ELCOREAPI DOCUMENTATION

Версия 5.3.7
01.08.2019

ОГЛАВЛЕНИЕ

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[1] для программ, запускаемых на устройствах.

[1] https://developer.arm.com/docs/dui0471/k/what-is-semihosting/what-is-semihosting

2. ТРЕБОВАНИЯ

Требуемые сторонние библиотеки:

- pyelftools[2] - 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`).

[2] https://pypi.python.org/pypi/pyelftools

3. УСТАНОВКА

Для работы с библиотекой рекомендуется использовать virtualenv[3]. Это позволит устанавливать любые пакеты без root прав с пользовательскими настройками proxy. Использование virtualenv позволяет производить любые модификации рабочего окружения python без влияния на работу других проектов.

Установка и создание virtualenv:

```
sudo pip install virtualenv  # --proxy proxy.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

[3] 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 `libpyapi.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* предаются 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 `libpyapi.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 плавающий 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 плавающий 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_symtab=False*)

Загрузить elf файл, который находится по адресу *elf_path*, в память устройства. Если параметр *update_symtab* 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 нультерминированная 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 произошел 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 аттрибутов. При формировании цепочки аттрибутов 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'а.

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(0x20000000, 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(0x20000000)
...
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 `libpyapi` (`-lpyapi5`, 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 *GDBDevice*. 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 *GDBDevice*.

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, сработает 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, использующихся 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: `r0`, `r1`, `r2`, `r0`. `r0:r1` 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 зна

`v0`. `v0:v1` 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 неё подсоединено 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 устройство с именем %NAME%.

    Решение:

    Закрыть все отрытые 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, использующихся 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

g

p

АЛФАВИТНЫЙ УКАЗАТЕЛЬ