ZX SDCC NOINIT

Система сборки проектов для Z80 на базе пакета SDCC

Оглавление

Нет неразрешимых проблем, есть неприятные решения.

Предупреждение

Автор ни в коем случае не считает свое мнение единственно правильным и может его изменить или дополнить под действием разумных аргументов.

Также я не претендую на истину в последней инстанции, не считаю себя великим разработчиком систем как для Z80, так и систем вообще.

Посему прошу простить и понять мои ошибки, неточности и прочие места, которые вам придутся не по вкусу.

Если же вам есть, что сказать — то можно всегда написать мне об этом.

SfS, автор сего труда.

Введение

В далеких 90х практически единственным достойным средством разработки для ZX были ассемблеры. Их была масса. Они плодились и размножались.

На тех, кто пытался писать на C или Pascal — большинство кодеров смотрело как на детоубийц-некрофилов. Единственно, что не вызывало у кодеров тотальной брезгливости из ЯВУ (язык высокого уровня) для ZX — это был встроенный бейсик. И то с оговорками.

Но время шло. Катилось галактическое кольцо жизни по своей орбите, радиусом 27700 световых лет. За прошедшие с тех пор, приблизительно, 30 лет изменилось многое. Появились и канули в лету малиновые пиджаки и пейджеры. Расцвели и завяли радиобарахолки. Микросхемы серий 1533, 1554, 580 и других из дефицита превратились в бросовый товар, который можно получить почти бесплатно. Да и мы сами изменились. Из стройных вьюношей превратились в не очень стройных мужчин. Обзавелись женами, детьми, любовницами, котами, собаками и рыбками. Но одно осталось у нас на всю жизнь — любовь к железкам, собранным своими руками, программам, написанным своей головой, музыкой и графикой, которая выжимает из 8 бит все, что только можно. Посему мы продолжаем пытаться изобретать что-то свое. Просто для удовольствия.

На фоне всех изменений пора бы измениться и взглядам на ЯВУ для ZX. Сегодня, наверное, единственный свободный кросс-компилятор для ZX — это **SDCC**.

Так как это не майнстрим и разработкой его занимается совсем немного народа — то «особенностей», осложняющих жизнь в нем тьма тьмущая. Но, как говорится, «за неимением гербовой, будем писать на обычной».

Множество не очень удачных попыток использования **sdcc** привело меня к мысли, что без создания автоматизированной среды сборки, содержащей кучу скриптов, купирующих родовые недостатки **SDCC**, жизнь продолжаться не может.

Несколько вариантов различного подхода привели меня к созданию того, что называется в данный момент ***SDCC-NOINIT***. Не спрашивайте, почему именно так. Сам не знаю.

1. Лирическое отступление

Каждый разработчик чего-либо, будь то ПО или космические корабли — что-то терял. Даже такие «типа серьёзные» организации, как НАСА теряли немало: исходные коды программ, видеофайлы полётов на Луну и Марс и прочие вещи, которые, казалось бы, потерять невозможно. Что уж говорить о нас, простых смертных?

Какое количество исходных кодов, схем, карт, идей потеряно в области ретро-компьютеров вообще и ZX в частности — и подумать страшно.

Я так же грешен. Бурное студенчество и последующая жизнь, связанная с переездами, ремонтами квартир и прочей бытовой шелухой привела к потере множества дискет, компов, журналов и прочего и прочего.

Когда жизнь немного устаканилась, я вновь почувствовал желание делать что-то для души. Но оказалось, что все потеряно… Ну кроме знаний в голове.

Так что, пришлось начинать спектрум-жизнь сначала, благо в интернете есть практически вся информация, куча сайтов и прочего. Чтобы не повторять по возможности своих ошибок и потерь, я решил, что необходимо сделать систему сборки проектов для ZX, в которой мне легко и просто будет что-то создавать.

Что же такое SDCC-NOINIT?

Итак, **SDCC-NOINIT** — это система сборки программ для ZX-Spectrum. Основана она на пакете **SDCC** (Small Device C Compiler).

Основные языки написания программ — **C** и **Assembler Z80**.

Автоматизация сборки ПО обеспечивается с помощью утилиты GNU Make и **shell**-скриптов.

Конфигурация библиотек, программ, утилит и всего что есть в **SDCC-NOINIT** осуществляется с помощью текстовых конфигурационных файлов.

Основные свойства — простота добавления, удаления и конфигурации программ и библиотек; возможность иметь индивидуальный сборочный скрипт для каждого из компонентов ПО; возможность иметь индивидуальную конфигурацию для каждого из компонентов ПО; возможность неограниченного расширения функциональных возможностей внешними программами и скриптами.

Кому нужна SDCC-NOINIT?

Как минимум — она нужна мне. Или тому, кому неинтересно писать в 100500й раз с нуля вывод символа на экран, простого спрайта, музыкальный плеер pt3 и stc или ещё какой-то велосипеж. Многое в **SDCC-NOINIT** уже есть. Пусть не идеальное, не самое быстрое — но, есть! Поэтому тот, кто хочет проверить свою идею — может взять **SDCC-NOINIT**, быстренько накидать свою программу (примеры там есть) и заниматься проверкой своей идеи, а не добыванием руды для изготовления первого топора. И, если его идея, например, новой игры, ему понравится — то тогда он может потихоньку оптимизировать свои процедуры, переписывая их на ассемблере, если это надо или разработав новую библиотеку. Если кто-то будет добавлять свои библиотеки или программы — я буду только рад. Так что берите и смотрите: https://github.com/salextpuru/sdcc-noinit/.

2. Что нам понадобится и где это найти

Современный мир дает много возможностей. Лет 30 назад никто и не представить не мог, что можно будет находясь в лесу, купить билет на самолёт, вылетающий через полгода. А теперь этим никого не удивить.

То же и со средствами разработки. Обилие программ, утилит, конвертеров и прочих средств настолько велико, что трудно найти нужное.

Поскольку я не маньяк и писать все с нуля не хочу, не могу и не буду — то я по максимуму пользуюсь тем, что уже написано благородными людьми и отдано публике.

Поэтому я предваряю сей труд описанием того, какие именно программы и продукты нам понадобятся для работы с SDCC-NOINIT.

Итак, **прежде всего** — это утилиты такие как **bash** и **make**. Если вы счастливый обладатель Linux — то у вас проблем с этим нет. Но если вы несчастный и потерянный заложник Windows — то вам необходимо поставить CygWin, который позволит вам пользоваться юникс-утилитами в виндовс. Взять его можно на официальном сайте: https://cygwin.com/install.html.

Второе, что нам понадобиться — это компилятор GCC. И так же — для Linux он обычно есть в системе, а для Windows — в CygWin.

Третье — это набор утилит, который вы соберете с помощью компилятора GCC. Это специфичные спектрум-утилиты. Для удобства я собрал их в один «типа-пакет» **speccy-toochain**, который можно взять с GitHub и откомпилировать: https://github.com/salextpuru/speccy-toochain.

Четвертое, что нам необходимо — это компилятор SDCC. Тут все просто: идем на его сайт http://sdcc.sourceforge.net/.

Далее - или качаем бинарную сборку https://sourceforge.net/projects/sdcc/files/ или исходники http://sdcc.sourceforge.net/snap.php#Source и собираем сами.

Ну и, наконец, **пятое**, самое последнее — это собственно SDCC-NOINIT. Находится она все на том же GitHube, как раз рядышком с коллекцией исходников спектрум-утилит: https://github.com/salextpuru/sdcc-noinit.

Скачали? Поставили ? Зайдите в корень SDCC-NOINIT и наберите:

 # make

Если у вас все верно (все утилиты есть, пути к ним прописаны и т. п.), то после сборки в каталоге bin появятся программки и плагины WC.

Ну а если что-то не пошло — то думайте, чего вам не хватает и исправляйте. Какой интерес жить, не решая проблем ? :)

3. Как транслируется программа на С

Чтобы понять почему что-то сделано так, а не иначе — надо разобраться как оно работает. Так что начнем издалека. Из глубины времен до нас дошел единственно правильный путь работы трансляторов ЯВУ. Тот самый путь, коего придерживаются разработчики с прошлого тысячелетия.

Пакет программ SDCC в этом смысле не отличается от других подобных пакетов программ — будь то GCC или IAR. Схему сборки любого проекта можно представить приблизительно в таком виде, как на рис.1.

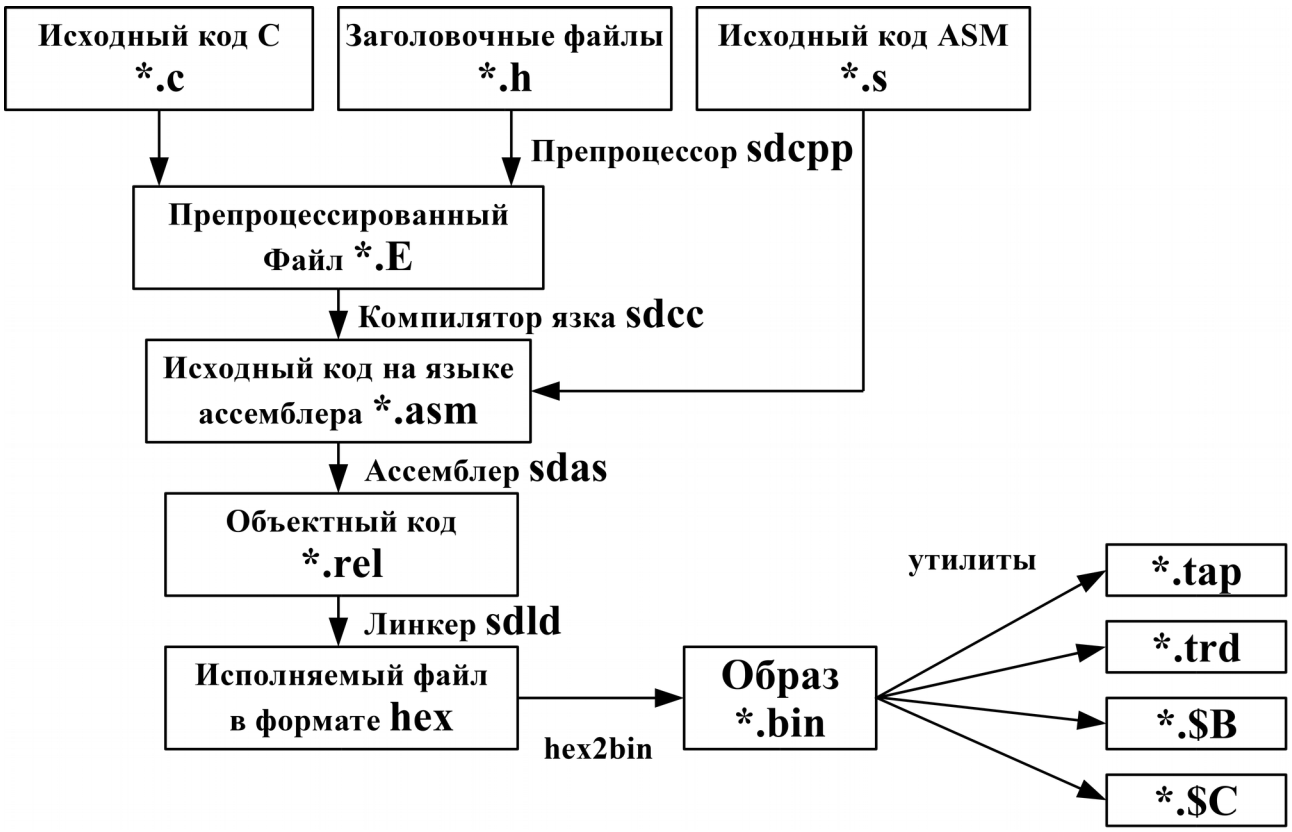

Рис. 1: Трансляция программы на языке С пакетом SDCC - от исходного кода до исполняемых файлов

Рассмотрим алгоритм, что видим мы на рис.1. Это основа основ. SDCC-NOINIT работает именно по этому алгоритму, дополняя его, но по сути не изменяя.

Рассмотрение алгоритма сделаем по возможности кратким, чтобы не погрязнуть в излишних технических подробностях. О них мы поговорим позже.

Итак, что у нас есть в начале? А в начале у нас есть слово. И слово это — на языках С и ассемблера, в файлах, что обычно носят расширения ***.c**, ***.h** и ***.s**.

Некоторые думают, что компилятор — это программа, которая из этих файлов делает бинарь. Они жестоко ошибаются! Чтобы получить бинарь — надо потеть нескольким программам по очереди, а то и не по одному разу.

Итак, что же происходит с файлами, что мы набрали в текстовом редакторе? Они передаются на программу-препроцессор, в нашем случае называемую **sdcpp**.

Что делает эта чудесная программа? Как ни удивительно, но она выполняет так называемые «директивы препроцессора».

Не пугайтесь этих загадочных слов. Что такое директива препроцессора? Это команда, начинающаяся со знака «решетка» (#), например #define, #if, #ifdef, #ifndef, #endif, #include.

Так вот — компилятор эти команды не понимает. Компилятору надо подать текст, очищенный от этих решеток. Иначе его может стошнить прямо на ваш дисплей. Некрасиво получится.

А вот как раз препроцессор никакие другие команды в тексте, кроме как начинающиеся с #, не интересуют. Поэтому директивы препроцессора с успехом можно использовать как в файлах на C, так и в файлах на языке ассемблера.

3.1. Препроцессор

На этой стадии, исходные тексты преобразуются в другие тексты в соответствии с директивами препроцессора. Рассмотрим очень коротко наиболее используемые директивы.

Самая важная, нужная, полезная и красивая из директив — это несомненно **#define**. Она позволяет заменять одно слово на другое.
Например, если у вас было написано:

```
// … Что-то до
#define WIN_W  16
#define WIN_H  12

// … Что-то после

clw(WIN_W,  WIN_H);
```

то после препроцессора все директивы имение его исчезнут, а на место определенных символов встанут их значения:

```
// … Что-то до

// … Что-то после
clw(16, 12);
```

Директивой **#define** можно определять и макросы.
Например, код

```
#define printat(x,y,s) \
        at(x,y); \
        printf(s);

printat(10,12,"String at 10,12");
```

после препроцессора даст

```
at(10,12);
printf("String at 10,12");
```

Определяя макросы — будьте внимательны. Если определять макросы через макросы можно такого наворотить… В общем — чтобы не по принципу «заставь дурака богу молиться».

Другая очень широко используемая и всем известная директива препроцессора — это **#include**. Она просто напросто вставляет указанный текстовый файл в указанное место.

Обычно это файлы с расширением ***.h**, в которых определены константы и макросы. Но вообще можно вставлять любые файлы. Если знаете что делаете, конечно.

С директивой **#include** надо помнить, что если имя файла взято в кавычки (**#include "file.h"**), то этот файл должен лежать в том же каталоге, что и компилируемый файл из которого происходит вызов директивы **#include**. Если же имя файла взято в угловые скобки (**#include <file.h>**), то препроцессор будет искать этот файл по всем, заданные опцией -L путям.

Ну и, наконец, если я не рассмотрю условные директивы — то покрою себя несмываемым позором. Условные директивы препроцессора — это директивы, которые могут вставлять или не вставлять в текст код, в зависимости от заданных условий.

Все они начинаются с #if (#if, #ifdef, #ifndef), и имеют, условно, такие формы: «if-endif», «if-else-endif», «if-elif-endif», «if-elif-else-endif».

Несколько образцов условных директив:

```
#if <условие>
        // код, включаемый если условие истинно
#endif

#ifdef SYMBOL
        // код, включаемый если символ SYMBOL определен
#endif

#ifndef SYMBOL
        // код, включаемый если символ SYMBOL не определен
#endif

#if <условие1>
        // код, включаемый если условие1 истинно
#elif <условие2>
        // код, включаемый если условие2 истинно,  условие1 - ложно
#elif <условие3>
        // код, включаемый если условие3 истинно,  условие1 и условие2 - ложны
#else
        // код, включаемый если все условия ложны
#endif
```

На этом закончим про препроцессор и вернемся к нему тогда, когда в этом будет необходимость. Если же кому хочется больше — то прошу в интернет по ссылке https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B5%D0%BF%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80_%D0%A1%D0%B8#.D0.92.D1.81.D1.82.D0.B0.D0.B2.D0.BA.D0.B0_.D1.84.D0.B0.D0.B9.D0.BB.D0.BE.D0.B2_.28.23include.29.

Пока запомним лишь, что препроцессор — чистит файл от решеток и даёт компилятору «чистый текст» и перейдем к следующей стадии, которая называется «компиляция».

Заметим, что файлы, написанные на языке ассемблера (***.s**) сразу после препроцессора становятся ***.asm** файлами, а файлы, написанные на C - ***.E** файлами.

3.2. Компиляция

Компиляция — это перевод текста программы, написанной на ЯВУ и обработанной препроцессором в текст на языке ассемблера. Собственно этим и занимается часть транслятора, именуемая компилятором. Многие часто путают трансляцию и компиляцию. Не надо опускаться так низко. Трансляция — «от исходного текста до бинаря». А компиляция «от препроцессора до ассемблера».

Исходя из определения компиляции становится понятно, что файлы на языке ассемблера компилировать не надо. Ибо они и так уже на ассемблере.

Итак на вход компилятора подаются файлы на языке С с расширением ***.E**, а на выходе получаются файлы на языке ассемблера с расширением ***.asm**.

Поскольку фронт-енд транслятора **sdcc** сам вызывает препроцессор и компилятор, то файлов с расширением ***.E** мы как правило не видим. Да они нас обычно и не интересуют.

После стадии компиляции различие между исходниками на С (***.c)** и ассемблере (***.s**) исчезают — все они становятся безликими файлами ***.asm**, хоть и различными путями.

И это прекрасно, потому что следующая стадия у всех одна — ассемблирование.

Компилятор пакета sdcc так и называется — **sdcc**, что вносит некоторую путаницу. Но делать нечего — компилятор объединен с фронт-ендом.

3.3. Ассемблирование

Ассемблирование — это процесс преобразования файла с текстом на языке ассемблера в объектный файл. В нашем случае объектные файлы имею расширение ***.rel**. Программа, которое совершает описанные зверства над ассемблеными файлами носит название **sdas**.

Что же такое «объектный файл»? Зачем он нужен? А нужен он, чтобы собрать из кусочков всю программу и привязать ее к нужным абсолютным адресам памяти.

Каждый объектный файл хранит три основные сущности: глобальные символы, откомпилированный объектный код и таблицу перемещения.

Глобальные символы — это символы, которые доступны из всех объектных файлов проекта. В противоположность им есть локальные символы, которые доступны только в пределах одного файла (не путайте их с локальными переменными, которые доступны только в пределах скобок {}!).

Откомпилированный объектный код — это исполняемый двоичный код с одним нюансом — вместо абсолютных адресов в нем указаны относительные. А для преобразования относительных адресов в абсолютные как раз и используется таблица перемещения.

На этом прервем, но не завершим пока рассказ об ассемблировании и перейдем к завершающей стадии сборки проекта — линковке.

3.4. Линковка

Линковка — это по сути процесс размещения объектных файлов по заданным абсолютным адресам.

Программе линкеру **sdld** передается такой ключевой параметр как адрес начала бинарного кода. И далее линкер размещает все объектные файлы, начиная с этого адреса. В процессе размещения каждого файла, линкер рассчитывает абсолютные адреса всех его символов и подставляет их на соответствующие места в файле.

Таким образом, в результате линковки получается некий «образ памяти», начиная с указанного адреса. Если полученный образ поместить в память с адреса начала бинарного кода и передать ему управление — то наша программа начнет выполняться.

В SDCC есть нюанс — линкер генерирует выходные файлы в текстовом формате intel hex. Для запуска же нам нужен двоичный код. Но утилита hex2bin без труда позволяет нам решить эту проблему.

3.5. Преобразования

Поскольку, бинарный файл не удобен тем, что в нем нет информации с какого адреса его грузить, да и вообще никакой дополнительной информации, то его преобразуют обычно в один из понимаемых ZX форматов - ***.tap**, ***.trd** или **Hobeta**.

Сам бинарь при этом не изменяется, но к нему добавляется заголовок или даже бейсик-загрузчик.

3.6. Небольшой итог

В итоге всех моих словестных изысков надо усвоить следующее:

- На вход транслятора подаются текстовые файлы на языках C и Ассемблера.

- Файлы прогоняются через препроцессор, который выполняет только свои директивы, преобразуя остальной код в соответствии с ними:
  ***.c, *.h → *.E**
  ***.s → *.asm**

- Препроцессированные файлы на языке C передаются компилятору; на выходе получается код ассемблера:
  ***.E → *.asm**

- Ассемблерные файлы передаются ассемблеру; на выходе получаются файлы с объектным кодом:
  ***.asm → *.rel**

- Файлы с объектным кодом линкуются в исполняемый файл в формате Intel Hex:
  ***.rel → <имя программы>.hex**

- Исполняемый файл в формате Intel Hex преобразуется в бинарный образ программы:
  <имя программы>.hex → <имя программы>.bin

- Если необходимо, то специальными утилитами создается загрузочный файл нужного формата:
  <имя программы>.bin → <имя программы>.tap
  <имя программы>.bin → <имя программы>.trd
  <имя программы>.bin → <имя программы>.$B
  <имя программы>.bin → <имя программы>.$C

Пока мы не рассматривали такие вещи, как стартовый код, библиотеки, излишние секции и многое другое.

4. Особенности SDCC

Как бы мы чего ни стандартизировали, каждой отдельной реализации этого «чего» будут свойственны свои уникальные особенности.

А если это «чего» делается группой энтузиастов из трёх с половиной человек — то многие вещи там будут уникальные и особенные. И все потому, что в порыве творческого энтузиазма — людям не до стандартов. А потом переделывать неохота.

Все это в полной мере относится к SDCC.

В настоящем разделе мы рассмотрим особенности транслятора SDCC и его ограничения.

4.1. Линковка и секции линкера

В предыдущих разделах говорилось, что линковка — это по сути процесс размещения объектных файлов по заданным абсолютным адресам. Но как же линкер знает где располагаются данные, код, стек и так далее?

А знает он это из первого встреченного им *.rel файла, в котором указан порядок следования секций линкера.

Что такое «секция линкера»? А это адресное пространство, имеющее свое уникальное имя и выделенное для определенных нужд.

Поясню на примере. Все мы знаем, что бейсик в ZX зашит в ПЗУ. С точки зрения программиста это значит, что по адресам процессора 0x0000 — 0x3FFF находится область памяти из которой можно читать данные, из неё можно исполнять код. Но в неё нельзя ничего записать.

Предположим, что мы пишем свой бейсик для спектрума. Как нам сказать линкеру, что в область 0x0000 — 0x3FFF можно писать только код и константы; в адреса 0x4000 — 0x5AFF писать и читать ничего нельзя (там экран); а в адресах 0x5B00-0xFFFF можно творить всё, что угодно?

Для разрешения этого вопроса были придуманы так называемые «секции линкера», каждая из которых имеет свое имя.

При запуске линкера, мы вольны указать абсолютный адрес начала любой секции по нашему разумению. Если же адрес секции не указан, то они располагаются одна за другой в том порядке, как был встречены впервые в объектных файлах.

Существует несколько типов секций, которые автоматически генерируются компилятором языка С. В SDCC такие секции жестко «прибиты гвоздями» и их названия изменить нельзя (может когда и будет можно?).

Что это за секции? Самые главные — это **_CODE**, **_DATA**, **_BSS**, **_INITIALIZED** и **_INITIALIZER**. Именно так, начиная с символа подчеркивания.

Секция **_CODE**. Здесь размещается исполняемый код и константы.

Секция **_DATA**. Здесь размещаются не инициализированные переменные. Для инициализированных переменных есть отдельная секция **_INITIALIZED**. О секции **_INITIALIZER** поговорим чуть позже.

И наконец, секция **_BSS** — это секция стека. Она не включается в размер исполняемого файла вообще.

Для определения секции в языке ассемблера sdas предназначено ключевое слово **.area**. Рассмотрим, какие данные в каких секциях размещает компилятор SDCC.

4.2. Секции, создаваемые компилятором SDCC

В табл.1 приведен список секций с краткими комментариями. Более подробно назначение секций разобрано ниже.

Таблица 1: Секции, используемые SDCC

№	Секция	Назначение	Примечание
1	_CODE	Код программы и константы	
2	_DATA	Неинициализированные переменные	Инициализируются нулем при старте программы на C
3	_INITIALIZER	Данные для инициализации секции _INITIALIZED	В SDCC-NOINIT — секция всегда расположена последней и удаляется из бинарника.
4	_INITIALIZED	Инициализированные переменные	При старте программы сюда копируются данные из секции _INITIALIZER. В SDCC-NOINIT данные копируются не во время выполнения, а специальным скриптом (см. ниже).
5	_DABS (ABS)	Абсолютные адреса (данные)	Не используются в SDCC-NOINIT
6	_CABS (ABS)	Абсолютные адреса (код)	Не используются в SDCC-NOINIT
7	_HOME	Так и не понял для чего она	Не используются в SDCC-NOINIT
8	_GSINIT	Начало кода инициализации во время выполнения программы	
9	_GSFINAL	Окончание кода инициализации во время выполнения программы	
10	_HEAP	Куча динамической памяти	Компилятором не генерируется. Дополнительная
11	_BSS	Стек	Компилятором не генерируется. Дополнительная

4.3. Размещение машинного кода

Секция _CODE.

Машинный код размещается компилятором в секции _CODE и более ни в каких других. Создатели SDCC предполагают, что из секции _CODE можно читать и выполнять код, но не записывать в неё.

На самом деле, так как никакой защиты от записи не предусмотрено, записывать тоже можно. Но компилятор не размещает в этой секции переменных.

4.4. Размещение переменных и констант

Секции _CODE, _DATA, _INITIALIZED, _INITIALIZER.

Данные в языке С довольно можно поделить на два типа — переменные и константы. Переменные могут произвольным образом изменяться, а константы доступны только для чтения.

Кроме того, данные могут быть как инициализированными при объявлении, так и не инициализированными.

Не забываем, что все не инициализированные при создании статические переменные в языке С должны при старте программы заполняться нулем.

В зависимости от того, как мы объявляем ту или иную переменную, описываем те или иные данные — будет генерироваться различный ассемблерный код.

Рассмотрим пример. Есть у нас три похожих по виду, но очень разных по сути определения переменных и констант:

Определение 1: **const int var_const=2;**
Определение 2: **int var_noinit;**
Определение 3: **int var_init=7;**

Что же генерирует компилятор на каждое из них? Давайте посмотрим. Обратите внимание, что на ассемблере к имени переменной, функции или другого объекта на С добавляется символ подчеркивания. Это общее правило, которое не следует забывать при связывании кода на С и ассемблере.

Определение 1. Константа.

C:

```
const int var_const=2;
```

ASM:

```
.area _CODE
_var_const:
        .dw #0x0002
```

Константа размещена в секции кода. Логично, так как её можно только читать. И даже если код в ПЗУ — ничего страшного, писать то в константу нельзя.

Определение 2. Не инициализированная при создании переменная.

C:

```
int var_noinit;
```

ASM:

```
.area _DATA
_var_noinit::
        .ds 2
```

Переменная размещена в секции данных. Когда происходит запуск программы на С, вся область **_DATA** инициализируется константой 0. Таким образом, для переменной

_var_noinit выделяется место, а инициализация происходит во время старта программы. Все прекрасно.

Определение 3. Инициализированная при создании переменная.

C:

```
int var_init=7;
```

ASM:

```
.area _INITIALIZED
_var_init::
        .ds 2

.area _INITIALIZER
__xinit__var_init:
        .dw #0x0007
```

А вот для инициализированных при создании переменных все крайне запутано. Каждая такая переменная располагается аж в двух секциях: **_INITIALIZED** - это секция, где переменная находится во время выполнения программы и **_INITIALIZER**, где хранится инициализирующее значение этой переменной.

Создатели SDCC, очевидно, предполагали, что код ВСЕГДА будет находится только в ПЗУ и никак иначе. Поэтому предполагается, что начальный код запуска программы на С будет копировть секцию **_INITIALIZER** в секцию **_INITIALIZED** и только потом запускать функцию **_main** (начало программы на С).

Само по себе ужасно не это, а то, то никоим образом нельзя сказать компилятору sdcc примерно следующее: «Милый мой компилятор! У меня вся программа будет в ОЗУ, так что инициализируй переменные сразу без копирования, как и константы». Ну например так:

```
.area _INITIALIZED
_var_init::
        .dw #0x0007
```

Увы. Пока такой функции в SDCC нет. Зато есть хитрые и злые кодеры вроде нас с вами. И выхода из ситуации два:

Выход первый. **Можно, но не нужно**.

Объявить все инициализированные при создании переменные как константы. В самом деле — раз у нас вся программа в ОЗУ, то какая разница — в секции кода или данных будет выделено место под переменную?

Недостатки такого подхода очевидны. Во-первых, придется все такие переменные вынести в отдельный файл и сделать к нему отдельный «обмынный» заголовочный файл:

Файл **var_init.c**:

```
const int var_init=7;
```

Файл **var_init.h**:

```
extern int var_init;
```

Почему так? Да чтобы компилятор не ругался, что мы пытаемся изменить константу. Компилируя файл **var_init.c**, компилятор будет думать, что **var_init —** это константа и выделит под неё памяить как под константу.

Компилируя другие файлы, куда подключен файл **var_init.h**, компилятор будет всецело уверен, что **var_init** — это переменная и будет спокойно в неё писать.

Мне такой подход не нравится. Ради переменных отдельные файлы.. Бррр..

Выход второй. **Правильный**.

Второй выход посложнее — написать утилиту, которая вырезает секцию **_INITIALIZER** и вставляет её на место секции **_INITIALIZED**. Именно такой подход и был осуществлен при написании утилиты **sdrmini**. На вход утилита принимает так называемый map-файл и бинарник программы. Далее, из файла вырезаются секции **_INITIALIZER** и **_INITIALIZED**. И, наконец, файл сшивается так, что секция **_INITIALIZER** встает на место секции **_INITIALIZED**. Ограничение такого метода — секция **_INITIALIZER** должна быть самая последняя в списке секций. Зато достоинства очевидны — не надо копировать данные. Да и объём бинаря сокращается на размер секции **_INITIALIZER**.

4.5. Секции абсолютной адресации

Секции _DABS (ABS) и _CABS (ABS).

По замыслу разработчиков, в данных секциях можно размещать данные и код, определяя абсолютные адреса директивой **.org**. Но практическое использование этих секций неудобно, потому что нет проверок перекрытия с другими секциями. Поэтому в SDCC-NOINIT данные секции не используются и вам не рекомендую с ними связываться.

4.6. Секции кода инициализации времени выполнения

Секции _GSINIT и _GSFINAL.

Некоторые конструкции языка С требуют выполнения кода инициализации до начала первой функции, то есть **main()**.

Компилятор помещает весь такой код в секцию **_GSINIT**. Секция **_GSFINAL**, следующая сразу за секцией **_GSINIT**, содержит единственную команду **ret**. При старте программы, стартовый код вызывает содержимое секции **_GSINIT**, как обычную подпрограмму.

Заметим, что после выполнения стартового кода, содержимое секций **_GSINIT**, **_GSFINAL** и **_INITIALIZER** становится не нужно. Поэтому в программе область памяти, занятую этими секциями можно использовать по своему усмотрению, например под стек.

4.7. Стартовый код crt0*.s

Выполнение программы начинается со стартового кода. В зависимости от того, что за программу, плагин или ещё что-то мы пишем — стартовый код может быть различным.

В SDCC-NOINIT предусмотрены различные файлы стартового кода, находящиеся в каталоге **libsrc/crt0**:

crt0.s - стартовый код обычной программы;
crt0wcplugin.s - стартовый код плагина WC.

Пользователь волен писать свой стартовый код, если ему нравится и хочется.

Файлы со стартовым кодом находятся в каталоге **libsrc/crt0** и называются **crt0*.s**. Вместо звездочки подставляем все, что нам угодно.

Какой бы ни был стартовый код, его функции примерно следующие:

а) инициализация указателя стека;
б) заполнение секции **_DATA** нулевым значением;
в) копирование секции **_INITIALIZER** в секцию**_INITIALIZED;**
г) вызов кода инициализации времени выполнения из секции **_GSINIT;**
д) вызов кода на языке С (функции main());
е) восстановление указателя стека;
ж) возврат в вызывающую программу.

Кроме этого, при линковке программы, файл **crt0*.rel** должен быть указан первым в списке объектных файлов, чтобы определить порядок следования секций. Иначе результат будет плачевным.

В SDCC-NOINIT проведена некоторая работа по оптимизации всех функций стартового кода. Поведение файла стартового кода регулируется для каждой программы файлом **config.mk**, находящимся в файле с программой.

Файл **config.mk —** файл конфигурации стартового кода. На его основе генерируется файл **autoconfig.asm**, который содержит определения, передаваемые в стартовый код.

Чтобы иметь доступ к символам, определённым в файле **config.mk**, в файл **crt0*.s** необходимо включить файл **autoconfig.asm**, написав в начале файла **crt0*.s** директиву:

```
.include /autoconfig.asm/
```

Рассмотрим, какие особенности имеет файл **crt0.s** SDCC-NOINIT:

а) инициализация указателя стека. Производится опционально. Определяется символом **CALLERSTACK** файла **config.mk**. Если **CALLERSTACK**=0, то использовать свой стек, а если **CALLERSTACK**=1 — то не трогать указатель стека, т. е. Использовать стек вызывающей программы. Это удобно;

б) заполнение секции **_DATA** нулевым значением. Так как снекция **_DATA** грузится целиком вместе с программой, то дополнительно забивать нулями её не нужно. Она уже обнулена во время компиляции;

в) копирование секции **_INITIALIZER** в секцию**_INITIALIZED**. Этот вопрос уже обсуждался ранее. Секции копируются скриптом **sdrmini** после компиляции и во время выполнения секция **_ INITIALIZER** уже не существует;

г) вызов кода инициализации времени выполнения из секции **_GSINIT**. Тут все по стандарту;

д) вызов кода на языке С (функции main());

е) восстановление указателя стека производится только если был использован свой собственный стек (**CALLERSTACK**=0);

ж) возврат в вызывающую программу. Тут все по стандарту.

Кроме того, предусмотрено управление прерываниями при старте программы. Если символ **DISTART=0** (файл **config.mk** данной программы), то при входе в программу прерывания запрещаются, а при выходе — разрешаются.

Подробнее о конфигурации программ, библиотек и прочего — ниже.

5. Ассемблер в С и С в Ассемблере

Известно, что на ZX без ассемблера никуда. И это хорошо. Но и без ЯВУ в 21м веке тоже не комфортоно.

Как же нам взять хорошее и от того и от другого? Для этого необходимо научиться ассемблерный код и код на С.

Этому и посвящена данная глава.

5.1. Два способа совмещать С и Ассемблер

В SDCC имеется два способа совмещать код на С и код на Ассемблере: раздельная компиляция и директивы встроенного ассемблера **__asm; - __endasm;**.

Вне зависимости от того, какой способ мы выберем (а можно и оба сразу) — все соглашения об именах, передаче параметров и прочие — одинаковы.

5.2. Соглашения об именах

Мы уже знаем, что компилятор преобразует код на языке С в код на языке ассемблера. Как же он называет различные переменные и функции? Так же как в С?

Рассмотрим простой пример:

Возьмем файл **example.c**:

```
int var_a;              // обычная переменная
static int var_b;       // локальная переменная (область видимости —
файл)

void function_a(){      // обычная функция
}
static void function_b(){     // локальная функция (область
видимости - файл)
}
```

Не будем пока углубляться в загадочную «область видимости», а откомпилируем этот файл и получим такой код на языке ассемблера **example.asm** (лишнее опущено):

```
.area _DATA
_var_a::
        .ds 2
_var_b:
        .ds 2

.area _CODE
_function_a::
        ret

_function_b:
        ret
```

Понимаете что произошло? Произошло очень простое и понятное. К каждому имени объекта (переменной, фугкции и проч.) компилятор добавил начальный символ подчеркивания.

Помните об этом всегда: если вы решили обратиться из ассемблера к переменной или функции, объявленной на С, то не забывайте добавить к имени знак подчеркивания. Так устроен SDCC и ничего тут не поделаешь. Зато удобно, когда все имена, начинающиеся с подчеркивания — это имена языка С. А с любой другой буквы — чисто ассемблерные имена.

Если вам надо из кода на С обратиться к переменной или функции, описанной на ассемблере — правило тоже работает. Только в языке С обязательно нужно указать, что у вас есть прототип функции или переменной.

Например имеется файл на ассемблере. Обратите внимание, что после имени стоит **два** двоеточия. Почему так — поговорим ниже.

```
.area _DATA
_MyAsmArray::
        .ds 10

.area _CODE
_MyAssemblerFunction::
        ret
```

Если в файле на С описать прототипы, то можно спокойно обращаться к этим переменным и функциям. Только надо учесть, что для вызова из С, эти переменные и функции должны начинаться с подчеркивания, а прототипы на С должны быть без подчеркивания. Пример кода на С:

```
// Прототипы

extern uint8_t MyAsmArray[];      // Массив из 10 байт
void MyAssemblerFunction();       // Функция

// Вызов (пример)
void main(){
        MyAsmArray[0] = 5;
        MyAssemblerFunction();
  }
```

Как видите, в преобразовании имен между С и ассемблером нет ничего сложного. Всего лишь один символ, и тот — подчеркивание. Займемся теперь загадочными двоеточиями. Почему в одном случае оно одно, а в другом их два? Где здесь разгадка?

А разгадка в том, что в языках С и Ассемблер есть такое понятие как «область видимости». И это очень важное и удобное понятие, если знать и уметь его использовать.

5.3. Локальный или глобальный?

Область видимости — это участок программы, где доступно то или иное имя переменной или функции.

Для языка Ассемблера **sdas** областей видимости всего две — в пределах одного файла (локальные переменные и функции) или в пределах всей программы (глобальные переменные и функции).

Для языка С областей видимости больше: есть переменные и функции видимые в пределах всей программы (глобальные); есть переменные и функции, видимые в пределах файла (локальные или статические); а есть и те, что видны только в пределах фигурных скобок (например в пределах функции) — локальные.

Здесь мы рассмотрим только две области видимости - в пределах файла и в пределах всей программы. Будем пользоваться терминологией из языка Ассембера, т. е. называть глобальной переменной или функцией ту, что видна везде и всем, а локальной переменной или функцией ту, что видна только в пределах текущего файла.

Чем же отличаются глобальные и локальные переменные? Все довольно просто.

Для переменных и функций, видимых в пределах всей программы, информация об именах переменных сохраняется в объектных файлах. Поэтому, к таким переменным и функциям можно обращаться из любого места программы, где описан прототип такой переменной или функции.

Для переменных и функций, видимых в пределах файла, никакой информации об именах в объектных файлах не сохраняется. Поэтому из других объектных файлов доступ к ним невозможен.

Вы можете иметь сколько угодно локальных переменных с одинаковыми именами, которые расположены в разных файлах. И это не доставит неудобств ни вам, ни компилятору.

5.3.1. Объявление глобальных переменных и функций

На языке С любая переменная или функция считается глобальной. Например, вот объявления глобальных переменных и функций:

```
int a;
void main(){}
```

На языке ассемблера все переменные и функции по умолчанию — напротив, локальные. Для того, чтобы объявить какую-то переменную или функцию как глобальную — надо воспользоваться одним из двух методов — ключевым словом **.globl** или двойным двоеточием после имени:

```
.globl  _MyAssemblerFunction

.area _CODE
_MyAssemblerFunction:
        ret

.area _DATA
_MyAsmArray::
        .ds 10
```

Функция **_MyAssemblerFunction** объявлена глобальной с помощью использования ключевого слова **.globl**. А переменная **_MyAsmArray** объявлена глобальной с помощью двойного двоеточия после имени. Эти объявления равнозначны.

5.3.2. Объявление локальных переменных и функций

На языке С область видимости, ограниченная одним файлом задаётся ключевым словом **static**. Например, вот объявления локальных переменных и функций, ограниченных по области видимости файлом:

```
static int local_a;
static void local_main(){}
```

На языке ассемблера для объявления локальных переменных и функций, ограниченных по области видимости файлом не надо прилагать никаких дополнительных усилий:

```
.area _CODE
_MyAssemblerFunctionLocal:
        ret

.area _DATA
_MyAsmArrayLocal:
        .ds 10
```

Хотя, надо отметить, что ключевое слово **.local** в ассемблере **sdas** все же существует. Дело в том, что в ассемблере **sdas** есть опция командной строки **-a**, которая включает режим «все символы по умолчанию — глобальные». В этом случае надо наоборот, указывать какие символы будут являться локальными. Никогда не используйте опцию **-a**! Иначе вас проклянут!

5.3.3. Раздельная компиляция

Раздельная компиляция подразумевает, что код на С и код на ассемблере находятся в разных файлах. Файлы компилируются (файл на С — с помощью **sdcc**, файл на ассемблере — с помощью **sdas**). В результате получаются несколько объектных ***.rel**-файлов, которые линкуются в программу. Раздельная компиляция удобна, когда ваш код на ассемблере — самостоятельный и может использоваться как в программе на С, так и отдельно.

5.3.4. Не выходя из С

Если ваши процедуры на ассемблере имеют небольшой объём и сильно интегрированы с кодом на С, то удобнее всего использовать встроенный ассемблер.

Как уже говорилось, имеются несколько директив SDCC, позволяющих встраивать ассемблерный код, прямо в файлы с исходными текстами на языке С.

И это очень удобно. Директивы, которые нам понадобятся — следующие:

__asm — обозначает начало ассемблерного кода;

__endasm — обозначает окончание ассемблерного кода;

__naked — (голая) обозначает, что процедура не имеет ***пролога*** и ***эпилога***.

Обычно, описание функции на встроенном ассемблере, выглядит так:

```
void    example_asm1()__naked{
__asm;
        ;// Тут любой код на ассемблере, метки и всё, что угодно.

        ret     ; Возврат из процедуры
__endasm;
}
```

Пока рассмотрим самый простой пример — функция не принимает и не возвращает никаких параметров.

Таким образом, при вызове нашей функции в другом месте программы

```
example_asm1();
```

просто выполнится вызов подпрограммы, примерно так (не забываем про конверсию имен!):

```
call _example_asm1
```

Тут вроде все понятно. Разберёмся теперь с загадочной голой директивой **__naked** и выясним, что такое ***пролог*** и ***эпилога*** функции.

Когда происходит компиляция функции на языке С, компилятор генерирует в начале каждой функции код, именуемый ***прологом***, а в конце — код выхода из функции, называемый ***эпилогом***. Что делает этот код? Ну, например, создаёт и уничтожает временные автоматические локальные переменные; запоминает и восстанавливает регистры; управляет стеком.

Когда мы пишем функцию полностью на встроенном ассемблере — нам все эти премудрости не только не нужны, но и вредны. Компилятор может вести себя по-разному и угадать, что он нагенерирует и как нам с эти работать — невозможно. Чтобы сказать компилятору: «хорош работать, не пиши тут никакого кода, я сам в этой процедуре всё напишу» и предназначена директива **__naked**.

Никто не запрещает нам написать часть функции на С, а часть на ассемблере. Но делайте так, только если очень хорошо знаете, что делаете и зачем. Иначе вы испортите, например, регистры или стек и будете долго удивляться волшебному поведению своей программы. Так что лучше взять за правило — или вся функция на С или вся функция на встроенном ассемблере. Но не коктейль.

В простейшем примере функция ничего не принимает и ничего не возвращает. К сожалению, в реальной жизни, так бывает редко. Потому в полный рост встаёт вопрос передачи параметров.

5.4. Передача параметров в функцию

Когда мы пишем на С функцию, например:

```
void    func(int a, int b){
        // Код работы с a и b
}
```

мы не задумываемся, как попадают в неё параметры и где хранятся. Нам важно только то, что: a и b — локальные копии каких-то целых чисел или выражений. И этого нам хватает.

Но если мы пишем на ассемблере, то все по-другому. Без знания — как получить параметры — нам никуда.

В языке С используются два способа передачи параметров — по значению и по ссылке. Заметим, что ссылка в языке С (не путать с С++!), называется ***указателем***. Как её не назови — ссылка или указатель — она (он) является адресом в памяти, по которому хранится тот объект (переменная, структура, массив), на который указывает эта ссылка.

В первом способе — передача параметров по значению — на стеке создаётся копия переменной, которая доступна функции. С этой копией можно делать что угодно — это никак не повлияет на переменную, которая передана по значению.

Во втором способе — передача параметров по ссылке — на стек помещается адрес переменной. И обращение производится по этому адресу к самой переменной, а не к её копии. То есть функция может изменять переменную. С помощью переменных, передаваемых по ссылке, функция может, например, возвращать результат.

Продвинутые кмпиляторы умеют передавать параметры через стек или регистры процессора, для ускорения. Но SDCC всегда передаёт параметры ***только через стек***.

Это упрощает нам жизнь в том смысле, что есть только один способ добраться до вожделенных параметров.

Итак, предположим, что мы вызываем нашу функцию из примера из какой-то части кода. Рассмотрим, как это выглядит на ассемблере:

Код на С:

```
func(19,77);
```

Тот же код на ассемблере (я упростил его для понимания):

```
ld     hl,#77          - помещаем на стек int b
push   hl

ld     hl, #19         - помещаем на стек int a
push   hl

call   _func           -  вызов функции (на стек помещается адрес возврата)

pop    af              - очистка стека
pop    af
```

Что мы видим? А видим мы много удивительного и полезного.

Во-первых — параметры помещаются на стек таким образом, что располагаются на нём от вершины к основанию в том же порядке, как они описаны в функции.

Во-вторых — в момент вызова функции, что логично, на стеке также сохраняется адрес возврата.

В-третьих — после вызова функции с параметрами, эти параметры удаляет ***вызывающая*** функция. То есть нам в ***вызываемой*** функции (**_func**) (а именно она у нас написана на ассемблере) ничего чистить не надо.

В-пятых — первый параметр располагается в ***вызываемой*** функции (**_func**) на стеке по адресу **SP+2**, так как два байта занимает адрес возврата.

Предположим, что мы решили переписать функцию на встроенном ассемблере. Теперь нам ясно, как добраться до параметров в функции ***void func(int a, int b)***:

```
void    func(int a, int b)__naked{
__asm;                  // Вход в функцию. Первый параметр по адресу SP+2
        push    ix      // Запомнили ix. Теперь параметр по адресу SP+4
        ld      ix,#4
        add     ix,sp   // Теперь ix указывает на int a;     ix+2 — на int b

        ld      l,0(ix) // hl=a
        ld      h,1(ix)

        ld      e,2(ix) // de=b
        ld      d,3(ix)

        ;// Делаем тут все что хотим

        pop     ix      // Восстанавливаем ix
        ret             // Возврат из функции
__endasm;
}
```

Как видим — ничего сложного. Помните только, что:

- не все параметры можно передать по значению. Массивы, строки, структуры передаются только по ссылке;
- параметр на стеке занимает ровно столько места, сколько составляет размер его типа. Для SDCC различные типы имею такие размеры:
  - **char — 1 байт (signed или unsigned - неважно)**
  - **int — 2 байта (signed или unsigned - неважно)**
  - **long — 4 байта (signed или unsigned - неважно)**
  - **long long — 8 байт (signed или unsigned - неважно)**
  - **float — 4 байта**
  - **void* — 2 байта (любой указатель или ссылка)**

5.5. Передача значений, возвращаемых функцией

Из функции можно вернуть только те типы, которые разрешено передавать по значению.

Правила возврата следующие:

- однобайтовые типы (**char**) возвращаются в регистре **L**;
- двухбайтовые типы (**int, void***) возвращаются в регистрах **HL**;
- четырёхбайтовые типы (**long, float**) возвращаются в регистрах **DEHL** (от старшего к младшему);
- восьмибайтовые типы (**long long**) — возвращаются на стек, где **вызывающая** функция зарезервировала место. Причем, на стеке хранится адрес параметра.

Рассмотрим примеры функций на ассемблере, которые корректно возвращают параметры в С. Чтобы не лопухнуться, необходимо, корректно описать прототип функции, указав все типы возвращаемых и принимаемых параметров.

Предположим, что у нас есть функция, которая возвращает один байт. Байт это — это в SDCC **char**, **signed char** или **unsigned char**. Я рекомендую всегда явно указывать типы с размерностями, например: **int8_t**, **uint16_t** и так далее. Описание этих типов находится в файле **stdint.h**, так что не забудьте его подключить. Опишем эту функцию:

```
#include <stdint.h>

uint8_t getByte()__naked{
        __asm;                      // Вход в функцию.

                ld      L,#123      // Значение возвращаемого значения
                                    //  заносим в регистр L.

                ret                 // Возврат из функции
        __endasm;
}
```

Функция **getByte()** в нашем примере всегда будет возвращать константу 123.

5.6. Примеры функций на ассемблере

В примерах будем считать, что наша программа запускается из бейсика. То есть Функции ПЗУ доступны. Это поможет нам избежать кучи не информативного кода.

Пример 1. Функция печати строки

Функция вывода символа на экран в ПЗУ ZX всем хорошо известна. Чтобы вывести строку на экран необходимо в регистр **A** поместить код символа, а затем вызвать программное прерывание **RST 16**. Таким образом можно напечатать латинские символы, но не русские (нет в ПЗУ кириллицы). Для нашего примера это не существенно. Исходные данные следующие:

- строка в языке C — это указатель на последовательность байт, т. е. **char***;
- строка в языке C завершается символом с кодом 0, т. е. **'\0'** или, что тоже байт **0**.

Глядя на исходные данные, можно сказать, что нам необходимо написать цикл печати всех символов, находящихся в памяти, начиная с указанного адреса, пока мы не встретим символ с кодом 0.

Попробуем выразить наши мысли на языке ассемблера. На самом деле это не сложно. Текст примера с комментариями приведён ниже.

```
void puts(const char* s)__naked{
        s;      // Это просто для того, чтобы компилятор не ругался
                // на неиспользуемую переменную s. Он ничего
                // не знает о том, что мы делаем в ассемблере.
    __asm;
            ;// Со стека надо извлечь одно слово — адрес строки
            ;// адрес строки лежит на стеке по адресам SP+2 и SP+3
            ld      hl,#0x0002
            add     hl,sp   ;// HL = SP+2: адрес на стеке, где лежит
```

```
                    ;// указатель на строку (то есть адрес  строки)

        ld      e,(hl)  ;// Извлекаем адрес строки в DE
        inc     hl
        ld      d,(hl)  ;// DE = адрес строки

        ;// Выводим строку, пока не встретим байт 0. (Символ  '\0')
puts_loop:
        ld      a,(de)  ;// Извлекаем очередной код символа в рег. A
        or      a       ;// Если код символа равен 0,
                        ;//то выставится флаг Z регистра флагов

        ret     z       ;// Если флаг Z=1, то выходим из функции, т. к.
                        ;// строка закончилась

        rst     0x10    ;// Иначе — печатаем символ
        inc     de      ;// Увеличиваем указатель на строку на 1,
                        ;// т. е. переходим к обработке
                        ;// следующего символа
        jr      puts_loop       ;// Переход на новую итерацию цикла
__endasm;
}
```

Вызвать нашу функцию можно из языка С очень просто, как и любую другую:

```
puts("My printed string");
```

Этот вызов прекрасно напечатает на экране указанную строку.

Пример 2. Функция печати строки с возвратом количества символов

Наша функция печати всем хороша. Любо-дорого глядеть, как она бойко выводит на экран всякие разные символы. Но есть в ней темные пятна — мы не знаем, сколько же символов вывела эта функция на экран. А иногда это очень полезно. Модифицируем её, чтобы устранить этот недостаток.

Главное отличие этого примера от предыдущего — это то, что имеется не только параметр, передаваемый в функцию, но и значение, возвращаемое из функции.

В нашем случае — это длина строки. Не будем себя ограничивать и условимся, что строки будут до **65535** байт длиной. Поэтому возвращать будем слово (**uint16_t**).

В коде мы рассмотрим только отличия от предыдущего примера.

```
uint16_t putsn(const char* s)__naked{
    s;
    __asm;
        ld      hl,#0x0002
        add     hl,sp
        ld      e,(hl)
        inc     hl
        ld      d,(hl)
```

```
        ;// -------- Отличие первое --------
        ;// В HL будет счетчик выведенных символов
        ld      hl,#0000        ;// Счетчик символов обнулен

putsn_loop:
        ld      a,(de)
        or      a
        ret     z       ;// Если происходит выход,
                        ;// то в HL — количество выведенных символов

        rst     0x10
        inc     de

        ;// -------- Отличие второе --------
        inc     hl      ;// С каждым выведенным символом,
                        ;// увеличиваем счетчик выведенных символов

        jr      putsn_loop
__endasm;
}
```

Так как нам надо вернуть слово, а мы знаем, что двухбайтовые значения помещаются в пару **HL** для возврата их из функции, то мы поместили счетчик выведенных символов именно в эту пару — **HL**. Как видим, отличия от предыдущего примера — всего две строчки (ну или четыре байта кода). А каков результат! Мы теперь не просто выводим символы, но и знаем, сколько ж их было в строке!

Пример 3. Функция сложения двух очень длинных слов

Для примера второго способа возврата параметров рассмотрим функцию сложения двух очень длинных слов, по 64 бита каждое.

Чтобы правильно написать такую функцию, надо понимать, на стеке располагаются параметры, передаваемые в функцию. Как уже говорилось, 64-битные значения возвращаются не через регистры, а через стек.

Наша функция будет описана как

```
uint64_t        addll(uint64_t a, uint64_t b);
```

Рассмотрим, что находится на стеке при входе в эту функцию.

Стек	***Размер***	***Что тут находится ?***
SP-20	8 байт	Возвращаемое значение 64 бита
SP-12	8 байт	Параметр b
SP-4	8 байт	Параметр a
SP-2	2 байта	Адрес возвращаемого 64-битного длинного-длинного целого, равный **SP-20**
SP	2 байта	Адрес возврата из подпрограммы (используется **ret**)

То есть на стеке мы займём 28 байт.

Обратите внимание, что адрес возвращаемого параметра хранится в начале списка параметров (сразу за адресом возврата из подпрограммы), а сам параметр — в конце.

Так устроен SDCC неспроста. Дело в том, что имеются функции с переменным числом входных параметров и было неудобно с ними работать, если бы параметры располагались в плавающем виде.

Таким образом, для получения доступа к возвращаемому параметру, нам необходимо снять его адрес со стека.

```
uint64_t addll( uint64_t a, uint64_t b )__naked{
    a;b;
    __asm;
        ld      hl,#4   ;// Вычисляем адрес параметра a = SP+4
        add     hl,sp
        ex      de,hl   ;// a = (de) — помещаем этот адрес в de
        ;//
        ld      hl,#2   ;// Вычисляем адрес возвращаемого значения
        add     hl,sp   ;// return = (SP+12)
        ld      c,(hl)
        inc     hl
        ld      b,(hl)  ;// return = (bc)  — помещаем этот адрес в bc
        ;//
        ld      hl,#12  ;// Вычисляем адрес параметра b = SP+12
        add     hl,sp   ;// b = (hl)  — помещаем этот адрес в hl

        ;// Делаем сложение младших байтов (без переноса)
        ld      a,(de)
        add     a,(hl)
        ld      (bc),a  ;// (bc) = (de)+(hl)  - без учета флага переноса
        inc     de
        inc     hl
        inc     bc

        ;// Повторяем сложение ещё 7 раз (с переносом)
        exx
        ld      b,#7
    addll7:
        exx
        ld      a,(de)
        adc     a,(hl)
        ld      (bc),a  ;// (bc) = (de)+(hl)  - с учетом флага переноса
        inc     de
        inc     hl
        inc     bc
        exx
        djnz    addll7

        ret
    __endasm;
    }
```

Как видим, для сложения длинных целых и функция получилась длинная. Не оптимально, но понятно.

6. Структура каталогов SDCC-NOINIT

Проект SDCC-NOINIT состоит из дерева каталогов различного назначения:

apps - каталог, содержащий исходники собираемых программ. Одна программа - один подкаталог.

Выходные файлы:
Для программ: *.tap *.$C *.tap *.bin
Для плагинов WC: *.wmf
Если что-то не нравится - всегда можно поправить Makefile конкретной программы.

configs — файлы глобальной конфигурации, определяющие какие программы и библиотеки и как собирать.

include - заголовочные файлы, которые доступны все библиотекам и программам всегда.

libsrc - каталог, содержащий исходники собираемых библиотек. Одна библиотека - один подкаталог. При сборке доступны заголовочные файлы только тех библиотек, которые разрешены.

scripts - скрипты, используемые для сборки (перемещение памяти, преобразование текста, автогенерация файлов ...)

Makefile - Головной Makefile. Лучше его не трогать.

6.1. Каталог с исходными текстами apps

Исходные тексты программ: один каталог — одна программа. Какие из программ собирать, определено в файле **configs/apps.mk**.

В каталоге с программой обязательно находятся файлы: **Makefie, config.mk** и **build.mk**.

Для программы и плагинов WC эти файлы различаются. Но основа у них одна.

Файл **config.mk —** файл конфигурации стартового кода. На его основе генерируется файл **autoconfig.asm**, который содержит определения, передаваемые в стартовый код.

Для стандартного стартового кода **crt0.s** определяются такие параметры:

```
# Начальный адрес загрузки и запуска программы
START=0x6000

# Размер стека
SSIZE=0x0400

# Управление прерываниями (0-при старте программы запрещаются,
# 1 — не меняются)
DISTART=0

# Использование собственного стека размером SSIZE (0) или
# стека вызывающей программы (1)
CALLERSTACK=0
```

Для стандартного стартового кода WC-плагина **crt0wcplugin.s** определяются такие параметры:

```
# Начальный адрес загрузки и запуска плагина
START=0x8000

# Начальный адрес загрузки заголовка плагина
STARTPLHEADER=0x6000
```

Пока что я создавал плагины размером 1 страницу.

Файл **build.mk —** содержит такую информацию:

```
# Имя программы
APP=zxkbdtest

# Список объектных файлов
OBJ=main.rel test.rel

# Список библиотек. Если он не задан — используются ВСЕ имеющиеся
# библиотеки. Очень полезная функция, если библиотеки конфликтуют.
# LIBS=libconio libz80

# Путь к файлу со стартовым кодом. Если не указан — то crt0.s
# CRT0PATH=$(TOP)/libsrc/crt0/$(CRT0)
```

Как видим, большей частью все просто и понятно. Отметим, что неважно — на Ассемблере или С написан файл — указывается только имя объектного файла. А уж система сама ищет из какого исходника её создать.

Помните, что нельзя в одном каталоге иметь два исходника с одинаковыми именами на разных языках, например **main.c** и **main.s**. В этом случае система не сможет определить — из какого же файла создавать **main.rel** ?

Если вам нужны другие локальные для программы параметры — создавайте свои ***.mk**-файлы. Все, что вы в них напишите — будет включено в **Makefile** программы в том каталоге, в котором вы их поместите..

6.2. Каталог глобальной конфигурации configs

Содержит файлы глобальной конфигурации. Любой ***.mk**-файл из каталога **configs** будет добавлен в описание глобальной конфигурации всех программ и библиотек.

Назначение некоторых глобальных конфигурационных файлов:

apps.mk — указывает какие программы-плагины собирать (список каталогов в **apps**);
crt0.mk — определяет шаблон стартового кода для программ (он берется по умолчанию);
crt0wcplugin.mk — определяет шаблон стартового кода для WC-плагинов;
library.mk — указывает какие библиотеки собирать (список каталогов в **libsrc**);
printf_float.mk - использовать плавающую точку в **printf** или нет (сильно влияет на размер);
sfs_float.mk — выбор стандартной реализации сложения **float** или моей;
tools.mk - определяет какие утилиты использовать. Менять можно, но очень-очень осторожно:).

Если вам нужны другие глобальные параметры — создавайте свои ***.mk**-файлы. Все, что вы в них напишите — будет включено во все **Makefile**.

6.3. Каталог с общими заголовочными файлами include

В данном каталоге находятся заголовочные файлы, доступные всем программам и библиотекам без исключения.

Можете добавлять сюда свои *.h-файлы. Но помните, что они будут видны всем.

6.4. Каталог с исходными текстами библиотек libsrc

Исходные тексты программ: один каталог — одна библиотека. Какие из библиотек собирать, определено в файле **configs/library.mk**.

Сборка библиотеки целиком определяется в **Makefile**.

Собственно, сборка состоит из двух шагов:

- из файлов ***.c** и ***.asm** генерируются ***.rel** файлы;
- с помощью специальной утилиты все полученные ***.rel** файлы собираются в один библиотечный ***.lib** файл.

И тут есть подводный камень: если у вас в одном ***.rel**-файле несколько функций, то все содержимое этого файла будет извлечено из библиотеки и вставлено в вашу программу. Даже если используется только одна функция.

Выход один — объединять в один файл только неразрывно связанные функции, которым друг без друга не жить.

Только так вы сможете минимизировать потери памяти при использовании библиотек.

Например, взгляните на **libz80**: одна функция — один файл почти везде. Поступайте подобным образом. Может SDCC когда-нибудь научится удалять мертвый код из библиотек…

Точно так же как и с каталоге с программами — в каждом каталоге с библиотекой имеются файлы **Makefie** и **build.mk**.

В файле **build.mk** указывается имя библиотеки и файлы из которых она состоит, например:

```
# Имя библиотеки
LIBNAME=libay.lib

# Объектные файлы, из которых она состоит.
OBJS=pt3xplayer_s.rel pt3xplayer.rel
OBJS+=stc_player_s.rel stc_player.rel
```

При необходимости можно добавлять в каталог библиотеки и дополнительные ***.mk** — файлы, например со специфичными для данной бмблиотеки параметрами сборка, ключами компилятора и т.д.

6.5. Каталог со скриптами scripts

Содержит сборочные скрипты для конфигурации, оптимизации, сборки. Кратко о назначении скриптов:

autoconfig — генерирует файл autoconfig.asm на базе файла config.mk. Можно использовать для своих нужд.

Пример. Имеется файл **config.mk**:

 # Start adress of module/program
START=0x6000

 # Stack Size
SSIZE=0x0400

 # Interrupt on start: 0-disable, 1-enable
DISTART=0

 # Store caller stack (1) or self stack (0)
CALLERSTACK=0

Запускаем скрипт **autoconfig**. Первый параметр — имя входного файла, второй — имя выходного:

#./autoconfig config.mk autoconfig.asm

Получаем файл **autoconfig.asm**:

;; This file generated automatically. Do not change it.

__START=0x6000
__SSIZE=0x0400
__DISTART=0
__CALLERSTACK=0

;; end of autoconfig

Догадались, что делает скрипт? Он удаляет все комментарии, начинающиеся с решетки (#) и добавляет ко всем именам переменных два символа подчеркивания. Такой файл уже можно включать в исходный код на ассемблере директивой **.include / autoconfig.asm/**.

sdrmini — удаляет переносит секцию **_INITIALIZER** на место секции **_INITIALIZED** и удаляет секцию **_INITIALIZER**. Для работы скрипта нужны два файла: бинарный файл скомпилированной программы и ***.map**-файл, который создаёт линкер при линковке. Запуск скрипта:

#./sdrmini app.map app.bin

bin2hob — преобразование bin-файла в *.$C файл с правильным заголовком. Скрипт требует обязательного наличия **z80asm** (Z80 Module Assembler (c) InterLogic). Запуск скрипта:

#./bin2hob name 0x6000 app.bin

где

name — имя (не более 8 символов);
0x6000 — стартовый адрес, указываемый в заголовке (укажите свой);
app.bin — скомпилированный двоичный файл.

plughdr_1page — синтезирует заголовок одностраничного плагина WC на ассемблере. Запуск скрипта:

#./plughdr_1page plheader,asm name [ext1] [ext2] … [extN]

где

plheader,asm — имя выходного файла на ассемблере;
name — имя плагина (не более 32 символов);
ext1 .. extN - расширения файлов для запуска плагина. Могут не указываться.

s2hs — преобразует определения констант на C в определение констант на ассемблере. Удаляет строки с комментариями в виде двух слэшей (//). Запуск скрипта:

#./s2hs infile.h

Создаёт выходной файл, добавляя к расширению S-заглаыную. В нашем примере — **infile.hS**.

Пример. Имеем файл **infile.h**:

```
#define SCR_TXT_W 0x20
#define SCR_TXT_H 0x18

#define SYM_AT 0x16
#define SYM_TAB 0x09
#define SYM_NL 0x0A
#define SYM_CR 0x0D

// Размер табуляции (2^K_TAB - размер)
#define K_TAB       0x02
```

После обработки утилитой s2hs получаем файл **infile.hS**:

```
SCR_TXT_W=0x20
SCR_TXT_H=0x18

SYM_AT=0x16
SYM_TAB=0x09
SYM_NL=0x0A
SYM_CR=0x0D

K_TAB=0x02
```

В будущем эти скрипты надо бы переработать, но пока используем то, что есть.

6.6. Каталог программ bin

Сюда копируются все собранные программы.

6.7. Каталог библиотек libs

Сюда копируются все собранные библиотеки.

7. Сборка программы в SDCC-NOINIT

7.1. Конфигурация компонентов — библиотек и программа

Конфигурация имеется глобальная (файлы находятся в каталоге **configs**) и локальная (***.mk** файлы в каталоге конкретной программы в **apps**).

В каталоге каждой программы имеется файл **config.mk**, в котором описаны параметры данной программы, такие как адрес загрузки и запуска данной программы, размер стека, разрешение прерываний по запуску программы и проч.

На основе файла **config.mk** генерируется стартовый код для программы **crt*.rel**.

Глобальная конфигурация доступна всем программам и библиотекам и разбита на отдельные текстовые ***.mk**-файлы для удобства.

Любой ***.mk**-файл из каталога **configs** будет добавлен в описание глобальной конфигурации.

Любой ***.mk**-файл из каталога программы будет добавлен в описание локальной конфигурации.

7.2. Добавление новой программы в систему сборки

Чтобы добавить свою программу в систему сборки, необходимо совсем немного. Все пути указываем относительно корневого каталога системы SDCC-NOINIT.

Шаг 1

Создаём каталог программы. Он располагается в каталоге **apps**. Например **apps/ myprogram**.

Шаг 2

Создаем файлы **Makefile**, **config.mk** и **build.mk**. Я рекомендую просто скопировать эти файлы из любого другого каталога с программой.

Шаг 3

Редактируем файлы **config.mk** и **build.mk**: указываем там имя нашей программы и список ***.rel** — файлов, которые мы должны получить из наших ***.c** и ***.s** файлов.

Например, если у нас есть один файл на языке С **main.c** и один файл на языке ассемблера **fast.s**, то нам надо указать в файле **build.mk**:

```
# Имя программы
APP= myprogram

# Объектные файлы
OBJ=main.rel        fast.rel
```

Сборочный скрипт сам определит как какой файл компилировать. Относительно остальных параметров файлов **config.mk** и **build.mk** уже было сказано в 6.1.

Шаг 4

Добавляем нашу программу в список сборки программ. Список сборки программ — это файл **configs/apps.mk**. Для нашего примера в него надо добавить строчки:

```
# Программа myprogram — первый опыт в SDCC-NOINIT
APPLICATIONS+= myprogram
```

Имя добавляемой программы должно совпадать с названием каталога с этой программой в каталоге **apps**.

Шаг 5

Пишем программу (те самые файлы, что были указаны в **build.mk**) и собираем её. Для сборки из корня SDCC-NOINIT надо запустить команду **make**:

```
#make
```

И, если все в порядке, то после сборки всех библиотек и программ у вас появятся файлы: **myprogram-basic.tap, myprogram-code.tap. myprogram.bin, myprogram.$C**.

То есть ваша программа будет вам представлена сразу в нескольких видах. Вы можете её запустить или приделать к ней свой загрузчик.

myprogram-basic.tap — программа с бейсик-загрузчиком (образ ленты);
myprogram-code.tap — программа без бейсик-загрузчика (образ ленты);
myprogram.bin — просто бинарный образ без загрузчика и без заголовка;
myprogram.$C — кодовый файл НОВЕТА (образ для диска).

Добавление нового плагина WC в систему сборки

Добавление нового WC-плагина осуществляется так же как и новой программы, но имеет два отличия:

- файлы **Makefile**, **config.mk** и **build.mk**.необходимо брать из каталога с примером WC-плагина, а не с примером программы;
- в результате сборки получается только один .wmf-файл с плагином.

В остальном — всё полностью аналогично тому, что говорилось про сборку программ.

7.3. Добавление новой библиотеки в систему сборки

Чтобы добавить свою библиотеку в систему сборки, необходимо совсем немного. Все пути указываем относительно корневого каталога системы SDCC-NOINIT.

Шаг 1

Создаём каталог программы. Он располагается в каталоге **libsrc**. Например **libsrc/ libmylib.**

Шаг 2

Создаем файлы **Makefile**, **build.mk**. Я рекомендую просто скопировать эти файлы из любого другого каталога с библиотекой.

Шаг 3

Редактируем файл **build.mk**: указываем там имя нашей библиотеки и список ***.rel** — файлов, которые мы должны получить из наших ***.c** и ***.s** файлов.

Например, если у нас есть один файл на языке С **interface.c** и один файл на языке ассемблера **funct.s**, то нам надо указать в файле **build.mk**:

```
# Имя библиотеки
LIBNAME=libmylib.lib

# Объектные файлы
OBJS=interface.rel    funct.rel
```

Сборочный скрипт сам определит как какой файл компилировать. Относительно остальных параметров файла **build.mk** уже было сказано в 6.4.

Шаг 4

Добавляем нашу библиотеку в список сборки библиотек. Список сборки библиотек — это файл **configs/library.mk**. Для нашего примера в него надо добавить строчки:

```
# Библиотека mylib — первый опыт в SDCC-NOINIT
LIBRARY+= libmylib
```

Имя добавляемой библиотеки должно совпадать с названием каталога с этой программой в каталоге **libsrc**.

Шаг 5

Для сборки библиотеки из корня SDCC-NOINIT надо запустить команду **make**:

```
#make
```

И, если все в порядке, то после сборки в каталоге **libs** появится файл с нашей библиотекой **libmylib.lib**.

8. Описание библиотек

8.1. Библиотека файлов начального запуска crt0

Данная библиотека содержит исходные тексты файлов начального запуска программ. Структура и назначение этих файлов описана в разделе 4.7.

8.2. Стандартная библиотека C (libz80)

Эта библиотека тесно связана с компилятором C и является усечённой версией библиотеки **libc**.

Основные части этой библиотеки:

1. **Математическая** — работа со стандартным float IEEE 754. Подробнее тут: http://man7.org/linux/man-pages/man0/math.h.0p.html. Разумеется, что для Z80 версия не совсем полная, но основное там есть.

2. **Работа со строками**, в том числе с «широкими» символами. Подробнее тут: https://linux.die.net/man/3/string. Опять же — не совсем всё есть, но основное работает шикарно.

3. **Вывод типа printf**. Подробнее тут: https://linux.die.net/man/3/printf. Вывод в строку — без проблем. Для вывода в стандартный поток (например, на экран), необходимо определить свою функцию вывода **putchar**:

```
void putchar( char c ){
        // Вывод символа куда угодно.
}
```

   Отметим, что в библиотеке **libconio** данная функция уже определена для вывода на стандартный экран ZX (256x192, цвета по знакоместам). Если вы используете **libconio**, то можно спокойно пользоваться функцией **printf** для вывода на экран.

4. **Работа с датой и временем**. Функции работы с датой-временем в формате UNIX-TIME. Очень полезные функции, описания которых можно найти тут https://linux.die.net/man/3/.
   Если вы хотите получать дату и время стандартной функцией **time()**, то вам необходимо определить свою функцию считывания даты-времени с часов реального времени (или ещё откудато):

```
unsigned char RtcRead(struct tm *timeptr) {
    // Заполнение структуры timeptr
    return 0;
}
```

   Остальные функции платформо-независимы.

5.

9. Краткий справочник-глоссарий

Здесь собраны термины, названия программ и сокращения с пояснениями.
В общем всё то, что может вызвать вопросы. Обращаю внимание, что это ***краткий*** справочник-глоссарий. То есть тут указаны основные вещи. Подробности смотрите в документации к соответствующим программам, если она есть в природе.

sdcc

Фронт-энд и компилятор C. У него есть множество опций. Для того, чтобы генерировать код в формате ассемблера Z80 используется опция **-mz80**.

Например комнада:

 # sdcc -mz80 test.c -o test.ihx

транслирует программу **test.c** полностью пройдя все стадии — препроцессор, компилятор, ассемблер и линкер. Если нет ошибок, то мы получим готовую программу в виде файла **test.ihx**.

Ключ **-o** позволяет задавать имя выходного файла, отличного от того, что создаётся по умолчанию.

Также можно прервать трансляцию на любой стадии:

Команда **# sdcc -mz80 -E -Wp,-P test.c -o test.E**

запустит только препроцессор; на выходе — препроцессированный файл **test.E**.

Команда **# sdcc -mz80 -S test.c -o test.asm**

запустит препроцессор и компилятор, но не запустит ассемблер и линкер; на выходе — файл на языке ассемблера **test.asm**, сгенерированный компилятором.

Команда **# sdcc -mz80 -c test.c -o test.rel**

запустит препроцессор, компилятор и ассемблер, но не запустит линкер; на выходе — объектный файл **test.rel**, сгенерированный ассемблером.

Препроцессор, ассемблер и линкер — это отдельные программы, которые можно запускать не из фронтенда **sdcc**, а отдельно.

sdcpp

Препроцессор языка C. Обрабатывает директивы препроцессора и нечем более не занимается. Запускается командой:

 # sdcpp -P [имя входного файла] [имя выходного файла]

Если имена не указаны (вместо имени стоит знак минус «-») — используются потоки стандартного ввода или вывода.

sdas

Общее название ассемблеров пакета **sdcc**. Для **Z80** ассемблер называется **sdasz80**. Общие правила запуска программ и ключи похожи. Программа **sdasz80** выполняет ровно одну функцию — преобразует код на языке ассемблера в объектный код:

*# **sdasz80 [опции] -o <имя>.rel <>.asm <>.asm …*** **- общий формат**

 # sdasz80 -o test.rel test.asm test.asm **- пример**

Обратите внимание, что в ассемблере **sdas** ключ **-o <имя выходного файла>** должен быть указан обязательно. Опции, если таковые имеются помимо **-o**, указываются до опции **-o**. Если указано несколько ассемблерных файлов — они ассемблируюся в один объектный файл.

sdld

Общее название линкеров пакета **sdcc**. Для **Z80** линкер называется **sdldz80**. Общие правила запуска программ и ключи похожи. Программа **sdldz80** выполняет ровно одну функцию — линкует один или более файлов, в образ программы, в заданном формате, например **IntelHex**:

 # sdldz80 [опции] -i <имя>.ihx <>.rel <>.rel … **- общий формат**

 # sdldz80 -i test.ihx test.rel test1.rel **- пример**

Опции, если таковые имеются помимо **-i**, указываются до опции **-i**.
Если указано несколько объектных файлов — они линкуются в один файл-образ.

hex2bin

Так как нам нам надо загружать в память и выполнять двоичные файлы, а линкер даёт только IntelHex, то нас выручит утилита **hex2bin**. Она преобразует файлы формата **IntelHex** в бинарные.

 # hex2bin -p <байт> -s <старт. адрес> <имя>.ihx ***- общий формат***

 # hex2bin -p 0 -s 0x6000 myprogram.ihx **- пример**

На выходе получим файл ***<имя>.bin*** (**myprogram.bin** в нашем примере).

Поясним, что всё это значит. Ключ **-p** задаёт байт-заполнитель. Дело в том, что в формате **IntelHex** могут быть разрывы. То есть значение некоторых участков памяти не определено. Образ же — непрерывен. Вот этим самым ключем **-p** мы и задаём, чтобы все разрывы заполнялись нулём (стандарт языка С, однако!). Если не задать — будут неприятные эффекты.

Ключ **-s** указывает, с какого адреса начинается наш образ (обычно, это стартовый адрес нашей программы). И этот адрес всегда указан в ***HEX-формате***. По умолчанию, утилита **hex2bin** считает, программа начинается с адреса **0x0000**. И заполняет всё пространство от **0x0000** до первого адреса, указанного в .ihx-файле — байтом-заполнителем. А нам этого совсем не надо. Так что не забывайте об этом ключе.