

# **ELCOREAPI DOCUMENTATION**

**Версия 5.3.7**

**04.02.2020**

## ОГЛАВЛЕНИЕ

<b>1</b>	<b>Введение</b>	<b>1</b>
<b>2</b>	<b>Требования</b>	<b>2</b>
<b>3</b>	<b>Установка</b>	<b>3</b>
<b>4</b>	<b>Использование</b>	<b>4</b>
4.1	Системные вызовы . . . . .	5
4.2	Параметры main (argc/argv) . . . . .	5
4.3	Переменные окружения (environ) . . . . .	6
4.4	Доступ к регистрам . . . . .	6
4.5	Доступ к памяти . . . . .	6
4.6	Примеры: GDB скрипты . . . . .	7
<b>5</b>	<b>Структура библиотеки</b>	<b>9</b>
5.1	core — Базовые классы и функции API . . . . .	9
5.2	platforms — Поддерживаемые платформы . . . . .	26
5.3	gdb — GDB расширения ElcoreAPI . . . . .	28
<b>6</b>	<b>Сообщения об ошибках</b>	<b>34</b>
	<b>Содержание модулей Python</b>	<b>45</b>

## 1. ВВЕДЕНИЕ

**ElcoreAPI** - библиотека для написания тестовых сценариев для различных устройств (симуляторы, отладочные платы, RTL) на языке Python 2.7. Управление отладкой осуществляется через классы устройств, базирующиеся на *BaseDevice*. Также библиотека предоставляет [semihosting](https://developer.arm.com/docs/dui0471/k/what-is-semihosting/what-is-semihosting)<sup>1</sup> для программ, запускаемых на устройствах.

---

<sup>1</sup> <https://developer.arm.com/docs/dui0471/k/what-is-semihosting/what-is-semihosting>

## 2. ТРЕБОВАНИЯ

Требуемые сторонние библиотеки:

- `pyelftools`<sup>2</sup> - библиотека для анализа ELF файлов

**ВАЖНО:** разрядность библиотеки отладчика/симулятора ОБЯЗАТЕЛЬНО должна совпадать с разрядностью интерпретатора Python. Что бы узнать разрядность интерпретатора, достаточно вызвать его в консоли. Информация о разрядности будет в первой строке вывода интерпретатора. В ОС Linux сведения о разрядности библиотек можно получить с помощью утилиты `objdump` (`objdump -a libname.so`). В ОС Windows для этого можно использовать утилиту `link.exe`, входящую в состав Microsoft Visual Studio (`link.exe /dump /headers libname.dll`).

---

<sup>2</sup> <https://pypi.python.org/pypi/pyelftools>

### 3. УСТАНОВКА

Для работы с библиотекой рекомендуется использовать `virtualenv`<sup>3</sup>. Это позволит устанавливать любые пакеты без root прав с пользовательскими настройками проху. Использование `virtualenv` позволяет производить любые модификации рабочего окружения python без влияния на работу других проектов.

Установка и создание `virtualenv`:

```
sudo pip install virtualenv # --проху проху.company.com:8888
mkdir ~/venv && cd ~/venv
virtualenv <venv_name>
```

Активация `virtualenv`:

```
source ~/venv/<venv_name>/bin/activate
```

Для установки ElcoreAPI в штатном режиме требуется доступ в Интернет.

Установка через `pip`:

- из архива:

```
pip install -U ElcoreAPI.zip
```

Установка из исходников:

- в `site-packages`:

```
python setup.py install
```

- без копирования (с указателем в `site-packages`):

```
python setup.py develop
```

Установка из пакета конечной поставки (не требуется Интернет соединение):

- пока не создан

---

<sup>3</sup> <https://adw0rd.com/2012/6/19/python-virtualenv/>

## 4. ИСПОЛЬЗОВАНИЕ

ElcoreAPI может быть использована при написании python скриптов для автоматизации исполнения и отладки тестов, а также для обработки системных вызовов в программах, исполняемых под gdb. Вместе с документацией предоставляются примеры python скриптов (`samples/python`) и gdbinit скриптов (`samples/gdb`) под все поддерживаемые платформы. Программы, запускаемые этими скриптами, должны быть собраны с библиотекой `libruar1.a`. Если программа собрана некорректно, будет выведено соответствующее сообщение об ошибке.

Примеры python скриптов имеют интерфейс runner'ов, т.е. принимают первым аргументом elf-файл, остальные аргументы передаются запускаемой программе (в виде `argv`):

```
python chip-platform.py program.elf bar 3
```

Примеры gdbinit скриптов содержат минимально необходимые настройки платформы, подключение обработчиков системных вызовов и инициализацию `argv`. Примеры не содержат директивы `file` и могут быть использованы в Eclipse/NetBeans IDE. Запуск из командной строки:

```
gdb -x chip-platform.gdbinit -ex "file program.elf" -ex "run bar 3"
```

**ВАЖНО:** для работы с python расширениями gdb должен быть собран с поддержкой python. Для проверки конфигурации нужно выполнить в командной строке отладчика команду `py import gdb`. Если отладчик сконфигурирован правильно, команда отработает без вывода. Если ответные сообщения содержат информацию вида `ImportError: No module named gdb` или `Python scripting is not supported in this copy of GDB`, воспользуйтесь другим релизом отладчика.

В python скриптах управление отладкой осуществляется через объект класса устройства (`device`), который наследует `BaseDevice`. В gdbinit скриптах ElcoreAPI выполняет роль коллекции подключаемых расширений, которые используются для обработки различных событий. Импорт всех доступных расширений:

```
python
from ElcoreAPI.gdb import *
end
```

Тем не менее доступ к объекту устройства можно получить через интерфейс `core`.

Использование большинства расширений ElcoreAPI в gdbinit скриптах требует работы с символьной информацией, а значит исполняемая программа должна быть указана до `python ... end`` вставки. Это можно сделать с помощью директивы `file` в самом скрипте или через аргументы командной строки.

## 4.1 Системные вызовы

ElcoreAPI позволяет эмулировать системные вызовы для программы, исполняемой на устройстве. Это даёт возможность создавать файловые потоки (`open/close`), использовать функции файлового ввода/вывода (`read/write`), получать информацию о файлах (`stat`), управлять рабочей директорией (`chdir`) и использовать некоторые другие функции операционной системы (`isatty`, `gettimeofday`, `times` и т.д.). Системные вызовы будут транслироваться в аналогичные вызовы операционной системы машины, на которой запущен отладчик. Файловые операции по относительным путям будут работать относительно рабочей директории устройства (`cwd`). Стандартные потоки ввода/вывода могут быть переопределены с помощью аргументов метода `enable_syscalls()`.

В python скриптах включение системных вызовов является элементом конфигурации устройства и не требует повторной инициализации при загрузке новой программы:

```
device.enable_syscalls()
```

В gdbinit скриптах системные вызовы включаются аналогично:

```
python
enable_syscalls()
end
```

## 4.2 Параметры main (argc/argv)

ElcoreAPI обеспечивает передачу параметров в функцию `main (argc/argv)`. В `argv` может быть передан любой объект, который может быть приведен к строке. Путь до исполняемой программы будет добавлен в `argv[0]` автоматически.

В python скриптах инициализация `argv` может быть произведена только после загрузки исполняемой программы:

```
device.load_elf('some_prog.elf')
device.set_argv('foo', 1, 3.0)
```

В gdbinit скриптах использование `set_argv()` без параметров приводит к сквозной передаче аргументов, которые были получены через интерфейсы gdb (параметры директивы `run` или аргументы опции `--args`). Тем не менее параметры могут быть указаны явно. В этом случае аргументы, переданные через интерфейсы gdb, будут проигнорированы:

```
python
set_argv()
end
run foo 1 3.0
```

## 4.3 Переменные окружения (environ)

ElcoreAPI предоставляет возможность инициализации массива `environ` в исполняемой программе, тем самым предоставляя доступ к переменным окружения хоста.

В python скриптах включения механизма загрузки переменных окружения является элементом конфигурации устройства и не требует повторной инициализации при загрузке новой программы:

```
device.enable_environ()
```

В gdbinit скриптах эта настройка выполняется аналогично:

```
python
enable_environ()
end
```

## 4.4 Доступ к регистрам

В gdbinit скриптах доступ к регистрам осуществляется через интерфейсы gdb. В python скриптах есть три способа получить доступ к значениям регистров или установить их значения:

- по id, который известен отладчику:

```
device.get_register('dsp0.r10:0')
device.set_register('dsp1.dcsr', 0x4000)
```

- через память, если регистры отображены на неё:

```
device.read_word(0x80001000)
device.write_word(0x80001100, 0x4000)
```

- через регистровые атрибуты устройства (`dsp/risc`) и классы с интерфейсом *Register*:

```
device.dsp[0].r[10].l.get()
device.dsp[1].pcu.dcsr.set(0x4000)
```

## 4.5 Доступ к памяти

В gdbinit скриптах доступ к памяти осуществляется через интерфейсы gdb. В python скриптах есть два способа работать с памятью:

- операции с 32-битными словами с указанием адреса:



```
device.read_word(0xe0001000)
device.write_word(0xe0001100, 0x4000)
```

- операции с массивами памяти с указанием адреса и размера в байтах или записываемого значения:

```
device.read_memory(0xe0001000, 64)
device.write_memory(0xe0001100, bytearray('some_data'))
```

## 4.6 Примеры: GDB скрипты

Создадим простой gdbinit скрипт для работы с симулятором процессора NVCom02T. Конфигурация устройства и подключение к нему происходит в блоке python кода (python/end). Для начала нужно импортировать методы для работы с устройством:

```
from ElcoreAPI.gdb import *
```

Далее необходимо создать экземпляр симулятора:

```
create_sim3x_mgdb_target('@NVCom-02T')
```

Теперь можно включить системные вызовы:

```
enable_syscalls()
```

Если используется нестандартный конфигурационный файл симулятора, gdb выведет сообщение unknown chip (chip-platform), ElcoreAPI is not supported. В этом случае класс устройства можно указать вручную:

```
from ElcoreAPI.gdb.targets import MIPS32
set_device(MIPS32)
```

Также необходимо добавить hook-run для загрузки программы:

```
define hook-run
  py gdb.execute('monitor loadelf {}'.format(gdb.objfiles()[0].filename))
end
```

Полный текст скрипта:

```
python
from ElcoreAPI.gdb import *
from ElcoreAPI.gdb.targets import MIPS32
create_sim3x_mgdb_target('@NVCom-02T')
set_device(MIPS32)
enable_syscalls()
end
```

```
define hook-run
  py gdb.execute('monitor loadelf {}'.format(gdb.objfiles()[0].filename))
end
```

Вывод скрипта при запуске стандартного “Hello, World!”:

```
(gdb) r
Starting program: nvcom02t-test.elf
Hello, World!
Program received signal SIGTRAP, Trace/breakpoint trap.
0x80000484 in _exit ()
```

## 5. СТРУКТУРА БИБЛИОТЕКИ

### 5.1 core — Базовые классы и функции API

---

Данный модуль предоставляет базовые абстрактные классы устройства и отладчика.

#### **class core.BaseDevice**

Базовый абстрактный класс для всех классов устройств (*GDBDevice*, *MDBDevice*, *Sim3xDevice*). *BaseDevice* предоставляет высокоуровневые отладочные методы, реализованные через интерфейс отладчика. Специфичные для архитектуры методы должны быть реализованы в классе-наследнике.

**\_\_init\_\_(self, \*args, \*\*kwargs)**

Конструктор. Инициализирует атрибуты *debugger* (через метод *allocate\_debugger()*, *args* и *kwargs* передаются методу без изменений), *description*, *active\_core*, *elf*, *files*, *symbols*, *callbacks*, *call\_descriptors*, *breakpoints*.

#### **debugger**

Атрибут объекта, содержит экземпляр низкоуровневого отладчика, реализующего *DebuggerInterface*.

#### **description**

Атрибут объекта, содержит регистровую карту устройства в виде вложенных словарей.

#### **active\_core**

Атрибут объекта, содержит id текущего активного ядра.

#### **elf**

Атрибут объекта, содержит путь до последнего загруженного elf-файла.

#### **files**

Атрибут объекта, содержит экземпляр класса *FileStreamsManager*, позволяющий управлять файловыми потоками, открытыми для устройства.

#### **symbols**

Атрибут объекта, содержит символьную таблицу загруженного elf-файла.

#### **callbacks**

Атрибут объекта, содержит таблицу зарегистрированных callback'ов.

#### **call\_descriptors**

Атрибут объекта, содержит дескрипторы вызова функций (*FunctionCallDescriptor*), сформированные методом *get\_call\_descriptor()* для отслеживаемых ядер устройства.

### **breakpoints**

Атрибут объекта, содержит список установленных breakpoint'ов.

### **symbol\_prefix**

Это свойство содержит декоратор меток, который используется компилятором при сборке программы под выбранную платформу. По умолчанию декоратор отсутствует. Это свойство может быть переопределено в классе-наследнике.

### **physical\_addressing**

Это свойство содержит флаг режима адресации по умолчанию. True - все адреса по умолчанию считаются физическими, False - виртуальными (значение по умолчанию). Это свойство может быть переопределено в классе-наследнике.

### **registers\_explorer**

Свойство содержит тип explorer'а регистрового описания устройства. Explorer позволяет обращаться к регистрам и регистровым группам устройства, используя имена ядер или имена периферийных устройств. Получение регистра или регистровой группы осуществляется через запрос атрибута базового устройства. Например, запрос `device.risc[0].pc` вернёт регистровый объект *REG32*, связанный с регистром pc ядра risc0. Запрос `device.dsp[0].pcu` вернёт объект класса *registers\_explorer*, связанный с группой регистров pcu ядра dsp0. Explorer по умолчанию - *BaseRegistersExplorer*. Это свойство может быть переопределено в классе-наследнике.

### **allocate\_debugger(\*args, \*\*kwargs)**

Функция создаёт низкоуровневый отладчик, имплементирующий интерфейс *DebuggerInterface*. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

### **cwd**

Это свойство содержит текущую рабочую директорию устройства. Определяется в момент доступа и может быть либо текущей рабочей директорией интерпретатора, либо директорией, в которой находится последний загруженный elf-файл. Рабочая директория может быть изменена через непосредственное присваивание нового значения или методом *chdir()*.

### **check\_backend\_version()**

Функция проверяет версию библиотеки *libyapi.a* в загруженном elf файле. Порождает исключение, если версия не может быть вычислена или не соответствует ожидаемой.

### **enable\_syscalls(stdin=sys.stdin, stdout=sys.stdout, stderr=sys.stderr)**

Функция включает semihosting для устройства: устанавливает точку останова на метку *SYSCALL\_ENTRY* и добавляет обработчики системных вызовов в таблицу call-

back'ов. *stdin*, *stdout*, *stderr* - стандартные потоки ввода/вывода ОС (подробнее о стандартных потоках в модуле *sys*).

#### **enable\_environ()**

Инициализировать массив *char\*\* environ* перед заходом в функцию *main*. Позволяет использовать переменные окружения хоста в пользовательской программе.

#### **set\_argv(\*argv)**

Записать массив *char\*\* argv* данными из *argv*. *argv* - последовательность значений, элементы которой будут приведены к строке и соединены в одну строку). Путь к elf файлу (*argv[0]*) будет добавлен в начало *argv*. Вызов *set\_argv()* без параметров приводит к записи пути до исполняемой программы в *argv[0]*.

#### **add\_callback(event, func)**

Добавить функцию в таблицу callback'ов. *event* должен быть объектом класса *DebugBp* или целым числом (*int*, код прерывания, первый аргумент функции *SYSCALL\_FUNCTION*). *func* должен быть функцией, которая принимает один аргумент (объект любого класса, производного от *BaseDevice*) и возвращает *bool*.

#### **find\_symbol(symbol)**

Найти *symbol* в таблице *symbols*. Возвращает -1, если символ не найден.

#### **chdir(path)**

Изменить текущую рабочую директорию (*cwd*) на *path*.

#### **add\_breakpoint(address, callback=None, core\_id=-1)**

Установить точку останова на *address*. *address* должен быть целым числом (*int*) или строкой (*str*, значение из *symbols*). *callback* должен быть функцией, которая принимает один аргумент (объект любого класса, производного от *BaseDevice*), возвращает *bool* и будет вызвана в момент останова. Если *core\_id* имеет значение -1 (по умолчанию), точка останова будет выставлена на память или на нулевое ядро, если точки останова на память не поддерживаются; иначе точка останова будет установлена для ядра с индексом *core\_id*.

#### **clocks(core\_id=0)**

Возвращает значение счётчика тактов для ядра с индексом *core\_id*. По умолчанию возвращает 0.

#### **get\_register(register\_name)**

Возвращает значение регистра с именем *register\_name*.

#### **set\_register(register\_name, value)**

Установить значение *value* для регистра с именем *register\_name*.

#### **read\_word(address, physical=False)**

Возвращает 32-битное целое (`int`) значение из памяти по адресу `address`. Если параметр `physical` установлен в `True`, адрес будет трактоваться как физический, иначе - как виртуальный.

**`write_word(address, value, physical=False)`**

Записать 32-битное целое (`int`) значение `value` в память по адресу `address`. Если параметр `physical` установлен в `True`, адрес будет трактоваться как физический, иначе - как виртуальный.

**`read_memory(address, size, physical=False)`**

Возвращает регион памяти размера `size` по адресу `address` в виде `bytearray`. Если параметр `physical` установлен в `True`, адрес будет трактоваться как физический, иначе - как виртуальный.

**`write_memory(address, value, physical=False)`**

Записать регион памяти по адресу `address` значением `value`. Если параметр `physical` установлен в `True`, адрес будет трактоваться как физический, иначе - как виртуальный.

**`run(timer=float('inf'), dt=0.001, core_id=0)`**

Перевести устройство в `run`. `timer` - число с плавающей точкой (`float`), которые регулирует возвращение статуса `timeout`. Если параметр `timer` установлен в `None`, устройство будет запущено в неблокирующем режиме (без обработки прерываний). `dt` - число с плавающей точкой (`float`), которые период опроса состояния устройства `timeout`. `core_id` должен быть целым числом (`int`) или итерируемым контейнером, содержащим индексы ядер, которые должны быть переведены в `run`. Возвращает объект класса `Stop`.

**`stop(core_ids=0)`**

Остановить ядро устройства с индексом `core_ids`. `core_ids` должен быть целым числом (`int`) или итерируемым контейнером, содержащим индексы ядер, которые должны быть остановлены. Функция останавливает каждое ядро и вызывает `DebuggerInterface.stop()`.

**`step(count)`**

Сделать `count` шагов. Вызывает `DebuggerInterface.step()`.

**`load_elf(elf_path, update_syntab=False)`**

Загрузить `elf` файл, который находится по адресу `elf_path`, в память устройства. Если параметр `update_syntab` установлен в `False`, таблица символов будет перезаписана, иначе она будет обновлена.

**`get_string(string_address, max_size=1024)`**

Функция возвращает строку (`str`) длины `max_size` или меньше, если это нуль-терминированная строка. `string_address` - стартовый адрес строки.

**`disable_all_breakpoints()`**

Деактивировать все точки останова.

**get\_call\_descriptor()**

Возвращает дескриптор вызова функции (*FunctionCallDescriptor*) для текущего активного ядра. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**get\_pc(core\_id=0)**

Возвращает текущее значение регистра *pc* для ядра с индексом *core\_id*. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**set\_pc(value, core\_id=0)**

Установить текущее значение регистра *pc* для ядра с индексом *core\_id*. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**get\_active\_core()**

Возвращает индекс ядра, инициировавшего режим отладки. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**run\_core(core\_id)**

Запустить ядро с индексом *core\_id*. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**stop\_core(core\_id)**

Остановить ядро с индексом *core\_id*. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**get\_state\_core(core\_id)**

Возвращает статус ядра с индексом *core\_id*. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**pre\_run\_hook()**

Функция вызывается перед выполнением любых действий в методе *run()*. По умолчанию ничего не делает.

**post\_stop\_hook()**

Функция вызывается после любого события останова в методе *run()* перед его обработкой. По умолчанию ничего не делает.

**setup()**

Функция настраивает устройство, вызывается после всех этапов инициализации объекта. По умолчанию ничего не делает.

**class core.DebuggerInterface**

Интерфейсный класс для всех классов отладчиков (*GDB*, *MDB*, *Sim3x*, *ImgtecDebugger* и т.п.). Предоставляет базовые имплементации основных отладочных методов и обязательные для реализации абстрактные методы.

**load\_elf(path)**

Загрузить elf файл по пути *path* в память устройства. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**add\_breakpoint**(*address*, *core\_id*)

Установить точку останова на *address* для ядра с индексом *core\_id*. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**delete\_all\_breakpoints**()

Убрать все точки останова. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**get\_register**(*register\_name*)

Возвращает значение регистра с именем *register\_name*. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**set\_register**(*register\_name*, *value*)

Установить значение *value* для регистра с именем *register\_name*. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**write\_memory**(*address*, *value*, *physical*)

Записать регион памяти по адресу *address* значением *value*. Если параметр *physical* установлен в True, адрес будет трактоваться как физический, иначе - как виртуальный. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**read\_memory**(*address*, *size*, *physical*)

Возвращает регион памяти размера *size* по адресу *address* в виде bytearray. Если параметр *physical* установлен в True, адрес будет трактоваться как физический, иначе - как виртуальный. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**write\_word**(*address*, *value*, *physical*)

Записать 32-битное целое (int) значение *value* в память по адресу *address*. Если параметр *physical* установлен в True, адрес будет трактоваться как физический, иначе - как виртуальный. По умолчанию реализуется через DebuggerInterface.write\_memory().

**read\_word**(*address*, *physical*)

Возвращает 32-битное целое (int) значение ячейки памяти по адресу *address*. Если параметр *physical* установлен в True, адрес будет трактоваться как физический, иначе - как виртуальный. По умолчанию реализуется через DebuggerInterface.read\_memory().

**get\_registers\_description**()

Возвращает регистровую карту устройства. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**stop**()



Остановить устройство. По умолчанию ничего не делает.

**state()**

Получить статус устройства. По умолчанию ничего не делает.

**reset()**

Сбросить устройство. По умолчанию ничего не делает.

**run()**

Запустить устройство. По умолчанию ничего не делает.

**step(step\_count)**

Сделать *step\_count* шагов. По умолчанию недоступен.

**class core.FileStreamsManager**

Класс имитирует работу операционной системы с файловыми потоками.

**add(file\_object)**

Функция добавляет файловый поток (*file\_object*) к объекту *FileSystem*, связывает его с новым индексом потока и возвращает его.

**pop(fd)**

Функция удаляет файловый поток с индексом *fd*.

**close()**

Закрыть все открытые файловые потоки.

**class core.DebugBp**

Класс точки останова.

**address**

Адрес точки останова.

**core\_id**

Ядро, на которое выставлена точка останова. Значение - 1 соответствует точке останова на память.

**is\_set**

Свойство, возвращающее статус точки останова.

**enable()**

Активировать точку останова.

**disable()**

Деактивировать точку останова.

**class core.Stop**

Класс описывает причину остановки устройства.

**reasons**

Таблица кодов остановки и их расшифровки.

**time**

Время исполнения программы.

**reason**

Код причины остановки.

**address**

Адрес остановки.

**location**

Ближайшая к адресу остановки метка в elf-файле.

**class core.SymbolTable**

Класс символьной таблицы. Хранит адреса символьных меток. Наследует UserDict.

**mirror**

Список пар адрес-метка. Используется при определении функции, в которой произошёл останов.

**update\_mirror()**

Обновить *mirror*.

**add\_symbols\_from\_elf(elf\_path)**

Функция добавляет в символьную таблицу метки и их адреса из elf-файла (*elf\_path*).

### 5.1.1 core.pretty\_registers — Классы регистров и регистровых групп

---

Данный модуль предоставляет классы для регистров и регистровых групп ядер RISC, ARM, DSP и т.д.

**class core.pretty\_registers.DebuggerWrapper**

Класс-обёртка для отладчиков даёт возможность использовать объекты классов, реализующих *DebuggerInterface*, в качестве параметра *dev* конструктора класса *Register*.

**\_\_init\_\_(debugger, only\_mapped\_registers=False, physical\_addressing=False)**

Конструктор. *debugger* - любой объект класса, который реализует *DebuggerInterface*. *only\_mapped\_registers* - флаг. Если он задан, обмен данными с устройством будет идти через *read\_word()/write\_word()* интерфейс, иначе данные будут передаваться через *get\_register()/set\_register()* интерфейс. *physical\_addressing* - см. *physical\_addressing*.

**set\_register(register\_name, value)**

Установить значение *value* для регистра с именем *register\_name*.

**get\_register**(*register\_name*)

Возвращает значение регистра с именем *register\_name*.

**write\_word**(*address*, *value*)

Записать 32-битное целое значение *value* в память по адресу *address*.

**read\_word**(*address*)

Возвращает 32-битное целое значение из памяти по адресу *address*.

**class core.pretty\_registers.Register**

Абстрактный базовый класс для всех регистровых классов.

**Register**(*dev*, *pid*, *name*)

Конструктор. *dev* должен быть объектом любого класса, производного от *BaseDevice*. Требования к *pid* определяются классом-наследником. *name* - строка, содержащая имя регистра, используется в строковом представлении. Если *dev* имеет атрибут *only\_mapped\_registers*, обмен данными с устройством будет идти через *read\_word()/write\_word()* интерфейс, иначе данные будут передаваться через *get\_register()/set\_register()* интерфейс.

**id**

Атрибут объекта, содержит идентификатор регистра (или идентификаторы нескольких регистров, если он составной).

**bit\_size**

Это свойство содержит битовый размер регистра. Это абстрактное свойство и должно быть определено в классе-наследнике.

**value**

Это свойство, как и методы *get()/set()*, даёт доступ к текущему значению регистра и позволяет его изменять.

**get()**

Возвращает текущее значение регистра. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**set**(*value*)

Установить значение *value* для регистра. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**set\_bit**(*bit\_pos*, *value*)

Установить бит *bit\_pos* в регистре в *value*.

**wait\_bit**(*bit\_pos*, *stop\_value*, *timeout=0.1*)

Ждать *timeout* секунд, пока бит *bit\_pos* в регистре не будет установлен в *value*.

**class core.pretty\_registers.REG16**

Класс, описывающий 16-битные регистры. Наследует класс *Register*.

**REG16**(*dev, pid, name*)

Конструктор. *pid* должен быть кортежем (*tuple*) из двух элементов, где первый элемент определяет часть 32-битного слова (0 - старшая, 1 - младшая), а второй элемент - идентификатор регистра или адрес памяти.

**to\_signed\_short**()

Возвращает *signed short* представление текущего значения регистра.

**class core.pretty\_registers.REG32**

Класс, описывающий 32-битные регистры. Наследует класс *Register*.

**REG32**(*dev, pid, name*)

Конструктор. *pid* должен быть идентификатором регистра или адресом памяти.

**to\_signed\_int**()

Возвращает *signed int* представление текущего значения регистра.

**to\_float**()

Возвращает *float* представление текущего значения регистра.

**class core.pretty\_registers.REG64**

Класс, описывающий 64-битные регистры. Наследует класс *Register*.

**REG64**(*dev, pid, name*)

Конструктор. *pid* должен быть кортежем (*tuple*) из двух элементов, где каждый элемент - идентификатор регистра или адрес памяти. Элементы формируют регистр от младшей части к старшей.

**to\_signed\_long**()

Возвращает *signed long* представление текущего значения регистра.

**to\_double**()

Возвращает *double* представление текущего значения регистра.

**class core.pretty\_registers.REG128**

Класс, описывающий 128-битные регистры. Наследует класс *Register*.

**REG128**(*dev, pid, name*)

Конструктор. *pid* должен быть кортежем (*tuple*) из четырёх элементов, где каждый элемент - идентификатор регистра или адрес памяти. Элементы формируют регистр от младшей части к старшей.

**Классы суффиксов**

Ниже представлены несколько суффиксных классов, описывающих регистры ядер DSP. Конструкторы этих классов имеют общий прототип, он описан ниже:

**class** `core.pretty_registers.Suffix`

Абстрактный базовый класс для всех суффиксных классов.

**\_\_init\_\_**(*device*, *pid\_table*, *base\_name*)

Конструктор. *device* должен быть объектом любого класса, производного от *BaseDevice*. *pid\_table* - таблица, в которой разрядности интерфейса соответствуют id регистров. *base\_name* - базовое имя регистра (полное имя без суффикса).

**\_init\_register**(*interface*)

Возвращает инициализированный объект класса *interface*. Ожидается, что *interface* - один из интерфейсных классов регистров (*REG16*, *REG32*, etc), соответствующий по разрядности суффиксу (*.l*, *.s*, etc), с которым связан.

**class** `core.pretty_registers.D`

Класс для DSP регистров, которые поддерживают суффикс *.d* (64 bit). Этот класс наследует *Suffix*.

**d**

Это свойство возвращает *None*, если id регистра не может быть вычислен, иначе возвращается объект класса *REG64*.

**class** `core.pretty_registers.SL`

Класс для DSP регистров, которые поддерживают суффиксы *.s* (16 bit) и *.l* (32 bit). Этот класс наследует *Suffix*.

**s**

Это свойство возвращает *None*, если id регистра не может быть вычислен, иначе возвращается объект класса *REG16*.

**l**

Это свойство возвращает *None*, если id регистра не может быть вычислен, иначе возвращается объект класса *REG32*.

**class** `core.pretty_registers.LDQ`

Класс для DSP регистров, которые поддерживают суффиксы *.l* (32 bit), *.d* (64 bit) и *.q* (128 bit). Этот класс наследует *Suffix*.

**l**

Это свойство возвращает *None*, если id регистра не может быть вычислен, иначе возвращается объект класса *REG32*.

**d**

Это свойство возвращает *None*, если id регистра не может быть вычислен, иначе возвращается объект класса *REG64*.

**q**

Это свойство возвращает *None*, если id регистра не может быть вычислен, иначе возвращается объект класса *REG128*.

**class core.pretty\_registers.SLD**

Класс для DSP регистров, которые поддерживают суффиксы *.s* (16 bit), *.l* (32 bit) и *.d* (64 bit). Этот класс наследует *Suffix* и может быть использован в качестве значения свойства *suffix*.

**s**

Это свойство возвращает *None*, если *id* регистра не может быть вычислен, иначе возвращается объект класса *REG16*.

**l**

Это свойство возвращает *None*, если *id* регистра не может быть вычислен, иначе возвращается объект класса *REG32*.

**d**

Это свойство возвращает *None*, если *id* регистра не может быть вычислен, иначе возвращается объект класса *REG64*.

**class core.pretty\_registers.SLDQ**

Класс для DSP регистров, которые поддерживают суффиксы *.s* (16 bit), *.l* (32 bit), *.d* (64 bit) и *.q* (128 bit). Этот класс наследует *Suffix* и может быть использован в качестве значения свойства *suffix*.

**s**

Это свойство возвращает *None*, если *id* регистра не может быть вычислен, иначе возвращается объект класса *REG16*.

**l**

Это свойство возвращает *None*, если *id* регистра не может быть вычислен, иначе возвращается объект класса *REG32*.

**d**

Это свойство возвращает *None*, если *id* регистра не может быть вычислен, иначе возвращается объект класса *REG64*.

**q**

Это свойство возвращает *None*, если *id* регистра не может быть вычислен, иначе возвращается объект класса *REG128*.

**class core.pretty\_registers.BaseRegistersExplorer**

Базовый класс *explorer*'а, обеспечивающий доступ к регистрам устройства через расширение его атрибутов. При формировании цепочки атрибутов нужно учитывать следующие правила:

1. Имена регистров, регистровых групп и устройств должны совпадать с именами в регистровой карте.
2. Индексирование поддерживают только регистры, имена которых заканчиваются числовым индексом.

3. Название регистровой группы можно опустить, если оно совпадает с названием устройства или соответствует одному из правил *skip\_rules*.

Доступ к регистру через explorer начинается с указания базового устройства. Это может быть любое периферийное устройство или вычислительное ядро, описанное в регистровой карте. Затем указывается регистровая группа, если это требуется, и имя самого регистра. Для регистров, поддерживающих суффиксы (в соответствии с *suffix\_classes*), таким образом можно получить доступ к суффиксной группе (*SLDQ*, *SLD* и т.п.), для прочих регистров возвращается 32х разрядный интерфейс *REG32*.

Регистровые группы, регистры в группах и все индексированные объекты поддерживают итерации.

### **suffix\_classes**

Это свойство содержит таблицу правил, по которым осуществляется выбор суффиксных групп для регистров, которые поддерживают суффиксные расширения. По умолчанию суффиксных правил нет. Это свойство может быть переопределено в классе-наследнике.

### **skip\_rules**

Это свойство содержит список правил в виде регулярных выражений, которые описывают имена базовых устройств, для которых можно опускать указание регистровых групп. По умолчанию можно не указывать группы для ядер risc и dsp. Это свойство может быть переопределено в классе-наследнике.

## **5.1.2 core.syscalls — Обработчики системных вызовов POSIX**

Данный модуль предоставляет обработчики для некоторых системных вызовов POSIX, доступные в newlib.

### **core.syscalls.SYSCALL\_FUNCTION**

Эта глобальная переменная содержит имя функции, которая является точкой входа для обработчиков системных вызовов.

### **core.syscalls.SYSCALL\_ENTRY**

Эта глобальная переменная содержит имя метки, на которую ставится точка останова (*add\_breakpoint()*) функцией *enable\_syscalls()*.

### **core.syscalls.ENVIRON\_INIT\_ENTRY**

Эта глобальная переменная содержит имя метки, на которую ставится точка останова (*add\_breakpoint()*) функцией *enable\_environ()*.

### **core.syscalls.ARGV\_INIT\_ENTRY**

Эта глобальная переменная содержит имя метки, на которую ставится точка останова средствами отладчика gdb функцией *set\_argv()*.

### **core.syscalls.BACKEND\_VERSION\_LABEL**

Эта глобальная переменная содержит имя метки, содержащий версию C backend'а системных вызовов.

**core.syscalls.MIN\_BACKEND\_VERSIONS**

Эта глобальная переменная содержит номер минимальной поддерживаемой версии C backend'a.

**core.syscalls.unpack\_args(dev, address, size=3, bit\_version=32)**

Функция возвращает кортеж (tuple) длины *size*, который содержит значения 32-битных ячеек памяти из диапазона [*address*: *address* + *ptr\_size* \* *size*], где *ptr\_size* - *bit\_version* / 8. *dev* должен быть объектом любого класса, производного от *BaseDevice*.

ElcoreAPI поддерживает модель обратных вызовов для точек останова. Обработчики событий могут быть связаны с точками останова при их объявлении (аргумент *callback* функции *add\_breakpoint()*) или через метод *add\_callback()* (таким образом могут быть зарегистрированы новые системные вызовы, т.е. вызовы, реализованные через *SYSCALL\_FUNCTION*). Функция-обработчик должна принимать 1 аргумент (объект любого класса-наследника *BaseDevice*) и возвращать любое значение, кроме None. Если функция-обработчик возвращает None, устройство будет остановлено по причине срабатывания точки останова.

Для точек останова обработчики могут быть назначены следующим образом:

```
def my_callback(dev):
    print 'i was called by this device:', dev
    return 0

device.add_breakpoint(0x200000000, callback=my_callback)
```

Или это можно сделать вручную с помощью *add\_callback()*:

```
def my_callback(dev):
    print 'i was called by this device:', dev
    return 0

bp = device.add_breakpoint(0x200000000)
...
device.add_callback(bp, my_callback)
```

Для обработчиков событий останова, использующих *SYSCALL\_FUNCTION*, код системного вызова (первый аргумент функции *SYSCALL\_FUNCTION*) должен быть передан в *add\_callback()* и поставлена точка останова на метку *SYSCALL\_ENTRY*. Если при этом в то же время требуется использовать стандартные обработчики системных вызовов, достаточно вызвать *enable\_syscalls()*. Пример:

```
// somewhere in C program, which is linked with libpyapi
extern int __syscall(int code, void* args);

void my_python_caller(void * args)
{
    int answer = __syscall(0xC0, args);
}
```

Код 0xC0 должен быть связан с функцией-обработчиком:



```
def my_callback(dev):
    print 'code: 0xC0. i was called by this device:', dev
    return 0

device.add_callback(0xC0, my_callback)
entry = device.find_symbol(SYSCALL_ENTRY)
if entry != -1:
    device.add_breakpoint(entry)
```

Ниже представлено несколько функций, обрабатывающих поддерживаемые в newlib системные вызовы POSIX. Эти функции реализуют в соответствии со стандартом поведение операционной системы при срабатывании соответствующих системных вызовов. Файловые операции осуществляются в системе, на которой запущен отладчик. Возвращаемое значение сохраняется в соответствии с соглашением о вызовах. Для работы с обработчиками системных вызовов программа должна быть собрана с библиотекой `libyapi` (`-lyapi5`, поставляется вместе с инструментами под все поддерживаемые платформы). Для активации обработчиков системных вызовов достаточно вызвать `enable_syscalls()` перед `run()`. Важно: программа должна быть загружена (`load_elf()`) до включения обработчиков.

```
core.syscalls.sys_read(device)
    Обработчик POSIX read().
```

```
core.syscalls.sys_write(device)
    Обработчик POSIX write().
```

```
core.syscalls.sys_lseek(device)
    Обработчик POSIX lseek().
```

```
core.syscalls.sys_open(device)
    Обработчик POSIX open().
```

```
core.syscalls.sys_close(device)
    Обработчик POSIX close().
```

```
core.syscalls.sys_fstat(device)
    Обработчик POSIX fstat().
```

```
core.syscalls.sys_stat(device)
    Обработчик POSIX stat().
```

```
core.syscalls.sys_times(device)
    Обработчик POSIX times(). Не рекомендуется к использованию при оценке времени (вызывается функцией clock). При заполнении структуры tms заполняется только поле tms_utime. Реализация считывания счётчика тактов отличается для различных устройств, вследствие чего полученные значения могут значительно отличаться. Рекомендуется использовать метод sys_gettimeofday().
```

```
core.syscalls.sys_gettimeofday(device)
    Обработчик POSIX gettimeofday().
```

```
core.syscalls.sys_link(device)
    Обработчик POSIX link().
```

`core.syscalls.sys_unlink(device)`

Обработчик POSIX unlink().

`core.syscalls.sys_exit(device)`

Обработчик POSIX exit().

`core.syscalls.sys_isatty(device)`

Обработчик POSIX isatty().

`core.syscalls.sys_chdir(device)`

Обработчик POSIX chdir().

`core.syscalls.SYSCALL_TABLE`

Таблица (dict), содержащая пары код-обработчик для поддерживаемых системных вызовов.

#### Таблица кодов обработчиков системных вызовов:

Код	Обработчик
0x01	sys_exit()
0x03	sys_read()
0x04	sys_write()
0x05	sys_open()
0x06	sys_close()
0x09	sys_link()
0x0a	sys_unlink()
0x0c	sys_chdir()
0x12	sys_stat()
0x13	sys_lseek()
0x1c	sys_fstat()
0x2b	sys_times()
0x4e	sys_gettimeofday()
0xbf	sys_isatty()

`core.syscalls.environ_handler(device)`

Обработчик останова по метке ENVIRON\_INIT\_ENTRY. Инициализирует массивы по меткам `environ_labels` переменными окружения хоста.

### 5.1.3 core.utils — Общие классы и функции ElcoreAPI

Данный модуль предоставляет различные функции и классы, которые могут быть полезны пользователю и используются самой библиотекой.

#### class core.utils.FunctionCallDescriptor

Объект `namedtuple` описывает дескриптор вызова функции для текущего активного ядра. Имеет атрибуты `arg0`, `arg1`, `arg2`, `return_reg`, `return_reg64`, которые содержат указатели на объекты классов, производных от класса `Register`, через которые передаются первые три аргумента функции и возвращаемое значение в соответствии с соглашением о вызовах.

#### class core.utils.CDLLMethod

Объект `namedtuple` описывает прототип C функции из загружаемой библиотеки.

Имеет следующие атрибуты: *name* - имя функции, *arg\_types* - кортеж типов *ctypes*, соответствующих типам параметров функции, *res\_type* - *ctypes* тип возвращаемого значения (None, если функция ничего не возвращает).

#### **class core.utils.MethodCall**

Объект *namedtuple* описывает вызов метода на стороне сервера отладки. Имеет следующие атрибуты: *method* - имя метода, *args* - параметры вызова.

#### **class core.utils.MethodResponse**

Объект *namedtuple* описывает возвращаемое значение, полученное на вызов метода на стороне сервера отладки. Имеет следующие атрибуты: *err\_state* - код ошибки (0 - нет ошибки), *value* - возвращаемое значение метода.

#### **class core.utils.CDLLWrapper**

Базовый класс для обёрток над загружаемыми библиотеками.

##### **cdll\_methods**

Кортеж элементов CDLLMethod.

##### **\_\_init\_\_(lib\_handler)**

Конструктор. *lib\_handler* - обработчик загруженной библиотеки.

##### **driver\_methods**

Таблица указателей на методы загруженной библиотеки. В качестве ключей используются имена функция.

#### **core.utils.load\_lib(library\_path)**

Функция возвращает обработчик загруженной библиотеки. *library\_path* может содержать только имя библиотеки или полный путь до неё. В первом случае её поиск будет проходить по путям переменных окружения.

#### **core.utils.normalize\_path(cwd, path)**

Функция вычисляет путь относительно рабочей директории. *cwd* - рабочая директория. *path* - относительный или абсолютный путь. Если *path* - абсолютный путь, функция вернёт *path*, иначе - *cwd* + *path*.

#### **core.utils.\_\_function\_\_()**

Метод возвращает имя функции, из которой он был вызван.

#### **core.utils.lower\_keys(source)**

Функция создаёт копию регистровой карты *source*, переводя все ключи в нижний регистр.

#### **class core.utils.SocketBasedTransfer**

Класс описывает протокол передачи данных по TCP/IP в режиме непрерывной сессии.

##### **head\_size**

Размер заголовка пакета.

##### **head\_format**

Формат заголовка пакета (синтаксис *struct*).

**block\_size**

Максимальный размер блока данных для чтения.

**get(connection)**

Получить пакет. *connection* - socket соединение. Ожидается, что перед пакетом будет следовать заголовок *head\_format*, содержащий его размер.

**send(connection, value)**

Отправить пакет. *connection* - socket соединение. *value* - пакет в виде `bytearray`. К отсылаемому пакету будет добавлен заголовок *head\_format*.

**class core.utils.Options**

Базовый класс для опций приложений с интерфейсом командной строки.

**create\_argument\_parser()**

Метод класса, создаёт объект `argparse.ArgumentParser`. Этот метод является абстрактным и должен быть имплементирован в классе-наследнике.

**from\_command\_line()**

Альтернативный конструктор объекта, создаёт парсер параметров с помощью *create\_argument\_parser*, вызывает разбор параметров командной строки, передаёт их в конструктор опций и возвращает получившийся объект.

## 5.2 platforms — Поддерживаемые платформы

---

Данный раздел содержит абстрактные классы, описывающие отладочную логику для поддерживаемых платформ, и классы, управляющие доступом к регистрам устройств.

### 5.2.1 platforms.explorers — Классы, управляющие доступом к регистрам

---

Данный модуль предоставляет классы `explorer`'ов регистровых описаний устройств.

**class platforms.explorers.Elcore30RegistersExplorer**

Класс обеспечивает доступ к суффиксным расширениям регистров ядер Elcore30, наследует *BaseRegistersExplorer*. Суффиксные расширения доступны для регистров `r(.s, .l, .d, .q)`, `ac(.l, .d, .q)` и `x(.d)`.

### 5.2.2 platforms.mcom02 — Отладочная логика процессора MCom02

---

**class platforms.mcom02.AbstractMCom02**

Абстрактный класс, описывающий отладочную логику процессора MCom02. Наследует класс *BaseDevice*.

**registers\_explorer**

*Elcore30RegistersExplorer*

**get\_call\_descriptor()**

Функция возвращает описание соглашения о вызовах для текущего активного ядра. r0, r1, r2 - регистры параметров, r0 - регистр возвращаемого значения.

**get\_pc(core\_id=0)**

Функция возвращает значение регистра risc[core\_id].pc.

**set\_pc(value, core\_id=0)**

Функция устанавливает значение регистра risc[core\_id].pc.

**get\_active\_core()**

Функция возвращает индекс текущего активного dsp ядра, всегда возвращает 0.

**run\_core(core\_id)**

Метод ничего не делает.

**stop\_core(core\_id)**

Метод ничего не делает.

**get\_state\_core(core\_id)**

Функция всегда возвращает True (признак run).

### 5.2.3 platforms.mct03p — Отладочная логика процессора MCT03P

---

**class platforms.mct03p.AbstractMCT03P**

Абстрактный класс, описывающий отладочную логику процессора MCT03P. Наследует класс *AbstractMIPS32*.

### 5.2.4 platforms.mips32 — Отладочная логика процессоров MIPS32

---

**class platforms.mips32.AbstractMIPS32**

Абстрактный класс, описывающий отладочную логику процессоров с MIPS32 CPU. Наследует класс *BaseDevice*.

**get\_call\_descriptor()**

Функция возвращает описание соглашения о вызовах для текущего активного ядра. `a0`, `a1`, `a2` - регистры параметров, `v0` - регистр возвращаемого значения.

**get\_pc**(*core\_id*=0)

Функция возвращает значение регистра `risc[core_id].pc`.

**set\_pc**(*value*, *core\_id*=0)

Функция устанавливает значение регистра `risc[core_id].pc`.

**get\_active\_core**()

Функция возвращает индекс текущего активного dsp ядра, всегда возвращает 0.

**run\_core**(*core\_id*)

Метод ничего не делает.

**stop\_core**(*core\_id*)

Метод ничего не делает.

**get\_state\_core**(*core\_id*)

Функция всегда возвращает `True` (признак `run`).

### 5.2.5 **platforms.nvcom02t** — Отладочная логика процессора NVCom02T

---

**class** `platforms.nvcom02t.AbstractNVCom02T`

Абстрактный класс, описывающий отладочную логику процессора NVCom02T. Наследует класс *AbstractMIPS32*.

**registers\_explorer**

*Elcore30RegistersExplorer*

### 5.2.6 **platforms.mc24r** — Отладочная логика процессора MC24R

---

**class** `platforms.mc24r.AbstractMC24R`

Абстрактный класс, описывающий отладочную логику процессора MC24R. Наследует класс *AbstractMIPS32*.

## 5.3 **gdb** — GDB расширения ElcoreAPI

---

Данный модуль предоставляет GDB расширения ElcoreAPI.

**`gdb.chdir(path)`**

Сменить рабочую директорию для обработчиков системных вызовов на *path*. По умолчанию, рабочей директорией является домашняя директория программы, если она может быть получена через отладчик, иначе это рабочая директория отладчика.

**`gdb.enable_syscalls(stdin=sys.stdin, stdout=sys.stdout, stderr=sys.stderr)`**

Статическая версия метода *enable\_syscalls()*, которая использует GDB интерфейсы.

**`gdb.enable_environ()`**

Статическая версия метода *enable\_environ()*, которая использует GDB интерфейсы.

**`gdb.set_argv(*argv)`**

Проинициализировать массив `char** argv` при старте программы, если платформой поддерживается метод `__get_elcore_argv`. Если функция вызвана без аргументов (*argv*), в `char** argv` будут записаны аргументы, переданные программе через gdb интерфейсы (через опцию `--args` при запуске отладчика или через параметры команды `run`), иначе в `char** argv` будут записаны объекты из *argv*, которые будут приведены к строке.

**`gdb.set_device(dev_class)`**

Задать класс устройства вручную. *dev\_class* должен быть классом-наследником *GDB-Device*. По умолчанию класс устройства будет выбран автоматически.

**`gdb.add_callback(event, func)`**

Добавить обработчик события останова в таблицу обработчиков. Ожидается, что *event* будет целым числом (`int`, код прерывания, первый аргумент метода *SYSCALL\_FUNCTION*). *func* - функция, принимающая один аргумент и возвращающая любое значение, кроме `None`. Если функция-обработчик возвращает `None`, отладка будет остановлена по причине срабатывания точки останова.

**`gdb.execute(command)`**

Выполнить команду gdb через интерфейс `gdb.post_event`. Такой механизм выполнения предпочтителен для команд, влияющих на состояние отладчика (`run`, `continue`, `step` и т.п.).

**`gdb.create_sim3x_mgdb_target(config, host='localhost', port=22222, log_file=None)`**

Функция создаёт экземпляр симулятора с заданной конфигурацией (*config*). Параметры *host* и *port* передаются `mgdbserver`'у и команде `gdb target extended-remote`. Файл логирования `mgdbserver`'а задаётся через параметр *log\_file*. По умолчанию логирование выключено.

**`gdb.core`**

Объект класса *CoreManager*, позволяет управлять устройством, как объектом *GDB-Device*.

**`class gdb.CallbackBreakpoint`**

Подкласс `gdb.Breakpoint`, используемый для реализации вызова callback-методов при срабатывании `internal breakpoint`'ов.

`__init__(device, *args, **kwargs)`

Конструктор. *device* - объект любого класса, наследующего *GDBDevice*. *args/kwargs* - параметры конструктора *gdb.Breakpoint*.

#### **action**

Атрибут объекта, содержит указатель на callback-метод. По умолчанию это `lambda _: None`, т.е. метод, всегда приводящий к останову.

#### **device**

Атрибут объекта, содержит указатель на текущее gdb-устройство (*GDBDevice*).

#### **stop()**

Функция, вызываемая при срабатывании точки останова. Вызывает метод *action*, передавая ему *device* в качестве параметра. Возвращаемое значение *action* проверяется на `None`. В случае, если callback вернёт `None` или породит исключение, срабатывает точка останова.

### **class gdb.GDBSymbolTable**

Класс управляет доступом к символьной информации загруженных в gdb исполняемых файлов. Поддерживает методы словаря `__getitem__`, `__contains__`, `__iter__` и `get`.

### **class gdb.GDB**

Класс отладчика gdb-устройств (*GDBDevice*), реализует интерфейс *DebuggerInterface*.

#### **load\_elf(path)**

Метод не поддерживается в этой реализации интерфейса.

#### **add\_breakpoint(address, core\_id)**

Метод не поддерживается в этой реализации интерфейса.

#### **get\_register(register\_name)**

Получить значение регистра с именем *register\_name* через `gdb.parse_and_eval` и синтаксис `$`. Значение приводится к `unsigned long`.

#### **set\_register(register\_name, value)**

Установить значение *value* для регистра с именем *register\_name* через `gdb.execute` и команду `set`.

#### **write\_memory(address, array\_of\_bytes, physical)**

Запись в память значения *array\_of\_bytes* через `gdb.Inferior.write_memory`. Параметр *physical* не используется.

#### **read\_memory(address, size, physical)**

Чтение памяти устройства через `gdb.Inferior.read_memory`. Память считывается блоками по 1 Кб. Параметр *physical* не используется.

#### **delete\_all\_breakpoints()**

Метод не поддерживается в этой реализации интерфейса.



**step(step\_count)**

Выполнить step\_count шагов через *execute()* и команду *step*.

**get\_registers\_description()**

Функция возвращает пустую регистровую карту.

**class gdb.GDBDevice**

Базовый абстрактный класс gdb-устройств, наследует *BaseDevice*.

**allocate\_debugger(\*args, \*\*kwargs)**

Функция создаёт экземпляр отладчика *GDB*.

**symbols**

Атрибут объекта, содержит символьную таблицу *GDBSymbolTable*.

**dev\_names**

Это свойство содержит список имен устройств, которые поддерживаются данным классом устройства. Это абстрактное свойство и оно должно быть определено в классе-наследнике.

**call\_descriptor\_ids**

Это свойство содержит имена регистров, используемых для передачи параметров в функцию и для возвращаемого значения. По умолчанию регистры не определены, обращение к свойству порождает исключение. Свойство может быть переопределено в классе-наследнике.

**add\_breakpoint(address, callback=None, core\_id=-1)**

Поставить точку останова на память в виде *CallbackBreakpoint*. Параметр *core\_id* не используется.

**get\_pc(core\_id=0)**

Функция возвращает значение регистра с именем *pc*. Параметр *core\_id* не используется.

**set\_pc(value, core\_id=0)**

Функция устанавливает значение регистра с именем *pc*. Параметр *core\_id* не используется.

**get\_active\_core()**

Функция возвращает индекс текущего активного ядра. Всегда возвращает 0.

**run\_core(core\_id)**

Метод не поддерживается в этой реализации интерфейса.

**stop\_core(core\_id)**

Метод не поддерживается в этой реализации интерфейса.

**get\_state\_core(core\_id)**

Метод не поддерживается в этой реализации интерфейса.

**class gdb.DSPGDBDevice**

Базовый абстрактный класс gdb-устройств, использующих DSP ядра. Наследует *GDBDevice*.

**clocks**(*core\_id*=0)

Возвращает значение счётчика тактов для ядра с индексом *core\_id*, которое находится с помощью команды `gdb monitor clock-count index`. В качестве параметра (индекс ядра) передаётся `0x1000 + core_id`.

**class gdb.RISCGDBDevice**

Базовый абстрактный класс gdb-устройств, использующих RISC ядра. Наследует *GDBDevice*.

**clocks**(*core\_id*=0)

Возвращает значение счётчика тактов для ядра с индексом *core\_id*, которое находится с помощью команды `gdb monitor clock-count index`. В качестве параметра (индекс ядра) передаётся 1.

**class gdb.CoreManager**

Класс, управляющий созданием экземпляра gdb-устройства (*GDBDevice*). Устройство может быть создано как в автоматическом режиме при доступе к свойству *device*, так и с помощью явного указания класса устройства (*device\_class*) через функцию *set\_device()*.

**device**

Это свойство содержит указатель на текущее gdb-устройство.

**device\_class**

Это свойство содержит указатель на класс текущего gdb-устройства.

**get\_target()**

Функция возвращает имя подключенного устройства и его тип через команды `mdb monitor show chipname` и `monitor show backend`.

### 5.3.1 gdb.targets — Классы gdb-устройств

---

Этот модуль предоставляет классы gdb-устройств для поддерживаемых чипов.

**class gdb.targets.MCom02**

Класс gdb-устройства для ядра ARM. Наследует *RISCGDBDevice* и *AbstractMCom02*.

**dev\_names**

Поддерживаемые имена устройств: `mcom02`, `mcom-02`.

**get\_call\_descriptor()**

Регистры, использующиеся для передачи параметров в функцию и для возвращаемого значения: `г0`, `г1`, `г2`, `г0`. `г0:г1` используются для возвращения 64х битного значения.

**class gdb.targets.MIPS32**

Класс gdb-устройства для ядра MIPS32. Класс наследует *RISCGDBDevice* и *AbstractMIPS32*.

**dev\_names**

Поддерживаемые имена устройств: `nvcom-02`, `nvcom02t`, `nvcom-02t`, `nvcom-01`, `mct-03p`, `mc-30sf6`, `mc-24r`, `mc-24m`, `lde-vega`, `mc-0428`, `mc-12m`, `mc-226m`, `mct-04r`, `mck-02`.

**get\_call\_descriptor()**

Регистры, использующиеся для передачи параметров в функцию и для возвращаемого значения: `г0`. `г0:г1` используются для возвращения 64х битного значения.

## 6. СООБЩЕНИЯ ОБ ОШИБКАХ

В этом разделе приведены сообщения об ошибках, которые могут быть получены в процессе работы с ElcoreAPI, расшифровка этих сообщений и способы их устранения. Некоторые сообщения приведены не полностью, т.к. должны содержать данные, относящиеся к конкретному сеансу отладки.

### 1. ImportError: No module named ElcoreAPI

Описание:

Библиотека ElcoreAPI не установлена.

Решение:

Установить библиотеку ElcoreAPI.

### 2. ImportError: No module named elftools

Описание:

Библиотека pyelftools не установлена.

Решение:

Установить библиотеку pyelftools (`pip install pyelftools`).

### 3. EnvironmentError: unsupported version of libpyapi

Описание:

Пользовательская программа не прошла проверку на используемую версию `libpyapi`, либо была собрана без неё.

Решение:

Удостовериться, что в маршруте сборки есть `libpyapi` и/или обновить инструменты сборки до последней версии.

### 4. RuntimeError: no symbol information, can't enable syscalls

Описание:

К моменту включения обработчиков системных вызовов пользовательская программа не была загружена, либо символьная информация из пользовательской программы была разобрана с ошибками.

Решение:

Удостовериться, что программа загружена и её символьная информация может быть разобрана с помощью `readelf.py` (входит в состав `pyelftools`).

### 5. OSError: path does not exist: %PATH%

Описание:

Новый путь рабочей директории недоступен.

Решение:

Создать требуемую иерархию.

6. RuntimeError: error while setting breakpoint at %ADDRESS%

Описание:

Ошибка со стороны отладчика при установке точки останова.

Решение:

Проверить значение точки останова и/или удостовериться, что она может быть установлена через иной интерфейс отладки.

7. RuntimeError: error while loading ELF

Описание:

Ошибка со стороны отладчика при загрузке программы.

Решение:

Проверить правильность сборки программы и/или удостовериться, что она может быть загружена через иной интерфейс отладки.

8. OSError: ELF file not found: %PATH%

Описание:

Загружаемая программа не найдена.

Решение:

Проверить наличие программы по указанному пути.

9. KeyError: can't find symbol %SYMBOL% in elf

Описание:

Метка, указывающая на массив argv, не найдена при его инициализации.

Решение:

Проверить правильность сборки программы.

10. ValueError: argv is too long (%SIZE% > %MAX\_SIZE%)

Описание:

Размер argv превысил максимально возможный.

Решение:

Уменьшить размер данных, передаваемых в `argv` и/или убедиться в том, что они корректно приводятся к строковому типу данных.

11. `RuntimeError: %PLATFORM% does not support argv`

Описание:

Выбранная платформа не поддерживает работу с `argv`.

Решение:

Отказаться от использования `set_argv()`.

12. `TypeError: Expected 'dev_class' to be subclass of %BASE_CLASS%; was %CLASS%`

Описание:

Выбранный класс устройства не является производным от базового класса GDB устройства.

Решение:

Наследование от класса `GDB`.

13. `SystemError: unknown chip (%CHIP%-%PLATFORM%), ElcoreAPI is not supported`

Описание:

Конфигурация устройства не описана ни в одном классе устройства и `semihosting` не может быть настроен автоматически.

Решение:

Использовать метод `set_device()` с одним из классов `gdb` устройств в качестве аргумента.

14. `RuntimeError: %METHOD%() is unsupported`

Описание:

Данный метод не поддерживается выбранным классом устройства.

Решение:

Не использовать данный метод.

15. `RuntimeError: can't allocate IModel3 instance, unknown problem`

Описание:

Ошибка при создании объекта отладки.

Решение:

Уведомить разработчика библиотеки.

16. `SystemError: can't find ports, platform is not connected`

Описание:

Отладчик не обнаружил ни одной подключенной платы.

Решение:

Подключить хотя бы одну плату к ПК и/или удостовериться, что она определяется в MDB/GDB.

17. EnvironmentError: can't find chip with name "%CHIP\_NAME"

Описание:

Отладчик не обнаружил выбранное устройство.

Решение:

Убедиться, что нужная плата подключена к ПК и к ней подведено питание.

18. KeyError: port %PORT% not found

Описание:

Выбранный порт устройства не найден среди подключенных устройств.

Решение:

Проверить значение параметра *port* при создании экземпляра устройства.

19. SystemError: attempt to access to unknown chip, access refused

Описание:

Отладчик не смог идентифицировать чип.

Решение:

Уведомить разработчика отладчика.

20. SystemError: can't connect to any chip with name %NAME%

Описание:

Новая отладочная сессия не может быть открыта, т.к. не удалось подключиться ни одному устройству с именем %NAME%.

Решение:

Закрыть все открытые отладочные сессии для выбранного устройства.

21. OSError: configuration file not found: %PATH%

Описание:

Конфигурационный файл симулятора не найден.

Решение:

Проверить существование конфигурационного файла по указанному пути.

22. RuntimeError: error while setting configuration: %CONFIG%

Описание:

Ошибка при создании симулятора выбранной конфигурации.

Решение:

Уведомить разработчика симулятора.

23. OSError: ddr init program not found: %PATH%

Описание:

Программа инициализации ddr не найдена.

Решение:

Проверить существование программы инициализации ddr по указанному пути.

24. SystemError: no free 'dbsar' registers

Описание:

Невозможно установить точку останова, т.к. все отладочные регистры заняты.

Решение:

Уменьшить количество одновременно используемых точек останова.

25. RuntimeError: connect

Описание:

Ошибка подключения к отладочной плате.

Решение:

Удостовериться в корректности имени устройства и/или в его доступности через Codescape Debugger.

26. RuntimeError: Fail: mode %MODE%

Описание:

Ошибка инициализации интерфейса отладки.

Решение:

Удостовериться в корректности инициализации интерфейса отладки через Codescape Debugger.

27. RuntimeError: Fail to run (stopped at address %ADDRESS%)

Описание:



Ошибка инициализации ddr.

Решение:

Удостовериться в корректности программы инициализации ddr и/или инициализации ddr через Codescape Debugger.

28. IndexError: register index out of range

Описание:

Выход за границу индексов, доступных для регистровой группы.

Решение:

Удостовериться в корректности вычисления индекса регистра.

29. OSError: python and lib platforms (32/64) must be the same

Описание:

Разрядность интерпретатора не соответствует разрядности загружаемой библиотеки.

Решение:

Использовать другой релиз загружаемой библиотеки.

30. OSError: bad path to lib, file not found: %PATH%

Описание:

Загружаемая библиотека не найдена.

Решение:

Проверить существование загружаемой библиотеки по указанному пути.

31. OSError: can't load lib, unknown problem

Описание:

Неизвестная ошибка при загрузке библиотеки.

Решение:

Уведомить разработчика библиотеки.

32. KeyError: undefined symbol: %SYMBOL%

Описание:

Запрашиваемый символ не найден в таблице символов загруженной программы.

Решение:

Удостовериться, что программа загружена, её символьная информация может быть разобрана с помощью readelf.py (входит в состав pyelftools) и искомый символ может быть найден.

33. ValueError: %HANDLER% is not callable

Описание:

Обработчик события не является вызываемым объектом.

Решение:

Проверить тип данных обработчика события.

34. Attempt to read a non-existent file stream with fd number %INDEX%

Описание:

Предупреждение обработчика системного вызова. Попытка чтения из несуществующего файлового потока.

Решение:

Проверить корректность операции чтения в загружаемой программе.

35. Attempt to write a non-existent file stream with fd number %INDEX%

Описание:

Предупреждение обработчика системного вызова. Попытка записи в несуществующий файловый поток.

Решение:

Проверить корректность операции записи в загружаемой программе.

36. Attempt to seek a non-existent file stream with fd number %INDEX%

Описание:

Предупреждение обработчика системного вызова. Попытка сдвига каретки в несуществующем файловом потоке.

Решение:

Проверить корректность операции сдвига каретки в загружаемой программе.

37. IOError: EINVAL: whence is not valid

Описание:

Предупреждение обработчика системного вызова. Ошибка сдвига каретки.

Решение:

Проверить корректность операции сдвига каретки в загружаемой программе.

38. Attempt to access a non-existent directory %PATH%

Описание:

Предупреждение обработчика системного вызова. Доступ к несуществующей директории.

Решение:

Проверить существование указанной иерархии.

39. Attempt to read a non-existent file %PATH%

Описание:

Предупреждение обработчика системного вызова. Доступ на чтение к несуществующему файлу.

Решение:

Проверить существование указанного файла.

40. Attempt to close a non-existent file stream with fd number %INDEX%

Описание:

Предупреждение обработчика системного вызова. Попытка закрытия несуществующего файлового потока.

Решение:

Проверить корректность операции закрытия файлового потока в загружаемой программе.

41. Attempt to check a non-existent file stream with fd number %INDEX%

Описание:

Предупреждение обработчика системного вызова. Попытка проверки на TTY несуществующего файлового потока.

Решение:

Проверить корректность операции проверки на TTY в загружаемой программе.

42. Attempt to change working directory to the non-existent path: %PATH%

Описание:

Предупреждение обработчика системного вызова. Изменение рабочей директории на несуществующий путь.

Решение:

Проверить существование указанного файла.

43. Error when opening: %MSG%

Описание:

Предупреждение обработчика системного вызова. Ошибка при открытии файла.

Решение:

Проверить возможность открытия такого файла в операционной системе отладчика.

44. symbol table is corrupted, any non ascii symbols in elf build path? (%PATH%)

Описание:

Предупреждения парсера символьной таблицы. Символьная таблица разобрана с ошибками, данные могут быть некорректны. Такое поведение возможно при использовании символов кириллицы в сборочных путях под Linux.

Решение:

Удостовериться, что символьная информация загружаемой программы может быть разобрана с помощью readelf.py (входит в состав pyelftools).

45. %METHOD%: NULL pointer access

Описание:

Предупреждение обработчика системного вызова. Один из параметров системного вызова является нулевым указателем, хотя не должен быть таковым.

Решение:

Проверить корректность алгоритма исполняемой программы.

46. KeyError: unknown register “%NAME%”

Описание:

Имя регистра не соответствует ни одному из известных отладчику.

Решение:

Проверить корректность имени регистра.

47. RuntimeError: error while executing command “%CMD%”

Описание:

При выполнении команды mdb возникла ошибка.

Решение:

Проверить корректность команды mdb с помощью отладчика.

48. SystemError: device %DEVICE% does not have a registered calling convention

Описание:

Выбранный класс GDB устройства не имеет описания соглашения о вызовах (информация о регистрах, используемых для передачи параметров в функцию и для возвращаемого значения).

Решение:

Переопределить свойство *call\_descriptor\_ids*.

49. RuntimeError: error while writing [%BEGIN%:%END%]

Описание:

Ошибка записи региона памяти со стороны низкоуровневого отладчика.

Решение:

Удостовериться в исправности отлаживаемого устройства, канала связи и актуальности используемых инструментов.

50. RuntimeError: error while reading [%BEGIN%:%END%]

Описание:

Ошибка чтения региона памяти со стороны низкоуровневого отладчика.

Решение:

Удостовериться в исправности отлаживаемого устройства, канала связи и актуальности используемых инструментов.

51. RuntimeError: error while loading elf

Описание:

Ошибка загрузки elf-файла в память устройства.

Решение:

Удостовериться в исправности отлаживаемого устройства, канала связи, актуальности используемых инструментов, корректности сборки программы.

52. RuntimeError: error while loading .dat

Описание:

Ошибка загрузки dat-файла в память устройства.

Решение:

Удостовериться в исправности отлаживаемого устройства, канала связи, актуальности используемых инструментов, корректности dat-файла.

53. ConnectionError

Описание:

Ошибка подключения к удалённому отладчику.

Решение:

Удостовериться в том, что сервер отладчика запущен и имеет актуальную версию. Проверить исправность канала связи.

54. RuntimeError: bit %BIT\_POS% of %REGISTER% was not set to %VALUE% in the expected time

Описание:

Ошибка установки бита регистра.

Решение:

Удостовериться с помощью документации, что бит не read only и timeout задан корректно.

55. ConnectionError: can't get the server version; update server

Описание:

Ошибка при попытке получить версию сервера отладчика.

Решение:

Обновить версию сервера.

56. ConnectionError: server version (%VERSION%) does not match expected (%VERSION%)

Описание:

Ошибка при попытке соединения с устаревшим сервером отладчика.

Решение:

Обновить версию сервера.

## СОДЕРЖАНИЕ МОДУЛЕЙ PYTHON

### C

core, 9

core.pretty\_registers, 16

core.syscalls, 21

core.utils, 24

### g

gdb, 28

gdb.targets, 32

### p

platforms, 26

platforms.explorers, 26

platforms.mc24r, 28

platforms.mcom02, 26

platforms.mct03p, 27

platforms.mips32, 27

platforms.nvcom02t, 28

## АЛФАВИТНЫЙ УКАЗАТЕЛЬ

\_\_function\_\_() (в модуле core.utils), 25  
 \_\_init\_\_() (метод core.BaseDevice), 9  
 \_\_init\_\_() (метод core.pretty\_registers.DebuggerWrapper), 16  
 \_\_init\_\_() (метод core.pretty\_registers.Suffix), 19  
 \_\_init\_\_() (метод core.utils.CDLLWrapper), 25  
 \_\_init\_\_() (метод gdb.CallbackBreakpoint), 29  
 \_init\_register() (метод core.pretty\_registers.Suffix), 19  
 AbstractMC24R (класс в platforms.mc24r), 28  
 AbstractMCom02 (класс в platforms.mcom02), 27  
 AbstractMCT03P (класс в platforms.mct03p), 27  
 AbstractMIPS32 (класс в platforms.mips32), 27  
 AbstractNVCom02T (класс в platforms.nvcom02t), 28  
 action (атрибут gdb.CallbackBreakpoint), 30  
 active\_core (атрибут core.BaseDevice), 9  
 add() (метод core.FileStreamsManager), 15  
 add\_breakpoint() (метод core.BaseDevice), 11  
 add\_breakpoint() (метод core.DebuggerInterface), 14  
 add\_breakpoint() (метод gdb.GDB), 30  
 add\_breakpoint() (метод gdb.GDBDevice), 31  
 add\_callback() (в модуле gdb), 29  
 add\_callback() (метод core.BaseDevice), 11  
 add\_symbols\_from\_elf() (метод core.SymbolTable), 16  
 address (атрибут core.DebugBp), 15  
 address (атрибут core.Stop), 16  
 allocate\_debugger() (метод core.BaseDevice), 10  
 allocate\_debugger() (метод gdb.GDBDevice), 31  
 ARGV\_INIT\_ENTRY (в модуле core.syscalls), 21  
 BACKEND\_VERSION\_LABEL (в модуле core.syscalls), 21  
 BaseDevice (класс в core), 9  
 BaseRegistersExplorer (класс в core.pretty\_registers), 20  
 bit\_size (атрибут core.pretty\_registers.Register), 17  
 block\_size (атрибут core.utils.SocketBasedTransfer), 25  
 breakpoints (атрибут core.BaseDevice), 10  
 call\_descriptor\_ids (атрибут gdb.GDBDevice), 31  
 call\_descriptors (атрибут core.BaseDevice), 9  
 CallbackBreakpoint (класс в gdb), 29  
 callbacks (атрибут core.BaseDevice), 9  
 cdll\_methods (атрибут core.utils.CDLLWrapper), 25  
 CDLLMethod (класс в core.utils), 24  
 CDLLWrapper (класс в core.utils), 25  
 chdir() (в модуле gdb), 28  
 chdir() (метод core.BaseDevice), 11  
 check\_backend\_version() (метод core.BaseDevice), 10  
 clocks() (метод core.BaseDevice), 11  
 clocks() (метод gdb.DSPGDBDevice), 32  
 clocks() (метод gdb.RISCGDBDevice), 32  
 close() (метод core.FileStreamsManager), 15  
 core (в модуле gdb), 29  
 core (модуль), 9  
 core.pretty\_registers (модуль), 16  
 core.syscalls (модуль), 21  
 core.utils (модуль), 24  
 core\_id (атрибут core.DebugBp), 15  
 CoreManager (класс в gdb), 32  
 create\_argument\_parser() (метод core.utils.Options), 26  
 create\_sim3x\_mgdb\_target() (в модуле gdb), 29  
 cwd (атрибут core.BaseDevice), 10  
 d (атрибут core.pretty\_registers.D), 19



d (атрибут core.pretty\_registers.LDQ), 19  
d (атрибут core.pretty\_registers.SLD), 20  
d (атрибут core.pretty\_registers.SLDQ), 20  
D (класс в core.pretty\_registers), 19  
DebugBp (класс в core), 15  
debugger (атрибут core.BaseDevice), 9  
DebuggerInterface (класс в core), 13  
DebuggerWrapper (класс в core.pretty\_registers), 16  
delete\_all\_breakpoints() (метод core.DebuggerInterface), 14  
delete\_all\_breakpoints() (метод gdb.GDB), 30  
description (атрибут core.BaseDevice), 9  
dev\_names (атрибут gdb.GDBDevice), 31  
dev\_names (атрибут gdb.targets.MCom02), 32  
dev\_names (атрибут gdb.targets.MIPS32), 33  
device (атрибут gdb.CallbackBreakpoint), 30  
device (атрибут gdb.CoreManager), 32  
device\_class (атрибут gdb.CoreManager), 32  
disable() (метод core.DebugBp), 15  
disable\_all\_breakpoints() (метод core.BaseDevice), 12  
driver\_methods (атрибут core.utils.CDLLWrapper), 25  
DSPGDBDevice (класс в gdb), 32  
Elcore30RegistersExplorer (класс в platforms.explorers), 26  
elf (атрибут core.BaseDevice), 9  
enable() (метод core.DebugBp), 15  
enable\_environ() (в модуле gdb), 29  
enable\_environ() (метод core.BaseDevice), 11  
enable\_syscalls() (в модуле gdb), 29  
enable\_syscalls() (метод core.BaseDevice), 10  
environ\_handler() (в модуле core.syscalls), 24  
ENVIRON\_INIT\_ENTRY (в модуле core.syscalls), 21  
execute() (в модуле gdb), 29  
files (атрибут core.BaseDevice), 9  
FileStreamsManager (класс в core), 15  
find\_symbol() (метод core.BaseDevice), 11  
from\_command\_line() (метод core.utils.Options), 26  
FunctionCallDescriptor (класс в core.utils), 24  
GDB (класс в gdb), 30  
gdb (модуль), 28  
gdb.targets (модуль), 32  
GDBDevice (класс в gdb), 31  
GDBSymbolTable (класс в gdb), 30  
get() (метод core.pretty\_registers.Register), 17  
get() (метод core.utils.SocketBasedTransfer), 26  
get\_active\_core() (метод core.BaseDevice), 13  
get\_active\_core() (метод gdb.GDBDevice), 31  
get\_active\_core() (метод platforms.mcom02.AbstractMCom02), 27  
get\_active\_core() (метод platforms.mips32.AbstractMIPS32), 28  
get\_call\_descriptor() (метод core.BaseDevice), 12  
get\_call\_descriptor() (метод gdb.targets.MCom02), 32  
get\_call\_descriptor() (метод gdb.targets.MIPS32), 33  
get\_call\_descriptor() (метод platforms.mcom02.AbstractMCom02), 27  
get\_call\_descriptor() (метод platforms.mips32.AbstractMIPS32), 27  
get\_pc() (метод core.BaseDevice), 13  
get\_pc() (метод gdb.GDBDevice), 31  
get\_pc() (метод platforms.mcom02.AbstractMCom02), 27  
get\_pc() (метод platforms.mips32.AbstractMIPS32), 28  
get\_register() (метод core.BaseDevice), 11  
get\_register() (метод core.DebuggerInterface), 14  
get\_register() (метод core.pretty\_registers.DebuggerWrapper), 17  
get\_register() (метод gdb.GDB), 30  
get\_registers\_description() (метод core.DebuggerInterface), 14

get\_registers\_description() (метод gdb.GDB), 31  
 get\_state\_core() (метод core.BaseDevice), 13  
 get\_state\_core() (метод gdb.GDBDevice), 31  
 get\_state\_core() (метод platforms.mcom02.AbstractMCom02), 27  
 get\_state\_core() (метод platforms.mips32.AbstractMIPS32), 28  
 get\_string() (метод core.BaseDevice), 12  
 get\_target() (метод gdb.CoreManager), 32  
 head\_format (атрибут core.utils.SocketBasedTransfer), 25  
 head\_size (атрибут core.utils.SocketBasedTransfer), 25  
 id (атрибут core.pretty\_registers.Register), 17  
 is\_set (атрибут core.DebugBp), 15  
 l (атрибут core.pretty\_registers.LDQ), 19  
 l (атрибут core.pretty\_registers.SL), 19  
 l (атрибут core.pretty\_registers.SLD), 20  
 l (атрибут core.pretty\_registers.SLDQ), 20  
 LDQ (класс в core.pretty\_registers), 19  
 load\_elf() (метод core.BaseDevice), 12  
 load\_elf() (метод core.DebuggerInterface), 13  
 load\_elf() (метод gdb.GDB), 30  
 load\_lib() (в модуле core.utils), 25  
 location (атрибут core.Stop), 16  
 lower\_keys() (в модуле core.utils), 25  
 MCom02 (класс в gdb.targets), 32  
 MethodCall (класс в core.utils), 25  
 MethodResponse (класс в core.utils), 25  
 MIN\_BACKEND\_VERSIONS (в модуле core.syscalls), 22  
 MIPS32 (класс в gdb.targets), 33  
 mirror (атрибут core.SymbolTable), 16  
 normalize\_path() (в модуле core.utils), 25  
 Options (класс в core.utils), 26  
 physical\_addressing (атрибут core.BaseDevice), 10  
 platforms (модуль), 26  
 platforms.explorers (модуль), 26  
 platforms.mc24r (модуль), 28  
 platforms.mcom02 (модуль), 26  
 platforms.mct03p (модуль), 27  
 platforms.mips32 (модуль), 27  
 platforms.nvcom02t (модуль), 28  
 pop() (метод core.FileStreamsManager), 15  
 post\_stop\_hook() (метод core.BaseDevice), 13  
 pre\_run\_hook() (метод core.BaseDevice), 13  
 q (атрибут core.pretty\_registers.LDQ), 19  
 q (атрибут core.pretty\_registers.SLDQ), 20  
 read\_memory() (метод core.BaseDevice), 12  
 read\_memory() (метод core.DebuggerInterface), 14  
 read\_memory() (метод gdb.GDB), 30  
 read\_word() (метод core.BaseDevice), 11  
 read\_word() (метод core.DebuggerInterface), 14  
 read\_word() (метод core.pretty\_registers.DebuggerWrapper), 17  
 reason (атрибут core.Stop), 16  
 reasons (атрибут core.Stop), 15  
 REG128 (класс в core.pretty\_registers), 18  
 REG128() (метод core.pretty\_registers.REG128), 18  
 REG16 (класс в core.pretty\_registers), 17  
 REG16() (метод core.pretty\_registers.REG16), 18  
 REG32 (класс в core.pretty\_registers), 18  
 REG32() (метод core.pretty\_registers.REG32), 18  
 REG64 (класс в core.pretty\_registers), 18  
 REG64() (метод core.pretty\_registers.REG64), 18  
 Register (класс в core.pretty\_registers), 17  
 Register() (метод core.pretty\_registers.Register), 17  
 registers\_explorer (атрибут core.BaseDevice), 10  
 registers\_explorer (атрибут platforms.mcom02.AbstractMCom02), 27  
 registers\_explorer (атрибут platforms.nvcom02t.AbstractNVCom02T), 27

- 28
- reset() (метод core.DebuggerInterface), 15
- RISCGDBDevice (класс в gdb), 32
- run() (метод core.BaseDevice), 12
- run() (метод core.DebuggerInterface), 15
- run\_core() (метод core.BaseDevice), 13
- run\_core() (метод gdb.GDBDevice), 31
- run\_core() (метод platformforms.mcom02.AbstractMCom02), 27
- run\_core() (метод platformforms.mips32.AbstractMIPS32), 28
- s (атрибут core.pretty\_registers.SL), 19
- s (атрибут core.pretty\_registers.SLD), 20
- s (атрибут core.pretty\_registers.SLDQ), 20
- send() (метод core.utils.SocketBasedTransfer), 26
- set() (метод core.pretty\_registers.Register), 17
- set\_argv() (в модуле gdb), 29
- set\_argv() (метод core.BaseDevice), 11
- set\_bit() (метод core.pretty\_registers.Register), 17
- set\_device() (в модуле gdb), 29
- set\_pc() (метод core.BaseDevice), 13
- set\_pc() (метод gdb.GDBDevice), 31
- set\_pc() (метод platformforms.mcom02.AbstractMCom02), 27
- set\_pc() (метод platformforms.mips32.AbstractMIPS32), 28
- set\_register() (метод core.BaseDevice), 11
- set\_register() (метод core.DebuggerInterface), 14
- set\_register() (метод core.pretty\_registers.DebuggerWrapper), 16
- set\_register() (метод gdb.GDB), 30
- setup() (метод core.BaseDevice), 13
- skip\_rules (атрибут core.pretty\_registers.BaseRegistersExplorer), 21
- SL (класс в core.pretty\_registers), 19
- SLD (класс в core.pretty\_registers), 19
- SLDQ (класс в core.pretty\_registers), 20
- SocketBasedTransfer (класс в core.utils), 25
- state() (метод core.DebuggerInterface), 15
- step() (метод core.BaseDevice), 12
- step() (метод core.DebuggerInterface), 15
- step() (метод gdb.GDB), 30
- Stop (класс в core), 15
- stop() (метод core.BaseDevice), 12
- stop() (метод core.DebuggerInterface), 14
- stop() (метод gdb.CallbackBreakpoint), 30
- stop\_core() (метод core.BaseDevice), 13
- stop\_core() (метод gdb.GDBDevice), 31
- stop\_core() (метод platformforms.mcom02.AbstractMCom02), 27
- stop\_core() (метод platformforms.mips32.AbstractMIPS32), 28
- Suffix (класс в core.pretty\_registers), 18
- suffix\_classes (атрибут core.pretty\_registers.BaseRegistersExplorer), 21
- symbol\_prefix (атрибут core.BaseDevice), 10
- symbols (атрибут core.BaseDevice), 9
- symbols (атрибут gdb.GDBDevice), 31
- SymbolTable (класс в core), 16
- sys\_chdir() (в модуле core.syscalls), 24
- sys\_close() (в модуле core.syscalls), 23
- sys\_exit() (в модуле core.syscalls), 24
- sys\_fstat() (в модуле core.syscalls), 23
- sys\_gettimeofday() (в модуле core.syscalls), 23
- sys\_isatty() (в модуле core.syscalls), 24
- sys\_link() (в модуле core.syscalls), 23
- sys\_lseek() (в модуле core.syscalls), 23
- sys\_open() (в модуле core.syscalls), 23
- sys\_read() (в модуле core.syscalls), 23
- sys\_stat() (в модуле core.syscalls), 23
- sys\_times() (в модуле core.syscalls), 23
- sys\_unlink() (в модуле core.syscalls), 23
- sys\_write() (в модуле core.syscalls), 23
- SYSCALL\_ENTRY (в модуле core.syscalls), 21
- SYSCALL\_FUNCTION (в модуле core.syscalls), 21
- SYSCALL\_TABLE (в модуле core.syscalls), 24
- time (атрибут core.Stop), 16

`to_double()` (метод `core.pretty_registers.REG64`), 18

`to_float()` (метод `core.pretty_registers.REG32`), 18

`to_signed_int()` (метод `core.pretty_registers.REG32`), 18

`to_signed_long()` (метод `core.pretty_registers.REG64`), 18

`to_signed_short()` (метод `core.pretty_registers.REG16`), 18

`unpack_args()` (в модуле `core.syscalls`), 22

`update_mirror()` (метод `core.SymbolTable`), 16

`value` (атрибут `core.pretty_registers.Register`), 17

`wait_bit()` (метод `core.pretty_registers.Register`), 17

`write_memory()` (метод `core.BaseDevice`), 12

`write_memory()` (метод `core.DebuggerInterface`), 14

`write_memory()` (метод `gdb.GDB`), 30

`write_word()` (метод `core.BaseDevice`), 12

`write_word()` (метод `core.DebuggerInterface`), 14

`write_word()` (метод `core.pretty_registers.DebuggerWrapper`), 17