**如何利用qemu搭建SOC protoype：80行代碼實現一個Cortex M4 模擬器**

category: [嵌入式開發](https://icode.best/category/%E5%B5%8C%E5%85%A5%E5%BC%8F%E5%BC%80%E5%8F%91)category: [C](https://icode.best/category/C)category: [qemu](https://icode.best/category/qemu) Posted on2021-01-22

<https://blog.csdn.net/u011280717/article/details/112983720>

**文章目錄**

[1. MY\_SOC Memory Map](https://icode.best/i/29831237731699#1_MY_SOC_Memory_Map_13)

[2. MY\_SOC 源碼](https://icode.best/i/29831237731699#2_MY_SOC__21)

[3. 測試代碼](https://icode.best/i/29831237731699#3__160)

[4 創建自己的IP](https://icode.best/i/29831237731699#4_IP_240)

<源碼文件 (https://github.com/JiaminMa/qemu_hw)>

隨著國內芯片公司越來越多，越來越多的底層程序員需要在pre silicon階段就要開發代碼。而在pre silicon階段測試方式有多種：

| **方式** | **優點** | **缺點** |
| --- | --- | --- |
| RTL simulation | 可以驗證最準確的硬件行為，可以測試SOC相關代碼 | 仿真速度非常慢，且rtl freeze之前硬件有bug |
| FPGA/ZEBU emulation | 可以驗證部分硬件行為，速度相對RTL simulation快 | 價格昂貴，難以佈署大量測試，且有些硬件沒法仿真 |
| 軟件模擬器，如QEMU | 速度最快，可以佈署大量測試 | 沒有現成的模擬器對應正在開發的SOC |

很明顯軟件模擬器的優點缺點顯而易見，如果開發人員可以在芯片開發前期定製出一個軟件模擬器對應SOC 原型，那麼可以大大提高pre silicon的效率。本文以一個小demo來演示如何在qemu源碼基礎上搭建一個最簡單的Cortex M4的SOC。 這裡就不講如何編譯qemu-system-arm了，網上有很多教程教學如何編譯qemu-system-arm。後文簡稱該SOC為MY\_SOC

**1. MY\_SOC Memory Map**

Cortex M自帶的system peripheral的地址這裡就不列出來了，比如systick，nvic這些都是arm規定的，沒法改，也沒必要改。這裡定義了最簡單三個外設。一段FLASH用來跑代碼，一段SRAM用來存數據，一個UART來打印。

|  | 起始地址 | 結束地址 |
| --- | --- | --- |
| FLASH(定義了一段4M 的FLASH) | 0x00000000 | 0x003FFFFF |
| SRAM(定義了一段16M的SRAM) | 0x20000000 | 0x20FFFFFF |
| UART0(使用ARM PL011 IP) | 0x40000000 | 0x40000FFF |

**2. MY\_SOC 源碼**

將my\_soc.c 放在qemu/hw/arm目錄下並且加入arm的makefile編譯即可。先貼出全部代碼再逐行解釋。加上頭文件include和宏定義一共77行代碼，可見利用qemu能夠很方便地搭出一個SOC模擬器原型。

#include "qemu/osdep.h"

#include "qapi/error.h"

#include "hw/arm/boot.h"

#include "hw/boards.h"

#include "qemu/log.h"

#include "exec/address-spaces.h"

#include "sysemu/sysemu.h"

#include "hw/arm/armv7m.h"

#include "hw/char/pl011.h"

#include "hw/irq.h"

#include "cpu.h"

#define MY\_SOC\_FLASH\_START (0x0)

#define MY\_SOC\_FLASH\_SIZE (4 \* 1024 \* 1024) //< 4M

#define MY\_SOC\_SRAM\_START (0x20000000)

#define MY\_SOC\_SRAM\_SIZE (16 \* 1024 \* 1024) //<16M

#define PL011\_UART0\_START (0x40000000)

#define PL011\_UART0\_IRQn (0)

#define NUM\_IRQ\_LINES 64

static void mysoc\_init(MachineState \*ms)

{

DeviceState \*nvic;

MemoryRegion \*sram = g\_new(MemoryRegion, 1);

MemoryRegion \*flash = g\_new(MemoryRegion, 1);

MemoryRegion \*system\_memory = get\_system\_memory();

/\* Flash programming is done via the SCU, so pretend it is ROM. \*/

memory\_region\_init\_ram(flash, NULL, "mysoc.flash", MY\_SOC\_FLASH\_SIZE, &error\_fatal);

memory\_region\_set\_readonly(flash, true);

memory\_region\_add\_subregion(system\_memory, MY\_SOC\_FLASH\_START, flash);

memory\_region\_init\_ram(sram, NULL, "mysoc.sram", MY\_SOC\_SRAM\_SIZE, &error\_fatal);

memory\_region\_add\_subregion(system\_memory, MY\_SOC\_SRAM\_START, sram);

nvic = qdev\_create(NULL, TYPE\_ARMV7M);

qdev\_prop\_set\_uint32(nvic, "num-irq", NUM\_IRQ\_LINES);

qdev\_prop\_set\_string(nvic, "cpu-type", ms->cpu\_type);

qdev\_prop\_set\_bit(nvic, "enable-bitband", true);

object\_property\_set\_link(OBJECT(nvic), OBJECT(get\_system\_memory()), "memory", &error\_abort);

/\* This will exit with an error if the user passed us a bad cpu\_type \*/

qdev\_init\_nofail(nvic);

pl011\_luminary\_create(PL011\_UART0\_START , qdev\_get\_gpio\_in(nvic, PL011\_UART0\_IRQn), serial\_hd(0));

armv7m\_load\_kernel(ARM\_CPU(first\_cpu), ms->kernel\_filename, MY\_SOC\_FLASH\_SIZE);

}

static void mysoc\_class\_init(ObjectClass \*oc, void \*data)

{

MachineClass \*mc = MACHINE\_CLASS(oc);

printf("%s entry\n", \_\_func\_\_);

mc->desc = "My SOC Cortex M4";

mc->init = mysoc\_init;

mc->ignore\_memory\_transaction\_failures = true;

mc->default\_cpu\_type = ARM\_CPU\_TYPE\_NAME("cortex-m4");

}

static const TypeInfo mysoc\_type = {

.name = MACHINE\_TYPE\_NAME("mysoc\_evb"),

.parent = TYPE\_MACHINE,

.class\_init = mysoc\_class\_init,

};

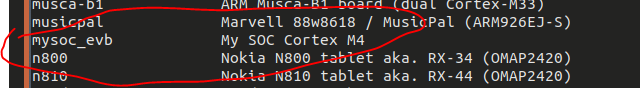
static void mysoc\_evb\_init(void)

{

type\_register\_static(&mysoc\_type);

}

type\_init(mysoc\_evb\_init)

代碼很簡單，從下往上看。這裡定義了一塊板子叫mysoc\_evb，通過type\_init宏上報給qemu，之後qemu在啟動地時候就能自動地調用mysoc\_init初始化soc外設。  
這裡定義了描述字符串為My SOC Cortex M4，cpu類型是cortex-m4，板子名字是mysoc\_evb。當這些結構體初始化完後，運行qemu-system-arm -machine help 就會出現我們自己地設備。  


static void mysoc\_class\_init(ObjectClass \*oc, void \*data)

{

MachineClass \*mc = MACHINE\_CLASS(oc);

printf("%s entry\n", \_\_func\_\_);

mc->desc = "My SOC Cortex M4";

mc->init = mysoc\_init;

mc->ignore\_memory\_transaction\_failures = true;

mc->default\_cpu\_type = ARM\_CPU\_TYPE\_NAME("cortex-m4");

}

static const TypeInfo mysoc\_type = {

.name = MACHINE\_TYPE\_NAME("mysoc\_evb"),

.parent = TYPE\_MACHINE,

.class\_init = mysoc\_class\_init,

};

static void mysoc\_evb\_init(void)

{

type\_register\_static(&mysoc\_type);

}

type\_init(mysoc\_evb\_init)

所以最關鍵地函數就是**mysoc\_init**這個函數，這個函數裡做的事情非常簡單，就是調用qemu的API創建相應的memory map以及設備就可以了。

* 創建4M flash並設置為只讀，定義了一個默認的加載文件mysoc.flash。如果mysoc.flash存在。那麼qemu就會把該文件的數據讀到這段flash中

memory\_region\_init\_ram(flash, NULL, "mysoc.flash", MY\_SOC\_FLASH\_SIZE, &error\_fatal);

memory\_region\_set\_readonly(flash, true);

memory\_region\_add\_subregion(system\_memory, MY\_SOC\_FLASH\_START, flash);

* 創建一段16M的SRAM， 並定義默認文件mysoc.sram

memory\_region\_init\_ram(sram, NULL, "mysoc.sram", MY\_SOC\_SRAM\_SIZE, &error\_fatal);

memory\_region\_add\_subregion(system\_memory, MY\_SOC\_SRAM\_START, sram);

* 配置NVIC, 設置64個外部中斷槽，使能bitband。

nvic = qdev\_create(NULL, TYPE\_ARMV7M);

qdev\_prop\_set\_uint32(nvic, "num-irq", NUM\_IRQ\_LINES);

qdev\_prop\_set\_string(nvic, "cpu-type", ms->cpu\_type);

qdev\_prop\_set\_bit(nvic, "enable-bitband", true);

object\_property\_set\_link(OBJECT(nvic), OBJECT(get\_system\_memory()), "memory", &error\_abort);

* 添加一個PL011 UART，地址為0x40000000， 中斷號為0。

pl011\_luminary\_create(PL011\_UART0\_START, qdev\_get\_gpio\_in(nvic, PL011\_UART0\_IRQn), serial\_hd(0));

* 設置kernel加載到flash中

armv7m\_load\_kernel(ARM\_CPU(first\_cpu), ms->kernel\_filename, MY\_SOC\_FLASH\_SIZE);

至此MY\_SOC就配置完了。我們寫一段測試代碼來測試這個模擬器能不能運行。

**3. 測試代碼**

鏈接文件，把代碼段放在FLASH中，data，bss段放在SRAM中

MEMORY

{

FLASH (rx) : ORIGIN = 0x00000000, LENGTH = 4M

SRAM (rwx) : ORIGIN = 0x20000000, LENGTH = 16M

}

SECTIONS

{

.text :

{

\_text = .;

KEEP(\*(.isr\_vector))

\*(.text\*)

\*(.rodata\*)

\_etext = .;

} > FLASH

/DISCARD/ :

{

\*(.ARM.exidx\*)

\*(.gnu.linkonce.armexidx.\*)

}

.data : AT(ADDR(.text) + SIZEOF(.text))

{

\_data = .;

\*(vtable)

\*(.data\*)

\_edata = .;

} > SRAM

.bss :

{

\_bss = .;

\*(.bss\*)

\*(COMMON)

\_ebss = .;

} > SRAM

. = ALIGN(32); /\*Not sure if this needs to be done, but why not.\*/

\_stack\_bottom = .; /\*Address of the bottom of the stack.\*/

. = . + 0x4000; /\*Allocate 4K for the Stack.\*/

\_stack\_top = 0x20008000; /\*Address of the top of the heap, also end of RAM.\*/

}

**startup.c** 定義了棧指針和函數入口main

\_\_attribute\_\_ ((section(".isr\_vector")))void (\*g\_pfnVectors[])(void) =

{

0x20008000, // StackPtr, set in RestetISR

main, // The reset handler

NmiSR, // The NMI handl

main函數裡就是往UART0裡輸出了Hello My SOC

#include <stdint.h>

static volatile uint32\_t \* const UART0\_DR = (uint32\_t \*)0x40000000;

void puts(char \*str)

{

while(\*str != 0) {

\*UART0\_DR = \*str;

str++;

}

}

int main()

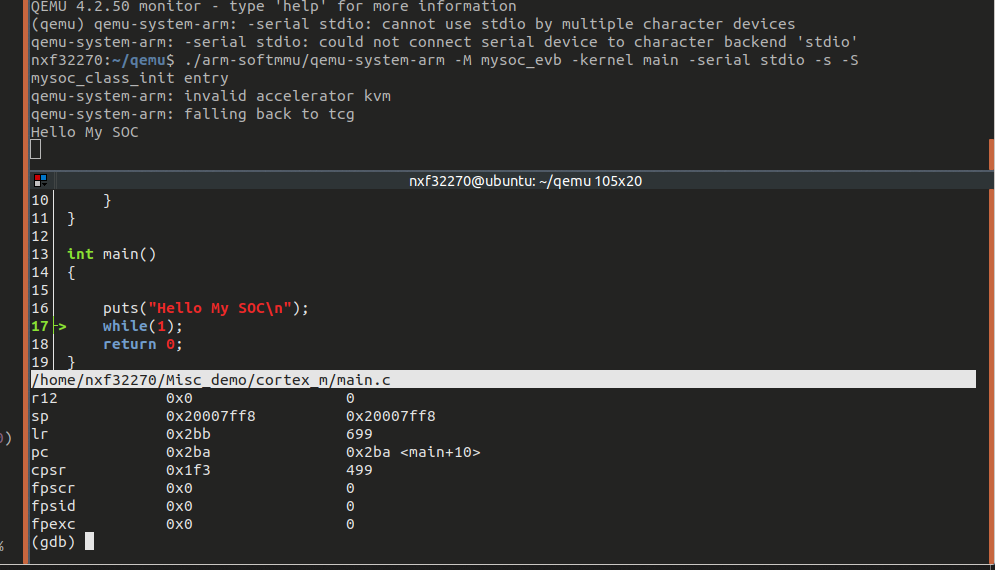
{

puts("Hello My SOC\n");

while(1);

return 0;

}

編譯運行看結果  


**4 創建自己的IP**

qemu/hw下面已經內置了許多可用的外設IP，如果有就可以簡單地直接在像搭積木一樣地配置一下就行。但如果在SOC開發中加入了一些自研的IP， 而此時在hw下面並沒有，這個時候就需要自己加入IP的模擬器代碼。本小節以一個最簡單的讀寫寄存器來示範如何在qemu模擬器中加入自己的IP。

**4.1 memory map**

| **register** | **offset** | **reset\_value** | **R/W** |
| --- | --- | --- | --- |
| ID0 | 0x0 | 0x54 (T) | RO |
| ID1 | 0x4 | 0045 (E) | RO |
| ID2 | 0x8 | 0x53 (S) | RO |
| ID3 | 0xc | 0x54 (T) | RO |
| ID4 | 0x10 | 0x20 (Space) | RO |
| ID5 | 0x14 | 0x40 (I) | RO |
| ID6 | 0x18 | 0x50 § | RO |
| Test Reg | 0x1c | 0 | RO |

寄存器功能很簡單，前面7個是只讀ID寄存器，最後一個是可讀寫的寄存器，沒什麼實際作用，就是demo用。沒有中斷產生。

**4.2 my\_test\_ip源碼**

#define MY\_TEST\_IP\_START (0x40001000)

#define NUM\_IRQ\_LINES 64

typedef struct {

SysBusDevice parent\_obj;

qemu\_irq irq;

MemoryRegion iomem;

uint32\_t id0; //T

uint32\_t id1; //E

uint32\_t id2; //S

uint32\_t id3; //T

uint32\_t id4; //

uint32\_t id5; //I

uint32\_t id6; //P

uint32\_t test\_reg;

} my\_test\_ip\_state;

#define TEST\_IP(obj) \

OBJECT\_CHECK(my\_test\_ip\_state, (obj), TYPE\_TEST\_IP)

static const VMStateDescription my\_test\_ip\_vm = {

.name = "my\_test\_ip",

.version\_id = 1,

.minimum\_version\_id = 1,

.fields = (VMStateField[]) {

VMSTATE\_UINT32(id0, my\_test\_ip\_state),

VMSTATE\_UINT32(id1, my\_test\_ip\_state),

VMSTATE\_UINT32(id2, my\_test\_ip\_state),

VMSTATE\_UINT32(id3, my\_test\_ip\_state),

VMSTATE\_UINT32(id4, my\_test\_ip\_state),

VMSTATE\_UINT32(id5, my\_test\_ip\_state),

VMSTATE\_UINT32(id6, my\_test\_ip\_state),

VMSTATE\_UINT32(test\_reg, my\_test\_ip\_state),

VMSTATE\_END\_OF\_LIST()

}

};

static uint64\_t my\_test\_ip\_read(void \*opaque, hwaddr offset,

unsigned size)

{

uint64\_t ret = 0;

my\_test\_ip\_state \*s = (my\_test\_ip\_state \*)opaque;

printf("%s hwaddr:%lx, size:%x\n", \_\_func\_\_, offset, size);

switch (offset) {

case 0x0:

ret = s->id0;

break;

case 0x4:

ret = s->id1;

break;

case 0x8:

ret = s->id2;

break;

case 0xc:

ret = s->id3;

break;

case 0x10:

ret = s->id4;

break;

case 0x14:

ret = s->id5;

break;

case 0x18:

ret = s->id6;

break;

case 0x1c:

ret = s->test\_reg;

break;

}

return ret;

}

static void my\_test\_ip\_write(void \*opaque, hwaddr offset,

uint64\_t value, unsigned size)

{

my\_test\_ip\_state \*s = (my\_test\_ip\_state \*)opaque;

printf("%s hwaddr:%lx, size:%x, value:%lx\n", \_\_func\_\_, offset, size, value);

switch(offset){

case 0x0:

case 0x4:

case 0x8:

case 0xc:

case 0x10:

case 0x14:

case 0x18:

printf("%s: cannot write the read only register\n", \_\_func\_\_);

break;

case 0x1c:

s->test\_reg = value;

break;

}

}

static const MemoryRegionOps my\_test\_ip\_ops = {

.read = my\_test\_ip\_read,

.write = my\_test\_ip\_write,

.endianness = DEVICE\_NATIVE\_ENDIAN,

};

static void my\_test\_ip\_init(Object \*obj)

{

printf("%s \n", \_\_func\_\_);

my\_test\_ip\_state \*s = TEST\_IP(obj);

memory\_region\_init\_io(&s->iomem, obj, &my\_test\_ip\_ops, s, TYPE\_TEST\_IP, 0x1000);

sysbus\_init\_mmio(SYS\_BUS\_DEVICE(obj), &s->iomem);

s->id0 = 0x54;

s->id1 = 0x45;

s->id2 = 0x53;

s->id3 = 0x54;

s->id4 = 0x20;

s->id5 = 0x49;

s->id6 = 0x50;

s->test\_reg = 0;

}

static void my\_test\_ip\_class\_init(ObjectClass \*klass, void \*data)

{

printf("%s \n", \_\_func\_\_);

DeviceClass \*dc = DEVICE\_CLASS(klass);

dc->vmsd = &my\_test\_ip\_vm;

}

static const TypeInfo my\_test\_ip = {

.name = TYPE\_TEST\_IP,

.parent = TYPE\_SYS\_BUS\_DEVICE,

.instance\_size = sizeof(my\_test\_ip\_state),

.instance\_init = my\_test\_ip\_init,

.class\_init = my\_test\_ip\_class\_init,

};

static void my\_test\_ip\_types(void)

{

printf("%s \n", \_\_func\_\_);

type\_register\_static(&my\_test\_ip);

}

type\_init(my\_test\_ip\_types)

**4.3 測試代碼**

#define MY\_TEST\_IP\_START 0x40001000

typedef struct my\_ip\_tag {

volatile uint32\_t id0;

volatile uint32\_t id1;

volatile uint32\_t id2;

volatile uint32\_t id3;

volatile uint32\_t id4;

volatile uint32\_t id5;

volatile uint32\_t id6;

volatile uint32\_t test\_reg;

} my\_test\_ip\_t;

void my\_test\_ip\_sample()

{

my\_test\_ip\_t \*ip = (my\_test\_ip\_t \*)MY\_TEST\_IP\_START;

char id[8];

id[0] = (char) ip->id0;

id[1] = (char) ip->id1;

id[2] = (char) ip->id2;

id[3] = (char) ip->id3;

id[4] = (char) ip->id4;

id[5] = (char) ip->id5;

id[6] = (char) ip->id6;

id[7] = 0;

puts(id);

puts("\n");

ip->test\_reg = 0x00414141;

id [0] = ip->test\_reg & 0xff;

id [1] = (ip->test\_reg & 0xff00) >> 8;

id [2] = (ip->test\_reg & 0xff0000) >> 16;

id [3] = (ip->test\_reg & 0xff000000) >> 24;

puts(id);

puts("\n");

}

運行結果： 紅框內是qemu打印的，藍框是測試程序打印出來的，可以看到能順利讀出ID，同時能讀寫test reg。  
