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 говориться, «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. 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.

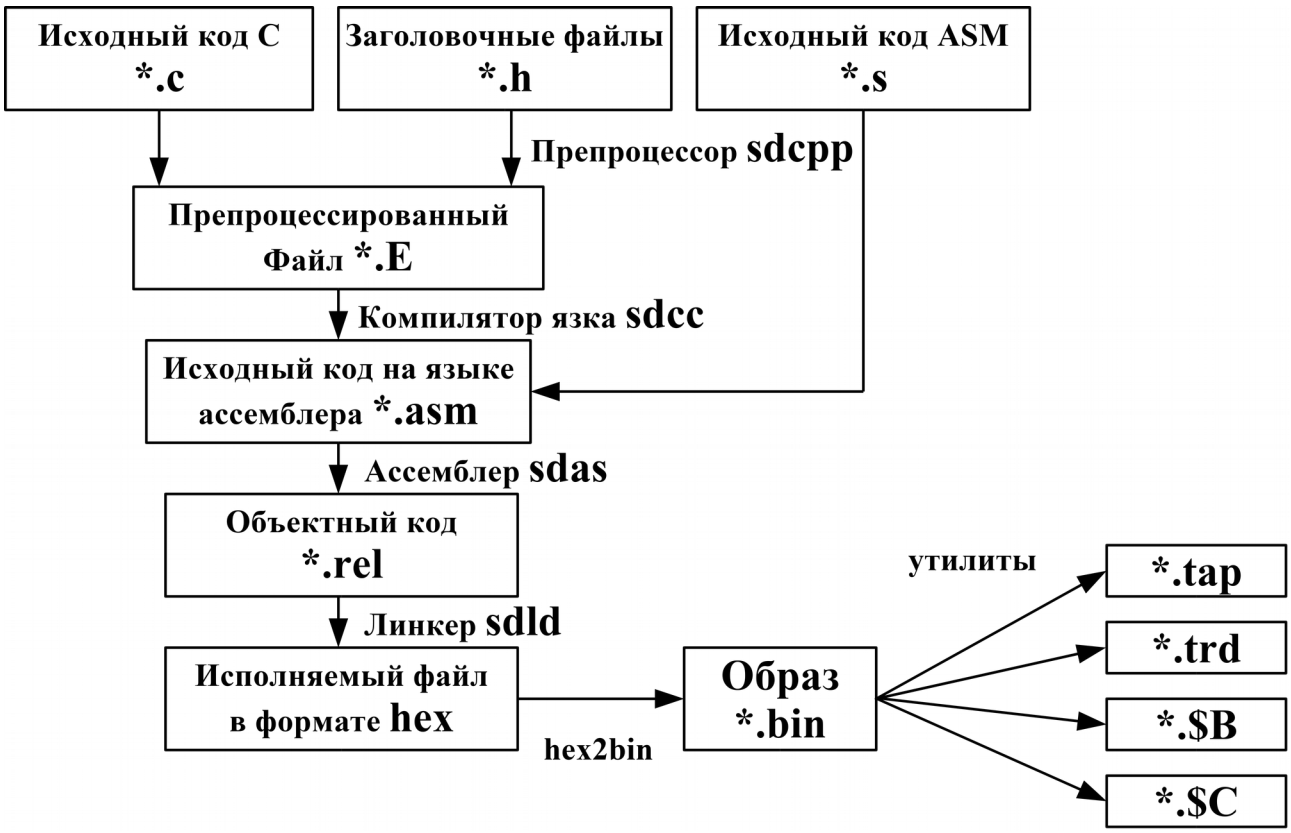

Рис. 1: Трансляция программы на языке С пакетом SDCC - от исходного кода до исполняемых файлов

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.

1.1. 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.

1.2. 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.

1.3. 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.

1.4. 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.

1.5. 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.

1.6. 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.

2. Особенности SDCC

2.1. Линковка и секции линкера

2.2. Компилятор

2.3. Стартовый код

3. Структура каталогов SDCC-NOINIT

3.1. Каталог с исходными текстами apps

3.2. Каталог с исходными текстами configs

3.3. Каталог с исходными текстами include

3.4. Каталог с исходными текстами libsrc

3.5. Каталог с исходными текстами scripts

3.6. Каталог программ bin

3.7. Каталог библиотек libs

4. Сборка программы в SDCC-NOINIT

4.1. Конфигурация компонентов — библиотек и программа

4.2. Добавление новой программы в систему сборки

4.3. Добавление новой библиотеки в систему сборки

4.4. Файлы глобальной конфигурации

4.5. Файлы локальной конфигурации программ

4.6. Файлы локальной конфигурации WC-плагинов

4.7. Файлы начального запуска crt0*.s

4.8. Процесс сборки программы

4.9. Процесс сборки библиотеки

4.10. Сборочные скрипты

5. Описание библиотек

5.1. Библиотека файлов начального запуска crt0

5.2. Библиотека libc для ZX: libz80

5.3. Библиотека музыкальных плееров libay

5.4. Библиотека вывода на стандартный экран ZX libconio

5.5. Библиотека работы с прерываниями libim2

5.6. Библиотека работы с кольцевыми буферами libringbuf

5.7. Библиотека вывода спрайтов на стандартный экран ZX libspr

5.8. Библиотека работы с таймерами libtimer

5.9. Библиотека работы с UARTами libuart

5.10. Библиотека создания плагинов для WC libwcplugin

5.11. Библиотека работы с клавиатурой ZX libzxkbd

5.12. Библиотека работы с клавиатурой PS2 libps2