НТЦ Метротек

«Научно-технический центр Метротек»

Лабораторная работа №1

Тема: Сбор статистики по сетевому трафику

Стажёр: Бронников Егор Игоревич

<bronnikov.40@mail.ru>
<t.me/endygamedev>

Санкт-Петербург

2022

Содержание

Постановка задачи

Цель задачи: создать набор программного обеспечения, который мог бы собирать и отображать статистику по трафику на заданном сетевом интерфейсе.

Требования:
1. ПО должно работать на ПК под управлением Debian GNU\Linux (версии 10 и новее).
2. Для реализации использовать язык программирования C.
3. Сборка должна осуществляться GNU Toolchain.
4. Дистрибуция должна осуществляться при помощи deb-пакета.
5. Сбор статистики должен вестись только по входящим UDP пакетам.
6. Должна быть реализована возможность указывать конкретные параметры учитываемых в статистике пакетов:
    1. IP-адрес источника
    2. IP-адрес назначения
    3. Порт источника
    4. Порт назначения
7. В статистике должно присутствовать количество принятых пакетов и суммарное количество байт в этих пакетах.

ПО организовать в виде двух отдельных утилит: первая читает данные с сетевого интерфейса и собирает статистику по пакетам, вторая при запуске получает собранную статистику у первой утилиты и выводит её на экран.

Утилита для сбора статистики. Нужно реализовать 2 варианта данной утилиты:
1. Два потока (pthread): первый читает пакеты при помощи Raw Socket (OSI L2) с интерфейса, проверяет параметры пакета и для

подходящих по заданным параметрам, передаёт статистику во второй поток. Второй суммирует статистику и отдаёт её по запросу извне.
2. Два потока (pthread): первый читает пакеты при помощи Raw Socket (OSI L2) с интерфейса, проверяет параметры пакета и для подходящих по заданным параметрам, суммирует статистику. Второй отдаёт её по запросу извне.

Самостоятельно провести профилирование обоих вариантов, оценить какой вариант эффективнее, с каким вариантом можно обеспечить большую пропускную способность.
В обоих вариантах: передача данных между потоками осуществляется любым способом, на усмотрение разработчика.
Утилита для вывода статистики на экран: запрашивает статистику у первой утилиты через ubus или через POSIX Message Queues. Рекомендация: попробовать реализовать оба варианта.

Программное обеспечение должно сопровождаться документацией, содержащей следующие разделы:
1. Описание - общая информация, что и как делает ПО.
2. Сборка - инструкции по сборке ПО из исходников: что установить в систему, какой командой запустить сборку, что должно получиться в итоге.
3. Запуск - как запустить ПО, как подать трафик на интерфейс, чтобы убедиться в корректности работы, что пользователь программ должен увидеть на экране.
4. Результаты профилирования двух реализованных вариантов утилиты для сбора статистики.
5. Авторство и лицензия - указать имя и электронную почту автора, указать лицензию.
Результат работы: архив git репозитория, содержащего исходники ПО и сопроводительную документацию.

Вспомогательная информация:

- The Linux Programming Interface: https://man7.org/tlpi/
- GCC: https://gcc.gnu.org
- pthread: https://man7.org/linux/man-pages/man7/pthreads.7.html
- Message Queues: https://man7.org/linux/man-pages/man7/mq_overview.7.html
- Raw Sockets: https://man7.org/linux/man-pages/man7/raw.7.html
- ubus: https://openwrt.org/docs/techref/ubus
- Сборка deb-пакета: https://www.debian.org/doc/devel-manuals#debmake-doc
- Профилирование: https://www.brendangregg.com/
- Git: https://git-scm.com/book/en/v2

1. Описание

Данная лабораторная работа разделена на две части: *ps-scanner* и *ps-stats*.

1.1 Сбор статистики: ps-scanner

Эта утилита создаёт два потока (*pthread*). Первый поток читает пакеты при помощи Raw Socket (OSI L2) с интерфейса, проверяет параметры пакета и для подходящих по заданным параметрам, суммирует статистику. Второй поток отдаёт её по запросу извне.
Передача статистики пользователю осуществляется через POSIX Message Queues.

1.2 Вывод статистики: ps-stats

В статистике отражается количество принятых пакетов и суммарное количество байт в этих пакетах. Также существует возможность фильтрации пакетов по следующим параметрам:

1. IP-адрес источника
2. IP-адрес назначения
3. Порт источника
4. Порт назначения

1.3 Структура исходников

Весь исходный код данного проекта содержится в папке `src/`:

```
src/
    basic.c
    colors.h
    main.c
    main.h
    Makefile
```

`recipient.c`

`main.c`:
Данный файл содержит системную утилиту, которая собирает информацию о пакетах и передаёт её пользователю через POSIX Message Queues.

`main.h`:
Заголовочный файл для `main.c`, который содержит прототипы функций `main.c`.

`basic.c`:
В данный файл вынесены общие функции, которые не связаны логикой с `main.c`.

`colors.h`:
Данный файл содержит константы цветов для более приятного пользовательского интерфейса.

`recipient.c`:
В данном файле содержатся функции, которые выводят статистику для пользователя. Данные приходят от `main` через POSIX Message Queues.

2. Сборка

Зависимости:

- Build Essential

  ```
  $ sudo apt-get install build-essential
  ```

- (*optional*) Linux Tools для профилирования

  ```
  $ sudo apt-get install linux-tools-common linux-tools-generic
  $ sudo apt-get install linux-tools-‘uname -r‘
  ```

2.1 Исходники

Просто для сборки проекта можно использоваться Makefile.

```
$ git clone https://github.com/endygamedev/network-traffic.git
$ cd ./network-traffic/src/
$ make
```

2.2 deb-пакет

Для сборки проекта можно воспользоваться deb-пакетом.

Установка:

1. Клонируем репозиторий с проектом:

   ```
   $ git clone https://github.com/endygamedev/network-traffic.git
   ```

2. Переходим в директорию для создания deb-пакета:

   ```
   $ cd ./network-traffic/deb/
   ```

3. Собираем запускаемые файлы:

   ```
   $ make
   ```

4. Собираем и устанавливаем deb-пакет:

   ```
   $ make install
   ```

Проверка установки пакета:

```
$ dpkg -l | grep packet-sniffer
```

Если всё установилось корректно, то должно получиться следующее сообщение:

Удаление:

```
$ make clean
```

или

```
$ sudo dpkg -r packet-sniffer
```

3. Запуск

Установленный deb-пакет содержит две утилиты:

1. `ps-scanner` – утилита для сбора статистики (системная);
2. `ps-stats` – пользовательская утилита, которая предоставляет собранную статистику.

3.1 ps-scanner

`ps-scanner` получает на вход 4 опциональных аргумента, которые помогают отфильтровать нужные пакеты:

- `-ips` или `--ip_source` – IP-адрес источника отслеживаемых пакетов;
- `-ipd` или `--ip_dest` – IP-адрес назначения отслеживаемых пакетов;
- `-ps` или `--port_source` – порт источника отслеживаемых пакетов;
- `-pd` или `--port_dest` – порт назначения отслеживаемых пакетов.

Запуск `ps-scanner`:

```
$ sudo ps-scanner -ips <IP source> -ipd <IP dest> -ps <port source>
   -pd <port dest>
```

Пример запуска `ps-scanner`:

```
$ sudo ps-scanner -ipd 192.168.1.2 -pd 9999
```

В данном примере отбираются только те пакеты, у которых IP-адрес назначения – 192.168.1.2 и порт назначения – 9999.

Также в файл `/var/log/ps-scanner.log` записываются все пакеты, которые прошли этап отбора, чтобы можно было проверить результат

работы программы.

3.2 ps-stats

Данная утилита идёт без аргументов и связывается с утилитой `ps-scanner` по названию (идентификатору) очереди (Message Queue Name).
`ps-stats` запускается после `ps-scanner`, если попытаться выводить статистику без предварительного сбора, то будет напечатано сообщение с ошибкой.

Запуск `ps-stats`:

```
$ sudo ps-stats
```

3.3 Скрипты

Для проверки работоспособности утилит можно воспользоваться файлами из папки `tests/`:

```
./tests/
    Makefile
    nc_client
    nc_server
    recipient_test
    sender_test
```

`sender_test`:

```
#!/bin/bash
sudo ps-scanner -ipd $(hostname -I | awk '{print $1}') -pd 9999
```

В данном скрипте запускается сканер, который отбирает только те пакеты, у которых IP-адрес назначения `localhost`, а порт назначения 9999.

`recipient_test`:

```
#!/bin/bash
sudo ps-stats
```

В данном скрипте запускается вывод собранной статистики `ps-scanner`.

`nc_server`:

```
#!/bin/bash
nc -u -l 9999
```

С помощью утилиты `netcat` запускается UDP-сервер на порте 9999, который ожидает сообщений от клиентов.

`nc_client`:
Данная утилита принимает 4 опциональных аргумента:

- `-c` – количество посланных пользователем сообщений;
- `-m` – текст сообщения, которое будет послано;
- `-s` – IP-адрес сервера;
- `-p` – порт сервера.

То есть после запуска данной программы на сервер `$count`-раз отправится сообщение `$message`.

3.4 Тесты

Итак, для тестирования, с помощью `netcat`, создаётся локальный UDP-сервер, на который посылаются UDP-пакеты (сообщения) клиентов.
Для начала стоит перейти в папку с тестом:

```
$ cd ./tests/
```

Затем в запускается UDP-сервер:

```
$ ./nc_server
```

После чего можно запустить `ps-scanner`:

```
$ ./sender_test
```

Далее стоит запустить `ps-stats`:

```
$ ./recipient_test
```

После чего можно отправлять различные пакеты на сервер и смотреть как отображается статистика:

```
$ ./nc_client
```

Также собранные пакеты можно посмотреть:

```
$ cat /var/log/ps-scanner.log
```

Видеопример последовательного запуска программ предстален в репозитории GitHub.

4. Профилирование

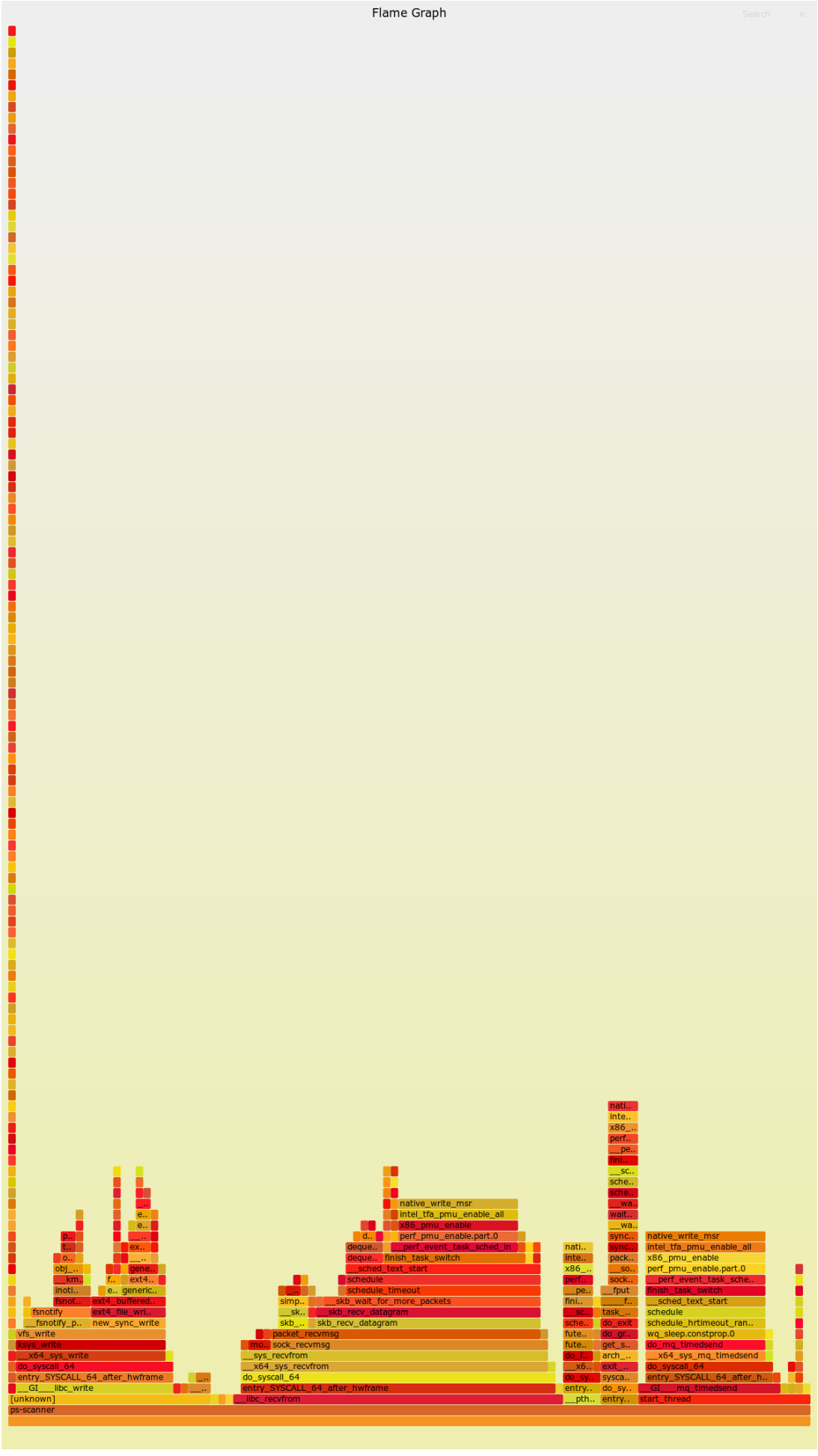
Профилирование проводилось с помощью утилиты `perf`.

Профилировалась утилита `ps-scanner`. Была собрана информация о (%) загруженности CPU, функциях программ в пользовательском пространстве и функциях ядра.
С помощью `perf record` была собрана данная статистика и сохранена в директории `./profiling/perf.data`, в данном случае профилировались все процессы (`-a`), а также собиралась трассировка стека (`-g`).

Далее с помощью `perf script` была выведена информация в виде текста, которые собрала `perf`. Полученные данные можно анализировать скриптом. Например Flamegraph-скриптом.

Flamegraph – это отличный способ визуализировать профилированные данные. На данном графе можно посмотреть сколько трассировок приходится на функции. Для более подробного просмотра можно воспользоваться браузером:

```
$ firefox ./profiling/perf.svg
```


Flame Graph
Search
ic
native_write_msr
intel_tfa_pmu_enable_all
x86_pmu_enable
perf_pmu_enable.part.0
__perf_event_task_sched_in
finish_task_switch
__sched_text_start
schedule
schedule_timeout
__skb_wait_for_more_packets
__skb_recv_datagram
skb_recv_datagram
packet_recvmsg
sock_recvmsg
__sys_recvfrom
__x64_sys_recvfrom
do_syscall_64
entry_SYSCALL_64_after_hwframe
__libc_recvfrom
vfs_write
ksys_write
__x64_sys_write
do_syscall_64
entry_SYSCALL_64_after_hwframe
__GI___libc_write
[unknown]
ps-scanner
__fput
task_...
do_exit
do_gr..
get_s..
arch_...
exit_...
sysca..
do_sy..
entry..
__GI___mq_timedsend
__x64_sys_mq_timedsend
do_mq_timedsend
wq_sleep.constprop.0
schedule_hrtimeout_ran..
schedule
__sched_text_start
finish_task_switch
__perf_event_task_sche..
perf_pmu_enable.part.0
x86_pmu_enable
native_write_msr
start_thread

5. Авторство и лицензия

Автор: Егор Бронников <bronnikov.40@mail.ru>

mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

```
network-traffic Copyright (C) 2022 Egor Bronnikov
This program comes with ABSOLUTELY NO WARRANTY; for details type
   ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, your program’s commands might be different; for a GUI interface, you would use an "about box".

You should also get your employer (if you work as a programmer) or school, if any, to sign a "copyright disclaimer" for the program, if necessary. For more information on this, and how to apply and follow the GNU GPL, see <https://www.gnu.org/licenses/>.

The GNU General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Lesser General Public License instead of this License. But first, please read <https://www.gnu.org/licenses/why-not-lgpl.html>.

Приложение А

Coding Style

C

В сторонних проектах с собственным описанным стилем оформления кода следует придерживаться правил этого стиля:

- U-Boot: https://www.denx.de/wiki/U-Boot/CodingStyle
- Linux: https://www.kernel.org/doc/html/v4.10/process/coding-style.html

В сторонних проектах без описанного стиля следует оформлять наш код по аналогии с остальными исходниками проекта.

В наших собственных проектах за основу взят стиль, принятый в linux с некоторыми изменениями.

Отступы

В качестве отступа используется один таб шириной в четыре пробела.

```
int main(void)
{
    int a = 0;
    return a;
}
```

Лишние табы и пробелы в конце строк следует удалять.

Длина строк

Следует избегать превышения ограничения в 80 символов на строку. Если выражение не помещается в 80 символов, его следует разделить

на части. При этом желательно, чтобы каждый аргумент функции находился в отдельной строке. Пример:

```
return_code = my_structure->callback_with_arguments(argument_number_1,
                                                    argument_number_2);
```

Строки, выводимые программой на экран, не следует сокращать при превышении ими ограничения.

Скобки

В условиях, циклах и объявлениях структур открывающая фигурная скобка не переносится на следующую строку. Закрывающая переносится:

```
while (a > b) {
    a--;
    b++;
    if (a == b) {
        do_something();
    }
}
```

Закрывающая фигурная скобка располагается на том же уровне отступов, что и начало всей конструкции.

В объявлениях функций открывающая фигурная скобка переносится на следующую строку:

```
int main(int argc, char **argv)
{
    return 0;
}
```

В условиях и циклах перед открывающей круглой скобкой ставится пробел. После закрывающей скобки ставится пробел:

```
if (a == 0) {
    return a;
}
```

Фигурные скобки должны присутствовать даже если в блоке всего одно выражение.

Пробелы

Бинарные и тернарные операторы окружаются пробелами:

```
= + - < > * / % | & ^ <= >= == != ? :
```

После унарных операторов не ставится пробел:

```
& * + - ~ ! sizeof typeof alignof __attribute__ defined
```

Пробел не ставится после префиксных инкремента и декремента:

```
++i
--i
```

и перед постфиксными:

```
i++
i--
```

Пробелы не ставятся вокруг операторов структур:

```
my_struct.element
my_struct->element
```

При объявлении указателя оператор * идет перед именем указателя, а не после его типа:

```
void *buffer
```

Имена и объявления

В именовании функций, переменных и типов используется *snake_ case*. Примеры:

```
void my_function(int my_argument);
struct my_struct *s;
```

В именах желательно стараться избегать сокращений, из-за которых становится не ясно назначение сущности. Например, вместо **dctl()** использовать более полное имя **device_control()**.

Объявлять переменные в функции желательно рядом с выражениями, их использующими, если функция достаточно длинная (от 10 строк), либо в начале функции, если короткая (до 10 строк). Например, переменную-счётчик для цикла for следует объявлять так:

```
for (int i = 0; i < 3; ++i) {
    actions();
}
```

Также нужно следить за количеством переменных в одной функции. Если их число превышает пять, то следует задуматься о декомпозиции функции.

Функции

Функции должны быть как можно короче и выполнять как можно меньше действий. Длинные функции следует разбивать на подфункции, действия внутри которых связаны по смыслу. Пример:

```
int collect_data_and_calculate_result(struct program_context *c)
{
    void *data = collect_data(c);

    return calculate_result(data);
}

int init_and_run_program(struct arguments *a){
    struct program_context *c = init_context(a);

    return collect_data_and_calculate_result(c);
```

```
}
```

Структуры перечисления

Их объявления должны выглядеть следующим образом.
Структуры только с именем:

```
struct my_struct_name {
    int i;
    int j;
    void *buffer;
};
```

Структуры с typedef должны иметь суффикс _t в имени типа:

```
typedef struct my_struct {
    int i;
    int j;
    void *buffer;
} my_struct_t;
```

Перечисление объявляются аналогичным образом. Все варианты перечисления должны именоваться в верхнем регистре:

```
typedef enum {
    ADDRESS_7BIT,
    ADDRESS_10BIT,
} addressing_mode_t;
```

Макросы

Желательно избегать написания макросов. Особенно вложенных, так как это приводит к сложностям в отладке.

Макросы именуются с использованием только верхнего регистра. Макрос и отдельно его входные параметры должны быть окружены круглыми скобками:

```
#define MIN(x, y)     ((x) < (y) ? (x) : (y))
```

Макросы с несколькими выражениями должны быть заключены в блок `do {} while(0)`:

```
#define DO_ACTION(a, b)                         \
                do {                            \
                    if (do_first(a) >= 0) {     \
                    do_next(b);                 \
                    }                           \
                } while(0)                      \
```

Использование goto

`goto` можно использовать для обработки ошибок. Переходя должны осуществляться только в пределах одной функции. Пример:

```
int configure_device(void)
{
    int ret = 0;
    struct my_device *d = malloc(sizeof(my_device));
    if (!d) {
        return -ENOMEM;
    }

    ret = init_stage_first(d);
    if (ret != 0) {
        goto err_free;
    }

    ret = init_stage_last(d);
    if (ret != 0) {
        goto err_free;
    }

    return 0;

err_deinit:
```

```
    deinit_stage_first(d);
err_free:
    free(d);
    return ret;
}
```

Комментарии

Комментарии следует использовать только в качестве документации и для пояснения каких-то не очевидных специфичных случаев.

Функции

Документировать следует функции, поведение, входные и выходные параметры которые не очевидны. Пример:

```
/*
 * Configures an I2C bus.
 *
 * addr_lenght should be ADDRESS_7BIT or ADDRESS_10BIT.
 * speed should be SPEED_100MHZ or SPEED_400MHZ.*
 * Returns 0 on success, -1 otherwise.
 */
int configure_i2c_bus(int bus_number, int addr_length, int speed)
```

Не следует документировать очевидные функции. Например, в достаточно очевидны назначение и принцип работы функции. Пример:

```
int sum_int(int a, int b)
{
    return a + b;
}
```

Магические числа

Если назначение числового значения не очевидно из имени переменной или макроса, которым оно присваивается, то это значение следу-

ет снабдить комментарием. Пример:

```
#define DEVICE_CONTROL_REGISTER_ADDR 0x4
/* Set required device mode on startup. See device datasheet page
   100, table 20. */
#define DONTROL_REGISTER_DATA 0x4f7a
```

Заголовочные файлы

Должны обязательно содержать защиту от повторного включения:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H
All the contents of the header file here
#endif
```

либо

```
#pragma once
Content
```

Содержимое может включать:

- дополнительные `include`
- объявление макросов
- объявление констант
- прототипы функций

но реализации функций должны содержаться только в файлах `.c`

Python

См. https://peps.python.org/pep-0008/