Unicorn-Engine API Documentation

Version	1.0.3

Official API document by kabeor

PDF File

Unicorn Engine is a lightweight , multi-platform , multi-architecture CPU simulator framework , the current version is based on Qemu 2.0.x development , it can be used instead of the CPU to simulate the execution of code , commonly used in malicious code analysis , Fuzzing , etc., the project is used in the Qiling virtual framework , Radare2 reverse analysis framework , GEF (gdb pwn analysis plugin), Pwndbg, Angr symbolic execution framework and many other famous projects.

0x0 Preparation for Development

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

Compile your own libs and dlls

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

Download and unzip

The file structure is as follows:

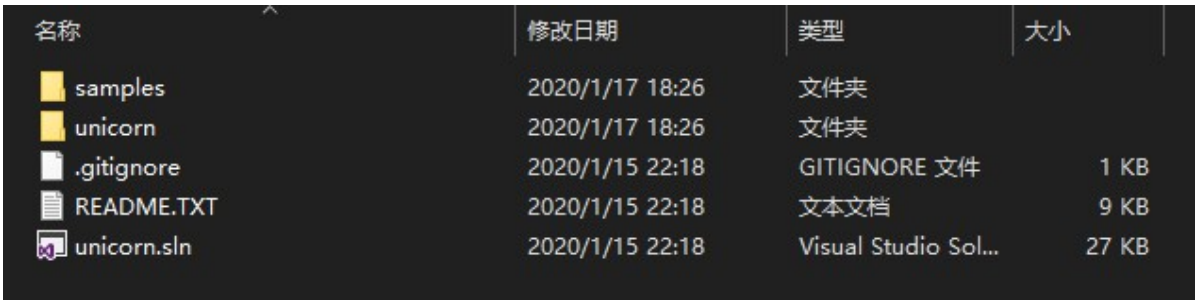
```
. <- Main engine core engine + README + Compile document COMPILE.TXT etc.
├── bindings <- bindings
│ ├── dotnet <- .Net bindings + test code
│ ├── go <- go binding + test code
│ ├── haskell <- Haskell binding + test code
│ ├── java <- Java binding + test code
│ ├── pascal <- Pascal binding + test code
│ ├── python <- Python bindings + test code
│ ├── ruby <- Ruby bindings + test code
│ ├── rust <- Rust bindings + test code
│ └── vb6 <- VB6 bindings + test code
├── docs <- documentation, mainly Unicorn implementation ideas
├── include <- C header file
├── msvc <- Microsoft Visual Studio Support (Windows)
out <- Build output
├── qemu <- qemu (modified) source code
├── samples <- Unicorn usage examples
└── tests <- C test cases
```

Below is a demonstration of Windows 10

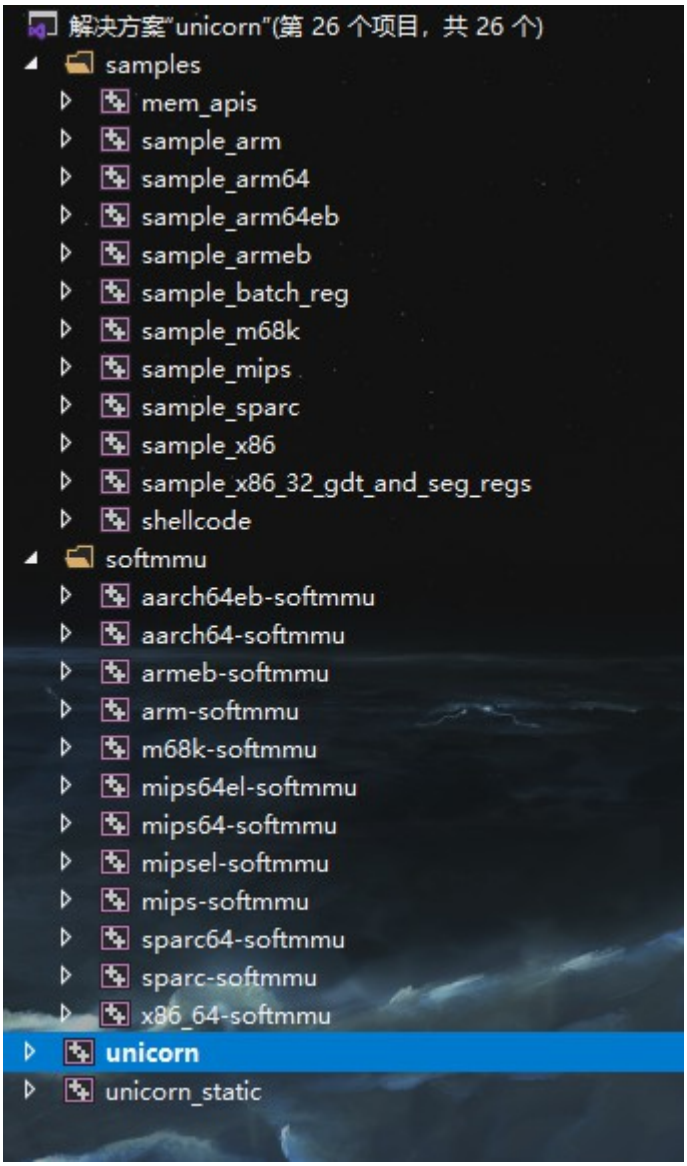
using Visual Studio 2019 to compile and open

the msvc folder with the following internal

structure

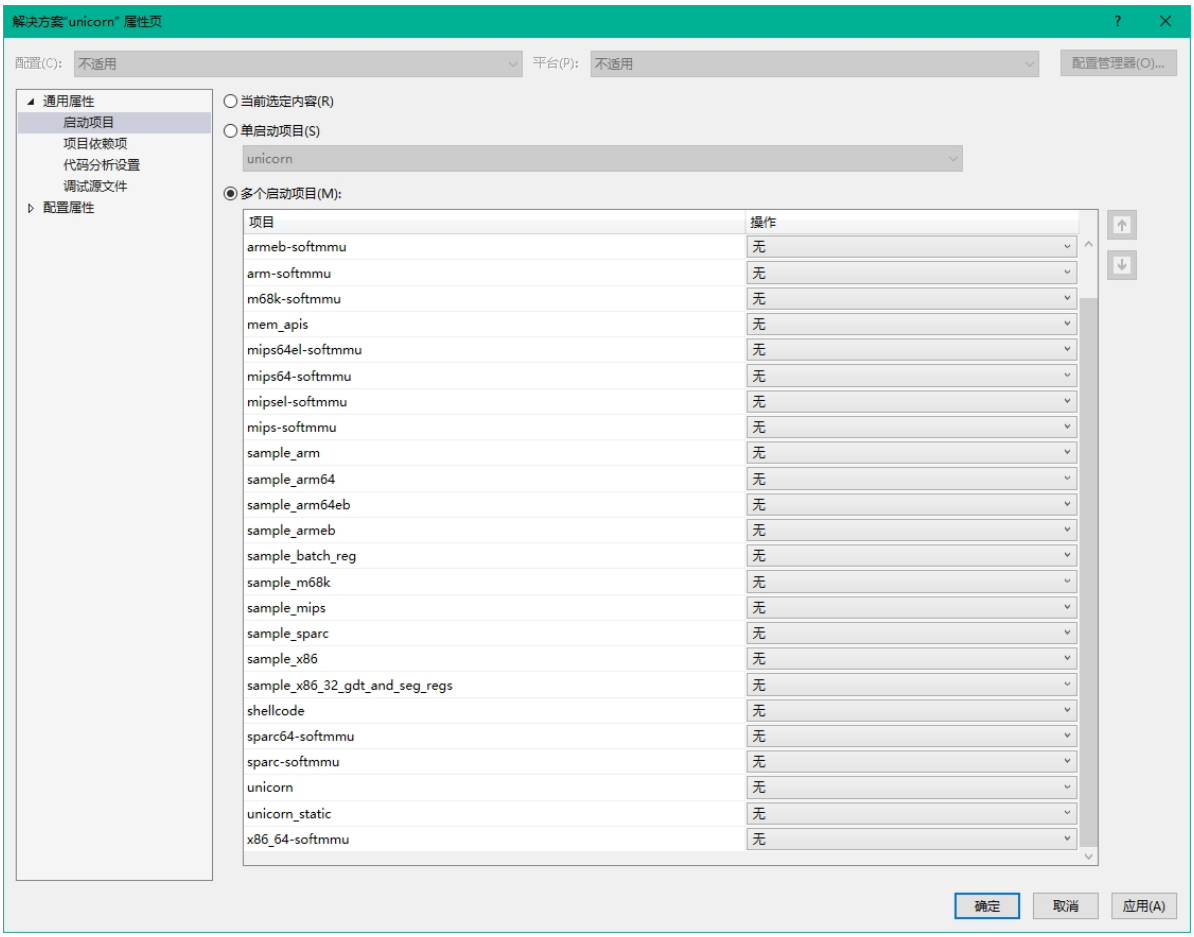


VS opens the unicorn.sln project file and the solution automatically loads these



If you need all of them, just compile them directly, if you need only a few of them, then right-click Solution->Properties->Configuration Properties->Generate Options andcheck the support items you need.

Multiple project operations can also be configured in the startup project as follows



After compilation, it will generate unicorn.lib static compilation library and unicorn.dll dynamic library in the current folder Debug directory so that you can start to develop with Unicorn.

The latest compiled version is 1.0.3, you can edit the source code of the latest version by

> yourself to get more available APIs. 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 of the later developments.

Click compile, go to unicorn\msvc\x32 or x64\Debug or Release and look for unicorn.dll and unicorn.lib.

Engine Call Testing

Create a new VS project, copy... \unicorn-master\include\unicorn in the header files, as well as compiled libs and dlls to the main directory of the new 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, add the existing item unicorn.h to the header file and unicorn.lib to the resource file, and regenerate the solution.

Next, we tested our generated

Unicorn engine master file code

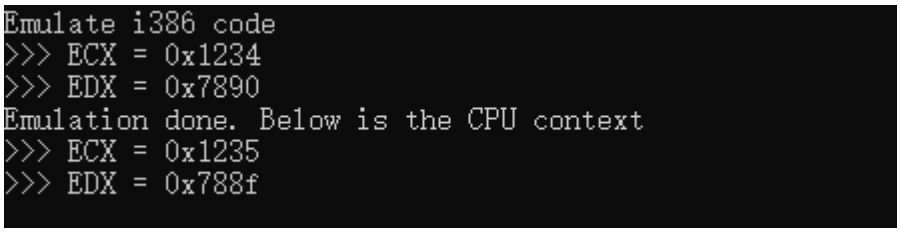
as follows

▶ Code

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

// Instructions to be simulated
#define X86_CODE32 "\x41\x4a" // INC ecx; DEC
edx

// Starting address
#define ADDRESS 0x1000000

int main()
{
    uc_engine*
    uc; uc_err
    err.                    // ECX
    int r_ecx = 0x1234;     register
    int r_edx = 0x7890.     // EDX
    printf("Emulate i386 code\n");.  register

    // X86-32bit mode initialization emulation
    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;
```

```
    }

    // Request 2MB of memory for the emulator
    uc_mem_map(uc, ADDRESS, 2 * 1024 * 1024, UC_PROT_ALL);

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

    // Initialize registers
    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);

     // Analog code
    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));
    }

    // Print register values
    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;
}
```

The results of the run are as follows



ecx+1 and edx-1 are successfully simulated.

0x1 Data type

> indexing

uc_arch

uc_mode

uc_err

---

uc_arch

Architecture Options

▶ Code

```
typedef enum uc_arch {
    UC_ARCH_ARM = 1, // ARM architectures (including
    Thumb, Thumb-2)       // ARM architecture (including
    Thumb, Thumb-2) UC_ARCH_ARM64, // ARM-64, also
    known as AArch64.     // ARM-64, also known as
    AArch64.
    UC_ARCH_MIPS.         // Mips Architecture
    UC_ARCH_X86, // X86 architecture (including x86 &
    x86-64)               // X86 架构 (包括 x86 & x86-
    64) UC_ARCH_PPC,      // PowerPC architecture (not
    supported yet) UC_ARCH_SPARC, // Sparc architecture
    (including x86 & x86-64)      // Sparc architecture
    UC_ARCH_M68K.         // M68K architecture
    UC_ARCH_MAX.
} uc_arch.
```

uc_mode

Mode Selection

▶ Code

```
typedef enum uc_mode {
    UC_MODE_LITTLE_ENDIAN =
    0,                          // Small end
    UC_MODE_BIG_ENDIAN = 1 << 30, // Big end-
    sequence mode                sequence mode
                                 (default)

    // arm / arm64
    UC_MODE_ARM = 0,             // ARM mode
    UC_MODE_THUMB = 1 << 4,      // THUMB mode (including Thumb-2)
    UC_MODE_MCLASS = 1 <<        // ARM's Cortex-M family (not supported
    5,                           yet)
    UC MODE V8 = 1 << 6,         // ARMv8 A32 encodings for ARM (not
                                 supported yet)
    // arm (32bit) cpu type
    UC_MODE_ARM926 = 1 << 7,     // ARM926 CPU type
    UC_MODE_ARM946 = 1 << 8.     // ARM946 CPU type
    UC_MODE_ARM1176 = 1 <<       // ARM1176 CPU type
    9.

    // mips
    UC_MODE_MICRO = 1 << 4,      // MicroMips mode (not
    UC_MODE_MIPS3 = 1 << 5,      supported yet)
    UC_MODE_MIPS32R6 = 1 <<      // Mips III ISA (not yet
    6,                           supported)
    UC_MODE_MIPS32 = 1 << 2,     // Mips32r6 ISA (not
    UC_MODE_MIPS64 = 1 << 3,     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)
    // sparc
    UC_MODE_SPARC32 = 1 << 2,     // 32-bit mode
    UC_MODE_SPARC64 = 1 << 3,     // 64-bit mode
    UC_MODE_V9 = 1 << 4,          // SparcV9 mode (not supported yet)
    // m68k
} uc_mode.
```

uc_err

Error type, is the return value of uc_errno()

▶ Code

```
typedef enum uc_err {
    UC_ERR_OK = 0,     // No error
    UC_ERR_NOMEM, // Insufficient memory.            // Insufficient memory: uc_open(), uc_emulate()
    UC_ERR_ARCH,       // Unsupported architectures: uc_open() UC_ERR_HANDLE, // Unavailable handles.
                       // Unavailable handle
    UC_ERR_MODE, // Unavailable/Unsupported Architecture.      // Unavailable/unsupported architecture: uc_open() UC_ERR_VERSION, // unsupported version (middleware)
    UC_ERR_READ_UNMAPPED, // Exit simulation due to reading on unmapped memory: uc_emu_start() UC_ERR_WRITE_UNMAPPED, // Exit simulation due to writing on unmapped memory: uc_emu_start() UC_ERR_FETCH_ UNMAPPED, // Exit simulation due to fetching data in unmapped memory: uc_emu_start() UC_ERR_HOOK, // Exit simulation due to fetching data in unmapped memory: uc_emu_start() UNMAPPED, // Exit simulation due to fetch on unmapped memory: uc_emu_start()
    UC_ERR_HOOK, // Invalid hook type. // Invalid hook type: uc_hook_add()
    UC_ERR_INSN_INVALID, // Exit simulation due to invalid instruction: uc_emu_start() UC_ERR_MAP, // Invalid memory map: uc_mem_map()
    UC_ERR_WRITE_PROT, // Stop simulation due to UC_MEM_WRITE_PROT conflict: uc_emu_start() UC_ERR_READ_PROT, // Stop simulation due to UC_MEM_READ_PROT conflict: uc_emu_start() UC_ERR_ FETCH_PROT, // Stop simulation due to UC_MEM_FETCH_PROT conflict: uc_emu_start() UC_ERR_ARG, // Provided to uc_xxx.       // Invalid argument provided to uc_xxx function
    UC_ERR_READ_UNALIGNED, // unaligned read
    UC_ERR_WRITE_UNALIGNED, // unaligned write
    UC_ERR_FETCH_UNALIGNED, // unaligned fetch
    UC_ERR_HOOK_EXIST, // the hook for this event is already present UC_ERR_ RESOURCE, // Insufficient resources.      // Insufficient resources: uc_emu_start() UC_ERR_EXCEPTION, // Unhandled CPU exception UC_ERR_TIMEOUT // Emulation timeout
} uc_err.
```

uc_mem_type

All memory access types for

UC_HOOK_MEM_*▶ Code

```
typedef enum uc_mem_type {
    uc_mem_read = 16,    // Memory
    from . UC_MEM_WRITE, // memory
    read from...             // Memory
    written to...
    UC_MEM_FETCH, // Memory was fetched.        //
    Memory was fetched UC_MEM_READ_UNMAPPED, //
    Unmapped memory from...        // Unmapped
    memory from... UC_MEM_WRITE_UNMAPPED, //
    Unmapped memory is read from... // Unmapped
    memory written to... UC_MEM_FETCH_UNMAPPED, //
    Unmapped memory is fetched from...   //
    Unmapped memory fetched UC_MEM_WRITE_PROT, //
    Memory write-protected, but mapped
    UC_MEM_READ_PROT, // Memory read-protected, but
    mapped.                // Memory read protected, but
    mapped UC_MEM_FETCH_PROT, // Memory not
    executable, but mapped UC_MEM_READ_AFTER, //
    Memory accessed from (successfully accessed)
                           // Memory read from
    (successfully accessed address)
} uc_mem_type.
```

uc_hook_type

All hook type arguments to

uc_hook_add()▶ 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 are 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 for reading memory on unmapped memory
    UC_HOOK_MEM_READ_UNMAPPED = 1 << 4,
    // Hook invalid memory write events
    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 fetch execution event
    UC_HOOK_MEM_FETCH = 1 << 12,
    //  Hook memory read events to allow only addresses that can be successfully accessed
    // Callbacks will be triggered on successful reads
    UC_HOOK_MEM_READ_AFTER = 1 << 13,
    // Hook Invalid Instruction Exception
    UC_HOOK_INSN_INVALID = 1 << 14,
```

```
} uc_hook_type.
```

hook_types

Macro Defining Hook Types

▶ Code

```
// Hook all events with unmapped memory accesses
#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 illegal memory reads.
#define UC_HOOK_MEM_READ_INVALID (UC_HOOK_MEM_READ_PROT +
UC_HOOK_MEM_READ_UNMAPPED)
// Hook all illegal memory writes.
#define UC_HOOK_MEM_WRITE_INVALID (UC_HOOK_MEM_WRITE_PROT +
UC_HOOK_MEM_WRITE_UNMAPPED)
// Hook all events that fetch memory illegally
#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 is not included in UC_HOOK_MEM_READ_PROT and
UC_HOOK_MEM_READ_UNMAPPED.
Pre-trigger .
//          So 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

Map memory regions by uc_mem_map() and uc_mem_map_ptr() Retrieve a list of such memory regions using uc_mem_regions()

▶ Code

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

uc_query_type

All query type parameters for

uc_query()▶ 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 contexts

▶ Code

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

uc_prot

Permissions for the new mapped area

▶ Code

```
typedef enum uc_prot {
   uc_prot_none = 0,     //
   None
   UC_PROT_READ = 1, //read
   //Read UC_PROT_WRITE = 2,
   //write UC_PROT_EXEC = 4,
   //executable UC_PROT_ALL = 7,
   //all privileges
   UC_PROT_WRITE = 2, //write
   //All privileges
} uc_prot.
```

0x2 API

> indexing

uc_version

uc_arch_supported

uc_open

uc_close

uc_query

uc_errno

uc_strerror

uc_reg_write

uc_reg_read

uc_reg_write_batch

uc_reg_read_batch

uc_mem_write

uc_mem_read

---

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 hexadecimal number, calculated as (major << 8 | minor)
```

Tip: This return value can be compared to 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; // macro
        *minor = UC_API_MINOR; // macro
    }

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

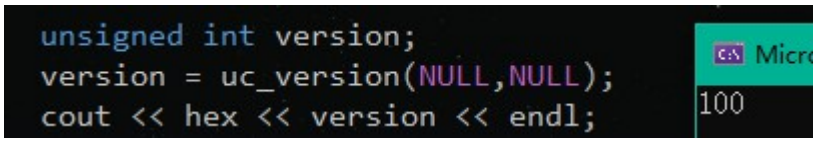
Compiled unchangeable, do not

accept customized versions of use

examples:

```
#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 version number 1.0.0

uc_arch_supported

```
bool uc_arch_supported(uc_arch arch).
```

Determine if Unicorn supports the current architecture

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

▶ source code implementation

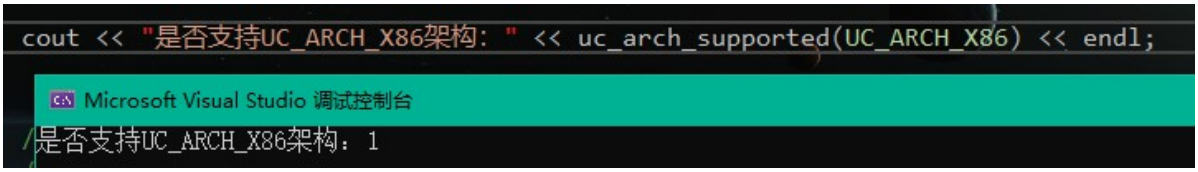
```
bool uc_arch_supported(uc_arch arch)
{
    switch (arch) {
#ifdef UNICORN_HAS_ARM
        case UC_ARCH_ARM.    return
        true.
#endif
#ifdef UNICORN_HAS_ARM64
        case UC_ARCH_ARM64: return true
#endif
#ifdef UNICORN_HAS_M68K
        case UC ARCH M68K: return true;
#endif
#ifdef UNICORN_HAS_MIPS
        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;
    }
}
```

Example of use:

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

int main()
{
    cout << "Is the UC_ARCH_X86 architecture supported:" <<
    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);
```

Creating a new instance of Unicorn

```
@arch: Architecture type
(UC_ARCH_*) @mode: Hardware
mode. Combined with UC_MODE_*.
@uc: pointer to uc_engine, updated on return

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err
enumeration.
```

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)); // request memory
        if (!uc) {
            // Insufficient memory
            return UC_ERR_NOMEM.
        }

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

```
        uc->mode = mode;

        // Initialization
        // 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) {     // Preprocess according to the 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;
#endif          break;
#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 {      // Small end 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;
```

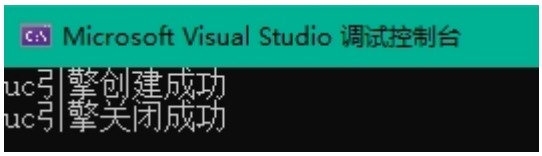
```
        }

        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 request heap memory, which must be freed with uc_close after use, otherwise a leak will occur.

Example of use:

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

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

    //// Initializing 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 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 shutdown successful" << endl;

    return 0;
}
```

exports



uc_close

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

Closing a uc instance will free up memory. It cannot be restored after closing.

```
@uc: pointer to the pointer returned by uc_open()

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err enumeration.
```

▶ source code implementation

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

    // Cleaning 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).

    // Clean up all 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->machine_state))
    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 the memory
    g_free(uc->system_memory).

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

    // Release of additional 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 hooks and hook lists
    for (i = 0; i < UC_HOOK_MAX; i++) {
        cur = uc->hook[i].head;
        // hook can exist in multiple lists and can be counted to get the time of release
        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 the uc itself
    memset(uc, 0,
    sizeof(*uc)); free(uc);

    return UC_ERR_OK.
}
```

Example of use is the same as uc_open()

uc_query

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

Internal state of the query engine

```
@uc: handle returned by uc_open()
@type: type of enumeration in uc_query_type

@result: a pointer to the internal state being queried

@return: if successful, return UC_ERR_OK , otherwise return other error types of uc_err
enumeration
```

▶ 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.
}
```

Example of use:

```
#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 created successfully" << endl;

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

exports

The architecture query result is 4, which corresponds to exactly UC_ARCH_X86

uc_errno

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

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

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

@return: if successful, return UC_ERR_OK , otherwise return other error types of uc_err
enumeration
```

source code implementation

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

Example of use:

```
#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 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;
}
```

exports

No error, output error number 0

uc_strerror

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

Returns the explanation of the given error number

```
@code: Error number

@return: String pointer to the explanation 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)";
    }
}
```

Example of use:

```
#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 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;
}
```

exports

uc_reg_write

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

Write value to register

```
@uc: the handle returned by
uc_open() @regid: the
register ID that will be
modified
@value: Pointer to the value to which the register will be modified.
```

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err
▶ enumeration.

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
        structure
confirm or agree with
    else
        return UC_ERR_EXCEPTION.

    return ret;
}
```

Example of use:

```
#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 created successfully" << endl;

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

exports

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: ID of the
register to be read @value:
pointer to the stored register
value

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err
enumeration.
```

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.
}
```

Example of use:

```
#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 created successfully" << endl;

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

    int recv_eax.
    err = uc_reg_read(uc, UC_X86_REG_ECX, &recv_eax);
    if (!err)
        cout << "Read successful: " << 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;
}
```

exports

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: array storing multiple register
IDs to be written @value: pointer to array
holding multiple values @count: length of
array of *regs and *vals

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err
enumeration.
```

source code implementation

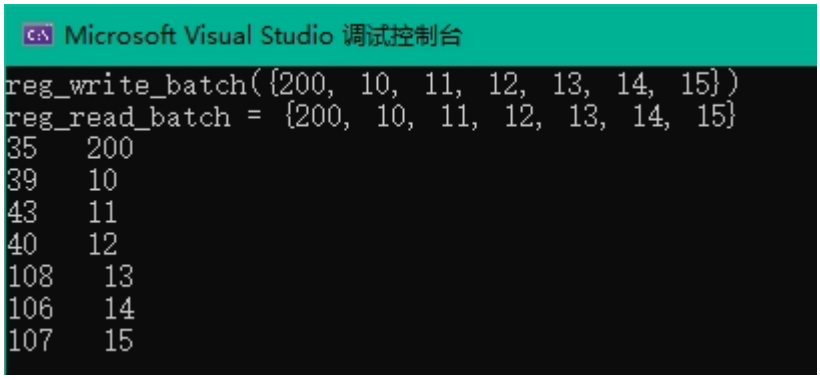
```
uc_err uc_reg_write_batch(uc_engine *uc, int *ids, void *const *vals, int count)
{
    int ret = UC_ERR_OK;
    if (uc->reg_write)
        ret = uc->reg_write(uc, (unsigned int *)ids, vals, count);
    else
        return UC_ERR_EXCEPTION.

    return ret;
}
```

Example of use:

```
#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;
}
```

exports



uc_reg_read_batch

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

Reads the values of multiple registers at the same time.

```
@uc: handle returned by uc_open()
@regid: array storing multiple register
IDs to be read @value: pointer to an
array holding multiple values @count:
length of the *regs and *vals arrays

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err
enumeration.
```

▶ 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.
}
```

Usage examples are 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);
```

Writes a piece of byte code in memory.

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

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

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err
enumeration.
```

▶ 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) {
```

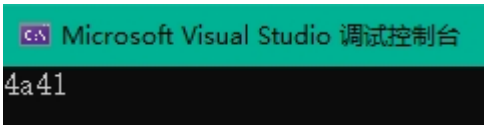
```
        address = uc->mem_redirect(address);
    }

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

    // Memory regions can overlap neighboring memory blocks
    while(count < size) {
        MemoryRegion *mr = memory_mapping(uc, address);
        if (mr) {
            uint32_t operms = mr->perms;
            if (! (operms & UC_PROT_WRITE)) // not write-protected
                // 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)) // not write-protected
                // Set write protection
                uc->readonly_mem(mr, true);

            count += len;
            address += len;
            bytes += len.
        } else // This address is not yet mapped
            break;
    }

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

Example of use:

```
#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;
}
```

exports



uc_mem_read

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

Reads bytes from memory.

`@uc`: handle returned by `uc_open()`
`@address`: start address of the byte to read
`@bytes`. Pointer to a pointer containing the data to be read from memory
`@size`. The size of the memory to be read.

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

`@return` Returns `UC_ERR_OK` if successful, otherwise returns other error types in the `uc_err` enumeration.

▶ 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 regions can overlap neighboring 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 is not yet mapped
            break;
    }

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

Usage examples are the same as 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);
```

Simulates machine code for a specified period of time.

`@uc`: handle returned by `uc_open()`
`@begin`: address to start simulation
`@until`: address of the analog stop (when reached)
`@timeout`: duration of the simulation code (in microseconds). When this value is `0`, there will be no time limit on the simulation code until the simulation is complete.
`@count`: the number of instructions to simulate. When this value is `0`, all executable code will be simulated until the simulation is complete

`@return` Returns `UC_ERR_OK` if successful, otherwise returns other error types in the `uc_err` enumeration.

source code implementation

```
uc_err uc_emu_start(uc_engine* uc, uint64_t begin, uint64_t until, uint64_t
timeout, size_t count)
{
    // Remake counters
    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 later added IPs and CSs
                    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;
```

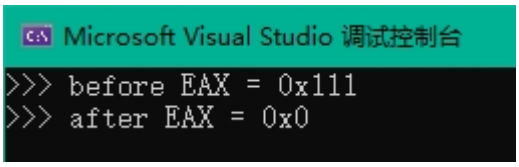
```
    uc->emu_count = count;
    // Remove the count hook if it is not needed.
    if (count <= 0 && uc->count_hook ! = 0) {
        uc_hook_del(uc, uc->count_hook);
        uc->count_hook = 0;
    }
    //  Set the number of commands to be logged by the counting hook
    if (count > 0 && uc->count_hook == 0) {
        uc_err err;
        //  The callback to the count instruction must run before all other operations, so the hook
        must be inserted at the beginning of the hook  list without the
It's an add-on 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 complete
    uc->emulation_done = true;

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

    if(uc->timed_out)
        return UC_ERR_TIMEOUT.

    return uc->invalid_error;
}
```

Example of use:

```
#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;
}
```

exports



uc_emu_stop

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

Stop the simulation

Typically called from a callback function registered through the tracing API.

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

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err enumeration.
```

▶ 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.
}
```

Example of use:

```
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, ...) ;
```

Registers callbacks for hook events, which will be called back when the hook event is triggered.

```
@uc: handle returned by uc_open()
@hh: Register the handle obtained by hook. Used in uc_hook_del().
@type: hook type
@callback: callback to be run when an instruction is hit
@user_data: user-defined data. Will be passed to the last parameter of the
callback function @user_data @begin: the start address of the area where
the callback will take effect (included)
@end: end address of the area where the callback will take effect (included)
   Note 1: Callbacks are only invoked if the address of the callback is in [@begin, @end].
Note 2: If @begin > @end, the callback is called whenever this hook
                        type is triggered.
@... : variable parameters (depends on @type)
   Note: If @type = UC_HOOK_INSN, this is the command ID (e.g. UC_X86_INS_OUT).
```

▶ `@return Returns UC_ERR_OK` if successful, otherwise returns other error types in the `uc_err` enumeration.

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 one extra
    parameter: the 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;
}
```

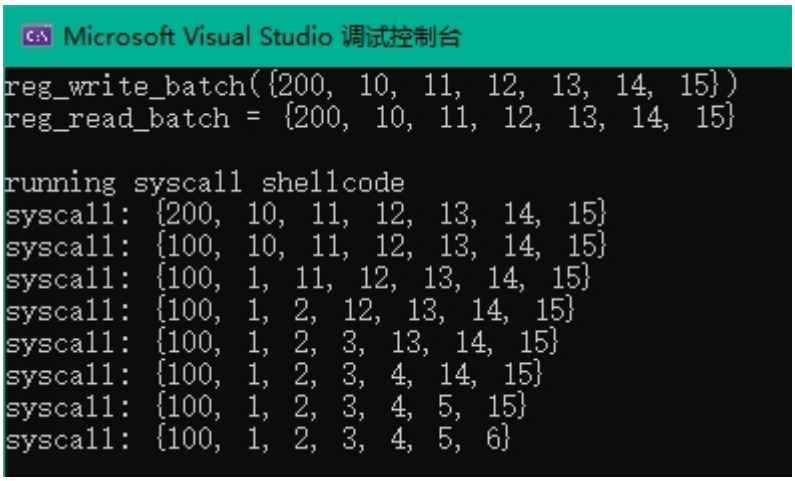
Example of use:

```
#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\x00\x00\x48\xc7\xc7\xc7\x01\x00\x00\x00\x48\xc7\xc6\x02\
x00\x0 0\x00\x48\xc7\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\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)); 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;
}
```

exports



Do a hook for each command

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` Returns `UC_ERR_OK` if successful, otherwise returns other error types in the `uc_err` enumeration.

▶ 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.
}
```

Example of use:

```
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 block of memory for the simulation.

`@uc`: handle returned by `uc_open()`
`@address`: The starting address of the new memory region to be mapped to. This address must be aligned to `4KB` or a `UC_ERR_ARG` error will be returned. `@size`: The size of the new memory region to be mapped. This size must be a multiple of `4KB or a UC_ERR_ARG` error will be returned. `@perms`: The permissions for the new mapped region. The parameter must be `UC_PROT_READ | UC_PROT_WRITE | UC_PROT_EXEC` or a combination of these or a `UC_ERR_ARG` error is returned.

`@return Returns UC_ERR_OK` if successful, otherwise returns other error types in the `uc_err` enumeration.

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
    safety check if (res)
        return res;
```

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

Usage examples are the same as 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 region to be mapped to. This address must be aligned to `4KB` or a `UC_ERR_ARG` error will be returned. `@size:` `The` size of the new memory region to be mapped. This size must be a multiple of `4KB or a UC_ERR_ARG` error will be returned. `@perms:` The permissions for the new mapped region. The parameter must be `UC_PROT_READ | UC_PROT_WRITE | UC_PROT_EXEC` or a combination of these or a `UC_ERR_ARG` error is returned.
`@ptr:` `Pointer` to host memory that supports the new mapped memory. The size of the mapped host memory should be the same as or larger than the `size and be` mapped using at least `PROT_READ | PROT_WRITE`, otherwise no mapping is defined.

▶ `@return Returns UC_ERR_OK` if successful, otherwise returns other error types in the `uc_err` enumeration.

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
    safety check if (res)
        return res;

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

Usage examples are the same as uc_mem_map().

uc_mem_unmap

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

Unmapping of simulated memory regions

`@uc`: handle returned by `uc_open()`
`@address`: The starting address of the new memory region to be mapped to. This address must be aligned to `4KB` or a `UC_ERR_ARG` error will be returned.
`@size`: The size of the new memory region to map to. This size must be a multiple of `4KB` or a `UC_ERR_ARG` error will be returned.

`@return Returns UC_ERR_OK` if successful, otherwise returns other error types in the `uc_err` enumeration.

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)
        // No areas to unmap
        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 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 if the entire block requested by the user is mapped
    if (!check_mem_area(uc, address, size))
        return UC_ERR_NOMEM.

    // If the region spans neighboring regions, it may be necessary to split the region
    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 ! mr =
        memory_mapping(uc, addr); if
        (mr !
           uc->memory_unmap(uc, mr);
        count += len;
        addr += len;
```

```
    }

    return UC_ERR_OK.
}
```

Example of use:

```
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);
```

Setting permissions for analog memory

`@uc`: handle returned by `uc_open()`
`@address`: The starting address of the new memory region to be mapped to. This address must be aligned to `4KB` or a `UC_ERR_ARG` error will be returned. `@size`: `The` size of the new memory region to be mapped. This size must be a multiple of `4KB or a UC_ERR_ARG` error will be returned. `@perms`: The new permissions for the mapped region. The parameter must be `UC_PROT_READ | UC_PROT_WRITE | UC_PROT_EXEC` or a combination of these or a `UC_ERR_ARG` error is returned.

`@return Returns UC_ERR_OK` if successful, otherwise returns other error types in the `uc_err` enumeration.

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.
}
```

Example of use:

```
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);
```

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

This API allocates memory for @regions, which the user must later free via free() to avoid memory leaks.

```
@uc: handle returned by uc_open()
@regions: pointer to an array of uc_mem_region structures. Claimed by Unicorn and must be freed by uc_free().
@count: pointer to the number of uc_mem_region structures contained in @regions

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err enumeration.
```

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.
}
```

Example of use:

```
#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 << "Start address: 0x" << hex << region->begin << " End address: 0x" << hex
<< region->end << " Memory permissions:     " << region->perms << " Number of
memory blocks requested: " << count <<
endl;
    if ((err = uc_free(region)))      //// Note the
        {     uc_perror("uc_free",      release of memory
        err); return 1;
    }

    return 0;
}
```

exports

uc_free

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

Free memory requested by uc_mem_regions()

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

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err
enumeration.
```

▶ source code implementation

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

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

Usage examples are the same as 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 fast save/rollback of the CPU context, including registers and internal metadata. Contexts cannot be shared between engine instances with different architectures or modes.

```
@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. These allocated memories must then be freed using
uc_context_free().

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err
enumeration.
```

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.
    }
}
```

usage example

```
#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 register

    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;
    }

    // Initialize registers
    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);

    // Request and save CPU contexts
    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);

    // Restore CPU context
    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).
}
```

exports

uc_context_save

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

Save current CPU context

```
@uc: handle returned by uc_open()
@context: handle returned by uc_context_alloc()

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err enumeration.
```

▶ 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;
}
```

Usage examples are the same as uc_context_alloc().

uc_context_restore

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

Restoring a saved CPU context

```
@uc: handle returned by uc_open()
@context: uc_context_alloc() Returns and has been saved with uc_context_save.

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err enumeration.
```

▶ 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;
}
```

Usage examples are the same as uc_context_alloc().

uc_context_size

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

Returns the size needed to store the cpu context. This can be used to allocate a buffer to contain the cpu context and directly call the
uc_context_save.

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

@return The size required to store 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).
}
```

Usage examples are the same as uc_context_alloc().

uc_context_free

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

Free the memory requested by uc_context_alloc().

```
@context: uc_context created by uc_context_alloc

@return Returns UC_ERR_OK if successful, otherwise returns other error types in the uc_err
enumeration.
```

Usage examples are the same as uc_context_alloc().