|  |
| --- |
|  |
| Codezero源码分析 |
| ——api目录 |

|  |
| --- |
|  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 当前版本： | 1.0 | | 日期： | 2014-07-07 | | 作者： | 康乔 | |

版本历史

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 版本状态 | 作者 | 参与者 | 起止日期 | 备注 |
| 1.0 | 柯杰伟 |  | 2014-07-07 |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 概述

# 总体功能描述

# Sconstruct文件

# ARM目录

## 文件1

### 功能描述（对于复杂操作建议画出流程图）

### 数据结构

### 代码注释

### 疑问或改进建议

### 体系结构和平台相关的代码

## Exception-common.c

### 功能描述（对于复杂操作建议画出流程图）

### 数据结构

### 代码注释

/\*

\* Common exception handling code

\*

\* Copyright (C) 2008 - 2010 B Labs Ltd.

\* Written by Bahadir Balban

\*/

#include <l4/generic/scheduler.h>

#include <l4/generic/thread.h>

#include <l4/api/thread.h>

#include <l4/generic/space.h>

#include <l4/generic/tcb.h>

#include <l4/generic/platform.h>

#include <l4/generic/debug.h>

#include <l4/lib/printk.h>

#include <l4/api/ipc.h>

#include <l4/api/kip.h>

#include <l4/api/errno.h>

#include INC\_ARCH(exception.h)

#include INC\_GLUE(memlayout.h)

#include INC\_GLUE(memory.h)

#include INC\_GLUE(mapping.h)

#include INC\_GLUE(message.h)

#include INC\_GLUE(ipc.h)

#include INC\_SUBARCH(mm.h)

void abort\_die(void)

{

disable\_irqs(); // 在v5/irq.h中定义，对CPSR的I位进行操作

print\_early("Unhandled kernel abort.\n");

print\_early("Kernel panic.\n");

print\_early("Halting system...\n");

while (1) // 禁止死循环，处于宕机状态

;

}

struct ipc\_state {

u32 mr[MR\_TOTAL]; // MR\_TOTAL = 6

unsigned int flags;

};

void ipc\_save\_state(struct ipc\_state \*state)

{

unsigned int \*mr0\_current = KTCB\_REF\_MR0(current); //系统调用时，从任务栈中获取r3寄存器

BUG\_ON(!mr0\_current); // 判断是否为空

/\* Save primary message registers \*/

// 保存主要的信息寄存器

for (int i = 0; i < MR\_TOTAL; i++)

state->mr[i] = mr0\_current[i];

/\* Save ipc flags \*/

state->flags = tcb\_get\_ipc\_flags(current);

}

void ipc\_restore\_state(struct ipc\_state \*state)

{

unsigned int \*mr0\_current = KTCB\_REF\_MR0(current);

BUG\_ON(!mr0\_current);

/\* Restore primary message registers \*/

for (int i = 0; i < MR\_TOTAL; i++)

mr0\_current[i] = state->mr[i];

/\* Restore ipc flags \*/

tcb\_set\_ipc\_flags(current, state->flags);

}

/\* Send data fault ipc to the faulty task's pager \*/

int \_\_attribute\_\_((optimize("O0")))

fault\_ipc\_to\_pager(u32 faulty\_pc, u32 fsr, u32 far, u32 ipc\_tag)

{

int err;

/\* mr[0] has the fault tag. The rest is the fault structure \*/

u32 mr[MR\_TOTAL] = {

[MR\_TAG] = ipc\_tag,

[MR\_SENDER] = current->tid

};

fault\_kdata\_t \*fault = (fault\_kdata\_t \*)&mr[MR\_UNUSED\_START];

/\* Fill in fault information to pass over during ipc \*/

fault->faulty\_pc = faulty\_pc;

fault->fsr = fsr;

fault->far = far;

/\*

\* Write pte of the abort address,

\* which is different on pabt/dabt

\*/

if (is\_prefetch\_abort(fsr))

fault->pte = virt\_to\_pte(faulty\_pc);

else

fault->pte = virt\_to\_pte(far);

/\*

\* System calls save arguments (and message registers)

\* on the kernel stack. They are then referenced from

\* the caller's ktcb. Here, we forge a fault structure

\* as if an ipc syscall has occured. Then the reference

\* to the fault structure is set in the ktcb such that

\* it lies on the mr0 offset when referred as the syscall

\* context.

\*/

/\*

\* Assign fault such that it overlaps

\* as the MR0 reference in ktcb.

\*/

current->syscall\_regs = (syscall\_context\_t \*)

((unsigned long)&mr[0] -

offsetof(syscall\_context\_t, r3));

/\* Set current flags to short ipc \*/

tcb\_set\_ipc\_flags(current, IPC\_FLAGS\_SHORT);

/\* Detect if a pager is self-faulting \*/

if (current == current->pager) {

printk("Pager (%d) faulted on itself. "

"FSR: 0x%x, FAR: 0x%x, PC: 0x%x pte: 0x%x CPU%d Exiting.\n",

current->tid, fault->fsr, fault->far,

fault->faulty\_pc, fault->pte, smp\_get\_cpuid());

thread\_destroy(current);

}

/\* Send ipc to the task's pager \*/

if ((err = ipc\_sendrecv(tcb\_pagerid(current),

tcb\_pagerid(current), 0)) < 0) {

BUG\_ON(current->nlocks);

/\* Return on interrupt \*/

if (err == -EINTR) {

printk("Thread (%d) page-faulted "

"and got interrupted by its pager.\n",

current->tid);

return err;

} else { /\* Suspend on any other error \*/

printk("Thread (%d) faulted in kernel "

"and an error occured during "

"page-fault ipc. err=%d. "

"Suspending task.\n",

current->tid, err);

current->flags |= TASK\_SUSPENDING;

sched\_suspend\_sync();

}

}

return 0;

}

/\*

\* When a task calls the kernel and the supplied user buffer is

\* not mapped, the kernel generates a page fault to the task's

\* pager so that the pager can make the decision on mapping the

\* buffer. Remember that if a task maps its own user buffer to

\* itself this way, the kernel can access it, since it shares

\* that task's page table.

\*/

int pager\_pagein\_request(unsigned long addr, unsigned long size,

unsigned int flags)

{

int err;

u32 abort = 0;

unsigned long npages = \_\_pfn(align\_up(size, PAGE\_SIZE));

struct ipc\_state ipc\_state;

set\_abort\_type(abort, ABORT\_TYPE\_DATA);

/\* Save current ipc state \*/

ipc\_save\_state(&ipc\_state);

/\* For every page to be used by the

\* kernel send a page-in request \*/

for (int i = 0; i < npages; i++)

if ((err = fault\_ipc\_to\_pager(0, abort,

addr + (i \* PAGE\_SIZE),

L4\_IPC\_TAG\_PFAULT)) < 0)

return err;

/\* Restore ipc state \*/

ipc\_restore\_state(&ipc\_state);

return 0;

}

/\*

\* @r0: The address where the program counter was during the fault.

\* @r1: Contains the fault status register

\* @r2: Contains the fault address register

\*/

void data\_abort\_handler(u32 faulted\_pc, u32 dfsr, u32 dfar, u32 spsr)

{

int ret;

system\_account\_dabort();

/\* Indicate abort type on dfsr \*/

set\_abort\_type(dfsr, ABORT\_TYPE\_DATA);

dbg\_abort("Data abort PC:0x%x, FAR: 0x%x, FSR: 0x%x, CPU%d\n",

faulted\_pc, dfar, dfsr, smp\_get\_cpuid());

/\*

\* Check abort type and tell

\* if it's an irrecoverable fault

\*/

if ((ret = check\_abort\_type(faulted\_pc, dfsr, dfar, spsr)) < 0)

goto die; /\* Die if irrecoverable \*/

else if (ret == ABORT\_HANDLED)

return;

/\* Notify the pager \*/

fault\_ipc\_to\_pager(faulted\_pc, dfsr, dfar, L4\_IPC\_TAG\_PFAULT);

/\*

\* FIXME:

\* Check return value of pager, and also make a record of

\* the fault that has occured. We ought to expect progress

\* from the pager. If the same fault is occuring a number

\* of times consecutively, we might want to kill the pager.

\*/

/\* See if current task has various flags set by its pager \*/

if (current->flags & TASK\_SUSPENDING) {

BUG\_ON(current->nlocks);

sched\_suspend\_sync();

}

return;

die:

dprintk("FAR:", dfar);

dprintk("PC:", faulted\_pc);

abort\_die();

}

void prefetch\_abort\_handler(u32 faulted\_pc, u32 ifsr, u32 ifar, u32 spsr)

{

int ret;

system\_account\_pabort();

/\* Indicate abort type on dfsr \*/

set\_abort\_type(ifsr, ABORT\_TYPE\_PREFETCH);

dbg\_abort("Prefetch abort PC:0x%x, FAR: 0x%x, FSR: 0x%x, CPU%d\n",

faulted\_pc, ifar, ifsr, smp\_get\_cpuid());

/\*

\* Check abort type and tell

\* if it's an irrecoverable fault

\*/

if ((ret = check\_abort\_type(0, ifsr, ifar, spsr)) < 0)

goto die; /\* Die if irrecoverable \*/

else if (ret == ABORT\_HANDLED)

return; /\* Return if handled internally \*/

/\* Notify the pager \*/

fault\_ipc\_to\_pager(faulted\_pc, ifsr, ifar, L4\_IPC\_TAG\_PFAULT);

/\*

\* FIXME:

\* Check return value of pager, and also make a record of

\* the fault that has occured. We ought to expect progress

\* from the pager. If the same fault is occuring a number

\* of times consecutively, we might want to kill the pager.

\*/

/\* See if current task has various flags set by its pager \*/

if (current->flags & TASK\_SUSPENDING) {

BUG\_ON(current->nlocks);

sched\_suspend\_sync();

}

return;

die:

dprintk("FAR:", ifar);

abort\_die();

}

void undefined\_instr\_handler(u32 undefined\_address, u32 spsr, u32 lr)

{

dbg\_abort("Undefined instruction. PC:0x%x", undefined\_address);

system\_account\_undef\_abort();

fault\_ipc\_to\_pager(undefined\_address, 0, undefined\_address,

L4\_IPC\_TAG\_UNDEF\_FAULT);

if (!is\_user\_mode(spsr)) {

dprintk("Undefined instruction occured in "

"non-user mode. addr=", undefined\_address);

goto die;

}

/\* See if current task has various flags set by its pager \*/

if (current->flags & TASK\_SUSPENDING) {

BUG\_ON(current->nlocks);

sched\_suspend\_sync();

}

return;

die:

abort\_die();

}

extern int current\_irq\_nest\_count;

/\*

\* This is called right where the nest count is increased

\* in case the nesting is beyond the predefined max limit.

\* It is another matter whether this limit is enough to

\* guarantee the kernel stack is not overflown.

\*

\* FIXME: Take measures to recover. (E.g. disable irqs etc)

\*

\* Note that this is called in irq context, and it \*also\*

\* thrashes the designated irq stack which is only 12 bytes.

\*

\* It really is assumed the system has come to a halt when

\* this happens.

\*/

void irq\_overnest\_error(void)

{

printk("Irqs nested beyond limit. Current count: %d",

current\_irq\_nest\_count);

print\_early("System halted...\n");

while(1)

;

}

### 疑问或改进建议

### 体系结构和平台相关的代码

# V5目录

## SConsript

# Inherit global environment

Import('env')

# The set of source files associated with this SConscript file.

src\_local = ['mapping.c', 'exception.c', 'mmu\_ops.S', 'cache.c', 'mutex.c', 'irq.c', 'init.c', 'atomic.S']

obj = env.Object(src\_local)

Return('obj')

## Atomic.S

### 功能描述

该文件仅包含一个函数，用于原子读取一个字节。

### 数据结构

### 代码注释

/\*

\* Copyright (C) 2010 B Labs

\*

\* Author: Bahadir Balban

\*/

#include INC\_ARCH(asm.h)

/\*

\* Atomically and destructively reads a byte. E.g.

\* byte is read and zero is written back. This is

\* useful on reading irq counts

\*

\* @r0 = byte address

\*/

// 本函数用于原子地读取一个字节，并且将源地址的数据置零。主要用于读取irq数量。

BEGIN\_PROC(l4\_atomic\_dest\_readb)

mov r1, #0 // r1 = 0

swpb r2, r1, [r0] // r2 = [r0], [r0] = r1 = 0。其中后缀b表示只取最后一个字节，其他的清零。

mov r0, r2 // r0 = r2 作为返回值。从这里看出r0即作为参数传入，又作为返回值返回

mov pc, lr // 设置pc为lr，返回上一层调用函数

END\_PROC(l4\_atomic\_dest\_readb)

### 疑问或改进建议

该函数的功能室原子性地读取字节。如何保证原子性，在这个函数中没有体现。是否在外围调用时，做一些限定？

读取irq计数的流程。

### 体系结构和平台相关的代码

## Cache.c

### 功能描述

定义了体系结构相关的cache调用函数。其具体实现在mmu\_ops.s文件中。

### 数据结构

### 代码注释

/\*

\* Generic layer over ARMv5 soecific cache calls

\*

\* Copyright B-Labs Ltd 2010.

\*/

#include INC\_SUBARCH(mmu\_ops.h)

void arch\_invalidate\_dcache(unsigned long start, unsigned long end)

{

arm\_invalidate\_dcache();

}

void arch\_clean\_invalidate\_dcache(unsigned long start, unsigned long end)

{

arm\_clean\_invalidate\_dcache();

}

void arch\_invalidate\_icache(unsigned long start, unsigned long end)

{

arm\_invalidate\_icache();

}

void arch\_clean\_dcache(unsigned long start, unsigned long end)

{

arm\_clean\_dcache();

}

void arch\_invalidate\_tlb(unsigned long start, unsigned long end)

{

arm\_invalidate\_tlb();

}

### 疑问或改进建议

### 体系结构和平台相关的代码

## Exception.c

### 功能描述

该文件处理进程空间的内存异常。这部分参考ARM Architecture Reference Manual的B4.5。

### 数据结构

### 代码注释

/\*

\* Memory exception handling in process context.

\*

\* Copyright (C) 2007, 2008 Bahadir Balban

\*/

#include <l4/generic/scheduler.h>

#include <l4/generic/thread.h>

#include <l4/api/thread.h>

#include <l4/generic/space.h>

#include <l4/generic/tcb.h>

#include <l4/generic/platform.h>

#include <l4/lib/printk.h>

#include <l4/api/ipc.h>

#include <l4/api/kip.h>

#include <l4/api/errno.h>

#include INC\_ARCH(exception.h)

#include INC\_GLUE(memlayout.h)

#include INC\_GLUE(memory.h)

#include INC\_GLUE(mapping.h)

#include INC\_GLUE(message.h)

#include INC\_GLUE(ipc.h)

#include INC\_SUBARCH(mm.h)

int check\_abort\_type(u32 faulted\_pc, u32 fsr, u32 far, u32 spsr)

// faulted\_pc：发生错误的指令；fsr：错误状态寄存器；far：错误地址寄存器；spsr：程序状态保存寄存器

{

int ret = 0;

/\*

\* On ARMv5, prefetch aborts dont have different

\* status values. We validate them here and return.

\*/

// 在v5中，预取指令中止没有不同的状态值。单独进行处理。

// 预取指令失败发生在处理器尝试执行非法指令时。当指令不执行，不会发生该异常。处理过程参考A2.6.5。

// #define is\_prefetch\_abort(fsr) ((fsr >> 8) & 0x1) 第八位判断是否预取指令异常

if (is\_prefetch\_abort(fsr)) {

dbg\_abort("Prefetch abort: 0x%x\n", faulted\_pc);

/\* Happened in any mode other than user \*/

if (!is\_user\_mode(spsr)) {

dprintk("Unhandled kernel prefetch "

"abort at address ", far);

return -EABORT;

}

return 0;

}

switch (fsr & FSR\_FS\_MASK) {

/\* Aborts that are expected on page faults: \*/

// 页错误能进行处理的异常

case DABT\_PERM\_PAGE:

dbg\_abort("Page permission fault 0x%x\n", far);

ret = 0;

break;

case DABT\_XLATE\_PAGE:

dbg\_abort("Page translation fault 0x%x\n", far);

ret = 0;

break;

case DABT\_XLATE\_SECT:

dbg\_abort("Section translation fault 0x%x\n", far);

ret = 0;

break;

/\* Aborts that can't be handled by a pager yet: \*/

// 页面程序不能处理的异常

case DABT\_TERMINAL:

dprintk("Terminal fault dabt ", far);

ret = -EABORT;

break;

case DABT\_VECTOR:

dprintk("Vector abort (obsolete!) ", far);

ret = -EABORT;

break;

case DABT\_ALIGN:

dprintk("Alignment fault dabt ", far);

ret = -EABORT;

break;

case DABT\_EXT\_XLATE\_LEVEL1:

dprintk("External LVL1 translation fault ", far);

ret = -EABORT;

break;

case DABT\_EXT\_XLATE\_LEVEL2:

dprintk("External LVL2 translation fault ", far);

ret = -EABORT;

break;

case DABT\_DOMAIN\_SECT:

dprintk("Section domain fault dabt ", far);

ret = -EABORT;

break;

case DABT\_DOMAIN\_PAGE:

dprintk("Page domain fault dabt ", far);

ret = -EABORT;

break;

case DABT\_PERM\_SECT:

dprintk("Section permission fault dabt ", far);

ret = -EABORT;

break;

case DABT\_EXT\_LFETCH\_SECT:

dprintk("External section linefetch "

"fault dabt ", far);

ret = -EABORT;

break;

case DABT\_EXT\_LFETCH\_PAGE:

dprintk("Page perm fault dabt ", far);

ret = -EABORT;

break;

case DABT\_EXT\_NON\_LFETCH\_SECT:

dprintk("External section non-linefetch "

"fault dabt ", far);

ret = -EABORT;

break;

case DABT\_EXT\_NON\_LFETCH\_PAGE:

dprintk("External page non-linefetch "

"fault dabt ", far);

ret = -EABORT;

break;

default:

dprintk("FATAL: Unrecognised/Unknown "

"data abort ", far);

dprintk("FATAL: FSR code: ", fsr);

ret = -EABORT;

}

/\*

\* Check validity of data abort's source.

\*

\* FIXME: Why not use spsr to do this?

\*/

if (is\_kernel\_address(faulted\_pc)) {

dprintk("Unhandled kernel data "

"abort at address ",

faulted\_pc);

ret = -EABORT;

}

return ret;

}

其中，各种异常类型在exception.h中定义：

\* Definitions for exception support on ARMv5

\*

\* Copyright (C) 2007 Bahadir Balban

\*/

#ifndef \_\_ARCH\_V5\_EXCEPTION\_H\_\_

#define \_\_ARCH\_V5\_EXCEPTION\_H\_\_

#include INC\_ARCH(asm.h)

/\*

\* v5 Architecture-defined data abort values for FSR ordered

\* in highest to lowest priority.

\*/

// 以下定义了15种数据（没有0x3）中止的类型，按照从高到低的优先级排列。通过fsr的低四位可以确定异常类型。

#define DABT\_TERMINAL 0x2

#define DABT\_VECTOR 0x0 /\* Obsolete \*/

#define DABT\_ALIGN 0x1

#define DABT\_EXT\_XLATE\_LEVEL1 0xC

#define DABT\_EXT\_XLATE\_LEVEL2 0xE

#define DABT\_XLATE\_SECT 0x5

#define DABT\_XLATE\_PAGE 0x7

#define DABT\_DOMAIN\_SECT 0x9

#define DABT\_DOMAIN\_PAGE 0xB

#define DABT\_PERM\_SECT 0xD

#define DABT\_PERM\_PAGE 0xF

#define DABT\_EXT\_LFETCH\_SECT 0x4

#define DABT\_EXT\_LFETCH\_PAGE 0x6

#define DABT\_EXT\_NON\_LFETCH\_SECT 0x8

#define DABT\_EXT\_NON\_LFETCH\_PAGE 0xA

#define FSR\_FS\_MASK 0xF

#endif /\* \_\_ARCH\_V5\_EXCEPTION\_H\_\_ \*/

### 疑问或改进建议

### 体系结构和平台相关的代码

## Init.c

### 功能描述

包含了v5特定的初始化函数。

### 数据结构

### 代码注释

/\*

\* ARM v5 specific init routines

\*

\* Copyright (C) 2007 - 2010 B Labs Ltd.

\*/

#include <l4/generic/tcb.h>

#include <l4/generic/scheduler.h>

#include <l4/generic/platform.h>

#include INC\_SUBARCH(mm.h)

#include INC\_SUBARCH(mmu\_ops.h)

#include INC\_GLUE(memory.h)

#include INC\_GLUE(mapping.h)

#include INC\_ARCH(linker.h)

SECTION(".data.pgd") ALIGN(PGD\_SIZE) pgd\_table\_t init\_pgd;

// 将页目录表放置在.data.pgd节中，并按照4页进行对齐。

/\* Type-checkable page table elements \*/

typedef u32 pmd\_t;

typedef u32 pte\_t;

/\* Page global directory made up of pgd\_t entries \*/

typedef struct pgd\_table {

pmd\_t entry[PGD\_ENTRY\_TOTAL]; // 一共4K项，每项为一个整型数

} pgd\_table\_t;

/\* Page middle directory made up of pmd\_t entries \*/

typedef struct pmd\_table {

pte\_t entry[PMD\_ENTRY\_TOTAL];

} pmd\_table\_t;

struct address\_space init\_space;

void system\_identify(void)

{

}

void jump(struct ktcb \*task)

{

\_\_asm\_\_ \_\_volatile\_\_ (

"mov lr, %0\n" /\* Load pointer to context area \*/

"ldr r0, [lr]\n" /\* Load spsr value to r0 \*/

"msr spsr, r0\n" /\* Set SPSR as ARM\_MODE\_USR \*/

"add sp, lr, %1\n" /\* Reset SVC stack \*/

"sub sp, sp, %2\n" /\* Align to stack alignment \*/

"ldmib lr, {r0-r14}^\n" /\* Load all USR registers \*/

"nop \n" /\* Spec says dont touch banked registers

\* right after LDM {no-pc}^ for one instruction \*/

"add lr, lr, #64\n" /\* Manually move to PC location. \*/

"ldr lr, [lr]\n" /\* Load the PC\_USR to LR \*/

"movs pc, lr\n" /\* Jump to userspace, also switching SPSR/CPSR \*/

:

: "r" (task), "r" (PAGE\_SIZE), "r" (STACK\_ALIGNMENT)

);

}

void switch\_to\_user(struct ktcb \*task)

{

arm\_clean\_invalidate\_cache();

arm\_invalidate\_tlb();

arm\_set\_ttb(virt\_to\_phys(TASK\_PGD(task)));

arm\_invalidate\_tlb();

jump(task);

}

/\* Maps the early memory regions needed to bootstrap the system \*/

void init\_kernel\_mappings(void)

{

//memset((void \*)virt\_to\_phys(&init\_pgd), 0, sizeof(pgd\_table\_t));

/\* Map kernel area to its virtual region \*/

add\_section\_mapping\_init(align(virt\_to\_phys(\_start\_text), SZ\_1MB),

align((unsigned int)\_start\_text, SZ\_1MB), 1,

cacheable | bufferable);

/\* Map kernel one-to-one to its physical region \*/

add\_section\_mapping\_init(align(virt\_to\_phys(\_start\_text), SZ\_1MB),

align(virt\_to\_phys(\_start\_text), SZ\_1MB),

1, 0);

}

/\*

\* Enable virtual memory using kernel's pgd

\* and continue execution on virtual addresses.

\*/

void start\_virtual\_memory()

{

/\*

\* TTB must be 16K aligned. This is because first level tables are

\* sized 16K.

\*/

if ((unsigned int)&init\_pgd & 0x3FFF)

dprintk("kspace not properly aligned for ttb:",

(u32)&init\_pgd);

// memset((void \*)&kspace, 0, sizeof(pgd\_table\_t));

arm\_set\_ttb(virt\_to\_phys(&init\_pgd));

/\*

\* This sets all 16 domains to zero and domain 0 to 1. The outcome

\* is that page table access permissions are in effect for domain 0.

\* All other domains have no access whatsoever.

\*/

arm\_set\_domain(1);

/\* Enable everything before mmu permissions are in place \*/

arm\_enable\_caches();

arm\_enable\_wbuffer();

arm\_enable\_high\_vectors();

/\*

\* Leave the past behind. Tlbs are invalidated, write buffer is drained.

\* The whole of I + D caches are invalidated unconditionally. This is

\* important to ensure that the cache is free of previously loaded

\* values. Otherwise unpredictable data aborts may occur at arbitrary

\* times, each time a load/store operation hits one of the invalid

\* entries and those entries are cleaned to main memory.

\*/

arm\_invalidate\_cache();

arm\_drain\_writebuffer();

arm\_invalidate\_tlb();

arm\_enable\_mmu();

/\* Jump to virtual memory addresses \*/

\_\_asm\_\_ \_\_volatile\_\_ (

"add sp, sp, %0 \n" /\* Update stack pointer \*/

"add fp, fp, %0 \n" /\* Update frame pointer \*/

/\* On the next instruction below, r0 gets

\* current PC + KOFFSET + 2 instructions after itself. \*/

"add r0, pc, %0 \n"

/\* Special symbol that is extracted and included in the loader.

\* Debuggers can break on it to load the virtual symbol table \*/

".global break\_virtual;\n"

"break\_virtual:\n"

"mov pc, r0 \n" /\* (r0 has next instruction) \*/

:

: "r" (KERNEL\_OFFSET)

: "r0"

);

/\*

\* Restore link register (LR) for this function.

\*

\* NOTE: LR values are pushed onto the stack at each function call,

\* which means the restored return values will be physical for all

\* functions in the call stack except this function. So the caller

\* of this function must never return but initiate scheduling etc.

\*/

\_\_asm\_\_ \_\_volatile\_\_ (

"add %0, %0, %1 \n"

"mov pc, %0 \n"

:: "r" (\_\_builtin\_return\_address(0)), "r" (KERNEL\_OFFSET)

);

/\* should never come here \*/

while(1);

}

### 疑问或改进建议

### 体系结构和平台相关的代码

## Irq.c

### 功能描述

在针对ARM体系结构的编程中，一般很难直接使用C语言产生操作协处理器的相关代码，因此使用汇编语言来实现就成为了唯一的选择。但如果完全通过汇编代码实现，又会过于复杂、难以调试。因此，C语言内嵌汇编的方式倒是一个不错的选择。然而，使用内联汇编的一个主要问题是，内联汇编的语法格式与使用的编译器直接相关，也就是说，使用不同的C编译器内联汇编代码时，它们的写法是各不相同的。下面介绍在ARM体系结构下GCC的内联汇编。GCC内联汇编的一般格式：

asm(

代码列表

: 输出运算符列表

: 输入运算符列表

: 被更改资源列表

);

那么这个新的表达式又该怎样解释呢？原来，在“r”(tmp)这个表达式中，tmp代表的正是C语言向内联汇编输入的变量，操作符“r”则代表tmp的值会通过某一个寄存器来传递。在GCC4中与之相类似的操作符还包括“m”、“I”，等等，其含义见下表：

|  |  |  |
| --- | --- | --- |
| **Constraint** | **Usage in ARM state** | **Usage in Thumb state** |
| f | Floating point registers f0 .. f7 | Not available |
| h | Not available | Registers r8..r15 |
| G | Immediate floating point constant | Not available |
| H | Same a G, but negated | Not available |
| I | Immediate value in data processing instructions  e.g. ORR R0, R0, #operand | Constant in the range 0 .. 255  e.g. SWI operand |
| J | Indexing constants -4095 .. 4095  e.g. LDR R1, [PC, #operand] | Constant in the range -255 .. -1  e.g. SUB R0, R0, #operand |
| K | Same as I, but inverted | Same as I, but shifted |
| L | Same as I, but negated | Constant in the range -7 .. 7  e.g. SUB R0, R1, #operand |
| l | Same as r | Registers r0..r7  e.g. PUSH operand |
| M | Constant in the range of 0 .. 32 or a power of 2  e.g. MOV R2, R1, ROR #operand | Constant that is a multiple of 4 in the range of 0 .. 1020  e.g. ADD R0, SP, #operand |
| m | Any valid memory address | |
| N | Not available | Constant in the range of 0 .. 31  e.g. LSL R0, R1, #operand |
| O | Not available | Constant that is a multiple of 4 in the range of -508 .. 508  e.g. ADD SP, #operand |
| r | General register r0 .. r15  e.g. SUB operand1, operand2, operand3 | Not available |
| W | Vector floating point registers s0 .. s31 | Not available |
| X | Any operand | |

与输入运算符列表的应用方法一致，当C语言需要利用内联汇编输出结果时，可以使用输出运算符列表来实现，其格式应该是下面这样的。

void test(void)

{

int tmp;

asm(

"mov %0,#1\n"

:"=r"(tmp)

:

);

}

在上面的代码中，原本应出现在输入运算符列表中的运算符，现在出现在了输出运算符列表中，同时变量tmp将会存储内联汇编的输出结果。这里有一点可能已经引起大家的注意了，上面的代码中操作符r的前面多了一个“=”。这个等号被称为约束修饰符，其作用是对内联汇编的操作符进行修饰。几种修饰符的含义如下表所示：

|  |  |
| --- | --- |
| **Modifier** | **Specifies** |
| = | Write-only operand, usually used for all output operands |
| + | Read-write operand, must be listed as an output operand |
| & | A register that should be used for output only |

更详细请参考<http://www.ethernut.de/en/documents/arm-inline-asm.html>。

本文件只涉及关IRQ和打开IRQ，以及判断IRQ是否打开的操作。其操作非常简单，只需要对CPSR的I位进行读写即可。

### 数据结构

### 代码注释

/\*

\* Low-level irq routines.

\*

\* Copyright (C) 2010 B Labs Ltd.

\* Written by Bahadir Balban

\* Prem Mallappa <prem.mallappa@b-labs.co.uk>

\*/

void irq\_local\_disable\_save(unsigned long \*state)

// CPSR第7位对应I位，关闭IRQ；第6位对应F位，关闭FIQ。

// 本函数关闭IRQ，并且将CPSR保存在state中。以便于之后恢复。

// 在irq.h中有irq\_local\_disable\_save和irq\_local\_restore的定义。该函数无条件禁止IRQ，// 同时保存之前的CPSR状态。如果该函数调用前已经禁止了IRQ，就可以通过state恢复。

{

unsigned int tmp, tmp2;

\_\_asm\_\_ \_\_volatile\_\_ (

"mrs %0, cpsr\_fc \n" // 将CPSR赋值到tmp

"orr %1, %0, #0x80 \n" // tmp2 = tmp | 0x80

"msr cpsr\_fc, %1 \n" // 将tmp2赋值到CPSR

: "=&r"(tmp), "=r"(tmp2) // 定义了输入，其中tmp/tmp2都分配寄存器存放，只写，tmp只作为输出

:

: "cc" // 更新的寄存器有条件码寄存器

);

\*state = tmp;

}

void irq\_local\_restore(unsigned long state)

{

\_\_asm\_\_ \_\_volatile\_\_ (

"msr cpsr\_fc, %0\n" // 将state赋值到CPSR，恢复CPSR的值

:

: "r"(state)

: "cc"

);

}

int irqs\_enabled(void)

// 判断IRQ是否使能

{

int tmp;

\_\_asm\_\_ \_\_volatile\_\_ (

"mrs %0, cpsr\_fc\n"

: "=r"(tmp)

); // 读取CPSR

if (tmp & 0x80) // 返回CPSR第I位的值。I值为1，表示禁止。

return 0;

return 1;

}

### 疑问或改进建议

### 体系结构和平台相关的代码

## Mmu\_ops.s

### 功能描述

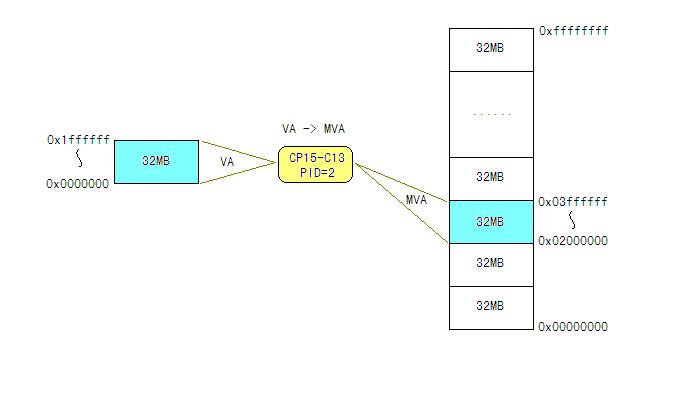
#### 地址分类

ARM v5使用协处理器CP15来实现MMU机制。ARM9有三种地址，VA（虚地址），MVA（修正后虚地址），PA（物理地址）

* 1. VA，是程序中的逻辑地址，0x00000000~0xFFFFFFFF。
  2. MVA，由于多个进程执行，逻辑地址会重合。所以，跟据进程号将逻辑地址分布到整个内存中。MVA = (PID << 25) | VA
  3. PA，MVA通过MMU转换后的地址。

由2可知，地址位共32位，PID占7位，所以最多只能有 128 个进程。而每个进程可访问的地址位为25位，故只能分到32MB的地址空间。（注：不是物理内存空间）

PID是存放在CP15协处理器的C13寄存器的高7位。



#### 虚拟内存转换

CP15从C2中获得页基址（TTB）。将 MVA 的高12位作为页表索引值。获得页表项：TTB [ MVA >> 20 ]。注意：页表项是32位的。

从上可知，一个页表最多有4096个页表项，也就是4K。那么，每个页表项可以表示1MB的地址空间。

得来的项表项分三种：

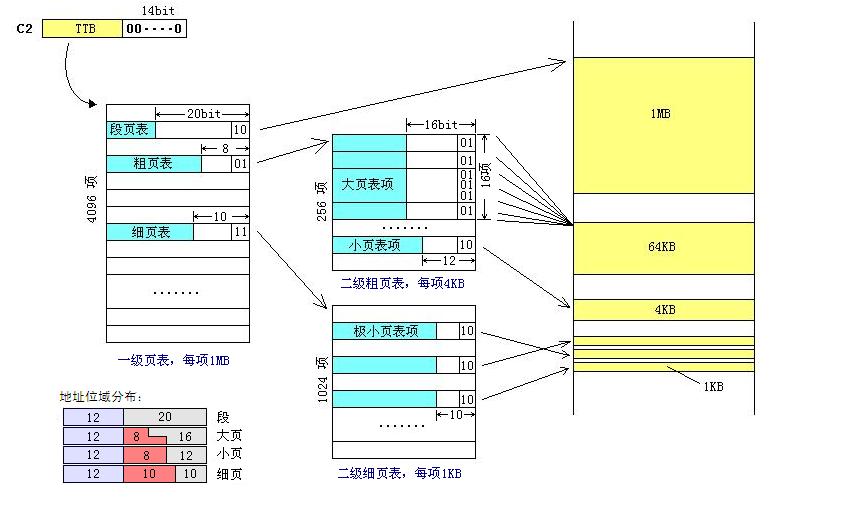
1. 段页描述符，直接指向1MB的内存空间。
2. 粗页描述符，有256个二级页表项，每个二级页表项指向4KB的内存空间。
3. 细页描述符，有1024个二级页表项，每个二级页表项指向1KB的内存空间。

粗页描述符中存放的是粗页表二级表的基址。 将MVA的[19~12]位用来进行二级页表查寻。粗页表二级表分两种：

1. 大页描述符，一个描述符可以对应64KB的内存地址，但16个二级描述符对应同一块内存。
2. 小页描述符，一个描述符只对应4KB的内存地址，每个二级描述符只对应一块内存。

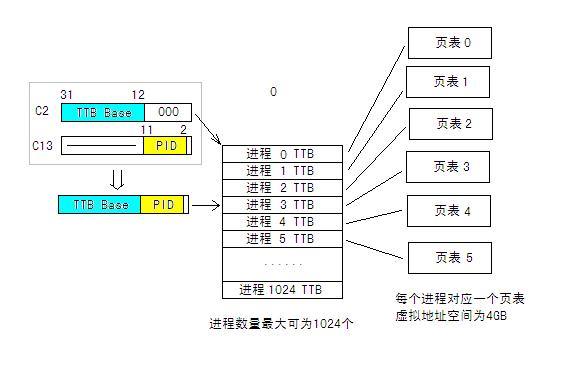
细页描述符中存放的是细页表二级表的基址。将MVA的[19~10]，共计10位用于进行二级页表索引。二级页表共1024个描述符。剩下的10位作为基址，可访问空间为1024B。

如下是内存转换图：



通过上面的学习，了解到ARM将4GB的地址访问空间分成128个32MB，每份供一个进程使用。如此以来，一个进程的地址访问空间只有32MB。如进程1的地址空间为［0x02000000~0x03FFFFFF］。如果超出这个范围，地址访问就是非法的。

那ARM9在设计CP15时为什么不为每一个进程指定一个单独的页表。这样以来，每个进程就可以独地拥有4GB的地址空间。



如此以来，进程数就不再受限于128个，可以多达1024个进程。而每一个进程的虚拟地址的空间可以扩展到4GB。

#### 相关指令

ARM 微处理器可支持多达 16 个协处理器，用于各种协处理操作，在程序执行的过程中，每个协处理器只执行针对自身的协处理指令，忽略 ARM 处理器和其他协处理器的指令。ARM 的协处理器指令主要用于 ARM 处理器初始化 ARM 协处理器的数据处理操作，以及在ARM 处理器的寄存器和协处理器的寄存器之间传送数据，和在 ARM 协处理器的寄存器和[存储器](http://www.aetelecom.com.cn/biaoji-137-1.html)之间传送数据。 ARM 协处理器指令包括以下 5 条：

— CDP 协处理器数操作指令

— LDC 协处理器数据加载指令

— STC 协处理器数据存储指令

— MCR ARM 处理器寄存器到协处理器寄存器的数据传送指令

— MRC 协处理器寄存器到ARM 处理器寄存器的数据传送指令

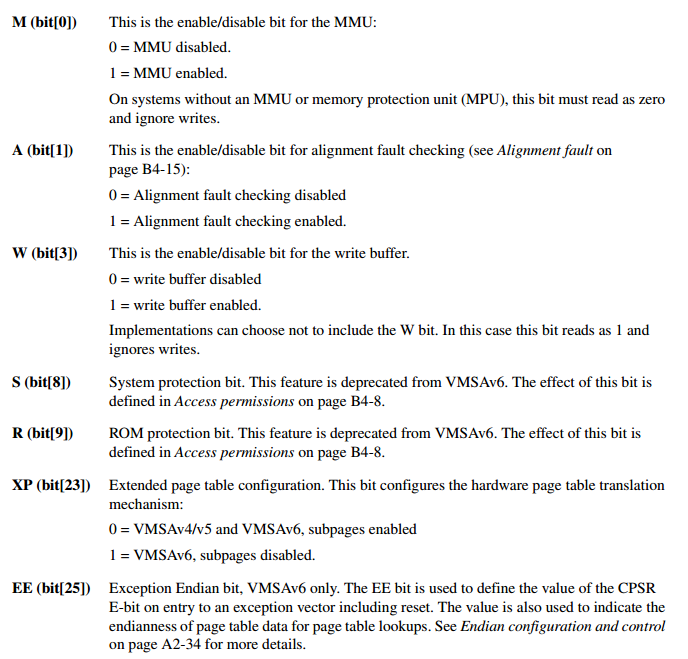
该文件定义了MMU的底层实现。MMU主要通过系统控制协处理器CP15进行控制。对CP15的指令格式如下：

MRC p15, 0, Rd, CRn, CRm, opcode\_2

MCR p15, 0, Rd, CRn, CRm, opcode\_2

其中CRn是CP15的寄存器。一般而言，CRm和opcode\_2都为0。

其中CP15的r1控制寄存器的各位定义如下，这里没有包含代码中所有的位定义。



### 数据结构

### 代码注释

/\*

\* low-level mmu operations

\*

\* Copyright (C) 2007 Bahadir Balban

\*/

#include INC\_ARCH(asm.h)

#define C15\_id c0 // 缓存类型寄存器（CACHE TYPE REGISTER），只读寄存器，包含了cache的信息。读这个寄存器的方式是通过设置协处理操作码为1

#define C15\_control c1 // 控制寄存器

#define C15\_ttb c2 // 转换表基址寄存器（Translation Table Base --TTB）

#define C15\_dom c3 // 域访问控制寄存器（Domain access control ）

#define C15\_fsr c5 // 异常状态寄存器（fault status -FSR）

#define C15\_far c6 // 异常地址寄存器（fault address -FAR）

#define C15\_tlb c8 // TLB操作寄存器

#define C15\_C0\_M 0x0001 /\* MMU \*/

#define C15\_C0\_A 0x0002 /\* Alignment \*/

#define C15\_C0\_C 0x0004 /\* (D) Cache \*/

#define C15\_C0\_W 0x0008 /\* Write buffer \*/

#define C15\_C0\_B 0x0080 /\* Endianness \*/

#define C15\_C0\_S 0x0100 /\* System \*/

#define C15\_C0\_R 0x0200 /\* ROM \*/

#define C15\_C0\_Z 0x0800 /\* Branch Prediction \*/

#define C15\_C0\_I 0x1000 /\* I cache \*/

#define C15\_C0\_V 0x2000 /\* High vectors \*/

/\* FIXME: Make sure the ops that need r0 dont trash r0, or if they do,

\* save it on stack before these operations.

\*/

// 需要确保对r0的操作不会丢弃r0，否则，进行操作之前应该先把r0放入堆栈中保存。

/\*

\* In ARM terminology, flushing the cache means invalidating its contents.

\* Cleaning the cache means, writing the contents of the cache back to

\* main memory. In write-back caches the cache must be cleaned before

\* flushing otherwise in-cache data is lost.

\*/

// 在ARM的术语中，flush意味着将cache中的内容无效。Clean意味着将cache的内容写回主存。对于写回策略的cache，flush前必须先执行clean操作，否则其中的数据会丢失。

BEGIN\_PROC(arm\_set\_ttb)

mcr p15, 0, r0, C15\_ttb, c0, 0

mov pc, lr

END\_PROC(arm\_set\_ttb)

BEGIN\_PROC(arm\_get\_domain)

mrc p15, 0, r0, C15\_dom, c0, 0

mov pc, lr

END\_PROC(arm\_get\_domain)

BEGIN\_PROC(arm\_set\_domain)

mcr p15, 0, r0, C15\_dom, c0, 0

mov pc, lr

END\_PROC(arm\_set\_domain)

BEGIN\_PROC(arm\_enable\_mmu)

mrc p15, 0, r0, C15\_control, c0, 0

orr r0, r0, #C15\_C0\_M

mcr p15, 0, r0, C15\_control, c0, 0

mov pc, lr

END\_PROC(arm\_enable\_mmu)

BEGIN\_PROC(arm\_enable\_icache)

mrc p15, 0, r0, C15\_control, c0, 0

orr r0, r0, #C15\_C0\_I

mcr p15, 0, r0, C15\_control, c0, 0

mov pc, lr

END\_PROC(arm\_enable\_icache)

BEGIN\_PROC(arm\_enable\_dcache)

mrc p15, 0, r0, C15\_control, c0, 0

orr r0, r0, #C15\_C0\_C

mcr p15, 0, r0, C15\_control, c0, 0

mov pc, lr

END\_PROC(arm\_enable\_dcache)

BEGIN\_PROC(arm\_enable\_wbuffer)

mrc p15, 0, r0, C15\_control, c0, 0

orr r0, r0, #C15\_C0\_W

mcr p15, 0, r0, C15\_control, c0, 0

mov pc, lr

END\_PROC(arm\_enable\_wbuffer)

BEGIN\_PROC(arm\_enable\_high\_vectors)

mrc p15, 0, r0, C15\_control, c0, 0

orr r0, r0, #C15\_C0\_V

mcr p15, 0, r0, C15\_control, c0, 0

mov pc, lr

END\_PROC(arm\_enable\_high\_vectors)

BEGIN\_PROC(arm\_invalidate\_cache)

mov r0, #0 @ FIX THIS

mcr p15, 0, r0, c7, c7, 0 @ Flush I cache and D cache

mov pc, lr

END\_PROC(arm\_invalidate\_cache)

BEGIN\_PROC(arm\_invalidate\_icache)

mov r0, #0 @ FIX THIS

mcr p15, 0, r0, c7, c5, 0 @ Flush I cache

mov pc, lr

END\_PROC(arm\_invalidate\_icache)

BEGIN\_PROC(arm\_invalidate\_dcache)

mov r0, #0 @ FIX THIS

mcr p15, 0, r0, c7, c6, 0 @ Flush D cache

mov pc, lr

END\_PROC(arm\_invalidate\_dcache)

BEGIN\_PROC(arm\_clean\_dcache)

mrc p15, 0 , r15, c7, c10, 3 @ Test/clean dcache line

bne arm\_clean\_dcache

mcr p15, 0, ip, c7, c10, 4 @ Drain WB

mov pc, lr

END\_PROC(arm\_clean\_dcache)

BEGIN\_PROC(arm\_clean\_invalidate\_dcache)

1:

mrc p15, 0, r15, c7, c14, 3 @ Test/clean/flush dcache line

@ COMMENT: Why use PC?

bne 1b

mcr p15, 0, ip, c7, c10, 4 @ Drain WB

mov pc, lr

END\_PROC(arm\_clean\_invalidate\_dcache)

BEGIN\_PROC(arm\_clean\_invalidate\_cache)

1:

mrc p15, 0, r15, c7, c14, 3 @ Test/clean/flush dcache line

@ COMMENT: Why use PC?

bne 1b

mcr p15, 0, ip, c7, c5, 0 @ Flush icache

mcr p15, 0, ip, c7, c10, 4 @ Drain WB

mov pc, lr

END\_PROC(arm\_clean\_invalidate\_cache)

BEGIN\_PROC(arm\_drain\_writebuffer)

mov r0, #0 @ FIX THIS

mcr p15, 0, r0, c7, c10, 4

mov pc, lr

END\_PROC(arm\_drain\_writebuffer)

BEGIN\_PROC(arm\_invalidate\_tlb)

mcr p15, 0, ip, c8, c7

mov pc, lr

END\_PROC(arm\_invalidate\_tlb)

BEGIN\_PROC(arm\_invalidate\_itlb)

mov r0, #0 @ FIX THIS

mcr p15, 0, r0, c8, c5, 0

mov pc, lr

END\_PROC(arm\_invalidate\_itlb)

BEGIN\_PROC(arm\_invalidate\_dtlb)

mov r0, #0 @ FIX THIS

mcr p15, 0, r0, c8, c6, 0

mov pc, lr

END\_PROC(arm\_invalidate\_dtlb)

### 疑问或改进建议

从以上代码可见，对MMU的控制主要涉及CP15协处理器的寄存器的读写操作。其基本流程是，先把协处理器寄存器读取到常规寄存器，通过对常规寄存器进行位操作，然后写回到协处理器寄存器中。

然而，在官方文档中对CP15的所有寄存器的各位定义并不完整。另外，对mcr/mrc的操作的第二个目标寄存器的作用也不明确。

### 体系结构和平台相关的代码

## Mutex.c

### 功能描述（对于复杂操作建议画出流程图）

### 数据结构

### 代码注释

/\*

\* ARM v5 Binary semaphore (mutex) implementation.

\*

\* Copyright (C) 2007-2010 B Labs Ltd.

\* Author: Prem Mallappa <prem.mallappa@b-labs.co.uk>

\*/

#include <l4/lib/printk.h>

/\* Recap on swp:

\* swp rx, ry, [rz]

\* In one instruction:

\* 1) Stores the value in ry into location pointed by rz.

\* 2) Loads the value in the location of rz into rx.

\* By doing so, in one instruction one can attempt to lock

\* a word, and discover whether it was already locked.

\*/

// swp指令的作用：swp rx, ry, [rz]

// 1）保存ry的值到rz所指向的内存

// 2）将rz所指向内存的值加载到rx

// 这样可以用一条指令来锁定一个字，同时测试该字是否已被锁定。

#define MUTEX\_UNLOCKED 0

#define MUTEX\_LOCKED 1

void \_\_spin\_lock(unsigned long \*s)

{

int tmp = 0, tmp2;

\_\_asm\_\_ \_\_volatile\_\_(

"1: \n"

"swp %0, %1, [%2] \n" // tmp2 = \*s, \*s = 0

"teq %0, %3 \n" // 判断tmp2是否为0

"bne 1b \n" // 直到tmp2为0，跳出循环

: "=&r" (tmp2)

: "r" (tmp), "r"(s), "r"(0)

: "cc", "memory"

);

}

void \_\_spin\_unlock(unsigned long \*s)

{

int tmp = 1, tmp2;

\_\_asm\_\_ \_\_volatile\_\_(

"1: \n"

"swp %0, %1, [%2] \n" // tmp2 = \*s, \*s = 1

"teq %0, %3 \n" // 判断tmp2是否为0

"bne 1b \n" // 直到tmp2为0，跳出循环

: "=&r" (tmp2)

: "r" (tmp), "r"(s), "r"(0)

: "cc", "memory"

);

}

int \_\_mutex\_lock(unsigned long \*s)

{

int tmp = MUTEX\_LOCKED, tmp2;

\_\_asm\_\_ \_\_volatile\_\_(

"swp %0, %1, [%2] \n"

: "=&r"(tmp2)

: "r"(tmp), "r"(s)

: "cc", "memory"

);

if (tmp2 == MUTEX\_UNLOCKED)

return 1;

return 0;

}

void \_\_mutex\_unlock(unsigned long \*s)

{

int tmp, tmp2=MUTEX\_UNLOCKED;

\_\_asm\_\_ \_\_volatile\_\_(

"swp %0, %1, [%2] \n"

: "=&r"(tmp)

: "r"(tmp2), "r"(s)

: "cc", "memory"

);

BUG\_ON(tmp != MUTEX\_LOCKED);

}

### 疑问或改进建议

### 体系结构和平台相关的代码

# 总结

参考文献

<http://blog.csdn.net/hevake_lcj/article/details/7400751>

<http://www.ethernut.de/en/documents/arm-inline-asm.html>