Хранимая программа. Шаги трансляции и запуска программы

Хохлов С.А.

МГТУ им. Н.Э. Баумана

23 сентября 2025 г.

Хранимая программа

Концепция хранимой программы

[2], с.68

- Программа, написанная на машинном языке, – это последовательность 32-битовых чисел, представляющих команды.
- Эти команды (числа) можно хранить в памяти.
- Такая концепция называется хранимой программой, именно в ней заключается главная причина могущества компьютеров.
- Для запуска новой программы не нужно тратить много времени и усилий на изменение или переконфигурацию аппаратного обеспечения, необходимо лишь записать новую программу в память.

- Хранимые программы, в отличие от специализированного аппаратного обеспечения, способны выполнять вычисления общего характера.
- Таким образом, компьютер может выполнять любые приложения, начиная от простого калькулятора и заканчивая текстовыми процессорами и проигрывателями видео, – нужно лишь изменить хранимую программу.
- В хранимой программе команды считываются, или выбираются из памяти, и выполняются процессором.
- Даже большие и сложные программы представляют собой просто последовательность операций чтения из памяти и выполнения команд.

Код на языке ассемблера	Машинный код
MOV R1, #100	0xE3A01064
MOV R2, #69	0xE3A02045
CMP R1, R2	0xE1510002
STRHS R3, [R1, #0x24]	0x25813024

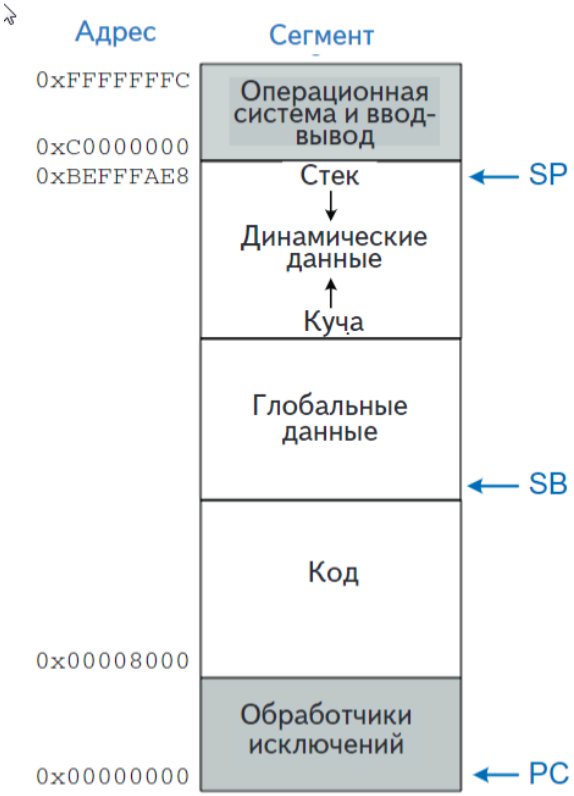
- На рисунке показано, как машинные команды хранятся в памяти.

- В программах для ARM команды обычно хранятся, начиная с младших адресов, в данном случае 0x00008000.
- Адресация памяти в архитектуре ARM побайтовая;
- 32-битовые (4-байтовые) адреса команд кратны четырем байтам.
- Чтобы запустить, или выполнить, хранимую программу, процессор последовательно выбирает ее команды из памяти.
- Далее выбранные команды декодируются и выполняются оборудованием.
- Адрес текущей команды хранится в 32-битовом регистре, называемом счетчиком команд (program counter, PC), в роли которого выступает регистр R15.

Карта памяти МК

Адресное пространство

[2], с.69-71



- На рисунке изображена карта памяти.
- Адресное пространство разделено на пять частей, или сегментов:
  - сегмент кода (.text),
  - сегмент глобальных данных (.bss для неициализированных данных, .data для инициализированных),
  - сегмент динамических данных (heap и stack)
  - сегменты для обработчиков исключений, операционной системы (ОС) и ввода-вывода.

- Так как в архитектуре ARM используются 32-битовые адреса, размер адресного пространства составляет 2^{32} байта = 4 гигабайта (ГБ).
- Адреса слов кратны 4 и располагаются в промежутке от 0 до 0xFFFFFFFC.

Сегмент кода (text segment)

- содержит машинные команды исполняемой программы.
- В ARM его называют также постоянным сегментом (read-only segment).
- Помимо кода, он может содержать литералы (константы) и данные, предназначенные только для чтения.

Сегмент глобальных данных

- содержит глобальные переменные, которые, в отличие от локальных переменных, находятся в области видимости всех функций программы.
- Глобальные переменные инициализируются при загрузке программы, но до начала ее выполнения.

Сегмент динамических данных

- содержит стек и кучу.
- В момент запуска программы этот сегмент не содержит данных – они динамически выделяются и освобождаются в процессе выполнения.

- Стек
  - В момент запуска операционная система устанавливает указатель стека (SP), так чтобы он указывал на вершину стека. Обычно стек растет вниз, как показано на рисунке выше.
  - В стеке хранятся временные данные и локальные переменные, например массивы, для которых не хватило регистров.
  - Функции также используют стек для сохранения и восстановления регистров.
  - Доступ к кадрам стека осуществляется в порядке последним пришел – первым ушел.

- Куча
  - В куче (heap) хранятся данные, динамически выделяемые программой во время работы.
  - В языке C выделение памяти осуществляется функцией malloc; в C++ и Java для этого служит оператор new.
    - По аналогии с кучей одежды, разбросанной по полу в спальне, данные, находящиеся в куче, можно использовать и отбрасывать в произвольном порядке.
    - Куча обычно растет вверх от нижней границы сегмента динамических данных.
- Если стек и куча прорастут друг в друга, данные программы могут быть повреждены. Распределитель памяти стремится избежать этой ситуации. Он возвращает ошибку нехватки памяти (out-of-memory error), если памяти недостаточно для размещения новых динамических данных.

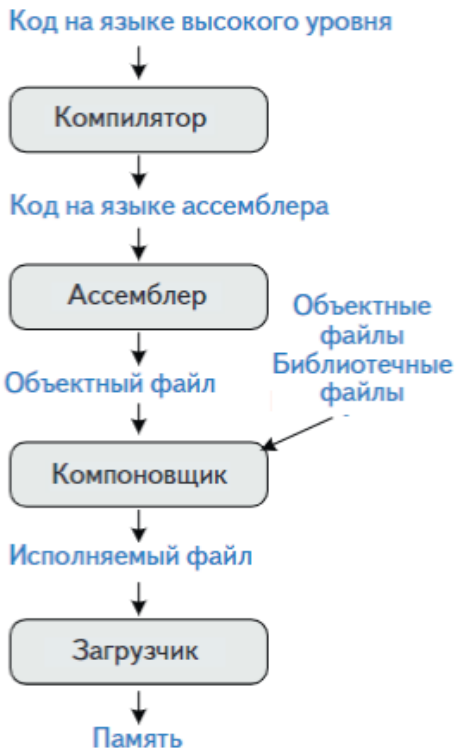
Сегменты обработчиков исключений, ОС и ввода-вывода

- Нижняя часть карты памяти в ARM, начиная с адреса 0x0, зарезервирована для таблицы векторов исключений и обработчиков исключений, а верхняя часть – для операционной системы и ввода-вывода, отображенного на память.

Трансляция и запуск

Шаги трансляции и запуска программы

[2], с.70-76



- На рисунке показаны шаги трансляции программы с языка высокого уровня на машинный язык и ее последующего запуска.

- Прежде всего компилятор транслирует программу с языка высокого уровня на язык ассемблера.
- Ассемблер транслирует ассемблерный код в машинный и записывает результат в объектный файл.
- Компоновщик объединяет машинный код с кодом из библиотек и других файлов и вычисляет адреса переходов и переменных, формируя файл исполняемой программы.
- Наконец, загрузчик загружает программу в памяти и начинает ее выполнение.

На практике большинство компиляторов выполняют все три шага: компиляцию, ассемблирование и компоновку.

Компоновка I

- Большие программы обычно содержат несколько файлов.
- Если программист изменяет только один из этих файлов, то заново компилировать и ассемблировать все остальные файлы – лишняя трата времени.
- В частности, программы нередко вызывают функции из библиотечных файлов, которые почти никогда не меняются.
- Работа компоновщика заключается в том, чтобы объединить все объектные файлы и начальный код в один исполняемый файл, содержащий машинный код.

Компоновка II

- Компоновщик формирует ELF-файл, в котором — машинный код, таблицы символов, отладочная информация.
- Дополнительно линкер раскладывает секции (.text, .data, .bss, стек, куча) по адресам памяти в соответствии с линкерным скриптом.
  - линкерный скрипт указывает, где должны лежать секции (.text, .data, .bss, стек, куча и т. д.) в адресном пространстве МК.

Программирование

- В программатор может быть загружен elf/bin/hex файл, в зависимости от того, что поддерживает программатор
  - Например, из ELF утилитой (objcopy) делают чистый бинарный файл или hex, содержащий только прошиваемый код и данные.
  - Программатор может поддерживать и сам формат ELF.
- Программатор (SWD/JTAG/bootloader) загружает бинарный код во флеш-память микроконтроллера по необходимым адресам.
- В начале флеш-памяти по адресу 0x00000000 находится векторная таблица:
  - - [0] — начальное значение MSP (Main Stack Pointer),
  - - [1] — адрес точки входа (Reset_Handler).

Аппаратный запуск (reset)

- После сброса ядро ARM Cortex-M:
  - считывает MSP из [0],
  - загружает PC из [1] и начинает выполнение с Reset_Handler.

Код запуска (startup code) I

В Reset_Handler обычно:

- инициализируется .data (копирование из флеша в RAM),
- обнуляется .bss,
- может вызываться системный инициализационный код (настройка тактирования, памяти),
- затем вызывается main().

Код запуска (startup code) II

[1] Код запуска написан на языке ассемблера для каждого целевого процессора. Можно переписать этот код на C для большей ясности.

Код запуска (startup code) выполняет следующие функции:

- Задание таблицы векторов для NVIC (__isr_vector). NVIC является контроллером прерываний. При возникновении исключения или прерывания он ищет адрес соответствующей ISR (interrupt service routine).
  - Таблица векторов для NVIC содержит значение инициализации стека, вектор сброса, все векторы исключений и внешние векторы прерываний.
  - Когда происходит сброс (reset) системы, выполнение начинается с вектора сброса.

Код запуска (startup code) III

- Инициализирует секцию памяти .bss нулями, используя символы __bss_start__ и __bss_end__ для получения диапазона адресов, который должен быть заполнен нулями. Эти символы определены в скрипте компоновщика.
- Инициализирует секцию .data, используя символы __data_start__, __data_end__ и __etext , определенные компоновщиком.
- Вызов функции инициализации платы
- Вызов функции main()

Code

```
#define DEFINE_DEFAULT_ISR(name) \
    extern "C" \
    __attribute__((interrupt)) \
    __attribute__((weak)) \
    __attribute__((noreturn)) \
    void name() { \
        while(true); \
    }

DEFINE_DEFAULT_ISR(defaultISR)
DEFINE_DEFAULT_ISR(NMI_Handler)
DEFINE_DEFAULT_ISR(HardFault_Handler)
DEFINE_DEFAULT_ISR(MemManage_Handler)
DEFINE_DEFAULT_ISR(BusFault_Handler)
DEFINE_DEFAULT_ISR(UsageFault_Handler)
DEFINE_DEFAULT_ISR(SVC_Handler)
DEFINE_DEFAULT_ISR(DebugMon_Handler)
DEFINE_DEFAULT_ISR(PendSV_Handler)
DEFINE_DEFAULT_ISR(SysTick_Handler)
DEFINE_DEFAULT_ISR(USART1_IRQHandler)

extern std::uint32_t __StackTop;
extern "C" void ResetHandler();

const volatile std::uintptr_t g_pfnVectors[]
__attribute__((section(".isr_vector"))) {
    // Stack Ptr initialization
    reinterpret_cast<std::uintptr_t>(&__StackTop),
    // Entry point
    reinterpret_cast<std::uintptr_t>(ResetHandler),
    // Exceptions
    reinterpret_cast<std::uintptr_t>(NMI_Handler),            /* NMI_Handler */
    reinterpret_cast<std::uintptr_t>(HardFault_Handler),      /* HardFault_Handler */
    reinterpret_cast<std::uintptr_t>(MemManage_Handler),      /* MemManage_Handler */
    reinterpret_cast<std::uintptr_t>(BusFault_Handler),       /* BusFault_Handler */
    reinterpret_cast<std::uintptr_t>(UsageFault_Handler),     /* UsageFault_Handler */
    reinterpret_cast<std::uintptr_t>(nullptr),                /* 0 */
    reinterpret_cast<std::uintptr_t>(nullptr),                /* 0 */
    reinterpret_cast<std::uintptr_t>(nullptr),                /* 0 */
    reinterpret_cast<std::uintptr_t>(nullptr),                /* 0 */
    reinterpret_cast<std::uintptr_t>(SVC_Handler),            /* SVC_Handler */
    reinterpret_cast<std::uintptr_t>(DebugMon_Handler),       /* DebugMon_Handler */
    reinterpret_cast<std::uintptr_t>(nullptr),                /* 0 */
    reinterpret_cast<std::uintptr_t>(PendSV_Handler),         /* PendSV_Handler */
    reinterpret_cast<std::uintptr_t>(SysTick_Handler),        /* SysTick_Handler */
    // External Interrupts
};
```

Code

```cpp
#include <algorithm>
#include <cstdint>
#include "core_cm7.h"

static void BoardInitialization() {
    SCB->CPACR |= ((3UL << 10*2)|(3UL << 11*2));  /* set CP10 and CP11 Full Access */
}

extern "C"
void ResetHandler() {
    // Initialize data section
    extern std::uint8_t __data_start__;
    extern std::uint8_t __data_end__;
    extern std::uint8_t __etext;
    std::size_t size = static_cast<size_t>(&__data_end__ - &__data_start__);
    std::copy(&__etext, &__etext + size, &__data_start__);

    // Initialize bss section
    extern std::uint8_t __bss_start__;
    extern std::uint8_t __bss_end__;
    std::fill(&__bss_start__, &__bss_end__, UINT8_C(0x00));

    // Initialize static objects by calling their constructors
    typedef void (*function_t)();
    extern function_t __init_array_start;
    extern function_t __init_array_end;
    std::for_each(&__init_array_start, &__init_array_end, [](const function_t pfn) {
        pfn();
    });

    BoardInitialization();

    // Jump to main
    asm ("bl main");
}
```

Выполнение программы

- Дальше работает пользовательский код.
- При вызове прерываний ядро берет адрес обработчика из векторной таблицы.

Линкерный скрипт

Пример кода и линкерного скрипта

```c
#include <stdint.h>

/* переменная в пользовательской секции */
__attribute__((section(".mydata")))
volatile uint32_t counter = 0x12345678;

/* функция в отдельной секции */
__attribute__((section(".myfunc")))
void blink_once(void)
{
    // здесь какая-нибудь логика
}
```

Пример кода и линкерного скрипта

```
/* области памяти */
MEMORY
{
  FLASH (rx)  : ORIGIN = 0x08000000, LENGTH = 512K
  RAM   (xrw) : ORIGIN = 0x20000000, LENGTH = 128K
  EXTRA (rw)  : ORIGIN = 0x08040000, LENGTH = 16K   /* наша область во флеше */
}
/* размещение секций */
SECTIONS
{
  .text : /* выходная секция */
  {
    /* входные секции */
    *(.isr_vector) /* `*` в начале — **маска имени объектного файла**. */
    *(.text*)  /* взять все входные секции,
          /* имена которых **начинаются** на `.text`*/
    *(.rodata*)
  } > FLASH
  /* наша секция данных во флеше, по адресу 0x08040000 */
  .mydata :
  {
    KEEP(*(.mydata))
  } > EXTRA
  /* функцию тоже можно в отдельный участок флеша */
  .myfunc :
  {
    KEEP(*(.myfunc))
  } > EXTRA
}
```

Комментарии I

Как читать (на примере выходной секции .text):

- Создать выходную секцию .text в области памяти FLASH.
- Поместить туда все входные секции из всех объектных файлов с именем .isr_vector, и все входные секции, чьё имя начинается на .text и .rodata.

Комментарии II

Локальные переменные и компоновщик

- к локальным переменным __attribute__((section())) не применяется (они в стеке).
- а к статическим локальным переменным – применять можно:

```
void f(void) {
    static int y __attribute__((section(".mystatic"))) = 42;
}
```

[1] Javier Alvarez. ARM Cortex-M Startup Code (for C and C++). 3 янв. 2019. URL: https://allthingsembedded.com/post/2019-01-03-arm-cortex-m-startup-code-for-c-and-c/ (дата обр. 17.09.2024).

[2] Дэвид М. Харрис и Сара Л. Харрис. Цифровая схемотехника и архитектура компьютера. Дополнение по архитектуре ARM / пер. с англ. Слинкин А. А. / науч. ред. Косолобов Д. А. Москва: ДМК Пресс, 2019. 356 с. ISBN: 978-5-9706-0650-6.