Unicorn-Engine API Documentation

Version	1.0.3

Official API document by kabeor

Translated by Nitr0-G

PDF File

Unicorn Engine - легкий, мультиплатформенный, мультиархитектурный фреймворк эмулятора процессора, текущая версия которого основана на Qemu 2.0.x он может заменить выполнение кода моделирования процессора и обычно используется для анализа вредоносного кода，Fuzzing, Qiling,Radare2,GEF,Pwndbg，Angr。

0x0 Подготовка к разработке

Unicorn http://www.unicorn-engine.org

Скомпилируйте lib и dll самостоятельно

Source code: https://github.com/unicorn-engine/unicorn/archive/master.zip

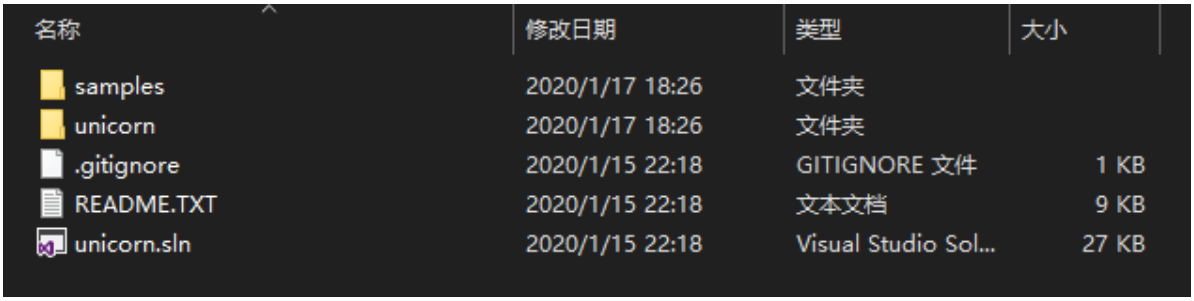
```
. <- 主要引擎core engine + README + 编译文档COMPILE.TXT 等
├── bindings <- 绑定
| ├── dotnet <- .Net 绑定 + 测试代码
| ├── go <- go 绑定 + 测试代码
| ├── haskell <- Haskell 绑定 + 测试代码
| ├── java <- Java 绑定 + 测试代码
| ├── pascal <- Pascal 绑定 + 测试代码
| ├── python <- Python 绑定 + 测试代码
| ├── ruby <- Ruby 绑定 + 测试代码
| ├── rust <- Rust 绑定 + 测试代码
| └── vb6 <- VB6 绑定 + 测试代码
├── docs <- 文档，主要是Unicorn的实现思路
├── include <- C头文件
├── msvc <- Microsoft Visual Studio 支持（Windows）
├── out <- Build 输出
├── qemu <- qemu(已修改)源码
├── samples <- Unicorn使用示例
└── tests <- C语言测试用例
```

Ниже показано, что Windows10 использует

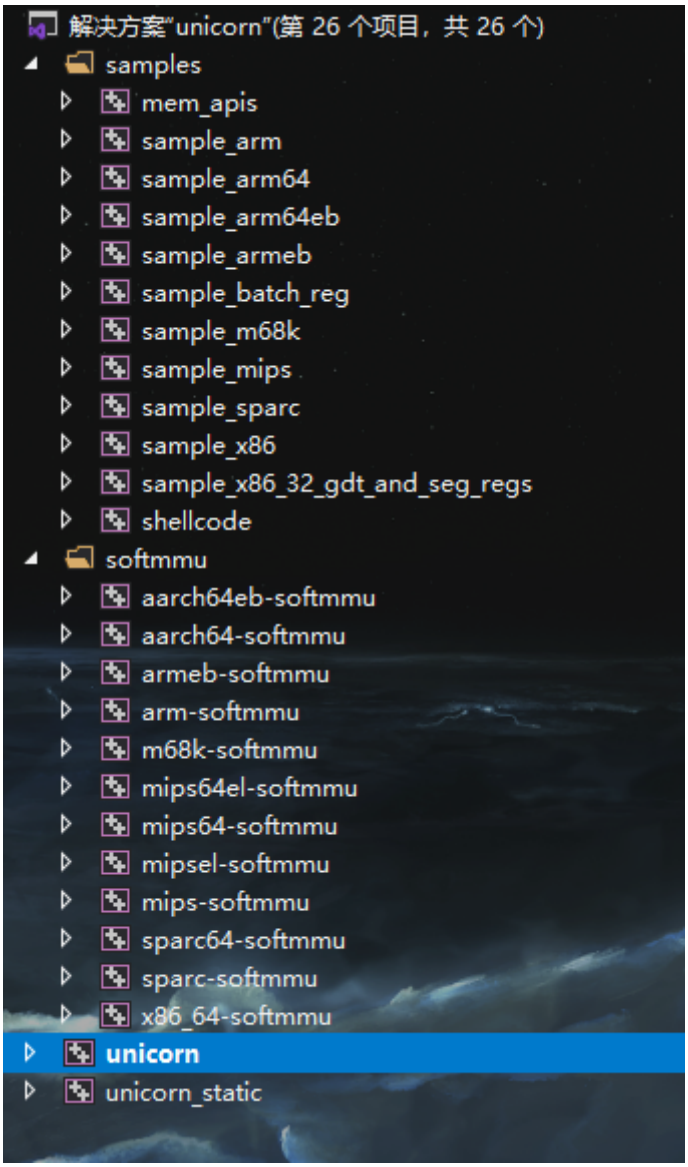
Visual Studio 2019 для компиляции и

открытия папки msvc. Внутренняя

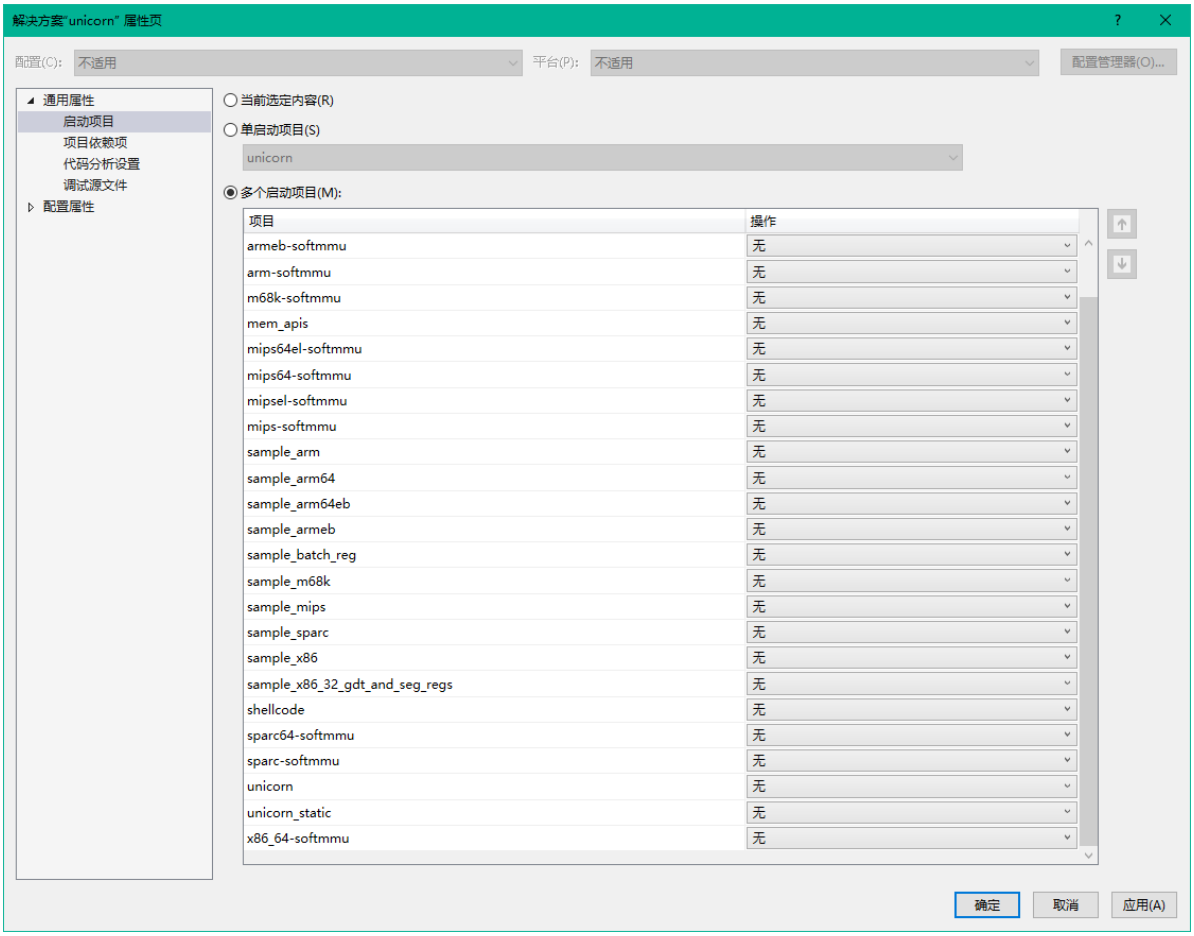
структура выглядит следующим образом



VS open unicorn.sln project files, the solution automatically loads these



Если вам нужны все они, просто скомпилируйте их напрямую. Вам нужны только некоторые из них. Затем щелкните правой кнопкой мыши Solution->Properties->Configuration Properties->Generate options и проверьте необходимые элементы поддержки.
Вы также можете настроить несколько проектов в стартовом проекте следующим образом:



После компиляции unicorn будет сгенерирован в каталоге Debug текущей папки.библиотека статической компиляции lib и unicorn.динамическая библиотека dll, чтобы вы могли начать использовать Unicorn для разработки.

Последняя скомпилированная версия, предоставляемая в настоящее время официальным сайтом, - версия 1.0.3. Вы можете самостоятельно отредактировать исходный код последней версии, чтобы получить больше доступных API.。

> Win32：https://github.com/unicorn-engine/unicorn/releases/download/1.0.1/unicorn-1.0.3-win32.zip

> Win64：https://github.com/unicorn-engine/unicorn/releases/download/1.0.1/unicorn-1.0.3-win64.zip

Note: Choosing x32 or x64 will affect the architecture developed later

Click to compile: unicorn\msvc\x32\x64\Debug\Release\unicorn.dll\unicorn.lib.

Engine call test

Create a new VS project and will..Copy all the header files in \unicorn-master\include\unicorn and the compiled lib and dll files to the home directory of the newly created project

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

In the VS solution, the header file adds the existing item unicorn.h, add unicorn to the resource file.lib, regenerate the solution

Затем протестируйте основной код файла движка Unicorn, который

мы сгенерировали следующим образом

▶ Code

```
#include <iostream>
#include "unicorn/unicorn.h"

// Инструкции, которые должны
быть смоделированы
#define X86_CODE32 "\x41\x4a" // INC ecx; DEC edx

// Начальный адрес
#define ADDRESS 0x1000000

int main()
{   uc_engine* uc;
    uc_err err;
    int r_ecx = 0x1234;     // ECX declaretion
    int r_edx = 0x7890;    // EDX declaretion

    printf("Emulate i386 code\n");

    // X86-32bit инициализация
    err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_open() with error returned: %u\n", err);
            return -1;
```

```
    }

    // Выделяем 2 миба для эмулятора
    uc_mem_map(uc, ADDRESS, 2 * 1024 * 1024, UC_PROT_ALL);

    // Записываем в память инструкции.
    if (uc_mem_write(uc, ADDRESS, X86_CODE32, sizeof(X86_CODE32) -
   1)) {printf("Failed to write emulation code to memory, quit!\n");
        return -1;
    }

    // Инициализация регистров
    uc_reg_write(uc, UC_X86_REG_ECX, &r_ecx);
    uc_reg_write(uc, UC_X86_REG_EDX, &r_edx);

    printf(">>> ECX = 0x%x\n", r_ecx);
    printf(">>> EDX = 0x%x\n", r_edx);

     // Старт
    err = uc_emu_start(uc, ADDRESS, ADDRESS + sizeof(X86_CODE32) - 1, 0, 0);
    if (err) {
        printf("Failed on uc_emu_start() with error returned %u: %s\n",
        err, uc_strerror(err));
    }

    // Значение регистров
    printf("Emulation done. Below is the CPU context\n");

    uc_reg_read(uc, UC_X86_REG_ECX, &r_ecx);
    uc_reg_read(uc, UC_X86_REG_EDX, &r_edx);
    printf(">>> ECX = 0x%x\n", r_ecx);
    printf(">>> EDX = 0x%x\n", r_edx);

    uc_close(uc);

    return 0;
}
```

Результаты операции следующие

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

ecx(inc) и edx(dec) были успешно смоделированы.

> Индексы

uc_arch

uc_mode

uc_err

---

uc_arch

Выбор структуры(Structure selection)

▶ Code

```
typedef enum uc_arch {
    UC_ARCH_ARM = 1,    // ARM ( Thumb, Thumb-2)
    UC_ARCH_ARM64,      // ARM-64, called AArch64
    UC_ARCH_MIPS,       // Mips
    UC_ARCH_X86,        // X86 (x86 & x86-64)
    UC_ARCH_PPC,        // PowerPC
    UC_ARCH_SPARC,      // Sparc
    UC_ARCH_M68K,       // M68K
    UC_ARCH_MAX,
} uc_arch;
```

uc_mode

Выбор режима(Mode selection)

▶ Code

```
typedef enum uc_mode {
    UC_MODE_LITTLE_ENDIAN = 0,    // Little-endian preface mode (default)
   UC_MODE_BIG_ENDIAN = 1 << 30, // Big endian mode

    // arm / arm64
    UC_MODE_ARM = 0,
    UC_MODE_THUMB = 1 << 4,
    UC_MODE_MCLASS = 1 << 5,
    UC_MODE_V8 = 1 << 6,

    // arm (32bit) cpu 类型
    UC_MODE_ARM926 = 1 << 7,
    UC_MODE_ARM946 = 1 << 8,
    UC_MODE_ARM1176 = 1 << 9,

    // mips
    UC_MODE_MICRO = 1 << 4,
    UC_MODE_MIPS3 = 1 << 5,
    UC_MODE_MIPS32R6 = 1 << 6,
    UC_MODE_MIPS32 = 1 << 2,
    UC_MODE_MIPS64 = 1 << 3,
```

```
 // ARM mode
 // THUMB mode (including Thumb-2)
 // ARM's Cortex-M series (Not supported
yet)
 // ARMv8 A32 encodings for ARM (Not
supported yet)


 // ARM926 CPU type // ARM946 CPU type //
ARM1176 CPU type


//MicroMips(mode)(Not.supported.yet)
//MipsIII(ISA)(Not.supported.yet)
//Mips32r6(ISA)(Not.supported.yet)
//Mips32(ISA)
// Mips64(ISA)
```

```
    // x86 / x64
    UC_MODE_16 = 1 << 1,          // 16-bit mode
    UC_MODE_32 = 1 << 2,          // 32-bit mode
    UC_MODE_64 = 1 << 3,          // 64-bit mode

    // ppc
    UC_MODE_PPC32 = 1 << 2,       //32-bit(mode)(Not.supported.yet)
    UC_MODE_PPC64 = 1 << 3,       //64-bit(mode)(Not.supported.yet)
    UC_MODE_QPX = 1 << 4,         //Quad Processing eXtensions(mode)
                                  (Not.supported.yet)

    // sparc
    UC_MODE_SPARC32 = 1 << 2,     //32-bit(SPARC32.mode)
    UC_MODE_SPARC64 = 1 << 3,     //64-bit(SPARC64.mode)
    UC_MODE_V9 = 1 << 4,          // SparcV9(mode)(Not.supported.yet)

    // m68k
} uc_mode;
```

uc_err

Error Type, uc_errno()The return value of

▶ Code

```
typedef enum uc_err {
    UC_ERR_OK = 0,
    UC_ERR_NOMEM,   // No.error
                   //Insufficient memory: uc_open(), uc_emulate()
    UC_ERR_ARCH,   // Unsupported architecture: uc_open()
    UC_ERR_HANDLE,  // Unavailable handle
    UC_ERR_MODE,    // Unavailable/unsupported architecture: uc_open()
    UC_ERR_VERSION, // Unsupported version (middleware)
    UC_ERR_READ_UNMAPPED, /reading on unmapped memory(Quit emu): uc_emu_start()
    UC_ERR_WRITE_UNMAPPED, /writing on unmapped memory(Quit emu): uc_emu_start()
    UC_ERR_FETCH_UNMAPPED, /obtaining data in unmapped memory(Quit emu): uc_emu_start()
    UC_ERR_HOOK,    /Invalid hook type: uc_hook_add()
    UC_ERR_INSN_INVALID, /invalid instruction(Quit emu): uc_emu_start()
    UC_ERR_MAP, /Invalid memory mapping: uc_mem_map()
    UC_ERR_WRITE_PROT, /UC_MEM_WRITE_PROT conflict(Stop emu): uc_emu_start()
    UC_ERR_READ_PROT, /UC_MEM_READ_PROT conflict(Stop emu): uc_emu_start()
    UC_ERR_FETCH_PROT, /UC_MEM_FETCH_PROT conflict(Stop emu): uc_emu_start()
    UC_ERR_ARG,     /Invalid parameter provided to uc_xxx function
    UC_ERR_READ_UNALIGNED,  /Misaligned reading
    UC_ERR_WRITE_UNALIGNED,  /Unaligned write
    UC_ERR_FETCH_UNALIGNED,  /Unaligned extraction
    UC_ERR_HOOK_EXIST,  /The hook for this event already exists
    UC_ERR_RESOURCE,    /Insufficient resources: uc_emu_start()
    UC_ERR_EXCEPTION, /Unhandled CPU exception
    UC_ERR_TIMEOUT /Emulate timeout
} uc_err;
```

uc_mem_type

All memory access types of
UC_HOOK_MEM_*

▶ Code

```
typedef enum uc_mem_type {
    UC_MEM_READ = 16,   /Memory from(...)read
    UC_MEM_WRITE,       /Memory is written to(...)
    UC_MEM_FETCH,       /Memory is acquired
    UC_MEM_READ_UNMAPPED,    /Unmapped memory slave(...)read
    UC_MEM_WRITE_UNMAPPED,   /Unmapped memory is written to(...)
    UC_MEM_FETCH_UNMAPPED,   /Unmapped memory is acquired
    UC_MEM_WRITE_PROT, /Memory write protection, but mapped
    UC_MEM_READ_PROT,   /Memory read protection, but mapped
    UC_MEM_FETCH_PROT, /Memory is not executable, but mapped
    UC_MEM_READ_AFTER, /Memory is read from (successfully accessed address)
} uc_mem_type;
```

uc_hook_type

uc_hook_add()All hook type
parameters

▶ Code

```
typedef enum uc_hook_type {
    // Hook - All interrupt/syscall events
    UC_HOOK_INTR = 1 << 0,
    // Hook - A specific instruction-only a very small subset
   of instructions is supported
    UC_HOOK_INSN = 1 << 1,
    // Hook — A piece of code
    UC_HOOK_CODE = 1 << 2,
    // Hook Basic block
    UC_HOOK_BLOCK = 1 << 3,
    // Hook used to read memory on unmapped memory
    UC_HOOK_MEM_READ_UNMAPPED = 1 << 4,
    // Hook Invalid memory write event
    UC_HOOK_MEM_WRITE_UNMAPPED = 1 << 5,
    // Hook Invalid memory for execution events
    UC_HOOK_MEM_FETCH_UNMAPPED = 1 << 6,
    // Hook Read-protected memory
    UC_HOOK_MEM_READ_PROT = 1 << 7,
    // Hook Write-protected memory
    UC_HOOK_MEM_WRITE_PROT = 1 << 8,
    // Hook Memory on non-executable memory
    UC_HOOK_MEM_FETCH_PROT = 1 << 9,
    // Hook Memory read event
    UC_HOOK_MEM_READ = 1 << 10,
    // Hook Memory write event
    UC_HOOK_MEM_WRITE = 1 << 11,
    // Hook Memory acquisition execution event
    UC_HOOK_MEM_FETCH = 1 << 12,
    // Hook Memory read events, only addresses that can be
   successfully accessed are allowed
   // A callback will be triggered after a successful read
    UC_HOOK_MEM_READ_AFTER = 1 << 13,
    // Hook Invalid instruction exception
    UC_HOOK_INSN_INVALID = 1 << 14,
```

```
} uc_hook_type;
```

hook_types

Macro definition
Hook type

▶ Code

```
// Hook All unmapped memory access events
#define UC_HOOK_MEM_UNMAPPED (UC_HOOK_MEM_READ_UNMAPPED +
UC_HOOK_MEM_WRITE_UNMAPPED + UC_HOOK_MEM_FETCH_UNMAPPED)
// Hook All illegal access events to protected memory
#define UC_HOOK_MEM_PROT (UC_HOOK_MEM_READ_PROT + UC_HOOK_MEM_WRITE_PROT +
UC_HOOK_MEM_FETCH_PROT)
// Hook All events of illegal memory reading
#define UC_HOOK_MEM_READ_INVALID (UC_HOOK_MEM_READ_PROT +
UC_HOOK_MEM_READ_UNMAPPED)
// Hook All events that are illegally written to memory
#define UC_HOOK_MEM_WRITE_INVALID (UC_HOOK_MEM_WRITE_PROT +
UC_HOOK_MEM_WRITE_UNMAPPED)
// Hook All events that illegally obtain memory
#define UC_HOOK_MEM_FETCH_INVALID (UC_HOOK_MEM_FETCH_PROT +
UC_HOOK_MEM_FETCH_UNMAPPED)
// Hook All illegal memory access events
#define UC_HOOK_MEM_INVALID (UC_HOOK_MEM_UNMAPPED + UC_HOOK_MEM_PROT)
// Hook All valid memory access events
// Note: UC_HOOK_MEM_READ in UC_HOOK_MEM_READ_PROT and UC_HOOK_MEM_READ_UNMAPPED
Triggered before,
//Therefore, this HOOK may trigger some invalid reads.
#define UC_HOOK_MEM_VALID (UC_HOOK_MEM_READ + UC_HOOK_MEM_WRITE +
UC_HOOK_MEM_FETCH)
```

uc_mem_region

By uc_mem_map() && uc_mem_map_ptr()Map memory area usage || uc_mem_regions()Retrieve a list of this memory area

▶ Code

```
typedef struct uc_mem_region {
   uint64_t begin; // Regional starting address (inclusive)
   uint64_t end;   // Area end address (inclusive)
   uint32_t perms; // Memory permissions for the area
} uc_mem_region;
```

uc_query_type

uc_query()All query type parameters

▶ Code

```
typedef enum uc_query_type {
    /Dynamically query the current hardware mode
   UC_QUERY_MODE = 1,
   UC_QUERY_PAGE_SIZE,
   UC_QUERY_ARCH,
} uc_query_type;
```

uc_context

Used with uc_context_*() to manage opaque storage of CPU context

▶ Code

```
struct uc_context;
typedef struct uc_context uc_context;
```

uc_prot

Permissions for the newly mapped area

▶ Code

```
typedef enum uc_prot {
   UC_PROT_NONE = 0,   //None
   UC_PROT_READ = 1,   //Read
   UC_PROT_WRITE = 2,  //Write
   UC_PROT_EXEC = 4,   //Executable
   UC_PROT_ALL = 7,    //All permissions
} uc_prot;
```

0x2 API

> Index

---

uc_version

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

Used to return Unicorn API major and minor version information

```
@major:API(Major.version.number)
@minor:API(Minor.version.number)
@return 16Hex Number, calculation method (major
<< 8 | minor)

Tip: The return value can be compared with the macro
UC_MAKE_VERSION
```

▶ Source code implementation

```
unsigned int uc_version(unsigned int *major, unsigned int *minor)
{
    if (major != NULL && minor != NULL) {
        *major = UC_API_MAJOR;  //宏
        *minor = UC_API_MINOR;  //宏
    }

    return (UC_API_MAJOR << 8) + UC_API_MINOR;   //(major << 8 | minor)
}
```

Cannot be changed after

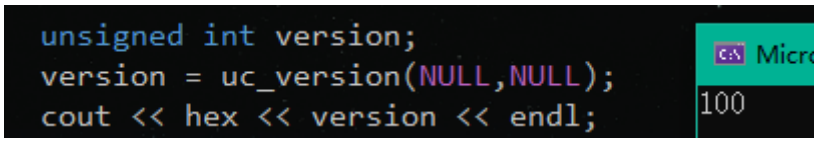
compilation, custom version

usage examples are not

accepted:

```
#include <iostream>
#include "unicorn/unicorn.h"
using namespace std;

int main()
{
    unsigned int version;
    version = uc_version(NULL,NULL);
    cout << hex << version << endl;
    return 0;
}
```

output:



Get the version number 1.0.0

uc_arch_supported

```
bool uc_arch_supported(uc_arch arch);
```

Determine whether Unicorn supports the current architecture

```
 @arch: Architecture type (UC_ARCH_*)
@return Return True if supported
```

▶ Source code implementation

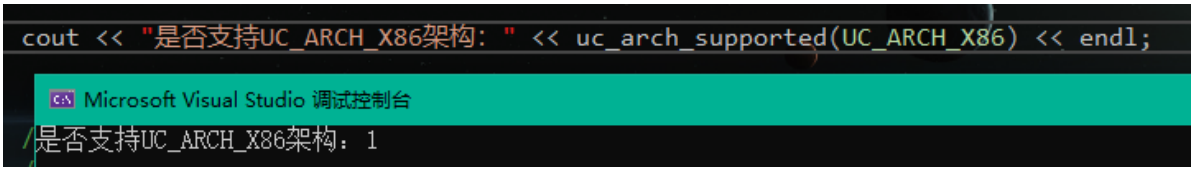
```
bool uc_arch_supported(uc_arch arch)
{
    switch (arch) {
#ifdef UNICORN_HAS_ARM
        case UC_ARCH_ARM:   return true;
#endif
#ifdef UNICORN_HAS_ARM64
        case UC_ARCH_ARM64: return true;
#endif
#ifdef UNICORN_HAS_M68K
        case UC_ARCH_M68K:  return true;
#endif
#ifdef UNICORN_HAS_MIPS
        case UC_ARCH_MIPS:  return true;
#endif
#ifdef UNICORN_HAS_PPC
        case UC_ARCH_PPC:   return true;
#endif
#ifdef UNICORN_HAS_SPARC
        case UC_ARCH_SPARC: return true;
#endif
#ifdef UNICORN_HAS_X86
```

```
        case UC_ARCH_X86:               return true;
#endif
        /* Invalid or disabled architecture */
        default:                        return false;
    }
}
```

Usage example:

```
#include <iostream>
#include "unicorn/unicorn.h"
using namespace std;

int main()
{
    cout << "Whether to support UC_ARCH_X86 architecture: " << uc_arch_supported(UC_ARCH_X86) << endl;
    return 0;
}
```

output:



uc_open

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

Create a new Unicorn instance

```
@arch: Architecture type (UC_ARCH_*)
@mode: Hardware mode. Combined by UC_MODE_*
@uc: Pointer to uc_engine, updated when returned

@return If successful, UC_ERR_OK is returned, otherwise other error
types enumerated by uc_err are returned
```

▶ Source code implementation

```
uc_err uc_open(uc_arch arch, uc_mode mode, uc_engine **result)
{
    struct uc_struct *uc;

    if (arch < UC_ARCH_MAX) {
        uc = calloc(1, sizeof(*uc));//Apply for memory
        if (!uc) {
            //Insufficient memory
            return UC_ERR_NOMEM;
        }

        uc->errnum = UC_ERR_OK;
        uc->arch = arch;
```

```
        uc->mode = mode;

        // Initialize
        // uc->ram_list = { .blocks = QTAILQ_HEAD_INITIALIZER(ram_list.blocks)
};
        uc->ram_list.blocks.tqh_first = NULL;
        uc->ram_list.blocks.tqh_last = &(uc->ram_list.blocks.tqh_first);

        uc->memory_listeners.tqh_first = NULL;
        uc->memory_listeners.tqh_last = &uc->memory_listeners.tqh_first;

        uc->address_spaces.tqh_first = NULL;
        uc->address_spaces.tqh_last = &uc->address_spaces.tqh_first;

        switch(arch) {   // Preprocessing according to architecture
            default:
                break;
#ifdef UNICORN_HAS_M68K
            case UC_ARCH_M68K:
                if ((mode & ~UC_MODE_M68K_MASK) ||
                        !(mode & UC_MODE_BIG_ENDIAN)) {
                    free(uc);
                    return UC_ERR_MODE;
                }
                uc->init_arch = m68k_uc_init;
                break;
#endif
#ifdef UNICORN_HAS_X86
            case UC_ARCH_X86:
                if ((mode & ~UC_MODE_X86_MASK) ||
                        (mode & UC_MODE_BIG_ENDIAN) ||
                        !(mode & (UC_MODE_16|UC_MODE_32|UC_MODE_64))) {
                    free(uc);
                    return UC_ERR_MODE;
                }
                uc->init_arch = x86_uc_init;
                break;
#endif
#ifdef UNICORN_HAS_ARM
            case UC_ARCH_ARM:
                if ((mode & ~UC_MODE_ARM_MASK)) {
                    free(uc);
                    return UC_ERR_MODE;
                }
                if (mode & UC_MODE_BIG_ENDIAN) {
                    uc->init_arch = armeb_uc_init;
                } else {
                    uc->init_arch = arm_uc_init;
                }

                if (mode & UC_MODE_THUMB)
                    uc->thumb = 1;
                break;
#endif
#ifdef UNICORN_HAS_ARM64
            case UC_ARCH_ARM64:
                if (mode & ~UC_MODE_ARM_MASK) {
                    free(uc);
```

```
                    return UC_ERR_MODE;
                }
                if (mode & UC_MODE_BIG_ENDIAN) {
                    uc->init_arch = arm64eb_uc_init;
                } else {
                    uc->init_arch = arm64_uc_init;
                }
                break;
#endif

#if defined(UNICORN_HAS_MIPS) || defined(UNICORN_HAS_MIPSEL) ||
defined(UNICORN_HAS_MIPS64) || defined(UNICORN_HAS_MIPS64EL)
            case UC_ARCH_MIPS:
                if ((mode & ~UC_MODE_MIPS_MASK) ||
                        !(mode & (UC_MODE_MIPS32|UC_MODE_MIPS64))) {
                    free(uc);
                    return UC_ERR_MODE;
                }
                if (mode & UC_MODE_BIG_ENDIAN) {
#ifdef UNICORN_HAS_MIPS
                    if (mode & UC_MODE_MIPS32)
                        uc->init_arch = mips_uc_init;
#endif
#ifdef UNICORN_HAS_MIPS64
                    if (mode & UC_MODE_MIPS64)
                        uc->init_arch = mips64_uc_init;
#endif
                } else {  // Little endian sequence
#ifdef UNICORN_HAS_MIPSEL
                    if (mode & UC_MODE_MIPS32)
                        uc->init_arch = mipsel_uc_init;
#endif
#ifdef UNICORN_HAS_MIPS64EL
                    if (mode & UC_MODE_MIPS64)
                        uc->init_arch = mips64el_uc_init;
#endif
                }
                break;
#endif

#ifdef UNICORN_HAS_SPARC
            case UC_ARCH_SPARC:
                if ((mode & ~UC_MODE_SPARC_MASK) ||
                        !(mode & UC_MODE_BIG_ENDIAN) ||
                        !(mode & (UC_MODE_SPARC32|UC_MODE_SPARC64))) {
                    free(uc);
                    return UC_ERR_MODE;
                }
                if (mode & UC_MODE_SPARC64)
                    uc->init_arch = sparc64_uc_init;
                else
                    uc->init_arch = sparc_uc_init;
                break;
#endif
        }

        if (uc->init_arch == NULL) {
            return UC_ERR_ARCH;
```

```
        }

        if (machine_initialize(uc))
            return UC_ERR_RESOURCE;

        *result = uc;

        if (uc->reg_reset)
            uc->reg_reset(uc);

        return UC_ERR_OK;
    } else {
        return UC_ERR_ARCH;
    }
}
```

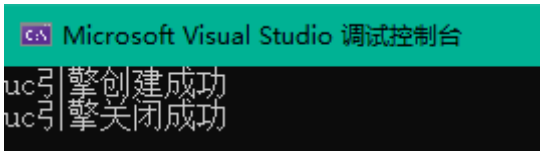
note: uc_open will apply for heap memory, and uc_close must be used

to release it after use, otherwise a leak will occur.

Usage example:

```
#include <iostream>
#include "unicorn/unicorn.h"
using namespace std;

int main()
{
    uc_engine* uc;
    uc_err err;

    //Initialize the X86-32bit mode emulator
    err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_open() with error returned: %u\n", err);
            return -1;
    }

    if (!err)
       cout << "uc engine was created successfully" << endl;

    //Close uc
    err = uc_close(uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_close() with error returned: %u\n", err);
        return -1;
    }

    if (!err)
       cout << "uc engine shut down successfully" << endl;

    return 0;
}
```

output



uc_close

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

Closing a uc instance will free up memory.Cannot be restored after shutdown.

```
@uc: Pointer to the one returned by uc_open()

@return If successful, UC_ERR_OK is returned, otherwise other error

types enumerated by uc_err are returned
```

▶ Source code implementation

```
uc_err uc_close(uc_engine *uc)
{
    int i;
    struct list_item *cur;
    struct hook *hook;

    //Clean up internal data
    if (uc->release)
        uc->release(uc->tcg_ctx);
    g_free(uc->tcg_ctx);

    //Clean up the CPU.
    g_free(uc->cpu->tcg_as_listener);
    g_free(uc->cpu->thread);

    // Own objects.
    OBJECT(uc->machine_state->accelerator)->ref = 1;
    OBJECT(uc->machine_state)->ref = 1;
    OBJECT(uc->owner)->ref = 1;
    OBJECT(uc->root)->ref = 1;

    object_unref(uc, OBJECT(uc->machine_state->accelerator));
    object_unref(uc, OBJECT(uc->machine_state));
    object_unref(uc, OBJECT(uc->cpu));
    object_unref(uc, OBJECT(&uc->io_mem_notdirty));
    object_unref(uc, OBJECT(&uc->io_mem_unassigned));
    object_unref(uc, OBJECT(&uc->io_mem_rom));
    object_unref(uc, OBJECT(uc->root));

    //Free up memory
    g_free(uc->system_memory);

    //Release related threads
    if (uc->qemu_thread_data)
        g_free(uc->qemu_thread_data);

   //Release other data
```

```
    free(uc->l1_map);

    if (uc->bounce.buffer) {
        free(uc->bounce.buffer);
    }

    g_hash_table_foreach(uc->type_table, free_table, uc);
    g_hash_table_destroy(uc->type_table);

    for (i = 0; i < DIRTY_MEMORY_NUM; i++) {
        free(uc->ram_list.dirty_memory[i]);
    }

    //Release hook and hook list
    for (i = 0; i < UC_HOOK_MAX; i++) {
        cur = uc->hook[i].head;
        //Hook can exist in multiple lists, and the release time can be obtained by counting
        while (cur) {
            hook = (struct hook *)cur->data;
            if (--hook->refs == 0) {
                free(hook);
            }
            cur = cur->next;
        }
        list_clear(&uc->hook[i]);
    }

    free(uc->mapped_blocks);

    //Finally release uc itself
    memset(uc, 0, sizeof(*uc));
    free(uc);

    return UC_ERR_OK;
}
```

The usage example is the same: uc_open()

uc_query

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

The internal state of the query engine

```
@uc: Handle returned by uc_open()

@type: The type enumerated in uc_query_type

@result: Pointer to save the internal state of the query

@return: UC_ERR_OK is returned if successful, otherwise other error

types enumerated by uc_err are returned
```

▶ Source code implementation

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

```
{
    if (type == UC_QUERY_PAGE_SIZE) {
        *result = uc->target_page_size;
        return UC_ERR_OK;
    }

    if (type == UC_QUERY_ARCH) {
        *result = uc->arch;
        return UC_ERR_OK;
    }

    switch(uc->arch) {
#ifdef UNICORN_HAS_ARM
        case UC_ARCH_ARM:
            return uc->query(uc, type, result);
#endif
        default:
            return UC_ERR_ARG;
    }

    return UC_ERR_OK;
}
```

Usage example:

```
#include <iostream>
#include "unicorn/unicorn.h"
using namespace std;
int main()
{
    uc_engine* uc;
    uc_err err;

    //// Initialize emulator in X86-32bit mode
    err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_open() with error returned: %u\n", err);
        return -1;
    }
    if (!err)
       cout << "uc instance was created successfully" << endl;

    size_t result[] = {0};
    err = uc_query(uc, UC_QUERY_ARCH, result);  //Query architecture
    if (!err)
       cout << "Query successfully: " << *result << endl;

    err = uc_close(uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_close() with error returned: %u\n", err);
        return -1;
    }
    if (!err)
       cout << "uc instance closed successfully << endl;

    return 0;
}
```

output

The result of the architecture query is 4, which corresponds to UC_ARCH_X86

uc_errno

```
uc_err uc_errno(uc_engine *uc);
```

When an API function fails, the last error number is reported. Once accessed, uc_errno may not retain its original value.

```
@uc: Handle returned by uc_open()

@return: UC_ERR_OK is returned if successful, otherwise other error types enumerated by uc_err are returned
```

▶ Source code implementation

```
uc_err uc_errno(uc_engine *uc)
{
    return uc->errnum;
}
```

Usage example:

```
#include <iostream>
#include "unicorn/unicorn.h"
using namespace std;

int main()
{
    uc_engine* uc;
    uc_err err;

    err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_open() with error returned: %u\n", err);
        return -1;
    }
    if (!err)
       cout << "uc instance was created successfully" << endl;

    err = uc_errno(uc);
    cout << "Error number:  " << err << endl;

    err = uc_close(uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_close() with error returned: %u\n", err);
        return -1;
    }
```

```
    if (!err)
        cout << "uc instance closed successfully" << endl;

    return 0;
}
```

output

No error, the output error number is 0

uc_strerror

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

Returns an explanation of the given error number

```
@code: error number

@return: A string pointer to the interpretation of the given error number
```

▶ Source code implementation

```
const char *uc_strerror(uc_err code)
{
    switch(code) {
        default:
            return "Unknown error code";
        case UC_ERR_OK:
            return "OK (UC_ERR_OK)";
        case UC_ERR_NOMEM:
            return "No memory available or memory not present (UC_ERR_NOMEM)";
        case UC_ERR_ARCH:
            return "Invalid/unsupported architecture (UC_ERR_ARCH)";
        case UC_ERR_HANDLE:
            return "Invalid handle (UC_ERR_HANDLE)";
        case UC_ERR_MODE:
            return "Invalid mode (UC_ERR_MODE)";
        case UC_ERR_VERSION:
            return "Different API version between core & binding
(UC_ERR_VERSION)";
        case UC_ERR_READ_UNMAPPED:
            return "Invalid memory read (UC_ERR_READ_UNMAPPED)";
        case UC_ERR_WRITE_UNMAPPED:
            return "Invalid memory write (UC_ERR_WRITE_UNMAPPED)";
        case UC_ERR_FETCH_UNMAPPED:
            return "Invalid memory fetch (UC_ERR_FETCH_UNMAPPED)";
        case UC_ERR_HOOK:
            return "Invalid hook type (UC_ERR_HOOK)";
```

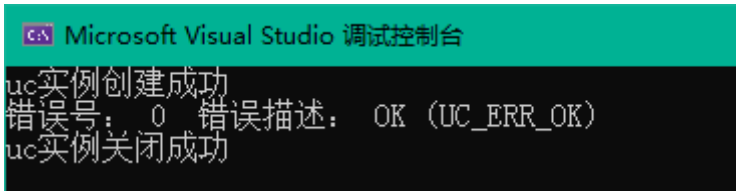
```
        case UC_ERR_INSN_INVALID:
            return "Invalid instruction (UC_ERR_INSN_INVALID)";
        case UC_ERR_MAP:
            return "Invalid memory mapping (UC_ERR_MAP)";
        case UC_ERR_WRITE_PROT:
            return "Write to write-protected memory (UC_ERR_WRITE_PROT)";
        case UC_ERR_READ_PROT:
            return "Read from non-readable memory (UC_ERR_READ_PROT)";
        case UC_ERR_FETCH_PROT:
            return "Fetch from non-executable memory (UC_ERR_FETCH_PROT)";
        case UC_ERR_ARG:
            return "Invalid argument (UC_ERR_ARG)";
        case UC_ERR_READ_UNALIGNED:
            return "Read from unaligned memory (UC_ERR_READ_UNALIGNED)";
        case UC_ERR_WRITE_UNALIGNED:
            return "Write to unaligned memory (UC_ERR_WRITE_UNALIGNED)";
        case UC_ERR_FETCH_UNALIGNED:
            return "Fetch from unaligned memory (UC_ERR_FETCH_UNALIGNED)";
        case UC_ERR_RESOURCE:
            return "Insufficient resource (UC_ERR_RESOURCE)";
        case UC_ERR_EXCEPTION:
            return "Unhandled CPU exception (UC_ERR_EXCEPTION)";
        case UC_ERR_TIMEOUT:
            return "Emulation timed out (UC_ERR_TIMEOUT)";
    }
}
```

Usage example:

```
#include <iostream>
#include "unicorn/unicorn.h"
using namespace std;

int main()
{
    uc_engine* uc;
    uc_err err;

    err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_open() with error returned: %u\n", err);
        return -1;
    }
    if (!err)
       cout << "uc instance was created successfully" << endl;

    err = uc_errno(uc);
    cout << "Error number:  " << err << "Error description:  " << uc_strerror(err) <<endl;

    err = uc_close(uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_close() with error returned: %u\n", err);
        return -1;
    }
    if (!err)
       cout << "uc instance closed successfully" << endl;
```

```
    return 0;
}
```

output



uc_reg_write

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

Write value to register

```
@uc: Handle returned by uc_open()
@regid: The register ID to be modified
@value: A pointer to the value that the register will be modified to
@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err
are returned
```

▶ Source code implementation

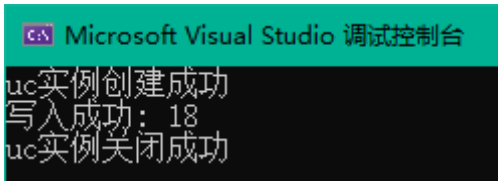
```
uc_err uc_reg_write(uc_engine *uc, int regid, const void *value)
{
    return uc_reg_write_batch(uc, &regid, (void *const *)&value, 1);
}

uc_err uc_reg_write_batch(uc_engine *uc, int *ids, void *const *vals, int count)
{
    int ret = UC_ERR_OK;
    if (uc->reg_write)
        ret = uc->reg_write(uc, (unsigned int *)ids, vals, count);//Write in the structure
入
    else
        return UC_ERR_EXCEPTION;

    return ret;
}
```

Usage example:

```
#include <iostream>
#include "unicorn/unicorn.h"
using namespace std;

int main()
{
    uc_engine* uc;
    uc_err err;
```

```
    err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_open() with error returned: %u\n", err);
        return -1;
    }
    if (!err)
       cout << "uc instance was created successfully" << endl;

    int r_eax = 0x12;
    err = uc_reg_write(uc, UC_X86_REG_ECX, &r_eax);
    if (!err)
       cout << "Write successfully: " << r_eax << endl;

    err = uc_close(uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_close() with error returned: %u\n", err);
        return -1;
    }
    if (!err)
       cout << "uc instance closed successfully" << endl;

    return 0;
}
```

output



uc_reg_read

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

Read the value of the register

```
@uc: Handle returned by uc_open()
@regid: The register ID to be read
@value: pointer to save register value
@return If successful, UC_ERR_OK is returned, otherwise other error
types enumerated by uc_err are returned
```

▶ Source code implementation

```
uc_err uc_reg_read(uc_engine *uc, int regid, void *value)
{
    return uc_reg_read_batch(uc, &regid, &value, 1);
}

uc_err uc_reg_read_batch(uc_engine *uc, int *ids, void **vals, int count)
{
    if (uc->reg_read)
```

```
        uc->reg_read(uc, (unsigned int *)ids, vals, count);
    else
        return -1;

    return UC_ERR_OK;
}
```

Usage example:

```
#include <iostream>
#include "unicorn/unicorn.h"
using namespace std;

int main()
{
    uc_engine* uc;
    uc_err err;

    err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_open() with error returned: %u\n", err);
        return -1;
    }
    if (!err)
       cout << "uc instance was created successfully" << endl;

    int r_eax = 0x12;
    err = uc_reg_write(uc, UC_X86_REG_ECX, &r_eax);
    if (!err)
       cout << "Write successfully: " << r_eax << endl;

    int recv_eax;
    err = uc_reg_read(uc, UC_X86_REG_ECX, &recv_eax);
    if (!err)
        cout << "读取成功: " << recv_eax << endl;

    err = uc_close(uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_close() with error returned: %u\n", err);
        return -1;
    }
    if (!err)
       cout << "uc instance closed successfully" << endl;

    return 0;
}
```

output

uc_reg_write_batch

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

Write multiple values to multiple registers at the same time

```
@uc: Handle returned by uc_open()
@regid: An array that stores multiple register IDs to be written
@value: pointer to an array holding multiple values
@count: The length of the *regs and *vals arrays
@return If successful, UC_ERR_OK is returned, otherwise other error
types enumerated by uc_err are returned
```

▶ Source code implementation

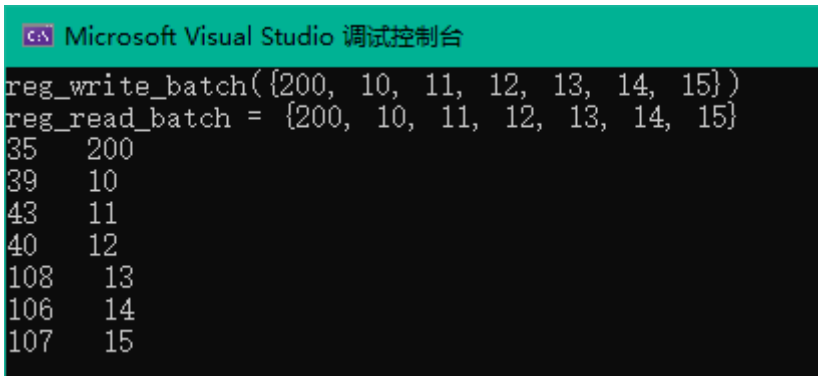
```
uc_err uc_reg_write_batch(uc_engine *uc, int *ids, void *const *vals, int count)
{
    int ret = UC_ERR_OK;
    if (uc->reg_write)
        ret = uc->reg_write(uc, (unsigned int *)ids, vals, count);
    else
        return UC_ERR_EXCEPTION;

    return ret;
}
```

Usage example:

```
#include <iostream>
#include <string>
#include "unicorn/unicorn.h"
using namespace std;

int syscall_abi[] = {
    UC_X86_REG_RAX, UC_X86_REG_RDI, UC_X86_REG_RSI, UC_X86_REG_RDX,
    UC_X86_REG_R10, UC_X86_REG_R8, UC_X86_REG_R9
};

uint64_t vals[7] = { 200, 10, 11, 12, 13, 14, 15 };

void* ptrs[7];

int main()
{
    int i;
    uc_err err;
    uc_engine* uc;

    // set up register pointers
    for (i = 0; i < 7; i++) {
        ptrs[i] = &vals[i];
    }

    if ((err = uc_open(UC_ARCH_X86, UC_MODE_64, &uc))) {
```

```
        uc_perror("uc_open", err);
        return 1;
    }

    // reg_write_batch
    printf("reg_write_batch({200, 10, 11, 12, 13, 14, 15})\n");
    if ((err = uc_reg_write_batch(uc, syscall_abi, ptrs, 7))) {
        uc_perror("uc_reg_write_batch", err);
        return 1;
    }

    // reg_read_batch
    memset(vals, 0, sizeof(vals));
    if ((err = uc_reg_read_batch(uc, syscall_abi, ptrs, 7))) {
        uc_perror("uc_reg_read_batch", err);
        return 1;
    }

    printf("reg_read_batch = {");

    for (i = 0; i < 7; i++) {
        if (i != 0) printf(", ");
        printf("%" PRIu64, vals[i]);
    }

    printf("}\n");

    uint64_t var[7] = { 0 };
    for (int i = 0; i < 7; i++)
    {
        cout << syscall_abi[i] << "    ";
        printf("%" PRIu64, vals[i]);
        cout << endl;
    }

    return 0;
}
```

output



uc_reg_read_batch

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

Read the values of multiple registers at the same time.

```
@uc: Handle returned by uc_open()
@regid: An array that stores multiple register IDs to be read
@value: pointer to an array holding multiple values
@count: The length of the *regs and *vals arrays
@return If successful, UC_ERR_OK is returned, otherwise other error
types enumerated by uc_err are returned
```

▶ Source code implementation

```
uc_err uc_reg_read_batch(uc_engine *uc, int *ids, void **vals, int count)
{
    if (uc->reg_read)
        uc->reg_read(uc, (unsigned int *)ids, vals, count);
    else
        return -1;

    return UC_ERR_OK;
}
```

The usage example is the same as uc_reg_write_batch().

uc_mem_write

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

Write a piece of bytecode in memory.

```
@uc: Handle returned by uc_open()
@address: The starting address of the byte written
@bytes:  A pointer to a pointer containing data to be written to memory.
@size:  The size of memory to be written to.

Note: @bytes must be large enough to contain @size bytes.

@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err are returned
```

▶ Source code implementation

```
uc_err uc_mem_write(uc_engine *uc, uint64_t address, const void *_bytes, size_t
size)
{
    size_t count = 0, len;
    const uint8_t *bytes = _bytes;

    if (uc->mem_redirect) {
```

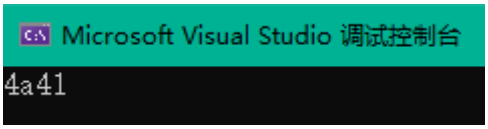
```
        address = uc->mem_redirect(address);
    }

    if (!check_mem_area(uc, address, size))
        return UC_ERR_WRITE_UNMAPPED;

    //Memory areas can overlap adjacent memory blocks
    while(count < size) {
        MemoryRegion *mr = memory_mapping(uc, address);
        if (mr) {
            uint32_t operms = mr->perms;
            if (!(operms & UC_PROT_WRITE))// No write protection
                // Marked as writable
                uc->readonly_mem(mr, false);

            len = (size_t)MIN(size - count, mr->end - address);
            if (uc->write_mem(&uc->as, address, bytes, len) == false)
                break;

           if (!(operms & UC_PROT_WRITE)) // No write protection
                // Set write protection
                uc->readonly_mem(mr, true);

            count += len;
            address += len;
            bytes += len;
        } else // This address has not been mapped yet
            break;
    }

    if (count == size)
        return UC_ERR_OK;
    else
        return UC_ERR_WRITE_UNMAPPED;
}
```

Usage example:

```
#include <iostream>
#include <string>
#include "unicorn/unicorn.h"
using namespace std;

#define X86_CODE32 "\x41\x4a" // INC ecx; DEC edx
#define ADDRESS 0x1000

int main()
{
    uc_engine* uc;
    uc_err err;

    err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_open() with error returned: %u\n", err);
        return -1;
    }
```

```
    uc_mem_map(uc, ADDRESS, 2 * 1024 * 1024, UC_PROT_ALL);

    if (uc_mem_write(uc, ADDRESS, X86_CODE32, sizeof(X86_CODE32) - 1)) {
        printf("Failed to write emulation code to memory, quit!\n");
        return -1;
    }

    uint32_t code;

    if(uc_mem_read(uc,ADDRESS,&code, sizeof(code))) {
        printf("Failed to read emulation code to memory, quit!\n");
        return -1;
    }

    cout << hex << code << endl;

    err = uc_close(uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_close() with error returned: %u\n", err);
        return -1;
    }
    return 0;
}
```

输出



uc_mem_read

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

Read bytes from memory.

```
@uc: Handle returned by uc_open()
@address: The starting address of the read byte
 @bytes:  A pointer to a pointer containing the data to be
 @size:  read in memory, the size of the memory to be read.

Note: @bytes must be large enough to contain @size bytes.

@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err are returned
```

▶ Source code implementation

```
uc_err uc_mem_read(uc_engine *uc, uint64_t address, void *_bytes, size_t size)
{
    size_t count = 0, len;
    uint8_t *bytes = _bytes;

    if (uc->mem_redirect) {
        address = uc->mem_redirect(address);
```

```
    }

    if (!check_mem_area(uc, address, size))
        return UC_ERR_READ_UNMAPPED;

    //Memory areas can overlap adjacent memory blocks
    while(count < size) {
        MemoryRegion *mr = memory_mapping(uc, address);
        if (mr) {
            len = (size_t)MIN(size - count, mr->end - address);
            if (uc->read_mem(&uc->as, address, bytes, len) == false)
                break;
            count += len;
            address += len;
            bytes += len;
        } else  //This address has not been mapped yet
            break;
    }

    if (count == size)
        return UC_ERR_OK;
    else
        return UC_ERR_READ_UNMAPPED;
}
```

The usage example is the same uc_mem_write()

uc_emu_start

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

Simulate the machine code within the specified time.

```
@uc: Handle returned by uc_open()
@begin: The address to start the simulation
@until: The address where the simulation stopped (when the address is reached)
@timeout: The duration of the simulation code (in microseconds).When this value is 0, there will be no
time limit to simulate the code until the simulation is completed.
@count: The number of instructions to be simulated.When this value is 0, all executable code will be
simulated until the simulation is complete
@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err are
returned
```

▶ Source code implementation

```
uc_err uc_emu_start(uc_engine* uc, uint64_t begin, uint64_t until, uint64_t
timeout, size_t count)
{
    //Remake counter
    uc->emu_counter = 0;
    uc->invalid_error = UC_ERR_OK;
    uc->block_full = false;
    uc->emulation_done = false;
    uc->timed_out = false;
```

```
    switch(uc->arch) {
        default:
            break;
#ifdef UNICORN_HAS_M68K
        case UC_ARCH_M68K:
            uc_reg_write(uc, UC_M68K_REG_PC, &begin);
            break;
#endif
#ifdef UNICORN_HAS_X86
        case UC_ARCH_X86:
            switch(uc->mode) {
                default:
                    break;
                case UC_MODE_16: {
                    uint64_t ip;
                    uint16_t cs;

                    uc_reg_read(uc, UC_X86_REG_CS, &cs);
                    //Offset the added IP and CS later
                    ip = begin - cs*16;
                    uc_reg_write(uc, UC_X86_REG_IP, &ip);
                    break;
                }
                case UC_MODE_32:
                    uc_reg_write(uc, UC_X86_REG_EIP, &begin);
                    break;
                case UC_MODE_64:
                    uc_reg_write(uc, UC_X86_REG_RIP, &begin);
                    break;
            }
            break;
#endif
#ifdef UNICORN_HAS_ARM
        case UC_ARCH_ARM:
            uc_reg_write(uc, UC_ARM_REG_R15, &begin);
            break;
#endif
#ifdef UNICORN_HAS_ARM64
        case UC_ARCH_ARM64:
            uc_reg_write(uc, UC_ARM64_REG_PC, &begin);
            break;
#endif
#ifdef UNICORN_HAS_MIPS
        case UC_ARCH_MIPS:
            // TODO: MIPS32/MIPS64/BIGENDIAN etc
            uc_reg_write(uc, UC_MIPS_REG_PC, &begin);
            break;
#endif
#ifdef UNICORN_HAS_SPARC
        case UC_ARCH_SPARC:
            // TODO: Sparc/Sparc64
            uc_reg_write(uc, UC_SPARC_REG_PC, &begin);
            break;
#endif
    }

    uc->stop_request = false;
```

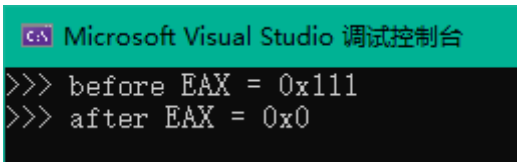
```
    uc->emu_count = count;
    // If counting is not required, remove the counting hook hook
    if (count <= 0 && uc->count_hook != 0) {
        uc_hook_del(uc, uc->count_hook);
        uc->count_hook = 0;
    }
    //Set the count hook to record the number of instructions
    if (count > 0 && uc->count_hook == 0) {
        uc_err err;
  //The callback to the counting instruction must be run before all other operations, so
the hook must be inserted at the beginning of the hook list instead of appending the hook
        uc->hook_insert = 1;
        err = uc_hook_add(uc, &uc->count_hook, UC_HOOK_CODE, hook_count_cb,
NULL, 1, 0);
        //Revert to uc_hook_add()
        uc->hook_insert = 0;  if
       (err != UC_ERR_OK) {
            return err;
        }
    }

    uc->addr_end = until;

    if (timeout)
        enable_emu_timer(uc, timeout * 1000);   // microseconds -> nanoseconds

    if (uc->vm_start(uc)) {
        return UC_ERR_RESOURCE;
    }

    //Simulation completed
    uc->emulation_done = true;

    if (timeout) {
        //Waiting for timeout
        qemu_thread_join(&uc->timer);
    }

    if(uc->timed_out)
        return UC_ERR_TIMEOUT;

    return uc->invalid_error;
}
```

Usage example:

```
#include <iostream>
#include <string>
#include "unicorn/unicorn.h"
using namespace std;

#define X86_CODE32 "\x33\xC0" // xor  eax, eax
#define ADDRESS 0x1000

int main()
{
    uc_engine* uc;
```

```
    uc_err err;

    int r_eax = 0x111;

    err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_open() with error returned: %u\n", err);
        return -1;
    }

    uc_mem_map(uc, ADDRESS, 2 * 1024 * 1024, UC_PROT_ALL);

    if (uc_mem_write(uc, ADDRESS, X86_CODE32, sizeof(X86_CODE32) - 1)) {
        printf("Failed to write emulation code to memory, quit!\n");
        return -1;
    }

    uc_reg_write(uc, UC_X86_REG_EAX, &r_eax);
    printf(">>> before EAX = 0x%x\n", r_eax);

    err = uc_emu_start(uc, ADDRESS, ADDRESS + sizeof(X86_CODE32) - 1, 0, 0);
    if (err) {
        printf("Failed on uc_emu_start() with error returned %u: %s\n",
        err, uc_strerror(err));
    }

    uc_reg_read(uc, UC_X86_REG_EAX, &r_eax);
    printf(">>> after EAX = 0x%x\n", r_eax);

    err = uc_close(uc);
    if (err != UC_ERR_OK) {
        printf("Failed on uc_close() with error returned: %u\n", err);
        return -1;
    }

    return 0;
}
```

output



uc_emu_stop

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

Stop simulation

It is usually called from a callback function

registered through the tracing API.

```
@uc: Handle returned by uc_open()

@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err are returned
```

▶ Source code implementation

```
uc_err uc_emu_stop(uc_engine *uc)
{
    if (uc->emulation_done)
        return UC_ERR_OK;

    uc->stop_request = true;

    if (uc->current_cpu) {
        /Exit the current thread
        cpu_exit(uc->current_cpu);
    }

    return UC_ERR_OK;
}
```

Usage example:

```
uc_emu_stop(uc);
```

uc_hook_add

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

Register a callback for the hook event, and a callback will be made when the hook event is triggered.

```
@uc: Handle returned by uc_open()
@hh: The handle obtained by registering hook. Used in uc_hook_del()
@type: hook type
@callback: The callback to be run when the instruction is hit
@user_data: User-defined data. The last parameter to be passed to the callback function
@user_data @begin: the starting address of the effective area of the callback (including)
@end: The end address of the effective area of the callback (inclusive)
Note 1: The callback will only be called if the address of the callback is in [@begin, @end]
Note 2: If @begin > @end, the callback will be called whenever this hook type is triggered
@...: Variable parameter (depends on @type)
 Note: if @type = UC_HOOK_INSN, here is the instruction ID (e.g. UC_X86_INS_OUT)

@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err are returned
```

▶ Source code implementation

```
uc_err uc_hook_add(uc_engine *uc, uc_hook *hh, int type, void *callback,
        void *user_data, uint64_t begin, uint64_t end, ...)
{
    int ret = UC_ERR_OK;
```

```
    int i = 0;

    struct hook *hook = calloc(1, sizeof(struct hook));
    if (hook == NULL) {

        return UC_ERR_NOMEM;
    }

    hook->begin = begin;
    hook->end = end;
    hook->type = type;
    hook->callback = callback;
    hook->user_data = user_data;  hook->refs = 0;
   *hh = (uc_hook)hook;

    //UC_HOOK_INSN has an additional parameter: instruction ID
    if (type & UC_HOOK_INSN) {
        va_list valist;

        va_start(valist, end);
        hook->insn = va_arg(valist, int);
        va_end(valist);

        if (uc->insn_hook_validate) {
            if (! uc->insn_hook_validate(hook->insn)) {
                free(hook);
                return UC_ERR_HOOK;
            }
        }

        if (uc->hook_insert) {
            if (list_insert(&uc->hook[UC_HOOK_INSN_IDX], hook) == NULL) {
                free(hook);
                return UC_ERR_NOMEM;
            }
        } else {
            if (list_append(&uc->hook[UC_HOOK_INSN_IDX], hook) == NULL) {
                free(hook);
                return UC_ERR_NOMEM;
            }
        }

        hook->refs++;
        return UC_ERR_OK;
    }

    while ((type >> i) > 0) {
        if ((type >> i) & 1) {
            if (i < UC_HOOK_MAX) {
                if (uc->hook_insert) {
                    if (list_insert(&uc->hook[i], hook) == NULL) {
                        if (hook->refs == 0) {
                            free(hook);
                        }
                        return UC_ERR_NOMEM;
                    }
                } else {
                    if (list_append(&uc->hook[i], hook) == NULL) {
```

```
                        if (hook->refs == 0) {
                            free(hook);
                        }
                        return UC_ERR_NOMEM;
                    }
                }
                hook->refs++;
            }
        }
        i++;
    }

    if (hook->refs == 0) {
        free(hook);
    }

    return ret;
}
```

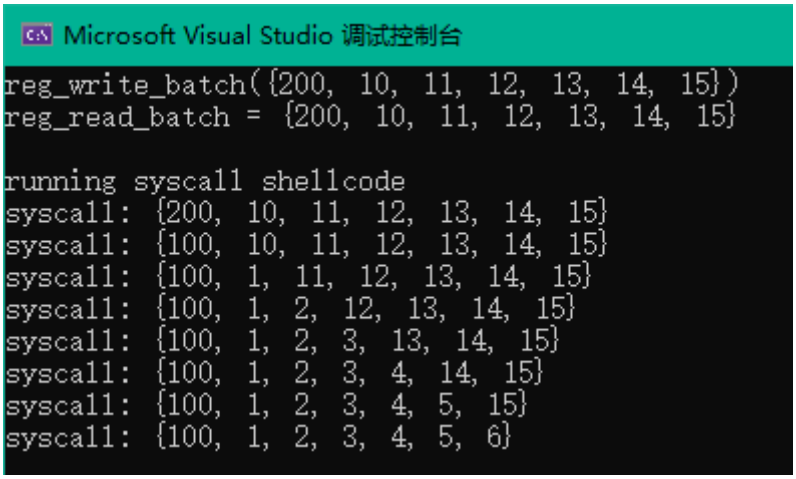
Usage example:

```
#include <iostream>
#include <string>
#include "unicorn/unicorn.h"
using namespace std;

int syscall_abi[] = {
    UC_X86_REG_RAX, UC_X86_REG_RDI, UC_X86_REG_RSI, UC_X86_REG_RDX,
    UC_X86_REG_R10, UC_X86_REG_R8, UC_X86_REG_R9
};

uint64_t vals[7] = { 200, 10, 11, 12, 13, 14, 15 };

void* ptrs[7];

void uc_perror(const char* func, uc_err err)
{
    fprintf(stderr, "Error in %s(): %s\n", func, uc_strerror(err));
}

#define BASE 0x10000

// mov rax, 100; mov rdi, 1; mov rsi, 2; mov rdx, 3; mov r10, 4; mov r8, 5; mov
r9, 6; syscall
#define CODE
"\x48\xc7\xc0\x64\x00\x00\x00\x48\xc7\xc7\x01\x00\x00\x00\x48\xc7\xc6\x02\x00\x0
0\x00\x48\xc7\xc2\x03\x00\x00\x00\x49\xc7\xc2\x04\x00\x00\x00\x49\xc7\xc0\x05\x0
0\x00\x00\x49\xc7\xc1\x06\x00\x00\x00\x0f\x05"

void hook_syscall(uc_engine* uc, void* user_data)
{
    int i;

    uc_reg_read_batch(uc, syscall_abi, ptrs, 7);

    printf("syscall: {");
```

```
    for (i = 0; i < 7; i++) {
        if (i != 0) printf(", ");
        printf("%" PRIu64, vals[i]);
    }

    printf("}\n");
}

void hook_code(uc_engine* uc, uint64_t addr, uint32_t size, void* user_data)
{
    printf("HOOK_CODE: 0x%" PRIx64 ", 0x%x\n", addr, size);
}

int main()
{
    int i;
    uc_hook sys_hook;
    uc_err err;
    uc_engine* uc;

    for (i = 0; i < 7; i++) {
        ptrs[i] = &vals[i];
    }

    if ((err = uc_open(UC_ARCH_X86, UC_MODE_64, &uc))) {
        uc_perror("uc_open", err);
        return 1;
    }

    printf("reg_write_batch({200, 10, 11, 12, 13, 14, 15})\n");
    if ((err = uc_reg_write_batch(uc, syscall_abi, ptrs, 7))) {
        uc_perror("uc_reg_write_batch", err);
        return 1;
    }

    memset(vals, 0, sizeof(vals));
    if ((err = uc_reg_read_batch(uc, syscall_abi, ptrs, 7))) {
        uc_perror("uc_reg_read_batch", err);
        return 1;
    }

    printf("reg_read_batch = {");

    for (i = 0; i < 7; i++) {
        if (i != 0) printf(", ");
        printf("%" PRIu64, vals[i]);
    }

    printf("}\n");

    // syscall
    printf("\n");
    printf("running syscall shellcode\n");

    if ((err = uc_hook_add(uc, &sys_hook, UC_HOOK_CODE, hook_syscall, NULL, 1,
0))) {
        uc_perror("uc_hook_add", err);
        return 1;
```

```
    }

    if ((err = uc_mem_map(uc, BASE, 0x1000, UC_PROT_ALL))) {
        uc_perror("uc_mem_map", err);
        return 1;
    }

    if ((err = uc_mem_write(uc, BASE, CODE, sizeof(CODE) - 1))) {
        uc_perror("uc_mem_write", err);
        return 1;
    }

    if ((err = uc_emu_start(uc, BASE, BASE + sizeof(CODE) - 1, 0, 0))) {
        uc_perror("uc_emu_start", err);
        return 1;
    }

    return 0;
}
```

output



Hook each instruction

uc_hook_del

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

Delete a registered hook event

```
@uc: Handle returned by uc_open()
@hh: Handle returned by uc_hook_add()
@return If successful, UC_ERR_OK is returned, otherwise other error
types enumerated by uc_err are returned
```

▶ Source code implementation

```
uc_err uc_hook_del(uc_engine *uc, uc_hook hh)
{
    int i;
```

```
    struct hook *hook = (struct hook *)hh;

    for (i = 0; i < UC_HOOK_MAX; i++) {
        if (list_remove(&uc->hook[i], (void *)hook)) {
            if (--hook->refs == 0) {
                free(hook);
                break;
            }
        }
    }
    return UC_ERR_OK;
}
```

Usage example:

```
if ((err = uc_hook_add(uc, &sys_hook, UC_HOOK_CODE, hook_syscall, NULL, 1, 0)))
{
    uc_perror("uc_hook_add", err);
    return 1;
}

if ((err = uc_hook_del(uc, &sys_hook))) {
    uc_perror("uc_hook_del", err);
    return 1;
}
```

uc_mem_map

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

Map a piece of memory for the simulation.

```
@uc: Handle returned by uc_open()
@address: The starting address of the new memory area to be mapped to.This address must be
aligned with 4KB, otherwise a UC_ERR_ARG error will be returned. @size: The size of the new
memory area to be mapped to.This size must be a multiple of 4KB, otherwise a UC_ERR_ARG error
will be returned.
@perms: Permissions for the newly mapped area.The parameter must be UC_PROT_READ | UC_PROT_WRITE
| UC_PROT_EXEC or a combination of these, otherwise a UC_ERR_ARG error is returned.
@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err
are returned
```

▶ Source code implementation

```
uc_err uc_mem_map(uc_engine *uc, uint64_t address, size_t size, uint32_t perms){

    uc_err res;

    if (uc->mem_redirect) {
        address = uc->mem_redirect(address);
    }

    res = mem_map_check(uc, address, size, perms);     //Memory security check
    if (res)
        return res;
```

```
    return mem_map(uc, address, size, perms, uc->memory_map(uc, address, size,
perms));
}
```

The usage example is the same uc_hook_add()

uc_mem_map_ptr

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

Map existing host memory in the simulation.

```
@uc: Handle returned by uc_open()
@address: The starting address of the new memory area to be mapped to.This address must be
aligned with 4KB, otherwise a UC_ERR_ARG error will be returned. @size: The size of the new
memory area to be mapped to.This size must be a multiple of 4KB, otherwise a UC_ERR_ARG error
will be returned.
@perms: Permissions for the newly mapped area.The parameter must be UC_PROT_READ | UC_PROT_WRITE
| UC_PROT_EXEC or a combination of these, otherwise a UC_ERR_ARG error is returned.
@ptr: A pointer to the host memory that supports newly mapped memory.The size of the mapped host
memory should be the same or larger as the size of size, and at least use PROT_READ | PROT_WRITE
for mapping, otherwise the mapping is not defined.
@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err
are returned
```

▶ Source code implementation

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

    if (ptr == NULL)
        return UC_ERR_ARG;

    if (uc->mem_redirect) {
        address = uc->mem_redirect(address);
    }

    res = mem_map_check(uc, address, size, perms);    //Memory security check
    if (res)
        return res;

    return mem_map(uc, address, size, UC_PROT_ALL, uc->memory_map_ptr(uc,
address, size, perms, ptr));
}
```

The usage example is the same uc_mem_map()

uc_mem_unmap

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

Unmap the simulated memory area

```
@uc: Handle returned by uc_open()
@address: The starting address of the new memory area to be mapped to.This address must be
aligned with 4KB, otherwise a UC_ERR_ARG error will be returned. @size: The size of the new
memory area to be mapped to.This size must be a multiple of 4KB, otherwise a UC_ERR_ARG error
will be returned.
@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err
are returned
```

▶ Source code implementation

```
uc_err uc_mem_unmap(struct uc_struct *uc, uint64_t address, size_t size)
{
    MemoryRegion *mr;
    uint64_t addr;
    size_t count, len;

    if (size == 0)
        / There is no area to unmap
        return UC_ERR_OK;

    / The address must be aligned to uc->target_page_size
    if ((address & uc->target_page_align) != 0)
        return UC_ERR_ARG;

    The size must be a multiple of uc->target_page_size  if
   ((size & uc->target_page_align) != 0)
        return UC_ERR_ARG;

    if (uc->mem_redirect) {
        address = uc->mem_redirect(address);  }

    // Check whether the entire block requested by the user is mapped
    if (!check_mem_area(uc, address, size))

        return UC_ERR_NOMEM;

    // If this area spans an adjacent area, you may need to divide the area
    addr = address;
    count = 0;
    while(count < size) {
        mr = memory_mapping(uc, addr);
        len = (size_t)MIN(size - count, mr->end - addr);
        if (!split_region(uc, mr, addr, len, true))
            return UC_ERR_NOMEM;

       // Unmap
        mr = memory_mapping(uc, addr);
        if (mr != NULL)
           uc->memory_unmap(uc, mr);
        count += len;
        addr += len;
```

```
    }

    return UC_ERR_OK;
}
```

Usage example:

```
if ((err = uc_mem_map(uc, BASE, 0x1000, UC_PROT_ALL))) {
    uc_perror("uc_mem_map", err);
    return 1;
}

if ((err = uc_mem_unmap(uc, BASE, 0x1000))) {
    uc_perror("uc_mem_unmap", err);
    return 1;
}
```

uc_mem_protect

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

Set permissions to simulate memory

```
@uc: Handle returned by uc_open()
@address: The starting address of the new memory area to be mapped to. This address must be
aligned with 4KB, otherwise a UC_ERR_ARG error will be returned. @size: The size of the new
memory area to be mapped to. This size must be a multiple of 4KB, otherwise a UC_ERR_ARG error
will be returned.
@perms: New permissions for the mapped area. The parameter must be UC_PROT_READ | UC_PROT_WRITE |
UC_PROT_EXEC or a combination of these, otherwise a UC_ERR_ARG error is returned.
@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err
are returned
```

▶ Source code implementation

```
uc_err uc_mem_protect(struct uc_struct *uc, uint64_t address, size_t size,
uint32_t perms)
{
    MemoryRegion *mr;
    uint64_t addr = address;
    size_t count, len;
    bool remove_exec = false;

    if (size == 0)
        // trivial case, no change
        return UC_ERR_OK;

    // address must be aligned to uc->target_page_size
    if ((address & uc->target_page_align) != 0)
        return UC_ERR_ARG;

    // size must be multiple of uc->target_page_size
    if ((size & uc->target_page_align) != 0)
        return UC_ERR_ARG;
```

```
    // check for only valid permissions
    if ((perms & ~UC_PROT_ALL) != 0)
        return UC_ERR_ARG;

    if (uc->mem_redirect) {
        address = uc->mem_redirect(address);
    }

    // check that user's entire requested block is mapped
    if (!check_mem_area(uc, address, size))
        return UC_ERR_NOMEM;

    // Now we know entire region is mapped, so change permissions
    // We may need to split regions if this area spans adjacent regions
    addr = address;
    count = 0;
    while(count < size) {
        mr = memory_mapping(uc, addr);
        len = (size_t)MIN(size - count, mr->end - addr);
        if (!split_region(uc, mr, addr, len, false))
            return UC_ERR_NOMEM;

        mr = memory_mapping(uc, addr);
        // will this remove EXEC permission?
        if (((mr->perms & UC_PROT_EXEC) != 0) && ((perms & UC_PROT_EXEC) == 0))
            remove_exec = true;
        mr->perms = perms;
        uc->readonly_mem(mr, (perms & UC_PROT_WRITE) == 0);

        count += len;
        addr += len;
    }

    // if EXEC permission is removed, then quit TB and continue at the same
place
    if (remove_exec) {
        uc->quit_request = true;
        uc_emu_stop(uc);
    }

    return UC_ERR_OK;
}
```

Usage example:

```
if ((err = uc_mem_protect(uc, BASE, 0x1000, UC_PROT_ALL))) {//Readable, writable, executable
    uc_perror("uc_mem_protect", err);
    return 1;
}
```

uc_mem_regions

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

Retrieve information about the memory mapped by uc_mem_map() and uc_mem_map_ptr().

This API allocates memory for @regions, and users must then release this memory through free() to avoid memory leaks.

```
@uc: Handle returned by uc_open()
@regions: A pointer to an array of uc_mem_region structures. Applied by Unicorn, these memory
must be released through uc_free()
@count: Pointer to the number of uc_mem_region structures contained in @regions
@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err
are returned
```

Source code analysis

▶ Code

```
uint32_t uc_mem_regions(uc_engine *uc, uc_mem_region **regions, uint32_t *count)
{
    uint32_t i;
    uc_mem_region *r = NULL;

    *count = uc->mapped_block_count;

    if (*count) {
        r = g_malloc0(*count * sizeof(uc_mem_region));
        if (r == NULL) {
            // Insufficient memory
            return UC_ERR_NOMEM;
        }
    }

    for (i = 0; i < *count; i++) {
        r[i].begin = uc->mapped_blocks[i]->addr;
        r[i].end = uc->mapped_blocks[i]->end - 1;
        r[i].perms = uc->mapped_blocks[i]->perms;
    }

    *regions = r;

    return UC_ERR_OK;
}
```

Usage example:

```
#include <iostream>
#include <string>
#include "unicorn/unicorn.h"
using namespace std;

int main()
{
    uc_err err;
```

```
    uc_engine* uc;

    if ((err = uc_open(UC_ARCH_X86, UC_MODE_64, &uc))) {
        uc_perror("uc_open", err);
        return 1;
    }

    if ((err = uc_mem_map(uc, BASE, 0x1000, UC_PROT_ALL))) {
        uc_perror("uc_mem_map", err);
        return 1;
    }

    uc_mem_region *region;
    uint32_t count;

    if ((err = uc_mem_regions(uc, &region, &count))) {
        uc_perror("uc_mem_regions", err);
        return 1;
    }

    cout << "Starting address: 0x" << hex << region->begin << "End address: 0x"
<< hex << region->end << "Memorpermissions：" <<region->perms << "Number of
memor blocks  requested：" << count << endl;

    if ((err = uc_free(region))) {  ////Pay attention to freeing up memory
        uc_perror("uc_free", err);
        return 1;
    }

    return 0;
}
```

输出

uc_free

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

Free up the memory requested by uc_mem_regions()

```
@mem: Memory requested by uc_mem_regions (return *regions)

@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err are returned
```

▶ Source code implementation

```
uc_err uc_free(void *mem)
{
    g_free(mem);
    return UC_ERR_OK;
}

void g_free(gpointer ptr)
{
   free(ptr);
}
```

The usage example is the same uc_mem_regions()

uc_context_alloc

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

Allocate an area that can be used with uc_context_{save, restore} to perform a quick save/rollback of the CPU context, including registers and internal metadata.Context cannot be shared between engine instances with different architectures or schemas.

```
@uc: Handle returned by uc_open()
@context: Pointer to uc_engine*.When this function returns successfully, it will be updated with a pointer to the new context.After that, you must use uc_context_free() to free up these allocated memory.
@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err are returned
```

▶ Source code implementation

```
uc_err uc_context_alloc(uc_engine *uc, uc_context **context)
{
    struct uc_context **_context = context;
    size_t size = uc->cpu_context_size;

    *_context = g_malloc(size);
    if (*_context) {
        (*_context)->jmp_env_size = sizeof(*uc->cpu->jmp_env);
        (*_context)->context_size = size - sizeof(uc_context) - (*_context)-
>jmp_env_size;
        return UC_ERR_OK;
    } else {
        return UC_ERR_NOMEM;
    }
}
```

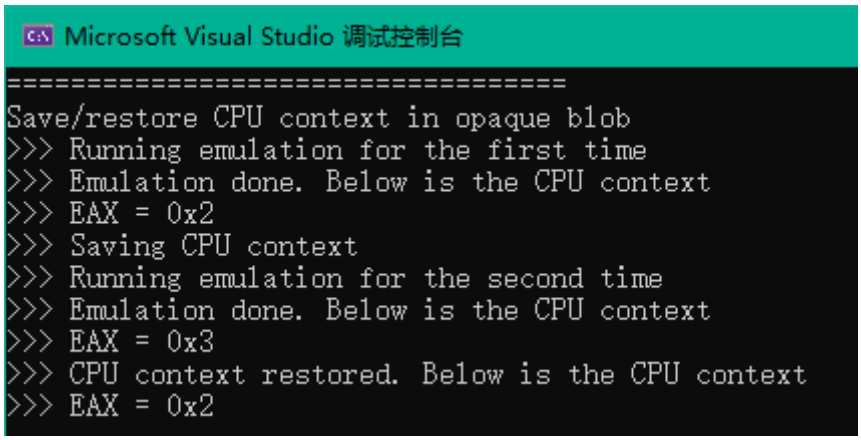
使用示例

```
#include <iostream>
#include <string>
#include "unicorn/unicorn.h"
using namespace std;

#define ADDRESS 0x1000
```

```c
#define X86_CODE32_INC "\x40"   // INC eax

int main()
{
    uc_engine* uc;
    uc_context* context;
    uc_err err;

    int r_eax = 0x1;   // EAX declaration

    printf("=================================\n");
    printf("Save/restore CPU context in opaque blob\n");

    err = uc_open(UC_ARCH_X86, UC_MODE_32, &uc);
    if (err) {
        printf("Failed on uc_open() with error returned: %u\n", err);
        return 0;
    }

    uc_mem_map(uc, ADDRESS, 8 * 1024, UC_PROT_ALL);

    if (uc_mem_write(uc, ADDRESS, X86_CODE32_INC, sizeof(X86_CODE32_INC) - 1)) {
        printf("Failed to write emulation code to memory, quit!\n");
        return 0;
    }

    //Initialization register
    uc_reg_write(uc, UC_X86_REG_EAX, &r_eax);

    printf(">>> Running emulation for the first time\n");

    err = uc_emu_start(uc, ADDRESS, ADDRESS + sizeof(X86_CODE32_INC) - 1, 0, 0);
    if (err) {
        printf("Failed on uc_emu_start() with error returned %u: %s\n",
            err, uc_strerror(err));
    }

    printf(">>> Emulation done. Below is the CPU context\n");

    uc_reg_read(uc, UC_X86_REG_EAX, &r_eax);
    printf(">>> EAX = 0x%x\n", r_eax);

    //Apply for and save CPU context
    printf(">>> Saving CPU context\n");

    err = uc_context_alloc(uc, &context);
    if (err) {
        printf("Failed on uc_context_alloc() with error returned: %u\n", err);
        return 0;
    }

    err = uc_context_save(uc, context);
    if (err) {
        printf("Failed on uc_context_save() with error returned: %u\n", err);
        return 0;
    }

    printf(">>> Running emulation for the second time\n");
```

```
    err = uc_emu_start(uc, ADDRESS, ADDRESS + sizeof(X86_CODE32_INC) - 1, 0, 0);
    if (err) {
        printf("Failed on uc_emu_start() with error returned %u: %s\n",
            err, uc_strerror(err));
    }

    printf(">>> Emulation done. Below is the CPU context\n");

    uc_reg_read(uc, UC_X86_REG_EAX, &r_eax);
    printf(">>> EAX = 0x%x\n", r_eax);

    // 恢复 CPU 上下文
    err = uc_context_restore(uc, context);
    if (err) {
        printf("Failed on uc_context_restore() with error returned: %u\n", err);
        return 0;
    }

    printf(">>> CPU context restored. Below is the CPU context\n");

    uc_reg_read(uc, UC_X86_REG_EAX, &r_eax);
    printf(">>> EAX = 0x%x\n", r_eax);

    //Release CPU context
    err = uc_context_free(context);
    if (err) {
        printf("Failed on uc_free() with error returned: %u\n", err);
        return;
    }

    uc_close(uc);
}
```

output



uc_context_save

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

Save the current CPU context

```
@uc: Handle returned by uc_open()
@context: The handle returned by uc_context_alloc()
@return If successful, UC_ERR_OK is returned, otherwise other error
types enumerated by uc_err are returned
```

▶ Source code implementation

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

The usage example is the same uc_context_alloc()

uc_context_restore

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

Restore the saved CPU context

```
@uc: Handle returned by uc_open()
@context: uc_context_alloc() returns a handle that has been saved using
uc_context_save
@return If successful, UC_ERR_OK is returned, otherwise other error types
enumerated by uc_err are returned
```

▶ Source code implementation

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

The usage example is the same uc_context_alloc()

uc_context_size

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

Returns the size required to store the cpu context.Can be used to allocate a buffer to contain the cpu context, and directly call uc_context_save.

```
@uc: Handle returned by uc_open()

@return stores the size required for the cpu context, of type size_t.
```

▶ Source code implementation

```
size_t uc_context_size(uc_engine *uc)
{
    return sizeof(uc_context) + uc->cpu_context_size + sizeof(*uc->cpu-
>jmp_env);
}
```

The usage example is the same uc_context_alloc()

uc_context_free

```
uc_err uc_context_free(uc_context *context);
```

Free up the memory requested by uc_context_alloc()

```
@context: uc_context created by uc_context_alloc

@return If successful, UC_ERR_OK is returned, otherwise other error types enumerated by uc_err are returned
```

The usage example is the same as uc_context_alloc()