Глава 2

Рабочее окружение и MCPU Toolchain

В этой главе мы подготовим рабочее окружение MCPU и научимся видеть весь путь программы от исходного текста до исполнения. Установим необходимые библиотеки и инструменты, создадим личное дерево `$HOME/.mcpu`, разберём порядок поиска заголовочных файлов, библиотек и CRT, познакомимся с форматом MELF и утилитами исследования объектных файлов. В конце главы мы соберём небольшой проверочный пример и убедимся, что препроцессор, ассемблер, компоновщик и среда исполнения работают как единая система. После этого можно будет перейти к первой настоящей программе на языке ассемблера MCPU.

Andrey V. Kosteltsev · октябрь 2026

Оглавление

2 Рабочее окружение и MCPU Toolchain 70

2.1 Рабочая среда и необходимые компоненты . . . 70
2.1.1 Где взять исходные тексты . . . 70
2.1.2 Порядок установки . . . 71
2.1.3 Сборка libmpu . . . 71
2.1.4 Сборка libmpuio . . . 71
2.1.5 Сборка mcpu-cpp . . . 72
2.1.6 Сборка LibMCPU . . . 72
2.1.7 Необязательный ZUBR . . . 73
2.1.8 Что будет подготовлено дальше . . . 73
2.2 Домашнее окружение MCPU . . . 73
2.2.1 Создание домашнего дерева . . . 74
2.2.2 От runtime root к effective paths . . . 75
2.2.3 Три уровня конфигурации . . . 75
2.2.4 Конфигурация mcpu-cpp . . . 76
2.2.5 Конфигурация mcpu-ld . . . 77
2.2.6 Проверка конфигурации mcpu-cpp . . . 78
2.2.7 Проверка конфигурации mcpu-ld . . . 78
2.2.8 Что именно мы получили . . . 79
2.3 Инструменты MCPU Toolchain . . . 80
2.3.1 Препроцессор mcpu-cpp . . . 80
2.3.2 Ассемблер mcpu-as . . . 80
2.3.3 Компоновщик mcpu-ld . . . 81
2.3.4 Среда исполнения mcpu-run . . . 81
2.3.5 Бинарные утилиты и их происхождение . . . 82
2.4 Объектный формат MELF . . . 82
2.4.1 Три формы MELF . . . 83
2.4.2 Секции, символы и релокации . . . 83
2.4.3 Программные заголовки и образ процесса . . . 84
2.4.4 Низкоуровневый жизненный цикл остаётся основой . . . 84
2.4.5 Где здесь будущий язык высокого уровня . . . 85
2.5 Исследование объектных файлов MELF . . . 85
2.5.1 mcpu-readelf: структура файла . . . 85
2.5.2 mcpu-nm: имена без лишних деталей . . . 87
2.5.3 mcpu-objdump: команды и релокации вместе . . . 87
2.5.4 Окончательная компоновка и mcpu-readelf для ET_EXEC . . . 88
2.5.5 mcpu-ar: статическая библиотека как архив ET_REL . . . 89
2.5.6 mcpu-strip и mcpu-objcopy . . . 89
2.6 Как выглядит программа до первой главы программирования . . . 89
2.6.1 Простая ПРОГРАММА: _start_life . . . 90
2.6.2 ПРОГРАММНАЯ МОДЕЛЬ: _start, _life и _fini . . . 90
2.6.3 Почему мы пока не начинаем с main() . . . 91
2.7 Проверка рабочего окружения . . . 92
2.7.1 Проверка установленных программ . . . 92
2.7.2 Проверка эффективной конфигурации . . . 93

2.7.3 Проверка всей цепочки сборки 93
2.8 Литература 96

2. Рабочее окружение и MCPU Toolchain

2.1. Рабочая среда и необходимые компоненты

Прежде чем переходить к первым программам на языке ассемблера MCPU, подготовим рабочую среду, в которой читатель сможет самостоятельно выполнять все действия, встречающиеся далее в книге: запускать препроцессор, собирать исходный текст ассемблером, компоновать объектные файлы, запускать готовую программу и исследовать структуру MELF-файлов специальными утилитами.

Для основной части книги необходимы четыре компонента:

- `libmpu` — библиотека машинной арифметики, на которой основана вычислительная модель MCPU;
- `libmpuio` — библиотека ввода-вывода и преобразования данных, используемая инструментами MCPU;
- `mcpu-cpp` — препроцессор исходных текстов;
- `libmcpu` — реализация виртуального процессора MCPU и комплект ассемблера, компоновщика, среды исполнения и бинарных утилит.

В рабочем окружении, использованном при подготовке этой главы, применяются `libmpu-1.0.25`, `libmpuio-1.0.4`, `mcpu-cpp-1.0.3` и `libmcpu-0.3.34`. Номера выпусков здесь важны для воспроизводимости примеров установки, но не являются частью архитектурного контракта MCPU.

2.1.1. Где взять исходные тексты

Исходные тексты `libmpu` доступны в Git-репозитории проекта Radix и в каталоге releases:

```
https://git.radix-linux.su/libs/libmpu.git/
https://ftp.radix-linux.su/libmpu/releases/
```

Для `libmpuio` используются:

```
https://git.radix-linux.su/libs/libmpuio.git/
https://ftp.radix-linux.su/libmpu/releases/
```

Исходные тексты `mcpu-cpp` находятся здесь:

```
https://git.radix-linux.su/mcpu/mcpu-cpp.git/
https://ftp.radix-linux.su/mcpu/
```

Выпуски `libmcpu`, используемые при подготовке книги, доступны в каталоге pre-releases:

```
https://ftp.radix-linux.su/mcpu/pre-releases/
```

Для Slackware готовые сценарии SlackBuild для `libmpu` и `libmpuio` также опубликованы на `slackbuilds.org`. Далее мы, однако, покажем обычную сборку из исходных архивов: она одинаково хорошо демонстрирует реальные префиксы установки и расположение файлов, которые понадобятся для понимания инструментария MCPU.

2.1.2. Порядок установки

Компоненты удобно устанавливать в следующем порядке:

1. `libmpu`;
2. `libmpuio`;
3. `mcpu-cpp`;
4. `libmcpu`.

Такой порядок отражает зависимости: сначала устанавливаются базовые библиотеки, затем препроцессор и только после этого полный комплект инструментов MCPU.

2.1.3. Сборка libmpu

Распакуем исходный архив и будем собирать библиотеку вне дерева исходных файлов:

```
tar xJvf libmpu-1.0.25.tar.xz
mkdir build-libmpu-1.0.25
cd build-libmpu-1.0.25/
../libmpu-1.0.25/configure \
  --prefix=/usr \
  --libdir=/usr/lib64 \
  --with-real-math-f-limit=16384 \
  --enable-static=no
make -j4
sudo make install
```

Параметр `--prefix=/usr` задаёт префикс установки, а `--libdir=/usr/lib64` — каталог библиотек основной системы на 64-bit системе. Статическая библиотека основной системы в этом варианте установки отключается опцией `--enable-static=no`.

2.1.4. Сборка libmpuio

Следующим устанавливается `libmpuio`:

```
tar xJvf libmpuio-1.0.4.tar.xz
mkdir build-libmpuio-1.0.4
cd build-libmpuio-1.0.4/
```

```
../libmpuio-1.0.4/configure \
  --prefix=/usr \
  --libdir=/usr/lib64 \
  --enable-static=no
make -j4
sudo make install
```

2.1.5. Сборка mcpu-cpp

После библиотек установим препроцессор MCPU:

```
tar xJvf mcpu-cpp-1.0.3.tar.xz
mkdir build-mcpu-cpp-1.0.3
cd build-mcpu-cpp-1.0.3/
../mcpu-cpp-1.0.3/configure \
  --prefix=/usr \
  --libdir=/usr/lib64
make -j4
sudo make install
```

В следующих разделах мы отдельно разберём, откуда `mcpu-cpp` получает конфигурацию, как формируется его **effective root** (*эффективный корень*) и в каком порядке выполняется поиск заголовочных файлов.

2.1.6. Сборка LibMCPU

Последним устанавливается полный инструментарий MCPU:

```
tar xJvf libmcpu-0.3.34.tar.xz
mkdir build-libmcpu-0.3.34
cd build-libmcpu-0.3.34/
../libmcpu-0.3.34/configure \
  --prefix=/usr \
  --libdir=/usr/lib64
make -j4
sudo make install
```

После установки в системе появляются ассемблер `mcpu-as`, компоновщик `mcpu-ld`, среда исполнения `mcpu-run` и набор утилит исследования объектных файлов MELF. Одновременно устанавливается целевое окружение MCPU: системные заголовочные файлы, конфигурация компоновщика и объекты CRT. Именно это целевое окружение позднее можно скопировать в домашний каталог и сделать независимым от системной установки.

2.1.7. Необязательный ZUBR

Для изучения языка ассемблера MCPU генератор синтаксических анализаторов `zubr` не требуется. Он может быть интересен читателю, который хочет глубже познакомиться с построением языковых процессоров и посмотреть простые примеры калькуляторов из книги «Построение интерпретаторов и компиляторов. BIZON, BYACC, ZUBR» [3].

Исходные тексты и примеры ZUBR доступны здесь:

```
https://git.radix-linux.su/prog/zubr.git/
https://ftp.radix-linux.su/zubr/
```

При необходимости `zubr-4.1.0` можно собрать тем же способом:

```
tar xJvf zubr-4.1.0.tar.xz
mkdir build-zubr-4.1.0
cd build-zubr-4.1.0/
../zubr-4.1.0/configure \
  --prefix=/usr \
  --libdir=/usr/lib64
make -j4
sudo make install
```

На дальнейшую работу с примерами этой книги наличие ZUBR не влияет.

2.1.8. Что будет подготовлено дальше

Установить программы недостаточно: необходимо понимать, какие целевые файлы они используют. Поэтому далее в этой главе мы последовательно разберём:

- понятие **effective root** (*эффективный корень*) MCPU;
- системную и домашнюю конфигурацию `mcpu-cpp`, `mcpu-ld` и `mcpu-run`;
- создание домашнего корня MCPU в `$HOME/.mcpu`;
- порядок поиска заголовочных файлов, библиотек и объектов CRT;
- диагностические опции, которые печатают значения переменных конфигурации и реальные пути поиска;
- назначение `mcpu-as`, `mcpu-ld`, `mcpu-run`, `mcpu-readelf`, `mcpu-nm`, `mcpu-objdump`, `mcpu-ar`, `mcpu-strip` и `mcpu-objcopy`.

После этого рабочая среда будет полностью готова к разбору первых программ: читатель сможет не только выполнить команды из книги, но и проверить, какие именно заголовочные файлы, библиотеки и целевые объектные файлы использует инструментарий.

2.2. Домашнее окружение MCPU

После установки всех компонентов у нас уже есть рабочий комплект программ. Если использовались параметры `--prefix=/usr` и `--libdir=/usr/lib64`, то исполняемые файлы MCPU до-

ступны через обычный PATH, а само дерево MCPU располагается в /usr/lib64/mcpu. На этом этапе можно было бы сразу переходить к первым примерам. Однако для дальнейшей работы полезно сделать ещё один шаг и отделить *программы инструментария* от *целевого окружения*, с которым работает пользователь.

Такое разделение особенно удобно во время разработки. Системный администратор может обновлять установленный LibMCPU, а программист — проверять новые заголовочные файлы, объекты CRT или библиотеки в своём домашнем каталоге, не изменяя системную установку. Для этого экосистема MCPU использует домашнее дерево $HOME/.mcpu.

Важно сразу представить себе это дерево не как ещё одну копию программы mcpu-ld или mcpu-cpp, а как личное **целевое окружение** (*target environment*). Сами утилиты могут по-прежнему запускаться из /usr/bin, но часть файлов, которые они ищут во время препроцессирования и компоновки, будет браться из домашнего дерева.

2.2.1. Создание домашнего дерева

Для материала этой книги достаточно следующей структуры:

```
$HOME/.mcpu/
|-- etc/
|   |-- mcpu-cpp.conf
|   `-- mcpu-ld.conf
|-- include/
|   `-- sys/
|       `-- asm/
|           |-- dirent.h
|           |-- errno.h
|           |-- fcntl.h
|           |-- mpu-errno.h
|           |-- select.h
|           |-- stat.h
|           |-- syscall.h
|           |-- time.h
|           |-- types.h
|           `-- unistd.h
`-- lib/
    `-- mcpu/
        |-- crt0.o
        |-- crt1.o
        |-- crti.o
        `-- crtn.o
```

Создадим необходимые каталоги и скопируем в них исходное состояние установленного целевого окружения:

```
mkdir -p $HOME/.mcpu/etc/
mkdir -p $HOME/.mcpu/lib/mcpu/
```

```
cp -a /usr/lib64/mcpu/etc/mcpu-cpp.conf $HOME/.mcpu/etc/
cp -a /usr/lib64/mcpu/etc/mcpu-ld.conf  $HOME/.mcpu/etc/
cp -a /usr/lib64/mcpu/lib/mcpu/crt*.o   $HOME/.mcpu/lib/mcpu/
cp -r /usr/lib64/mcpu/include           $HOME/.mcpu/
```

После этих команд системное дерево остаётся нетронутым, а в домашнем каталоге появляется его рабочая копия. Однако *одного копирования файлов недостаточно*. На этом этапе домашнее дерево ещё нельзя считать полностью настроенным: скопированные конфигурационные файлы нужно отредактировать так, чтобы пути поиска `mcpu-cpp` и `mcpu-ld` указывали на каталоги внутри `$HOME/.mcpu`. Это мы сделаем ниже, в подразделах о конфигурации `mcpu-cpp` и `mcpu-ld`. До этого момента следует считать, что мы только подготовили файлы будущего домашнего окружения, но ещё не включили это окружение в работу инструментария.

В следующих главах мы подробно разберём назначение объектов CRT. Сейчас достаточно понимать, что это объектные файлы целевой MCPU-среды, которые компоновщик должен уметь находить при сборке программ высокого уровня. Для первых ассемблерных примеров нам прежде всего понадобятся заголовочные файлы из `$HOME/.mcpu/include`. В частности, каталог `include/sys/asm` содержит определения, непосредственно связанные с низкоуровневым ABI: номера системных вызовов, типы, значения `errno`, флаги и структуры. Уже в первых программах мы будем подключать эти заголовочные файлы вместо того, чтобы вручную повторять числовые константы системных вызовов.

2.2.2. От runtime root к effective paths

Здесь полезно различать два близких, но не одинаковых понятия.

Когда `mcpu-cpp`, `mcpu-ld` или `mcpu-run` запускаются обычным способом, программа сначала определяет расположение своего реального исполняемого файла. Если он находится в `<runtime-root>/bin`, то родительский каталог считается **runtime root** (*корнем времени выполнения*). Из него можно получить штатные пути:

```
<runtime-root>/etc
<runtime-root>/include
<runtime-root>/lib
```

При типичной установке это будет дерево `/usr/lib64/mcpu`. Но runtime root — лишь начальная точка. После этого инструмент читает конфигурационные файлы. Они могут заменить пути, полученные по умолчанию. Именно итоговые значения, которыми реально пользуется программа, будем называть **effective paths** (*эффективными путями*).

Это различие позволяет оставить исполняемые программы в системном каталоге, но направить поиск заголовочных файлов и библиотек в домашнее дерево. Поэтому переносимость окружения MCPU не требует, чтобы все её части обязательно жили в одном физическом каталоге.

2.2.3. Три уровня конфигурации

Для `mcpu-cpp` конфигурация читается в следующем порядке:

```
<runtime-root>/etc/mcpu-cpp.conf
/etc/mcpu/mcpu-cpp.conf
$HOME/.mcpu/etc/mcpu-cpp.conf
```

Для `mcpu-ld` и `mcpu-run` действует такая же последовательность:

```
<runtime-root>/etc/mcpu-ld.conf
/etc/mcpu/mcpu-ld.conf
$HOME/.mcpu/etc/mcpu-ld.conf
```

Порядок здесь имеет смысл. Первый файл поставляется вместе с конкретным деревом среды MCPU и задаёт его штатное окружение. Второй файл — необязательная конфигурация всей машины; через него администратор может изменить общую политику для пользователей. Последним читается домашний файл. Поэтому он имеет наивысший приоритет и позволяет пользователю изменить настройки, не затрагивая ни дерево установки, ни `/etc/mcpu`.

Если одна и та же переменная встретилась на нескольких уровнях, используется её последнее эффективное значение. Благодаря этому домашняя конфигурация работает как обычное переопределение, а не как отдельный режим работы инструментария.

2.2.4. Конфигурация mcpu-cpp

Откроем файл `$HOME/.mcpu/etc/mcpu-cpp.conf`. Для нашей домашней среды его основная часть выглядит так:

```
MCPU_CPP_INCLUDE_PATH = $HOME/.mcpu/include;
MCPU_CPP_DIFF_INCLUDE_PATH = $HOME/.mcpu/include/diff;
MCPU_CPP_DIFT_INCLUDE_PATH = $HOME/.mcpu/include/dift;
MCPU_CPP_ALG_INCLUDE_PATH = $HOME/.mcpu/include/alg;
MCPU_CPP_AS_INCLUDE_PATH = $HOME/.mcpu/include/as;
MCPU_CPP_AVM_INCLUDE_PATH = $HOME/.mcpu/include/avm;
MCPU_CPP_ACS_INCLUDE_PATH = $HOME/.mcpu/include/acs;

MCPU_CPP_AFTER_INCLUDE_PATH = ;
```

Первая переменная задаёт общий пользовательский каталог заголовочных файлов. Он доступен независимо от активного языка. Переменные `MCPU_CPP_<LANG>_INCLUDE_PATH` задают пользовательские каталоги конкретных языковых состояний препроцессора, выбираемых директивой `#lang`. Эти каталоги не имеют непосредственного отношения к одноимённым утилитам инструментария. В частности, `MCPU_CPP_AS_INCLUDE_PATH` не является путём поиска для `mcpu-as`: это корневой каталог заголовочных файлов языкового состояния `as` внутри `mcpu-cpp`.

Пока для препроцессора предусмотрены следующие языковые состояния:

Имя	Назначение
diff	дифференциальные уравнения
dift	разностные уравнения
alg	алгебраические уравнения
as	MCPU assembler
avm	схемы аналоговых вычислительных машин
ACS	структурные схемы систем автоматического управления

Здесь имя as пока следует понимать только как имя языкового состояния препроцессора. Окончательно ещё не решено, будет ли соответствующий язык непосредственно низкоуровневым MCPU assembler, отдельным HLL assembler или embedded assembler будущего языка высокого уровня. В частности, MCPU_CPP_AS_INCLUDE_PATH сам по себе никак не задаёт путь для программы mcpu-as.

На данном этапе важно понимать сам принцип: корневой язык высокого уровня сможет переключать языковое состояние директивой #lang, а вместе с ним — и соответствующий набор каталогов заголовочных файлов. Точная семантика этих будущих языков относится уже к следующим этапам развития MCPU и здесь нам не нужна.

Переменная MCPU_CPP_AFTER_INCLUDE_PATH задаёт резервный список путей, который рассматривается уже после обычных путей поиска. В приведённой конфигурации он пуст: дополнительный резервный путь нам пока не нужен.

Отдельное место занимает переменная MCPU_CPP_SYSTEM_INCLUDE_PATH. Если её нигде не переопределять, mcpu-cpp сам получает штатное значение <runtime-root>/include. Поэтому в установленном конфигурационном файле абсолютный путь обычно не записывается. Но более приоритетная конфигурация может полностью заменить системное дерево заголовочных файлов, например:

```
MCPU_CPP_SYSTEM_INCLUDE_PATH = $HOME/.mcpu/include;
```

В нашей конфигурации домашний $HOME/.mcpu/include уже введён как пользовательский путь поиска заголовочных файлов и поэтому участвует в поиске раньше штатного системного дерева. Это удобно для разработки: скопированный заголовочный файл можно изменить дома и получить его прежде системного оригинала. Если же требуется полностью изолировать эксперимент от установленного системного дерева заголовочных файлов, тогда имеет смысл явно заменить MCPU_CPP_SYSTEM_INCLUDE_PATH домашним каталогом.

Значения путей в конфигурационных файлах допускают ссылки $NAME и ${NAME}. Поэтому запись с $HOME не привязывает файл к конкретному имени пользователя и остаётся переносимой между домашними каталогами.

2.2.5. Конфигурация mcpu-ld

У компоновщика задача проще: ему нужен итоговый системный путь поиска библиотек. Для домашнего окружения в $HOME/.mcpu/etc/mcpu-ld.conf достаточно определить:

```
MCPU_LD_SYSTEM_LIBRARY_PATH = $HOME/.mcpu/lib;
```

Без такого переопределения `mcpu-ld` и `mcpu-run` используют `<runtime-root>/lib`. После чтения домашнего файла итоговый системный путь поиска библиотек становится `$HOME/.mcpu/lib`. Именно поэтому объекты CRT мы положили в `$HOME/.mcpu/lib/mcpu`: относительно выбранного корня библиотек их положение совпадает с положением в системном окружении MCPU.

Позднее в этом дереве появятся и библиотеки MCPU. Таким образом, один и тот же домашний корень сможет содержать заголовочные файлы, объекты CRT и целевые библиотеки, а конфигурационные файлы будут связывать это дерево с системно установленными утилитами.

2.2.6. Проверка конфигурации mcpu-cpp

Конфигурацию лучше не проверять чтением трёх файлов вручную. Для этого сам `mcpu-cpp` имеет диагностические режимы.

Команда

```
mcpu-cpp -dconfig
```

печатает итоговые переменные конфигурации уже после чтения всех уровней конфигурации и раскрытия ссылок на переменные окружения. Поэтому здесь следует увидеть пути, содержащие реальный домашний каталог, а не буквальный текст `$HOME`.

Но значения переменных ещё не показывают реальный порядок поиска. Для этого предназначена команда:

```
mcpu-cpp -dsearch-dirs
```

Она выводит итоговые каталоги заголовочных файлов в том семантическом порядке, в котором их рассматривает препроцессор. В этот результат попадают пути из командной строки, пользовательские каталоги выбранного языкового состояния, общий пользовательский путь, системное дерево заголовочных файлов и резервные пути. Такой вывод особенно полезен тогда, когда в двух разных местах имеется заголовочный файл с одинаковым именем: вместо догадок можно сразу увидеть, какой каталог будет проверен раньше.

Для нашего домашнего дерева важно убедиться как минимум в том, что в списке присутствуют `$HOME/.mcpu/include` и необходимые каталоги выбранных языковых состояний.

2.2.7. Проверка конфигурации mcpu-ld

У компоновщика используются похожие, но намеренно более явно названные опции. Команда

```
mcpu-ld --dump-config
```

показывает runtime root, состояние конфигурационных слоёв и итоговое значение `MCPU_LD_SYSTEM_LIBRARY_PATH`. После созданной нами домашней настройки итоговое значение должно указывать на `$HOME/.mcpu/lib`.

Следующая команда показывает уже не переменную, а фактический порядок поиска библиотек:

```
mcpu-ld --print-search-dirs
```

Каталоги, явно заданные опциями `-L`, стоят впереди настроенного системного пути поиска библиотек. Поэтому можно проверить не только постоянную настройку, но и конкретную командную строку, например:

```
mcpu-ld -L ./lib --print-search-dirs
```

В этом случае локальный `./lib` должен находиться перед `$HOME/.mcpu/lib`.

Обе утилиты имеют также режимы `--no-config` и `--sys-root`. Первый полезен для диагностики: конфигурационные файлы не читаются, а значения по умолчанию, полученные из корня времени исполнения остаются видимыми. Второй задаёт корень MCPU явно для конкретного запуска и тем самым позволяет проверить отдельное дерево без изменения постоянных конфигурационных файлов. Подробно эти опции понадобятся нам позже, когда появятся несколько параллельных вариантов целевого окружения.

2.2.8. Что именно мы получили

Теперь картина рабочего окружения становится законченной. Системная установка предоставляет исполняемые утилиты и исходное дерево среды исполнения. Домашний каталог `$HOME/.mcpu` хранит пользовательскую часть целевого окружения. Конфигурационные файлы связывают эти два мира, а диагностические опции позволяют увидеть результат этой связи глазами самого инструментария.

Это важный практический принцип. Если дальнейший пример не находит заголовочный файл или библиотеку, первым действием должно быть не копирование файлов в случайные каталоги, а проверка итоговой конфигурации и путей поиска. Тогда ошибка поиска превращается из скрытой особенности среды в обычную наблюдаемую часть процесса сборки.

В следующем разделе мы используем уже подготовленное окружение для знакомства с программами инструментария MCPU: ассемблером, компоновщиком, средой исполнения и утилитами исследования объектных файлов. После этого можно будет перейти к первой небольшой программе и проследить весь путь исходного текста от `.s` до выполняемого MELF-файла.

2.3. Инструменты MCPU Toolchain

В предыдущем разделе мы подготовили домашнее MCPU-окружение и научились проверять, откуда препроцессор и компоновщик берут заголовочные файлы, библиотеки и объекты CRT. Теперь можно посмотреть на сами программы инструментария как на одну систему. До сих пор каждое имя — `mcpu-cpp`, `mcpu-as`, `mcpu-ld`, `mcpu-run` — встречалось отдельно. Для дальнейшей работы важнее увидеть не набор команд, а путь, который проходит программа от исходного текста до исполнения.

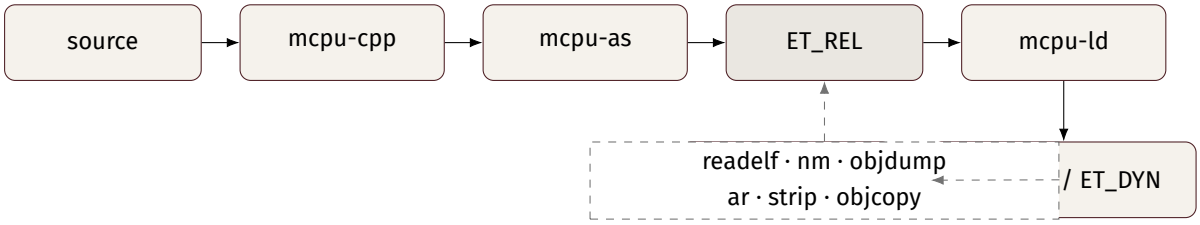


Рис. 2.1. Основной путь программы через инструментарий MCPU.

Слева на этой схеме находится текст, который пишет программист. Справа — уже исполняемый образ MCPU. Между ними расположены несколько стадий, и у каждой стадии есть свой результат, который можно сохранить, исследовать и при необходимости использовать отдельно.

2.3.1. Препроцессор mcpu-cpp

`mcpu-cpp` работает с исходным текстом до ассемблера. Именно здесь обрабатываются `#include`, определения макросов, условная компиляция, `#line` и переключение языковых состояний посредством `#lang`. Для обычного исходного текста на ассемблере удобно различать два имени файла:

```
program.s     # исходный текст, написанный программистом
program.S     # результат работы mcpu-cpp
```

Это соглашение не является требованием MELF или ассемблера, но хорошо показывает границу между двумя стадиями. В следующей главе мы будем пользоваться именно такой цепочкой, чтобы читатель мог при желании открыть `.S`-файл и увидеть, какой текст фактически получил ассемблер.

2.3.2. Ассемблер mcpu-as

Ассемблер преобразует текст программы в перемещаемый объектный файл MELF:

```
mcpu-as -o program.o program.S
```

Результат имеет тип `ET_REL`. Это ещё не исполняемая программа. У секций нет окончательных адресов в образе процесса, внешние символы могут оставаться неопределёнными, а места, которые компоновщик должен будет исправить после размещения кода и данных, описываются

записями релокаций.

Именно здесь впервые появляется принципиальная граница между *исходным представлением* и *объектным представлением*. В исходном тексте мы видим метки, директивы и мнемоники. В объектном файле остаются байты секций, таблицы символов, релокации и метаданные, необходимые следующим стадиям сборки.

2.3.3. Компоновщик mcpu-ld

`mcpu-ld` принимает один или несколько `ET_REL` и связывает их между собой. В простейшем случае он строит статический исполняемый файл:

```
mcpu-ld --stack-size=4K -o program program.o
```

Но компоновщик умеет создавать и другие формы MELF:

```
mcpu-ld -r -o combined.o a.o b.o
mcpu-ld --shared --soname=libcalc.so.1 -o libcalc.so calc.o
mcpu-ld --dynamic -o program program.o
```

Первый вариант выполняет частичную компоновку и снова создаёт `ET_REL`. Второй создаёт разделяемый объект `ET_DYN`. Третий строит динамический исполняемый файл `ET_EXEC`, способный иметь динамические зависимости. Подробности динамической компоновки относятся к более поздним главам; сейчас важно увидеть, что один компоновщик обслуживает все основные формы объектного мира MCPU.

2.3.4. Среда исполнения mcpu-run

Готовый `ET_EXEC` передаётся программе `mcpu-run`. Это одновременно загрузчик, управляющая программа MCPU и удобный испытательный стенд. `mcpu-run` проверяет метаданные MELF, создаёт память для адресных пространств, формирует область запуска, загружает PROGRAM/DATA и передаёт управление в точку входа жизненного цикла.

В отличие от обычного запуска исполняемого файла основной системы, здесь управляющая программа может непосредственно наблюдать состояние MCPU. Например:

```
mcpu-run --int-reg=r64@0x00018 -- program
mcpu-run --int-data=64@16 -- program
mcpu-run --raw-heap=64@16 -- program
```

Поэтому `mcpu-run` полезен не только как средство исполнения, но и как инструмент обучения. Когда далее мы будем обсуждать регистры, DATA, STACK, HEAP или модельные пространства, их можно будет увидеть в живой машине, а не только на рисунке.

2.3.5. Бинарные утилиты и их происхождение

Названия многих утилит MCPU намеренно следуют привычной UNIX-традиции и восходят к назначению программ GNU Binutils. Системное программирование в Linux и практическое применение классических UNIX-инструментов подробно рассматриваются, например, в книге Stewart N. Weiss [1]. Программист, знакомый с `as`, `ld`, `ar`, `nm`, `readelf`, `objdump`, `strip` и `objcopy`, сразу узнаёт общий смысл соответствующих инструментов MCPU.

Однако это сходство не следует понимать буквально. Утилиты MCPU не являются копией GNU Binutils и не являются обёрткой над обычным ELF. Они реализованы для собственной архитектуры MCPU и работают с объектным форматом MELF. Там, где у MELF нет задачи, соответствующей исторической возможности Binutils, такая возможность не переносится только ради внешнего сходства.

Например, `mcpu-objcopy` не предназначен для превращения исполняемого файла в плоский двоичный файл, Intel HEX или S-record. MELF уже является собственным контейнером объектных и исполняемых файлов MCPU и хранит именно ту структурную информацию, которая нужна компоновщику и среде исполнения.

Основные бинарные утилиты можно представить так:

Утилита	Назначение
`mcpu-readelf`	структура MELF-файла
`mcpu-nm`	таблицы символов в компактном виде
`mcpu-objdump`	секции, релокации и дизассемблирование
`mcpu-ar`	статические архивы из объектов ET_REL
`mcpu-strip`	безопасное удаление лишних символьных метаданных
`mcpu-objcopy`	безопасное редактирование метаданных MELF

Далее мы познакомимся с ними не по справке `--help`, а на одном небольшом объекте. Это важный методический приём: разные утилиты будут показывать разные стороны одного и того же файла, поэтому символы, релокации и секции не будут восприниматься как разрозненные термины.

2.4. Объектный формат MELF

Когда ассемблер завершил работу, текстовая программа перестаёт быть главным источником информации для последующих стадий сборки. Компоновщик не перечитывает мнемоники, а среда исполнения не анализирует метки исходного файла. Между ними существует самостоятельное двоичное представление — **MELF** (*MCPU Executable and Linkable Format*).

По замыслу MELF играет для MCPU ту же системную роль, которую объектный формат играет в обычном инструментарии компиляции: он связывает ассемблер, компоновщик, утилиты исследования объектных файлов и загрузчик. При этом MELF является собственным форматом MCPU и фиксирует особенности именно этой виртуальной архитектуры — метаданные целевой машины, жизненный цикл, адресные пространства MCPU, профиль STACK/HEAP и необходимые виды релокаций.

Полный нормативный контракт MELF описан в отдельном руководстве `libmcpu-melf-abi-ru.pdf`, установленном вместе с документацией LibMCPU. Здесь нам нужна не спецификация каждого поля, а рабочая картина, достаточная для чтения вывода бинарных утилит. Общие принципы работы компоновщиков и загрузчиков, независимо от конкретного объектного формата, подробно разобраны в классической книге Джона Левина [2].

2.4.1. Три формы MELF

В повседневной работе чаще всего встречаются три типа файла:

Тип	**Форма**	**Основное назначение**
`ET_REL`	перемещаемый объект	вход компоновщика; частичная компоновка; элемент архива
`ET_EXEC`	исполняемый файл	главный исполняемый образ MCPU
`ET_DYN`	разделяемый объект	модуль динамически скомпонованной программы

`ET_REL` ближе всего к тому, что программист обычно называет объектным файлом с расширением `.o`. Он содержит секции и символы, но окончательное расположение программы ещё неизвестно. Поэтому в нём допустимы неопределённые GLOBAL/WEAK-символы и сохраняются RELA-записи, которые описывают будущую работу компоновщика.

`ET_EXEC` — законченный образ главной программы. Здесь уже определены точки входа жизненного цикла, программные заголовки и профиль ресурсов исполняемого файла. STACK является частью ABI исполняемого файла; HEAP может присутствовать или быть отключён при компоновке.

`ET_DYN` — разделяемый объект. Он не является самостоятельной программой и потому не владеет жизненным циклом, STACK или HEAP. Его PROGRAM/DATA размещаются в образе процесса при начальной динамической компоновке, а необходимые runtime-relocations применяются после назначения баз модулей.

2.4.2. Секции, символы и релокации

Чтобы не утонуть в деталях формата, полезно представить `ET_REL` как набор трёх взаимосвязанных слоёв.

Первый слой — *секции*. `.text` содержит машинные команды, `.data` — инициализированные данные, `.bss` — память, не требующую хранения нулевых байтов в самом файле. Кроме них существуют служебные секции: таблицы символов, таблицы строк, таблицы релокаций и специфические для MCPU метаданные.

Второй слой — *символы*. Метка из исходного текста на ассемблере может стать записью таблицы символов. LOCAL-символ виден только внутри объектного файла; GLOBAL или WEAK может участвовать в связывании нескольких объектов. Неопределённый GLOBAL-символ означает не

ошибку ассемблера, а обещание: его определение должно появиться на одной из следующих стадий компоновки.

Третий слой — *релокации*. Если команда или объект данных содержит ссылку, окончательное значение которой ассемблер ещё не может знать, в объектном файле сохраняется запись релокации. Она отвечает на три вопроса: где находится исправляемое поле, какой символ нужен и как именно вычислить итоговое значение.

Поэтому компоновщик работает не с «текстовыми именами вообще», а с очень конкретной структурой: секция + символ + релокация.

2.4.3. Программные заголовки и образ процесса

У перемещаемого объектного файла программные заголовки обычно отсутствуют: этот файл ещё не описывает готовый образ процесса. После окончательной компоновки у `ET_EXEC` появляются описатели загружаемых сегментов и ресурсов исполнения. Например, PROGRAM и DATA получают LOAD-записи, STACK — `PT_MCPU_STACK`, HEAP — `PT_MCPU_HEAP`.

Это различие удобно запомнить как переход:

```
исходный текст
      |
      v
ET_REL: объектное представление
      |
      v
ET_EXEC / ET_DYN: результат компоновки
      |
      v
образ процесса MCPU, созданный mcpu-run
```

Исходный файл отвечает на вопрос «что написал программист». ET_REL отвечает на вопрос «что должен связать компоновщик». ET_EXEC отвечает на вопрос «что должна загрузить среда исполнения». Образ процесса — это уже состояние работающей виртуальной машины.

2.4.4. Низкоуровневый жизненный цикл остаётся основой

Контейнер MELF определяет машинные точки входа жизненного цикла `_start`, `_life` и необязательный `_fini`. Именно эти символы являются основой низкоуровневого assembler ABI.

Если отдельная фаза INIT не нужна, ассемблер предоставляет удобную запись:

```
.text
_start_life:
        ...
```

Псевдометка `_start_life:` означает совпадающие `_start` и `_life` в одной точке `.text`. Так

устроено большинство простых программ, с которых мы начнём изучение ассемблера.

Если нужна отдельная инициализация, программа получает `_start` в `.init` и `_life` в `.text`. Необязательная завершающая фаза задаётся `_fini` в `.fini`.

2.4.5. Где здесь будущий язык высокого уровня

В текущем LibMCPU уже существует слой компоновщика, рассчитанный на будущий инструментарий языка высокого уровня. Если один определяющий `ET_REL` имеет форму `main+fini`, компоновщик может классифицировать его как HLL PROGRAM; форма `init+life+fini` соответствует HLL MODEL. После этого автоматически выбираются объекты CRT, которые строят обязательный низкоуровневый жизненный цикл.

Важно не смешивать два уровня. Имена `main`, `init`, `life`, `fini` относятся к будущему контракту законченной программы языка высокого уровня. Сам формат MELF и среда исполнения по-прежнему работают через `_start`, `_life`, `_fini`.

В отдельном каталоге `crt-libc-hll` комплекта примеров уже можно увидеть ассемблерную имитацию будущего вывода компилятора и временную MCPU Libc. Этот материал предназначен для читателей, которым интересна самостоятельная разработка libc, CRT и собственного языка высокого уровня. В основных примерах ближайших глав мы сознательно остаёмся на более прямом и прозрачном низкоуровневом пути.

2.5. Исследование объектных файлов MELF

Теперь посмотрим на настоящий перемещаемый объектный файл. В сопровождающем книгу архиве примеров эти исходные тексты находятся в каталоге `environment-test`. Первый файл, `inspect-main.s`, содержит жизненный цикл и ссылки на функцию и данные, определённые во втором файле, `inspect-lib.s`. Такое разделение специально создаёт неопределённые символы и релокации, которые удобно исследовать до окончательной компоновки.

Соберём оба файла:

```
mcpu-as -o inspect-main.o inspect-main.s
mcpu-as -o inspect-lib.o  inspect-lib.s
```

2.5.1. mcpu-readelf: структура файла

Начнём с заголовка MELF:

```
mcpu-readelf -h inspect-main.o
```

Существенная часть вывода выглядит так:

```
Class:                             MELF64
```

```
Data:                              little endian
OS/ABI:                            MCPU
ABI Version:                       1
Type:                              REL (Relocatable file)
Machine:                           MCPU (0x4d43)
Entry point address:               0x0000000000000000
Number of program headers:         0
Number of section headers:         7
```

Нулевая точка входа здесь закономерна. Перед нами `ET_REL`, а не готовая программа. Компоновщик ещё не построил жизненный цикл исполняемого файла и образ процесса.

Посмотрим секции:

```
mcpu-readelf -S inspect-main.o
```

В полном выводе будут видны:

```
[ 1] .text         PROGBITS
[ 2] .rela.text    RELA
[ 3] .symtab       SYMTAB
[ 4] .strtab       STRTAB
[ 5] .mcpu.target MCPU_TARGET
[ 6] .shstrtab     STRTAB
```

Даже такой короткий список уже раскрывает устройство объектного файла. Машинные команды лежат в `.text`; ссылки, требующие работы компоновщика, — в `.rela.text`; имена и характеристики связывания символов — в `.symtab`/`.strtab`; архитектурный профиль объектного файла — в `.mcpu.target`.

Теперь посмотрим таблицу символов:

```
mcpu-readelf -s inspect-main.o
```

```
Num: Value             Type    Bind     Ndx Name
  1: 0000000000000000  FUNC    GLOBAL     1 _start
  2: 0000000000000000  FUNC    GLOBAL     1 _life
  3: 0000000000000000  NOTYPE  GLOBAL   UND add_two
  4: 0000000000000000  NOTYPE  GLOBAL   UND shared_value
```

`_start` и `_life` определены в этом объектном файле. `add_two` и `shared_value` пока имеют индекс `UND`: их определения должны быть найдены в другом входном объектном файле или библиотеке.

Наконец, релокации:

```
mcpu-readelf -r inspect-main.o
```

```
Offset             Type                Sym.Name
0000000000000019   R_MCPU_DATA64       shared_value
0000000000000029   R_MCPU_DATA64       shared_value
000000000000000f   R_MCPU_CODE64       add_two
```

Здесь особенно хорошо видно различие между ссылкой на код и ссылкой на данные. Вызов внешней функции требует `R_MCPU_CODE64`; обращения к объекту DATA — `R_MCPU_DATA64`. После окончательной компоновки эти временные поля будут заменены архитектурными адресами.

2.5.2. mcpu-nm: имена без лишних деталей

Для быстрого просмотра символов полный вывод `mcpu-readelf` обычно избыточен:

```
mcpu-nm inspect-main.o
```

```
0000000000000000 T _life
0000000000000000 T _start
                 U add_two
                 U shared_value
```

Буква `T` показывает символ, определённый в секции исполняемого кода; `U` — неопределённый символ. Такой вывод очень удобен при диагностике ошибок компоновки: за несколько строк видно, что объектный файл экспортирует и чего ему не хватает.

2.5.3. mcpu-objdump: команды и релокации вместе

`mcpu-objdump` соединяет представление объектного файла с представлением машинных команд:

```
mcpu-objdump -d -r inspect-main.o
```

Фрагмент дизассемблирования:

```
0000000000000000 <_start>:
 ...  LDI    r64@0x00000, 40
 ...  CALL   0x0000000000000000 <add_two>
      R_MCPU_CODE64  add_two
 ...  STORE  0x0000000000000000 <shared_value>, r64@0x00000
```

```
      R_MCPU_DATA64  shared_value
...   LOAD   r64@0x00018, 0x0000000000000000 <shared_value>
      R_MCPU_DATA64  shared_value
...   EXIT   r32@0x00030
```

Нулевые операнды не означают, что `CALL` действительно передаст управление по нулевому адресу. Перед нами всё ещё перемещаемое представление. Сразу под командой напечатана запись релокации, объясняющая компоновщику, чем временное значение должно стать позднее.

Так один и тот же объект можно видеть на трёх уровнях:

```
mcpu-readelf  -> структура MELF
mcpu-nm       -> символы
mcpu-objdump  -> команды + релокации
```

2.5.4. Окончательная компоновка и mcpu-readelf для ET_EXEC

Свяжем объектные файлы:

```
mcpu-ld --stack-size=512 \
        -o inspect-program inspect-main.o inspect-lib.o
```

Теперь заголовок уже сообщает:

```
Type:                              EXEC (Executable file)
Entry point address:               0x2000000000000000
LIFE entry address:                0x2000000000000000
Number of program headers:         4
```

В нашем примере отдельной `.init` нет, поэтому `_start==_life`. В программных заголовках можно увидеть два LOAD-сегмента и ресурсы исполняемого файла:

```
LOAD        ...  R E
LOAD        ...  RW
MCPU_STACK  ...  RW
MCPU_HEAP   ...  RW
```

Запустим программу:

```
mcpu-run --int-reg=r64@0x00018 -- inspect-program
```

Ожидаемый результат:

```
42
```

Таким образом мы прошли полный путь от неопределённых символов и RELA-записей в ET_REL до готового ET_EXEC, в котором адреса уже согласованы и программа выполняется.

2.5.5. mcpu-ar: статическая библиотека как архив ET_REL

Статическая библиотека MCPU не вводит нового типа MELF. Это обычный детерминированный UNIX-архив ar, элементы которого являются самостоятельными ET_REL:

```
mcpu-ar rcs libinspect.a inspect-lib.o
mcpu-ar t libinspect.a
```

Вывод:

```
inspect-lib.o
```

Такая библиотека участвует только на стадии компоновки. Компоновщик извлекает необходимые элементы архива и связывает их так же, как если бы соответствующие .o-файлы были перечислены непосредственно в командной строке.

2.5.6. mcpu-strip и mcpu-objcopy

Две оставшиеся утилиты сознательно консервативны. mcpu-strip удаляет только те обычные символьные метаданные, которые не нужны среде исполнения. Для ET_REL таблица символов необходима компоновщику, поэтому она сохраняется.

mcpu-objcopy выполняет копирование MELF в MELF и безопасное редактирование метаданных, не относящихся к ALLOC. Он умеет, например, выгружать, добавлять, обновлять или переименовывать разрешённые секции, но не позволяет разрушить размещаемый образ, динамические метаданные или релокации времени исполнения.

Для начинающего программиста главное правило простое: readelf, nm и objdump — прежде всего инструменты наблюдения; strip и objcopy уже изменяют объектный файл и потому требуют понимания того, какие части MELF относятся к контракту исполнения.

2.6. Как выглядит программа до первой главы программирования

После знакомства с инструментарием и MELF-форматом можно вернуться к исходному коду на ассемблере и увидеть в нём уже не набор директив, а будущую структуру исполняемого файла. Пока мы не будем подробно разбирать команды. Наша задача скромнее: научиться узнавать две основные формы низкоуровневой программы, чтобы в следующей главе спокойно сосредоточиться на «Здравствуй, мир!».

2.6.1. Простая ПРОГРАММА: _start_life

Самая простая форма не требует отдельной фазы INIT:

```
.text
_start_life:
        ...
        EXIT    r32@0x00030
```

Ассемблер разворачивает `_start_life:` в два нормативных GLOBAL/FUNC-символа `_start` и `_life`, находящихся по одному адресу. Поэтому в готовом исполняемом файле:

```
e_entry == e_life_entry
```

а `e_init_entry` равен нулю.

Именно эту форму мы будем считать «привычной землёй» ближайших глав. Программист видит весь жизненный цикл непосредственно и не зависит от скрытой работы CRT.

2.6.2. ПРОГРАММНАЯ МОДЕЛЬ: _start, _life и _fini

Если программе требуется отдельная параметризация или начальная инициализация, структура становится богаче:

```
.init
_start:
        ...
        END

.text
_life:
        ...
        END
        JMP     _life

.fini
_fini:
        ...
        END
```

Здесь `END` нужно понимать как границу между фазами работы, а не как безусловный «конец процесса». После границы INIT среда исполнения переходит к LIFE. Фаза LIFE может повторяться многократно. При штатном завершении среда исполнения, если присутствует `_fini`, выполняет FINI и завершает её последним `END`.

Такой жизненный цикл особенно естественен для программных моделей: INIT задаёт начальные параметры, LIFE выполняет шаг вычислительного эксперимента, FINI завершает работу и

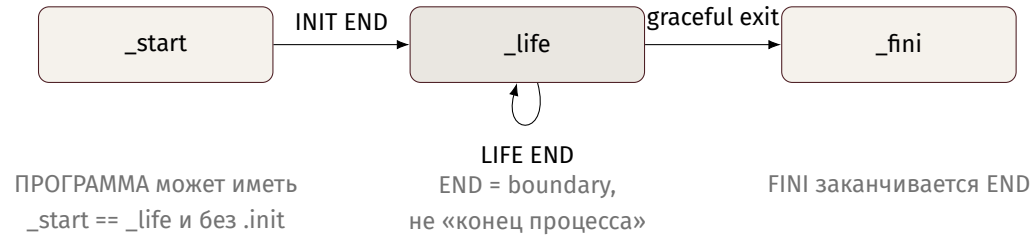


Рис. 2.2. Низкоуровневый жизненный цикл MCPU.

освобождает внешние ресурсы.

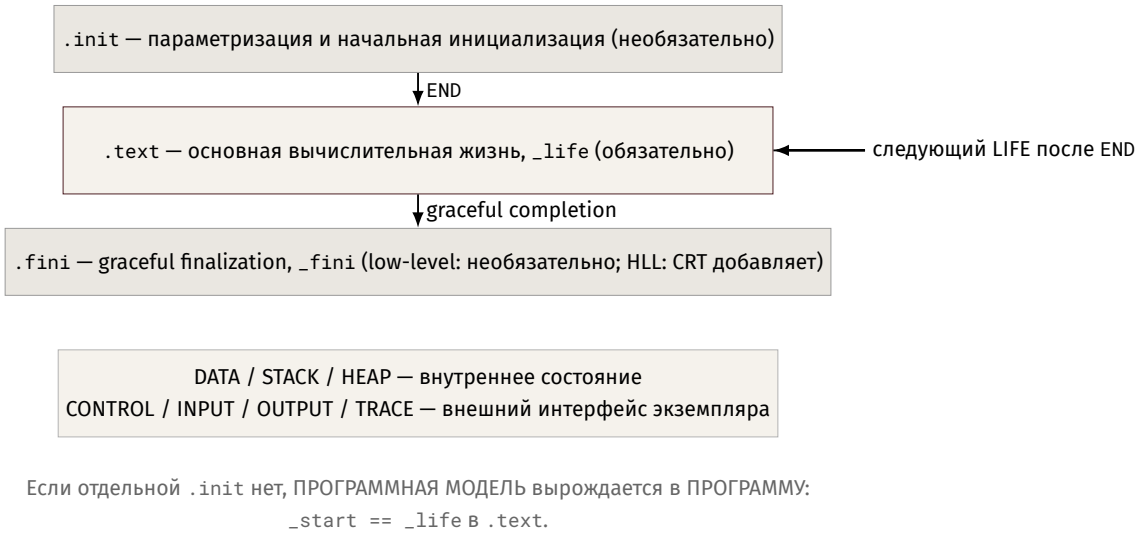


Рис. 2.3. Общая структура ПРОГРАММНОЙ МОДЕЛИ и вырожденной ПРОГРАММЫ.

Кроме PROGRAM/DATA модель может использовать CONTROL, INPUT, OUTPUT и TRACE. Эти пространства не являются обязательным украшением исполняемого файла: они появляются только тогда, когда выражают реальный внешний интерфейс модели.

2.6.3. Почему мы пока не начинаем с main()

Читатель, привыкший к C, естественно ожидает увидеть функцию `main`. В MCPU такая возможность уже предусмотрена для будущей среды языков высокого уровня, но начинать изучение низкоуровневого ассемблера с неё было бы методически неверно.

Форма вида

```
main(...)
fini(...)
```

или

```
init(...)
```

```
life(...)
fini(...)
```

предназначена для вывода будущего компилятора языка высокого уровня. Компоновщик распознаёт такую форму законченной программы и автоматически добавляет CRT, который уже создаёт обязательные `_start`, `_life` и `_fini`. То есть часть машинного жизненного цикла в этом случае принадлежит не программисту, написавшему исходный текст, а системному слою CRT и библиотеки времени исполнения.

Это очень интересная тема, и она понадобится позже, когда мы будем обсуждать самостоятельное написание MCPU Libc, CRT и собственного языка высокого уровня. В сопровождающем книгу архиве примеров каталог `crt-libc-hll` уже демонстрирует этот будущий путь на ассемблерной имитации вывода компилятора.

Но основная линия книги пока остаётся другой:

```
исходный текст программиста
          |
          v
_start / _life / _start_life / _fini
          |
          v
mcpu-as -> ET_REL -> mcpu-ld -> ET_EXEC
```

Так читатель сначала понимает саму машину и её объектный формат и жизненный цикл, а уже затем сможет оценить, какую работу за него выполняют компилятор, CRT и libc.

2.7. Проверка рабочего окружения

Перед переходом к первой настоящей программе полезно один раз проверить всю цепочку. Это не регрессионный тест LibMCPU, а пользовательская проверка: исполняемые программы находятся через `PATH`, домашняя конфигурация читается, заголовочные файлы доступны, препроцессор обрабатывает исходный текст, ассемблер создаёт объектный файл MELF, компоновщик строит исполняемый файл, а среда исполнения способна его запустить.

2.7.1. Проверка установленных программ

Начнём с версий:

```
mcpu-cpp --version
mcpu-as --version
mcpu-ld --version
mcpu-run --version
mcpu-readelf --version
mcpu-nm --version
```

```
mcpu-objdump --version
mcpu-ar --version
mcpu-strip --version
mcpu-objcopy --version
```

Конкретные номера выпусков могут меняться. Важно, чтобы команды выполнялись из того дерева установки, с которым читатель собирается работать.

2.7.2. Проверка эффективной конфигурации

Далее полезно ещё раз посмотреть итоговые пути поиска:

```
mcpu-cpp -dconfig
mcpu-cpp -dsearch-dirs
mcpu-ld --dump-config
mcpu-ld --print-search-dirs
mcpu-run --dump-config
mcpu-run --print-search-dirs
```

Если домашнее окружение настроено по предыдущему разделу, в выводе должны быть видны пути внутри `$HOME/.mcpu`. Это самый надёжный способ убедиться, что инструментарий действительно использует выбранное целевое окружение, а не только то, что нужные файлы физически скопированы в домашний каталог.

2.7.3. Проверка всей цепочки сборки

Теперь соберём маленькую программу целиком. Те же исходные файлы находятся в архиве примеров программ, сопровождающем данную книгу, поэтому их не обязательно перепечатывать вручную. Тем не менее здесь они приведены полностью: читатель может создать пустой каталог, скопировать в него три приведённых ниже файла и повторить все команды на своей машине.

Первый файл, `inspect-main.s`, содержит начало и основную фазу программы. Функция `add_two` и объект данных `shared_value` намеренно определены в другом файле, поэтому до окончательной компоновки в `inspect-main.o` останутся внешние символы и записи релокаций:

```
; Пример объектного файла для исследования в главе 2.
;
; Определения add_two и shared_value находятся в inspect-lib.s. Поэтому до
; окончательной компоновки этот объект содержит неопределённые GLOBAL-символы
; и записи RELA.

.global _life
.extern add_two
.extern shared_value

.text
_start_life:
```

```
        LDI     r64@0x00000, 40
        CALL    add_two
        STORE   shared_value, r64@0x00000
        LOAD    r64@0x00018, shared_value
        LDI     r32@0x00030, 0
        EXIT    r32@0x00030
```

Второй файл, `inspect-lib.s`, содержит недостающие определения:

```
; Определения, используемые файлом inspect-main.s.

.global add_two
.global shared_value
.global shared_ptr
.global add_two_ptr

.data
.align 32
shared_value:
        .integer 64, 0
shared_ptr:
        .ptr shared_value
add_two_ptr:
        .cptr add_two

.text
add_two:
        LDI     r64@0x00008, 2
        IADD    r64@0x00000, r64@0x00000, r64@0x00008
        RET
```

Наконец, создадим `Makefile`. Он проводит оба исходных файла через полный путь `mcpu-cpp->mcpu-as->mcpu-ld` и добавляет цели для запуска и исследования получившихся MELF-файлов:

```
.SUFFIXES:

MCPU_CPP     ?= mcpu-cpp
MCPU_AS      ?= mcpu-as
MCPU_LD      ?= mcpu-ld
MCPU_RUN     ?= mcpu-run
MCPU_READELF ?= mcpu-readelf
MCPU_NM      ?= mcpu-nm
MCPU_OBJDUMP ?= mcpu-objdump
MCPU_AR      ?= mcpu-ar

.PHONY: all run inspect archive clean

all: inspect-program
```

```
inspect-main.S: inspect-main.s
        $(MCPU_CPP) -o $@ $<

inspect-lib.S: inspect-lib.s
        $(MCPU_CPP) -o $@ $<

inspect-main.o: inspect-main.S
        $(MCPU_AS) -o $@ $<

inspect-lib.o: inspect-lib.S
        $(MCPU_AS) -o $@ $<

inspect-program: inspect-main.o inspect-lib.o
        $(MCPU_LD) --stack-size=512 -o $@ $^

run: inspect-program
        $(MCPU_RUN) --int-reg=r64@0x00018 -- ./inspect-program

inspect: inspect-main.o inspect-program
        $(MCPU_READELF) -h inspect-main.o
        $(MCPU_READELF) -S inspect-main.o
        $(MCPU_READELF) -s inspect-main.o
        $(MCPU_READELF) -r inspect-main.o
        $(MCPU_NM) inspect-main.o
        $(MCPU_OBJDUMP) -d -r inspect-main.o
        $(MCPU_READELF) -h inspect-program
        $(MCPU_READELF) -l inspect-program

archive: inspect-lib.o
        $(MCPU_AR) rcs libinspect.a inspect-lib.o
        $(MCPU_AR) t libinspect.a

clean:
        rm -f *.S *.o inspect-program libinspect.a
```

В настоящем `Makefile` строки рецептов, разумеется, начинаются символом табуляции. При копировании листинга следует сохранить эти символы; в сопровождающем книгу архиве примеров файл уже записан в правильном виде.

Команда

```
make
```

последовательно создаёт `inspect-main.S`, `inspect-lib.S`, два объектных файла `ET_REL` и окончательный `ET_EXEC`. Запуск

```
make run
```

должен вывести:

```
42
```

После этого можно выполнить:

```
make inspect
```

и сравнить между собой вывод `mcpu-readelf`, `mcpu-nm` и `mcpu-objdump` уже на своей системе. Цель

```
make archive
```

дополнительно создаёт статический архив `libinspect.a` и показывает его содержимое через `mcpu-ar`.

Если эти действия выполняются успешно, рабочее окружение готово. Мы знаем, где находятся целевые заголовочные файлы и библиотеки, понимаем разницу между исходным текстом, `ET_REL` и `ET_EXEC`, умеем посмотреть символы и релокации и представляем низкоуровневый жизненный цикл программы.

Следующая глава может начинаться с программирования без дополнительных подготовительных отступлений. Первым примером станет «Здравствуй, мир!»: мы проведём его через `mcpu-cpp`, `mcpu-as`, `mcpu-ld` и `mcpu-run`, а затем разберём каждую строку исходного текста и каждую стадию сборки.

2.8. Литература

1. Weiss S. N. *System Programming in Linux: A Hands-On Introduction*. — San Francisco: No Starch Press, 2025. — 1048 p. — ISBN 978-1-7185-0356-4. — `https://nostarch.com/system-programming-linux/`. Примеры программ к книге: `https://github.com/stewartweiss/intro-linux-sys-prog`.
2. Levine J. R. *Linkers and Loaders*. — San Francisco: Morgan Kaufmann, 2000. — 256 p. — ISBN 1-55860-496-0.
3. Костельцев А. В. *Построение интерпретаторов и компиляторов: Использование программ BIZON, BYACC, ZUBR*. — СПб.: Наука и Техника, 2001. — 218, [1] с.: ил.; 22 см. — (Конспект программиста). — ISBN 5-94387-033-4. — `https://search.rsl.ru/ru/record/01000734042`