Lab 7 Writeup

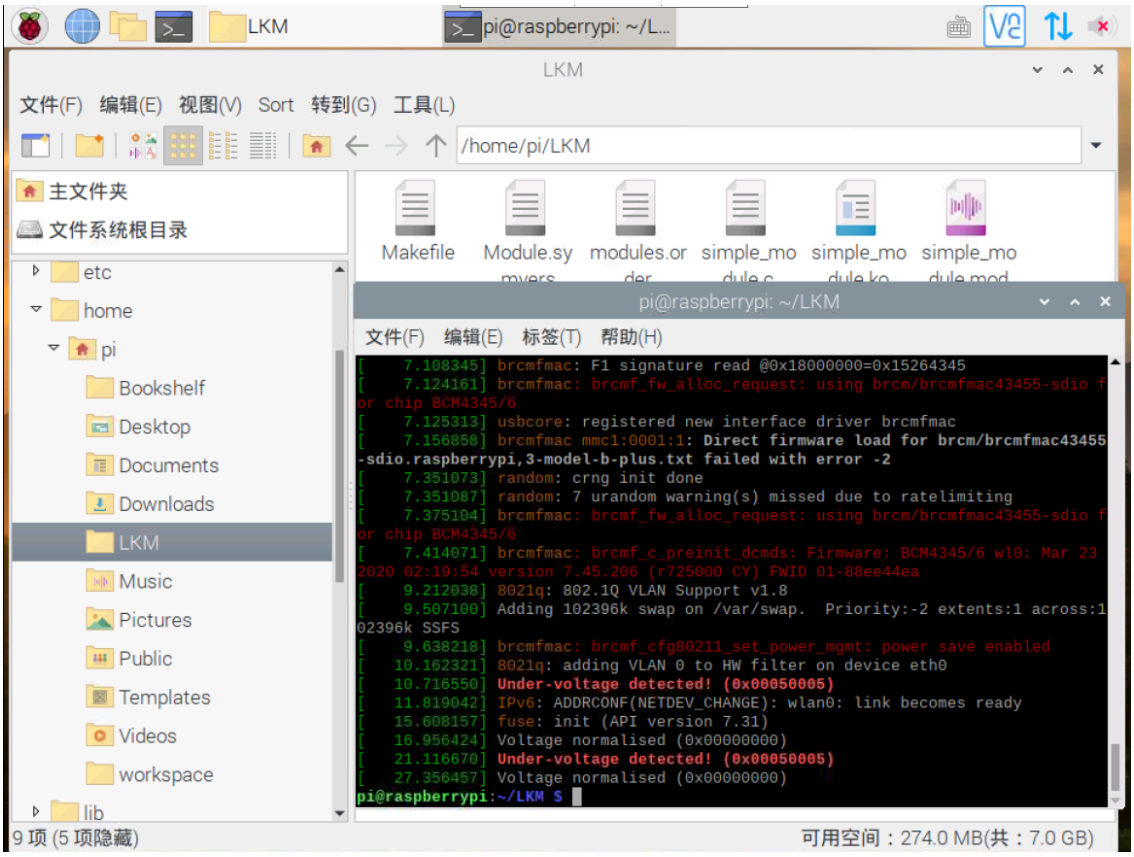
孙永康　11911409

Task 1

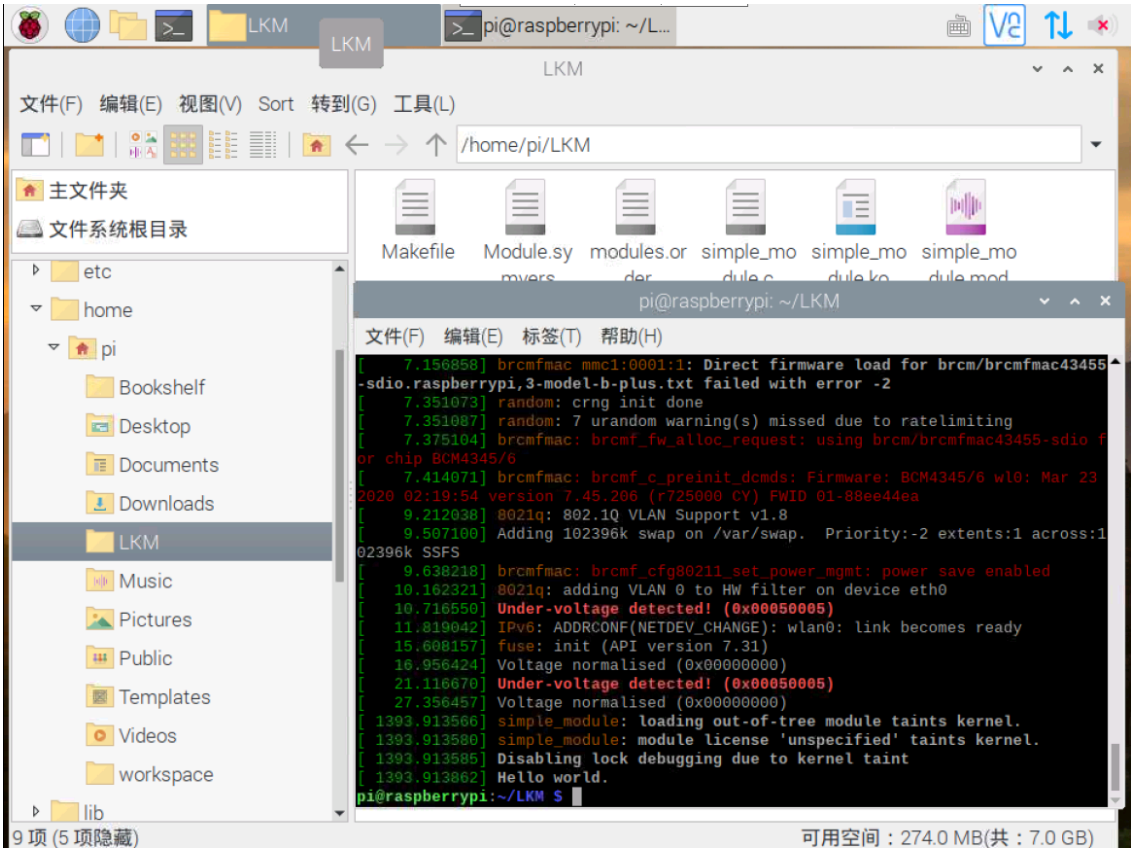
After copying the c code into `simple_module.c` file, I compile and execute it in the following order.

```
$ ls
simple_moduel.c Makefile
$ make
$ dmesg
$ sudo insmod simple_module.ko
$ dmesg
$ sudo rmmod simple_module
```

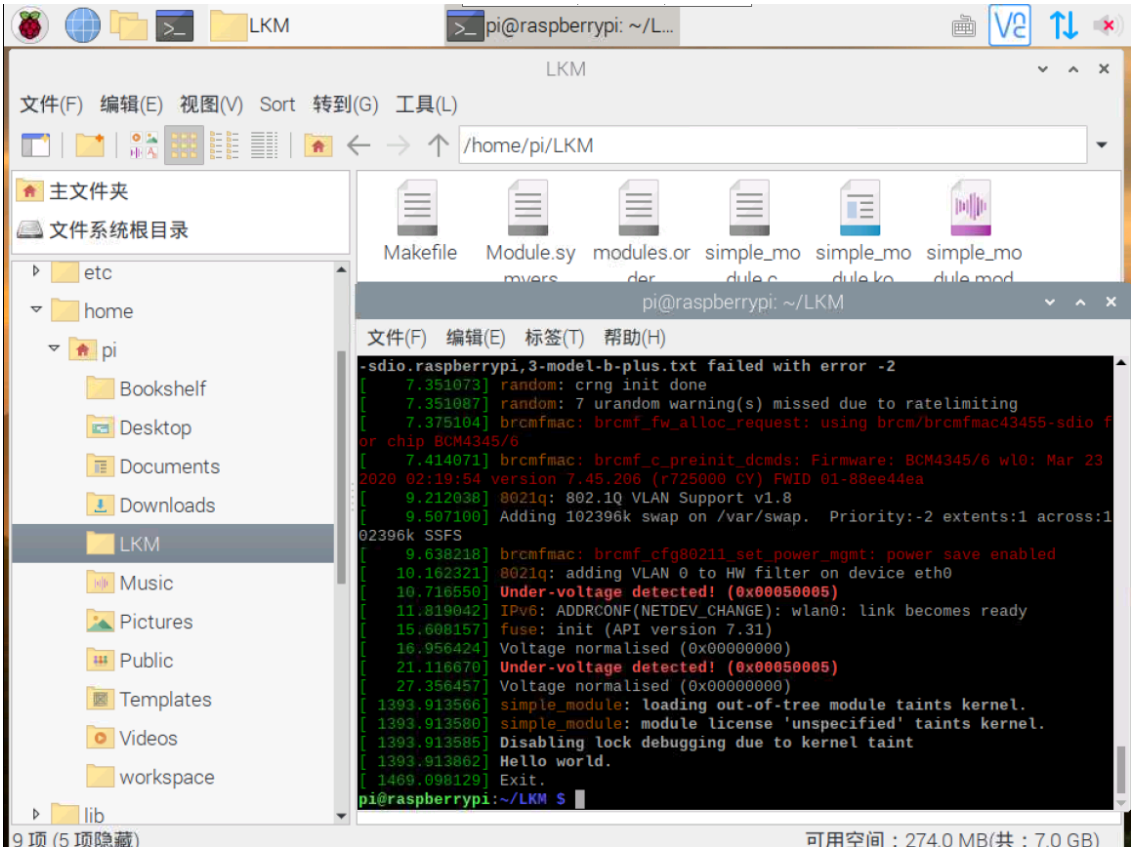
1. `dbmsg` after `make`



2. `dbmsg` after `sudo insmod simple_module.ko`



3. `dbmsg` after `sudo rmmod simple_module`



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

Answer : The exception level is EL2 and security state is non-secure state.

Task 2

a.

This is the code I modified. It is according to the manual.

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

Accessing SCR

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
0b1111	0b000	0b0001	0b0001	0b000

```
if PSTATE.EL == EL0 then
    UNDEFINED;
elsif PSTATE.EL == EL1 then
    if EL2Enabled() && !ELUsingAArch32(EL2) && HSTR_EL2.T1 == '1' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif EL2Enabled() && ELUsingAArch32(EL2) && HSTR.T1 == '1' then
        AArch32.TakeHypTrapException(0x03);
    elsif !ELUsingAArch32(EL2) && SCR_EL3.<NS,EEL2> == '01' then
        AArch64.AArch32SystemAccessTrap(EL2, 0x03);
    elsif !ELUsingAArch32(EL3) && SCR_EL3.NS == '0' then
        AArch64.AArch32SystemAccessTrap(EL3, 0x03);
    else
        UNDEFINED;
elsif PSTATE.EL == EL2 then
    UNDEFINED;
elsif PSTATE.EL == EL3 then
    return SCR;
```

After compiling it I receive a segment fault.

```
pi@raspberrypi:~/LKM $ sudo insmod simple_module.ko
段错误
```

After checking it through `dbmsg`, I found that the `register $PC` stop at 0x14. I do not know how to find the instructions that cause this segment fault. So, I ask my friend for help, he disassemble this .o file and find the 14th instructions is `mrc p15, 0, %0, c1, c1, 0`.

```
Hardware name: BCM2835
PC is at simple_module_init+0x14/0x1000 [simple_module]
LR is at do_one_initcall+0x50/0x23c
pc : [<7f006014>]    lr : [<801031c8>]    psr: 60000013
sp : b257dd88  ip : b257dd98  fp : b257dd94
r10: b5fa6f40  r9 : b5fa6f00  r8 : 00000000
r7 : 80d04f48  r6 : b257c000  r5 : 7f006000  r4 : 7f199000
r3 : 6b6f6758  r2 : 6b6f6758  r1 : 35ec4000  r0 : 00000000
Flags: nZCv  IRQs on  FIQs on  Mode SVC_32  ISA ARM  Segment user
Control: 10c5383d  Table: 179c006a  DAC: 00000055
```

b.

So, the reason this instruction could cause segment fault is that SCR can only be accessed in **secure state** while our LKM is running under **non-secure state**.

Task 3

a.

The instruction is in the below figure. This is according to the manual.

```
/* simple_module.c */
#include <linux/module.h> // included for all kernel modules
#include <linux/kernel.h> // included for KERN_INFO
#include <linux/init.h> // included for __init and __exit macros
#include <asm/io.h>
/*
* The init function of the module.
*/
static int __init simple_module_init(void)
{
uint32_t reg;
asm volatile("mrc p14, 0, %0, c7, c14, 6":"=r"(reg));
printk(KERN_INFO "DBGAUTHSTATUS: %x.\n", reg);
return 0;
}
/*
* The cleanup function of the module.
*/
static void __exit simple_module_exit(void)
{
printk(KERN_INFO "Exit.\n");
}
module_init(simple_module_init);
"simple_module.c" 25L, 621C                                   1,1           顶端
```

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

b.

After compiling and running in LKM, we receive this feedback.

```
[    5.900924] bcm2835-codec bcm2835-codec: Loaded V4L2 encode
[    5.924954] bcm2835-codec bcm2835-codec: Device registered as /dev/video12
[    5.925099] bcm2835-codec bcm2835-codec: Loaded V4L2 isp
[    6.050841] cfg80211: Loading compiled-in X.509 certificates for regulatory database
[    6.209200] cfg80211: Loaded X.509 cert 'sforshee: 00b28ddf47aef9cea7'
[    6.295443] brcmfmac: F1 signature read @0x18000000=0x15264345
[    6.308803] brcmfmac: brcmf_fw_alloc_request: using brcm/brcmfmac43455-sdio for chip BCM4345/6
[    6.309719] usbcore: registered new interface driver brcmfmac
[    6.344614] brcmfmac mmc1:0001:1: Direct firmware load for brcm/brcmfmac43455-sdio.raspberrypi
,3-model-b-plus.txt failed with error -2
[    6.560646] random: crng init done
[    6.560659] random: 7 urandom warning(s) missed due to ratelimiting
[    6.573109] brcmfmac: brcmf_fw_alloc_request: using brcm/brcmfmac43455-sdio for chip BCM4345/6
[    6.613659] brcmfmac: brcmf_c_preinit_dcmds: Firmware: BCM4345/6 wl0: Mar 23 2020 02:19:54 ver
sion 7.45.206 (r725000 CY) FWID 01-88ee44ea
[    8.789900] 8021q: 802.1Q VLAN Support v1.8
[    9.046605] Adding 102396k swap on /var/swap.  Priority:-2 extents:1 across:102396k SSFS
[    9.170331] brcmfmac: brcmf_cfg80211_set_power_mgmt: power save enabled
[    9.819003] 8021q: adding VLAN 0 to HW filter on device eth0
[   11.364379] IPv6: ADDRCONF(NETDEV_CHANGE): wlan0: link becomes ready
[   15.270335] Bluetooth: Core ver 2.22
[   15.270499] NET: Registered protocol family 31
[   15.270525] Bluetooth: HCI device and connection manager initialized
[   15.270566] Bluetooth: HCI socket layer initialized
[   15.270608] Bluetooth: L2CAP socket layer initialized
[   15.270678] Bluetooth: SCO socket layer initialized
[   15.282411] Bluetooth: HCI UART driver ver 2.3
[   15.282426] Bluetooth: HCI UART protocol H4 registered
[   15.282471] Bluetooth: HCI UART protocol Three-wire (H5) registered
[   15.282637] Bluetooth: HCI UART protocol Broadcom registered
[   15.566298] Bluetooth: BNEP (Ethernet Emulation) ver 1.3
[   15.566314] Bluetooth: BNEP filters: protocol multicast
[   15.566344] Bluetooth: BNEP socket layer initialized
[   15.658005] Bluetooth: RFCOMM TTY layer initialized
[   15.658032] Bluetooth: RFCOMM socket layer initialized
[   15.658056] Bluetooth: RFCOMM ver 1.11
[   16.304903] fuse: init (API version 7.31)
[   81.986689] simple_module: loading out-of-tree module taints kernel.
[   81.986708] simple_module: module license 'unspecified' taints kernel.
[   81.986715] Disabling lock debugging due to kernel taint
[   81.987071] DBGAUTHSTATUS: ff.
pi@raspberrypi:~/LKM$
```

According to the manual, the result `ff`, means that both those 4 debug mode, **Secure Non-Invasive Debug** , **Secure Invasive Debug** , **Non-Secure Non-Invasive Debug** , **Non-Secure Invasive Debug** , are all implemented and enabled.

SNID, bits [7:6]

When FEAT_Debugv8p4 is implemented:

SNID

Secure Non-Invasive Debug.

This field has the same value as DBGAUTHSTATUS.SID.

Otherwise:

SNID

Secure Non-Invasive Debug.

0b00	Not implemented. EL3 is not implemented and the Effective value of SCR.NS is 1.
0b10	Implemented and disabled. ExternalSecureNoninvasiveDebugEnabled() == FALSE.
0b11	Implemented and enabled. ExternalSecureNoninvasiveDebugEnabled() == TRUE.

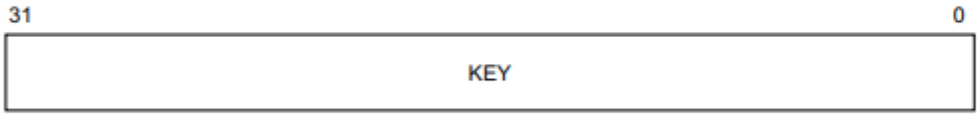
All other values are reserved.

Task 4

Below is what I found in the manual about the `EDLAR`.

Field descriptions

The EDLAR bit assignments are:



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.

Accessing the EDLAR:

EDLAR can be accessed through a memory-mapped interface access to the external debug interface:

Component	Offset
Debug	`0xFB0`

Below is what I found in the manual about the `OSLAR`.

Field descriptions

The OSLAR_EL1 bit assignments are:

Bits [31:1]

Reserved, RES0.

OSLK, bit [0]

On writes to OSLAR_EL1, bit[0] is copied to the OS lock.

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

Accessing the OSLAR_EL1:

OSLAR_EL1 can be accessed through the external debug interface:

Component	Offset
Debug	`0x300`

So I modify this code according to it :

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

```
iowrite32(0xC5ACCE55, param->debug_register + EDLAR_OFFSET);
iowrite32(0xC5ACCE55, param->cti_register + EDLAR_OFFSET);
iowrite32(0x00000000, param->debug_register + OSLAR_OFFSET);
iowrite32(0x00000000, param->cti_register + OSLAR_OFFSET);
```

Task 5

a.

This is the code I modified. The first instructions is quite the same in task2, but the register is changed to R1 respect to the question.

b.

The second instruction is copied from step 5, which is a good example about how to put it to local variable `scr`. The only difference is that the first register is changed to R1.

```
// Step 7: Read the SCR
printk(KERN_INFO "Step 7: Read SCR\n");
// 0xee111f11 <=> mrc p15, 0, R1, C1, C1, 0
execute_ins_via_itr(param->debug_register, 0x1f11ee11);
// 0xee001e15 <=> mcr p14, 0, R1, C0, C5, 0
execute_ins_via_itr(param->debug_register, 0x1e15ee00);
scr = ioread32(param->debug_register + DBGDTRTX_OFFSET);
```

Task 6

This is what I receive after building and running this code. The `SCR` value is **0x00000131**. Convert it into binary, **0000 0000 0000 0000 0000 0001 0011 0001**.

```
Step 1: Unlock debug and cross trigger registers
Step 2: Enable halting debug
Step 3: Halt the target processor
Step 4: Wait the target processor to halt
Step 5: Save context
Step 6: Switch to EL3
Step 7: Read SCR
Step 8: Restore context
Step 9: Send restart request to the target processor
Step 10: Wait the target processor to restart
All done! The value of SCR is 0x00000131
```

The manual explain what each bit means. Since it is too long, I can not put a screen shot here. So, this `SCR` means :

1. `TERR, bit [15]` is 0 : Trap Error record accesses. Generate a Monitor Trap exception on accesses to some of the registers from modes other than Monitor mode. When it is 0, this control does not cause any instructions to be trapped.
2. `TWE, bit [13]` is 0 : Traps WFE instructions to Monitor mode. When it is 0, this control does not cause any instructions to be trapped.
3. `TWI, bit [12]` is 0 : Traps WFI instructions to Monitor mode. When it is 0, this control does not cause any instructions to be trapped.

4. `SIF, bit [9]` is 0 : Secure instruction fetch. When the PE is in Secure state, this bit disables instruction fetch from Non-secure memory. When it is 0, secure state instruction fetches from Non-secure memory are permitted.
5. `HCE, bit [8]` is 1 : Hypervisor Call instruction enable. If EL2 is implemented, enables execution of HVC instructions at Non-secure EL1 and EL2. When it is 1, HVC instructions are enabled at Non-secure EL1 and EL2.
6. `SCD, bit [7]` is 0 : Secure Monitor Call disable. Disables SMC instructions. When it is 0, SMC instructions are enabled.
7. `nET, bit [6]` is 0 : Not Early Termination. This bit disables early termination. When it is 0, early termination permitted, execution time of data operations can depend on the data values.
8. `AW, bit [5]` is 1 : When the value of SCR.EA is 1 and the value of HCR.AMO is 0, this bit controls whether PSTATE.A masks an External abort taken from Non-secure state. When it is 1, external aborts taken from either Security state are masked by PSTATE.A. When PSTATE.A is 0, the abort is taken to EL3.
9. `FW, bit [4]` is 1 : When the value of SCR.FIQ is 1 and the value of HCR.FMO is 0, this bit controls whether PSTATE.F masks an FIQ interrupt taken from Non-secure state. When it is 1, an FIQ taken from either Security state is masked by PSTATE.F. When PSTATE.F is 0, the FIQ is taken to EL3.
10. `EA, bit [3]` is 0 : External Abort handler. This bit controls which mode takes External aborts and SError interrupt exceptions. When it is 0, External aborts taken to Abort mode.
11. `FIQ, bit [2]` is 0 : FIQ handler. This bit controls which mode takes FIQ exceptions. When it is 0, FIQs taken to FIQ mode.
12. `IRQ, bit [1]` is 0 : IRQ handler. This bit controls which mode takes IRQ exceptions. When it is 0, IRQs taken to IRQ mode.
13. `NS, bit [0]` is 1 : Non-secure bit. Except when the PE is in Monitor mode, this bit determines the Security state of the PE. When it is 1, PE is in Non-secure state.

Submission :

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)

The base address is in the code:

```
#define CTI_REGISTER_ADDR 0x40038000
```

This is the offset of `CTICONTROL` described in the manual.

Accessing the CTICONTROL:

CTICONTROL can be accessed through the external debug interface:

Component	Offset	Instance
CTI	0x000	CTICONTROL

So, the global address of `CTACONTROL` is `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.

Mentioned in the manual, It can also be unlocked by `DBGOSLAR`. Setting it to `0xc5acce55` to lock and to `0x00000000` to unlock.

G8.3.22 DBGOSLAR, Debug OS Lock Access Register

The DBGOSLAR characteristics are:

Purpose

Provides a lock for the debug registers. The OS Lock also disables some debug exceptions and debug events.

Configurations

This register is present only when EL1 is capable of using AArch32. Otherwise, direct accesses to DBGOSLAR are UNDEFINED.

The OS Lock can also be locked or unlocked using the AArch64 System register OSLAR_EL1 and External register OSLAR_EL1.

Attributes

DBGOSLAR is a 32-bit register.

Field descriptions

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

Use `DBGOSLSR` to check OSLOCK's status.

G8.3.23 DBGOSLSR, Debug OS Lock Status Register

The DBGOSLSR characteristics are:

Purpose

Provides status information for the OS Lock.

Configurations

AArch32 System register DBGOSLSR bits [31:0] are architecturally mapped to AArch64 System register OSLSR_EL1[31:0].

This register is present only when AArch32 is supported at EL0. Otherwise, direct accesses to DBGOSLSR are UNDEFINED.

The OS Lock status is also visible in the external debug interface through EDPRSR.

Attributes

DBGOSLSR is a 32-bit register.

OSLK, bit [1]

OS Lock Status. The possible values are:

`0b0` OS Lock unlocked.

`0b1` OS Lock locked.

The OS Lock is locked and unlocked by writing to the OS Lock Access Register.

The reset behavior of this field is:

- On a Cold reset, this field resets to 1.

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

coproc	opc1	CRn	CRm	opc2
0b1110	0b000	0b0001	0b0001	0b100

This is the test code I use, and from the log, we can know that it is successful locked and unlock.

```
uint32_t reg;
asm volatile("mrc p14, 0, %0, c1, c1, 4":"=r"(reg));
printk(KERN_INFO "DBGOSLSR before lock: 0x%.08x.\n", reg);

asm volatile("mcr p14, 0, %0, c1, c0, 4"::"r"(0xC5ACCE55));
printk(KERN_INFO "use DBGOSLAR lock OSLOCK.\n");

asm volatile("mrc p14, 0, %0, c1, c1, 4":"=r"(reg));
printk(KERN_INFO "DBGOSLSR after lock: 0x%.08x.\n", reg);

asm volatile("mcr p14, 0, %0, c1, c0, 4"::"r"(0x00000000));
printk(KERN_INFO "use DBGOSLAR unlock OSLOCK.\n");

asm volatile("mrc p14, 0, %0, c1, c1, 4":"=r"(reg));
printk(KERN_INFO "DBGOSLSR after unlock: 0x%.08x.\n", reg);
return 0;
```

```
DBGOSLSR before lock: 0x00000008.
use DBGOSLAR lock OSLOCK.
DBGOSLSR after lock: 0x0000000a.
use DBGOSLAR unlock OSLOCK.
DBGOSLSR after unlock: 0x00000008.
```