Ассемблер

3

3.1. Введение

Ассемблер семейства ADSP-2101 переводит вашу программу, написанную на языке ассемблер, в объектный код. Часть программы на языке ассемблер называют модулем; модули, которые являются входными данными ассемблера, называются исходными модулями. Каждый исходный модуль должен содержаться в отдельном файл. Отдельно ассемблированные модули связываются в единую исполняемую программу.

Ваш исходный код может быть записан на языке ассемблер ADSP-2101 или создан С компилятором семейства ADSP-2101. Для определения переменных, буферов данных и макросов используют директивы ассемблера. Создать исходный кода можно с помощью любого простого текстового редактора. Не используйте текстовые процессоры, например, Microsoft Word, которые вставляют специальные управляющие коды.

В этой главе рассмотрены различные методы программирования. Дополнительную информацию и примеры программ можно найти в разделах «Использование библиотечных файлов ваших подпрограмм» и «Системы с многостраничной памятью начальной загрузки» главы 4 «Редактор связей».

Рис.3.1 (на следующей странице) показывает входные и выходные файлы ассемблера. Ассемблер читает входной файл и создает четыре типа выходных файлов: объектный файл (*object file*) `.OBJ`, файл кода (*code file*) `.CDE`, файл листинга или сообщений транслятора (*list file*) `.LST`, файл инициализации (*initialization file*) `.INT`.

Объектный файл содержит информацию о размещении памяти и символьных определениях. Размещение памяти это процесс, в котором редактор связей решает, где сохранить ваш код программы и фрагменты данных. Файл кода содержит инструкции кодов операций ADSP-21xx с помеченными неразрешенными символами. Файлы инициализации содержат данные для инициализации буферов данных. Файл листинга предоставляет дополнительную информацию, помогая понять и документировать процесс ассемблирования.

(**Примечание**: так как директива ассемблера `.VAR` используется для объявления как однословных переменных данных, так и многословных буферов данных, термин «*буфер данных*» используется для обозначения как переменных, так и буферов.)

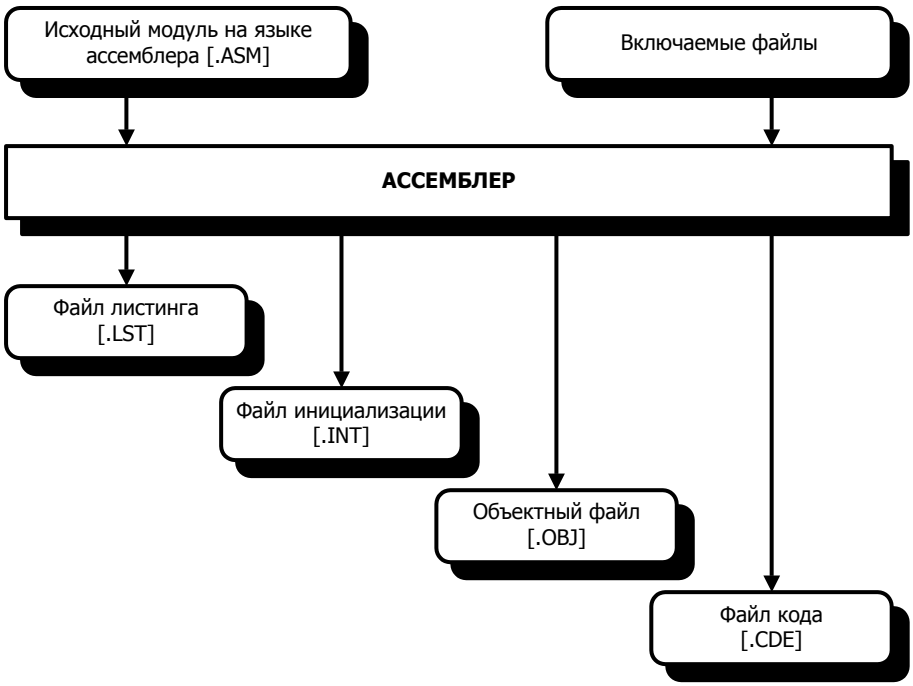


Рис.3.1. Вход/выход ассемблера.

3.2. Препроцессоры ассемблера

Ассемблер включает три исполняемых компонента:

- Препроцессор языка С
- Препроцессор ассемблера
- Ядро ассемблера

Два препроцессора ассемблера являются препроцессором языка С ANSI-стандарта и препроцессором ассемблера. Препроцессор языка С обрабатывает директивы С, например, `#define` и `#include`. Препроцессор ассемблера обрабатывает директивы ассемблера ADSP-21xx, например, `.MODULE` и `.VAR`. Рис.3.2 показывает последовательность процесса выполнения ассемблера.

Препроцессор ассемблера языка С позволяет использовать в ассемблерном коде директивы С препроцессора, например, `#include`. Препроцессор С обрабатывает эти директивы таким же образом, как это делает препроцессор компилятора. Для примера использования этой возможности смотрите раздел «Использование С препроцессора» .

Заметьте, что препроцессор С не допускает комментарии, заключенные в скобки `{ }`. Чтобы разместить комментарии на строке с директивой С препроцессора, используйте комментарии по правилам языка С:

```
#директива              /*комментарий*/
```

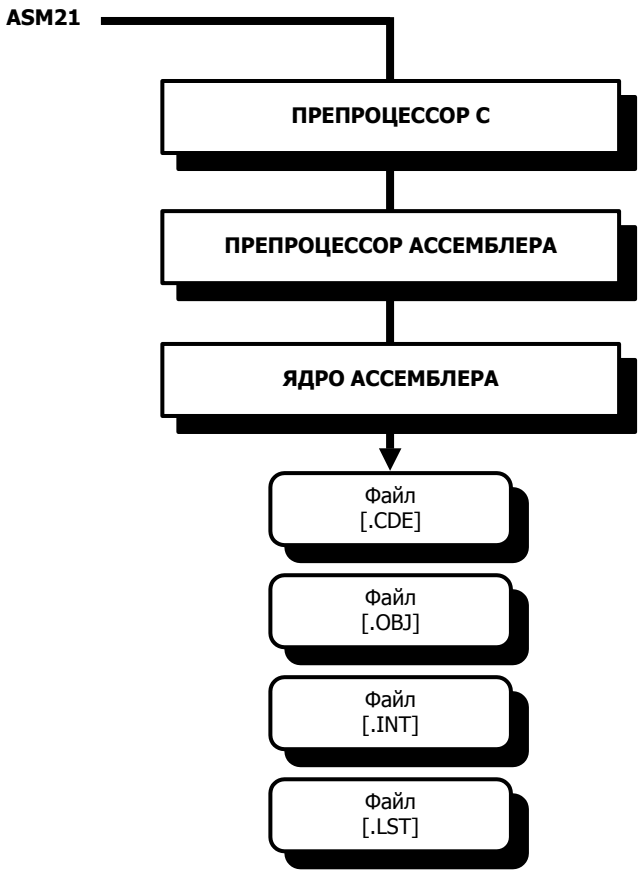


Рис.3.2. Процесс выполнения ассемблера.

3.3. Запуск ассемблера

Чтобы запустить ассемблер из вашей операционной системы, введите в командной строке:

```
ASM21 filename  [.ext][-switch ...]
```

Где, `filename` входной файл, содержащий ассемблерный модуль исходного кода. Имя файла может иметь некоторое расширение; если его нет, то по умолчанию ассемблер добавляет расширение `.dsp`.

Добавление ключей управляет работой ассемблера. Они могут быть введены в верхнем или в нижнем регистре. Составные ключи должны быть разделены не менее чем одним пробелом.

Если вы забыли синтаксис командной строки системного конфигуратора, наберите:

```
ASM21 -help
```

Этот ключ позволяет просмотреть список возможных команд. Ключ `-help` работает во всех программах средств разработки.

Если вы ассемблируете код, созданный С компилятором ADSP-21xx, ассемблер должен быть запущен с ключами `-c` и `-s`. Поскольку компилятор и средства разработки С чувствительны к регистру символов, ключ `-c` делать ассемблер также чувствительным к регистру.

Применение ключа `-s` обусловлено тем, что компилятор может генерировать многофункциональные инструкции, которые не являются обычными, логически упорядоченными. Эти инструкции, тем не менее, будут правильно ассемблироваться и выполняться. Для получения дополнительной информации обратитесь к *ADSP-2100 Family C Tools Manual* и *ADSP-2100 Family C Runtime Library Manual*.

3.3.1. Ключи ассемблера

Список ключей ассемблера приведен в таблице 3.1; приведены также некоторые обязательные аргументы:

Ключ	*Воздействие*
`-c`	устанавливает чувствительность к регистру символов
`-d`*ид* `[=`*символ*`]`	определяет идентификатор для С препроцессора
`-i [`*глубина*`]`	раскрывает содержимое включенных файлов в листинге
`-l`	создает файл листинг
`-m [`*глубина*`]`	раскрывает макросы в файле листинга
`-o` *имя_файла*	переименование выходного файла
`-s`	отменяет семантическую проверку многофункциональных инструкций
`-215g`	специальная ассемблерная инструкция для процессоров ADSP-21msp5x
`-2171`	специальные ассемблерные инструкции для процессоров ADSP-217x
`-2181`	специальные ассемблерные инструкции для процессоров ADSP-217x

Таблица 3.1. Ключи ассемблера.

3.3.2. Чувствительность к регистру (-c)

Ключ `-c` делает ассемблер чувствительным к регистру символов, в основном для совместимости с кодом, созданным С компилятором семейства ADSP-2100. Если ключ `-c` не используется, ассемблер не сможет различить символы верхнего и нижнего регистров и все символы будут выводиться в верхнем регистре (прописные). Вы должны использовать этот ключ для сохранения символов нижнего регистра (строчные). Если вы ассемблируете код, произведенный компилятором, вы должны использовать ключ `-c`, когда вызываете ассемблер (обычно компилятор сам вызывает ассемблер, используя ключ `-c` в вызове).

3.3.3. Определение идентификатора (-d)

Ключ `-d` определяет идентификатор в стиле С для препроцессора ассемблера таким же образом, как директива `#define`. Ключ применяют следующим образом:

`-d`*идентификатор* `[=`*символ*`]`

Идентификатор это строка символов и цифр, определенная по стандарту C. Идентификатор может быть установлен равным константе или строчному литералу. Пример использования ключа `-d`:

```
-Djunk
-dten=10
-dname="Jake"
```

Один способ использования ключа `-d` в ассемблерном коде:

```
     CNTR=n;
     Do this_loop UNTIL CE;
                . . .
this_loop:      . . .           {последняя инструкция цикла}
```

Теперь вы можете выбрать величину для счетчика циклов `n` при вызове ассемблера. Положим, что `this_loop` находится во входном файле `source_file`:

```
asm21 source_file -dn=100
```

Для примера использования ключа `–d` при реализации условного ассемблирования смотрите раздел «Использование C препроцессора».

3.3.4. Раскрытие включаемых файлов в листинге (-i)

Ключ `-i [`*`глубина`*`]` раскрывает включаемые файлы, указанные директивой ассемблера `.INCLUDE`, в выходном файле листинга `.LST`. Параметр `[`*`глубина`*`]` определяет глубину раскрытия включаемых файлов. Если параметр не задан, то по умолчанию в файле листинга будут изложены все вложенные файлы. Пример использования ключа `-i`:

```
-i
-i 3
```

Если ключ `-i` не используется, то в файле листинга будет записана информация в виде «`.INCLUDE filename`». Для уточнения деталей смотрите раздел «Включение других исходных файлов» этой главы.

3.3.5. Раскрытие макросов в файле листинге (-m)

Ключ `–m [`*`глубина`*`]` вызывает раскрытие макросы в файле листинга `.LST`. Это означает, что будет показаны все инструкции, заключенные в макросе. Параметр `[глубина]` определяет глубину вложения макровызовов, которые должны быть раскрыты. Если параметр не задан, то все вложенные вызовы макросов будут раскрыты в файле листинга. Пример использования ключа `-m`:

```
-m
-m 2
```

Если ключ `-m` не используется, то в файле листинга будет записана информация в виде вызова «`macroname`». Для уточнения деталей смотрите раздел «Макросы» этой главы.

3.3.6. Создание файла листинга (-l)

Ключ `-l` вызывает создание ассемблером файла листинга `.LST`. Этот файл приводит адреса и коды операций, позволяющие интерпретировать результаты ассемблирования. Формат файла листинга `.LST` описан в разделе «Формат файла листинга» в конце этой главы

3.3.7. Переименование выходного файла (-o)

Ключ `-o` может использоваться для переименования выходных файлов ассемблера (`.OBJ`, `.CDE`, `.INT`, `.LST`). Например, если вы ввели имя файла `source1.dsp` и хотите переименовать выходные файлы в `src1.obj`, `src1.cde`, `src1.int`, `src1.lst`, вызывайте ассемблер следующим образом:

```
asm21 source1 -o src1
```

3.3.8. Отмена семантической проверки (-s)

Ключа `-s` освобождает ассемблер от проверки семантики (предписывающих условий) многофункциональных инструкций в вашей программе. Многофункциональные инструкции ADSP-21XX могут иметь составные операторы, которые имеют логический порядок слева направо. Операторы могут, тем не менее, быть написаны в другом порядке и все же ассемблироваться корректно. Если это случается, ассемблер обычно предупреждает об ошибке; ключ `-s` подавляет эти предупреждения.

До тех пор пока корректны отдельные операторы многофункциональной инструкции, будет генерироваться соответствующий код операции, несмотря на порядок, в котором они (операторы) даны. Предупреждения ассемблерные об ошибке указывают на то, что выходные инструкции могут появиться в искаженном виде.

3.4. Правила языка ассемблер

Эта секция описывает правила специфичные для языка ассемблер. Соглашения и правила, касающиеся систем счисления, символов и набора символов были рассмотрены в главе 1.

3.4.1. Символы и ключевые слова

Символы и ключевые слова составляют символьные строки. Символы - это строка, которая определяется на языке ассемблер: имя может быть названием модуля, переменной, буфером данных, меткой, портом ввода/вывода, макросом или константой. Имя может быть длиной до 32 символов и не может начинаться с цифры. Пример типичных имен:

```
main_prog           имя программного модуля
xoperand            переменная
input_array         буфер данных
subroutinel         метка
AD_INPUT            порт отраженный в памяти
```

Символы могут быть введены в любой комбинации знаков верхнего и нижнего регистра, но ассемблер будет преобразовывать все символы в верхний регистр, если не используется ключ `-c`.

Одинаковые символы могут быть объявлены и использоваться отдельно в разных модулях. Символы распознаются только в пределах границ, определяемых модулем - пока они не объявлены как `GLOBAL` или `ENTRY`. Такой тип символов может быть связан со всеми модулями и должен быть определен только в одном. Смотрите описание директив ассемблера `.GLOBAL`, `.ENTRY` и `.EXTERNAL` далее в этой главе.

Таблица 3.2 содержит список зарезервированных ключевых слов ассемблера. Вы не должны использовать ключевые слова в качестве символов в вашем коде. Так как ассемблер по умолчанию не различает регистров печати, обе версии верхнего и нижнего регистров ключевых слов резервируются.

ABS	DM	INCLUDE	MR0	RTS
AC	DO	INIT	MR1	RX0
AF	EMODE	JUMP	MR2	RX1
ALT_REG	ENA	L0	MSTAT	SAT
AND	ENDMACRO	L1	MV	SB
AR	ENDMOD	L2	MX0	SEG
AR_SAT	ENTRY	L3	MX1	SEGMENT
ASHIFT	EQ	L4	MY0	SET
ASTAT	EXP	L5	MY1	SHIFT
AUX	EXPADJ	L6	NAME	SI
AV	EXTERNAL	L7	NE	SR
AV_LATCH	FOREVER	LE	NEG	SR0
AX0	FLAG_IN	LOCAL	NEWPAGE	SR1
AX1	FLAG_OUT	LOOP	NOP	SS
AY0	GE	LSHIFT	NORM	SSTAT
AY1	GLOBAL	LT	NOT	STATIC
BIT_REV	GT	M0	OR	STS
BM	I0	M1	PASS	SU
BY	I1	M2	PC	TEST
C	I2	M3	PM	TIMER
CACHE	I3	M4	POP	TOGGLE
CALL	I4	M5	PORT	TOPPCSTACK
CE	I5	M6	POS	TRAP
CIRC	I6	M7		TRUE
CLR	I7	MACRO	PUSH	TX1
CLEAR	ICTRL	MF	RAM	TX0
CNTR	IDLE	M_MODE	REGBANK	UNTIL
CONST	IF	GO_MODE	RESET	US
DIS	IFC	MODIFY	RND	UU
DIVS	IMASK	MODULE	ROM	VAR
DIVQ		MR	RTI	XOR

Таблица 3.2. Ключевые слова зарезервированные ассемблером.

3.4.2. Выражения ассемблера

Ассемблер семейства ADSP-2100 может вычислять простые выражения в исходном коде., которое может быть использовано вместо числовой величины. Разрешены два вида выражений:

- арифметические или логические операции над двумя или более целыми константами.

  пример: `29+129` `(128-48)*3` `0x55&0x0F`

- символ плюс или минус целая константа

  пример: `data-8` `data_buffer+15` `startup+2`

Символ может быть как переменной, так и буфером данных, или программной меткой. Все символы реально представляют собой адресные величины, которые определяются редактором связей. Увеличение или уменьшение константы указывает на смещение адреса.

(**Примечание**: так как директива ассемблера `.VAR` используется для объявления как однословных переменных данных, так и многословных буферов данных, термин «*буфер данных*» используется для обозначения как переменных, так и буферов.)

Простые арифметические или логические выражения могут быть использованы для объявления символьных констант с помощью директивы системного конфигуратора и ассемблера `.CONST`. Эти выражения могут использовать выражения, которые могут составлять следующий набор операторов, распознаваемый средствами языка С:

```
( )             левая, правая скобки
~ -             поразрядное дополнение, унарный минус
* / %           умножение, деление, модуль
+ -             сложение, вычитание
<< >>           битовые сдвиги
&               поразрядная операция AND
|               поразрядная операция OR
^               поразрядная операция XOR
```

Выражения могут использоваться также для ввода команды одну из программ моделирования семейства ADSP-2100. Симуляторы распознают дополнительный набор выражений и операторов.

Самое важное различие между выражениями ассемблера и выражениями симулятора состоит в том, что содержимое памяти (например, переменные), и содержимое регистров процессора может использоваться как операнд, *только в симуляторе*.

Ассемблер не может преобразовать числовые выражения в памяти и регистровые величины во время ассемблирования, в то время как симулятор имеет доступ к мгновенным величинам моделирующих состояние памяти и регистров .

3.4.3 Адрес буфера и операторы определения длины

Ассемблер распознает два специальных оператора. Операторы *определения указателя адреса* (*address pointer*) ^ и *определения длины* (*length of*) % применяются к имени буфера:

```
^buffer_name     преобразуется в значение базового адреса буфера
%buffer_name     преобразуется в значения длины (число слов) буфера
```

Оператор ^ может применяться к переменными, которые представляют собой простейшие однопозиционные буфера:

```
^variable_name   преобразуется в значение базового адреса переменной
```

Сложением или вычитанием констант могут быть образованы простые выражения:

```
^buffer_name ± constant
%buffer_name ± constant
```

Например:

```
^array + 3
%array - 10
```

Операторы ассемблера используются для загрузки регистров L (длины) и I (индекса) при настройке циклического буфера:

```
      VAR/DM/RAM/CIRC real_date[n];     {n = число входных точек}

      I5 = ^real_data;                  {базовый адрес буфера }
      L5 = %real_data;                  {длина буфера}
      M4 = 1;                           {пост-модификация I5}

      CNTR = %real_data;                {счетчик = длине буфера}
      DO loop UNTIL CE;
            AX0 = DM(I5, M4);           {получение следующей точки}
            ....
            {обработка значения, сохраненного в AX0}
loop:       ...
```

Этот фрагмент программы инициализирует I5 и L5 базовым адресом и длиной относительно циклического буфера `real_data`. Длина буфера хранится в L5 и определяет циклический адрес возврата в начало буфера. Более подробную информацию об организации и работе циклических буферов вы найдете в разделе «Переменные и буферы данных» этой главы и в главе «Передача данных» руководства *ADSP-2100 Family User's Manual*.

3.4.4. Комментарии

Вы можете вставлять комментарии в любом месте исходного кода программы заключив их в скобки { }, за исключением строк директив С препроцессора. Применение вложенных комментариев не разрешена. Многострочные комментарии заключенные в одну пару скобок не должны включать знак «решетки» (#) в начале строки.

Директивы ассемблера могут содержать только однострочные комментарии; они не могут быть продолжены на следующей строке. Если требуется ввести больший комментарий, начните новое поле комментария на следующей строке. Заметьте, что С препроцессор не может воспринимать комментарии заключенные в скобки (в стиле ассемблера). Чтобы разместить комментарий на той же строке, где находится директива С препроцессора (начинающаяся с символа «#»), используйте правила С:

```
#директива              /* комментарий * /
```

3.5. Использование С препроцессора

Ассемблер ADSP-21xx включает С препроцессор, который позволяет использовать такие директивы, как `#define`, `#ifdef`, `#include`, и др. в вашей программе на ассемблере.

Препроцессор обрабатывает эти директивы, как и код включающий их (например, макрос созданный с директивой `#define`).

Ниже приведены директивы С препроцессора, которые могут быть использованы:

Директива	*Значение*
`#include`	Вставить текст из другого исходного файла
`#define`	Макроопределение
`#undef`	Отмена макроопределения
`#if`	Условно включаемый текст по величине выражения, которое задает константу
`#ifdef`	Условно включаемый текст, по наличию выбранного макроопределения
`#ifndef`	Условно включаемый текст, по отсутствию выбранного макроопределения
`#else`	Включение текста по альтернативной ветви выражений `#if`, `#ifdef` или `#ifndef`
`#endif`	Завершение включения условного текста

Эти директивы допускают различную технику программирования. Например, вы можете реализовать условное ассемблирование или определить макрос через использование директив С препроцессора `#ifdef` и `#define`. Примеры применений приведены ниже.

3.5.1. Пример условного ассемблирования

Ключ ассемблера `-d` предназначен для совместного использования С препроцессором. Этот ключ позволяет определить для препроцессора идентификатор, как при использовании директивы `#define`. Самое распространенное использование этой возможности заключено в применении условного ассемблирования, как показано в следующем примере.

Если часть ассемблерного кода предназначена только для отладки, она может быть включена в модуль вашей главной программы и при необходимости условно ассемблирована. Для реализации этого, отладочный код помещают внутри блока `#ifdef`, который обрабатывается препроцессором языка С.

Возьмите для примера следующий блок программы:

```
#ifdef debug          /* ассемблировать, если debug определено */
                ...
                ... отладочный код
                ...
#endif
```

Если *debug* определено с ключом `-d`, препроцессор C удалит директивы `#ifdef` и `#endif`, ассемблируя отладочный код в программный модуль. Если *debug* не определен, то препроцессор C удалит введенный блок до ассемблирования.

Для включения отладочного кода, ассемблер должен быть запущен следующим образом:

```
asm21 source_file  -ddebug
```

3.5.2. Пример макроса в стиле C

В то время как директива ассемблера `.MACRO` позволяет создавать макрос в вашем исходном коде (смотри раздел «Определение макросов» далее в этой главе), препроцессор C также может быть использован для определения макроса. Это осуществляется директивой `#define` в соответствии с правилами языка C:

```
#define  mac  MR = MR + MX0*MY0(RND)

AR  = AX1-AY1;
MY0 = AR;
MX0 = DM(I1,M0);
mac;
```

В приведенном примере макрос *mac* определен как составная инструкция ADSP-21xx. Макрос заменят эту инструкцию там, где он появляется в вашем исходном коде. Заметьте, что строка директивы `#define` не требует завершающей точки с запятой, в отличие от вызова макроса.

Макровызов не может не содержать дополнительных операторов программы (т.е. инструкций, директив препроцессора или других макровызовов) на той же строке исходного кода программы.

Вы можете передавать аргументы в макрос. Следующий пример показывает макрос, который копирует слово из памяти данных в память программ:

```
#define copy (src,dest) \
AX0 = DM(src); \
PM(dest)=AX0;
```

(Символ обратной косой черты показывает, что макроопределение продолжается на следующей строке). При вызове макроса с абсолютными адресами для *src* и *dest*, он выполняет копирование, использующее прямую адресацию:

```
copy (0×3F, 0×C0)
```

3.6. Написание программ

Оставшаяся часть этой главы рассказывает, как написать программу на языке ассемблер для семейства микропроцессоров ADSP-2100. Вы можете создавать ваши программы с помощью любого простого текстового редактора. Не используйте текстовых процессоров, например, Microsoft Word, которые вставляют специальные управляющие коды.

Программа состоит из инструкций языка ассемблер, директив ассемблера и директив С препроцессора. Этот раздел объясняет использование ассемблерных директив.

Дополнительные примеры программирования представлены в двух разделах главы 4: «Использование библиотечных файлов ваших подпрограмм» и «Системы с многостраничной памятью начальной загрузки». Примеры программирования различных приложений приведены в *Digital Signal Processing Applications Using the ADSP-2100 Family*.

3.6.1. Структура программы

Базовой единицей программы для ADSP-21xx является модуль. Программа состоит из одного или нескольких модулей, которые отдельно ассемблируются и затем связываются вместе.

Модуль определяется двумя директивами:

```
.MODULE имя_модуля;
...
.ENDMOD;
```

Каждый модуль должен содержаться в собственном файл; другими словами, только один модуль может находиться в одном файле. Каждый оператор внутри модуля может быть инструкцией, директивой или макровызовом.

Точка с запятой завершают каждый оператор. Программные метки размещаются в начале строки и завершаются двоеточием:

```
startup:   10=2;            {начало программы}
```

Отдельные строки в исходном файле должны быть длиной не более 200 символов.

3.6.2. Настройка регистров управления, отраженных в памяти

Все процессоры ADSP-2101, ADSP-2105, ADSP-2111 и ADSP-21msp50 имеют набор регистров управления отраженных (размещенных) в памяти, которые настраивают различные режимы работы процессора. Эти регистры расположены в скрытой части внутренней памяти данных на каждом кристалле. Это пространство памяти расположено в верхней части 1К внутренней памяти данных по адресу 0×3C00 - 0×3FFF.

Каждый регистр должен быть загружен в программе записью слова данных по соответствующему адресу. Адрес и формат регистров приведены в приложении Е.

Для доступа к регистрам управления может использоваться как косвенная, так и прямая адресация; тем не менее, для ясности, рекомендуется использовать прямую адресацию с применением символических имен регистров. Этот метод, вместе с соответствующими комментариями сделает ваши программы легкими для чтения и понимания. Рис. 3.3 показывает простой модуль подпрограммы, который именует и инициализирует 17 управляющих регистров ADSP-2101. Заметьте, что для определения символических имен адресов регистров используется директива ассемблера .CONST. Чтобы установить управляющие регистры, как показано в этом примере, вы должны вызвать подпрограмму в вашей программе следующей инструкцией:

```
CALL  init2101;
```

Таблица 3.3 показывает набор регистров управления отраженных в памяти ADSP-2101, адреса регистров в памяти данных и предлагаемые мнемонические символы для каждого из регистров. Таблицы 3.4, 3.5 и 3.6 приводят подобную информацию для других процессоров семейства ADSP-21xx.

Имя регистра	*Адрес DM*	*Символическое имя*
System Control Register	0x3FFF	Sys_Ctrl_Reg
Data Memory Wait State Control Register	0x3FFE	Dm_Wait_Reg
Timer Period	0x3FFD	Tperiod_Reg
Timer Count	0x3FFC	Tcount_Reg
Timer Scaling Factor	0x3FFB	Tscale_Reg
SPORT0 Multichannel Receive	0x3FFA	Sport0_Rx_Words1
Word Enable Register (32-bit)	0x3FF9	Sport0_Rx_Words0
SPORT0 Multichannel Transmit	0x3FF8	Sport0_Tx_Words1
Word Enable Register (32-bit)	0x3FF7	Sport0_Tx_Words0
SPORT0 Control Register	0x3FF6	Sport0_Ctrl_Reg
SPORT0 Serial Clock Divide Modulus	0x3FF5	Sport0_Sclkdiv
SPORT0 Rcv Frame Sync Divide Modulus	0x3FF4	Sport0_Rfsdiv
SPORT0 Autobuffer Control Register	0x3FF3	Sport0_Autobuf_Ctrl
SPORT1 Control Register	0x3FF2	Sport1_Ctrl_Reg
SPORT1 Serial Clock Divide Modulus	0x3FF1	Sport1_Sclkdiv
SPORT1 Rcv Frame Sync Divide Modulus	0x3FF0	Sport1_Rfsdiv
SPORT1 Autobuffer Control Register	0x3FEF	Sport1_Autobuf_Ctrl

Таблица 3.3. Управляющие регистры ADSP-2101 и предлагаемые символические имена.

Если вы примете предложенные символы, вам не потребуется выписывать объявления .CONST, как сделано в подпрограмме на Рис.3.3. Объявления по умолчанию приведены для вас в четырех файлах, включаемых с программным обеспечением средств разработки:

Имя файла	*Содержит объявления .CONST для символических адресов в*
DEF2101.H	Таблица 3.3.
DEF2105.H	Таблица 3.4.
DEF2111.H	Таблица 3.5.
DEF2150.H	Таблица 3.6.

Все, что вам необходимо сделать - это включить соответствующий файл в вашу исходную программу с помощью директивы .INCLUDE. Например::

```
.INCLUDE  <DEF2105.H>;
```

Если файл, который должен быть включен, находится в текущей директории вашей операционной системы, требуется только указать имя файла в скобках. Если файл находится в другой директории, вы должны указать путь к этой директории с именем файла (смотрите раздел «Включение других исходных файлов» далее в этой главе).

```
{ Настройка управляющих регистров последовательного порта и таймера ADSP-2105}.
{ DM(0x3FEF) - DM(0x3FFF) инициализируются за 35 тактов}

.MODULE/BOOT=0  Set_Up_2101_Ctrl_Regs;
.CONST  Sport1_Autobuf_Ctrl   =0x3FEF;
.CONST  Sport1_Rfsdiv         =0x3FF0;
.CONST  Sport1_Sclkdiv        =0x3FF1;
.CONST  Sport1_Ctrl_Reg       =0x3FF2;
.CONST  Sport0_Autobuf_Ctrl   =0x3FF3;
.CONST  Sport0_Rfsdiv         =0x3FF4;
.CONST  Sport0_Sclkdiv        =0x3FF5;
.CONST  Sport0_Ctrl_Reg       =0x3FF6;
.CONST  Sport0_Tx_Words0      =0x3FF7;
.CONST  Sport0_Tx_Words1      =0x3FF8;
.CONST  Sport0_Rx_Words0      =0x3FF9;
.CONST  Sport0_Rx_Words1      =0x3FFA;
.CONST  Tscale_Reg            =0x3FFB;
.CONST  Tcount_Reg            =0x3FFC;
.CONST  Tperiod_Reg           =0x3FFD;
.CONST  Dm_Wait_Reg           =0x3FFE;
.CONST  Sys_Ctrl_Reg          =0x3FFF;
.ENTRY  init2101;

init2101:
{============ установка регистров SPORT1 ============}
AX0=0;  DM (Sport1_Autobuf_Ctrl) =AX0;    {Автобуферизация отключена}
AX0=0;  DM (Sport1_Rfsdiv)       =AX0;    {RESDIV не используется}
AX0=0;  DM (Sport1_Sclkdiv)      =AX0;    {SCLKDIV не используется}
AXO=0;  DM (Sport1_Ctrl_Reg)     =AX0;    {функции Ctrl_Reg отключены}
{============ установка регистров SPORT0 ============}
AX0=0;  DM(Sport0_Autobuf_Ctrl)  =AX0;    {Автобуферизация отключена}
AX0=255;DM(Sport0_Rfsdiv)        =AXO;    {RFSDIV=255 для частоты 8 КГц}
AX0=2;  DM(Sport0_Sclkdiv)       =AXO;    {SCLKDIV=2 дает 2.048 МГц SCLK}

{Многоканальность отключена, SCLK внутр., RFS нужна, TFS нужна, нормальная
 кадровая синхронизация внутр. RFS, внутр. TFS, посл. данные u-law, 8 разр. ИКМ}
AX0=0x6B27; DM(Sport0_Ctrl_Reg)  =AXO;
AX0=0;  DM(Sport0_Tx_Words0)     =AX0;    {TX на TDM каналах 15-00}
AX0=0;  DM(Sport0_Tx_Words1)     =AX0;    {TX на TDM каналах 31-16}
AX0=0;  DM(Sport0_Rx_Words0)     =AX0;    {RX на TDM каналах 15-00}
AX0=0;  DM(Sport0_Rx_Words1)     =AX0;    {RX на TDM каналах 31-16}
{============ установка регистров таймера ============}
AX0=0;  DM(Tscale_Reg)           =AX0;    {таймер не используется}
AX0=0;  DM(Tcount_Reg)           =AX0;
AX0=0;  DM(Tperiod_Reg)          =AX0;
{============ установка системы и памяти ============}
AX0=0;  DM(Dm_Wait_Reg)          =AX0;    {нет состояний ожидания DM}
AX0=0x1018;  DM(Sys_Ctrl_Reg)    =AX0;    {SPORT0 установлен, SPORT1 недоступен}
RTS;

.ENDMOD;
```

Рис.3.3. Инициализация управляющих регистров, использующих символические адреса.

Имя регистра	*Адрес DM*	*Символическое имя*
System Control Register	0x3FFF	Sys_Ctrl_Reg
Data Memory Wait State Control Register	0x3FFE	Dm_Wait_Reg
Timer Period	0x3FFD	Tperiod_Reg
Timer Count	0x3FFC	Tcount_Reg
Timer Scaling Factor	0x3FFB	Tscale_Reg
SPORT1 Control Register	0x3FF2	Sport1_Ctrl_Reg
SPORT1 Serial Clock Divide Modulus	0x3FF1	Sport1_Sclkdiv
SPORT1 Rcv Frame Sync Divide Modulus	0x3FF0	Sport1_Rfsdiv
SPORT1 Autobuffer Control Register	0x3FEF	Sport1_Autobuf_Ctrl

Таблица 3.4. Управляющие регистры ADSP-2105 и предлагаемые символические имена.

Имя регистра	*Адрес DM*	*Символическое имя*
System Control Register	0x3FFF	Sys_Ctrl_Reg
Data Memory Wait State Control Register	0x3FFE	Dm_Wait_Reg
Timer Period	0x3FFD	Tperiod_Reg
Timer Count	0x3FFC	Tcount_Reg
Timer Scaling Factor	0x3FFB	Tscale_Reg
SPORT0 Multichannel Receive	0x3FFA	Sport0_Rx_Words1
Word Enable Register (32-bit)	0x3FF9	Sport0_Rx_Words0
SPORT0 Multichannel Transmit	0x3FF8	Sport0_Tx_Words1
Word Enable Register (32-bit)	0x3FF7	Sport0_Tx_Words0
SPORT0 Control Register	0x3FF6	Sport0_Ctrl_Reg
SPORT0 Serial Clock Divide Modulus	0x3FF5	Sport0_Sclkdiv
SPORT0 Rcv Frame Sync Divide Modulus	0x3FF4	Sport0_Rfsdiv
SPORT0 Autobuffer Control Register	0x3FF3	Sport0_Autobuf_Ctrl
SPORT1 Control Register	0x3FF2	Sport1_Ctrl_Reg
SPORT1 Serial Clock Divide Modulus	0x3FF1	Sport1_Sclkdiv
SPORT1 Rcv Frame Sync Divide Modulus	0x3FF0	Sport1_Rfsdiv
SPORT1 Autobuffer Control Register	0x3FEF	Sport1_Autobuf_Ctrl
HIP Interrupt Mask Register	0x3FE8	Hmask_Reg
HIP Status Register 7	0x3FE7	HSR7_Reg
HIP Status Register 6	0x3FE6	HSR6_Reg
HIP Data Register 5	0x3FE5	HSR5_Reg
HIP Data Register 4	0x3FE4	HSR4_Reg
HIP Data Register 3	0x3FE3	HSR3_Reg
HIP Data Register 2	0x3FE2	HSR2_Reg
HIP Data Register 1	0x3FE1	HSR1_Reg
HIP Data Register 0	0x3FE0	HSR0_Reg

Таблица 3.5. Управляющие регистры ADSP-2111 и предлагаемые символические имена.

Имя регистра	*Адрес DM*	*Символическое имя*
System Control Register	0x3FFF	Sys_Ctrl_Reg
Data Memory Wait State Control Register	0x3FFE	Dm_Wait_Reg
Timer Period	0x3FFD	Tperiod_Reg
Timer Count	0x3FFC	Tcount_Reg
Timer Scaling Factor	0x3FFB	Tscale_Reg
SPORT0 Multichannel Receive	0x3FFA	Sport0_Rx_Words1
Word Enable Register (32-bit)	0x3FF9	Sport0_Rx_Words0
SPORT0 Multichannel Transmit	0x3FF8	Sport0_Tx_Words1
Word Enable Register (32-bit)	0x3FF7	Sport0_Tx_Words0
SPORT0 Control Register	0x3FF6	Sport0_Ctrl_Reg
SPORT0 Serial Clock Divide Modulus	0x3FF5	Sport0_Sclkdiv
SPORT0 Rcv Frame Sync Divide Modulus	0x3FF4	Sport0_Rfsdiv

(продолжение)

Имя регистра	*Адрес DM*	*Символическое имя*
SPORT0 Autobuffer Control Register	0x3FF3	Sport0_Autobuf_Ctrl
SPORT1 Control Register	0x3FF2	Sport1_Ctrl_Reg
SPORT1 Serial Clock Divide Modulus	0x3FF1	Sport1_Sclkdiv
SPORT1 Rcv Frame Sync Divide Modulus	0x3FF0	Sport1_Rfsdiv
Analog Autobuffer/Powerdown Ctrl Reg	0x3FEF	Codec_Autobuf_Ctrl
Analog Control Register	0x3FEE	Codec_Ctrl_Reg
ADC Receive Data Register	0x3FED	Codec_Rx_Data
DAC Transmit Data Register	0x3FEC	Codec_Tx_Data
HIP Interrupt Mask Register	0x3FE8	Hmask_Reg
HIP Status Register 7	0x3FE7	HSR7_Reg
HIP Status Register 6	0x3FE6	HSR6_Reg
HIP Data Register 5	0x3FE5	HSR5_Reg
HIP Data Register 4	0x3FE4	HSR4_Reg
HIP Data Register 3	0x3FE3	HSR3_Reg
HIP Data Register 2	0x3FE2	HSR2_Reg
HIP Data Register 1	0x3FE1	HSR1_Reg
HIP Data Register 0	0x3FE0	HSR0_Reg

Таблица 3.6. Управляющие регистры ADSP-21msp50 и предлагаемые символические имена.

Для косвенной адресации регистров размещенных в памяти, используют регистры I и M. Управляющие регистры будут некорректно установлены, если эти регистры DAG (генератора адреса данных) некорректно инициализированы или переписаны с ошибкой. Этот тип неисправности может быть трудным для обнаружения в программе. (**Примечание**: Вы можете заметить, что несколько примеров программ ADSP-21xx, которые приведены в этом и других руководствах, используют косвенную адресацию для установки управляющих регистров. Эти примеры показывают, что хоть и рекомендуется прямая адресация, косвенная адресация может быть также применена).

3.7. Директивы ассемблера

Директивы ассемблера управляют процессом ассемблирования. Они обрабатываются препроцессором, но в отличие от инструкций не генерируют при ассемблировании код. Директива ассемблера начинается с точки и заканчивается точкой с запятой. Некоторые директивы имеют параметры и аргументы. Параметры следуют сразу за директивой и разделяются косой чертой; аргументы следуют после параметров.

```
.DIRECTIVE/параметр/параметр ... аргумент;        {комментарий}
```

Директивы ассемблера могут содержать только однострочный комментарий, который не может быть продолжен на следующей строке. Если требуется продолжить комментарий, начните новое поле на следующей строке.

3.7.1. Программные модули (. MODULE)

Директива .MODULE обозначает начало программного модуля и определяет название модуля. Исходный файл может содержать только один модуль. Директива имеет форму:

```
.MODULE/параметр/параметр ... имя_модуля;
```

В качестве параметров могут выступать:

```
RAM или ROM       тип памяти
ABS=адрес         абсолютный стартовый адрес(не используйте с STATIC)
SEG=seg_name      размещение в указанном сегменте
BOOT=0-7          размещение копии на странице начальной загрузки
STATIC            статичное размещение модуля в памяти
```

Параметры BOOT и STATIC используются только в системах с памятью начальной загрузки (т.е. все процессоры семейства, за исключением ADSP-2100). Второй способ размещения модулей на страницах начальной загрузки реализуется при использовании ключа редактора связей -i; для получения дополнительной информации смотрите раздел «Размещение модулей на страницах начальной загрузки» в главе 4.

Если тип памяти не определен, то по умолчанию принимается тип RAM (ОЗУ). Параметр ABS размещает коды модулей программ в определенных адресах памяти программ, что делает их неперемещаемыми. Это означает, что редактор связей резервирует память для модулей по указанным адресам. Модули, которые не имеют параметра ABS, перемещаемы.

Параметр SEG размещает модуль в указанный сегмент памяти, который объявлен в файле системной конфигурации. Если вы определяете оба параметра ABS и SEG, и указываете абсолютные адреса, которых нет в данном сегменте, вы увидите сообщение об ошибке при запуске редактора связей.

Параметр BOOT используется для размещения копии модуля на странице памяти начальной загрузки с указанным номером. Модуль будет сохранен в памяти начальной загрузки памяти до тех пор, пока он не будет загружен и выполнен. Вы можете разместить копии модулей на нескольких загрузочных страницах, позволяя получить доступ к его программам или данным, например .MODULE/BOOT=0/BOOT=1/BOOT=2. Другой способ реализации этого, использование параметра STATIC, который сохраняет модуль в памяти программ, когда загружаются страницы начальной загрузки (смотрите раздел «модули STATIC»).

Параметр BOOT применяется также ко всем переменным .VAR и объявлениям буферов данных внутри модуля - помните, что память начальной загрузки и память программ, обычно основном, содержат как программу, так и данные.

Директива .ENDMODE указывает на завершение программного модуля. Программа ассемблера останавливается, когда достигает директивы .ENDMODE. Примеры объявления модулей:

```
.MODULE/SEG=fir filter-routine;
```

В этом примере объявляется перемещаемый модуль *filter-routine*, размещенный в сегменте памяти с именем *fir*, который определен в выходном .ACH файле системного конфигуратора.

```
.MODULE/RAM/ABS=0×0040 main-prog;
```

В этом примере объявляется модуль *main-prog*, который должен быть размещен в памяти RAM по адресу 40 (шестнадцатеричный).

3.7.1.1. Загружаемые модули

Система может иметь до 8 загружаемых страниц (**Примечание**: у систем ADSP-2100 пространство памяти начальной загрузки отсутствует).

Когда вы выбираете атрибуты загружаемых модулей параметрами RAM, ROM, SEG и ABC, они применяются к памяти, где размещен код, во время выполнения, а не к памяти начальной загрузки. Таким образом, при конфигурировании распределения памяти во время исполнения, вы должны оперировать терминами памяти программ и памяти данных.

Редактор связей определяет расположение программ и данных в памяти, в соответствии с вашими объявлениями сегментов для системного конфигуратора и вашим объявлением модуля на ассемблере. Редактор связей конструирует также страницы памяти начальной загрузки, но вы не можете напрямую указать расположение модуля в загрузочной памяти.

Здесь приведен один из способов понять разницу между адресами памяти начальной загрузки и адресами памяти программ: процессор не может получить и выполнить инструкцию из памяти начальной загрузки; содержимое страницы памяти начальной загрузки должно быть вначале загружено во внутреннюю память программ, и только затем код выполняется. Если вы хотите, чтобы модуль (или переменная/буфер) существовали во внутренней ли внешней памяти процессора, в течение выполнения некоторой страницы начальной загрузки, необходимо ассоциировать ее определителем `BOOT` с этой страницей. Это заставит редактор связей оставить пространство для объекта во время выполнения кода страницы.

Выходной файл листинга редактора связей (`.map`) показывает расположение вашей программы в области загрузочной памяти, также как соответствующее отражение в памяти программ во время выполнения.

Например, следующая директива заново объявляет модуль *`main_prog`* от объявленного ранее. Модуль будет сохранен на странице 0:

```
.MODULE/RAM/ABS=0×0040/BOOT=0 main_prog;
```

параметры `RAM` и `ABS` этой директивы применяются к внутренней памяти программ процессора (считая, что MMAP=0).

Здесь приведен пример, который сохраняет копии перемещаемого модуля на нескольких загрузочных страницах:

```
.MODULE/RAM/BOOT=0/BOOT=2/BOOT=3 shifter;
```

(Для получения подобной информации смотрите раздел из главы 4 «Системы с несколькими страницами начальной загрузки»).

3.7.1.2. Статические модули

Если вы пишете программный модуль, который будет использоваться на нескольких страницах начальной загрузки, например, содержащий подпрограммы, вы захотите, чтобы код оставался на месте при загрузки различных страниц. Для того чтобы достигнуть этого,

при объявлении имени модуля нужно указать параметр STATIC. (Параметр ABS не может использоваться совместно с параметром STATIC).

Определитель STATIC предотвратит перезапись модуля, как в том случае, когда при сбросе загружается страница 0 (если MMAP=0), так и в том, когда программно вызываются страницами 1-7. Редактор связей гарантирует это при размещении вашей программы в памяти. Если модуль не объявлен как STATIC, он может быть частично или полностью перезаписан содержимым какой-либо загрузочной страницы. Это применимо к модулю как в случае с внутренней, так и с внешней памятью.

Когда редактор связей распределяет память для хранения вашей программы, он разбивает ее на 9 независимых частей: незагружаемая память программ/данных и загружаемые страницы 0-7. Незагружаемая память определена как начальная структура памяти программ и памяти данных перед загрузкой какой-либо страницы или исполнения кода.

Девять частей структуры содержатся независимо одна от другой, если в объявлении программного модуля (или буфера данных) не используется параметр STATIC. При отсутствии параметра STATIC, редактор связей принимает, что каждая из девяти частей стартует с чистого состояния памяти программ и памяти данных, и что каждая загружаемая страница имеет доступ ко всему пространству памяти.

Следуйте следующему правилу: если у вас есть программный модуль или буфер данных, который не должен быть перезаписан при загрузке новой страницы, вы должны использовать при его объявлении параметр STATIC. В противном случае, редактор связей принимает, что для загрузки новой страницы доступна вся память, и допустимо переписывать любые существующие программы/ данные.

Рис.3.4 и Рис.3.5 иллюстрируют действие параметра STATIC. Например, у вас есть подпрограмма с именем *routine1*, расположенная во внешней памяти программ ADSP-2101. Подпрограмма вызывается программой, сохраненной на загрузочной странице 0. Эта загрузочная страница содержит также 16-словный буфер с именем *coeffs*, который объявлен в модуле *bootfilter.* И подпрограмма *routine1* и буфер *coeffs* располагаются в пределах сегмента памяти *ext_pm*, определенном системным конфигуратором.

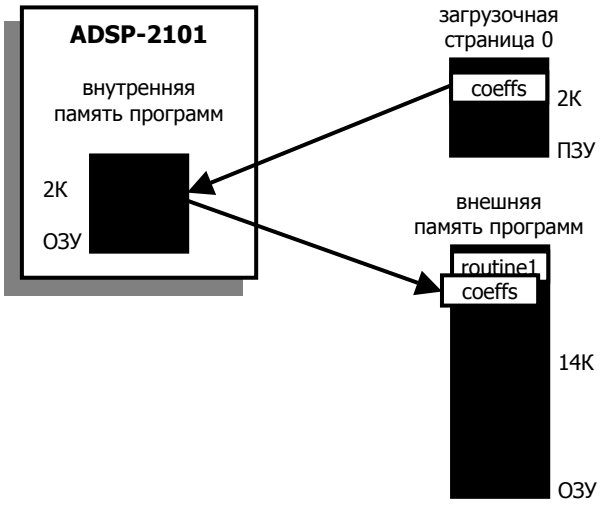


Рис.3.4. Перезапись астатического модуля.

```
.MOD/SEG=ext_pm/RAM     routine1;
.MOD/RAM/BOOT=0         bootfilter;
.VAR/SEG=ext_pm/PM/RAM  coeffs[16];
```

Пока подпрограмма *routine1* не объявлена как STATIC, редактор связей игнорирует ее расположение в памяти при определении места для буфера *coeffs* в памяти программ. Поэтому редактор связей может зарезервировать место для *coeffs* в области памяти, которая частично перекрывает *routine1*. При загрузке страницы 0, буфер *coeffs* загружают во внутреннюю память программ ADSP-2101, откуда она копируется во внешнюю память программ PM. Рис.3.4. показывает, что *coeffs* может перекрыть *routine1*.

Тем не менее, если при объявлении *routine1* указать параметр STATIC, редактор связей зарезервирует ее адреса в пространстве памяти программ и расположит *coeffs* где-нибудь в другом месте внешней памяти программ, как показано на Рис.3.5.

```
.MOD/SEG=ext_pm/RAM/STATIC  routine1;
```

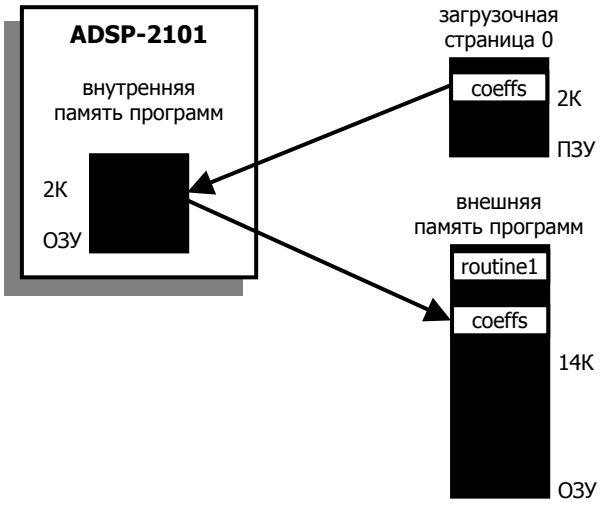


Рис.3.5. Защита статического модуля.

3.7.2. Переменные данные и буфера (.VAR)

Директива .VAR объявляет буферы данных. Буфер данных – это множество адресов расположения данных. Переменная объявляется как буфер с единственным адресом. Вы должны объявить все переменные и буфера до использования их в программе. Если буфер инициализирован директивой .INIT, объявление и инициализация должны выполняться в одном и том же модуле.

Директива .VAR имеет форму:

```
.VAR/параметр/параметр ... имя_буфера[длина], ... ;
```

Объявление по умолчанию подразумевает однословную переменную перемещаемую в памяти RAM. Одна директива .VAR может объявить несколько буферов, разделенных запятыми на одной строке (до 200 символов).

При многочисленном объявлении переменных и буферов в одной строке, редактор связей размещает их в смежных областях памяти. Если при таком объявлении используется параметр CIRC, то создается единственный кольцевой буфер, а отдельные буфера будут линейными. (Смотрите примеры в разделе «Дополнительно о кольцевых буферах»).

С директивой могут использоваться следующие параметры:

PM *или* DM	размеры в программной памяти или данных
RAM *или* ROM	тип памяти
ABS=адрес	абсолютный адрес (не использовать со STATIC)
SEG=*сегмент*	размещение буфера в сегменте, объявленным системным конфигуратором
CIRC	кольцевой буфер
STATIC	предотвращает перезапись буфера во время загрузки Страницы начальной загрузки

(Параметр STATIC используют только для систем с памятью начальной загрузки, т.е. для всего семейства процессоров, за исключением ADSP-2100).

Буфера могут быть размещены как в памяти программ, PM, так и в памяти данных, DM, по умолчанию. Тип памяти по умолчанию устанавливается в RAM для памяти DM и PM. Параметр ABS размещает буфер с указанного стартового адреса, делая его неперемещаемым. Параметр SEG размещает буфер в указанном сегменте памяти, который был объявлен в файле системного конфигуратора.

Параметр CIRC определяет кольцевой буфер. Буфер будет адресоваться в линейном виде, если не применен атрибут CIRC.

Параметр STATIC предотвращает перезапись буфера, когда загружается страница начальной загрузки. Если вы хотите использовать буфер в программах с нескольких загрузочных страниц, он должен оставаться неизменным во время загрузки. Это выполняется приданием буферу атрибута STATIC. Статические буфера управляются редактором связей точно так же, как статические модули - смотрите раздел «Статические модули» для уточнения деталей.

Для объявления переменной применяется директива .VAR без указания длины буфера:

```
.VAR/DM/RAM/ABS=0x000A  seed;
```

Этот оператор объявляет однословную переменную с именем *seed* в памяти данных RAM по адресу 10 (десятичный). Следующий пример показывает объявление буфера:

```
.VAR/PM/RAM/SEG=pmdata  coefficients[10];
```

Здесь линейный буфер объявлен в памяти программ RAM, который перемещаем в пределах сегмента с именем *pmdata*. Название буфера *coefficients* и он состоит из 10 записей в памяти программ. Длина буфера должна быть помещена внутри скобок.

(В этом руководстве скобки обычно используются для выделения необязательных аргументов. Директивы .VAR, .INIT и .INCLUDE являются примерами синтаксиса ассемблера, где требуются круглые и угловые скобки).

Ниже приведен пример объявления перемещаемого кольцевого буфера, длина которого определяется величиной константы *taps*.

```
.CONST  taps = 15;
.VAR/DM/CIRC  data_buffer[taps];
```

3.7.2.1. Дополнительно о кольцевых буферах

Кольцевой буфер может быть размещен в памяти с некоторыми ограничениями, связанными с характеристиками процессоров ADSP-21xx по аппаратной реализации адресации кольцевого буфера. Вообще, кольцевой буфер должен стартовать с базового адреса, который кратен 2^n, где n – количество бит требуемых для представления длины буфера в двоичном виде (Обратитесь к следующему разделу для обсуждения специальных случаев, когда длина буфера равна 2^n).

Редактор связей будет придерживаться этих требований для размещения кольцевых буферов. Вы должны иметь это в виду, если явно выбрали базовый адрес буфера параметром ABS. Приведенные ниже примеры помогут понять, где можно размещать циклические буферы в памяти.

Следующий оператор объявляет кольцевой буфер из пяти позиций:

```
.VAR/CIRC  aa[5];
```

Так как для представления длины *aa* необходимо три разряда, редактор связей присвоит буферу базовый адрес кратный 8. Три младших значимых разряда (МЗР) этого адреса нули.

Если на одной строке объявлены несколько буферов и используется параметр CIRC, создается один кольцевой буфер, а отдельные буфера будут только простыми линейными буферами. Например, следующее объявление создает один 15-словный кольцевой буфер (изображенный на Рис.3.6):

```
.VAR/CIRC  aa[5], bb[5], cc[5];
```

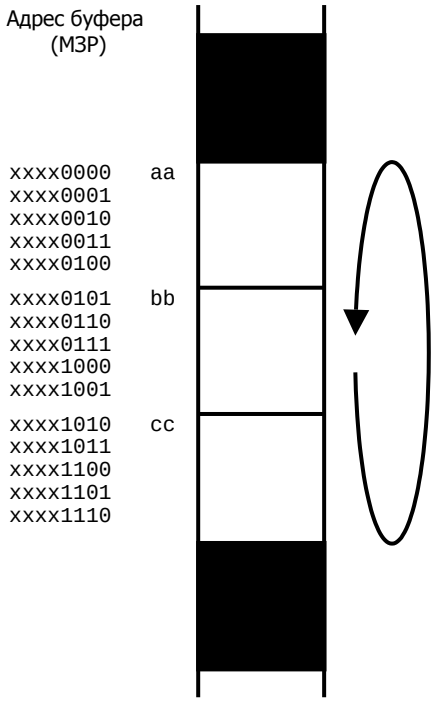
Базовый адрес кольцевого буфера равен адресу *aa*; этот символ будет использоваться для доступа к буферу в программе. Адрес *bb* это *aa+5*, а адрес *cc* это *aa+10*. Три пятисловных буфера могут быть индивидуально доступны как линейные.

Так как величина 15 требует четырех разрядов для двоичного представления, кольцевой буфер *aa* будет размещен по адресу, который кратен 16 (четыре младших значащих разряда равны нулю).

Следующий пример показывает использование трех директив для объявления трех различных циклических буферов:

```
.VAR/CIRC  aa[5];
.VAR/CIRC  bb[5];
.VAR/CIRC  cc[5];
```

Поскольку они объявлены отдельно, буфера не будут объединены (см. Рис.3.7).



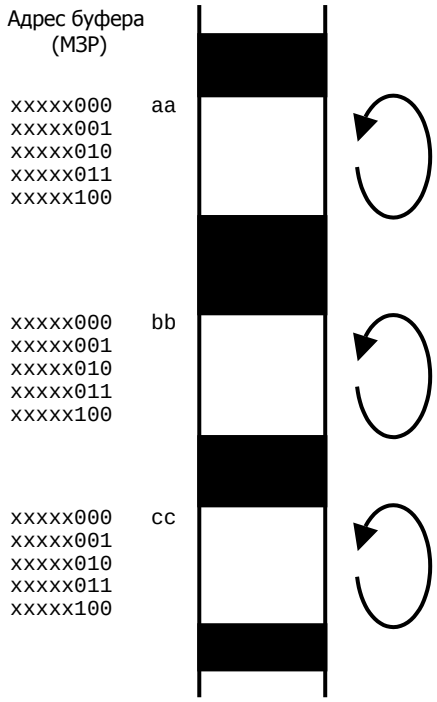


Рис.3.6. Составной кольцевой буфер. **Рис.3.7. Отдельные кольцевые буфера.**

В следующем примере создаются структуры для поисковых таблиц синуса/косинуса:

```
.VAR/CIRC  sin[256], cos[768];
```

Здесь определен один кольцевой буфер, который имеет длину 1024. Чтобы получить доступ к буферу из программы, вы должны инициализировать индексные регистры DAG и регистры длины буфера следующими инструкциями:

```
I0 = ^cos;              {^ это оператор "указателя адреса" }
L0 = 1024;
I1 = ^sin;
L1 = 1024;
```

Эти инструкции загружают регистры I0 и I1 базовыми адресами *cos* и *sin*. Соответствующие регистры L загружаются значением длины кольцевого буфера для разрешения кольцевой адресации. Кольцевой буфер реализуется только при ненулевом значении регистра L.

Обратитесь к главе «Data Transfer» из *ADSP-2100 Family User's Manual* для уточнения информации по кольцевым буферам.

(**Примечание**: Для линейной непрямой адресации, L регистры должны быть установлены в нуль. Не думайте, что L регистры процессора автоматически инициализируются или могут быть игнорированы, если вы не используете циклические буферы; регистры I, M, L содержат случайные величины, которые следует устанавливать. Ваша программа должна инициализировать L регистры, соответствующие используемым I регистрам).

3.7.2.2. Особый случай: кольцевой буфер длиной 2^n

Существует одно отличие между ADSP-2100 и всеми другими процессорами ADSP-21xx для размещения кольцевого буфера, когда его длина соответствует степени 2. Во всех случаях, определенное число младших значащих разрядов базового адреса кольцевого буфера должно быть установлено в ноль. Однако ADSP-2100 требует на 1 нулевой бит больше.

Например, все ADSP-21xx процессоры, кроме ADSP-2100, могут иметь два восьмисловных циклических буфера, размещенных в последовательных блоках памяти. Однако ADSP-2100 использует память менее эффективно, должен оставить восьмисловный блок памяти между двумя буферами. Другими словами, базовый адрес восьмисловного буфера ADSP-2100 должен быть кратен 16, в то время как для кольцевого буфера ADSP-21xx базовый адрес должен быть кратен только 8.

3.7.3. Инициализация переменных и буферов (.INIT)

Директива `.INIT` используется для инициализации переменных и буферов в ПЗУ. Редактор связей помещает данные инициализации в файл образа памяти, который загружается разделителем программ для записи в ППЗУ. Разделитель переводит ПЗУ части этого файла в формат, совместимый с промышленным стандартом программатора ППЗУ.

Инициализирующие значения могут быть перечислены в директиве или указаны во внешнем файле; директива `.INIT` принимает одну из следующих форм:

```
.INIT  имя_буфера : константа, константа, ...;
.INIT  имя_буфера : ^другой_буфер или %другой_буфер, ...;
.INIT  имя_буфера : <имя_файла>;
```

Операторы `^` и `%` использоваться для инициализации буфера или переменной базовым адресом или длиной, или даже другими буферами. Любые комбинации констант, указателей адресов буфера и величин длины буфера могут быть заданы через запятую. Примеры:

```
.INIT seed: 0×3FFF;
```

Этот оператор инициализирует переменную *seed* шестнадцатеричной константой.

```
.INIT seed_values: 1,2,3,5,7;
```

Этот оператор инициализирует буфер *seed_values* списком констант.

```
.INIT buffer_ptr: ^input_buf;
```

Здесь переменная *buffer_ptr* инициализируется указателем стартового адреса буфера *input_buf*. Вы можете инициализировать только часть данных буфера, задавая смещение:

```
.INIT buffer_name[offset] :
```

Теперь инициализирующие величины будут размещены, начиная с адреса: *buffer_name + offset*.

Следующий оператор, например, инициализирует восьмой, девятый и десятый элементы буфера *coeffs* величинами 2, 3 и 4:

```
.INIT  coeffs[7] : 2,3,4;
```

Третья форма директивы .INIT указывает имя файла, который содержит инициализирующие величины. Ассемблер устанавливает указатель на этот файл, и данные присоединяются при запуске редактора связей.

Следующий пример заставляет редактор связей инициализировать буфер *cos* содержимым файла *cosines.dat*:

```
.INIT  cos: <cosines.dat>;
```

Если файл с данными находится в текущей директории операционной системы, только необходимо указать в скобках только имя файла. Если файл находится в другом каталоге, вы должны указать путь к этому каталогу и имя файла. Например, если файл *inits.dat* для буфера с именем *samples* размещен в директории DOS C:\2101\filter3\, тогда директива .INIT должна быть применена следующим образом:

```
.INIT  samples : < c:\2101\filter3\inits.dat>
```

Это позволит редактору связей найти файл. Данный способ широко используется для загрузки буферов данными, выработанными другими программами, такими как нахождение коэффициентов фильтра. После того, как редактор связей считает и присоединит содержимое файла, изменение данных потребует лишь выполнить повторную компоновку программы.

Переменные данных и буферов могут еще быть инициализированы с помощью семиразрядного ASCII кода. Следующие пример инициализирует один четырехпозиционный буфер данных *inputs* ASCII кодами A, B, C, D. ASCII коды размещаются в семи МЗР памяти данных (16 разрядов) или памяти программ (8, 16 или 24 разрядов).

```
.INIT  inputs : 'ABCD';
```

Специальный синтаксис директивы .INIT, .INIT24, позволяет сохранять 24-разрядные данные в памяти программ. Это позволяет получить доступ к младшим 8 разрядам каждого 24-разрядного слова памяти программ при инициализации буферов данных или переменных в исходной программе.

Например, в то время как эта запись вычисляет 16-разрядный адрес:

```
.INIT  var : ^label + 10
```

То следующая запись вычисляет 24-разрядный адрес:

```
.INIT24  var : ^label + 10
```

3.7.3.1. Инициализация данных в системном оборудовании

Созданный редактором связей .EXE образ памяти содержит, содержащий ваши программные коды и инициализирующие данные. Генерация этого файла не гарантирует, тем не менее, что программа и данные будут загружены в память; файл только перечисляет, что *должно быть* представлено в памяти, чтобы программа запустилась корректно. Вы должны предоставить данные, которыми память будет по-настоящему загружена.

После того как редактор связей создаст файл образа памяти для вашей программы, существует два способа завершить инициализацию: 1) запрограммировать микросхему ППЗУ для буферов, расположенных в ПЗУ. 2) записать в вашу программу код, копирующий инициализирующие данные в буферы, расположенные в ОЗУ.

Вы можете инициализировать буферы, расположенные в ПЗУ используя разделитель программ для записи в ППЗУ одним из следующих способов:

- создание файлов для программирования приборов ППЗУ для внешней памяти программ или данных, *или*
- создание файлов для программирования приборов ППЗУ для памяти начальной загрузки; переменные и буферы во внутренней памяти программ будут инициализированы во время загрузки (для процессоров ADSP-21xx с внутренней и загрузочной памятью).

Переменные и буферы, расположенные в памяти программ или данных ОЗУ должны быть инициализированы вашей программой. Исключением является внутренняя память программ ОЗУ ADSP-2101, ADSP-2105, ADSP-21msp50. Это пространство ОЗУ может быть инициализировано при загрузке (как описано выше).

Три типа памяти должны быть инициализированы в исходном коде:

- внешняя память программ ОЗУ
- внешняя память данных ОЗУ
- внутренняя память данных ОЗУ

Чтобы инициализировать буферы в этих областях памяти вы должны иметь данные, сохраненные в ПЗУ, и включить в вашу программу код, копирующий данные в соответствующую область памяти. Пример такой подпрограммы показан ниже:

```
.VAR/PM/ROM  sin_init[64];
.VAR/DM/RAM  sin_table[64];
.INIT  sin_init : <sin.dat> ;
{копировать инициализированный буфер, sin_init, из PM ROM в DM RAM}
      M0=1;
      M4=1;
      I0=^sin_table;
      I4=^sin_init;
      CNTR=%sin_table;
      D0 sin_copy UNTIL CE;
      AX0=PM(I4,M4);
sin_copy: DM(I0,M0)=AX0;
```

Для инициализации данных, сохраненных в загрузочной памяти, загрузчик должен выполнить операцию копирования из внутренней памяти программ, после окончания загрузки.

3.7.4. Обозначение портов для ассемблера (.PORT)

Директива .PORT обозначает порт ввода/вывода, отраженный в памяти, который уже был объявлен системным конфигуратором (и определен в файле .ACH). Параметры порта и адрес памяти указаны в объявлении системного конфигуратора и не требуют повторного указания. Директива .PORT имеет формат:

```
.PORT имя_порта;
```

Если требуется доступ к порту в других модулях вашей программы, вы должны объявить его в этом модуле двумя директивами .PORT и .GLOBAL. В дальнейшем вы должны указать порт в других модулях директивой .EXTERNAL. (Смотрите описание .GLOBAL и .EXTERNAL). Редактор связей читает информацию о порте из файла системного конфигуратора .ACH и устанавливает все ссылки на него.

3.7.5. Включение других исходных файлов (.INCLUDE)

Директива .INCLUDE используется для включения других исходных файлов в файл предназначенный для ассемблирования. Ассемблер открывает, читает и ассемблирует указанный файл, когда он встречает строку оператора .INCLUDE. Ассемблированный код объединяется в выходном файле .OBJ. Когда ассемблер достигает конца включенного файла, он возвращается в первичный исходный файл и продолжает обработку. Директива .INCLUDE имеет формат:

```
.INCLUDE <имя_файла>;
```

Если файл, который должен быть включен директивой .INCLUDE, находится в текущей директории вашей операционной системы, в скобках требуется указать только имя файла . Если файл находится в другом каталоге, вы должны задать путь к этому каталогу и имя файла (или использовать переменную среды окружения ADII; смотрите ниже). Например, если файл, который должен быть включен, называется newcode и расположен в директории C:\2111\filters\ , тогда директива .INCLUDE должна быть задана следующим образом:

```
.INCLUDE <C:\2111\filters\newcode>;
```

Это позволит ассемблеру найти файл. В другом случае, вы можете указать путь, используя переменную среды окружения ADII. Установка ADII равной пути к каталогу позволит ассемблеру обнаружить файл. В этом случае вы можете задавать имя файла без указания полного пути. Файл, включенный директивой .INCLUDE, может также содержать внутри себя директиву .INCLUDE – вложение файлов директивами INCLUDE ограничивается только размером свободной оперативной памяти. Файлы, включенные директивой .INCLUDE, не могут содержать директив С препроцессора (таких как #define). Для включения таких файлов используйте директиву С препроцессора #include.

Директива . INCLUDE допускает использовать принцип модульного программирования. Например, во многих случаях она используется, чтобы развить библиотеку подпрограмм или макросов, которые применяются в различных программах. Вместо того чтобы каждый раз переписывать подпрограммы, вы можете присоединить макробиблиотеку в ассемблерный модуль, воспользовавшись директивой .INCLUDE. Пример:

```
.INCLUDE  <библиотека_макросов>;
```

3.7.6. Макросы

Макрос создается с помощью ассемблерной директивы .MACRO. Макрос используется для повторения часто используемых последовательностей инструкций в вашем исходном коде. Передачей аргументов макросу реализуется подобие подпрограммы, которая может быть использована в различных программах.

Макро вложения ограничены только размером свободной оперативной памяти. Вложенные макросы должны быть объявлены следующим образом: внутренний макрос первый, ..., внешний макрос последний. Все константы, используемые в макросах, должны быть объявлены перед объявлением макросов.

3.7.6.1. Определение макроса (.MACRO)

Макрос определяется двумя директивами:

```
.MACRO имя_макроса (аргумент, аргумент, ...);
...
...
.ENDMACRO;
```

Каждый оператор внутри макроса может быть инструкцией, директивой или макро включением. Директива .ENDMACRO отмечает конец макроопределения. Макрос вызывается по своему имени. Чтобы выполнить макрос с именем quickloop, используйте следующую команду в вашем исходном коде:

```
quickloop;  вызов макроса
```

Макровызов не должно содержать дополнительных операторов (т.е. инструкций, директив препроцессора или других макровключений) на той же строке исходного кода. Аргументы макроса принимают форму:

```
%n      n = 0, 1, 2, ..., 9
```

Следующий пример определяет макрос с тремя аргументами:

```
.MACRO memory_transf (%0, %1, %2);
```

В коде макроса, аргументы маркируются служебными символами %1, %2, %3, и т.д. При вызове макроса служебные символы замещаются величинами аргументов, переданных в макрос. Должно быть передано правильное число аргументов. Передаваемые аргументы могут быть одними из:

Аргумент	*Исключения*
константы или выражения	нет
символы	все, кроме MACRO, ENDMACRO,CONST,INCLUDE
^символ	"^%n" не разрешено
%буфер	"%%n" не разрешено

Таблица 3.7. Допустимые аргументы MACRO.

Операторы ^ и % не могут быть использованы с аргументами, замещающими служебные символы в макроопределении. Тем не менее, аргументы переданные в макрос могут использовать эти операторы. Например:

```
reed_data (^input);
```

(**Примечание**: другой способ определить макрос это директива C препроцессора `#define`).

3.7.6.2. Локальная метка в макросах (.LOCAL)

Директивой `.LOCAL` задают программные метки, используемые в макросе. Директива `.LOCAL` указывает ассемблеру создавать уникальную версию метки при каждом включении макроса. Это предотвращает ошибку дублирования меток в случае, когда макрос вызывается несколько раз в одном программном модуле. Директива `.LOCAL` имеет формат:

```
.LOCAL  метка_макроса, ...;
```

Ассемблер создает уникальные версии `метки_макроса` добавляя к ней номер; это может посмотреть в программе моделирования или в файле листинга `.LST`, разрешено раскрытие макросов. Смотрите Рис.3.8. как пример директивы `.LOCAL`.

3.7.6.3. Пример макроса

Рис.3.8. показывает пример макрообъявления и вызова. Макрос реализует подпрограмму, которая переносит содержимое буфера данных из одной области памяти в другую.

```
{MACRO объявление}
.MACRO memory_transf (%0, %1, %2, %3, %4) {допускает 5 аргументов}
.LOCAL transf;

I4=%0;                      {устанавливает I4 как адрес источника}
I5=%1;                      {устанавливает I5 как адрес приемника}
M4=1                        {устанавливает указатель на инкремент 1}
CNTR=%2                     {устанавливает длину буфера}
DO transf UNTIL CE;         {перенос данных}
    SI=%3(I4, M4)           {переносит из типа %3 памяти}
transf: %4(I5, M4)=SI;      {переносит в тип %4 памяти}

.ENDMACRO

{MACRO вызов}
memory_transf (^coeff_table, ^buffer, buff_length, PM,DM);
```

Рис.3.8. Пример макроса.

Заметьте, что зарезервированные ключевые слова `PM` и `DM` переданы как аргументы.

3.7.7. Глобальные структуры данных (.GLOBAL)

Директива .GLOBAL позволяет переменным, буферам и портам быть доступными извне модуля, в котором они объявлены. Для доступа к одной из этих структур из других модулей вы должны объявить ее с директивой .GLOBAL. Директива .GLOBAL имеет формат:

```
.GLOBAL внутренний_символ, ...;
```

Пример:

```
.VAR/PM/RAM  coeffs[10];
.GLOBAL      coeffs;        {делает буфер видимым снаружи модуля}
```

С тех пор как символ сделан глобальным, другие модули могут обращаться к нему идентифицируя символ как внешний.

3.7.8. Глобальные программные метки (.ENTRY)

Директива .ENTRY позволяет обращаться к программным меткам в других модулях. Это позволяет использовать метку для вызова подпрограммы или межмодульных переходов. Директива .ENTRY имеет формат:

```
.ENTRY  программная_метка, ...;
```

Пример:

```
.ENTRY  fir_start;          {делает метку видимой снаружи модуля}
```

С тех пор, как метка объявлена директивой .ENTRY другие модули могут обращаться к ней, идентифицируя метку как внешнюю.

3.7.9. Внешние символы (.EXTERNAL)

Директива .EXTERNAL позволяет программному модулю обращаться к глобальным структурам данных (переменным, буферам и портам) и программным меткам, объявленным в других модулях.

Символ должен быть определен до этого с помощью директив .GLOBAL или .ENTRY в тех модулях, где он впервые объявлен. Другие модули должны использовать директиву .EXTERNAL для открытия доступа к внешним символам. Директива имеет формат:

```
.EXTERNAL  внешний_символ, ...;
```

Пример:

```
.EXTERNAL  fir_start;       {метка в другом модуле}
```

3.7.10. Ассемблерные константы (.CONST)

Директива .CONST определяет ассемблерные константы. После объявления символической константы, вы можете использовать ее вместо реального числа. Директива имеет формат:

```
.CONST  имя_константы = константа или выражение, ...;
```

В выражениях разрешаются только арифметические или логические действия над двумя или более целыми константами; использование символы не допускается. Одна директива .CONST может содержать на одной строке несколько объявлений констант, разделенных запятыми. Список множества объявлений не может быть продолжен на следующей строке. Пример:

```
.CONST  taps=15, taps_less_one=14;
```

3.7.11. Размещение программ и данных в сегментах памяти (.PMSEG, .DMSEG)

Директивы .PMSEG и .DMSEG подобны параметру /SEG директив .MODULE и .VAR и имеют следующий формат:

```
.PMSEG имя_сегмента_pm;
.DMSEG имя_сегмента_dm;
```

Директива .PMSEG указывает редактору связей на необходимость разместить все программы и данные модуля в сегменте *имя_сегмента_pm* памяти программ. Директива .DMSEG указывает редактору связей на необходимость разместить все структуры данных модуля в сегменте *имя_сегмента_dm* памяти данных. Сегменты *имя_сегмента_pm* и *имя_сегмента_dm* должны быть предварительно определены в файле описания архитектуры .ACH системного конфигуратора. Обычно, чтобы расположить все программы и данные исходного модуля в определенном системным конфигуратором сегменте памяти, вы должны повторить параметр /SEG в директиве .MODULE и всех директивах .VAR внутри модуля. Директивы .PMSEG и .DMSEG используются для исключения многократного повторения параметров /SEG.

Директивы .PMSEG и .DMSEG должны быть размещены в вашем исходном файле перед директивой .MODULE. Ниже приводится пример, в котором модуль располагают в памяти данных в сегмент с именем Audio_Samples:

```
.DMSEG  Audio_Samples;
.MODULE/RAM  Sample_Input;

.VAR/DM/RAM/CIRC  sample_buffer[15];
.VAR/DM/RAM  other_buffer[5];
.VAR/DM/RAM  another_buffer[5];
.VAR/DM/RAM  variable1;

... программа SAMPLE_INPUT ...

.ENDMOD;
```

Программа для подпрограммы SAMPLE_INPUT будет размещения в памяти программ.

3.7.12. Системы страничной памяти (.PAGE)

Директива системного конфигуратора .ADSP2101P создает файл описания архитектуры для систем со страничной организацией памяти. Директива ассемблера .PAGE должна использоваться во всех исходных программных модулях, которые составляют часть системы со страничной организацией памяти:

```
.PAGE;
```

Директива .PAGE должна быть размещена перед директивой .MODULE. Специальный ассемблерный оператор, PAGE *имя_переменной*, используется для выделения номера страницы (верхние разряды адреса) переменной или буфера данных:

```
AX0 = PAGE array0;          {получения номера страницы array0}
```

Эта инструкция определяет номер страницы буфера *array0* и загружает его в регистр AX0. Заметьте, что оператор PAGE подобен ассемблерным операторам указателя адреса (^) и длины (%). Программные модули, буферы данных, переменные данных, которые сохраняются в страничной памяти, должны быть ограничены своей собственной страницей и не могут пересекать ее границу.

3.8. Инициализация вашей программы в памяти

Редактор связей читает выходные .OBJ файлы ассемблера и создает исполняемый .EXE файл. После ассемблирования и связывания вашей программы, у вас есть файл для загрузки в память, который содержит все программные коды и данные. Этот файл представляет собой подробную «фотографию» системной памяти, предшествующей началу исполнения программы. Однако, простое создание этого файла, не гарантирует, что программа и данные будут инициализированы в памяти; файл определяет только то, что должно быть представлено в памяти для корректного запуска программы. Необходимо обеспечить значения, которыми память RAM и память ROM реально загружаются.

Существуют два способа реализации этого: 1) устройства программирования ППЗУ и 2) включение подпрограмм для копирования программ и данных в определенные области памяти. Эти методы описаны в следующих двух разделах. Рис.3.9. показывает, какой из методов может быть применен для каждой из областей памяти и типов памяти процессоров.

МЕТОД ИНИЦИАЛИЗАЦИИ	ADSP-2100	ADSP-21xx (другие)
Программирование ППЗУ	Внешняя PM ROM Внешняя DM ROM	Внешняя PM ROM Внешняя DM ROM Внутренняя PM RAM
Написание исходного кода для копирования программ/данных из внешней ROM	Внешняя PM RAM Внешняя DM RAM	Внутренняя DM RAM Внешняя PM RAM Внешняя DM RAM
Написание исходного кода для копирования программ/данных из внутренней PM RAM (после загрузки)		

Рис.3.9. Как инициализировать память.

(Программы моделирования ADSP-21XX автоматически инициализируют все разделы внутренней или внешней памяти ОЗУ или ПЗУ; это сделано для того чтобы упростить процесс отладки. Однако вы должны помнить, что автоматическая инициализация не поддерживается на аппаратном уровне.)

3.8.1. Использование разделителя программ для инициализации ПЗУ

После того, как редактор связей создал исполняемый файл вашей программы, вы можете использовать разделитель программ для записи в ППЗУ для того, чтобы завершить инициализацию одним из двух способов:

- посредством создания файлов для программаторов ППЗУ для внешней памяти программ или данных, или
- посредством создания файлов для программаторов ППЗУ для загрузочной памяти; внутренняя память программ будет инициализирована, когда осуществится загрузка (для ADSP-21xx процессоров с внутренней и загрузочной памятью).

Разделитель переводит ПЗУ части `.EXE` файла в формат промышленного стандарта программаторов ППЗУ.

3.8.2. Инициализация системной памяти ОЗУ в исходной программе

Системная память ОЗУ должна быть инициализирована вашей программой. Исключение из этого правила составляют процессоры ADSP-2101, ADSP-2105, ADSP-2111, и ADSP-21msp50. Это пространство памяти может быть инициализировано загрузочной операцией (как описано в предыдущем разделе). В исходной программе должны быть инициализированы три типа памяти:

- внешняя память программ ОЗУ
- внешняя память данных ОЗУ
- внутренняя память данных ОЗУ (для процессоров ADSP-21xx).

Для инициализации этих пространств памяти, у вас должны быть программа и данные, сохраненные в микросхемах ПЗУ. Необходимо включить в программу исходный код, который будет копировать информацию в определенные ячейки памяти. Для каждого несмежного сегмента памяти требуется один цикл копирования.

3.9. Формат файла листинга

Файл листинга `.LST` позволяет представить результаты процесса ассемблирования. Пример файла листинга показан на Рис.3.10. В этом файле представлена следующая информация:

addr	смещение от базового адреса модуля
inst	код операции (добавление «u» показывает, что код операции содержит ненайденные поля)
source line	номер строки исходного файла и код.

Для формирования выходного файла листинга используются пять директив ассемблера. Директива .NEWPAGE вставляет разделители страниц. Пример:

```
.NEWPAGE;
```

Директива .PAGELENTH вставляет разделитель страниц после указанного количества строк:

```
.PAGELENTH lines
```

Директива .LEFTMARGIN оставляет левое поля указанного числа столбцов. Пример:

```
.LEFTMARGIN columns
```

Директива .INDENT оставляет отступ в исходном коде указанного число столбцов:

```
.INDENT columns
```

Директива .PAGEWIDTH оставляет правое поля указанного числа столбцов. Пример:

```
.PAGEWIDTH columns
```

Директивы .NEWPAGE и .PAGELENGTH могут быть использованы для нумерации страниц, в то время как директивы .LEFTMARGIN, .INDENT и .PAGEWIDTH используются для того, чтобы сделать каждую страницу удобной для чтения. Эти директивы могут быть помещены в любом месте исходного файла.

```
Analog Devices Inc.          ADSP-21xx Assembler Version 3.0
C:\2101_System\fir2101.app   Fri May 13 11:04:39  1990  Page 1

addr inst   source line
            1  .MODULE/RAM/BOOT=0 FIR_ROUTINE;  {перемещаемый}
            2                                   {модуль обслуживания прерываний}
            3  .CONST  TAPS = 15;
            4  .ENTRY  FIR_START;               {делает метку видимой снаружи}
            5
            6  .EXTERNAL DATA_BUFFER, COEFFICIENT; {глобальные буферы видимы}
            7
            8
            9
0000 3C00E5 10  FIR_START:  CNTR=14;            {N-1 проходов между D0 LOOP}
0001 0D0388 11              SI=RXO;             {читать из SPORT0}
0002 680080 12              DM(I0,M0)=SI;
0003 E89800 13              MR=0, MY0=PM(I4,M4), MX0=DM(I0,M0);
            14
            15
0004 1400Eu 16              DO CONVOLUTION UNTIL CE;
0005 E80000 17  CONVOLUTION:MR=MR+MXO*MY0(SS), MY0=PM(I4,M4), MX0=DM(I0,M0);
            18
            19
0006 20400F 20              MR=MR+MXO*MY0(RND); {N циклических проходов}
0007 050000 21              IF MV SAT MR;       {насыщение, если переполнение}
0008 0D0C9C 22              TX0 = MR1;          {запись в SPORT0}
0009 0A001F 23              RTI;                {возврат из прерывания}
            24  .ENDMOD;
```

Рис.3.10. Файл листинга.