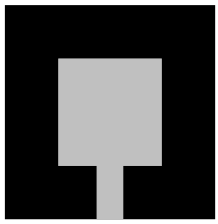
Семейство процессоров ADSP-2100

Руководство пользователя Ассемблера



Второе издание [оригинал] (11/1994)
Предварительное издание [перевод] (02/2001)

АВТОРЫ ПЕРЕВОДА : ПОКРОВСКИЙ В.Н., СИЛАНТЬЕВ В.И.
ЗАКАЗ И ОРГАНИЗАЦИЯ ПЕРЕВОДА ЦУКАНОВ Ю.В.

Литература

Руководства пользователя ADSP-2100

ADSP-2100 Family Assembler Tools & Simulator Manual
ADSP-2100 Family C Tools Manual
ADSP-2100 Family C Runtime Library Manual
ADSP-2100 Family EZ Tools Manual

Руководства по применению (Prentice Hall)

Digital Signal Processing Applications Using the ADSP-2100 Family, Volume 1
Включены темы арифметики, фильтров, БПФ, предикативного линейного кодирования, модемов, алгоритмов, графики, кодово-импульсной модуляции, многоступенчатых фильтров, двухтонального многочастотного набора, многопроцессорной обработки, хост-интерфейса и гидролокации.

Digital Signal Processing Applications Using the ADSP-2100 Family, Volume 2
Включены темы модемов, предикативного линейного кодирования, GSM кодеков, АДИКМ кодирования, распознавания речи, дискретного косинусного преобразования, цифрового тонального детектирования, разработки цифровых систем управления, биквадратных БИХ-фильтров, организации программного и аппаратного UART интерфейса.

Информация по спецификациям

ADSP-2100/ADSP2100A Data Sheet
ADSP-21xx Data Sheet
ADSP-21msp50A/55A/56A Data Sheet
ADSP-21msp58/59 Preliminary Data Sheet
ADSP-2171/72/73 Data Sheet
ADSP-2181 Preliminary Data Sheet

ЗАМЕЧАНИЯ ПО ИЗДЕЛИЯМ И ДОКУМЕНТАЦИИ ANALOG DEVICES
Analog Devices оставляет за собой право изменить данное изделие и его документацию без заблаговременного уведомления. Информация предоставляемая Analog Devices проверена и может считаться достоверной. Однако, Analog Devices не несет ответственности за использование изделия, также как за нарушение прав или других прав третьих лиц, которые могут быть затронуты в результате использования изделия. Не предоставляются никакие лицензии при включении или использовании патентных прав Analog Devices.

ВТОРОЕ ОРИГИНАЛЬНОЕ ИЗДАНИЕ 11/1994
ПЕРВОЕ ПЕРЕВЕДЕННОЕ ИЗДАНИЕ 02/2001

Введение & Установка

1

1.1. Введение

Средства разработки для семейства процессоров ADSP-2100 включают в себя комплект программных инструментов проектирования. Программное обеспечение состоит из инструментов программирования на языках ассемблер и С, а также программ моделирования работы процессоров, позволяющие разрабатывать и отлаживать системы DSP. Программное обеспечение разработчика может быть запущено на IBM (или IBM- совместимых) PC и рабочих станциях SUN4.

Средства разработки состоят из нескольких программ: системного конфигуратора (*system builder*), ассемблера (*assembler*), редактора связей или компоновщика (*linker*), программы разделителя для программатора ППЗУ (*PROM splitter*), программы моделирования или симулятора (*simulator*) и С компилятора (*compiler*). Это руководство описывает первые пять программ.

Для получения информации С компиляторе для семейства процессоров ADSP-2100, обратитесь к *ADSP-2100 Family C Tools Manual & ADSP-2100 Family C Runtime Library Manual,* соответственно.

Для получения информации по архитектуре и системному интерфейсу каждого процессора, обратитесь к *ADSP-2100 Family User's Manual.*

Семейство ADSP-2100 включает в себя следующие процессоры:

Процессор	*Описание*
ADSP-2100/ADSP-2100A	микропроцессор DSP
ADSP-2101	микрокомпьютер DSP
ADSP-2105	микрокомпьютер DSP
ADSP-2115	микрокомпьютер DSP
ADSP-2111	микрокомпьютер DSP с Хост портом
ADSP-21msp50/55/56	микрокомпьютер DSP смешанного сигнала
ADSP-2171	микрокомпьютер DSP с Хост портом

Это руководство содержит полную информацию по разработке программ для всех перечисленных процессоров.

Версии процессоров с ПЗУ, программируемым фотошаблонами, такие как ADSP-2102 и ADSP-2106, не перечислены в списке, однако программы для этих изделий создаются тем же способом, как и для стандартных компонентов.

Другие процессоры, которые пополнят семейство ADSP-2100 в будущем, будут полностью программно-совместимыми и позволят использовать средства разработки ADSP-21xx и данное руководство.

В этом руководстве, термин «ADSP-21xx» используют в основном для ссылки на один или все процессоры семейства ADSP-2100. Термин «ADSP-2100» использован для обозначения ADSP-2100 и ADSP-2100A.

Заметьте, что любые возможности программного обеспечения или текстовые ссылки относящиеся к памяти начальной загрузки, внутренней памяти или памяти на кристалле, могут быть отнесены ко всем процессорам ADSP-21xx, исключая ADSP-2100. Этот процессор не включает память на кристалле и не использует память начальной загрузки.

Каждая версия программного обеспечения поставляется с сопровождающими замечаниями. Эти замечания описывают текущую версию и дают информацию по любой замене программного обеспечения. ***Пожалуйста, позаботьтесь вернуть регистрационную карту с вашими координатами!*** *Это позволит нам поддерживать вас информацией о последующих версиях программного обеспечения.*

1.2. Содержание

Руководство содержит следующие главы, описывающие программное обеспечение:

- Глава 2: Системный конфигуратор

Системный конфигуратор это программный инструмент для описания особенностей оборудования. Вы создаете исходный файл состава системы, который содержит соотношение памяти ОЗУ и ПЗУ, размещение памяти программ и данных, размещение портов ввода/вывода целевого оборудования. Чтобы облегчить эту задачу используют высокоуровневые конструкции.

- Глава 3: Ассемблер

Ассемблер переводит ваши программы на языке ассемблер. Он поддерживает синтаксис инструкций семейства ADSP-2100 и поддерживает возможность гибких макропроцессов. Препроцессор поддерживает директивы С препроцессора в исходном коде. Исходный код разделяют в определенные установками модули (файлы). Прилагается полный набор диагностики.

- Глава 4: Редактор связей

Редактор связей осуществляет процесс редактирования отдельно -ассемблированных модулей и генерирует исполняемый файл. Он может выполнить связывание модулей в библиотеку исполняемых подпрограмм. Он размещает скомпонованную программу и данные в аппарате целевой системы, как указано в выходном файле построителя системы и может создать несколько загружаемых страниц исполняемых файлов для процессоров с загружаемой памятью.

- Глава 5: Разделитель программ для записи в ППЗУ

Разделитель программ для записи в ППЗУ использует выходные данные редактора связей и генерирует файл для записи в ППЗУ в форматах различных промышленных стандартов. Форматы этих файлов записаны в Приложении B.

Следующая информация не включена в перевод, и может быть найдена в оригинале руководства *ADSP-2100 Family Assembler Tools & Simulator Manual*.

- Главы 6 – 13: Программа моделирования (симулятор)
- Приложение A, Коды инструкций.
- Приложение B, Форматы файлов
- Приложение C, Сообщения об ошибках
- Приложение D, Таблицы векторов прерываний
- Приложение E, Сообщения об ошибках симулятора.
- Приложение F, Форматы файлов данных симулятора.

1.3. Процесс разработки системы

Рис.1.1 показывает этапы процесса разработки системы ADSP-21xx. Процесс разработки начинается с задачи определения аппаратного обеспечения системы. Определить аппаратное окружение можно с использованием программного инструмента –системного конфигуратора. Необходимо написать файл описания системы, который будет содержать входные данные для системного конфигуратора; этот файл специфицирует конкретную систему. Системный конфигуратор считывает данные и генерирует файл описания архитектуры системы, который предоставляет информацию о системе редактору связей, программе моделирования и эмулятору (в случае его использования).

Вы начинаете написание программы с создания исходных модулей на языке ассемблер. Ассемблерный программный модуль это термин языка ассемблер включающий главную программу, подпрограмму или объявление переменных. Каждый программный модуль обрабатывается ассемблером отдельно.

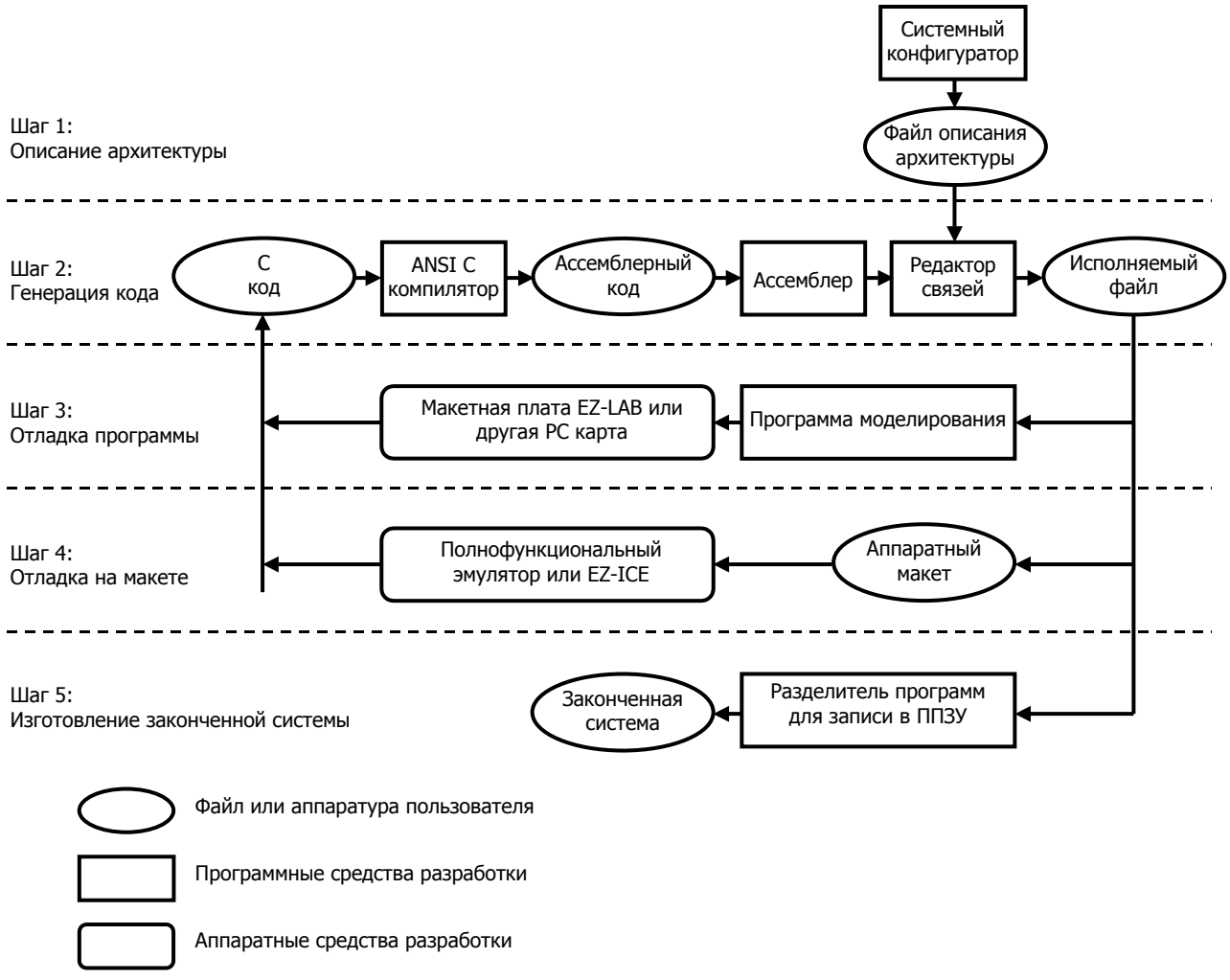


Рис.1.1. Процесс разработки системы ADSP-21xx.

Редактор связей читает информацию о конкретной системе из файла описания архитектуры, чтобы определить размещение фрагментов программы и данных. В ассемблерных модулях вы можете определять каждый фрагмент программы/данных, как полностью перемещаемый, перемещаемый в пределах определенного сегмента памяти, или неперемещаемый (размещаемый по абсолютному адресу). Неперемещаемые модули программ или данных размещают по указанным адресам памяти. Перемещаемые объекты размещаются в памяти редактором связей.

Пользуясь файлом описания архитектуры и ассемблированными программными модулями, редактор связей определяет размещение перемещаемых модулей программ и данных и присваивает всем модулям в памяти адреса с правильными атрибутами (CODE или DATA, RAM или ROM). Редактор связей генерирует исполняемый файл, содержащий копию содержимого памяти, который может быть использован для проверки программой моделирования или эмулятором.

Симулятор предусматривает окна, которые показывают различные части аппаратных средств. Для отображения аппаратных средств, симулятор конфигурирует собственную память в соответствии с файлом описания архитектуры и моделирует порты ввода/вывода отображенные в памяти. Моделирование позволяет провести отладку системы и проанализировать исполнение программы перед загрузкой в аппаратный макет.

После полного моделирования вашей системы и проверки программного обеспечения, к аппаратному макету подключают эмулятор, чтобы проверить схему, синхронизацию и выполнение программы в реальном режиме времени. Эмулятор имеет оверлейную память, которая может быть использована вместо компонент памяти конкретной системы.

Разделитель программ для записи в ППЗУ - это программный инструмент, переводящий выходные программы редактора связей (копии содержимого памяти) в формат файла промышленного стандарта для программатора ППЗУ. После того, как вы запрограммировали микросхему ППЗУ и подключили ее к процессору ADSP-21xx, размещенному на целевой плате, ваша система готова к запуску.

1.4. Константы

Константы включают в себя числовые константы и символы определенные как константы. Символические константы могут использоваться вместо числовых. Символ должен быть объявлен как константа для системного конфигуратора или ассемблера с использованием директивы `.CONST`. Объявление константы для системного конфигуратора не переносится в ассемблер, таким образом вы должны написать новые объявления для констант в вашей исходной программе на языке ассемблер.

1.5. Основания счисления

Основания счисления, которые могут быть использованы в исходной программе включают шестнадцатеричную, восьмеричную, двоичную и десятичная. Их обозначают следующим образом.

Для шестнадцатеричных чисел, префикс `0x` (ноль и x) или `H#`:

```
0x24FF      H#CF8A
```

Для восьмеричных , префикс `0` (ноль):

```
0777
```

Двоичные числа обозначаются с префиксом `B#`:

```
B#01110100
```

Десятичные числа используются по умолчанию (префикс отсутствует):

```
1024        +1024       -55
```

1.6. Набор символов

Программное обеспечение средств разработки ADSP-21xx распознает следующие символы:

- Буквы верхнего регистра от «A» до «Z»
- Буквы нижнего регистра от «a» до «z»
- Цифры от «0» до «9»
- ASCII графические символы - т.е. печатные символы отличные от букв и цифр (пунктуации и т.д.)
- ASCII не графические: пробел, табуляция, возврат каретки, перевод строки, перевод страницы (символ или символы «новая строка» интерпретируются в соответствии с установками, при которых символ был получен.)

1.7. Символы

Символ это строка из знаков, используемая одним из двух способов. Символы, которые вы определяете во входном файле системного конфигуратора или программе на языке ассемблер используются для представления, например, сегмента памяти, адреса или значения данных. Другие символы резервируют ключевые слова распознаваемые системным конфигуратором или ассемблером. Зарезервированные символы приведены в главах 2 и 3.

Символ, определенный на языке ассемблер может быть именем модуля, переменной, буфером данных, программной меткой, портом ввода/вывода, макросом или константой.

Символы начинаются со знака из следующих наборов:

- прописные буквы от «A» до «Z»
- строчные буквы от «a» до «z»
- знак подчеркивания «_»

с последующими знаками из наборов:

- прописные буквы от «A» до «Z»
- строчные буквы от «a» до «z»
- знак подчеркивания «_»
- цифры от «0» до «9»

Другими словами, ваши символы не должны начинаться с цифры. Символ может быть длиной до 32 знаков. Здесь приведено несколько примеров типичных символов и что они могут обозначать:

`main_prog`	программный модуль на языке ассемблер
`xoperand`	переменная данных
`input_array`	буфер данных
`subroutine1`	программная метка
`AtoD_INPUT`	порт, отображенный в памяти

1.7.1. Чувствительность к регистру

Программное обеспечение средств разработки на языке ассемблер ADSP-21xx может быть установлено как чувствительным к регистру, с различием между буквами верхнего и нижнего регистров, так и нечувствительным к регистру, принимая буквы верхнего и нижнего регистров за одни и те же знаки.

По умолчанию чувствительность к регистру отключена, и вы можете вводить текст в любых комбинациях верхнего и нижнего регистров. Язык C, тем не менее различает регистр символов, и если вы используете компилятор C семейства ADSP-2100, то системный конфигуратор и ассемблер должны быть также установлены чувствительными к регистру. Это достигается включением в командную строку ключа (`-c`). Для уточнения деталей смотрите главы 2 и 3.

1.8. Выражения ассемблера

Ассемблер семейства ADSP-2100 может преобразовывать отдельные выражения в исходном коде. Выражение может быть использовано там, где вычисляется числовая константа .

Разрешены два вида выражений:

- арифметические или логические операции с двумя или более целыми константами

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

- символ + или - целая константа

  примеры: `data–8` `data_buffer+15` `startup+2`

Символы могут быть переменными, буферами данных или программными метками. Все эти символы представляют собой значение адреса, которое определяется редактором связей. Сложение или вычитание константы обозначает смещение от адреса.

Простые арифметические или логические выражения могут быть использованы для объявления символических констант с помощью директивы `.CONST`. Эти выражения могут использовать следующие наборы операторов, распознаваемых языком программирования C (записаны в порядке предпочтения):

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

Выражения также могут быть использованы при вводе команды в одной из программ моделирования семейства ADSP-2100. Симуляторы распознают в дополнительный набор выражений и операторов, которые описаны в разделе «Выражения» главы 6.

Самое важное различие между ассемблерными выражениями и выражениями симулятора состоит в том, что содержимое памяти (например, переменные) и содержимое регистров процессора могут быть использованы как операнды *только в симуляторе.* Ассемблер не может преобразовывать значения в памяти и значения регистров во время ассемблирования, однако, симулятор имеет доступ к мгновенным значениям в памяти и регистров во время моделирования.

1.9. Принятые соглашения

Этот раздел приводит список принятых в этом руководстве соглашений.

- Ключевые слова (символы зарезервированные системой) представлены в тексте знаками верхнего регистра, хотя реально они могут быть введены знаками любого регистра. Обе формы ключевых слов зарезервированы.

- Символы в нижнем регистре выделенные курсивом, например, *`jumplabel`*, обычно представляют символы определяемые пользователем. такие как программная метка, переменная или имя файла.

- Квадратные скобки, [ ] , заключают необязательные величины, размер сегмента памяти (в директиве `.SEG` системного конфигуратора) или длину буфера данных (в директиве ассемблера `.VAR`)

- Многоточие, . . . , показывает , что предшествующие элементы могут быть повторены.

- Термин **ADSP-21xx** используется для ссылки на один или все процессоры семейства ADSP-2100.

- Термин **ADSP-2100** используется для обозначения как процессора ADSP-2100, так и процессора ADSP-2100A.

- Сокращения **DM, PM** и **BM** используются вместо памяти данных, памяти программ и памяти начальной загрузки, соответственно.

- С того момента, когда директива ассемблера `.VAR` была использована для объявления и однословной переменной данных, и многословных буферов данных, термин **буфер данных (*data buffer*)** обозначает как переменную, так и буфер.

1.10. Установка и замечания по версии

Детали процесса установки программного обеспечения могут отличаться от версии к версии; замечания по версии сопровождают каждый новый релиз должны и содержать эти детали. Вы должны всегда проверять выполнение указаний для используемой версии перед продолжением установки.

Инструкции по установке для различных базисных платформ приведены отдельно для каждой версии.

1.10.1. Файлы и переменные окружения

Программа установки создает на вашем жестком диске различные поддиректории и файлы. Файлы могут размещаться в каталоге, принятом по умолчанию или в другом каталоге, заданном пользователем. Вы должны найти установленными следующие исполняемые программные файлы:

Имя файла	*Описание*
BLD21.EXE	системный конфигуратор
ASM21.EXE	ассемблер С препроцессор
ASMPP.EXE	ассемблер препроцессор
ASM2.EXE	ассемблер
LD21.EXE	редактор связей (компоновщик)
LIB21.EXE	библиотекарь (программа для работы с библиотеками программ)
SPL21.EXE	разделитель программ для программатора ППЗУ
HSPL21.EXE	разделитель программ для загрузки через порт HIP (для использования с ADSP-2111 и ADSP-21msp50)

Новые версии программного обеспечения могут включать другие файлы. Смотрите информацию по изменениям в замечаниях.

Создаются следующие переменные окружения, которым во время установки присваиваются значения по умолчанию:

Переменные окружения	*Описание*
ADI_DSP	путь к директории содержащей инсталлированные файлы
ADII	путь(и) к директориям INCLUDE, используемых ассемблером

Это полный набор переменных окружения для программного обеспечения средств разработки семейства ADSP-21xx, включая программу моделирования (симулятор) и С компилятор.

Переменные окружения создаются когда вы устанавливаете любую часть программного обеспечения. Переменные окружения требуются для правильной работы программного обеспечения.

Когда программное обеспечение успешно установлено, вы подготовлены к тому, чтобы писать программу и использовать программные инструменты.

1.10.2. Пример архитектуры и исходные программные файлы

Несколько примеров программирования систем включено в программное обеспечение. Эти файлы расположены в директории названной \EXAMPLES, которая создается в корне каталога, выбранного при установке. Примеры включены для того, чтобы помочь вам научиться как писать программы для ADSP-21xx и использовать программное обеспечение.

Файлы названы в соответствии с номером примера. Каждый пример включает файл системной архитектуры .SYS, один или более файлов ассемблерных модулей .DSP, и командный файл .BAT. Запуск командного файла будет вызывать системный конфигуратор, ассемблер и редактор связей в порядке создания исполняемого .EXE файла, который может быть загружен и запущен на симуляторе ADSP-21xx.

Системный конфигуратор 2

2.1. Введение

Системный конфигуратор это программный инструмент для описания вашего оборудования. Каждая система ADSP-21xx может иметь уникальную аппаратную конфигурацию и использовать различную структуру памяти. Системный конфигуратор вводит определения вашей аппаратной конфигурации, включая память и порты ввода/вывода, в виде файла, читаемом редактором связей и программой моделирования (симулятором). Редактору связей эта информация требуется для распределения хранения программ и данных в доступном пространстве памяти. Симулятор должен моделировать вашу системную архитектуру и процессор ADSP-21xx, базирующийся на ней.

Системный конфигуратор может быть использован для предварительной установки распределения памяти для эмулятора ADSP-21xx. Подробнее данный вопрос рассмотрен в разделе «Распределение сегментов для эмулятора».

Рис.2.1 показывает конфигурацию памяти, позволяющую адресовать максимальное число слов в каждом пространстве памяти для каждого процессора. Системный конфигуратор позволяет создавать описания системной архитектуры только в пределах этих ограничений.

	ADSP-2100 (вся память внешняя)	ADSP-2105 ADSP-2115	ADSP-2101 ADSP-2111
Память данных 16-бит данные	16 К максимум	14.5 К максимум (0.5 К внутренней, 14 К внешней)	15 К максимум (1 К внутренней, 14 К внешней)
Память программ 24-бит программы, или 16/24-бит данные	16 К смешанной памяти программ и данных, или 32 К (16 К программ и 16 К данных)	15 К максимум (1 К внутренней, 14 К внешней)	16 К максимум (2 К внутренней, 14 К внешней)
Память начальной загрузки 24-бит программы, 16/24-бит данные, дополненные до 32-битных слов	----------	8 К максимум (32 Кб, организованных в 32-битные слова)	16 К максимум (64 Кб, организованных в 32-битные слова)

Рис.2.1. Конфигурации памяти семейства ADSP-2100 .

Каждое пространство памяти адресуется раздельно. Адреса в памяти программ отличаются от адресов в памяти данных. Память начальной загрузки адресуется особым образом и используется процессором во время загрузки (**Примечание**: в ADSP-2100 память начальной загрузки отсутствует).

Вы должны указать для системного конфигуратора *исходный файл системной конфигурации* (*system specification source file*), обычно с расширением `.SYS`; этот файл описывает ваше целевое оборудование. В этой главе приведены директивы системного конфигуратора, которые используются для написания файла.

Системный конфигуратор обрабатывает входной файл и создает *файл описания архитектуры* (*architecture description file*) с расширением `.ACH`. Файл описания архитектуры расшифровывается редактором связей для определения размещения программ и фрагментов данных в памяти. Этот файл используется и симулятором для моделирования конфигурации памяти системы и для установки размещения памяти целевой системы эмулятором. Системный конфигуратор выводит сообщения об ошибках, если они обнаружены, если они не обнаружены, показывает на экране полную архитектуру. Если необходимо обратится к полной архитектуре с целью отладки или документирования, вам потребуется перенаправить этот вывод в файл, используя действующую систему команд операционной системы.

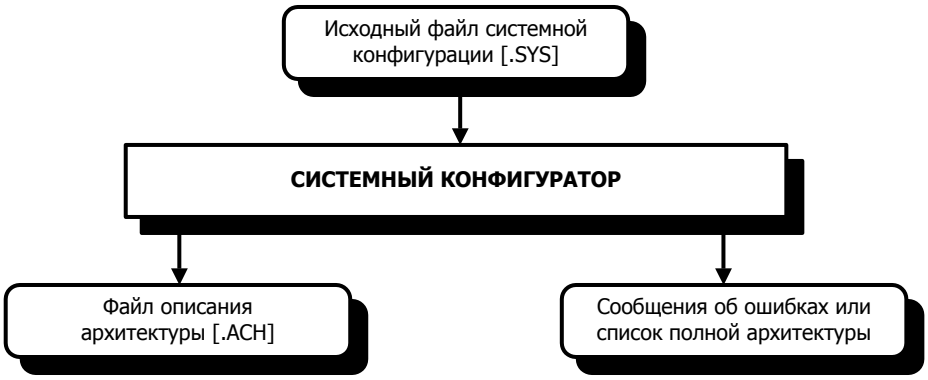


Рис.2.2. Вход/выход системного конфигуратора.

2.1.1. Регистры управления отображенные в памяти

Все процессоры ADSP-21xx, кроме ADSP-2100, имеют набор *регистров управления отображенных в памяти* (*memory-mapped control registers*), которые конфигурируют различные режимы работы процессора. Эти регистры расположены в скрытой части внутренней памяти. Это пространство памяти находится в верхней части 1K внутренней памяти данных по адресам 0x3C00-0x3FF. Вы не можете объявлять сегмент памяти в этом пространстве, и управляющие регистры не могут быть определены во входном файле системного конфигуратора. Вместо этого, каждый регистр должен быть инициализирован в вашей программе на языке ассемблер записью слов данных по присвоенному адресу. (Запись и чтение в регистры разрешены, даже если этот сегмент памяти не может быть объявлен директивой системного конфигуратора `.SEG`). Адрес и формат каждого регистра управления приведены в приложении E этого руководства и в описании регистров управления/состояния *ADSP-2100 Family User's Manual.*

2.2. Запуск системного конфигуратора

Для запуска системного конфигуратора наберите:

```
BLD21 filename[.ext] [-c]
```

где `filename` это ваш исходный файл системной конфигурации. Имя файла может иметь расширение, если оно отсутствует, системный конфигуратор добавляет по умолчанию расширение `.SYS`. Системный конфигуратор создает выходной файл описания архитектуры `filename.ACH`, использующий имя вашего входного файла .

Существует единственный дополнительный ключ включаемый в командную строку вызова системного конфигуратора. Ключ `-c` делает системный конфигуратор зависимым к регистру, различая использование верхних или нижних регистров (прописных или строчных) символов. Этот ключ предназначен для совместимости с компилятором семейства ADSP-2100.

Если ключ `-c` не используется, то выход системного конфигуратора составляют символы верхнего регистра. Вы должны использовать это ключ для сохранения введенных в нижнем регистре символов (строчных букв). При использовании этого ключа ассемблер различает регистр, как это требуется при ассемблировании скомпилированного C кода. Если вы обратитесь (в ассемблерном коде) к сегменту памяти в нижнем регистре написания, и ассемблер запущен в режиме зависимости к регистру, то имя сегмента не будет распознано до тех пор, пока системный конфигуратор не будет запущен с ключом `–c`. Если вы забыли синтаксис командной строки системного конфигуратора, наберите:

```
BLD21 -help
```

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

2.3. Использование символов и зарезервированные слова

В файле описания конфигурации вы назначаете символические имена целевой системе, сегментам памяти и портам ввода/вывода отображенным в памяти. Вы можете использовать имена сегментов памяти в ваших программах, чтобы назначать фрагменты программ и данных этим сегментам. Это назначение проходит от ассемблера к редактору связей и определяет точное размещение вашей программы в памяти. Все символические имена должны быть уникальны. Символическое имя это строка из букв, цифр и подчеркнутых знакомест, или подчеркнутое знакоместо перед первым символом. Небольшая группа символов резервируется для использования в качестве ключевых слов системного конфигуратора - вы не можете использовать эти символы в ваших файлах описания конфигурации. Таблица 2.1 приводит ключевые слова системного конфигуратора.

ABC	CODE	PM	DM
ADSP2100	DATA	PORT	ROM
ADSP2101	CONST	SEG	RAM
ADSP2105	ENDSYS	SYSTEM	BOOT
ADSP2111	ADSP2150	MMAP0	MMAP1

Таблица 2.1. Ключевые слова системного конфигуратора.

Ключевые слова ассемблера представлены в главе 3, они также зарезервированы для использования в программах средств разработки. Пожалуйста проверьте этот список перед тем, как вы выберете имена для ваших системных компонент. При использовании зарезервированных слов редактор связей выдаст сообщение об ошибке.

2.4. Файл системной конфигурации

Исходный файл системной конфигурации определяет соотношение памяти программ и данных в вашей системе. Для процессоров с памятью начальной загрузки файл объявляет также каждую страницу памяти начальной загрузки, которая будет использоваться. Комментарии заключаются в скобки {} и могут быть расположены в любом месте файла. Вложенные комментарии не разрешены.

Собственный файл можно создать с помощью любого простого текстового редактора. Не используйте текстовые процессоры, например Microsoft Word, которые вставляют специальные управляющие коды.

2.4.1. Пример файла системной конфигурации ADSP-2100

На Рис.2.3 приведен пример исходного файла системной конфигурации для ADSP-2100. (Термин «ADSP-2100» используется для определения ADSP-2100 и ADSP-2100A).

```
.SYSTEM fir_system;                          {системное имя}
.ADSP2100;                                   {определение процессора}
.SEG/PM/ROM/ABS=0/CODE        prog_mem [4096];  {программа}
.SEG/PM/RAM/ABS=4096/DATA     coeff_table [15]; {таблица коэффициентов}
.SEG/DM/RAM/ABS=0/DATA        delay_line [15];  {данные}
.PORT/DM/ABS=16382            ad_sample;        {порт отраженный в памяти}
.PORT/DM/ABS=16383            da_data;          {порт отраженный в памяти}
.ENDSYS;
```

Рис.2.3. Файл системной конфигурации ADSP-2100.

Первая директива в файле .SYSTEM. Она назначает имя fir_system описанию архитектуры и обозначает начало файла. Команда .ADSP2100 идентифицирует тип процессора, в приведенном примере это процессор ADSP-2100. Данная команда обязательна.

Директивы .SEG объявляют сегменты системной памяти и их характеристики. Сегменты памяти могут быть объявлены в любом порядке. В этом примере объявлены три сегмента. Первый, prog_mem - это сегмент размером 4 Кслова, который будет хранить код программы. Сегмент coeff_table - это блок размером 15 слов памяти программ объявлен для хранения данных; данные устанавливают коэффициенты КИХ-фильтра. Третий сегмент, delay_line, требуется для сохранения промежуточных данных алгоритма фильтрации; этот сегмент находится в памяти данных ADSP-2100.

Две директивы .PORT объявляют порты ввода/вывода отраженные в памяти. Имена портов ad_sample и da_data предполагает внешнюю связь с АЦП и ЦАП. Порты расположены в памяти данных с абсолютными адресами, заданными параметром ABS. Имена портов становятся символами, которые могут использоваться в программе для чтения и записи данных. Последнее объявление в файле системной конфигурации это директива .ENDSYS. Системный конфигуратор завершает свою работу при достижении директивы .ENDSYS.

2.4.2. Пример файла системной конфигурации ADSP-2101

На Рис.2.4 приведен пример исходного файла системной конфигурации для ADSP-2101. Первая директива в файле это директива .SYSTEM. Она присваивает имя fir_system описанию архитектуры и обозначает начало файла. Команда .ADSP2101 идентифицирует тип процессора, здесь подразумевается процессор ADSP-2101. Данная команда обязательна.

Директива .MMAP0 определяет в этой системе состояние вывода MMAP на ADSP-2101. Определение MMAP как 0 означает, что программа в памяти начальной загрузки должна быть загружена во внутреннюю память программ начиная с адреса 0x0000.

Директива .SEG объявляет сегменты системной памяти и их характеристики. Сегменты памяти могут быть объявлены и другим способом. В этом примере сегменты объявлены в полной конфигурации внешней и внутренней памяти программ и данных ADSP-2101.

```
.SYSTEM fir_system;                                  {системное имя}
.ADSP2101;                                           {определение процессора}
.MMAP0;                          {загрузка программы из памяти начальной загрузки}
.SEG/PM/ROM/BOOT=0 boot_mem [2048];  {страница блока памяти начальной загрузки}
.SEG/PM/RAM/ABS=0/CODE/DATA    int_pm [2048];    {внутренняя память программ}
.SEG/PM/RAM/ABS=2048/CODE/DATA ext_pm [14336];   {внешняя память программ}
.SEG/PM/RAM/ABS=0/DATA         ext_dm [14336];   {внешняя память данных}
.SEG/PM/RAM/ABS=0/DATA         int_dm [1024];    {внутреняя память данных}
.ENDSYS;
```

Рис.2.4. Файл системной конфигурации ADSP-2101.

(**Примечание**: обращение к памяти программ и данных, не включает память начальной загрузки, которую следует рассматривать как уникальную область памяти в системной архитектуре). Сегмент boot_mem это сегмент из 2 Кслов одной страницы памяти. Если система основана на ADSP-2105, размер сегмента должен быть 1К (или меньше) .

Объявление int_pm идентифицирует внутреннюю (расположенную на чипе) память программ размером 2 Кслова, начинающейся с адреса 0x0000. В ADSP-2101 (также как и в ADSP-2105, ADSP-2111, ADSP-21msp50) эта память может хранить как программу, так и данные и должна быть объявлена.

Следующая строка объявляет ext_pm, как сегмент размером 14 Кслов внешней памяти программ, начинающейся с адреса 2048, которая может содержать программу и данные. Следующая строка объявляет ext_dm как сегмент размером 14 Кслов внешней памяти данных, начинающейся с адреса 0. Предпоследняя строка объявляет int_dm как сегмент размером 1 Кслов внутренней памяти данных, начинающейся с адреса 14336. Верхние 1К памяти данных резервируется под регистры управления, отраженные в памяти и не могут быть объявлены как сегмент.

Последняя строка в файле системной конфигурации включает директивы .ENDSYS. Системный конфигуратор завершает свою работу при достижении директивы .ENDSYS.

2.5. Директивы системного конфигуратора

Этот раздел описывает директивы системного конфигуратора и их синтаксис. Формат некоторых директив включает аргументы и параметры. Параметры следуют сразу за директивой и отделяются косой чертой, /, слешем; аргументы следуют за параметрами. Основная форма директивы выглядит следующим образом:

```
.DIRECTIVE /параметр/параметр/..аргумент;
```

2.5.1. Название системы (.SYSTEM)

Директива .SYSTEM должна быть первой командой в исходном файле системной конфигурации. Вы присваиваете имя вашей системе ADSP-21xx, указывая его в аргументе этой директивы. Системное имя будет показано в программе моделирования. Директива .SYSTEM имеет формат:

```
.SYSTEM имя_системы;
```

Директива .ENDSYS должна быть последней командой в файле. Системный конфигуратор останавливает свою работу на директиве .ENDSYS. Директива .ENDSYS имеет форму:

```
.ENDSYS;
```

2.5.2. Определение процессора (.ADSP21XX)

Эта директива определяет какой процессор семейства ADSP2100 используется в вашей системе. Эта информация передается редактору связей и симулятору через выходной файл с расширением .ACH.

После этого редактор связей в состоянии определить место расположения программы и данных в соответствии с организацией памяти для каждого процессора. Эта директива принимает одну из следующих форм:

```
.ADSP2100;       для ADSP-2100 и ADSP-2100A
.ADSP2101;
.ADSP2105;
.ADSP2111;
.ADSP2150;       для ADSP-21msp50 и ADSP-21msp55
.ADSP2151;       для ADSP-21msp51 и ADSP-21msp56

.ADSP2101MV;     для вариантной системы ADSP-2101 (ADSP-2115)
.ADSP2101P;      для системы со страничной памятью ADSP-2101
```

Когда симулятор ADSP-21msp50 загружается с файлом архитектуры ADSP-21msp51, он автоматически конфигурирует себя для распределения памяти ADSP-21msp51. Если используется файл архитектуры ADSP-21msp51 и разряд ROMENABLE установлен в 1, программа моделирования конфигурирует память программ PM[0x800] - PM[0x1000] как ПЗУ. Разряд ROMENABLE отображается в окне регистров управления.

2.5.3. Вывод MMAP (.MMAP)

Эта директива используется только для тех процессоров семейства ADSP-2100, которые обладают встроенной памятью, памятью начальной загрузки и выводом MMAP (т.е. все за исключением ADSP-2100). Директива определяет логическое состояние вывода MMAP процессора в вашей целевой системе. Эта директива принимает одну из двух форм:

```
.MMAP0                вывод MMAP поддерживает низкий уровень
.MMAP1                вывод MMAP поддерживает высокий уровень
```

Если используется .MMAP0, загрузка программы из памяти начальной загрузки происходит после перезагрузки и начинается с адреса 0x000 внутренней памяти программ. Если используется .MMAP1, загрузка программы из памяти начальной загрузки недоступна и внутренняя память программ отображается на верхние адреса пространства памяти программ. Когда эта директива пропускается, программа моделирования принимает по умолчанию значение MMAP=1.

2.5.4. Объявление сегмента памяти (.SEG)

Директива .SEG определяет специальную секцию системной памяти и описывает ее атрибуты. Не существует разделения памяти по умолчанию – вы должны определить системную память с помощью директив .SEG самостоятельно. Эта информация передается редактору связей, симулятору и эмулятору через выходной файл с расширением .ACH. Директива .SEG имеет форму:

```
.SEG/параметр/параметр .имя_сегмента[размер];
```

Сегменту присваивают символическое имя имя_сегмента. Присвоенное имя позволит вам точно разместить фрагменты программы и данных в памяти. На языке ассемблер это достигается параметром SEG. Вы должны указать длину сегмента внутри скобок. Эта величина интерпретируется как количество слов (16-разрядных данных или 24-разрядных инструкций) в сегменте. Размер сегмента памяти данных в битах составляет 2 x числа слов, размер сегмента памяти программ в битах составляет 3 x числа слов. Размер сегмента памяти начальной загрузки в байтах составляет 4 x числа слов, из-за расширения памяти начальной загрузки дополнительным байтом для достижения выравнивая слова.

Для директивы .SEG необходимо указать два параметра:

```
PM или DM или BOOT=0-7          пространство памяти
RAM или ROM                     тип памяти
```

Четыре параметра являются необязательными:

```
ABS=address                     абсолютный стартовый адрес
DATA или CODE или DATA/CODE     что хранится в сегменте
EMULATOR или TARGET             распределение памяти для эмулятора
INTERNAL                        расположение в памяти процессора
                                ADSP-2101 с вариантной памятью
```

Параметры PM/DM/BOOT показывают какой сегмент пространства памяти занят памятью программ, памятью данных, памятью начальной загрузки.

Оставшиеся опции определяют тип памяти, начальный адрес сегмента, тип содержимого (данные и/или программы) и распределение памяти эмулятора. Если вы используете один из эмуляторов ADSP-21xx, обратитесь к разделу «Распределение сегментов для эмулятора».

Каждое пространство памяти адресуется раздельно: адрес 0x10 в памяти программ отличается от адреса 0x10 в памяти данных. Сегмент памяти программ PM может хранить только CODE (программы), только DATA (данные) или одновременно программы и данные. Если вы не укажете ни одну из опций, по умолчанию принимается хранение программ. Для сегмента PM содержащего программу и данные должны быть указаны оба параметра. Процессоры ADSP-21xx, требующие передачи данных из или в память программ должны быть объявлены с сегментами, которые имеют параметр данных. Если ваша система требует, чтобы исполняемая программа была записана или прочитана процессором, сегменты, чтобы быть доступными, должны быть объявлены с обоими параметрами CODE (программа) и DATA (данные) опциями.

Сегменты памяти данных DM должны быть только DATA; это значение принимается по умолчанию, если параметр DATA пропущен. Если сегменту DM присвоен параметр CODE выдается сообщение об ошибке. Сегменты памяти начальной загрузки принимают по умолчанию параметры CODE и DATA, с тех пор как в большинстве систем используется для хранения и данных, и программ, поэтому эти параметры могут пропущены. Параметр памяти начальной загрузки должен определять только номер страницы, например, BOOT=0. Необходимо разделить объявление сегмента для каждой страницы памяти начальной загрузки вашей системы.

Система может иметь до 8 страниц памяти начальной загрузки с номерами от 0 до 7. Каждая страница памяти начальной загрузки для ADSP-2101, ADSP-2111, ADSP21ms50 может сохранять до 2К программ и данных. Каждая страница памяти начальной загрузки для ADSP-2105, ADSP-2115 может сохранять до 1К слов. Параметр ABS для страниц памяти начальной загрузки не используется – системный конфигуратор самостоятельно присваивает соответствующие адреса.

2.5.4.1. Пример директивы .SEG

Пример

```
.SEG/PM/RAM/ABS=0/CODE/DATA  restart [2048];
```

объявляет сегмент памяти программ ОЗУ, называемый restart, который расположен по адресу 0. Сегмент может содержать до 2048 слов программ и данных.

Пример

```
SEG/ROM/BOOT/=0  boot_mem[1536];
```

объявляет сегмент памяти начальной загрузки boot_mem, который расположен на странице номер 0 памяти начальной загрузки (автоматически соответствует адресу 0 в памяти начальной загрузки). Длина сегмента составляет 1536 слов. Сегменты памяти начальной загрузки должны всегда быть объявлены как ROM (ПЗУ).

2.5.4.2. Распределение сегментов для эмулятора

Системный конфигуратор позволяет вам заранее установить распределение памяти для эмулятора ADSP-21xx через файл .ACH. Для этой цели используются два дополнительных параметра директивы .SEG:

```
      /EMULATOR        размещает сегмент для оверлейной памяти эмулятора
или
      /TARGET          размещает сегмент памяти целевой платы
```

Эти установки настраивают распределение памяти эмулятора, когда программа эмулятора будет запущена. Сегменты должны быть разделены на блоки по 1К или больше. После запуска эмулятора вы сможете изменить распределение памяти командами эмулятора.

Если вы используете эмулятор в автономном режиме (без целевой платы), вы должны распределить *все* сегменты памяти для эмулятора. Можно распределить все сегменты для эмулятора в течение начальной стадии интеграции аппаратной/аппаратно-программной части системы. Для разрешения вопросов обратитесь к *ADSP-21xx Emulator Manual*.

2.5.5. Порты ввода/вывода отображенные в памяти (.PORT)

Директива .PORT объявляет порт ввода/вывода отображенный в памяти. Вы должны присвоить уникальное имя и адрес каждому порту в вашей системе. Порты могут быть определены в памяти данных или памяти программ. Для процессоров с внутренней памятью, портам могут быть присвоены адреса только во внешней памяти.

Директива .PORT принимает одну из двух форм:

```
      .PORT/DM/ABS=address  port_name;
или
      .PORT/PM/ABS=address  port_name;
```

Опция DM показывает, что порт отображен в памяти данных; PM показывает, что порт отображен в памяти программ. Если параметр не задан, по умолчанию принимается DM.

Порт располагается по абсолютному адресу, который вы определите (параметром ABS), и присваивается символическому имени port_name. Этот символ могут использовать инструкции ассемблера для доступа к порту, например:

```
      .PORT/DM/ABS=0x0400   ad_sample;
```

Объявляет имя порта ad_sample, который отображен в памяти данных по адресу 0x4000 (шестнадцатеричный) или 1024 (десятичный). Ассемблерная программа обращается к этому символу, чтобы представить его редактору связей базирующемуся на содержании файла описания системной конфигурации .ACH. Отображение порта в памяти данных позволяет 16-разрядное чтение/запись, в то время как отображение порта в памяти программ допускает 16- или 24-разрядную передачу. (Обратитесь к *ADSP-2100 Family User's Manual* для описания 24-разрядной передачи данных).

2.5.6. Константы системного конфигуратора (.CONST)

Директива .CONST определяет константы системного конфигуратора. Однажды объявив символическую константу, вы можете использовать её вместо реального числа. Это символическое определение распознается только системным конфигуратором, поэтому не переходит на ассемблер и программу моделирования. Директива .CONST имеет форму:

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

В выражениях могут использоваться только арифметические или логические операции из двух или более целочисленных констант; использование символов не разрешено.

Одна директива .CONST может содержать одно или более объявлений констант на одной строке, разделенных запятыми. Список объявлений не может быть продолжен на следующей строке.

Чтобы приравнять символ *taps* к 15, для примера, вы должны написать следующую директиву .CONST:

```
.CONST taps=15;
```

2.6. Рассмотрение особенностей процессоров

Благодаря некоторым уникальным характеристикам ADSP-2100, ADSP-2105 и ADSP-2115, следующие руководящие указания должны быть приняты во внимание при написании файла системной конфигурации для этих процессоров.

2.6.1. Системы ADSP-2100

Для ADSP-2100 возможны две различные конфигурации памяти программ:

- 16К слов смешанных программ и данных

или

- 32К слов - 16К программ, 16К данных.

Расширенная конфигурация 32К требует использования вывода процессора PMDA как дополнительной (высшего порядка) адресной линии для памяти программ. Если эта конфигурация работает, нижние 16К должны быть только программными. Сегменты PM, объявленные в этой области, должны иметь параметр только CODE. Верхние 16К пространства памяти должны быть только данными. Сегменты PM, объявленные в этой области, должны иметь параметр только DATA.

Если ваша система ADSP-2100 включает 32К расширенного пространства памяти программ, системный конфигуратор будет выдавать сообщение об ошибке при попытке объявить сегмент PM с обоими параметрами CODE и DATA.

2.6.2. Системы ADSP-2105 и ADSP-2115

Поскольку ADSP-2105 и ADSP-2115 имеют половину внутренней памяти ADSP-2101, пространство внутренней памяти, которое может быть объявлено, ограничено системным конфигуратором. Вы не можете объявить сегменты памяти в любых порциях в диапазоне следующих адресов:

Внутренние адреса памяти данных	14848 – 15359	(0x3A00 - 0x3BFF).
Внутренние адреса PM (MMAP=0)	1024 – 2047	(0x3C00 - 0x07FF).
Внутренние адреса PM (MMAP=1)	15360 – 16383	(0x3C00 - 0x3FFF).

Эти диапазоны соответствуют верхним 1K внутренней памяти `PM` ADSP-2101 и верхним 1/2K внутренней памяти `DM` ADSP-2101, которые отсутствуют в ADSP-2105/ADSP-2115. Поскольку редактор связей выделяет информацию из файла `.ACH`, выработанного системным конфигуратором, он не размещает программы или данные в этих диапазонах памяти.

Симулятор ADSP-2101 используется для моделирования систем ADSP-2105 и ADSP-2115. При запуске этого симулятора с архитектурным файлом ADSP-2105 или ADSP-2115 (процессор с вариантной памятью `.ADSP2101MV`), симулятор конфигурируется на различное количество внутренней памяти. Часть памяти на чипе ADSP-2101, которая не существует в этих процессорах будет показана как несуществующая.

2.6.2.1. Создание 1K страниц начальной загрузки

Так как ADSP-2105 и ADSP-2115 имеют страницы начальной загрузки объемом 1K, ваше объявление страницы должно специфицировать сегмент размером 1024 слов (или меньше). Например::

```
.SEG/BOOT=0/ROM  page_0[1024];
```

При последующем использовании программы разделителя для подготовки исходного файла для программатора, необходимо включить ключ `-bs` для создания страниц размером 1K (смотрите главу 5, Разделитель программ для записи в ППЗУ).

2.6.2.2. Замена ADSP-2105/ADSP-2115 на ADSP-2101

ADSP-2105 и ADSP-2115 являются совместимыми по выводам с ADSP-2101, позволяя делать прямую замену компоненты в вашей системе. Существует несколько деталей, касающихся такой замены, которые должны быть приняты во внимание при проектировании.

Чтобы произвести замену без модификации оборудования, необходимо изначально разработать аппаратную и аппаратно-программную часть для использования загрузочных страниц размером 2K (которые хранят только 1024 слов). Для этого нужно объявить сегменты страниц начальной загрузки размером 2048 слов:

```
.SEG/BOOT=0/ROM  page_0[2048];
```

Поскольку ваша программа для ADSP-2105/ADSP-2115 должна быть загружена в 1К страницы, система не будет использовать 1К памяти между каждыми страницами программ/данных. Процессор будет правильно загружать каждую 1К страницу программы, игнорируя неиспользуемые пустые сегменты между ними.

Другими словами, загружаемые страницы размером 2К будут содержать программы/данные в нижней половине каждой страницы.(адреса слов 0-1023), а верхняя половина будет пустая. Когда вы замените систему на ADSP-2101, вы сможете использовать разделитель программ для записи в ППЗУ для создания страниц начальной загрузки размером 2К.

2.7. Процессоры ADSP-2101 с вариантной памятью

Процессоры с вариантной памятью, такие как ADSP-2115, это производные ADSP-2101, которые содержат различные конфигурации памяти, расположенной на кристалле. Чтобы поддержать моделирование процессоров с вариантной памятью, программное обеспечение средств разработки позволяет легко конфигурировать системный архитектурный файл и симулятор для процессора, который вы используете.

Системы, основанные на процессорах с вариантной памятью, могут быть определены с любым количеством памяти программ (PM) 0 – 16К слов и с любым количеством памяти данных (DM) 0 - 15К слов. Часть PM и DM пространства памяти могут быть свободно определены как ROM и/или INTERNAL (т.е. расположенная на кристалле). Эти определения вводятся в файле системной конфигурации .SYS. После обработки вашего .SYS файла системным конфигуратором, создается файл .ACH, который используется редактором связей и симулятором ADSP-2101.

Необходимо предпринять следующие действия для моделирования процессора ADSP-2101 с вариантной памятью:

1. Используйте следующую директиву системного конфигуратора в вашем .SYS файле системного конфигуратора:

   ```
   .ADSP2101MV
   ```

2. Для любых ваших сегментов памяти, расположенных в вариантной памяти на кристалле, используйте параметр INTERNAL в соответствующей директиве .SEG:

3. Для любых из ваших сегментов памяти ПЗУ, используйте параметр /ROM в соответствующей директиве .SEG.

4. Ассемблируйте, связывайте и моделируйте вашу программу обычным способом. Редактор связей и симулятор ADSP-2101 прочитают архитектурный файл .ACH, созданный системным конфигуратором, определяя внутренние и/или ПЗУ сегменты памяти соответственно.

2.8. Проектирование систем со страничной памятью

Программное обеспечение средств разработки позволяет проектировать ADSP-2101 системы, которые адресуются большим пространством внешней памяти, применяя схему со страничной памятью данных. Такое расширение можно применить только к памяти данных.

Рис.2.5 показывает пример систем такого типа, где память данных ADSP-2101 расширена тремя дополнительными страницами. Страница 0 - это стандартное пространство 16К памяти данных ADSP-2101. В системе со страничной памятью, страница 0 разделена на *пространство данных*, *пространство ввода/вывода* и память данных на чипе (адреса 0x3800-3FFF).

Величина `PAGESIZE` (размер страницы), которую вы должны обозначить в системном конфигураторе, определяет границу между *пространством данных* и *пространством ввода/вывода*. Величина `DMIOEND`, которую также необходимо определить и которая не может быть больше чем 0x37FF, это последний адрес *пространства ввода/вывода*. *Пространство ввода/вывода* содержит порты отраженные в памяти и специальную ячейку расположенную в памяти, называемую *регистр страницы памяти данных*.

Модули программ, буферы данных и переменные, которые вы хранятся в страничной памяти должны быть заключены внутри собственной страницы и не пересекать ее границы.

2.8.1. Функции системного конфигуратора для страничной памяти

Для создания файла архитектуры для системы со страничной памятью, необходимо использовать директиву `.ADSP2101P`. Например, чтобы организовать 4К страницы, нужно использовать следующие параметры в вашем `.SYS` файле :

```
.ADSP2101P/PAGESIZE=4096;          {страничная система памяти}
                                   {размер страницы 4К}
```

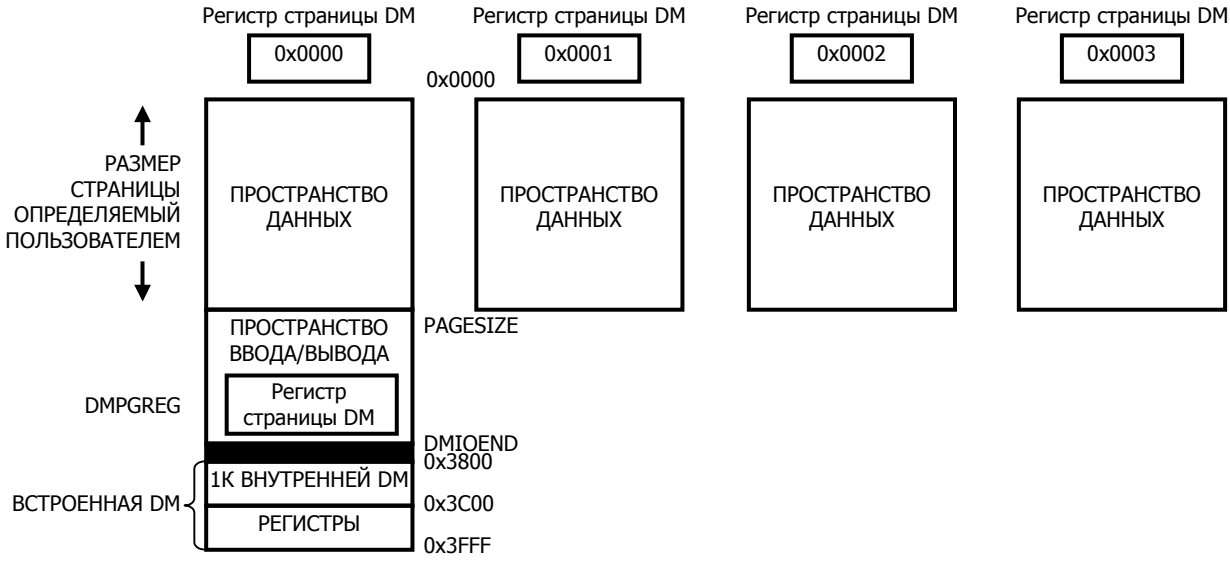


Рис.2.5. Страничная система памяти данных ADSP-2101.

Параметр PAGESIZE определяет количество слов пространства данных в каждой странице памяти. По умолчанию размер страницы равен 8192 словам. Вы должны также использовать системный конфигуратор, чтобы определить для вашей системы регистр страницы памяти данных. Этот регистр используется для адресации на различные страницы. В примере, показанном на Рис.2.5, значение регистра страницы памяти данных изменяется от 0 до 3.

Регистр страницы памяти данных определяется директивой .PORT и должен быть назван DMPGREG. Чтобы определить регистр страницы памяти данных с адресом 8192 (0x2000), необходимо произвести следующую запись:

```
.PORT/DM/ABS=0x2000 DMPGREG;      {регистр страницы памяти данных}
```

Регистр страницы памяти данных реализуется как регистр отраженный в памяти вашего системы. Значение регистра используется как старшие разряды адреса всей памяти данных; эти разряды производят выбор между различными страницами.

2.8.2. Использование имен сегментов для страниц

Специальный синтаксис параметра /ABS (директивы .SEG системного конфигуратора) позволяет вам определить и назвать сегмент памяти, который размещен на отдельной странице памяти данных. Формат этой директивы:

```
.SEG/параметры/ABS=pq#:addr  имя_сегмента[длина_сегмента];
```

Этот синтаксис определяет номер страницы и стартовый адрес сегмента (в десятичном формате от 0 до 16,384). Параметры DM, RAM/ROM и DATA должны быть также указаны. Например, чтобы определить имена сегментов для страниц памяти данных, показанных на Рис.2.5 должны быть использованы следующие директивы:

```
.SEG/DM/RAM/DATA/ABS=0:0  page0[4096];
.SEG/DM/RAM/DATA/ABS=1:0  page1[4096];
.SEG/DM/RAM/DATA/ABS=2:0  page2[4096];
.SEG/DM/RAM/DATA/ABS=3:0  page3[4096];
```

Ваш исходный C код и/или исходные ассемблерный модули могут быть размещены в одном из этих определенных на страницах сегментов, используя параметр ассемблера .SEG или ключ C компилятора -DMSEG.

2.8.3. Функции ассемблера для страничной памяти

Ассемблер распознает особую директиву, которая указывает на системы со страничной памятью. Эта директива называется .PAGE, и она должна быть применена во всех исходных модулях на языке ассемблер, которые составляют части системы со страничной памятью:

```
.PAGE;
```

Новый оператор ассемблера PAGE *buffer_name* используется для получения номера страницы (старших разрядов адреса) буфера данных или переменной:

```
AX0=PAGE array0;          (получить номер страницы array0)
```

Эта инструкция определяет номер страницы буфера array0 и загружает его в регистр AX0. Заметьте, что оператор PAGE работает подобно оператору получения указателя адреса (^) и оператору определения размера (%).

2.8.4. Функции С компилятора для страничной памяти

С компилятор распознает несколько параметров в командой строке, которые указывают на поддержку системы со страничной памятью. При компиляции программы для системы со страничной памятью используют ключ -FARDATA:

```
CC21 source.c -FARDATA
```

Ключ -FARDATA сообщает компилятору, что переменные и множества в файле source.c будут использоваться в системе со страничной памятью, и поэтому для этих переменных/множеств должны быть создана постранично адресуемая информация (т.е. использующая старшие разряды адреса). Для переменных и множеств, размещенных в DM, компилятор генерирует коды, которые включают номер страницы, содержащейся в регистре страницы памяти данных DMPGREG, предварительно определенном в системном конфигураторе. Если вы используете ключ -FARDATA для того чтобы сохранить данные в страничной памяти, вы должны определить имена сегментов, в которых ваши данные будут размещены. Ключ -FARDATA указывает компилятору разместить все данные из файла source.c в сегменте DM по умолчанию, называемом DDEFAULT. Необходимо определить DDEFAULT в системном конфигураторе перед компиляцией и редактированием связей. Для системы, показанной на Рис.2.5, страница 0 может быть определена как сегмент DDEFAULT следующей командой системного конфигуратора :

```
.SEG/DM/RAM/DATA/ABS=0:0  DDEFAULT[4096];
```

Ключ компилятора -DMSEG используется для переопределения размещения данных. Например, чтобы разместить данные DM из src.c в сегмент памяти, названный table3 (вместо DDEFAULT), необходимо указать в командной строке следующие параметры:

```
cc21 src.c -fardata -dmseg table3
```

Ключ -FARDATA оказывает несколько другое воздействие на компилируемый код и выполнение программы:

1. Стек исполняемой программы должен быть размещен во внутренней памяти ADSP-2101. По умолчанию стек располагается в памяти данных, если не используется ключ компилятора -pmstack.

2. Компилятор назначает страницу памяти данных каждой откомпилированной функции. При выполнении, она получает доступ только к тем данным, которые расположены на этой странице.

3. Часть функций С библиотеки рабочих программ используют небольшую часть памяти - эта память будет размещена во внутренней памяти ADSP-2101.

2.8.5. Использование адресов страниц в программе моделирования

При работе с симулятором ADSP-2101, когда моделируется система со страничной памятью, вы можете специфицировать адреса памяти информацией о странице. Синтаксис для адресации страничной памяти включает номер страницы и адрес (в десятичном формате от 0 до 16.383):

синтаксис	*пример*
`DM(pg#:addr)`	`dm(0:8193)`

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.)

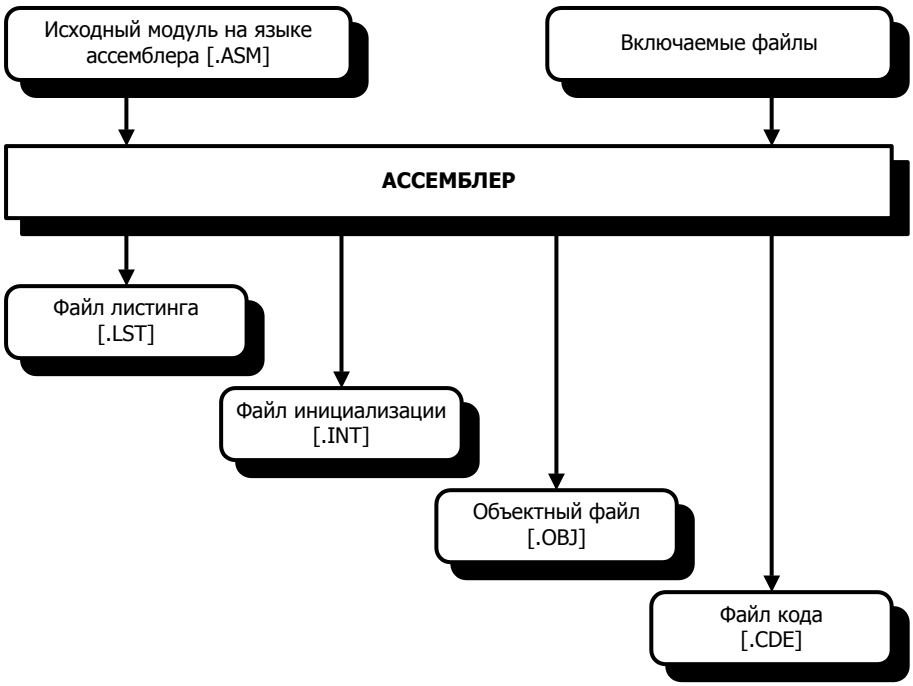


Рис.3.1. Вход/выход ассемблера.

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*/
```

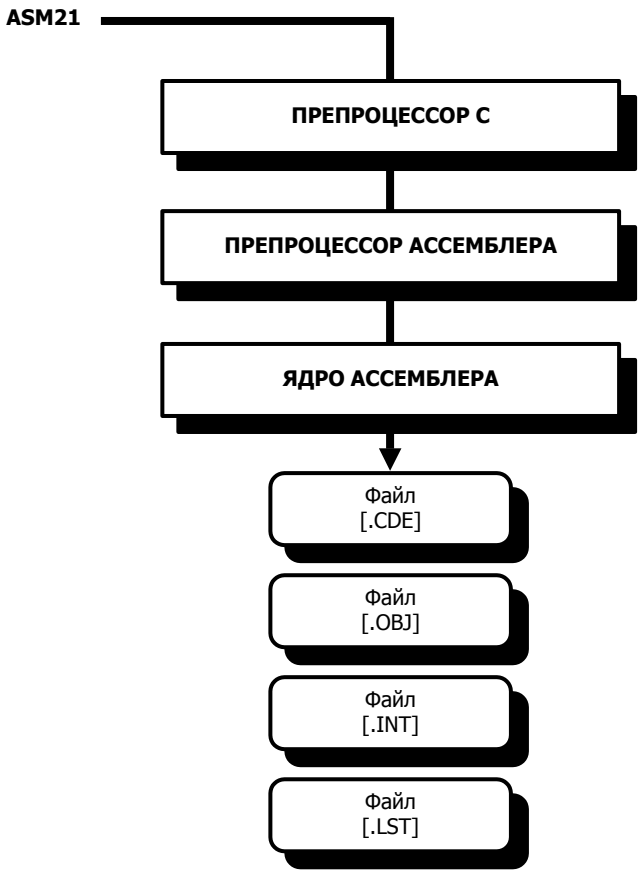


Рис.3.2. Процесс выполнения ассемблера.

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.

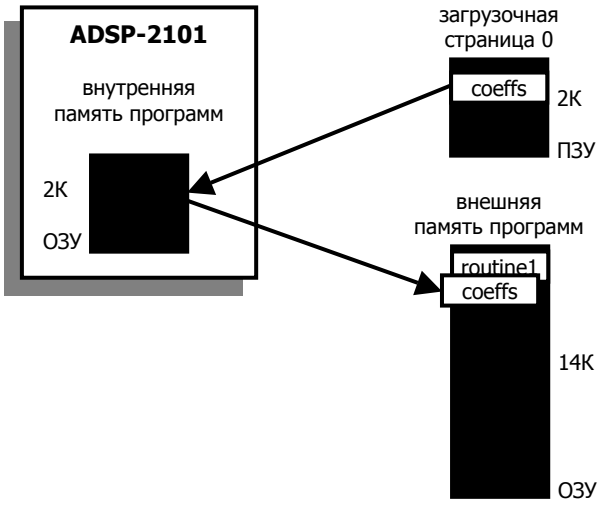


Рис.3.4. Перезапись астатического модуля.

```
.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;
```

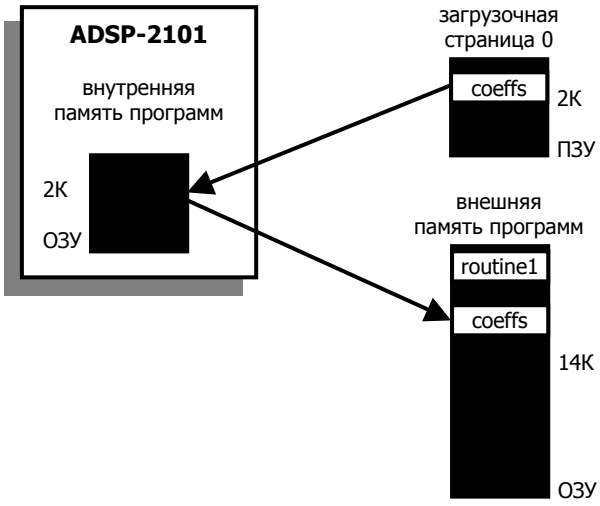


Рис.3.5. Защита статического модуля.

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];
```

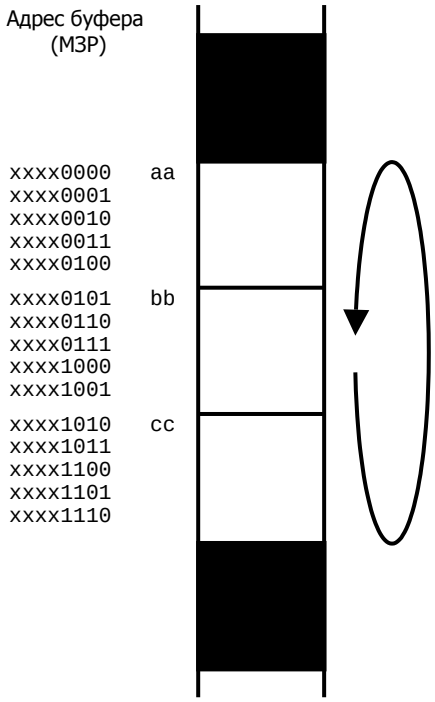
Базовый адрес кольцевого буфера равен адресу *aa*; этот символ будет использоваться для доступа к буферу в программе. Адрес *bb* это *aa+5*, а адрес *cc* это *aa+10*. Три пятисловных буфера могут быть индивидуально доступны как линейные.

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).



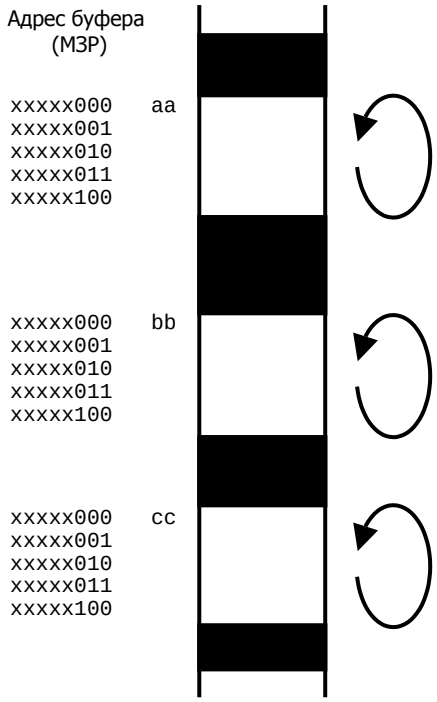


Рис.3.6. Составной кольцевой буфер. **Рис.3.7. Отдельные кольцевые буфера.**

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.

Редактор связей

4

4.1. Введение

Редактор связей (компоновщик) ADSP-21xx генерирует исполняемую программу путем связывания отдельно ассемблированных модулей. Основным выходом редактора связей является файл отображения памяти с расширением `.EXE`. Этот файл загружают в программу моделирования для отладки. После того, как программа полностью отлажена, применяют программу разбиения памяти для программатора ППЗУ.

Как было описано в предыдущей главе, ассемблер обрабатывает каждого модуль исходного кода и создает объектный файл (`.OBJ`), файл кода (`.CDE`) и файл инициализации (`.INT`). Объектный файл содержит информацию по размещению в памяти и символьную информацию, в то время как файл кода содержит коды операций ADSP-21xx с помеченными неразрешенными символами. Файл инициализации содержит информацию, относящуюся к переменным данным и буферам. Инициализация данных должна обеспечиваться файлом данных, указанным с помощью директивы ассемблера `.INIT`. Редактор связей читает данные из этого файла и объединяет их в файл `.EXE`. Изменения в инициализируемых данных потребует повторного вызова редактора связей. На следующей странице Рис.4.1. показывает входные и выходные файлы для редактора связей.

Редактор связей сканирует каждый ассемблированный модуль и разрешает обращения между модулями к глобальным и внешним символам. Он назначает (раздает) адреса перемещаемых программ и фрагментов данных. Редактор связей читает также файл описания архитектуры `.ACH` для создания карты системной памяти и размещения в ней программы и данных. Идентификация архитектурного файла для редактора связей происходит включением ключа `-a` в строке вызова программы.

Редактор связей может создавать три различных файла. Файл отображения памяти (*memory image file*) `.EXE` создается всегда – это исполняемая программа, которая содержит реальные коды операций и данные, которые должны быть расположены в памяти.

Вспомогательный файл распределения (*memory listing file*) `.MAP` суммирует информацию, относящуюся к созданной программе.

Вспомогательный файл таблицы символов (*symbol table file*) `.SYM` перечисляет все символы, встреченные редактором связей и их абсолютные адреса. Этот файл также используется программами моделирования (симуляторами) ADSP-21xx и эмуляторами.

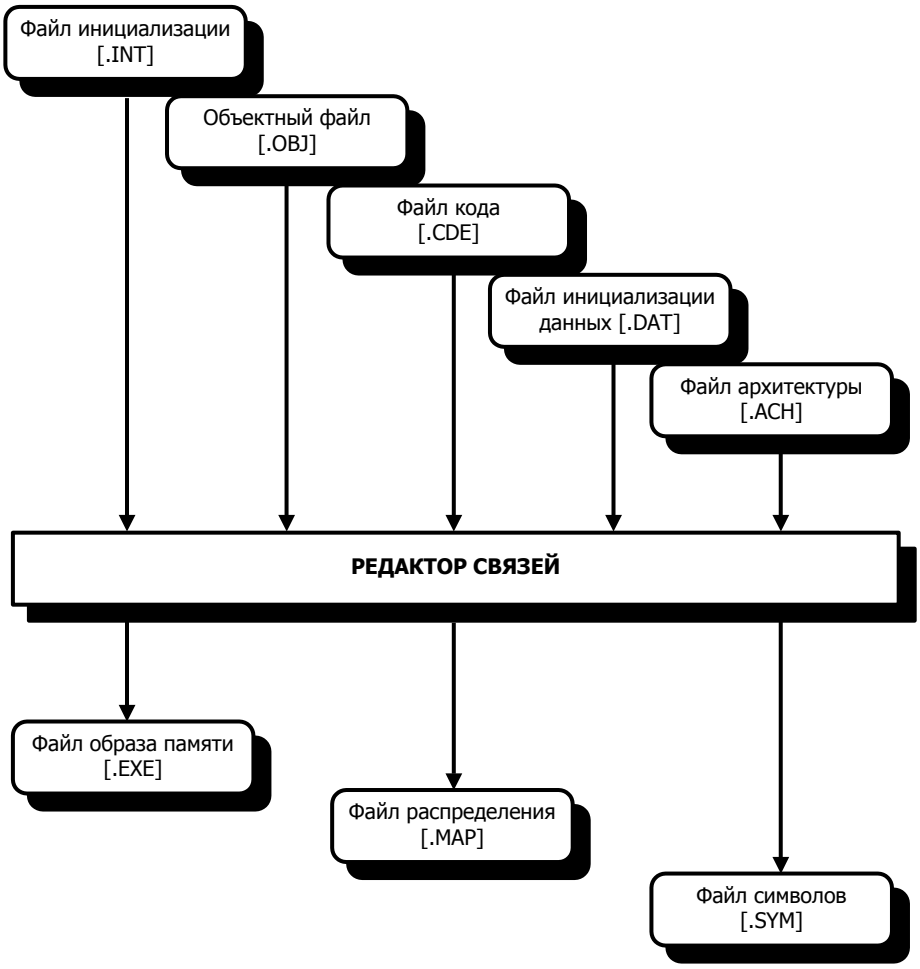


Рис.4.1. Вход/выход редактора связей.

Инициализирующие файлы данных (`.DAT`) не обозначены запуске редактора связей явно, поскольку они перечислены (директивой `.INIT`) в файле исходного кода. Файлы данных объединяют с помощью редактора связей. Если в файлах с данными сделаны изменения, необходимо просто запустить редактор связей заново. (**Примечание**: с тех пор как директива ассемблера `.VAR` используется для объявления как однословных переменных, так и многословных буферов данных, термин «буфер данных» включает в себя как переменные, так и буфера.)

4.2. Запуск редактора связей

Редактор связей вызывают перечисляя файлы, которые должны быть скомпонованы:

```
LD21  файл1 [файл2...] [-ключ ... ]
```

Например:

```
LD21  mine subl subl2  -a archfile
```

Каждый входной файл должен содержать только один модуль. Если файлы отсутствуют в текущем каталоге вашей операционной системы, с каждым именем файла должен быть путь к нему. Имена файлов должны идентифицировать ассемблерные выходные файлы без расширений (т.е. `.CDE`, `.OBJ`, `.INT`). Выходные файлы редактора связей получают по умолчанию имя 210X. Вы можете переименовать эти файлы, используя ключ `-e` (смотрите раздел «Параметры ключей редактора связей»).

Редактор связей может быть также вызван с ключом `-i`, который называет отдельный файл, содержащий список файлов для компоновки:

```
LD21  -i файл_все  [-ключ ...]
```

В этом случае редактор связей читает дополнительный файл *файл_все*, который является простым текстовым файлом, содержащим в каждой строке одно имя файла.

Другие ключи управляют различными операциями редактора связей. Они могут быть введены как в верхнем, так и в нижнем регистре. Несколько ключей должны быть отделены друг от друга хотя бы одним пробелом.

Редактор связей размещает память для модулей в соответствии с порядком, в котором они были перечислены. Для модулей, которые содержат объявления циклических буферов данных, изменения в порядке входных файлов могут определить, сможет ли программа быть успешно скомпонована в доступном пространстве памяти. Поэтому:

Модули, содержащие циклические буферы различных размеров должны быть перечислены для редактора связей в порядке понижения размеров буферов.

Это позволит редактору связей размещать вначале буфера большего размера, которые имеют большие ограничения на допустимые базовые адреса.

Если вы забыли синтаксис для вызова редактора связей, наберите в командной строке:

```
LD21  -help
```

Эта команда покажет синтаксис вызова программы и покажет список доступных ключей.

Если вы ассемблировали исходную программу, сгенерированную С компилятором, редактор связей должен быть вызван с ключом `-c`. Для получения детальной информации обратитесь к *ADSP-2100 Family C Tools Manual* и *ADSP-2100 Family C Runtime Library Manual*.

4.2.1. Размещение модулей на загрузочных страницах

Редактор связей предоставляет альтернативу ассемблерным директивам `.SEG/BOOT` для размещения модулей на страницах памяти начальной загрузки. Для этого в командном файле применяется специальный параметр. Чтобы заставить редактор связей разместить копию модуля на одной или более загрузочных страницах, необходимо перечислить имена файлов модулей следующим образом:

```
имя_файла  B  hh
```

Параметр *hh* представляет собой двухсимвольный шестнадцатеричный номер страницы памяти начальной загрузки. В этом 8-разрядном селекторе для страниц 0-7, любой разряд, установленный в «1» соответствует размещению модуля на этой странице:

$hh_{7\text{-}0}$ =

pg7	**pg6**	**pg5**	**pg4**	**pg3**	**pg2**	**pg1**	**pg0**

В следующем примере редактор связей разместит копии модуля `subprog` на загрузочных страницах с 0 по 4. Компоновщик вызывается с ключом `-i`, называя командный списочный файл. Заметьте, что селектор *hh* не требует шестнадцатеричного префикса «`0x`».

```
subprog  B  1F
```

4.2.2. Ключи редактора связей

Ключи редактора связей перечислены в таблице 4.1; часть из них требуют аргументы.

Ключ	*Воздействие*
-a *имя_файла*	указание файла описания архитектуры *имя_файла* .ACH
-c	создание стека для компилированных C программ (DM)
-dir *каталоги*;	указание директорий для поиска файлов библиотек
-dryrun	быстрый запуск для теста на ошибки (не создает .EXE файл)
-e *имя_файла*	присваивает выходным файлам имя *имя_файла*
-g	создает файл таблицы символов .SYM
-i *файл_все*	указание командного списочного файла
-lib	компоновка с C библиотекой рабочих программ (используется только с «-c»)
-p	размещает копию библиотечной подпрограммы на загрузочных страницах
-pmstack	перемещение C стека в память программ (PM) (используется только с «-c»)
-rom	используется ПЗУ версия C библиотеки программ (используется только с «-c»)
-s *размер*	создает динамическую память («кучу») C (используется только с «-c»)
-user *имя_файла*	поиск библиотечного файла, созданного утилитой построителя библиотеки LIB21
-x	создает файл распределения памяти .MAP

Таблица 4.1. Ключи редактора связей.

4.2.2.1. Указание файла описания архитектуры (-a)

Вы должны указать файл описания архитектуры .ACH с помощью ключа -a.. Имя файла может быть дано без расширения .ACH:

```
LD21  файл1  файл2  -a имя_файла
```

Редактору связей требуется информация об архитектуре системы, чтобы разместить программу и данные вашей программы в доступной системной памяти.

4.2.2.2. Создание стека исполняемой С программы (-c)

Ключ -c должен использоваться при компоновке программных модулей, сгенерированных компилятором семейства ADSP-2100. Эти программы требуют наличия стека исполняемой программы. Задание ключа -c приводит к следующему. Во-первых, редактор связей создает метку:

```
____top_of_ram        (метка с четырьмя символами подчеркивания)
```

которая присваивается самым верхним доступным адресам памяти данных (или память программ, если был задан ключ -pmstack). Во-вторых, редактор связей находит и подсоединяет исполняемый заголовок С программы, который представляет собой файл на языке ассемблере, требуемый для откомпилированных С программ. Метка ____top_of_ram используется исполняемой программой для расположения и инициализации стека в памяти процессора. Исполняемый заголовок программы устанавливает регистры, используемые для управления стеком и вызывает главную подпрограмму С программы. Стек не имеет ограничений по размерам; это позволяет увеличивать его (в направлении младших адресов).

Для каждого из процессоров ADSP-21XX используются различные заголовки исполняемой программы. Эти файлы поставляются с программным обеспечением компилятора С и имеют уникальное имя файла, указывающее с каким из процессоров его следует использовать. Необходимо выбрать соответствующие файлы для вашего системного процессора и переименовать их перед компоновкой в:

```
run_hdr.OBJ
run_hdr.CDE
run_hdr.INT
```

Редактор связей ищет файлы именно с этими именами. Вы можете также выбрать исходный ассемблерный файл .DSP для модификации. В этом случае вы должны отредактировать и заново ассемблировать модуль заголовка исполняемой программы перед компоновкой. Для получения детальной информации обратитесь к *ADSP-2100 Family C Tools Manual* и *ADSP-2100 Family C Runtime*.

Переменная окружения ADIRTH используется для поиска заголовка исполняемой программы. Редактор связей ищет этот файл в соответствии с назначенным путем. Вы должны установить переменную окружения, указывающую на каталог, в котором сохранен файл.

Например:

```
SET  ADIRTH=C:\ADI_DSP\LIB\
```

Обязательно должен присутствовать завершающий символ косой черты (слеш); не включайте дополнительные пробелы. Если редактор связей вызван с ключом `-c`, но не может найти файл заголовка исполняемой программы, это вызовет сообщение об ошибке.

4.2.2.3. Пути поиска для библиотечных подпрограмм (-dir)

Редактор связей позволяет вам использовать часто применяемые файлы с подпрограммами как библиотеки. Когда программа компонуется с вызовом одной из библиотечных подпрограмм, редактор связей автоматически разыскивает и присоединяет соответствующий файл. Необходимо указать редактору связей, где найти ваши библиотечные файлы, используя ключ `-dir` или переменную окружения `ADIL`. Редактор связей сначала осуществляет поиск по каталогам, указанным в ADIL, и затем смотрит аргументы ключа `-dir` (если необходимо). Например, чтобы установить переменную среды `ADIL` на каталог C:\DSP\LIBRARY\ вы должны ввести:

```
SET  ADIL=C:\DSP\LIBRARY\
```

Если перечисляются несколько путей DOS, они должны быть разделены точками с запятой. Обязательно должен присутствовать завершающий символ косой черты (слеш); не включайте дополнительные пробелы.

В операционной системе Unix на рабочих станциях Sun, аналогичная команда должна выглядеть следующим образом:

```
setenv  ADIL "/dsp/library/INCLUDE/"
```

4.2.2.4. Указание имен выходных файлов (-e)

Ключ `-e` позволяет указать имена выходных файлов редактора связей. Если этот ключ не используется, по умолчанию присваиваются следующие имена:

```
210X.EXE, 210X.SYM и 210X.MAP.
```

Пример использования ключа `-e`:

```
LD21  файл1  файл2  -e имя_файла
```

4.2.2.5. Компоновка С библиотеки рабочих программ ADSP-21XX (-lib)

Этот ключ должен использоваться, если компонуемая программа была сгенерирована С компилятором семейства ADSP-2100 и подключает некоторые функции С библиотеки рабочих программ ADSP-21XX. Эта библиотека является набором подпрограмм ANSI стандарта для цифровой обработки сигнала.

Ключ `-lib` принуждает редактор связей искать С библиотеку и компоновать любые вызванные функции. Ключ `-lib` используется только вместе с ключом `-c`.

4.2.2.6. Копирование библиотечных подпрограмм в загрузочные страницы (-p)

Ключ `-p` указывает редактору связей, что необходимо разместить копию библиотечной подпрограммы в каждой загрузочной странице, откуда она вызывается. Этот ключ должен быть применен даже в том случае, когда подпрограмма (или подпрограммы) вызывается на одной странице.

Обычно этот ключ применяется для указания часто используемых подпрограмм в файлах текущего каталога совместно с ключом `-dir`:

```
LD21  файл1 файл2  -p  -dir .
```

Теперь редактор связей найдет все вызовы этих подпрограмм в ваших загрузочных страницах и присвоит им соответствующий код. (**Примечание**: редактор связей не будет автоматически искать файлы в текущей директории. Если вы сохранили там ваши библиотечные файлы, необходимо использовать ключ `-dir` или переменную окружения `ADIL`, чтобы указать это). Ключ `-p` может применяться только для систем с памятью начальной загрузки (все процессоры ADSP-21XX, за исключением ADSP-2100).

4.2.2.7. С стек исполняемой программы в памяти программ (-pmstack)

Ключ `-c` указывает редактору связей реализовать стек исполняемой программы в памяти данных для программ сгенерированных С компилятором ADSP-21XX. Ключ `-pmstack` передвигает стек в память программ.

Если программа была откомпилирована с ключом `-pmstack`, необходимо компоновать ее также с ключом `-pmstack`. Ключ используется только вместе с ключом `-c`.

4.2.2.8. ПЗУ версия С библиотеки рабочих программ ADSP-21XX (-rom)

Функции библиотеки рабочих программ ADSP-21XX, могут быть размещены редактором связей в ПЗУ или ОЗУ (по умолчанию). Если программа была откомпилирована с ключом `-crom`, необходимо компоновать ее с ключом `-rom`. Ключ используется только вместе с ключом `-c`.

4.2.2.9. Создание динамической памяти С (-s)

Ключ -s указывает редактору связей на необходимость резервирования области памяти, выделяемой программе для динамически размещаемых структур данных, «кучи», для программ, сгенерированных С компилятором ADSP-21XX. Ключ `-s` может применяться только вместе с ключом `-c`, который вызывает создание стека исполняемой программы.

```
LD21  файл1 файл2  -c  -s размер_кучи
```

Аргумент *размер_кучи* должен быть целым числом, преобразуемым редактором связей в единицы слов памяти. Необходимо указывать ключ `-s`, для использования таких С функций, как `malloc`, `calloc` или `free`.

Обычно вершина стека размещена по самому старшему доступному адресу памяти данных (или памяти программ, если задан ключ `-pmstack`) и перемещается в направлении нижних адресов. Когда задан ключ `-s`, «куча» размещается в самых старших доступных адресах памяти. Стек сдвигается вниз, и начинается сразу после окончания блока «кучи».

Если используется ключа `-s`, редактор связей создает метку:

```
____top_of_stack      (метка с четырьмя символами подчеркивания)
```

адрес которой равен

```
____top_of_stack = ____top_of_ram - heap_size
```

4.2.2.10. Поиск библиотечного файла быстрого доступа (-user)

Ключ `-user` указывает библиотечный файл, который получен с помощью утилиты построения библиотек LIB21. При использовании этого ключа редактор связей будет искать только обозначенный файл для компоновки библиотечных подпрограмм. Смотрите раздел «Построение единой библиотеки для быстрого доступа».

4.3. Как работает редактор связей

Данные раздел дает представление о том, как работает редактор связей, когда он обрабатывает программные модули. При лучшем понимании принципов работы редактора связей, вы сможете структурировать свои программы для более эффективного использования памяти.

Работа редактора связей состоит в комбинировании ассемблированных модулей и инициализации данных в исполняемой программе, называемой файлом отображения памяти (`.EXE`). Две основные задачи: размещение памяти и разрешение символов.

4.3.1. Распределение памяти

Редактор связей читает каждый модуль кода и объявления переменных данных/буферов для определения типа памяти, в которой они должны быть расположены – ОЗУ или ПЗУ, память программ или память данных, имя сегмента и т.д.

Редактор связей читает также содержимое файла описания архитектуры `.ACH`, чтобы определить какие пространства памяти доступны и какие у них характеристики.

На основе этой информации происходит размещение каждого модуля, буфера и переменной в соответствующем типе памяти. Если объект имеет абсолютный адрес, указанный параметром `ABS`, его называют неперемещаемым. Если абсолютный адрес не присвоен, объект считается перемещаемым. Редактор связей присваивает адрес каждому перемещаемому объекту.

Если определитель сегмента (SEG) объединяется с параметром ABS, объявленный объект также считается неперемещаемым. Если определитель SEG используется без абсолютной адресации, тогда объявленный объект считается перемещаемым в пределах названного сегмента.

Редактор связей размещает объекты в памяти в следующей последовательности:

1. **неперемещаемые объекты** – буферы данных и модули с параметром ABS.

2. **перемещаемые внутри сегмента циклические буферы** – буферы данных с параметрами CIRC и SEG.

3. **перемещаемые внутри сегмента нециклические буферы и модули** – модули и буферы данных с параметрами SEG

4. **перемещаемые циклические буферы** – буферы данных с параметрами CIRC.

5. **перемещаемые модули и нециклические буферы** – все оставшиеся модули и нециклические буферы.

Для процессоров ADSP-21XX с памятью начальной загрузки на кристалле, редактор связей разместит как можно больше загрузочных кода и данных во внутренней памяти, перед тем, как использовать внешнюю.

Циклические буферы могут быть размещены в некоторых пределах памяти, принимая во внимание характеристики аппаратной адресации циклических буферов процессоров ADSP-21XX. Обычно циклический буфер должен начинаться с базового адреса, который кратен 2^n, где n - число разрядов, требуемых для представления длины буфера в двоичном выражении.

Редактор связей размещает циклические буферы в памяти, в соответствии с этим ограничением. Если длина циклического буфера равна, например, 13 словам, редактор связей размещает его с базового адреса, который кратен 16.

4.3.1.1. Размещение памяти начальной загрузки

Система может иметь до 8 страниц памяти начальной загрузки. Одна страница может хранить до 2048, 24 разрядных слов памяти программ для ADSP-2101, ADSP-2111 и ADSP-21msp50. Страницы начальной загрузки ADSP-2105 и ADSP-2115 размещают до 1024 слов.

Следует помнить важную вещь, касающуюся разницы между тем, что происходит при компоновке и тем, что происходит во время выполнения программы. Любой модуль, объявленный с параметром BOOT, редактор связей размещает в пространстве памяти начальной загрузки. Это размещение, тем не менее, касается только хранения программ перед исполнением (когда страница загружается и выполняется).

Редактор связей должен также разместить адресное пространство для данных и программ загружаемого модуля в памяти программ или данных, где они располагаются во время выполнения. Таким образом, редактор связей назначает области в памяти начальной загрузки и памяти программа/данных для всех загружаемых модулей, обеспечивая логический интерфейс между загрузочной памятью и памятью во время выполнения.

Если вы захотите, чтобы в памяти процессора (внешней или внутренней) во время выполнения отдельной загрузочной страницы существовала незагружаемая подпрограмма или буфер данных, вы должны использовать параметр BOOT, чтобы связать его с этой страницей. При совместном использовании с параметром STATIC, редактор связей зарезервирует пространство для объекта во время выполнения страницы.

Выходной файл карты распределения памяти программы (.MAP) показывает как расположение вашей программы в памяти начальной загрузки, так и соответствующее отображение кода в памяти во время выполнения (после выполнения начальной загрузки).

Этот файл помогает понять перенос из памяти начальной загрузки в рабочую память. Вы не можете указать область, где модуль будет размещен в памяти начальной загрузки - редактор связей самостоятельно реализует эту функцию, составляя эффективно упакованные загрузочные страницы.

4.3.2. Разрешение символов

Чтобы разрешить программные символы, редактор связей должен поставить в соответствие каждому символу определенный адрес в пространстве памяти. Имена программных меток, переменных/буферов являются символами, которые определены в исходном коде. Ассемблер просто пропускает их редактору связей, задача которого определить адрес каждого из символов, после размещения в памяти всех модулей.

Обращение к символу может происходить только внутри модуля, где он определен, пока он не определен директивами ENTRY или GLOBAL. Эти директивы ассемблера расширяют диапазон обращения к символу. Другие модули должны объявить этот символ директивой EXTERNAL перед обращением к нему.

Для каждой ссылки EXTERNAL, редактор связей ищет в других модулях определения ENTRY или GLOBAL. При нахождении нескольких совпадений выводится сообщение об ошибке. Если поиск не выявил определений символа, редактор связей включает поиск библиотечного файла в соответствии с последовательностью, описанной в разделе «Последовательность поиска библиотеки». Этот поиск включает просмотр переменной среды окружения ADIL и аргументов ключей -user, -dir.

Если подключаемый символ не найден с помощью поиска в библиотеке, редактор связей выводит сообщение об ошибке.

После того как распределение памяти завершено, и все внешние ссылки разрешены, редактор связей присваивает каждому символу значение адреса.

Редактор связей вырабатывает файл таблицы символов .SYM (если задан ключ -g), который содержит список всех встреченных программных символов и их адресов. Этот файл показывает какие символы могут быть доступны каждому модулю.

4.4. Использование библиотечных файлов ваших подпрограмм

Редактор связей ADSP-21XX позволяет вам использовать библиотечные файлы ваших часто используемых программ и подпрограмм. Библиотечные подпрограммы делаются доступными для любых компонуемых программ. Простой ссылкой на функцию вы заставляете редактор связей искать и компоновать соответствующий библиотечный файл.

Чтобы приготовить свою библиотеку, вы должны сделать следующее:

Что делаете вы:

1. Пишете библиотечные подпрограммы, располагая в начале каждой из них метку. Объявляете эти метки как глобальные через директиву `ENTRY`.

2. Объявляете начальную метку библиотечной подпрограммы как внешнюю в начале каждого модуля, использующего эту функцию с помощью директивы ассемблера `EXTERNAL`.

3. Ассемблируйте библиотечные подпрограммы в одном или более модулях (один модуль на файл).

4. Указываете редактору связей, где можно найти ваши библиотечные файлы (задавая путь у каталогу в переменной среду окружения `ADIL` или ключом `-dir`).

Теперь, когда вы компонуете программу, которая использует ваши библиотечные подпрограммы, редактор связей делает следующее:

Что делает редактор связей:

1. Располагает программу в памяти, собирает все символы в программе и пытается определить адрес обращения к каждой метке (известное как «разрешение символов», описанное ранее).

2. Поскольку метки библиотечных подпрограмм являются неразрешенными (ненайденными) символами, редактор связей автоматически открывает и ищет их в каждом объектном файле (`.OBJ`), находящимся в каталогах, обозначенных в переменной `ADIL` или ключе `-dir`. Проверяются все метки, определенные через директиву `ENTRY`.

3. Любой файл, содержащий метку, на которую ссылается программа, включается процесс компоновки.

Для систем ADSP-21XX с несколькими страницами начальной загрузки, если применяется какая-либо библиотечная подпрограмма, должен быть указан ключ `-p`. Этот ключ вынуждает редактор связей разместить копии подпрограмм на всех страницах начальной загрузки, где это требуется.

4.4.1. Построение единой библиотеки для быстрого доступа

Программное обеспечение разработчика ADSP-21XX включает построитель библиотеки – утилиту **LIB21**, которая позволяет вам записать несколько библиотечных подпрограмм и упаковать их в один файл для быстрого доступа. После создания библиотечного файла необходимо вызвать редактор связей с ключом `-user` *`имя_библиотеки`*, который позволяет находить, выделять и компоновать необходимые подпрограммы из файла *`имя_библиотеки`*.

Использование утилиты LIB21 и ключа `-user` подразумевает то же самое, что и применение ключа `-dir` или переменной среды окружения `ADIL`.

Преимущество утилиты LIB21 состоит в том, что редактор связей работает быстрее. Утилита LIB21 вызывается одним из двух способов:

```
LIB21  имя_библиотеки  файл1 [файл2...]  [-V версия]
```

или

```
LIB21  имя_библиотеки  -i списочный_файл  [ -V версия]
```

Выходной файл *`имя_библиотеки.A`* объединяет в себе отдельно ассемблированные модули, перечисленные в командной строке (*`файл1`*, *`файл2`* и т. д.). Эти модули должны быть указаны без расширений. Ключ `-i` имеет такое же действие, как и при вызове редактора связей. Файл *`списочный_файл`* содержит список с одним именем файла на строке. Ключ `-V` позволяет вставить номер версии для подпрограмм в библиотечном модуле; аргумент *`версия`* является символьной строкой. Строка версии не влияет на исполнение программы.

Ниже приведен пример создания библиотечного файла быстрого доступа:

```
LIB21  filler  taps  coeffs  start_input  -v V1.0
```

Утилита LIB21 создаст файл `FILTER.A`, объединяющий три входных модуля. Символьная строка версии «`V1.0`» вставляется в файл. Редактор связей можно вызывать строкой:

```
LD21  main  sum  graph  -user filter
```

Что вызовет компоновку модулей `main`, `sum` и `graph`. Полагается, что из файла `FILTER.A` будет использована одна или более библиотечных подпрограмм, и редактор связей выделит требуемые функции и включит их в процесс компоновки.

4.4.2. Последовательность поиска библиотеки

Существует несколько путей использования библиотечных подпрограмм в сочетании с редактором связей: ключ `-dir`, переменная среды окружения `ADIL`, утилита LIB21 и ключ `-user`, ключ `-lib`. При использовании различных комбинации, редактор связей разбирает их в следующей последовательности:

1. Если задан ключ `–user`, ищется и открывается файл библиотеки быстрого доступа. Последовательность поиска каталога с файлом:

    a) текущая директория, затем
    b) директории, перечисленные в переменной `ADIL` (если необходимо)

2. Если задан ключ `-lib`, ищется С библиотека рабочих программ ADSP-21XX. Просматриваются каталоги, перечисленные в переменной `ADIL`.

3. Для неразрешенных ссылок просматриваются директории, перечисленные в переменной `ADIL`.

4. Если задан ключ `-dir`, просматриваются директории, перечисленные в аргументах.

4.5. Системы с несколькими страницами начальной загрузки

Реализация систем с несколькими страницами начальной загрузки может быть простой или сложной, зависящей от того, сколько программ и данных требуется разместить. Если каждая страница полностью независима и содержит всю необходимую информацию, распределение памяти редактором связей проходит относительно просто.

Однако множество систем должны распределять программы или структуры данных между загрузочными страницами. Стандартная операция компоновки не позволяет сделать это, поскольку редактор связей стирает ранее существующую карту разделения памяти для каждой страницы. Для предотвращения этого, вы должны использовать один из следующих технических приемов:

- повторение параметра `BOOT` в модуле (содержащем программы и/или структуры данных), с указанием каждой страницы, которую он требует.

- задание ключа `-p` редактора связей, чтобы копировать библиотеку рабочих подпрограмм на каждую страницу начальной загрузки.

- использование определителя `STATIC`, чтобы оставить модуль или буфер/переменную в памяти, во время загрузки новых страниц.

Для достижения различных результатов эти технические приемы могут быть использованы отдельно или в некоторой комбинации. Первые два приема имеют одинаковый конечный результат, в то время как определитель `STATIC` обладает другим механизмом действия. Применение этих определителей предотвращает перезапись модуля (или буфера) при загрузке страницы, как страницы 0 при перезапуске (если MMAP=0), так и страниц 1-7 под управлением программного обеспечения. Это относится к модулям/буферам как для внутренней, так и для внешней памяти процессора.

Когда редактор связей определяет память для размещения программы, он рассматривает девять независимых частей памяти: незагружаемые память программ и данных, и загружаемые страницы 0-7. Незагружаемую память определяют как начальное состояние памяти программ и данных, перед загрузкой любой страницы и выполнением кода.

До применения определителя `STATIC` эти девять частей рассматривают совершенно независимо друг от друга. В отсутствии любых объявлений `STATIC`, редактор связей полагает, что каждая из девяти частей начинается с чистого листа, и каждая загружаемая страница имеет собственное распределение памяти, доступное при загрузке.

4.5.1. Перезагрузка под управлением программы

Загрузка новой страницы во время выполнения программы описана в главе «Интерфейс памяти» руководства пользователя семейства ADSP-2100. Выполнение перехода от одной страницы к другой происходит перезагрузкой под управлением программного обеспечения. Перезагрузка управляется регистром `System Control Register`, отображенным во внутренней памяти данных по адресу 0×3FFF. Этот регистр содержит разряд `BFORCE` (управляющий загрузкой) и поле `BPAGE` (выбора загружаемой страницы).

Программы, реализованные в нескольких страницах начальной загрузки, выполняются следующим образом:

1. Страница 0 загружается и выполняется после перезапуска системы.

2. Когда должна быть загружена новая страница, устанавливают разряды поля `BPAGE`.

3. Для перехода на новую страницу устанавливают бит `BFORCE`. Внутренняя память программ загружается из новой страницы; внутренняя память данных не меняется.

4.5.2. Пример совместного использования статического буфера

Этот раздел приводит пример совместного использования буфера данных на нескольких загрузочных страницах. Этот же метод может быть использован для разделения подпрограмм, разрешая вызывать их кодом, расположенным на разных страницах. Этот пример использует страницы начальной загрузки размером 2К, которые могут быть реализованы на процессорах ADSP-2101, ADSP-2111 и ADSP-21msp50. В этом примере для защиты от перезаписи буфера используется определитель STATIC. Страницы 0, 1 и 2 совместно используют этот буфер; код загруженный из этих страниц получает доступ к данным. Буфер объявлен в исходном коде на загрузочной странице 0:

```
.VAR/PM/RAM/STATIC  dat1[64];
.GLOBAL  dat1;
.INIT  dat1: <frames.dat>;
```

Буфер с именем `dat1` и длиной в 64 слова определен как глобальный, чтобы сделать его видимым снаружи модуля. Другие модули должны определить `dat1`, как внешний, чтобы иметь к нему доступ.

Редактор связей считывает значения 64 слов данных из файла `frames.dat` для инициализации ими буфера `dat1`. Данные включены в порции загружаемой памяти `EXE` файла. Этот файл переводится разделителем программ для записи в ППЗУ в файл формата программатора ППЗУ.

Рис.4.2. показывает память нулевой, первой и второй страницы начальной загрузки. Заметьте, что редактор связей размещает статический объект `dat1` на вершине страницы 0 и резервирует 64 верхних адреса на страницах 1 и 2. Если на этих страницах необходимо будет расположить больше кода, и адресное пространство для буфера `dat1` будет перекрыто, редактор связей выведет сообщение об ошибке.

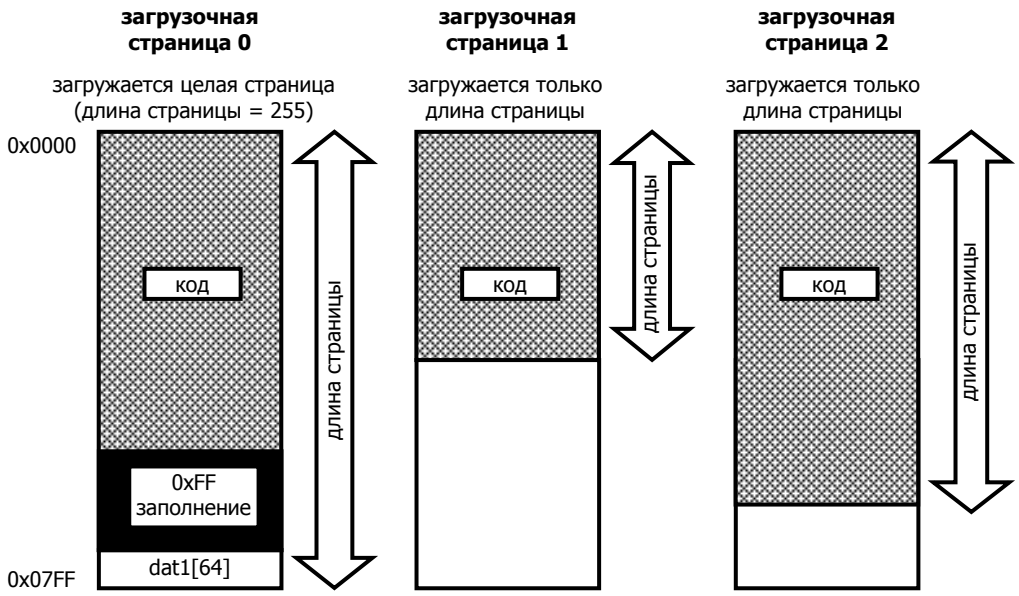


Рис.4.2. Статический буфер на страницах начальной загрузки.

На рисунке видно, что размещение защищает буфер от перезаписи, поскольку страницы 1 и 2 загружаются только длиной страниц. Схема загрузки адресов процессоров ADSP-21XX загружает внутреннюю память программ, начиная с самого старшего используемого адреса (как определено длиной страницы). Поскольку код этих страниц содержит меньше, чем (2048-64) слов, при загрузке он не будет переписывать `dat1`. Начальные данные для `dat1` содержатся в копии страницы 0, поэтому длина загружаемой страницы 255 слов (страницы загружена полностью). Редактор связей размещает заполняющие байты со значением 0×FF в области нулевой страницы между окончанием кода и буфером `dat1`.

(**Примечание**: длина страницы определяется как:
pagelength = (число 24-разрядных PM слов / 8) – 1

Еще одно применение определителя `STATIC`, это создание в памяти «разделенной» (совместно используемой) подпрограммы или буфера с абсолютным адресом. Адрес должен быть выбран таким, чтобы он был больше, чем длина самой большой загружаемой страницы. В этом случае вы делаете работу за редактора связей - размещаете объект в памяти, гарантируя, что он не перекроется в течение загрузки новых страниц. Однако бывает легче, при разработке сложных систем, позволить редактору связей сделать эту работу самостоятельно.

4.5.3. Пример использования статических и динамических сегментов

Этот раздел описывает пример системы с многостраничной организацией памяти начальной загрузки. Будет использоваться процессор ADSP-2101, для которого создается файл описания архитектуры, определяющий статические и динамические сегменты памяти. Статический сегмент будет содержать код, используемый всеми загружаемыми страницами, в то время как динамический сегмент будет содержать код, определенный для каждой страницы.

Система будет использовать страницы 0, 1 и 2. Страница 0 загружается при перезапуске системы. При последующей загрузке страниц 1 и 2, каждая из них будет частично перезаписывать внутреннюю память программ. Перезаписываемая часть будет «динамическим» сегментом, а защищенная часть будет «статическим» сегментом. Ниже приведен файл описания архитектуры для системного конфигуратора:

```
.SYSTEM overlay_test;
.ADSP2101;
.MMAPO;

.SEG/PM/RAM/CODE/ABS=0           dynamic[0x400];{нижние 1K внутренней PM}
.SEG/PM/RAM/CODE/DATA/ABS=0x400 fixed[0x400];  {верхние 1K внутренней PM}
.SEG/DM/RAM/DATA/ABS=0x3800      int_dm[0x400]; {внутренняя DM}

.SEG/BOOT=0/ROM  boot 0[0x800];  {загрузочная страница 0}
.SEG/BOOT=1/ROM  boot 1[0x800];  {загрузочная страница 1}
.SEG/BOOT=2/ROM  boot 2[0x800];  {загрузочная страница 2}

.ENDSYS;
```

Сегменты `dynamic` и `fixed` расположены во внутренней памяти программ. Общий код для всех загружаемых страниц помещен в сегмент `fixed`; все коды, не являющиеся общими размещены в сегменте `dynamic`. Размер сегмента выбран здесь для примера – для реального приложения размер сегмента должен определяться размером кода. Следующие операторы объявляют три постранично определенных модуля:

```
.MODULE/RAM/SEG=dynamic/BOOT=0  mod01;
.MODULE/RAM/SEG=dynamic/BOOT=0  mod02;
.MODULE/RAM/SEG=dynamic/BOOT=0  mod03;

.MODULE/RAM/SEG=dynamic/BOOT=1  mod11;
.MODULE/RAM/SEG=dynamic/BOOT=1  mod12;
.MODULE/RAM/SEG=dynamic/BOOT=1  mod13;

.MODULE/RAM/SEG=dynamic/BOOT=2  mod21;
.MODULE/RAM/SEG=dynamic/BOOT=2  mod22;
.MODULE/RAM/SEG=dynamic/BOOT=2  mod23;
```

Следующий оператор объявляет единственный модуль, который будет использоваться на всех трех загружаемых страницах:

```
.MODULE/RAM/STATIC/SEG=fixed/BOOT=0  common_mod;
```

Наконец, предположим, что следующее объявление буфера содержится в некотором другом модуле, загружаемом из страницы 0:

```
.VAR/DM/RAM/STATIC/SEG=int_dm  common_buf[100];
```

Буфер данных хранит 100 слов, которые должны быть доступны из всех трех загружаемых страниц. Следовательно, необходимо использовать объявление `STATIC`. Модуль, который объявляет и инициализирует `common_buf`, должен включать программу копирования буфера после загрузки из внутренней памяти программ во внутреннюю память данных, так как редактор связей только резервирует место, но не обеспечивает механизм записи данных. Рис.4.3. показывает копии загрузочных страниц 0, 1 и 2 после компоновки программы.

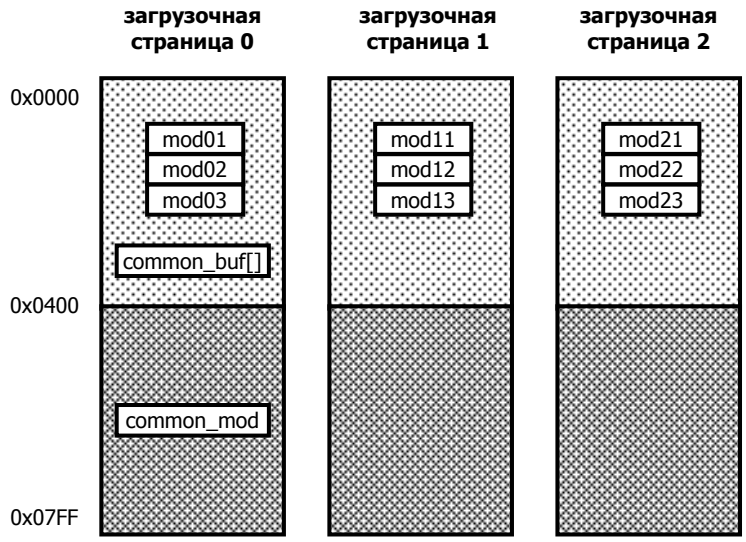


Рис.4.3. Статические и динамические объекты в памяти начальной загрузки.

Рис.4.4 показывает состояние внутренней памяти ADSP-2101 после загрузки страницы 0 и копирования буфера `common_buf` в память данных. Цикл начальной загрузки процессоров ADSP-21xx загружает внутреннюю память программ, начиная самого старшего используемого адреса, декрементируя его до нуля. Младшие адреса загружаются с каждой перезагрузкой. Поскольку страницы 1 и 2 только загружают код в сегмент `dynamic`, сегмент `fixed` не перезаписывается. Это позволяет модулю `common_mod` постоянно оставаться в памяти.

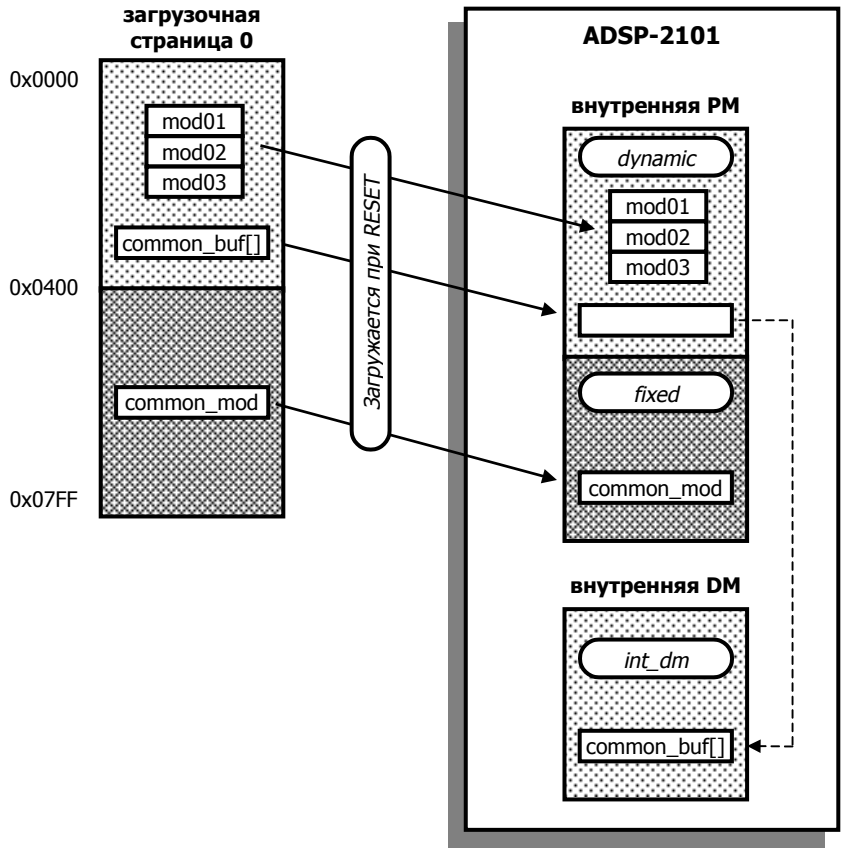


Рис.4.4. Состояние памяти страницы 0 во время выполнения программы.

4.6. Структура файла карты распределения памяти

Файл карты распределения памяти (.MAP) помогает вам интерпретировать результаты компоновки. Редактор связей генерирует этот файл, если был указан ключа -x. Данный файл обеспечивает следующую информацию:

- Символы

Список перекрестных ссылок всех программных символов, отсортированный по модулям. Для каждого символа предоставляется следующая информация:

Тип символа	название модуля, буфера/переменной или программной метки
Адрес	базовый адрес для модулей и буферов
Длина	для модулей и буферов
Область памяти	PM, DM или BM

- Сегменты памяти

Структура сегментов памяти, объявленных системным конфигуратором и описанная в файле архитектуры .ACH. Для каждого сегмента показаны базовый адрес, длина, атрибуты памяти.

- Память начальной загрузки память программ во время выполнения

Карта адресов модулей и структур данных на каждой странице начальной загрузки и соответствующая карта кода, загруженного во внутреннюю память программ. Информация о загрузочных байтовых адресах ППЗУ и требуемом размере ППЗУ.

- Статическая и динамическая память

Структура статической памяти программ, динамической памяти данных и статической памяти данных. Включена информация об адресе, длине и атрибутах памяти.

- Сообщения об ошибках

- Библиотеки

Список найденных библиотечных файлов, включенных в процесс компоновки.

Пример структуры файла карты распределения памяти показан на Рис.4.5.

```
ADSP-21XX Linker Version 3.0              Analog Devices, Inc.
final (final.exe) mapped according to FIR_SYSTEM (sysb2101.ach)

xref for module: MAIN_ROUTINE       boot memory page(s) 0,
   MAIN_ROUTINE               pm 0:0000 [003B]        module(global)
   DATA_BUFFER                dm 0:3800 [000F]        variable(global)
   COEFFICIENT                pm 0:0040 [000F]        variable(global)
   RESTARTER                  pm 0:001C               label
   CLEAR_BUFFER               pm 0:0024               label
   WAIT                       pm 0:0039               label
   FIR_START                     0:004F [0000]        extern(FIR_ROUTINE)

xref for module: FIR_ROUTINE        boot memory page(s) 0,
   FIR_ROUTINE                pm 0:004F [000A]        module(global)
   FIR_START                  pm 0:004F               label
   CONVOLUTION                pm 0:0054               label
   COEFFICIENT                   0:0040 [000F]        extern(MAIN_ROUTINE)
   DATA_BUFFER                   0:3800 [000F]        extern(MAIN_ROUTINE)

21xx memory per FIR_SYSTEM (sysb2101.ach):
   internal 2101 pm ram mapped to 0000 - 0800 (auto booted at reset)
   internal 2101 dm ram mapped to 3800 - 3BFF
   0000 - 07FF [  2048.] external bm rom code BOOT_MEM
   0000 - 07FF [  2048.] internal pm ram data/code INT_PM
   0800 - 3FFF [ 14336.] external pm ram data/code EXT_PM
   3800 - 3BFF [  1024.] internal dm ram data INT_DM
   0000 - 37FF [ 14336.] external dm ram data EXT_DM

boot memory and bootable runtime program memory map:
   boot page 0 (auto boot)

   bm:0000-003A (x8rom:0000-00EB) pm:0000-003A [59.]
   ram module           MAIN_ROUTINE of MAIN_ROUTINE

   bm:0040-004E (x8rom:0100-013B) pm:0040-004E [15.]
   ram circ variable    COEFFICIENT of MAIN_ROUTINE

   bm:004F-0058 (x8rom:013C-0163) pm:004F-0058 [10.]
   ram module           FIR_ROUTINE of FIR_ROUTINE

   8k of boot memory rom space required for this bootable runtime map.
   Most convenient boot memory rom size is 8k bytes (64k bits).

fixed program memory map:
   fixed program memory rom: 0.
   fixed program memory ram: 0.

dynamic data memory map:
   boot page 0
   3800 - 380E [15.]    ram circ variable DATA_BUFFER of MAIN_ROUTINE

fixed data memory map:
   fixed data memory rom: 0.
   fixed data memory ram: 0.

21xxlnk: final, 21xx memory use:
   program memory rom: 0.; program memory ram: 0.;
   data memory rom: 0.; data memory ram: 0.
```

Рис.4.5. Структура файла карты распределения памяти.

Разделитель программ для записи в ППЗУ 5

5.1. Введение

Программа разделителя программ выделяет ПЗУ части из выходного `.EXE` файла редактора связей и формирует информацию для использования программаторами ППЗУ.

Программа разделителя программ может генерировать файлы для памяти программ, памяти данных или памяти начальной загрузки. Три файла создаются для памяти программ, чтобы организовать ППЗУ в трехбайтовых словах, соответствующих 24-разрядным инструкциям ADSP-21xx. Два файла создаются для памяти данных, чтобы сгруппировать данные в двухбайтовые слова, и один файл байтового формата создается для памяти начальной загрузки.

Файлы байтовых потоков могут быть созданы для памяти программ и памяти данных, чаще вертикальной организации слов, чем горизонтальной. Программа разделения программ для программаторов ППЗУ может генерировать файлы в форматах **Motorola S** или **Intel Hex**. Кроме того, поддерживается формат **Motorola S2** для вывода байтовых потоков.

5.2. Запуск программы разделения

Для запуска программы используется следующая команда:

```
SPL21  файл_образа  ППЗУфайл  [-ключ ...]
```

При каждом запуске генерируются выходные файлы для одного типа памяти ADSP-21xx – памяти программ, памяти данных или памяти начальной загрузки. Если требуется создать файлы для более чем одного типа памяти, необходимо запустить программу разделителя программ для программаторов ППЗУ несколько раз.

Входной файл *файл_образа* является `.EXE` файлом вашей программы, сгенерированным редактором связей. Этот файл должен иметь расширение `.EXE`, добавленное редактором связей; в противном случае программа разделителя не распознает его.

Каждый раз, когда запускается программа разделителя, необходимо называть имя выходного файла *ППЗУфайл*. При создании файлов для более чем одного типа памяти (PM, DM, BM), необходимо каждый раз выбирать различные имена файлов, чтобы избежать наложения на результаты предшествующих запусков.

При запуске программы разделителя в командной строке могут использоваться пять ключей (один обязательный, остальные дополнительные).

Ключ	*Возможные формы*	
Тип памяти	**-pm, -dm, -bm**	(обязательный)
Формат файла ППЗУ	**-s,-i,-us,-us2,-ui**	(по умолчанию -s)
Размер страниц	**-bs pagesize**	(по умолчанию 2K)
Границы страниц	**-bb boundary**	(по умолчанию 2K)
Программа начальной загрузки	**-loader**	

Ключи, определяющие пространство памяти и формат файла ППЗУ:

-pm	память программ	
-dm	память данных	
-bm	память начальной загрузки	
-s	формат Motorola S	(по умолчанию)
-i	формат Intel Hex	
-us	формат Motorola S, байтовый поток	(только с `-pm` или `-dm`)
-us2	формат Motorola S2, байтовый поток	(только с `-pm` или `-dm`)
-ui	формат Intel Hex, байтовый поток	(только с `-pm` или `-dm`)

Ключи `-bs`, `-bb` и `-loader` описаны в следующих разделах этой главы.

5.2.1. Пример создания файлов для памяти PM и DM

Чтобы создать файлы для программаторов ППЗУ для памяти программ и памяти данных, нужно запустить программу разделителя дважды:

```
spl21  fir_sys  pmburn  -pm
spl21  fir_sys  dmburn  -dm
```

В приведенном примере имеется файл, созданный редактором связей – FIR_SYS.EXE, который содержит программы и данные для записи в ППЗУ. Заметьте, что каждому выходному файлу программы разделителя дано уникальное имя. Выходные файлы созданы в формате Motorola S, так как отсутствует ключ указания формата ППЗУ.

5.2.2. Пример создания файла только для памяти BM

Многие системы ADSP-21xx используют для хранения программ только память начальной загрузки. Для этих систем программу разделителя запускают один раз, чтобы создать файл для памяти начальной загрузки ППЗУ, например:

```
spl21  iir-sys  bootburn -bm -i
```

Здесь используется ключ `-i`, чтобы сгенерировать файл в формате Intel Hex.

5.3. Выходные файлы программы разделителя

Когда установлен ключ `-pm`, программа разделителя для программаторов ППЗУ создает три файла байтовой ширины. Один файл содержит старшие байты 24-разрядных слов и имеет расширение `.BNU`. Второй файл содержит средние байты и имеет расширение `.BNM`, и третий файл содержит младшие байты и имеет расширение `.BNL`.

Когда установлен ключ `-dm`, программа разделителя для программаторов ППЗУ создает два файла байтовой ширины. Один файл содержит старшие байты 16-разрядных слов и имеет расширение `.BNM`. Второй файл содержит младшие байты и имеет расширение `.BNL`. Файл с расширением `.BNU` не создается.

Когда установлен ключ `-bm`, программа разделителя для программаторов ППЗУ создает единственный файл с расширением `.BNM`, который включает программы и данные для всех страниц начальной загрузки. Файлы с расширением `.BNU` и `.BNL` не создаются.

Рис.5.1 показывает выходные файлы разделителя программ для записи в ППЗУ.

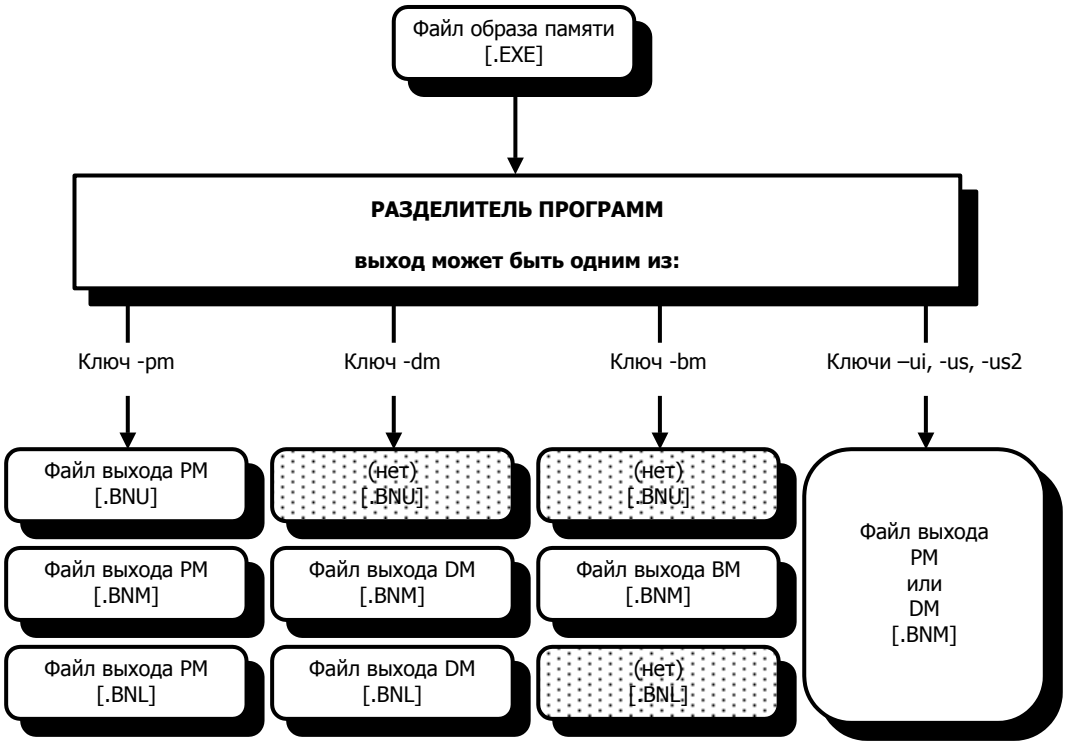


Рис.5.1. Вход/выход разделителя программ для программаторов ППЗУ.

5.3.1. Выходной байтовый поток для PM и DM

Ключами `-us`, `-us2` или `-ui` выбирается выходной байтовый поток и генерируется единственный файл (`.BNM`) с вертикальной организацией слов в памяти. Этот формат может быть выбран только для памяти программ или памяти данных.

Выходной файл байтового потока построен от самого старшего байта каждого слова к самому(ым) младшему(им) байту(ам), другими словами, старший байт каждого слова находится по младшему адресу. 24-разрядные слова памяти программ требуют последовательности из трех байт, в то время как 16-разрядные слова памяти данных последовательности из двух.

Если были назначены любые абсолютные адреса объектам памяти, такая информация в формате байтового потока будет утеряна.

Нельзя также создавать файлы байтового потока из входной программы, которая содержит несмежные блоки памяти (неперемещаемые сегменты с неиспользуемыми блоками памяти между ними). Программные модули должны быть или перемещаемыми, или их необходимо поместить в смежные блоки памяти.

5.3.2. Организация памяти начальной загрузки

Загружаемая память обладает байтовой шириной и организована в вертикальные группы из четырехбайтовых слов. Старший байт каждого слова размещают по младшему адресу четырехбайтовой группы.

Каждое 32-разрядное слово состоит из 24-разрядной инструкции, дополненной байтом. Добавленный байт (0×FF) игнорируется, за исключением первого байта каждой загружаемой страницы. Дополнительный байт первого слова на странице содержит информацию о длине страницы начальной загрузки.

Длины страниц подсчитываются программой разделителя и вставляются в файл образа ППЗУ: для страницы 0 эта величина размещена в ППЗУ по байтовому адресу 0×0003.

Разделитель программ для программаторов ППЗУ подсчитывает длину каждой страницы начальной загрузки следующим образом:

$$Длина\ страницы = \frac{Количество\ 24-разрядных\ инструкций}{8} - 1$$

(количество инструкций должно быть округлено до кратного восьми).

Например, нулевая длина страницы показывает восемь слов, представленных 32 последовательными байтами. Максимальная длина страницы = 255 показывает 2048 слов.

Каждая страница начальной загрузки должна содержать количество слов, которое кратно восьми, если необходимо, программа разделителя добавляет дополнительные слова (0×FFFFFFFF) в конце страницы.

5.4. Страницы начальной загрузки меньше 2К

Ключи `-bs` и `–bb` разрешают создание загружаемых страниц размером меньше 2К. Системы ADSP-21xx с памятью начальной загрузки обычно используют страницы размером 2К слов. Однако такие системы, как ADSP-2105 и ADSP-2115 требуют, чтобы размер страниц начальной загрузки был 1К. Использование страниц меньшего размера позволит использовать преимущества многостраничных систем с небольшими программами, так как для них необходимо меньшие по объему EPROM.

Когда программа разделителя запущена без ключа `-bs`, размер загружаемой страницы по умолчанию формируется равным 2К, что позволяет хранить 2048 слов.

Применяя ключ `-bs` можно создать загрузочные страницы меньшего размера; аргумент *`pagesize`* может быть величины от 0 до 2048 и должен быть введен в десятичном виде.

```
0 ≤ pagesize ≥ 2048                    (по умолчанию = 2048)
```

Ключ `-bb` используют, чтобы сделать небольшие страницы непрерывными в памяти. Аргумент *`boundary`* определяет стартовый адрес последующих страниц и может принимать следующие значения:

```
boundary = 2048,1024, 512 или 256      (по умолчанию = 2048)
```

Например, если размер страницы =1024 и значение ограничения = 2048, то страницы размером 1К будут начинаться с адресов, кратных 2048: 0, 2048, 4096, 6144, 8192. Это приводит к тому, что остаются неиспользованными 1024 адреса пространства ППЗУ.

Заметьте, что программы моделирования ADSP-2101, ADSP-2111 и ADSP-21msp50 в состоянии моделировать только загрузку страниц размером 2К. Если вы создадите загрузочные страницы меньшего размера, вы не сможете использовать команду симулятора LR, чтобы загрузить файл образа ППЗУ. Тем не менее, команда L может быть применена, чтобы загрузить и моделировать исполняемый `.EXE` файл.

5.4.1. Страницы начальной загрузки размером 1К для ADSP-2105/2115

Для создания страниц начальной загрузки размером 1К, непрерывных в памяти, для систем ADSP-2105 или ADSP-2115, используйте ключи `-bs` и `-bb` следующим образом:

```
spl21  файлEXE  файлППЗУ  -bm  -bs 1024  –bb 1024
```

5.4.2. Использование адресной линии для памяти начальной загрузки

Если вы используете ключи `-bs` и `-bb` для создания страниц начальной загрузки размером меньше 2К в системах ADSP-2101, ADSP-2111 или ADSP-21msp50, или для создания загружаемых страниц меньших 1К для систем ADSP--2105 или ADSP-2115, необходимо изменить подключение шины адреса к памяти начальной загрузки:

Размер страницы	*Отсоединить*
1024	A12*
512	A12, A11
256	A12, A11, A10

* Нет необходимости отключать A12 в системах ADSP-2105 или ADSP-2115.

Эти изменения позволяют процессору адресовать загружаемые страницы меньшие 2К без незаполненных блоков памяти. Обычно (со страницами 2К) ADSP-2101 используют следующие соединения с адресами памяти начальной загрузки:

- D23, D22, A13 выбирают страницы начальной загрузки (0-7);
- A12 – A0 выбирают байтовый адрес (0-8191) на каждой странице.

Отключая A12 и A11, ADSP-2101 может загрузить страницы размером 1/2К. Связи D23, D22, A13 и A10–A0 адресуют тогда полное пространство памяти начальной загрузки в 4К слов.

Чтобы адресовать загружаемые страницы размером 1К ADSP-2105 и ADSP-2115 используют следующие соединения:

- D23, D22, A13 выбирают страницы начальной загрузки (0-7);
- A11–A0 выбирают байтовый адрес (0-4095) на каждой странице.

5.5. Использование ключа –loader.

Опция загрузчика (ключ `-loader`) разделителя программ для записи программаторами ППЗУ создает загружаемую версию программы, которая автоматически инициализирует все необходимое пространство ОЗУ ADSP-21xx во внутренней памяти программ. Чтобы осуществлять инициализацию загрузки каждой следующей страницы, добавляются подпрограммы загрузки памяти.

Стандартная операция загрузки ADSP-2101, ADSP-2111 и ADSP-21msp50 работает при перезапуске только с 2К внутренней памятью программ. Ни внутренняя память данных, ни внешняя память программ, или внешняя память данных не загружаются.

Если ваша программа начальной загрузки требует инициализацию любой из перечисленных выше областей памяти (в ОЗУ), необходимо написать подпрограммы для копирования кодов/ данных из внутренней памяти программ, или использовать ключ `-loader` (можно также использовать запрограммированные ПЗУ устройства для внешней памяти).

Смотрите раздел «Инициализация вашей программы в памяти» в главе 3.

Ключ `-loader` позволяет также создавать многостраничные программы, которые полностью загружаются при перезапуске – коды и данные загружаются во внутреннюю память программ и копируются из нее в другие области, для каждой страницы одновременно.

Чтобы использовать опцию загрузчика для систем ADSP-2101, ADSP-2111 или ADSP-21msp50 с размером страниц начальной загрузки 2К, вызовите программу разделителя только с ключом `-loader` и ключом формата ППЗУ файла:

```
spl21  файлEXE  файлППЗУ  -loader  [-s]  [-i]
```

Чтобы использовать опцию загрузчика для систем ADSP-2105 или ADSP-2115 со страницами начальной загрузки размером 1К, вызовите разделитель программ для записи в ППЗУ с ключом `-loader`, ключом формата ППЗУ файла и ключами `-bs` и `-bb` с аргументами *`pagesize`* и *`boundary`*, устанавливающими значение 1024:

```
spl21  файлEXE  файлППЗУ  -bs 1024  -bb 1024  -loader  [-s]  [-i]
```

При этом нельзя использовать ключи типа памяти `-bm`, `-pm` или `-dm`. Автоматически генерируется файл с расширением `.BNM`. Выходной файл должен использоваться для программирования загрузочной памяти EPROM. В этом файле включена ваша полная исполняемая программа, включая все коды и данные, которые должны быть скопированы в память программ и память данных ОЗУ.

Для страниц размером 2К процесс загрузки и копирования изображен на Рис.5.2.

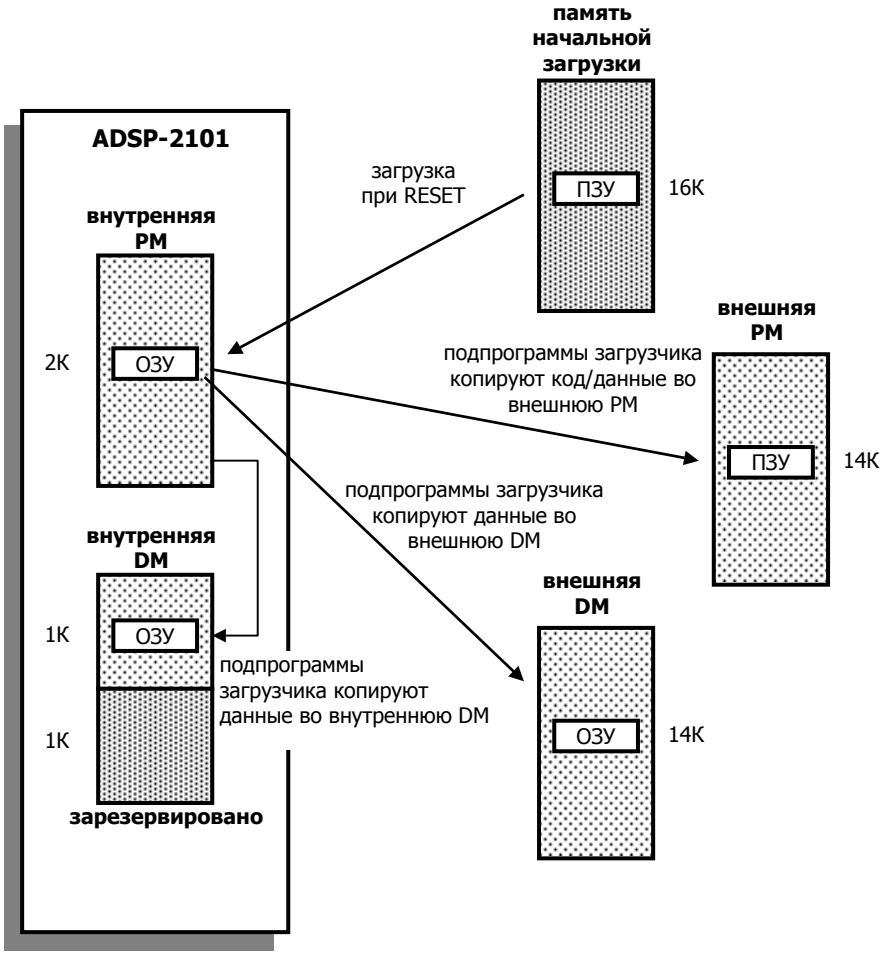


Рис.5.2. Загрузка программы и инициализация памяти.

Ключ `-loader` указывает разделителю программ просматривать входной исполняемый файл для поиска PM сегментов внешней памяти ОЗУ и DM сегментов внутренней или внешней памяти ОЗУ; при этом создается столько загружаемых страниц, сколько необходимо для того чтобы вместить все коды и данные, не обращая внимание на то, как много страниц объявлено в файле описания системной архитектуры. Вдобавок, небольшие загружающие подпрограммы создаются и размещаются в начале (младшие адреса) каждой страницы.

Рис.5.3. показывает программу, созданную с ключом `-loader`, которая содержится на страницах начальной загрузки 0, 1 и 2. Подпрограммы загрузки на страницах 0 и 1 содержат несколько циклов, каждая из которых после загрузки копирует сегмент кода или данных в соответствующую память программ или данных. Один цикл копирования требуется для каждого отдельного (несвязанного) сегмента.

В конце каждой страницы подпрограмм загрузчика следует последовательность инструкций, которая программно вызывает следующую страницу. Последняя страница (в приведенном примере страница 2) не имеет подпрограмм загрузки - она содержит первую 2К страницу исполняемой программы. После загрузки этой страницы, выполнение программы начнется с адреса 0 во внутренней памяти программ ADSP-21xx:

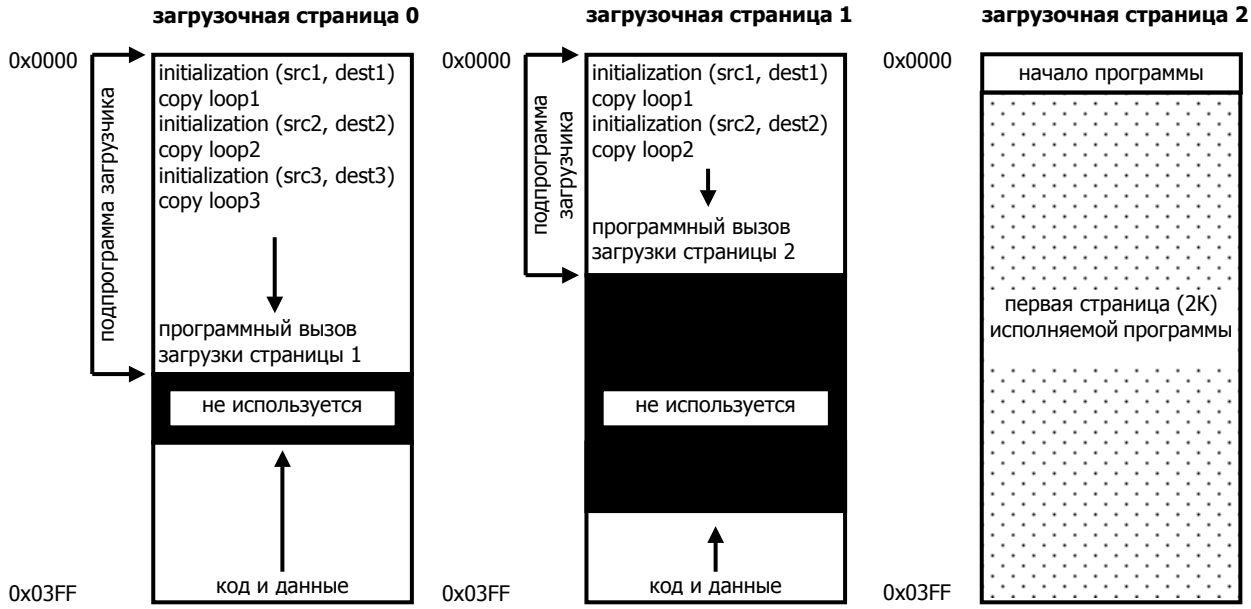


Рис.5.3. Пример программы начальной загрузки.

Ключ `-loader` заставляет программу разделителя размещать коды и данные на каждой загружаемой странице (кроме последней), начиная с самого старшего адреса вниз (Рис.5.3). Неиспользованная область, обычно, остается посередине.

После загрузки страницы 0 сегменты кодов и данных копируются подпрограммами загрузчика в соответствии с их назначениями. Затем страница 0 осуществляет программный вызов страницы 1, где подпрограммы загрузчика выполняют подобные операции. Последовательная загрузка продолжается до тех пор, пока программа не загрузится целиком, максимум до 15К памяти начальной загрузки.

На примере Рис.5.3. осуществляется следующая последовательность событий после перезапуска системы:

1. Страница 0 загружается во внутреннюю память программ ADSP-21xx. Выполняются подпрограммы загрузчика страницы 0, копирующие коды и данные со страницы 0 во внутреннюю DM, внешнюю PM и/или внешнюю DM.
2. Страница 1 загружается подпрограммой загрузчика страницы 0. Выполняются подпрограммы загрузчика страницы 1, копирующие коды и данные. Это завершает процесс инициализации.
3. Страница 2 загружается подпрограммой загрузчика страницы 1. Основная исполняемая программа стартует с адреса 0.

При установке системного управляющего регистра отраженного в памяти для вызова программной начальной загрузки, подпрограммы загрузчика по умолчанию устанавливают в поле BWAIT этого регистра значение 3.

5.5.1. Как подготовить программу для начального загрузчика

Если предполагается использование ключа `-loader` программы разделителя, чтобы создать загружаемую программу с автоматической инициализацией, необходимо предпринять следующие шаги:

1. Объявить максимальное допустимое количество загружаемых страниц, восемь, в вашем системном файле архитектуры.

2. Записать ваши ассемблерные модули без использования определителя `/BOOT` в директиве `.MODULE`.

3. Ассемблировать и скомпоновать.

4. Вызвать разделитель программ для программаторов ППЗУ с ключом `-loader`. Во время обработки показывается отображение процесса созданных страниц (Чтобы сохранить эту информацию воспользуйтесь командами операционной системы для перенаправления текста в файл).

Когда используется опция загрузчика нельзя присваивать программным модулям ассемблерный определитель `/BOOT`. Это позволит программе определить размещение загружаемой страницы для каждого модуля.

Заметьте, что при копировании кодов/данных во внешнюю память программ или данных подпрограммы загрузчика, добавленные к вашей программе, могут записать другие адреса памяти, чем те, что используются основной программой. Например, если основная программа инициализирует адреса DM[0×03FF] и DM[0×0401], и определен порт ввода/вывода отображенный по адресу DM[0×0400], подпрограмма загрузчика может записать в порт неопределенную величину. Необходимо гарантировать, что в системе не будет таких ложных записей. Один из способов сделать это состоит в объявлении для всех портов уникального сегмента памяти (директивой `.SEG` системного конфигуратора) в файле описания архитектуры. Если код или данные не размещаются в этом сегменте, порты не будут перезаписаны в течение инициализации памяти.

5.5.2. Моделирование программы начального загрузчика

Так как функция начальной загрузки реализуется программой разделителем, а не редактором связей, необходимо использовать выходной файл `.BNM` программы разделителя для моделирования загружаемой программы. Выходной файл редактора связей не содержит информации о памяти начальной загрузки.

Чтобы загрузить файл `.BNM` в программу моделирования ADSP-21xx или эмулятор, используйте команду LR, вместо команды L.

5.6. Файлы HIP загрузки (HIP splitter)

HIP разделитель программ – это утилита, подобная разделителю программ для записи в ППЗУ, которая позволяет создавать файлы, которые могут быть загружены через интерфейс управляющего порта (host interface port – HIP) ADSP-2111 и ADSP-21msp50.

HIP разделитель программ вызывается следующей командой:

```
hspl21  файлEXE  [addr]  [-boot]  [-i]
```

Входной файл *файлEXE* – это файл копии содержимого памяти редактора связей. Этот файл должен иметь расширение `.EXE`, добавленное редактором связей; в противном случае, HIP разделитель программ его не распознает. Дополнительный (необязательный) параметр `addr` характеризует отступление от нулевого стартового адреса для загрузочного HIP файла. Этот параметр должен быть введен десятичным номером, без префикса «0x».

Выходной файл HIP разделителя всегда называется `IMAGEFILE.HIP`. Формат файла подобен файлу `.BNM`, который создает разделитель программ для записи в ППЗУ для памяти программ или памяти начальной загрузки – первый байт загружается в регистр процессора HDR3, содержащий длину страницы. По умолчанию используется формат **Motorola S**. Используя ключ `-i` можно создавать файлы в формате **Intel Hex**.

Если HIP разделитель вызван без ключа `-boot`, `IMAGEFILE.HIP` содержит 24-разрядные слова памяти программ, размещенные в следующем порядке: **старший байт**, **средний байт**, **младший байт**. Чтобы разместить эти три байта в порядке, используемом операциями начальной загрузки ADSP-21xx: **старший байт**, **младший байт**, **средний байт**, используйте ключ `-boot`.