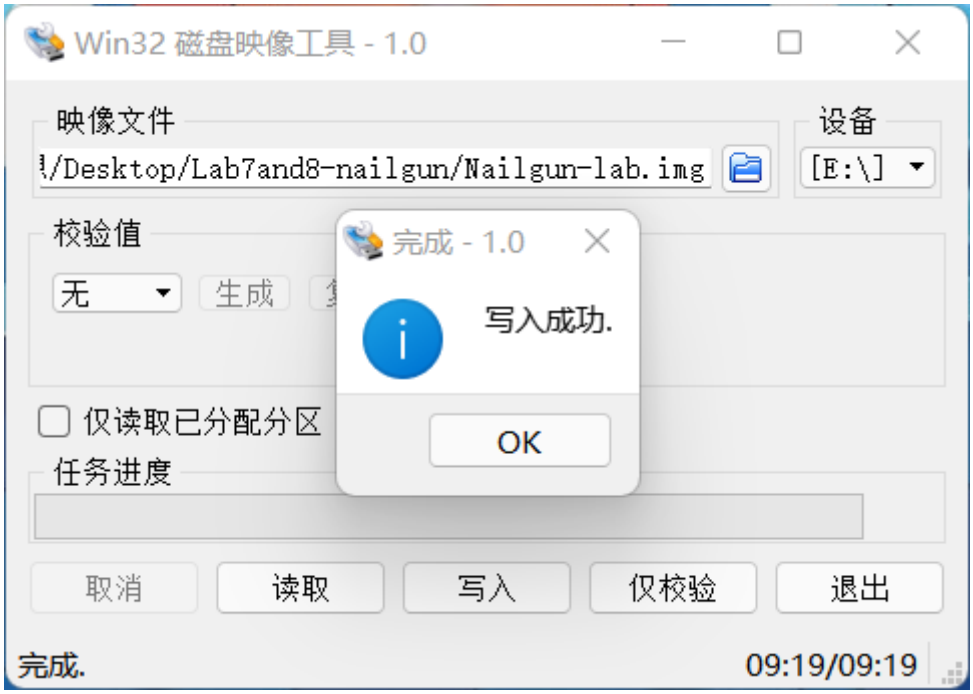
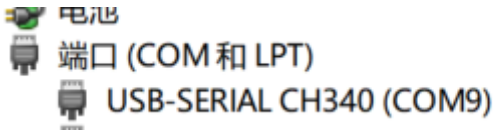
# Lab7

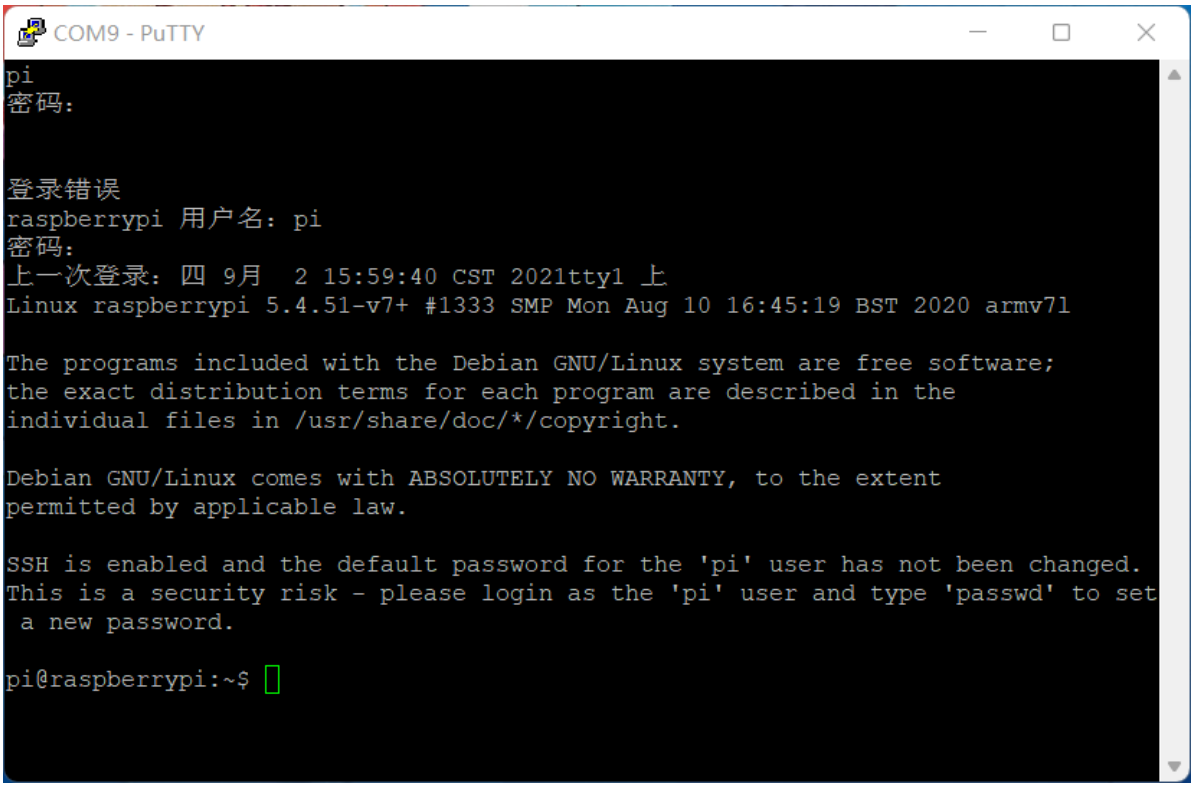
## Connect the Raspberry Pi




Service Manage recognize it sucessfully




log in it:




Write simple_module.c and makefile:

```
pi@raspberrypi:~/workspace$ ls
Makefile  simple_module.c
pi@raspberrypi:~/workspace$
```

make and run it:

```
pi@raspberrypi:~/workspace$ make
make -C /lib/modules/5.4.51-v7+/build M=/home/pi/workspace modules
make[1]: 进入目录"/usr/src/linux-headers-5.4.51-v7+"
  CC [M]  /home/pi/workspace/simple_module.o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /home/pi/workspace/simple_module.o
see include/linux/module.h for more information
  CC [M]  /home/pi/workspace/simple_module.mod.o
  LD [M]  /home/pi/workspace/simple_module.ko
make[1]: 离开目录"/usr/src/linux-headers-5.4.51-v7+"
pi@raspberrypi:~/workspace$
```

```
283 [ 1272.936297] simple module: loading out-of-tree module taints kernel.
284 [ 1272.936315] simple module: module license 'unspecified' taints kernel.
285 [ 1272.936321] Disabling lock debugging due to kernel taint
286 [ 1272.937985] Hello world.
287 [ 1304.749963] Exit.
288
```

## Task 1: Question: What is the exception level and the security state of the core with loaded LKM?

```
Exeception level:
    EL1: OS kernel and other privileged code
security state:
    non secure
```

## Task 2.a: What is the complete instruction?

```
static int __init simple_module_init(void)
{
    //printk("Hello world.\n");
    uint32_t reg;
    asm volatile("mrc p15, 0, %0, c1, c1, 0":"=r"(reg));
    printk(KERN_INFO "SCR %x.\n", reg);
    return 0;
}
```

Compile and execute:

```
Message from syslogd@raspberrypi at Sep  2 17:04:11 ...
 kernel:[ 3886.676113] ffe0: 7efd1bc8 7efd1bb8 00022cb8 76beaaf0 60000010 00000003 0

Message from syslogd@raspberrypi at Sep  2 17:04:11 ...
 kernel:[ 3886.785526] Code: e8bd4000 e309003c e3470f15 eb45e84e (ee111f11)
段错误
```

dmesg:

] Exception stack(0xb664ffa8 to 0xb664fff0)
] ffa0:                   77023e00 7efd1d94 00000003 0002d064 00000000 00000004
] ffc0: 77023e00 7efd1d94 0003fce8 0000017b 0191e7e0 00000000 00000002 00000000
] ffe0: 7efd1bc8 7efd1bb8 00022cb8 76beaaf0
]  r9:b664e000 r8:801011c4 r7:0000017b r6:0003fce8 r5:7efd1d94 r4:77023e00
] Code: e8bd4000 e309003c e3470f15 eb45e84e (ee111f11)
] ---[ end trace 39dac41b9227c3d9 ]---

## Task 2.b: Question: Why the segmentation fault occurs

```
It's because the Privilge Level is not enough to access SCR in EL1 level.
SCR is noly accessible at secure state.
```

## Task 3.a: Question: What is the instruction to read DBGAUTHSTATUS? Suppose we want to store it in Rt.

### Accessing DBGAUTHSTATUS

Accesses to this register use the following encodings in the System register encoding space:

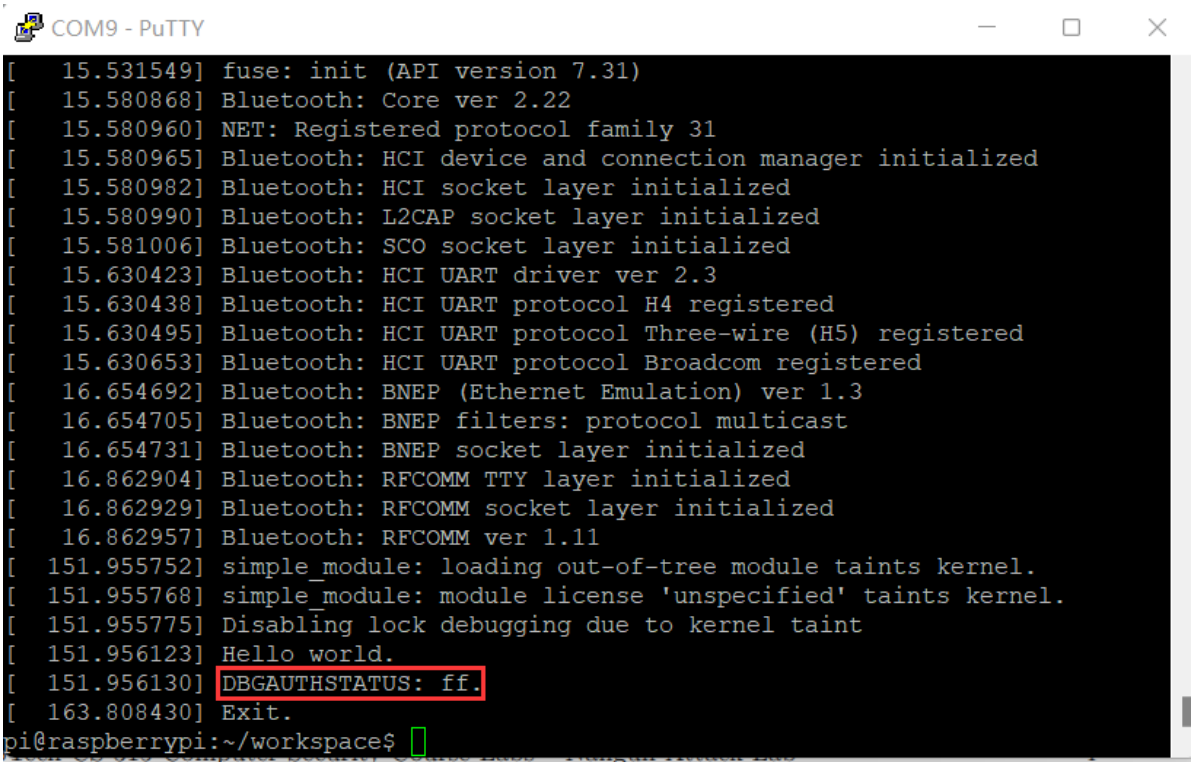
***MRC{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}***

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 0b1110 | 0b000 | 0b0111 | 0b1110 | 0b110 |

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif EL2Enabled() && !ELUsingAArch32(EL2) && MDCR_EL2.<TDE,TDA> != '00' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x05);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HDCR.<TDE,TDA> != '00' then
        AArch32.TakeHypTrapException(0x05);
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        if Halted() && EDSCR.SDD == '1' then
            UNDEFINED;
        else
            AArch64.AArch32SystemAccessTrap(EL3, 0x05);
    else
        return DBGAUTHSTATUS;
elsif PSTATE.EL == EL2 then
    if Halted() && HaveEL(EL3) && EDSCR.SDD == '1' && boolean IMPLEMENTATION_DEFINED "EL3 trap priority
when SDD == '1'" && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
        UNDEFINED;
    elsif HaveEL(EL3) && !ELUsingAArch32(EL3) && MDCR_EL3.TDA == '1' then
```

```
mrc p14, 0, R0, c7, c6, 6
```

## Task 3.b: Take single_module.c as an example, write your kernel module to read the signal. A screenshot of the result is needed, and tell us what is the meaning of the result. What kind of debug events are enabled?




| Bits | |
|---|---|
| | [31:8] RES0. |
| | [7:6] SNID, Secure non-invasive debug. |
| | [5:4] SID, Secure invasive debug. |
| | [3:2] NSNID, Non-secure non-invasive debug. |
| | [1:0] NSID, Non-secure invasive debug. |

```
So 0xff means all four debug are enabled. That is:
DBGEN: invasive debug enable
SPIDEN: secure invasive debug enable
SPNIDEN: secure non-invasive debug enable
NIDEN: non-invasive debug enable
```

## Task 4:

**OSLA, bits [31:0]**

OS Lock Access. Writing the value 0xC5ACCE55 to the DBGOSLAR sets the OS Lock to 1. Writing any other value sets the OS Lock to 0.

Use DBGOSLSR.OSLK to check the current status of the lock.

### Accessing DBGOSLAR

Accesses to this register use the following encodings in the System register encoding space:

***MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}***

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 0b1110 | 0b000 | 0b0001 | 0b0000 | 0b100 |

- OSLAR_EL1 - OS Lock Access Register - offset 0x300

  This register controls the OS Lock. Unlocking the OS Lock allows you to access the debug registers. This is a write only regist

### Field descriptions

***When Software Lock is implemented:***


**KEY, bits [31:0]**

Lock Access control. Writing the key value 0xC5ACCE55 to this field unlocks the lock, enabling write accesses to this component's registers through a memory-mapped interface.

Writing any other value to this register locks the lock, disabling write accesses to this component's registers through a memory mapped interface.

***Otherwise:***


Otherwise

**Bits [31:0]**

Reserved, RES0.

### Accessing the EDLAR:

EDLAR can be accessed through its memory-mapped interface:

| Component | Offset | Instance |
|---|---|---|
| Debug | 0xFB0 | EDLAR |

So the complete code is:

```
#define OSLAR_OFFSET 0x300
#define EDLAR_OFFSET 0xFB0

iowrite32(0xC5ACCE55, param->debug_register + EDLAR_OFFSET);
iowrite32(0xC5ACCE55, param->cti_register + EDLAR_OFFSET);
iowrite32(0x1, param->debug_register + OSLAR_OFFSET);
iowrite32(0x1, param->cti_register + OSLAR_OFFSET);
```

### Task 5.a: We mention how to access SCR directly in Task 2. You need to prepare an instruction, who reads SCR and store it to R1. Then convert it to machine code (do it on yourself) and execute it on the target.

```
// 0x1f11ee11 <=>mrc p15, 0, R1, c1, c1, 0  ==>store in R1
execute_ins_via_itr(param->debug_register, 0x1f11ee11);
```

### Task 5.b: After you finish Task 5.a, you need to transfer the value in R1 on core 0, to the local variable scr. It will be printed later. DBGDTRTXint and DBGDTRTX would be helpful in your implementation.

**Accessing DBGDTRTXint**

Data can be loaded from memory into this register using LDC (immediate) and LDC (literal).

Accesses to this register use the following encodings in the System register encoding space:

***MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}***

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 0b1110 | 0b000 | 0b0000 | 0b0101 | 0b000 |

```
//0x1e15ee00 <=> mcr p14,0,R1,c0,c5,0
execute_ins_via_itr(param->debug_register, 0x1e15ee00);
scr = ioread32(param->debug_register + DBGDTRTX_OFFSET);
```

### Task 6: Build and run nailgun.ko, and see what happen. Explain the value of SCR.

```
pi@raspberrypi:~/zxy_workspace/nailgun$ make
make -C /lib/modules/5.4.51-v7+/build M=/home/pi/zxy_workspace/nailgun modules
make[1]: 进入目录“/usr/src/linux-headers-5.4.51-v7+”
  CC [M]  /home/pi/zxy_workspace/nailgun/nailgun.o
  Building modules, stage 2.
  MODPOST 1 modules
WARNING: modpost: missing MODULE_LICENSE() in /home/pi/zxy_workspace/nailgun/nailgun.o
see include/linux/module.h for more information
  CC [M]  /home/pi/zxy_workspace/nailgun/nailgun.mod.o
  LD [M]  /home/pi/zxy_workspace/nailgun/nailgun.ko
make[1]: 离开目录“/usr/src/linux-headers-5.4.51-v7+”
pi@raspberrypi:~/zxy_workspace/nailgun$ sudo rmmod nailgun
pi@raspberrypi:~/zxy_workspace/nailgun$ sudo insmod nailgun.ko
pi@raspberrypi:~/zxy_workspace/nailgun$ dmesg
```

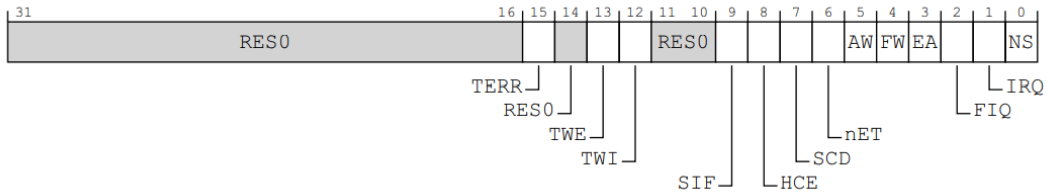
```
[ 2718.833911] Step 1: Unlock debug and cross trigger registers
[ 2718.833921] Step 2: Enable halting debug
[ 2718.833926] Step 3: Halt the target processor
[ 2718.833932] Step 4: Wait the target processor to halt
[ 2718.833936] Step 5: Save context
[ 2718.833942] Step 6: Switch to EL3
[ 2718.833946] Step 7: Read SCR
[ 2718.833951] Step 8: Restore context
[ 2718.833956] Step 9: Send restart request to the target processor
[ 2718.833961] Step 10: Wait the target processor to restart
[ 2718.833968] All done! The value of SCR is 0x00000131
pi@raspberrypi:~/zxy_workspace/nailgun$
```

The result:

```
0x0131 ==>
0x0000 0001 0011 0001
```

```
0(NS)-1:PE is in Non-secure state.
1(IRQ)-0:IRQs taken to IRQ mode
2(FIQ)-0:FIQs taken to FIQ mode
3(EA)-0:External aborts taken to Abort mode.
4(FW)-1: An FIQ taken from either Security state is masked by PSTATE.F. When
PSTATE.F is 0, the FIQ is taken to EL3.
5(AW)-1:External aborts taken from either Security state are masked by PSTATE.A.
When PSTATE.A is 0, the abort is taken to EL3.
6(nET)-0:Early termination permitted. Execution time of data operations can
depend on the data values.
7(SCD)-0:SMC instructions are enabled
8(HCE)-1:HVC instructions are enabled at Non-secure EL1 and EL2.
9(SIF)-0:Secure state instruction fetches from Non-secure memory are permitted
Bits [11:10]:Reserved, RES0
12(TWI)-0:This control does not cause any instructions to be trapped
13(TWE)-0:This control does not cause any instructions to be trapped
Bit [14]:Reserved, RES0.
15(TERR)-0:This control does not cause any instructions to be trapped.
Bits [31:16] : Reserved, RES0
```

**Field descriptions**




## Questions:

1. **During this lab, what is the base address of Cross Trigger Interface in Raspberry Pi 3? Can your find the global address of CTICONTROL register in Raspberry Pi 3 according to the Arm Reference Manual? Answer the address value and show your calculation. (hint: Find the offset)**

```
// 0x40038000 is the base address of the cross trigger interface registers on Core 0
#define CTI_REGISTER_ADDR 0x40038000
#define CTI_REGISTER_SIZE 0x1000
```

base address: 0x40038000

**Accessing the CTICONTROL:**

CTICONTROL can be accessed through the external debug interface:

| Component | Offset | Instance |
|---|---|---|
| CTI | 0x000 | CTICONTROL |

the global address of CTICONTROL register = 0x40038000+0x000=0x40038000

2.**Do we have another way to unlock the OS Lock in this lab except memory mapping? If yes, how to do that? Justify your answer.**

Use the instruction to unlock directly.

### Accessing DBGOSLAR

Accesses to this register use the following encodings in the System register encoding space:

***MCR{<c>}{<q>} <coproc>, {#}<opc1>, <Rt>, <CRn>, <CRm>{, {#}<opc2>}***

| coproc | opc1 | CRn | CRm | opc2 |
|---|---|---|---|---|
| 0b1110 | 0b000 | 0b0001 | 0b0000 | 0b100 |

```
asm volatile("mcr p14, 0, R0, c1, c0, 4");
```

```
asm volatile("mcr p14, 0, R0, c1, c0, 4");
// iowrite32(0x1, param->debug_register + OSLAR_OFFSS

        // iowrite32(0x1, param->cti_register + OSLARR
```

run it and get the result:

```
1945.292804] Step 7: Read SCR
1945.295679] Step 8: Restore context
1945.299164] Step 9: Send restart request to the target processor
1945.305165] Step 10: Wait the target processor to restart
1945.310561] All done! The value of SCR is 0x00000131
i@raspberrypi:~/zxy_workspace/nailgun$
```

Happy ending!