F9-Kernel Note

大家到Wiki上新增一下自己的名字還有ID吧！

http://wiki.csie.ncku.edu.tw/embedded/f9-kernel

-
-
- **在STM32F4Discovery上執行**

需要的套件：

- arm-none-eabi- toolchain，必須支援Cortex-M4F
  - Sourcey CodeBench
    - arm-2012.03, arm-2013.05, arm-2013.11
  - GNU Tools for ARM Embedded Processors
    - 4.8-2013-q4-major
- libncurses5-dev
- stlink

步驟：

1. git clone https://github.com/f9micro/f9-kernel.git f9-kernel
1. cd f9-kernel
1. make config(使用預設設定即可)
1. make
1. 將STMF4Discovery接上電腦
1. make flash

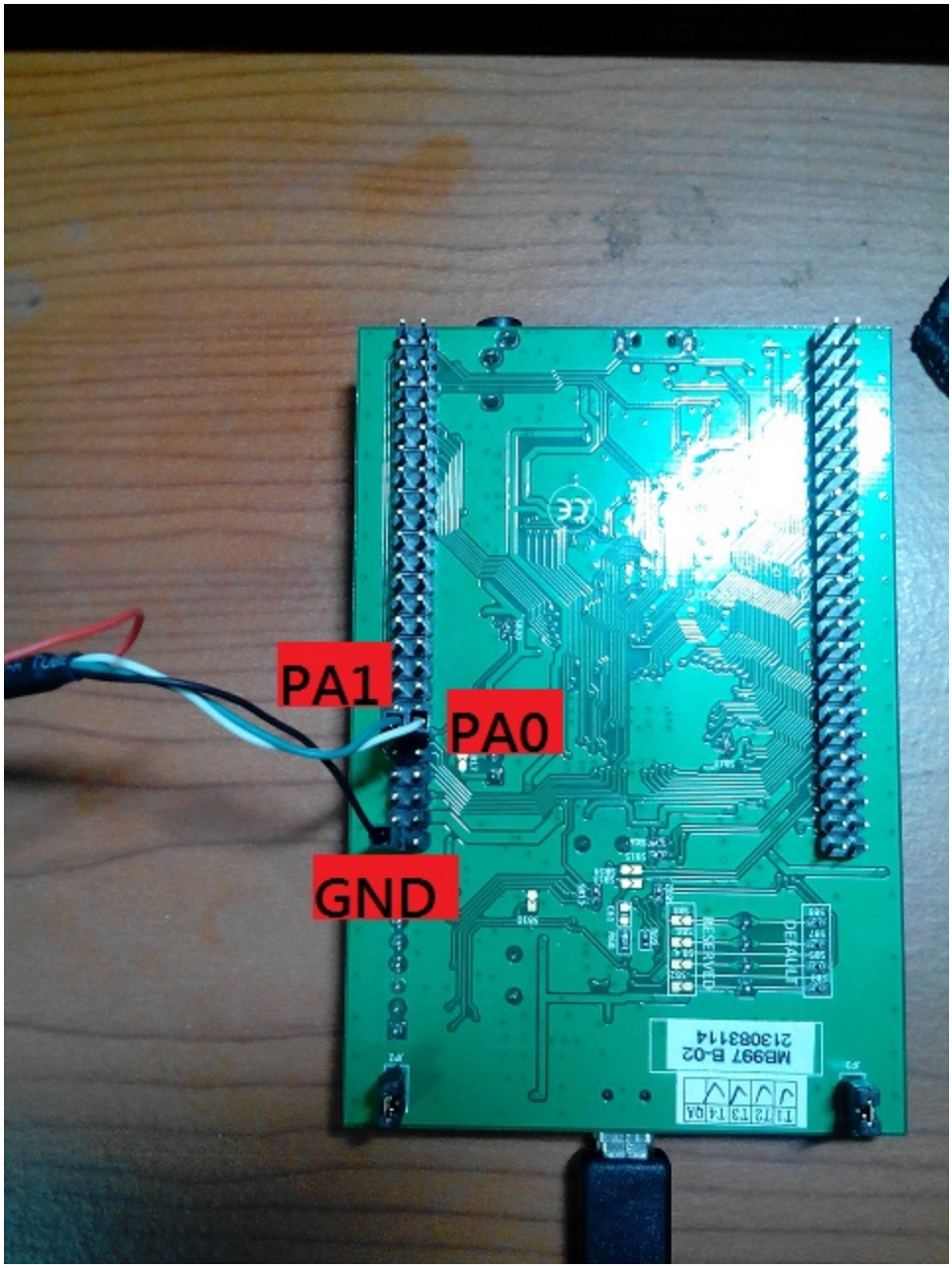
KDB(in-kernel debugger)

Debug工具，預設是會在開機時啟動，支援以下指令

- a： dump address spaces
- m：dump memory pools
- t：dump threads
- s：show softriqs
- n：show timer(now)
- e：dump ktimer events
- K：print kernel tables

步驟：

1. 預設會使用USART4：PA0(TX), PA1(RX)，所以先將pin腳接上，白線接PA0、綠線接PA1、黑線接GND
1. 確認USB連上電腦
1. screen /dev/ttyUSB0 115200 8n1



Troubleshooting

- 無法make flash - 電腦要先連接上STM32F4Discovery才能進行燒錄
- 沒有`screen` - 執行sudo apt-get install screen
- screen連接失敗 - 執行sudo screen /dev/ttyUSB0 115200 8n1
- screen傳輸的資料有問題 - 重新接線，重新啟動STM32F4

可改用 neocon: http://wiki.openmoko.org/wiki/NeoCon

neocon 可以自動連線，會比 screen 方便

試了一下 neocon 在 mac os x 10.9.2 也能跑

neocon 很久沒維護了，也許之後可以改進 keybinding

- **The GNU linker(Linker Scripts)**

Linker scripts的主要目的是描述要如何map輸入檔的section到輸出檔，以及控制輸出檔的memory layout，大部分的linker script都只做了這個。`arm-none-eabi-ld --verbose`可以看到預設的linker script。

Basic Linker Script Concepts

linker會將多個輸入檔組合成一個單一的輸出檔，每一個輸入檔與輸出檔都是object file，輸出檔也叫做executable，每一個object file都會有一串section，有時候會稱輸入檔的section為input section；而輸出檔的section為output section。

每一個section都會有名字與大小，大部分的section還會有一塊相關連的資料區塊(data block)，叫做`section contents`，一個section可能會被做上標記：

- `loadable` - 代表輸出檔在執行時，這塊section的content要被load到記憶體
- `allocatable` - 一個section沒有content時可能被做上這個標記，代表區域在記憶體中應該被放置到一旁，但不會有東西要load進來（有的時候這些記憶體應該被歸0）
- 兩個都不是 - 通常會包含一些debug資訊

每一個`loadable`或是`allocatable`的output section都會有兩個地址：

- VMA(virtual memory address) - 代表這個輸出檔在執行時，這個section的位置
- LMA(load memory address) - 這是在section load進去時的位置

大部分時候這兩個值會一樣，舉一個會不同的例子：當一個data section被load進ROM，接著在程式開始執行時被複製到RAM(這個技巧通常是在一個ROM system中初始化全域變數時用到)，在這個例子中，LMA是ROM的位置，VMA是RAM的位置。

`arm-none-eabi-objdump -h`可以看到輸出檔的section資訊。

每一個object file都會有一串`symbols(symbol table)`，一個symbol可以是defined或是undefined，每一個symbol都會有名字，而每一個defined symbol會有一個地址。如果是編譯C或C++的程式，會從每一個defined function、defined global variable、static variable取得defined symbol，每一個undefined function、undefined global variable會變成undefined symbol。

`arm-none-eabi-objdump -t`可以看到輸出檔的symbol資訊

Linker Script Format

Linker Script是個純文字檔，linker script會是一連串的指令，每一個指令可能會是一個`keyword`後面接著參數或是一個symbol的賦值，使用分號分隔指令，空白會被忽略。

字串可以直接輸入，如果檔名中有逗號會被當作分開的檔案對待，可以用雙引號框起來，檔名中不能有雙引號，不然無法處理。

註解方式與C一樣：`/* ... */`

Simple Linker Script Example

大部分的linker script都是很簡單的，最簡單的linker script只有一個指令：`SECTIONS`，使用`SECTIONS`，去描述記憶體中的memory layout。

`SECTIONS`是一個很強大的指令，這邊會用一個簡單的例子示範。假設你的程式只有包含：code、initialized data、uninitialized data，這樣應該會有`.text`、`.data`、`.bss`區段，假設這些是輸入檔中唯一的區段。

在這個範例中，我們要讓code被load到`0x10000`，而data應該從`0x8000000`開始：

- SECTIONS
- {

```
    . = 0x10000;
    .text : { *(.text) }
    . = 0x80000000;
    .data : { *(.data) }
    .bss : { *(.bss) }
}
```

`.`是一個location counter，如果沒有用其他方法指定output section的地址，則location counter會設定成當前位址，接著加上output section的大小。在SECTIONS指令一開始，location counter會等於0，第四行會定義一個output section `.text`，冒號這邊先不介紹，但這是一個必要的syntax，接著在output section name後面用大括號將要放在這個output section中的input section列出來，`*`會對應到所有的檔案名稱，`*(.text)`代表所有輸入檔的`.text`這段input section。

因為定義output section `.text`的時候，location counter等於`0x10000`，所以linker會將`.text`區段放在輸出檔的`0x10000`位置。

接下來的指令會將`.data`放在`0x80000000`，而location counter在定義完`.data`區段後會等於`0x80000000`加上`.data`的大小，所以`.bss`區段會接在`.data`區段的後面

linker會透過增加location counter去確保每一個區段有對齊的需求。在這個範例中`.text`跟`.data`區段可能不需要對齊，但可能會在`.data`與`.bss`區段中塞入空白。

Simple Linker Script Command

- **Setting The Entry Point**

程式執行的第一道指令叫做`entry point`，可以使用`ENTRY`這個指令設定程式進入點，參數是一個symbol name：`ENTRY(symbol)`。

有許多方法可以設定程式進入點，linker會按順序嘗試以下的試定方法，直到有其中一個成功：

    - `-e` command line選項
    - `ENTRY(symbol)`指令
    - 如果target有定義特定的symbol，大部分的target上是`start`，但PE跟BeOS的系統會檢查一連串可能的entry symbol
    - `.text`區段的第一個byte
    - 位址0
- **Command Dealing with Files**

`INCLUDE filename`，include一個linker script檔案，會在當前目錄搜尋檔案或是任何用`-L`選項指定的路徑，可以巢狀呼叫`INCLUDE`達十層的深度，可以將`INCLUDE`放在top level、MEMORY或是SECTION指令中、output section description。

`INPUT(file, file, ...)、INPUT(file file ...)`，直接include一個檔案到linker，例如想要在每次link的時候include `subr.o`，但又不能將他放到每一個link command line中，那就可以在linker script中用`INPUT(subr.o)`。

- **Assign alias names to memory regions**

透過`MEMORY`指令可以建立alias name

- **F9開機過程**
    - // kernel/start.c
    - void __l4_start(void)
    - {
    - run_init_hook(INIT_LEVEL_EARLIEST);
    -
    -
    - ...
    -
    -

```
    /* entry point */
    main();
}


int main(void) {
    run_init_hook(INIT_LEVEL_PLATFORM_EARLY);
    ...
}


// kernel/init.c
int run_init_hook(unsigned int level)
{
    unsigned int max_called_level = last_level;


    for (const init_struct *ptr = init_hook_start; ptr != init_hook_end;
++ptr)
            if ((ptr->level > last_level) && (ptr->level <= level)) {
                    max_called_level = MAX(max_called_level, ptr->level);
                    ptr->hook();
            }


    last_level = max_called_level;
```

```
    return last_level;
}
```

NOTE: extra efforts inside linker script in oder to hook up these registered init functions.

Tip: http://kunyichen.wordpress.com/2014/04/18/f9-kernel-%E4%B9%8B-init_hook/

OK! I wrote down how I trace on init hook mechanism. It's really a cool thing.

Init Hook

F9-kernel用了一個global initialization hook的技巧，這個技巧可以在任意地方定義一段要在系統初始化時執行的code。一個init hook會在特定的run level被呼叫，hook可以保證依據level順序呼叫，單不能保證在同一個level中呼叫的順序，下面是一個init hook的結構：

```
// include/init_hook.h
typedef struct {
    unsigned int level;
    init_hook_t hook;
    const char *hook_name;
} init_struct;
```

其中包含要在哪個level呼叫、要執行的code位置、名稱，定義這個結構的方法如下：

```
// include/init_hook.h
#define INIT_HOOK(_hook, _level)                                \
    const init_struct _init_struct_##_hook                      \
                __attribute__((section(".init_hook"))) = {      \
```

```
        .level = _level,                                \
        .hook = _hook,                                  \
        .hook_name = #_hook,                                    \
    };
```

第四行的用意在把這個結構放在`.init_hook`這個section中，接著看linker script：

```
// loader/loader.ld
SECTIONS {
    .text 0x08000000:
    {
        KEEP(*(.isr_vector))
        . = TEXT_BASE;
        text_start = .;
        *(.text*)
        *(.rodata*)
        .init_hook_start = .;
        KEEP(*(.init_hook))
        init_hook_end = .;
        text_end = .;
    } > MFlash
    ...
}
```

在`KEEP(*(.init_hook))`前後各紀錄了一個位置，`init_hook_start`會是section `.init_hook`的開始，`init_hook_end`會是section `.init_hook`的結束。

在F9-kernel中可以已經有一些地方使用到`INIT_HOOK`：

```
$ grep INIT_HOOK kernel/* platform/*
kernel/kdb.c:INIT_HOOK(kdb_init, INIT_LEVEL_KERNEL);
```

- kernel/kprobes.c:INIT_HOOK(kprobe_init, INIT_LEVEL_KERNEL);
- kernel/ksym.c:INIT_HOOK(ksym_init, INIT_LEVEL_KERNEL_EARLY);
- kernel/ktimer.c:INIT_HOOK(ktimer_event_init, INIT_LEVEL_KERNEL);
- kernel/memory.c:INIT_HOOK(memory_init, INIT_LEVEL_KERNEL_EARLY);
- kernel/sched.c:INIT_HOOK(sched_init, INIT_LEVEL_KERNEL_EARLY);
- kernel/syscall.c:INIT_HOOK(syscall_init, INIT_LEVEL_KERNEL);
- kernel/thread.c:INIT_HOOK(thread_init_subsys, INIT_LEVEL_KERNEL);
- platform/debug_device.c:INIT_HOOK(dbg_device_init_hook, INIT_LEVEL_PLATFORM);

接著看一下`init_hook_start`跟`init_hook_end`的值：

- $ arm-none-eabi-readelf -s f9.elf | grep "init_hook_start|init_hook_end" -E
- 765: 08005924 0 NOTYPE GLOBAL DEFAULT 1 init_hook_end
- 934: 080058b8 0 NOTYPE GLOBAL DEFAULT 1 init_hook_start

最後看一下剛剛定義的init_struct在哪邊：

- $ arm-none-eabi-objdump -d f9.elf | grep init_struct
- 080058b8 <_init_struct_dbg_device_init_hook>:
- 080058c4 <_init_struct_ktimer_event_init>:
- 080058d0 <_init_struct_memory_init>:
- 080058dc <_init_struct_sched_init>:
- 080058e8 <_init_struct_syscall_init>:
- 080058f4 <_init_struct_thread_init_subsys>:
- 08005900 <_init_struct_kdb_init>:
- 0800590c <_init_struct_kprobe_init>:
- 08005918 <_init_struct_ksym_init>:

可以發現`0x080058b8~0x08005924`剛好就是剛剛定義的init_struct內容(一個init_struct的大小是12byte，所以最後一個是0x8005918+12=0x8005924)。剩下的就是如何執行這些code：

```
// kernel/init.c
extern const init_struct init_hook_start[];
extern const init_struct init_hook_end[];
static unsigned int last_level = 0;


int run_init_hook(unsigned int level)
{
    unsigned int max_called_level = last_level;


    for (const init_struct *ptr = init_hook_start;
            ptr != init_hook_end; ++ptr)
            if ((ptr->level > last_level) &&
                    (ptr->level <= level)) {
                    max_called_level = MAX(max_called_level, ptr->level);
                    ptr->hook();
            }


    last_level = max_called_level;
```

```
    return last_level;
}
```

這段程式會從`init_hook_start`開始掃過一遍，當發現一個hook的level是大於上次呼叫`run_init_hook`而且小於等於這次要run的level時，就執行對應的hook function，並且更新最大呼叫過的level。

Refer from

- http://kunyichen.wordpress.com/2014/04/18/f9-kernel-%E4%B9%8B-init_hook/
- Documentation/init-hooks.txt from f9-kernel

- **Thread**

Thread Control Block

thread is being used at here rather than task

```
//include/thread.h
struct tcb {
    l4_thread_t t_globalid;
    l4_thread_t t_localid;
    thread_state_t state;

    memptr_t stack_base;
    size_t stack_size;

    context_t ctx;

    as_t *as;
```

```
    struct utcb *utcb;

    l4_thread_t ipc_from;

    struct tcb *t_sibling;
    struct tcb *t_parent;
    struct tcb *t_child;

};
```

state是說明目前thread的狀態。

sibling, parent, child是建立thread tree所用。

ipc_from根據名字看起來是儲存ipc的目標，可是ipc只有一個嗎？還是負責ipc的thread?

> *請參閱 L4 文件:*
> *http://os.inf.tu-dresden.de/l4env/doc/html/l4sys-l4v2/main.html*
> *能力不足沒辦法從l4文件中看出關聯，我會從F9裡面找原因*
> *另一份: http://www.cse.unsw.edu.au/~cs9242/05/lectures/02-l4.pdf (L4 Programming)*
> *IPC中有 propagation功能，所以ipc_from是真正的發送者？可是utcb中也有sender的存在....*
> *UTCB is a per-thread data structure designated as thread local storage for IPC message registers and private data. The UTCB area in a thread's address space is a virtual address region that is unique for each thread available on the system.*
> *in f9, UTCB area maps to specific virtual registers.*
> *-> http://dev.b-labs.com/api-reference/hypervisor-api/utcb/*
> *UTCB's sender means the actual sender, and the ipc_from is the target that*

thread is waiting for receiving ipc? Did I get the concept right?

The statement is seemingly right. L4's IPC is synchronous. seL4, the third generation microkernel, takes another approach to implement IPC though.

stack_base指向stack位置

為什麼要有localid globalid?

- #define GLOBALID_TO_TID(id) (id >> 14)
- #define TID_TO_GLOBALID(id) (id << 14)

globalid和localid的互換功能，id的type為l4_thread_t

- typedef uint32_t l4_thread_t;

所以我們知道id為unsigned 32bit int，所以global id會是local id往左移動14bit。

Context

ctx應該是為了context switch，放在這邊可能是為了避免建立global stack。

```
//include/thread.h
typedef struct {
    uint32_t sp;
    uint32_t ret;
    uint32_t ctl;
    uint32_t regs[8];
 #ifdef CONFIG_FPU
    uint64_t fp_regs[8];
    uint32_t fp_flag;
 #endif
} context_t;
```

從context_t發現裡面包 含了context switch所需要的資料

- sp是stack pointer, ret應該是return address (lr),

- ctl不知道是什麼，在thread_init_ctx()當中，發現ctl分別設為0x0 和0x3, 是某種control,是thread所擁有的權限，該權限包括了分配 KIP和 UTCB區域的能力。

  不確定F9是否有設定communication的權限需求。

- fp_regs & fp_flag是屬於浮點運算部分。
- context裡面的regs只有八個，分別對應 r4..r11。至於 r0..r3, r12, lr, pc, xpsr屬於不在此結構中。設在sp當中。

  ctl = thread control (register), Check L4 programming for details

  page 16 of http://www.l4ka.org/l4ka/l4-x2-r7.pdf ?

  I saw tcr in page 16, do you mean some part of it same with the tcb in f9?

  I guess it might be the thread control register described in page 16. As description, it's a place for sharing infomration between user and kernel. The content will differ from the requests. It can be seen in context_switch, where it's loaded to r2 register. However I cannot connect the description inside the l4x2-r7.pdf with source because all the data fields defined in page 16 seem to exist in utcb already.

User-level thread control block

User-level thread control block is include in tcb

```
//include/l4/utcb.h
struct utcb {
 /* +0w */
    l4_thread_t t_globalid;
    uint32_t    processor_no;
    uint32_t    user_defined_handle;    /* NOT used by kernel */
    l4_thread_t t_pager;
 /* +4w */
    uint32_t    exception_handler;
    uint32_t    flags;          /* COP/PREEMPT flags (not used) */
```

```
    uint32_t      xfer_timeouts;
    uint32_t      error_code;
  /* +8w */
    l4_thread_t intended_receiver;
    l4_thread_t sender;
    uint32_t      thread_word_1;
    uint32_t      thread_word_2;
  /* +12w */
    uint32_t      mr[8];        /* MRs 8-15 (0-8 are laying in
                  r4..r11 [thread's context]) */
  /* +20w */
    uint32_t      br[8];
  /* +28w */
    uint32_t      reserved[4];
  /* +32w */
  };
```

mr[8]屬於 message registers, br[8]屬於buffer registers,這兩種register都屬於IPC register.

> *F9 implements partial L4 x.2 ABI, and you can check:*
> *http://www.l4ka.org/l4ka/l4-x2-r7.pdf*

Thread create

```
tcb_t *thread_create(l4_thread_t globalid, utcb_t *utcb)
{
    tcb_t *thr;
    int id;
```

```
id = GLOBALID_TO_TID(globalid);


assert(caller != NULL);


if (id < THREAD_SYS ||
globalid == L4_ANYTHREAD ||
globalid == L4_ANYLOCALTHREAD) {
set_caller_error(UE_TC_NOT_AVAILABLE);
return NULL;
}
/* 這邊會先init thread，而且把tcb 空間配置好後回傳 */
thr = thread_init(globalid, utcb);
/* 呼叫create thread的thread為 parent */
thr->t_parent = caller;


/* 以下只是把thread放到 caller的child */
/* Place under */
if (caller->t_child) {
tcb_t *t = caller->t_child;


while (t->t_sibling != 0)
```

```
            t = t->t_sibling;
    t->t_sibling = thr;
    /* thread的 local id由child的數序決定，因此可能不同的thread有同样的local id*/
    thr->t_localid = t->t_localid + (1 << 6);
    } else {
    /* That is first thread in child chain */
    caller->t_child = thr;


    thr->t_localid = (1 << 6);
    }


    return thr;
}
```

最後會回傳tcb，要注意的是這邊的utcb是已经配好的空間，thread_create不會帮忙配置空間，global id也是要預先獲得。

Global id獲得方式

```
L4_INLINE L4_ThreadId_t L4_GlobalId(L4_Word_t threadno, L4_Word_t version)
{
    L4_ThreadId_t t;
    t.global.X.thread_no = threadno;
    t.global.X.version = version;
    return t;
```

```
}
```

Global id 由 18bits 的 thread_no，14 bits的 version所組成。

重點在於 thread_init部分，所以往那部分去看。

Thread Initialization

```
tcb_t *thread_init(l4_thread_t globalid, utcb_t *utcb)
{
    tcb_t *thr;
    /* 首先從thread_table中獲得記憶體 */
    thr = (tcb_t *) ktable_alloc(&thread_table);


    if (!thr) {
    set_caller_error(UE_OUT_OF_MEM);
    return NULL;
    }
    /* 把thread放進thread map中 */
    thread_map_insert(globalid, thr);
    /* local id過後會根据所屬parent的child順序決定，因此這邊先给0x0 */
    thr->t_localid = 0x0;
    ...

    thr->as = NULL;
    thr->utcb = utcb;
    thr->state = T_INACTIVE;
```

```
    dbg_printf(DL_THREAD, "T: New thread: %t @[%p] \n", globalid,
thr);


    return thr;
}
```

首先從thread table中獲得tcb所需的記憶體，不包含utcb，之後把thr放進thread map中，thread map是整理好的數序陣列。目前只看到thread map在 ipc_deliver中會被逐個檢查是否有recv_block & send_block 的情況。

Thread start過程

F9利用IPC來啟動一個thread，以 user/root_thread.c中的 start_thread為例子

```
static void __USER_TEXT start_thread(L4_ThreadId_t t, L4_Word_t ip,
                                     L4_Word_t sp, L4_Word_t stack_size)
{
    L4_Msg_t msg;


    L4_MsgClear(&msg);
    L4_MsgAppendWord(&msg, ip);
    L4_MsgAppendWord(&msg, sp);
    L4_MsgAppendWord(&msg, stack_size);
    L4_MsgLoad(&msg);
```

```
    L4_Send(t);
}
```

只要把 ip, sp, stack_size透過IPC傳送給想要啟動的thread, IPC在偵測到目標thread為T_INACTIVE時候，首先會init thread 的context， 接着就把該thread的狀态改為T_RUNNABLE。

```
// kernel/ipc
void sys_ipc()
{
    ...
    else if (to_thr && to_thr->state == T_INACTIVE &&
                    GLOBALID_TO_TID(to_thr->utcb->t_pager) ==
GLOBALID_TO_TID(caller->t_globalid)) {
            if (ipc_read_mr(caller, 0) == 0x00000003) {
            /* thread start protocol */
            /* stack pointer */
            memptr_t sp = ipc_read_mr(caller, 2);
            /* stack size */
            size_t stack_size = ipc_read_mr(caller, 3);


            dbg_printf(DL_IPC, "IPC: %t thread start\n", to_tid);


            to_thr->stack_base = sp - stack_size;
```

```
            to_thr->stack_size = stack_size;
            /* ipc_read_mr return ip here */
            thread_init_ctx((void *) sp, (void *) ipc_read_mr(caller, 1),
to_thr);
            caller->state = T_RUNNABLE;


            /* Start thread */
            to_thr->state = T_RUNNABLE;


            return;
            }
    ...
}
```

當 thread狀态更改為 T_RUNNABLE之後，就會被scheduler考虑進排程里頭了。

- **IPC**

F9的IPC 性質為同步傳送，舉個同步和非同步的例子。

同步：當 A 要傳送資料给 B 時候，會先檢查 B 是否已经準備好，如果是的話就直接傳送，不是的話就等待对方。傳送過程中不會经過其他buffer，而是直接傳給 B。

非同步：當A要傳送資料給 B 時候，會把資料丢進系統IPC準備好的 queue/buffer 中。當 B 要接收資料的時候，會從 queue/buffer中尋找。過程中需要系统IPC的 buffer/queue做為中轉。

IPC register有两種- message register & buffer register

Message Register

属於IPC Register, 共有 16個,其中0~8在r4~r11,接下来才是mr[0~8].

在L4中，每個MR的值只能被使用一次，一次之後讀取的話會出现undefined結果。

```
//include/ipc.h
static uint32_t ipc_read_mr(tcb_t *from, int i)
{
  if (i >= 8)
   return from->utcb->mr[i - 8];
  return from->ctx.regs[i];
}
```

内容可分為

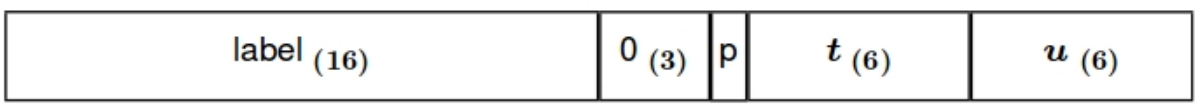
- untyped word
- typed item
  - MapItem
  - GrantItem
  - CtrlXferItem (目前未完成）
  - StringItem （目前未完成）

從 ipc讀取message register的code中可以發現 8以下的mr属于ctx裡的regs。

Message分為三塊區域，分別為：

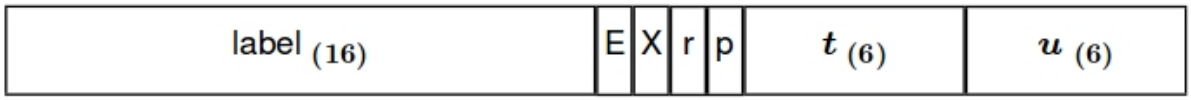
- Message Tag ： MR[0]
- Untyped Words ： MR[1...u]
- Typed Words ： MR[u+1 ... u+t]

mr[0]是 Message Tag,説明發送message的內容。



- u :内容中 untyped words的數量
- t : words裡面有typed item的數量
- label :使用者自定 opcode
- 0 :保留
- p : 擴展性

除此之外，mr[0]也當做收到的message的內容的Message Tag.



- u :收到的untyped words數量
- t :收到的typed items數量
- E :發生错误，從UTCB中查看ErrorCode
- X :從其他CPU送来的message
- r : message被重新導向
- p :發送者使用propagation,可以從UTCB中找出真正的發送者

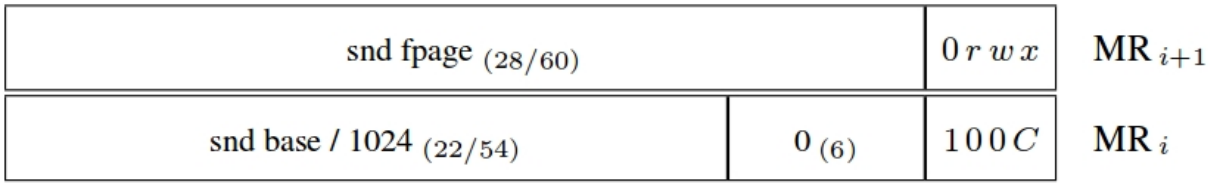
```
//include/l4/ipc.h
typedef union {
    struct {
    /* Number of words */
    uint32_t n_untyped : 6;
    uint32_t n_typed : 6;


    uint32_t prop       : 1;
    uint32_t reserved   : 3;  /* Type of operation */
    uint16_t label;
```

```
    } s;
    uint32_t raw;
} ipc_msg_tag_t;
```

MapItem

Map的動作是透過将要map的fpage組成部分message傳送給Mappee。

Fpage細節由两個words來組成：



- r w x :權限
- snd base :說明 fpage的被mapped位置

  fpage 的設計相較於傳統 UNIX paging，有什麼好處？

  首先是flexible，傳統的UNIX paging的size固定，而fpage不是，因此可以有各種功用。

  fpage裡面就有MPU，可以直接保护fpage的存取。目前看到這兩個，請問还有嗎？

GrantItem

如同Map, Grant也是透過傳送 message完成。

Fpage細節也是两個words:

100C部分由 101C 取代。

CtrlXferItem

Control transfer Item,負责轉換message接收者的一些權限狀态如 instruction

pointer, stack pointer, 或者 general purpose register。

從 L4 繼承而来，但是在F9未找到相關的程式碼，應該是未完成。

StringItem

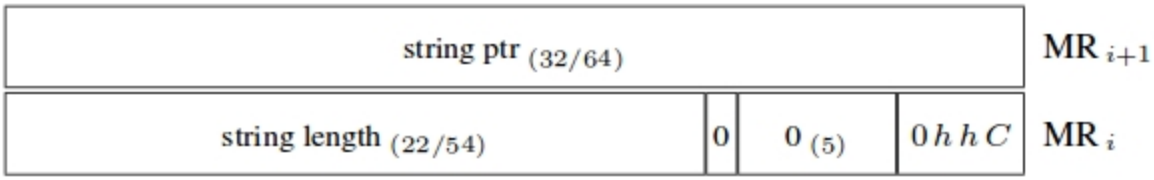
指定user space中一段順序的bytes。最大值為4MB (L4)，在F9中不確定。在發送時候，這字串會被直接複製到接收者的buffer中。

在接收段部分，string item用來指定接收到string的buffer register。

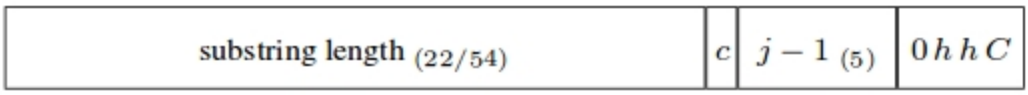
在F9中也是未完成。

StringItem又可以分為連續和不連續的string：

- Simple String
  - 連續性的bytes。



    - string ptr :要發送的字串起始位置或者是接收到字串的buffer起始位置。字串和buffer需要完全符合使用者空間可用位置。
    - string length :要發送的string長度或者是接收的buffer長度。
    - hh :Cache設定。00為處理器預設cache。
- Compound String
  - 一個不連續/鄰近的字串，由落在使用者空間中多個連續性的子字串組成。子字串之間不可以重疊。



    - 0hhC在第一個string descriptor word才需要，後面會被忽略。
    - j :接下来連續的str-ptr word的數量。
    - c :如果是 0，則這compound string descriptor word只有 j 個word的string之後就结束。

- 如果是 1 ，j個word以後會有新的string descriptor word。

Buffer Register

不同於Message Register,全部都在utcb當中，而且BR的值是固定的，直到下次被更改。

BR是StringItem和control transfer Item指定Buffer的目的地。

IPC過程

每一次的IPC syscall都會有發送和接收兩個階段，這兩階段皆可以被忽略。

若目標thread没有在等待接收，則caller thread會進入T_SEND_ BLOCKED狀态。

等待接收的目標可以設定成兩種

- Closed receive:特定thread
- Open wait:任何thread

關於untyped和 typed words,typed Item之後是下一個word會是MapItem, GrantItem 或者 StringItem,其中StringItem部分功能尚未完成,而MapItem和 GrantItem皆屬於Memory配置部分, F9将這一步放到IPC中做處理。Untyped words 是會直接寫進目的thread的mr中。

從pingpong開始trace整个IPC流程，

- //user/apps/pingpong/main.c
- void __USER_TEXT pong_thread(void)
- {

```
    ...
    while(1) {
            msgtag = L4_Receive(threads[PING_THREAD]);
            L4_MsgStore(msgtag, &msg);
    }
}
```

一開始pong會從ping中接收到message tag.

```
//user/include/l4/ipc.h
L4_INLINE L4_MsgTag_t L4_Receive(L4_ThreadId_t from)
{
    /* call L4_Receive_Timeout with no timeout */
    return L4_Receive_Timeout(from, L4_Never);
}


L4_INLINE L4_MsgTag_t L4_Receive_Timeout(
    L4_ThreadId_t from,
    L4_Time_t RcvTimeout)
{
    L4_ThreadId_t dummy;
    /*
    Call L4_Ipc, the reason that using another function call is
    ipc required syscall, and it's different with different hw, so
    using another function call to separate the hw-dependent and
    not-hw-dependent codes.
    */
```

```
    return L4_Ipc(L4_nilthread, from, (L4_Word_t) RcvTimeout.raw,
&dummy);
}
```

這邊利用L4_Receive_Timeout & L4_Ipc把硬體相關和不相關的code分開。

```
// user/lib/l4/platform/syscalls.c
L4_MsgTag_t L4_Ipc(L4_ThreadId_t to,
                   L4_ThreadId_t FromSpecifier,
                   L4_Word_t Timeouts,
                   L4_ThreadId_t *from)
{
    L4_MsgTag_t result;
    L4_ThreadId_t from_ret;


    __asm__ __volatile__(
    "svc %[syscall_num]\n"
    "str r0, %[from]\n"
    : [from] "=m"(from_ret)
    : [syscall_num] "i"(SYS_IPC));


    result.raw = __L4_MR0;


    if (from != NULL)
    *from = from_ret;
```

```
    return result;
}
```

在 L4_Ipc中，呼叫svc,其中變數儲存位置 to在R0, FromSpecifier在R1， Timeouts在R2。

從pong的接收呼叫來看，則是 R0 = L4_nilthread, R1 = ping threads, R2 = L4_NEVER。

接下來在 syscall_handler中，發现是属於 SYS_IPC的呼叫，會将 caller->sp當成參數呼叫 sys_ipc。

```
//kernel/ipc.c
void sys_ipc(uint32_t *param1)
{
    /* TODO: Checking of recv-mask */
    tcb_t *to_thr = NULL;
    l4_thread_t to_tid = param1[REG_R0], from_tid = param1[REG_R1];
    uint32_t timeout = param1[REG_R2];
    /* 所以從 R0 讀取 to_tid, R1 讀取 from_tid, R2 讀取 Timeout */
    /* 當 to_tid == L4_NILTHREAD 時候，表示说不發送资料，只是等待接收 */
    if (to_tid == L4_NILTHREAD && timeout) { /* Timeout/Sleep */
    ipc_time_t t = { .raw = timeout };
    caller->state = T_INACTIVE;
    ktimer_event_create((t.period.m << t.period.e) /
((1000000)/(CORE_CLOCK/CONFIG_KTIMER_HEARTBEAT)), /* millisec
```

```
to ticks */
                    ipc_timeout, caller);
    return;
    }
    /* 當 to_tid != L4_NILTHREAD, 就是要發送資料 */
    if (to_tid != L4_NILTHREAD) {
    to_thr = thread_by_globalid(to_tid);


    if (to_tid == TID_TO_GLOBALID(THREAD_LOG)) {
            user_log(caller);
            caller->state = T_RUNNABLE;
            return;
    } else if ((to_thr && to_thr->state == T_RECV_BLOCKED)
                    || to_tid == caller->t_globalid) {
            /* 這邊要確定 to_thr 的狀态是在等待接收才可以傳送 */
            /* To thread who is waiting for us or sends to myself */
            do_ipc(caller, to_thr);
            return;
    } else if (to_thr && to_thr->state == T_INACTIVE &&
                    GLOBALID_TO_TID(to_thr->utcb->t_pager) ==
GLOBALID_TO_TID(caller->t_globalid)) {
            /* 如果thread狀态是 T_INACTIVE, 則啟動它 */
            if (ipc_read_mr(caller, 0) == 0x00000003) {
            /* thread start protocol */


```

```
            memptr_t sp = ipc_read_mr(caller, 2);
            size_t stack_size = ipc_read_mr(caller, 3);


            dbg_printf(DL_IPC, "IPC: %t thread start\n", to_tid);


            to_thr->stack_base = sp - stack_size;
            to_thr->stack_size = stack_size;


            thread_init_ctx((void *) sp, (void *) ipc_read_mr(caller, 1),
to_thr);
            caller->state = T_RUNNABLE;


            /* Start thread */
            to_thr->state = T_RUNNABLE;


            return;
            } else  {
            /* 如果没有任何 thread在等待接收，則讓自己進入等待發送階段 */
            /* No waiting, block myself */
            caller->state = T_SEND_BLOCKED;
            caller->utcb->intended_receiver = to_tid;
```

```
        dbg_printf(DL_IPC, "IPC: %t sending\n", caller->t_globalid);


        return;
    }
    }
    /* 如果 from_tid == L4_NILTHREAD， 就是不接收资料 */
    if (from_tid != L4_NILTHREAD) {
    /* Only receive phases, simply lock myself */
    caller->state = T_RECV_BLOCKED;
    /* 進入等待接收階段 */
    caller->ipc_from = from_tid;
    /* 設定等待的目標 */
    dbg_printf(DL_IPC, "IPC: %t receiving\n", caller->t_globalid);


    return;
    }


    caller->state = T_SEND_BLOCKED;
}
```

可以看得出，sys_ipc當中包含了發送和接收的階段，可以透過 to_tid和 from_tid的值知道是否要發送或接收。

從 L4_Receive_Timeout中可以發现to_tid的值被設为 L4_NILTHREAD, 因此知道這

次的呼叫只是接收。

sys_ipc比較特別的一點是它包含了啟動 thread,透過發送特定資料（thread start protocol）给目標thread就可以啟動它。這部分的呼叫過程将在thread啟動過程里面叙述。

當 send的目標狀态為 T_RECV_BLOCKED時候，則用 do_ipc把from的message傳送给to。

在 do_ipc()當中，一開始會先讀取tag，然后寫到目的thread的mr[0]當中。

```
//kernel/ipc.c
static void do_ipc(tcb_t *from, tcb_t *to)
{
    ipc_msg_tag_t tag;
    ...
    tag.raw = ipc_read_mr(from, 0);
    ...
    ipc_write_mr(to, 0，tag.raw);
    /* Copy untyped words */
    for (untyped_idx = 1; untyped_idx < untyped_last; ++untyped_idx)
{
    ipc_write_mr(to, untyped_idx, ipc_read_mr(from, untyped_idx));
    }

    typed_item_idx = -1;
    /* Copy typed words
    * FSM: j - number of byte */
```

```
    for (typed_idx = untyped_idx; typed_idx < typed_last; ++typed_idx)
{
    uint32_t mr_data = ipc_read_mr(from, typed_idx);


    /* Write typed mr data to 'to' thread */
    ipc_write_mr(to, typed_idx, mr_data);


    if (typed_item_idx == -1) {
            /* If typed_item_idx == -1 - read typed item's tag */
            typed_item.raw = mr_data;
            ++typed_item_idx;
    } else if (typed_item.s.header & IPC_TI_MAP_GRANT) {
            /* MapItem / GrantItem have 1xxx in header */
            typed_data = mr_data;


            map_area(from->as, to->as,
                    typed_item.raw & 0xFFFFFFC0,
                    typed_data & 0xFFFFFFC0,
                    (typed_item.s.header & IPC_TI_GRANT) ? GRANT : MAP,
                    thread_ispriviliged(from));
            typed_item_idx = -1;
    }


```

```
    /* TODO: StringItem support */
    }

    /* I guess it checked whether the caller waiting for next phase
    (receive phase) at here, if it's not waiting, then unblock it and return
    It checked if to and from stack pointer is not available
    */
    if (!to->ctx.sp || !from->ctx.sp) {
    caller->state = T_RUNNABLE;
    return;
    }
    to->utcb->sender = from->t_globalid;


    to->state = T_RUNNABLE;
    to->ipc_from = L4_NILTHREAD;
    ((uint32_t*)to->ctx.sp)[REG_R0] = from->t_globalid;


    /* If from has receive phases, lock myself */
    from_recv_tid = ((uint32_t*)from->ctx.sp)[REG_R1];
    if (from_recv_tid == L4_NILTHREAD) {
    from->state = T_RUNNABLE;
    } else {
    from->state = T_RECV_BLOCKED;
    from->ipc_from = from_recv_tid;
```

```
    dbg_printf(DL_IPC, "IPC: %t receiving\n", from->t_globalid);
    }
    ...
}
```

當to接收到message之後，狀态将更改為T_RUNNABLE;

當from的send完成後，如果有receive階段，會把from的狀态改成T_RECV_BLOCKED, 并且更改ipc_from。

我是透過這部分猜測 ipc_from代表要從哪邊接收到message.

處理 T_SEND_BLOCKED & T_RECV_BLOCKED

比較特别的一點是，當from在發送時發现to還没有進入T_RECV_BLOCKED狀态時，from會把自己的狀态更改為 T_SEND_BLOCKED。同理，當要等待message時會把自己狀态改為 T_RECV_BLOCKED，因此，就會有專门的schedule來處理它们。

在kernel開始跑起来的時候，有這么一段：

```
// kernel/start.c : main()
ktimer_event_create(64, ipc_deliver, NULL);
```

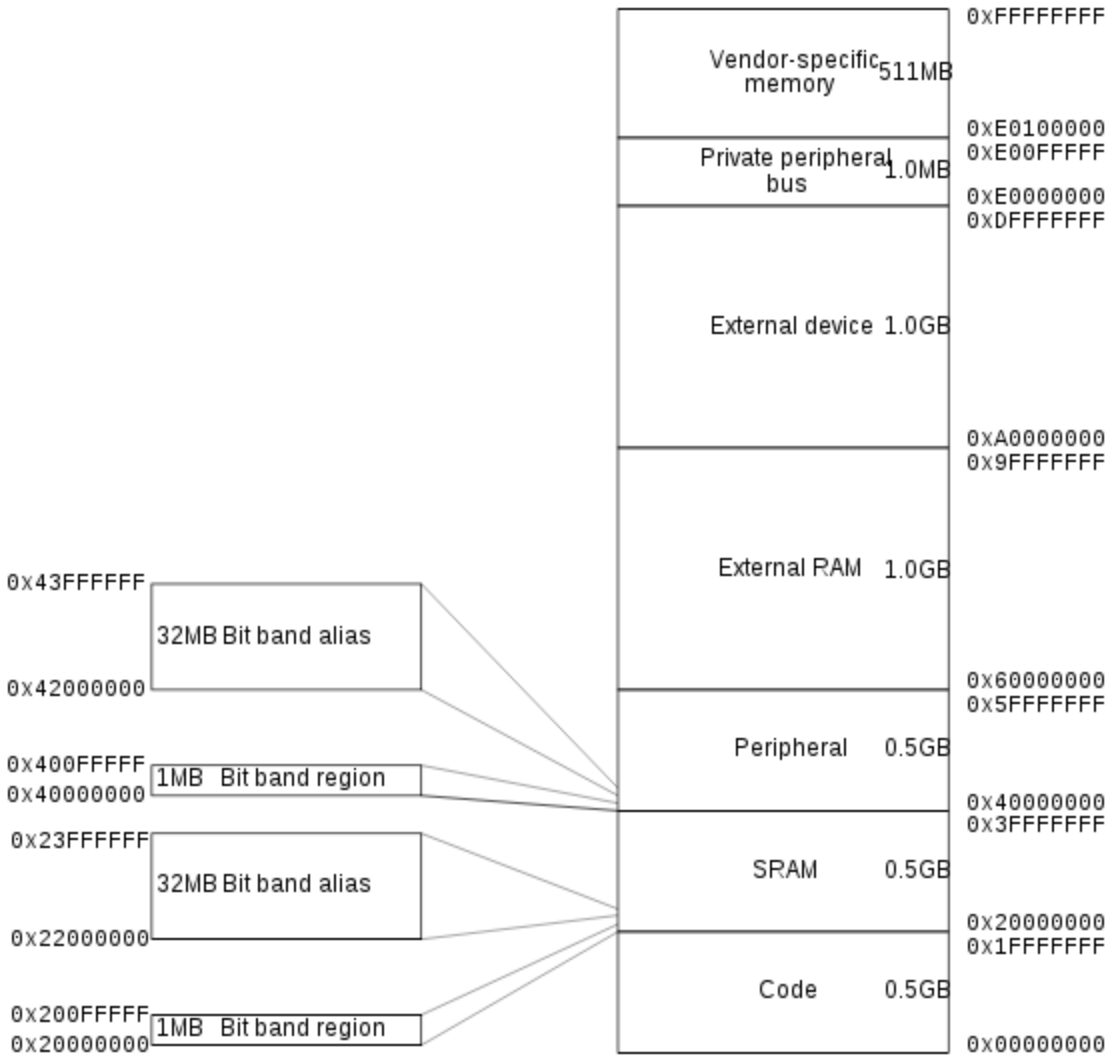
64為 ticks， ipc_deliver是handler，最後是data。ktimer_event_create相關可以往下找ktimer標題。然而這邊就是新增一個event, 然後進行排程，event發生時會呼叫ipc_deliver。

ipc_deliver做的工作就是從thread map中找是否有 T_SEND_BLOCKED或者T_RECV_BLOCKED的 thread，然後檢查ipc目標是否同样處在對應的T_RECV_BLOCKED/T_SEND_BLOCKED狀态，如果是的話，讓它们進行do_ipc()。

```
// kernel/ipc.c
uint32_t ipc_deliver(void *data)
{
    tcb_t *thr = NULL, *from_thr = NULL, *to_thr = NULL;
    l4_thread_t receiver;
    int i;


    for (i = 1; i < thread_count; ++i) {
    thr = thread_map[i];
    switch (thr->state) {
    case T_RECV_BLOCKED:
            if (thr->ipc_from != L4_NILTHREAD &&
            thr->ipc_from != L4_ANYTHREAD) {
            from_thr = thread_by_globalid(thr->ipc_from);
            if (from_thr->state == T_SEND_BLOCKED)
                    do_ipc(from_thr, thr);
            }
            break;
    case T_SEND_BLOCKED:
            receiver = thr->utcb->intended_receiver;
            if (receiver != L4_NILTHREAD && receiver != L4_ANYTHREAD)
{
            to_thr = thread_by_globalid(receiver);
            if (to_thr->state == T_RECV_BLOCKED)
                    do_ipc(thr, to_thr);
            }
```

```
            break;
        default:
            break;
    }
    }


    return 4096;
}
```

- **Memory management system**



memory map會被切成幾個區域，每個區域有定義的memory type，有些區域還會有額外的記憶體屬性，這些type與屬性決定了memory區域存取的行為。

Memory types:

- Normal - The processor can re-order transactions for efficiency, or perform speculative reads
- Device - The processor preserves transaction order relative to other transactions to Device or Strongly-ordered memory.
- Strongly-ordered - The processor preserves transaction order relative to all other transactions.

Device跟Strongly-ordered兩種不同的ordering requirement代表memory system可以

buffer一個寫入到Device memory但不能是Strongly-ordered

Memory attributes:

- shareable - 如果一個記憶體區塊是shareable的話，memory system會在多重的bus中提供資料同步。如果有多個bus可以存取一個non-shareable的區域的話，software要自己確保資料一致性
- Execute Never(XN) - 處理器會避免instruction access，如果有instruction從XN區域執行，則會產生fault exception

- **KIP(Kernel Interface Page)**
  - Kernel memory object
    - map to address space(AS)
  - 包含一些硬體與kernel的資訊
    - kernel版本
    - 支援的feature(page size)
    - physical memory layout
    - system call address

```
// include/l4/kip_types.h
/*
 * NOTE: kip_mem_desc_t differs from L4 X.2 standard
 */
typedef struct {
    uint32_t    base;       /* Last 6 bits contains poolid */
    uint32_t    size;       /* Last 6 bits contains tag */
} kip_mem_desc_t;


typedef union {
    struct {
```

```
            uint8_t  version;
            uint8_t  subversion;
            uint8_t  reserved;
    } s;
    uint32_t raw;
} kip_apiversion_t;


typedef union {
    struct {
            uint32_t  reserved : 28;
            uint32_t  ww : 2;
            uint32_t  ee : 2;
    } s;
    uint32_t raw;
} kip_apiflags_t;


typedef union {
    struct {
            uint16_t memory_desc_ptr;
            uint16_t n;
    } s;
    uint32_t raw;
} kip_memory_info_t;


```

```
typedef union {
    struct {
            uint32_t user_base;
            uint32_t system_base;
    } s;
    uint32_t raw;
} kip_threadinfo_t;


struct kip {
    /* First 256 bytes of KIP are compliant with L4 reference
            * manual version X.2 and built in into flash (lower kip)
            */
    uint32_t kernel_id;
    kip_apiversion_t api_version;
    kip_apiflags_t api_flags;
    uint32_t kern_desc_ptr;


    uint32_t reserved1[17];


    kip_memory_info_t memory_info;


    uint32_t reserved2[20];

```

```
    uint32_t utcb_info;             /* Unimplemented */
    uint32_t kip_area_info;         /* Unimplemented */


    uint32_t reserved3[2];


    uint32_t boot_info;             /* Unimplemented */
    uint32_t proc_desc_ptr;                 /* Unimplemented */
    uint32_t clock_info;            /* Unimplemented */
    kip_threadinfo_t thread_info;
    uint32_t processor_info;        /* Unimplemented */


    /* Syscalls are ignored because we use SVC/PendSV instead of
            * mapping SC into thread's address space
            */
    uint32_t syscalls[12];
};


typedef struct kip kip_t;


// kernel/kip.c
kip_t kip __KIP = {
```

```
    .kernel_id = 0x00000000,


    .api_version.raw = 0x84 << 24 | 7 << 16,        /* L4 X.2, rev 7 */
    .api_flags.raw      = 0x00000000,           /* Little endian 32-bit */
};
```

KIP(L4-x2)

KIP包含了API、kernel版本資訊、system descriptor including memory descriptor、system call link，剩下的部份還未定義。這個page是一個mircokernel object，在AS建立時會直接映射到每個AS。他不是被pager映射，而且不能被map或grant到其他的AS，也不能被unmap。新AS的創建者可以決定要讓KIP被映射到哪些位置，而這些位置在AS的生命週期中會一直存在，任意的thread都能透過KERNELINTERFACE這個system call取得這些位置。

> *What's memory descriptor? I found the definition "The kernel represents a process's address space with a data structure called the memory descriptor." from linux kernel map. I will use this definition temporary until I know what the memory descriptor is exactly.*

- API Version

versio n	subversio n	
0x02		Version 2
0x83	0x80	Experimental Version X.0
0x83	0x81	Experimental Version X.1
0x84	rev	Experimental Version X.2(Revision

rev)

0x85 Dresden L4.Sec

0x86 rev NICTA N1(Revision rev)

0x04 rev Version 4(Revision rev)

- F9-kernel：`.api_version.raw = 0x84 << 24 | 7 << 16` ，L4 X.2, rev 7
- API Flags
  - ee
    - 00：little endian
    - 01：big endian
  - ww
    - 00：32-bit API
    - 01：64-bit API
  - F9-kernel：`.api_flags.raw = 0x00000000` ，Little endian 32-bit
- ProcessInfo

s(4 ~(12/44 processors-1(1

)) 6)

- s：一個一個processor description佔據2^s的空間，第一個processor的description field放在ProcDescPtr，其餘的description會直接放在前一個的後面
- processors：系統可用的processor數量
- F9-kernel沒有實作
- ThreadInfo

UserBase(12SystemBase(1 t(8

) 2))

- UserBase：User thread可以用到的最低的thread number，前三個thread number會給初始thread σ0、σ1、還有root task使用，version number會與這三個中的一個相等。

這邊的前三個thread要在確認與F9-kernel中的關係

- SystemBase：System thread**可以用到的最低的**thread number，**小於這個**thread number**的是硬體中斷**
- t：**有效的**thread number bits
- F9-kernel**沒有實作**t

- ClockInfo

SchedulePrecision(16ReadPrecision(16

))

- SchedulePrecsion：**指定藉由**SYSTEMCLOCK**這個**system call**可以取得最小的時間間隔**
- ReadPrecision：
- F9-kernel**沒有實作**

- UtcbInfo
  - F9-kernel**沒有實作**
- KipAreaInfo
  - F9-kernel**沒有實作**
- BootInfo
  - F9-kernel**沒有實作**
- ProcDescPtr
  - F9-kernel**沒有實作**
- KernDescPtr
  - Points to a region that contains 4 kernel-version words (see below) followed by a number of 0-terminated plaintext strings. The first plaintext string identifies the current kernel followed by further optional kernel-specific versioning information. The remaining plaintext strings identify architecture dependent kernel features (see Appendix A.3). A zero length string (i.e., a string containing only a 0-character) terminates the list of feature descriptions. KernelDescPtr is given as an address relative to the kernel interface page's base address.
  - F9-kerne**沒有用到**
- KernelID
  - **用來辨識**kernel

id	subid	kernel	supplier
0	1	L4/486	GMD
0	2	L4/Pentium	IBM
0	3	L4/x86	UKa
1	1	L4/Mips	UNSW
2	1	L4/Alpha	TUD, UNSW
3	1	Fiasco	TUD
4	1	L4Ka::Hazelnut	UKa
4	2	L4Ka::Pistachio	UKa, UNSW, NICTA
4	3	L4Ka::Strawberry	UKa
5	1	NICTA::Pistachio-embedded	NICTA

- Syscalls
  - F9中使用SVC/PendSV代替映射SC到AS
- MemoryInfo

MemDescPtr(1 n(16

6))

  - MemDescPtr：第一個memory descriptor的位置，之後的memory descriptor會跟在後面，如果有overlap，較晚的descriptor會優先於比較早的descriptor
  - n：memory descriptor的數量
- MemDesc
  - F9的MemDesc與L4 x.2不同

Refer from

- *http://www.l4ka.org/l4ka/l4-x2-r7.pdf*

MPU

MPU會將memory map切成幾塊區域，並定義每一個區域的位置、大小、存取權限還有記憶體屬性

- 對每一個區域獨立設定
- overlapping區域
- export記憶體屬性給系統

記憶體屬性會影響那個區域的記憶體存取，Cortex-M4定義了：

- 八個獨立的記憶體區域，0-7
- 一個背景區域

如果記憶體的區域有overlap，則記憶體存取會受到較高數字區域的屬性影響，例如當有其他的區域與他overlap區域7時，區域7會有最高的優先度。

背景區域會與預設的memory map有同樣的屬性，但只能被priviledge的software存取。

Cortex-M4的memory map是unified的，也就是instruction access與data access會有同樣的區域設定。

如果有程式要存取一塊被MPU所保護的記憶體位置，處理器會觸發一個`MemManage fault`，這會造成fault exception，在OS環境下處理序可能會被中止。在一個OS環境中，kernel可以動態的依據要執行的處理序改變MPU區域設定。一般來說，一個嵌入式OS會使用MPU作為memory protection。

與傳統L4不一樣，主要focus在小的MCU上

- 沒有虛擬記憶體與分頁(virtual memory and page)
- RAM很小，但實體位置空間大(32-bit)，包含硬體、flash、bit-band區域
- 只有八個區域有MPU(memory protection unit)

記憶體管理分成三個設計

- Memory pool(mempool_t) - represent area of PAS with specific attributes(hardcoded in memmap table)
- Flexible page(fpage_t) - 與傳統L4不同，這邊代表MPU區域

- Address page(as_t) - sorted list of fpages bound to specific thread

在cortex-m中只支援2^n大小的MPU區域，所以如果我們要建立一個page是96byte的話，要先切成記小塊的區域，並建立fpage chain包含一個32byte跟一個64byte的fpage，這是實作複雜的原因，最簡單的方法是直接使用一塊標準大小的區域(128byte)，但這樣會非常浪費。

```
// include/memory.h
// 表示實體記憶體位置，flag是kernel與user的存取權限，還有fpage的create rule
typedef struct {
    char *name;
    memptr_t start;
    memptr_t end;


    uint32_t flags;
    uint32_t tag;
} mempool_t;


#define DECLARE_MEMPOOL(name_, start_, end_, flags_, tag_) \
{                                       \
    .name = name_,                      \
    .start = (memptr_t) (start_),       \
    .end  = (memptr_t) (end_),          \
    .flags = flags_,                    \
    .tag = tag_                         \
```

```
}


#define DECLARE_MEMPOOL_2(name, prefix, flags, tag) \
    DECLARE_MEMPOOL(name, \
                    &(prefix ## _start), &(prefix ## _end), flags, tag)
```

請教一個問題如何將MPT_DEVICES (AHB1_1DEV) 授權給user-space thread ? 目前正實作把部分的 GPIO driver code 從 kernel space 移到 user-space.

- **Map Action**
  - `// include/memory.h`
  - `typedef enum { MAP, GRANT, UNMAP } map_action_t;`
  - MAP - **分享記憶體，**memory page**被傳送給另一位使用者，且可被兩者使用**
  - GRANT - memory page**會被授權給新的使用者，而且不能在被前一位使用者使用**
  - UNMAP - **被**map**到其他使用者的**memory page**，會從他們的**AS**中被洗掉**

    這邊是依據L4的動作行為推測，尚未確認在F9中是否一致，看過F9-kernel的README後，確認與L4的動作行為一致

Memory Pool

Flexible Page(fpage)

- `// include/fpage.h`

```
#define FPAGE_ALWAYS      0x1    /*! Fpage is always mapped in MPU
*/
#define FPAGE_CLONE       0x2    /*! Fpage is mapped from other AS */
#define FPAGE_MAPPED      0x4    /*! Fpage is mapped with MAP (
                                 unavailable in original AS) */


/**
 * Flexible page (fpage_t)
 *
 * as_next - next in address space chain
 * map_next - next in mappings chain (cycle list)
 *
 * base - base address of fpage
 * shift - size of fpage == 1 << shift
 * rwx - access bits
 * mpid - id of memory pool
 * flags - flags
 */
struct fpage {
    struct fpage *as_next;
    struct fpage *map_next;
    struct fpage *mpu_next;


    union {
            struct {
```

```
                    uint32_t base;
                    uint32_t mpid : 6;
                    uint32_t flags : 6;
                    uint32_t shift : 16;
                    uint32_t rwx : 4;
            } fpage;
            uint32_t raw[2];
    };


    // for KDE
    int used;
};


int map_fpage(as_t *src, as_t *dst, fpage_t *fpage, map_action_t
action)
{
    fpage_t *fpmap = (fpage_t *) ktable_alloc(&fpage_table);


    /* FIXME: check for fpmap == NULL */
    fpmap->as_next = NULL;
    fpmap->mpu_next = NULL;


    /* Copy fpage description */
```

```
    fpmap->raw[0] = fpage->raw[0];
    fpmap->raw[1] = fpage->raw[1];


    /* Set flags correctly */
    if (action == MAP)
            fpage->fpage.flags |= FPAGE_MAPPED;
    fpmap->fpage.flags = FPAGE_CLONE;


    /* Insert into mapee list */
    fpmap->map_next = fpage->map_next;
    fpage->map_next = fpmap;


    /* Insert into AS */
    insert_fpage_to_as(dst, fpmap);


    dbg_printf(DL_MEMORY, "MEM: %s fpage %p from %p to %p\n",
                    (action == MAP) ? "mapped" : "granted", fpage, src,
dst);


    return 0;
}
```

Address Space(AS)

```
typedef struct {
    uint32_t as_spaceid;      /*! Space Identifier */
    struct fpage *first;      /*! head of fpage list */


    struct fpage *mpu_first;          /*! head of MPU fpage list */
    struct fpage *mpu_stack_first;          /*! head of MPU stack fpage list
*/
    uint32_t shared;          /*! shared user number */
} as_t;
```

Refer from

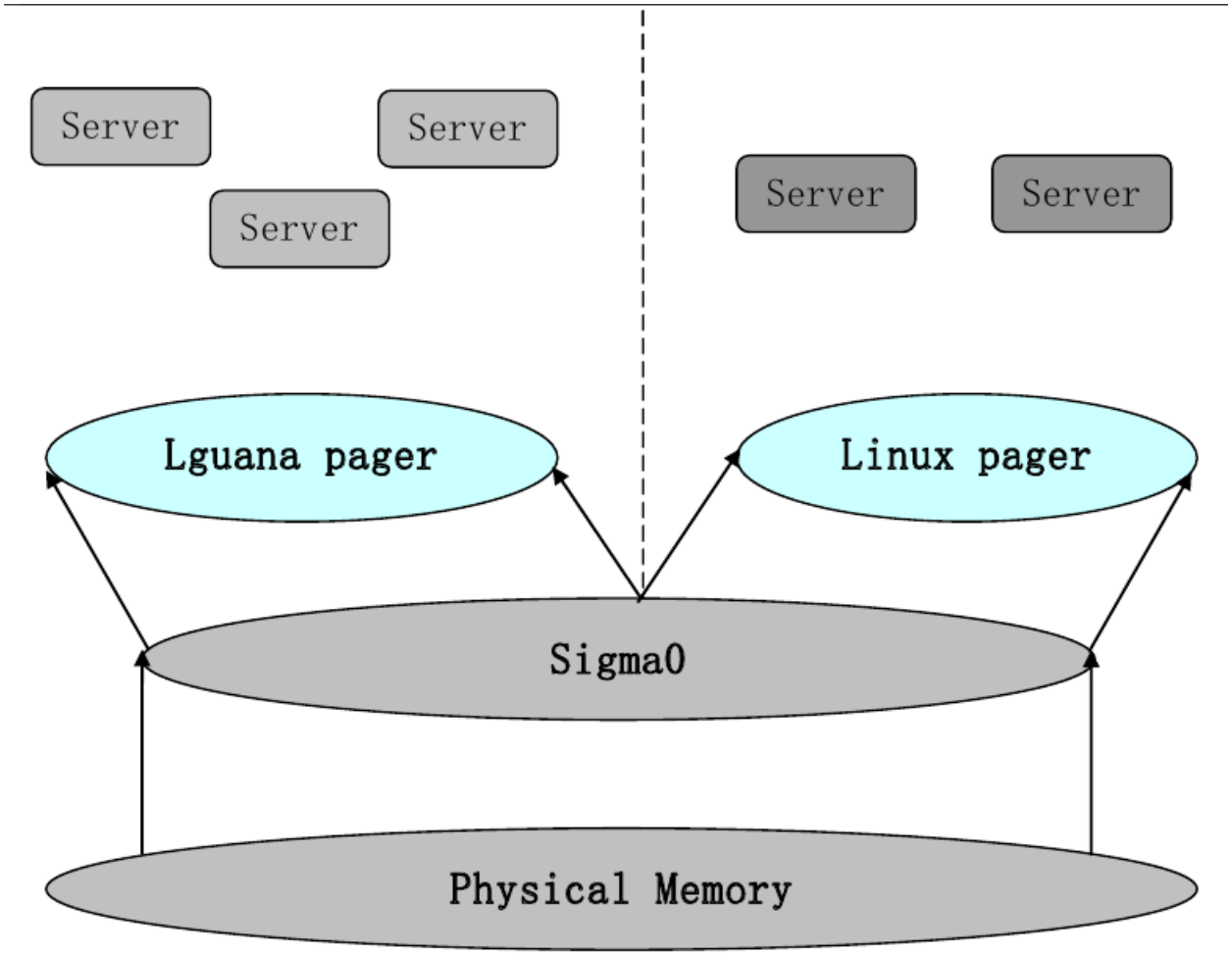
- ARM Information Center(2.2. Memory model)
- ARM Information Center(4.5. Optional Memory Protection Unit)

Memory Management System(L4-Kernel)

- **Memory Mapping Through Pagers**

不同的page可能被劃分給不同的軟體區域，例如：

- Lguana系統提供專屬獨立的page
- Linux系統提供專屬獨立的page



- **User Level Pagers**

Pages提供一層抽象界面給軟體，降低硬體中MPU的複雜度。Pages構成了一層(hierarchy)，該層建立最頂層的策略(policy)，在啟動時從root page搶佔memory，他們在外部的軟體區域間分散的進行記憶體管理，提供更高的安全性。

  - Sigma0是root pager，並且是AS創建的第一個User
    - 內核禁止使用所有物理memory page
    - sigma0是一個與物理memory等冪(1:1)的映射
  - page mapping from sigma0 follow these rules
    - 每個page只被映射一次
    - 先來先服務(first come, first serverd)
  - page被其他page映射

    - transparently during a page fault
    - explicity with the corresponding kernel operation(map, grant)
- **The Root AS**
    - Root AS是AS創建的第二個用戶
    - Root AS的任務是啟動所有那些不是由kernel自己直接啟動的AS
    - Root AS是唯一可以執行priviledge system call的AS

fpage(L4 X.2)

fpage(flexpage)是一塊虛擬地址空間(virtual address space)的區域，一個fpage包含了所有實際上映射到這個區域的page，除了kernel mapped object(UTCB、KIP)。fpage最少有1K，在特定的處理器上，可能更大，KIP上面會有被硬體或kernel支援的大小資訊。

mapped fpage是不可切割的物件，也就是說，當fpage被映射後，mapper不能在部份unmap或map page，整個fpage必須被在一次操作中被unmap，不過mappee可以切割fpage並映射小部份的fpage，部份unmap可能可以也可能不可以在某些系統上運作，kernel無法指出這些動作是否成功。

- **Bitmap**

bit array(bitmap, bitset, bit string, bit vector)是一種緊湊儲存位元的陣列結構，可以用來實作簡單的set結構。在硬體上操作bit-level時，bitmap是一種很有效的方法，一個典型的bitmap會儲存kw個位元，w代表一個單位需要w個位元(byte、word)，k則是一個非負的整數，如果w無法被要儲存的位元整除，則有些空間會因為內部片段被浪費。

定義

bitmap會從某一個domain mapping到一個集合{0, 1}，這個值可以代表valid/invalid、dark/light等等，重點在只會有兩個可能的值，所以可以被存在一個位元中。

基本操作

雖然大部分的機器無法取得或操作記憶體中的單一位元，但是可以透過bitwise操作一個word進而改變單一位元的資料：

- OR可以用來set一個位元為1：11101010 OR 00000100 = 11101110(set 3rd bit 1)
- AND可以用來set一個位元為0：11101010 AND 11111101 = 11101000(set 2nd bit 0)
- AND可以用來判斷某一個位元是否為1：11101010 AND 00000001 = 0(check 1st bit is 1)
- XOR可以用來toggle一個位元：11101010 XOR 00000100 = 11101110(toggle 3rd bit)
- NOT用來invert：NOT 11101010 = 00010101

只要n/w個bitwise operation用來算出兩個相同大小bitmap的union、intersection、difference、complement

```
for i from 0 to n/w-1
    complement[i] := not a[i]
    union[i]      := a[i] or b[i]
    intersection[i] := a[i] and b[i]
    difference[i]   := a[i] and (not b[i])
```

如果要iterate bitmap中的所有bit，只要用一個雙層的迴圈就能有效率的掃完，只需要n/w次的memory access

```
 for i from 0 to n/w-1
    index := 0   // if needed
    word := a[i]
    for b from 0 to w-1
```

- value := word **and** 1 ≠ 0
- word := word **shift right** 1
- *// do something with value*
- index := index + 1 *// if needed*

Bit-banding

bit-banding會將一塊較大記憶體中的word對應到一個較小的bit-band區域中的單一bit，例如寫到其中一個alias，可以set或是clear一個bit-band區域中對應的bit。

這使得bit-band區域中每一個獨立的bit都可以透過LDR指令搭配一個word-aligned的地址進行存取，也能讓每一個獨立bit被直接toggle，而不須經過read-modify-write的指令操作。

處理器的memory map包含了兩塊bit-band區域，分別是在SRAM以及Peripheral中最低位的1MB。

System bus interface包含了一個bit-band的存取邏輯

- remap一個bit-band alias到bit-band區域
- 讀取時，會將requested bit放在回傳資料的Least Significant Bit中
- 寫入時，會將read-modify-write轉換成一個atomic的動作
- 處理器在bit-band操作中不會stall，除非試圖在bit-band操作中存取system bus

記憶體中有兩塊32MB的alias對應到兩塊1MB的bit-band區域

- 32MB可存取的SRAM alias區域對應到1MB的bit-band SRAM區域
- 32MB可存取的peripheral alias區域對應到1MB的bit-band peripheral區域

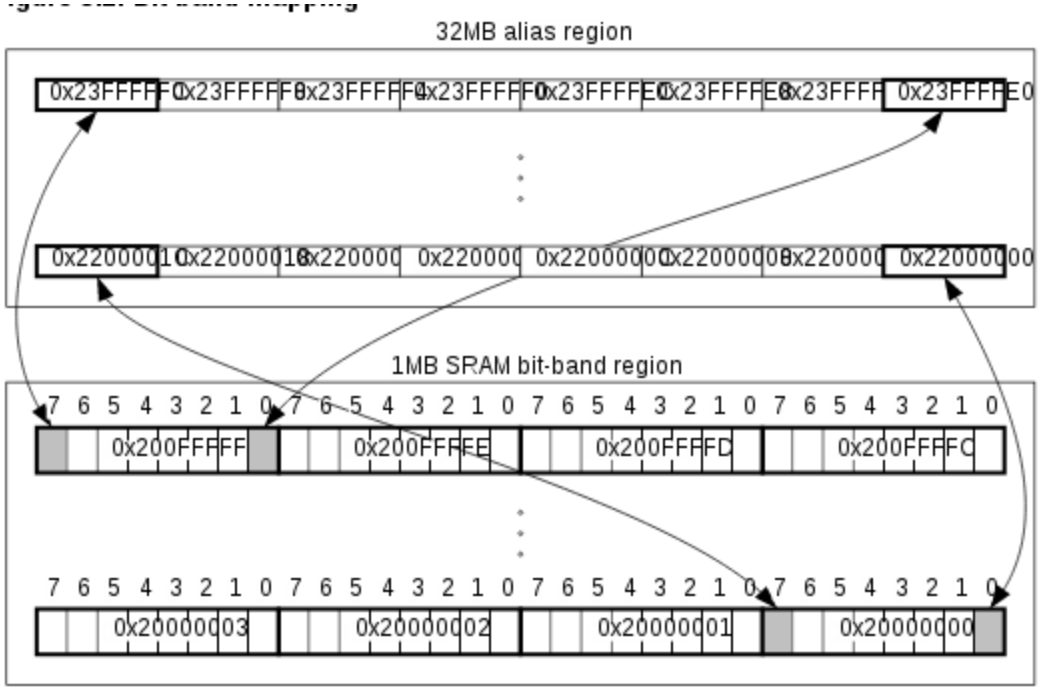
有一個mapping公式可以將alias轉換成對應的bit-band位置

- bit_word_offset = (byte_offset x 32) + (bit_number × 4)
- bit_word_addr = bit_band_base + bit_word_offset
- bit_word_offset是target bit在bit-band區域中的位置
- bit_word_addr是target bit在alias中對應的地址

- bit_band_base是alias區域的起始位置
- byte_offset是target bit在bit-band區域中的第幾個byte
- bit_number是target bit的bit位置，從0到7

下面是範例

- The alias word at 0x23FFFFE0 maps to bit [0] of the bit-band byte at 0x200FFFFF: 0x23FFFFE0 = 0x22000000 + (0xFFFFF*32) + 0*4.
- The alias word at 0x23FFFFFC maps to bit [7] of the bit-band byte at 0x200FFFFF: 0x23FFFFFC = 0x22000000 + (0xFFFFF*32) + 7*4.
- The alias word at 0x22000000 maps to bit [0] of the bit-band byte at 0x20000000: 0x22000000 = 0x22000000 + (0*32) + 0*4.
- The alias word at 0x2200001C maps to bit [7] of the bit-band byte at 0x20000000: 0x2200001C = 0x22000000 + (0*32) + 7*4.
- bit-band[0x20000000] <-> alias[0x22000000~0x2200001C](8格)
- bit-band 0x20000000[0]-0x20000000[1]-0x20000000[2]-0x20000000[3]-0x20000000[4]
- alias 0x22000000 -0x20000004 -0x20000008 -0x2000000C -0x20000010



直接存取alias

直接寫一個word到alias上與target bit的read-modify-write動作有同樣效果，Bit[0]代表要寫入target bit的值，Bit[31:1]沒有用處，所以寫入`0x01`跟`0xFF`是一樣的，都會寫入1到target bit；寫入`0x00`跟`0x0E`是一樣的，都會寫入0到target bit。

從alias讀取一個word會得到`0x01`或是`0x00`，Bit[31:1]會為0

直接存取bit-band區域

可以透過一般的read、write指令存取bit-band區域

> *Verify this*
> *http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0439b/Behcjiic.html*
> *Do you mean access bit-band region directly? I got the statement "You can also access the base bit-band region itself in the same way as normal memory, using word, half word, and byte accesses." from http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dai0179b/CHDJHIDF.html*

F9-kernel(Bitmap)

Bit-band bitmap被放在AHB SRAM中，使用BitBang地址存取bit，使用bitmap cursor(type bitmap_cusor_t)iterate bitmap。

```
// include/lib/bitmap.h
// 宣告一塊bitmap
#define DECLARE_BITMAP(name, size) \
    static __BITMAP uint32_t name [ALIGNED(size, BITMAP_ALIGN)];


// ADDR_BITBAND指的是target bit所在byte對應到的align，還沒加上
```

bit_number

- // ((ptr_t) addr) & 0xFFFFF) 可以抓出addr在bit-band區域中的第幾個byte
- #define BITBAND_ADDR_SHIFT 5
- #define ADDR_BITBAND(addr) \
- (bitmap_cursor_t) (0x22000000 + \
- ((((ptr_t) addr) & 0xFFFFF) << BITBAND_ADDR_SHIFT))
- #define BIT_SHIFT 2
-
-
- // bitmap_cursor是加上bit_number後的值，也就是target bit正確的align
- #define bitmap_cursor(bitmap, bit) \
- ((ADDR_BITBAND(bitmap) + (bit << BIT_SHIFT)))
-
- // bitmap_cursor_id可以取得bit_number
- // ((1 << (BITBAND_ADDR_SHIFT + BIT_SHIFT)) - 1) 取得 0b1111111 也就是七位的mask，與cursor進行完AND操作並右移兩位後，會留下兩位的byte_offset以及bit_number，也就是BBXXX(B:byte_offset、X:bit_number)

  這邊不知道為什麼要取這樣的值當作ID
- #define bitmap_cursor_id(cursor) \
- (((ptr_t) cursor & ((1 << (BITBAND_ADDR_SHIFT + BIT_SHIFT)) - 1)) >> BIT_SHIFT)
-
- // bitmap_cursor_goto_next 可以把cursor往前推一格(+= 4)
- #define bitmap_cursor_goto_next(cursor) \
- cursor += 1 << BIT_SHIFT
-

```
// for_each_in_bitmap 可以從某一個bitmap的start開始訪問完一塊bitmap
#define for_each_in_bitmap(cursor, bitmap, size, start) \
    for (cursor = bitmap_cursor(bitmap, start); \
            bitmap_cursor_id(cursor) < size;       \
            bitmap_cursor_goto_next(cursor))
```

這邊看到用bitmap_cursor_id跟size做比較

- bitmap_set_bit(bitmap_cursor_t cursor) - 將cursor設為1
- bitmap_clear_bit(bitmap_cursor_t cursor) - 將cursor設為0
- bitmap_get_bit(bitmap_cursor_t cursor) - 取得cursor值
- bitmap_test_and_set_bit(bitmap_cursor_t cursor) - 測試cursor是否被使用並設為1

這邊應該是用到了exclusive access，避免同一個位置被不同的thread配置

Refer from

- http://en.wikipedia.org/wiki/Bit_array
- ARM Information Center(2.5. Bit-banding)

- **ktable**

```
// include/lib/ktable.h
struct {
    char *tname;
    bitmap_ptr_t bitmap;
    ptr_t data;
    size_t num;
    size_t size;
};
```

問一個笨問題，這邊為什麼要bitmap?

我現在在查bitmap的用法，目前是看到這是一種在處理bit-level上很有效率的結構。

ktable的構造應該是這樣：tname是table的名字、bitmap是用來紀錄data內區塊的使用情況(已配置、空閒)、data是實際存放資料的地方，會被切成num個區塊，每個區塊有size大小。

取得跟釋放一個區塊都是透過ktable提供的API，alloc跟free，他們都會先進去查bitmap，然後設定bitmap，最後把data區塊中的位置傳回來。(以上是我的推論XD)

了解！Thanks!

"Fast object pool management" from pg31 of Viller Hsiao's slide,透过ktable加速 allocate和 free。

```
// 宣告一個ktable
// $ arm-none-eabi-readelf f9.elf -s | grep fpage_table
// 263: 10000000    32 OBJECT  LOCAL  DEFAULT    8 kt_fpage_table_bitmap
// 265: 2000c4e0  6144 OBJECT  LOCAL  DEFAULT    4 kt_fpage_table_data
// 918: 20000640    20 OBJECT  GLOBAL DEFAULT    3 fpage_table
#define DECLARE_KTABLE(type, name, num_)                \
    DECLARE_BITMAP(kt_ ## name ## _bitmap, num_);           \
    static __KTABLE type kt_ ## name ## _data[num_];    \
    ktable_t name = {                                   \
                .tname = #name,                         \
                .bitmap = kt_ ## name ## _bitmap,       \
                .data = (ptr_t) kt_ ## name ## _data,   \
                .num = num_, .size = sizeof(type)       \
    }

```

```
/* 提供的API */
// 將kt中的bitmap全部設為0
void ktable_init(ktable_t *kt);
// 檢查第i個元素是否已經被配置
int ktable_is_allocated(ktable_t *kt, int i);
// 配置第i個元素，回傳元素的位置
void *ktable_alloc_id(ktable_t *kt, int i);
// 配置到第一個free的元素，回傳元素的位置
void *ktable_alloc(ktable_t *kt);
// 釋放元素
void  ktable_free(ktable_t *kt, void *element);
// 取得該元素位在ktable內的id
uint32_t ktable_getid(ktable_t *kt, void *element);
```

- **ktimer**

```
// kernel/systhread.c
static void idle_thread(void) {
    while(1) ktimer_enter_tickless();
}


// kernel/ktimer.c
// 宣告一個 ktimer_event_table 的 ktable
DECLARE_KTABLE(ktimer_event_t, ktimer_event_table,
CONFIG_MAX_KT_EVENTS);
```

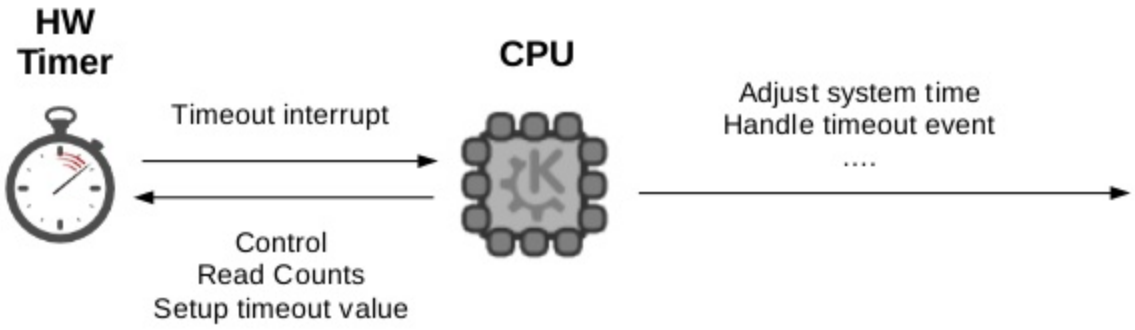
```
// 全域變數，存了下一個要執行的event queue
ktimer_event_t *event_queue = NULL;


// include/ktimer.h
typedef struct ktimer_event {
    struct ktimer_event *next;
    ktimer_event_handler_t handler;


    uint32_t delta;
    void *data;
} ktimer_event_t;


// 初始化ktimer，設定systick reload value，設定 ktimer的handler
void ktimer_event_init(void);
//
int ktimer_event_schedule(uint32_t ticks, ktimer_event_t *kte);
// 新增一個event，並且呼叫ktimer_event_schedule進行排程
int ktimer_event_create(uint32_t ticks, ktimer_event_handler_t
handler, void *data);
// 會將最後一個還不需要開始執行的event放到event_queue，接著走訪所有要被
執行的event，並判斷是否需要re-scheduling，最後如果還有event在排程，則
enable ktimer
```

- `void ktimer_event_handler(void);`

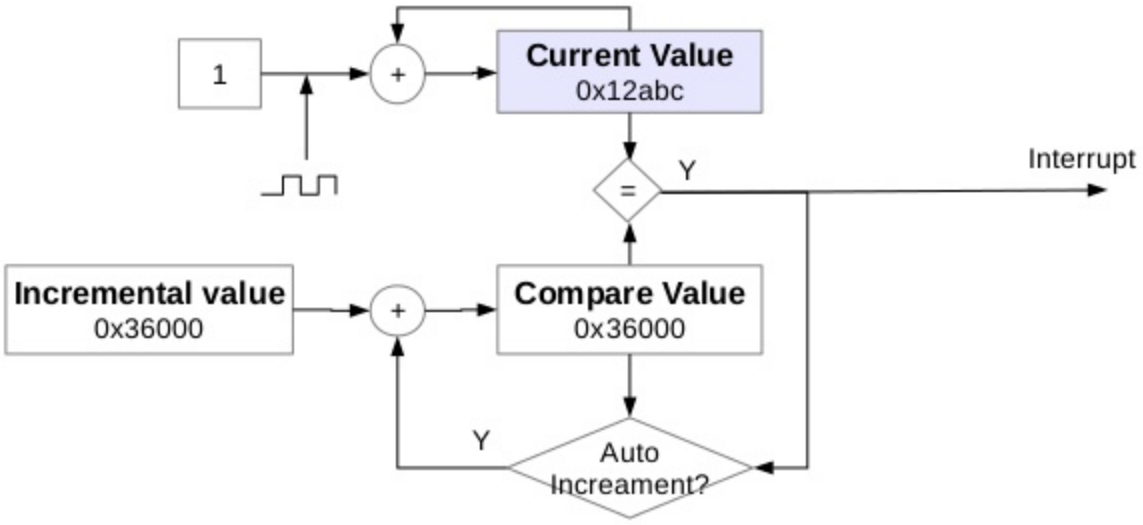
- **Timers**

system time represents a computer system's notion of the passing of time. system time is measured by a *system clock*, which is typically implemented as a simple count of the number of *ticks*that have transpired since some arbitrary starting date, called the *epoch*.

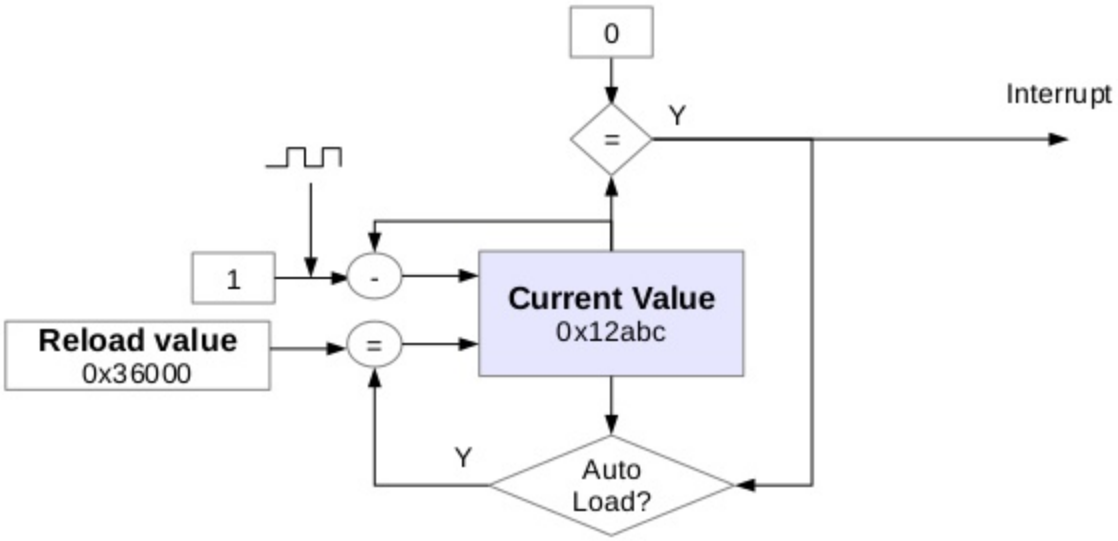
Tick的實作方法



- Incremental timer
  - Example：Cortex-A9 MP Global Timer
  - 每一個周期會將counter + 1直到設定的大小，接著產生interrupt

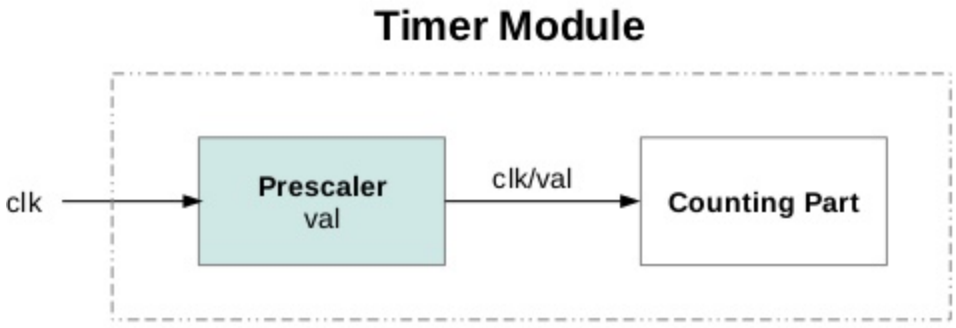


- Counting down timer
  - Example：Cortex-M4 SysTick
  - 先將值存入counter，每一個週期會將counter - 1，到0時產生interrupt

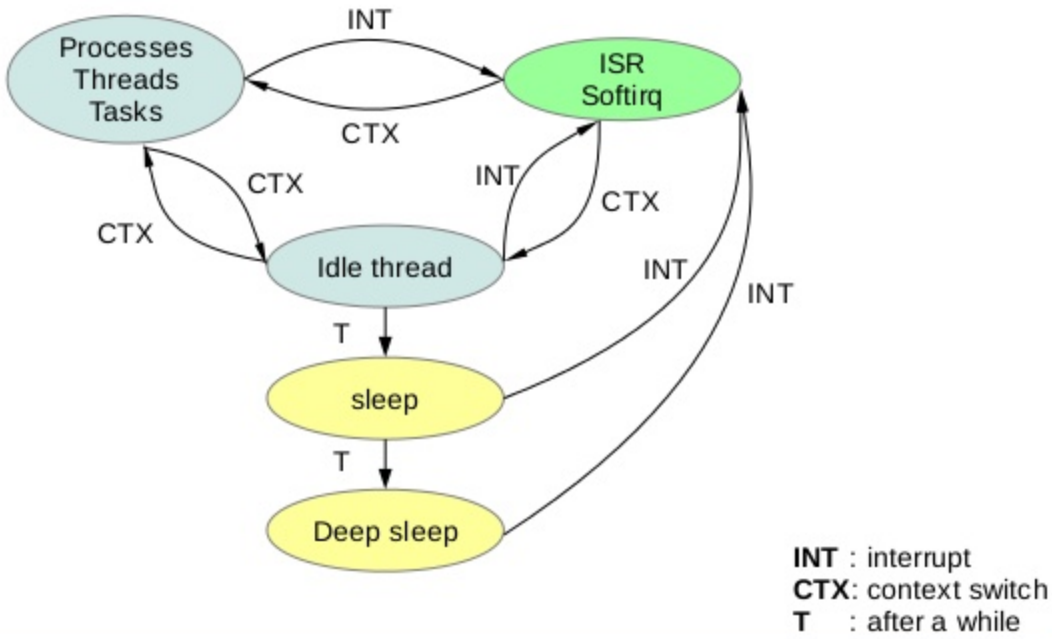


- Prescaler(除頻器)
  - 利用整數除法將高頻率的電子訊號轉換成低頻率的電子訊號，例如將CPU的clock頻率在經過除頻器後在餵給timer
  - 除頻器的用途在於將clock的頻率調整到你想要的，在8-bit或是16-bit的timer上常常會需要在解析度(高解析度就要有高頻率的clock)還有範圍(高頻率的clock會讓timer常常overflow)間調整。例如在16-bit的timer上

無法以1us的解析度配上1sec的最大區間



- Timeout ISR
  - 增加System ticks
  - 執行timeout事件的handler
  - 重新排程



- **GPIO**

第一版本的 GPIO API PR#86

- based on feedback from **georgekang**'s opinion should move driver to user-space.

- but I encountered issue that memory fault while accessing RCC_AHB1ENR
  - I submit a issue f9micro/f9 kernel#87 with detail debug log.
    Currently, I'm stuck here. Any idea please let me know, thanks.

    *feedback from **georgekang** ? Does your user space get the authority of AHB1_1DEV?*

Feature: support gpio output pin for user app

File: user/include/gpioer.h

/* action */

#define GPIO_HIGH 1

#define GPIO_LOW 0

void gpioer_config_output(uint8_t port, uint8_t pin, uint8_t pupd, uint8_t speed);

void gpioer_out(uint8_t port, uint8_t pin, uint8_t action);

Hook 方式仿照 THREAD_LOG for printf 方式，實作了 THREAD_GPIOER。

example: builtin 4 led control

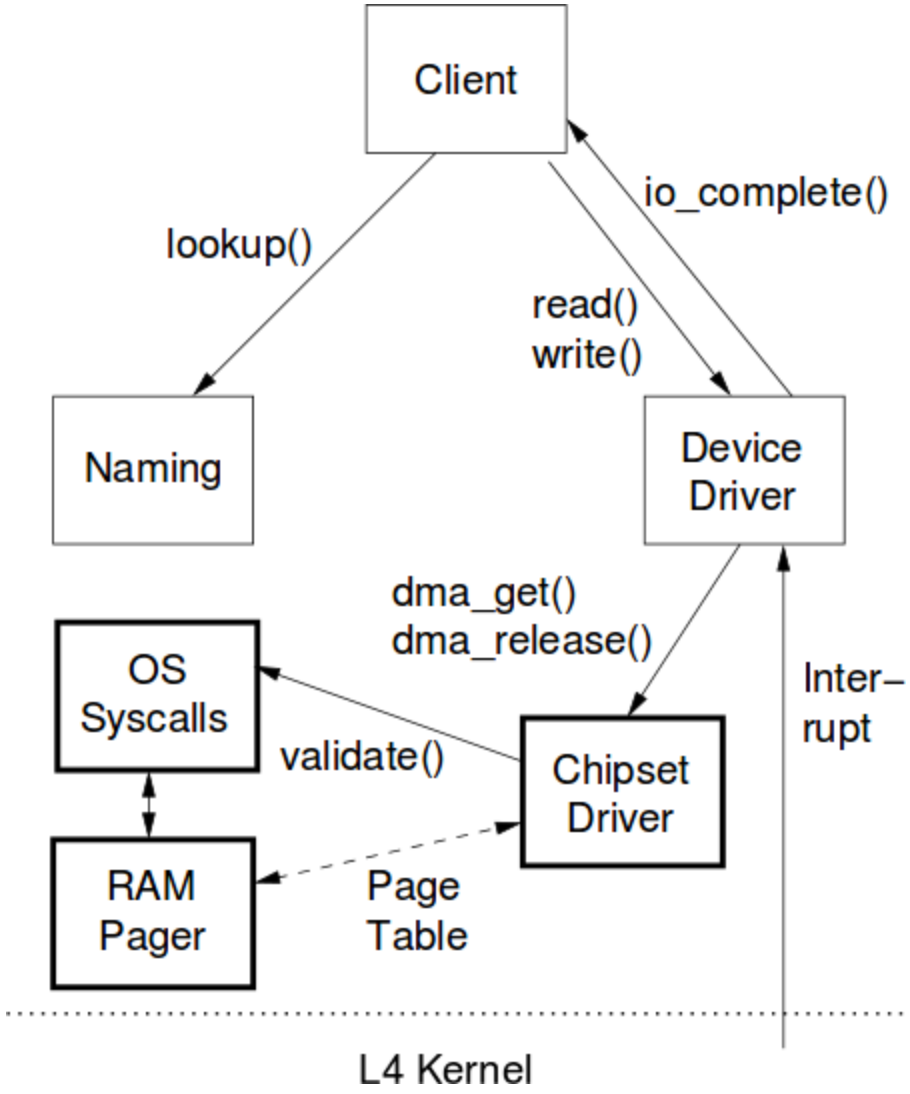
user/apps/gpioer

presentation:

http://www.slideshare.net/benuxwei/f9-microkernel-app-development-part-2-gpio-meets-led

第二版：符合 microkernel 規範的 driver model

參考: http://www.cse.unsw.edu.au/~cs9242/03/lectures/lect13a.pdf



device driver 其實就是 user task ，並且接收 interrupt IPC 並回應

> *我目前在將 GPIO Driver移到 user space, 遇到 dma_get() 的問題，請問 DMA_get() 要如何實作，才能讓 GPIO driver 直接存取 GPIO 相關的記憶位置呢？ issue: f9micro/f9 kernel#87 I don't know how to make a thread with read/write authority of AHB1_1DEV*

- **工具**

cscope

一套用來瀏覽source code的開發者工具

1. 安裝`cscope` - sudo apt-get install cscope
1. 下載cscope_maps.vim，放到`$HOME/.vim/plugin`
1. 到一個有C code的資料夾執行`cscope -R`，接著會進入cscope的GUI畫面，可以有一些操作
    1. 方向鍵 - 移動游標
    1. tab - 切換搜索結果與搜尋類別
    1. 數字鍵 - 打開搜尋結果文件
    1. CTRL-D 離開cscope
1. 啟動vim，可以利用`vim -t main`指定C symbol
1. 將游標指到一個symbol，輸入`CTRL-\` + `s`，接著可以看到搜尋結果出現在下方，輸入編號或是鍵入enter都能進入搜尋結果，如果要跳回剛剛的位置只要輸入`CTRL-t`
1. 將游標指到一個symbol，輸入`CTRL-spacebar` + `s`，會出現跟剛剛一樣的結果，不過這次進入搜尋結果會開啟一個水平分頁
1. 將游標指到一個symbol，輸入`CTRL-spacebar-spacebar` + `s`，會出現跟剛剛一樣的結果，不過這次進入搜尋結果會開啟一個垂直分頁
1. 也可以直接輸入command搜尋，`:cscope find symbol foo`
1. 剛剛的`find symbol X`會找到所有用到symbol X的地方
    1. `find g X` - 找到X的全域定義
    1. `find c X` - 找到呼叫X的地方
    1. `find f X` - 開啟檔案X(標頭檔)
1. `cscope -b`，重新生成資料庫，不開啟GUI

    跟大家分享一些常用的軟體:

    1. 把vim打造成像source insight一樣:

    http://stenlyho.blogspot.tw/2010/03/vim-trinity-source-explorer-tag-list.html

    2. 比對軟體: meld, vimdiff

    3. git相關: tig

- **問題**
    - 什麼是tickless？
        - 讓ARM Cortex-M只有在需要的時候喚醒(排程的時間、interrupt事件)

，

- **比起使用SysTick保持高頻率運轉的clock，更能降低耗能**

  原本使用systick_handler來中斷task，讓kernel可以執行scheduling。現在讓硬體sleep,然後在scheduled time or interrupt happened的時候起來。可以減少power consumption。參考投影片29頁

  是我下面貼的那份投影片的p29嗎？

  jserv那份，我加上去了

- FPB - Page8~Page9 (MPU pg8)

- **暫存紀錄**

  - UTCB IPC - chinese-mk-2013(page. 26)

- **參考資料**

http://www.slideshare.net/benuxwei/f9-microkernel-app-development-part-1

http://www.slideshare.net/vh21/2014-0109f9kernelktimer

https://github.com/f9micro/f9-kernel/blob/master/README.md

http://www.slideshare.net/jserv/f9-microkernel

http://www.cse.unsw.edu.au/~cs9242/05/lectures/02-l4.pdf

- JuluOSDev 的 hackpad: https://juluos.hackpad.com/
  - Lab: https://juluos.hackpad.com/F9-kernel-Labs-skF7vQyB7KZ

Memory Management from UNIX v6, BSD, MINIX, to L4

http://www.slideshare.net/jserv/l4-microkernel-design-overview

http://www.l4ka.org/l4ka/l4-x2-r7.pdf

在晶心平台實作 ROM patch