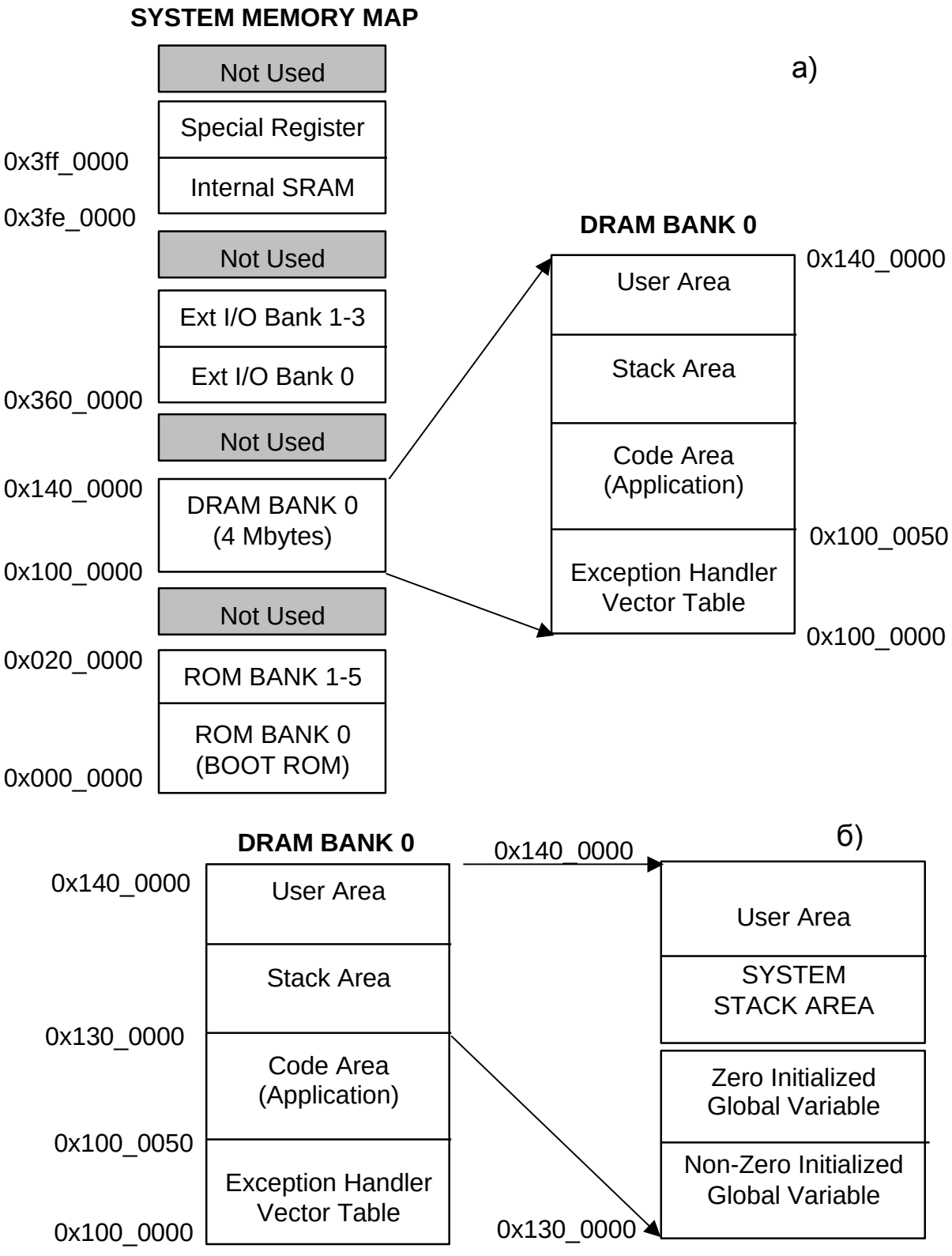


Рис. 4.21. Карта памяти микросистемы SNDS100

ПЗУ в представленном на рис. 4.21 распределении адресного пространства начинается с нулевого адреса. Это не является общим правилом. ПЗУ может быть расположено и по старшим адресам (на дне памяти). Более того, после загрузки системы на первоначально принадлежащее ПЗУ адресное пространство

верхней области памяти может быть отображено ОЗУ. Необходимость в этом бывает вызвана тем, что в постоянной памяти располагается (начиная с нулевого адреса) таблица векторов прерывания. Однако ПЗУ обычно обладает большим временем доступа в сравнении с ОЗУ и, кроме того, при вызовах процедур обработки прерываний через расположенную в ПЗУ таблицу увеличивается время реакции системы на запросы прерывания. В том случае, когда это имеет значение, оперативную память отображают на верхнюю (начинающуюся с нулевого адреса) область памяти, замещая таким образом находившееся там ПЗУ и размещая теперь уже в ОЗУ таблицу прерываний. Модель памяти, представленная на рис. 4.21 таких действий не предусматривает.

При выполнении программы инструкции из памяти выбираются последовательно по программному счётчику. С помощью программного счётчика осуществляются также переходы и вызовы. Нормальный ход выполнения программы может быть нарушен исключительными ситуациями (*exceptions*), обусловленными внутренними или внешними событиями в системе. В ответ на эти события генерируются сигналы, на которые процессор так или иначе реагирует. К числу таких событий относятся, например:

- запросы прерываний от устройств ввода/вывода,
- попытка процессора выполнить неопределенную инструкцию (*undefined instruction*),
- обращение пользовательских заданий к привилегированной функции операционной системы и др.

Таблица 4.1. Перечень исключительных ситуаций для ARM процессоров

Vector address	**Exception type**	**Exception mode**	**Priority (1=high, 6= low)**
0x0	Reset	Supervisor (SVC)	1
0x4	Undefined Instruction	Undef	6
0x8	Software Interrupt (SWI)	Supervisor (SVC)	6
0xC	Prefetch abort	Abort	5
0x10	Data abort	Abort	2
0x14	*Reserved*	*Not applicable*	*Not applicable*
0x18	Interrupt (IRQ)	Interrupt (IRQ)	4
0x1C	Fast Interrupt (FIQ)	Fast Interrupt (FIQ)	3

Получив сигнал о произошедшем событии, процессор переходит на обработку исключительной ситуации (в дальнейшем – исключения), предварительно сохраняя информацию о своем текущем состоянии. Исключительные ситуа-

ции различаются по типу (таблица 4.1) и зависят от режима (*mode*), в котором они возникли. Для ARM процессоров различают:

1) режим супервизора (*supervisor*),
2) режим аварийного прекращения выполнения программы (*abort*) из-за ошибки в процессе предварительной выборке инструкции из памяти (*prefetch abort*) или ошибки обращения по адресу данных (*data abort*),
3) режимы обычных (*Interrupt* – IRQ) и быстрых (*Fast Interrupt* – FIQ) прерываний.

Более подробно исключительные ситуации описаны в таблице 4.2.

Таблица 4.2. Описание исключительных ситуаций для ARM процессоров

Exception	**Описание**
Reset	Происходит, когда сигнал сброса поступает на соответствующий контакт микросхемы CPU. Сигнал генерируется при включении источника питания или при аппаратной начальной установке процессора. Сброс может быть выполнен программно через вектор (0x0000) в таблице прерываний.
Undefined Instruction	Имеет место, если ни процессор, ни сопроцессор (если таковой имеется) не распознают текущую инструкцию.
Software Interrupt (SWI)	Программное прерывание не является асинхронным и инициируется в процессе исполнения программы. Программные прерывания обычно используются при запросах привилегированных операций, выполняемых в режиме супервизора. К их числу таких операций относятся, например, те которые поддерживаются функциями RTOS.
Prefetch Abort	Возникает при попытке процессора выполнить инструкцию, которая не была предварительно извлечена из памяти из-за обращения по недействительному адресу инструкций [2].
Data Abort	Возникает, когда при исполнении инструкция записи или чтения памяти произошло обращение по недействительному адресу [9].
IRQ	Происходит, когда процессору передаются запросы прерываний от внешних устройств ввода/вывода через соответствующие контакты микросхемы процессора. Процессор воспринимает эти запросы в том случае, если они разрешены установкой соответствующего бита в слове состояния процессора.
FIQ	Происходит, когда процессору передаются запросы прерываний от внешних устройств ввода/вывода через соответствующие контакты микросхемы процессора. Процессор воспринимает эти запросы в том случае, если они разрешены установкой соответствующего бита в слове состояния процессора.

Таблица векторов прерывания расположена в предназначенной для неё области памяти и занимает 32 байта, начиная с нулевого адреса. Каждому эле-

[2] Это может произойти из-за отсутствия на текущий момент возможности отображения виртуального адреса в физический или при нарушении прав доступа к соответствующей области памяти из-за несоответствия этих прав и текущего режима процессора.

менту таблицы соответствует одно 32-разрядное слово. В нём обычно размещается команда перехода на процедуру обработки. Отметим, что вектор прерываний для FIQ запросов, требующих быстрой реакции в таблице располагается последним. В силу этого программу обработчика FIQ можно размещать сразу же после таблицы (включая и её последний элемент). Это позволяет существенно сократить время перехода на процедуру обработки быстрых прерываний.

Как правило, приложения исполняются в пользовательском режиме (*user mode*), но обслуживание этих приложений требует привилегированных операций, которые не могут быть выполнены в режиме пользователя. Исключения изменяют режим процессора. Регистровый блок ARM процессоров имеет 16 32-разрядных регистров R0 – R15 (рис. 4.22). Особенностью регистрового блока является то, что некоторая часть регистров для каждого из режимов процессора имеет *альтернативные* (*вторичные*) *регистры*. Выполняются такие регистры не в единичном экземпляре, а с подключением дополнительного набора (банка) регистров. При этом число регистров в таком наборе определяется режимом работы процессора. В дальнейшем такие регистры будем называть банкированными (*banked registers*).

Регистры процессора и их организация показаны на рис. 4.22. При этом банкированными регистрами являются

- регистр R13, или указатель стека SP_*mode* (*Stack Pointer*),
- регистр R14, или регистр связи LR_*mode* (*Link Register*),
- регистр хранения состояния программы SPSR_*mode* (*Saved Processor Status Register*)

В FIQ-режиме обработчику прерываний доступны ещё 5 альтернативных регистра (R8_FIQ – R12_FIQ).

Текущее состояние процессора заключено в регистре CPSR (*Current Processor Status Register*). Регистр R15 – это программный счетчик (PC). В регистре связи R14 (или LR) сохраняется адрес возврата (содержимое PC) при вызовах процедур обработки прерываний, чем и обусловлено такое название R14.

Таким образом, при переходе процессора на обработку исключений текущее содержимое банкированных регистров автоматически сохраняется в соответствующем режиму альтернативном регистре. Все остальные регистры при необходимости должны сохраняться в стеке. При этом каждому режиму процессора выделяется своя область стековой памяти.

На то, что каждому из режимов соответствует свой альтернативный регистр, показывает суффикс «*mode*» в именах регистров. На рис. 4.22 этот суффикс представлен обозначающими режим аббревиатурами. Однако в командах Ассемблера соответствие регистров тому или иному режиму не показывается и используются обычные имена регистров R0 – R12, R13 (или SP), R14 (или LR) и R15 (или PC) без суффиксных определений (действует правило «по умолчанию»).

MODES

Privileged modes

Exception modes

User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Показывает, что обычные регистры, используемые в режимах «User» и «System», в других режимах работы могут быть замещены альтернативными регистрами.

Рис. 4.22. Регистры ARM процессоров.

Механизм банкированных регистров позволяет значительно сократить время переключения контекста, особенно в тех случаях, когда при обработке исключений не используются другие регистры, кроме банкированных. Если необходимость в использовании других регистров всё же имеется, скорость переключения контекста можно повысить за счёт инструкций групповой загруз-

ки/сохранения регистров процессора. Действие таких инструкций рассмотрим позднее при обсуждении последовательности вызова процедур обработки прерываний.

Приоритеты исключительных ситуаций

Когда несколько исключительных ситуаций случаются одновременно, обработка их должна вестись в соответствии с установленными для них приоритетами. Распределение исключений по приоритетам показано в таблице 6.1.

Реакция процессора на исключения

Когда процессор получает сигнал о произошедшем в связи с возникшей исключительной ситуацией событии, он выполняет следующие действия:

1. Сохраняет текущее состояния процессора (регистр `CPSR`) в регистре `SPSR`, соответствующем режиму обработки исключения. Это обусловлено необходимостью сохранения идентификатора текущего режима, флагов операционного блока и бит маскирования прерываний при переходе процессора на обработку исключения.
2. Содержимое `CPSR` модифицируется в соответствии с возникшей исключительной ситуацией. Это делается для того, чтобы:
   - изменить соответствующим образом режим процессора, отобразить `CPSR` на соответствующий вновь устанавливаемому режиму `SPSR` и
   - запретить прерывания. (При этом IRQ прерывания запрещаются в любых исключительных ситуациях, а FIQ прерывания запрещаются только при наличии FIQ запросов и при сбросе процессора.)
3. В регистр связи `LR_`*mode* помещается адрес возврата (содержимое `PC`).
4. В программном счетчике `PC` устанавливается адрес, взятый из таблицы прерываний, что обеспечивает переход на соответствующую исключению процедуру обработки.

В случае использования в процессе обработки исключения других (небанкированных) регистров эти регистры должны сохраняться в стеке. Направленные на сохранение регистров действия выполняет процедура обработки. Для этого предусмотрены специальные инструкции сохранения в памяти группы регистров:

```
STMFD sp!,{reglist}.
```

Не вдаваясь в подробное описание этих инструкций, заметим, что аббревиатура `STMFD` соответствует словосочетанию «*Story Memory + Full Descending*» и обозначает сохранение в стеке со смещением вершины стека в сторону меньших адресов и с указателем стека, показывающим на его последний элемент[3]. Итоговый адрес вершины стека (значение которого определяется содержимым ука-

[3] Допускаются и иная организация стековой памяти, что соответствующим образом отображается в командах Ассемблера.

зателя стека) после сохранения всех регистров из списка reglist сохраняется в SP. На это показывает суффикс «!» при SP.

Действия процессора при возврате из процедуры обработки исключения зависят от того, производились или нет операции со стеком при её вызове. Содержимое регистров можно восстановить, используя инструкцию групповой загрузки из стека. Например, при возврате из процедуры обработки исключения регистры (включая и программный счетчик, если нельзя ограничиться одним уровнем вложения процедур обработки) можно восстановить одной инструкцией

```
LDMFD sp!,{r0-r12,pc}^.
```

Заметим, что в этой инструкции к списку регистров добавлен квалификатор ^, который в команду включать не обязательно. Квалификатор ^ показывает, что при возврате из процедуры обработки исключения наряду с перечисленными в списке регистрами по содержимому SPSR восстанавливается CPSR. Однако такой механизм действует только в привилегированных режимах, каковыми не являются режим пользователя и низкоприоритетный режим системы.

На рис. 4.23 представлена диаграмма, на которой отображены действия, необходимые для восприятия и обработки запросов прерывания. Эти действия предусматривают

1. инициализацию
   - регистра режима прерываний INTMOD с целью установки FIQ или IRQ режима,
   - регистра маски прерываний INTMSK,
   - регистра прерываний INTPND (*в соответствии с аббревиатурой* INTPND *следовало бы сказать «регистра отложенных прерываний», поскольку в этом регистре фиксируются все незамаскированные запросы прерываний, но обрабатывается тот из запросов, который обладает наивысшим приоритетом в текущий момент времени*),
2. установку приоритетов (регистры INTPRI 0-5), если неприемлемо их распределение по умолчанию (*число регистров* 6*, поскольку контроллер прерываний может воспринимать запросы от* 21 *источника и для их кодирования по уровням приоритета используется битовое поле размером в шесть* 32*-разрядных слов*),
3. установку вектора прерываний ISR в таблице прерываний (*для этого используется системная функция, например, такая, как представленная на* рис. 4.23 *функция* SysSetInterrupt(index, void(*handler()))),
4. разрешение всех незамаскированных в регистре маски прерываний с помощью соответствующего бита INTMSK[20] в регистре INTMSK и разрешение прерываний путём установки соответствующих бит в регистре состояния программы CPSR.

Рис. 4.23. Последовательность действий при обработке исключительных ситуаций

Передаваемый в `SysSetInterrupt( index, void(*handler()) )` параметр `index` идентифицирует источник прерываний и используется для обращения к имеющейся в процедуре обработки прерывания таблице, содержащей точки входа в те части программного кода ISR, которые отвечают за обработку прерывания от обозначенного `index`'ом источника. Как правило, элементами таблицы являются инструкции перехода по адресу конкретного обработчика в составе ISR. Поскольку обычные инструкции ARM процессора имеют размер в 32 бита, то смещение каждого элемента в таблице кратно 4-м байтам. Значение параметра `index` берётся из специального регистра `INTOFFSET` контроллера прерываний, и это значение соответствует тому из прерываний, которое на текущий момент времени обладает наиболее высоким приоритетом. Значение регистра `INTOFFSET` равно численному выражению смещения позиции процедуры обработки прерывания, вызываемой из функции (рис. 4.24)

```
void ISR_IrqHandler(void)
{
 intOffset=(U32)INTOFFSET;
 Clear_PendingBit(intOffset>>2);
(*InterruptHandlers[intOffset>>2])(); //Вызов процедуры
```

```
                                        //обработки прерывания.
}
```

Здесь `U32` обозначает беззнаковые 32-разрядные целые.

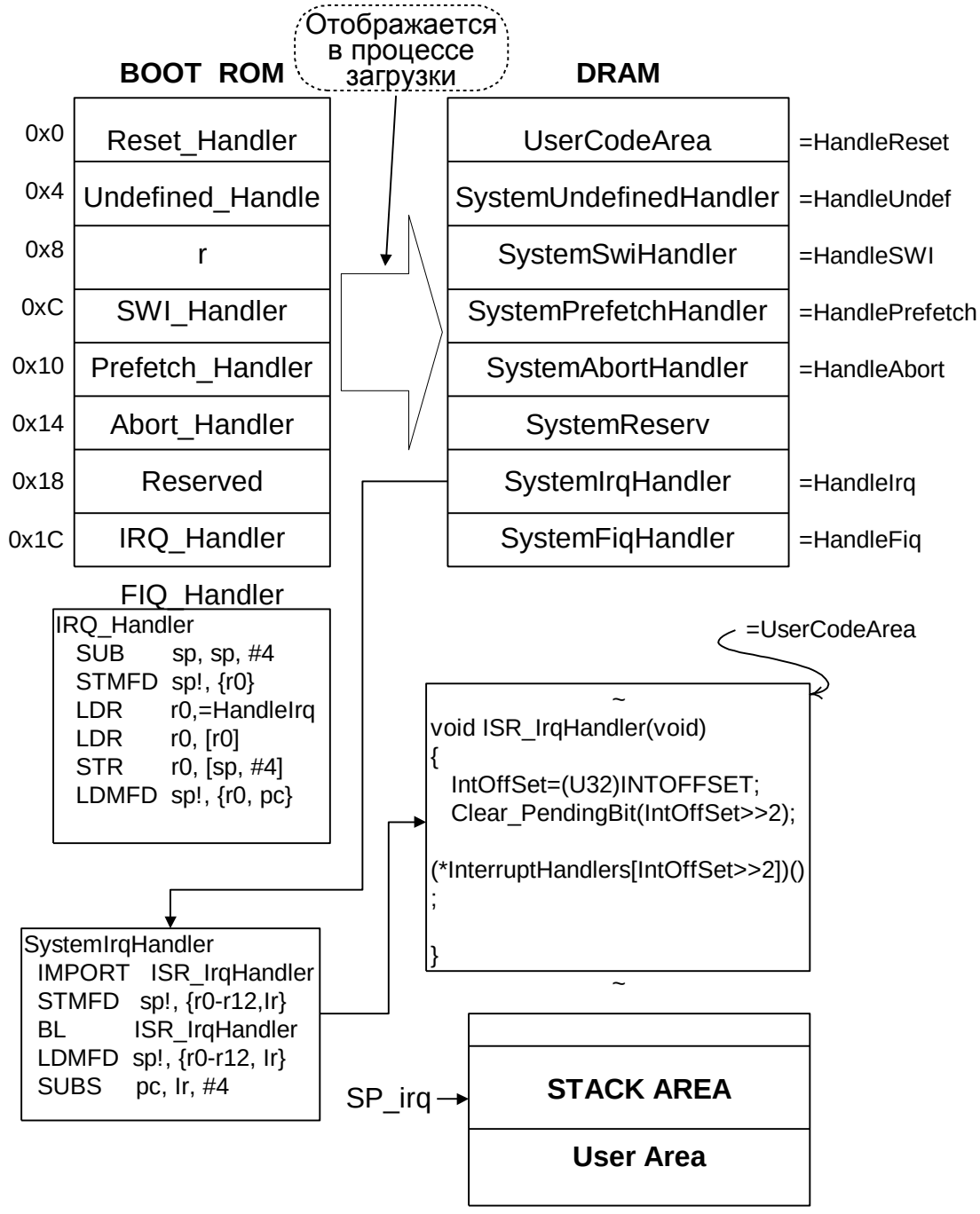


Рис. 4.24. Пример последовательности обработки IRQ прерываний

Пример последовательности обработки IRQ прерываний представлен на рис. 4.24. В соответствии с требованиями ARM архитектуры таблица векторов исключительных ситуаций (исключений) располагается в верхней области памяти, начиная с адреса 0x0 и кончая адресом 0x1c. Полагаем, что ПЗУ расположено в вершине памяти, начиная с нулевого адреса. В таком случае после начального сброса, расположенные в ПЗУ вектора исключительных ситуаций,

отображаются на предназначенное для них адресное пространство ОЗУ. Для этого используются обработчики исключений по умолчанию. На рис. 4.24 представлен только один из них - обработчик по умолчанию IRQ_Handler. С его помощью вектор IRQ_Handler отображается в вектор SystemIrqHandler, который представляет собой точку входа в следующую ступень обработки прерывания. Для этой цели используется последовательность инструкций, представленных в листинге 4.1.

Листинг 4.1

```
IRQ_Handler
  SUB   sp,sp,#4       ;(sp)=(sp)-4 без изменения флагов
                       ;операционного блока – смещение указателя
                       ;стека
  STMFD sp!,{r0}       ;Сохранить r0 в стеке
  LDR   r0,=HandleIrq  ;Загрузить в r0 указатель HandleIrq
  LDR   r0,[r0]        ;Загрузить в r0 вектор SystemIrqHahdler
  STR   r0,[sp,#4]     ;Сохранить r0 в стеке (адрес = (sp)+4)
  LDMFD sp!,{r0,pc}    ;Восстановить r0, а в pc поместить
                       ;вектор SystemIrqHandler
```

Здесь символ «#» обозначает непосредственное значение, а знак «=» в мнемокоде инструкции LDR – литерал (в данном случае адрес вектора обработчика прерывания).

После перехода на SystemIrqHandler происходит

1) сохранение регистров r0-r12 и регистра связи lr – инструкция STMFD sp!,{r0-r12,lr},
2) переход на процедуру обработки прерывания ISR_IrqHandler() с сохранением содержимого программного счетчика pc в регистре связи lr – инструкция BL ISR_IrqHandler,
3) вызов обработчика прерывания в соответствии со значением регистра INTOFFSET,
4) восстановление регистров процессора после возврата из обработчика прерывания – инструкция LDMFD sp!,{r0-r12,lr} и
5) возврат из процедуры посредством восстановления программного счетчика по содержимому регистра связи lr – инструкция SUBS pc,lr,#4, в соответствии с которой (pc)=(lr)-4. (Суффикс S после мнемоники команды вычитания SUB означает, что в результате операции модифицируются флаги операционного блока).

Вычитание из содержимого lr числа 4 на последнем этапе, связано с тем, что при переходе на обработку запроса IRQ процессор автоматически сохраняет в регистре связи (в lr_irq) уменьшенное на 4 значение программного счетчика (то есть pc-4). Тем не менее, данный процессор наряду с текущей производит предварительную выборку следующей инструкции и таким образом при каждом обращении к памяти из неё берется не одна, а две инструкции. Этим и обусловлена производимая коррекция lr. Нужно сказать, что принципиального значения это не имеет, а характеризует лишь особенности ARM процессоров.

Программные модули, которые отвечают за инициализацию системы обработки прерываний и установку их обработчиков с соответствующими комментариями представлены в листинге 4.2.

Листинг 4.2

```
/*-------------------------------------------------------------------------------------------------*/
/*InitIntHandlerTable: Инициализация таблицы обработчиков прерываний  */
/*    Примечание: Вызывается на стадии инициализации системы          */
/*-------------------------------------------------------------------------------------------------*/
void InitIntHandlerTable(void)
{
     REG32 i;

     for (i = 0; i < MAXHNDLRS; i++)
     InterruptHandlers[i] = DummyIsr;
}

/*-------------------------------------------------------------------------------------------------*/
/*SysSetInterrupt: Установить таблицу векторов обработчиков прерываний    */
/*-------------------------------------------------------------------------------------------------*/
void SysSetInterrupt(REG32 vector, void (*handler)())
{
     InterruptHandlers[vector] = handler;
}

/*-------------------------------------------------------------------------------------------------*/
/*InitInterrupt: Инициализировать таблице обработчиков прерываний          */
/*-------------------------------------------------------------------------------------------------*/
void InitInterrupt(void)
{
     ClrIntStatus();          // Очистить все запросы прерываний
     InitIntHandlerTable() ;
}
```

На рис. 4.25 дана временная диаграмма, поясняющая реакцию системы на запросы прерывания.

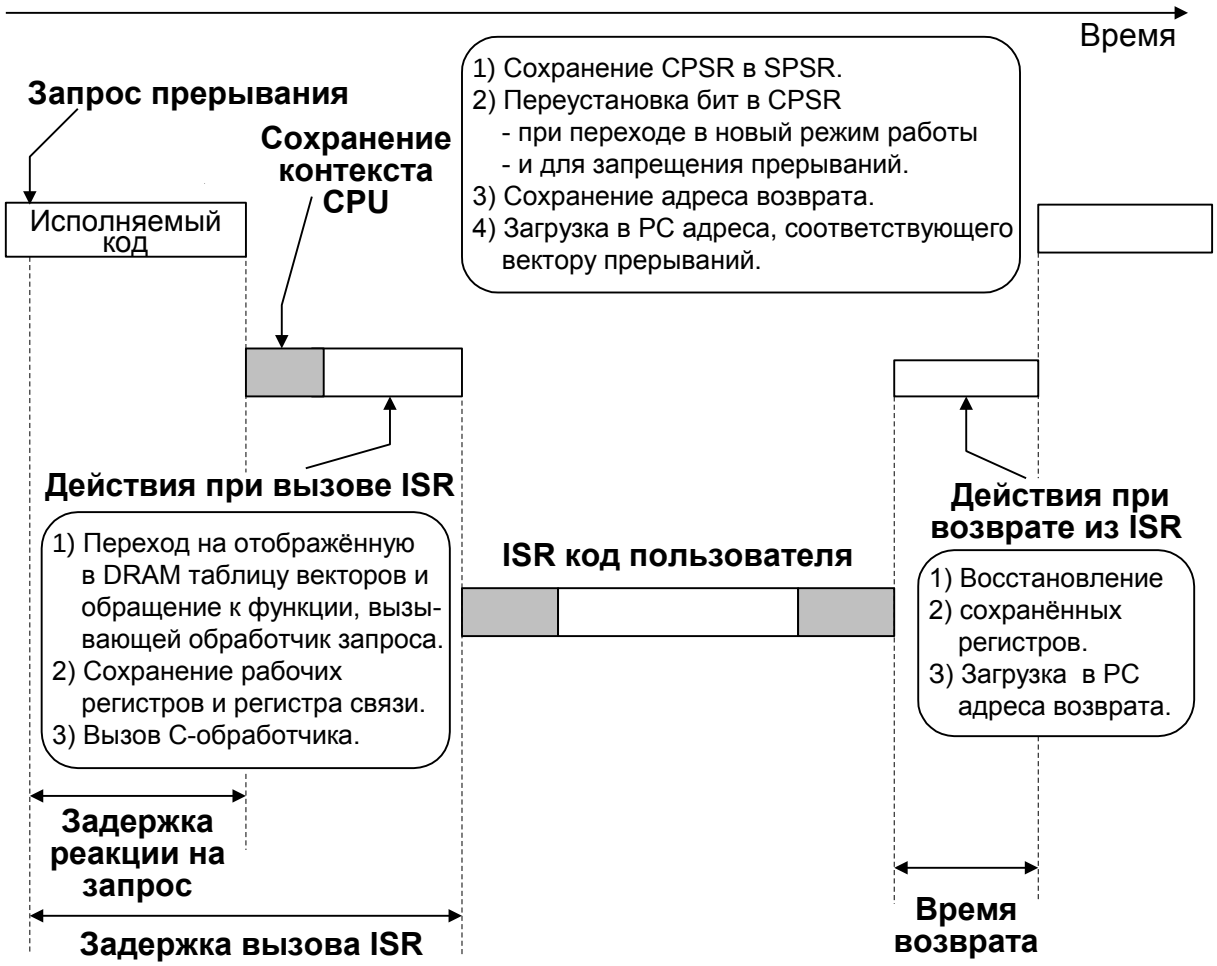


Рис. 4.25. Временная диаграмма последовательности обработки запроса прерывания

4.6. Системы на базе ARM микроконтроллеров

Архитектуру ARM отличает то, что на её основе создаются как простые встроенные микросистемы с низким энергопотреблением, так и сложные многофункциональные аппаратно-программные комплексы, работающие под управлением операционных систем.

В первом случае системы используют ресурсы микропроцессорного ядра, интегрированного в один кристалл вместе с устройствами ввода-вывода (включая простейшие параллельные и последовательные цифровые порты и сложные, поддерживающие различные сетевые приложения). Сюда следует отнести, в частности, коммуникационные МК. Обычно их используют в тех приложениях, в которых конечный пользователь никогда не добавляет программное обеспечение к системе.

Структурная схема одного из микроконтроллеров для коммуникационных приложений (МК ARM7DMI) представлена на рис. 4.26.

Центральный процессорный элемент (ЦПЭ, CPU – *Central Processor Unit*) через встроенный интерфейс (CPU *Interface*) связан с кэш-памятью и с буфером записи (*Write Buffer*). Интегрированные в кристалл компоненты объединены 32-

разрядной системной магистралью (32-*bit System Bus*). Связь ЦПЭ с УВВ и с системной магистралью осуществляется через шинный коммутатор (BUS *Router*).

Буфер записи используется при передаче содержимого регистров ЦПЭ в память: при выполнении инструкции записи в память ЦПЭ делает эту запись не непосредственно в память, а через буфер записи и, не ожидая перезаписи содержимого буфера в основную память, продолжает свою работу. Данные из буфера записи в основную память переносятся в то время, когда системная шина свободна и не занята каким-либо другим активным устройством, или когда механизм шинного арбитража предоставит системную шину в распоряжение процессора. Это освобождает ЦПЭ от необходимости ожидания магистрали при обращении к памяти.

Шинный коммутатор управляет перемещением информации между ЦПЭ и подключёнными к системной магистрали устройствами. Среди устройств, включенных в состав микроконтроллера, имеются:

1. Контроллер прерываний (Interrupt Controller).

2. Двухканальный контроллер прямого доступа к памяти (CDMA – *General Direct Memory Access*).

3. Два универсальных асинхронных приёмо-передатчика UART – *Universal Asynchronous Receiver/Transmitter* и интерфейс I^2C – *Inter-Integrated Circuit* с линией передачи адреса/данных (SDA – *Serial Data/Address Line*) и с линией синхронизации (SCL – *Serial Clock Line*).

4. Порт с 18-ю двунаправленными входами/выходами (контактами), которые могут быть сконфигурированы как самостоятельные цифровые порты ввода-вывода и как контакты, предназначенные для передачи входных и выходных сигналов входящих в состав МК микросхемы устройств. Среди них 4 сигнала запроса прерывания (*Ext* INT *req*.), два запроса ПДП (*Ext* DMA REQ.) и два сигнала подтверждения ПДП (*Ext* DMA ACK) от устройств, расположенных за пределами кристалла микроконтроллера, два выходных сигналов таймеров (*Timer out* (0, 1)).

5. Два контроллера, поддерживающие высокоуровневый протокол канала покадровой передачи данных (HDLCs – *High Level Data Link Controllers*) для связи с удаленными портами (*Remote port* A, B) и имеющие полностью независимую приёмную и передающие части. В состав каждого контроллеры входят контроллеры прямого доступа к памяти DMA (*Direct Memory Access*) для обмена данными с основной памятью МК.

6. Ethernet контроллер – поддерживает множественный доступ к коммуникационной сети через MII-интерфейс (*Media-Independent Interface*), а также через 7-проводной (7-*wire*) последовательный интерфейс. Имеет встроенный контроль сетевого доступа (MAC – *Media Access Control*) с двумя организованными как очередь FIFO (*First In First Out*) буфера памяти – один на приём, другой на передачу.

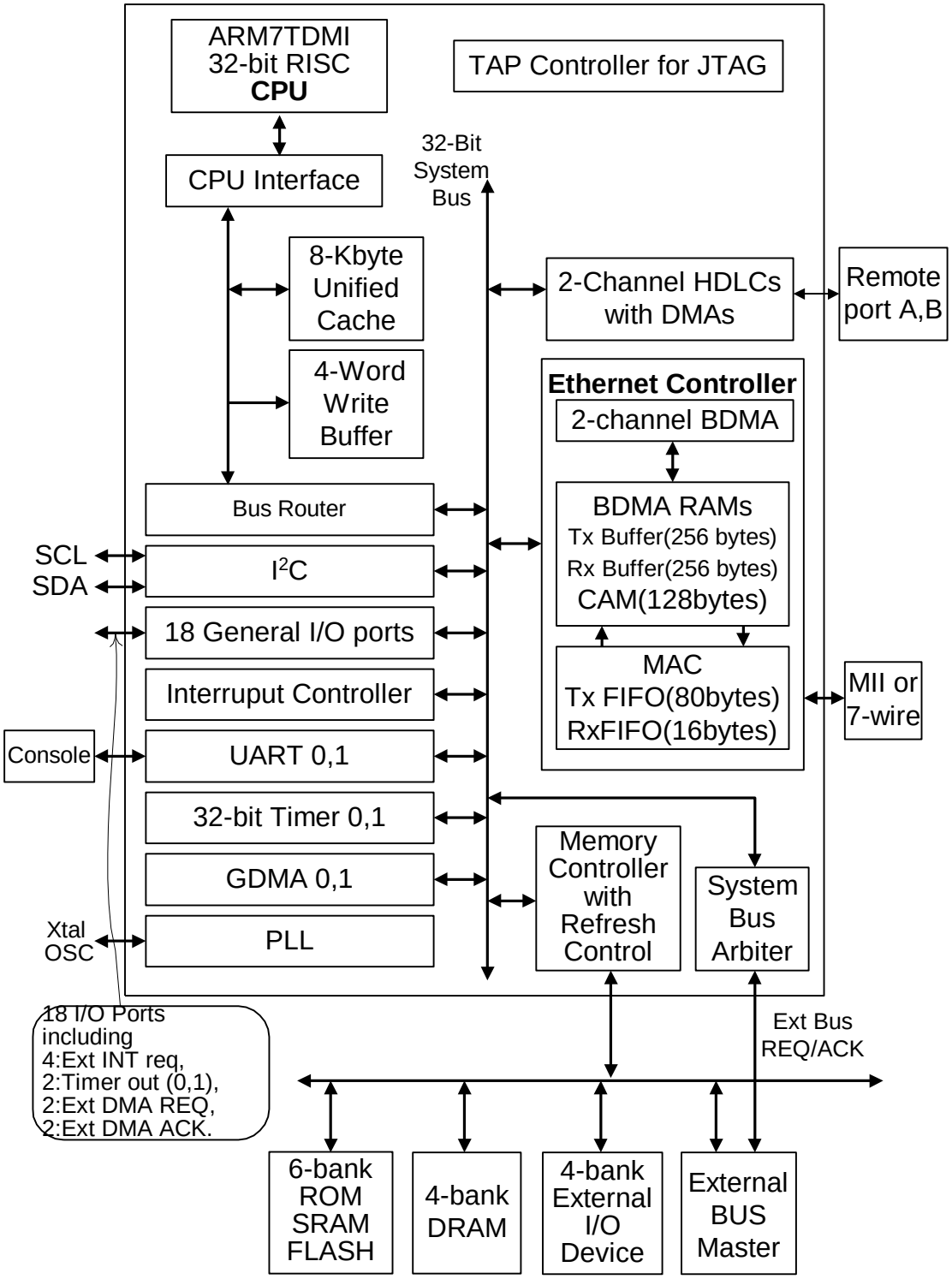


Рис. 4.26. Структура микроконтроллера для коммуникационных приложений

Для обмена данными с основной памятью Ethernet контроллер использует системную шину. Операции чтения и записи выполняются под управлением 2-канального контроллера ПДП (*Buffered* DMA) с отдельными для принимаемых и передаваемых данных буферными оперативными запоминающими устройствами (RAM). При приеме Ethernet-пакетов адресная информация сравнивается с содержимым ассоциативной памяти CAM (*Content Addressable Memory*). Если обнаруживается совпадение, то управляющая приемом микропрограммный ав-

томат продолжает прием пакета информации. В противном случае могут быт предприняты действия с целью посылки источнику информации контрольных пакетов, используемых для дальнейшего продолжения связи.

7. Два 32-разрядных таймера – *Timer* (0, 1).

Работа микроконтроллера тактируется внешним синхронизирующим сигналом 10-40 МГц, частота которого задается кварцевым резонатором Xtal. Тактовая частота внутри МК микросхемы увеличивается в пять раз, для чего используется цифровая система фазовой автоподстройки PLL (*Phase-Locked Loop*)

Микроконтроллер может работать с разными типами постоянной (ROM) и оперативной (динамической и статической) памяти (RAM) и взаимодействовать с находящимися за пределами микросхемы УВВ (*External* I/O *Devices*), а также с другими процессорами или сопроцессорами в составе многопроцессорной системы. К числу сопроцессоров относится, в частности, тот, который расширяет возможности микроконтроллера при работе с памятью, увеличивая размер адресного пространства и реализуя механизм виртуальной адресации и с необходимыми средствами защиты.

Память системы делится на банки и для каждого банка устанавливается своё время доступа, что даёт возможность отображения на память портов внешних устройств ввода/вывода. Управление внешней шиной осуществляет контроллер памяти (*Memory Controller with Refresh Control*), одной из функций которого является регенерация динамической памяти, если таковая в системе имеется.

Возможности ARM-систем могут быть расширены добавлением внешних сопроцессоров, в частности, сопроцессоров с плавающей точкой и управляющего сопроцессора. Последний предоставляет дополнительные средства для конфигурации и управления системой, включая средства защиты памяти, что важно для многозадачных приложений. При наличии других ведущих на внешней системной магистрали (*External BUS Master*), возникает необходимость шинного арбитража – разделения ресурсов шины между отдельными потенциально ведущими. Эту задачу решает арбитр системной шины (*System Bus Arbiter*).

При создании сложных многофункциональные аппаратно-программные комплексов используются ARM микропроцессоры с кэшированным макроядром, способным выполнять операции с многоразрядными данными и работать совместно с встроенными устройствами управления системой (с встроенным в кристалл управляющим сопроцессором). Такое макроядро ориентировано на использование в «открытых» системах, для которых необходимы полное управление виртуальной памятью и развитая защита собственной памяти.

Всё это позволяет создавать масштабируемые системы с возможностью выбора необходимых для конкретных приложений микропроцессорных ресурсов. Этому служит то, что ARM-компоненты могут быть выполнены как на отдельных микросхемах (например, отдельно ЦПЭ с устройствами ввода/вывода,

отдельно сопроцессор управления памятью и сопроцессор с плавающей точкой), так и быть интегрированными в одну микросхему. Кроме этого, ARM-процессоры, работающие обычно с 32-разрядными командами, могут управляться укороченными до 16-ти разрядов Thumb-инструкциями, что повышает компактность создаваемого для них программного кода.

ARM МП являются RISC-процессорами, поэтому

- операции в них выполняются только с данными, находящимися в регистрах,
- отдельно выполняются команды загрузки/сохранения регистров,
- режимы адресации используют только значения, взятые из регистров или из определенных в командах полей,
- все команды имеют фиксированный размер (32 бита при обычной работе и 16 бит при работе в Thumb-режиме),
- все команды являются условно исполняемыми.

В состав ARM микропроцессора помимо перечисленных устройств входят расширения отладки: контроллер TAP (*Test Access Port*) сканирования состояния входов и входов интегрированных в МП ядро функциональных узлов (*boundary scan*) . Ядро процессора с подключенными к нему средствами отладки называют ARM-макроядром. Аппаратные расширения отладки облегчают разработку пользовательского прикладного программного обеспечения, операционных систем и самих аппаратных средств. Аппаратные расширения отладки позволяют останавливать ядро или при выборке заданной команды (в контрольной точке), или при обращении к данным (в точке просмотра), или асинхронно – по запросу в процессе отладки. В точках останова через JTAG последовательный интерфейс могут быть исследованы внутреннее состояние ядра, находящегося в состоянии отладки, и внешние признаки состояния системы. По завершении процесса отладки состояния ядра и системы могут быть восстановлены, а выполнение программы продолжено.

Режим отладки устанавливается запросом по одной из линий внешнего интерфейса отладки и программируется в последовательном режиме с использованием контроллера TAP – средства управления работой цепочек сканирования через последовательный интерфейс JTAG.

Некоторые ARM-микропроцессоры для расширения возможностей отладки имеют встроенный сопроцессор.

4.7. Многоядерные ARM-процессоры

Завершая обсуждение процессоров компании ARM Limited, остановимся ещё на одной их разновидности – многоядерных процессорах. К ним относится процессор ARM11. Структурная схема этого процессора представлена на рис. 4.27.

Процессор имеет четыре микропроцессорных ядра MP11 (CPU), поддерживающих стандартные ARM инструкции, инструкции с укороченными 16-ти разрядными (Thumb) инструкциями, а также байтовые инструкции, ориентированные на непосредственное исполнение Java-программ. Каждый CPU имеет своё устройство управления памятью (MMU), обеспечивающее работу с кэш и внешней памятью и реализующее механизм виртуальной адресации. Поддержка когерентности кэш-памяти данных первого уровня отдельных CPU возлагается на предназначенное для этого устройство – *Snoop Control Unit* (SCU). Имеется контроллер для управления распределённой системой прерываний (*Distributed Interrupt Controller*). Для связи с кэш-памятью второго уровня используются два высокоскоростных интерфейса – *Advanced eXtensible Interface* (AXI). Имеются встроенные сопроцессоры CP14 и CP15. Первый для расширения возможностей средств отладки, а второй – для управления микропроцессорной системой. Предусмотрена возможность работы с сопроцессорами с плавающей точкой. Каждый CPU имеет свой сторожевой таймер (*watchdog*) и свой таймер реального времени (*timer*).

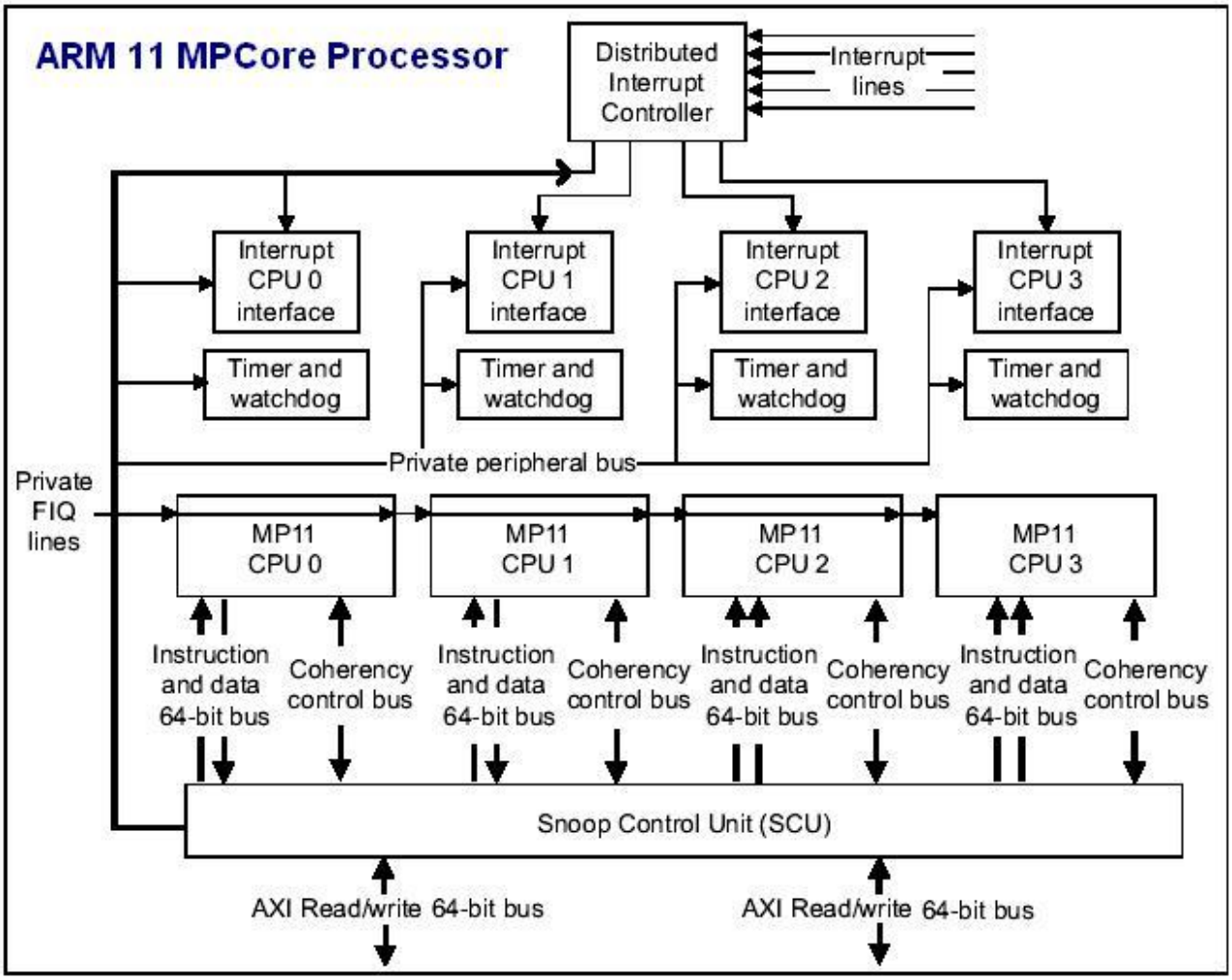


Рис. 4.27. Структурная схема процессора ARM 11

В структуре процессора ARM 11 отражена общая тенденция повышения производительности микропроцессоров за счёт распараллеливания операций путём интегрирования в один кристалл нескольких ядер. Эта тенденция была отмечена и в ранее рассмотренных процессорах фирм Analog Devices и Texas Instruments. Различия в технических решениях свидетельствуют о разнообразии возможных вариантов распараллеливания.

5. МНОГОПРОЦЕССОРНЫЙ ПАРАЛЛЕЛИЗМ. ТРАНСПЬЮТЕРЫ

При создании многопроцессорных систем, работа которых основана на идее параллельных вычислений, приходится иметь дело с двумя фундаментальными понятиями – параллелизм и связь.

Перемещения данных в таких системах могут происходить при использовании

1) глобальной потактовой синхронизации, когда действия процессорных элементов (ПЭ) по передаче локальных данных (данных от одного процессора к другому) контролируется потоком внешних управляющих сигналов, и
2) локального потока управления (управление со стороны взаимодействующих ПЭ) дополнительно к локальному потоку данных, что даёт возможность самосинхронизации при управлении потоком данных.

В последнем случае ход исполнения процессором программы зависит только от доступности данных и требуемых ресурсов. Снижается роль централизованного управления и глобальной памяти. Примером могут послужить потоковые многопроцессорные системы. Однако и для них критическими остаются проблемы конфликтов, затрагивающие память и взаимодействие посредством операций ввода/вывода. Эти проблемы можно разрешить, если потоковые системы строить как модульные с локальными связями. Именно отсюда вытекает значимость принципа локальности передачи данных – передачи не любым, а только соседним (объединённым потоком данных) процессорам.

Если следовать этому принципу, то связь надо понимать как связь через операции ввода/вывода под управлением потока данных с локальной синхронизацией. При этом порт ввода/вывода в любой момент времени имеет состояние

готов к вводу

или

готов к выводу.

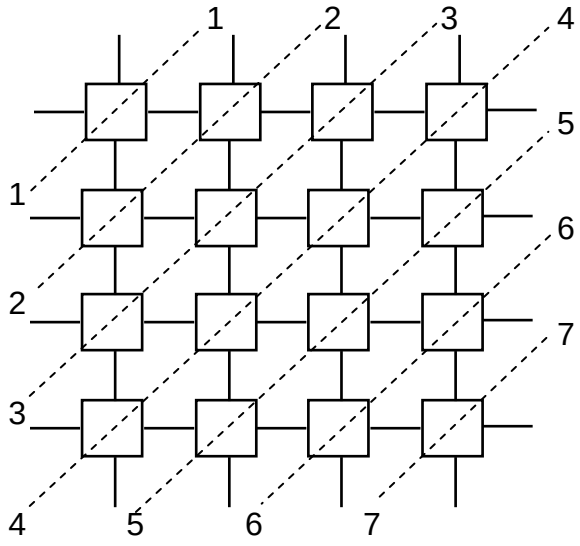


Рис. 5.1. Двумерный матричный массив процессоров

В первом случае инициатором обмена данными является выводящий процессор, и пересылка задерживается до тех пор, пока партнёр по связи в одном из своих обращений не прочтёт текущее содержимое своего порта. Во втором случае обмен инициирует процессор, принимающий данные, и данные принимаются при их поступлении в порт передающего процессора. В результате взаимодействующие процессоры производят обмен данными только тогда, когда один готов передать, а другой готов принять пе-

редаваемые данные.

Идея распараллеливания и конвейеризации вычислений наиболее полно реализуется в *волновых матричных процессорах* (ВМП), имеющих регулярную и поддающуюся аналитическому описанию структуру. Термин «волновой матричный процессор» связан с представлением о том, как в текущий момент времени распределяются вычислительные операции на параллельно работающие процессоры, и в какой последовательности активность одних параллельно работающих[4] процессоров передаётся другим. В волновом процессоре переход активности от одних процессоров к другим можно представить как распространение волны вычислений по массиву процессоров. На рис. 5.1 представлен двумерный массив процессоров и показано, как волновой фронт 1-1 на этом массиве с течением времени перемещается в положение 2-2, 3-3, …, 7-7. При этом на смену ушедшей волне вычислений всякий раз приходит новая волна, идущая вслед за ушедшей.

Имеют место два подхода к программированию матричных волновых процессоров – *глобальный* и *локальный*. Глобальная волновая программа описывает алгоритм с точки зрения волнового фронта, проходящего через все процессоры. Локальная волновая программа описывает операции отдельного процессора с учётом возможности обработки серии проходящих через него волновых фронтов.

Реализация программ а такой системе требует преобразования с помощью препроцессора глобального волнового описания на высоком уровне в набор программ (или микроинструкций) на низком уровне для отдельных процессоров (или процессорных элементов).

Рассмотрение волновых матричных процессоров в данном пособии не преследует цели их детального изучения. Основной задачей является ознакомление со средствами и способами межпроцессорного (в том числе межзадачного) взаимодействия и, в частности, с правилами и приёмами, которым необходимо следовать, когда речь заходит о построении многопроцессорной системы.

Заметим также, что процессорные элементы (будь то процессоры «целиком» или их крупные составные части) могут быть интегрированы в одну микросхему матричного процессора. Тенденция развития такова, что со временем ПЭ должны превратиться в объекты, расположенные внутри одной программируемой микросхемы, подобной микросхемам FPGA (*Field Programmable Gate Array*). Уже в настоящее время разработчикам цифровых устройств предлагаются микросхемы FPOA (*Field Programmable Object Array*), программирование которых заключается в прокладывании соединительных трасс между объектами без вмешательства в то, что находится внутри интегрированных в микросхему объектов. В дополнение к этому предлагается комплект программных инструментов и язык программирования – OHDL (*Object Hardware Description Lan-*

[4] О пространственном распределении активности параллельно работающих процессоров можно говорить как о волне вычислений. Отсюда происходит понятие «волновой процессор».

guage), ориентированный на описание ресурсов микросхем. Производитель – фирма MathStar.

Как иллюстрацию основных идей программирования параллельных и конвейерных операций можно назвать язык *Occam*, который был разработан фирмой *Inmos Ltd*. для программирования ей же разработанных волновых матричных процессоров, сконструированных из транспьютерных кристаллов.

5.1. Транспьютер как процессорный элемент для волнового процессора

Процессорные элементы (ПЭ)[5] для *волновых матричных процессоров* (ВМП) должны удовлетворять определённым требованиям, позволяющим им работать в многопроцессорной среде. И прежде всего это относится в средствам межпроцессорного взаимодействия. Идея распараллеливания операций привела к созданию процессора, предназначенного для работы в вычислительной сети, – *транспьютера*. Такое название можно интерпретировать как сочетание слов ***trans****lating com****puter***.

Некоторые специалисты понимают термин «транспьютер» как название конкретного продукта фирмы *Inmos*, другие трактуют его как обобщённое наименование микропроцессоров со встроенными каналами межпроцессорного обмена. Используется также термин «транспьютероподобный микропроцессор», чтобы, с одной стороны, подчеркнуть, что речь идёт не о продукции фирмы *Inmos*, а с другой, указать, что микропроцессор имеет встроенные линки для образования параллельных систем. Вполне возможно, что стремительное развитие микроэлектроники не позволит термину «транспьютер» устояться, и он будет поглощён более общим термином «микропроцессор», т. к. отличительный признак транспьютера – встроенные межпроцессорные каналы появятся в том или ином виде у всех микропроцессоров. Достаточно упомянуть в связи с этим процессоры фирмы *Analog Devices* серии SHARC с встроенными в них линк-портами (раздел 6).

Хотя транспьютер и является одиночным процессором, однако он не рассчитан на работу в однопроцессорной конфигурации, а ориентирован на параллельную работу нескольких транспьютеров. Первые транспьютеры представляли собой 16- и 32-разрядные процессоры и не поддерживали операций с плавающей точкой. К числу 32-разрядных относится микропроцессор T414. Затем в состав транспьютера был включён сопроцессор с плавающей точкой (МП T800). Последние версии транспьютеров (T9000 – Chameleon) стали 64-

[5] Процессорные элементы (ПЭ) – это те же процессоры, но ориентированные преимущественно на работу в вычислительной сети. Ими могут быть, в частности, простые вычислительные устройства предназначенные для выполнения заданного набора операций и интегрированные в одну микросхему матричного процессора.

разрядными. На рис. 5.2 представлена структурная схема 32-разрядного транспьютера T800.

Все транспьютеры имеют RISC-архитектуру с фиксированным 8-разрядным размером команд, Наличие внутренней памяти позволило обойтись небольшим количеством регистров при проектировании центрального процессора (ЦП).

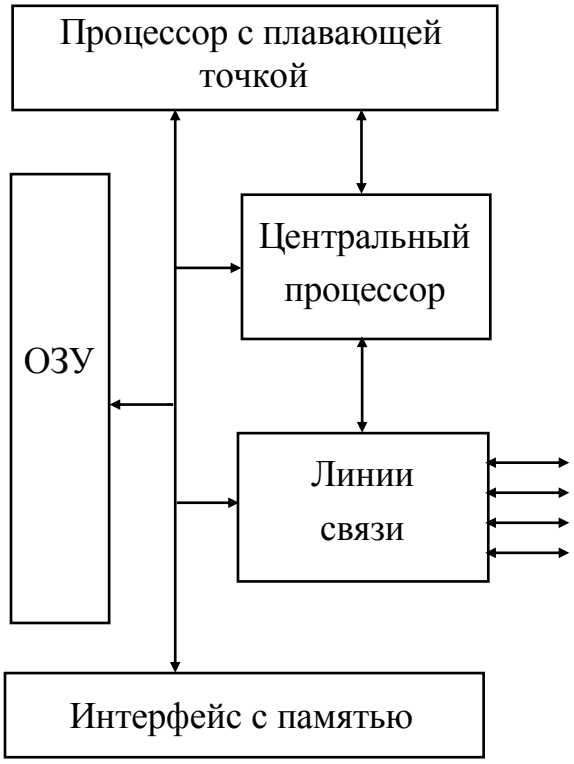


Рис. 5.2. Структурная схема 32 - разрядного транспьютера T800.

Этими регистрами (рис. 5.3) являются:

1) указатель на рабочую область – на область памяти, где хранятся локальные переменные;
2) счётчик команд – указатель на следующую инструкцию;
3) регистр операнда – используется для формирования операндов инструкций;
4) регистры A, B и C – используются при работе с целыми числами и адресами и сформированы как аппаратный вычислительный стек.

Механизм вычислительного стека работает так, что не требуется явного обращения к нему со стороны программы. Поэтому команды транспьютера не содержат операндов и при исполнении команды операнды определяются на аппаратном уровне. О таких операндах можно говорить как об *операндах по умолчанию*.

При записи в вычислительный стек содержимое регистра B записывается в регистр C, содержимое регистра A в регистр B, и лишь после этого в регистр A заносится новое значение. При чтении из стека извлекается значение регистра A, после чего содержимое регистра B записывается в регистр A, а содержимое регистра C – в регистр B. Стек используется также для вычисления выражений. Например, инструкция *add* (сложение) складывает два верхних элемента стека и помещает результат на вершину стека. Статистика, набранная при анализе большого числа программ, показала, что три регистра обеспечивают баланс между компактностью кода и сложностью технической реализации.

Соответственно *процессор с плавающей точкой* (ППТ) содержит трёхрегистровый стек для хранения операндов и результатов при работе с числами с плавающей точкой. Адреса переменных с плавающей точкой формируются в стеке ЦП. ЦП управляет также обменом данными между памятью и вычислительным стеком ППТ. Так как стек ЦП используется только для хранения адре-

сов переменных с плавающей точкой, разрядность ЦП не зависит от разрядности ППТ. Следовательно, один и тот же ППТ может быть использован в транспьютерах с разной разрядностью ЦП.

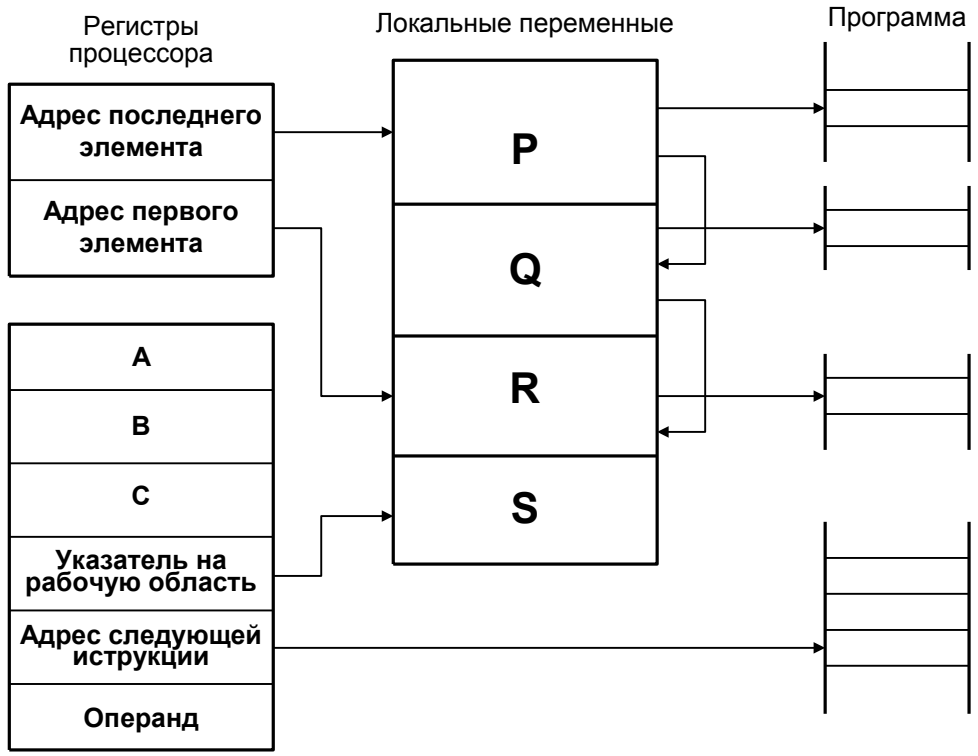


Рис. 5.3. Регистровая модель транспьютера.

Каждая инструкция транспьютера занимает один байт, который разделён на два 4-битовых поля (рис. 5.4). Четыре старшие бита этого байта являются кодом функции, а младшие четыре бита содержат операнд.

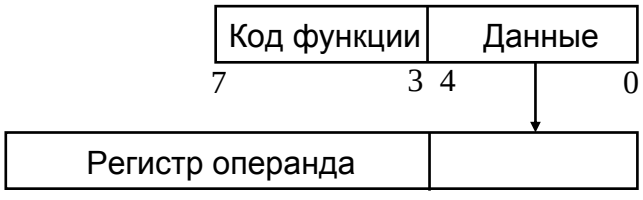


Рис. 5.4. Формат инструкций

Кроме операндов по умолчанию в инструкциях можно указывать непосредственные операнды при условии, что для их представления достаточно четырёх бит поля операнда.

Увеличить размер операндов можно, используя специальные префиксные инструкции (*prefix* и *negative prefix*). При исполнении инструкции *prefix* четыре бита данных замещают четыре младших бита регистра операнда, содержимое которого затем смещается на четыре позиции влево. Посредством последовательности префиксных инструкций операнды могут быть расширены до любой длины вплоть до разрядности регистра операнда.

При выполнении инструкции *negative prefix* происходит то же самое, только до сдвига содержимое регистра операнда переводится в дополнительный код.

Таким образом, за счёт префиксных инструкций процессор может единым образом работать с операндами любой длины. При этом способ формирование операндов не зависит от разрядности процессора, поскольку максимальная длина определяется разрядностью регистра операнда, а разрядность регистра операнда соответствует разрядности процессора.

Поддержка параллелизма

В транспьютере есть микропрограммный планировщик, позволяющий параллельно выполнять любое число процессов в режиме разделения времени процессора, что позволяет обходиться без ядра операционной системы. Этот процессор не имеет средств динамического выделения памяти, и память для параллельных процессов выделяет компилятор с языка *Occam.*

Параллельно исполняемые процессы могут находиться в следующих состояниях:

активное – выполняется,
– находится в очереди ожидания на выполнение;
пассивное – готов к вводу,
– готов к выводу,
– ожидает, пока на наступит указанное время.

Пассивные процессы не занимают процессорное время. Активные процессы, ожидающие выполнения, находятся в списке. Это связанный список рабочих областей процессов, реализованный с использованием двух регистров, один из которых указывает на первый процесс из списка, другой – на последний процесс. На рис. 5.3 процесс S выполняется, а процессы P, Q и R активны и ожидают выполнения.

Процесс выполняется до тех пор, пока не перейдёт в режим ожидания ввода, вывода или таймера. Как только процесс не может выполняться, его счётчик команд сохраняется в его рабочей области и выбирается следующий процесс из списка для выполнения. Время переключения с процесса на процесс очень маленькое, так как не нужно сохранять информацию о состоянии. Нет необходимости сохранять при передиспетчеризации вычислительный стек.

В процессоре реализованы специальные операции, поддерживающие понятие «процесс». Сюда входят *start process* и *end process*.

При выполнении параллельной конструкции используется операция *start process* для создания необходимых параллельных процессов. Инструкция *start process* создаёт новый процесс и добавляет новую рабочую область в конец списка диспетчеризации, что позволяет новому процессу выполняться вместе с уже существующими.

Параллельная конструкция при корректном завершении выполняет инструкцию *end process*. Эта инструкция использует позицию рабочей области с числом компонентов, равным числу процессов в параллельной конструкции.

Каждому процессу соответствует своя позиция, фиксируемая по счётчику, значение которого устанавливается при включении процесса в список диспетчеризации. При завершении процесса (при выполнении инструкции *end process*) значение счётчика уменьшается и проверяется. Для всех компонентов, кроме последнего, счётчик имеет ненулевое значение. При этом условии поступление инструкции *end process* выводит процесс из списка очерёдности на выполнение. Для последнего элемента списка счётчик нулевой, и процесс продолжает выполняться.

Связь

Связь между процессами осуществляется с помощью каналов и взаимодействие является двухточечным, синхронным и небуферизованным. Вследствие этого для канала нет необходимости в очереди процессов и сообщений, а также не нужны буферы сообщений.

Канал связи между процессами, выполняющимися на одном транспьютере, реализуется с помощью одного слова в памяти; канал между процессами, выполняющимися на разных транспьютерах, – с помощью двухточечных линий связи. Обмен сообщениями поддерживается рядом операций, наиболее важными из которых являются *input message* и *output message*.

При исполнении инструкций *input message* и *output message* анализируется адрес канала и определяется является ли канал внутренним или внешним. Один и тот же тип инструкций может быть использован как для программно, так и для аппаратно реализуемых каналов.

Связь осуществляется тогда, когда и вводящий, и выводящий процессы готовы к обмену информацией. Следовательно, процесс, первый инициировавший обмен, должен ждать, пока второй процесс не будет готов к обмену.

Когда процесс выполняет операцию ввода или вывода, в вычислительный стек загружаются указатель на сообщение (регистр С), адрес канала (регистр В) и число байтов для передачи (регистр А), после чего выполняется инструкция *input message* или *output message*.

Коммуникация по внутренним каналам

В каждый момент времени внутренний канал (одно слово в памяти) содержит или идентификатор процесса, или специальное значение *empty* (свободен). Если канал не используется, то он содержит значение *empty*.

При посылке сообщения через канал в него записывается идентификатор процесса, который первым инициировал связь, и этот процесс исключается из списка диспетчеризации. Процессор начинает выполнение следующего в порядке очерёдности процесса. Как только второй процесс, использующий этот же канал, становится готовым к обмену, ожидающий процесс включается в диспетчерский список, и канал переустанавливается в состояние «свободен». При этом не имеет значения, какой процесс (вводящий или выводящий) первым инициировал обмен по каналу.

Последовательность событий при выводе в «свободный» канал С показана на рис. 5.5 и 5.6. Когда процесс Р выполняет инструкцию вывода, в регистры

вычислительного стека заносятся указатель на сообщение, идентификатор (адрес) канала С и число байтов в сообщении. В рабочую область, принадлежащую процессу Р, заносится содержимое регистров А и С – указатель на передаваемое сообщение и число передаваемых байт (рис. 5.5). При этом в канал С помещается указатель на рабочую область процесса Р (рис. 5.6).

Процесс Р блокируется (выводится из списка диспетчеризации), и процессор приступает к выполнению следующего по списку процесса. Перед этим значение счётчика команд сохраняется в рабочей области процесса Р. Это необходимо для того, чтобы выполнение процесса Р могло быть продолжено после того, как вывод данных будет завершён.

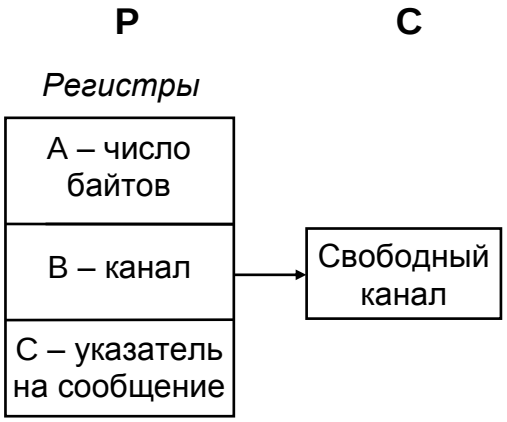


Рис. 5.5.

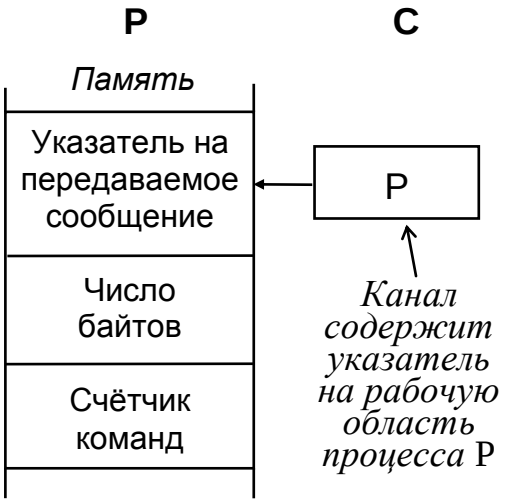


Рис. 5.6.

В таком состоянии канал С и процесс Р находятся до тех пор, пока второй процесс Q не выполнит инструкцию ввода из канала С, обратившись к каналу, используя адрес из регистра В (рис.5.7).

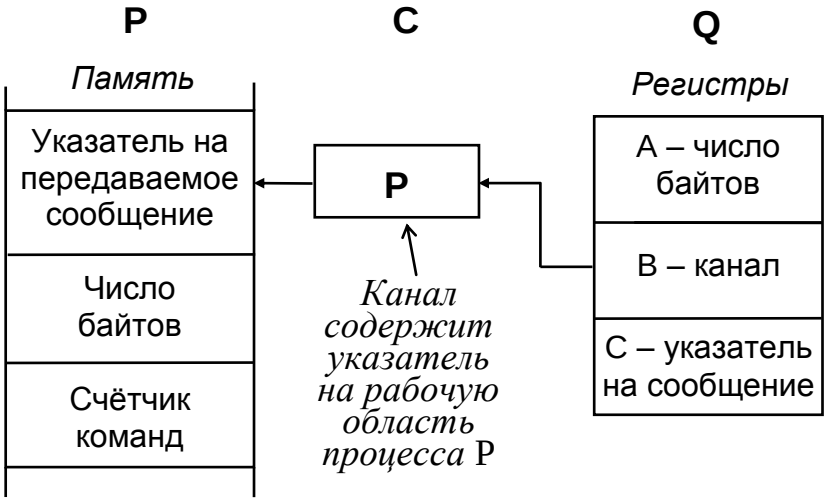


Рис. 5.7.

Процесс Q получает информацию о месте расположения и объёме передаваемого сообщения из рабочей области процесса Р, размещая её в регистрах

A и C. Затем сообщение копируется в память процесса Q, ожидающий процесс P включается в список диспетчеризации, а канал C устанавливается состояние «свободен».

Коммуникация по внешним каналам

При посылке сообщения по внешнему каналу процессор инициирует работу автономного интерфейса связи по передаче этого сообщения, и процесс исключается из списка диспетчеризации. Когда сообщение будет передано, интерфейс связи сигнализирует диспетчеру о том, что ожидающий завершения сеанса связи процесс должен быть включён в список диспетчеризации. Такой механизм позволяет процессору продолжать выполнение других процессов, в то время, когда идёт передача сообщения по внешнему каналу.

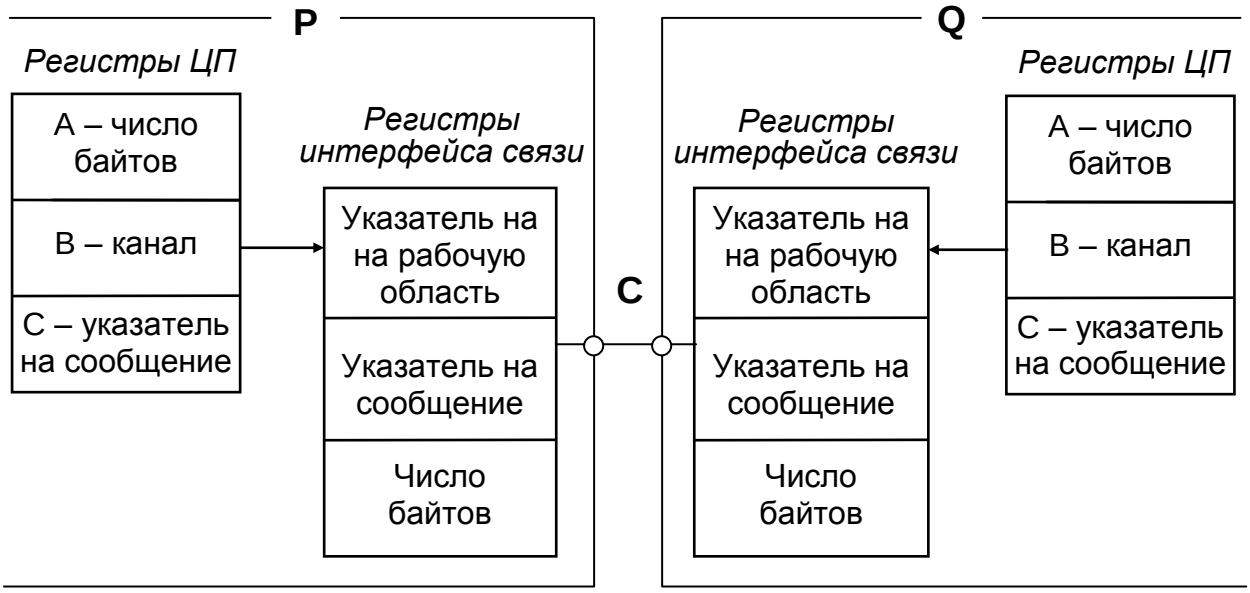


Рис. 5.8. Связь между транспьютерами.

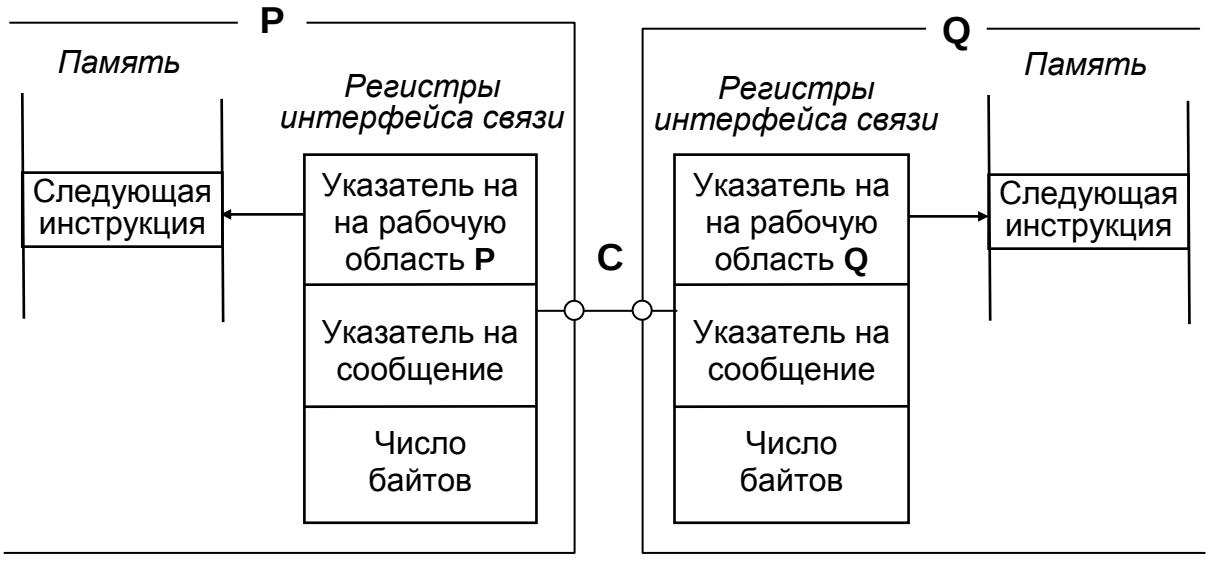


Рис. 5.9. Состояние канала связи, когда процессы P и Q исключены из списка диспетчеризации.

Каждый интерфейс связи использует три регистра (рис. 5.8):

- указатель на рабочую область процесса;
- указатель на сообщение;
- число байтов в сообщении.

На рис.5.9 показано, как взаимодействуют процессы (процессы P и Q), выполняющиеся на разных транспьютерах, используя внешний канал C. Когда процесс P выполняет инструкцию *вывода* в канал C, регистры интерфейса связи транспьютера, на котором выполняется этот процесс, инициализируются и процесс P исключается из списка диспетчеризации.

При выполнении процессом Q инструкции *ввода* инициализируются регистры интерфейса связи транспьютера, выполняющего процесс Q, и процесс Q также исключается из списка диспетчеризации.

В рабочих областях блокированных процессов P и Q сохраняется необходимая для их продолжения информация (в частности, значения счётчиков команд). Место расположения этих областей определяется по регистрам-указателям на рабочие области, входящим в состав связных интерфейсов канала C (рис.5.9).

Сообщение передается по линии связи в режиме автономной работы интерфейса связи, после чего процессы P и Q вновь восстанавливаются в своих списках диспетчеризации.

Линии связи

Связь между двумя транспьютерами устанавливается путём соединения интерфейса связи одного транспьютера с интерфейсом связи другого транспьютера двумя однонаправленными сигнальными линиями, по которым данные передаются последовательно. Эти две линии представляют аппаратную реализацию двух программных каналов (по одному в каждом направлении), описываемых на языке *Occam*. Для реализации каналов требуется установить протокол и способ временного мультиплексирования передаваемых данных.

Рис. 5.10. Перекрытие во времени при передаче подтверждающих сообщений в линиях связи

Сообщения передаются в асинхронном режиме как последовательность байтов, причем на каждый переданный байт должно прийти подтверждение. Байты посылаются в сопровождении двух стартовых бит и одного стопового бита (рис.5.10). Сигнал подтверждения состоит из старт-бита и стоп-бита. Подтверждение означает, что есть готовность принять следующий байт данных. При этом передатчик может получить сигнал подтверждения до того, как весь пакет данных будет передан целиком. В этом случае передатчик может посылать следующий пакет данных сразу вслед за текущим. Такое перекрытие иллюстрирует рис. 5.10.

Таймер

Таймер позволяет отслеживать текущее время. Значение таймера можно прочитать с помощью инструкции

read time.

Инструкция

timer input

приостанавливает выполнение процесса до тех пор, пока не наступит указанное время. Если указанное время ещё на наступило, процесс исключается из списка диспетчеризации. При наступлении указанного времени процесс снова включается в список диспетчеризации.

5.2. Основные конструкции языка *Occam*

Язык *Occam* позволяет представить прикладную программу как набор параллельных процессов, взаимодействующих через каналы. Канал используется для передачи данных между процессами, один из которых вводит из канала, другой – выводит в канал. Процессы не могут взаимодействовать посредством общих данных. Единственным способом взаимодействия процессов является посылка сообщений по каналам. Такое соглашение в языке *Occam* действует и для процессов, выполняющихся на одном транспьютере, и для процессов, выполняющихся на разных транспьютерах. Таким образом, одна и та же модель параллелизма используется как для одного, так и для нескольких процессоров.

В язык *Occam* включены два фундаментальных понятия: процесс (RPOC) и канал (CHAN).

Процесс – это оператор или группа операторов или даже группа процессов. Процесс означает независимое и автономное вычисление, обладая своими собственными программой и данными. Он способен связываться с другими процессами по явно определённым каналам. В отличие от параллелизма на микроуровне в машинах с управлением потоком данных параллелизм в языке *Occam* находится на процедурном уровне.

В основе лежат три примитивные процесса:

1) *присваивание*, например,

```
a := b + c;
```

2) *ввод* – получение значения из канала и присваивание переменной, например,

```
chan1 ? a
```

– ввод из канала *chan1* и присваивание полученного значения переменной *a*;

3) вывод – передача значения переменной в канал, например,

```
chan2 ! b + c.
```

Типы и описания переменных и каналов

Язык *Occam* наряду с описанием переменных включает описание каналов:

- BOOL – двоичные переменные;
- BYTE – байтовый тип;
- INT16, INT32, INT64 – 16-, 32- и 64-разрядные целочисленные переменные;
- REAL32, REAL64 – переменные с плавающей точкой.
- Массивы переменных и каналов описываются с указанием размерности:
- [20] INT vector – одномерный массив целых;
- [100] CHAN OF INT switch – одномерный массив каналов switch для передачи целочисленных значений;
- [8][8] BYTE – двумерный массив байтовых переменных.

Основными конструкциями языка *Occam* являются:

- SEQ – последовательная конструкция. Входящие в конструкцию процессы выполняются один за другим;
- PAR – параллельная конструкция. Все составляющие конструкцию процессы выполняются параллельно;
- ALT – конструкция альтернативного выбора. Выполняется первый из готовых к связи процессов;
- IF – условные конструкции. Может быть выполнен более чем один из составляющих конструкцию процессов.

Условная конструкция предназначена для управления вычислениями. Кроме неё есть ещё другие управляющие конструкции:

конструкция выбора – CASE,
конструкция повторения с проверкой условия – WHILE,
конструкция повторения – FOR.

Протоколы каналов

Представленный выше способ описания каналов является наиболее простым. Полное описание канала предполагает указание протокола передачи и приёма данных.

1) Канал может иметь протокол типа «*счётный массив*». При обмене данными сначала передаётся число элементов массива, а затем сами элементы. Элементами могут быть переменные любого типа. Программа передачи байт может выглядеть следующим образом:

```
PROTOCOL counted.byte.array IS INT::[]BYTE:
CHAN OF counted.byte.array c:
[20] BYTE buffer:
INT len:
PAR
    c ! 7:: "goodbye cruel world"
    c ? len:: buffer
```

В представленном примере канал c, имеющий протокол counted.byte.array, используется для передачи семи байт из строки "goodbye cruel world" («прощай жестокий мир») одним процессом и для приёма этих байт другим процессом с сохранением принятого числа байт в переменной len, а принятых символов – в массиве buffer. Отметим, что каждая директива описания заканчивается двоеточием. Группа строк программы, начинающаяся с одной позиции, находится на одном программном уровне и рассматривается как одна конструкция, в данном случае конструкция PAR из двух параллельно исполняемых процессов. Применение символ «точка» допустимо в именах переменных.

2) *Последовательный протокол* применим тогда, когда по каналу передаются значения переменных разных типов. В приведённом ниже примере протокол byte.int.pair соответствует передаче последовательности из одного байта и одной целочисленной переменной:

```
PROTOCOL byte.int.pair IS BYTE;INT:
CHAN OF byte.int.pair c:
BYTE b:
INT i:
PAR
    c ! 'a'; 42
    c ? b; i
```

Один процесс передаёт в канал c значение символа 'a' и число 42. Другой процесс принимает эти значения и сохраняет в своих переменных b и i. Перечисленные в списке переменные отделяются символом `';'`. В общем случае элементы списка могут быть как скалярами, так и массивами, а сам список может содержать разное число элементов.

3) *Протокол с вариациями*, например,

```
PROTOCOL data.variant
CASE
char; BYTE
number; INT
string; [256] BYTE
nothing:
```

описывает возможность использования разных форматов. При этом каждое сообщение начинается с тега (в приведённом примере тегами являются char, number, string, nothing). После тега следует перечисление типов переменных, которые могут входить в сообщение.

Пример использования протокола с вариациями data.variant:

```
CHAN OF data.variant data.chan:
BYTE ch:
INT i:
[256] BYTE str:
SEQ
      …..некоторые действия…..
```

```
data.chan ? CASE
char; ch
…..некоторые действия для char…..
number; i
…..некоторые действия для number…..
string; str
…..некоторые действия для str…..
nothing;
…..некоторые действия для nothing…..
```

В примере приведена последовательная конструкция SEQ с процессом, принимающим данные из канала. Исполняемые процессом действия зависят от того, какой тег предшествует поступающим из канала данным.

Процедуры и функции

Язык *Occam* имеет средства описания процедур и функций.

При описании *процедуры* задаются её имя и набор формальных параметров, которые при вызове заменяются на действительные. Действительными параметрами могут быть значения выражений, переменные, *каналы*, массивы переменных и *массивы каналов*. Существенным отличием от языков последовательного программирования является то, что в качестве параметров процедурам могут передаваться каналы.

Функции подобны процедурам, которые возвращают одно или несколько значений. Однако функции не могут изменять значения внешних по отношению к ним переменных, и также осуществлять связь по каналам.

Пример программирования на языке *Occam*

В качестве примера программирования на языке *Occam* возьмём перемножение квадратных матриц $\boldsymbol{A}$ и $\boldsymbol{B}$ на массиве процессоров, показанном на рис. 5.1. Размерность массива процессоров $n \times n$ совпадает с размерностью матриц:

$$\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B} \text{ или } c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}.$$

Относящийся к перемножению матриц фрагмент программы представлен в листинге 5.1.

Листинг 5.1. Перемножение квадратных матриц

```
[n*(n+1)]CHAN OF INT vertical:
[n*(n+1)]CHAN OF INT horizontal:
PAR i = 0 FOR n
   PAR j = 0 FOR n
         mult(vertical[n*i + j], vertical[n*i+ j + 1],
               horizontal[n*i + j], horizontal[n*(i+1) + j])
PROC mult(CHAN OF INT up, down, left, right)
VAR acc, a, b:
SEQ
   acc := 0
```

```
SEQ i = 0 FOR n
   SEQ
      PAR
         up ? a
         left ? b
      acc := acc + a*b
      PAR
         down ! a
         right ! b
```

Примечание: *константа n должна быть определена предварительно*.

Одна матрица поступает слева (по столбцам), а другая – сверху (по строкам). По мере продвижения вниз и вправо данные перемножаются и накапливаются. После завершении этого продвижения каждый процессорный элемент будет содержать элементы матрицы ***C***.

Заметим, что конструкция i = 0 FOR n не является оператором циклического повторения, а применяется для описания процессов, которые исполняются параллельно, если i = 0 FOR n включена в конструкцию PAR, и последовательно, если используются в конструкции SEQ. Например, конструкция SEQ I = 0 FOR n говорит о том, что *n* процессов исполняются последовательно один за другим.

Программирование в реальном масштабе времени

Разработка транспьютеров производилась с учётом возможности работы их в системах реального времени. Эти системы взаимодействуют с внешними устройствами и должны реагировать на события за вполне определённое время.

Асинхронная природа внешнего события такова, что событие можно отобразить на связь по каналам. При этом процедуре обработке события предоставляется выделенный для этого канал, поскольку отвечающие за событие внешние устройства могут соединяться с транспьютером по каналам связи. Кроме этого транспьютер сам имеет вывод «*Event*», который может быть отображён на канал в прикладной программе. Когда вывод «*Event*» возбуждается внешней системой, процесс, ожидающий ввода по этому каналу, включается в список диспетчеризации.

На структуру процессов, обрабатывающих события накладываются приоритеты. Если два процесса одновременно окажутся готовыми к выполнению, то к исполнению будет принят процесс с большим приоритетом, что позволит избежать конфликта.

Подытоживая сказанное здесь о транспьютерах добавим, что в середине 90-х годов XX века в эпоху расцвета микропроцессоров семейства *транспьютеров* фирма *Inmos* поставляла широкий набор *трэмов* (трэм – *TRansputer Extension Module*) – устройства ввода/вывода с линком в качестве интерфейса. Появились трэмы, специализированные на вычислениях, на операциях ввода/вывода, на работе с дисковыми накопителями, с графическими изображениями; трэмы, позволявшие подключить через линк адаптеры Ethernet или

SCSI. На этом фоне вполне естественным было появление разнообразных параллельных компьютеров и супер-ЭВМ с оригинальными архитектурными решениями.

6. Микропроцессоры с гарвардской архитектурой и микросистемы на их основе

Очевидным способом повышения производительности процессоров является распараллеливание операций, которое может быть выполнено разными способами, в частности, за счёт организации параллельной работы устройств, отвечающих за вычислительные операции при обработке данных и при вычислении исполнительных адресов данных и команд. Такое распараллеливание свойственно гарвардской архитектуре с отдельными памятью программ и памятью данных. Гарвардская архитектура широко используется в цифровых процессорах сигналов (ЦПС). Черты гарвардской архитектуры можно найти в микроконтроллерах и в универсальных процессорах. Одним из её проявлений является использование раздельных кэш-памяти программ и кэш-памяти данных.

6.1. Цифровые процессоры сигналов семейства ADSP 21xx

Особенности функционирования процессоров с гарвардской архитектурой рассмотрим на примере цифровых процессорах сигналов фирмы Analog Devices. Обращение к процессорам этой фирмы обусловлено тем, что в них наиболее ясно выражены не только аппаратные решения, но и средства программирования процессоров с гарвардской архитектурой.

6.1.1. Структура ядра

Структура ядра цифровых процессоров сигналов, принадлежащих одному из первых семейств ЦПС фирмы Analog Devices – семейства ADSP-21xx, – представленная на рис. 6.1. Процессоры имеют интегрированную в МП кристалл память программ PM (*Program Memory*) и память данных DM (*Data Memory*). При необходимости к этой памяти может быть добавлена внешняя, расположенная за пределами микропроцессора память.

Доступ к памяти системы осуществляется по четырем шинам:

- к памяти программ по шине адреса памяти программ PMA (*Program Memory Address*) и шине данных памяти программ PMD (*Program Memory Data*);
- к памяти данных по шине адреса памяти данных DMA (*Data Memory Address*) и шине данных памяти данных DMD (*Data Memory Data*).

Ядро ЦПС семейства ADSP-21xx включает:

- устройство управления последовательностью выполнения программы (*Program Sequencer*),
- два генератора адресов памяти данных (*Data Address Generator* – DAG1,2),

– арифметическо-логическое устройство (*Arithmetic-Logic Unit* – ALU),
– умножитель-аккумулятор (*Multiplier-Accumulator* – MAC) и
– сдвигатель (*Shifter*).

Структура ядра оптимизирована для выполнения наиболее часто встречающихся при цифровой обработке сигналов операций (вычисление свертки, функции корреляции, быстрого преобразования Фурье и др). Результаты операций ALU, MAC и сдвигателя всегда размещаются в выходных регистрах, которые при необходимости могут быть сохранены в памяти или в качестве операндов переданы от одного операционного блока к другому по дополнительной шине – шине результата (R BUS – *Result Bus*).

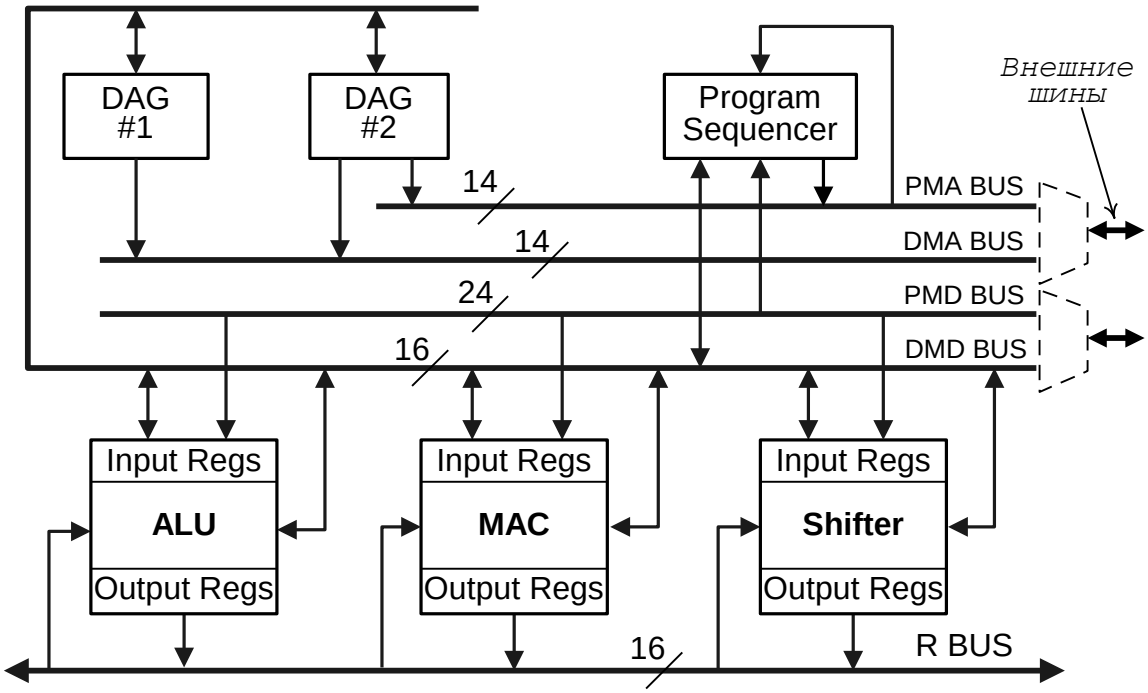


Рис. 6.1. Структура ядра цифрового процессора сигналов семейства ADSP-21xx

Все операции выполняются только с содержимым регистров – входных (Input Regs) или выходных (Output Regs). По этому признаку МП ADSP 21xx можно классифицировать как RISC-процессоры. Все команды процессора умещаются в 24-разрядные слова, чем определяется размер ячеек памяти программ и разрядность шины PMD. Единицей памяти данных ЦПС ADSP 21xx являются 16-разрядные слова, поэтому 16-разрядной является шина данных DMD.

Вместе с ядром и внутренней памятью программ и данных на кристалле микропроцессора размещены таймеры, устройства ввода-вывода и другие компоненты, состав которых зависит от конкретной модели МП ADSP 21xx.

На рис. 6.2 представлена структура микропроцессорной системы на базе ЦПС семейства ADSP-21xx с интегрированными в кристалл таймером и двумя последовательными портами (SPORT1,2). Другие МП этого семейства могут иметь контроллер прямого доступа к памяти или хост-интерфейс для связи с управляющей (главной) ЭВМ, например, с персональным компьютером.

Внешняя системная магистраль содержит одну шину адреса и одну шину данных, которые формируются путём мультиплексирования внутренних шин PMA и DMA, PMD и DMD. Обращение к внешней памяти программ и памяти данных происходит с разделением времени в сопровождении селектирующих сигналов $\overline{\text{PMS}}$ (*Program Memory Select*) и $\overline{\text{DMS}}$ (*Data Memory Select*) и сигналов синхронизации $\overline{\text{WR}}$ и $\overline{\text{RD}}$.

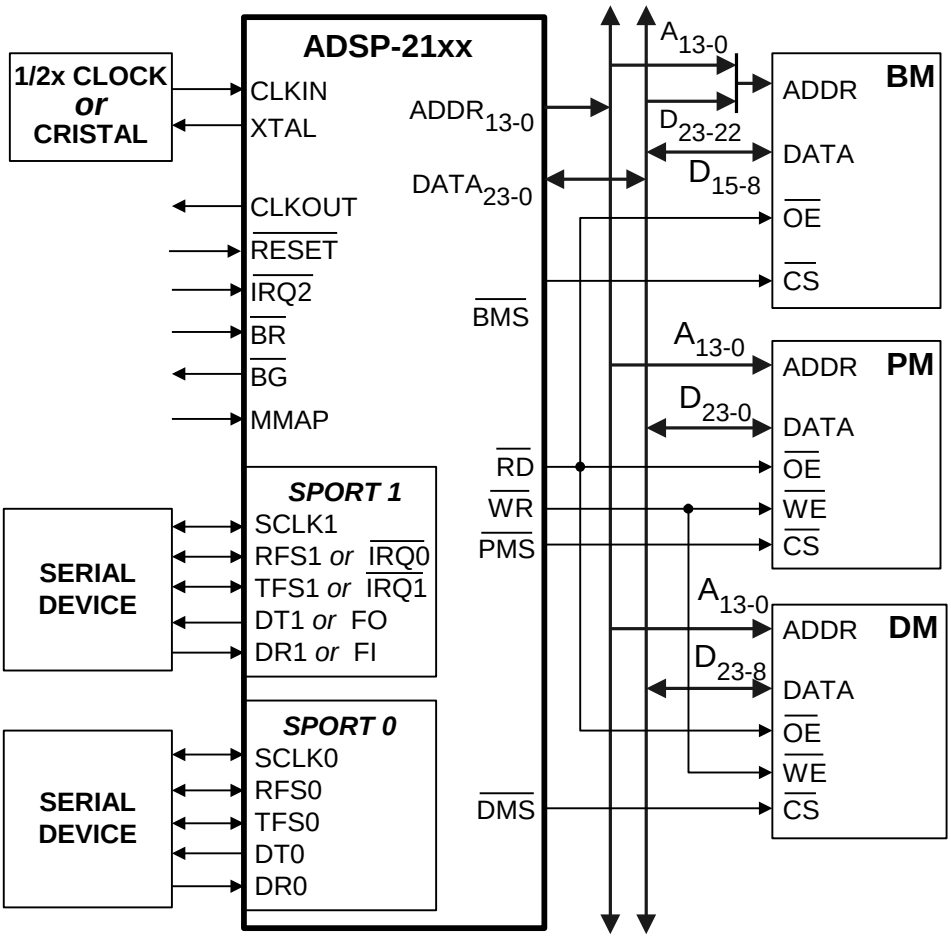


Рис. 6.2. ADSP-21xx система с внешней памятью

Память данных не является однородной и делится на блоки с устанавливаемым для каждого блока временем доступа. Часть адресного пространства DM выделена для регистров (портов) ввода/вывода.

Кроме оперативной памяти PM и DM в состав микросистемы входит внешнее перепрограммируемое постоянное запоминающее устройство (ППЗУ) *с байтовой организацией* BM (*Boot Memory*). Обычно это 8-разрядное ППЗУ, в котором хранится код начальной загрузки, переносимый в память программ при старте ЦПС или при его программном сбросе. Нередко в качестве ППЗУ применяется флэш-память. Для доступа к BM используется селектирующий сигнал $\overline{\text{BMS}}$ (*Boot Memory Select*). В случае, когда объём BM превышает свойственное процессору адресное пространство, недостающие для BM биты адреса передаются по внешней шине данных в мультиплексном режиме. В представ-

ленной на рис. 6.2 системе для этого используются линии D_{23-22} старших разрядов.

Интерфейс памяти с байтовой организацией можно построить так, что он будет поддерживать загрузку по ходу выполнения программы. При этом содержимое определённого числа ячеек внутренней памяти ЦПС может быть передано во внешнюю память или получено из неё без перерыва основной работы процессора.

В системе на рис. 6.2 использован процессор из конкретного семейства. Поэтому есть особенности его применения, требующие пояснений. Касаются они размещенных на кристалле ЦПС устройств ввода/вывода и принципиального значения не имеют. Кроме того, пояснений требуют сигналы, которыми процессор обменивается с внешней средой.

1) В составе ЦПС имеются два синхронных последовательных порта – SPORT 0 и SPORT 1. Принимаемые и передаваемые через эти порты данные (линии DR0,1 – *Data Receive* и DT0,1 – *Data Transmit* соответственно) формируются в кадры и независимо от направления передачи сопровождаются сигналами побитовой SCLK0,1 и кадровой RFS0,1 (*Receive Frame Synchronization*) и TFS0,1 (*Transmit Frame Synchronization*) синхронизации. Сигналы побитовой синхронизации представляют собой периодическую последовательность импульсов SKLC0 или SKLC1. Линии сигналов синхронизации – двунаправленные. Это делает возможной синхронизацию либо со стороны последовательного порта, либо со стороны связанного с ним устройства.

2) Последовательный порт SPORT 1 наделен дополнительными функциями: его можно сконфигурировать не как сериальный порт, а как порт для восприятия запросов прерываний $\overline{\text{IRQ0}}$ и $\overline{\text{IRQ1}}$ от внешних устройств, а также для передачи флагов готовности IF (*Input Flag*) и OF (*Output Flag*) по вводу и выводу.

3) Для внешних запросов прерываний имеется выделенная линия $\overline{\text{IRQ2}}$. Эта линия обычно используется для управления работой ADSP 21xx системы. Управляющим устройством может быть, например, компьютер, к которому ADSP 21xx система подключена как устройство ввода/вывода. Управляющее воздействие принимается по запросу прерывания, передаваемому по линии $\overline{\text{IRQ2}}$. Процессор, получив запрос $\overline{\text{IRQ2}}$, запускает процедуру обработки прерывания, действия которой сводятся к чтению командного слова и вызову соответствующей этому командному слову функции.

4) Сигналы $\overline{\text{BR}}$ (*Bus Request*) и $\overline{\text{BG}}$ (*Bus Grant*) – это, соответственно, сигнал запроса и сигнал подтверждения прямого доступа к памяти. Сигнал $\overline{\text{BR}}$ поступает со стороны внешнего по отношению к ADSP 21xx ведущего на системной магистрали, например, от управляющей ЭВМ тогда, когда необходимо прочитать данные из памяти или записать данные в память ADSP 21xx системы.

5) Для синхронизации процессора используется либо внешний сигнал CLKIN, либо сигнал внутреннего генератора с подключенным к нему через контакты CLKIN и XTAL кристаллом кварца.

6) Для аппаратного сброса процессора применяется сигнал $\overline{\text{RESET}}$. После сброса процессор считывает из ВМ программный код, заносит его в свою оперативную память программ и приступает к выполнению программы.

7) Высоким или низким уровнями напряжения на входе MMAP (*Memory Map*) устанавливается одна из двух возможных конфигураций памяти данных, определяющих то, как распределено адресное пространство между внутренней и внешней памятью DM.

6.1.2. Управление выполнением программы

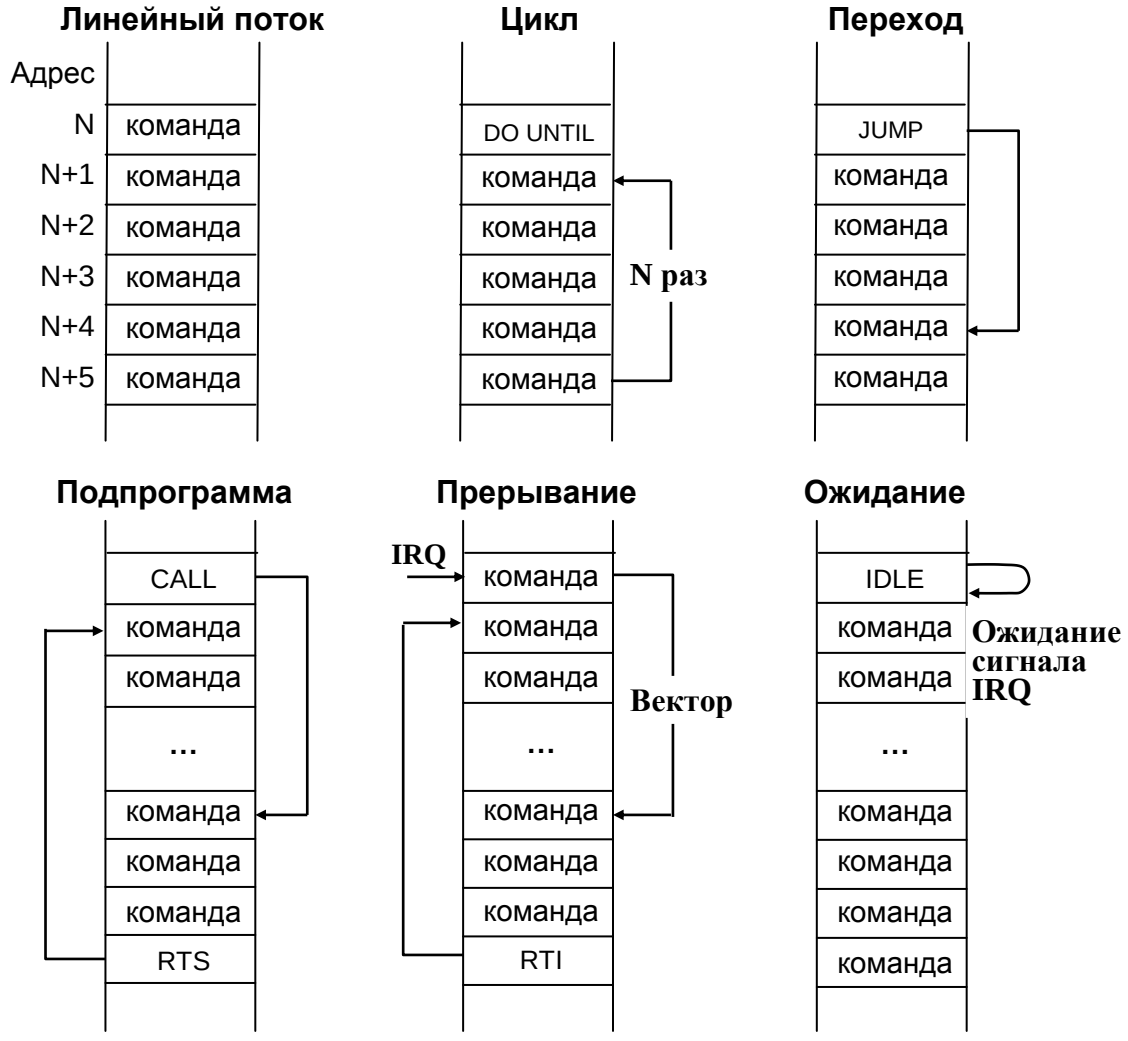


Рис. 6.3. Структуры, изменяющие линейный поток программы

Большей частью процессор выполняет программу линейно, последовательно извлекая команды из памяти и исполняя их (рис. 6.3). Изменения в линейном потоке команд обусловливаются *переходами* (JUMP), *вызовами проце-*

дур (CALL) *и циклами* (DO UNTIL – делай, пока выполнено условие). Вызов процедуры предполагает возврат, который выполняется по команде RTS – *Returne from Subroutine*.

Линейное исполнение программы нарушается также вследствие *запросов прерываний* (*Interrupt Request* – IRQ), когда выполнение основной программы прерывается событием, которое происходит во время её выполнения. Процедура обработки прерывания всегда завершается командой возврата из прерывания RTI – *Return from Interrupt*.

Большое значение для встроенных систем имеет наличие возможности перехода в режим пониженного потребления энергии. Особенно это касается микроконтроллеров и цифровых процессоров сигналов. Для этого используются встраиваемые в МП средства и, в частности, перевод микропроцессора в энергосберегающий режим работы с помощью специальной команды (команда IDLE на рис. 6.3). По этой команде процессор прекращает операции по обработке данных и переходит в режим о*жидания* с пониженным энергопотреблением, оставаясь восприимчивым к запросам прерываний. Когда запрос прерывания поступает, процессор обрабатывает его, после чего продолжает стандартное выполнение программы.

Вызовы, прерывания и циклы могут поддерживаться как аппаратными, так и программными средствами. Однако алгоритмическая сторона этой поддержки в своей основе остается без изменений. Применение аппаратных средств сокращает накладные расходы (связанные, например, с обращениями к стеку) и временные затраты на вызовы, возвраты и организацию циклов

Выполнением программы цифрового процессора сигналов семейства ADSP 21xx управляет программный автомат (*Program Sequencer*). Особенностью этого устройства (рис. 6.4) является наличие аппаратных (интегрированных в кристалл МП) стеков для хранения адресов возврата из процедур и поддержки циклов.

Пока процессор выполняет текущую команду, программный автомат (ПА) осуществляет предварительную выборку следующей команды. Адрес этой команды определяется одним из четырёх источников:

- инкрементор программного счётчика (*Program Counter* – PC),
- стек программного счётчика (PC стек),
- регистр команд IR,
- контроллер прерываний.

Каждый из этих источников связан с соответствующим портом мультиплексора следующего адреса. Коммутацией портов управляет устройство выбора следующего адреса, работа которого основывается на данных, поступающих

1) из регистра команд,
2) от логического устройства проверки выполнения условий переходов/вызовов,

3) от компаратора циклов, который проверяет адрес текущей команды на совпадение с адресом последней команды в теле цикла, что обеспечивает переход к его первой команде, если условие выхода из цикла не выполнено,
4) из контроллера прерываний.

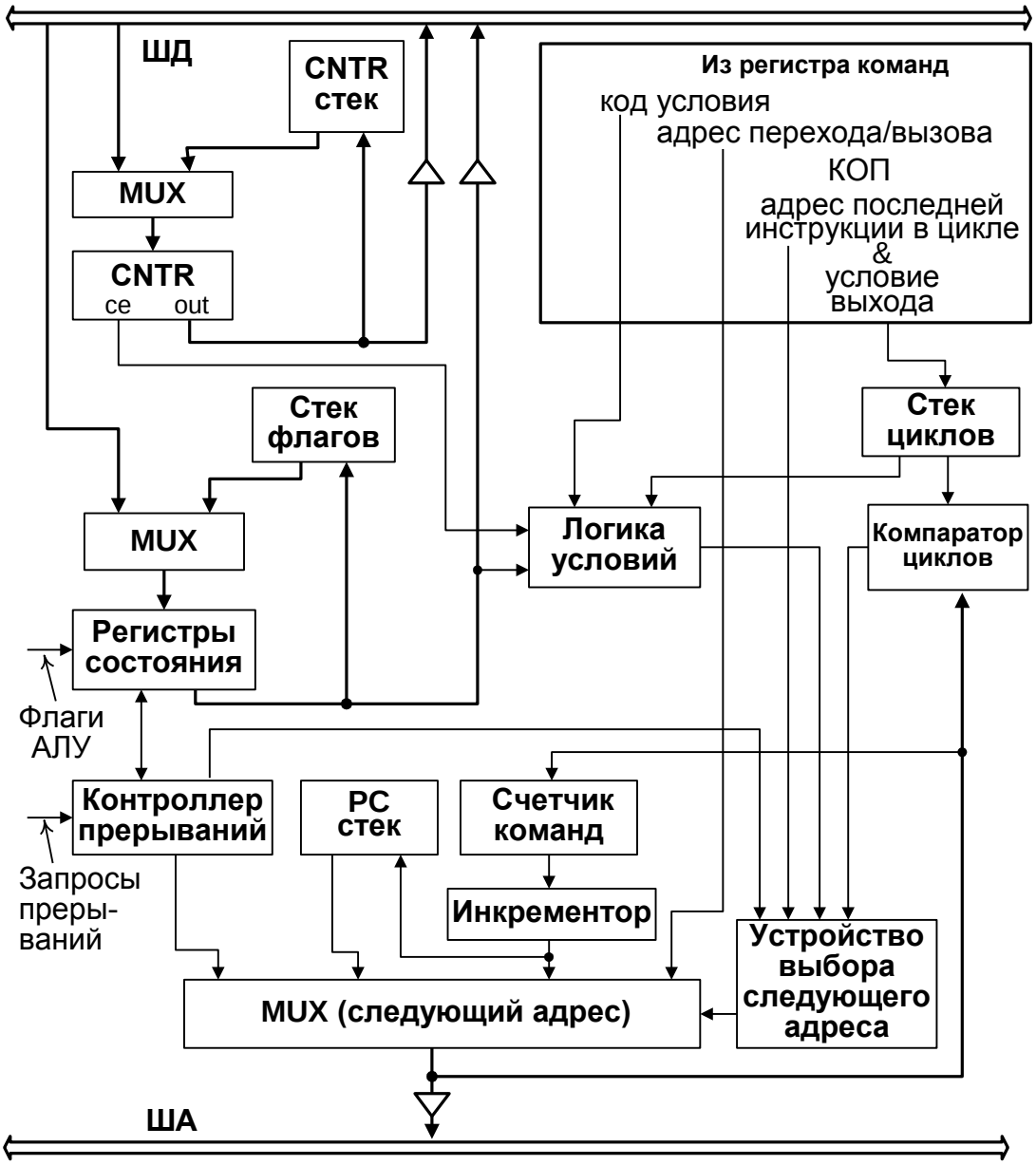


Рис. 6.4. Программный автомат МП семейства ADSP 21xx.

Рассмотрим сначала работу устройства управления без участия контроллера прерываний, о функциях которого поговорим позднее после рассмотрения действий ПА, не связанных с реакцией на события. Такой режим работы может быть установлен в любом процессоре с помощью специального запрещающего восприятие запросов прерывания флага IF – *Interrupt Flag* в регистре (или в одном из регистров) состояния процессора.

При последовательном выполнении программы, когда не предпринимается переход или возврат из процедуры, или когда заканчивается цикл DO UNTIL, в качестве источника следующего адреса выбирается инкрементор программного счётчика PC. Выходное значение инкрементора выводится на шину адреса ША, по которой адрес передаётся в память, и одновременно загружается в PC, заменяя его предшествующее значение для последующей выборки команды из памяти.

Микропроцессоры семейства ADSP 21xx, в формате команд которых отражены черты RISC архитектуры (все команды МП семейства ADSP 21xx упакованы в 24-разрядные слова), обладают тем преимуществом, что при линейном выполнении программы значение программного счётчика всегда наращивается на единицу, и для этого не ружны какие-либо операции по вычислению исполнительного адреса.

При выполнении перехода по адресу, указанному в команде (команда типа JUMP), выход мультиплексора следующего адреса переключается на вход, связанный с регистром команд IR, и адрес перехода берётся из команды, зафиксированной в IR во время предыдущего цикла выборки команды из памяти. Если переход условный, то предварительно *логика условий* проверяет выполнение условия перехода, сравнивая флаги ALU с указанным в коде операции условием. Переход реализуется, если условие выполнено, и откладывается в противном случае.

Точно также при выполнении команды вызова (команды типа CALL) из регистра команд берется адрес перехода на подпрограмму. Но при этом дополнительно используется стек PC, в который при выполнении команды вызова подпрограммы помещается значение инкрементора. При возвращении из подпрограммы по команде RTS выход мультиплексора следующего адреса переключается на вход, связанный со стеком PC, и сохраненный в стеке PC адрес используется как адрес возврата.

Стек PC используется при выполнении циклических операций по команде DO UNTIL. Адрес первой команды в теле цикла сохраняется в вершине стека PC. Одновременно адрес последней команды и условие завершения цикла размещаются в *стеке цикла*. При каждом очередном проходе тела цикла *компаратор циклов* сравнивает генерируемый программным автоматом адрес с адресом последней команды тела цикла (этот адрес содержится в команде DO UNTIL). При выполнении последней команды цикла процессор проверяет условие выхода из цикла и, если оно не выполнено, делает условный переход на начало цикла, выбирая в качестве источника адреса стек PC. Выполнение цикла контролирует логика сравнения флагов АЛУ с указанным в команде DO UNTIL условием. Для поддержки вложенных циклов используется *стек циклов*, на вершине которого всегда находятся параметры выполняемого в текущий момент времени цикла.

Циклы могут выполняться по заданному числу итераций. Для организации таких циклов используются *счётчик циклов* **(**CNTR**)** и *стек счётчика*

(CNTR стек**)**. Счётчик цикла работает в режиме вычитания и перед исполнением цикла в него через шину данных ШД заносится число, задающее количество проходов тела цикла. Окончание цикла фиксирует *логика условий* по обнулению счетчика – по условию CE (счётчик пуст).

Стек счётчика циклов позволяет организовывать вложенные циклы. Каждый раз при активном сигнале CE содержимое вершины стека автоматически извлекается и загружается в счётчик цикла, что позволяет продолжить выполнение внешнего цикла (если таковой имеется). Для случаев, когда требуется преждевременный выход из цикла, предусматривается возможность непосредственного извлечения содержимого стека счётчика. Текущее значение счётчика автоматически помещается в стек при загрузке в него нового значения с шины данных ШД.

В стеке PC сохраняется также адрес инструкции, при исполнении которой возник запрос прерывания. *При обработке прерываний* кроме адреса возврата сохраняется состояние (флаги) процессора. Для этого используется *стек флагов*. При завершении процедуры обработки прерывания – состояние процессора восстанавливается[6]. Этим отличаются действия процессора при вызове процедуры обработки прерывания от действий при программном вызове.

Запросы прерываний от устройств ввода/вывода (УВВ), расположенных на кристалле процессора или за его пределами, принимает *контролер прерываний*. Контроллер прерываний разбирается в приоритетах запросов и наиболее высокоприоритетный из них транслирует процессору. В схеме на рис. 6.4 сигнал о таком запросе поступает на *устройство выбора следующего адреса*, а адрес, по которому определяется путь к программе обработки запроса прерывания (ISR), передаётся на вход мультиплексора следующего адреса. Получив сигнал от контроллера прерываний, устройство управления мультиплексором следующего адреса переключает выход мультиплексора на вход, связанный с контроллером прерываний, и на шину адреса ША передается сформированный контроллером адрес. Этот адрес относится к одному из элементов таблицы, расположенной в основной памяти и называемой *таблицей прерываний*.

Каждый из элементов таблицы прерываний содержит либо стартовый адрес ISR (или команду перехода на ISR), либо саму ISR, если размер ее программного кода не превышает размера элемента таблицы. Число элементов в таблице определяется числом устройств, которые процессор может обслуживать в режиме прерываний. Таблица прерываний обычно расположена, начиная с нулевого адреса памяти.

В качестве примера в таблице 6.1 представлена таблица прерываний МП ADSP 2171, каждый элемент которой содержит по четыре слова. Первому по списку элементу соответствует наиболее высокий приоритет, а последнему –

[6] Стеки счетчика команд, циклов, счетчика циклов и состояния имеют ограниченное количество вложений. Однако глубина стеков подобрана таким образом, чтобы по возможности исключить обращение к памяти при выполнении наиболее часто выполняемых процедур. Обращение к памяти становится необходимым, если уровни вложений превышены.

наиболее низкий. Особенностью таблицы прерываний МП семейства ADSP 21xx является то, что длина слова в ней, как и длина слов в памяти, где расположена программа, равна 24 битам. При байтовой организации памяти адреса элементов в таблице прерываний определяются адресами соответствующих байтов.

Помимо аппаратных запросов прерывания в таблице представлены два вектора программных прерываний, использование которых вызывает действия, подобные действиям при вызове подпрограммы, но с сохранением состояния процессора в стеке.

Таблица 6.1

Источник прерывания	**Адрес вектора прерываний**
Запуск программы после сигнала «сброс» (или после выхода из режима пониженного энергопотребления)	0x0000 (высший приоритет)
Понижение потребляемой мощности	0x002C
Запрос прерывания о внешнего источника, поступивший на контакт микросхемы IRQ2	0x0004
Порт интерфейса с управляющей цифровым процессором машиной в режиме записи	0x0008
Порт интерфейса с управляющей цифровым процессором машиной в режиме чтения	0x000C
Интегрированный в МП кристалл последовательный порт SPORT0 в режиме «передача»	0x0010
Интегрированный в МП кристалл последовательный порт SPORT0 в режиме «прием»	0x0014
Программируемое прерывание	0x0018
Программируемое прерывание	0x001C
Интегрированный в МП кристалл последовательный порт SPORT1 в режиме «передача» или запрос прерывания от внешнего источника через контакт микросхемы IRQ1	0x0020
Интегрированный в МП кристалл последовательный порт SPORT1 в режиме «прием» или запрос прерывания от внешнего источника через контакт микросхемы IRQ0	0x0024
Интегрированный в кристалл МП таймер	0x0028 (низший приоритет)

Переходя к обработке запроса прерывания, процессор автоматически сбрасывает флаг прерываний IF в соответствующем регистре состояния процессора, запрещая тем самым обработку других запросов. Реакция на другие запросы возможна только после того, как процедура обработки текущего запроса установит флаг IF или тогда, когда выполнится команда RTI возврата из прерывания, по которой IF вместе с другими флагами процессора восстанавливается автоматически, благодаря чему вновь разрешается восприятие запросов прерываний. Именно этим обусловлены действия по сохранению регистров (или регистра, если он один) состояния процессора при вызовах ISR. Этот механизм

обслуживания запросов прерываний является универсальным и свойственен в основных своих чертах всем современным микропроцессорам.

При обработке прерываний статусные регистры процессоров ADSP 21xx сохраняются в стеке флагов, а при окончании обработки прерывания – восстанавливаются.

Есть два типа запросов прерывания – маскируемые и немаскируемые. К немаскируемым запросам прерывания относится, например, запрос от источника питания, возникающий при сбое в его работе. Немаскируемые прерывания за счет аппаратно реализованных стеков обрабатываются без задержки (кроме задержки, связанной с синхронизацией). Маскируемые (не обрабатываемые в текущее время) прерывания обозначаются в регистре маски контроллера прерываний, являющемся программно доступным регистром.

Для обслуживания высокоприоритетных запросов предусмотрен механизм вложенных прерываний, который позволяет обслуживать запросы, если они поступают во время исполнения ранее вызванной процедуры обработки прерывания ISR, и если уровень приоритета вновь поступившего запроса превышает приоритет обрабатываемого. Но такое обслуживание возможно только в том случае, если восприятие вложенных запросов прерывания обеспечено со стороны процессора.

При получении запроса прерывания процессор фиксирует его на время, которое необходимо, чтобы закончилось выполнение текущей команды. Затем контроллер прерываний сравнивает поступивший запрос с регистром маски прерываний. В случае немаскируемого прерывания программный автомат помещает текущее значение счетчика команд в стек PC. Статусные регистры (в том числе и регистр маски) помещаются в стек состояния. В регистр маски загружается новое значение, которое определяет, разрешается или не разрешается дальнейшее использование вложенных прерываний.

Затем процессор выполняет команду NOP (нет операции), одновременно выбирая команду, расположенную по адресу вектора прерывания. По возвращении из подпрограммы обслуживания прерывания восстанавливаются значения счётчика команд и регистров состояния (включая регистр маски) путём извлечения их содержимого из стека PC и стека состояний и выполнение прерванной программы возобновляется.

Каждое отдельное прерывание (внешнее или внутреннее)

- маскируется или разрешается,
- конфигурируется на восприятие по перепаду или по уровню.

Имеется возможность принудительного прерывания, а также сброса задержанного чувствительного к перепаду прерывания. Для этого устанавливаются соответствующие биты в одном из регистров процессора, отвечающем за конфигурацию прерываний. Запросы внешних прерываний $\overline{\text{IRQ0}}$, $\overline{\text{IRQ1}}$ и $\overline{\text{IRQ2}}$ (см. рис. 6.2) могут быть сконфигурированы как воспринимаемые по перепаду или по уровню.

При обработке сконфигурированных по перепаду прерываний запрос на прерывание фиксируется каждый раз, когда уровень сигнала запроса $\overline{IRQ0-2}$ на соответствующем контакте микросхемы процессора изменяется от высокого к низкому. Запрос фиксируется до окончания обслуживания прерывания, а затем он автоматически сбрасывается. Задержанные прерывания по перепаду могут также сбрасываться программно путем установки соответствующего бита в одном из регистров управления прерываниями.

Прерывания по перепаду требуют, как правило, меньших аппаратных затрат, чем прерывания по уровню, что позволяет использовать в качестве запросов прерывания такие сигналы, как синхроимпульсы (например, импульсы, определяющие частоту дискретизации аналого-цифровых преобразователей).

Прерывания по уровню не фиксируются процессором и должны оставаться подтвержденными до окончания обслуживания. При этом действие запроса прекращается устройством, от которого запрос поступил. Если этого не произошло, то «не сброшенное» прерывание будет обработано повторно. У прерываний по уровню есть особенность, заключающееся в том, что за счет логического комбинирования одна и та же запросная линия может быть использована несколькими источниками прерывания.

Возможность использования вложенных прерываний задается программным путем. В режиме, не разрешающем использование вложенных прерываний, все запросы на прерывания автоматически маскируются и запрещаются при входе в процедуру обработки прерывания. В режиме, разрешающем использование вложенных прерываний действует система приоритетов.

Когда вложенные прерывания заблокированы, то при входе в процедуру обработки прерывания всех уровней автоматически маскируются. Если вложенность прерываний разрешена, то регистр маски устанавливается таким образом, что маскируются только прерывания с равным или более низким приоритетом. Конфигурация прерываний с более высокими приоритетами сохраняется.

Если сигнал прерывания по перепаду возникает в то время, когда это прерывание замаскировано, то запрос фиксируется, но не обслуживается. Так возникает отложенное прерывание. Затем это прерывание может быть обработано. Любое из отложенных прерываний можно сбросить программными средствами с использованием соответствующих команд.

В процессорах ADSP 2171, ADSP 2181 предусмотрены глобальные команды разрешения и блокирования прерываний ENA INTS и DIS INTS,

6.1.3. Устройства обработки данных

На рис. 6.5 и 6.6 представлены блок-схемы двух устройств, входящих в ядро ЦПС ADSP 21xx – арифметическо-логического устройства (ALU) и умножителя-аккумулятора (MAC).

Входными для ALU являются два регистра AX (AX0 и AX1) и два регистра AY (AY0 и AY1). Результат выполняемой в ALU операции сохраняется в регистре результата AR (ALU *Result*) или в регистре временного хранения AF (регистре обратной связи – ALU *Feedback*). ALU формирует также признаки (флаги) нулевого (AZ) и отрицательного результата (AN), переполнения (AV) и переноса (AC). Имеется операция, связанная с определением знака операнда, принимаемого через порт X. Результатом этой операции является признак знака X-операнда. Вход CI ALU предназначен для приема бита переноса, что бывает необходимо в операциях двойной точности.

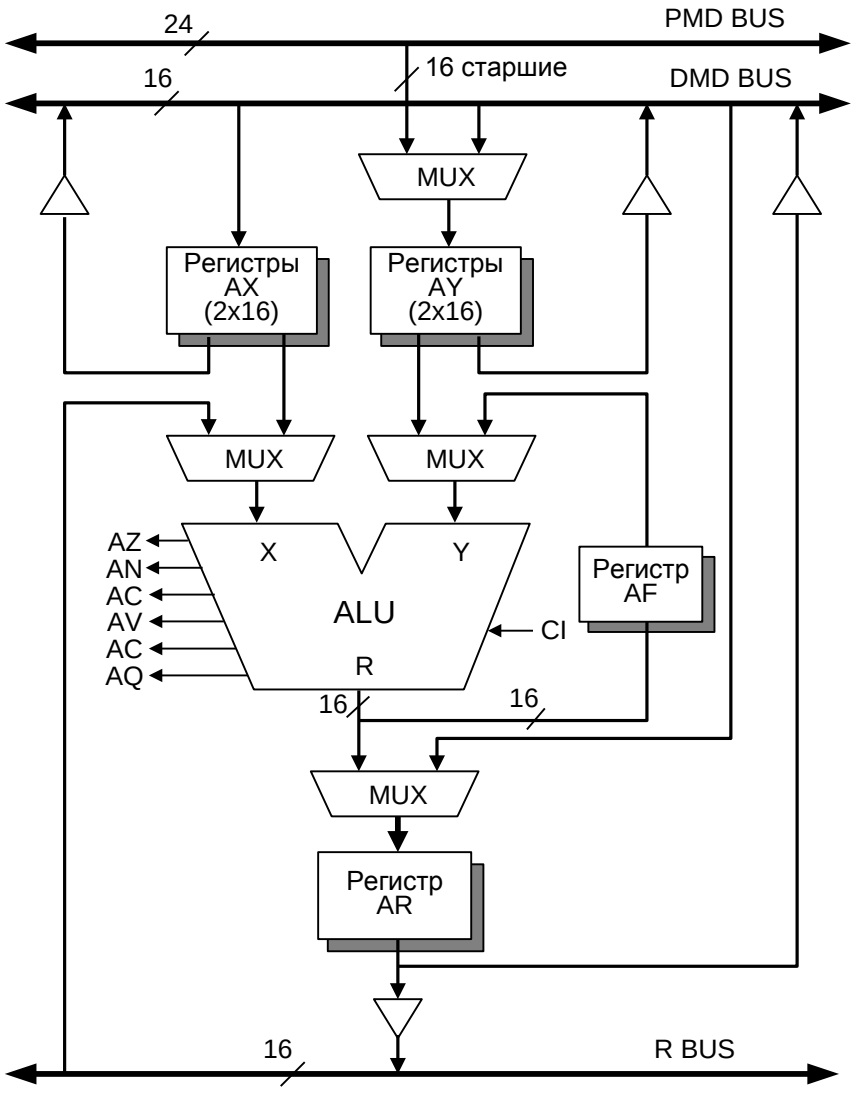


Рис. 6.5. Арифметическо-логическое устройство ЦПС ADSP 21xx.

В ALU выполняются также операции целочисленного деления. Деление происходит в пошаговом режиме со сдвигом делимого и каждый шаг вычисления сопровождается формированием бита частного, оформляемого как признак AQ.

Умножитель-аккумулятор также имеет два входных порта с регистрами MX (MX0 и MX1) и MY (MY0 и MY1). Регистр результата MR (MAC *Result*)

составлен из двух 16-разрядных (MR0 и MR1) и одного 8-разрядного (MR2) регистров. Формируется один бит состояния – бит переполнения умножения MV. Имеется регистр временного хранения (регистр обратной связи) MF – MAC *Feedback*.

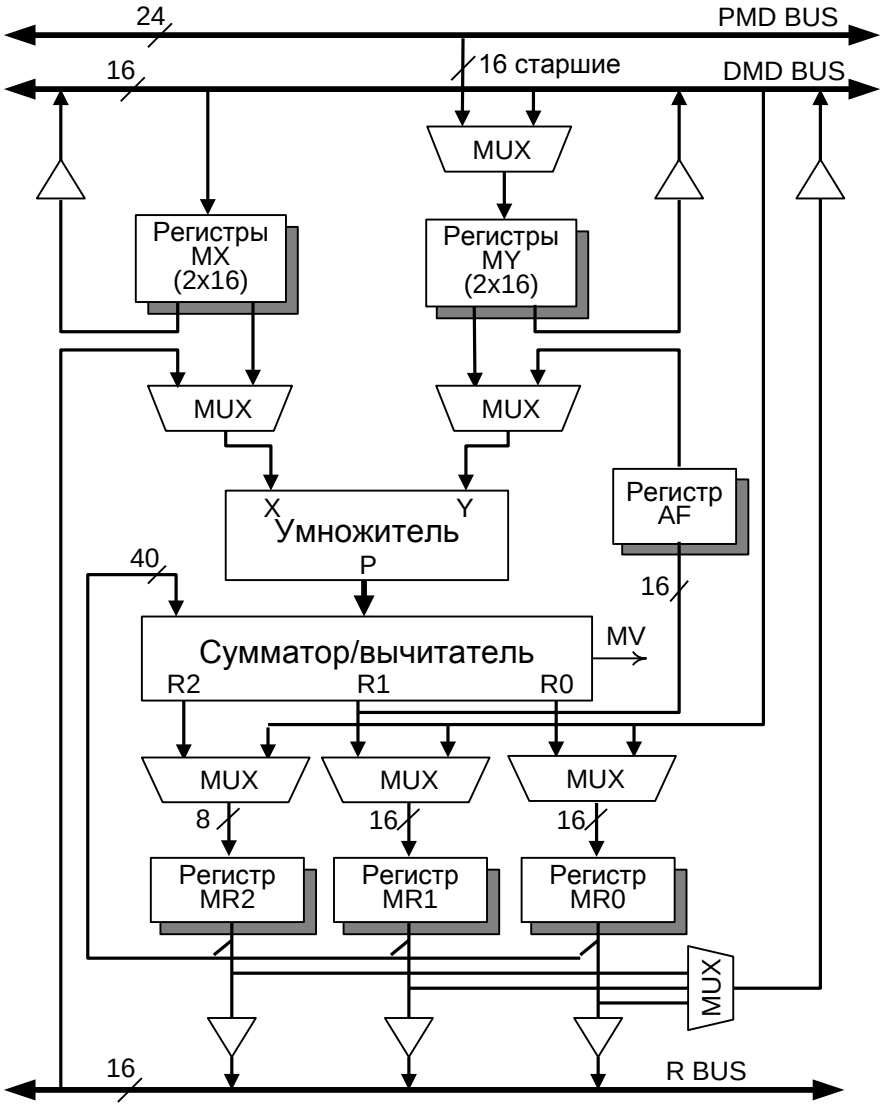


Рис. 6.6. Умножитель-аккумулятор ЦПС ADSP 21xx.

Устройство сдвига помимо стандартных сдвиговых операций выполняет операции нормализации и денормализации мантисс и операции с показателями чисел с плавающей точкой. Поскольку результат сдвиговых операций требует большего чем у операнда на входе числа разрядов, то регистр результата сдвигателя SR (Shifter *Result*) составлен из двух 16-разрядных регистров SR1 и SR2. Кроме SR имеются входной регистр SI (Shifter *Input*) и относящий к числу выходных регистр показателя SE (Shifter *Exponent*), необходимый при работе с числами с плавающей точкой.

Все регистры ALU, MAC и устройства сдвига доступны для чтения и записи через шину данных памяти данных DMD BUS. Через порты Y в ALU и MAC могут быть приняты значения шины данных памяти программ PMD BUS.

Кроме этого, выходные регистры каждого из вычислительных устройств (регистры AR, MR и выходной регистр сдвигателя SR) могут быть операндами любого другого вычислительного устройства благодаря тому, что обмен информацией между ALU, MAC и устройством сдвига может осуществляться через шину результата R BUS.

Альтернативные (вторичные) регистры и контекстное переключение

Архитектурные решения компонент операционного блока МП ADSP 21xx мало отличаются от стандартных. Существенной особенностью является то, что в ЦПС семейства ADSP 21xx, как и во всех последующих, предусмотрен дополнительный (альтернативный, вторичный) набор регистров, которые в нужное время могут заменить основные регистры. К ним относятся входные и выходные регистры AX, AY, AR и AF арифметическо-логического устройства, регистры MX, MY, MR и MF умножителя-аккумулятора , а также входной SI и выходные SR и SE регистры устройства сдвига. На рис. 6.5и 6.6 регистры, при которых есть альтернативные регистры, отмечены оттенёнными прямоугольниками.

Необходимость в подмене основных регистров альтернативными возникает при вызовах процедур, особенно при обработке запросов прерываний. Переключение с одного набора регистров на другой происходит под управлением специальных команд

ENA SEC_REG – разрешить вторичные регистры и
DIS SEC_REG – запретить вторичные регистры.

Вызываемое этими командами переключение с одного набора регистров на другой происходит без затрат времени на сохранение использовавшихся до вызова регистров. Задержка переключения равна длительности одного такта процессора. Действия, связанные с сохранением/восстановлением регистров при вызовах и возвратах называют *сохранением/восстановлением контекста*, или *контекстным переключением*. Время контекстного переключения является одной из важнейших характеристик систем реального времени.

Механизм поддержки реального времени, основанный на использовании альтернативных регистров, имеет ограничения, обусловленные тем, что допускает только один уровень вложения процедур с контекстным переключением. Поэтому его целесообразно применять лишь при многократно повторяющихся вызовах одной и той же (желательно короткой) процедуры, предпочтительно процедуры обработки прерывания. Примером может послужить обработка прерываний при вводе данных от аналого-цифрового преобразователя (АЦП), когда запросы прерывания идут с частотой дискретизации, а сама процедура обработки состоит их небольшого числа команд, предназначенных для ввода данных из регистра данных АЦП и сохранения их в памяти. Поскольку процедура работает с альтернативными регистрами, в нее дополнительно необходимо

включать исполнение команд ENA SEC_REG при входе в процедуру и DIS SEC_REG перед выходом из нее.

Здесь уместно напомнить о механизме регистровых окон в процессорах с регистрово-ориентированной архитектурой, используемом при аппаратных и программных вызовах процедур: операции по сохранению регистров (контекста) процессора заменяются предоставлением вызываемой процедуре отдельного регистрового окна (см. раздел 8.2). Регистры МП ADSP 21xx вместе с альтернативными регистрами можно рассматривать «двухоконный» набор регистров. Отличие от действительно двухоконного набора состоит в том, что регистры процессоров ADSP 21xx не оформлены как отдельный регистровый блок, а распределены по компонентам операционного блока – находятся в ALU, в MAC и в устройстве сдвига.

6.1.4. Адресация памяти данных

В каждом процессоре семейства ADSP 21xx имеются два генератора адресов данных DAG1 и DAG2, реализующих косвенно-регистровую адресацию данных с автоматической модификацией адреса. В обоих генераторах предусмотрена возможность организации циклических буферов. Генератор DAG1 работает только с памятью данных, а DAG2 может генерировать адреса памяти программ и памяти данных, чем обеспечивается *одновременный доступ к данным*, расположенным в DM и в PM.

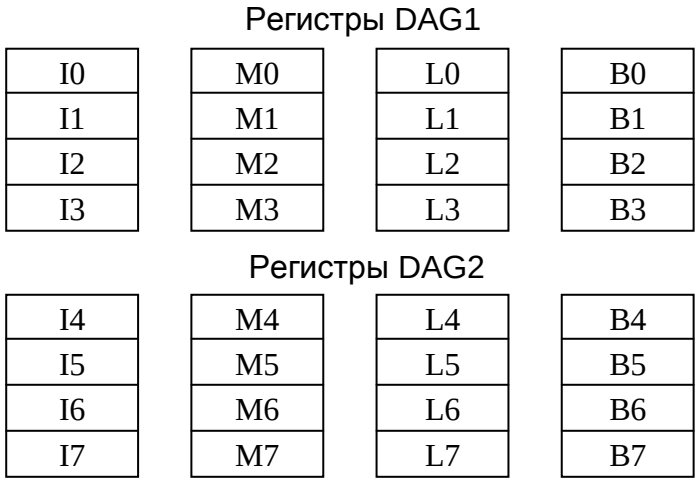


Рис. 6.7. Регистровая модель генераторов адресов данных

Регистровая модель генераторов адресов данных представлена на рис. 6.7. В DAG1 входят регистры базы B*n* (*n* = 0–3), индексные регистры I*n* регистры-модификаторы M*n* и регистры L*n* для работы с циклическими массивами. Точно такие же регистры B*n*, I*n*, M*n* и L*n* с номерами *n* = 4–7 имеются и в DAG2.

При работе с массивами базовый адрес массива указывается в регистре B*n*. Индексные регистры I*n* всегда работают в паре с соответствующим по номеру *n* регистром базы B*n*, и в них содержится адрес текущего элемента массива, который при необходимости может автоматически наращиваться или уменьшаться для указания на следующий элемент. Предусмотрено автоматическое изменение содержимого индексных регистров с помощью регистров-модификаторов M*n*. Регистры M*n* не имеют строгой привязки к индексным регистрам. В них содержится лишь набор модификаторов, значения и порядок использования которых зависят от размеров элементов массива, а также от шага и направления движения по массиву. Но при этом регистры-модификаторы одного генератора адресов данных не могут быть использованы в другом

Обычные и циклические массивы различаются по значениям регистров L*n*. Ненулевое значение регистра L*n* определяет длину организованных как очередь массивов. Для индексации элементов циклического массива используются регистры L*n* с номером *n*, совпадающим с номером регистра базы B*n*. Если содержимое какого-либо из регистров L*n* равно нулю, то это означает, что соответствующий массив не является циклическим. Следствием этого является требование обязательной инициализации регистров L*n* до вступления генератора адреса данных в работу

Вычислительные устройства и устройство сдвига процессоров ADSP 21xx работают только с операндами, расположенными в регистрах. Обращение к памяти поддерживается отдельным набором инструкций, которые служат для загрузки и сохранения регистров в памяти. Для примера можно привести следующие инструкции:

`AX0 = DM(I3,M0)` – загрузка входного регистра ALU AX0 содержимым ячейки памяти данных DM, адрес которой указан в регистре I3, с последующей модификацией I3 по содержимому M0;

`MR1 = PM(I5,M5)` – загрузка регистра MR1 умножителя-аккумулятора содержимым ячейки памяти программ PM, адрес которой указан в регистре I5, с последующей модификацией I5 по содержимому M5;

`PM(I4,M4) = AR` – сохранение регистра AR ALU в ячейке памяти программ PM по адресу, указанному в регистре I1, с последующей модификацией I1 по содержимому M3;

`DM(I1,M3) = MR0` – сохранение регистра результата MR0 MAC в ячейке памяти данных DM, адрес которой указан в регистре I4, с последующей модификацией I4 по содержимому M4;

Если модификация адреса не требуется, то модификатор в команде все равно указывается, но его значение берется равным нулю. В командах Ассемблера L- и B-регистры не показываются и их значения определяются на стадии компиляции и компоновки программы.

Наряду с адресацией по указателю к памяти можно обращаться по адресу, указанному в команде – в режиме прямой адресации, например, так как это показано в командах

DM(*адрес*) = AR – сохранить регистр в DM по указанному адресу,
MX0 = DM(*адрес*) – загрузить регистр MX0 из ячейки DM с указанным адресом

6.1.5. Особенности программирования

Есть две формы написания инструкций на языке Ассемблера: мнемоническая и алгебраическая. Традиционно наиболее широко распространена мнемоническая форма. Для написания инструкций цифровых процессоров, производимых фирмой Analog Devices, используется алгебраическая форма, которая ближе к языкам высокого уровня.

Все команды ALU, MAC и устройства сдвига МП ADSP 21xx выполняются по условию. Однако использование в команде кода условия не является обязательным и применяется лишь в случае необходимости. В качестве примера можно привести команду ALU сложения с переносом:

```
IF AC AR = AX0 + AY0 + C.
```

В этой команде присутствует необязательное условие IF AC. Поэтому в данном случае проверяется на наличие бита переноса ALU (AC), и команда выполняется, если по результатам предыдущей команды был выставлен флаг переноса. В противном случае выполняется NOP и выполнение программы продолжается со следующей команды. Алгебраическое выражение AX0+AY0+C означает, что в регистр результата ALU (AR) записывается сумма значений входных регистров плюс значение бита переноса.

В качестве другого примера возьмём команду для умножителя-аккумулятора

```
IF NOT MV MR = MR + MX0*MY0(UU).
```

По условию IF NOT MV в ней проверяется состояние бита переполнения умножителя. Если условие ложно, выполняется NOP. Выражение MR=MR+MX0*MY0 обозначает операцию умножения с накоплением: в регистр результата MR, образованный двумя 16-разрядными регистрами MR1 и MR0 и одним 8-разрядным регистром MR2, записывается сумма его текущего значения и произведения операндов X и Y из выбранных регистров ввода. Модификатор в скобках (UU) означает, что операнды являются беззнаковыми величинами. Наряду с UU применяются другие модификаторы:

- модификатор SS для операции со знаковыми операндами;
- модификаторы SU и US для случаев, когда один из операндов представляет число со знаком, а другой – число без знака.

Имеется еще один модификатор, означающий округление (подразумевается знакового) результата.

[IF *условие*] | AR / AF | = *xop* | + *yop* / + C / +*yop* + C / + *const* / + *const* + C |;

[IF *условие*] | AR / AF | = *xop* | – *yop* / – *yop* + C – 1 / + C – 1 / – *const* / – *const* + C – 1 |;

[IF *условие*] | AR / AF | = *xop* | AND / OR / XOR | | *yop* / *const* |;

DIVS *yop*, *xop*; {*Определить знак результата*}
DIVQ *xop*; {*Определить разряд в частном*}

Рис. 6.8. Команды ALU МП ADSP 21xx.

[IF *условие*] | MR / MF | = *xop* * | *yop* / *xop* | (| SS / SU / US / UU / RND |);

[IF *условие*] | MR / MF | = MR + *xop* * | *yop* / *xop* | (| SS / SU / US / UU / RND |);

[IF *условие*] | MR / MF | = MR – *xop* * | *yop* / *xop* | (| SS / SU / US / UU / RND |);

Рис. 6.9. Команды MAC МП ADSP 21xx.

На рис. 6.8 и 6.9 представлены команды ALU и MAC. Они не составляют полного перечня исполняемых этими операционными блоками команд и выбраны для того, чтобы проиллюстрировать правило, по которому строятся команды для МП ADSP 21xx.

В списке команд *условие* используется для обозначения всех возможных условий, которые могут быть проверены, а аббревиатуры *xop* и *yop* – для обозначения регистров-операндов по X и Y вводам ALU и MAC. X-операндами

каждого из операционных блоков могут быть свои входные X-регистры и регистры результата AR и MR вычислительных устройств и регистр результата устройства сдвига SR. Y-операндами являются свои входные Y-регистры и только свои регистры результата и регистры обратной связи AF и MF.

На рисунках

- квадратные скобки обозначают дополнительную (необязательную) часть команды;
- между параллельными вертикальными линиями помещается список операндов. Выбирается один из перечисленных в списке операндов;
- заглавными буквами обозначены команды и имена регистров;
- аббревиатура *const* обозначает непосредственные операнды;
- аббревиатура *dreg* (см. ниже) относится к любому из регистров данных (к регистрам ALU, MAC и устройства сдвига).

ALU выполняет также команды деления. Деление начинается с определения знака результата (команда DIVS *xop, yop*), а затем в пошаговом режиме путем исполнения команды DIVQ *xop* последовательно, начиная со старшего определяются биты результата. Число шагов равно числу значащих бит результата. Команда DIVQ имеет один операнд (делитель), поскольку второй операнд (делимое) до начала процедуры деления должен находиться в регистрах AY и AF – в двух регистрах потому, что делимое имеет вдвое большее число разрядов, чем делитель.

Здесь не представлены команды устройства сдвига, т. к. с точки зрения построения команд их рассмотрение не дает существенно нового.

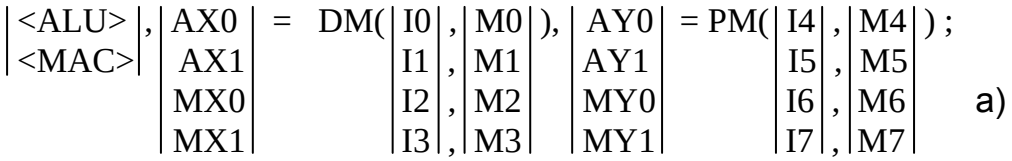
Помимо простых команд, относящихся только к выполнению операций с данными, имеются многофункциональные команды, позволяющие эффективно использовать преимущества гарвардской архитектуры – одновременное выполнение за один цикл пересылок данных, операций записи/считывания в/из память(и) и вычислений. На рис. 6.10 приведен список многофункциональных команд. Команды внутри многофункциональной команды отделяются запятой, комбинированная многофункциональная команда оканчивается точкой с запятой.

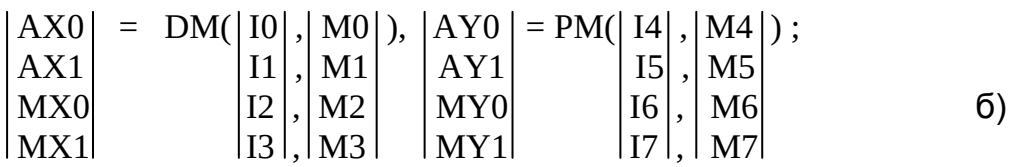
Операции ALU и MAC (рис. 6.10а) могут выполняться одновременно с загрузкой регистров AX, AY или MX, MY из памяти данных DM и памяти программ PM.

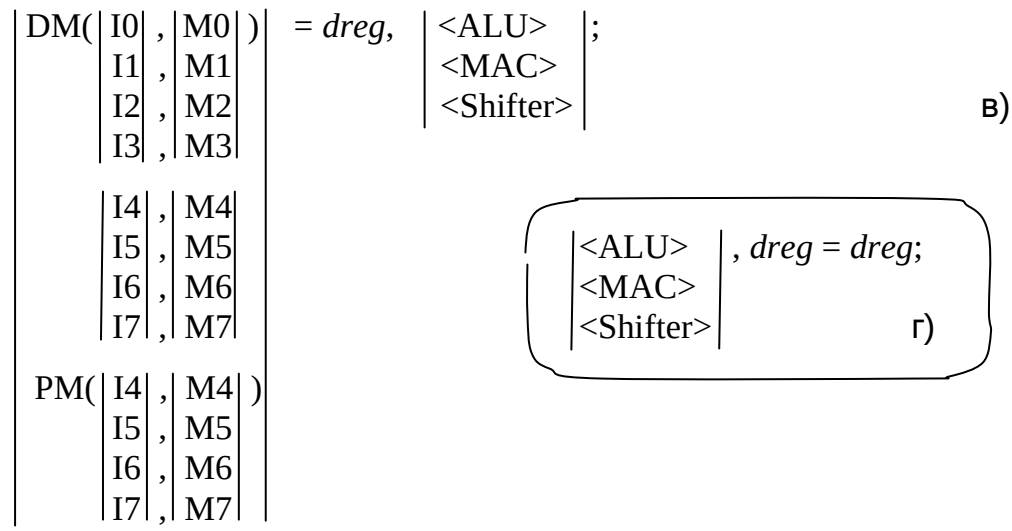
Адреса ячеек DM определяются генератором адресов данных DAG1 с привлечением в качестве указателей индексных регистров I0-I3[7] с регистрами-модификаторами M0-M3, а адреса PM – генератором DAG2 с индексными регистрами I4-I7 и модификаторами M4-M7. Каждый из регистров-модификаторов задает приращение (уменьшение) значения следующего адреса по отношению к текущему, содержащемуся в индексном регистре, с которым соответствующий модификатор работает. Модификатор в команде должен быть

[7] Имена регистров I0-I3, M0-M3, I4-I7 и M4-M7 показываются в командах.

обязательно указан. Если адрес не модифицируется, то используется регистр-модификатор с нулевым содержимым.







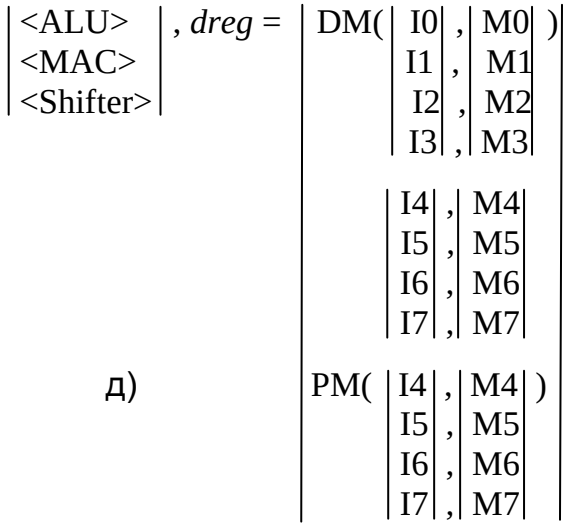


Рис. 6.10. Многофункциональные команды МП ADSP 21xx.

Частным случаем являются команды, связанные только с обращением к памяти (рис. 6.10б).

Операции устройства сдвига, как и операции ALU и MAC, могут выполняться одновременно с загрузкой/сохранением значений регистров данных либо в памяти программ PM, либо в памяти данных DM – рис. 6.10г, д.

Пересылки данных между регистрами данных также могут выполняться одновременно с операциями ALU, MAC и устройства сдвига (рис. 6.10д).

6.2. Средства повышения производительности ядра процессора и многопроцессорные системы

Цифровые процессоры сигналов развиваются в направлении увеличения производительности и расширения функциональных возможностей. Это отражается в устройствах обработки данных, в организации и объеме памяти, в работе с устройствами ввода-вывода. Важное значение имеет то, насколько эффективно используются вычислительные ресурсы процессора. В значительной степени повышение этой эффективности достигается за счет конвейера операций, при котором действия процессора распределяются на отдельные фазы (циклы), связанные

- с определением адреса следующей выбираемой из памяти команды,
- с циклом считывания команды из памяти,
- с декодированием команды и выработкой условий и сигналов, управляющих выполнением команды и, наконец,
- с выполнением команды.

При последовательном выполнении программы в то время, когда одна команда выбирается, команда выбранная несколькими (в представленном перечне тремя) циклами ранее, выполняется. Кроме этого, отдельные этапы конвейера могут быть разбиты на дополнительные подэтапы. Однако нужно заметить, что повышение числа уровней конвейера влечет за собой сложности в управлении, обусловленные, прежде всего, возникающими по ходу исполнения программы ветвлениями и переходами. Наличие последних заставляет прибегать к механизму опережающей выборки команд и к предсказанию ветвлений. В противном случае эффективность конвейера операций значительно снижается из-за необходимости его перезагрузки при поступлении команд переходов.

Производительность процессора во многом зависит от эффективности использования регистров процессора. Доступность регистров со стороны различных вычислительных устройств ядра процессора дает возможность параллельного исполнения ими вычислительных. Отсюда следует, что регистровый блок процессора должен быть многопортовым.

Конвейер операций и эффективное использование регистровой памяти – это одна из ключевых особенностей RISC процессорам. Поэтому RISC архитектурные решения рассматриваются как наиболее рациональные, и их широко используют производители как простых микроконтроллеров, так и сложных однокристальных вычислительных систем

6.2.1. Ядро ЦПС семейства ADSP 2106x

Тенденцию развития микропроцессорных систем проследим, продолжив рассмотрение цифровых процессоров сигналов фирмы Analog Devices.

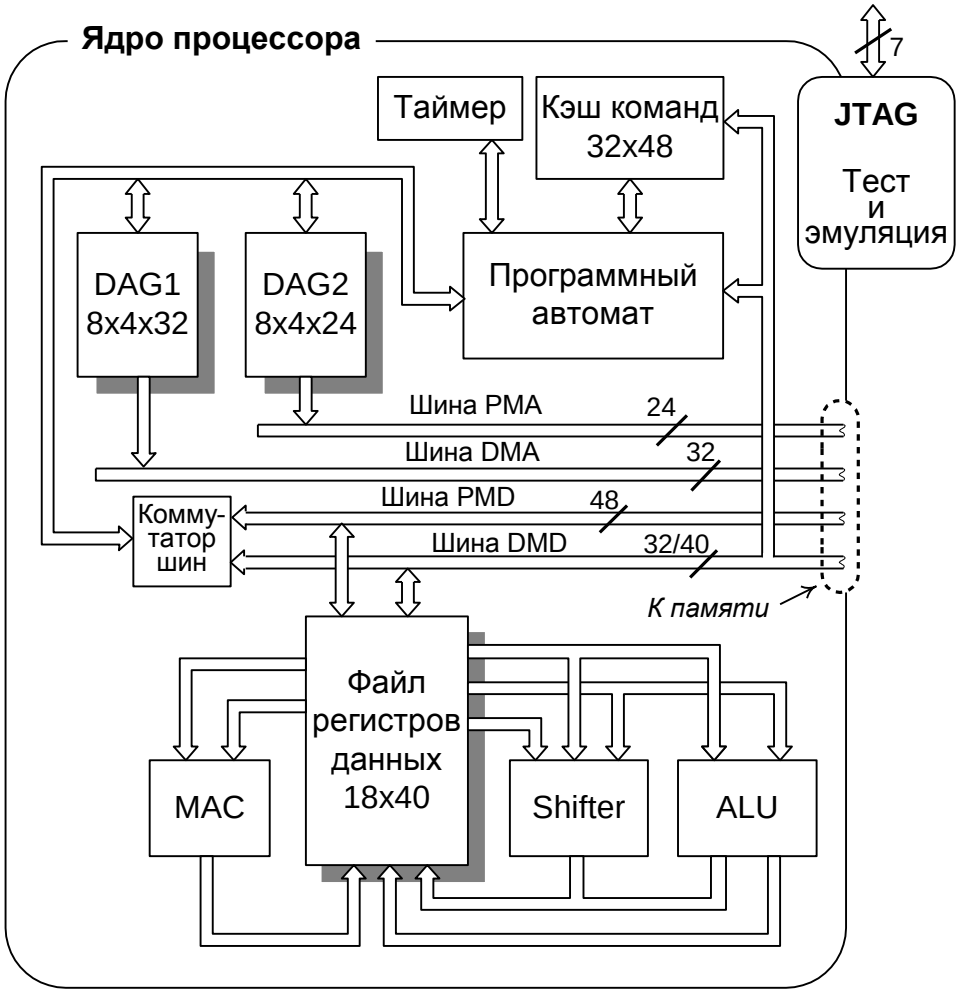


Рис. 6.11. Структура ядра ЦПС семейства ADSP 2106x

На рис. 6.11 представлена структурная схема ядра ЦПС семейства ADSP 2106x с архитектурой SHARC. В состав ядра ADSP 2106x входят те же функциональные блоки, что и в процессорах семейства ADSP 21xx: программный автомат, арифметическо-логическое устройство ALU, умножитель-аккумулятор MAC, устройство сдвига Shifter и два генератора адресов данных DAG1,2. Но повышена разрядность регистров и шин PMA, DMA, PMD и DMD.

Шина адреса памяти программы PMA имеет 24-разряда и, как следствие, 24-разрядными являются регистры генератора адресов данных DAG2. Число разрядов в шине данных памяти программы PMD, как и в ADSP 21xx, обусловлено разрядностью команд процессора – все команды МП ADSP 2106x являются 48-разрядными. Это позволило расширить возможности многофункциональных команд – все они стали условными, за исключением команд, управляющих выполнением операций с регистрами при одновременном доступе к памяти программы и к памяти данных.

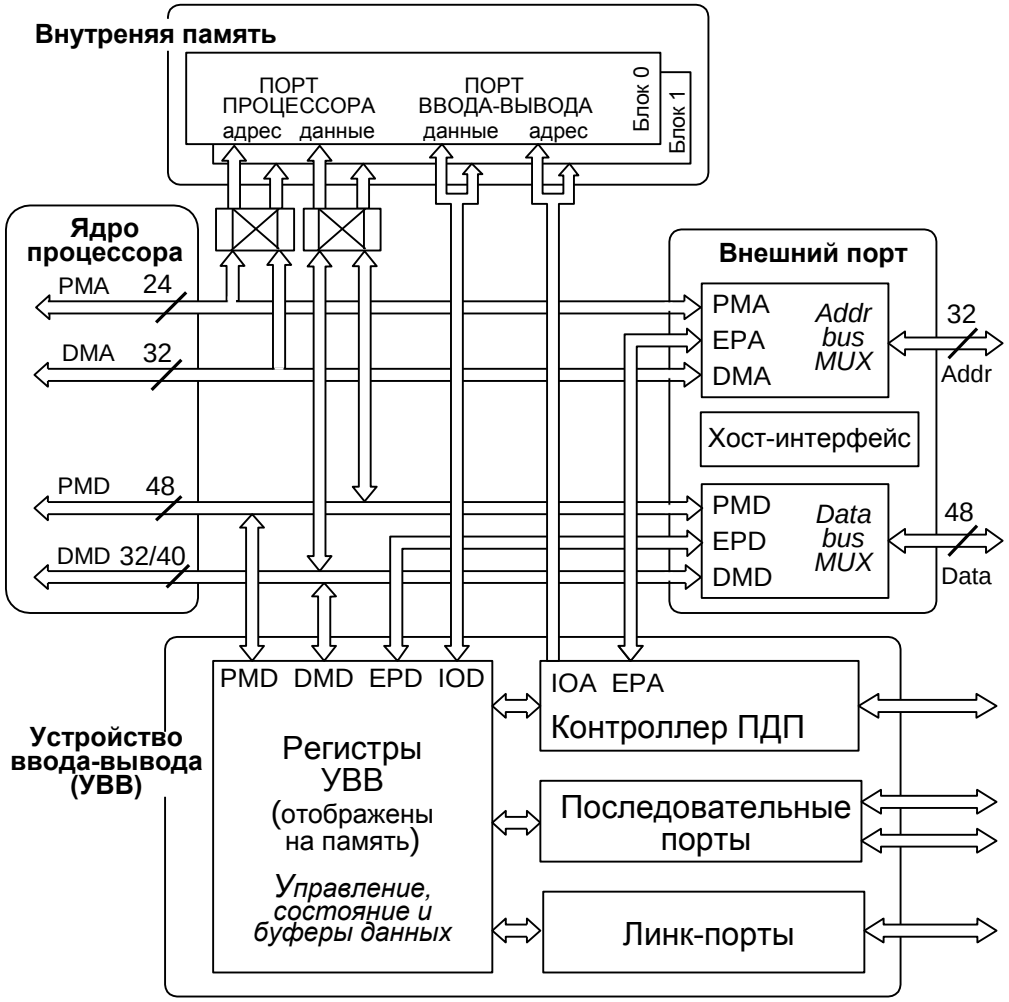


Рис. 6.12. Структура процессора ADSP 2106x

Расширено адресное пространство памяти данных, обращение к которой происходит по 32-разрядной шине адреса DMA и 40-разрядной шине данных DMD.

Операнды и промежуточные результаты работы MAC, ALU и устройства сдвига хранятся и запоминаются в 10-портовом файле регистров данных. В каждом цикле синхронизации регистровый файл может поддерживать следующие операции: (1) запись и считывание из регистрового файла двух операндов, (2) выдача двух операндов на входы ALU, (3) выдачу двух операндов на входы MAC и (4) сохранение результатов операций ALU и MAC.

Процессор ADSP 2106x обрабатывает команду за три цикла:

1. В цикле *выборки* команда считывается из кэша команд или из памяти программы.
2. В цикле *декодирования* производится декодирование команды определяются условия, необходимые для управления выполнением команды.

3. В цикле *выполнения* завершаются предписанные командой действия.

Эти циклы перекрываются и образуют конвейер. При последовательном выполнении программы, в то время, когда одна команда выбирается, команда, выбранная в предыдущем цикле, декодируется, а команда, выбранная двумя циклами ранее, выполняется.

Для быстрого переключения контекста, например, при обработке прерываний, регистровый файл имеет два набора регистров (первичный и вторичный), каждый из которых включает 16 40-разрядных регистров.

Два набора регистров имеют также генераторы адресов данных. Регистры, активные после сброса процессора, относятся к первичным, а другие – к дополнительным (или вторичным). Какой из наборов регистров является активным в определенный момент времени, определяют соответствующие биты в регистре управления режимом работы процессора.

У процессоров ADSP 2106x *нет физического разделения памяти на память программ и память данных*. В этом состоит особенность гарвардской архитектуры ADSP 2106x. Внутренняя память процессора делится на блоки (рис. 6.12), в которых могут быть размещены и программа, и данные. Внешняя память разбита на банки. Место нахождения программы и данных определяется тем, как память сконфигурирована.

Обычно программный автомат выбирает команду из памяти в каждом машинном цикле. Иногда возникают ситуации, в которых одновременно (в одном цикле) инициируются обращения по шине PMD к данным и к программе. Если команда выбирается из памяти программ в то время, когда выполняется команда пересылки данных по шине данных PMD, может возникнуть конфликтная ситуация. Для устранения подобных конфликтных ситуаций процессор кэширует команды.

Когда возникают проблемы с выборкой какой-то команды, процессор записывает эту команду в кэш для того, чтобы в будущем исключить возникновение конфликта. Программный автомат проверяет кэш команд при каждом обращении по шине данных PMD. Если необходимая команда находится в кэше, то для ее выборки не требуется обращения к памяти программ и команда из кэша выбирается параллельно с обращением к данным памяти программы и без задержки. Если необходимой команды в кэше нет, то команда выбирается из памяти в цикле, следующем за обращением к данным памяти программы. Таким образом добавляется один непроизводительный цикл. Команда загружается в кэш, и если кэш разблокирован, то она будет доступна при повторных возникновениях подобного рода конфликтов.

Если при выполнении команды, находящейся по адресу n, делается обращение к данным по шине PMD и возникает конфликт при обращении блоку памяти, то этот конфликт связан с выборкой команды по адресу n+3 (предполагается последовательное выполнение программы), а не с исполняемой командой по адресу n. Это обусловлено опережающей выборкой и конвейерным выпол-

нением команд. Как следствие, в кэше сохраняется команда с адресом *n*+3, а не та команда, для выполнения которой потребовалось обращение к данным памяти программы. При последовательном выполнении программы это не сказывается на эффективности работы кэш-памяти. Однако при ветвлениях и переходах вероятность кэш-промахов увеличивается.

Режимом работы кэш-памяти можно управлять. За это отвечает соответствующий регистр процессора. В частности, использование кэша можно запретить или разрешить.

JTAG-порт поддерживает комплекс действий по тестированию соединений в системе, использующих методику последовательной проверки результатов ввода-вывода во всех компонентах системы в соответствии с предложенным IEEE стандартом.

6.2.2. Подсистема ввода-вывода ЦПС семейства ADSP 2106x

Процессоры ADSP 2106x содержат внутреннюю статическую оперативную память в виде двух двухпортовых блока, которые могут конфигурироваться для различных комбинаций хранения кода и данных. В одном цикле к каждому блоку памяти могут независимо обращаться ядро процессора (с использованием шин PMA, PMD и DMA, DMD) и контроллер DMA в устройстве ввода-вывода (с использованием шин IOA – *Input-Output Address* и IOD – *Input-Output Data*). Используя шины DMD и PMD, ядро процессора может обращаться к внешней или внутренней памяти в одном и том же машинном цикле.

Имеется система арбитража шин для обработки конфликтующих обращений. Приоритеты арбитража фиксированы и установлены в следующем порядке (по убыванию): обращения по шине DMD, обращения по шине PMD, обращения со стороны устройства ввода-вывода (УВВ) с использованием шины IOD. Кроме того, время обращения УВВ (которое в пакетном режиме может быть достаточно большим) не может выходить за устанавливаемые временные пределы, поэтому предоставление шины ядру процессора никогда не задерживаются более чем на 4 цикла.

В состав устройства ввода-вывода входят

- порт стандартного последовательного интерфейса с периферийными устройствами SPI – *Serial Peripheral Interface*,
- порт универсального асинхронного приемопередатчика UART – *Universal Asynchronous Receiver/Transmitter*,
- шесть 4-разрядных портов связи – линк-портов для организации многопроцессорных систем и
- контроллер прямого доступа к памяти – контроллер ПДП.

Обмен данными с периферийными устройствами (ПУ) происходит преимущественно в режиме прямого доступа к памяти под управлением контроллера ПДП, что позволяет более эффективно использовать ресурсы ядра при вы-

полнении операций по обработке поступающих от периферийных устройств данных. Для конфигурации и управления периферийными устройствами, для промежуточного хранения передаваемых и принимаемых от ПУ данных, а также для конфигурации и управления работой контроллера ПДП служат отображенные на память *регистры УВВ*. Обращение к регистрам УВВ происходит с использованием шин DMA и DMD. По этим же шинам при необходимости ядро процессора может обращается к входящим в состав устройства ввода-вывода регистрам данных.

Интерфейс с внешней памятью и внешними периферийными устройствами обеспечивает *внешний порт* с 32-разрядной шиной адреса и 48-разрядной шиной данных, которые образованы путем мультиплексирования внутренних шин адреса (PMA, DMA и EPA – *External Port Address*) и данных (PMD, DMD и EPD – *External Port Data*). Кроме этого через внешний порт осуществляется связь с хост-машиной (с главной машиной).

6.2.3. Ресурсы и архитектура многопроцессорной системы на базе ADSP 2106x

Процессоры семейства ADSP 2106x имеют функциональные характеристики, которые позволяют создавать многопроцессорные системы. Используются две схемы связи между процессорами. По одной схеме с помощью портов связи (линк-портов) реализуются соединения «точка-точка». По другой процессоры совместно используют глобальную память системы и связь между процессорами осуществляется по общей шине, для чего имеются встроенный арбитраж шины и возможность обращения к внутренней памяти и регистрам устройств ввода-вывода других ADSP 2106x. Число объединённых общей шиной процессоров не должно превышать шести.

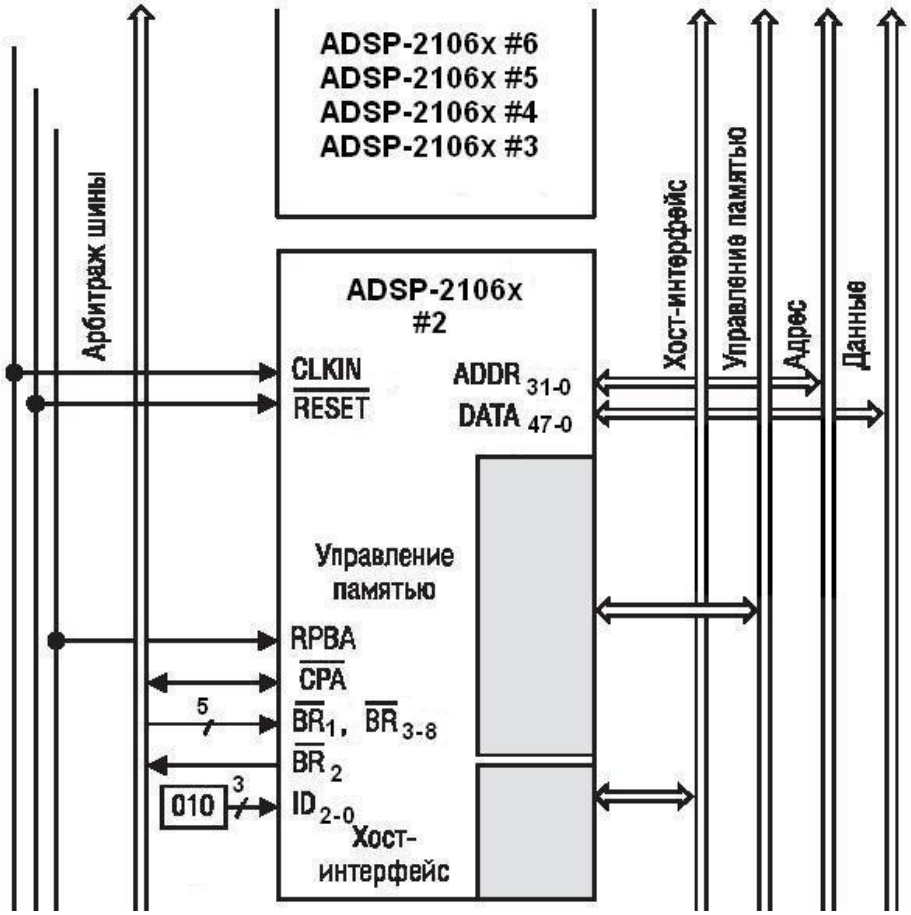
Процессоры ADSP 2106x обращаются к внешней памяти и портам ввода-вывода через внешний порт. Внешнее адресное пространство включает *пространство памяти многопроцессорной системы* (память на кристаллах других ADSP 2106x), а также *пространство внешней памяти* (область памяти, расположенной вне кристаллов).

Ядро процессора и устройство ввода-вывода имеют доступ к внешним шинам ($DATA_{47\text{-}0}$, $ADDR_{31\text{-}0}$) через внешний порт ADSP 2106x (рис. 6.12, 6.13). Внешний порт обеспечивает доступ к памяти, размещённой вне кристалла, и к периферийным устройствам. Через него можно обращаться к внутренней памяти других ADSP 2106x, соединенных в многопроцессорную систему. Схема соединения с общей шиной позволяет реализовывать одно объединенное адресное пространство, в котором могут храниться и код, и данные.

Внешняя память может быть 16-, 32- или 48-разрядная; контроллер прямого доступа в память автоматически упаковывает внешние данные в слова соответствующей разрядности: 48-разрядные команды или 32-разрядные данные.

В многопроцессорной системе любой из процессоров может стать ведущим. Ведущий процессор берёт на себя управление шиной, которая включает линии шины данных $DATA_{47-0}$ и шины адреса $ADDR_{31-0}$, а также линии управления. Ведущий ADSP 2106x способен блокировать шину для выполнения неделимой последовательности операций *чтение-модификация-запись* для семафоров.

В таблице 6.2 перечислены и определены выводы микросхемы ADSP 2106x, используемые для интерфейса с внешней памятью. Сигналы управления памятью дают возможность соединения как с быстрыми (например, со статическими), так и с медленными устройствами памяти, а также с разными по быстродействию периферийные устройствами (адресное пространство ввода/вывода в процессорах ADSP 2106x отображено на память). Пользователь программными средствами может определять нужную комбинацию состояний ожидания и аппаратных сигналов подтверждения связи. Сигналы на выводах $\overline{SBTS}$ (перевод шины в третье состояние) и PAGE (граница страницы) могут использоваться для интерфейса с динамической памятью (DRAM).



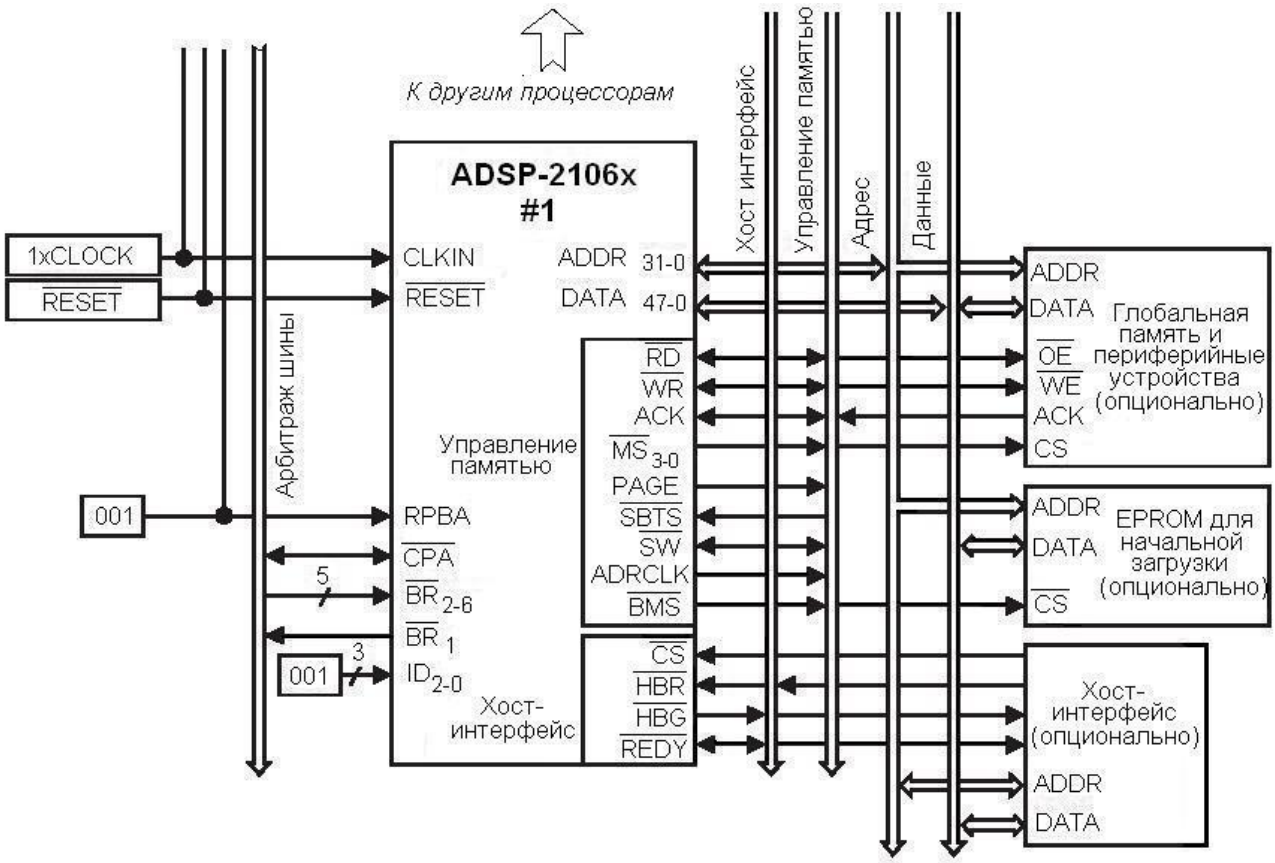


Рис. 6.13. Многопроцессорная система на базе ADSP 2106x

Таблица 6.2

Вывод	Тип	Функция
$ADDR_{31-0}$	I/O/T	**Внешняя шина адреса.** По линиям внешней шины процессор выводит адреса внешней памяти и периферийных устройств. В многопроцессорной системе их активизирует ведущий процессор, обращаясь к внутренней памяти других ADSP 2106x или к регистрам УВВ. ADSP 2106x принимает адресные сигналы от хост-процессора или другого ведущего в многопроцессорной системе, когда производится считывание или запись в его внутреннюю память или в регистры УВВ.
$DATA_{47-0}$	I/O/T	**Внешняя шина данных.** По линиям шины данных процессор получает и выводит данные и команды. 32-разрядные данные с фиксированной точкой и 32-разрядные данные с плавающей точкой передаются битами 47-16, 40-разрядные данные с плавающей точкой повышенной точности – битами 47-8, 16-разрядные короткие слова данных – битами 31-16.
$\overline{MS}_{3-0}$	O/T	**Линии выбора памяти.** Сигнал низкого уровня на этих линиях используется для выбора кристалла соответствующего банка внешней памяти. Размер банка памяти должен быть определен в регистре управления системой. В многопроцессорной системе сигналы на линиях выводятся ведущим на системной магистрали.
$\overline{RD}$	I/O/T	**Строб чтения памяти**. Этот сигнал сопровождает считывание из внешней памяти или из внутренней памяти других ADSP 2106x. Сигнал $\overline{RD}$ выставляют также внешние устройства (включая другие ADSP 2106x) для считывания данных из внутренней памяти ADSP 2106x. В многопроцессорных системах сигнал $\overline{RD}$ выводится ведущим и принимается всеми другими процессорами.
$\overline{WR}$	I/O/T	**Строб записи в память**. Активизируется, когда процессор выполняет запись во внешнюю или во внутреннюю память других ADSP 2106x. Внешние устройства (включая другие ADSP 2106x) должны выставлять строб $\overline{WR}$ для записи данных во внутреннюю память ADSP 2106x. В многопроцессорных системах сигнал $\overline{WR}$ выводится ведущим и принимается всеми другими ADSP 2106x.
PAGE	O/T	**Граница страницы DRAM.** Сигнал активизируется при пересечении границы страницы внешней динамической памяти (DRAM). Размер страницы DRAM определяется в регистре управления временем доступа к памяти WAIT. DRAM может быть размещена только в банке 0 внешней памяти; сигнал PAGE может быть использован только для обращений к банку 0. В многопроцессорных системах PAGE выводится ведущим.

Продолжение таблицы 6.2		
$\overline{SW}$	I/O/T	**Выбор синхронной записи** (*Synchronous Write Select*). Этот сигнал используется для синхронного взаимодействия ADSP 2106x с устройствами памяти (включая внутреннюю память других ADSP 2106x). Процессор использует $\overline{SW}$ для предварительного указания на то, что предполагается выполнение операции записи, которая может и не произойти, если строб $\overline{WR}$ не будет установлен (например, при выполнении условной команды записи). В многопроцессорной системе сигналом $\overline{SW}$ управляет ведущий процессор, а все остальные его принимают. $\overline{SW}$ выставляется тогда, когда выводится адрес. Хост-процессор, использующий синхронную запись, должен выставлять этот сигнал при записи в ADSP 2106x.
ACK	I/O/S	**Подтверждение доступа к памяти** (*Memory Acknowledge*). ACK используется устройствами ввода/вывода, контроллерами памяти, другими периферийными устройствами, чтобы путём сброса этого сигнала продлить цикл обращения к внешней памяти. ADSP 2106x в режиме синхронной записи в его внутреннюю память может сбросить выходной сигнал подтверждения ACK, если с целью обеспечения синхронизма требуется продлить цикл обращения к его внутренней памяти.

Пояснения к таблице: I/O – вход/выход, S – синхронный, T – с тремя состояниями.

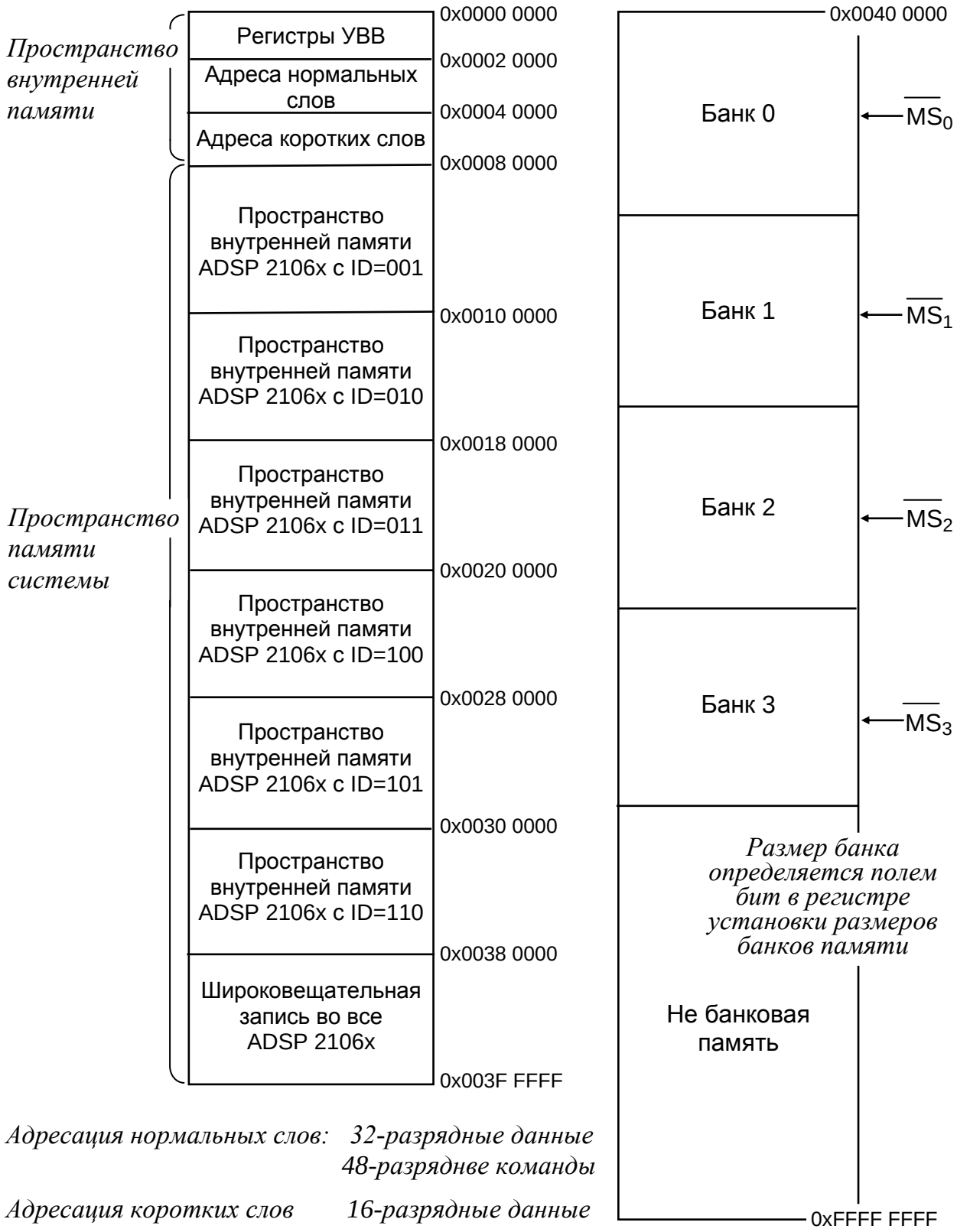


Рис. 6.14. Карта памяти ADSP 2106x

Карта памяти ADSP-2106x

Адресное пространство МП ADSP 2106x делится на три части: пространство внутренней памяти, пространство памяти многопроцессорной системы и пространство внешней памяти.

Внешняя память разделена на четыре равных банка (рис. 6.14). Для каждого из них устанавливается собственное время доступа, что позволяет отображать адреса портов медленных периферийных устройств в адресное пространство того банка, для которого установлены подходящие периферийному устройству способ синхронизации и время доступа.

Поля адресных кодов, генерируемых ADSP 2106x при обращении по шинам DM и PM, показаны на рис. 6.15. Адреса шины DM генерирует DAG1, а адреса шины PM – программный автоматом (для команд) или DAG2 (для данных).

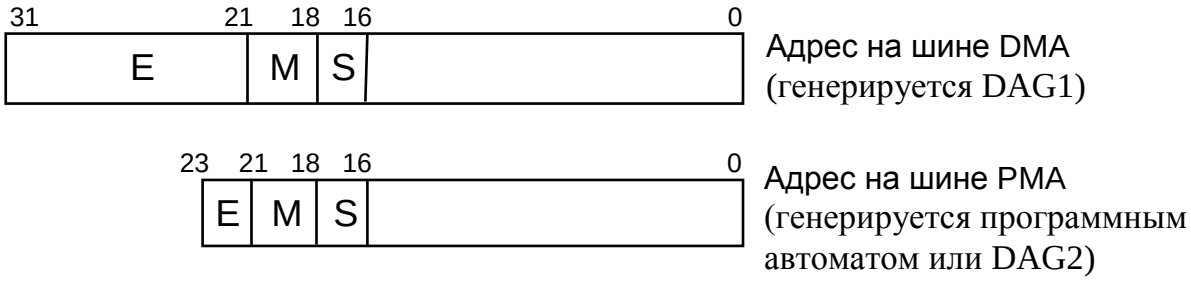


Рис. 6.15. Формат адресных кодов, передаваемых по шинам DMA и PMA.

Таблица 6.3

Поле	Значение	Пояснение
E	не ноль	Адрес внешней памяти.
	все нули	Адрес внутренней памяти процессора или внутренней памяти другого ADSP-2106x *(*M *и* S *активизированы*).
M	000	Адрес собственной внутренней памяти процессора.
	не ноль	M = ID, адрес внутренней памяти другого процессора, где ID – идентификатор МП ADSP 2106x.
	111	Широковещательная запись во внутреннюю память всех ADSP 2106x.
S	00	Адрес регистров устройства ввода-вывода (IOP).
	01	Адрес в пространстве адресов нормальных слов.
	1x	Арес в пространстве адресов коротких слов (x – *старшие биты адреса короткого слова*).

Устройство ввода-вывода проверяет адреса всех обращений к памяти и направляет их в соответствующее пространство памяти. Поля адреса E (внешний), M (многопроцессорный) и S (внутренний) декодируются устройством ввода/вывода. Если поле E нулевое, то поля M и S станут активными и будут

декодироваться. Более подробное объяснение назначения адресных полей E, M и S дано в таблице 6.3.

Устройству ввода-вывода принадлежат 256 отображенных в адресном пространстве памяти регистров (регистры УВВ на рис. 6.14). Регистры УВВ служат для управления и конфигурации системы ADSP 2106x, а также для управления выполнением операций ввода-вывода. Адресное пространство между адресами регистров УВВ и адресами нормальных слов с 0x0000 0100 по 0x0001 FFFF является не используемой памятью, и в него нельзя производить запись.

Когда ADSP 2106x генерирует адрес для одного из четырёх банков, то активизируются соответствующие линии выбора памяти $\overline{MS}_{3-0}$. Сигналы $\overline{MS}_{3-0}$ могут быть использованы для выбора микросхем памяти или внешних устройств. Тем самым устраняется надобность во внешней декодирующей логике. $\overline{MS}_0$ в комбинации с сигналом PAGE обеспечивает выбор банка динамической памяти DRAM.

Размер банка памяти должен быть равен степени двойки и может изменяться от 8 килослов до 256 мегаслов. Размеры банков памяти задаются путем установки соответствующих бит в одном из регистров процессора, отвечающих за конфигурацию памяти.

Небанковая память

Область памяти выше банков 0-3 называется небанковым пространством внешней памяти. Для доступа в это адресное пространство линии выбора памяти не используются. Временные параметры доступа к небанковому пространству памяти при необходимости также можно изменять.

Начальная загрузка памяти

Когда ADSP 2106x сконфигурирован для начальной загрузки из EPROM, то начальная загрузка производится из отдельного внешнего пространства памяти в сопровождении сигнала $\overline{BMS}$ (*Boot Memory Select*). В многопроцессорной системе сигналом $\overline{BMS}$ управляет только ведущий ADSP 2106x.

Многопроцессорная система с общей системной магистралью

Структура многопроцессорной системы, в которой связь между процессорами осуществляется по общей параллельной шине, представлена на рис. 6.13. В состав такой системы могут входить до шести процессоров ADSP2106x и хост-процессор.

Управление шиной в многопроцессорной системе

В многопроцессорной системе внешняя шина может использоваться совместно несколькими процессорами без дополнительного устройства управления. Этому способствует заложенная в каждый процессор логика арбитража

шины, которая позволяет включать в состав системы до шести ADSP 2106x и хост-процессор.

Управление шиной выполняется с использованием сигналов $\overline{BR}_{1-6}$, $\overline{HBR}$ и $\overline{HBG}$. Сигналы $\overline{BR}_{1-6}$ управляют переключением между несколькими ADSP 2106x, а $\overline{HBR}$ и $\overline{HBG}$ управляют передачей шины от ведущего процессора к хост-процессору и обратно.

В таблице 6.4 определены сигналы микросхемы ADSP 2106x, предназначенные для использования в многопроцессорных системах.

Для арбитража шины используется две схемы приоритетов: схема с фиксированными и с циклически изменяющимися приоритетами. Высокий уровень сигнала на входе RPBA (*Rotating Priority Bus Arbitration Select*) устанавливает схему с циклически изменяющимися приоритетами.

Когда процессор получает управление шиной, он может обращаться не только к внешней памяти, но и к внутренней памяти и регистрам УВВ всех других процессоров. Процессор может прямо передавать данные в другой процессор или инициализировать канал ПДП для передачи данных. Все процессоры отображены в общей карте памяти системы. Для идентификации адресного пространства любого процессора внутри объединенной карты памяти каждому процессору присваивается уникальный идентификатор ID.

Таблица 6.4

Вывод	**Тип**	**Функция**
$\overline{BR}_{1-6}$	I/O/S	**Запрос шины в многопроцессорной системе** (*Bus Requests*). Используются в многопроцессорных системах ADSP 2106x для арбитража шины. Каждый процессор управляет сигналом только на своей линии $\overline{BR}_X$, определяемой значением двоичного кода его идентификатора на входах ID_{2-0}, и контролирует все другие сигналы.
$\overline{CPA}$	I/O	**Приоритетный доступ ядра** (*Core Priority Access*). Сигнал позволяет ядру ведомого ADSP 2106x прервать фоновый ПДП и получить доступ к внешней шине. Выход $\overline{CPA}$ является выходом с открытым стоком. Выводы всех сигналов $\overline{CPA}$ в системе соединены вместе.
ID_{2-0}	I	**Многопроцессорный идентификатор**. Определяет, какая из запросных линий $\overline{BR}_{1-6}$ используется процессором с идентификационным номером ID. ID = 001 соответствует $\overline{BR}_1$, ID = 010 соответствует $\overline{BR}_2$ и т. д. ID = 000 используется в системах с одним процессором. Эти линии определяют конфигурацию системы, и уровни напряжений на них устанавливаются аппаратно.

PRBA	I/S	**Выбор вращающихся приоритетов для арбитража шины** (*Rotating Priority Bus Arbitration Select*). Когда RPBA установлен, то это означает, что для арбитража шины многопроцессорной системы выбрана схема вращающихся приоритетов. Когда RPBA сброшен, выбрана схема фиксированных приоритетов. Этот сигнал определяет конфигурацию системы и должен устанавливаться одинаковым во всех ADSP 2106x.

Внутренняя память МП ADSP 2106x организована таким образом, чтобы повысить эффективность операций ввода-вывода в многопроцессорных системах. Двухпортовая RAM, расположенная на кристалле, позволяет осуществлять высокоскоростную передачу данных между процессорами, а также параллельно выполнять двойной доступ к памяти со стороны ядра процессора.

Блокировка шины и семафоры

Для совместного использования ресурсов многопроцессорной системы, таких как память или ввод-вывод, могут использоваться семафоры. Семафор – это флаг, который может считываться и записываться любым из процессоров, совместно использующим данный ресурс, и его значение определяет возможность доступа к этому ресурсу. Семафоры полезны также для синхронизации задач, выполняемых различными процессорами в многопроцессорной системе.

Ключевым требованием при работе с семафорами в многопроцессорной (многозадачной) среде является считывание и изменение семафора в одной неделимой операции. Это можно сделать, применяя блокировку шины. Из-за того, что внешняя память, а также внутренняя память и регистры УВВ каждого процессора доступны любому другому, семафоры могут размещаться почти везде.

Операция *чтение-модификация-запись* семафора выполняется корректно, если следовать двум простым правилам:

- процессор не должен записывать семафор, если он не является ведущим. Это особенно важно, если семафор размещён в собственной внутренней памяти или в регистрах УВВ;
- при операции чтение-модификация-запись семафора процессор должен управлять шиной на протяжении всей операции.

Этих правил придерживаются и тогда, когда процессор использует блокировку шины, предотвращая одновременный доступ к ней (и к семафорам) со стороны других процессоров.

Блокировка шины может использоваться в комбинации с широковещательной записью для реализации *взаимных семафоров* в многопроцессорной системе. Взаимный семафор должен размещаться по одним и тем же адресам во внутренней памяти (или регистре УВВ) каждого процессора. Для проверки семафора каждый процессор может считывать его из собственной внутренней памяти. Для изменения семафора процессор сначала запрашивает блокировку шины, а затем выполняет широковещательную запись по адресу семафора в

каждый процессор, включая себя. Внешняя шина, в этом случае, используется только для модификации взаимных семафоров, но не для их считывания. Это значительно уменьшает трафик шины. Перед тем, как изменить значение семафора, необходимо проверить его состояние, чтобы убедиться, что над семафором не производились действия со стороны других процессоров.

Пример: взаимные семафоры при совместном использовании канала ПДП

Несколько процессоров могут совместно использовать один канал прямого доступа к памяти, используя *регистр управления каналом* в качестве взаимного семафора. Регистр управления каналом ПДП – это отображенный в памяти каждого процессора регистр устройства ввода-вывода. В нем содержится адрес *дескриптора канала* – начальный адрес области памяти, где хранится информация для инициализации канала ПДП: адрес предоставляемой для прямого доступа области памяти, количество и размер передаваемых слов, направление передачи, источник и приемник передаваемых данных. При обращении к регистру управления информация, хранимая в дескрипторе, заносится в регистры канала, после чего канал автоматически начинает сою работу. При этом не нужны никакие действия со стороны процессора. Эти действия должны быть предприняты заранее и направлены на инициализацию дескриптора канала ПДП.

Если содержимое регистра управления равно нулю, то регистр управления как семафор не используется никаким процессором. Если канал ПДП занят, то содержимое регистра управления не равно нулю.

Перед получением доступа к каналу ПДП, необходимо проверить принадлежащий ему семафор. Если канал свободен (значение семафора равно нулю), то процессор может запросить блокировку шины, а затем выполнить операцию чтение-модификация-запись для установки нового состояния семафора. Новое значение семафора в режиме широковещательной записи передается всем совместно использующим этот канал ПДП процессорам. Перед выполнением операции чтение-модификация-запись процессор должен повторно проверить семафор, чтобы убедиться, что канал ПДП все еще свободен. Получив в распоряжение семафор, процессор может выполнять запланированную передачу по ПДП. После завершения передачи семафор необходимо освободить, обнулив его значение и в режиме широковещательной записи сообщить об этом другим процессорам.

Межпроцессорные сообщения и векторные прерывания

Ведущий процессор может связываться с ведомыми путем записи сообщений в их регистры УВВ. Для передачи сообщений между процессорами могут использоваться предназначенные для этого универсальные регистры (восемь регистров сообщений) устройства ввода-вывода. Эти регистры могут быть полезны также при реализации семафоров и для совместного использования ресурсов разными процессорами.

К числу универсальных регистров относится также регистр многопроцессорного векторного прерывания VIRPT (*Vector Interrupt*).

Векторные прерывания используются для межпроцессорного взаимодействия в многопроцессорных системах. Внешний (другой или хост-) процессор вызывает векторное прерывание посредством записи стартового адреса процедуры обработки в регистр VIRPT.

Все названные универсальные регистры могут использоваться для передачи сообщения следующими способами:

- *Передача сообщения*. Ведущий процессор может связываться с ведомым путем записи и/или считывания любого из восьми регистров сообщений ведомого.
- *Векторные прерывания*. Ведущий процессор может запрашивать векторное прерывание у ведомого. Производится это путём записи адреса программы обработки прерывания в регистр VIRPT ведомого. Это вызывает прерывание с высшим приоритетом в ведомом процессоре, который переходит на определённую в векторе запроса программу обработки прерывания.

Таблица прерываний

Таблица векторов прерываний может размещаться как во внутренней, так и во внешней памяти. Это зависит от того, используется или не используется режим начальной загрузки, а также от установки соответствующего бита в имеющемся у процессора регистре конфигурации системы. Адреса в таблице представляют собой смещения относительно базового адреса. Для таблицы векторов прерываний во внутренней памяти базовый адрес – 0x0002 0000 (начало блока 0); для таблицы векторов прерываний во внешней памяти базовый адрес – 0x0040 0000. Каждому вектору выделяются четыре 48-разрядные ячейки памяти.

Кластерная многопроцессорная система

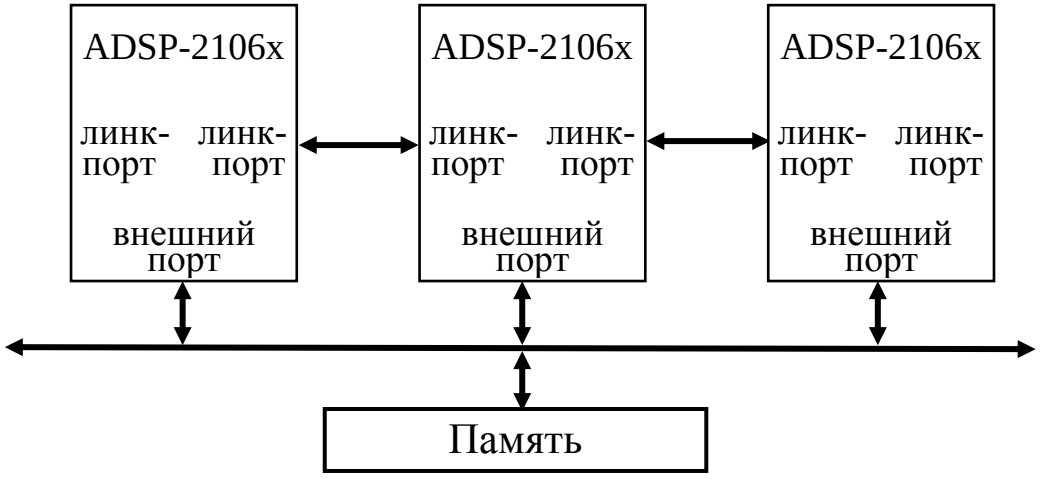


Рис. 6.16. Кластерная многопроцессорная система.

В общем случае многопроцессорная система может быть организована, исходя из совместного использования системной магистрали и линк-портов. Такую систему можно характеризовать как кластерную (имеющую не более шести процессоров ADSP 2106x и хост-процессор) многопроцессорную систему, которая обладает боле развитой функциональностью, но требует значительно более трудоемкого программирования. Конфигурация кластерной многопроцессорной системы представлена на рис. 6.16.

6.2.4. Направление развития МП с архитектурой SHARC

Особенностью микропроцессоров с архитектурой SHARC является наличие в них средств межпроцессорного взаимодействия в виде линк-портов. Эта особенность сохраняется и следующих версиях этих МП. Развитие идет в сторону увеличения вычислительных возможностей, в частности, путем создания многоядерных процессоров и увеличения разрядности внутренних шин. Примером может послужить процессор ADSP-21160 (рис. 6.17), структурное построение которого похоже на структуру ядер процессоров семейства ADSP-2106x, за исключением ширины шин и второго вычислительного блока с собственным умножителем, АЛУ, устройством сдвига и регистровым файлом. Такая структура свойственна процессорам с SIMD (*Single Instruction Multiple Data*) архитектурой.

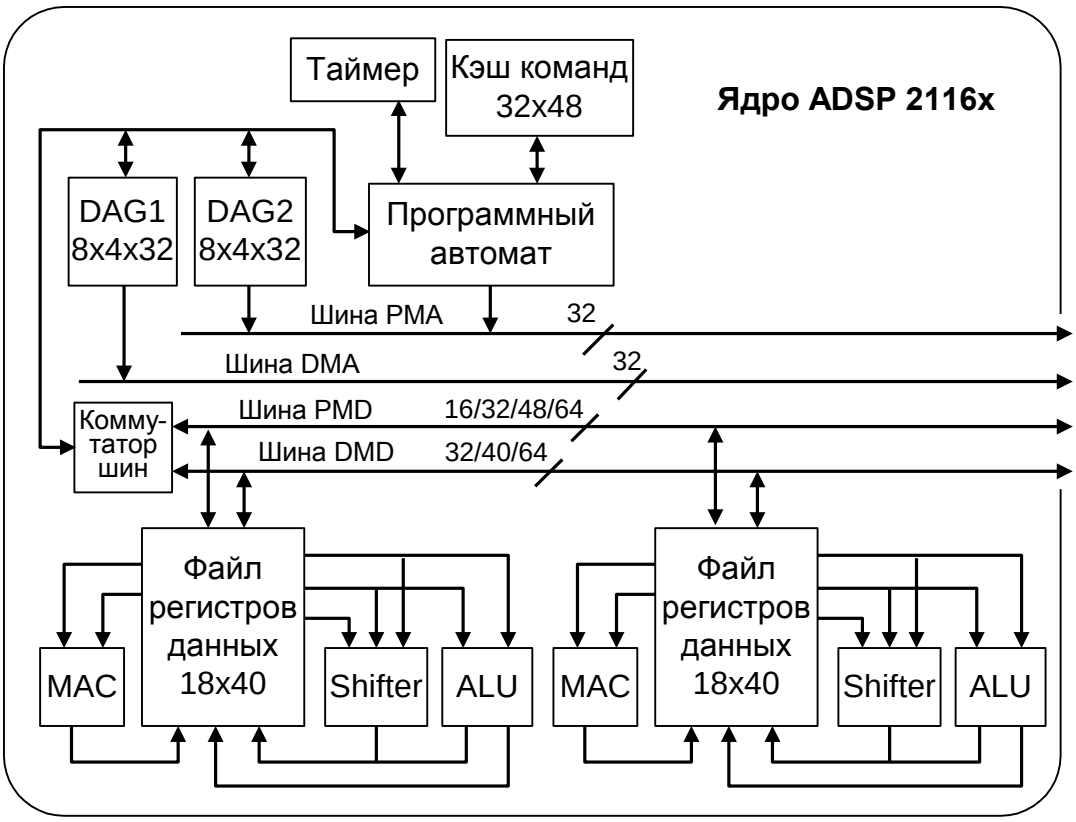


Рис. 6.17. Ядро микропроцессоров семейства ADSP 21116x

Процессоры ADSP 2116x имеют два процессорных элемента, которые могут одновременно выполнять команду каждый над своими данными, т.е. обрабатывать два потока данных параллельно при незначительном изменении программной модели.

Пути дальнейшего повышения производительности компания Analog Devices связывает со статическим выявлением параллелизма уровня команд. Примером может послужить процессор ADSP-TS001 – TigerSHARC. В нем возможности цифровой обработки сигналов основаны на сочетании особенностей RISC и VLIW (*Very Long Instruction Word*) архитектур.

Черты RISC отражены в фиксированной структуре команд и их конвейерном выполнении с предсказанием переходов, в наличии большого регистрового файла и, как следствие, в эффективной загрузке вычислительных блоков.

VLIW подход заключается в планировании загрузки функциональных блоков. Чтобы обеспечить поступление команд во все функциональные блоки, необходимо эффективно использовать доступную ширину слова команды. Иначе говоря, заложенные в многофункциональные команды управляющие воздействия должны подаваться на вычислительные блоки одновременно и связанный с этим параллелизм выполнения операций должен планироваться заранее, до непосредственного выполнения программы. Выявление параллелизма уровня команд и возможность независимого задания в программе порядка загрузки функциональных блоков происходит на этапе компиляции.

С точки зрения аппаратных решений VLIW подход требует увеличения разрядности шин и организации памяти с целью поддержки обращений к памяти программы и операций одновременной загрузки и сохранения данных. Аппаратные решения процессоров ADSP-TS001 выглядит как развитие представленной на рис. 6.17 структуры ADSP 2116x в направлении расширения функциональных возможностей составляющих ее блоков.

6.2.5. Отражение гарвардской архитектуры в микроконтроллерах и процессорах общего назначения

В 1991 г. компании Motorola, IBM и Apple Computers объявили об организации консорциума для совместной разработки и внедрения RISC микропроцессоров новой архитектуры. Для решения этой задачи были сделаны крупные инвестиции (около 1 млрд долл.) и построен новый центр проектирования в г. Остин (Техас), открытый в мае 1992 г. Штат центра составили 300 ведущих специалистов из компаний IBM и Motorola, в результате работы которых уже в октябре 1992 г. были получены первые образцы 32-разрядных RISC микропроцессоров семейства PowerPC 60x, а с апреля 1993 г. начался их серийный выпуск. В октябре 1994 г. было положено начало новой ветви 64-разрядных процессоров.

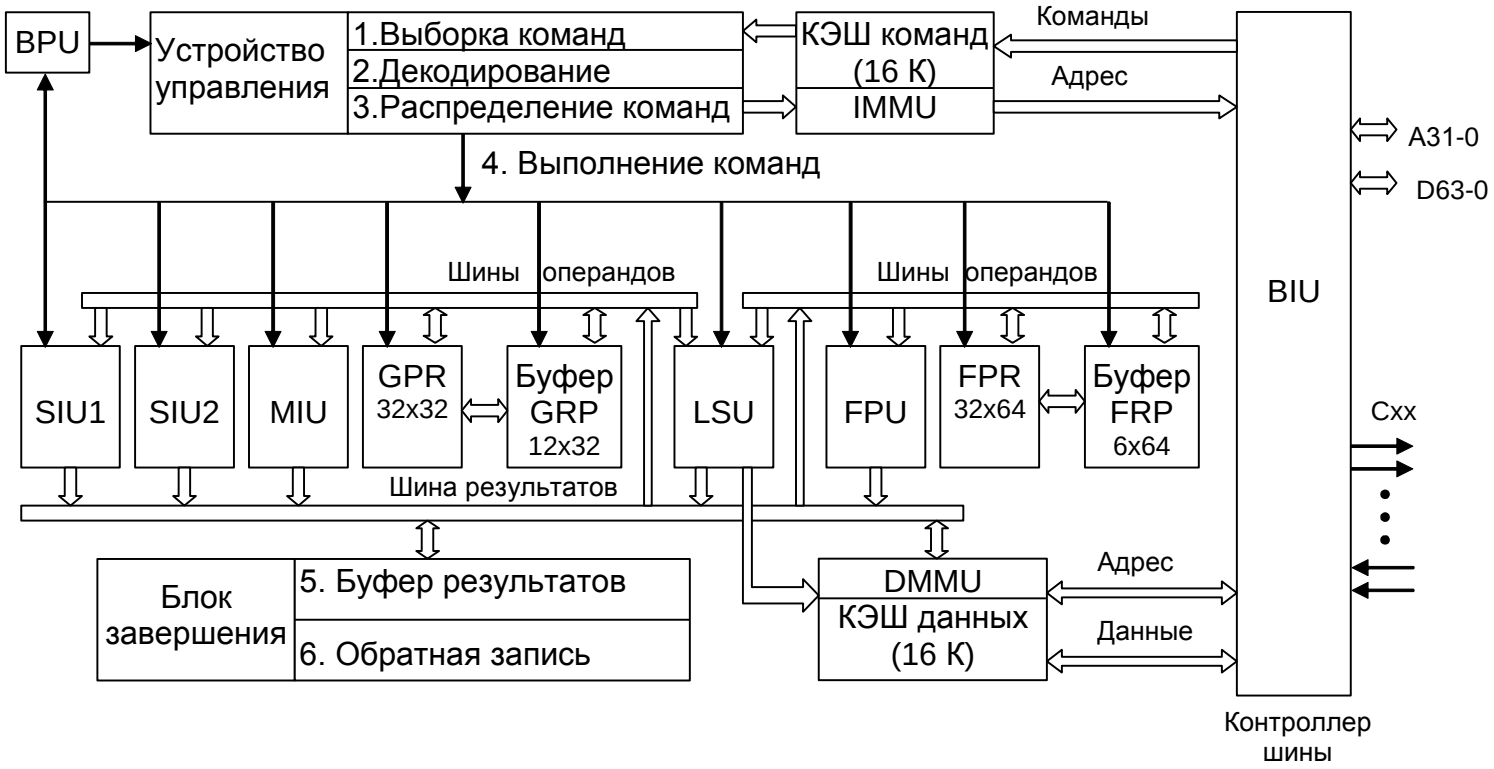


Рис. 6.18. Структура микропроцессора MCP604

Модели микропроцессоров семейства PowerPC 60x, выпускаемые фирмой Motorola, входят в семейство MPC60x. Все микропроцессоры семейства содержат от 4 до 6 параллельно работающих исполнительных устройств, обеспечивающих одновременное выполнение нескольких команд. Эти микропроцессоры послужили также основой для создания новых семейств микроконтроллеров MPC5xxx и коммуникационных контроллеров MCP860.

Рассмотрим структуру (рис. 6.18) и функционирование наиболее производительной 32-разрядной модели MPC604.

В микропроцессорах PowerPC, как и в семействе MCP6xx, реализован принцип выделения отдельных ресурсов для решения задач пользователя и супервизора (операционной системы). В соответствии с этим принципом архитектура PowerPC содержит регистры, входящие в модели пользователя или супервизора, а также ряд привилегированных команд, выполняемых только в режиме супервизора.

В регистровую модель пользователя, которая является общей для всего семейства PowerPC, входят:

- тридцать два 32-разрядных регистра общего назначения GRP31–0 для хранения целочисленных операндов;
- тридцать два 64-разрядных регистра FPR31–0 для хранения операндов с плавающей точкой;
- 32-разрядные регистр условий (признаков) CR и регистр FPSR состояния при обработке чисел с плавающей точкой;
- три 32-разрядных регистра, используемые при обработке исключительных ситуаций (событий внутри микропроцессора и прерываний).

Эта регистровая модель не содержит программного счетчика PC и указателя стека SP. Отсутствие PC связано с тем, что микропроцессоры PowerPC не выполняют команд записи или чтения программного счетчика (действия со счетчиком видны только на аппаратном уровне). Эти микропроцессоры не выполняют также автоматических операций со стеком. В случае необходимости стек реализуется программно, поэтому в регистровой модели отсутствует SP.

Микропроцессор MPC604 (рис. 6.18) содержит шесть параллельно работающих исполнительных устройств: блок обработки ветвлений BPU, два устройства для выполнения простых (одноцикловых) целочисленных операций SIU1 и SIU2, одно устройство для выполнения сложных (многоцикловых) целочисленных операций MIU, устройство обработки чисел с плавающей точкой FPU и блок выборки операндов из памяти LSU. При этом обеспечивается одновременное выполнение четырех команд. Все операции обработки данных выполняются с регистровой адресацией. Выборка данных из памяти производится только командами пересылки, которые выполняются блоком LSU и осуществляют загрузку данных в регистры GRP и FRP или запись содержимого этих регистров в память.

Выборка команд из памяти и обращение к данным осуществляются с использованием отдельных шин адреса и данных и отдельных устройств управления памятью команд IMMU (*Instruction Memory Management Unit*) и памятью данных DMMU (*Data Memory Management Unit*). Такое решение подобно тому, которое

используется в процессорах с гарвардской архитектурой, но с той разницей, что непосредственное обращение к основной памяти по шинам адреса и данных отсутствует, поскольку внутренняя память процессора MPC604 представлена кэш-памятью команд и кэш-памятью данных.

При параллельной работе исполнительных устройств возможно их одновременное обращение к одним регистрам. Чтобы избежать ошибок, возникающих при этом, в случае записи в регистр нового содержимого до того, как другим устройством будет считано предыдущее, введены буферные регистры – 12 для GRP и 8 для FRP. Эти регистры служат для промежуточного хранения операндов, дублируя регистры GRP и FRP. После завершения этих операций производится перезапись полученных результатов в GRP и FRP (обратная запись).

Конвейер выполнения команд имеет шесть основных каскадов: выборка команды (1) и ее дешифрация (2), распределение команд по исполнительным устройствам (3), выполнение команд (4), запись результатов в буфер (5) и обратная запись результатов в регистры GRP или FRP. Устройство управления одновременно выбирает из кэша и декодирует четыре команды, которые распределяются в соответствующие исполнительные устройства. Большинство команд выполняется за один такт. Эти команды реализуются с помощью SIU1 и SIU2, которые осуществляют сложение/вычитание, сравнение, сдвиги, логические операции. Выполнение ряда команд требует нескольких тактов. Например, целочисленное умножение выполняется за 3-4 такта, деление за 20 тактов. Эти операции производятся MIU. Большинство операций с плавающей точкой (кроме деления) выполняется за три такта, поэтому во внутренней структуре FPU реализовано три исполнительных каскада. Блок завершения обеспечивает получение правильной последовательности результатов выполняемых операций. Он содержит буферную память типа очереди, в которую по мере поступления записываются результаты, получаемые от исполнительных устройств. Обратная запись результатов из буфера в регистры GPP и FRP производится в соответствии с порядком поступления команд, выбираемых устройством управления, т.е. восстанавливается необходимая последовательность результатов.

Блок обработки ветвлений BPU предсказывает направление ветвления программы при поступлении соответствующих команд. В микропроцессорах семейства MPC6xx реализуются два механизма предсказания переходов: статическое предсказание, которое дается программистом и содержится в тексте выполняемой программы, и динамическое предсказание, которое выполняется BPU по результатам предыдущих ветвлений. Блок BPU для предсказания использует кэш адресов ветвлений BTAC и таблицу истории ветвлений BHT. Кэш BTAC хранит 64 адреса ветвлений, выполнявшихся предыдущими командами. Если поступившая команда ветвления содержит адрес, совпадающий с одним из имеющихся в BTAC, то процессор предсказывает ветвление по этому адресу и выбирает команды из соответствующей ветви в конвейер (ветвление выполняется). Если адрес ветвления не содержится в BTAC, то выбирается следующая команда программы (ветвление отсутствует).

При поступлении команд условных ветвлений BPU анализирует таблицу BHT, которая представляет собой кэш-память, где содержатся 512 адресов предыдущих условных ветвлений. Для каждого адреса в таблице имеются два бита, определяющие вероятность ветвления по данному адресу: 00 – практически не выполняется, 01 – выполняется редко, 10 – выполняется часто, 11 – выполняется очень часто. Значения этих битов меняются после каждой операции ветвления: если производится ветвление по данному адресу, то вероятность увеличивается на единицу (максимальное значение 11), если данный адрес не использован, то вероятность уменьшается на единицу (минимальное значение 00). Новые адреса вводятся в BHT вместо адресов с нулевой вероятностью. Ветвления по имеющемуся в таблице адресу предсказывается, если его вероятность больше 01. Правильность предсказания проверяется после определения соответствующих признаков в регистре условий CR. Если условия не выполняются (неправильное предсказание), то процессор освобождает конвейер от команд из неправильно выбранной ветви и загружает новые команды из другой ветви. Таким образом, при неправильных предсказаниях требуется перезагрузка конвейера.

Так как одновременно могут выполняться несколько команд, изменяющих значения признаков в регистре CR, то в BPU имеются восемь буферных регистров, дублирующих содержимое CR. Эти регистры используются различными исполнительными устройствами, а после завершения соответствующих операций их содержимое переносится в основной регистр условий CR (обратная запись), обеспечивая правильный порядок выполнения условных команд.

Микропроцессор MPC604 осуществляет раздельную выборку команд и данных с помощью двух устройств управления памятью IMMU, DMMU, которые могут выполнять сегментную, страничную и блочную адресацию. Работа IMMU, DMMU обеспечивается с помощью 8 пар регистров IBAT, DBAT, 16 сегментных регистров SR0-SR15 и регистра SDR1, обращение к которым осуществляется только в режиме супервизора.

Процессор может генерировать адреса в реальном режиме, когда сформированный адрес поступает на адресные выводы в качестве физического адреса ячейки памяти или порта ввода-вывода. В этом случае IMMU, DMMU отключены и действия по трансляции адреса, подобные представленным в разделе 6.7, не производятся. Включение IMMU, DMMU производится путем установки соответствующих битов в одном из регистров, отвечающих за конфигурацию управления процессором. В этом случае сформированный адрес команды или данных воспринимается как логический, который с помощью IMMU или DMMU транслируется в физический.

Рассмотрим реализуемые IMMU, DMMU варианты адресной трансляции.

1. *Блочная трансляция* обеспечивает обращение к блокам внешней памяти заданного объема – от 128 Кбайт до 256 Мбайт. Возможна организация четырех блоков для хранения команд и четырех – для хранения данных. Параметры каждого блока задаются дескриптором, который содержится в соответствующей паре регистров IBAT для команд и паре регистров DBAT.

Если устройство IMMU или DMMU включено, то старшие разряды сформированного логического адреса команды или данных, определяющие базовый адрес блока заданного размера, сравниваются со значениями индекса в дескрипторах блоков с целью идентификации блока. Младшие разряды, указывающие относительное положение байта (смещение), транслируются в младшие разряды физического адреса.

Помимо адресной информации в дескрипторах содержатся биты, обеспечивающие защиту памяти и определяющие работу кэша.

2. *Сегментная трансляция* обеспечивает обращение к сегментам памяти емкостью 256 Мбайт, которые разбиваются на страницы размером по 4 Кбайт, размещаемые в ОЗУ, или представляют массивы данных, расположенные во внешних устройствах. Для обращения к сегментам используются 16 сегментных регистров SR0-SR15, которые содержат дескрипторы сегментов. четыре старшие разряда логического адреса задают номер регистра SRi, из которого выбирается дескриптор соответствующего сегмента. В зависимости от установленного типа дескриптора реализуется страничная трансляция памяти или адресация портов ввода-вывода.

При страничной трансляции устройство управления памятью IMMU или DMMU формирует 52-разрядный виртуальный адрес VA0-51[8], составленный из виртуального адреса сегмента VSID, индекса страницы (разряды LA4-19 логического адреса) и относительного адреса байта в странице (разряды LA20-31 логического адреса). Старшие разряды VA0-39 виртуального адреса, представляющие виртуальный номер страницы VPN, поступают в блок страничной трансляции (БСТ), определяющий физический номер страницы RPN, который служит разрядами PA0-19 физического адреса. Младшие разряды физического адреса совпадают с адресом байта на странице PA20-31 = LA20-31. Для определения RPN используются дескрипторы страниц PD, которые хранятся в специальной таблице TPD, хранящейся в ОЗУ. Базовый адрес этой таблицы задается содержимым регистра SDR1.

Дескрипторы страниц, к которым выполнялось обращение, хранятся в специальной кэш-памяти TLB, входящей в состав IMMU и DMMU. Каждый TLB содержит 128 дескрипторов PD. Работа TLB организована по такому же принципу, как и работа кэш-памяти данных и кэш-памяти команд (см. разделы 6.6 и 6.7).

При организации многопроцессорных систем обычно делается так, чтобы кэш-память данных каждого процессора была доступной для других процессоров, но при этом необходимы аппаратные решения по синхронизации кэшей разных процессоров для того чтобы исключить возможность получения из кэш-памяти недействительных данных как со стороны своего, так и со стороны других процессоров. К своему собственному кэшу процессор обращается по виртуальному адресу. Обращение со стороны других процессоров происходит со стороны системной магистрали и только по физическому адресу. Поэтому в многопроцессорных системах кэш-память данных наряду с памятью виртуальных тегов должна

[8] Здесь 0 относится к старшему разряду, а 51 – к младшему.

содержать память физических тегов, а внутренняя связанная с DMMU шина адреса является двунаправленной.

Контроллер шины BIU обеспечивает интерфейс микропроцессора с внешними устройствами. Системная шина передает 32-разрядный адрес и 64-разрядные данные. Увеличенная разрядность шины данных позволяет быстрее производить загрузку строк кэш-памяти.

Рассмотренные применительно к архитектуре MSP6xx структурные и архитектурные решения в той или иной степени свойственны широкому классу как специализированных микропроцессоров, так и процессоров общего назначения. Их можно найти в процессорах семейства P6, в которых отражена общая линия микропроцессоров Intel 80/86, в 32-разрядных RISC процессорах семейств ARM9, ARM10 компании ARM Limited, используемых в качестве ядра для различных встроенных приложений, в процессорах серии SPARC фирмы Sun Microsystems. Перечень можно продолжать. Но и в том, что обозначено, можно найти характерные особенности развития:

- внутренняя гарвардская архитектура с разделением потоков команд и данных;
- одновременное выполнение нескольких команд в параллельно работающих исполнительных устройствах;
- конвейер операций с предсказанием ветвлений.

На базе RISC процессоров с архитектурой MSP6xx разработаны микроконтроллеры и коммуникационные контроллеры для использования в системах управления и связи. Их структурное построение мало чем отличается от рассмотренных в разделах 9.3 и 9.4 структур микроконтроллеров фирм Texas Instruments и ARM Limited. Различия имеются в составе периферийных устройств, а также, что более существенно, в архитектуре центральных процессорных элементов ЦПЭ, которые могут выглядеть как простые с архитектурой фон-Неймана процессорные элементы, так и как сложные функционально развитыми аппаратные комплексы.

В сфере встраиваемых приложений активно развивается производство высокопроизводительных 8-разрядных RISC-микроконтроллеров. Одним из признанных лидеров в этом направлении является фирма Atmel Corp. Широкое распространение нашли выпускаемые этой фирмой микроконтроллеры AVR.

Остановимся лишь на семействе AVR Classic, включающем в общей сложности 17 моделей микроконтроллеров (МК). Микроконтроллеры этого семейства построены по гарвардской архитектуре, в которой память программ выполнена как ПЗУ, составленное из FLASH- и EEPROM-памяти, а память данных – как статическое ОЗУ (рис. 6.19).

Микроконтроллер всегда работает под управлением программы, загруженной в ПЗУ. Для загрузки ПЗУ используется последовательный синхронный интерфейс SPI. Конфигурация микросистемы осуществляется под управлением программы путем записи соответствующей информации в регистры управления.

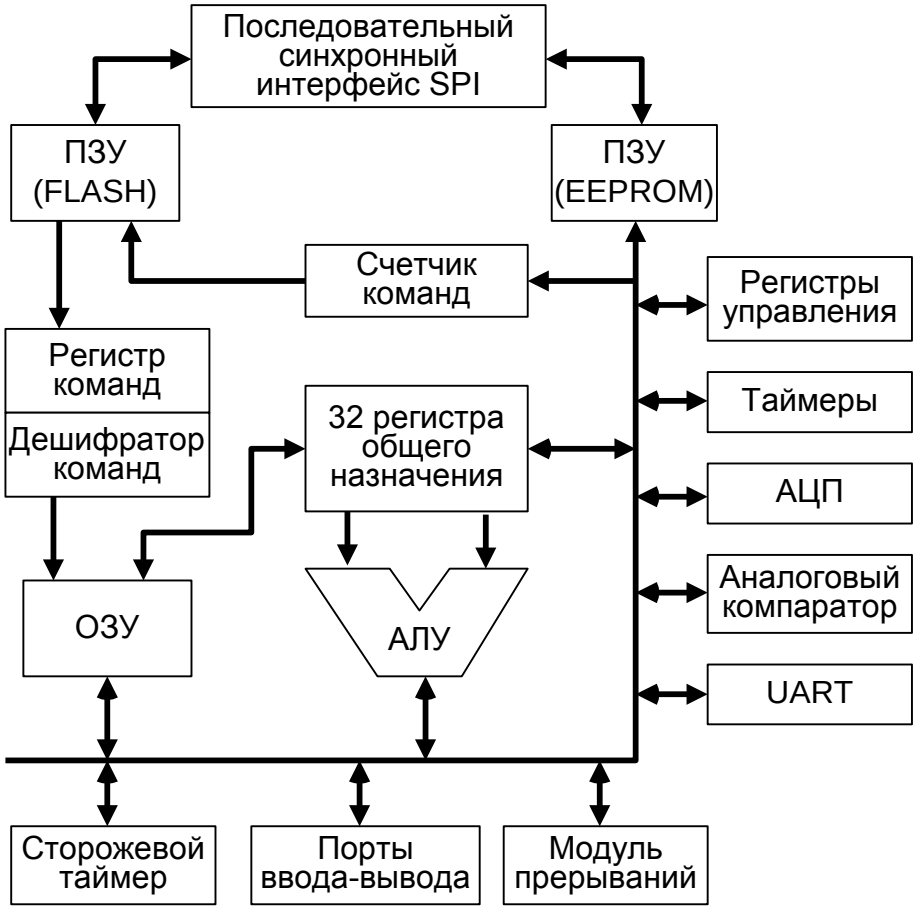


Рис. 6.19. Блок-схема микроконтроллера AVR

Набор периферийных устройств, интегрированных в микросхему МК, зависит от требований по компоновке. На рис. 6.19 представлен вариант такой компоновки.

Приложение 1

Последовательность действий ОС µC/OS-II при работе с функциями OSMBoxPend() и OSMboxPost()

```
*********************************************************************************
*                                TASK CONTROL BLOCK (TCB)
*********************************************************************************
typedef struct os_tcb {
    OS_STK        *OSTCBStkPtr;    /* Pointer to current top of stack */

#if OS_TASK_CREATE_EXT_EN
    void          *OSTCBExtPtr;      /* Pointer to user definable data for TCB
                                     /* extension */
    OS_STK        *OSTCBStkBottom;   /* Pointer to bottom of stack    */
    INT32U         OSTCBStkSize;     /* Size of task stack (in bytes) */
    INT16U         OSTCBOpt;         /* Task options as passed by */
                                     /* OSTaskCreateExt()*/
    INT16U         OSTCBId;          /* Task ID (0..65535) */
#endif

    struct os_tcb *OSTCBNext;     /* Pointer to next TCB in the TCB list */
    struct os_tcb *OSTCBPrev;     /* Pointer to previous TCB in the TCB list */

#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
    OS_EVENT *OSTCBEventPtr;     /* Pointer to event control block */
#endif

#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    void *OSTCBMsg;     /* Message received from OSMboxPost() or OSQPost() */
#endif

    INT16U OSTCBDly;    /* Nbr ticks to delay task or, timeout waiting for event*/
    INT8U  OSTCBStat;   /* Task status */
    INT8U  OSTCBPrio;   /* Task priority (0 == highest, 63 == lowest) */

    INT8U  OSTCBX;  /* Bit position in group corresponding to task priority (0..7)
*/
    INT8U  OSTCBY;  /* Index into ready table corresponding to task priority */
    INT8U  OSTCBBitX;   /* Bit mask to access bit position in ready table */
    INT8U  OSTCBBitY;   /* Bit mask to access bit position in ready group */

#if OS_TASK_DEL_EN
    BOOLEAN OSTCBDelReq;  /* Indicates whether a task needs to delete itself */
#endif
} OS_TCB;
```

```
/*      PEND ON MAILBOX FOR A MESSAGE
 Description: This function waits for a message to be sent to a mailbox
 Arguments:
    pevent is a pointer to the event control block associated with the desired
     mailbox
    timeout is an optional timeout period (in clock ticks).  If non-zero, your
     task will wait for a message to arrive at the mailbox up to the amount of
     time specified by this argument.  If you specify 0, however, your task will
     wait forever at the specified mailbox or, until a message arrives.
    err is a pointer to where an error message will be deposited.  Possible error
     messages are:
      OS_NO_ERR         The call was successful and your task received a message.
      OS_TIMEOUT        A message was not received within the specified timeout
      OS_ERR_EVENT_TYPE   Invalid event type
      OS_ERR_PEND_ISR     If you called this function from an ISR and the
                          result would lead to a suspension.
 Returns:
    != (void *)0  is a pointer to the message received
    == (void *)0  if no message was received or you didn't pass the proper
       pointer to the event control block. */
void *OSMboxPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{   void  *msg;
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {  //Validate event block type
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
    msg = pevent->OSEventPtr;
    if (msg != (void *)0) {                 //See if there is already a message
        pevent->OSEventPtr = (void *)0;     //Clear the mailbox
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
    } else if (OSIntNesting > 0) {          //See if called from ISR
        OS_EXIT_CRITICAL();                 //can't PEND from an ISR
        *err = OS_ERR_PEND_ISR;
    } else {
        OSTCBCur->OSTCBStat |= OS_STAT_MBOX;  //Message not available, task will
                                              //pend
        OSTCBCur->OSTCBDly   = timeout;       //Load timeout in TCB
        OSEventTaskWait(pevent);              //Suspend task until event or
                                              //timeout occurs
        OS_EXIT_CRITICAL();
        OSSched();                 //Find next highest priority task ready to run
        OS_ENTER_CRITICAL();
        if ((msg = OSTCBCur->OSTCBMsg) != (void *)0) { //See if we were given the
                                                       //message
            OSTCBCur->OSTCBMsg = (void *)0;          //Yes, clear message received
            OSTCBCur->OSTCBStat = OS_STAT_RDY;
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0; //No longer waiting for event
            OS_EXIT_CRITICAL();
            *err = OS_NO_ERR;
        } else if (OSTCBCur->OSTCBStat & OS_STAT_MBOX) {  //If status is not
                                                          //OS_STAT_RDY, timed out
            OSEventTO(pevent);                            //Make task ready
            OS_EXIT_CRITICAL();
            msg  = (void *)0;                 //Set message contents to NULL
            *err = OS_TIMEOUT;                //Indicate that a timeout occured
        } else {
            msg = pevent->OSEventPtr;         //Message received
            pevent->OSEventPtr = (void *)0;   //Clear the mailbox
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
            OS_EXIT_CRITICAL();
```

```
            *err = OS_NO_ERR;
        }
    }
    return (msg);                       //Return the message received (or NULL)
}

/*     MAKE TASK WAIT FOR EVENT TO OCCUR
  Description: This function is called by other uC/OS-II services to suspend a
               task because an event has not occurred.
  Arguments: pevent is a pointer to the event control block for which the task
             will be waiting for.
  Returns: none */
#if  (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
void  OSEventTaskWait (OS_EVENT *pevent)
{
    OSTCBCur->OSTCBEventPtr = pevent;    //Store pointer to event control block
                                         //in TCB
    if((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX)==0) {//Task no
                                                               //longer ready
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;  //Put task in
                                                                  //waiting list
    pevent->OSEventGrp                   |= OSTCBCur->OSTCBBitY;
}
#endif

/*             SCHEDULER
  Description: This function is called by other uC/OS-II services to determine
               whether a new, high priority task has been made ready to run.  This
               function is invoked by TASK level code and is not used to
               reschedule tasks from ISRs (see OSIntExit() for ISR rescheduling).
  Arguments: none
  Returns: none
  Notes: 1) This function is INTERNAL to uC/OS-II and your application should not
         call it.
         2) Rescheduling is prevented when the scheduler is locked (see
         OSSchedLock())   */
void OSSched (void)
{   INT8U y;
    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) {  //Task scheduling must be enabled
                                                //and not ISR level
        y = OSUnMapTbl[OSRdyGrp];               //Get pointer to highest priority
                                                //task ready to run
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur) {       //No context switch if current
                                                //task is highest ready
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;                       //Increment context switch counter
            OS_TASK_SW();                       // Perform a context switch
        }
    }
    OS_EXIT_CRITICAL();
}

;     void OS_TASK_SW(void)
;     ;     Perform a context switch.
;     On entry, OSTCBCur and OSPrioCur hold the current TCB and priority
;     and OSTCBHighRdy and OSPrioHighRdy contain the same for the task
;     to be switched to.
;
;     The following code assumes that the virtual memory is directly
```

```
;     mapped into  physical memory. If this is not true, the cache must
;     be flushed at context switch to avoid address aliasing.
      EXPORT      OS_TASK_SW
OS_TASK_SW
      STMFD sp!, {r0-r12, lr} ; save register file and ret address
      MRS   r4, CPSR
      STMFD sp!, {r4}         ; save current PSR
      MRS   r4, SPSR          ; YYY+
      STMFD sp!, {r4}         ; YYY+ save SPSR

      ; OSPrioCur = OSPrioHighRdy
      LDR   r4, addr_OSPrioCur
      LDR   r5, addr_OSPrioHighRdy
      LDRB  r6, [r5]
      STRB  r6, [r4]

      ; Get current task TCB address
      LDR   r4, addr_OSTCBCur
      LDR   r5, [r4]
      STR   sp, [r5]          ; store sp in preempted tasks's TCB

      ; Get highest priority task TCB address
      LDR   r6, addr_OSTCBHighRdy
      LDR   r6, [r6]
      LDR   sp, [r6]          ; get new task's stack pointer

      ; OSTCBCur = OSTCBHighRdy
      STR   r6, [r4]          ; set new current task TCB address

      LDMFD sp!, {r4}         ; YYY+
      LDMFD sp!, {r5}         ; YYY+
      MSR   CPSR_cxsf, r5           ; YYY+ Switch to task's context
      MSR   SPSR_cxsf, r4           ; YYY+ Restore task's SPSR
      LDMFD sp!, {r0-r12, lr} ; YYY+
 IF :DEF: THUMB_AWARE
      BX    lr
 ELSE
      MOV   pc,lr
 ENDIF

/*   MAKE TASK READY TO RUN BASED ON EVENT TIMEOUT
  Description: This function is called by other uC/OS-II services to make a task
               ready to run because a timeout occurred.
  Arguments: pevent is a pointer to the event control block which is readying a
             task.
  Returns: none     */
#if  (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
void  OSEventTO (OS_EVENT *pevent)
{
    if ((pevent->OSEventTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBStat     = OS_STAT_RDY;    //Set status to ready
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;  //No longer waiting for event
}
#endif

--------------------------------------------------------------------------------

/*     POST MESSAGE TO A MAILBOX
  Description: This function sends a message to a mailbox
  Arguments:
      pevent is a pointer to the event control block associated with the desired
```

```
       mailbox
       msg is a pointer to the message to send. You MUST NOT send a NULL pointer.
  Returns:
     OS_NO_ERR          The call was successful and the message was sent
     OS_MBOX_FULL       If the mailbox already contains a message.  You can can
                        only send one message at a time and thus, the message MUST
                        be consumed before you are allowed to send another one.
     OS_ERR_EVENT_TYPE  If you are attempting to post to a non mailbox.    */
INT8U OSMboxPost (OS_EVENT *pevent, void *msg)
{   OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {  //Validate event block type
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {                 //See if any task pending on mailbox
        OSEventTaskRdy(pevent, msg, OS_STAT_MBOX);  //Ready highest priority task
                                                    //waiting on event
        OS_EXIT_CRITICAL();
        OSSched();                          //Find highest priority task ready to run
        return (OS_NO_ERR);
    } else {
        if (pevent->OSEventPtr != (void *)0) {      //Make sure mailbox doesn't
                                                    //already have a msg
            OS_EXIT_CRITICAL();
            return (OS_MBOX_FULL);
        } else {
            pevent->OSEventPtr = msg;               //Place message in mailbox
            OS_EXIT_CRITICAL();
            return (OS_NO_ERR);
        }
    }
}

/*   MAKE TASK READY TO RUN BASED ON EVENT OCCURING
  Description: This function is called by other uC/OS-II services and is used to
  ready a task that was waiting for an event to occur.
  Arguments:
       pevent is a pointer to the event control block corresponding to the event.
       msg is a pointer to a message. This pointer is used by message oriented
       services such as MAILBOXEs and QUEUEs. The pointer is not used when called
       by other service functions.
       msk is a mask that is used to clear the status byte of the TCB. For
       example, OSSemPost() will pass OS_STAT_SEM, OSMboxPost() will pass
       OS_STAT_MBOX etc.
  Returns:    none
  Note: This function is INTERNAL to uC/OS-II and your application should not
callit.*/
#if  (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
void  OSEventTaskRdy (OS_EVENT *pevent, void *msg, INT8U msk)
{
    OS_TCB *ptcb;
    INT8U   x;
    INT8U   y;
    INT8U   bitx;
    INT8U   bity;
    INT8U   prio;

    y = OSUnMapTbl[pevent->OSEventGrp]; //Find highest prio. task waiting for
                                        //message
    bity = OSMapTbl[y];
    x    = OSUnMapTbl[pevent->OSEventTbl[y]];
    bitx = OSMapTbl[x];
    prio = (INT8U)((y << 3) + x);  //Find priority of task getting the msg
```

```
    if ((pevent->OSEventTbl[y] &= ~bitx) == 0) {//Remove this task from the
                                                //waiting list
       pevent->OSEventGrp &= ~bity;
    }
    ptcb           = OSTCBPrioTbl[prio];        //Point to this task's OS_TCB
    ptcb->OSTCBDly = 0;                  //Prevent OSTimeTick() from readying task
    ptcb->OSTCBEventPtr = (OS_EVENT *)0;        //Unlink ECB from this task
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    ptcb->OSTCBMsg = msg;              //Send message directly to waiting task
#else
    msg = msg;                         //Prevent compiler warning if not used
#endif
    ptcb->OSTCBStat &= ~msk;           //Clear bit associated with event type
    if (ptcb->OSTCBStat == OS_STAT_RDY) { //See if task is ready (could be
                                          //susp'd)
        OSRdyGrp        |=  bity;         //Put task in the ready to run list
        OSRdyTbl[y]     |=  bitx;
    }
}
#endif
```

eg1.c

```
/* File:        example1.c
 * uC/OS Real-time multitasking kernel for the ARM processor.
 * Simple example of using multiple tasks and mailboxes.*/

#include  "includes.h"              /* uC/OS interface */

/* allocate memory for tasks' stacks */
#ifdef SEMIHOSTED
#define   STACKSIZE (64+SEMIHOSTED_STACK_NEEDS)
#else
#define   STACKSIZE 64
#endif
OS_STK Stack1[STACKSIZE];
OS_STK Stack2[STACKSIZE];
OS_STK Stack3[STACKSIZE];

/* mailbox event control blocks */
OS_EVENT *Mbox1;
OS_EVENT *Mbox2;
OS_EVENT *Mbox3;

char PassMsg[] = "Example 1";

/* Task running at the lowest priority.
 * Wait for a message in the first mailbox and post
 * a messages to the third mailbox. */

void Task1(void *Id)
{   char *Msg;
    INT8U err;
    uHALr_printf("Task1() called\n");
    for (;;)
    {   /* wait for a message from the input mailbox */
        Msg = (char *)OSMboxPend(Mbox1, 0, &err);

        /* print task's id */
        uHALr_printf("%c", *(char *)Id);

        /* post the input message to the output mailbox */
        OSMboxPost(Mbox2, Msg);
    }
}

void Task2(void *Id)
{   char *Msg;
    INT8U err;
    uHALr_printf("Task2() called\n");
    for (;;)
    {    /* wait for a message from the input mailbox */
        Msg = (char *)OSMboxPend(Mbox2, 0, &err);
```

```
        /* print task's id */
        uHALr_printf("%c", *(char *)Id);

/* post the input message to the output mailbox */
        OSMboxPost(Mbox3, Msg);
    }
}

void Task3(void *Id)
{   char *Msg;
    INT8U err;
    uHALr_printf("Task3() called\n");
    for (;;)
    {     /* wait for a message from the input mailbox */
        Msg = (char *)OSMboxPend(Mbox3, 0, &err);

        /* print task's id */
        uHALr_printf("%c", *(char *)Id);

        /* post the input message to the output mailbox */
        OSMboxPost(Mbox1, Msg);
    }
}

/* Main function.*/
int main(int argc, char **argv)
{   char Id1 = '1';
    char Id2 = '2';
    char Id3 = '3';

    /* do target (uHAL based ARM system) initialisation */
    ARMTargetInit();

    /* needed by uC/OS */
    OSInit();

    /* create the first mailbox in the pipeline with a message
     * in it to get the first task started.*/

    Mbox1 = OSMboxCreate(PassMsg);

    /* create the remaining mailboxes empty */
    Mbox2 = OSMboxCreate((void *)0);
    Mbox3 = OSMboxCreate((void *)0);

    /* create the tasks in uC/OS and assign increasing
     * priorities to them so that Task3 at the end of
     * the pipeline has the highest priority.
     */
    OSTaskCreate(Task1,(void*)&Id1,(void*)&Stack1[STACKSIZE-1],3);
    OSTaskCreate(Task2,(void*)&Id2,(void*)&Stack2[STACKSIZE-1],2);
    OSTaskCreate(Task3,(void*)&Id3,(void*)&Stack3[STACKSIZE-1],1);
```

```
    /* Start the (uHAL based ARM system) system running */
    ARMTargetStart();

    /* start the game */
    OSStart();
    /* never reached */
}                                   /* main */
```

includes.h

```
/*********************************************
#include    "uhal.h"
#include    "os_cpu.h"
#include    "os_cfg.h"
#include    "ucos_ii.h"

#ifdef      EX3_GLOBALS
#define     EX3_EXT
#else
#define     EX3_EXT  extern
#endif

/*  DATA TYPES*/
typedef struct {
    char    TaskName[30];
    INT16U  TaskCtr;
    INT16U  TaskExecTime;
    INT32U  TaskTotExecTime;
} TASK_USER_DATA;

/*    VARIABLES */
EX3_EXT  TASK_USER_DATA  TaskUserData[10];

/*    FUNCTION PROTOTYPES */
void   DispTaskStat(INT8U id);
```

os_cfg.h

```
/*    Configuration for Intel 80x86 (Large)*/
/*    uC/OS-II CONFIGURATION  */

#define OS_MAX_EVENTS     20    //Max. number of event control blocks in your
                                //application. MUST be >= 2
#define OS_MAX_MEM_PART   10    //Max. number of memory partitions. MUST be >= 2
#define OS_MAX_QS          5    //Max. number of queue control blocks in your
                                //application. MUST be >= 2
#define OS_MAX_TASKS      32    //Max. number of tasks in your application.
                                //MUST be >= 2

#define OS_LOWEST_PRIO    63    //Defines the lowest priority that can be
                                //assigned. MUST NEVER be higher than 63!
#define OS_TASK_IDLE_STK_SIZE  512 //Idle task stack size (# of 16-bit wide
                                   //entries)
#define OS_TASK_STAT_EN          0 //Enable (1) or Disable(0) the statistics
                                   //task
#define OS_TASK_STAT_STK_SIZE   512  //Statistics task stack size (# of 16-bit
```

```
                                        //wide entries)
#define OS_CPU_HOOKS_EN           1     //uC/OS-II hooks are NOT found in the
                                        //processor port files
#define OS_MBOX_EN                1       //Include code for MAILBOXES
#define OS_MEM_EN                 0       //Include code for MEMORY MANAGER (fixed
                                          //sized memory blocks)
#define OS_Q_EN                   1       //Include code for QUEUES
#define OS_SEM_EN                 1       //Include code for SEMAPHORES
#define OS_TASK_CHANGE_PRIO_EN    0       //Include code for OSTaskChangePrio()
#define OS_TASK_CREATE_EN         1       //Include code for OSTaskCreate()
#define OS_TASK_CREATE_EXT_EN     0       //Include code for OSTaskCreateExt()
#define OS_TASK_DEL_EN            0       //Include code for OSTaskDel()
#define OS_TASK_SUSPEND_EN        1       //Include code for OSTaskSuspend() and
                                          //OSTaskResume()
#define OS_TICKS_PER_SEC        200       //Set the number of ticks in one second
```

os_core.c

```
/*      CORE FUNCTIONS  */

#ifndef  OS_MASTER_FILE
#define  OS_GLOBALS
#include "includes.h"
#endif

/*      LOCAL GLOBAL VARIABLES   */
static  INT8U     OSIntExitY;     //Variable used by 'OSIntExit' to prevent using
                                  //locals
static  OS_STK    OSTaskIdleStk[OS_TASK_IDLE_STK_SIZE];   //Idle task stack

#if     OS_TASK_STAT_EN
static  OS_STK   OSTaskStatStk[OS_TASK_STAT_STK_SIZE];    //Statistics task stack
#endif

static  OS_TCB   OSTCBTbl[OS_MAX_TASKS + OS_N_SYS_TASKS]; //Table of TCBs

/*$PAGE*/
/*     MAPPING TABLE TO MAP BIT POSITION TO BIT MASK
 * Note: Index into table is desired bit position, 0..7
 *       Indexed value corresponds to bit mask */

INT8U const OSMapTbl[]={0x01,0x02,0x04,0x08,0x10,0x20,0x40,0x80};

/*      PRIORITY RESOLUTION TABLE
 * Note: Index into table is bit pattern to resolve highest priority
 *       Indexed value corresponds to highest priority bit position (i.e. 0..7)
*/
INT8U const OSUnMapTbl[] = {
    0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    7, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    6, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
    5, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0,
```

```
    4, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0
};

/*$PAGE*/
/*   MAKE TASK READY TO RUN BASED ON EVENT OCCURING
  Description: This function is called by other uC/OS-II services and is used to
  ready a task that was waiting for an event to occur.
  Arguments:
       pevent is a pointer to the event control block corresponding to the event.
       msg is a pointer to a message. This pointer is used by message oriented
       services such as MAILBOXEs and QUEUEs. The pointer is not used when called
       by other service functions.
       msk is a mask that is used to clear the status byte of the TCB. For
       example, OSSemPost() will pass OS_STAT_SEM, OSMboxPost() will pass
       OS_STAT_MBOX etc.
  Returns    : none
  Note: This function is INTERNAL to uC/OS-II and your application should not call
        it.
*/
#if  (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
void  OSEventTaskRdy (OS_EVENT *pevent, void *msg, INT8U msk)
{
    OS_TCB *ptcb;
    INT8U   x;
    INT8U   y;
    INT8U   bitx;
    INT8U   bity;
    INT8U   prio;

    y = OSUnMapTbl[pevent->OSEventGrp];  //Find highest prio. task waiting for
                                         //message
    bity = OSMapTbl[y];
    x    = OSUnMapTbl[pevent->OSEventTbl[y]];
    bitx = OSMapTbl[x];
    prio = (INT8U)((y << 3) + x);  //Find priority of task getting the msg

    if ((pevent->OSEventTbl[y] &= ~bitx) == 0) {//Remove this task from the
                                                //waiting list
       pevent->OSEventGrp &= ~bity;
    }
    ptcb           = OSTCBPrioTbl[prio];       //Point to this task's OS_TCB
    ptcb->OSTCBDly = 0;                //Prevent OSTimeTick() from readying task
    ptcb->OSTCBEventPtr = (OS_EVENT *)0;    //Unlink ECB from this task
#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    ptcb->OSTCBMsg   = msg;        //Send message directly to waiting task
#else
    msg              = msg;        //Prevent compiler warning if not used
#endif
    ptcb->OSTCBStat &= ~msk;       //Clear bit associated with event type
    if (ptcb->OSTCBStat == OS_STAT_RDY) { //See if task is ready (could be
                                          //susp'd)
        OSRdyGrp        |=  bity;         //Put task in the ready to run list
        OSRdyTbl[y]     |=  bitx;
    }
}
#endif
/*$PAGE*/
/*     MAKE TASK WAIT FOR EVENT TO OCCUR
  Description: This function is called by other uC/OS-II services to suspend a
               task because an event has not occurred.
  Arguments: pevent is a pointer to the event control block for which the task
             will be waiting for.
  Returns: none
```

```
  Note: This function is INTERNAL to uC/OS-II and your application should not cal
        it.*/

#if  (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
void  OSEventTaskWait (OS_EVENT *pevent)
{
    OSTCBCur->OSTCBEventPtr = pevent;    //Store pointer to event control block in
                                         //TCB
    if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) { //Task no
                                                                    //longer ready
        OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
    }
    pevent->OSEventTbl[OSTCBCur->OSTCBY] |= OSTCBCur->OSTCBBitX;  //Put task in
                                                                   //waiting list
    pevent->OSEventGrp                   |= OSTCBCur->OSTCBBitY;
}
#endif
/*$PAGE*/
/*   MAKE TASK READY TO RUN BASED ON EVENT TIMEOUT
  Description: This function is called by other uC/OS-II services to make a task
               ready to run because a timeout occurred.
  Arguments: pevent is a pointer to the event control block which is readying a
             task.
  Returns: none
  Note: This function is INTERNAL to uC/OS-II and your application should not call
        it.
*/
#if  (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
void  OSEventTO (OS_EVENT *pevent)
{
    if ((pevent->OSEventTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {
        pevent->OSEventGrp &= ~OSTCBCur->OSTCBBitY;
    }
    OSTCBCur->OSTCBStat     = OS_STAT_RDY;    //Set status to ready
    OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;  //No longer waiting for event
}
#endif
/*$PAGE*/
/*       INITIALIZE EVENT CONTROL BLOCK'S WAIT LIST
  Description: This function is called by other uC/OS-II services to initialize
               the event wait list.
  Arguments: pevent is a pointer to the event control block allocated to the
             event.
  Returns: none
  Note: This function is INTERNAL to uC/OS-II and your application should not call
        it.
*/
#if  (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
void  OSEventWaitListInit (OS_EVENT *pevent)
{
    INT8U i;


    pevent->OSEventGrp = 0x00;       //No task waiting on event
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
        pevent->OSEventTbl[i] = 0x00;
    }
}
#endif
/*$PAGE*/
/*    INITIALIZATION
  Description: This function is used to initialize the internals of uC/OS-II and
               MUST be called prior to creating any uC/OS-II object and, prior to
```

```
              calling OSStart().
  Arguments: none
  Returns: none
*/
void OSInit (void)
{
    INT16U i;

    OSTime        = 0L;              //Clear the 32-bit system clock
    OSIntNesting  = 0;               //Clear the interrupt nesting counter
    OSLockNesting = 0;               //Clear the scheduling lock counter
#if OS_TASK_CREATE_EN  || OS_TASK_CREATE_EXT_EN || OS_TASK_DEL_EN
    OSTaskCtr     = 0;               //Clear the number of tasks
#endif
    OSRunning     = FALSE;           //Indicate that multitasking not started
    OSIdleCtr     = 0L;              //Clear the 32-bit idle counter
#if OS_TASK_STAT_EN && OS_TASK_CREATE_EXT_EN
    OSIdleCtrRun  = 0L;
    OSIdleCtrMax  = 0L;
    OSStatRdy     = FALSE;           //Statistic task is not ready
#endif
    OSCtxSwCtr    = 0;               //Clear the context switch counter
    OSRdyGrp      = 0;               //Clear the ready list
    for (i = 0; i < OS_RDY_TBL_SIZE; i++) {
        OSRdyTbl[i] = 0;
    }

    OSPrioCur     = 0;
    OSPrioHighRdy = 0;
    OSTCBHighRdy  = (OS_TCB *)0;                //TCB Initialization
    OSTCBCur      = (OS_TCB *)0;
    OSTCBList     = (OS_TCB *)0;
    for (i = 0; i < (OS_LOWEST_PRIO + 1); i++) { //Clear the priority table
        OSTCBPrioTbl[i] = (OS_TCB *)0;
    }
    for (i = 0; i < (OS_MAX_TASKS + OS_N_SYS_TASKS - 1); i++) { //Init. list of
                                                                // free TCBs
        OSTCBTbl[i].OSTCBNext = &OSTCBTbl[i + 1];
    }
    OSTCBTbl[OS_MAX_TASKS + OS_N_SYS_TASKS - 1].OSTCBNext = (OS_TCB *)0;
                                                    //Last OS_TCB
    OSTCBFreeList                                        = &OSTCBTbl[0];

#if OS_MAX_EVENTS >= 2
    for (i = 0; i < (OS_MAX_EVENTS - 1); i++) {  //Init. list of free EVENT
                                                 //control blocks
        OSEventTbl[i].OSEventPtr = (OS_EVENT *)&OSEventTbl[i + 1];
    }
    OSEventTbl[OS_MAX_EVENTS - 1].OSEventPtr = (OS_EVENT *)0;
    OSEventFreeList                          = &OSEventTbl[0];
#endif

#if OS_Q_EN && (OS_MAX_QS >= 2)
    OSQInit();                       //Initialize the message queue structures
#endif

#if OS_MEM_EN && OS_MAX_MEM_PART >= 2
    OSMemInit();                     //Initialize the memory manager
#endif

#if OS_STK_GROWTH == 1
    #if OS_TASK_CREATE_EXT_EN
    OSTaskCreateExt(OSTaskIdle,
```

```
                    (void *)0,               /No arguments passed to OSTaskIdle()
                    &OSTaskIdleStk[OS_TASK_IDLE_STK_SIZE - 1], //Set Top-Of-Stack
                    OS_IDLE_PRIO,                          //Lowest priority level
                    OS_TASK_IDLE_ID,
                    &OSTaskIdleStk[0],                     //Set Bottom-Of-Stack
                    OS_TASK_IDLE_STK_SIZE,
                    (void *)0,                             //No TCB extension
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);//Enable stack
                                                          //checking + clear stack
    #else
    OSTaskCreate(OSTaskIdle, (void *)0, &OSTaskIdleStk[OS_TASK_IDLE_STK_SIZE - 1],
OS_IDLE_PRIO);
    #endif
#else
    #if OS_TASK_CREATE_EXT_EN
    OSTaskCreateExt(OSTaskIdle,
                    (void *)0,              //No arguments passed to OSTaskIdle()
                    &OSTaskIdleStk[0],      //Set Top-Of-Stack
                    OS_IDLE_PRIO,           //Lowest priority level
                    OS_TASK_IDLE_ID,
                    &OSTaskIdleStk[OS_TASK_IDLE_STK_SIZE - 1], //Set Bottom-Of-
                                                               // Stack
                    OS_TASK_IDLE_STK_SIZE,
                    (void *)0,                                 //No TCB extension
                    OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);//Enable stack
                                                            //checking + clear stack
    #else
    OSTaskCreate(OSTaskIdle, (void *)0, &OSTaskIdleStk[0], OS_IDLE_PRIO);
    #endif
#endif

#if OS_TASK_STAT_EN
    #if OS_TASK_CREATE_EXT_EN
        #if OS_STK_GROWTH == 1
        OSTaskCreateExt(OSTaskStat,
                        (void *)0,              //No args passed to OSTaskStat()
                        &OSTaskStatStk[OS_TASK_STAT_STK_SIZE - 1],//Set Top-Of-
                                                                  //Stack
                        OS_STAT_PRIO,        //One higher than the idle task
                        OS_TASK_STAT_ID,
                        &OSTaskStatStk[0],   //Set Bottom-Of-Stack
                        OS_TASK_STAT_STK_SIZE,
                        (void *)0,                        //No TCB extension
                        OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);  //Enable
                                                          //stack checking + clear
        #else
        OSTaskCreateExt(OSTaskStat,
                        (void *)0,               //No args passed to OSTaskStat()
                        &OSTaskStatStk[0],       //Set Top-Of-Stack
                        OS_STAT_PRIO,            //One higher than the idle task
                        OS_TASK_STAT_ID,
                        &OSTaskStatStk[OS_TASK_STAT_STK_SIZE - 1],//Set Bottom-Of-
                                                                  //Stack
                        OS_TASK_STAT_STK_SIZE,
                        (void *)0,                    //No TCB extension
                        OS_TASK_OPT_STK_CHK | OS_TASK_OPT_STK_CLR);  //Enable
                                                        //stack checking + clear
        #endif
    #else
        #if OS_STK_GROWTH == 1
        OSTaskCreate(OSTaskStat,
                     (void *)0,                  //No args passed to OSTaskStat()
                     &OSTaskStatStk[OS_TASK_STAT_STK_SIZE - 1],   //Set Top-Of-
```

```
                                                                    //Stack
                    OS_STAT_PRIO);               //One higher than the idle task
        #else
        OSTaskCreate(OSTaskStat,
                    (void *)0,                  //No args passed to OSTaskStat()
                    &OSTaskStatStk[0],          //Set Top-Of-Stack
                    OS_STAT_PRIO);              //One higher than the idle task
        #endif
    #endif
#endif
}
/*$PAGE*/
/*   ENTER ISR
  Description: This function is used to notify uC/OS-II that you are about to
               service an interrupt service routine (ISR). This allows uC/OS-II to
               keep track of interrupt nesting and thus only perform rescheduling
               at the last nested ISR.
  Arguments: none
  Returns: none
  Notes: 1) Your ISR can directly increment OSIntNesting without calling this
         function because OSIntNesting has been declared 'global'. You MUST,
         however, be sure that the increment is performed 'indivisibly' by your
         processor to ensure proper access to this critical resource.
         2) You MUST still call OSIntExit() even though you increment OSIntNesting
         directly.
         3) You MUST invoke OSIntEnter() and OSIntExit() in pair.  In other words,
         for every call to OSIntEnter() at the beginning of the ISR you MUST have
         a call to OSIntExit() at the end of the ISR.
*/
void OSIntEnter (void)
{
  //    OS_ENTER_CRITICAL();
    OSIntNesting++;                  //Increment ISR nesting level
  //    OS_EXIT_CRITICAL();
}
/*$PAGE*/
/*       EXIT ISR
  Description: This function is used to notify uC/OS-II that you have completed
               serviving an ISR.  When the last nested ISR has completed, uC/OS-II
               will call the scheduler to determine whether a new, high-priority
               task, is ready to run.
  Arguments: none
  Returns: none
  Notes: 1) You MUST invoke OSIntEnter() and OSIntExit() in pair.  In other words,
         for every call to OSIntEnter() at the beginning of the ISR you MUST have
         a call to OSIntExit() at the end of the ISR.
         2) Rescheduling is prevented when the scheduler is locked (see
         OSSchedLock())
*/
void OSIntExit (void)
{
  //    OS_ENTER_CRITICAL();
    if ((--OSIntNesting | OSLockNesting) == 0) { //Reschedule only if all ISRs
                                                 //completed & not locked
        OSIntExitY    = OSUnMapTbl[OSRdyGrp];
        OSPrioHighRdy = (INT8U)((OSIntExitY << 3) +
OSUnMapTbl[OSRdyTbl[OSIntExitY]]);
        if (OSPrioHighRdy != OSPrioCur) {  //No context switch if current task is
                                           //highest ready
            OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;          //Keep track of the number of context switches
            OSIntCtxSw();          //Perform interrupt level context switch
        }
```

```
    }
    //    OS_EXIT_CRITICAL();
}
/*$PAGE*/
/*   SCHEDULER
  Description: This function is called by other uC/OS-II services to determine
               whether a new, high priority task has been made ready to run.  This
               function is invoked by TASK level code and is not used to
               reschedule tasks from ISRs (see OSIntExit() for ISR rescheduling).
  Arguments: none
  Returns: none
  Notes: 1) This function is INTERNAL to uC/OS-II and your application should not
         call it.
         2) Rescheduling is prevented when the scheduler is locked (see
         OSSchedLock())
*/
void OSSched (void)
{
    INT8U y;


    OS_ENTER_CRITICAL();
    if ((OSLockNesting | OSIntNesting) == 0) {  //Task scheduling must be enabled
                                                //and not ISR level
        y             = OSUnMapTbl[OSRdyGrp];   //Get pointer to highest priority
                                                //task ready to run
        OSPrioHighRdy = (INT8U)((y << 3) + OSUnMapTbl[OSRdyTbl[y]]);
        if (OSPrioHighRdy != OSPrioCur) {       //No context switch if current
                                                //task is highest ready
            OSTCBHighRdy = OSTCBPrioTbl[OSPrioHighRdy];
            OSCtxSwCtr++;                       //Increment context switch counter
            OS_TASK_SW();                        /* Perform a context switch
        }
    }
    OS_EXIT_CRITICAL();
}
/*$PAGE*/
/*    PREVENT SCHEDULING
  Description: This function is used to prevent rescheduling to take place.  This
               allows your application to prevent context switches until you are
               ready to permit context switching.
  Arguments: none
  Returns: none
  Notes: 1) You MUST invoke OSSchedLock() and OSSchedUnlock() in pair.  In other
         words, for every call to OSSchedLock() you MUST have a call to
         OSSchedUnlock().
*/
void OSSchedLock (void)
{
    if (OSRunning == TRUE) {      //Make sure multitasking is running
        OS_ENTER_CRITICAL();
        OSLockNesting++;          //Increment lock nesting level
        OS_EXIT_CRITICAL();
    }
}
/*$PAGE*/
/*   ENABLE SCHEDULING
  Description: This function is used to re-allow rescheduling.
  Arguments: none
  Returns: none
  Notes: 1) You MUST invoke OSSchedLock() and OSSchedUnlock() in pair.  In other
         words, for every call to OSSchedLock() you MUST have a call to
         OSSchedUnlock().
```

```
*/
void OSSchedUnlock (void)
{
    if (OSRunning == TRUE) {       //Make sure multitasking is running
        OS_ENTER_CRITICAL();
        if (OSLockNesting > 0) {   //Do not decrement if already 0
            OSLockNesting--;       //Decrement lock nesting level
            if ((OSLockNesting | OSIntNesting) == 0) { //See if scheduling re-
                                                       //enabled and not an ISR
                OS_EXIT_CRITICAL();
                OSSched();         //See if a higher priority task is ready
            } else {
                OS_EXIT_CRITICAL();
            }
        } else {
            OS_EXIT_CRITICAL();
        }
    }
}
/*$PAGE*/
/*     START MULTITASKING
  Description: This function is used to start the multitasking process which lets
               uC/OS-II manages the task that you have created.  Before you can call
               OSStart(), you MUST have called OSInit() and you MUST have created at
               least one task.
  Arguments: none
  Returns: none
  Note: OSStartHighRdy() MUST:
        a) Call OSTaskSwHook() then,
        b) Set OSRunning to TRUE.
*/
void OSStart (void)
{
    INT8U y;
    INT8U x;


    if (OSRunning == FALSE) {
        y             = OSUnMapTbl[OSRdyGrp]; //Find highest priority's task
                                              //priority number
        x             = OSUnMapTbl[OSRdyTbl[y]];
        OSPrioHighRdy = (INT8U)((y << 3) + x);
        OSPrioCur     = OSPrioHighRdy;
        OSTCBHighRdy  = OSTCBPrioTbl[OSPrioHighRdy]; //Point to highest priority
                                                     //task ready to run
        OSTCBCur      = OSTCBHighRdy;
        OSStartHighRdy();            //Execute target specific code to start task
    }
}
/*$PAGE*/
/*    STATISTICS INITIALIZATION
  Description: This function is called by your application to establish CPU usage
               by first determining how high a 32-bit counter would count to in 1
               second if no other tasks were to execute during that time.  CPU
               usage is then determined by a low priority task which keeps track
               of this 32-bit counter every second but this time, with other tasks
               running.  CPU usage is determined by:
           OSIdleCtr
           CPU Usage (%) = 100 * (1 - ------------)
           OSIdleCtrMax
  Arguments: none
  Returns: none
*/
```

```
#if OS_TASK_STAT_EN
void OSStatInit (void)
{
    OSTimeDly(2);          //Synchronize with clock tick
    OS_ENTER_CRITICAL();
    OSIdleCtr    = 0L;     //Clear idle counter
    OS_EXIT_CRITICAL();
    OSTimeDly(OS_TICKS_PER_SEC);    //Determine MAX. idle counter value for 1
                                    //second
    OS_ENTER_CRITICAL();
    OSIdleCtrMax = OSIdleCtr;       //Store maximum idle counter count in 1 second
    OSStatRdy    = TRUE;
    OS_EXIT_CRITICAL();
}
#endif
/*$PAGE*/
/*      IDLE TASK
  Description: This task is internal to uC/OS-II and executes whenever no other
       higher priority tasks executes because they are waiting for event(s) to
       occur.
  Arguments: none
  Returns: none
*/
void OSTaskIdle (void *pdata)
{
    pdata = pdata;           //Prevent compiler warning for not using 'pdata'
    for (;;) {
        OS_ENTER_CRITICAL();
        OSIdleCtr++;
        OS_EXIT_CRITICAL();
    }
}
/*$PAGE*/
/*      STATISTICS TASK
  Description: This task is internal to uC/OS-II and is used to compute some
      statistics about the multitasking environment.  Specifically, OSTaskStat()
      computes the CPU usage. CPU usage is determined by:
         OSIdleCtr
         OSCPUUsage = 100 * (1 - ------------)     (units are in %)
         OSIdleCtrMax
  Arguments: pdata  this pointer is not used at this time.
  Returns: none
  Notes: 1) This task runs at a priority level higher than the idle task.  In
         fact, it runs at the next higher priority, OS_IDLE_PRIO-1.
         2) You can disable this task by setting the configuration #define
         OS_TASK_STAT_EN to 0.
         3) We delay for 5 seconds in the beginning to allow the system to reach
         steady state and have all other tasks created before we do statistics.
         You MUST have at least a delay of 2 seconds to allow for the system to
         establish the maximum value for the idle counter.
*/
#if OS_TASK_STAT_EN
void OSTaskStat (void *pdata)
{
    INT32U run;
    INT8S  usage;

    pdata = pdata;            //Prevent compiler warning for not using 'pdata'
    while (OSStatRdy == FALSE) {
        OSTimeDly(2 * OS_TICKS_PER_SEC);   //Wait until statistic task is ready
    }
    for (;;) {
        OS_ENTER_CRITICAL();
```

```
        OSIdleCtrRun = OSIdleCtr;      //Obtain the of the idle counter for the
                                       //past second
        run          = OSIdleCtr;
        OSIdleCtr    = 0L;             //Reset the idle counter for the next
                                       //second
        OS_EXIT_CRITICAL();
        if (OSIdleCtrMax > 0L) {
            usage = (INT8S)(100L - 100L * run / OSIdleCtrMax);
            if (usage > 100) {
                OSCPUUsage = 100;
            } else if (usage < 0) {
                OSCPUUsage =   0;
            } else {
                OSCPUUsage = usage;
            }
        } else {
            OSCPUUsage = 0;
        }
        OSTaskStatHook();                  //Invoke user definable hook
        OSTimeDly(OS_TICKS_PER_SEC);       //Accumulate OSIdleCtr for the next
                                           //second
    }
}
#endif
/*$PAGE*/
/* INITIALIZE TCB
  Description: This function is internal to uC/OS-II and is used to initialize a
         Task Control Block when a task is created (see OSTaskCreate() and
         OSTaskCreateExt()).
  Arguments: prio  is the priority of the task being created
    ptos is a pointer to the task's top-of-stack assuming that the CPU registers
     have been placed on the stack.  Note that the top-of-stack corresponds to a
     'high' memory location is OS_STK_GROWTH is set to 1 and a 'low' memory
     location if OS_STK_GROWTH is set to 0.  Note that stack growth is CPU
     specific.
    pbos is a pointer to the bottom of stack.  A NULL pointer is passed if called
     by 'OSTaskCreate()'.
    id is the task's ID (0..65535)
    stk_size is the size of the stack (in 'stack units'). If the stack units are
     INT8Us then, 'stk_size' contains the number of bytes for the stack.  If the
     stack units are INT32Us then, the stack contains '4 * stk_size' bytes.  The
     stack units are established by the #define constant OS_STK which is CPU
     specific.  'stk_size' is 0 if called by 'OSTaskCreate()'.
    pext is a pointer to a user supplied memory area that is used to extend the
     task control block.  This allows you to store the contents of floating-point
     registers, MMU registers or anything else you could find useful during a
     context switch.  You can even assign a name to each task and store this name
     in this TCB extension.  A NULL pointer is passed if called by OSTaskCreate().
    opt options as passed to 'OSTaskCreateExt()' or, 0 if called from
     'OSTaskCreate()'.
  Returns: OS_NO_ERR         if the call was successful OS_NO_MORE_TCB    if there
     are no more free TCBs to be allocated and thus, the task cannot be created.
  Note: This function is INTERNAL to uC/OS-II and your application should not call
        it.
*/
INT8U OSTCBInit (INT8U prio, OS_STK *ptos, OS_STK *pbos, INT16U id, INT16U
stk_size, void *pext, INT16U opt)
{
    OS_TCB *ptcb;

    OS_ENTER_CRITICAL();
    ptcb = OSTCBFreeList;           //Get a free TCB from the free TCB list
    if (ptcb != (OS_TCB *)0) {
```

```
        OSTCBFreeList        = ptcb->OSTCBNext;  //Update pointer to free TCB list
        OS_EXIT_CRITICAL();
        ptcb->OSTCBStkPtr    = ptos;             //Load Stack pointer in TCB
        ptcb->OSTCBPrio      = (INT8U)prio;      //Load task priority into TCB
        ptcb->OSTCBStat      = OS_STAT_RDY;      //Task is ready to run
        ptcb->OSTCBDly       = 0;                //Task is not delayed

#if OS_TASK_CREATE_EXT_EN
        ptcb->OSTCBExtPtr    = pext;            //Store pointer to TCB extension
        ptcb->OSTCBStkSize   = stk_size;        //Store stack size
        ptcb->OSTCBStkBottom = pbos;            //Store pointer to bottom of stack
        ptcb->OSTCBOpt       = opt;             //Store task options
        ptcb->OSTCBId        = id;              //Store task ID
#else
        pext                 = pext;        //Prevent compiler warning if not used
        stk_size             = stk_size;
        pbos                 = pbos;
        opt                  = opt;
        id                   = id;
#endif

#if OS_TASK_DEL_EN
        ptcb->OSTCBDelReq    = OS_NO_ERR;
#endif

        ptcb->OSTCBY         = prio >> 3;    //Pre-compute X, Y, BitX and BitY
        ptcb->OSTCBBitY      = OSMapTbl[ptcb->OSTCBY];
        ptcb->OSTCBX         = prio & 0x07;
        ptcb->OSTCBBitX      = OSMapTbl[ptcb->OSTCBX];

#if     OS_MBOX_EN || (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_SEM_EN
        ptcb->OSTCBEventPtr  = (OS_EVENT *)0;     //Task is not pending on an
                                                  //event
#endif

#if     OS_MBOX_EN || (OS_Q_EN && (OS_MAX_QS >= 2))
        ptcb->OSTCBMsg       = (void *)0;    //No message received
#endif

        OS_ENTER_CRITICAL();
        OSTCBPrioTbl[prio]   = ptcb;
        ptcb->OSTCBNext      = OSTCBList;    //Link into TCB chain
        ptcb->OSTCBPrev      = (OS_TCB *)0;
        if (OSTCBList != (OS_TCB *)0) {
            OSTCBList->OSTCBPrev = ptcb;
        }
        OSTCBList               = ptcb;
        OSRdyGrp               |= ptcb->OSTCBBitY;  //Make task ready to run
        OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
        OS_EXIT_CRITICAL();
        return (OS_NO_ERR);
    } else {
        OS_EXIT_CRITICAL();
        return (OS_NO_MORE_TCB);
    }
}
/*$PAGE*/
/*        PROCESS SYSTEM TICK
  Description: This function is used to signal to uC/OS-II the occurrence of a
       'system tick' (also known as a 'clock tick').  This function should be
       called by the ticker ISR but, can also be called by a high priority task.
  Arguments: none
  Returns: none
```

```
*/
void OSTimeTick (void)
{
    OS_TCB *ptcb;

    OSTimeTickHook();                          //Call user definable hook
    ptcb = OSTCBList;                          //Point at first TCB in TCB list
    while (ptcb->OSTCBPrio != OS_IDLE_PRIO) {  //Go through all TCBs in TCB list
      //        OS_ENTER_CRITICAL();
        if (ptcb->OSTCBDly != 0) {             //Delayed or waiting for event with TO
            if (--ptcb->OSTCBDly == 0) {       //Decrement nbr of ticks to end of
                                               //delay
                if (!(ptcb->OSTCBStat & OS_STAT_SUSPEND)) {  //Is task suspended?
                    OSRdyGrp               |= ptcb->OSTCBBitY; //No,  Make task
                                                        //Rdy to Run (timed out)
                    OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                } else {                               //Yes, Leave 1 tick to prevent
                    ptcb->OSTCBDly = 1;                //...loosing the task when the
                }                                      //...suspension is removed.
            }
        }
        ptcb = ptcb->OSTCBNext;                //Point at next TCB in TCB list
      //        OS_EXIT_CRITICAL();
    }
    //    OS_ENTER_CRITICAL();                 //Update the 32-bit tick counter
    OSTime++;
    //    OS_EXIT_CRITICAL();
}
/*$PAGE*/
/*       GET VERSION
  Description: This function is used to return the version number of uC/OS-II.
       The returned value corresponds to uC/OS-II's version number multiplied by
       100. In other words, version 2.00 would be returned as 200.
  Arguments: none
  Returns: the version number of uC/OS-II multiplied by 100.
*/
INT16U OSVersion (void)
{
    return (OS_VERSION);
}
```

os_cpu.h

```
/*
*********************************************************************************
**********************
*                                               uC/OS-II
*                                         The Real-Time Kernel
*
*                        (c) Copyright 1992-1998, Jean J. Labrosse, Plantation, FL
*                                           All Rights Reserved
*                        (c) Copyright ARM Limited 1999.  All rights reserved.
*
*                                           ARM Specific code
*
* File : OS_CPU.H
*********************************************************************************
**********************
*/

#ifdef  OS_CPU_GLOBALS
#define OS_CPU_EXT
#else
```

```
#define OS_CPU_EXT  extern
#endif

/*
*********************************************************************************
**********************
*                                             DATA TYPES
*                                          (Compiler Specific)
*********************************************************************************
**********************
*/

typedef unsigned char  BOOLEAN;
typedef unsigned char  INT8U;                    /* Unsigned  8 bit quantity
*/
typedef signed   char  INT8S;                    /* Signed    8 bit quantity
*/
typedef unsigned int   INT16U;                   /* Unsigned 16 bit quantity
*/
typedef signed   int   INT16S;                   /* Signed   16 bit quantity
*/
typedef unsigned long  INT32U;                   /* Unsigned 32 bit quantity
*/
typedef signed   long  INT32S;                   /* Signed   32 bit quantity
*/
typedef float          FP32;                     /* Single precision floating
point                    */
typedef double         FP64;                     /* Double precision floating
point                    */

typedef unsigned int   OS_STK;                   /* Each stack entry is 16-bit
wide                     */

#define BYTE           INT8S                     /* Define data types for backward
compatibility ...   */
#define UBYTE          INT8U                     /* ... to uC/OS V1.xx.  Not actu-
ally needed for ...   */
#define WORD           INT16S                    /* ... uC/OS-II.
*/
#define UWORD          INT16U
#define LONG           INT32S
#define ULONG          INT32U

/*
*********************************************************************************
**********************
*                               ARM, various architectures
*
*********************************************************************************
**********************
*/
#define     OS_ENTER_CRITICAL()     ARMDisableInt()
#define     OS_EXIT_CRITICAL()      ARMEnableInt()

/*
 * Definitions specific to ARM/uHAL
 */
#define     SVC32MODE   0x13

/* stack stuff */
#define OS_STK_GROWTH    1

/* angel takes up stack */
```

```
#define SEMIHOSTED_STACK_NEEDS 1024

/* idle task stack size (words) */
#ifdef SEMIHOSTED
#define OS_IDLE_STK_SIZE        (32+SEMIHOSTED_STACK_NEEDS)
#else
#define OS_IDLE_STK_SIZE        32
#endif

/*
 * Functions specific to uHAL
 */
/* defined in os_cpu_c.c */
extern void IrqStart(void);
extern PrVoid IrqFinish(void);
extern void ARMTargetInit(void);
extern void ARMTargetStart(void);

/* defined in os_cpu_a.s */
extern void OS_TASK_SW(void);
extern void ARMDisableInt(void);
extern void ARMEnableInt(void);
```

os_cpu_a.s

```
;*
;* File: os_cpu_a.s
;*
;*            (c) Copyright ARM Limited 1999.  All rights reserved.
;*
;*                                ARM Specific code
;*
;*
;
;     Functions defined in this module:
;
;     void ARMDisableInt(void)      /* disable interrupts when in SVC */
;     void ARMEnableInt(void)       /* enable interrupts when in SVC */
;     void OS_TASK_SWAP(void)       /* context switch */
;     void OSStartHighRdy(void)     /* start highest priority task */

SwiV        EQU   0x08
IrqV        EQU   0x18
FiqV        EQU   0x1C
NoInt       EQU   0x80

SVC32Mode   EQU   0x13
IRQ32Mode   EQU   0x12
FIQ32Mode   EQU   0x11
USR32Mode   EQU   0x10

OSEnterSWI  EQU   0x00

      AREA  |subr|, CODE, READONLY

      ; Improper use of locations within a READONLY area
SavedIRQ    DCD   0x0
SavedFIQ    DCD   0x0
SavedSWI    DCD   0x0

      ; External symbols we need the addresses of
                  IMPORT      OSTCBCur
addr_OSTCBCur           DCD   OSTCBCur
                  IMPORT      OSTCBHighRdy
```

```
addr_OSTCBHighRdy DCD   OSTCBHighRdy
                  IMPORT      OSPrioCur
addr_OSPrioCur          DCD   OSPrioCur
                  IMPORT      OSPrioHighRdy
addr_OSPrioHighRdy      DCD   OSPrioHighRdy

      EXPORT IRQContextSwap
IRQContextSwap
      ; NOTE: The following code assumes that all threads use r13 as
      ; the stack-pointer, and that it is a APCS conformant stack.
      ; i.e. there is never any data stored beneath the current
      ; stack-pointer.
      ; The above needs to be true to enable the context switches
      ; started as a return from an interrupt to use the current
      ; threads stack as the state save area.
      ;
      LDMFD sp!,{r12}         ; recover SPSR value from stack
      ; if NE then we need to check if we were a nested interrupt
      AND   r11,r12,#0x1F           ; mask out all but the mode bits
      TEQNE r11,#IRQ32Mode          ; check for interrupted IRQ thread
      TEQNE r11,#USR32Mode
      ; if EQ then we can return immediately
      MSREQ SPSR_cxsf,r12           ; restore the SPSR
      LDMEQFD     sp!,{r0-r12,pc}^  ; and return to the interrupted thread

      ; We now need to perform a context switch.
      ; r12 = SPSR describing the interrupted thread.
      ; r11 = interrupted thread processor mode

      ; We need to protect the SPSR before we actually perform the
      ; return to the interrupted thread, since we don't want to
      ; lose the value by another interrupt occuring between the
      ; SPSR load and the PC+CPSR load. Similarly we need to protect
      ; the IRQ stack and threading code while we setup the state
      ; required to enter the context switch code from an interrupt
      ; routine. We rely on the interrupted thread having IRQs
      ; enabled (since we would never have reached this point
      ; otherwise).
      ; We have recovered the SPSR value, so only r0-r12,lr are on the stack.

      LDR   r4,[sp,#(13 * 4)] ; load return address: nasty use of
                              ; a constant
 IF :DEF: THUMB_AWARE
      ANDS  r0,r12,#0x20            ; mask out all but Thumb bit
      ORRNE r4,r4,#1          ; convert to a Thumb return address
      STR   r4,[sp,#(13 * 4)] ; store it back
      BIC   r12,r12,#0x20
 ENDIF

      ; r11 contains the mode bits describing the interrupted thread
      MRS   r0,CPSR                 ; get current mode info.
      ORR   r0,r0,#0x80       ; and set IRQ disable flag
      BIC   r1,r0,#0x1F       ; clear mode bits
      ORR   r1,r1,r11         ; insert interrupted thread mode bits
      MSR   CPSR_c,r1               ; and change to that mode

      ; We are now in the interrupted thread mode with IRQs
      ; disabled.
      MOV   r3,lr             ; copy the current lr
      MRS   r1,SPSR                 ; copy current SPSR
      MRS   r2,CPSR                 ; copy current CPSR
      STMFD sp!,{r1,r2,r3,r4} ; and construct return stack
      MSR   CPSR_c,r0               ; return to IRQ mode
```

```
      ; IRQ mode; IRQs disabled
      ; r12 = SPSR describing interrupted thread
      ORR   r12,r12,#NoInt          ; disable interrupts
      MSR   SPSR_cxsf,r12           ; restore SPSR_irq ready for return
      LDMFD sp!,{r0-r12,lr}         ; restore all the registers
      SUBS  pc,pc,#0          ; and return to the interrupted mode
      NOP                     ; flush the pipeline
      NOP
      NOP

      ; we are now executing in the interrupted mode with IRQs enabled
      BL    OS_TASK_SW        ; perform the context switch
      LDMFD sp!,{lr}
      MSR   SPSR_c,lr         ; recover the SPSR when the thread
                              ; was interrupted
      LDMFD sp!,{lr}
      BIC   lr,lr,#NoInt            ; re-enable interrupts
      MSR   CPSR_c,lr         ; recover the CPSR when the thread
                              ; was interrupted
 IF :DEF: THUMB_AWARE
      LDMFD sp!,{lr}          ; restore link register
      LDMFD sp!,{r3}          ; get return address

      BX    r3
 ELSE
      LDMFD sp!,{lr,pc}       ; return to the interrupted thread
 ENDIF


;     void DisableInt(void)
;     void EnableInt(void)
;
;     Disable and enable IRQ and FIQ preserving current CPU mode.
;
      EXPORT      ARMDisableInt
ARMDisableInt
      MRS   r12, CPSR
      ORR   r12, r12, #NoInt
      MSR   CPSR_c, r12
 IF :DEF: THUMB_AWARE
      BX    lr
 ELSE
      MOV   pc, lr
 ENDIF

      EXPORT      ARMEnableInt
ARMEnableInt
      MRS   r12, CPSR
      BIC   r12, r12, #NoInt
      MSR   CPSR_c, r12
 IF :DEF: THUMB_AWARE
      BX    lr
 ELSE
      MOV   pc, lr
 ENDIF



;     void OS_TASK_SW(void)
;
;     Perform a context switch.
;
;     On entry, OSTCBCur and OSPrioCur hold the current TCB and priority
```

```
;     and OSTCBHighRdy and OSPrioHighRdy contain the same for the task
;     to be switched to.
;
;     The following code assumes that the virtual memory is directly
;     mapped into  physical memory. If this is not true, the cache must
;     be flushed at context switch to avoid address aliasing.

      EXPORT      OS_TASK_SW
OS_TASK_SW
      STMFD sp!, {r0-r12, lr} ; save register file and ret address
      MRS   r4, CPSR
      STMFD sp!, {r4}         ; save current PSR
      MRS   r4, SPSR          ; YYY+
      STMFD sp!, {r4}         ; YYY+ save SPSR

      ; OSPrioCur = OSPrioHighRdy
      LDR   r4, addr_OSPrioCur
      LDR   r5, addr_OSPrioHighRdy
      LDRB  r6, [r5]
      STRB  r6, [r4]

      ; Get current task TCB address
      LDR   r4, addr_OSTCBCur
      LDR   r5, [r4]
      STR   sp, [r5]          ; store sp in preempted tasks's TCB

      ; Get highest priority task TCB address
      LDR   r6, addr_OSTCBHighRdy
      LDR   r6, [r6]
      LDR   sp, [r6]          ; get new task's stack pointer

      ; OSTCBCur = OSTCBHighRdy
      STR   r6, [r4]          ; set new current task TCB address

      LDMFD sp!, {r4}         ; YYY+
      LDMFD sp!, {r5}         ; YYY+
      MSR   CPSR_cxsf, r5           ; YYY+ Switch to task's context
      MSR   SPSR_cxsf, r4           ; YYY+ Restore task's SPSR
      LDMFD sp!, {r0-r12, lr} ; YYY+
 IF :DEF: THUMB_AWARE
      BX    lr
 ELSE
      MOV   pc,lr
 ENDIF


;     void OSStartHighRdy(void)
;
;     Start the task with the highest priority;
;
      EXPORT      OSStartHighRdy
OSStartHighRdy
      ; Get current task TCB address
      LDR   r4, addr_OSTCBCur
      ; Get highest priority task TCB address
      LDR   r5, addr_OSTCBHighRdy
      LDR   r5, [r5]          ; get stack pointer
      LDR   sp, [r5]          ; switch to the new stack

      STR   r5, [r4]          ; set new current task TCB address

      LDMFD sp!, {r4}         ; YYY
      LDMFD sp!, {r4}         ; get new state from top of the stack
```

```
     MSR   CPSR_c, r4        ; CPSR should be SVC32Mode
     LDMFD sp!, {r0-r12, lr} ; start the new task
 IF :DEF: THUMB_AWARE
     BX    lr
 ELSE
     MOV   pc,lr
 ENDIF

     END
```

os_cpu_c.c

```
#define  OS_CPU_GLOBALS
#include "includes.h"

#ifndef BUILT_FOR
#define BUILT_FOR "an UNKNOWN board"
#endif

#define UCOS_II_BANNER BUILT_FOR
static const char ucosii_banner[] = UCOS_II_BANNER;

/*    INITIALIZE A TASK'S STACK
  Description: This function is called by either OSTaskCreate() or
      OSTaskCreateExt() to initialize the stack frame of the task being created.
      This function is highly processor specific.
  Arguments:
     task is a pointer to the task code
     pdata is a pointer to a user supplied data area that will be passed to the
      task when the task first executes.
     ptos is a pointer to the top of stack.  It is assumed that 'ptos' points to
      a 'free' entry on the task stack.  If OS_STK_GROWTH is set to 1 then
      'ptos' will contain the HIGHEST valid address of the stack.  Similarly, if
      OS_STK_GROWTH is set to 0, the 'ptos' will contains the LOWEST valid address
      of the stack.
     opt specifies options that can be used to alter the behavior of
      OSTaskStkInit(). (see uCOS_II.H for OS_TASK_OPT_???).
  Returns: Always returns the location of the new top-of-stack' once the processor
      registers have been placed on the stack in the proper order.
  Note(s): Interrupts are enabled when your task starts executing. You can change
     this by setting the PSW to 0x0002 instead.  In this case, interrupts would be
     disabled upon task startup.  The application code would be responsible for
     enabling interrupts at the beginning of the task code.  You will need to
     modify OSTaskIdle() and OSTaskStat() so that they enable interrupts.  Failure
     to do this will make your system crash!
*/
void *OSTaskStkInit (void (*task)(void *pd), void *pdata, void *ptos, INT16U opt)
{
    unsigned int *stk ;

    opt    = opt;            //'opt' is not used, prevent warning
    stk    = (unsigned int *)ptos;      //Load stack pointer

    /* build a context for the new task */
    *--stk = (unsigned int) task;       /* lr */
    *--stk = 0;                         /* r12 */
    *--stk = 0;                         /* r11 */
    *--stk = 0;                         /* r10 */
    *--stk = 0;                         /* r9 */
    *--stk = 0;                         /* r8 */
    *--stk = 0;                         /* r7 */
    *--stk = 0;                         /* r6 */
    *--stk = 0;                         /* r5 */
```

```
    *--stk = 0;                          /* r4 */
    *--stk = 0;                          /* r3 */
    *--stk = 0;                          /* r2 */
    *--stk = 0;                          /* r1 */
    *--stk = (unsigned int) pdata;       /* r0 */
    *--stk = SVC32MODE;                  /* spsr YYY+ */
    *--stk = SVC32MODE;                  /* psr */
    return ((void *)stk);
}

/*$PAGE*/
#if OS_CPU_HOOKS_EN
/*       TASK CREATION HOOK
  Description: This function is called when a task is created.
  Arguments:
      ptcb is a pointer to the task control block of the task being created.
  Note(s):
    1) Interrupts are disabled during this call.
*/
void OSTaskCreateHook (OS_TCB *ptcb)
{
    ptcb = ptcb;                        //Prevent compiler warning
}

/*        TASK DELETION HOOK
  Description: This function is called when a task is deleted.
  Arguments:
     ptcb is a pointer to the task control block of the task being deleted.
  Note(s):
   1) Interrupts are disabled during this call.
*/
void OSTaskDelHook (OS_TCB *ptcb)
{
    ptcb = ptcb;                //Prevent compiler warning
}

/*       TASK SWITCH HOOK
*
  Description: This function is called when a task switch is performed.  This
      allows you to perform other operations during a context switch.
  Arguments: none
  Note(s): 1) Interrupts are disabled during this call.
     2) It is assumed that the global pointer 'OSTCBHighRdy' points to the TCB of
     the task that will be 'switched in' (i.e. the highest priority task) and,
     'OSTCBCur' points to the task being switched out (i.e. the preempted task).
*/
void OSTaskSwHook (void)
{
}

/*       STATISTIC TASK HOOK
  Description: This function is called every second by uC/OS-II's statistics task.
    This allows your application to add functionality to the statistics task.
  Arguments: none
*/
void OSTaskStatHook (void)
{
}

/*      TICK HOOK
  Description: This function is called every tick.
  Arguments: none
  Note(s): 1) Interrupts may or may not be ENABLED during this call.
```

```
*/
void OSTimeTickHook (void)
{
}
#endif

#define BUILD_DATE "Date: " __DATE__ "\n"

/*
 * Initialize an ARM Target board
 */
void
 ARMTargetInit(void)
{
    /* ---- Tell the world who we are ---------------------------------- */
      uHALr_InitHeap();
    uHALr_printf("\n\n\nuCOS-II Running on ") ;
#if defined(EBSA285)
    uHALr_printf("n EBSA-285 (21285 evaluation board)\n") ;
#elif defined(BRUTUS)
    uHALr_printf(" Brutus (SA-1100 verification platform)\n") ;
#else
    uHALr_printf("%s\n",ucosii_banner) ;
#endif
    uHALr_printf(uHAL_VERSION_STRING);
    uHALr_printf("\n") ;
    uHALr_printf(BUILD_DATE);
    uHALr_printf("\n") ;

#ifdef DEBUG
    uHALr_printf("Initialising target\n");
#endif

    /* ---- disable the MMU        ------------------------------------- */
    uHALr_ResetMMU();

    /* ---- disable interrupts (IRQs) ---------------------------------- */
    ARMDisableInt();

    /* ---- soft vectors ----------------------------------------------- */
#ifdef DEBUG
    uHALr_printf("Setting up soft vectors\n");
#endif
    /* Define pre & post-process routines for Interrupt */
    uHALir_DefineIRQ(IrqStart, IrqFinish, (PrVoid) 0);
    uHALr_InitInterrupts();

#ifdef DEBUG
    uHALr_printf("Timer init\n");
#endif
    uHALr_InitTimers();

#ifdef DEBUG
    uHALr_printf("targetInit() complete\n");
#endif
}                               /* targetInit */

/* start the ARM target running */
void
 ARMTargetStart(void)
{
#ifdef DEBUG
    uHALr_printf("Starting target\n") ;
```

```
#endif

    /* request the system timer */
    if (uHALr_RequestSystemTimer(
                        (PrHandler) OSTimeTick,
                        (const unsigned char *)"uCOS-II") <= 0)
        uHALr_printf("Timer/IRQ busy\n");

    /* Start system timer & enable the interrupt. */
    uHALr_InstallSystemTimer();
}

extern void IRQContextSwap(void);  //post DispatchIRQ processing (the _real_ one)

/* just 'notice' that we need to change context */
static need_to_swap_context = 0 ;
void OSIntCtxSw(void) {
    need_to_swap_context = 1;
}

/* This is what uCOS does at the start of an IRQ */
void IrqStart(void)
{
    /* increment nesting counter */
    OSIntNesting++;
}

/* This is what uCOS does at the end of an IRQ */
PrVoid IrqFinish(void)
{
    OSIntExit() ;
    if (need_to_swap_context) {
        need_to_swap_context = 0 ;
        return (IRQContextSwap) ;
    } else {
        return NULL ;
    }
}
```

os_mbox.c

```
#ifndef  OS_MASTER_FILE
#include "includes.h"
#endif

#if OS_MBOX_EN
/*     ACCEPT MESSAGE FROM MAILBOX
  Description: This function checks the mailbox to see if a message is available.
      Unlike OSMboxPend(), OSMboxAccept() does not suspend the calling task if a
      message is not available.
  Arguments:
      pevent is a pointer to the event control block
  Returns:
      != (void *)0  is the message in the mailbox if one is available.  The
      mailbox is cleared so the next time OSMboxAccept() is called, the mailbox
      will be empty.
       == (void *)0  if the mailbox is empty or if you didn't pass the proper
      event pointer.
*/
void *OSMboxAccept (OS_EVENT *pevent)
{
    void  *msg;
```

```
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) { //Validate event block type
        OS_EXIT_CRITICAL();
        return ((void *)0);
    }
    msg = pevent->OSEventPtr;
    if (msg != (void *)0) {              //See if there is already a message
        pevent->OSEventPtr = (void *)0;  //Clear the mailbox
    }
    OS_EXIT_CRITICAL();
    return (msg);                        //Return the message received (or NULL)
}
/*$PAGE*/
/*     CREATE A MESSAGE MAILBOX
  Description: This function creates a message mailbox if free event control
       blocks are available.
  Arguments:
     msg is a pointer to a message that you wish to deposit in the mailbox.  If
      you set this value to the NULL pointer (i.e. (void *)0) then the mailbox
      will be considered empty.
  Returns:
    != (void *)0  is a pointer to the event control clock (OS_EVENT) associated
    with the created mailbox
    == (void *)0  if no event control blocks were available
*/
OS_EVENT *OSMboxCreate (void *msg)
{
    OS_EVENT *pevent;

    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;           //Get next free event control block
    if (OSEventFreeList != (OS_EVENT *)0) {  //See if pool of free ECB pool was
                                             //empty
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {
        pevent->OSEventType = OS_EVENT_TYPE_MBOX;
        pevent->OSEventPtr  = msg;      //Deposit message in event control block
        OSEventWaitListInit(pevent);
    }
    return (pevent);                     //Return pointer to event control block
}
/*$PAGE*/
/*      PEND ON MAILBOX FOR A MESSAGE
  Description: This function waits for a message to be sent to a mailbox
  Arguments:
     pevent is a pointer to the event control block associated with the desired
      mailbox
     timeout is an optional timeout period (in clock ticks).  If non-zero, your
      task will wait for a message to arrive at the mailbox up to the amount of
      time specified by this argument.  If you specify 0, however, your task will
      wait forever at the specified mailbox or, until a message arrives.
     err is a pointer to where an error message will be deposited.  Possible error
      messages are:
       OS_NO_ERR         The call was successful and your task received a message.
       OS_TIMEOUT        A message was not received within the specified timeout
       OS_ERR_EVENT_TYPE   Invalid event type
       OS_ERR_PEND_ISR     If you called this function from an ISR and the
                           result would lead to a suspension.
  Returns:
     != (void *)0  is a pointer to the message received
```

```
     == (void *)0  if no message was received or you didn't pass the proper
        pointer to the event control block.
*/
void *OSMboxPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
    void  *msg;


    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {  //Validate event block type
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
    msg = pevent->OSEventPtr;
    if (msg != (void *)0) {               //See if there is already a message
        pevent->OSEventPtr = (void *)0;   //Clear the mailbox
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
    } else if (OSIntNesting > 0) {        //See if called from ISR ...
        OS_EXIT_CRITICAL();               //... can't PEND from an ISR
        *err = OS_ERR_PEND_ISR;
    } else {
        OSTCBCur->OSTCBStat |= OS_STAT_MBOX;   //Message not available, task will
                                               //pend
        OSTCBCur->OSTCBDly   = timeout;        //Load timeout in TCB
        OSEventTaskWait(pevent);               //Suspend task until event or
                                               //timeout occurs
        OS_EXIT_CRITICAL();
        OSSched();                  //Find next highest priority task ready to run
        OS_ENTER_CRITICAL();
        if ((msg = OSTCBCur->OSTCBMsg) != (void *)0) {  //See if we were given the
                                                        //message
            OSTCBCur->OSTCBMsg      = (void *)0;     //Yes, clear message received
            OSTCBCur->OSTCBStat     = OS_STAT_RDY;
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0; //No longer waiting for event
            OS_EXIT_CRITICAL();
            *err                    = OS_NO_ERR;
        } else if (OSTCBCur->OSTCBStat & OS_STAT_MBOX) {  //If status is not
                                                          //OS_STAT_RDY, timed out
            OSEventTO(pevent);                            //Make task ready
            OS_EXIT_CRITICAL();
            msg                 = (void *)0;    //Set message contents to NULL
            *err                = OS_TIMEOUT;   //Indicate that a timeout occured
        } else {
            msg                     = pevent->OSEventPtr;  //Message received
            pevent->OSEventPtr      = (void *)0;           //Clear the mailbox
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
            OS_EXIT_CRITICAL();
            *err                    = OS_NO_ERR;
        }
    }
    return (msg);                          //Return the message received (or NULL)
}
/*$PAGE*/
/*     POST MESSAGE TO A MAILBOX
  Description: This function sends a message to a mailbox
  Arguments:
      pevent is a pointer to the event control block associated with the desired
       mailbox
      msg is a pointer to the message to send. You MUST NOT send a NULL pointer.
  Returns:
     OS_NO_ERR          The call was successful and the message was sent
```

```
     OS_MBOX_FULL       If the mailbox already contains a message.  You can can
                        only send one message at a time and thus, the message MUST
                        be consumed before you are allowed to send another one.
     OS_ERR_EVENT_TYPE  If you are attempting to post to a non mailbox.
*/
INT8U OSMboxPost (OS_EVENT *pevent, void *msg)
{
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {  //Validate event block type
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {                 //See if any task pending on mailbox
        OSEventTaskRdy(pevent, msg, OS_STAT_MBOX);  //Ready highest priority task
                                                    //waiting on event
        OS_EXIT_CRITICAL();
        OSSched();                      //Find highest priority task ready to run
        return (OS_NO_ERR);
    } else {
        if (pevent->OSEventPtr != (void *)0) {      //Make sure mailbox doesn't
                                                    //already have a msg  */
            OS_EXIT_CRITICAL();
            return (OS_MBOX_FULL);
        } else {
            pevent->OSEventPtr = msg;               //Place message in mailbox
            OS_EXIT_CRITICAL();
            return (OS_NO_ERR);
        }
    }
}
/*$PAGE*/
/*      QUERY A MESSAGE MAILBOX
  Description: This function obtains information about a message mailbox.
  Arguments:
    pevent is a pointer to the event control block associated with the desired
           mailbox
    pdata  is a pointer to a structure that will contain information about the
           message mailbox.
  Returns:
     OS_NO_ERR          The call was successful and the message was sent
     OS_ERR_EVENT_TYPE  If you are attempting to obtain data from a non mailbox.
*/
INT8U OSMboxQuery (OS_EVENT *pevent, OS_MBOX_DATA *pdata)
{
    INT8U  i;
    INT8U *psrc;
    INT8U *pdest;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_MBOX) {  //Validate event block type
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pdata->OSEventGrp = pevent->OSEventGrp;       //Copy message mailbox wait list
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
        *pdest++ = *psrc++;
    }
    pdata->OSMsg = pevent->OSEventPtr;              //Get message from mailbox
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

```
#endif
```

os_mem.c

```
#ifndef  OS_MASTER_FILE
#include "includes.h"
#endif

#if OS_MEM_EN && OS_MAX_MEM_PART >= 2
/*   LOCAL GLOBAL VARIABLES */

static  OS_MEM   *OSMemFreeList;   //Pointer to free list of memory partitions
static  OS_MEM   OSMemTbl[OS_MAX_MEM_PART]; //Storage for memory partition manager
/*$PAGE*/
/*      CREATE A MEMORY PARTITION
  Description : Create a fixed-sized memory partition that will be managed by
                uC/OS-II.
  Arguments:
     addr is the starting address of the memory partition
     nblks is the number of memory blocks to create from the partition.
     blksize is the size (in bytes) of each block in the memory partition.
     err is a pointer to a variable containing an error message which will be set
      by this function to either:
    OS_NO_ERR            if the memory partition has been created correctly.
    OS_MEM_INVALID_PART  no free partitions available
    OS_MEM_INVALID_BLKS  user specified an invalid number of blocks (must be >= 2)
    OS_MEM_INVALID_SIZE  user specified an invalid block size (must be greater
                         than the size of a pointer)
  Returns:
    != (OS_MEM *)0  is the partition was created
    == (OS_MEM *)0  if the partition was not created because of invalid arguments
                    or, no free partition is available.
*/
OS_MEM *OSMemCreate (void *addr, INT32U nblks, INT32U blksize, INT8U *err)
{
    OS_MEM  *pmem;
    INT8U   *pblk;
    void   **plink;
    INT32U   i;

    if (nblks < 2) {                  //Must have at least 2 blocks per partition
        *err = OS_MEM_INVALID_BLKS;
        return ((OS_MEM *)0);
    }
    if (blksize < sizeof(void *)) {    //Must contain space for at least a pointer
        *err = OS_MEM_INVALID_SIZE;
        return ((OS_MEM *)0);
    }
    OS_ENTER_CRITICAL();
    pmem = OSMemFreeList;              //Get next free memory partition
    if (OSMemFreeList != (OS_MEM *)0) { //See if pool of free partitions was empty
        OSMemFreeList = (OS_MEM *)OSMemFreeList->OSMemFreeList;
    }
    OS_EXIT_CRITICAL();
    if (pmem == (OS_MEM *)0) {               //See if we have a memory partition
        *err = OS_MEM_INVALID_PART;
        return ((OS_MEM *)0);
    }
    plink = (void **)addr;              //Create linked list of free memory blocks
    pblk  = (INT8U *)addr + blksize;
    for (i = 0; i < (nblks - 1); i++) {
        *plink = (void *)pblk;
        plink  = (void **)pblk;
```

```
        pblk   = pblk + blksize;
    }
    *plink = (void *)0;                 //Last memory block points to NULL
    OS_ENTER_CRITICAL();
    pmem->OSMemAddr     = addr;         //Store start address of memory partition
    pmem->OSMemFreeList = addr;        //Initialize pointer to pool of free blocks
    pmem->OSMemNFree    = nblks;       //Store number of free blocks in MCB
    pmem->OSMemNBlks    = nblks;
    pmem->OSMemBlkSize  = blksize;     //Store block size of each memory blocks
    OS_EXIT_CRITICAL();
    *err   = OS_NO_ERR;
    return (pmem);
}
/*$PAGE*/
/*       GET A MEMORY BLOCK
  Description: Get a memory block from a partition
  Arguments:
     pmem is a pointer to the memory partition control block
     err  is a pointer to a variable containing an error message which will be set
          by this function to either:
   OS_NO_ERR           if the memory partition has been created correctly.
   OS_MEM_NO_FREE_BLKS if there are no more free memory blocks to allocate to
                       caller
  Returns: A pointer to a memory block if no error is detected
               A pointer to NULL if an error is detected
*/
void *OSMemGet (OS_MEM *pmem, INT8U *err)
{
    void    *pblk;


    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree > 0) {         //See if there are any free memory blocks
        pblk                = pmem->OSMemFreeList;    //Yes, point to next free
                                                      //memory block
        pmem->OSMemFreeList = *(void **)pblk;   //Adjust pointer to new free list
        pmem->OSMemNFree--;             //One less memory block in this partition
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;               //No error
        return (pblk);                  //Return memory block to caller
    } else {
        OS_EXIT_CRITICAL();
        *err = OS_MEM_NO_FREE_BLKS;  //No, Notify caller of empty memory partition
        return ((void *)0);          //Return NULL pointer to caller
    }
}
/*$PAGE*/
/*      INITIALIZE MEMORY PARTITION MANAGER
  Description: This function is called by uC/OS-II to initialize the memory parti-
tion manager. Your application MUST NOT call this function.
  Arguments: none
  Returns: none
*/
void OSMemInit (void)
{
    OS_MEM  *pmem;
    INT16U   i;

    pmem = (OS_MEM *)&OSMemTbl[0];          //Point to memory control block (MCB)
    for (i = 0; i < (OS_MAX_MEM_PART - 1); i++) {  //Init. list of free memory
                                                   //partitions
        pmem->OSMemFreeList=(void *)&OSMemTbl[i+1]; //Chain list of free
                                                    //partitions
```

```
        pmem->OSMemAddr    =(void *)0;   //Store start address of memory partition
        pmem->OSMemNFree    = 0;                      //No free blocks
        pmem->OSMemNBlks    = 0;                      //No blocks
        pmem->OSMemBlkSize  = 0;                      //Zero size
        pmem++;
    }
    OSMemTbl[OS_MAX_MEM_PART - 1].OSMemFreeList = (void *)0;
    OSMemFreeList                               = (OS_MEM *)&OSMemTbl[0];
}
/*$PAGE*/
/*    RELEASE A MEMORY BLOCK
  Description: Returns a memory block to a partition
  Arguments:
     pmem    is a pointer to the memory partition control block
     pblk    is a pointer to the memory block being released.
  Returns:
   OS_NO_ERR    if the memory block was inserted into the partition
   OS_MEM_FULL  if you are returning a memory block to an already FULL memory
                partition (You freed more blocks than you allocated!)
*/
INT8U OSMemPut (OS_MEM  *pmem, void *pblk)
{
    OS_ENTER_CRITICAL();
    if (pmem->OSMemNFree >= pmem->OSMemNBlks) {  //Make sure all blocks not
                                                 //already returned
        OS_EXIT_CRITICAL();
        return (OS_MEM_FULL);
    }
    *(void **)pblk    = pmem->OSMemFreeList;   //Insert released block into free
                                               //block list
    pmem->OSMemFreeList = pblk;
    pmem->OSMemNFree++;                  //One more memory block in this partition
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);             //Notify caller that memory block was released
}
/*$PAGE*/
/*      QUERY MEMORY PARTITION
  Description : This function is used to determine the number of free memory
blocks and the number of used memory blocks from a memory partition.
  Arguments:
    pmem    is a pointer to the memory partition control block
    pdata   is a pointer to a structure that will contain information about the
            memory partition.
  Returns:
     OS_NO_ERR         Always returns no error.
*/
INT8U OSMemQuery (OS_MEM *pmem, OS_MEM_DATA *pdata)
{
    OS_ENTER_CRITICAL();
    pdata->OSAddr     = pmem->OSMemAddr;
    pdata->OSFreeList = pmem->OSMemFreeList;
    pdata->OSBlkSize  = pmem->OSMemBlkSize;
    pdata->OSNBlks    = pmem->OSMemNBlks;
    pdata->OSNFree    = pmem->OSMemNFree;
    OS_EXIT_CRITICAL();
    pdata->OSNUsed    = pdata->OSNBlks - pdata->OSNFree;
    return (OS_NO_ERR);
}
#endif
```

os_q.c

```
#ifndef  OS_MASTER_FILE
```

```
#include "includes.h"
#endif

#if OS_Q_EN && (OS_MAX_QS >= 2)
/*       LOCAL DATA TYPES  */

typedef struct os_q {           //QUEUE CONTROL BLOCK
    struct os_q   *OSQPtr;      //Link to next queue control block in list of free
                                //blocks
    void         **OSQStart;    //Pointer to start of queue data
    void         **OSQEnd;      //Pointer to end   of queue data
    void         **OSQIn;       //Pointer to where next message will be inserted
                                //in   the Q
    void         **OSQOut;      //Pointer to where next message will be extracted
                                //from the Q
    INT16U         OSQSize;     //Size of queue (maximum number of entries)
    INT16U         OSQEntries;  //Current number of entries in the queue
} OS_Q;

/*      LOCAL GLOBAL VARIABLES  */
static  OS_Q        *OSQFreeList;   //Pointer to list of free QUEUE control blocks
static  OS_Q         OSQTbl[OS_MAX_QS];     //Table of QUEUE control blocks

/*$PAGE*/
/*     ACCEPT MESSAGE FROM QUEUE
  Description: This function checks the queue to see if a message is available.
  Unlike OSQPend(), OSQAccept() does not suspend the calling task if a message is
  not available.
  Arguments:
    pevent        is a pointer to the event control block
  Returns:
     != (void *)0  is the message in the queue if one is available.  The message
        is removed from the so the next time OSQAccept() is called, the queue will
        contain one less entry.
     == (void *)0  if the queue is empty if you passed an invalid event type
*/
void *OSQAccept (OS_EVENT *pevent)
{
    void  *msg;
    OS_Q  *pq;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {     // Validate event block type
        OS_EXIT_CRITICAL();
        return ((void *)0);
    }
    pq = pevent->OSEventPtr;                 //Point at queue control block
    if (pq->OSQEntries != 0) {               //See if any messages in the queue
        msg = *pq->OSQOut++;          //Yes, extract oldest message from the queue
        pq->OSQEntries--;             //Update the number of entries in the queue
        if (pq->OSQOut == pq->OSQEnd) {   //Wrap OUT pointer if we are at the end
                                          //of the queue
            pq->OSQOut = pq->OSQStart;
        }
    } else {
        msg = (void *)0;                  //Queue is empty
    }
    OS_EXIT_CRITICAL();
    return (msg);                         //Return message received (or NULL)
}
/*$PAGE*/
/*     CREATE A MESSAGE QUEUE
```

```
  Description: This function creates a message queue if free event control blocks
are available.
  Arguments:
    start  is a pointer to the base address of the message queue storage area.The
     storage area MUST be declared as an array of pointers to 'void' as follows
     void *MessageStorage[size]
    size   is the number of elements in the storage area
  Returns:
    != (void *)0  is a pointer to the event control clock (OS_EVENT) associated
       with the created queue
    == (void *)0  if no event control blocks were available
*/
OS_EVENT *OSQCreate (void **start, INT16U size)
{
    OS_EVENT *pevent;
    OS_Q     *pq;

    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;              //Get next free event control block
    if (OSEventFreeList != (OS_EVENT *)0) {  //See if pool of free ECB pool was
                                             //empty
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {      //See if we have an event control block
        OS_ENTER_CRITICAL();            //Get a free queue control block
        pq = OSQFreeList;
        if (OSQFreeList != (OS_Q *)0) {
            OSQFreeList = OSQFreeList->OSQPtr;
        }
        OS_EXIT_CRITICAL();
        if (pq != (OS_Q *)0) {        //See if we were able to get a queue control
                                      //block
            pq->OSQStart        = start;         //Yes, initialize the queue
            pq->OSQEnd          = &start[size];
            pq->OSQIn           = start;
            pq->OSQOut          = start;
            pq->OSQSize         = size;
            pq->OSQEntries      = 0;
            pevent->OSEventType = OS_EVENT_TYPE_Q;
            pevent->OSEventPtr  = pq;
            OSEventWaitListInit(pevent);
        } else {               //No,  since we couldn't get a queue control block
            OS_ENTER_CRITICAL();          //Return event control block on error
            pevent->OSEventPtr = (void *)OSEventFreeList;
            OSEventFreeList    = pevent;
            OS_EXIT_CRITICAL();
            pevent = (OS_EVENT *)0;
        }
    }
    return (pevent);
}
/*$PAGE*/
/*        FLUSH QUEUE
  Description : This function is used to flush the contents of the message queue.
  Arguments: none
  Returns:
    OS_NO_ERR          upon success
    OS_ERR_EVENT_TYPE  If you didn't pass a pointer to a queue
*/
INT8U OSQFlush (OS_EVENT *pevent)
{
    OS_Q  *pq;
```

```
OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {     //Validate event block type
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pq             = pevent->OSEventPtr;       //Point to queue storage structure
    pq->OSQIn      = pq->OSQStart;
    pq->OSQOut     = pq->OSQStart;
    pq->OSQEntries = 0;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}

/*$PAGE*/
/*       QUEUE MODULE INITIALIZATION
  Description : This function is called by uC/OS-II to initialize the message
queue module. Your application MUST NOT call this function.
  Arguments:  none
  Returns: none
*/
void OSQInit (void)
{
    INT16U i;

    for (i = 0; i < (OS_MAX_QS - 1); i++) {      //Init. list of free QUEUE
                                                 //control blocks
        OSQTbl[i].OSQPtr = &OSQTbl[i+1];
    }
    OSQTbl[OS_MAX_QS - 1].OSQPtr = (OS_Q *)0;
    OSQFreeList                  = &OSQTbl[0];
}

/*$PAGE*/
/*      PEND ON A QUEUE FOR A MESSAGE
  Description: This function waits for a message to be sent to a queue
  Arguments:
    pevent is a pointer to the event control block associated with the
     desired queue
    timeout is an optional timeout period (in clock ticks).  If non-zero, your
     task will wait for a message to arrive at the queue up to the amount of time
     specified by this argument.  If you specify 0, however, your task will wait
     forever at the specified queue or, until a message arrives.
    err is a pointer to where an error message will be deposited.  Possible error
     messages are:
     OS_NO_ERR         The call was successful and your task received a message.
     OS_TIMEOUT        A message was not received within the specified timeout
     OS_ERR_EVENT_TYPE You didn't pass a pointer to a queue
     OS_ERR_PEND_ISR    If you called this function from an ISR and the result
                        would lead to a suspension.
  Returns:
    != (void *)0  is a pointer to the message received
    == (void *)0  if no message was received or you didn't pass a pointer to a
                  queue.
*/
void *OSQPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
    void  *msg;
    OS_Q  *pq;


    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {//Validate event block type
```

```
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
        return ((void *)0);
    }
    pq = pevent->OSEventPtr;                    //Point at queue control block
    if (pq->OSQEntries != 0) {                 //See if any messages in the queue
        msg = *pq->OSQOut++;          //Yes, extract oldest message from the queue
        pq->OSQEntries--;             //Update the number of entries in the queue
        if (pq->OSQOut == pq->OSQEnd) {  //Wrap OUT pointer if we are at the end
                                         // of the queue
            pq->OSQOut = pq->OSQStart;
        }
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
    } else if (OSIntNesting > 0) {              //See if called from ISR ...
        OS_EXIT_CRITICAL();                     //... can't PEND from an ISR
        *err = OS_ERR_PEND_ISR;
    } else {
        OSTCBCur->OSTCBStat    |= OS_STAT_Q;    //Task will have to pend for a
                                                //message to be posted
        OSTCBCur->OSTCBDly      = timeout;      //Load timeout into TCB
        OSEventTaskWait(pevent);                //Suspend task until event or
                                                //timeout occurs
        OS_EXIT_CRITICAL();
        OSSched();                              //Find next highest priority task
                                                //ready to run
        OS_ENTER_CRITICAL();
        if ((msg = OSTCBCur->OSTCBMsg) != (void *)0) {//Did we get a message?
            OSTCBCur->OSTCBMsg      = (void *)0;      //Extract message from TCB
                                                      //(Put there by QPost)
            OSTCBCur->OSTCBStat     = OS_STAT_RDY;
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;  //No longer waiting for
                                                      //event
            OS_EXIT_CRITICAL();
            *err                    = OS_NO_ERR;
        } else if (OSTCBCur->OSTCBStat & OS_STAT_Q) { //Timed out if status
                                                      //indicates pending on Q
            OSEventTO(pevent);
            OS_EXIT_CRITICAL();
            msg                     = (void *)0;      //No message received
            *err                    = OS_TIMEOUT;     //Indicate a timeout occured
        } else {
            msg = *pq->OSQOut++;                      //Extract message from queue
            pq->OSQEntries--;                         //Update the number of
                                                      //entries in the queue
            if (pq->OSQOut == pq->OSQEnd) {           //Wrap OUT pointer if we are
                                                      //at the end of Q
                pq->OSQOut = pq->OSQStart;
            }
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
            OS_EXIT_CRITICAL();
            *err = OS_NO_ERR;
        }
    }
    return (msg);                              //Return message received (or NULL)
}
/*$PAGE*/
/*       POST MESSAGE TO A QUEUE
  Description: This function sends a message to a queue
  Arguments:
   pevent is a pointer to the event control block associated with the desired
    queue
   msg is a pointer to the message to send.  You MUST NOT send a NULL pointer.
```

```
  Returns:
   OS_NO_ERR      The call was successful and the message was sent
   OS_Q_FULL      If the queue cannot accept any more messages because it is full.
   OS_ERR_EVENT_TYPE  If you didn't pass a pointer to a queue.
*/
INT8U OSQPost (OS_EVENT *pevent, void *msg)
{
    OS_Q   *pq;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {     //Validate event block type
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {                   //See if any task pending on queue
        OSEventTaskRdy(pevent, msg, OS_STAT_Q);  //Ready highest priority task
                                                 //waiting on event
        OS_EXIT_CRITICAL();
        OSSched();                       //Find highest priority task ready to run
        return (OS_NO_ERR);
    } else {
        pq = pevent->OSEventPtr;         //Point to queue control block
        if (pq->OSQEntries >= pq->OSQSize) {       //Make sure queue is not full
            OS_EXIT_CRITICAL();
            return (OS_Q_FULL);
        } else {
            *pq->OSQIn++ = msg;           //Insert message into queue
            pq->OSQEntries++;             //Update the nbr of entries in the queue
            if (pq->OSQIn == pq->OSQEnd) { //Wrap IN ptr if we are at end of queue
                pq->OSQIn = pq->OSQStart;
            }
            OS_EXIT_CRITICAL();
        }
        return (OS_NO_ERR);
    }
}
/*$PAGE*/
/*     POST MESSAGE TO THE FRONT OF A QUEUE
  Description: This function sends a message to a queue but unlike OSQPost(), the
message is posted at the front instead of the end of the queue.  Using
 OSQPostFront() allows you to send'priority' messages.
  Arguments:
    pevent is a pointer to the event control block associated with the desired
           queue
    msg is a pointer to the message to send.  You MUST NOT send a NULL pointer.
  Returns:
   OS_NO_ERR     The call was successful and the message was sent
   OS_Q_FULL     If the queue cannot accept any more messages because it is full.
   OS_ERR_EVENT_TYPE  If you didn't pass a pointer to a queue.
*/
INT8U OSQPostFront (OS_EVENT *pevent, void *msg)
{
    OS_Q   *pq;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {     //Validate event block type
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {                   //See if any task pending on queue
        OSEventTaskRdy(pevent, msg, OS_STAT_Q);       //Ready highest priority
                                                      //task waiting on event
        OS_EXIT_CRITICAL();
```

```
        OSSched();                          //Find highest priority task ready to run
        return (OS_NO_ERR);
    } else {
        pq = pevent->OSEventPtr;                //Point to queue control block
        if (pq->OSQEntries >= pq->OSQSize) {    //Make sure queue is not full
            OS_EXIT_CRITICAL();
            return (OS_Q_FULL);
        } else {
            if (pq->OSQOut == pq->OSQStart) {    //Wrap OUT ptr if we are at
                                                 //the 1st queue entry */
                pq->OSQOut = pq->OSQEnd;
            }
            pq->OSQOut--;
            *pq->OSQOut = msg;            //Insert message into queue
            pq->OSQEntries++;             //Update the nbr of entries in the queue
            OS_EXIT_CRITICAL();
        }
        return (OS_NO_ERR);
    }
}
/*$PAGE*/
/*    QUERY A MESSAGE QUEUE
  Description: This function obtains information about a message queue.
  Arguments:
    pevent is a pointer to the event control block associated with the desired
           mailbox
    pdata  is a pointer to a structure that will contain information about the
           message queue.
  Returns:
    OS_NO_ERR          The call was successful and the message was sent
    OS_ERR_EVENT_TYPE  If you are attempting to obtain data from a non queue.
*/
INT8U OSQQuery (OS_EVENT *pevent, OS_Q_DATA *pdata)
{
    OS_Q   *pq;
    INT8U   i;
    INT8U  *psrc;
    INT8U  *pdest;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_Q) {     //Validate event block type
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pdata->OSEventGrp = pevent->OSEventGrp;      //Copy message mailbox wait list
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
        *pdest++ = *psrc++;
    }
    pq = (OS_Q *)pevent->OSEventPtr;
    if (pq->OSQEntries > 0) {
        pdata->OSMsg = pq->OSQOut;      //Get next message to return if available
    } else {
        pdata->OSMsg = (void *)0;
    }
    pdata->OSNMsgs = pq->OSQEntries;
    pdata->OSQSize = pq->OSQSize;
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
#endif
```

os_sem.c

```
#ifndef  OS_MASTER_FILE
#include "includes.h"
#endif

#if OS_SEM_EN
/*   ACCEPT SEMAPHORE
  Description: This function checks the semaphore to see if a resource is
  available or, if an event occurred.  Unlike OSSemPend(), OSSemAccept() does not
  suspend the calling task if the resource is not available or the event did not
  occur.
  Arguments:
    pevent  is a pointer to the event control block
  Returns:
   >  0  if the resource is available or the event did not occur the semaphore is
          decremented to obtain the resource.
   == 0  if the resource is not available or the event did not occur or,
          you didn't pass a pointer to a semaphore
*/
INT16U OSSemAccept (OS_EVENT *pevent)
{
    INT16U cnt;

    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {   //Validate event block type
        OS_EXIT_CRITICAL();
        return (0);
    }
    cnt = pevent->OSEventCnt;
    if (cnt > 0) {                            //See if resource is available
        pevent->OSEventCnt--;                 //Yes, decrement semaphore and
                                              //notify caller
    }
    OS_EXIT_CRITICAL();
    return (cnt);                                     //Return semaphore count
}

/*$PAGE*/
/*   CREATE A SEMAPHORE
  Description: This function creates a semaphore.
  Arguments:
    cnt is the initial value for the semaphore.  If the value is 0, no resource is
     available (or no event has occurred).  You initialize the semaphore to a
     non-zero value to specify how many resources are available (e.g. if you have
     10 resources, you would initialize the semaphore to 10).
  Returns:
   != (void *)0  is a pointer to the event control clock (OS_EVENT)  associated
      with the created semaphore
   == (void *)0  if no event control blocks were available
*/
OS_EVENT *OSSemCreate (INT16U cnt)
{
    OS_EVENT *pevent;

    OS_ENTER_CRITICAL();
    pevent = OSEventFreeList;                 //Get next free event control block
    if (OSEventFreeList != (OS_EVENT *)0) {   //See if pool of free ECB pool was
                                              //empty
        OSEventFreeList = (OS_EVENT *)OSEventFreeList->OSEventPtr;
    }
    OS_EXIT_CRITICAL();
    if (pevent != (OS_EVENT *)0) {                  //Get an event control block
        pevent->OSEventType = OS_EVENT_TYPE_SEM;
```

```
        pevent->OSEventCnt  = cnt;                          //Set semaphore value
        OSEventWaitListInit(pevent);
    }
    return (pevent);
}

/*$PAGE*/
/*       PEND ON SEMAPHORE
  Description: This function waits for a semaphore.
  Arguments:
   pevent  is a pointer to the event control block associated with the desired
           semaphore.
   timeout is an optional timeout period (in clock ticks).  If non-zero, your task
           will wait for the resource up to the amount of time specified by this
           argument. If you specify 0, however, your task will wait forever at the
           specified semaphore or, until the resource becomes available (or the
           event occurs).
   err is a pointer to where an error message will be deposited.  Possible error
           messages are:
       OS_NO_ERR      The call was successful and your task owns the resource
                      or, the event you are waiting for occurred.
       OS_TIMEOUT     The semaphore was not received within the specified
                      timeout.
       OS_ERR_EVENT_TYPE  If you didn't pass a pointer to a semaphore.
       OS_ERR_PEND_ISR    If you called this function from an ISR and the result
                          would lead to a suspension.
  Returns: none
*/
void OSSemPend (OS_EVENT *pevent, INT16U timeout, INT8U *err)
{
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {   //Validate event block type
        OS_EXIT_CRITICAL();
        *err = OS_ERR_EVENT_TYPE;
    }
    if (pevent->OSEventCnt > 0) {   //If sem. is positive, resource available ...
        pevent->OSEventCnt--;       //... decrement semaphore only if positive.
        OS_EXIT_CRITICAL();
        *err = OS_NO_ERR;
    } else if (OSIntNesting > 0) {   //See if called from ISR ...
        OS_EXIT_CRITICAL();          //... can't PEND from an ISR
        *err = OS_ERR_PEND_ISR;
    } else {                         //Otherwise, must wait until event occurs
        OSTCBCur->OSTCBStat    |= OS_STAT_SEM;    //Resource not available, pend
                                                  //on semaphore
        OSTCBCur->OSTCBDly      = timeout;        //Store pend timeout in TCB
        OSEventTaskWait(pevent);      //Suspend task until event or timeout occurs
        OS_EXIT_CRITICAL();
        OSSched();                    //Find next highest priority task ready
        OS_ENTER_CRITICAL();
        if (OSTCBCur->OSTCBStat & OS_STAT_SEM) {   //Must have timed out if still
                                                   //waiting for event
            OSEventTO(pevent);
            OS_EXIT_CRITICAL();
            *err = OS_TIMEOUT;         //Indicate that didn't get event within TO
        } else {
            OSTCBCur->OSTCBEventPtr = (OS_EVENT *)0;
            OS_EXIT_CRITICAL();
            *err = OS_NO_ERR;
        }
    }
}
/*$PAGE*/
```

```
/*    POST TO A SEMAPHORE
  Description: This function signals a semaphore
  Arguments:
   pevent is a pointer to the event control block associated with the desired
    semaphore.
  Returns:
   OS_NO_ERR  The call was successful and the semaphore was signaled.
   OS_SEM_OVF If the semaphore count exceeded its limit.  In other words, you have
              signalled the semaphore more often than you waited on it with either
              OSSemAccept() or OSSemPend().
   OS_ERR_EVENT_TYPE  If you didn't pass a pointer to a semaphore
*/
INT8U OSSemPost (OS_EVENT *pevent)
{
    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {   //Validate event block type
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    if (pevent->OSEventGrp) {             //See if any task waiting for semaphore
        OSEventTaskRdy(pevent, (void *)0, OS_STAT_SEM);   //Ready highest prio
                                                          //task waiting on event
        OS_EXIT_CRITICAL();
        OSSched();                       //Find highest priority task ready to run
        return (OS_NO_ERR);
    } else {
        if (pevent->OSEventCnt < 65535) { /* Make sure semaphore will not overflow
            pevent->OSEventCnt++;   //Increment semaphore count to register event
            OS_EXIT_CRITICAL();
            return (OS_NO_ERR);
        } else {                        //Semaphore value has reached its maximum
            OS_EXIT_CRITICAL();
            return (OS_SEM_OVF);
        }
    }
}
/*      QUERY A SEMAPHORE
  Description: This function obtains information about a semaphore
  Arguments:
    pevent is a pointer to the event control block associated with the desired
           semaphore
    pdata  is a pointer to a structure that will contain information about the
           semaphore.
  Returns:
   OS_NO_ERR     The call was successful and the message was sent
   OS_ERR_EVENT_TYPE  If you are attempting to obtain data from a non semaphore.
*/
INT8U OSSemQuery (OS_EVENT *pevent, OS_SEM_DATA *pdata)
{
    INT8U  i;
    INT8U *psrc;
    INT8U *pdest;


    OS_ENTER_CRITICAL();
    if (pevent->OSEventType != OS_EVENT_TYPE_SEM) {   //Validate event block type
        OS_EXIT_CRITICAL();
        return (OS_ERR_EVENT_TYPE);
    }
    pdata->OSEventGrp = pevent->OSEventGrp;      //Copy message mailbox wait list
    psrc              = &pevent->OSEventTbl[0];
    pdest             = &pdata->OSEventTbl[0];
    for (i = 0; i < OS_EVENT_TBL_SIZE; i++) {
```

```
        *pdest++ = *psrc++;
    }
    pdata->OSCnt      = pevent->OSEventCnt;               //Get semaphore count
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
#endif
```

os_task.c

```
#ifndef  OS_MASTER_FILE
#include "includes.h"
#endif

/*    LOCAL FUNCTION PROTOTYPES  */
#if OS_TASK_DEL_EN
static  void  OSDummy(void);

/*      DUMMY FUNCTION
  Description:
 This function doesn't do anything.  It is called by OSTaskDel() to
 ensure that interrupts are disabled immediately after they are enabled.
  Arguments: none
  Returns: none
*/
static void  OSDummy (void)
{
}
#endif

/*$PAGE*/
/*     CHANGE PRIORITY OF A TASK
  Description: This function allows you to change the priority of a
  task dynamically.  Note that the new priority MUST be available.
  Arguments:
    oldp     is the old priority
    newp     is the new priority
  Returns:
   OS_NO_ERR        is the call was successful
   OS_PRIO_INVALID  if the priority you specify is higher that the
                    maximum allowed (i.e. >= OS_LOWEST_PRIO)
   OS_PRIO_EXIST    if the new priority already exist.
   OS_PRIO_ERR      there is no task with the specified OLD priority
                    (i.e. the OLD task does not exist.
*/
#if OS_TASK_CHANGE_PRIO_EN
INT8U OSTaskChangePrio (INT8U oldprio, INT8U newprio)
{   OS_TCB   *ptcb;
    OS_EVENT *pevent;
    INT8U     x;
    INT8U     y;
    INT8U     bitx;
    INT8U     bity;

    if ((oldprio >= OS_LOWEST_PRIO && oldprio != OS_PRIO_SELF)  ||
         newprio >= OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[newprio] != (OS_TCB *)0) {  //New priority
                                                 //must not already exist
        OS_EXIT_CRITICAL();
        return (OS_PRIO_EXIST);
    } else {
        OSTCBPrioTbl[newprio] = (OS_TCB *)1;    //Reserve the entry
                                                //to prevent others
        OS_EXIT_CRITICAL();
        y    = newprio >> 3;     //Precompute to reduce INT. latency
        bity = OSMapTbl[y];
        x    = newprio & 0x07;
        bitx = OSMapTbl[x];
        OS_ENTER_CRITICAL();
```

```
        if (oldprio == OS_PRIO_SELF) {        //See if changing self
            oldprio = OSTCBCur->OSTCBPrio;    //Yes, get priority
        }
        if ((ptcb = OSTCBPrioTbl[oldprio]) != (OS_TCB *)0) {  //Task
                                                  //to change must
            OSTCBPrioTbl[oldprio] = (OS_TCB *)0;    //Remove TCB from
                                                  //old priority
            if (OSRdyTbl[ptcb->OSTCBY] & ptcb->OSTCBBitX) {  //If
                                  //task is ready make it not ready
                if((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX)==0) {
                    OSRdyGrp &= ~ptcb->OSTCBBitY;
                }
                OSRdyGrp    |= bity; //Make new priority ready to run
                OSRdyTbl[y] |= bitx;
            } else {
                if ((pevent = ptcb->OSTCBEventPtr) != (OS_EVENT *)0) {
                             /* Remove from event wait list  */
                    if((pevent->OSEventTbl[ptcb->OSTCBY]&= ~ptcb->OSTCBBitX)==0) {
                        pevent->OSEventGrp &= ~ptcb->OSTCBBitY;
                    }
                    pevent->OSEventGrp    |= bity;  //Add new priority to wait
                                                  //list
                    pevent->OSEventTbl[y] |= bitx;
                }
            }
            OSTCBPrioTbl[newprio] = ptcb;    //Place pointer to TCB @ new priority
            ptcb->OSTCBPrio       = newprio;    //Set new task priority
            ptcb->OSTCBY          = y;
            ptcb->OSTCBX          = x;
            ptcb->OSTCBBitY       = bity;
            ptcb->OSTCBBitX       = bitx;
            OS_EXIT_CRITICAL();
            OSSched();                  //Run highest priority task ready
            return (OS_NO_ERR);
        } else {
            OSTCBPrioTbl[newprio] = (OS_TCB *)0;    //Release the reserved prio.
            OS_EXIT_CRITICAL();
            return (OS_PRIO_ERR);                   //Task to change didn't exist
        }
    }
}
#endif
/*$PAGE*/
/*    CREATE A TASK
  Description: This function is used to have uC/OS-II manage the execution of a
  task. Tasks can either be created prior to the start of multitasking or by a
 running task.  A task cannot be created by an ISR.
  Arguments:
    task     is a pointer to the task's code
    pdata    is a pointer to an optional data area which can be used to pass
     parameters to the task when the task first executes.  Where the task is
     concerned it thinks it was invoked and passed the argument 'pdata' as
     follows:
       void Task (void *pdata)
         {
          for (;;) {
            Task code;
          }
         }
   ptos    is a pointer to the task's top of stack.  If the configuration
    constant OS_STK_GROWTH is set to 1, the stack is assumed to grow downward
    (i.e. from high memory to low memory).  'pstk' will thus point to the highest
    (valid) memory location of the stack.  If OS_STK_GROWTH is set to 0, 'pstk'
```

```
    will point to the lowest memory location of the stack and the stack will grow
    with increasing memory locations.
   prio    is the task's priority.  A unique priority MUST be assigned to each
    task and the lower the number, the higher the priority.
  Returns:
   OS_NO_ERR        if the function was successful.
   OS_PRIO_EXIT     if the task priority already exist
                    (each task MUST have a unique priority).
   OS_PRIO_INVALID  if the priority you specify is higher that the maximum allowed
                    (i.e. >= OS_LOWEST_PRIO)
*/
#if OS_TASK_CREATE_EN
INT8U OSTaskCreate (void (*task)(void *pd), void *pdata, OS_STK *ptos, INT8U prio)
{   void   *psp;
    INT8U   err;

    if (prio > OS_LOWEST_PRIO) {  //Make sure priority is within allowable range
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) { //Make sure task doesn't already
                                             //exist at this priority
        OSTCBPrioTbl[prio] = (OS_TCB *)1;    //Reserve the priority to prevent
                                             //others from doing ...
                                       //... the same thing until task is created.
        OS_EXIT_CRITICAL();
        psp = (void *)OSTaskStkInit(task, pdata, ptos, 0); //Initialize the task's
                                                           //stack
        err = OSTCBInit(prio, psp, (void *)0, 0, 0, (void *)0, 0);
        if (err == OS_NO_ERR) {
            OS_ENTER_CRITICAL();
            OSTaskCtr++;                           //Increment the #tasks counter
            OSTaskCreateHook(OSTCBPrioTbl[prio]);  //Call user defined hook
            OS_EXIT_CRITICAL();
            if(OSRunning) {    //Find highest priority task if multitasking has
                               //started
                OSSched();
            }
        } else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;//Make this priority available to
                                             //others
            OS_EXIT_CRITICAL();
        }
        return (err);
    } else {
        OS_EXIT_CRITICAL();
        return (OS_PRIO_EXIST);
    }
}
#endif
/*$PAGE*/
/*    CREATE A TASK (Extended Version)
  Description: This function is used to have uC/OS-II manage the execution of a
 task.  Tasks can either be created prior to the start of multitasking or by a
 running task.  A task cannot be created by an ISR.  This function is similar to
 OSTaskCreate() except that it allows additional information about a task to be
 specified.
  Arguments:
    task     is a pointer to the task's code
    pdata    is a pointer to an optional data area which can be used to pass
     parameters to the task when the task first executes.  Where the task is
     concerned it thinks it was invoked and passed the argument 'pdata' as
```

```
     follows:
       void Task (void *pdata)
         {
           for (;;) {
             Task code;
           }
         }
    ptos     is a pointer to the task's top of stack.  If the configuration
     constant OS_STK_GROWTH is set to 1, the stack is assumed to grow downward
     (i.e. from high memory to low memory).  'pstk' will thus point to the highest
     (valid) memory location of the stack.  If OS_STK_GROWTH is set to 0, 'pstk'
     will point to the lowest memory location of the stack and the stack will grow
     with increasing memory locations.  'pstk' MUST point to a valid 'free' data
     item.
    prio     is the task's priority.  A unique priority MUST be assigned to each
     task and the lower the number, the higher the priority.
    id       is the task's ID (0..65535)
    pbos     is a pointer to the task's bottom of stack.  If the configuration
     constant OS_STK_GROWTH is set to 1, the stack is assumed to grow downward
     (i.e. from high memory to low memory).  'pbos' will thus point to the LOWEST
     (valid) memory location of the stack.  If OS_STK_GROWTH is set to 0, 'pbos'
     will point to the HIGHEST memory location of the stack and the stack will
     grow with increasing memory locations.  'pbos' MUST point to a valid 'free'
     data item.
    stk_size is the size of the stack in number of elements.  If OS_STK is set to
     INT8U, 'stk_size' corresponds to the number of bytes available.  If OS_STK is
     set to INT16U, 'stk_size' contains the number of 16-bit entries available.
     Finally, if OS_STK is set to INT32U, 'stk_size' contains the number of 32-bit
     entries available on the stack.
    pext     is a pointer to a user supplied memory location which is used as a
     TCB extension. For example, this user memory can hold the contents of
     floating-point registers during a context switch, the time each task takes to
     execute, the number of times the task has been switched-in, etc.
    opt      contains additional information (or options) about the behavior of
     the task.  The LOWER 8-bits are reserved by uC/OS-II while the upper 8 bits
     can be application specific.  See OS_TASK_OPT_??? in uCOS-II.H.
  Returns:
    OS_NO_ERR        if the function was successful.
    OS_PRIO_EXIT     if the task priority already exist (each task MUST have a
                     unique priority).
    OS_PRIO_INVALID  if the priority you specify is higher that the maximum
                     allowed (i.e. > OS_LOWEST_PRIO)
*/
/*$PAGE*/
#if   OS_TASK_CREATE_EXT_EN
INT8U OSTaskCreateExt (void   (*task)(void *pd),
                       void    *pdata,
                       OS_STK  *ptos,
                       INT8U    prio,
                       INT16U   id,
                       OS_STK  *pbos,
                       INT32U   stk_size,
                       void    *pext,
                       INT16U   opt)
{   void    *psp;
    INT8U    err;
    INT16U   i;
    OS_STK  *pfill;

    if (prio > OS_LOWEST_PRIO) {   //Make sure priority is within allowable range
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
```

```
    if (OSTCBPrioTbl[prio] == (OS_TCB *)0) { //Make sure task doesn't already
                                             //exist at this priority
        OSTCBPrioTbl[prio] = (OS_TCB *)1;    //Reserve the priority to prevent
                                             //others from doing ...
                                       //... the same thing until task is created.
        OS_EXIT_CRITICAL();

        if (opt & OS_TASK_OPT_STK_CHK) {  //See if stack checking has been enabled
            if (opt & OS_TASK_OPT_STK_CLR) { //See if stack needs to be cleared
                pfill = pbos;                //Yes, fill the stack with zeros
                for (i = 0; i < stk_size; i++) {
                    #if OS_STK_GROWTH == 1
                    *pfill++ = (OS_STK)0;
                    #else
                    *pfill-- = (OS_STK)0;
                    #endif
                }
            }
        }

        psp = (void *)OSTaskStkInit(task, pdata, ptos, opt); //Initialize the
                                                             //task's stack
        err = OSTCBInit(prio, psp, pbos, id, stk_size, pext, opt);
        if (err == OS_NO_ERR) {
            OS_ENTER_CRITICAL();
            OSTaskCtr++;                             //Increment the #tasks counter
            OSTaskCreateHook(OSTCBPrioTbl[prio]);  //Call user defined hook
            OS_EXIT_CRITICAL();
            if (OSRunning) {              //Find highest priority task if
                                          //multitasking has started
                OSSched();
            }
        } else {
            OS_ENTER_CRITICAL();
            OSTCBPrioTbl[prio] = (OS_TCB *)0;//Make this priority available to
                                             //others
            OS_EXIT_CRITICAL();
        }
        return (err);
    } else {
        OS_EXIT_CRITICAL();
        return (OS_PRIO_EXIST);
    }
}
#endif
/*$PAGE*/
/*    DELETE A TASK
  Description: This function allows you to delete a task.  The calling task can
  delete itself by its own priority number.  The deleted task is returned to the
  dormant state and can be re-activated by creating the deleted task again.
  Arguments:
    prio    is the priority of the task to delete.  Note that you can explicitely
     delete the current task without knowing its priority level by setting 'prio'
     to OS_PRIO_SELF.
  Returns:
    OS_NO_ERR           if the call is successful
    OS_TASK_DEL_IDLE    if you attempted to delete uC/OS-II's idle task
    OS_PRIO_INVALID     if the priority you specify is higher that the maximum
       allowed (i.e. >= OS_LOWEST_PRIO) or, you have not specified OS_PRIO_SELF.
    OS_TASK_DEL_ERR     if the task you want to delete does not exist
    OS_TASK_DEL_ISR     if you tried to delete a task from an ISR
  Notes: 1) To reduce interrupt latency, OSTaskDel() 'disables' the task:
            a) by making it not ready
```

```
            b) by removing it from any wait lists
            c) by preventing OSTimeTick() from making the task ready to run.
            The task can then be 'unlinked' from the miscellaneous structures
               in uC/OS-II.
            2) The function OSDummy() is called after OS_EXIT_CRITICAL() because,
            on most processors, the next instruction following the enable
            interrupt instruction is ignored.  You can replace OSDummy() with a
            macro that basically executes a NO OP (i.e. OS_NOP()). The NO OP macro
            would avoid the execution time of the function call and return.
         3) An ISR cannot delete a task.
         4) The lock nesting counter is incremented because, for a brief instant,
         if the current task is being deleted, the current task would not be
         able to be rescheduled because it is removed from the ready list.
         Incrementing the nesting counter prevents another task from being
         schedule.  This means that the ISR would return to the current task
         which is being deleted.  The rest of the deletion would thus be able to
         be completed.
*/
/*$PAGE*/
#if OS_TASK_DEL_EN
INT8U OSTaskDel (INT8U prio)
{   OS_TCB   *ptcb;
    OS_EVENT *pevent;

    if (prio == OS_IDLE_PRIO) {                 //Not allowed to delete idle task
        return (OS_TASK_DEL_IDLE);
    }
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) { //Task priority valid ?
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (OSIntNesting > 0) {                     //See if trying to delete from ISR
        OS_EXIT_CRITICAL();
        return (OS_TASK_DEL_ISR);
    }
    if (prio == OS_PRIO_SELF) {                 //See if requesting to delete self
        prio = OSTCBCur->OSTCBPrio;             //Set priority to delete to current
    }
    if ((ptcb = OSTCBPrioTbl[prio]) != (OS_TCB *)0) { //Task to delete must exist
        if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {//Make task not
                                                                //ready
            OSRdyGrp &= ~ptcb->OSTCBBitY;
        }
        if ((pevent = ptcb->OSTCBEventPtr) != (OS_EVENT *)0) {  //If task is
                                                                //waiting on event
            if ((pevent->OSEventTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {
                                                            //... remove task from
                pevent->OSEventGrp &= ~ptcb->OSTCBBitY;     //... event ctrl block
            }
        }
        ptcb->OSTCBDly  = 0;                //Prevent OSTimeTick() from updating
        ptcb->OSTCBStat = OS_STAT_RDY;      //Prevent task from being resumed
        OSLockNesting++;
        OS_EXIT_CRITICAL();                 //Enabling INT. ignores next instruc.
        OSDummy();                          //... Dummy ensures that INTs will be
        OS_ENTER_CRITICAL();                //... disabled HERE!
        OSLockNesting--;
        OSTaskDelHook(ptcb);                //Call user defined hook
        OSTaskCtr--;                        //One less task being managed
        OSTCBPrioTbl[prio] = (OS_TCB *)0;   //Clear old priority entry
        if (ptcb->OSTCBPrev == (OS_TCB *)0) {   //Remove from TCB chain
            ptcb->OSTCBNext->OSTCBPrev = (OS_TCB *)0;
            OSTCBList                  = ptcb->OSTCBNext;
```

```
        } else {
            ptcb->OSTCBPrev->OSTCBNext = ptcb->OSTCBNext;
            ptcb->OSTCBNext->OSTCBPrev = ptcb->OSTCBPrev;
        }
        ptcb->OSTCBNext = OSTCBFreeList;       //Return TCB to free TCB list
        OSTCBFreeList   = ptcb;
        OS_EXIT_CRITICAL();
        OSSched();                              //Find new highest priority task
        return (OS_NO_ERR);
    } else {
        OS_EXIT_CRITICAL();
        return (OS_TASK_DEL_ERR);
    }
}
#endif
/*$PAGE*/
/*  REQUEST THAT A TASK DELETE ITSELF
  Description: This function is used to:
       a) notify a task to delete itself.
       b) to see if a task requested that the current task delete itself.
       This function is a little tricky to understand.  Basically, you have a task
       that needs to be deleted however, this task has resources that it has
       allocated (memory buffers, semaphores, mailboxes, queues etc.).  The task
       cannot be deleted otherwise these resources would not be freed.  The
       requesting task calls OSTaskDelReq() to indicate that the task needs to be
       deleted.  Deleting of the task is however, deferred to the task to be
       deleted.  For example, suppose that task #10 needs to be deleted.  The
       requesting task example, task #5, would call OSTaskDelReq(10).  When task
       #10 gets to execute, it calls this function by specifying OS_PRIO_SELF and
       monitors the returned value.  If the return value is OS_TASK_DEL_REQ,
       another task requested a task delete.  Task #10 would look like this:
          void Task(void *data)
            {.
             .
             while (1) {
               OSTimeDly(1);
                 if (OSTaskDelReq(OS_PRIO_SELF) == OS_TASK_DEL_REQ) {
                      Release any owned resources;
                      De-allocate any dynamic memory;
                      OSTaskDel(OS_PRIO_SELF);
                 }
             }
            }
  Arguments: prio    is the priority of the task to request the delete from
  Returns:
    OS_NO_ERR          if the task exist and the request has been registered
    OS_TASK_NOT_EXIST  if the task has been deleted.  This allows the caller to
                       know whether the request has been executed.
    OS_TASK_DEL_IDLE   if you requested to delete uC/OS-II's idle task
    OS_PRIO_INVALID    if the priority you specify is higher that the maximum
        allowed (i.e. >= OS_LOWEST_PRIO) or, you have not specified OS_PRIO_SELF.
    OS_TASK_DEL_REQ    if a task (possibly another task) requested that the
        running task be deleted.
*/
/*$PAGE*/
#if OS_TASK_DEL_EN
INT8U OSTaskDelReq (INT8U prio)
{   BOOLEAN  stat;
    INT8U    err;
    OS_TCB  *ptcb;

    if (prio == OS_IDLE_PRIO) {                 //Not allowed to delete idle task
        return (OS_TASK_DEL_IDLE);
```

```
    }
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {  //Task priority valid?
        return (OS_PRIO_INVALID);
    }
    if (prio == OS_PRIO_SELF) {               //See if a task is requesting to ...
        OS_ENTER_CRITICAL();                  //... this task to delete itself
        stat = OSTCBCur->OSTCBDelReq;         //Return request status to caller
        OS_EXIT_CRITICAL();
        return (stat);
    } else {
        OS_ENTER_CRITICAL();
        if ((ptcb = OSTCBPrioTbl[prio]) != (OS_TCB *)0) {  //Task to delete must
                                                           //exist
         ptcb->OSTCBDelReq=OS_TASK_DEL_REQ; //Set flag indicating task to be DEL.
         err               = OS_NO_ERR;
        } else {
            err            = OS_TASK_NOT_EXIST;          //Task must be deleted
        }
        OS_EXIT_CRITICAL();
        return (err);
    }
}
#endif
/*$PAGE*/
/*   RESUME A SUSPENDED TASK
  Description: This function is called to resume a previously suspended task.
  This is the only call that will remove an explicit task suspension.
  Arguments: prio     is the priority of the task to resume.
  Returns:
    OS_NO_ERR      if the requested task is resumed
    OS_PRIO_INVALID  if the priority you specify is higher that the maximum llowed
        (i.e. >= OS_LOWEST_PRIO)
    OS_TASK_RESUME_PRIO    if the task to resume does not exist
    OS_TASK_NOT_SUSPENDED  if the task to resume has not been suspended
*/
#if OS_TASK_SUSPEND_EN
INT8U OSTaskResume (INT8U prio)
{   OS_TCB   *ptcb;

    if (prio >= OS_LOWEST_PRIO) {            //Make sure task priority is valid
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) { //Task to suspend must exist
        OS_EXIT_CRITICAL();
        return (OS_TASK_RESUME_PRIO);
    } else {
        if (ptcb->OSTCBStat & OS_STAT_SUSPEND) {      //Task must be suspended
            if (((ptcb->OSTCBStat &= ~OS_STAT_SUSPEND) == OS_STAT_RDY) &&
                                                       //Remove suspension
                 (ptcb->OSTCBDly  == 0)) {            //Must not be delayed
                OSRdyGrp               |= ptcb->OSTCBBitY;  //Make task ready to
                                                            //run
                OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
                OS_EXIT_CRITICAL();
                OSSched();
            } else {
                OS_EXIT_CRITICAL();
            }
            return (OS_NO_ERR);
        } else {
            OS_EXIT_CRITICAL();
            return (OS_TASK_NOT_SUSPENDED);
```

```
        }
    }
}
#endif
/*$PAGE*/
/*   STACK CHECKING
  Description: This function is called to check the amount of free memory left on
  the specified task's stack.
  Arguments:
    prio     is the task priority
    pdata    is a pointer to a data structure of type OS_STK_DATA.
  Returns:
    OS_NO_ERR           upon success
    OS_PRIO_INVALID     if the priority you specify is higher that the maximum
        allowed (i.e. > OS_LOWEST_PRIO) or, you have not specified OS_PRIO_SELF.
    OS_TASK_NOT_EXIST   if the desired task has not been created
    OS_TASK_OPT_ERR     if you did NOT specified OS_TASK_OPT_STK_CHK when the task
                        was created
*/
#if   OS_TASK_CREATE_EXT_EN
INT8U OSTaskStkChk (INT8U prio, OS_STK_DATA *pdata)
{   OS_TCB  *ptcb;
    OS_STK  *pchk;
    INT32U   free;
    INT32U   size;

    pdata->OSFree = 0;                              //ssume failure, set to 0 size
    pdata->OSUsed = 0;
    if (prio > OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {  //Make sure task
                                                          //priority is valid
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                         //See if check for SELF
        prio = OSTCBCur->OSTCBPrio;
    }
    ptcb = OSTCBPrioTbl[prio];
    if (ptcb == (OS_TCB *)0) {                          //Make sure task exist
        OS_EXIT_CRITICAL();
        return (OS_TASK_NOT_EXIST);
    }
    if ((ptcb->OSTCBOpt & OS_TASK_OPT_STK_CHK) == 0) { //Make sure stack checking
                                                        //option is set
        OS_EXIT_CRITICAL();
        return (OS_TASK_OPT_ERR);
    }
    free = 0;
    size = ptcb->OSTCBStkSize;
    pchk = ptcb->OSTCBStkBottom;
    OS_EXIT_CRITICAL();
#if OS_STK_GROWTH == 1
    while (*pchk++ == 0) {        //Compute the number of zero entries on the stk
        free++;
    }
#else
    while (*pchk-- == 0) {
        free++;
    }
#endif
    pdata->OSFree = free * sizeof(OS_STK);  //Compute number of free bytes on the
                                            //stack
    pdata->OSUsed = (size - free) * sizeof(OS_STK);   //Compute number of bytes
                                                      //used on the stack
```

```
    return (OS_NO_ERR);
}
#endif
/*$PAGE*/
/*    SUSPEND A TASK
  Description: This function is called to suspend a task.  The task can be the
  calling task if the priority passed to OSTaskSuspend() is the priority of the
  calling task or OS_PRIO_SELF.
  Arguments:
    prio  is the priority of the task to suspend. If you specify OS_PRIO_SELF, the
     calling task will suspend itself and rescheduling will occur.
  Returns:
    OS_NO_ERR             if the requested task is suspended
    OS_TASK_SUSPEND_IDLE  if you attempted to suspend the idle task which is
                          not allowed.
    OS_PRIO_INVALID       if the priority you specify is higher that the maximum
        allowed (i.e. >= OS_LOWEST_PRIO) or, you have not specified OS_PRIO_SELF.
    OS_TASK_SUSPEND_PRIO     if the task to suspend does not exist
  Note: You should use this function with great care.  If you suspend a task that
  is waiting for an event (i.e. a message, a semaphore, a queue ...) you will
  prevent this task from running when the event arrives.
*/
#if OS_TASK_SUSPEND_EN
INT8U OSTaskSuspend (INT8U prio)
{   BOOLEAN   self;
    OS_TCB   *ptcb;

    if (prio == OS_IDLE_PRIO) {               //Not allowed to suspend idle task
        return (OS_TASK_SUSPEND_IDLE);
    }
    if (prio >= OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {  //Task priority valid?
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                            //See if suspend SELF
        prio = OSTCBCur->OSTCBPrio;
        self = TRUE;
    } else if (prio == OSTCBCur->OSTCBPrio) {            //See if suspending self
        self = TRUE;
    } else {
        self = FALSE;                                   //No suspending another task
    }
    if ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) { /Task to suspend must exist
        OS_EXIT_CRITICAL();
        return (OS_TASK_SUSPEND_PRIO);
    } else {
        if ((OSRdyTbl[ptcb->OSTCBY] &= ~ptcb->OSTCBBitX) == 0) {   //Make task not
                                                                   //ready
            OSRdyGrp &= ~ptcb->OSTCBBitY;
        }
        ptcb->OSTCBStat |= OS_STAT_SUSPEND;       //Status of task is 'SUSPENDED'
        OS_EXIT_CRITICAL();
        if (self == TRUE) {                       //Context switch only if SELF
            OSSched();
        }
        return (OS_NO_ERR);
    }
}
#endif
/*$PAGE*/
/*    QUERY A TASK
  Description: This function is called to obtain a copy of the desired task's TCB.
  Arguments: prio     is the priority of the task to obtain information from.
```

```
  Returns:
    OS_NO_ERR       if the requested task is suspended
    OS_PRIO_INVALID if the priority you specify is higher that the maximum allowed
       (i.e. > OS_LOWEST_PRIO) or, you have not specified OS_PRIO_SELF.
    OS_PRIO_ERR     if the desired task has not been created
*/
INT8U OSTaskQuery (INT8U prio, OS_TCB *pdata)
{   OS_TCB *ptcb;

    if (prio > OS_LOWEST_PRIO && prio != OS_PRIO_SELF) {  //Task priority valid?
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    if (prio == OS_PRIO_SELF) {                            //See if suspend SELF
        prio = OSTCBCur->OSTCBPrio;
    }
    if ((ptcb = OSTCBPrioTbl[prio]) == (OS_TCB *)0) {   //Task to query must exist
        OS_EXIT_CRITICAL();
        return (OS_PRIO_ERR);
    }
    *pdata = *ptcb;                              //Copy TCB into user storage area
    OS_EXIT_CRITICAL();
    return (OS_NO_ERR);
}
```

os_time.c

```
#ifndef  OS_MASTER_FILE
#include "includes.h"
#endif

/*     DELAY TASK 'n' TICKS   (n from 0 to 65535)
  Description: This function is called to delay execution of the currently running
  task until the specified number of system ticks expires.  This, of course,
  directly equates to delaying the current task for some time to expire.  No delay
  will result If the specified delay is 0.  If the specified delay is greater than
  0 then, a context switch will result.
  Arguments: ticks  is the time delay that the task will be suspended in number of
            clock 'ticks'. Note that by specifying 0, the task will not be delayed.
  Returns: none
*/
void OSTimeDly (INT16U ticks)
{   if (ticks > 0) {                                       //0 means no delay!
        OS_ENTER_CRITICAL();
        if ((OSRdyTbl[OSTCBCur->OSTCBY] &= ~OSTCBCur->OSTCBBitX) == 0) {  //Delay
                                                                          //current task
            OSRdyGrp &= ~OSTCBCur->OSTCBBitY;
        }
        OSTCBCur->OSTCBDly = ticks;                      //Load ticks in TCB
        OS_EXIT_CRITICAL();
        OSSched();                                         //Find next task to run!
    }
}
/*$PAGE*/
/*    DELAY TASK FOR SPECIFIED TIME
  Description: This function is called to delay execution of the currently running
  task until some time expires.  This call allows you to specify the delay time in
  HOURS, MINUTES, SECONDS and MILLISECONDS instead of ticks.
  Arguments:
    hours       specifies the number of hours that the task will be delayed
                (max. is 255)
    minutes   specifies the number of minutes (max. 59)
    seconds   specifies the number of seconds (max. 59)
    milli     specifies the number of milliseconds (max. 999)
```

```
  Returns: OS_NO_ERR
           OS_TIME_INVALID_MINUTES
           OS_TIME_INVALID_SECONDS
           OS_TIME_INVALID_MS
           OS_TIME_ZERO_DLY
  Note(s): The resolution on the milliseconds depends on the tick rate.  For
  example, you can't do a 10 mS delay if the ticker interrupts every 100 mS.  In
  this case, the delay would be set to 0.  The actual delay is rounded to the
  nearest tick.
*/
INT8U OSTimeDlyHMSM (INT8U hours, INT8U minutes, INT8U seconds, INT16U milli)
{   INT32U ticks;
    INT16U loops;

    if (hours > 0 || minutes > 0 || seconds > 0 || milli > 0) {
        if (minutes > 59) {
            return (OS_TIME_INVALID_MINUTES);    //Validate arguments to be within
                                                 //range
        }
        if (seconds > 59) {
            return (OS_TIME_INVALID_SECONDS);
        }
        if (milli > 999) {
            return (OS_TIME_INVALID_MILLI);
        }
                            //Compute the total number of clock ticks required..
                            //.. (rounded to the nearest tick)
        ticks = ((INT32U)hours * 3600L + (INT32U)minutes * 60L + (INT32U)seconds)
              * OS_TICKS_PER_SEC
              + OS_TICKS_PER_SEC*((INT32U)milli+500L/OS_TICKS_PER_SEC)/1000L;
        loops = ticks / 65536L; //Compute the integral number of 65536 tick delays
        ticks = ticks % 65536L;          //Obtain  the fractional number of ticks
        OSTimeDly(ticks);
        while (loops > 0) {
            OSTimeDly(32768);
            OSTimeDly(32768);
            loops--;
        }
        return (OS_NO_ERR);
    } else {
        return (OS_TIME_ZERO_DLY);
    }
}
/*$PAGE*/
/*    RESUME A DELAYED TASK
  Description: This function is used resume a task that has been delayed through a
  call to either OSTimeDly() or OSTimeDlyHMSM().  Note that you MUST NOT call this
  function to resume a task that is waiting for an event with timeout.  This
  situation would make the task look like a timeout occurred (unless you desire
  this effect).  Also, you cannot resume a task that has called OSTimeDlyHMSM()
  with a combined time that exceeds 65535 clock ticks.  In other words, if the
  clock tick runs at 100 Hz then, you will not be able to resume a delayed task
  that called OSTimeDlyHMSM(0, 10, 55, 350) or higher.
              (10 Minutes * 60 + 55 Seconds + 0.35) * 100 ticks/second.
  Arguments: prio      specifies the priority of the task to resume
  Returns: OS_NO_ERR            Task has been resumed
           OS_PRIO_INVALID      if the priority you specify is higher that the
                                maximum allowed (i.e. >= OS_LOWEST_PRIO)
           OS_TIME_NOT_DLY      Task is not waiting for time to expire
           OS_TASK_NOT_EXIST    The desired task has not been created
*/
INT8U OSTimeDlyResume (INT8U prio)
{   OS_TCB *ptcb;
```

```
    if (prio >= OS_LOWEST_PRIO) {
        return (OS_PRIO_INVALID);
    }
    OS_ENTER_CRITICAL();
    ptcb = (OS_TCB *)OSTCBPrioTbl[prio];           //Make sure that task exist
    if (ptcb != (OS_TCB *)0) {
        if (ptcb->OSTCBDly != 0) {                     //See if task is delayed
            ptcb->OSTCBDly  = 0;                       //Clear the time delay
            if (!(ptcb->OSTCBStat & OS_STAT_SUSPEND)) {  //See if task is ready
                                                         //to run
               OSRdyGrp               |= ptcb->OSTCBBitY; //Make task ready to run
               OSRdyTbl[ptcb->OSTCBY] |= ptcb->OSTCBBitX;
               OS_EXIT_CRITICAL();
               OSSched();                  //See if this is new highest priority
            } else {
                OS_EXIT_CRITICAL();               //Task may be suspended
            }
            return (OS_NO_ERR);
        } else {
            OS_EXIT_CRITICAL();
            return (OS_TIME_NOT_DLY);        //Indicate that task was not delayed
        }
    } else {
        OS_EXIT_CRITICAL();
        return (OS_TASK_NOT_EXIST);                    //The task does not exist
    }
}
/*$PAGE*/
/*     GET CURRENT SYSTEM TIME
  Description: This function is used by your application to obtain the current
  value of the 32-bit counter which keeps track of the number of clock ticks.
  Arguments: none
  Returns: The current value of OSTime
*/
INT32U OSTimeGet (void)
{   INT32U ticks;

    OS_ENTER_CRITICAL();
    ticks = OSTime;
    OS_EXIT_CRITICAL();
    return (ticks);
}

/*   SET SYSTEM CLOCK
  Description: This function sets the 32-bit counter which keeps track of the
  number of clock ticks.
  Arguments: ticks    specifies the new value that OSTime needs to take.
  Returns: none
*/
void OSTimeSet (INT32U ticks)
{   OS_ENTER_CRITICAL();
    OSTime = ticks;
    OS_EXIT_CRITICAL();
}
```

ping.c

```
/* This program is an example of using semaphore to implement task rendevous.
*/
#include    "includes.h"               /* uC/OS interface */

/* allocate memory for tasks' stacks */
#ifdef SEMIHOSTED
```

```
#define STACKSIZE       (SEMIHOSTED_STACK_NEEDS+64)
#else
#define     STACKSIZE   64
#endif
unsigned int Stack1[STACKSIZE];
unsigned int Stack2[STACKSIZE];

/* semaphores event control blocks */
OS_EVENT *Sem1;
OS_EVENT *Sem2;

/* Task running at the highest priority. */
void Task1(void *i)
{   INT8U Reply;

    for (;;)
    {   /* wait for the semaphore  */
        OSSemPend(Sem2, 0, &Reply);

        uHALr_printf("1+");

        /* wait a short while */
        OSTimeDly(100);

        uHALr_printf("1-");

        /* signal the semaphore */
        OSSemPost(Sem1);
    }
}

void Task2(void *i)
{   INT8U Reply;

    for (;;)
    {  /* wait for the semaphore */
        OSSemPend(Sem1, 0, &Reply);

        uHALr_printf("[");

        /* wait a short while */
        OSTimeDly(1000);

        uHALr_printf("2]");

        /* signal the semaphore */
        OSSemPost(Sem2);
    }
}

/* Main function. */
int main(int argc, char **argv)
{   char Id1 = '1';
    char Id2 = '2';

    /* do target (uHAL based ARM system) initialisation */
    ARMTargetInit();

    /* needed by uC/OS */
    OSInit();

    OSTimeSet(0);
    /*  create the semaphores   */
```

```
    Sem1 = OSSemCreate(1);
    Sem2 = OSSemCreate(1);

    /*  create the tasks in uC/OS and assign decreasing priority to them  */
    OSTaskCreate(Task1, (void *)&Id1, (void *)&Stack1[STACKSIZE - 1], 1);
    OSTaskCreate(Task2, (void *)&Id2, (void *)&Stack2[STACKSIZE - 1], 2);

    /* Start the (uHAL based ARM system) system running */
    ARMTargetStart();

    /* start the game */
    OSStart();

    /* never reached */
}                               /* main */
```

ucos_ii.c

```
#define  OS_GLOBALS                  //Declare GLOBAL variables
#include "includes.h"

#define  OS_MASTER_FILE   //Prevent the following files from including includes.h
#include "os_core.c"
#include "os_mbox.c"
#include "os_mem.c"
#include "os_q.c"
#include "os_sem.c"
#include "os_task.c"
#include "os_time.c"
```

ucos_ii.h

```
/*       MISCELLANEOUS  */
#define  OS_VERSION         200   //Version of uC/OS-II (Vx.yy multiplied by 100)

#ifdef   OS_GLOBALS
#define  OS_EXT
#else
#define  OS_EXT  extern
#endif

#define  OS_PRIO_SELF           0xFF   //Indicate SELF priority

#if      OS_TASK_STAT_EN
#define  OS_N_SYS_TASKS            2   //Number of system tasks
#else
#define  OS_N_SYS_TASKS            1
#endif

#define  OS_STAT_PRIO       (OS_LOWEST_PRIO - 1)  //Statistic task priority
#define  OS_IDLE_PRIO       (OS_LOWEST_PRIO)      //IDLE task priority

#define  OS_EVENT_TBL_SIZE ((OS_LOWEST_PRIO) / 8 + 1)   //Size of event table
#define  OS_RDY_TBL_SIZE   ((OS_LOWEST_PRIO) / 8 + 1)   //Size of ready table

#define  OS_TASK_IDLE_ID       65535   //I.D. numbers for Idle and Stat tasks
#define  OS_TASK_STAT_ID       65534

          //TASK STATUS (Bit definition for OSTCBStat)
#define  OS_STAT_RDY            0x00   //Ready to run
#define  OS_STAT_SEM            0x01   //Pending on semaphore
#define  OS_STAT_MBOX           0x02   //Pending on mailbox
#define  OS_STAT_Q              0x04   //Pending on queue
#define  OS_STAT_SUSPEND        0x08   //Task is suspended
```

```
#define  OS_EVENT_TYPE_MBOX        1
#define  OS_EVENT_TYPE_Q           2
#define  OS_EVENT_TYPE_SEM         3

           //  TASK OPTIONS (see OSTaskCreateExt())
#define  OS_TASK_OPT_STK_CHK  0x0001   //Enable stack checking for the task
#define  OS_TASK_OPT_STK_CLR  0x0002   //Clear the stack when the task is create
#define  OS_TASK_OPT_SAVE_FP  0x0004   //Save the contents of any floating-point
                                       //registers

#ifndef  FALSE
#define  FALSE                     0
#endif

#ifndef  TRUE
#define  TRUE                      1
#endif

/*     ERROR CODES   */
#define OS_NO_ERR                 0
#define OS_ERR_EVENT_TYPE         1
#define OS_ERR_PEND_ISR           2

#define OS_TIMEOUT               10
#define OS_TASK_NOT_EXIST        11

#define OS_MBOX_FULL             20

#define OS_Q_FULL                30

#define OS_PRIO_EXIST            40
#define OS_PRIO_ERR              41
#define OS_PRIO_INVALID          42

#define OS_SEM_OVF               50

#define OS_TASK_DEL_ERR          60
#define OS_TASK_DEL_IDLE         61
#define OS_TASK_DEL_REQ          62
#define OS_TASK_DEL_ISR          63

#define OS_NO_MORE_TCB           70

#define OS_TIME_NOT_DLY          80
#define OS_TIME_INVALID_MINUTES  81
#define OS_TIME_INVALID_SECONDS  82
#define OS_TIME_INVALID_MILLI    83
#define OS_TIME_ZERO_DLY         84

#define OS_TASK_SUSPEND_PRIO     90
#define OS_TASK_SUSPEND_IDLE     91

#define OS_TASK_RESUME_PRIO     100
#define OS_TASK_NOT_SUSPENDED   101

#define OS_MEM_INVALID_PART     110
#define OS_MEM_INVALID_BLKS     111
#define OS_MEM_INVALID_SIZE     112
#define OS_MEM_NO_FREE_BLKS     113
#define OS_MEM_FULL             114

#define OS_TASK_OPT_ERR         130
```

```
/*$PAGE*/
/*     EVENT CONTROL BLOCK    */
#if (OS_MAX_EVENTS >= 2)
typedef struct {
    void   *OSEventPtr;     //Pointer to message or queue structure
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE]; //List of tasks waiting for event to
                                           //occur
    INT16U  OSEventCnt;             //Count of used when event is a semaphore
    INT8U   OSEventType;            //OS_EVENT_TYPE_MBOX, OS_EVENT_TYPE_Q or
                                    //OS_EVENT_TYPE_SEM
    INT8U   OSEventGrp;  //Group corresponding to tasks waiting for event to occur
} OS_EVENT;
#endif

/*$PAGE*/
/*    MESSAGE MAILBOX DATA    */
#if OS_MBOX_EN
typedef struct {
    void   *OSMsg;                          //Pointer to message in mailbox
    INT8U   OSEventTbl[OS_EVENT_TBL_SIZE]; //List of tasks waiting for event to
                                           //occur
    INT8U   OSEventGrp;  //Group corresponding to tasks waiting for event to occur
} OS_MBOX_DATA;
#endif

/*    MEMORY PARTITION DATA STRUCTURES   */
#if OS_MEM_EN && (OS_MAX_MEM_PART >= 2)
typedef struct {                        //MEMORY CONTROL BLOCK
    void   *OSMemAddr;                  //Pointer to beginning of memory partition
    void   *OSMemFreeList;              //Pointer to list of free memory blocks
    INT32U  OSMemBlkSize;               //Size (in bytes) of each block of memory
    INT32U  OSMemNBlks;                 //Total number of blocks in this partition
    INT32U  OSMemNFree;      //Number of memory blocks remaining in this partition
} OS_MEM;

typedef struct {
   void  *OSAddr;     //Pointer to the beginning address of the memory partition
   void  *OSFreeList; //Pointer to the beginning of the free list of memory blocks
   INT32U  OSBlkSize;           //Size (in bytes) of each memory block
   INT32U  OSNBlks;             //Total number of blocks in the partition
   INT32U  OSNFree;             //Number of memory blocks free
   INT32U  OSNUsed;             //Number of memory blocks used
} OS_MEM_DATA;
#endif

/*$PAGE*/
/*      MESSAGE QUEUE DATA   */
#if OS_Q_EN
typedef struct {
   void   *OSMsg;       //Pointer to next message to be extracted from queue
   INT16U  OSNMsgs;     //Number of messages in message queue
   INT16U  OSQSize;     //Size of message queue
   INT8U   OSEventTbl[OS_EVENT_TBL_SIZE];  //List of tasks waiting for event to
                                           //occur
   INT8U   OSEventGrp;  //Group corresponding to tasks waiting for event to occur
} OS_Q_DATA;
#endif

/*    SEMAPHORE DATA   */
#if OS_SEM_EN
typedef struct {
  INT16U  OSCnt;                           //Semaphore count
  INT8U   OSEventTbl[OS_EVENT_TBL_SIZE];  //List of tasks waiting for event to
```

```
                                          //occur
  INT8U   OSEventGrp;   //Group corresponding to tasks waiting for event to occur
} OS_SEM_DATA;
#endif

/*   TASK STACK DATA   */
#if OS_TASK_CREATE_EXT_EN
typedef struct {
    INT32U  OSFree;                    //Number of free bytes on the stack
    INT32U  OSUsed;                    //Number of bytes used on the stack
} OS_STK_DATA;
#endif

/*$PAGE*/
/*     TASK CONTROL BLOCK  */
typedef struct os_tcb {
    OS_STK        *OSTCBStkPtr;        //Pointer to current top of stack

#if OS_TASK_CREATE_EXT_EN
  void         *OSTCBExtPtr; //Pointer to user definable data for TCB extension
  OS_STK       *OSTCBStkBottom;     //Pointer to bottom of stack
  INT32U        OSTCBStkSize;       //Size of task stack (in bytes)
  INT16U        OSTCBOpt;           //Task options as passed by OSTaskCreateExt()
  INT16U         OSTCBId;           //Task ID (0..65535)
#endif

    struct os_tcb *OSTCBNext;  //Pointer to next     TCB in the TCB list
    struct os_tcb *OSTCBPrev;  //Pointer to previous TCB in the TCB list

#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN || OS_SEM_EN
    OS_EVENT      *OSTCBEventPtr;      //Pointer to event control block
#endif

#if (OS_Q_EN && (OS_MAX_QS >= 2)) || OS_MBOX_EN
    void          *OSTCBMsg;     //Message received from OSMboxPost() or OSQPost()
#endif

  INT16U     OSTCBDly;  //Nbr ticks to delay task or, timeout waiting for event
  INT8U      OSTCBStat;          //Task status
  INT8U      OSTCBPrio;          //Task priority (0 == highest, 63 == lowest)

  INT8U      OSTCBX; //Bit position in group corresponding to task priority (0..7)
  INT8U      OSTCBY;      //Index into ready table corresponding to task priority
  INT8U      OSTCBBitX;   //Bit mask to access bit position in ready table
  INT8U      OSTCBBitY;   //Bit mask to access bit position in ready group

#if OS_TASK_DEL_EN
    BOOLEAN    OSTCBDelReq;  //Indicates whether a task needs to delete itself
#endif
} OS_TCB;

/*$PAGE*/
/*    GLOBAL VARIABLES    */
OS_EXT  INT32U       OSCtxSwCtr;     //Counter of number of context switches

#if     (OS_MAX_EVENTS >= 2)
OS_EXT  OS_EVENT    *OSEventFreeList;  //Pointer to list of free EVENT control
                                       //blocks
OS_EXT  OS_EVENT     OSEventTbl[OS_MAX_EVENTS];//Table of EVENT control blocks
#endif

OS_EXT  INT32U       OSIdleCtr;         //Idle counter
```

```
#if     OS_TASK_STAT_EN
OS_EXT  INT8S   OSCPUUsage;        //Percentage of CPU used
OS_EXT  INT32U  OSIdleCtrMax; //Maximum value that idle counter can take in 1 sec.
OS_EXT  INT32U  OSIdleCtrRun; //Value reached by idle counter at run time in 1sec.
OS_EXT  BOOLEAN  OSStatRdy;   //Flag indicating that the statistic task is ready
#endif

OS_EXT  INT8U        OSIntNesting;             //Interrupt nesting level
OS_EXT  INT8U        OSLockNesting;            //Multitasking lock nesting level
OS_EXT  INT8U        OSPrioCur;                //Priority of current task
OS_EXT  INT8U        OSPrioHighRdy;            //Priority of highest priority task
OS_EXT  INT8U        OSRdyGrp;                 //Ready list group
OS_EXT  INT8U        OSRdyTbl[OS_RDY_TBL_SIZE]; //Table of tasks which are ready
                                                //to run
OS_EXT  BOOLEAN      OSRunning;           //Flag indicating that kernel is running

#if     OS_TASK_CREATE_EN  || OS_TASK_CREATE_EXT_EN || OS_TASK_DEL_EN
OS_EXT  INT8U        OSTaskCtr;                      //Number of tasks created
#endif

OS_EXT  OS_TCB      *OSTCBCur;                //Pointer to currently running TCB
OS_EXT  OS_TCB      *OSTCBFreeList;           //Pointer to list of free TCBs
OS_EXT  OS_TCB      *OSTCBHighRdy;  //Pointer to highest priority TCB ready to run
OS_EXT  OS_TCB      *OSTCBList;     //Pointer to doubly linked list of TCBs
OS_EXT  OS_TCB      *OSTCBPrioTbl[OS_LOWEST_PRIO + 1];//Table of pointers to
                                                      //created TCBs

OS_EXT  INT32U       OSTime;           //Current value of system time (in ticks)

extern  INT8U const  OSMapTbl[];       //Priority->Bit Mask lookup table
extern  INT8U const  OSUnMapTbl[];     //Priority->Index    lookup table

/*$PAGE*/
/*         FUNCTION PROTOTYPES
     (Target Independant Functions) */

/*      MESSAGE MAILBOX MANAGEMENT  */
#if         OS_MBOX_EN
void       *OSMboxAccept(OS_EVENT *pevent);
OS_EVENT   *OSMboxCreate(void *msg);
void       *OSMboxPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
INT8U       OSMboxPost(OS_EVENT *pevent, void *msg);
INT8U       OSMboxQuery(OS_EVENT *pevent, OS_MBOX_DATA *pdata);
#endif

/*     MEMORY MANAGEMENT     */
#if         OS_MEM_EN && (OS_MAX_MEM_PART >= 2)
OS_MEM     *OSMemCreate(void *addr, INT32U nblks, INT32U blksize, INT8U *err);
void       *OSMemGet(OS_MEM *pmem, INT8U *err);
INT8U       OSMemPut(OS_MEM *pmem, void *pblk);
INT8U       OSMemQuery(OS_MEM *pmem, OS_MEM_DATA *pdata);
#endif

/*     MESSAGE QUEUE MANAGEMENT    */
#if         OS_Q_EN && (OS_MAX_QS >= 2)
void       *OSQAccept(OS_EVENT *pevent);
OS_EVENT   *OSQCreate(void **start, INT16U size);
INT8U       OSQFlush(OS_EVENT *pevent);
void       *OSQPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
INT8U       OSQPost(OS_EVENT *pevent, void *msg);
INT8U       OSQPostFront(OS_EVENT *pevent, void *msg);
INT8U       OSQQuery(OS_EVENT *pevent, OS_Q_DATA *pdata);
#endif
```

```
/*$PAGE*/
/*     SEMAPHORE MANAGEMENT   */
#if         OS_SEM_EN
INT16U      OSSemAccept(OS_EVENT *pevent);
OS_EVENT   *OSSemCreate(INT16U value);
void        OSSemPend(OS_EVENT *pevent, INT16U timeout, INT8U *err);
INT8U       OSSemPost(OS_EVENT *pevent);
INT8U       OSSemQuery(OS_EVENT *pevent, OS_SEM_DATA *pdata);
#endif

/*    TASK MANAGEMENT   */
#if         OS_TASK_CHANGE_PRIO_EN
INT8U       OSTaskChangePrio(INT8U oldprio, INT8U newprio);
#endif

INT8U   OSTaskCreate(void (*task)(void *pd),void *pdata,OS_STK *ptos,INT8U prio);

#if         OS_TASK_CREATE_EXT_EN
INT8U       OSTaskCreateExt(void  (*task)(void *pd),
                            void   *pdata,
                            OS_STK *ptos,
                            INT8U   prio,
                            INT16U  id,
                            OS_STK *pbos,
                            INT32U  stk_size,
                            void   *pext,
                            INT16U  opt);
#endif

#if         OS_TASK_DEL_EN
INT8U       OSTaskDel(INT8U prio);
INT8U       OSTaskDelReq(INT8U prio);
#endif

#if         OS_TASK_SUSPEND_EN
INT8U       OSTaskResume(INT8U prio);
INT8U       OSTaskSuspend(INT8U prio);
#endif

#if         OS_TASK_CREATE_EXT_EN
INT8U       OSTaskStkChk(INT8U prio, OS_STK_DATA *pdata);
#endif

INT8U       OSTaskQuery(INT8U prio, OS_TCB *pdata);

/*     TIME MANAGEMENT   */
void     OSTimeDly(INT16U ticks);
INT8U    OSTimeDlyHMSM(INT8U hours, INT8U minutes, INT8U seconds, INT16U milli);
INT8U    OSTimeDlyResume(INT8U prio);
INT32U   OSTimeGet(void);
void     OSTimeSet(INT32U ticks);
void     OSTimeTick(void);

/*      MISCELLANEOUS   */

void        OSInit(void);

void        OSIntEnter(void);
void        OSIntExit(void);

void        OSSchedLock(void);
void        OSSchedUnlock(void);
```

```
void        OSStart(void);

void        OSStatInit(void);

INT16U      OSVersion(void);

/*$PAGE*/
/*    INTERNAL FUNCTION PROTOTYPES
  (Your application MUST NOT call these functions)  */

#if         OS_MBOX_EN || OS_Q_EN || OS_SEM_EN
void        OSEventTaskRdy(OS_EVENT *pevent, void *msg, INT8U msk);
void        OSEventTaskWait(OS_EVENT *pevent);
void        OSEventTO(OS_EVENT *pevent);
void        OSEventWaitListInit(OS_EVENT *pevent);
#endif

#if         OS_MEM_EN && (OS_MAX_MEM_PART >= 2)
void        OSMemInit(void);
#endif

#if         OS_Q_EN
void        OSQInit(void);
#endif

void        OSSched(void);

void        OSTaskIdle(void *data);

#if         OS_TASK_STAT_EN
void        OSTaskStat(void *data);
#endif

INT8U  OSTCBInit(INT8U prio,OS_STK *ptos,OS_STK *pbos,INT16U id,INT16U stk_size,
                           void *pext, INT16U opt);

/*$PAGE*/
/*     FUNCTION PROTOTYPES
     (Target Specific Functions)   */
void     OSCtxSw(void);
void     OSIntCtxSw(void);
void     OSStartHighRdy(void);
void     OSTaskCreateHook(OS_TCB *ptcb);
void     OSTaskDelHook(OS_TCB *ptcb);
void     OSTaskStatHook(void);
void    *OSTaskStkInit(void (*task)(void *pd),void *pdata,void *ptos,INT16U opt);
void     OSTaskSwHook(void);
void        OSTickISR(void);
void     OSTimeTickHook(void);
```

Приложение 3

Пример

Синхронизация потоков в операционной системе eCos **(*Embedded Configurable Operating System*).**

В этой системе вообще отсутствует понятие «задача», как и понятие процесс. Вместо них используется понятие «управляющий поток» – thread.

eCos two-threaded program listing (***Компилируется с помощью*** GCC **(*General Cross Compiler*).**

```
#include <cyg/kernel/kapi.h>
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
/* Now declare (and allocate space for) some kernel objects,
 like the two threads we will use */
cyg_thread thread_s[2];/* space for two thread objects */
char stack[2][4096];   /* space for two 4K stacks */

/* Now the handles for the threads */
cyg_handle_t simple_threadA, simple_threadB;

/* and now variables for the procedure which is the thread */
cyg_thread_entry_t simple_program;

/* and now a mutex to protect calls to the C library */
cyg_mutex_t cliblock;

/* We install our own startup routine which sets up threads */
void cyg_user_start(void)
{
 printf("Entering twothreads' cyg_user_start() function\n");

 cyg_mutex_init(&cliblock);

 cyg_thread_create(4, simple_program, (cyg_addrword_t) 0,
    "Thread A", (void *) stack[0], 4096,
    &simple_threadA, &thread_s[0]);
 cyg_thread_create(4, simple_program, (cyg_addrword_t) 1,
    "Thread B", (void *) stack[1], 4096,
    &simple_threadB, &thread_s[1]);

 cyg_thread_resume(simple_threadA);
 cyg_thread_resume(simple_threadB);
}
```

```
/* This is a simple program which runs in a thread */
void simple_program(cyg_addrword_t data)
{
 int message = (int) data;
 int delay;

 printf("Beginning execution; thread data is %d\n", message);

 cyg_thread_delay(200);

 for (;;) {
  delay = 200 + (rand() % 50);

  /* Note: printf() must be protected by a
     call to cyg_mutex_lock() */
  cyg_mutex_lock(&cliblock);
  printf("Thread %d: and now a delay of %d clock ticks\n",
           message, delay);
  cyg_mutex_unlock(&cliblock);
  cyg_thread_delay(delay);
  }
}
```

When you run the program (by typing `run` at the (**gdb** – *General Developer Board* ?) prompt the output should look like this:

----------Результат работы----------

```
Starting program: BASE_DIR/examples/twothreads.exe
Entering twothreads' cyg_user_start() function
Beginning execution; thread data is 0
Beginning execution; thread data is 1
Thread 0: and now a delay of 240 clock ticks
Thread 1: and now a delay of 225 clock ticks
Thread 1: and now a delay of 234 clock ticks
Thread 0: and now a delay of 231 clock ticks
Thread 1: and now a delay of 224 clock ticks
Thread 0: and now a delay of 249 clock ticks
Thread 1: and now a delay of 202 clock ticks
Thread 0: and now a delay of 235 clock ticks
```

NOTE

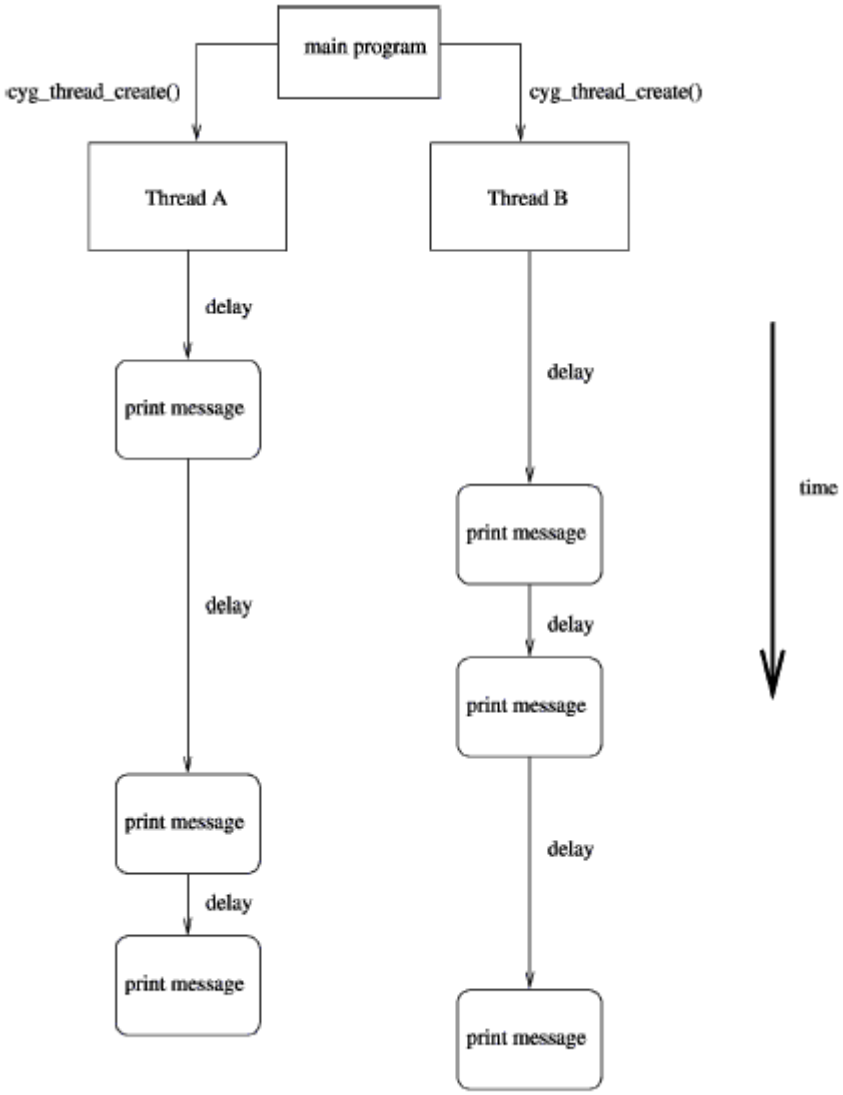
When running in a simulator the delays might be quite long. On a hardware board (where the clock speed is 100 ticks/second) the delays should average to about 2.25 seconds. In simulation, the delay will depend on the speed of the processor and will almost always be much slower than the actual board. You might want to reduce the delay parameter when running in simulation.

Two threads with simple print statements after random delays shows how this multitasking program executes. Note that apart from the thread creation system calls, this program also creates and uses a *mutex* for synchronization between the *printf()* calls in the two threads. This is because the C library standard I/O (by default) is configured not to be thread-safe, which means that if more than one thread is using standard I/O they might corrupt each other. This is fixed by a mutual exclusion (or *mutex*)

lockout mechanism: the threads do not call *printf()* until *cyg_mutex_lock()* has returned, which only happens when the other thread calls *cyg_mutex_unlock().*

You could avoid using the mutex by configuring the C library to be thread-safe (by selecting the component `CYGSEM_LIBC_STDIO_THREAD_SAFE_STREAMS`). Keep in mind that if the C library is thread-safe, you can no longer use *printf()* in *cyg_user_start().*

Two threads with simple print statements after random delays



A Sample Program with Alarms

`simple-alarm.c` (in the examples directory) is a short program that creates a thread that creates an alarm. The alarm is handled by the function `test_alarm_func()`, which sets a global variable. When the main thread of execution sees that the variable has changed, it prints a message.

A sample program that creates an alarm

```
 /* This is a very simple program meant to demonstrate
 a basic use of time, alarms and alarm-handling functions
 in eCos */
#include <cyg/kernel/kapi.h>
#include <stdio.h>

#define NTHREADS 1
#define STACKSIZE 4096

static cyg_handle_t thread[NTHREADS];

static cyg_thread thread_obj[NTHREADS];
static char stack[NTHREADS][STACKSIZE];

static void alarm_prog( cyg_addrword_t data );

/* We install our own startup routine which sets up
 threads and starts the scheduler */
void cyg_user_start(void)
{
 cyg_thread_create(4, alarm_prog, (cyg_addrword_t) 0,
    "alarm_thread", (void *) stack[0],
    STACKSIZE, &thread[0], &thread_obj[0]);
 cyg_thread_resume(thread[0]);
}

/* We need to declare the alarm handling function (which is
 defined below), so that we can pass it to
 cyg_alarm_initialize() */
cyg_alarm_t test_alarm_func;

/* alarm_prog() is a thread which sets up an alarm which is then
 handled by test_alarm_func() */
static void alarm_prog(cyg_addrword_t data)
{
 cyg_handle_t test_counterH, system_clockH, test_alarmH;
 cyg_tick_count_t ticks;
 cyg_alarm test_alarm;
 unsigned how_many_alarms = 0, prev_alarms = 0, tmp_how_many;

 system_clockH = cyg_real_time_clock();
```

```
 cyg_clock_to_counter(system_clockH, &test_counterH);
 cyg_alarm_create(test_counterH, test_alarm_func,
    (cyg_addrword_t) &how_many_alarms,
    &test_alarmH, &test_alarm);
 cyg_alarm_initialize(test_alarmH, cyg_current_time()+200, 200);

 /* Get in a loop in which we read the current time and
 print it out, just to have something scrolling by */
 for (;;) {
  ticks = cyg_current_time();
  printf("Time is %llu\n", ticks);
  /* Note that we must lock access to how_many_alarms, since the
  alarm handler might change it. This involves using the
  annoying temporary variable tmp_how_many so that I can keep the
  critical region short */
  cyg_scheduler_lock();
  tmp_how_many = how_many_alarms;
  cyg_scheduler_unlock();
  if (prev_alarms != tmp_how_many)
   { printf(" --- alarm calls so far: %u\n", tmp_how_many);
     prev_alarms = tmp_how_many;
   }
  cyg_thread_delay(30);
 }
}

/* Test_alarm_func() is invoked as an alarm handler, so
 it should be quick and simple. in this case it increments
 the data that is passed to it. */
void test_alarm_func(cyg_handle_t alarmH, cyg_addrword_t data)
{
 ++*((unsigned *) data);
}
```

When you run this program (by typing `run` at the (**gdb** – *General Developer Board* ?) prompt) the output should look like this:

----------Результат работы----------

```
Starting program: BASE_DIR/examples/simple-alarm.exe
Time is 0
Time is 30
Time is 60
Time is 90
Time is 120
Time is 150
Time is 180
Time is 210
  --- alarm calls so far: 1
Time is 240
Time is 270
```

```
Time is 300
Time is 330
Time is 360
Time is 390
Time is 420
  --- alarm calls so far: 2
Time is 450
Time is 480
```

NOTE

When running in a simulator the delays might be quite long. On a hardware board (where the clock speed is 100 ticks/second) the delays should average to about 0.3 seconds (and 2 seconds between alarms). In simulation, the delay will depend on the speed of the processor and will almost always be much slower than the actual board. You might want to reduce the delay parameter when running in simulation.

Here are a few things you might notice about this program:

- It used the *cyg_real_time_clock();* this always returns a handle to the default system real-time clock.
- Alarms are based on counters, so the function *cyg_alarm_create()* uses a counter handle. The program used the function *cyg_clock_to_counter()* to strip the clock handle to the underlying counter handle.
- Once the alarm is created it is initialized with *cyg_alarm_initialize(),* which sets the time at which the alarm should go off, as well as the period for repeating alarms. It is set to go off at the current time and then to repeat every 200 ticks.
- The alarm handler function *test_alarm_func()* conforms to the guidelines for writing alarm handlers and other *delayed service routines* : it does not invoke any functions which might lock the scheduler. This is discussed in detail in the *eCos Reference Manual*, in the chapter *Requirements for programs.*
- There is a *critical region* in this program: the variable *how_many_alarms* is accessed in the main thread of control and is also modified in the alarm handler. To prevent a possible (though unlikely) race condition on this variable, access to *how_many_alarms* in the principal thread is protected by calls to *cyg_scheduler_lock()* and *cyg_scheduler_unlock().* When the scheduler is locked, the alarm handler will not be invoked, so the problem is averted.