

# H1 Micro Unicorn-Engine API Documentation

Warning: This is an unofficial API document by [kabeor](#). If there are any mistakes, welcome to ask.

**注意：**这是由kabeor制作的非官方API参考文档，如有错误欢迎提出，觉得不错可以给个star鼓励我

之前对Capstone反汇编引擎的API分析文档已经被[官方](#)收录 <https://github.com/kabeor/Micro-Capstone-Engine-API-Documentation>，在实现自己想要做出的调试器的路上，又遇到了与Capstone同作者的国外大佬aquynh的另一个著名项目Unicorn，不巧的是，详尽的API文档仍然较少，更多的是大篇幅的代码，因此决定继续分析Unicorn框架，包括数据类型，已开放API及其实现。

Unicorn是一个轻量级, 多平台, 多架构的CPU模拟器框架，基于qemu开发，它可以代替CPU模拟代码的执行，常用于恶意代码分析，Fuzz等，该项目被用于Radare2逆向分析框架，GEF(gdb的pwn分析插件)，Pwndbg，Angr符号执行框架等多个著名项目。接下来我也将通过阅读源码和代码实际调用来写一个简单的非官方版本的API手册。

Blog: [kabeor.cn](http://kabeor.cn)

## H2 0xo 开发准备

Unicorn官网: <http://www.unicorn-engine.org>

## H3 自行编译lib和dll方法

源码: <https://github.com/unicorn-engine/unicorn/archive/master.zip>

下载后解压

文件结构如下:

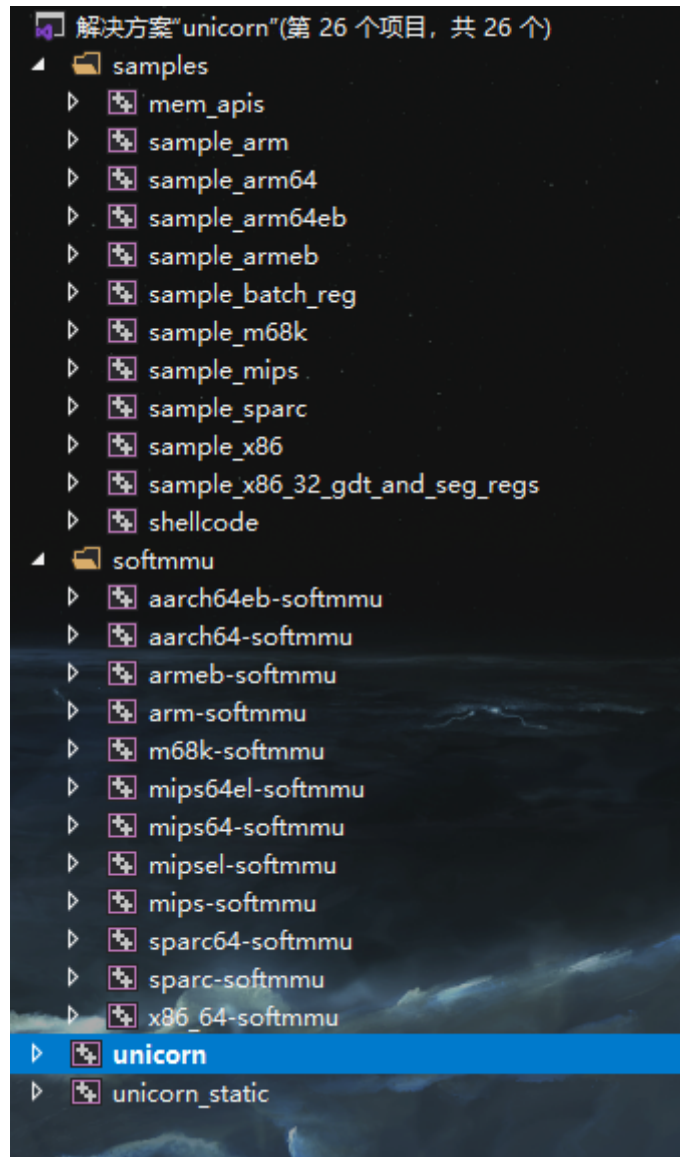
```
1  . <- 主要引擎core engine + README + 编译文档COMPILE.TXT 等
2  |— arch <- 各语言反编译支持的代码实现
3  |— bindings <- 中间件
4  |   |— dotnet <- .Net 中间件 + 测试代码
5  |   |— go <- go 中间件 + 测试代码
6  |   |— haskell <- Haskell 中间件 + 测试代码
7  |   |— java <- Java 中间件 + 测试代码
8  |   |— pascal <- Pascal 中间件 + 测试代码
9  |   |— python <- Python 中间件 + 测试代码
10 |   |— ruby <- Ruby 中间件 + 测试代码
11 |   |— vb6 <- VB6 中间件 + 测试代码
12 |— docs <- 文档，主要是Unicorn的实现思路
13 |— include <- C头文件
14 |— msvc <- Microsoft Visual Studio 支持 (Windows)
15 |— qemu <- qemu框架源码
16 |— samples <- Unicorn使用示例
17 |— tests <- C语言测试用例
```

下面演示Windows10使用Visual Studio2019编译

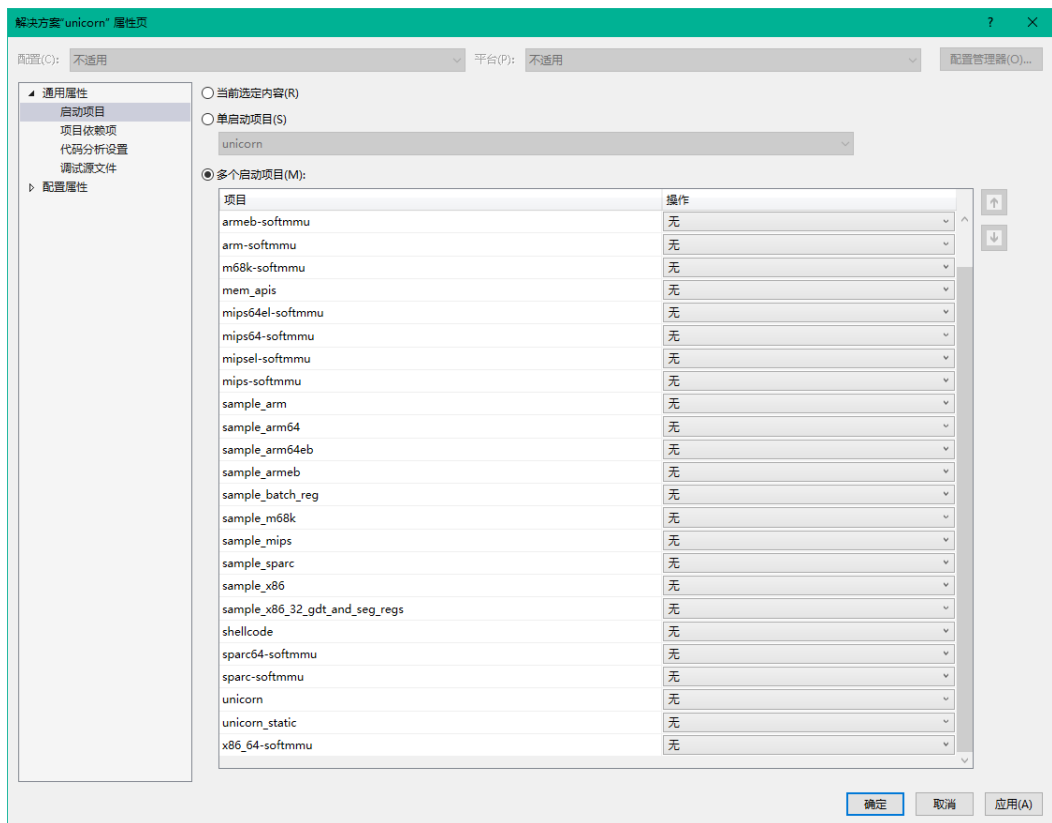
打开msvc文件夹，内部结构如下

名称	修改日期	类型	大小
samples	2020/1/17 18:26	文件夹	
unicorn	2020/1/17 18:26	文件夹	
.gitignore	2020/1/15 22:18	GITIGNORE 文件	1 KB
README.TXT	2020/1/15 22:18	文本文档	9 KB
unicorn.sln	2020/1/15 22:18	Visual Studio Sol...	27 KB

VS打开unicorn.sln项目文件，解决方案自动载入这些



如果都需要的话，直接编译就好了，只需要其中几种，则右键解决方案->属性->配置属性如下



生成选项中勾选你需要的支持项即可

项目编译属性为：

1. 使用多字节字符集
2. 不使用预编译头
3. 附加选项 /wd4018 /wd4244 /wd4267
4. 预处理器定义中添加 `_CRT_SECURE_NO_WARNINGS`

编译后会在当前文件夹Debug目录下生成unicorn.lib静态编译库和unicorn.dll动态库这样就可以开始使用Unicorn进行开发了

编译到最后项可能会报错系统找不到指定的路径，查看makefile发现问题出现在此处

```
error MSB3073: 命令“mkdir ..\Unicorn\unicorn-master\msvc\distro\include”
error MSB3073: mkdir ..\Unicorn\unicorn-master\msvc\distro\include
error MSB3073: mkdir ..\Unicorn\unicorn-master\msvc\distro\include\unicorn
error MSB3073: mkdir ..\Unicorn\unicorn-master\msvc\distro\include\unicorn
error MSB3073: copy ..\Unicorn\unicorn-master\msvc\Release\unicorn.dll ..\Unicorn\unicorn-master\msvc\distro\include\unicorn
error MSB3073: copy ..\Unicorn\unicorn-master\msvc\Release\unicorn.lib ..\Unicorn\unicorn-master\msvc\distro\include\unicorn
error MSB3073: copy ..\Unicorn\unicorn-master\msvc\include\unicorn*.h ..\Unicorn\unicorn-master\msvc\distro\include\unicorn
error MSB3073: :VCEnd”已退出，代码为 9009。
```

事实上只不过是未能将生成的lib和dll复制到新建的文件夹而已，只需要到生成目录去找即可。

官方目前提供的最新已编译版本为1.0.1版本，比较老，建议自己编辑最新版本源码，以获得更多可用API。

Win32: <https://github.com/unicorn-engine/unicorn/releases/download/1.0.1/unicorn-1.0.1-win32.zip>

Win64: <https://github.com/unicorn-engine/unicorn/releases/download/1.0.1/unicorn-1.0.1-win64.zip>

**注意：**选x32或x64将影响后面开发的位数

### H3 引擎调用测试

新建一个VS项目，将..\unicorn-master\include\unicorn中的头文件以及编译好的lib和dll文件全部拷贝到新建项目的主目录下

名称	修改日期	类型	大小
.vs	2020/1/17 17:23	文件夹	
Debug	2020/1/17 17:30	文件夹	
unicorn	2020/1/17 17:25	文件夹	
x64	2020/1/17 17:30	文件夹	
unicorn.dll	2020/1/17 17:02	应用程序扩展	4,479 KB
unicorn.lib	2020/1/17 17:02	Object File Library	7 KB
Unicorn_Demo.cpp	2020/1/17 17:38	c file	3 KB
Unicorn_Demo.sln	2020/1/17 17:23	Visual Studio Sol...	2 KB
Unicorn_Demo.vcxproj	2020/1/17 17:32	VC++ Project	8 KB
Unicorn_Demo.vcxproj.filters	2020/1/17 17:30	VC++ Project Fil...	2 KB
Unicorn_Demo.vcxproj.user	2020/1/17 17:23	Per-User Project...	1 KB

在VS解决方案中，头文件添加现有项unicorn.h，资源文件中添加unicorn.lib，重新生成解决方案



接下来测试我们生成的unicorn框架

主文件代码如下

```

1  #include <iostream>
2  #include "unicorn/unicorn.h"
3
4  // 要模拟的指令
5  #define X86_CODE32 "\x41\x4a" // INC ecx; DEC edx
6
7  // 起始地址
8  #define ADDRESS 0x1000000
9
10 int main()
11 {
12     uc_engine* uc;
13     uc_err err;
14     int r_ecx = 0x1234;    // ECX 寄存器
15     int r_edx = 0x7890;    // EDX 寄存器
16
17     printf("Emulate i386 code\n");
18
19     // X86-32bit 模式初始化模拟
20     err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
21     if (err != UC_ERR_OK) {
22         printf("Failed on uc_open() with error returned:
23         %u\n", err);
24     }
25     return -1;

```

```

24     }
25
26     // 给模拟器申请 2MB 内存
27     uc_mem_map(uc, ADDRESS, 2 * 1024 * 1024, UC_PROT_ALL);
28
29     // 将要模拟的指令写入内存
30     if (uc_mem_write(uc, ADDRESS, X86_CODE32,
31 sizeof(X86_CODE32) - 1)) {
32         printf("Failed to write emulation code to memory,
33         quit!\n");
34         return -1;
35     }
36
37     // 初始化寄存器
38     uc_reg_write(uc, UC_X86_REG_ECX, &r_ecx);
39     uc_reg_write(uc, UC_X86_REG_EDX, &r_edx);
40
41     printf(">>> ECX = 0x%x\n", r_ecx);
42     printf(">>> EDX = 0x%x\n", r_edx);
43
44     // 模拟代码
45     err = uc_emu_start(uc, ADDRESS, ADDRESS +
46 sizeof(X86_CODE32) - 1, 0, 0);
47     if (err) {
48         printf("Failed on uc_emu_start() with error
49         returned %u: %s\n",
50         err, uc_strerror(err));
51     }
52
53     // 打印寄存器值
54     printf("Emulation done. Below is the CPU context\n");
55
56     uc_reg_read(uc, UC_X86_REG_ECX, &r_ecx);
57     uc_reg_read(uc, UC_X86_REG_EDX, &r_edx);
58     printf(">>> ECX = 0x%x\n", r_ecx);
59     printf(">>> EDX = 0x%x\n", r_edx);
60
61     uc_close(uc);
62
63     return 0;
64 }

```

运行结果如下

```

Emulate i386 code
>>> ECX = 0x1234
>>> EDX = 0x7890
Emulation done. Below is the CPU context
>>> ECX = 0x1235
>>> EDX = 0x788f

```

ecx+1和edx-1成功模拟。

### H3 uc\_arch

架构选择

```
1  typedef enum uc_arch {
2      UC_ARCH_ARM = 1,      // ARM 架构 (包括 Thumb, Thumb-2)
3      UC_ARCH_ARM64,      // ARM-64, 也称 AArch64
4      UC_ARCH_MIPS,      // Mips 架构
5      UC_ARCH_X86,      // X86 架构 (包括 x86 & x86-64)
6      UC_ARCH_PPC,      // PowerPC 架构 (暂不支持)
7      UC_ARCH_SPARC,      // Sparc 架构
8      UC_ARCH_M68K,      // M68K 架构
9      UC_ARCH_MAX,
10 } uc_arch;
```

### H3 uc\_mode

模式选择

```
1  typedef enum uc_mode {
2      UC_MODE_LITTLE_ENDIAN = 0,      // 小端序模式 (默认)
3      UC_MODE_BIG_ENDIAN = 1 << 30,  // 大端序模式
4
5      // arm / arm64
6      UC_MODE_ARM = 0,      // ARM 模式
7      UC_MODE_THUMB = 1 << 4,      // THUMB 模式 (包括 Thumb-
8      2)
9      UC_MODE_MCLASS = 1 << 5,      // ARM's Cortex-M 系列 (暂
10     不支持)
11     UC_MODE_V8 = 1 << 6,      // ARMv8 A32 encodings
12     for ARM (暂不支持)
13
14     // arm (32bit) cpu 类型
15     UC_MODE_ARM926 = 1 << 7,      // ARM926 CPU 类型
16     UC_MODE_ARM946 = 1 << 8,      // ARM946 CPU 类型
17     UC_MODE_ARM1176 = 1 << 9,      // ARM1176 CPU 类型
18
19     // mips
20     UC_MODE_MICRO = 1 << 4,      // MicroMips 模式 (暂不支
21     持)
22     UC_MODE_MIPS3 = 1 << 5,      // Mips III ISA (暂不支持)
23     UC_MODE_MIPS32R6 = 1 << 6,      // Mips32r6 ISA (暂不支持)
24     UC_MODE_MIPS32 = 1 << 2,      // Mips32 ISA
25     UC_MODE_MIPS64 = 1 << 3,      // Mips64 ISA
26
27     // x86 / x64
28     UC_MODE_16 = 1 << 1,      // 16-bit 模式
29     UC_MODE_32 = 1 << 2,      // 32-bit 模式
30     UC_MODE_64 = 1 << 3,      // 64-bit 模式
31
32     // ppc
```

```

29     UC_MODE_PPC32 = 1 << 2,          // 32-bit 模式 (暂不支持)
30     UC_MODE_PPC64 = 1 << 3,          // 64-bit 模式 (暂不支持)
31     UC_MODE_QPX = 1 << 4,            // Quad Processing
    extensions 模式 (暂不支持)
32
33     // sparc
34     UC_MODE_SPARC32 = 1 << 2,        // 32-bit 模式
35     UC_MODE_SPARC64 = 1 << 3,        // 64-bit 模式
36     UC_MODE_V9 = 1 << 4,             // SparcV9 模式 (暂不支持)
37
38     // m68k
39 } uc_mode;

```

### H3 uc\_err

错误类型，是uc\_errno()的返回值

```

1  typedef enum uc_err {
2      UC_ERR_OK = 0,          // 无错误
3      UC_ERR_NOMEM,          // 内存不足: uc_open(), uc_emulate()
4      UC_ERR_ARCH,           // 不支持的架构: uc_open()
5      UC_ERR_HANDLE,         // 不可用句柄
6      UC_ERR_MODE,           // 不可用/不支持架构: uc_open()
7      UC_ERR_VERSION,        // 不支持版本 (中间件)
8      UC_ERR_READ_UNMAPPED,  // 由于在未映射的内存上读取而退出模
    拟: uc_emu_start()
9      UC_ERR_WRITE_UNMAPPED, // 由于在未映射的内存上写入而退出模
    拟: uc_emu_start()
10     UC_ERR_FETCH_UNMAPPED, // 由于在未映射的内存中获取数据而退
    出模拟: uc_emu_start()
11     UC_ERR_HOOK,           // 无效的hook类型: uc_hook_add()
12     UC_ERR_INSN_INVALID,   // 由于指令无效而退出模拟:
    uc_emu_start()
13     UC_ERR_MAP,            // 无效的内存映射: uc_mem_map()
14     UC_ERR_WRITE_PROT,     // 由于UC_MEM_WRITE_PROT冲突而停止模
    拟: uc_emu_start()
15     UC_ERR_READ_PROT,      // 由于UC_MEM_READ_PROT冲突而停止模拟:
    uc_emu_start()
16     UC_ERR_FETCH_PROT,     // 由于UC_MEM_FETCH_PROT冲突而停止模
    拟: uc_emu_start()
17     UC_ERR_ARG,            // 提供给uc_xxx函数的无效参数
18     UC_ERR_READ_UNALIGNED, // 未对齐读取
19     UC_ERR_WRITE_UNALIGNED, // 未对齐写入
20     UC_ERR_FETCH_UNALIGNED, // 未对齐的提取
21     UC_ERR_HOOK_EXIST,     // 此事件的钩子已经存在
22     UC_ERR_RESOURCE,        // 资源不足: uc_emu_start()
23     UC_ERR_EXCEPTION,       // 未处理的CPU异常
24     UC_ERR_TIMEOUT // 模拟超时
25 } uc_err;

```

### H3 uc\_mem\_type

UC\_HOOK\_MEM\_\*的所有内存访问类型

```
1  typedef enum uc_mem_type {
2      UC_MEM_READ = 16,    // 内存从..读取
3      UC_MEM_WRITE,        // 内存写入到..
4      UC_MEM_FETCH,        // 内存被获取
5      UC_MEM_READ_UNMAPPED, // 未映射内存从..读取
6      UC_MEM_WRITE_UNMAPPED, // 未映射内存写入到..
7      UC_MEM_FETCH_UNMAPPED, // 未映射内存被获取
8      UC_MEM_WRITE_PROT,    // 内存写保护，但是已映射
9      UC_MEM_READ_PROT,     // 内存读保护，但是已映射
10     UC_MEM_FETCH_PROT,     // 内存不可执行，但是已映射
11     UC_MEM_READ_AFTER,     // 内存从（成功访问的地址）读入
12 } uc_mem_type;
```

### H3 uc\_hook\_type

uc\_hook\_add()的所有hook类型参数

```
1  typedef enum uc_hook_type {
2      // Hook 所有中断/syscall 事件
3      UC_HOOK_INTR = 1 << 0,
4      // Hook 一条特定的指令 - 只支持非常小的指令子集
5      UC_HOOK_INSN = 1 << 1,
6      // Hook 一段代码
7      UC_HOOK_CODE = 1 << 2,
8      // Hook 基本块
9      UC_HOOK_BLOCK = 1 << 3,
10     // 用于在未映射的内存上读取内存的Hook
11     UC_HOOK_MEM_READ_UNMAPPED = 1 << 4,
12     // Hook 无效的内存写事件
13     UC_HOOK_MEM_WRITE_UNMAPPED = 1 << 5,
14     // Hook 执行事件的无效内存
15     UC_HOOK_MEM_FETCH_UNMAPPED = 1 << 6,
16     // Hook 读保护的内存
17     UC_HOOK_MEM_READ_PROT = 1 << 7,
18     // Hook 写保护的内存
19     UC_HOOK_MEM_WRITE_PROT = 1 << 8,
20     // Hook 不可执行内存上的内存
21     UC_HOOK_MEM_FETCH_PROT = 1 << 9,
22     // Hook 内存读取事件
23     UC_HOOK_MEM_READ = 1 << 10,
24     // Hook 内存写入事件
25     UC_HOOK_MEM_WRITE = 1 << 11,
26     // Hook 内存获取执行事件
27     UC_HOOK_MEM_FETCH = 1 << 12,
28     // Hook 内存读取事件，只允许能成功访问的地址
29     // 成功读取后将触发回调
30     UC_HOOK_MEM_READ_AFTER = 1 << 13,
```



```

31      // Hook 无效指令异常
32      UC_HOOK_INSN_INVALID = 1 << 14,
33  } uc_hook_type;

```

### H3 宏定义Hook类型

```

1  // Hook 所有未映射内存访问的事件
2  #define UC_HOOK_MEM_UNMAPPED (UC_HOOK_MEM_READ_UNMAPPED +
   UC_HOOK_MEM_WRITE_UNMAPPED + UC_HOOK_MEM_FETCH_UNMAPPED)
3  // Hook 所有对受保护内存的非法访问事件
4  #define UC_HOOK_MEM_PROT (UC_HOOK_MEM_READ_PROT +
   UC_HOOK_MEM_WRITE_PROT + UC_HOOK_MEM_FETCH_PROT)
5  // Hook 所有非法读取存储器的事件
6  #define UC_HOOK_MEM_READ_INVALID (UC_HOOK_MEM_READ_PROT +
   UC_HOOK_MEM_READ_UNMAPPED)
7  // Hook 所有非法写入存储器的事件
8  #define UC_HOOK_MEM_WRITE_INVALID (UC_HOOK_MEM_WRITE_PROT +
   UC_HOOK_MEM_WRITE_UNMAPPED)
9  // Hook 所有非法获取内存的事件
10 #define UC_HOOK_MEM_FETCH_INVALID (UC_HOOK_MEM_FETCH_PROT +
   UC_HOOK_MEM_FETCH_UNMAPPED)
11 // Hook 所有非法的内存访问事件
12 #define UC_HOOK_MEM_INVALID (UC_HOOK_MEM_UNMAPPED +
   UC_HOOK_MEM_PROT)
13 // Hook 所有有效内存访问的事件
14 // 注意：UC_HOOK_MEM_READ 在 UC_HOOK_MEM_READ_PROT 和
   UC_HOOK_MEM_READ_UNMAPPED 之前触发，
15 // 因此这个Hook可能会触发一些无效的读取。
16 #define UC_HOOK_MEM_VALID (UC_HOOK_MEM_READ +
   UC_HOOK_MEM_WRITE + UC_HOOK_MEM_FETCH)

```

### H3 uc\_mem\_region

由uc\_mem\_map()和uc\_mem\_map\_ptr()映射内存区域  
使用uc\_mem\_regions()检索该内存区域的列表

```

1  typedef struct uc_mem_region {
2      uint64_t begin; // 区域起始地址 (包括)
3      uint64_t end;   // 区域结束地址 (包括)
4      uint32_t perms; // 区域的内存权限
5  } uc_mem_region;

```

### H3 uc\_query\_type

uc\_query()的所有查询类型参数

```

1  typedef enum uc_query_type {
2      // 动态查询当前硬件模式
3      UC_QUERY_MODE = 1,
4      UC_QUERY_PAGE_SIZE,
5      UC_QUERY_ARCH,
6  } uc_query_type;

```

### H3 uc\_context

与uc\_context\_\*()一起使用，管理CPU上下文的不透明存储

```

1  struct uc_context;
2  typedef struct uc_context uc_context;

```

### H3 uc\_prot

新映射区域的权限

```

1  typedef enum uc_prot {
2      UC_PROT_NONE = 0,    //无
3      UC_PROT_READ = 1,   //读取
4      UC_PROT_WRITE = 2,  //写入
5      UC_PROT_EXEC = 4,   //可执行
6      UC_PROT_ALL = 7,    //所有权限
7  } uc_prot;

```

## H2 0x2 API分析

### H3 uc\_version

```

1  unsigned int uc_version(unsigned int *major, unsigned int
    *minor);

```

用于返回Unicorn API主次版本信息

```

1  @major: API主版本号
2  @minor: API次版本号
3  @return 16进制数，计算方式 (major << 8 | minor)
4
5  提示：该返回值可以和宏UC_MAKE_VERSION比较

```

源码实现

```

1 unsigned int uc_version(unsigned int *major, unsigned int
  *minor)
2 {
3     if (major != NULL && minor != NULL) {
4         *major = UC_API_MAJOR; //宏
5         *minor = UC_API_MINOR; //宏
6     }
7
8     return (UC_API_MAJOR << 8) + UC_API_MINOR; //(major
  << 8 | minor)
9 }

```

编译后不可更改，不接受自定义版本

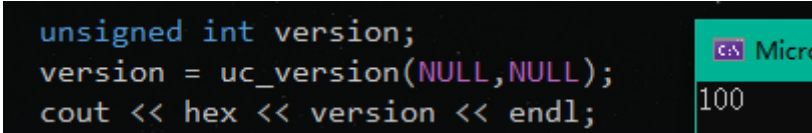
使用示例：

```

1 #include <iostream>
2 #include "unicorn/unicorn.h"
3 using namespace std;
4
5 int main()
6 {
7     unsigned int version;
8     version = uc_version(NULL, NULL);
9     cout << hex << version << endl;
10    return 0;
11 }

```

输出：



```

unsigned int version;
version = uc_version(NULL, NULL);
cout << hex << version << endl;

```

得到版本号1.0.0

### H3 uc\_arch\_supported

```

1 bool uc_arch_supported(uc_arch arch);

```

确定Unicorn是否支持当前架构

```

1 @arch: 架构类型 (UC_ARCH_*)
2 @return 如果支持返回True

```

源码实现

```

1 bool uc_arch_supported(uc_arch arch)
2 {
3     switch (arch) {
4 #ifdef UNICORN_HAS_ARM
5         case UC_ARCH_ARM: return true;

```

```

6  #endif
7  #ifdef UNICORN_HAS_ARM64
8      case UC_ARCH_ARM64: return true;
9  #endif
10 #ifdef UNICORN_HAS_M68K
11     case UC_ARCH_M68K: return true;
12 #endif
13 #ifdef UNICORN_HAS_MIPS
14     case UC_ARCH_MIPS: return true;
15 #endif
16 #ifdef UNICORN_HAS_PPC
17     case UC_ARCH_PPC: return true;
18 #endif
19 #ifdef UNICORN_HAS_SPARC
20     case UC_ARCH_SPARC: return true;
21 #endif
22 #ifdef UNICORN_HAS_X86
23     case UC_ARCH_X86: return true;
24 #endif
25     /* 无效或禁用架构 */
26     default: return false;
27 }
28 }

```

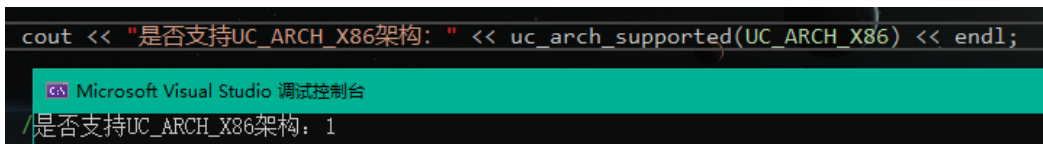
使用示例:

```

1  #include <iostream>
2  #include "unicorn/unicorn.h"
3  using namespace std;
4
5  int main()
6  {
7      cout << "是否支持UC_ARCH_X86架构: " <<
7      uc_arch_supported(UC_ARCH_X86) << endl;
8      return 0;
9  }

```

输出:



```

cout << "是否支持UC_ARCH_X86架构: " << uc_arch_supported(UC_ARCH_X86) << endl;
是否支持UC_ARCH_X86架构: 1

```

### H3 uc\_open

```

1  uc_err uc_open(uc_arch arch, uc_mode mode, uc_engine **uc);

```

创建新的Unicorn实例

- 1 @arch: 架构类型 (UC\_ARCH\_\*)
- 2 @mode: 硬件模式. 由 UC\_MODE\_\* 组合
- 3 @uc: 指向 uc\_engine 的指针, 返回时更新
- 4
- 5 @return 成功则返回UC\_ERR\_OK , 否则返回 uc\_err 枚举的其他错误类型

#### 源码实现

```
1  uc_err uc_open(uc_arch arch, uc_mode mode, uc_engine
   **result)
2  {
3      struct uc_struct *uc;
4
5      if (arch < UC_ARCH_MAX) {
6          uc = calloc(1, sizeof(*uc)); //申请内存
7          if (!uc) {
8              // 内存不足
9              return UC_ERR_NOMEM;
10         }
11
12         uc->errnum = UC_ERR_OK;
13         uc->arch = arch;
14         uc->mode = mode;
15
16         // 初始化
17         // uc->ram_list = { .blocks =
   QTAILQ_HEAD_INITIALIZER(ram_list.blocks) };
18         uc->ram_list.blocks.tqh_first = NULL;
19         uc->ram_list.blocks.tqh_last = &(uc-
   >ram_list.blocks.tqh_first);
20
21         uc->memory_listeners.tqh_first = NULL;
22         uc->memory_listeners.tqh_last = &(uc-
   >memory_listeners.tqh_first);
23
24         uc->address_spaces.tqh_first = NULL;
25         uc->address_spaces.tqh_last = &(uc-
   >address_spaces.tqh_first);
26
27         switch(arch) { // 根据架构进行预处理
28             default:
29                 break;
30 #ifdef UNICORN_HAS_M68K
31             case UC_ARCH_M68K:
32                 if ((mode & ~UC_MODE_M68K_MASK) ||
33                     !(mode & UC_MODE_BIG_ENDIAN)) {
34                     free(uc);
35                     return UC_ERR_MODE;
36                 }
37                 uc->init_arch = m68k_uc_init;
```

```

38         break;
39     #endif
40     #ifdef UNICORN_HAS_X86
41         case UC_ARCH_X86:
42             if ((mode & ~UC_MODE_X86_MASK) ||
43                 (mode & UC_MODE_BIG_ENDIAN) ||
44                 !(mode &
45 (UC_MODE_16|UC_MODE_32|UC_MODE_64))) {
46                 free(uc);
47                 return UC_ERR_MODE;
48             }
49             uc->init_arch = x86_uc_init;
50             break;
51     #endif
52     #ifdef UNICORN_HAS_ARM
53         case UC_ARCH_ARM:
54             if ((mode & ~UC_MODE_ARM_MASK)) {
55                 free(uc);
56                 return UC_ERR_MODE;
57             }
58             if (mode & UC_MODE_BIG_ENDIAN) {
59                 uc->init_arch = armb_uc_init;
60             } else {
61                 uc->init_arch = arm_uc_init;
62             }
63             if (mode & UC_MODE_THUMB)
64                 uc->thumb = 1;
65             break;
66     #endif
67     #ifdef UNICORN_HAS_ARM64
68         case UC_ARCH_ARM64:
69             if (mode & ~UC_MODE_ARM_MASK) {
70                 free(uc);
71                 return UC_ERR_MODE;
72             }
73             if (mode & UC_MODE_BIG_ENDIAN) {
74                 uc->init_arch = arm64eb_uc_init;
75             } else {
76                 uc->init_arch = arm64_uc_init;
77             }
78             break;
79     #endif
80
81     #if defined(UNICORN_HAS_MIPS) ||
82         defined(UNICORN_HAS_MIPSEL) || defined(UNICORN_HAS_MIPS64)
83         || defined(UNICORN_HAS_MIPS64EL)
84         case UC_ARCH_MIPS:
85             if ((mode & ~UC_MODE_MIPS_MASK) ||
86                 !(mode &
87 (UC_MODE_MIPS32|UC_MODE_MIPS64))) {
88                 free(uc);

```

```

86         return UC_ERR_MODE;
87     }
88     if (mode & UC_MODE_BIG_ENDIAN) {
89 #ifdef UNICORN_HAS_MIPS
90         if (mode & UC_MODE_MIPS32)
91             uc->init_arch = mips_uc_init;
92 #endif
93 #ifdef UNICORN_HAS_MIPS64
94         if (mode & UC_MODE_MIPS64)
95             uc->init_arch = mips64_uc_init;
96 #endif
97     } else { // 小端序
98 #ifdef UNICORN_HAS_MIPSEL
99         if (mode & UC_MODE_MIPS32)
100             uc->init_arch = mipsel_uc_init;
101 #endif
102 #ifdef UNICORN_HAS_MIPS64EL
103         if (mode & UC_MODE_MIPS64)
104             uc->init_arch = mips64el_uc_init;
105 #endif
106     }
107     break;
108 #endif
109
110 #ifdef UNICORN_HAS_SPARC
111     case UC_ARCH_SPARC:
112         if ((mode & ~UC_MODE_SPARC_MASK) ||
113             !(mode & UC_MODE_BIG_ENDIAN) ||
114             !(mode &
115 4 (UC_MODE_SPARC32|UC_MODE_SPARC64))) {
116             free(uc);
117             return UC_ERR_MODE;
118         }
119         if (mode & UC_MODE_SPARC64)
120             uc->init_arch = sparc64_uc_init;
121         else
122             uc->init_arch = sparc_uc_init;
123         break;
124 #endif
125     }
126
127     if (uc->init_arch == NULL) {
128         return UC_ERR_ARCH;
129     }
130
131     if (machine_initialize(uc))
132         return UC_ERR_RESOURCE;
133
134     *result = uc;
135
136     if (uc->reg_reset)
137         uc->reg_reset(uc);

```

```

16
13         return UC_ERR_OK;
18     } else {
19         return UC_ERR_ARCH;
20     }
21 }

```

**注意：**uc\_open会申请堆内存，使用完必须用uc\_close释放，否则会发生泄露

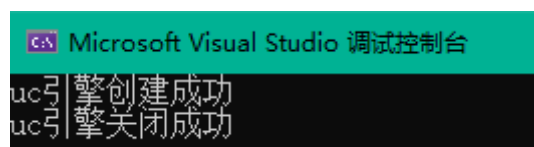
使用示例：

```

1  #include <iostream>
2  #include "unicorn/unicorn.h"
3  using namespace std;
4
5  int main()
6  {
7      uc_engine* uc;
8      uc_err err;
9
10     ///// 初始化 X86-32bit 模式模拟器
11     err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
12     if (err != UC_ERR_OK) {
13         printf("Failed on uc_open() with error returned:
14         %u\n", err);
15         return -1;
16     }
17
18     if (!err)
19         cout << "uc引擎创建成功" << endl;
20
21     ///// 关闭uc
22     err = uc_close(uc);
23     if (err != UC_ERR_OK) {
24         printf("Failed on uc_close() with error returned:
25         %u\n", err);
26         return -1;
27     }
28
29     if (!err)
30         cout << "uc引擎关闭成功" << endl;
31
32     return 0;
33 }

```

输出





```
1 uc_err uc_close(uc_engine *uc);
```

关闭一个uc实例，将释放内存。关闭后无法恢复。

```
1 @uc: 指向由 uc_open() 返回的指针
2
3 @return 成功则返回UC_ERR_OK，否则返回 uc_err 枚举的其他错误类型
```

### 源码实现

```
1 uc_err uc_close(uc_engine *uc)
2 {
3     int i;
4     struct list_item *cur;
5     struct hook *hook;
6
7     // 清理内部数据
8     if (uc->release)
9         uc->release(uc->tcg_ctx);
10    g_free(uc->tcg_ctx);
11
12    // 清理 CPU.
13    g_free(uc->cpu->tcg_as_listener);
14    g_free(uc->cpu->thread);
15
16    // 清理所有 objects.
17    OBJECT(uc->machine_state->accelerator)->ref = 1;
18    OBJECT(uc->machine_state)->ref = 1;
19    OBJECT(uc->owner)->ref = 1;
20    OBJECT(uc->root)->ref = 1;
21
22    object_unref(uc, OBJECT(uc->machine_state-
23    >accelerator));
24    object_unref(uc, OBJECT(uc->machine_state));
25    object_unref(uc, OBJECT(uc->cpu));
26    object_unref(uc, OBJECT(&uc->io_mem_notdirty));
27    object_unref(uc, OBJECT(&uc->io_mem_unassigned));
28    object_unref(uc, OBJECT(&uc->io_mem_rom));
29    object_unref(uc, OBJECT(uc->root));
30
31    // 释放内存
32    g_free(uc->system_memory);
33
34    // 释放相关线程
35    if (uc->qemu_thread_data)
36        g_free(uc->qemu_thread_data);
37
38    // 释放其他数据
39    free(uc->l1_map);
```

```

40     if (uc->bounce.buffer) {
41         free(uc->bounce.buffer);
42     }
43
44     g_hash_table_foreach(uc->type_table, free_table, uc);
45     g_hash_table_destroy(uc->type_table);
46
47     for (i = 0; i < DIRTY_MEMORY_NUM; i++) {
48         free(uc->ram_list.dirty_memory[i]);
49     }
50
51     // 释放hook和hook列表
52     for (i = 0; i < UC_HOOK_MAX; i++) {
53         cur = uc->hook[i].head;
54         // hook 可存在于多个列表，可通过计数获取释放的时间
55         while (cur) {
56             hook = (struct hook *)cur->data;
57             if (--hook->refs == 0) {
58                 free(hook);
59             }
60             cur = cur->next;
61         }
62         list_clear(&uc->hook[i]);
63     }
64
65     free(uc->mapped_blocks);
66
67     // 最后释放uc自身
68     memset(uc, 0, sizeof(*uc));
69     free(uc);
70
71     return UC_ERR_OK;
72 }

```

使用实例同uc\_open()

### H3 uc\_query

```

1  uc_err uc_query(uc_engine *uc, uc_query_type type, size_t
   *result);

```

查询引擎的内部状态

```

1  @uc: uc_open() 返回的句柄
2  @type: uc_query_type 中枚举的类型
3
4  @result: 保存被查询的内部状态的指针
5
6  @return: 成功则返回UC_ERR_OK , 否则返回 uc_err 枚举的其他错误
          类型

```

## 源码实现

```
1  uc_err uc_query(uc_engine *uc, uc_query_type type, size_t
    *result)
2  {
3      if (type == UC_QUERY_PAGE_SIZE) {
4          *result = uc->target_page_size;
5          return UC_ERR_OK;
6      }
7
8      if (type == UC_QUERY_ARCH) {
9          *result = uc->arch;
10         return UC_ERR_OK;
11     }
12
13     switch(uc->arch) {
14 #ifdef UNICORN_HAS_ARM
15         case UC_ARCH_ARM:
16             return uc->query(uc, type, result);
17 #endif
18         default:
19             return UC_ERR_ARG;
20     }
21
22     return UC_ERR_OK;
23 }
```

## 使用示例:

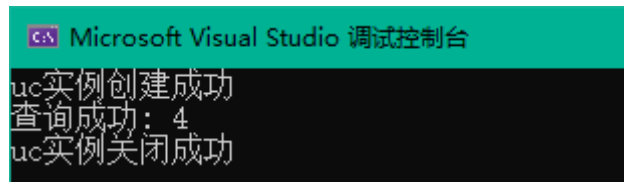
```
1  #include <iostream>
2  #include "unicorn/unicorn.h"
3  using namespace std;
4  int main()
5  {
6      uc_engine* uc;
7      uc_err err;
8
9      ///// Initialize emulator in X86-32bit mode
10     err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
11     if (err != UC_ERR_OK) {
12         printf("Failed on uc_open() with error returned:
13         %u\n", err);
14         return -1;
15     }
16     if (!err)
17         cout << "uc实例创建成功" << endl;
18
19     size_t result[] = {0};
20     err = uc_query(uc, UC_QUERY_ARCH, result); // 查询架构
21     if (!err)
22         cout << "查询成功: " << *result << endl;
```

```

23     err = uc_close(uc);
24     if (err != UC_ERR_OK) {
25         printf("Failed on uc_close() with error returned:
26         %u\n", err);
27         return -1;
28     }
29     if (!err)
30         cout << "uc实例关闭成功" << endl;
31     return 0;
32 }

```

输出



架构查询结果为4，对应的正是UC\_ARCH\_X86

### H3 uc\_errno

```

1 uc_err uc_errno(uc_engine *uc);

```

当某个API函数失败时，报告最后的错误号，一旦被访问，uc\_errno可能不会保留原来的值。

```

1 @uc: uc_open() 返回的句柄
2
3 @return: 成功则返回UC_ERR_OK，否则返回 uc_err 枚举的其他错误类型

```

源码实现

```

1 uc_err uc_errno(uc_engine *uc)
2 {
3     return uc->errnum;
4 }

```

使用示例:

```

1 #include <iostream>
2 #include "unicorn/unicorn.h"
3 using namespace std;
4
5 int main()
6 {
7     uc_engine* uc;
8     uc_err err;
9
10    err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);

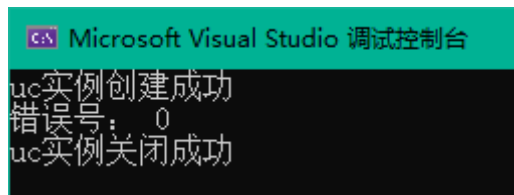
```

```

11     if (err != UC_ERR_OK) {
12         printf("Failed on uc_open() with error returned:
13         %u\n", err);
14         return -1;
15     }
16     if (!err)
17         cout << "uc实例创建成功" << endl;
18
19     err = uc_errno(uc);
20     cout << "错误号: " << err << endl;
21
22     err = uc_close(uc);
23     if (err != UC_ERR_OK) {
24         printf("Failed on uc_close() with error returned:
25         %u\n", err);
26         return -1;
27     }
28     if (!err)
29         cout << "uc实例关闭成功" << endl;
30
31     return 0;
32 }

```

输出



无错误，输出错误号为0

### H3 uc\_strerror

```
1 const char *uc_strerror(uc_err code);
```

返回给定错误号的解释

```

1 @code: 错误号
2
3 @return: 指向给定错误号的解释的字符串指针

```

源码实现

```

1 const char *uc_strerror(uc_err code)
2 {
3     switch(code) {
4         default:
5             return "Unknown error code";
6         case UC_ERR_OK:
7             return "OK (UC_ERR_OK)";

```

```
8         case UC_ERR_NOMEM:
9             return "No memory available or memory not
present (UC_ERR_NOMEM)";
10        case UC_ERR_ARCH:
11            return "Invalid/unsupported architecture
(UC_ERR_ARCH)";
12        case UC_ERR_HANDLE:
13            return "Invalid handle (UC_ERR_HANDLE)";
14        case UC_ERR_MODE:
15            return "Invalid mode (UC_ERR_MODE)";
16        case UC_ERR_VERSION:
17            return "Different API version between core &
binding (UC_ERR_VERSION)";
18        case UC_ERR_READ_UNMAPPED:
19            return "Invalid memory read
(UC_ERR_READ_UNMAPPED)";
20        case UC_ERR_WRITE_UNMAPPED:
21            return "Invalid memory write
(UC_ERR_WRITE_UNMAPPED)";
22        case UC_ERR_FETCH_UNMAPPED:
23            return "Invalid memory fetch
(UC_ERR_FETCH_UNMAPPED)";
24        case UC_ERR_HOOK:
25            return "Invalid hook type (UC_ERR_HOOK)";
26        case UC_ERR_INSN_INVALID:
27            return "Invalid instruction
(UC_ERR_INSN_INVALID)";
28        case UC_ERR_MAP:
29            return "Invalid memory mapping (UC_ERR_MAP)";
30        case UC_ERR_WRITE_PROT:
31            return "Write to write-protected memory
(UC_ERR_WRITE_PROT)";
32        case UC_ERR_READ_PROT:
33            return "Read from non-readable memory
(UC_ERR_READ_PROT)";
34        case UC_ERR_FETCH_PROT:
35            return "Fetch from non-executable memory
(UC_ERR_FETCH_PROT)";
36        case UC_ERR_ARG:
37            return "Invalid argument (UC_ERR_ARG)";
38        case UC_ERR_READ_UNALIGNED:
39            return "Read from unaligned memory
(UC_ERR_READ_UNALIGNED)";
40        case UC_ERR_WRITE_UNALIGNED:
41            return "Write to unaligned memory
(UC_ERR_WRITE_UNALIGNED)";
42        case UC_ERR_FETCH_UNALIGNED:
43            return "Fetch from unaligned memory
(UC_ERR_FETCH_UNALIGNED)";
44        case UC_ERR_RESOURCE:
45            return "Insufficient resource
(UC_ERR_RESOURCE)";
```

```

46         case UC_ERR_EXCEPTION:
47             return "Unhandled CPU exception
         (UC_ERR_EXCEPTION)";
48         case UC_ERR_TIMEOUT:
49             return "Emulation timed out (UC_ERR_TIMEOUT)";
50     }
51 }

```

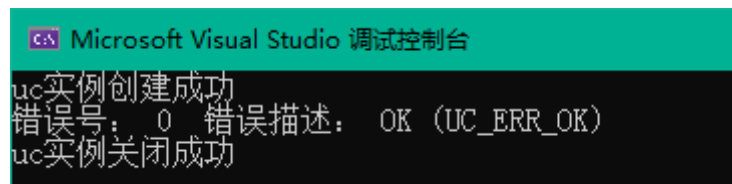
使用示例:

```

1  #include <iostream>
2  #include "unicorn/unicorn.h"
3  using namespace std;
4
5  int main()
6  {
7      uc_engine* uc;
8      uc_err err;
9
10     err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
11     if (err != UC_ERR_OK) {
12         printf("Failed on uc_open() with error returned:
        %u\n", err);
13         return -1;
14     }
15     if (!err)
16         cout << "uc实例创建成功" << endl;
17
18     err = uc_errno(uc);
19     cout << "错误号: " << err << "  错误描述: " <<
        uc_strerror(err) << endl;
20
21     err = uc_close(uc);
22     if (err != UC_ERR_OK) {
23         printf("Failed on uc_close() with error returned:
        %u\n", err);
24         return -1;
25     }
26     if (!err)
27         cout << "uc实例关闭成功" << endl;
28
29     return 0;
30 }

```

输出



Microsoft Visual Studio 调试控制台

```

uc实例创建成功
错误号: 0  错误描述: OK (UC_ERR_OK)
uc实例关闭成功

```

### H3 uc\_reg\_write

```
1 uc_err uc_reg_write(uc_engine *uc, int regid, const void
    *value);
```

将值写入寄存器

```
1 @uc: uc_open()返回的句柄
2 @regid: 将被修改的寄存器ID
3 @value: 指向寄存器将被修改成的值的指针
4
5 @return 成功则返回UC_ERR_OK , 否则返回 uc_err 枚举的其他错误类型
```

源码实现

```
1 uc_err uc_reg_write(uc_engine *uc, int regid, const void
    *value)
2 {
3     return uc_reg_write_batch(uc, &regid, (void *const
        *)&value, 1);
4 }
5
6 uc_err uc_reg_write_batch(uc_engine *uc, int *ids, void
    *const *vals, int count)
7 {
8     int ret = UC_ERR_OK;
9     if (uc->reg_write)
10         ret = uc->reg_write(uc, (unsigned int *)ids, vals,
            count); //结构体中写入
11     else
12         return UC_ERR_EXCEPTION;
13
14     return ret;
15 }
```

使用示例:

```
1 #include <iostream>
2 #include "unicorn/unicorn.h"
3 using namespace std;
4
5 int main()
6 {
7     uc_engine* uc;
8     uc_err err;
9
10     err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
11     if (err != UC_ERR_OK) {
12         printf("Failed on uc_open() with error returned:
            %u\n", err);
```

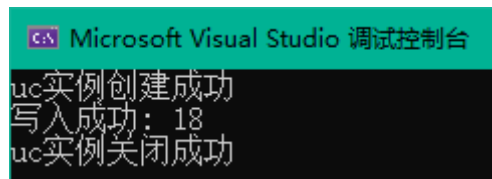


```

13         return -1;
14     }
15     if (!err)
16         cout << "uc实例创建成功" << endl;
17
18     int r_eax = 0x12;
19     err = uc_reg_write(uc, UC_X86_REG_ECX, &r_eax);
20     if (!err)
21         cout << "写入成功: " << r_eax << endl;
22
23     err = uc_close(uc);
24     if (err != UC_ERR_OK) {
25         printf("Failed on uc_close() with error returned:
26 %u\n", err);
27         return -1;
28     }
29     if (!err)
30         cout << "uc实例关闭成功" << endl;
31
32     return 0;
33 }

```

输出



### H3 uc\_reg\_read

```
1 uc_err uc_reg_read(uc_engine *uc, int regid, void *value);
```

读取寄存器的值

```

1 @uc: uc_open()返回的句柄
2 @regid: 将被读取的寄存器ID
3 @value: 指向保存寄存器值的指针
4
5 @return 成功则返回UC_ERR_OK , 否则返回 uc_err 枚举的其他错误类型

```

源码实现

```

1 uc_err uc_reg_read(uc_engine *uc, int regid, void *value)
2 {
3     return uc_reg_read_batch(uc, &regid, &value, 1);
4 }
5
6 uc_err uc_reg_read_batch(uc_engine *uc, int *ids, void
7 **vals, int count)

```

```

7  {
8      if (uc->reg_read)
9          uc->reg_read(uc, (unsigned int *)ids, vals, count);
10     else
11         return -1;
12
13     return UC_ERR_OK;
14 }

```

使用示例:

```

1  #include <iostream>
2  #include "unicorn/unicorn.h"
3  using namespace std;
4
5  int main()
6  {
7      uc_engine* uc;
8      uc_err err;
9
10     err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
11     if (err != UC_ERR_OK) {
12         printf("Failed on uc_open() with error returned:
13 %u\n", err);
14         return -1;
15     }
16     if (!err)
17         cout << "uc实例创建成功" << endl;
18
19     int r_eax = 0x12;
20     err = uc_reg_write(uc, UC_X86_REG_ECX, &r_eax);
21     if (!err)
22         cout << "写入成功: " << r_eax << endl;
23
24     int recv_eax;
25     err = uc_reg_read(uc, UC_X86_REG_ECX, &recv_eax);
26     if (!err)
27         cout << "读取成功: " << recv_eax << endl;
28
29     err = uc_close(uc);
30     if (err != UC_ERR_OK) {
31         printf("Failed on uc_close() with error returned:
32 %u\n", err);
33         return -1;
34     }
35     if (!err)
36         cout << "uc实例关闭成功" << endl;
37
38     return 0;
39 }

```

输出



### H3 uc\_reg\_write\_batch

```
1 uc_err uc_reg_write_batch(uc_engine *uc, int *regs, void
  *const *vals, int count);
```

同时将多个值写入多个寄存器

```
1 @uc: uc_open()返回的句柄
2 @regid: 存储将被写入的多个寄存器ID的数组
3 @value: 指向保存多个值的数组的指针
4 @count: *regs 和 *vals 数组的长度
5
6 @return 成功则返回UC_ERR_OK, 否则返回 uc_err 枚举的其他错误类型
```

源码实现

```
1 uc_err uc_reg_write_batch(uc_engine *uc, int *ids, void
  *const *vals, int count)
2 {
3     int ret = UC_ERR_OK;
4     if (uc->reg_write)
5         ret = uc->reg_write(uc, (unsigned int *)ids, vals,
        count);
6     else
7         return UC_ERR_EXCEPTION;
8
9     return ret;
10 }
```

使用示例:

```
1 #include <iostream>
2 #include <string>
3 #include "unicorn/unicorn.h"
4 using namespace std;
5
6 int syscall_abi[] = {
7     UC_X86_REG_RAX, UC_X86_REG_RDI, UC_X86_REG_RSI,
8     UC_X86_REG_RDX,
9     UC_X86_REG_R10, UC_X86_REG_R8, UC_X86_REG_R9
10 };
11 uint64_t vals[7] = { 200, 10, 11, 12, 13, 14, 15 };
12
```

```

13 void* ptrs[7];
14
15 int main()
16 {
17     int i;
18     uc_err err;
19     uc_engine* uc;
20
21     // set up register pointers
22     for (i = 0; i < 7; i++) {
23         ptrs[i] = &vals[i];
24     }
25
26     if ((err = uc_open(UC_ARCH_X86, UC_MODE_64, &uc))) {
27         uc_perror("uc_open", err);
28         return 1;
29     }
30
31     // reg_write_batch
32     printf("reg_write_batch({200, 10, 11, 12, 13, 14,
33 15})\n");
34     if ((err = uc_reg_write_batch(uc, syscall_abi, ptrs,
35 7))) {
36         uc_perror("uc_reg_write_batch", err);
37         return 1;
38     }
39
40     // reg_read_batch
41     memset(vals, 0, sizeof(vals));
42     if ((err = uc_reg_read_batch(uc, syscall_abi, ptrs,
43 7))) {
44         uc_perror("uc_reg_read_batch", err);
45         return 1;
46     }
47
48     printf("reg_read_batch = {");
49
50     for (i = 0; i < 7; i++) {
51         if (i != 0) printf(", ");
52         printf("%" PRIu64, vals[i]);
53     }
54
55     printf("}\n");
56
57     uint64_t var[7] = { 0 };
58     for (int i = 0; i < 7; i++)
59     {
60         cout << syscall_abi[i] << " ";
61         printf("%" PRIu64, vals[i]);
62         cout << endl;
63     }
64 }

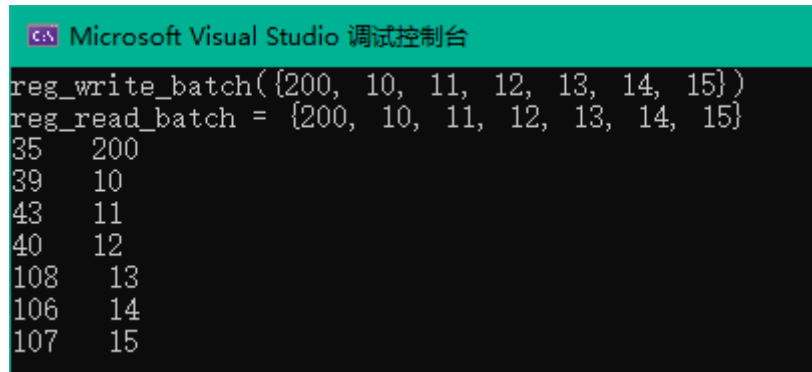
```

```

62     return 0;
63 }

```

输出



```

Microsoft Visual Studio 调试控制台
reg_write_batch({200, 10, 11, 12, 13, 14, 15})
reg_read_batch = {200, 10, 11, 12, 13, 14, 15}
35    200
39    10
43    11
40    12
108   13
106   14
107   15

```

### H3 uc\_reg\_read\_batch

```

1  uc_err uc_reg_read_batch(uc_engine *uc, int *regs, void
    **vals, int count);

```

同时读取多个寄存器的值。

```

1  @uc: uc_open()返回的句柄
2  @regid: 存储将被读取的多个寄存器ID的数组
3  @value: 指向保存多个值的数组的指针
4  @count: *regs 和 *vals 数组的长度
5
6  @return 成功则返回UC_ERR_OK，否则返回 uc_err 枚举的其他错误类型

```

源码实现

```

1  uc_err uc_reg_read_batch(uc_engine *uc, int *ids, void
    **vals, int count)
2  {
3      if (uc->reg_read)
4          uc->reg_read(uc, (unsigned int *)ids, vals, count);
5      else
6          return -1;
7
8      return UC_ERR_OK;
9  }

```

使用示例同uc\_reg\_write\_batch()。

### H3 uc\_mem\_write

```

1  uc_err uc_mem_write(uc_engine *uc, uint64_t address, const
    void *bytes, size_t size);

```

在内存中写入一段字节码。

```
1  @uc: uc_open() 返回的句柄
2  @address: 写入字节的起始地址
3  @bytes: 指向一个包含要写入内存的数据的指针
4  @size: 要写入的内存大小。
5
6  注意: @bytes 必须足够大以包含 @size 字节。
7
8  @return 成功则返回UC_ERR_OK , 否则返回 uc_err 枚举的其他错误类型
```

源码实现

```
1  uc_err uc_mem_write(uc_engine *uc, uint64_t address, const
    void *_bytes, size_t size)
2  {
3      size_t count = 0, len;
4      const uint8_t *bytes = _bytes;
5
6      if (uc->mem_redirect) {
7          address = uc->mem_redirect(address);
8      }
9
10     if (!check_mem_area(uc, address, size))
11         return UC_ERR_WRITE_UNMAPPED;
12
13     // 内存区域可以重叠相邻的内存块
14     while(count < size) {
15         MemoryRegion *mr = memory_mapping(uc, address);
16         if (mr) {
17             uint32_t operms = mr->perms;
18             if (!(operms & UC_PROT_WRITE)) // 没有写保护
19                 // 标记为可写
20                 uc->readonly_mem(mr, false);
21
22             len = (size_t)MIN(size - count, mr->end -
                address);
23             if (uc->write_mem(&uc->as, address, bytes, len)
                == false)
24                 break;
25
26             if (!(operms & UC_PROT_WRITE)) // 没有写保护
27                 // 设置写保护
28                 uc->readonly_mem(mr, true);
29
30             count += len;
31             address += len;
32             bytes += len;
33         } else // 此地址尚未被映射
34             break;
35     }
```

```

36
37     if (count == size)
38         return UC_ERR_OK;
39     else
40         return UC_ERR_WRITE_UNMAPPED;
41 }

```

使用示例:

```

1  #include <iostream>
2  #include <string>
3  #include "unicorn/unicorn.h"
4  using namespace std;
5
6  #define X86_CODE32 "\x41\x4a" // INC ecx; DEC edx
7  #define ADDRESS 0x1000
8
9  int main()
10 {
11     uc_engine* uc;
12     uc_err err;
13
14     err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
15     if (err != UC_ERR_OK) {
16         printf("Failed on uc_open() with error returned:
17 %u\n", err);
18         return -1;
19     }
20
21     uc_mem_map(uc, ADDRESS, 2 * 1024 * 1024, UC_PROT_ALL);
22
23     if (uc_mem_write(uc, ADDRESS, X86_CODE32,
24 sizeof(X86_CODE32) - 1)) {
25         printf("Failed to write emulation code to memory,
26 quit!\n");
27         return -1;
28     }
29
30     uint32_t code;
31
32     if(uc_mem_read(uc, ADDRESS, &code, sizeof(code))) {
33         printf("Failed to read emulation code to memory,
34 quit!\n");
35         return -1;
36     }
37
38     cout << hex << code << endl;
39
40     err = uc_close(uc);
41     if (err != UC_ERR_OK) {
42         printf("Failed on uc_close() with error returned:
43 %u\n", err);

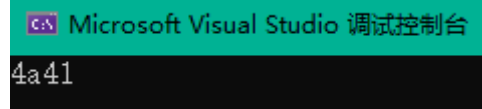
```

```

39         return -1;
40     }
41     return 0;
42 }

```

输出



Microsoft Visual Studio 调试控制台  
4a41

### H3 uc\_mem\_read

```

1  uc_err uc_mem_read(uc_engine *uc, uint64_t address, void
    *bytes, size_t size);

```

从内存中读取字节。

```

1  @uc: uc_open() 返回的句柄
2  @address: 读取字节的起始地址
3  @bytes: 指向一个包含要读取内存的数据的指针
4  @size: 要读取的内存大小。
5
6  注意: @bytes 必须足够大以包含 @size 字节。
7
8  @return 成功则返回UC_ERR_OK , 否则返回 uc_err 枚举的其他错误类型

```

源码实现

```

1  uc_err uc_mem_read(uc_engine *uc, uint64_t address, void
    *_bytes, size_t size)
2  {
3      size_t count = 0, len;
4      uint8_t *bytes = _bytes;
5
6      if (uc->mem_redirect) {
7          address = uc->mem_redirect(address);
8      }
9
10     if (!check_mem_area(uc, address, size))
11         return UC_ERR_READ_UNMAPPED;
12
13     // 内存区域可以重叠相邻的内存块
14     while(count < size) {
15         MemoryRegion *mr = memory_mapping(uc, address);
16         if (mr) {
17             len = (size_t)MIN(size - count, mr->end -
address);
18             if (uc->read_mem(&uc->as, address, bytes, len)
== false)

```



```

19         break;
20         count += len;
21         address += len;
22         bytes += len;
23     } else // 此地址尚未被映射
24         break;
25 }
26
27 if (count == size)
28     return UC_ERR_OK;
29 else
30     return UC_ERR_READ_UNMAPPED;
31 }

```

使用示例同uc\_mem\_write()

### H3 uc\_emu\_start

```

1 uc_err uc_emu_start(uc_engine *uc, uint64_t begin, uint64_t
  until, uint64_t timeout, size_t count);

```

在指定的时间内模拟机器码。

```

1 @uc: uc_open() 返回的句柄
2 @begin: 开始模拟的地址
3 @until: 模拟停止的地址（当到达该地址时）
4 @timeout: 模拟代码的持续时间(以微秒计)。当这个值为0时，将在无限时
   间内模拟代码，直到代码完成。
5 @count: 要模拟的指令数。当这个值为0时，将模拟所有可用的代码，直到
   代码完成
6
7 @return 成功则返回UC_ERR_OK，否则返回 uc_err 枚举的其他错误类
   型

```

源码实现

```

1 uc_err uc_emu_start(uc_engine* uc, uint64_t begin, uint64_t
  until, uint64_t timeout, size_t count)
2 {
3     // 重制计数器
4     uc->emu_counter = 0;
5     uc->invalid_error = UC_ERR_OK;
6     uc->block_full = false;
7     uc->emulation_done = false;
8     uc->timed_out = false;
9
10    switch(uc->arch) {
11        default:
12            break;
13    #ifdef UNICORN_HAS_M68K
14        case UC_ARCH_M68K:

```

```

15         uc_reg_write(uc, UC_M68K_REG_PC, &begin);
16         break;
17     #endif
18     #ifdef UNICORN_HAS_X86
19         case UC_ARCH_X86:
20             switch(uc->mode) {
21                 default:
22                     break;
23                 case UC_MODE_16: {
24                     uint64_t ip;
25                     uint16_t cs;
26
27                     uc_reg_read(uc, UC_X86_REG_CS, &cs);
28                     // 抵消后面增加的 IP 和 CS
29                     ip = begin - cs*16;
30                     uc_reg_write(uc, UC_X86_REG_IP, &ip);
31                     break;
32                 }
33                 case UC_MODE_32:
34                     uc_reg_write(uc, UC_X86_REG_EIP,
35 &begin);
36                     break;
37                 case UC_MODE_64:
38                     uc_reg_write(uc, UC_X86_REG_RIP,
39 &begin);
40                     break;
41             }
42             break;
43     #endif
44     #ifdef UNICORN_HAS_ARM
45         case UC_ARCH_ARM:
46             uc_reg_write(uc, UC_ARM_REG_R15, &begin);
47             break;
48     #endif
49     #ifdef UNICORN_HAS_ARM64
50         case UC_ARCH_ARM64:
51             uc_reg_write(uc, UC_ARM64_REG_PC, &begin);
52             break;
53     #endif
54     #ifdef UNICORN_HAS_MIPS
55         case UC_ARCH_MIPS:
56             // TODO: MIPS32/MIPS64/BIGENDIAN etc
57             uc_reg_write(uc, UC_MIPS_REG_PC, &begin);
58             break;
59     #endif
60     #ifdef UNICORN_HAS_SPARC
61         case UC_ARCH_SPARC:
62             // TODO: Sparc/Sparc64
63             uc_reg_write(uc, UC_SPARC_REG_PC, &begin);
64             break;
65     #endif
66 }

```

```

65
66     uc->stop_request = false;
67
68     uc->emu_count = count;
69     // 如果不需要计数，则移除计数挂钩hook
70     if (count <= 0 && uc->count_hook != 0) {
71         uc_hook_del(uc, uc->count_hook);
72         uc->count_hook = 0;
73     }
74     // 设置计数hook记录指令数
75     if (count > 0 && uc->count_hook == 0) {
76         uc_err err;
77         // 对计数指令的回调必须在所有其他操作之前运行，因此必须在
hook列表的开头插入hook，而不是附加hook
78         uc->hook_insert = 1;
79         err = uc_hook_add(uc, &uc->count_hook,
UC_HOOK_CODE, hook_count_cb, NULL, 1, 0);
80         // 恢复到 uc_hook_add()
81         uc->hook_insert = 0;
82         if (err != UC_ERR_OK) {
83             return err;
84         }
85     }
86
87     uc->addr_end = until;
88
89     if (timeout)
90         enable_emu_timer(uc, timeout * 1000);    //
microseconds -> nanoseconds
91
92     if (uc->vm_start(uc)) {
93         return UC_ERR_RESOURCE;
94     }
95
96     // 模拟完成
97     uc->emulation_done = true;
98
99     if (timeout) {
100         // 等待超时
100         qemu_thread_join(&uc->timer);
100     }
100
100     if(uc->timed_out)
100         return UC_ERR_TIMEOUT;
100
100     return uc->invalid_error;
100 }

```

使用示例:

```

1  #include <iostream>
2  #include <string>

```

```

3  #include "unicorn/unicorn.h"
4  using namespace std;
5
6  #define X86_CODE32 "\x33\xC0" // xor  eax, eax
7  #define ADDRESS 0x1000
8
9  int main()
10 {
11     uc_engine* uc;
12     uc_err err;
13
14     int r_eax = 0x111;
15
16     err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
17     if (err != UC_ERR_OK) {
18         printf("Failed on uc_open() with error returned:
19 %u\n", err);
20         return -1;
21     }
22
23     uc_mem_map(uc, ADDRESS, 2 * 1024 * 1024, UC_PROT_ALL);
24
25     if (uc_mem_write(uc, ADDRESS, X86_CODE32,
26 sizeof(X86_CODE32) - 1)) {
27         printf("Failed to write emulation code to memory,
28 quit!\n");
29         return -1;
30     }
31
32     uc_reg_write(uc, UC_X86_REG_EAX, &r_eax);
33     printf(">>> before EAX = 0x%x\n", r_eax);
34
35     err = uc_emu_start(uc, ADDRESS, ADDRESS +
36 sizeof(X86_CODE32) - 1, 0, 0);
37     if (err) {
38         printf("Failed on uc_emu_start() with error
39 returned %u: %s\n",
40 err, uc_strerror(err));
41     }
42
43     uc_reg_read(uc, UC_X86_REG_EAX, &r_eax);
44     printf(">>> after EAX = 0x%x\n", r_eax);
45
46     err = uc_close(uc);
47     if (err != UC_ERR_OK) {
48         printf("Failed on uc_close() with error returned:
49 %u\n", err);
50         return -1;
51     }
52
53     return 0;
54 }

```

输出

```
Microsoft Visual Studio 调试控制台
>>> before EAX = 0x111
>>> after EAX = 0x0
```

### H3 uc\_emu\_stop

```
1 uc_err uc_emu_stop(uc_engine *uc);
```

停止模拟

通常是从通过 tracing API 注册的回调函数中调用。

```
1 @uc: uc_open() 返回的句柄
2
3 @return 成功则返回 UC_ERR_OK , 否则返回 uc_err 枚举的其他错误类型
```

源码实现

```
1 uc_err uc_emu_stop(uc_engine *uc)
2 {
3     if (uc->emulation_done)
4         return UC_ERR_OK;
5
6     uc->stop_request = true;
7
8     if (uc->current_cpu) {
9         // 退出当前线程
10        cpu_exit(uc->current_cpu);
11    }
12
13    return UC_ERR_OK;
14 }
```

使用示例:

```
1 uc_emu_stop(uc);
```

### H3 uc\_hook\_add

```
1 uc_err uc_hook_add(uc_engine *uc, uc_hook *hh, int type,
2 void *callback,
3 void *user_data, uint64_t begin, uint64_t end,
4 ...);
```

注册hook事件的回调, 当hook事件被触发将会进行回调。

```

1  @uc: uc_open() 返回的句柄
2  @hh: 注册hook得到的句柄. uc_hook_del() 中使用
3  @type: hook 类型
4  @callback: 当指令被命中时要运行的回调
5  @user_data: 用户自定义数据. 将被传递给回调函数的最后一个参数
    @user_data
6  @begin: 回调生效区域的起始地址(包括)
7  @end: 回调生效区域的结束地址(包括)
8      注意 1: 只有回调的地址在[@begin, @end]中才会调用回调
9      注意 2: 如果 @begin > @end, 每当触发此hook类型时都会调用回调
10 @...: 变量参数 (取决于 @type)
11      注意: 如果 @type = UC_HOOK_INSN, 这里是指令ID (如:
    UC_X86_INS_OUT)
12
13 @return 成功则返回UC_ERR_OK , 否则返回 uc_err 枚举的其他错误类型

```

### 源码实现

```

1  uc_err uc_hook_add(uc_engine *uc, uc_hook *hh, int type,
    void *callback,
2      void *user_data, uint64_t begin, uint64_t end, ...)
3  {
4      int ret = UC_ERR_OK;
5      int i = 0;
6
7      struct hook *hook = calloc(1, sizeof(struct hook));
8      if (hook == NULL) {
9          return UC_ERR_NOMEM;
10     }
11
12     hook->begin = begin;
13     hook->end = end;
14     hook->type = type;
15     hook->callback = callback;
16     hook->user_data = user_data;
17     hook->refs = 0;
18     *hh = (uc_hook)hook;
19
20     // UC_HOOK_INSN 有一个额外参数: 指令ID
21     if (type & UC_HOOK_INSN) {
22         va_list valist;
23
24         va_start(valist, end);
25         hook->insn = va_arg(valist, int);
26         va_end(valist);
27
28         if (uc->insn_hook_validate) {
29             if (! uc->insn_hook_validate(hook->insn)) {
30                 free(hook);
31                 return UC_ERR_HOOK;

```

```

32         }
33     }
34
35     if (uc->hook_insert) {
36         if (list_insert(&uc->hook[UC_HOOK_INSN_IDX],
hook) == NULL) {
37             free(hook);
38             return UC_ERR_NOMEM;
39         }
40     } else {
41         if (list_append(&uc->hook[UC_HOOK_INSN_IDX],
hook) == NULL) {
42             free(hook);
43             return UC_ERR_NOMEM;
44         }
45     }
46
47     hook->refs++;
48     return UC_ERR_OK;
49 }
50
51 while ((type >> i) > 0) {
52     if ((type >> i) & 1) {
53         if (i < UC_HOOK_MAX) {
54             if (uc->hook_insert) {
55                 if (list_insert(&uc->hook[i], hook) ==
NULL) {
56                     if (hook->refs == 0) {
57                         free(hook);
58                     }
59                     return UC_ERR_NOMEM;
60                 }
61             } else {
62                 if (list_append(&uc->hook[i], hook) ==
NULL) {
63                     if (hook->refs == 0) {
64                         free(hook);
65                     }
66                     return UC_ERR_NOMEM;
67                 }
68             }
69             hook->refs++;
70         }
71     }
72     i++;
73 }
74
75 if (hook->refs == 0) {
76     free(hook);
77 }
78
79 return ret;

```

使用示例:

```

1  #include <iostream>
2  #include <string>
3  #include "unicorn/unicorn.h"
4  using namespace std;
5
6  int syscall_abi[] = {
7      UC_X86_REG_RAX, UC_X86_REG_RDI, UC_X86_REG_RSI,
8      UC_X86_REG_RDX,
9      UC_X86_REG_R10, UC_X86_REG_R8, UC_X86_REG_R9
10 };
11
12 uint64_t vals[7] = { 200, 10, 11, 12, 13, 14, 15 };
13
14 void* ptrs[7];
15
16 void uc_perror(const char* func, uc_err err)
17 {
18     fprintf(stderr, "Error in %s(): %s\n", func,
19         uc_strerror(err));
20 }
21
22 #define BASE 0x10000
23
24 // mov rax, 100; mov rdi, 1; mov rsi, 2; mov rdx, 3; mov
25 // r10, 4; mov r8, 5; mov r9, 6; syscall
26
27 #define CODE
28 "\x48\x7c\x00\x64\x00\x00\x00\x48\x7c\x7\x01\x00\x00\x00\x
29 48\x7c\x6\x02\x00\x00\x00\x48\x7c\x2\x03\x00\x00\x00\x49\
30 xc7\x2\x04\x00\x00\x00\x49\x7c\x0\x05\x00\x00\x00\x49\x7
31 \xc1\x06\x00\x00\x00\x0f\x05"
32
33 void hook_syscall(uc_engine* uc, void* user_data)
34 {
35     int i;
36
37     uc_reg_read_batch(uc, syscall_abi, ptrs, 7);
38
39     printf("syscall: {");
40
41     for (i = 0; i < 7; i++) {
42         if (i != 0) printf(", ");
43         printf("%" PRIu64, vals[i]);
44     }
45
46     printf("}\n");
47 }

```



```

41 void hook_code(uc_engine* uc, uint64_t addr, uint32_t size,
42 void* user_data)
43 {
44     printf("HOOK_CODE: 0x%" PRIx64 " ", 0x%x\n", addr, size);
45 }
46 int main()
47 {
48     int i;
49     uc_hook sys_hook;
50     uc_err err;
51     uc_engine* uc;
52
53     for (i = 0; i < 7; i++) {
54         ptrs[i] = &vals[i];
55     }
56
57     if ((err = uc_open(UC_ARCH_X86, UC_MODE_64, &uc))) {
58         uc_perror("uc_open", err);
59         return 1;
60     }
61
62     printf("reg_write_batch({200, 10, 11, 12, 13, 14,
63 15})\n");
64     if ((err = uc_reg_write_batch(uc, syscall_abi, ptrs,
65 7))) {
66         uc_perror("uc_reg_write_batch", err);
67         return 1;
68     }
69
70     memset(vals, 0, sizeof(vals));
71     if ((err = uc_reg_read_batch(uc, syscall_abi, ptrs,
72 7))) {
73         uc_perror("uc_reg_read_batch", err);
74         return 1;
75     }
76
77     printf("reg_read_batch = {");
78
79     for (i = 0; i < 7; i++) {
80         if (i != 0) printf(", ");
81         printf("%" PRIu64, vals[i]);
82     }
83
84     printf("}\n");
85
86     // syscall
87     printf("\n");
88     printf("running syscall shellcode\n");
89
90     if ((err = uc_hook_add(uc, &sys_hook, UC_HOOK_CODE,
91 hook_syscall, NULL, 1, 0))) {

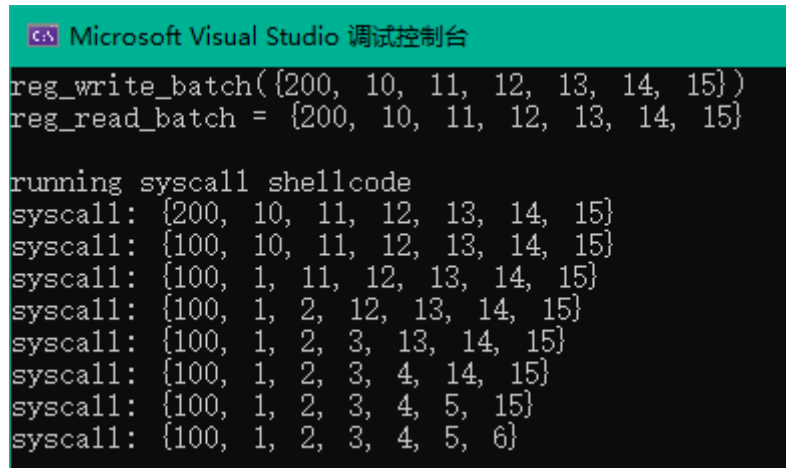
```

```

88         uc_perror("uc_hook_add", err);
89         return 1;
90     }
91
92     if ((err = uc_mem_map(uc, BASE, 0x1000, UC_PROT_ALL)))
93     {
94         uc_perror("uc_mem_map", err);
95         return 1;
96     }
97     if ((err = uc_mem_write(uc, BASE, CODE, sizeof(CODE) -
98 1))) {
99         uc_perror("uc_mem_write", err);
100        return 1;
101    }
102
103    if ((err = uc_emu_start(uc, BASE, BASE + sizeof(CODE) -
104 2, 1, 0, 0))) {
105        uc_perror("uc_emu_start", err);
106        return 1;
107    }
108
109    return 0;
110 }

```

输出



```

Microsoft Visual Studio 调试控制台
reg_write_batch({200, 10, 11, 12, 13, 14, 15})
reg_read_batch = {200, 10, 11, 12, 13, 14, 15}

running syscall shellcode
syscall: {200, 10, 11, 12, 13, 14, 15}
syscall: {100, 10, 11, 12, 13, 14, 15}
syscall: {100, 1, 11, 12, 13, 14, 15}
syscall: {100, 1, 2, 12, 13, 14, 15}
syscall: {100, 1, 2, 3, 13, 14, 15}
syscall: {100, 1, 2, 3, 4, 14, 15}
syscall: {100, 1, 2, 3, 4, 5, 15}
syscall: {100, 1, 2, 3, 4, 5, 6}

```

对每条指令都进行hook

### H3 uc\_hook\_del

```
1 uc_err uc_hook_del(uc_engine *uc, uc_hook hh);
```

删除一个已注册的hook事件

```
1 @uc: uc_open() 返回的句柄
2 @hh: uc_hook_add() 返回的句柄
3
4 @return 成功则返回UC_ERR_OK , 否则返回 uc_err 枚举的其他错误类型
```

源码实现

```
1 uc_err uc_hook_del(uc_engine *uc, uc_hook hh)
2 {
3     int i;
4     struct hook *hook = (struct hook *)hh;
5
6     for (i = 0; i < UC_HOOK_MAX; i++) {
7         if (list_remove(&uc->hook[i], (void *)hook)) {
8             if (--hook->refs == 0) {
9                 free(hook);
10                break;
11            }
12        }
13    }
14    return UC_ERR_OK;
15 }
```

使用示例:

```
1 if ((err = uc_hook_add(uc, &sys_hook, UC_HOOK_CODE,
2     hook_syscall, NULL, 1, 0))) {
3     uc_perror("uc_hook_add", err);
4     return 1;
5 }
6 if ((err = uc_hook_del(uc, &sys_hook))) {
7     uc_perror("uc_hook_del", err);
8     return 1;
9 }
```

### H3 uc\_mem\_map

```
1 uc_err uc_mem_map(uc_engine *uc, uint64_t address, size_t
2     size, uint32_t perms);
```

为模拟映射一块内存。

- 1 @uc: uc\_open() 返回的句柄
- 2 @address: 要映射到的新内存区域的起始地址。这个地址必须与4KB对齐，否则将返回UC\_ERR\_ARG错误。
- 3 @size: 要映射到的新内存区域的大小。这个大小必须是4KB的倍数，否则将返回UC\_ERR\_ARG错误。
- 4 @perms: 新映射区域的权限。参数必须是UC\_PROT\_READ | UC\_PROT\_WRITE | UC\_PROT\_EXEC或这些的组合，否则返回UC\_ERR\_ARG错误。
- 5
- 6 @return 成功则返回UC\_ERR\_OK，否则返回 uc\_err 枚举的其他错误类型

#### 源码实现

```
1  uc_err uc_mem_map(uc_engine *uc, uint64_t address, size_t
   size, uint32_t perms)
2  {
3      uc_err res;
4
5      if (uc->mem_redirect) {
6          address = uc->mem_redirect(address);
7      }
8
9      res = mem_map_check(uc, address, size, perms);    //内存
   安全检查
10     if (res)
11         return res;
12
13     return mem_map(uc, address, size, perms, uc-
   >memory_map(uc, address, size, perms));
14 }
```

使用示例同uc\_hook\_add。

### H3 uc\_mem\_map\_ptr

```
1  uc_err uc_mem_map_ptr(uc_engine *uc, uint64_t address,
   size_t size, uint32_t perms, void *ptr);
```

在模拟中映射现有的主机内存。

- 1 @uc: uc\_open() 返回的句柄
- 2 @address: 要映射到的新内存区域的起始地址。这个地址必须与4KB对齐, 否则将返回UC\_ERR\_ARG错误。
- 3 @size: 要映射到的新内存区域的大小。这个大小必须是4KB的倍数, 否则将返回UC\_ERR\_ARG错误。
- 4 @perms: 新映射区域的权限。参数必须是UC\_PROT\_READ | UC\_PROT\_WRITE | UC\_PROT\_EXEC或这些的组合, 否则返回UC\_ERR\_ARG错误。
- 5 @ptr: 指向支持新映射内存的主机内存的指针。映射的主机内存的大小应该与size的大小相同或更大, 并且至少使用PROT\_READ | PROT\_WRITE进行映射, 否则不定义映射。
- 6
- 7 @return 成功则返回UC\_ERR\_OK, 否则返回 uc\_err 枚举的其他错误类型

#### 源码实现

```
1  uc_err uc_mem_map_ptr(uc_engine *uc, uint64_t address,
2  size_t size, uint32_t perms, void *ptr)
3  {
4      uc_err res;
5
6      if (ptr == NULL)
7          return UC_ERR_ARG;
8
9      if (uc->mem_redirect) {
10         address = uc->mem_redirect(address);
11     }
12
13     res = mem_map_check(uc, address, size, perms);    //内存
14     安全检查
15     if (res)
16         return res;
17
18     return mem_map(uc, address, size, UC_PROT_ALL, uc-
19     >memory_map_ptr(uc, address, size, perms, ptr));
20 }
```

使用示例同uc\_mem\_map

### H3 uc\_mem\_unmap

```
1  uc_err uc_mem_unmap(uc_engine *uc, uint64_t address, size_t
2  size);
```

取消对模拟内存区域的映射

- 1 @uc: uc\_open() 返回的句柄
- 2 @address: 要映射到的新内存区域的起始地址。这个地址必须与4KB对齐，否则将返回UC\_ERR\_ARG错误。
- 3 @size: 要映射到的新内存区域的大小。这个大小必须是4KB的倍数，否则将返回UC\_ERR\_ARG错误。
- 4
- 5 @return 成功则返回UC\_ERR\_OK，否则返回 uc\_err 枚举的其他错误类型

## 源码实现

```
1  uc_err uc_mem_unmap(struct uc_struct *uc, uint64_t address,
2                      size_t size)
3  {
4      MemoryRegion *mr;
5      uint64_t addr;
6      size_t count, len;
7
8      if (size == 0)
9          // 没有要取消映射的区域
10         return UC_ERR_OK;
11
12     // 地址必须对齐到 uc->target_page_size
13     if ((address & uc->target_page_align) != 0)
14         return UC_ERR_ARG;
15
16     // 大小必须是 uc->target_page_size 的倍数
17     if ((size & uc->target_page_align) != 0)
18         return UC_ERR_ARG;
19
20     if (uc->mem_redirect) {
21         address = uc->mem_redirect(address);
22     }
23
24     // 检查用户请求的整个块是否被映射
25     if (!check_mem_area(uc, address, size))
26         return UC_ERR_NOMEM;
27
28     // 如果这个区域跨越了相邻的区域，可能需要分割区域
29     addr = address;
30     count = 0;
31     while(count < size) {
32         mr = memory_mapping(uc, addr);
33         len = (size_t)MIN(size - count, mr->end - addr);
34         if (!split_region(uc, mr, addr, len, true))
35             return UC_ERR_NOMEM;
36
37         // 取消映射
38         mr = memory_mapping(uc, addr);
39         if (mr != NULL)
40             uc->memory_unmap(uc, mr);
```

```

40         count += len;
41         addr += len;
42     }
43
44     return UC_ERR_OK;
45 }

```

使用示例:

```

1  if ((err = uc_mem_map(uc, BASE, 0x1000, UC_PROT_ALL))) {
2      uc_perror("uc_mem_map", err);
3      return 1;
4  }
5
6  if ((err = uc_mem_unmap(uc, BASE, 0x1000))) {
7      uc_perror("uc_mem_unmap", err);
8      return 1;
9  }

```

### H3 uc\_mem\_protect

```

1  uc_err uc_mem_protect(uc_engine *uc, uint64_t address,
    size_t size, uint32_t perms);

```

设置模拟内存的权限

```

1  @uc: uc_open() 返回的句柄
2  @address: 要映射到的新内存区域的起始地址。这个地址必须与4KB对齐，
    否则将返回UC_ERR_ARG错误。
3  @size: 要映射到的新内存区域的大小。这个大小必须是4KB的倍数，否则
    将返回UC_ERR_ARG错误。
4  @perms: 映射区域的新权限。参数必须是UC_PROT_READ |
    UC_PROT_WRITE | UC_PROT_EXEC或这些的组合，否则返回UC_ERR_ARG错
    误。
5
6  @return 成功则返回UC_ERR_OK，否则返回 uc_err 枚举的其他错误类
    型

```

源码实现

```

1  uc_err uc_mem_protect(struct uc_struct *uc, uint64_t
    address, size_t size, uint32_t perms)
2  {
3      MemoryRegion *mr;
4      uint64_t addr = address;
5      size_t count, len;
6      bool remove_exec = false;
7
8      if (size == 0)
9          // trivial case, no change

```

```

10         return UC_ERR_OK;
11
12         // address must be aligned to uc->target_page_size
13         if ((address & uc->target_page_align) != 0)
14             return UC_ERR_ARG;
15
16         // size must be multiple of uc->target_page_size
17         if ((size & uc->target_page_align) != 0)
18             return UC_ERR_ARG;
19
20         // check for only valid permissions
21         if ((perms & ~UC_PROT_ALL) != 0)
22             return UC_ERR_ARG;
23
24         if (uc->mem_redirect) {
25             address = uc->mem_redirect(address);
26         }
27
28         // check that user's entire requested block is mapped
29         if (!check_mem_area(uc, address, size))
30             return UC_ERR_NOMEM;
31
32         // Now we know entire region is mapped, so change
permissions
33         // We may need to split regions if this area spans
adjacent regions
34         addr = address;
35         count = 0;
36         while(count < size) {
37             mr = memory_mapping(uc, addr);
38             len = (size_t)MIN(size - count, mr->end - addr);
39             if (!split_region(uc, mr, addr, len, false))
40                 return UC_ERR_NOMEM;
41
42             mr = memory_mapping(uc, addr);
43             // will this remove EXEC permission?
44             if (((mr->perms & UC_PROT_EXEC) != 0) && ((perms &
UC_PROT_EXEC) == 0))
45                 remove_exec = true;
46             mr->perms = perms;
47             uc->readonly_mem(mr, (perms & UC_PROT_WRITE) == 0);
48
49             count += len;
50             addr += len;
51         }
52
53         // if EXEC permission is removed, then quit TB and
continue at the same place
54         if (remove_exec) {
55             uc->quit_request = true;
56             uc_emu_stop(uc);
57         }

```



```

58
59     return UC_ERR_OK;
60 }

```

使用示例:

```

1  if ((err = uc_mem_protect(uc, BASE, 0x1000, UC_PROT_ALL)))
    { //可读可写可执行
2      uc_perror("uc_mem_protect", err);
3      return 1;
4  }

```

### H3 uc\_mem\_regions

```

1  uc_err uc_mem_regions(uc_engine *uc, uc_mem_region
    **regions, uint32_t *count);

```

检索由 uc\_mem\_map() 和 uc\_mem\_map\_ptr() 映射的内存的信息。

这个API为@regions分配内存，用户之后必须通过free()释放这些内存来避免内存泄漏。

```

1  @uc: uc_open() 返回的句柄
2  @regions: 指向 uc_mem_region 结构体的数组的指针。由Unicorn申
    请，必须通过uc_free()释放这些内存
3  @count: 指向@regions中包含的uc_mem_region结构体的数量的指针
4
5  @return 成功则返回UC_ERR_OK，否则返回 uc_err 枚举的其他错误类
    型

```

源码分析

```

1  uint32_t uc_mem_regions(uc_engine *uc, uc_mem_region
    **regions, uint32_t *count)
2  {
3      uint32_t i;
4      uc_mem_region *r = NULL;
5
6      *count = uc->mapped_block_count;
7
8      if (*count) {
9          r = g_malloc0(*count * sizeof(uc_mem_region));
10         if (r == NULL) {
11             // 内存不足
12             return UC_ERR_NOMEM;
13         }
14     }
15
16     for (i = 0; i < *count; i++) {
17         r[i].begin = uc->mapped_blocks[i]->addr;
18         r[i].end = uc->mapped_blocks[i]->end - 1;
19         r[i].perms = uc->mapped_blocks[i]->perms;

```

```

20     }
21
22     *regions = r;
23
24     return UC_ERR_OK;
25 }

```

使用示例:

```

1  #include <iostream>
2  #include <string>
3  #include "unicorn/unicorn.h"
4  using namespace std;
5
6  int main()
7  {
8      uc_err err;
9      uc_engine* uc;
10
11     if ((err = uc_open(UC_ARCH_X86, UC_MODE_64, &uc))) {
12         uc_perror("uc_open", err);
13         return 1;
14     }
15
16     if ((err = uc_mem_map(uc, BASE, 0x1000, UC_PROT_ALL)))
17     {
18         uc_perror("uc_mem_map", err);
19         return 1;
20     }
21
22     uc_mem_region *region;
23     uint32_t count;
24
25     if ((err = uc_mem_regions(uc, &region, &count))) {
26         uc_perror("uc_mem_regions", err);
27         return 1;
28     }
29
30     cout << "起始地址: 0x" << hex << region->begin << " 结
31     束地址: 0x" << hex << region->end << " 内存权限: "
32     << region->perms << " 已申请内存块数: " << count << endl;
33
34     if ((err = uc_free(region))) {      ////注意释放内存
35         uc_perror("uc_free", err);
36         return 1;
37     }
38
39     return 0;
40 }

```

输出



### H3 uc\_free

```
1 uc_err uc_free(void *mem);
```

释放由 uc\_context\_alloc 和 uc\_mem\_regions 申请的内存

```
1 @mem: 由uc_context_alloc (返回 *context), 或由
      uc_mem_regions (返回 *regions)申请的内存
2
3 @return 成功则返回UC_ERR_OK , 否则返回 uc_err 枚举的其他错误类型
```

源码实现

```
1 uc_err uc_free(void *mem)
2 {
3     g_free(mem);
4     return UC_ERR_OK;
5 }
6
7 void g_free(gpointer ptr)
8 {
9     free(ptr);
10 }
```

使用示例同uc\_mem\_regions

### H3 uc\_context\_alloc

```
1 uc_err uc_context_alloc(uc_engine *uc, uc_context
      **context);
```

分配一个可以与uc\_context\_{save,restore}一起使用的区域来执行CPU上下文的快速保存/回滚, 包括寄存器和内部元数据。上下文不能在具有不同架构或模式的引擎实例之间共享。

```
1 @uc: uc_open() 返回的句柄
2 @context: 指向uc_engine*的指针。当这个函数成功返回时, 将使用指向
      新上下文的指针更新它。之后必须使用uc_free()释放这些分配的内存。
3
4 @return 成功则返回UC_ERR_OK , 否则返回 uc_err 枚举的其他错误类型
```

源码实现

```

1  uc_err uc_context_alloc(uc_engine *uc, uc_context
   **context)
2  {
3      struct uc_context **_context = context;
4      size_t size = cpu_context_size(uc->arch, uc->mode);
5
6      *_context = malloc(size + sizeof(uc_context));
7      if (*_context) {
8          (*_context)->size = size;
9          return UC_ERR_OK;
10     } else {
11         return UC_ERR_NOMEM;
12     }
13 }

```

### 使用示例

```

1  #include <iostream>
2  #include <string>
3  #include "unicorn/unicorn.h"
4  using namespace std;
5
6  #define ADDRESS 0x1000
7  #define X86_CODE32_INC "\x40"    // INC eax
8
9  int main()
10 {
11     uc_engine* uc;
12     uc_context* context;
13     uc_err err;
14
15     int r_eax = 0x1;    // EAX 寄存器
16
17     printf("=====\n");
18     printf("Save/restore CPU context in opaque blob\n");
19
20     err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
21     if (err) {
22         printf("Failed on uc_open() with error returned:
23 %u\n", err);
24         return 0;
25     }
26
27     uc_mem_map(uc, ADDRESS, 8 * 1024, UC_PROT_ALL);
28
29     if (uc_mem_write(uc, ADDRESS, X86_CODE32_INC,
30 sizeof(X86_CODE32_INC) - 1)) {
31         printf("Failed to write emulation code to memory,
32 quit!\n");
33         return 0;
34     }
35 }

```

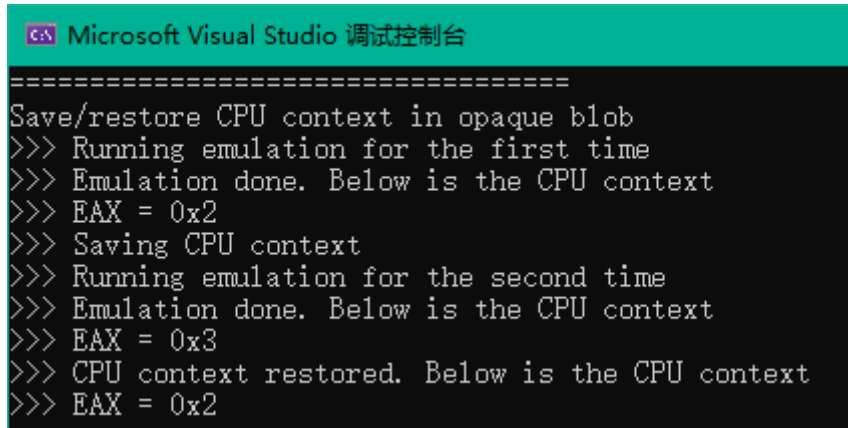
```
32
33     // 初始化寄存器
34     uc_reg_write(uc, UC_X86_REG_EAX, &r_eax);
35
36     printf(">>> Running emulation for the first time\n");
37
38     err = uc_emu_start(uc, ADDRESS, ADDRESS +
39 sizeof(X86_CODE32_INC) - 1, 0, 0);
39     if (err) {
40         printf("Failed on uc_emu_start() with error
41 returned %u: %s\n",
42             err, uc_strerror(err));
43     }
44     printf(">>> Emulation done. Below is the CPU
45 context\n");
46
47     uc_reg_read(uc, UC_X86_REG_EAX, &r_eax);
48     printf(">>> EAX = 0x%x\n", r_eax);
49
50     // 申请并保存 CPU 上下文
51     printf(">>> Saving CPU context\n");
52
53     err = uc_context_alloc(uc, &context);
54     if (err) {
55         printf("Failed on uc_context_alloc() with error
56 returned: %u\n", err);
57         return 0;
58     }
59
60     err = uc_context_save(uc, context);
61     if (err) {
62         printf("Failed on uc_context_save() with error
63 returned: %u\n", err);
64         return 0;
65     }
66
67     printf(">>> Running emulation for the second time\n");
68
69     err = uc_emu_start(uc, ADDRESS, ADDRESS +
70 sizeof(X86_CODE32_INC) - 1, 0, 0);
71     if (err) {
72         printf("Failed on uc_emu_start() with error
73 returned %u: %s\n",
74             err, uc_strerror(err));
75     }
76
77     printf(">>> Emulation done. Below is the CPU
78 context\n");
79
80     uc_reg_read(uc, UC_X86_REG_EAX, &r_eax);
81     printf(">>> EAX = 0x%x\n", r_eax);
```

```

76
77     // 恢复 CPU 上下文
78     err = uc_context_restore(uc, context);
79     if (err) {
80         printf("Failed on uc_context_restore() with error
returned: %u\n", err);
81         return 0;
82     }
83
84     printf(">>> CPU context restored. Below is the CPU
context\n");
85
86     uc_reg_read(uc, UC_X86_REG_EAX, &r_eax);
87     printf(">>> EAX = 0x%x\n", r_eax);
88
89     // 释放 CPU 上下文
90     err = uc_free(context);
91     if (err) {
92         printf("Failed on uc_free() with error returned:
%u\n", err);
93         return 0;
94     }
95
96     uc_close(uc);
97
98     return 0;
99 }

```

输出



```

Microsoft Visual Studio 调试控制台
=====
Save/restore CPU context in opaque blob
>>> Running emulation for the first time
>>> Emulation done. Below is the CPU context
>>> EAX = 0x2
>>> Saving CPU context
>>> Running emulation for the second time
>>> Emulation done. Below is the CPU context
>>> EAX = 0x3
>>> CPU context restored. Below is the CPU context
>>> EAX = 0x2

```

### H3 uc\_context\_save

```
1 uc_err uc_context_save(uc_engine *uc, uc_context *context);
```

保存当前CPU上下文

1 @uc: uc\_open() 返回的句柄  
2 @context: uc\_context\_alloc() 返回的句柄  
3  
4 @return 成功则返回UC\_ERR\_OK , 否则返回 uc\_err 枚举的其他错误类型

源码实现

```
1 uc_err uc_context_save(uc_engine *uc, uc_context *context)
2 {
3     struct uc_context *_context = context;
4     memcpy(_context->data, uc->cpu->env_ptr, _context->size);
5     return UC_ERR_OK;
6 }
```

使用示例同uc\_context\_alloc()

### H3 uc\_context\_restore

```
1 uc_err uc_context_restore(uc_engine *uc, uc_context *context);
```

恢复已保存的CPU上下文

1 @uc: uc\_open() 返回的句柄  
2 @context: uc\_context\_alloc() 返回并且已使用 uc\_context\_save 保存的句柄  
3  
4 @return 成功则返回UC\_ERR\_OK , 否则返回 uc\_err 枚举的其他错误类型

源码实现

```
1 uc_err uc_context_restore(uc_engine *uc, uc_context *context)
2 {
3     struct uc_context *_context = context;
4     memcpy(uc->cpu->env_ptr, _context->data, _context->size);
5     return UC_ERR_OK;
6 }
```

使用示例同uc\_context\_alloc()

### H3 uc\_context\_size

```
1 size_t uc_context_size(uc_engine *uc);
```

返回存储cpu上下文所需的大小。可以用来分配一个缓冲区来包含cpu上下文，并直接调用uc\_context\_save。

- 1 @uc: uc\_open() 返回的句柄
- 2
- 3 @return 存储cpu上下文所需的大小，类型为 size\_t.

#### 源码实现

```
1  size_t uc_context_size(uc_engine *uc)
2  {
3      return cpu_context_size(uc->arch, uc->mode);
4  }
5
6  static size_t cpu_context_size(uc_arch arch, uc_mode mode)
7  {
8      switch (arch) {
9          #ifdef UNICORN_HAS_M68K
10             case UC_ARCH_M68K: return M68K_REGS_STORAGE_SIZE;
11          #endif
12          #ifdef UNICORN_HAS_X86
13             case UC_ARCH_X86: return X86_REGS_STORAGE_SIZE;
14          #endif
15          #ifdef UNICORN_HAS_ARM
16             case UC_ARCH_ARM: return mode &
17                 UC_MODE_BIG_ENDIAN ? ARM_REGS_STORAGE_SIZE_armeb :
18                 ARM_REGS_STORAGE_SIZE_arm;
19          #endif
20          #ifdef UNICORN_HAS_ARM64
21             case UC_ARCH_ARM64: return mode &
22                 UC_MODE_BIG_ENDIAN ? ARM64_REGS_STORAGE_SIZE_aarch64eb :
23                 ARM64_REGS_STORAGE_SIZE_aarch64;
24          #endif
25          #ifdef UNICORN_HAS_MIPS
26             case UC_ARCH_MIPS:
27                 if (mode & UC_MODE_MIPS64) {
28                     if (mode & UC_MODE_BIG_ENDIAN) {
29                         return MIPS64_REGS_STORAGE_SIZE_mips64;
30                     } else {
31                         return
32                             MIPS64_REGS_STORAGE_SIZE_mips64el;
33                     }
34                 } else {
35                     if (mode & UC_MODE_BIG_ENDIAN) {
36                         return MIPS_REGS_STORAGE_SIZE_mips;
37                     } else {
38                         return MIPS_REGS_STORAGE_SIZE_mipsel;
39                     }
40                 }
41             #endif
42          #ifdef UNICORN_HAS_SPARC
```



```
38         case UC_ARCH_SPARC: return mode & UC_MODE_SPARC64 ?  
        SPARC64_REGS_STORAGE_SIZE : SPARC_REGS_STORAGE_SIZE;  
39     #endif  
40         default: return 0;  
41     }  
42 }
```

使用示例同uc\_context\_alloc()