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

|  |
| --- |
|  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | 当前版本： | 1.0 | | 日期： | 2014-07-10 | | 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 | 康乔 | 无 | 2014-06-30到2014年8月1日 |  |
|  |  |  |  |  |
|  |  |  |  |  |

# 概述

本目录为codezero向上层提供的应用程序编程接口。

# 总体功能描述

主要包含以下内容：

高速缓存操作: cache.c

有关进程权能（capability）的系统调用 cap.c

交换寄存器的操作 exreg.c

进程间通信的方法 ipc.c

用户空间中断请求的管理 irq.c

kernel interface page kip.c

与空间有关的系统调用 map.c

互斥量 mutex.c

若干系统调用 syscall.c

与线程有关的系统调用 thread.c

# Sconstruct文件

Sconstruct 代码如下：

# Inherit global environment

Import('env')

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

src\_local = ['kip.c', 'syscall.c', 'thread.c', 'ipc.c', 'map.c',

'mutex.c', 'cap.c', 'exregs.c', 'irq.c', 'cache.c']

obj = env.Object(src\_local)

Return('obj')

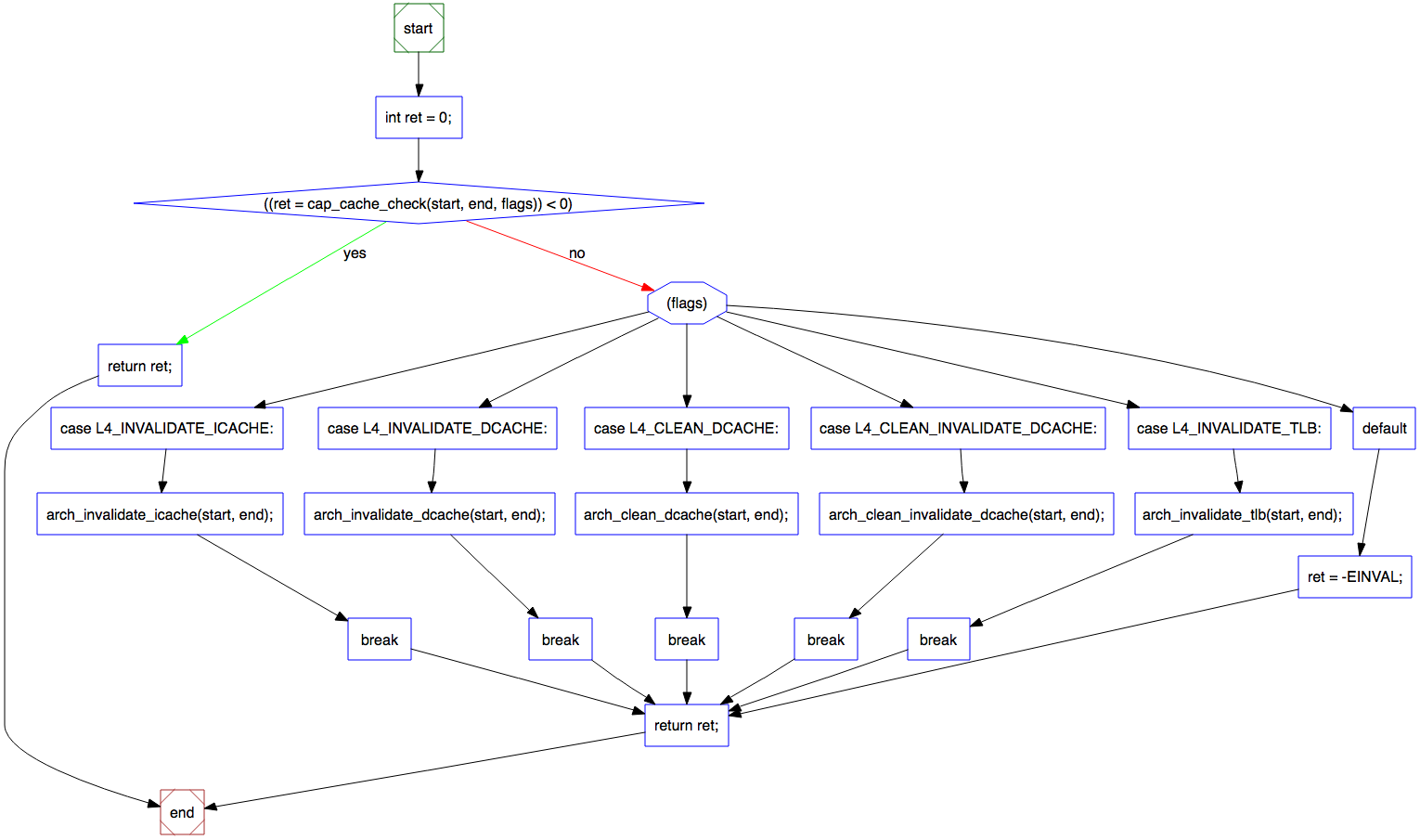
其作用是把该目录下的所有.c文件都编译成.o文件，然后把目标文件集合返回。

# 文件1：cache.c

## 功能描述（对于复杂操作建议画出流程图）

包含一个对高速缓存进行操作的函数：

sys\_cache\_control**(unsigned** **long** start**,** **unsigned** **long** end**,** **unsigned** **int** flags**)**



执行流程简述：

1. 首先调用cap\_cache\_check判断相应的区域是否有权限使用系统调用，如果没有，则返回-1，结束该函数。
2. 权限检查通过后，根据参数flag的值的不同，对start-end这一区域进行不同的操作。

2.1如果flags=L4\_INVALIDATE\_DCACHE，则使该内存区域对应的dcache失效

2.2如果flags=L4\_INVALIDATE\_ICACHE，则使该内存区域对应的icache失效

2.3 如果flags=L4\_CLEAN\_DCACHE，则清除对应的dcache

2.4 如果flags=L4\_CLEAN\_INVALIDATE\_DCACHE, 清除对应的失效的DCACHE

2.5 如果flags=L4\_INVALIDATE\_TLB, 则使对应的TLB失效

3、返回0，结束函数。

## 数据结构

无

## 代码注释

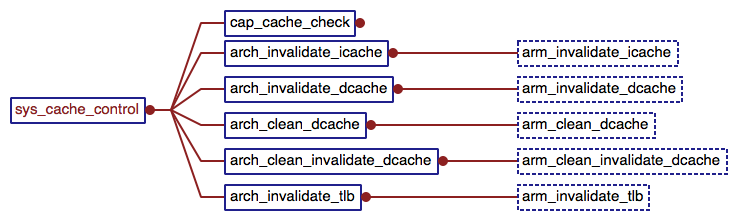
略

## 疑问或改进建议

暂无

## 体系结构和平台相关的代码

对于不同的flag的值，调用了不同的arch\_xxxxx函数，这一函数又会调用arm\_xxxx函数，如下图所示



这些arm\_xxx函数使用汇编语言实现，位于/arch/arm/v5或6/mmu\_ops.S中

# 文件2：cap.c

## 功能描述

包含两个跟权能（capability）有关的操作。

有关权能的解释（来自capability.h）：权能是特定的资源拥有的安全限定的一种表示。其数据结构capability将在下文中分析。

1、读取当前进程的所有权能：

**函数原型：int** cap\_read\_all**(struct** capability **\***caparray**)**

该函数的执行比较简单，执行结果为把当前进程的所有权能拷贝到指针caparray指定的内存位置中。

1. 读取或者操纵权能：

函数原型：**int** sys\_capability\_control**(unsigned** **int** req**,** **unsigned** **int** flags**,** **void** **\***userbuf**)**

函数执行流程图如下：

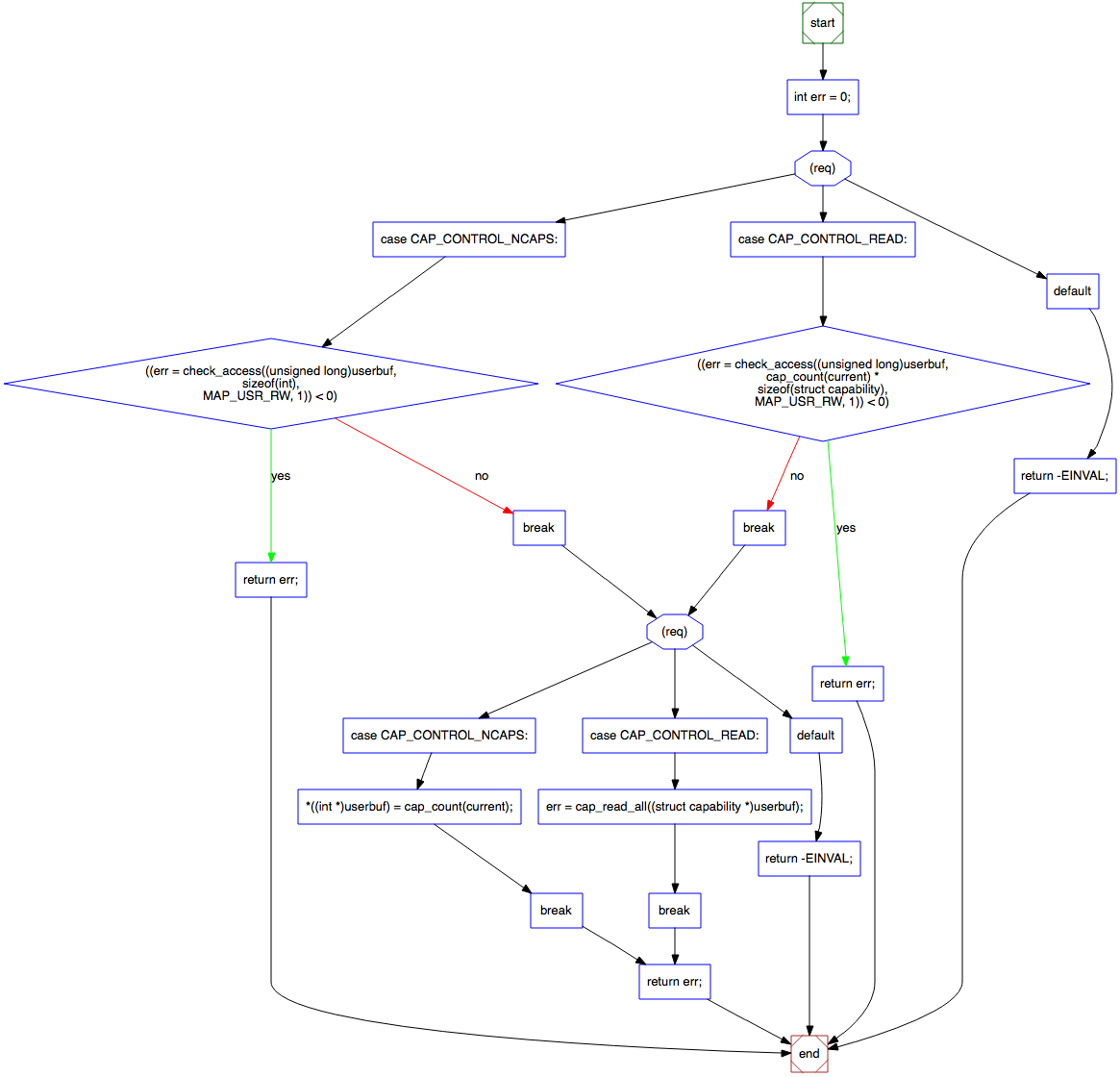


图 1

该函数的执行过程简述：

1. 根据req的值的不同，对提供的用户缓冲区进行检验，是否符合要求。如果不符合，则返回-EINVAL，并结束函数。
2. 根据req的值的不同，执行不同的操作
   1. 如果req为CAP\_CONTROL\_NCAPS，则调用cap\_count函数对权能计数，写入用户缓冲区中。
   2. 如果req为CAP\_CONTROL\_READ,则调用本文件中的第一个函数cap\_read\_all把当前进程的权能写入用户缓冲区中。

## 数据结构

1. 权能数据结构capability （在capability.h中定义），代码如下：

**struct** capability **{**  
    **struct** link list**;**  
  
    */\* 权能的标示符 \*/*  
    l4id\_t capid**;**       */\* 独有的权能ID \*/*  
    l4id\_t owner**;**       */\* 权能拥有着的ID，这个ID通常是一个线程ID \*/*  
    l4id\_t resid**;**       */\* 这一权能的目的资源的ID\*/*  
    **unsigned** **int** type**;**  */\* 种类包含了两个域，既有权能的类型，也有目的资源的类型 \*/*  
  
    */\* Capability limits/permissions \*/*  
    u32 access**;**     */\* Permitted operations \*/*  
  
    */\* Limits on the resource (NOTE: must never have signed type) \*/*  
    **unsigned** **long** start**;**    */\* Resource start value \*/*  
    **unsigned** **long** end**;**  */\* Resource end value \*/*  
    **unsigned** **long** size**;** */\* Resource size \*/*  
  
    */\* Use count of resource \*/*  
    **unsigned** **long** used**;**  
  
    */\* Device attributes, if this is a device. \*/*  
    **unsigned** **int** attr**;**  
    l4id\_t irq**;**  
**};**

补充：

1. 其中，type域中的目的资源类型域表示了该权能够在何种资源上执行操作，例如一个线程、或者线程组等都可能是该域的值。
2. type域中的权能类型表明了在资源上执行何种操作。例如如果资源类型为线程或者线程组，该域的值则可能为thread\_control, ipc, 交换寄存器， map操作等。

## 代码注释

暂无

## 疑问和改进建议

暂无

## 体系结构和平台相关的代码

无

# 文件3：exreg.c

## 功能描述

提供了三个函数，用于设置线程的寄存器上下文。

1. 把所有寄存器（如果有效位为1）的值拷贝到任务的上下文中。

函数原型：**void** exregs\_write\_registers**(struct** ktcb **\***task**,** **struct** exregs\_data **\***exregs**)**

函数执行的效果：是把数据结构exregs中的所有寄存器的值拷贝到task的上下文中。

1. 读取任务的所有寄存器上下文。

函数原型：**void** exregs\_read\_registers**(struct** ktcb **\***task**,** **struct** exregs\_data **\***exregs**)**

函数执行的效果：是把任务task的寄存器上下文读取到数据结构exregs中

1. 为某一个线程设定寄存器上下文

函数原型：**int** sys\_exchange\_registers**(struct** exregs\_data **\***exregs**,** l4id\_t tid**)**

函数执行的效果：是把exregs中保存的所有寄存器的值赋给tid指定的线程。

执行流程图如下所示：

函数执行流程简述：

1. 检查是否对exregs有相应权限，若没有，则返回错误，并结束。
2. 针对线程号tid，找到tcb，并把tcb赋予task结构。若错误，则返回-ESRCH，并结束。
3. 给task加锁，以防它被系统调用无意地重设；如果失败，则返回-EAGAIN，并结束。
4. 进行一系列检查工作，如果不通过，则返回错误并结束。
5. 进行寄存器的拷贝。
6. 对task解锁，返回0并结束。

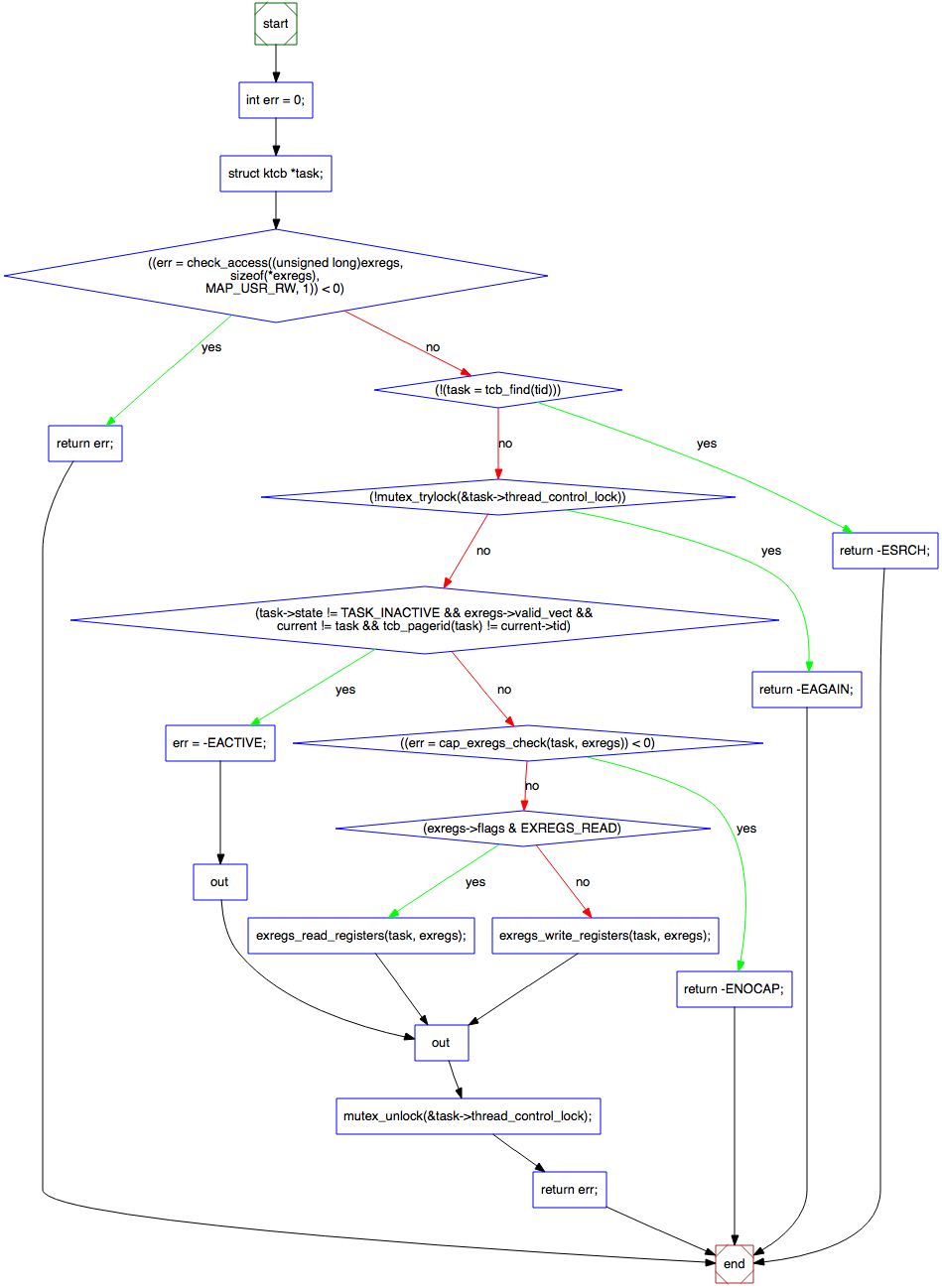


图 2

## 数据结构

本文件涉及到了两个主要的数据结构，分别是ktcb和exregs\_data，在这两个数据结构中，又包含了其他若干重要的结构。

1、ktcb

ktcb实际就是一个线程控制块的数据结构，它的定义如下（来自tcb.h）：

**struct** ktcb **{**  
    /\* User context \*/  
    task\_context\_t context**;**  
  
    /\*  
     \* Reference to the context on stack  
     \* saved at the beginning of a syscall trap.  
     \*/  
    syscall\_context\_t **\***syscall\_regs**;**  
  
    /\* Runqueue related \*/  
    **struct** link rq\_list**;**  
    **struct** runqueue **\***rq**;**  
  
    /\* Thread Id information (See space for space id) \*/  
    l4id\_t tid**;**     /\* Global thread id \*/  
    l4id\_t tgid**;**        /\* Global thread group id \*/  
  
    /\* CPU affinity \*/  
    **int** affinity**;**  
  
    /\* Flags to indicate various task status \*/  
    **unsigned** **int** flags**;**  
  
    /\* IPC flags \*/  
    **unsigned** **int** ipc\_flags**;**  
  
    /\* Lock for blocking thread state modifications via a syscall \*/  
    **struct** mutex thread\_control\_lock**;**  
  
    /\* To protect against thread deletion/modification \*/  
    **struct** spinlock thread\_lock**;**  
  
    u32 ts\_need\_resched**;**    /\* Scheduling flag \*/  
    **enum** task\_state state**;**  
  
    **struct** link task\_list**;** /\* Global task list. \*/  
  
    /\* UTCB related, see utcb.txt in docs \*/  
    **unsigned** **long** utcb\_address**;** /\* Virtual ref to task's utcb area \*/  
  
    /\* Thread times \*/  
    u32 kernel\_time**;**    /\* Ticks spent in kernel \*/  
    u32 user\_time**;**      /\* Ticks spent in userland \*/  
    u32 ticks\_left**;**     /\* Timeslice ticks left for reschedule \*/  
    u32 ticks\_assigned**;** /\* Ticks assigned to this task on this HZ \*/  
    u32 sched\_granule**;**  /\* Granularity ticks left for reschedule \*/  
    **int** priority**;**       /\* Task's fixed, default priority \*/  
  
    /\* Number of locks the task currently has acquired \*/  
    **int** nlocks**;**  
  
    /\* Task exit code \*/  
    **unsigned** **int** exit\_code**;**  
  
    /\* Page table information \*/  
    **struct** address\_space **\***space**;**  
  
    /\* Container \*/  
    **struct** container **\***container**;**  
  
    /\* Other related threads \*/  
    **struct** ktcb **\***pager**;**  
    **int** nchild**;**  
  
    /\* Fields for ipc rendezvous \*/  
    **struct** waitqueue\_head wqh\_recv**;**  
    **struct** waitqueue\_head wqh\_send**;**  
    l4id\_t expected\_sender**;**  
  
    /\* Waitqueue for notifiactions \*/  
    **struct** waitqueue\_head wqh\_notify**;**  
  
    /\* Waitqueue for pagers to wait for task states \*/  
    **struct** waitqueue\_head wqh\_pager**;**  
  
    /\* Tells where we are when we sleep \*/  
    **struct** spinlock waitlock**;**  
    **struct** waitqueue\_head **\***waiting\_on**;**  
    **struct** waitqueue **\***wq**;**  
  
    /\*  
     \* Extended ipc size and buffer that  
     \* points to the space after ktcb  
     \*/  
    **unsigned** **long** extended\_ipc\_size**;**  
    **char** extended\_ipc\_buffer**[];**  
**};**

本文件中还用到数据结构task\_content\_t，该结构在ktcb中首先被定义。它的定义如下（来自include/l4/glue/arm/context.h）：

**typedef** **struct** arm\_context **{**  
    u32 spsr**;**   */\* 0x0 \*/*  
    u32 r0**;**     */\* 0x4 \*/*  
    u32 r1**;**     */\* 0x8 \*/*  
    u32 r2**;**     */\* 0xC \*/*  
    u32 r3**;**     */\* 0x10 \*/*  
    u32 r4**;**     */\* 0x14 \*/*  
    u32 r5**;**     */\* 0x18 \*/*  
    u32 r6**;**     */\* 0x1C \*/*  
    u32 r7**;**     */\* 0x20 \*/*  
    u32 r8**;**     */\* 0x24 \*/*  
    u32 r9**;**     */\* 0x28 \*/*  
    u32 r10**;**    */\* 0x2C \*/*  
    u32 r11**;**    */\* 0x30 \*/*  
    u32 r12**;**    */\* 0x34 \*/*  
    u32 sp**;**     */\* 0x38 \*/*  
    u32 lr**;**     */\* 0x3C \*/*  
    u32 pc**;**     */\* 0x40 \*/*  
**}** \_\_attribute\_\_**((**\_\_packed\_\_**))** task\_context\_t**;**

可以看到，task\_content\_t其实就是指的线程的寄存器上下文环境，另外要主要的是，该除定义指向了arm架构的寄存器定义，属于体系结构相关的代码。

1. 数据结构exregs\_data用于交换寄存器，定义如下（来自exregs.h）：

**struct** exregs\_data **{**  
    exregs\_context\_t context**;**  
    u32 valid\_vect**;**  
    u32 flags**;**  
    l4id\_t pagerid**;**  
    **unsigned** **long** utcb\_address**;**  
**};**

该结构主要包含一个数据结构exregs\_context\_t，该定义也位于include/l4/glue/arm/context.h，定义内容与task\_context\_t相同，都是寄存器上下文，定义如下:

typedef struct arm\_exregs\_context {

u32 r0; /\* 0x4 \*/

u32 r1; /\* 0x8 \*/

u32 r2; /\* 0xC \*/

u32 r3; /\* 0x10 \*/

u32 r4; /\* 0x14 \*/

u32 r5; /\* 0x18 \*/

u32 r6; /\* 0x1C \*/

u32 r7; /\* 0x20 \*/

u32 r8; /\* 0x24 \*/

u32 r9; /\* 0x28 \*/

u32 r10; /\* 0x2C \*/

u32 r11; /\* 0x30 \*/

u32 r12; /\* 0x34 \*/

u32 sp; /\* 0x38 \*/

u32 lr; /\* 0x3C \*/

u32 pc; /\* 0x40 \*/

} \_\_attribute\_\_((\_\_packed\_\_)) exregs\_context\_t;

可见，该结构与task\_content\_t一样，都是与体系结构相关的代码。

## 代码注释

略

## 疑问和改进建议

略

## 体系结构和平台相关的代码

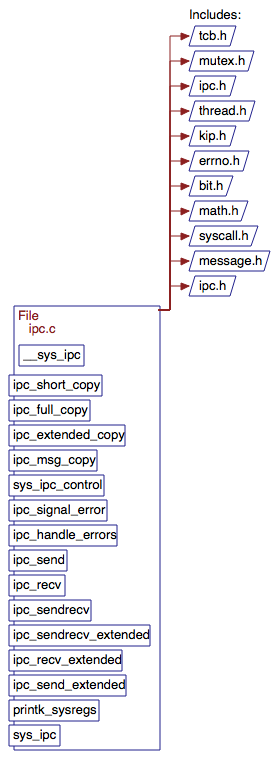
本文件代码没有平台相关代码。

但是涉及到的include/l4/glue/arm/context.h文件中的两个数据结构task\_content\_t和exregs\_content\_t是体系结构相关的，详见6.2。

# 文件4：ipc.c

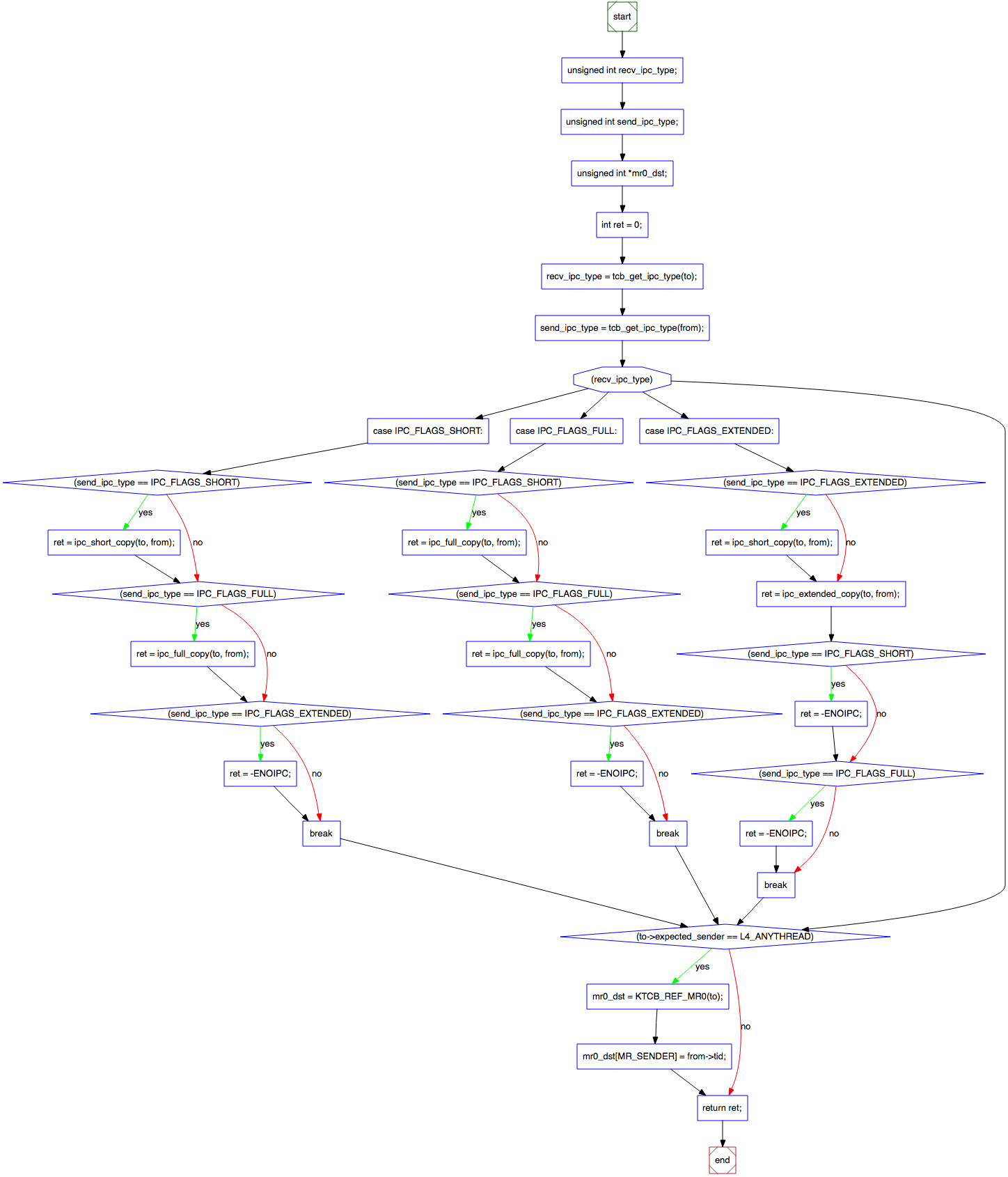
## 功能描述

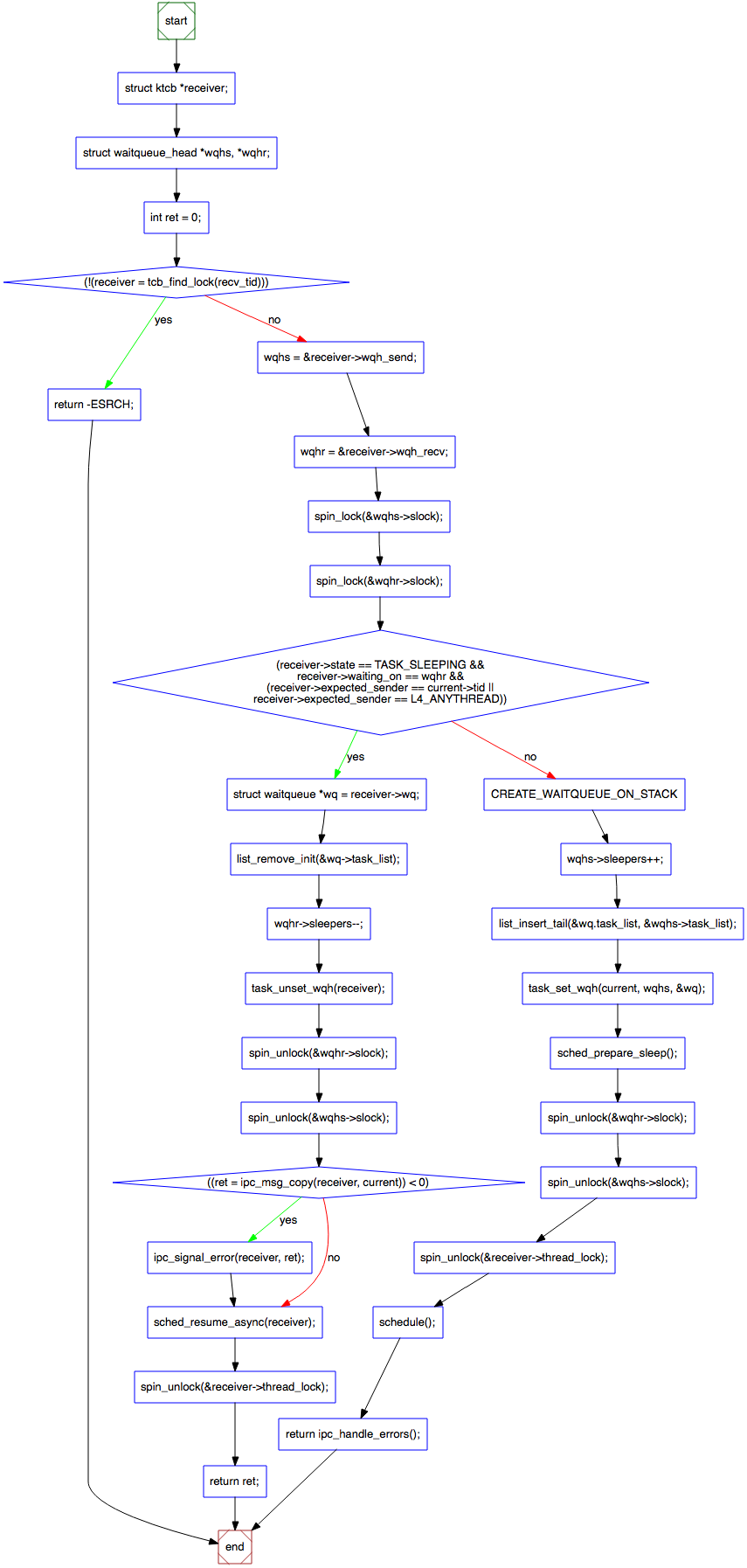
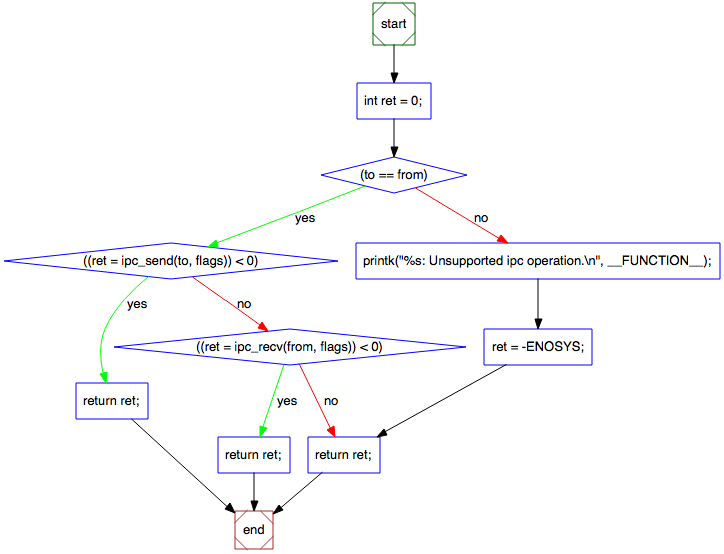
本文件包含了程序间通信（inter-process communication，即ipc）的API代码，包含的函数比较多。所有的函数及包含的头文件如下图所示：



1. ipc.c中首先提供了4个用于在线程间复制信息的函数，下面分别介绍：
   1. int ipc\_short\_copy(struct ktcb \*to, struct ktcb \*from)。   
      该函数的功能是以两个内核tcb为输入，把from线程上下文的MR0寄存器拷贝to线程的上下文中。其中，MR0寄存器在include/l4/glue/arm/message.h中定义为r3。
   2. **int** ipc\_full\_copy**(struct** ktcb **\***to**,** **struct** ktcb **\***from**)**该函数同样以两个内核tcb为输入，效果是把所有的用户空间tcb即utcb的内容进行拷贝。分为三步进行：
      1. 调用ipc\_short\_copy进行MR0的拷贝
      2. 检查两个utcb的内存访问是否会出错，若有错误，则返回错误并结束。
      3. 把from的utcb的寄存器信息拷贝到to中，结束。
   3. **int** ipc\_extended\_copy**(struct** ktcb **\***to**,** **struct** ktcb **\***from**)**该函数的作用是把from的内核栈缓冲区拷贝至to线程。
   4. **int ipc\_msg\_copy(struct ktcb\*to, struct ktcb \*from)**该函数的功能较为强大，能够根据ktcb中的ipc\_type字段进行信息的复制。Ipc\_type字段有三种可能性，分别是short, full和extended。进行何种复制根据from和to的ipc\_type字段来进行。  
      函数的流程图如下所示：  
        
      根据流程图可知，信息复制的原则是这样的：

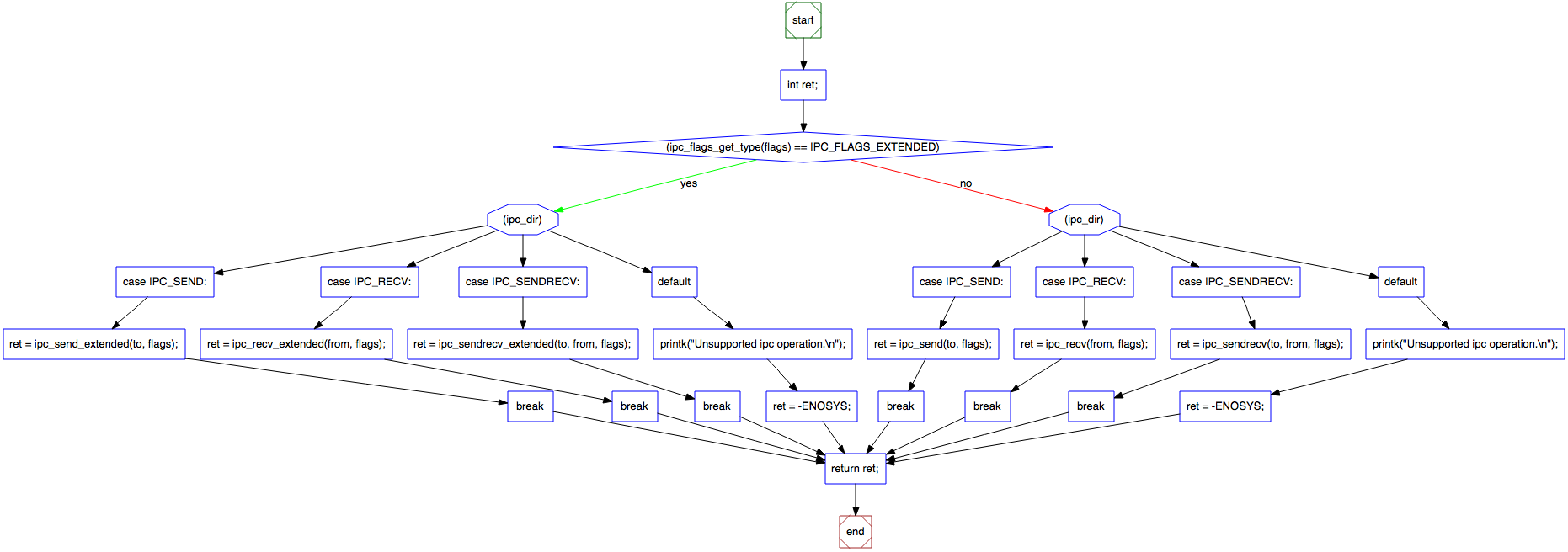
*\*SHORT    SHORT       -> SHORT IPC  
\*FULL     FULL/SHORT  -> FULL IPC  
\*EXTENDED EXTENDED    -> EXTENDED IPC  
\*EXTENDED NON-EXTENDED    -> ENOIPC*



1. int sys\_ipc\_control(void)   
   函数简单地返回-ENOSYS
2. **void** ipc\_signal\_error**(struct** ktcb **\***sleeper**,** **int** retval**)**用于处理ipc中发生的错误。作用是根据retval值的不同，为sleeper线程设置ipc\_flags字段。
3. **int ipc\_handle\_errors(void)**该函数的作用是在ipc发送与接收函数的最后，如果当前线程被挂起，则调用该函数。用于检测当前线程的ipc\_flags字段是否被标记，清除ipc\_flags字段并返回合适的错误代码。
4. **int** ipc\_send**(**l4id\_t recv\_tid**,** **unsigned** **int** flags)  
   该函数的作用是以recv\_tid线程为目的线程，当前线程（current线程）为源线程，调用ipc\_msg\_copy进行信息的传递。其执行流程较为复杂，如下图所示  
     
   可以看出，在进行复制前，该函数对目的线程进行了检查，如果可以进行接收，则直接进行信息的复制，否则就要建立等待队列，并调用schedule进行调度。
5. **int** ipc\_recv**(**l4id\_t senderid**,** **unsigned** **int** flags**)**与ipc\_send()函数类似，以senderid为源线程，以当前线程为目的线程，进行消息的传递（使用ipc\_msg\_copy）。其原理和执行流程与ipc\_send()函数类似，在此不再赘述。
6. **int** ipc\_sendrecv**(**l4id\_t to**,** l4id\_t from**,** **unsigned** **int** flags**)**显而易见，该函数是把ipc\_recv和ipc\_send函数合并到了一个函数中。该函数最常用于客户端和服务器线程之间的通信。  
   其执行流程如下：  
   
7. 此外，还提供了两个函数ipc\_send\_extended和ipc\_recv\_extended，作为5和6的扩展。扩展后的ipc函数的区别在于在执行真正的5或者6的ipc函数之前，相应的接收或者发送缓冲区是缺页的。
8. **static** inline **int** \_\_sys\_ipc**(**l4id\_t to**,** l4id\_t from**,** **unsigned** **int** ipc\_dir**,** **unsigned** **int** flags**)**这是一个内置的静态函数。用于在sys\_ipc中调用。是对上述素有函数的包装  
   该函数的功能是根据参数ipc\_dir和flags的值，调用上面的不同的函数进行操作。  
   其中，ipc\_dir的值可能为
   1. **IPC\_SEND**
   2. **IPC\_REVC**
   3. **IPC\_SENDREVC**
   4. **IPC\_INVALID**

而flags则可能为IPC\_FLAGS\_EXTENDED。  
根据二者的组合不同调用不同的函数。例如当ipc\_dir为IPC\_SEND而flags没有EXTENDED标志，则调用\_\_sys\_ipc等于调用ipc\_send(to, flags)

其函数调用流程图如下所示：



1. **void** printk\_sysregs**(**syscall\_context\_t **\***regs**)**该函数的作用是打印参数中的syscall\_context\_t结构中的所有寄存器的值。
2. **int** sys\_ipc**(**l4id\_t to**,** l4id\_t from**,** **unsigned** **int** flags**)  
   该函数是对\_\_sys\_ipc函数的包装，在进行一系列参数合法性和权能的检查后，调用\_\_sys\_ipc函数进行操作。**

## 数据结构

1、struct utcb，即userspace tcb, 用户空间的线程控制块  
定义如下（来自include/l4/glue/tests/utcb.h）  
struct utcb {  
    u32 global\_id;  
    u32 error\_code;gt  
};  
 可见，utcb中的内容比ktcb中少的多，只包含两个无符号整数，一个作为全局的线程号，一个错误号。  
 那ktcb和utcb如何联系在一起呢？  
 在ktcb中有一个utcb\_address字段，指向了utcb的地址。

1. 此外，在本文件中还用到了两个与等待队列的结构：  
    struct waitquene\_head  
    struct waitquene  
   这两个结构定义在include/l4/lib/wait.h中，与本部分联系不大，故略过

## 代码注释

略

## 疑问和改进建议

暂无

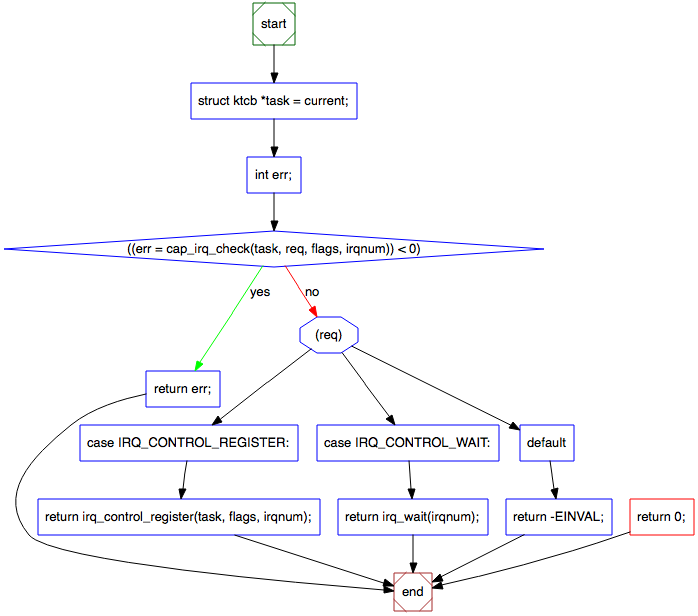
## 体系结构和平台相关的代码

无

# 文件5：irq.c

## 功能描述

本文件定义了对来自用户空间的中断请求进行处理和管理的函数。一共定义了4个函数。

1. **int** irq\_thread\_notify**(struct** irq\_desc **\***desc**)**该函数是用于处理中断请求的函数。执行的效果是将irq\_desc结构对应的utcb的通知栏加1，并唤醒所有在desc上的线程。
2. **int** irq\_control\_register**(struct** ktcb **\***task**,** **int** slot**,** l4id\_t irqnum**)**将给定的中断请求号irqnum对应的中断请求注册到当前线程current。即调用  
   irq\_register(current, slot, irqnum)。  
   根据我的理解，在调用irq\_control\_register时，传入的第一个参数应该始终是current
3. **int** irq\_wait**(**l4id\_t irq\_index**)**该函数的执行效果是使得当前进程current等待索引为irq\_index的中断请求。
4. **int** sys\_irq\_control**(unsigned** **int** req**,** **unsigned** **int** flags**,** l4id\_t irqnum**)**该函数是irq\_control和irq\_wait的包装函数。  
   根据req的值的不同，执行不同的操作。  
   其流程如下：  
     
   即当
   1. req=IRQ\_CONTROL\_REGISTERS时调用irq\_control\_register
   2. req=IRQ\_CONTROL\_WAIT时，调用irq\_wait

## 数据结构

与本文件有关的数据结构为struct irq\_desc，定义如下（include/l4/generic/irq.h）

**struct** irq\_desc **{**  
    **char** name**[**8**];**  
    **struct** irq\_chip **\***chip**;**  
  
    */\* Thread registered for this irq \*/*  
    **struct** ktcb **\***task**;**  
  
    */\* Notification slot for this irq \*/*  
    **int** task\_notify\_slot**;**  
  
    */\* Waitqueue head for this irq \*/*  
    **struct** waitqueue\_head wqh\_irq**;**  
  
    */\* NOTE: This could be a list for multiple handlers for shared irqs \*/*  
    irq\_handler\_t handler**;**  
**};**

## 代码注释

略

## 疑问和改进建议

1. 本文件中的函数**int** irq\_control\_register**(struct** ktcb **\***task**,** **int** slot**,** l4id\_t irqnum**)**第一个参数为一个线程tcb，但函数体中实际上没有用到这个参数，而是只用到了当前线程current。  
   这里是否可以把该参数删掉？

## 体系结构和平台相关的代码

无

# 文件6：kip.c

本文件中无代码。  
数据结构kip在kip.h中的定义如下：

**struct** kip **{**  
    */\* System descriptions \*/*  
    u32 magic**;**  
    u16 version\_rsrv**;**  
    u8  api\_subversion**;**  
    u8  api\_version**;**  
    u32 api\_flags**;**  
  
    u32 container\_control**;**  
    u32 time**;**  
  
    u32 irq\_control**;**  
    u32 thread\_control**;**  
    u32 ipc\_control**;**  
    u32 map**;**  
    u32 ipc**;**  
    u32 capability\_control**;**  
    u32 unmap**;**  
    u32 exchange\_registers**;**  
    u32 thread\_switch**;**  
    u32 schedule**;**  
    u32 getid**;**  
    u32 mutex\_control**;**  
    u32 cache\_control**;**  
      
    u32 arch\_syscall0**;**  
    u32 arch\_syscall1**;**  
    u32 arch\_syscall2**;**  
  
    u32 utcb**;**  
  
    **struct** kernel\_descriptor kdesc**;**  
**}** \_\_attribute\_\_**((**\_\_packed\_\_**));**

可见，kip结构包含了对系统的描述。

此外，kip.h中还定义了时间、版本号等结构。

# 文件7：map.c

## 功能描述

该文件定义了与空间有关（space-related）的系统调用。

1. **int** user\_map\_flags\_validate**(unsigned** **int** flags**)**该函数的功能是检查flags的值是否为下面的几项：
   1. **MAP\_USR\_RO**
   2. **MAP\_USR\_RW**
   3. **MAP\_USR\_RWX**
   4. **MAP\_USR\_RX**
   5. **MAP\_USR\_IO**

如果是，则返回1；否则判断为无效，返回-1。

1. **int** sys\_map

**(unsigned** **long** phys**,** **unsigned** **long** virt**, unsigned** **long** npages**,** **unsigned** **int** flags**,** l4id\_t tid**)**该函数对flags参数调用1中的user\_map\_validate进行检查后，陆续检查前三个参数，通过后，调用函数add\_mapping\_space函数。其功能暂时不详。

1. **int sys\_unmap**(unsigned long virtual, unsigned long npages, unsigned int tid)  
   该函数为2中函数sys\_map的逆操作。最后调用的函数为remove\_mapping\_space

## 数据结构

未引入新的数据结构

## 代码注释

略

## 疑问和改进建议

对于本文件存在较大的疑问：

1. 本文件是有关“space”的系统调用，这个space是否指的是user space和kernel space？
2. 系统调用syscall的map操作该如何理解？与空间space又有何关联？这是codezero中独有的吗？

## 体系结构和平台相关的代码

无

# 文件8：mutex.c

## 功能描述

本文件描述互斥量（mutex）在用户空间的实现。

1. 首先，本文件包含了一些针对互斥量队列、互斥量队列头部的基本操作，下面列出了这些函数的声明，并介绍其功能。
2. **void** init\_mutex\_queue\_head**(struct** mutex\_queue\_head **\***mqhead**)**该函数初始化一个互斥量队列头部
3. **void** mutex\_queue\_head\_lock**(struct** mutex\_queue\_head **\***mqhead**)**该函数对一个互斥量队列头部加锁。
4. **void** mutex\_queue\_head\_unlock**(struct** mutex\_queue\_head **\***mqhead**)**该函数对一个互斥量队列头部解锁
5. **void** mutex\_queue\_init**(struct** mutex\_queue **\***mq**,** **unsigned** **long** physical**)**该函数初始化一个互斥量队列项
6. **void** mutex\_control\_add**(struct** mutex\_queue\_head **\***mqhead**,** **struct** mutex\_queue **\***mq**)**该函数把一个互斥量队列项mq加到一个以mqhead为头部的互斥量队列中。
7. **void** mutex\_control\_remove**(struct** mutex\_queue\_head **\***mqhead**,** **struct** mutex\_queue **\***mq**)**该函数从mqhead中移除队列项mq
8. **struct** mutex\_queue **\***mutex\_control\_find**(struct** mutex\_queue\_head **\***mqhead**, unsignedlong** mutex\_physical**)**该函数遍历以mqhead为头部的队列，对其中的每一个mutex\_queue项，检查其physical字段是否等于mutex\_physical。如果找到这个mutex\_queue，则返回之；否则返回0。
9. **struct** mutex\_queue **\***mutex\_control\_create**(unsigned** **long** mutex\_physical**)**该函数创建一个互斥量队列项mutex\_queue，并初始化之（调用mutex\_queue\_init），并返回之。
10. **void** mutex\_control\_delete**(struct** mutex\_queue **\***mq**)**该函数删除**互斥量队列项mq**
11. 两个互斥量控制函数
    1. **int** mutex\_control\_lock**(struct** mutex\_queue\_head **\***mqhead**,**  
                      **unsigned** **long** mutex\_address**)**
    2. **int** mutex\_control\_unlock**(struct** mutex\_queue\_head **\***mqhead**,**  
                    **unsigned** **long** mutex\_address**,** **int** contenders**)**
12. 包装函数**int** sys\_mutex\_control**(unsigned** **long** mutex\_address**,** **int** mutex\_flags**)**该函数是上述2中两个函数mutex\_control\_lock和mutex\_control\_unlock的包装函数，根据mutex\_flags的值的不同进行不同的操作。  
    mutex\_flags的取值可能为
    1. MUTEX\_CONTROL\_LOCK,
    2. MUTEX\_CONTROL\_UNLOCK

## 数据结构

1. mutex，互斥量的数据结构，定义如下：

**struct** mutex **{**  
    **struct** waitqueue\_head wqh**;**   
    **unsigned** **int** lock**;**  
**};**

该结构包含一个等待队列头部项和一个整数项lock。

2、mutex\_queue\_head：互斥量队列头部  
**struct** mutex\_queue\_head **{**  
    **struct** link list**;**  
    **struct** mutex mutex\_control\_mutex**;**  
    **int** count**;**  
**};**

**该结构**包含了用户空间的全部互斥量。

3、mutex\_queue：互斥量队列项：  
**struct** mutex\_queue **{**  
    **int** contenders**;**  
    **unsigned** **long** physical**;**  
    **struct** link list**;**  
    **struct** waitqueue\_head wqh\_contenders**;**  
    **struct** waitqueue\_head wqh\_holders**;**  
**}；**

该结构包含了两个等待队列头部项，分别是wqh\_contenders和wqh\_holders，代表竞争该互斥量的线程队列和持有锁的线程队列。

## 代码注释

暂略

## 疑问和改进建议

暂无

## 体系结构和平台相关的代码

无

# 文件9：syscall.c

## 功能描述

本文件包含了几个系统调用。

1. **void** print\_syscall\_context**(struct** ktcb **\***t**)**该函数打印指定线程t的系统调用上下文。
2. **int** sys\_schedule**(void)**打印当前函数的函数名\_\_FUNCTION，和当前线程tid
3. **int** sys\_getid**(struct** task\_ids **\***ids**)**获得当前线程的各种序号，包含tid, spid, 和 tgid，保存到ids中。
4. **int** sys\_container\_control**(unsigned** **int** req**,** **unsigned** **int** flags**,** **void** **\***userbuf**)**该函数简单返回0

## 数据结构

1、表示线程各种id的简单数据结构task\_ids：

**struct** task\_ids **{**  
    l4id\_t tid**;**  
    l4id\_t spid**;**  
    l4id\_t tgid**;**  
**};**

## 代码注释

略

## 疑问和改进建议

1、sys\_container\_control函数有何作用？

## 体系结构和平台相关的代码

无

# 文件10：thread.c

## 功能描述

本文件包含与线程有关的系统调用。

本文件包含的函数较多，为了更好地理解文件内容，我们从最后一个函数sys\_thread\_control开始，与之前的文件一样，这个函数是前面所有函数的一个包装，根据传入参数flags的值的不同，对指定id的线程进行操作。

**int** sys\_thread\_control**(unsigned** **int** flags**,** **struct** task\_ids **\***ids**)**

函数中对于flags的值进行判断的代码如下：

**switch** **(**flags **&** THREAD\_ACTION\_MASK**)** **{**  
**case** THREAD\_CREATE**:**  
ret **=** thread\_create**(**ids**,** flags**);**  
**break;**  
**case** THREAD\_RUN**:**  
ret **=** thread\_start**(**task**);**  
**break;**  
**case** THREAD\_SUSPEND**:**  
ret **=** thread\_suspend**(**task**);**  
**break;**  
**case** THREAD\_DESTROY**:**  
ret **=** thread\_destroy**(**task**,** flags**);**  
**break;**  
**case** THREAD\_RECYCLE**:**  
ret **=** thread\_recycle**(**task**);**  
**break;**  
**case** THREAD\_WAIT**:**  
ret **=** thread\_wait**(**task**);**  
**break;**

由此可以总结出，该文件提供了以上6中线程操作，分别是：

1. 线程创建，调用thread\_create函数
2. 启动线程，调用thread\_start函数
3. 挂起线程，调用thread\_suspend函数
4. 销毁线程，调用thread\_destroy函数
5. 再生线程，调用thread\_recycle函数
6. 线程等待，调用thread\_wait函数

## 数据结构

无

## 代码注释

## 疑问和改进建议

## 体系结构和平台相关的代码

本文件中存在两个函数含有体系结构有关的代码，他们的函数命中都带有arch字样。具体与体系结构有关的代码已经用黄色标出。

1、**int** arch\_clear\_thread**(struct** ktcb **\***tcb**)**  
**{**  
    */\* Remove from the global list \*/*  
    tcb\_remove**(**tcb**);**  
    */\* Sanity checks \*/*  
    BUG\_ON**(!**is\_page\_aligned**(**tcb**));**  
    BUG\_ON**(**tcb**->**wqh\_pager**.**sleepers **>** 0**);**  
    BUG\_ON**(**tcb**->**wqh\_send**.**sleepers **>** 0**);**  
    BUG\_ON**(**tcb**->**wqh\_recv**.**sleepers **>** 0**);**  
    BUG\_ON**(!**list\_empty**(&**tcb**->**task\_list**));**  
    BUG\_ON**(!**list\_empty**(&**tcb**->**rq\_list**));**  
    BUG\_ON**(**tcb**->**rq**);**  
    BUG\_ON**(**tcb**->**nlocks**);**  
    BUG\_ON**(**tcb**->**waiting\_on**);**  
    BUG\_ON**(**tcb**->**wq**);**  
  
    */\* Reinitialise the context \*/*  
    memset**(&**tcb**->**context**,** 0**,** **sizeof(**tcb**->**context**));**  
    tcb**->**context**.**spsr **=** ARM\_MODE\_USR**;**  
    */\* Clear the page tables \*/*  
    remove\_mapping\_pgd\_all\_user**(**tcb**->**space**,** **&**current**->**space**->**cap\_list**);**  
    */\* Reinitialize all other fields \*/*  
    tcb\_init**(**tcb**);**  
    */\* Add back to global list \*/*  
    tcb\_add**(**tcb**);**  
    **return** 0**;**  
**}**

**2、int** arch\_setup\_new\_thread**(struct** ktcb **\***new**,** **struct** ktcb **\***orig**,**  
              **unsigned** **int** flags**)**  
**{**  
    */\* New threads just need their mode set up \*/*  
    **if** **(**flags **&** TC\_NEW\_SPACE**)** **{**  
        BUG\_ON**(**orig**);**  
        new**->**context**.**spsr **=** ARM\_MODE\_USR**;**  
        **return** 0**;**  
    **}**  
    BUG\_ON**(!**orig**);**  
    */\* If original has no syscall context yet, don't copy \*/*  
    **if** **(!**orig**->**syscall\_regs**)**  
        **return** 0**;**  
    */\*  
     \* For duplicated threads pre-syscall context is saved on  
     \* the kernel stack. We copy this context of original  
     \* into the duplicate thread's current context structure,  
     \*  
     \* No locks needed as the thread is not known to the system yet.  
     \*/*  
    BUG\_ON**(!(**new**->**context**.**spsr **=** orig**->**syscall\_regs**->**spsr**));** */\* User mode \*/*  
    new**->**context**.**r0 **=** orig**->**syscall\_regs**->**r0**;**  
    new**->**context**.**r1 **=** orig**->**syscall\_regs**->**r1**;**  
    new**->**context**.**r2 **=** orig**->**syscall\_regs**->**r2**;**  
    new**->**context**.**r3 **=** orig**->**syscall\_regs**->**r3**;**  
    new**->**context**.**r4 **=** orig**->**syscall\_regs**->**r4**;**  
    new**->**context**.**r5 **=** orig**->**syscall\_regs**->**r5**;**  
    new**->**context**.**r6 **=** orig**->**syscall\_regs**->**r6**;**  
    new**->**context**.**r7 **=** orig**->**syscall\_regs**->**r7**;**  
    new**->**context**.**r8 **=** orig**->**syscall\_regs**->**r8**;**  
    new**->**context**.**r9 **=** orig**->**syscall\_regs**->**r9**;**  
    new**->**context**.**r10 **=** orig**->**syscall\_regs**->**r10**;**  
    new**->**context**.**r11 **=** orig**->**syscall\_regs**->**r11**;**  
    new**->**context**.**r12 **=** orig**->**syscall\_regs**->**r12**;**  
    new**->**context**.**sp **=** orig**->**syscall\_regs**->**sp\_usr**;**  
    */\* Skip lr\_svc since it's not going to be used \*/*  
    new**->**context**.**pc **=** orig**->**syscall\_regs**->**lr\_usr**;**  
    */\* Distribute original thread's ticks into two threads \*/*  
    new**->**ticks\_left **=** **(**orig**->**ticks\_left **+** 1**)** **>>** 1**;**  
    **if** **(!(**orig**->**ticks\_left **>>=** 1**))**  
        orig**->**ticks\_left **=** 1**;**  
    **return** 0**;**  
**}**

# 总结

1. 版本1.0总结：本次完成1.0版本主要对api目录下的各个文件进行了以下工作：
   1. 列举所有的函数声明，用文字简要表述其功能。
   2. 针对复杂的函数，特别是向上层提供的api函数（一般以sys\_开头），对其流程进行分析，具体表现在画出流程图，并用文件简要介绍其执行流程。
   3. 列举所有第一次出现的数据结构，用文字概括其含义。
   4. 列举出所有与体系结构相关的代码。