**3 Скрипты компоновщика.**

Каждая компоновка управляется скриптом компоновщика. Этот скрипт написан на языке команд компоновщика.

Главная цель скрипта компоновщика заключается в описании того, как секции во входных файлах должны быть спроецированы на выходной файл, и в управлении распределением памяти выходного файла. Большинство скриптов компоновщика не делают ничего кроме этого. Однако, когда необходимо, скрипт компоновщика может также направлять компоновщик на выполнение многих других операций, используя команды, описанные ниже.

Компоновщик всегда использует скрипт. Если вы не снабдите себя таким скриптом, компоновщик будет использовать скрипт по умолчанию, который компилируется в исполняемый файл компоновщика (linker executable). Вы можете использовать опцию командной строки «--verbose», чтобы показать скрипт компоновщика по умолчанию. Некоторые опции командной строки, такие как «-r» или «-N» будут влиять на скрипт компоновщика по умолчанию.

Вы можете предоставить ваш собственный скрипт компоновщика, используя опцию командной строки «-T». Когда вы сделаете это, ваш скрипт будет замещать скрипт по умолчанию.

**3.1 Базовая концепция скрипта компоновщика.**

Нам нужно определить некоторые базовые концепции и терминологию для того, чтобы описать язык скриптов компоновщика.

Компоновщик объединяет входные файлы в единственный выходной файл. Выходной файл и каждый входной файл представлены в специальном формате данных, известном как object file format. Каждый файл называется объектным файлом. Выходной файл часто называют исполняемым, но для наших целей мы будем также называть его объектным файлом. Каждый объектный файл среди прочего имеет список секций. Иногда мы называем секцию во входном файле как «входную секцию», аналогично, секцию в выходном файле называем «выходной секцией».

Каждая секция в объектном файле имеет имя и размер. Большинство секций также имеют соответствующий блок данных, называемый содержимым секции (section contents). Секция может быть помечена как загружаемая (loadable), что означает, что содержимое должно быть загружено в память, когда исполняемый файл запущен. Секция без содержимого может быть выделяемой (allocatable), что означает, что область памяти следует отделить, но туда не должно быть загружено ничего особенного (в некоторых случаях эта память должна быть заполнена нулями). Секция, которая не является ни загружаемой, ни выделяемой обычно содержит некоторую отладочную информацию.

Каждая загружаемая и размещаемая выходная секция имеет два адреса. Первый – это VMA, или виртуальный адрес памяти (virtual memory address). Этот адрес будет иметь секция, когда будет исполняться выходной файл. Второй адрес – это LMA, или загрузочный адрес памяти (load memory address). Это адрес, с которого будет загружена секция. В большинстве случаев, эти два адреса будут одинаковы. Пример, при котором они могут быть различными – это когда секция данных загружена в ROM, и затем скопирована в RAM в процессе отработки стартового кода (эта техника часто используется для инициализации глобальных переменных в ROM-based системах). В этом случае адрес в ROM будет адресом LMA, а адрес в RAM будет VMA.

Вы можете посмотреть секции в объектном файле с помощью программы objdump с опцией «-h». Каждый объектный файл также имеет список символов, называемый таблицей символов (symbol table). Символ может быть определённым и неопределённым. Каждый символ имеет имя, и каждый определённый символ имеет адрес кроме прочей информации. Если вы компилируете C или C++ программу в объектный файл, вы получите определённый символ для каждой определённой (NB: нестатической) функции и глобальной или статической переменной. Каждая неопределённая функция или глобальная переменная, на которую ссылается входной файл, становится неопределённым символом.

Вы можете увидеть символы в объектном файле с помощью программы nm, или используя программу objdump с опцией «-t».

**3.2 Формат скрипта компоновщика.**

Скрипты компоновщика — это текстовые файлы. Каждая команда — это либо ключевое слово, за которым следуют возможные аргументы, либо присвоение символу. Вы можете отделять команды с помощью точки с запятой. Пробельные символы вообще игнорируются.

Такие строки как имена файлов или форматов могут быть введены напрямую как обычно. Если имя файла содержит символ, такой как «запятая», которая с другой стороны должна служить разделителем имён файлов, вы можете заключить имя файла в двойные кавычки. Не существует способа использовать символ двойных кавычек в имени файла.

Вы можете включать комментарии в скрипт компоновщика просто как в C, выделяя их в /\* \*/. Как и в C, комментарии синтаксически эквивалентны пробельным символам.

**3.3 Простой пример скрипта компоновщика.**

Множество скриптов компоновщика довольно просты. В простейшем случае скрипт компоновщика содержит всего одну команду “SECTIONS”. Вы используете команду “SECTIONS” для описания разметки памяти выходного файла.

Команда “SECTIONS” - это мощная команда. Здесь мы опишем простейшее её использование. Давайте примем, что ваша программа состоит только из кода, инициализированных данных, и не инициализированных данных. Они будут размещаться соответственно в секциях .text, .data, .bss. Давайте примем в дальнейшем, что только эти секции появляются в ваших входных файлах.

Для этого примера, давайте скажем, что код должен быть загружен по адресу 0x10000, и что данные должны начинаться с адреса 0x8000000. Далее приведён скрипт компоновщика, который делает это:

SECTIONS

{

. = 0x10000;

.text : { \*(.text) }

. = 0x8000000;

.data : { \*(.data) }

.bss : { \*(.bss) }

}

Вы пишете команду “SECTIONS” как ключевое слово “SECTIONS”, за которой следует серия символьных заданий и описаний выходных секций, заключенные в фигурные скобки.

Первая строчка внутри команды SECTIONS вышеприведённого примера устанавливает значение для специального символа ‘.’, который представляет счётчик положения. Если вы не определяете адрес выходной секции каким-то другим образом (будут описаны позже), адрес устанавливается исходя из текущего значения счётчика положения. Счётчик положения затем инкрементируется на размер выходной секции. В начале команды SECTIONS счётчик положения имеет значение ‘0’.

Вторая строчка определяет выходную секцию, ‘.text’. Двоеточие – это требуемый синтаксис, который пока может быть проигнорирован. Внутри фигурных скобок после имени выходной секции, вы перечисляете имена входных секций, которые должны быть размещены в этой выходной секции. ‘\*’ – это подстановочный знак, которому соответствует любое имя файла. Выражение ‘\*(.text)’ означает все входные секции ‘.text’ во всех входных файлах.

Так как счётчик положения сейчас установлен в 0x10000, то когда определяется выходная секция ‘.text’, компоновщик установит адрес секции ‘.text’ в выходном файле в значение ‘0x10000’.

Оставшиеся строчки определяет секции ‘.data’ и ‘.bss’ в выходном файле. Компоновщик поместит выходную секцию ‘.data’ по адресу 0x8000000. После того, как компоновщик разместит выходную секцию ‘.data’, значение счётчика положения будет 0x8000000 + размер выходной секции ‘.data’. В результате компоновщик поместит в памяти выходную секцию ‘.bss’ немедленно после выходной секции ‘.data’.

Компоновщик гарантирует, что выходная секция будет иметь требуемое выравнивание, увеличивая счётчик положения, если это необходимо. В этом примере указанный адрес для секций ‘.text’ и ‘.data’ будет вероятно удовлетворять ограничениям по выравниванию, но компоновщик может создать небольшой зазор между секциями ‘.data’ и ‘.bss’.

Вот и всё! Это простейший и полностью завершенный скрипт компоновщика.

**3.4 Простые команды скрипта компоновщика.**

В этом разделе мы опишем простые команды скрипта компоновщика.

**3.4.1 Установка точки входа.**

Первая инструкция для исполнения в программе называется точкой входа (entry point). Вы можете использовать команду скрипта компоновщика ENTRY для установки точки входа. Аргументом является имя символа:

ENTRY(symbol)

Существует несколько путей для установки точки входа. Компоновщик установит точку входа, пытаясь использовать каждый из последующих методов по порядку, и остановится, когда один из них завершится успехом.

* ‘-e’ опция командной строки entry.
* ENTRY(symbol) команда в скрипте компоновщика.
* Значение специального для платформы символа, если он определён. Для многих платформ это start, но, например, системы на основе PE и BeOS проверяют список возможных символов точек входа, устанавливая в соответствие первый найденный.
* Адрес первого байта секции ‘.text’, если таковая имеется.
* Адрес 0.

**3.4.2 Команды для работы с файлами.**

**3.4.3 Команды для работы с файлами объектных форматов.**

**3.4.4 Назначение псевдонимов имён регионам памяти.**

Псевдонимы имён могут быть добавлены для существующих регионов памяти, созданных с помощью команды MEMORY. Каждое имя соответствует не более чем одному региону памяти.

REGION\_ALIAS(alias, region)

Функция создаёт псевдоним alias для региона памяти region. Это позволяет гибко проецировать выходные секции на регионы памяти. Ниже приведён пример.

Предположим, что у нас есть приложение для встраиваемых систем, которое работает с различными устройствами памяти. Все системы требуют наличия оперативной памяти RAM, которая позволяет исполнять код, или хранить данные. Некоторые могут иметь read-only память ROM, которая позволяет исполнять код, и читать данные. Последний вариант – это read-only память ROM2 с доступом к данным только на чтение, и не позволяющая исполнять из неё код. Мы имеем четыре выходных секции:

* .text – программный код;
* .rodata – read-only данные;
* .data – инициализированные данные read-write;
* .bss – инициализированные нулём данные read-write.

Цель – предоставить командный файл компоновщика, который содержит системно-независимую часть, определяющую выходные секции и системно-зависимую часть, проецирующую выходные секции на регионы памяти, доступные в системе. Наши встраиваемые системы имеют три различных конфигурации памяти A, B и C:

|  |  |  |  |
| --- | --- | --- | --- |
| Секция | A | B | C |
| .text | RAM | ROM | ROM |
| .rodata | RAM | ROM | ROM2 |
| .data | RAM | RAM/ROM | RAM/ROM2 |
| .bss | RAM | RAM | RAM |

Запись RAM/ROM или RAM/ROM2 означает, что эта секция загружается в регион ROM или ROM2 соответственно. Пожалуйста заметьте, что адрес загрузки секции .data начинается для всех трёх вариантов в конце секции .rodata.

Базовый скрипт компоновщика, который работает с выходными секциями, описан далее. Он включает системно-зависимый файл linkcmds.memory, который описывает разметку памяти.

INCLUDE linkcmds.memory

SECTIONS

{

.text : {\*(.text) } > REGION\_TEXT

.rodata : {\*(.rodata) rodata\_end = .;} > REGION\_RODATA

.data : AT (rodata\_end){data\_start = .;\*(.data)} > REGION\_DATA

data\_size = SIZEOF(.data);

data\_load\_start = LOADADDR(.data);

.bss : {\*(.bss)} > REGION\_BSS

}

Теперь нам нужны три различных системно-зависимых файла linkcmds.memory для определения регионов памяти и псевдонимов имён. Далее следует содержимое этих файлов:

A. Здесь все направляется в RAM.

MEMORY

{

RAM : ORIGIN = 0, LENGTH = 4M

}

REGION\_ALIAS("REGION\_TEXT", RAM);

REGION\_ALIAS("REGION\_RODATA", RAM);

REGION\_ALIAS("REGION\_DATA", RAM);

REGION\_ALIAS("REGION\_BSS", RAM);

B. Программный код и read-only данные направляется в ROM. Read-write данные направляются в RAM. Образ инициализированных данных загружается в ROM и будет скопирован в RAM при исполнении стартового кода.

MEMORY

{

ROM : ORIGIN = 0, LENGTH = 3M

RAM : ORIGIN = 0x10000000, LENGTH = 1M

}

REGION\_ALIAS("REGION\_TEXT", ROM);

REGION\_ALIAS("REGION\_RODATA", ROM);

REGION\_ALIAS("REGION\_DATA", RAM);

REGION\_ALIAS("REGION\_BSS", RAM);

C. Программный код направляется в ROM. Read-only данные следуют в ROM2. Read-write данные следуют в RAM. Образ инициализированных данных загружается в ROM2 и будет скопирован в RAM при выполнении стартового кода системы.

MEMORY

{

ROM : ORIGIN = 0, LENGTH = 2M

ROM2 : ORIGIN = 0x10000000, LENGTH = 1M

RAM : ORIGIN = 0x20000000, LENGTH = 1M

}

REGION\_ALIAS("REGION\_TEXT", ROM);

REGION\_ALIAS("REGION\_RODATA", ROM2);

REGION\_ALIAS("REGION\_DATA", RAM);

REGION\_ALIAS("REGION\_BSS", RAM);

Возможно написать общий обработчик инициализации системы, чтобы скопировать .data секцию из ROM или ROM2 в RAM, если это необходимо:

#include <string.h>

extern char data\_start [];

extern char data\_size [];

extern char data\_load\_start [];

void copy\_data(void)

{

if (data\_start != data\_load\_start)

{

memcpy(data\_start, data\_load\_start, (size\_t) data\_size);

}

}

**3.4.5 Другие команды скрипта компоновщика.**

**3.5. Присвоение значений символам.**

Вы можете назначать значение символу в скрипте компоновщика. Это определит символ и поместит его в символьную таблицу с глобальной областью видимости.

**3.5.1 Простые присвоения.**

Вы можете выполнять присвоение, используя любой из операторов присвоения C:

symbol = expression ;

symbol += expression ;

symbol -= expression ;

symbol \*= expression ;

symbol /= expression ;

symbol <<= expression ;

symbol >>= expression ;

symbol &= expression ;

symbol |= expression ;

Первый случай определит symbol со значением expression. В остальных случаях, symbol должен быть уде определён и его значение будет корректироваться соответствующим образом.

Специальный символ ‘.’ Индицирует счётчик положения. Вы можете его использовать только в пределах команды SECTIONS (см. раздел 3.10.5 [счётчик положения], стр. 73).

После expression требуется точка с запятой.

Expressions определены ниже (см. раздел 3.10 [выражения], стр.72).

Вы можете писать символьные присвоения как команды, с их собственными правами, или как операторы в пределах команды SECTIONS, или как часть описания выходной секции в команде SECTIONS. Секция символа (NB: вместо выходной секции пишем имя символа) будет установлена из секции выражения (см. раздел 3.10.8 [выражение секции], стр.77).

Здесь приводится пример, показывающий три различных места, которые могут быть использованы для символьных присвоений:

floating\_point = 0; /\* как команда \*/

SECTIONS

{

.text : {\*(.text) \_etext = .; /\* как оператор \*/

}

\_bdata = (. + 3) & ~ 3; /\* как часть описания выходной секции \*/

.data : { \*(.data) }

}

В этом примере, символ ‘floating\_point’ будет определён как ноль. Символ ‘\_etext’ будет определён как адрес после последней входной секции ‘.text’. Символ ‘\_bdata’ будет определён как адрес после выходной секции ‘.text’, выровненный по границе в 4 байта.

**3.5.2 HIDDEN.**

**3.5.3 PROVIDE.**

В некоторых случаях, желательно определять в скрипте компоновщика символ, только если на него ссылаются, и он не определяется каким-либо объектом, включенным в компоновку. Например, традиционные компоновщики определяют символ ‘etext’. Однако, ANSI C требует, чтобы пользователь мог использовать ‘etext’ как имя функции без столкновения с ошибкой. Ключевое слово PROVIDE может быть использовано для определения символа, такого как ‘etext’ только если на него ссылаются, но не определяют. Синтаксис:

PROVIDE(symbol=expression).

Здесь приведён пример использования PROVIDE для определения ‘etext’:

SECTIONS

{

.text : {\*(.text) \_etext = .; PROVIDE(etext = .);}

}

В этом примере, если программа определит ‘\_etext’ (начинающийся с символа подчёркивания), компоновщик выдаст несколько ошибок определения. Если, с другой стороны, программа определит ‘etext’ (без знака подчёркивания), компоновщик тихо использует определение в программе. Если программа ссылается на ‘etext’, но не определяет его, компоновщик будет использовать определение в скрипте.

NB: таким образом, в ‘PROVIDE’ оборачивается символьное определение, которое может быть без ошибок компоновки переопределено в программе.

**3.5.4 PROVIDE\_HIDDEN.**

**3.5.5 Ссылки в исходном коде.**

Обращение к определённым в скрипте компоновщика переменным из исходного кода не интуитивно. В частности, символ скрипта компоновщика не является эквивалентом объявления переменной на языке высокого уровня. Вместо этого, он является символом, который не имеет значения.

Прежде чем двигаться дальше, важно заметить, что компиляторы часто преобразовывают имена в исходном коде в другие имена при сохранении их в таблице символов. Например, C++ выполняет расширение ‘name mangling’ (NB: коверканье имён). Таким образом, может быть несоответствие между именем переменной, как если бы она использовалась в исходном коде, и именем той же переменной, которая определена в скрипте компоновщика. Например, в C ссылка на переменную скрипта компоновщика может иметь вид:

extern int foo;

Но в скрипте компоновщика она может быть определена как:

\_foo = 1000;

В остальных примерах, однако, предполагается, что никаких трансформаций имён не произошло.

Когда символ определяется на языке высокого уровня, таком как C, происходит две вещи. Первая – компилятор резервирует достаточное пространство памяти программы для хранения значения символа. Вторая – компилятор создаёт запись в символьной таблице программы, которая хранит адрес символа. Т.е. символьная таблица содержит адрес блока памяти, по которому хранится значение символа. Так, для примера следующего C объявления в пределах видимости файла:

int foo = 1000;

создаётся запись, называемая ‘foo’ в символьной таблице. Эта запись хранит адрес блока памяти размером ‘int’, в котором изначально сохранено число 1000.

Когда программа ссылается на символ, компилятор производит код, который сперва обращается к символьной таблице, чтобы найти адрес блока памяти символа, а затем код, который считывает значение из этого блока. Итак:

foo = 1;

ищет символ «foo» в символьной таблице, получает соответствующий адрес и затем пишет значение 1 в этот адрес. Тогда как:

int \*a = &foo;

Ищет символ ‘foo’ в таблице символов, получает его адрес и затем копирует этот адрес в блок памяти, соответствующий переменной ‘a’.

NB: пример:

// foo = 7;

movs r2, #7 // загружает константу 7 в регистр r2

ldr r3, .L3 // загружает адрес метки .L3 (переменной foo) в регистр r3

str r2, [r3] // загружает содержимое регистра r2 по адресу, указанному r3

Объявления символов скриптов компоновщика, напротив, создают запись в символьной таблице, но не выполняют назначение им какой либо памяти. Таким образом, они являются адресами без значения. Итак, например, в скрипте имеется определение:

foo = 1000;

Создаёт запись в символьной таблице, называемую ‘foo’, которая хранит адрес позиции памяти 1000, но ничего при этом не сохраняет по адресу 1000 (NB: т.е. каждый символ, определенный в скрипте компоновщика – это именованный адрес в памяти). Это означает, что вы не можете получить значение определенной в скрипте переменной – у неё нет значения – все, что вы можете сделать, это получить адрес определенного в скрипте компоновщика символа.

Следовательно, когда вы используете символ, определённый в скрипте компоновщика, вы должны всегда брать адрес символа, и никогда не пытаться использовать его значение. Для примера положим, что вы хотите скопировать содержимое секции памяти с именем .ROM в секцию с именем .FLASH и скрипт компоновщика содержит объявления:

start\_of\_ROM = .ROM

end\_of\_ROM = .ROM + sizeof(.ROM)

start\_of\_FLASH = .FLASH

Исходный код C, выполняющий копирование, должен быть следующим:

extern char start\_of\_ROM, end\_of\_ROM, start\_of\_FLASH;

memcpy (& start\_of\_FLASH, & start\_of\_ROM, & end\_of\_ROM - & start\_of\_ROM);

Заметьте, что используется оператор взятия адреса ‘&’. Это корректно (NB: start\_of\_ROM = 5 было бы не корректной записью, так как реальной памяти не зарезервировано!). Альтернативно, символы могут рассматриваться как имена векторов или массивов и затем код будет снова работать, как и ожидалось:

extern char start\_of\_ROM[], end\_of\_ROM[], start\_of\_FLASH[];

memcpy (start\_of\_FLASH, start\_of\_ROM, end\_of\_ROM - start\_of\_ROM);

Обратите внимание, что такой способ не требует применения операции взятия адреса ‘&’.

NB: ещё один пример использования символов, определённых в скрипте компоновщика – указатель стека.

Имеется запись в скрипте:

\_\_stack = ORIGIN(RAM) + LENGTH(RAM);

\_estack = \_\_stack; /\* STM specific definition \*/

Здесь определяется символ \_estack для таблицы символов, в которой содержится его адрес (адрес вершины стека). Далее этот символ используется в исходном коде (vectors\_DEVICE.c):

extern unsigned int \_estack;

typedef void (\* const pHandler)(void);

\_\_attribute\_\_ ((section(".isr\_vector"),used)) pHandler \_\_isr\_vectors[] =

{ //

(pHandler) &\_estack, // The initial stack pointer

Reset\_Handler, // The reset handler

…

Здесь определяется массив константных указателей на функцию, не принимающую аргументов, и не возвращающую значений. В качестве первого элемента массива указан адрес символа \_estack, определённого в скрипте линковщика. Таким образом, адрес стека определяется в скрипте, а его значение может быть получено в стартовом коде через операцию взятия адреса ‘&’.

**3.6 Команда SECTIONS.**

Команда SECTIONS говорит компоновщику, как проецировать входные секции на выходные секции, и как размещать выходные секции в памяти. Формат команды следующий:

SECTIONS

{

sections-command

sections-command

...

}

Каждый sections-command может быть:

1. Командой ENTRY.

2. Символьным присваиванием.

3. Описанием выходной секции.

4. Описанием наложения.

Команда ENTRY и символьные присвоения допускаются внутри команды SECTIONS для удобства использования счётчика положения в этих командах. Это может также сделать скрипт компоновщика легче для понимания, потому что вы можете использовать эти команды в значимых точках в разметке выходного файла.

Описания выходных секций и наложений приводятся ниже.

Если вы не используете команду SECTIONS в вашем скрипте компоновщика, он поместит каждую входную секцию в выходную секцию с идентичным именем в порядке, в котором секции впервые встречаются во входных файлах. Например, если все входные секции присутствуют в первом файле, порядок секций в выходном файле будет соответствовать порядку в первом входном файле. Первая секция будет расположена по нулевому адресу.

**3.6.1 Описание выходной секции.**

Полное описание выходной секции похоже на это:

section [address] [(type)] :

[AT(lma)]

[ALIGN(section\_align) | ALIGN\_WITH\_INPUT]

[SUBALIGN(subsection\_align)]

[constraint]

{

output-section-command

output-section-command

...

} [>region] [AT>lma\_region] [:phdr :phdr ...] [=fillexp] [,]

Большинство опциональных атрибутов секций не используется в большинстве описаний секций. Вокруг «секция» требуются пробельные символы, так что имя секции однозначно. Двоеточие и фигурные скобки также требуются. Запятая в конце может потребоваться, если fillexp используется и следующая section-command похожа на продолжение выражения. Перенос строки и другие пробельные символы являются опциональными.

Каждый output-section-command может быть одним из:

1. Символьное присвоение.

2. Описание входной секции.

3. Значения данных для непосредственного включения.

4. Специальное ключевое слово выходной секции.

**3.6.2 Имя выходной секции.**

Имя выходной секции — это section. Section должна удовлетворять ограничениям вашего выходного формата. В форматах, которые поддерживают только ограниченное количество секций, такие как a.out, имя должно быть одним из тех, что поддерживаются форматом (a.out, например, допускает только .text, .data или .bss). Если выходной формат поддерживает любое количество секций, но с номерами, а не именами (как в случае с Oasys), имя должно быть представлено как числовая строка в кавычках. Имя секции может состоять из любой последовательности символов, но имя, которое содержит любые нетипичные символы, такие как запятые, должны заключаться в кавычки.

Имя выходной секции /DISCARD/ - специальное.

**3.6.3 Адрес выходной секции.**

Address — это выражение для VMA выходной секции. Этот адрес опционален, но если он предоставляется, то выходной адрес будет установлен точно в соответствии с указанным (NB: дополнительного выравнивания не будет – адрес устанавливается непосредственно).

Если выходной адрес не указан, то он будет выбран для секции на основе эвристики ниже. Этот адрес будет скорректирован, чтобы удовлетворить требованию выравнивания выходной секции. Требование выравнивания является строжайшим выравниванием любой входной секции, содержащейся внутри выходной секции (NB: т. е. входная секция, имеющая более строгое выравнивание, чем другие, задаёт выравнивание для выходной секции).

Эвристика адреса выходной секции следующая:

1. Если выходной регион памяти установлен для секции, то она добавляется к этому региону и её адрес будет следующим свободным адресом в этом регионе.

2. Если была использована команда MEMORY для создания списка регионов памяти, то первый регион, который имеет атрибуты, совместимые с секцией, выбирается для её включения. Выходной адрес секции будет следующим свободным адресом в этом регионе.

3. Если никакие регионы памяти не были определены, или нет соответствия секций, то выходной адрес секции будет определён на основе текущего значения счётчика положения.

Например:

.text . : {\*(.text)} и .text : {\*(.text)} — тонко отличаются. Первая запись установит адрес выходной секции .text в текущее значение счётчика положения. Вторая установит его в текущее значение счётчика положения, выровненное по значению выравнивания любой из входных секций .text.

Адрес может быть произвольным выражением. Например, если вы хотите выравнивать секцию по 16-ти байтовой границе, то младшие 4 бита адреса секции — нулевые. Вы можете сделать что-то вроде:

.text ALIGN(0x10) : {\*(.text)}

Это работает, потому что ALIGN возвращает счётчик текущего положения, выровненный вверх по указанной границе.

Указание адреса для секции изменит значение счётчика положения, при условии, что секция не пустая.

**3.6.3 Описание входной секции.**

Наиболее общей output-section-command является описание входной секции. Описание входной секции — это самая базовая операция в скрипте компоновщика. Вы используете описания входной секции, чтобы сказать компоновщику, как ему проецировать входные файлы на память.

**3.6.4.1 Основы входных секций.**

Описание входной секции состоит из имени файла, за которым следует опциональный список имён секций в круглых скобках. Имя файла и имя секции может быть подстановочными шаблонами, которые мы опишем в дальнейшем.

Наиболее распространённое описание входной секции — это включение всех входных секций с определённым именем в выходную секцию. Например, чтобы включить все входные секции .text, вы должны записать:

\*(.text)

Здесь \* это подставочный символ, которому соответствует любое имя файла. Для запрета «попадания» списка файлов, может быть использован EXCLUDE\_FILE — для сопоставления всех файлов, кроме некоторых, указанных в списке EXCLUDE\_FILE. Например:

\*(EXCLUDE\_FILE (\*crtend.o \*otherfile.o).ctors)

Такая запись включит все секции .ctors всех файлов, кроме файла crtend.o и otherfile.o. Существует два способа включения более чем одной секции:

\*(.text .rdata)

\*(.text) \*(.rdata)

Разница между ними заключается в порядке, в котором входные секции .text и .rdata будут появляться в выходной секции. В первом примере, они будут перемешаны, появляясь в том же порядке, в каком они подаются на вход компоновщика. Во втором примере, все входные секции .text будут появляться первыми, а следом все входные секции .rdata.

Вы можете указать имя файла для включения секций из определённого файла. Вы могли бы сделать это, если один или более ваших файлов содержат специальные данные, которые необходимо размещать в определённом месте памяти. Например:

data.o(.data)

Для определения секций, которые включаются на основе флагов секции входной секции, может быть использован INPUT\_SECTION\_FLAGS. Пример, иллюстрирующий флаги заголовка секции для ELF секций:

SECTIONS

{

.text : {INPUT\_SECTION\_FLAGS(SHF\_MERGE & SHF\_STRINGS) \*(.text)}

.text2 : {INPUT\_SECTION\_FLAGS (!SHF\_WRITE) \*(.text)}

}

В этом примере выходная секция .text будет состоять из любых входных секций, подходящих по имени \*(.text), чьи флаги заголовка секции SHF\_MERGE и SHF\_STRINGS будут установлены. Выходная секция .text2 будет состоять из любых входных секций, подходящих по имени \*(.text), чей флаг заголовка секции SHF\_WRITE будет сброшен.

Вы можете также указать файлы внутри архивов написанием шаблона, соответствующего архиву, двоеточие, затем шаблона, соответствующего файлу, без пробельных символов вокруг двоеточия.

‘archive:file’ соответствует файлу в архиве

‘arhcive’ соответствует всему архиву

‘:file’ соответствует файлу, но not one in an archive.

<…>

**3.6.4.2 Подстановочные шаблоны входной секции.**

**3.6.4.3 Входная секция для общих символов.**

Для общих символов необходима специальная запись, потому что во многих форматах объектных файлов общие символы не имеют определённой входной секции. Компоновщик рассматривает общие символы как если бы они находились во входной секции ‘COMMON’.

Вы можете использовать имена файлов с секцией ‘COMMON’ просто как с любыми другими входными секциями. Вы можете использовать это для помещения общих символов из определённого входного файла в одну секцию, в то время как общие символы из других входных файлов помещаются в другую секцию.

В большинстве случаев, общие символы во входных файлах будут помещены в секцию .bss, в выходной файл. Например:

.bss { \*(.bss) \*(COMMON) }

<…>

**3.6.4.4 Входная секция и уборка мусора.**

Когда используется уборка мусора на этапе компоновки (--gc-sections), часто полезно отмечать секции, которые не должны исключаться. Это достигается заключением шаблонной записи секции в KEEP(), как например KEEP(\*(.init)), или KEEP(SORT\_BY\_NAME(\*)(.ctors)).

**3.6.4.5. Пример входной секции.**

Следующий пример – это полностью завершенный скрипт компоновщика. Он говорит компоновщику читать все секции из файла ‘all.o’ и размещать их в начале выходной секции ‘outputa’, которая начинается с адреса 0x10000. Вся секция ‘.input1’ из файла ‘foo.o’ следуют немедленно, в той же выходной секции. Вся секция ‘.input2’ из файла ‘foo.o’ направляется в выходную секцию ‘outputb’, за которой следует секция ‘.input1’ из файла ‘foo1.o’. Все остальные секции ‘.input1’ и ‘.input2’ из любых файлов записываются в выходную секцию ‘outputc’.

SECTIONS

{

outputa 0x10000 : {all.o foo.o (.input1)}

outputb : {foo.o (.input2) foo1.o (.input1)}

outputc : {\*(.input1) \*(.input2)}

}

**3.6.5 Данные выходной секции.**

Вы можете включать явно байты данных в выходную секцию, используя BYTE, SHORT, LONG, QUAD или SQUAD в качестве output-section-command. За каждым ключевым словом следует выражение в круглых скобках, выражающее значение для хранения. Значение выражения сохраняется по текущему адресу счётчика положения.

BYTE, SHORT, LONG, QUAD сохраняют 1, 2, 4, 8 байтов соответственно. После сохранения байтов, счётчик положения увеличивается на количество сохранённых байтов.

Например, это сохранит 1 байт, за которым следуют 4 байта значения символа ‘addr’:

BYTE(1)

LONG(addr)

Когда используются 64-битный хост или платформа, QUAD и SQUAD не различаются. Они оба хранят 8-байтовое, или 64-битное значение. Когда и хост и платформа 32-битные, выражение вычисляется как 32-битное. В этом случае QUAD хранит 32-битное значение, расширяемое нулём до 64-битного, а SQUAD хранит 32-битное значение, расширяемое знаком до 64-ёх битов.

Если формат выходного объектного файла имеет явный endianness, что является нормальным, значение будет сохранено в этом endianness. Когда формат объектного файла не имеет явного endianness, что справедливо, например, S-записей, значение будет сохранено в endianness первого входного объектного файла.

Заметьте, что эти команды работают только внутри описания секции, но не между ними, так что следующая запись приведёт к ошибке компоновщика:

SECTIONS

{

.text : { \*(.text) }

LONG(1)

.data : { \*(.data) } }

В то время как вот эта запись будет работать:

SECTIONS

{

.text : { \*(.text) ; LONG(1) }

.data : { \*(.data) }

}

Вы можете использовать команду FILL для установки паттерна заполнения для текущей секции. За ней следует выражение в скобках. Любые неопределённые регионы памяти внутри секции (например, щели между секциями, возникающие из-за требования выравнивания входных секций) заполняются значением выражения, при необходимости, повторяющимся. Оператор FILL охватывает ячейки памяти после точки, в которой он встречается при описании секции. Включая более одного оператора FILL, вы можете иметь различные паттерны заполнения в различных частях выходной секции.

Следующий пример показывает, как заполнить неопределённые регионы памяти значением 0x90:

FILL(0x90909090)

Команда FILL похожа на атрибут ‘=fillexp’ выходной секции, но она влияет только на часть секции, следующей за командой FILL, а не на всю секцию. Если имеют место быть оба способа, то команда FILL имеет приоритет.

**3.6.6 Ключевые слова выходной секции.**

**3.6.7 Отбрасывание выходной секции.**

Компоновщик нормально не создаёт выходные секции без содержимого. Это удобно когда есть ссылки на входные секции, которые могут, или не могут присутствовать в каких-либо входных файлах. Например:

.foo : { \*(.foo) }

Выходная секция ‘.foo’ будет создана в выходном файле, если существует секция ‘.foo’ хотя бы в одном входном файле, и если входная секция не пустая. Выходная секция также будет создана, если есть другие директивы скрипта компоновщика, которые выделяют место в выходной секции. Также при присваивании точке, даже если это не создаёт пространство, но за исключением:

. = 0

. = . + 0

. = sym (при sym = 0)

. = . + sym (при sym = 0)

. = ALIGN(. != 0, expr, 1)

Это позволяет вам принудительно создать выходную пустую секцию с помощью ‘. = .’.

Компоновщик будет игнорировать адресные назначения на отброшенных секциях, кроме случаев, когда скрипт компоновщика определяет символы в выходной секции, при этом возможно продвигая точку, даже если секция отброшена.

Специальное имя секции ‘/DISCARD/’ может быть использовано для отбрасывания входных секций. Любые входные секции, которые привязаны к выходной секции с именем ‘/DISCARD/’, не включаются в выходной файл.

**3.6.8 Атрибуты выходной секции.**

Мы показали выше, что полное описание выходной секции похоже на:

section [address] [(type)] :

[AT(lma)]

[ALIGN(section\_align)]

[SUBALIGN(subsection\_align)]

[constraint]

{

output-section-command

output-section-command

...

} [>region] [AT>lma\_region] [:phdr :phdr ...] [=fillexp]

Мы уже описали из этой структуры: section, address и output-section-command. В этом разделе мы опишем оставшиеся атрибуты секции.

**3.6.8.1 Тип выходной секции.**

Каждая выходная секция может иметь тип. Тип – это ключевое слово в скобках. Определены следующие типы:

NOLOAD – секция, которая помечена как не загружаемая, так что она не будет загружаться в память при запуске программы.

DSECT, COPY, INFO, OVERLAY – эти имена типов поддерживаются для обратной совместимости и редко используются. Они все работают одинаково: секция должна быть отмечена как не выделяемая (not allocatable), так что никакой памяти не выделяется для секции, когда программа запущена.

Компоновщик обычно устанавливает атрибуты выходной секции на основе входной секции, которая в неё размещается. Вы можете переопределить их, используя тип. Например, в примере скрипта ниже, секция ‘ROM’ адресована в позицию памяти ‘0’ не должна загружаться при запуске программы.

SECTIONS {

ROM 0 (NOLOAD) : { ... }

...

}

**3.6.8.2 LMA выходной секции.**

Каждая секция имеет виртуальный адрес (VMA) и загрузочный адрес (LMA). Определение виртуального адреса было рассмотрено в разделе 3.6.3. Загрузочный адрес указывается с помощью ключевого слова AT или AT>. Указание загрузочного адреса опционально.

Ключевое слово AT принимает выражение в качестве аргумента. Оно определяет точный адрес загрузки секции. Ключевое слово AT> принимает имя региона в качестве аргумента. Смотри раздел 3.7 [MEMORY]. Загрузочный адрес секции устанавливается на следующий свободный адрес в регионе, выровненный в соответствии с требованием выравнивания секции.

Если для размещаемой секции не указан ни AT, ни AT>, то компоновщик будет использовать следующую эвристику для определения загрузочного адреса:

* Если секция имеет определённый VMA адрес, то он используется также как LMA адрес тоже.
* Если секция не выделяемая (not allocatable), то её LMA устанавливается в значение её VMA.
* В противном случае, если может быть найден регион памяти, совместимый с текущей секцией, и этот регион содержит, по крайней мере, одну секцию, то LMA устанавливается так, что разница между VMA и LMA была такой же, как и разница VMA и LMA у последней секции в регионе размещения.
* Если никакие регионы памяти не были объявлены, то для предыдущего шага будет использоваться регион по умолчанию, который охватывает всё адресное пространство.
* Если подходящий регион не может быть найден, или предыдущей секции не было, то LMA устанавливается равным VMA.

Эта возможность введена, чтобы легко можно было собрать ROM образ. Например, следующий скрипт компоновщика создаёт три выходные секции: одна называется ‘.text’, начинается с адреса 0x1000, вторая называется ‘.mdata’, которая загружается c конца секции ‘.text’, хотя её VMA равен 0x2000, и третья называется ‘.bss’, которая хранит неинициализированные данные по адресу 0x3000. Символ \_data определен со значением 0x2000, и показывает, что счётчик положения хранит VMA значение, не LMA.

SECTIONS

{

.text 0x1000 : { \*(.text) \_etext = . ; }

.mdata 0x2000 : AT ( ADDR (.text) + SIZEOF (.text) )

{

\_data = . ; \*(.data); \_edata = . ;

}

.bss 0x3000 :

{

\_bstart = . ; \*(.bss) \*(COMMON) ; \_bend = . ;

}

}

Стартовый код должен включать что-то похожее на ниже приведённый пример, для того, чтобы скопировать данные из образа ROM по адресу, который программа использует при выполнении. Обратите внимание, как этот код использует преимущество определяемых в скрипте компоновщика символов.

extern char \_etext, \_data, \_edata, \_bstart, \_bend;

char \*src = &\_etext;

char \*dst = &\_data;

/\* ROM has data at end of text; copy it. \*/

while (dst < &\_edata)

\*dst++ = \*src++;

/\* Zero bss. \*/

for (dst = &\_bstart; dst< &\_bend; dst++)

\*dst = 0;