СЕРИЯ
НАРОДНАЯ ЭЛЕКТРОНИКА
ВЫПУСК 2

GCC Cortex-M3

А. В. Немоляев

Екатеринбург 2015

УДК 621.325.5
ББК 32.84

Немоляев А.В.
GCC Cortex-M3. − Екатеринбург.: Живая мысль, 2015. − 53 с.
vk.com.protocols@yandex.ru
vk.com/protocols

В сжатой, но доступной форме излагается процесс программирования микроконтроллеров с ядром Cortex-M3, средствами GCC, материал ориентирован на начинающих разработчиков микропроцессорных систем и радиолюбителей.

УДК 621.325.5
ББК 32.84

Все права защищены. Никакая часть этого издания не может быть воспроизведена в любой форме или любыми средствами, электронными или механическими, включая фотографирование, ксерокопирование или иные средства копирования или сохранения информации, без письменного разрешения автора.

© Немоляев А.В., 2015

Оглавление

Оглавление 3

Предисловие 4

Форматы и секции..5

Первая программа..8

Компоновка 11

Отладчик и сервер отладки..13

Отладка 15

Связывание двух модулей 17

LMA и VMA..20

Программа на С 23

Реальная программа 26

Библиотеки и карта компоновки 29

Программа make 33

Литература 35

Предисловие

В отличие от программистов создающих программы для персональных компьютеров работающих под управлением операционной системы, программист, пишущий для встраиваемых систем должен знать всю «кухню» приготовления бинарного образа загружаемого в память целевой системы. В коммерческих средах разработки программ для встраиваемых систем, создаётся комфортная среда, где не нужно особо вникать в процесс сборки программы. А для мира GCC, для мира Free Software – навыки «кроить» и «шить» код из разных источников просто необходимы. Знание одного компилятора GCC и сопутствующих инструментов, позволяет работать в различных областях программной индустрии. Это и программирование в среде различных операционных систем, таких как: Linux, FreeBSD, uClinux, FreeRTOS, Android, QNX. И огромное количество готовых программ и библиотек, использование которых сокращает сроки разработки и затраты на программирование. Этот компилятор поддерживает великое множество платформ, почти на все случаи жизни. Использование готовых программ и библиотек для своих нужд, способно экономить огромные ресурсы. Некоторые коммерческие среды разработки используют GCC , но процесс компиляции и компоновки «не прозрачен» для разработчика.

В интернете много фрагментарных материалов по применению GCC для компилирования программ для микроконтроллеров. Чаще всего – это переводы кратких рецептов с английского языка, без объяснения логики процесса. Если что-то пойдёт не так, даже из-за пустяковой причины, разработчик оказывается в тупике. Этот материал – попытка «заглянуть под капот».

Мой материал подаётся как набор усложняющихся практических упражнений в ОС Linux. Истинность того, или иного утверждения, подтверждается реальными примерами. Я думаю, что очевидно, почему я использовал Linux для экспериментов с GCC. Многие задачи решать легче в середе Linux, если работаете с GCC.

Для примеров использовался порт GCC для встраиваемых систем на процессоре ARM, из проекта «GCC ARM Embedded». В составе этого пакета поставляется всё необходимое для создания программ для микроконтроллеров с архитектурой Cortex-M3. В дальнейшем, для краткости этот набор программ буду называть просто – инструментарий.

Микроконтроллеры STM32 пользуются большой популярностью у разработчиков, но предлагаемые производителем микроконтроллеров примеры программирования, состоят из проектов для коммерческих сред разработки: Keil, IAR, Atollic TrueSTUDIO.

Многие программы в составе инструментария имеют аналоги для различных систем Unix. К примеру, информацию по программе редактора связей, можно найти в книгах по Unix 20-летней давности. Информация об отладчике gdb в составе Linux, на 100 процентов подходит и для порта отладчика GDB из инструментария.

Форматы и секции

Дистрибутив инструментария можно скачать с сайта http://launchpad.net/gcc-arm-embedded. Установка не требует специальных усилий, в Linux устанавливается копированием. Достаточно отредактировать переменную PATH, добавив путь к каталогу, где находится arm-none-eabi-gcc и сопутствующие утилиты. Скомпилированные порты аналогичных компиляторов имеются и в репозиториях Debian и Ubuntu, про остальные клоны Linux, не могу ничего сказать. Правильность установки проверяется командой:

```
arm-none-eabi-gcc -v
```

Компилятор GCC пришёл из мира UNIX . В UNIX выполнение программы начинается с создания в памяти её образа и связанных с процессом структур ядра ОС, а затем инициализации и передаче управления инструкциям программы. Программа может быть загружена в память, если она имеет специальный двоичный формат. В настоящее время большинство систем UNIX использует формат выполнения и компоновки ELF (Excecutable and Linking Format). Стандарт ELF приняли многие производители. Этот стандарт нашёл применение и в мире портативных и встраиваемых систем.

Важное понятие – ABI (Application Binary Interface). В прежние времена тема ABI, не часто затрагивалась, так как как не было такого разнообразия вычислительных архитектур и операционных систем и их комбинаций. Практическая задача, нужно скомпилировать программу под смартфон, или иное портативное устройство. Совместимости на уровне API будет недостаточно, для работы программы, необходима совместимость ABI. Вопрос легко решается, если у Вас есть родной инструментарий (toolchain), который позволяет создать программу для целевой системы. Кроме того, даже имея подходящий инструментарий, нужно ещё знать, как его

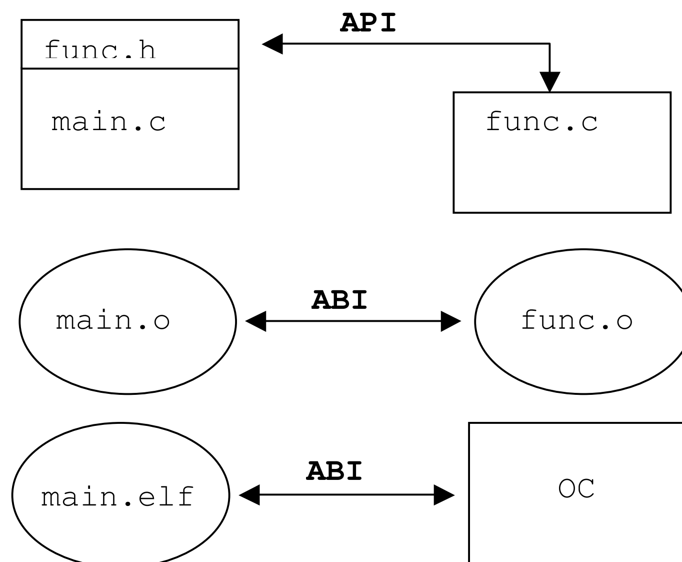

Рисунок 1

настроить. Для решения подобного класса задач требуется понимание принципов ABI.

Интерфейс – это набор соглашений по взаимодействию. API – интерфейс для доступа к ресурсам программ, написанных на языке высокого уровня или на ассемблере. Для программ на языке С - это чаще всего заголовочные файлы. ABI – это интерфейс взаимодействия объектного файла с компоновщиком, а так же интерфейс взаимодействия исполнимого файла с операционной системой. Выражаясь точнее, порядок загрузки операционной системой исполнимого файла в память. На рисунке 1 поясняются отличия ABI от API.

Интерфейс уровня ABI, находится ниже API. Подробности ABI нужно знать разработчикам компиляторов, компоновщиков и операционных систем. Но некоторые знания важны и для программистов встраиваемых систем.

Архитектура современной вычислительной системы – это иерархия систем нескольких уровней абстракции. Программа претерпевает ряд трансформаций, от набора текстовых файлов, до процесса в памяти. Для идентификации переменных и команд на разных этапах жизненного цикла программы используются символьные имена, виртуальные адреса и физические адреса. Символьные имена присваивает программист при написании программы. Использовать символические имена и адреса вместо двоичных и восьмеричных намного удобнее. Виртуальные адреса вырабатывает транслятор. Поскольку во время трансляции не известно, в какое место памяти будет загружена программа, то виртуальным адресам присваиваются некоторые условные, промежуточные значения или они считаются не имеющими определённого значения. Физические адреса - область памяти, где в действительности будут расположены переменные и программы. Очень давно все объекты программ, размещаемых в памяти, стали делить на три категории, код, данные и не инициализированные данные. В соответствии с этим память стали делить на 3 области, где эти объекты размещали. В разных системах, эти области стали называть секциями или сегментами. Объектный файл делится на секции, содержимое некоторых секций переносится в память целевой системы. Кроме основных секций программы, в объектном файле имеется ряд секций выполняющих вспомогательные функции. Информация в этих секциях содержит указания для компоновщика, отладчика и загрузчика операционной системы.

Секция text – исполнимый код, только для чтения. Секция data – инициализированные данные или секция, для чтения и записи, но не для выполнения. Секция bss – секция не инициализированных данных, допустимо чтение данных и запись в секцию.

На рисунке 2 приведена схема связывания двух объектных файлов, полученных в результате трансляции.

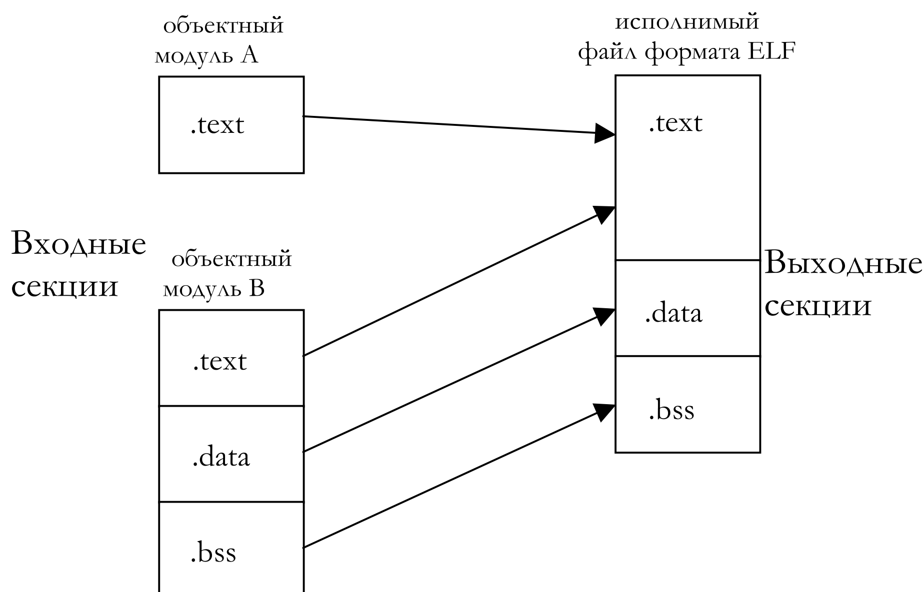

Рисунок 2

Результатом связывания будет исполнимый файл. В модуле A имеется только секция .text , а в модуле B , секции .text , .data и .bss. В процессе связывания выделяют 2 основных процесса, размещение (relocation) и разрешение символов (symbol resolution).

В данном контексте, под символом подразумевается объект программы, имеющий текстовое наименование. Переменная, определённая в тексте программы, имеет символьное обозначение, а так же этой переменной будет отведён участок памяти. В других местах программы, имеются ссылки на эту переменную, и везде эта переменная идентифицируется по символьному названию. Ясно, какую важную роль играет символьное название в процессе компоновки.

Процесс связывания считается выполненным, когда разрешены все символы. Иначе говоря, когда нет ссылок на символы, расположение которых не удалось определить.

Первая программа

Ассемблерная программа определяет не только машинные команды, которые нужно выполнять процессору, но и команды, которые нужно выполнять самому ассемблеру (например, выделить немного памяти или выдать листинг). Управляют процессом ассемблирования директивы ассемблера. Директивы начинаются с точки. Ниже приведён ассемблерный текст.

```
1 .syntax unified
2 .cpu cortex-m3
3 .thumb
4 .equ  STACKINIT, 0x20002000
5  .text
6  .global _start                  @
7       .word STACKINIT         @ stack pointer
8       .word _start + 1        @ reset
9       .word _nmi_handler + 1  @ vectors
10      .word _hard_fault  + 1
11      .word _memory_fault + 1
12      .word _bus_fault + 1
13      .word _usage_fault + 1
14 _start:              @ start of execution examples
15    mov r0, #5           @  5 -> r0
16    adr r1, value        @  address const value -> r1
17    ldr r2, [r1]         @  mem_32[r1] -> r2
18    add r1, r2, r0      @  r2 + r0 , -> r1
19    push {r1}            @  r1 -> stack
20    mov  r1, #0          @  0  -> r1
21   pop  {r1}            @  stack -> r1
22   blx _start           @  this bad command

23 .ascii "example" @ data in section code
24 .align  0x0     @
25     _dummy:          @ loop for handlers
26     _nmi_handler:
27     _hard_fault:
28     _memory_fault:
29     _bus_fault:
30     _usage_fault:
31          add r0, 1
32          add r1, 1
33         b _dummy
34 .align  0x0
35 value:  .word 0x11223344
 36 .end
```

По ходу изложения, список директив ассемблера будет расширяться, будут вводиться новые понятия.

В первой строке директива, задающая поддержку синтаксиса стандарта UAL (Unified Assembler Language), по-русски – это унифицированный язык ассемблера. Разработка компании ARM. Вторая директива задаёт целевой процессор. Третья – указывает на необходимость генерировать команды множества Thumb-2. Этот набор команд является расширением набора Thumb, впервые введённого в архитектуре стандарта V4T. Набор команд Thumb-2 введён начиная с архитектуры V6 и выше. Альтернативная директива `.code 16` , заставит генерировать 2-х байтные команды множества Thumb. В 4-ой строке директива `.equ` присваивает значение строке символов. Ассемблер подставит числовое значение там, где будет задан символ. Символу `_start` , присвоено свойство - глобальный, он будет доступен из других модулей, в процессе ассемблирования ему будет присвоено значение, а точнее адрес.

Директива `.text` указывает на начало секции кода. В программе задана только одна секция кода.

В 7-ой строке, директива `.word` выделяет 4 байта и в выделенном пространстве размещает значение, присвоенное символу `STACKINIT`. Для Cortex-M3 в начале секции кода, должно быть значение указателя стека. Не трудно заметить, что указывается в область RAM.

Всё что после директивы `.text` будет направлено в секцию кода, по сути, будет размещаться в ROM, до тех пор, пока в тексте программы не встретится директива `.data` или `.bss` , тогда ассемблер направит свой вывод в соответственные директивам секции.

Далее следует таблица первых 6-ти системных исключений. В строке 8 задан адрес вектора сброса, в этом месте памяти расположится значение метки `_start` . Процессор перейдёт по этой метке по сбросу. Младший бит каждого вектора указывает, в каком состоянии должен выполняться соответствующий обработчик. Будет процессор выполнять команды обработчика в состоянии ARM или THUMB. Поскольку процессор Cortex-M3 поддерживает только команды Thumb, младшие биты всех векторов исключений должны быть установлены в 1. Каждый вектор в программе увеличен на 1.

Целая серия векторов перехода для различных исключений указывают на один адрес, на метку `_dummy` . Там имеется простейшая ловушка, бесконечный цикл.

В строке 23 задана строковая константа. В секции кода не возбраняется размещать константы. Затем идёт директива `.align` , эта директива указывает ассемблеру на необходимость расположить следующие за константой команды на границе слова.

После сброса, CPU тактируется от HSI (internal high-speed oscilator). Код инициализации отсутствует. Источник синхронизации не настроен на максимальную частоту и доступ к флэш памяти не сконфигурирован оптимально. Но для учебных целей, существующая конфигурация вполне пригодна.

В строках с 15 по 22 задан некоторый набор команд процессора. В строке 22 стоит команда, которая для архитектуры Cortex-M3 не является корректной и вызывает генерацию исключения Usage Fault.

Ассемблируем командой:

```
arm-none-eabi-gcc -c -mthumb -mcpu=cortex-m3 -g -o ex1.o ex1.s
```

Программа arm-none-eabi-gcc – это программа драйвер или менеджер. Она управляет процессом компилирования и сборкой программ, вызывает необходимые программы, с нужными параметрами. Этот менеджер может применяться для компилирования программ на языке С , ассемблер и для компоновки.

Параметры командной строки:

`c` – задаёт необходимость получения только объектных файлов, не выполняется линковка,

`mthumb , mcpu` – задаёт формат кода для целевой системы,

`g` – указывает на необходимость генерации отладочной информации,

`o` – задаёт имя выходного файла. К слову сказать, параметры, задающие формат выходного кода, ARM или THUMB, не обязательны, так как этот формат задан в ассемблерном тексте.

После успешного ассемблирования, будет получен объектный файл ex1.o. Посмотреть имеющиеся символьные имена, или просто символы, в сгенерированном объектном файле можно командой:

```
arm-none-eabi-nm ./ex1.o
    00000040 t _bus_fault
    00000040 t _dummy
    00000040 t _hard_fault
    00000040 t _memory_fault
    00000040 t _nmi_handler
    20002000 a STACKINIT
    0000001c T _start
    00000040 t _usage_fault
    0000004c t value
```

В первом столбце идёт значение символа, затем тип символа и название.

t – локальный символ в секции .text,

T – глобальный символ в секции .text,

a – символ, имеющий абсолютное значение, при линковке его значение не меняется. Для символов типов t и T , значение – это адрес от начала секции. Эти адреса виртуальные, после линковки они могут претерпеть изменения, конечно кроме символа с абсолютным значением.

Посмотреть информацию о секциях в объектном файле, можно командой:

```
arm-none-eabi-size -A ./ex1.o
    ./ex1.o  :
    section           size   addr
   .text               80      0
   .data                0      0
   .bss                 0      0
   .ARM.attributes     33      0
   .debug_line         65      0
   .debug_info         66      0
   .debug_abbrev       20      0
   .debug_aranges      32      0
    Total              296
```

Секции `.data` и `.bss` пустые. Так же имеются секции для целей отладки, это результат действия флага `-g` при ассемблировании. Секция `.ARM.attributes` и секции отладки, выполняют служебные функции и в памяти целевой системы не будут размещены.

Компоновка

Компоновка – процесс связывания и преобразования объектных файлов. Иногда применяют термин линковка и программу называют, линковщиком или линкером. Связывание можно выполнить командой:

```
arm-none-eabi-ld  -Tl.ld ex1.o -o ex1.elf
```

Параметр `-T` задаёт скрипт линкера. Это текстовый файл, на специальном языке управления процессом линковки. Параметр `-o` задаёт имя выходного файла, по умолчанию генерируется файл формата ELF исполнимый. Среди параметров задаётся список объектных файлов подлежащих связыванию.

Скрипт линкера:

```
1 SECTIONS {
2          . = 0x00000000;
3          .text : {
4            ex1.o (.text);
5                  }
6         }
```

Важнейшая команда - `SECTIONS` . Именно эта команда определяет, как будут связаны секции и по каким адресам размещены. В командном файле линкера может располагаться только одна команда `SECTIONS`. Напомню, что секции объектного файла, называются входными секциями, а секции исполнимого файла – это выходные секции.

Специальная переменная линкера «.» («точка») всегда содержит текущее значение адреса. Во второй строке скрипта, текущее значение адреса устанавливается в ноль. Хотя это не обязательно, так как по умолчанию начальное значение этой переменной ноль. Но допустимо присваивать переменной линкера произвольное значение. В 3-ей строке задано имя выходной секции `.text` , именно в эту выходную секцию будет загружаться информация. С учётом предыдущего оператора, начало секции `.text` , будет располагаться с нулевого адреса. В 4-ой строке указано, из какого источника, из объектного файла ex1.o переместить секцию `.text`. Так же линкер будет выполнять и другие операции, например настройку адресов. Так как название объектного файла указано в скрипте, то задавать это название в параметрах командной строки, при вызове линкера, не обязательно. Возможен и такой вариант:

```
1 SECTIONS {
2          . = 0x00000000;
3          .text : {
4            * (.text);
5                  }
6         }
```

Здесь название конкретного файла заменено шаблоном «*». Нужно понимать так, секции всех объектных файлов заданных в командной строке, перенести в выходную секцию.

Результатом процесса связывания станет исполнимый файл формата ELF. Получить бинарный образ для записи во флэш, можно командой:

```
arm-none-eabi-objcopy -O binary ./ex1.elf ./ex1.bin
```

Шестнадцатеричный дамп бинарного файла ex1.bin:

```
     hd ./ex1.bin
00000000  00 20 00 20 1d 00 00 00  41 00 00 00 41 00 00 00  |. . ....A...A...|
00000010  41 00 00 00 41 00 00 00  41 00 00 00 4f f0 05 00  |A...A...A...O...|
00000020  0f f2 28 01 0a 68 02 eb  00 01 02 b4 4f f0 00 01  |..(..h......O...|
00000030  02 bc ff f7 f4 ef 65 78  61 6d 70 6c 65 00 00 bf  |......example...|
00000040  00 f1 01 00 01 f1 01 01  ff f7 fa bf 44 33 22 11  |............D3".|
00000050
```

В начале можно увидеть значение указателя стека, далее вектор сброса и значения остальных векторов прерывания. При анализе надо учесть обратный порядок следования байтов в бинарном файле (little endian). После заданной в программе строки текста, идут 2 нулевых байта, результат действия директивы `.aling` . Содержимое файла – двоичный код без служебной информации. Записать бинарный файл в микроконтроллер можно командой:

```
st-flash  write  v1  ./ex1.bin  0x8000000
```

Отладчик и сервер отладки

Все примеры программ будут запускаться на фирменной оценочной плате - STM32VLDISCOVERY. Что особенно ценно, в составе дешёвой платы, кроме целевого процессора, имеется аппаратный отладчик, с помощью которого можно вести трассировку кода. Удачный маркетинговый ход фирмы STM. Для отладки не нужно покупать дорогостоящий отладчик.

В код аппаратного отладчика Stlink v1 , дополнительно встроена программа, выполняющая функции флэш диска, устройство USB класса MSD. Для нормальной работы требуется настроить Linux так, чтобы MSD устройство с кодами 0483:3744 игнорировалось, так как в реализации имеются ошибки. Для этого модуль ядра usb_storage должен запускаться с параметром quirks=483:3744:i. Это можно выполнить различными способами. Можно из командной строки, размонтировать все флэш диски, выгрузить модуль usb_storage , а затем загрузить с нужными параметрами:

```
modprobe -r usb_storage
modprobe usb_storage quirks=483:3744:i .
```

Либо запускать ядро Linux , задав параметры. В моём случае, в файле /etc/default/grub, отредактировал строку :

```
GRUB_CMDLINE_LINUX_DEFAULT="usb-storage.quirks=483:3744:i".
```

А затем обновил update-grub.
Параметры модуля usb_storage можно посмотреть командой:

```
cat /sys/module/usb_storage/parameters/quirks .
```

В файле должно быть:

```
483:3744:i .
```

Когда модуль usb_storage будет загружен соответственным образом, можно двигаться дальше. Для отладки требуется наличие программы stlink. Это сервер отладки, он должен подключаться к аппаратному отладчику через USB и работать в фоновом режиме. А к этому серверу подключается клиент, в данном случае это будет arm-none-eabi-gdb, входящий в пакет инструментария.

В случае с Ubuntu, нужны библиотеки libusb-1.0 . Установить библиотеки можно командой:

```
sudo apt-get -y install libusb-1.0-0-dev
sudo apt-get -y install libusb-1.0
```

С помощью клиента git скачиваем исходный текст из репозитория:

```
git clone https://github.com/texane/stlink.git
```

В каталоге с исходным текстом, запускаем:

```
./autogen.sh
./configure
make  .
```

После успешной компиляции, в наличии файлы st-util и st-flash. Для того, чтобы можно было работать с отладчиком из учётной записи с обычными правами, требуется скопировать файл 49-stlinkv1.rules , находящийся в каталоге с исходным текстом, в каталог ./etc/udev/rules.d и выполнить:

```
udevadm control --reload-rules
udevadm trigger .
```

Либо перезагрузиться.

Подключаем отладчик и запускаем :

```
st-util  -1
```

Если всё сделано правильно, то на экране можно видеть что-то подобное:

```
            alex@big:/usr/local$ st-util
      2015-06-23T12:50:25 INFO src/stlink-usb.c: -- exit_dfu_mode
      2015-06-23T12:50:25 INFO src/stlink-common.c: Loading device parameters....
      2015-06-23T12:50:25 INFO src/stlink-common.c: Device connected is: F1
Medium/Low-density Value Line device, id 0x10016420
      2015-06-23T12:50:25 INFO src/stlink-common.c: SRAM size: 0x2000 bytes (8
KiB), Flash: 0x20000 bytes (128 KiB) in pages of 1024 bytes
      2015-06-23T12:50:25 INFO gdbserver/gdb-server.c: Chip ID is 00000420, Core
ID is  1ba01477.
      libusb_handle_events() | has_error
      [!] send_recv
2015-06-23T12:50:25 INFO gdbserver/gdb-server.c: Listening at *:4242…
```

Сервер отладки будет ждать подключений на порт 4242.

Хочется добавить, что можно использовать и широко известный сервер отладки OpenOCD.(Open On-Chip Debugger), но он имеет несколько усложнённую настройку, хотя вполне работоспособен.

Отладка

Для отладки мной используется целая цепочка средств: аппаратный отладчик, сервер отладки и порт программы gdb (GNU Debugger) из инструментария.

Как сообщалось ранее, для успешной работы gdb требуется, чтобы был запущен в фоновом режиме сервер отладки st-util. Программа gdb взаимодействует с сервером отладки, используя механизм межпроцессного обмена – сокеты. Этот механизм позволяет обмениваться данными процессам запущенным не только на одном компьютере, но и на разных компьютерах в сети.

В микроконтроллер загружена программа, плата подключена к компьютеру. В фоновом режиме запущена программа st-util с нужными параметрами командной строки, она видит микроконтроллер и ждёт подключения на порт 4242. Если все эти условия выполнены, то можно запускать gdb:

```
  arm-none-eabi-gdb
This GDB was configured as "--host=i686-linux-gnu --target=arm-none-eabi".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb)
```

Отладчик готов и ждёт дальнейших команд. Подключаемся к серверу:

```
(gdb) target extended localhost: 4242
```

Загружаем файл ex1.elf с отладочной информацией:

```
(gdb) symbol-file ./ex1.elf
```

Проверяем содержимое регистра pc:

```
       (gdb) info reg pc
       pc             0x1c      0x1c
```

По сбросу микроконтроллер перешёл на метку `_start` . Листинг программы:

```
(gdb) list _start
10            .word _hard_fault  + 1
11            .word _memory_fault + 1
12            .word _bus_fault + 1
13            .word _usage_fault + 1
14     _start:                @ start of execution examples
15           mov r0, #5          @  5 -> r0
16           adr r1, value       @  address const value -> r1
17           ldr r2, [r1]        @  mem_32[r1] -> r2
18           add r1, r2, r0      @  r2 + r0 , -> r1
19           push {r1}           @  r1 -> stack
```

Дизассемблируем часть кода с начального адреса:

```
(gdb) disassemble
Dump of assembler code for function _start:
=> 0x0000001c <+0>:mov.w  r0, #5
   0x00000020 <+4>:addw   r1, pc, #40  ; 0x28
   0x00000024 <+8>:ldr    r2, [r1, #0]
   0x00000026 <+10>:      add.w r1, r2, r0
   0x0000002a <+14>:      push  {r1}
   0x0000002c <+16>:      mov.w r1, #0
   0x00000030 <+20>:      pop   {r1}
   0x00000032 <+22>:      blx   0x1c <_start>
   0x00000036 <+26>:      stclvs 8, cr7, [r1, #-404]!     ; 0xfffffe6c
   0x0000003a <+30>:      rsbeq r6, r5, r0, ror r12
   0x0000003e <+34>:      nop
End of assembler dump.
```

Выполняем одну инструкцию:

```
(gdb) stepi
16    adr r1, value        @  address const value -> r1
```

Смотрим содержимое регистра `r0` :

```
(gdb) info reg r0
r0             0x5  5
```

Смотрим память, 7 байт с адреса 0x36 в формате hex:

```
(gdb) x/7xb 0x36
0x2c <stop+4>:     0x65   0x78   0x61   0x6d   0x70   0x6c   0x65
```

Количество аппаратных точек останова может быть ограничено и зависит от аппаратуры отладчика и микроконтроллера. Аппаратные точки останова можно ставить без проблем в области ROM. Возможна установка и удаление точек останова без прерывания сеанса отладки. Ставим аппаратную точку отладки в строку 18 ассемблерного текста:

```
(gdb) hbreak ex1.s:18
Hardware assisted breakpoint 1 at 0x26: file ex1.s, line 18.
```

Запускаем программу до точки останова:

```
(gdb) continue
Continuing.

18          add r1, r2, r0     @  r2 + r0 , -> r1
```

Получаем информацию об имеющихся точках останова:

```
        (gdb) info breakpoints
Num     Type           Disp Enb Address    What
1       hw breakpoint  keep   0x00000026 ex1.s:18
      breakpoint already hit 1 time
```

Удаляем точку останова:

```
(gdb) del break 1
```

В интернете можно найти документацию по отладчику GDB на русском языке. Существует множество программ предоставляющих более удобный интерфейс пользователя (front-end gdb), нежели командная строка: Insight , Eclipse CDT , Code::Blocks.

Связывание двух модулей

Большинство программ создаётся из нескольких модулей. В нашем следующем примере, простейшая программа создаётся из 2-х модулей, которые затем связываются в один бинарный файл. Текст программы 1-го модуля из файла ex2_m.s:

```
@ main program example 2
    .syntax unified
    .cpu cortex-m3
    .thumb
 .equ  STACKINIT, 0x20002000
       .text
 .global start
    .word STACKINIT         @ stack pointer
    .word start + 1         @ reset
    .word _nmi_handler + 1  @
    .word _hard_fault  + 1  @
    .word _memory_fault + 1 @
    .word _bus_fault + 1    @
    .word _usage_fault + 1  @
 start:
    ldr r0 , =bar          @ load begin of array
    ldr r1 , =ear          @ load end of array
    bl  sum                @ call proc
    ldr r1 , =var1         @ set address var1
    strb r3 , [r1]         @ save in memory
    mov r3, #1
    ldr r1 , =var2
    strb r3 , [r1]
 stop:                     @ trap
       b stop
        .section .rodata
 bar:  .byte 1, 2, 3, 4, 5  @ array of numbers
 ear:
        .text
     _dummy:
     _nmi_handler:
     _hard_fault:
     _memory_fault:
     _bus_fault:
     _usage_fault:
        add r0, 1
        add r1, 1
        b _dummy
  .end
```

Массив констант выделен в отдельную секцию, применена директива `.section .rodata` . Начало массива отмечено меткой `bar` , а конец отмечен меткой `ear` . Секция `.rodata` будет размещена в ROM. Адреса начала и конца массива передаются в процедуру `sum` через регистры. В этой процедуре выполняется суммирование элементов массива. Она определена в другом модуле. Результат возвращается в регистре r3 и сохраняется в переменных `var1` и `var2` . Эти переменные определены в секции `.bss` , а сама секция размещена в RAM.

После ассемблирования ex2_m.s , смотрим секции и символы объектного файла:

Символы	Секции
U sum U var1 U var2 00000000 r bar 00000005 r ear 0000001c T start 00000030 t stop 00000034 t _bus_fault 00000034 t _dummy 00000034 t _hard_fault 00000034 t _memory_fault 00000034 t _nmi_handler 00000034 t _usage_fault 20002000 a STACKINIT	section size addr .text 0x50 0x0 .data 0x0 0x0 .bss 0x0 0x0 .rodata 0x5 0x0 .ARM.attributes 0x21 0x0 .debug_line 0x4a 0x0 .debug_info 0x44 0x0 .debug_abbrev 0x14 0x0 .debug_aranges 0x20 0x0 Total 0x138

Секция `.rodata` имеет размер 5 байт. Символы типа U – это неопределённые символы, адреса не имеют. Переменные `var1` и `var2` , а так же процедура `sum` , определены в другом модуле, поэтому имеют тип U. Символы типа r – символы данных определённых в ROM, для чтения, но не для исполнения.

Текст второй программы из файла ex2_s.s:

```
@ subroutine example 2
    .syntax unified
    .cpu cortex-m3
    .thumb
 .global sum, var1, var2
     .text
  sum:
       mov r3, #0
 loop:       ldrb r2, [r0], #1
             add  r3, r2, r3
             cmp  r0, r1
             bne  loop
             mov  pc, lr
    .bss
      var1:  .skip 1
      var2:  .skip 1
    buffer:  .skip 32 0
 .end
```

Метка sum задана как глобальная, это условие видимости метки из других модулей. Это же относится и к var1 , var2 . Для задания переменных в секции .bss используется директива .skip . Эта директива позволяет определять буфер произвольного размера. Переменные var1 и var2 по байту и buffer в 32 байта. Таблица символов и секций:

Символы	Секции
00000000 T sum 00000000 B var1 00000001 B var2 00000002 b buffer 00000004 t loop	section size addr .text 0x12 0x0 .data 0x0 0x0 .bss 0x22 0x0 .ARM.attributes 0x21 0x0 .debug_line 0x3a 0x0 .debug_info 0x44 0x0 .debug_abbrev 0x14 0x0 .debug_aranges 0x20 0x0 Total 0x107

Символ типа B – глобальный символ в области неинициализированных данных, именно там расположены var1 и var2. Символ типа b – локальный символ. Получаем исполнимый файл командой:

```
arm-none-eabi-ld  -Tl.ld ex2_m.o ex2_s.o -o ex2.elf
```

Текст скрипта линковки:

```
MEMORY {
       ROM (RX): ORIGIN = 0, LENGTH = 128K
       RAM (RW): ORIGIN = 0x20000000 LENGTH = 8K
       }
SECTIONS {
          .text : {
            * (.text);   } > ROM
          .rodata : {
            * (.rodata); } > ROM
          .bss    :  {
            * (.bss);
                       } > RAM }
```

Здесь приведен пример с использованием команды `MEMORY`, как и команда `SECTIONS`, это команда может встречаться единожды в скрипте. В фигурных скобках указаны названия областей памяти и атрибуты этих областей. Как легко догадаться атрибут `RW` - это область памяти с доступом чтения-записи, а атрибут `X` , означает возможность выполнения команд. Слово `ORIGIN` – начало области, `LENGTH` – её размер. Знаком «>» указывается, в какой области памяти расположить секции.

После секции кода следует секция данных «только чтение», а затем секция не инициализированных данных, располагающаяся в RAM.

На результат линковки влияет последовательность, в которой указаны объектные файлы, из которых собирается программа. Если порядок файлов поменять в командной строке, то результат окажется не работоспособным. Ниже приведены две таблицы символов исполнимых ELF файлов, полученных при разном порядке задания файлов в команде линковки.

Правильно скомпонованный	Неправильно скомпонованный
0000001c T start	00000000 T sum
00000030 t stop	00000004 t loop
00000034 t _bus_fault	00000030 T start
00000034 t _dummy	00000044 t stop
00000034 t _hard_fault	00000048 t _bus_fault
00000034 t _memory_fault	00000048 t _dummy
00000034 t _nmi_handler	00000048 t _hard_fault
00000034 t _usage_fault	00000048 t _memory_fault
00000050 T sum	00000048 t _nmi_handler
00000054 t loop	00000048 t _usage_fault
00000062 r bar	00000064 r bar
00000067 r ear	00000069 r ear
20000000 B var1	20000000 B var1
20000001 B var2	20000001 B var2
20000002 b buffer	20000002 b buffer
20002000 a STACKINIT	20002000 a STACKINIT

Это понятно, в каком порядке подаются файлы, в таком порядке и размещаются их секции. В скрипте можно явно указать порядок линковки таким образом:

```
MEMORY {
       ROM (RX): ORIGIN = 0, LENGTH = 128K
       RAM (RW): ORIGIN = 0x20000000 LENGTH = 8K
       }
SECTIONS {
          .text : { ex2_m.o (.text);
                    ex2_s.o (.text);  } > ROM
          .rodata :  {  * (.rodata); } > ROM
          .bss    :  {  * (.bss);  } > RAM }
```

Другой вариант, выделить код, который требуется разместить по абсолютному адресу в отдельную секцию, и расположить эту секцию по абсолютному адресу. Но как это реализовать, станет ясно из последующего изложения.

Правильность функционирования этого примера можно проверить, загрузив в микроконтроллер и прогнав в отладчике.

Иногда при компоновке, выдаётся сообщение об ошибке, линковщик не может разрешить, переменную или функцию. В этом случае, нужно определить файл исходного текста, где этот символ определён, а затем скомпилировать этот модуль, и включить в процесс сборки.

LMA и VMA

В этом примере рассмотрим работу с секцией инициализированных данных. Организация этой секции важна для связывания модулей написанных на языках высокого уровня. На рисунке 3 представлено, как располагаются секции, когда требуется задать секцию инициализированных данных. Необходимость определения инициализированных данных можно обойти, создав необходимые константы, которые будут размещены в ROM, а затем в программе будут

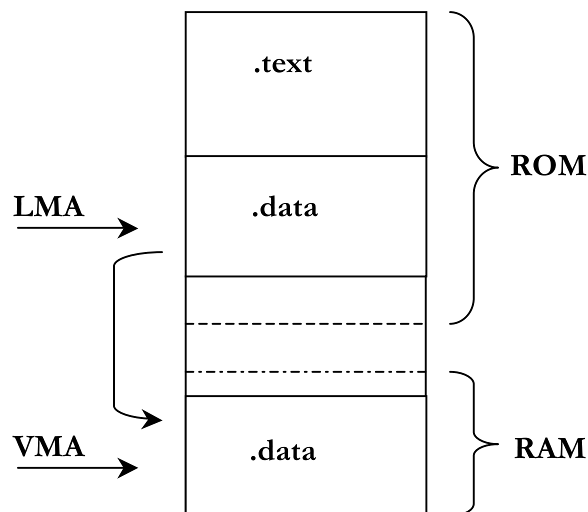

Рисунок 3

копироваться в область неинициализированных данных (секцию .bss). Но для поддержки языков высокого уровня требуется другая методика.

Секцию .data размещают в ROM, а затем, её содержимое переносится в RAM . Таким образом, перед выполнением прикладной программы, нужно выполнить некоторый код инициализации, а так же задать соответственные команды в скрипте линковки. Каждый символ в секции .data будет иметь два различных адреса. Один адрес в области ROM - LMA (Load Memory Address). А другой адрес в RAM – VMA (Virtual Memory Address).

Конечная программа будет работать с адресами VMA.

Скрипт компоновщика приведён ниже:

```
MEMORY {  ROM (RX) : ORIGIN = 0x0 LENGTH = 128K
          RAM (RW) : ORIGIN = 0x020000000 LENGTH = 8K
       }
SECTIONS { .text  :      {
            * (.text);   } > ROM
          .rodata :      {
            * (.rodata);
            flash_sd = .; } > ROM
          .bss :      {
            * (.bss); } > RAM
          .data  : AT (flash_sd) {
              ram_sd = . ;
            * (.data);
              ram_ed = . ;        } > RAM
          d_size = SIZEOF(.data) ; }
```

Первыми идут секции кода. В стандарте языка линкера, предусмотрена возможность задания программистом символов хранящих адреса и константы. Значения этих символов доступны из программного кода. В скрипте определяется символ `flash_sd` , он хранит адрес конца секции констант и начало области, где будут размещаться символы секции .data в ROM. Символы `ram_sd` и `ram_ed` , содержат адреса начала и конца секции .data в RAM. Символ `d_size` хранит размер секции `.data` . Для определения размера секции `.data` использовалась стандартная функция `SIZEOF` . В коде инициализации, значения этих символов будут использоваться для копирования секции `.data` из ROM в RAM. Ключевое слово `AT` определяет адрес загрузки секции в ROM. Линковщик расположит её за секцией `.rodata` в ROM, в то время как адреса символов принадлежащих этой секции, будут соответствовать расположению в RAM. Чтобы всё стыковалось, и программа нормально работала, содержимое секции из ROM нужно скопировать в RAM программным кодом инициализации. Текст на ассемблере приведён ниже.

```
@ example 3
    .syntax unified
    .cpu cortex-m3
    .thumb
 .equ  STACKINIT, 0x20002000
  .section .text
   .type start , %function
   .type _nmi_handler , %function
   .type _hard_fault , %function
   .type _memory_fault , %function
   .type _bus_fault , %function
   .type _usage_fault , %function
     .word STACKINIT        @ stack pointer value
     .word start            @ reset
     .word _nmi_handler
     .word _hard_fault
     .word _memory_fault
     .word _bus_fault
     .word _usage_fault
 start:
   ldr r0, =flash_sd  @ copy section data
   ldr r1, =ram_sd
   ldr r2, =d_size
 copy:
   ldrb r4, [r0], #1
   strb r4, [r1], #1
   subs r2, r2, #1
   bne copy          @ end copy section data

   ldr r0, =var1
   ldr r1, =var2

  ldr r2, [r0]
  ldr r3, [r1]
  add r4, r2, r3
  ldr r0, =res
  str r4, [r0]
     stop:   b stop    @ trap
  _dummy:              @ handlers
  _nmi_handler:
  _hard_fault:
  _memory_fault:
  _bus_fault:
  _usage_fault:   add r0, 1
                  add r1, 1
                  b _dummy
 .section .rodata
   .word 0x55555555 , 0xAAAAAAAA , 0x55555555 , 0xAAAAAAAA
 .section .data
     var1 :    .word 11
     var2 :    .word 22
     res :     .word 0
 .section .bss
     buf :    .skip 32 * 4
 .end
```

В этом примере используется директива .type . В данном случае она применяется для задания типа метки программного кода. При использовании этой директивы, необходимость в увеличении на единицу значения адреса, как в предыдущих примерах, отпадает.

В начале программы команды, выполняющие копирование секции инициализированных данных, а затем простейший код, выполняющий суммирование значений 2-х переменных.

Реальные значения символов определённых в скрипте линкера, можно узнать, воспользовавшись утилитой arm-none-eabi-nm, из инструментария.

Для контроля результатов компоновки, выполняем команду:

```
     arm-none-eabi-objdump -h ./ex3.elf
Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .text         00000068  00000000  00000000  00008000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
  1 .rodata       00000010  00000068  00000068  00008068  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .bss          00000080  20000000  20000000  00010000  2**0
                  ALLOC
  3 .data         0000000c  20000080  00000078  00008080  2**0
                  CONTENTS, ALLOC, LOAD, DATA
  4 .ARM.attributes 00000021  00000000  00000000  0000808c  2**0
                  CONTENTS, READONLY
  5 .debug_line   0000004e  00000000  00000000  000080ad  2**0
                  CONTENTS, READONLY, DEBUGGING
  6 .debug_info   00000042  00000000  00000000  000080fb  2**0
                  CONTENTS, READONLY, DEBUGGING
  7 .debug_abbrev 00000014  00000000  00000000  0000813d  2**0
                  CONTENTS, READONLY, DEBUGGING
  8 .debug_aranges 00000020  00000000  00000000  00008158  2**3
                  CONTENTS, READONLY, DEBUGGING
```

Кроме размеров секций, дамп вывода содержит VMA и LMA адреса начала секций. У секции .data , VMA и LMA адреса не совпадают. Дополнительно, эта утилита выводит информацию об атрибутах секций. Атрибут LOAD (loadable section) – содержимое секции будет загружено в память целевой системы. Атрибут ALLOC (allocatable section) – секция будет занимать место в адресном пространстве во время выполнения. Секция bss не имеет данных для загрузки в память целевой системы, но имеет размер. Поэтому она имеет только атрибут ALLOC. Секция ARM.attributes – выполняет служебные функции и не имеет данных для размещения в памяти целевой системы, поэтому не имеет атрибутов ALLOC и LOAD. Секции отладки так же не размещаются в памяти целевой системы, эти секции использует отладчик. Значения остальных атрибутов очевидны.

Программа на С

В этом примере рассмотрим компиляцию и линковку программы на С. Выше говорилось, что для запуска программ написанных на языке высокого уровня требуется некоторая предварительная работа по подготовке памяти. Кроме подготовки секции .data , необходимо заполнить секцию .bss нулями. Загрузочный образ будет собираться из 2-х модулей, на ассемблере и на С. На ассемблере написан код инициализации, текст программы, файл ex4.s :

```
@ example 4
    .syntax unified
    .cpu cortex-m3
    .thumb
 .equ  STACKINIT, 0x20002000
  .section .text
   .type start , %function
   .type _nmi_handler , %function
   .type _hard_fault , %function
   .type _memory_fault , %function
   .type _bus_fault , %function
   .type _usage_fault , %function
     .word STACKINIT        @ stack pointer value
     .word start            @ reset
     .word _nmi_handler
     .word _hard_fault
     .word _memory_fault
     .word _bus_fault
     .word _usage_fault
start:   ldr r2, =d_size       @ d_size -- size of .data
         cbz r2, fill_0        @ if (r2 == 0) goto fill_0
         ldr r0, =flash_sd
         ldr r1, =s_data
move:    ldrb r4, [r0], #1
         strb r4, [r1], #1
         subs r2, r2, #1
         bne move             @ end copy section
fill_0:  ldr r2, =b_size      @ b_size --  size of .bss
         cbz r2, go           @ if (b_size == 0) goto
         ldr r0, =s_bss
         mov r4, #0
fill_1:  strb r4, [r0], #1
         subs r2, r2, #1
         bne fill_1
go:      bl main
_dummy:                 @ handlers
  _nmi_handler:
  _hard_fault:
  _memory_fault:
  _bus_fault:
  _usage_fault:   add r0, 1
                  add r1, 1
                  b _dummy
 .end
```

Скрипт линкера:

```
MEMORY {
        ROM (RX) : ORIGIN = 0x0 LENGTH = 128K
        RAM (RW) : ORIGIN = 0x020000000 LENGTH = 8K
       }
SECTIONS {
          .text  :        {
            * (.text);
                          } > ROM
          .rodata :       {
            * (.rodata);
            flash_sd = .;
                          } > ROM
          .bss : { s_bss = . ;
            * (.bss);
                 } > RAM
          .data  : AT (flash_sd) {
              s_data = . ;
            * (.data);
                                 } > RAM
          d_size = SIZEOF(.data) ;
          b_size = SIZEOF(.bss) ;  }
```

В скрипте определено несколько символов. Символ `flash_sd` хранит адрес конца секции `.rodata`.

Начиная с этого адреса, будут размещаться данные секции `.data` в ROM. Именно в ROM! Адрес начала секции `.bss` в `s_bss` , а секции `.data` в `s_data` . Символы `d_size` и `b_size` , хранят размеры секций в байтах. В программе на ассемблере эти символы используются для копирования данных и подготовки секции `.bss` . После инициализации, вызывается подпрограмма `main` из модуля на C. Текст на C, файл ex4_1.c:

```
 int src[]={1,2,3,4,5,6,7,8,9,10};
int dst[10];
const int n = 10;
void main()
{ int i;
    for (i=0;i<n;i++)   {
       dst[i] = src[i]; }
     while(1);
}
```

Стандарт C поддерживает 4 спецификатора класса памяти: extern, static, register, auto. Если класс не определён явно, то по умолчанию принимается класс extern для объектов, описанных вне функций, и класс auto в пределах описания функций. Автоматические объекты (auto) существуют только во время выполнения данного блока и при выходе из блока теряют свои значения и освобождают занимаемую память. Классы static и extern касаются объектов, существующих в течение всего времени выполнения программы. Разница между ними в том, что объекты класса extern доступны из всех модулей, в то время как объекты класса static , доступны только в пределах того модуля, где они описаны. Это краткое напоминание об основах C.

Компилируем программу на C, командой:

```
arm-none-eabi-gcc  -mcpu=cortex-m3  -mthumb  -O0 -g  -c ./ex4_1.c -o ./ex4_1.o
```

Флаг `-O0` указывает компилятору, что не нужно выполнять оптимизацию программы, так как возможны трудности с отладкой. Смотрим символы полученного объектного файла:

```
arm-none-eabi-nm -t  x -n ./ex4_1.o
    00000000 T main
    00000000 R n
    00000000 D src
    00000028 C dst
```

Все символы глобальные. Тип C – общедоступные символы (common). Тип D – глобальные символы в секции `.data` . Если программа состоит из нескольких модулей компилируемых раздельно, и в этих модулях имеются символы типа C с одинаковыми именами, то эти символы будут ссылаться на один и тот же участок памяти. А если эти символы разного размера, то под символ будет выбран участок памяти максимального размера, из имеющихся участков. И что

важно, линковщик не сообщит, о том, что в разных модулях определены символы с одинаковыми именами.

Проведём эксперимент, создадим 2-ой модуль на С, в котором объявим не инициализированную переменную dst, с классом памяти extern , и меньшего размера. Текст 2-го модуля на С, из файла ex4_2.c совсем короткий, из одной строки.

```
int  dst ; //common symbol
```

Скомпилируем командой:

```
arm-none-eabi-gcc  -mcpu=cortex-m3  -mthumb  -O0 -g  -c ./ex4_2.c -o ./ex4_2.o
```

Скомпонуем командой:

```
arm-none-eabi-ld -v -Tl.ld ex4.o ex4_1.o ex4_2.o -o ex4.elf
```

Посмотрим символ `dst` в ELF файле:

```
arm-none-eabi-nm -t x -n ./ex4.elf|grep dst
    20000000 B dst
```

Как видим, в исполнимом файле создан символ типа B – не инициализированные данные в секции .bss. Символы типа C не создаются в исполнимых файлах ELF. Никаких предупреждающих сообщений не последовало об определении нескольких символов с одним именем в разных модулях. Нужно различать определение, от описания переменной. Описание переменной - это ссылка на участок памяти. Определение переменной – это выделение ей участка памяти.

Можно запретить компилятору создавать символы типа C (common). В параметрах компилятора задаётся опция `-fno-common`. В этом случае для глобальных, не инициализированных переменных, будут создаваться символы типа B. А при компоновке, линковщик сообщит об ошибке, о наличии нескольких символов с одним именем в разных модулях.

Переменная `int i` , по умолчанию имеет класс памяти auto. Для этой переменной компилятор не создаёт символ (именованный участок памяти в RAM), её значение хранится в регистре процессора. Кто интересуется, может получить ассемблерный текст, сгенерированный компилятором С:

```
arm-none-eabi-gcc -mcpu=cortex-m3 -mthumb -S ./ex4_1.c
```

Проведём ещё один эксперимент, изменим класс памяти переменных и функции на static:

```
static int src[]={1,2,3,4,5,6,7,8,9,10};
 static int dst[10];
 const int n = 10;
 static void main(void)
  { int i;
     for (i=0;i<n;i++)    {
        dst[i] = src[i]; }
      while(1); }
```

Компилируем командой:

arm-none-eabi-gcc -fno-common -mcpu=cortex-m3 -mthumb -O0 -c ./ex4_1.c -o ./ex4_1.o

Таблица символов:

```
00000000 b dst
00000000 t main
00000000 R n
00000000 d src
```

Переменные dst и src перестали быть доступными из других модулей, а функция main стала недоступной для вызова из модуля на ассемблере. Символ dst – типа b , не общедоступный (common), как в предыдущем примере. Это результат действия `-fno-common` . При попытке компоновки, будет выдано сообщение, о невозможности разрешить символ `main`. Для функции `main`, класс памяти static излишний, нужно изменить класс памяти функции main.

Реальная программа

В этом примере скомпилируем и скомпонуем один из примеров программы, поставляемый в составе пакета программ для демонстрационной платы, «STM32VLDISCOVERY firmware package». STMicroelectronics как-то не жалует GCC, например у Atmel имеется серия примеров компиляции проектов для GCC. Демонстрационные проекты Atmel могут собираться из командной строки средствами GCC, а большинство демонстрационных проектов STM рассчитаны на коммерческие среды разработки.

В составе демо пакета имеется набор файлов стандарта CMSIS и файлы библиотеки SPL (Standard peripherial library).

CMSIS – стандарт программного интерфейса микроконтроллеров с ядром Cortex. Этот стандарт разрабатывает и продвигает компания ARM, существует несколько версий и стандарт продолжает развиваться. В использованном в примере пакете, версия CMSIS 1.30.

Стартовая программа должна соответствовать некоторым требованиям, предъявляемым к ней стандартом CMSIS. Названия системных исключений должны соответствовать CMSIS. Инициализация аппаратуры микроконтроллера (в том числе настройка синхронизации), выполняется функцией SystemInit() , из состава CMSIS.

Средства CMSIS предоставляют сервисы базового уровня, эти сервисы используют средства SPL.

Стартовый код на ассемблере, из предыдущих примеров необходимо доработать. Произвольные названия векторов прерывания, нужно изменить на стандартные названия. Ниже приведён образец системного исключения по сбросу:

```
.word Reset_Handler + 1
```

Обработчик по сбросу задан в стартовом коде и ни где в программе не должен встречаться. Для обработчиков прочих системных исключений, применяется специальная техника. Кроме как в стартовом коде, обработчик может быть описан в другом модуле, с большим приоритетом. После компиляции, обработчики в стартовом коде будут игнорироваться, а реально выполняться обработчики, реализованные в другом модуле. Для этого существует ассемблерная директива `.weak`. Символ после этой директивы будет «слабым». Если в другом модуле будет объявлен символ с таким же именем, то «слабый» символ «проиграет», этот символ будет игнорироваться. Конечно, можно реализовать обработчик на ассемблере в тексте стартового модуля. В примере, обработчик NMI объявлен таким образом:

```
.word    NMI_Handler + 1
.weak    NMI_Handler
.thumb_set NMI_Handler,Default_Handler
```

Символ `NMI_Handler` , задан как «слабый». Директива `.thumb_set` специфична для ARM. Эта директива сообщает, что NMI_Handler – это адрес функции Thumb и его значение равно значению символа Default_Handler. А обработчик Default_Handler – это простейшая ловушка цикл.

В библиотеке SPL обработчики прерываний вынесены в отдельный файл stm32f10x_it.c. Если этот файл скомпилировать и скомпоновать с остальными модулями, то будут действовать обработчики из этого файла, а обработчики стартового кода будут игнорироваться.

Ещё одно изменение, это необходимость вызова `SystemInit()` . Можно вызывать эту функцию из стартового кода, а можно из функции main. Если вызывать из стартового кода, перед вызовом main, нужно вызвать `SystemInit()`.

Структура каталогов проекта:

```
├── Libraries
│   ├── CMSIS
│   │   └── CM3
│   │       ├── CoreSupport
│   │       └── DeviceSupport
│   │           └── ST
│   │               └── STM32F10x
│   └── STM32F10x_StdPeriph_Driver
│       ├── inc
│       └── src
├── Obj
│   ├── main.o
│   ├── misc.o
│   ├── stm32f10x_exti.o
│   ├── stm32f10x_flash.o
│   ├── stm32f10x_gpio.o
│   ├── stm32f10x_it.o
│   ├── stm32f10x_rcc.o
│   ├── STM32vldiscovery.o
│   ├── st.o
│   └── system_stm32f10x.o
├── Src
│   ├── main.c
│   ├── main.h
│   ├── stm32f10x_conf.h
│   ├── stm32f10x_it.c
│   ├── stm32f10x_it.h
│   └── st.s
├── Utilities
├── l.ld
├── lnk.sh
├── main.bin
├── main.elf
└── mk.sh
```

Каталоги /Libraries и /Utilities скопированы без изменений из демо пакета. Из подкаталога /Project/Examples/RCC , файлы примеров скопированы в каталог Src.

Для удобства компиляции файлов демо проекта, я написал скрипт командного интерпретатора bash. Вот его текст:

```
#!/bin/bash

INCL="-ILibraries/CMSIS/CM3/CoreSupport/ \
-ILibraries/CMSIS/CM3/DeviceSupport/ST/STM32F10x \
-ILibraries/STM32F10x_StdPeriph_Driver/inc \
-IUtilities \
-ISrc"

MACROS="-DSTM32F10X_MD_VL -DUSE_STDPERIPH_DRIVER"
OPT="-O0"
DEBUG="-g"
CFLAGS=" -mcpu=cortex-m3 -mthumb -fno-common  $OPT $DEBUG"
 echo "compiling st.s"
    arm-none-eabi-gcc $CFLAGS $INCL -c ./Src/st.s -o ./Obj/st.o
 echo "compiling system_stm32f10x.c"
  arm-none-eabi-gcc $MACROS  $CFLAGS \
  $INCL -c ./Libraries/CMSIS/CM3/DeviceSupport/ST/STM32F10x/system_stm32f10x.c \
       -o ./Obj/system_stm32f10x.o
 echo "compilimg stm32f10x_it.c"
    arm-none-eabi-gcc $MACROS $CFLAGS \
    $INCL -c ./Src/stm32f10x_it.c -o ./Obj/stm32f10x_it.o
 echo "compiling main.c"
    arm-none-eabi-gcc $MACROS $CFLAGS $INCL -c ./Src/main.c -o ./Obj/main.o
 echo "compiling STM32vldiscovery.c"
    arm-none-eabi-gcc $MACROS $CFLAGS \
    $INCL -c ./Utilities/STM32vldiscovery.c -o ./Obj/STM32vldiscovery.o
 echo "compiling stm32f10x_gpio.c"
   arm-none-eabi-gcc $MACROS $CFLAGS \
   $INCL -c ./Libraries/STM32F10x_StdPeriph_Driver/src/stm32f10x_gpio.c\
   -o ./Obj/stm32f10x_gpio.o
 echo "compiling stm32f10x_flash.c"
   arm-none-eabi-gcc $MACROS $CFLAGS \
   $INCL -c ./Libraries/STM32F10x_StdPeriph_Driver/src/stm32f10x_flash.c \
```

```
  -o ./Obj/stm32f10x_flash.o
echo "compilig stm32f10x_rcc.c"
   arm-none-eabi-gcc $MACROS $CFLAGS \
   $INCL -c ./Libraries/STM32F10x_StdPeriph_Driver/src/stm32f10x_rcc.c \
   -o ./Obj/stm32f10x_rcc.o
echo "compiling stm32f10x_exti.c"
  arm-none-eabi-gcc $MACROS $CFLAGS \
  $INCL -c ./Libraries/STM32F10x_StdPeriph_Driver/src/stm32f10x_exti.c\
  -o ./Obj/stm32f10x_exti.o
echo "compiling misc.c"
  arm-none-eabi-gcc $MACROS $CFLAGS \
  $INCL -c ./Libraries/STM32F10x_StdPeriph_Driver/src/misc.c -o ./Obj/misc.o
```

Одно замечание. Для управления процессом компиляции и сборки программ, имеется ряд специализированных утилит, самая известная из них, это make. Но в некоторых случаях, можно обходиться и скриптом командного интерпретатора. К тому же, при использовании скрипта, становится понятнее процесс сборки программ, и яснее назначение программы make.

Скрипт компилирует файлы проекта, запуская компилятор с нужными параметрами. Для удобства, параметры компиляции сохраняются в переменных, которые подставляются в команды. Опция `-I` указывает компилятору имя каталога для поиска включаемых в программу заголовочных файлов. Несколько путей поиска, хранится в переменной `INCL`. Опция `-D`, позволяет задать макрос в командной строке вызова компилятора. Точно так же, как если бы код программы содержал соответствующую директиву макроопределения. Эта опция широко применяется для управления процессом компиляции из командной строки. В примере, макрос `STM32F10X_MD_VL` указывает конкретный тип микроконтроллера, для которого будет компилироваться код. Значения подобных макросов обычно даются в описании библиотеки, либо получаются путём анализа исходного текста. Проверка значения макроса выполняется в файле stm32f10x.h из состава CMSIS. Макрос `USE_STDPERIPH_DRIVER` , указывает, что в программе будут использоваться библиотеки SPL. Значение этого макроса проверяется в том же файле stm32f10x.h. Важным является файл stm32f10x._conf.h , посредством этого файла включаются заголовочные файлы библиотек. В файле STM32vldiscovery.c , функции специфичные для демо платы.

Оптимизация кода отключена и указано, что нужно добавлять информацию для отладчика. Если не требуется отладка программы, то оптимизацию можно включить, флаг «`-O2`» или «`-Os`». А флаг «`-g`» опустить вовсе. После успешной компиляции, в каталоге Obj обнаружится набор объектных файлов.

Для компоновки и создания двоичного образа применяется отдельный командный файл :

```
#!/bin/bash
echo  "Linking"
arm-none-eabi-gcc -nostdlib -mcpu=cortex-m3 -mthumb -mlittle-endian -Tl.ld \
   -o main.elf \
   ./Obj/st.o \
   ./Obj/system_stm32f10x.o \
   ./Obj/stm32f10x_it.o \
   ./Obj/main.o \
   ./Obj/misc.o \
   ./Obj/stm32f10x_flash.o \
   ./Obj/stm32f10x_gpio.o \
   ./Obj/stm32f10x_rcc.o \
   ./Obj/stm32f10x_exti.o \
   ./Obj/STM32vldiscovery.o
echo  "make bin"
arm-none-eabi-objcopy -O binary main.elf main.bin
```

Опция `-nostdlib` запрещает подключение стандартных библиотек, из имеющихся в составе инструментария. После загрузки бинарного файла в микроконтроллер, светодиоды на плате должны мигать. Самодельный стартовый модуль оказался вполне работоспособен, так же как и скрипт компоновщика.

Библиотеки и карта компоновки

Некоторые часто используемые подпрограммы можно хранить в объектных файлах, а не компилировать каждый раз из исходных текстов. Множество объектных файлов собирается в статические библиотеки. Компоновщик работает с объектными файлами библиотеки напрямую, без необходимости извлечения отдельных объектных файлов. В статической библиотеке вместе с объектными модулями находится список, в котором содержатся имена функций и данных принадлежащих библиотеке. Посмотреть содержимое библиотеки можно утилитой arm-none-eabi-nm из инструментария. При разрешении имён, линковщик включит подходящие символы из первого подходящего объектного файла библиотеки и прекратит дальнейшее сканирование библиотеки. Если в библиотеке имеется несколько версий функции в разных объектных файлах, важен порядок расположения файлов в библиотеке.

Далее упражнение на создание и использование библиотек.

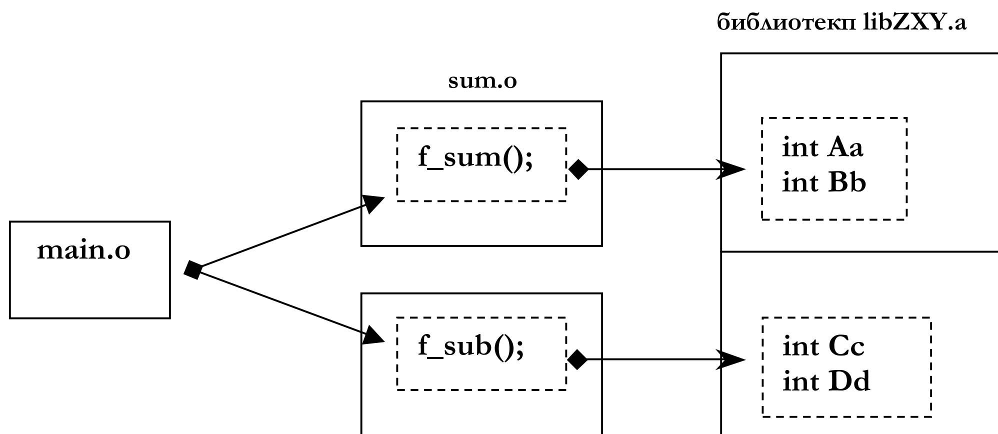

Рисунок 4

Пример программ намеренно создан простым. Библиотека `libZXY.a` содержит два модуля, `AB.o` и `CD.o` . В модулях sum.o и sub.o имеются ссыки на символы в модулях библиотеки. В свою очередь модуль `main.o` сылается на модули `sum.o` и `sub.o`. Тексты программ приведёны ниже.

Файл AB.c:
```
 int Bb = 5 ;
 int Ee = 1024 ;
 int Aa = 20 ;
```
Файл CD.c:
```
 int Cc = 20 ;
 int Dd = 5 ;
 const char Ff[]= “trash“;
```
Файл sub.c:
```
extern Cc,Dd;
 int f_sub(void)
  { return (Cc - Dd);
  }
//=================
 int ff_sub(void)
   { return (2*Cc - 3*Dd);
    }
```

Файл sum.c:

```
extern Bb,Aa;
 int f_sum(void)
  { return (Bb + Aa);
   }
```

Файл main.c:

```
 int f_sum(void);
 int f_sub(void);
 int i;
 void main(void)
 {  i = f_sum();
    i = f_sub();
    for(;;);
 }
```

Командный файл компиляции:

```
#!/bin/bash

CFLAGS=" -mcpu=cortex-m3 -mthumb -fno-common -O0 "
arm-none-eabi-gcc  $CFLAGS -c ./Src/st.s -o ./Obj/st.o
arm-none-eabi-gcc  $CFLAGS -c ./Src/AB.c -o ./Obj/AB.o
arm-none-eabi-gcc  $CFLAGS -c ./Src/CD.c -o ./Obj/CD.o
arm-none-eabi-gcc  $CFLAGS -c ./Src/main.c -o ./Obj/main.o
arm-none-eabi-gcc  $CFLAGS -c ./Src/sub.c -o ./Obj/sub.o
arm-none-eabi-gcc  $CFLAGS -c ./Src/sum.c -o ./Obj/sum.o
```

Объектные файлы `AB.o` и `CD.o` помещаем в библиотеку:

```
  arm-none-eabi-ar -r libZXY.a ./Obj/AB.o ./Obj/CD.o
```

Компонуем командой:

```
arm-none-eabi-ld -nostdlib  -Map=main.map  -Tl.ld \
 -o main.elf  ./Obj/st.o ./Obj/sub.o ./Obj/sum.o ./Obj/main.o ./libZXY.a
```

Файл библиотеки указан в конце списка объектных файлов подлежащих компоновке, и это не случайно. Если расположить, например библиотеку после файла `st.o` , то некоторые символы окажутся не разрешёнными и сборки программы не произойдёт. По умолчанию компоновщик сканирует библиотеку на предмет поиска не разрешённых ссылок, только если встретит в командной строке файл библиотеки.

В модуле `sub.o` имеются ссылки на символы в модуле библиотеки, но если библиотека указана раньше файла `sub.o` , то повторного сканирования библиотеки не произойдёт. А если библиотека указана после `sub.o` , то символы будут успешно разрешены. Когда список объектных файлов в командной строке заканчивается, то линковщиком формируется полный список не разрешённых ссылок и после сканирования библиотеки, которая находится в конце списка, они успешно разрешаются.

Но это не всё. Если библиотек несколько, то между отдельными модулями библиотек могут быть зависимости. Порядок указания библиотек тоже важен. Стандартные библиотеки компилятора, рекомендуется располагать в конце списка, в таком порядке:

```
  libc.a   libm.a   libgcc.a
```

Особый случай, когда зависимости между библиотеками циклические. Для решения подобных проблем, имеется универсальное средство. Список библиотек и объектных файлов можно задать в командной строке в виде:

```
  --start-group libK.a libL.a file.o --end-group
```

Если файлы заданы в командной строке в таком виде, то будет происходить сканирование списка библиотек при любом разрешении ссылок и до известной степени можно не заботиться о месте и порядке расположения библиотек в командной строке компоновщика. Правда это приводит к некоторому снижению скорости линковки. Можно прописать библиотеки в скрипте линковки, для этого используют команду:

```
GROUP(libK.a , libL.a , file.o).
```

Насколько мне удалось выяснить, объединять в группы, можно не только библиотеки, но и объектные файлы. Хотя в документации говорится только о библиотечных файлах. Единственный минус от группировки файлов, увеличение времени компоновки.

Благодаря флагу `-Map=file.map` , генерируется файл карты сборки проекта. Это текстовый файл с ценной информацией о компоновке программы.

Рассмотрим структуру сгенерированного файла карты сборки.

Первым идёт раздел с описанием включенных в процесс сборки библиотек. Строка из моей карты сборки:

```
libZXY.a(AB.o)            ./Obj/sum.o (Bb)
```

В строке сообщается, что в конечный образ программы включались секции из модуля AB.o входящего в библиотеку `libZXY.a` , из-за ссылок в модуле sum.o. В скобках показывается один из символов, виновников включения секций модуля AB.o – это символ Bb. Часто компилятор включает свои библиотеки по умолчанию и в этом разделе файла карты сборки, можно определить из каких библиотек, какие модули, были включены и почему.

В разделе «Memory Configuration» описана использованная конфигурация памяти. Затем описание загруженных файлов. После описания загруженных файлов, состав секций и вклад каждого объектного модуля в выходную секцию, с описанием символов видимых извне. Ниже, фрагмент файла карты:

```
.text           0x00000000       0xf4
 *(.text)
 .text           0x00000000       0x6c ./Obj/st.o
                 0x0000001c                start
```

Выходная секция `.text` , начинается с адреса 0x0 , размер 0xf4. С отступом ниже, входная секция из файла `st.o` , размером 0x6c. Ещё ниже символ `start` из этой входной секции и его значение. И так далее, все входные секции с указанием размеров и символов.

В некоторых случаях, компоновщик генерирует секции заглушки кода, на английском языке, linker stabs. Например, для некоторых архитектур ARM, для перехода из режима ARM в THUMB, необходимы такие заглушки. Иногда такие вставки требуются для обхода некоторых различий в системе команд различных реализаций ARM. Секции, отмеченные в карте сборки, как linker stabs, относятся к этой категории, в конечный образ они не входят, так как имеют нулевой размер.

При анализе файла карты сборки, можно сделать вывод, что все секции объектного файла используются как входные, даже если данные и код этих секций не используются в других объектных файлах. Секция `.rodata` модуля `CD.o` загружена, хотя ссылок на строковую константу `Ff` из других модулей нет. Легко обнаружить в двоичном образе программы строку «trash» - значение константы `Ff`. Функция `ff_sub` файла `sub.c` , не используется в других частях программы, но занесена в выходную секцию.

Флаг компоновщика `--gc-sections` , заставляет его не использовать входные секции, код и данные которых, нигде не используются в программе. Если при линковке задать этот флаг, то в конечном образе программы не будет строки «trash». А в файле карты компоновки, появится раздел «Discarded input sections». В этом разделе перечислены входные секции всех файлов использованных в процессе сборки и рядом размер отброшенной секции. Если размер секции ноль - секция не отброшена. Такое, несколько странное представление информации. Из карты сборки видно, что 8 байт входной секции .rodata , модуля CD.o не вошли в выходной файл. В то же время, если во входной секции имеются объекты, как используемые, так и не используемые в других частях программы, то в этом случае секция используется целиком, примером служит функция `ff_sub` модуля `sub.o`. Интересно, что если не задать точку входа, к примеру, в скрипте линкера, командой `ENTRY( start )`, где метка `start` , объявлена как глобальная в стартовом модуле, то линкер отбросит все секции. Конечный образ будет нулевого размера.

Для более тонкой чистки программы, на предмет не нужного кода и данных, применяют флаги компилятора, `-function-sections` и `-fdata-sections`. В этом случае компилятор помещает каждую функцию в отдельную секцию. А затем при компоновке, ненужные секции исключаются из сборки, конечно, если указан флаг линкера `-gc-sections` . Перекомпилируем программы с новыми флагами и посмотрим, какие образовались секции:

```
arm-none-eabi-size -A ./sub.o
section            size   addr
.text                 0      0
.data                 0      0
.bss                  0      0
.text.f_sub          32      0
```

```
.text.ff_sub        40      0
.comment           113      0
.ARM.attributes     51      0
Total              236
```

Компилятор поместил каждую функцию в отдельную секцию, а имя создал составное, добавив к названию секции, название функции через точку. Секции модуля с данными:

```
arm-none-eabi-size  -A  ./AB.o

section            size   addr
.text                 0      0
.data                 0      0
.bss                  0      0
.data.Bb              4      0
.data.Ee              4      0
.data.Aa              4      0
.comment            113      0
.ARM.attributes      51      0
Total               176
```

Виден результат действия флага `-fdata-sections` , каждый объект данных помещён в отдельную секцию. В документации к компилятору говорится, что эти флаги компиляции могут создать проблемы при отладке и усложняют структуру исполнимого файла, поэтому без особой нужды их применять не следует.

Для успешной сборки, необходимо доработать скрипт линкера. В разделе SECTIONS , после описания входных секций, добавить описание секций с шаблоном «*», вот так это выглядит для секции `.text`:

```
.text :
* (.text);
* (.text*);
```

Если использовать эти флаги для компиляции примеров библиотеки SPL, то результат будет просто разительный.

Программа make

Выше, мной приводился пример автоматизации сборки проекта командным файлом командного интерпретатора. Вполне работоспособная методика. Но фактическим стандартом, стало использование программы make для сборки программ. Проект, собираемый вызовом командной строки средствами make, можно легко встраивать во многие интегрированные среды разработки (IDE). Широко известный Eclipse, позволяет это делать.

Руководством к действию для make служит файл описания зависимостей модулей. По умолчанию этот файл называется Makefile или makefile. Вот его текст:

```
CC=arm-none-eabi-gcc
MACROS=-DSTM32F10X_MD_VL -DUSE_STDPERIPH_DRIVER
INCL=-ILibraries/CMSIS/CM3/CoreSupport/ \
-ILibraries/CMSIS/CM3/DeviceSupport/ST/STM32F10x \
-ILibraries/STM32F10x_StdPeriph_Driver/inc \
-IUtilities -ISrc

OPT=-Os

CFLAGS= $(INCL) $(MACROS) -Wall -mthumb -mcpu=cortex-m3 $(OPT) \
 -fno-common -ffunction-sections -fdata-sections

PDrv=./Libraries/STM32F10x_StdPeriph_Driver/src
PCMSIS=./Libraries/CMSIS/CM3/DeviceSupport/ST/STM32F10x

OBJECTS=./Obj/s.o ./Obj/system_stm32f10x.o ./Obj/stm32f10x_it.o \
  ./Obj/main.o ./Obj/misc.o ./Obj/stm32f10x_flash.o \
  ./Obj/stm32f10x_gpio.o ./Obj/stm32f10x_rcc.o \
  ./Obj/stm32f10x_exti.o ./Obj/STM32vldiscovery.o

LFLAGS= -T L.ld -Wl,--gc-sections -Wl,-Map=main.map -nostdlib

main.elf: $(OBJECTS) L_.ld
	arm-none-eabi-gcc $(LFLAGS) -o main.elf $(OBJECTS)
	arm-none-eabi-objcopy -O binary main.elf main.bin

./Obj/s.o: ./Src/s.s
	$(CC) $(CFLAGS) -c ./Src/s.s -o ./Obj/s.o

./Obj/system_stm32f10x.o: $(PCMSIS)/system_stm32f10x.c
	$(CC) $(CFLAGS) -c $(PCMSIS)/system_stm32f10x.c -o ./Obj/system_stm32f10x.o

./Obj/stm32f10x_it.o: ./Src/stm32f10x_it.c
	$(CC) $(CFLAGS) -c ./Src/stm32f10x_it.c -o ./Obj/stm32f10x_it.o

./Obj/main.o: ./Src/main.c
	$(CC) $(CFLAGS) -c ./Src/main.c -o ./Obj/main.o

./Obj/misc.o: $(PDrv)/misc.c
	$(CC) $(CFLAGS) -c $(PDrv)/misc.c -o ./Obj/misc.o

./Obj/stm32f10x_flash.o: $(PDrv)/stm32f10x_flash.c
	$(CC) $(CFLAGS) -c $(PDrv)/stm32f10x_flash.c -o ./Obj/stm32f10x_flash.o

./Obj/stm32f10x_gpio.o: $(PDrv)/stm32f10x_gpio.c
	$(CC) $(CFLAGS) -c $(PDrv)/stm32f10x_gpio.c -o ./Obj/stm32f10x_gpio.o

./Obj/stm32f10x_rcc.o: $(PDrv)/stm32f10x_rcc.c
```

```
	$(CC) $(CFLAGS) -c $(PDrv)/stm32f10x_rcc.c -o ./Obj/stm32f10x_rcc.o

./Obj/stm32f10x_exti.o: $(PDrv)/stm32f10x_exti.c
	$(CC) $(CFLAGS) -c $(PDrv)/stm32f10x_exti.c -o ./Obj/stm32f10x_exti.o

./Obj/STM32vldiscovery.o: ./Utilities/STM32vldiscovery.c
	$(CC) $(CFLAGS) -c ./Utilities/STM32vldiscovery.c -o
./Obj/STM32vldiscovery.o

.PHONY: clean

clean:
	- rm -f *.elf *.map
	- rm -f ./Obj/*.o
	- rm -f *.bin
```

Этот файл похож на командный файл сборки проекта, который приводился раньше. Команда make позволяет использовать макроопределения. Макроопределения дают возможность добиться большей наглядности и компактности. В строках с двоеточием заданы зависимости между файлами. Ниже, строка с описанием команд, выполняемых для достижения цели. Перед строкой команд обязателен символ табуляции, если этот символ отсутствует, то будет выдано сообщение об ошибке. Слева от двоеточия – это цель. Справа от двоеточия – это предусловие, файл, который должен существовать и иметь более позднее время модификации, чем файл цели. Цель достигнута, если файл цели и предусловия существуют, и между временем модификации этих файлов имеется определённое соотношение. Как это работает? При первом запуске программа make обнаружит отсутствие файла main.elf и зависимых от него объектных файлов. Для достижения цели, требуется создание объектных файлов. А каждый объектный файл получается компилированием. Утилита выполнит поиск всех зависимостей, а потом начнёт компилирование в обратном порядке. Процесс закончится компоновкой main.elf. Если будет изменён файл `STM32vldiscovery.c` , а затем выполнена команда make. То эта утилита начнёт проверку зависимостей между существующими файлами и обнаружит, что `STM32vldiscovery.o` устарел и запустит компиляцию только этого файла.

Можно создавать и фиктивные цели, этим целям не соответствуют никакие реальные файлы. Цель `clean` – из этого разряда. Её назначение, чистка проекта, удаление файлов с расширением «о» и сопутствующих файлов. Для указания цели `clean` , выполнить, `make clean` . Фиктивные цели всегда выполняются, если конечно они указаны. Специальная цель `.PHONY` используется для подчёркивания, что цель `clean` – фиктивная.

Достоинством этого makefile , является его простота. Неплохой старт для освоения make. Недостаток этого файла makefile, что не отслеживаются изменения в заголовочных файлах. Этот недостаток легко преодолеть, если перед компиляцией выполнить чистку проекта, выполнив команду `make clean` .

Дальнейшее развитие темы – это встраивание проекта make, в какую-либо интегрированную среду разработки программ. Одно из назначений IDE – это удобная навигация по исходному тексту проекта. Современные программные проекты для встраиваемых систем содержат десятки файлов и без средств навигации, полноценно работать просто невозможно. Использование IDE – это большая, отдельная тема.

Ценность GCC в том, что используемая техника программирования, стала практически стандартом, старый добротный инструмент. Освоив работу с GCC для встроенных систем, приобретается опыт применения GCC и в прочих областях. Многие программы, распространяемые по лицензии GNU, легко собираются только GCC. Каждый ли разработчик в одиночку сможет создать видеоплеер, поддерживающий все современные форматы? Но можно портировать встроенный Linux совместно с программой Mplayer, трудозатраты намного меньшие. Думаю, что весь предыдущий материал окажет большую пользу начинающим разработчикам. Кто-то захочет дополнить картину и опубликует своё видение темы.

Литература

Артур Гриффитс. GCC. Настольная книга пользователей, программистов и ситемных администраторов. – К.: ООО «ТИД ДС», 2004.-624 с.

Дозеф Ю. Ядро Cortex-M3 компании ARM. Полное руководство. – М.:Додэка-XXI, 2012.-552 с.

Солдатов В. П. Make, Build, Autotools. Управление программными проектами – М.: ООО «Бином-Пресс», 2007.-384 с.

Levine J. Linkers & Loaders. – Morgan Kaufmann Publishers, 2000.-256 с.

Lynch James. Using Open Source Tools for AT91SAM7S Cross Development. – Grand Island, 2007.-198 с.

Vijay Kumar B. Embedded Programming with the GNU Toolchain. http://www.bravegnu.org/gnu-eprog/index.html