SDCC-NOINIT

Перемещаемые модули

Алексей, aka SfS

Первая версия генератора перемещаемых модулей

Введение

Продолжая совершенствование своей поделки под названием SDCC-NOINIT, я подумал, а не замахнуться ли мне на ~~Вильяма нашего Шекспира~~ создание подгружаемых и перемещаемых программных модулей, например библиотек?

Для чего это надо

Как говорил Сунь Цзы, а может и не он, но кто-то точно говорил — точность слов есть основа понимания. Посему для начала определимся, что такое «подгружаемый и перемещаемый программный модуль», который в дальнейшем мы будем именовать просто «модуль» или «динамическая библиотека».

Итак, «подгружаемый и перемещаемый программный модуль» - это такой программный модуль, который независимо от основной программы (или программ) может быть загружен в память и выгружен из неё. Кроме того, загружаться в память, такой модуль может по любому произвольному адресу.

Собственно, такие модули вам хорошо известны — это **.so** — библиотеки в Linux или **.dll** в Windows. На ZX-Spectrum такого не хватает.

Удобство таких модулей очевидно. Если это библиотека, то можно написать несколько разных версий и загружать требуемую без компиляции программы. Если это драйвер — то понятно, что его можно и нужно загружать только тогда, когда он нужен, а не впихивать в тело программы навечно. Это может быть и плагин или уровень игры — загрузили, отработали, удалили.

Есть у такого подхода и некоторые недостатки — это дополнительный код для вызова процедур, например.

Две основные проблемы, которые пришлось решать, реализуя механизм подгружаемых и перемещаемых программных модулей — это привязка к адресам и связь модуля с другими модулями или основной программой.

Сразу оговорюсь, то, что здесь приведено — это первая версия, где все сделано «на скорую руку». Но это работает и может использоваться, а значит достойно того, чтобы выкинуть в свет — большинство, конечно пройдёт мимо, но может кто-то и заинтересуется.

Привязка к адресам или перемещение

К сожалению, родовые ~~травмы~~ особенности **SDCC** и моя лень не позволили сделать модули загружаемыми по любому произвольному адресу. Поэтому пришлось сделать загрузку перемещаемых модулей по адресу, кратному 256. На практике это не очень серьезное ограничение, поскольку выносить программный модуль в 10-100 байт смысла не имеет.

Я пошел по пути наименьшего сопротивления — вычисляю таблицу перемещений путем сравнения двух версий одного модуля, первая версия линкуется с адреса **0**, вторая — с

адреса **256**. На основании сравнений строится таблица перемещений и присоединяется в конец модуля.

После того, как модуль загружается в память, с помощью функции **soReloc()** специальной библиотеки **libso** модуль привязывается к адресу загрузки.

Функция **void* soReloc(void* load_adr)**; очень проста в обращении. На вход данная функция принимает адрес загруженного в память модуля **load_adr**, а возвращает она первый свободный адрес за концом загруженного модуля. Причем, таблица перемещения отсекается — ведь во время работы она не нужна. На рис.1 это показано подробно.

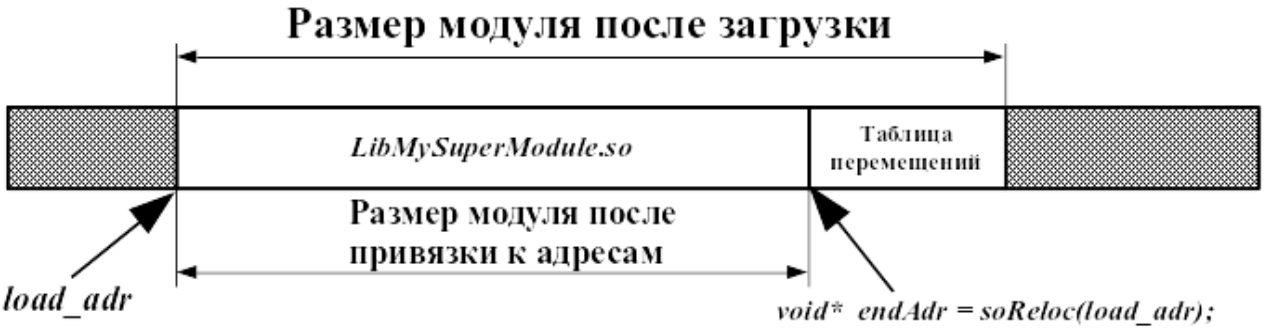

Рис. 1: Размещение модуля и привязка к адресам

load_adr — это адрес загрузки модуля;

endAdr — адрес после того, как функция **soReloc()** привяжет модуль к адресам с помощью таблицы перемещений.

Видно, что память расходуется довольно экономно, не засоряется ненужными данными.

Ниже, в практическом примере, мы ещё увидим как работает данная функция, а пока займемся филофским вопросом — как связать модули (или модуль и прогпрамму).

Связь программных модулей

Программные модули могут связываться различным образом. Самый старый и простой способ связи — это статическая линковка. Ей мы постоянно пользуемся. Этот тот самый способ, когда множество ***.rel** и ***.lib** файлов связываются в единый бинарный файл-образ программы. Разумеется, что в готовом бинарном файле мы не можем уже произвольно менять тот или иной программный модуль. Вообще, при статической линковке разбиение на некие «программные модули» условно — итог всё равно единый неделимый бинарь.

Другой способ — динамическая линковка. Это такой способ, когда программные модули загружаются в память по отдельности, а затем как-то связываются между собой. Способов связи или «интерфейсов» может быть несколько.

Во-первых, можно сделать так, чтобы программа после настройки загруженного модуля прозрачно использовала его функции. Именно этот способ продемонстрирован в примере ниже. Но такой способ не позволит использовать несколько подгружаемых модулей с одинаковым интерфейсом (например, драйверов). Поэтому есть и другой способ - «во-вторых».

Во-вторых, можно сделать так, чтобы у модуля была функция, возвращающая указатель на его «интерфейс». Таким интерфейсом может быть, например, структура, в которой хранятся ссылки на определенные функции. Понятно, что таких структур может быть несколько и тогда программа может использовать несколько модулей с одинаковым интерфейсом. Поскольку, первый способ наиболее сложен в реализации, я начал именно с него.

Как устроен подгружаемый модуль

Разберемся, как устроен подгружаемый модуль. Собственно модуль — это файл с расширением **.so**. Расширение я выбрал как в Linux — **s**hared **o**bject.

Модуль состоит из заголовка, интерфейсной части, программного кода, данных и таблицы перемещения, как показано на рис.2.

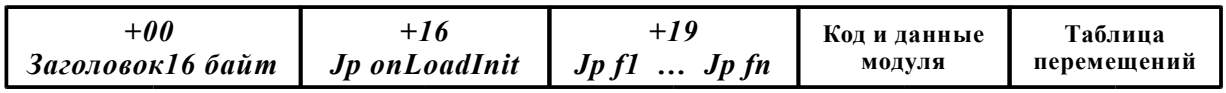

Рис. 2: Структура модуля версии 1

16-байтный заголовок имеет простой формат:

```
// +00 от начала модуля
.dw    0x424C      ;//+00 сигнатура LB
.dw    0x0001      ;//+02 Версия
.dw    libName     ;//+04 адрес ASCIIZ-строки названия модуля
.dw    reloc_t;    ;//+06 Адрес таблицы перемещений
.dw    0x0000      ;//+08 Счетчик ссылок на модуль
.dw    0x0000      ;//+0A — не используется, всегда 0
.dw    0x0000      ;//+0C — не используется, всегда 0
.dw    0x0000      ;//+0E — не используется, всегда 0
```

За заголовком следует интерфейсная часть. Самой первой функцией, которая вызывается после того, как модуль привязан к адресам, является внутренняя функция **onLoadInit**, инициализирующая некоторые переменные. Пользователь ей обычно не пользуется, поэтому не будем на ней останавливаться.

Далее, со смещения **+19** от начала файла, расположены команды перехода на функции модуля — по при байта на команду. В настоящее время реализовано только бинарное связывание, поэтому имена функций не сохраняются. В дальнейшем, возможно доработаю, чтобы можно было связывать и по имени функции.

Как уже говорилось, после привязки модуля к адресу загрузки, таблица перемещений становится не нужна и её можно удалить и использовать место для других нужд.

Прозрачный вызов функций

Если нам необходимо организовать прозрачный вызов функций модуля, то к подгружаемому модулю добавляется ещё и обычная статическая библиотека **.lib**, в которой описана интерфейсная резидентная часть. На рис.3 показано как происходит «прозрачный» вызов функции подгружаемого модуля из программы.

test-so — головная программа.

libtestso.lib — резидентная (интерфейсная) часть модуля, статически линкуемая с головной программой.

testso_v1.so — подгружаемый модуль.

Разберёмся для чего нужен такой зверь, как «интерфейсная резидентная часть». Когда программа линкуется, адреса всех вызываемых функций должны быть определены. Но, поскольку, подгружаемый модуль будет загружен отдельно, да ещё и неизвестно по каким адресам — то определить адрес функции из этого модуля на этапе линковки невозможно. Как нам тут быть?

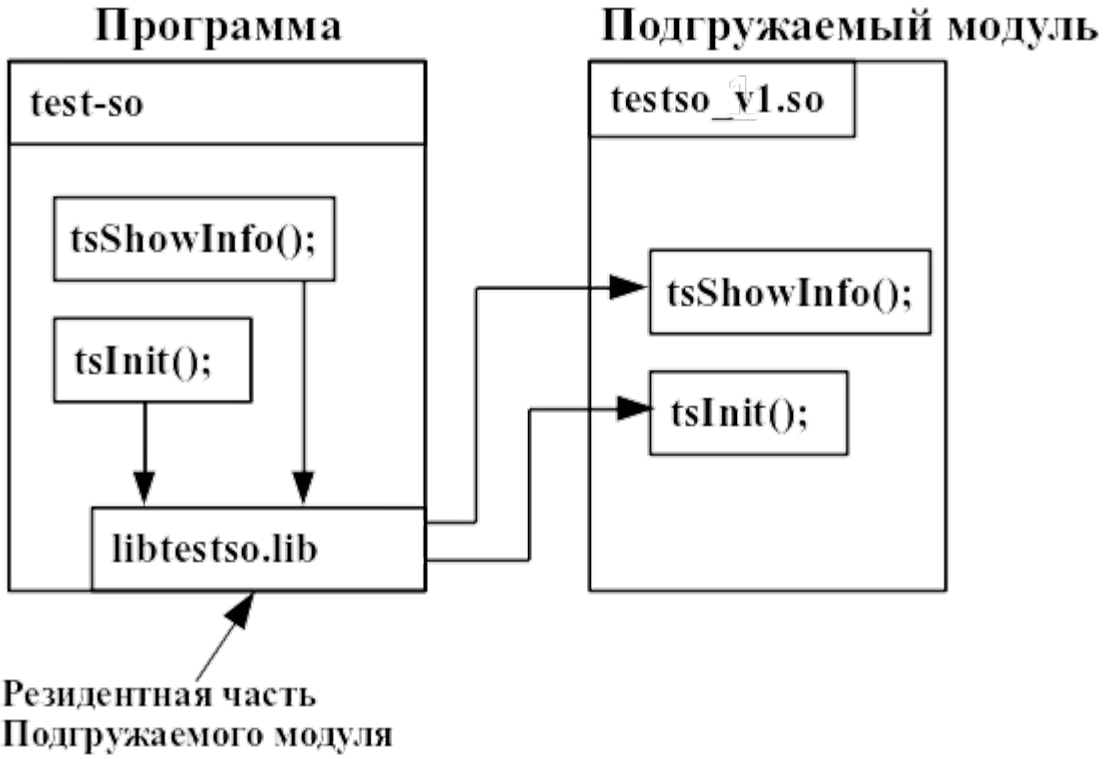

Рис. 3: «Прозрачный» вызов функций подгружаемого модуля из основной программы.

Выход нашелся довольно быстро: для каждой функции, которая объявлена как интерфейс в специальном файле, компилируется коротенький код перехода на специальную процедуру вызова подпрограммы модуля. Таким образом библиотека, называемая «интерфейсная резидентная часть» содержит псевдо-функции всех функций модуля и процедуру перехода от псевдофункции к реальной функции подгружаемого модуля.

Рассмотрим, например, как происходит вызов функции **tsShowInfo()**. Во время сборки подгружаемого модуля, генерируется функция **tsShowInfo()**, включаемая в библиотеку **libtestso.lib**. Данная «функция» содержит всего две команды (код на ассемблере, генерируется автоматически):

```
        ;// +22 — смещение функции от начала подгружаемого модуля
        .globl _testso_hl_jumper
        .globl _tsShowInfo
_tsShowInfo:
        ld      hl,#22 — смещение функции от начала подгружаемого модуля
        jp      _testso_hl_jumper ;// переход на реальную функцию модуля
```

То есть каждая функция подгружаемого модуля занимает дополнительные 6 байт в основной программе. Если функция не вызывается вообще, то её код в основную программу никогда не компилируется.

То есть, программа «думает», что у неё есть функция **tsShowInfo()**, хотя на самом деле она находится в другом модуле.

Вызов происходит так:

Программа вызывает «функцию» **tsShowInfo()**. Эта «функция» загружает смещение (в нашем примере **22**) от начала модуля в регистровую пару **HL** и переходит на единую для всех функций данного модуля процедуру ***_testso_hl_jumper***. Данная процедура абсолютно незатейлива — она прибавляет адрес загрузки модуля к регистровой паре **HL** и осуществляет

переход по вычисленному адресу — к реальной функции **tsShowInfo()**, находящейся внутри подгружаемого модуля.

Сразу оговорюсь для тех, кто будет смотреть код: там предусмотрен флаг для работы со страницами памяти, но это ещё не реализовано.

Каждая интерфейсная резидентная часть имеет свой заголовок размером 8 байт + имя модуля. Структура заголовка такая:

```
typedef struct {
        void*       next_link;    // +00 адрес следующего заголовка
        void*       load_adr;     // +02 адрес загрузки модуля
                                  //       load_adr ==00 — модуль не загружен
        uint16_t    page;         // +04 номер страницы модуля (не использ. в v1)
        uint16_t    flags;        // +08 флаги
        const char  name[];       // +0A название модуля (ASCIIZ)
} shared_link;
```

Все заголовки-описатели резидентных частей модуля хранятся в памяти программы друг за другом, поэтому можно найти любой из них, зная имя модуля. Это используется при настройке модуля в библиотеке **libso**.

Для настройки резидентной части модуля используется функция **void* tune_shared_obj(soHeader* lib)** библиотеки **libso**. На вход данной функции передаётся адрес загрузки **уже привязанного к адресам** модуля. После настройки модуля, функция **tune_shared_obj()** возвращает адрес заголовка настроенного модуля или **NULL**, если резидентной части такого модуля не найдено.

Порядок настройки модуля при использовании модели «прозрачного» вызова функции

Подведем краткий итог. Чтобы использовать программу с подгружаемыми модулями и моделью «прозрачного» вызова функций, необходимо:

- Создать (см. ниже) подгружаемый модуль (**xxx.so**) и его резидентную часть (**xxx.lib**).
- При сборке основной программы статически линковать программу с резидентной частью (**xxx.lib**).
- Загрузить программу и модуль (**xxx.so**) в память, причем модуль должен быть загружен по адресу, кратному **256**.
- В программе перед использованием модуля, необходимо настроить его на адрес загрузки функцией **soReloc()**.
- Настроить резидентную часть модуля функцией **tune_shared_obj()**.
- Все. Теперь функции модуля прозрачно вызываются.

Создаем подгружаемый модуль и тестовую программу

В дереве каталогов SDCC-NOINIT подгружаемый модуль находится в **libsrc/libtestso**, а тестовая программа для него — в **apps/test-so**. Используем модель «прозрачного вызова» функций.

Итак, вы решили сделать перемещаемый программный модуль, например библиотеку.

Сначала вы пишите её обычным образом. То есть один или несколько **.c** и .**h** файлов. В нашем примере — это файлы **testso.h, testso_common.h, testso_common.c, testso_v1.c, testso_v2.c**.

Файл **testso_common.c** содержит общие для всех версий библиотеки функции, процедуры, переменные.

Файлы **testso_v1.c** и **testso_v2.c** содержат разные версии нашего модуля.

Файл **testso.h** — заголовочный файл с интерфейсом модуля. Он содержит прототипы тех функций, которые доступны из ***вызывающей*** программы. В нашем примере это функции **tsInit()** и **tsShowInfo()**. Это самый обычный заголовочный файл, но в файле **build.mk** данный файл указан как интерфейсный. Это значит, что он анализируется и для каждой, описанной в нем функции, создается «псевдо-функция» в резидентной части модуля.

Теперь лезем в файл под названием **build.mk** и смотрим, что у нас там интересного.

Итак, прежде всего — внутреннее имя библиотеки **INTLIBNAME**. Это то имя, которое будет внесено в заголовок модуля.

INTLIBNAME="testso"

Каждая библиотека имеет свое внутреннее имя, которое позволяет её привязывать к резидентной части во время инициализации модуля.

Имя резидентной части **LIBNAME**. Именно эта библиотека линкуется с программой и обеспечивает выполнение функций подгружаемого модуля, как «родных». Сюда линкуются псевдо-функции, считанные из файла **testso.h**.

LIBNAME=libtestso.lib

Интерфейс модуля **IFACE_H**. В этом файле описываются те функции, которые доступны в вызывающей программе. В нашем примере это функции **tsInit()** и **tsShowInfo()**.

IFACE_H=testso.h

Имя файла с кодом инициализации модуля **CRT0PATH**. Пока что есть только один вариант, поэтому просто не меняйте его. Файл **crt0shared.asm** создаётся автоматически скриптами синтеза кода.

CRT0PATH=crt0shared.asm

Головные файлы модулей **MMSOLIB**. В нашем примере имеются две версии модуля, имеющие, разумеется, идентичные интерфейсы.

MMSOLIB=testso_v1 testso_v2

Общая часть для всех версий модуля **OBJ**. Разумеется, может быть указано несколько файлов.

OBJ+=testso_common.rel

Статические библиотеки, линкуемые в модуль **LIBS**. Заметьте, что эти библиотеки ***целиком*** линкуются в модуль. Так что осторожнее с такими вещами — объем может быть очень большой. В нашем примере в **каждый модуль** слинкован своё **printf()**, свой шрифт и свой модуль **conio**. Оцените объем — каждый модуль почти 4Кбайта! Для примера не важно, но в реальности так делать нельзя:)

LIBS=libz80 libconio

После сборки проекта получим два подгружаемых перемещаемых модуля — **testso_v1.so** и **testso_v2.so** и библиотеку, содержащую резидентную часть — **libtestso.lib**.

Программа, в которой мы хотим использовать подгружаемые модули должна быть слинкована с резидентной частью подгружаемых модулей, в нашем примере — **libtestso.lib**.

Рассмотрим, что происходит в программе **test-so**. Напомню, что к моменту запуска программы (она у нас загружена по адресу **0x6000**) в память загружены также два подгружаемых модуля: **testso_v1.so** по адресу **0x8000** и **testso_v2.so** по адресу **0x9000**.

Итак, программа и модули в памяти. Что происходит дальше?

Сначала происходит инициализация первого модуля с помощью функции **loadLib(so_v1_load_addr)**. Эта функция принимает на вход адрес загруженного модуля и делает две вещи:

- вызывает функцию **soReloc()**, которая привязывает модуль к адресам.
- вызывает функцию **tune_shared_obj()**, которая ищет по имени модуля его резидентную часть и связывает интерфейс с подгружаемым модулем.

После того, как модуль настроен на адреса и привязан к резидентной части, его функции можно вызывать точно так же как и любые другие:

```
tsInit();
tsShowInfo();
```

Просто, не правда ли?:) Но это ещё не все! У нас подгружены две версии одного модуля. Делаем **loadLib(so_v2_load_addr)** и получаем, что функции **tsInit()** и **tsShowInfo()** привязаны теперь ко второму модулю! Вывод на экран подтверждает это!

Более того, если есть необходимость, всегда можно вновь вернуться к работе с первым модулем, вызовом функции **loadLib(so_v1_load_addr)**. И так можно переключаться сколько угодно раз, пока модули находятся в памяти (рис.4).

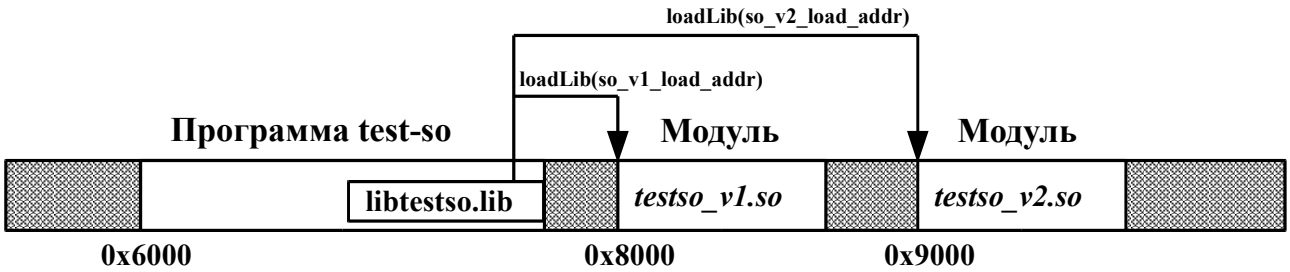

Рис. 4 Настройка резидентной части модуля на заданный модуль и переключение между модулями

Ссылки

Исходные коды, как всегда, тут: https://github.com/salextpuru/sdcc-noinit

Документация тут: https://github.com/salextpuru/sdcc-noinit/tree/master/doc
Тестовая программа **test-so** тут:
https://github.com/salextpuru/sdcc-noinit/tree/master/apps/test-so
Тестовый модуль **testso** тут:
https://github.com/salextpuru/sdcc-noinit/tree/master/libsrc/libtestso

Заключение или планы на будущее

Хорошие слова, которые не раз подтверждались опытом - «хочешь рассмешить бога, расскажи ему о своих планах». Так что планов не оглашаю, но намерения у меня такие:

- доделать работу с модулями, загружаемыми в разные страницы (фактически нужен менеджер памяти);
- доделать линковку по имени функций;
- доделать нормальный «непрозрачный» интерфейс, удобный для драйверов и плагинов.

Пока все.

Всегда рад ответить на ваши вопросы на форуме **http://zx-pk.ru/forum.php**. Если вопрос простой и мелкий — можно в личку. Если глобальный и большой — то создавайте тему.

Алексей, aka SfS