實作輕量級 RTOS 網路堆疊

Apr 24, 2009

Jim Huang(黃敬群 /**jserv**)
Email: <**jserv.tw@gmail.com**>
Blog: **http://blog.linux.org.tw/jserv/**

自我介紹

Phone/PDA, GPS, Mobile TV/Digital TV 代工設計

參與自由軟體開發 / 社群組織

- 新酷音輸入法
- Kaffe.org
- GNU Classpath
- FreeDesktop
- OpenMoko
- ...
- Xenomai
- Orz Microkernel
- pcmanx
- LXDE
- OpenAVS
- ...
- Debian Taiwan
- KDE Taiwan
- TOSSUG
- 0xlab (4/27)

警告：本議程僅探討實務面，忽略理論部份

提綱

動機

Bell 定律：每十年運算型態的移轉

RTOS 與嵌入式裝置的角色

實現 TCP/IP 的難題

回歸現實：剪裁與調適

技術討論

動機

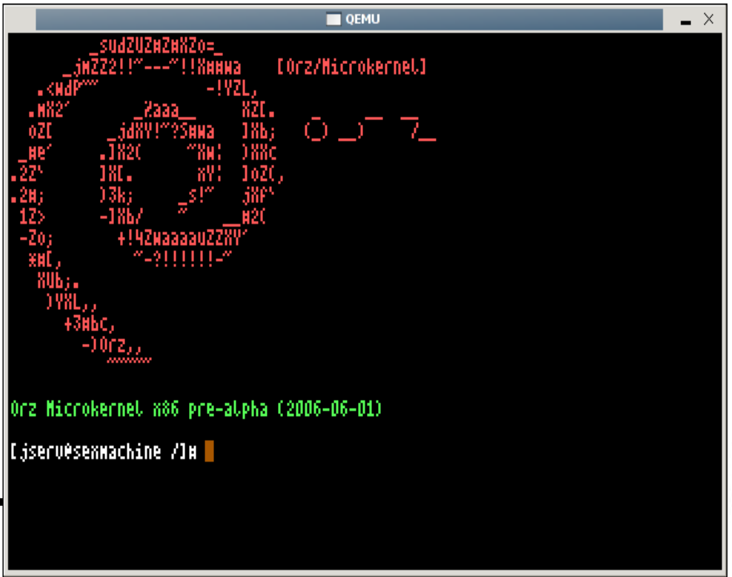
每年寫一套作業系統當作業（對不起，遲交）

- JK (2001)
- Orz Microkernel (2006)
- RT nanokernel (2007)
- Jamei RTOS (2007)
- CuRT (2009)
- ??? (2009)

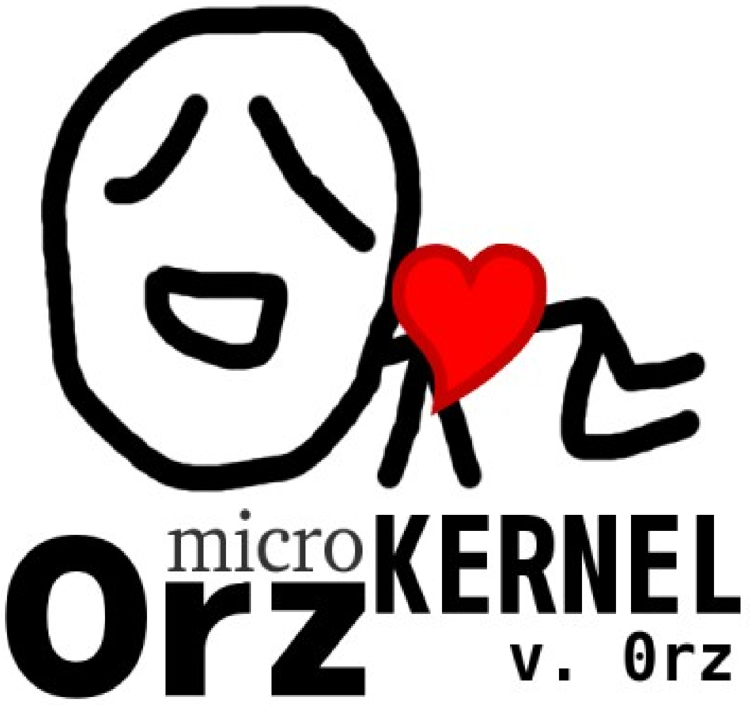
動機 (1)

Everything can be Orz.

Orz Microkernel



```
         _sudZUZ#Z#XZo=_
      _jmZZ2!!~---~!!X##wa    [Orz/Microkernel]
   .<wdP~~               -!YZL,
  .mX2'        _%aaa__       XZ[.
  oZ[       _jdXY!~?S#wa    ]Xb;   (_) __)‾ ‾7_
 _#e'      .]X2(     ~Xw|   )XXc
.2Z`       ]X[.       xY|   ]oZ(,
.2#;       )3k;     _s!~    jXf`
 1Z>       -]Xb/    ~    __#2(
 -Zo;        +!4ZwaaaauZZXY'
  *#[,         ~-?!!!!!!-~
   XUb;.
    )YXL,,
      +3#bc,
        -)Orz,,

Orz Microkernel x86 pre-alpha (2006-06-01)

[jserv@sexmachine /]# 
```



動機 (2)

Orz Microkernel 的啓發

- 學習作業系統與相關的系統程式該如何設計
- 建立自信：原來一個作業系統只需幾 kb 的空間就實做出來

動機 (3)

設計作業系統也可很有趣

以實體的機器人設計作爲主軸體驗如何親手打造嵌入式系統並著手設計相關軟硬體建設

動機 (4)

從零到有，設計即時作業系統

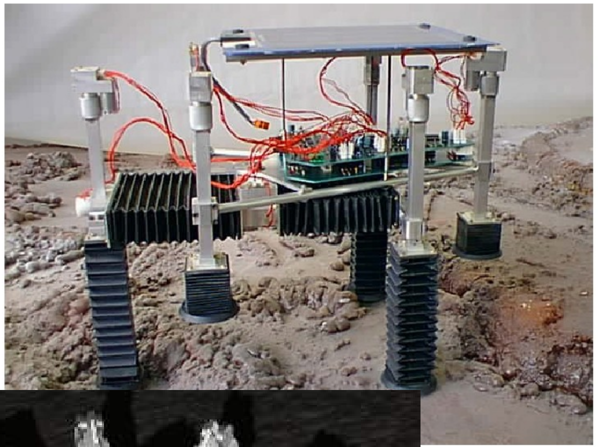
- 杜威博士：「作中學」
- RT nanokernel (OSDC.tw 2007)
- Jamei (COSCUP 2007)
- 模仿 Linux 經典設計並建構具體而微的 RTOS

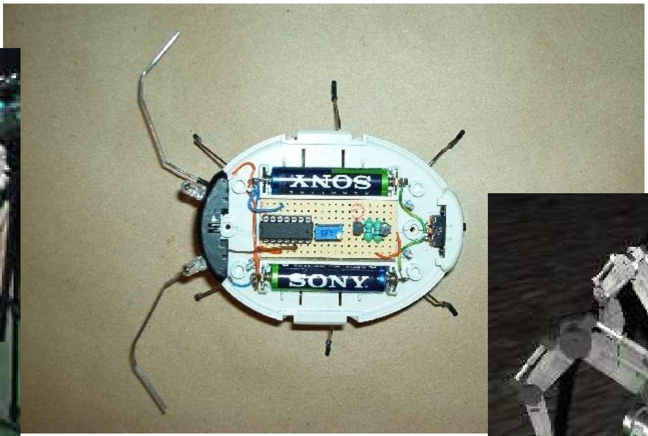
動機 (5)

- 滿足自動控制系統需求
  - 即時處理
  - 建構嵌入式環境
  - 網路通訊能力

應用型態

國防軍事
科學搜救探索
彷生物型態
機器人足球賽





Copyright 1997 Dan Caron

動機 (6)

簡化設計，適用更廣範圍的硬體

- CuRT (2009)
- 硬體： Marvell/Intel PXA255
- 特徵
  - Preemptive Multi-threading
  - Priority-base Round-Robin Scheduling
  - Thread Management
  - Semaphore Management Support
  - IPC: mailbox, message queue

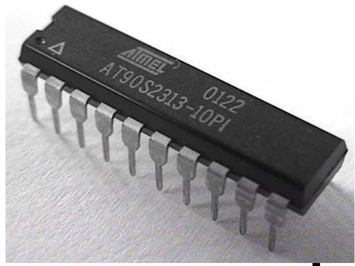
Bell 定律：每十年運算型態的移轉

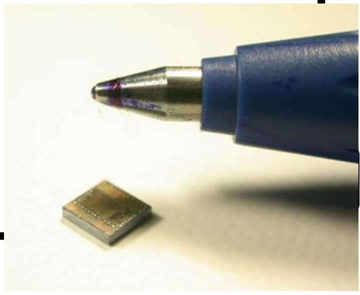
1980 年代： PC revolution

1990 年代： Internet revolution

2000 年代： embedded revolution

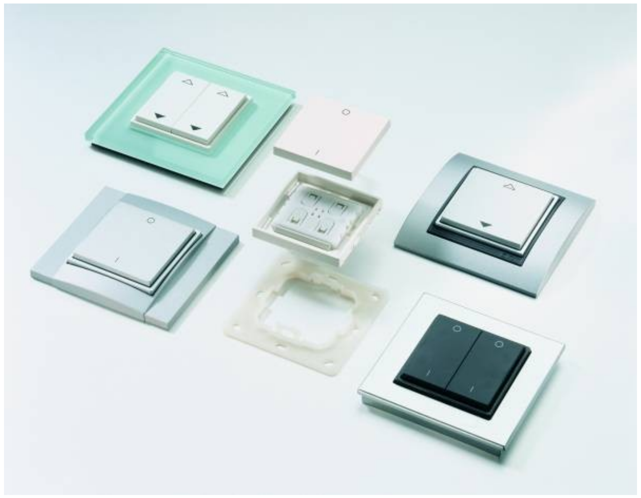
2010 年代： embedded Internet revolution



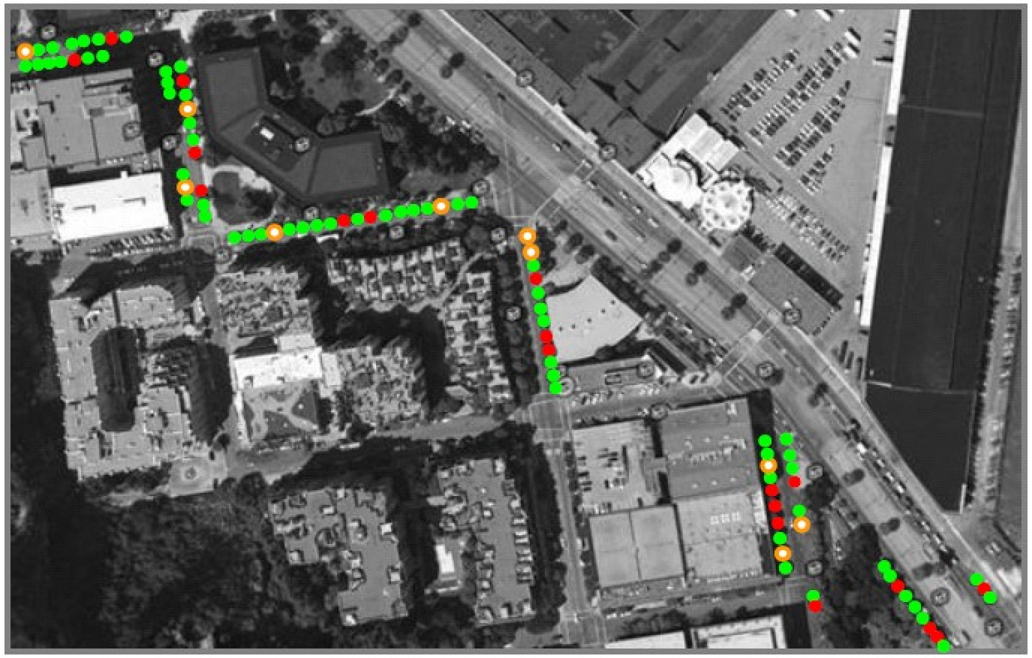


RTOS 與嵌入式裝置

enocean.com



Building automation

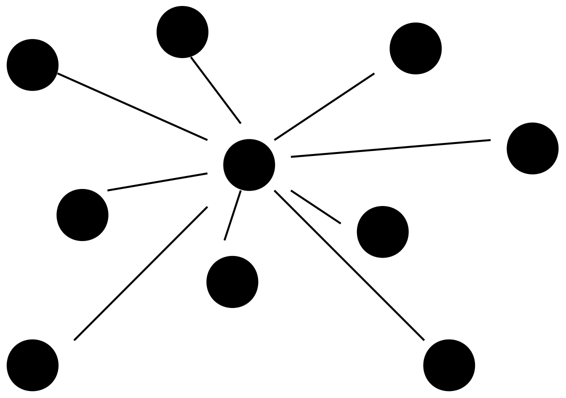


streetlinenetworks.com

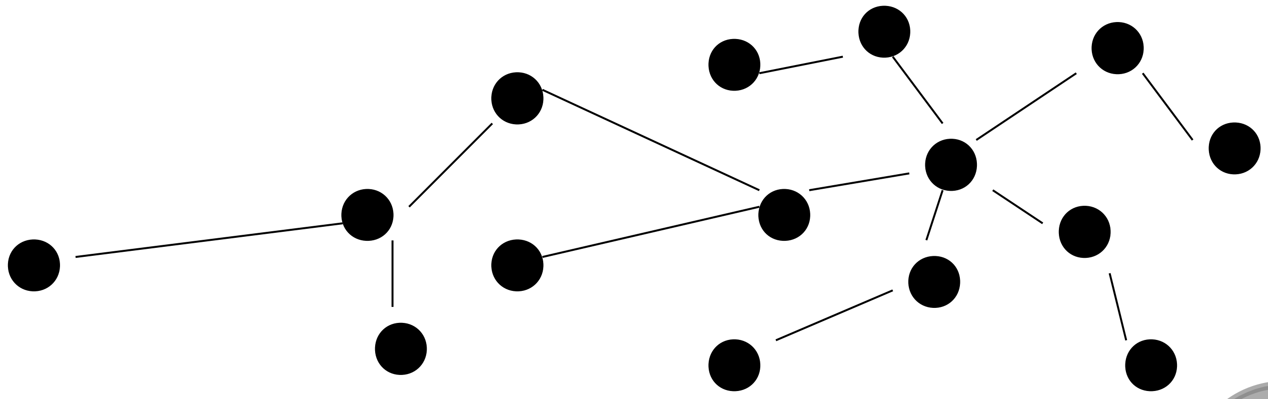
Wirelss parking management

嵌入式無線網路

Star networks

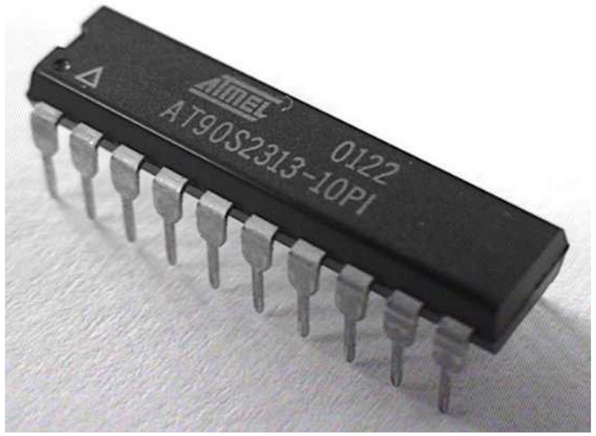


Mesh networks



典型嵌入式硬體

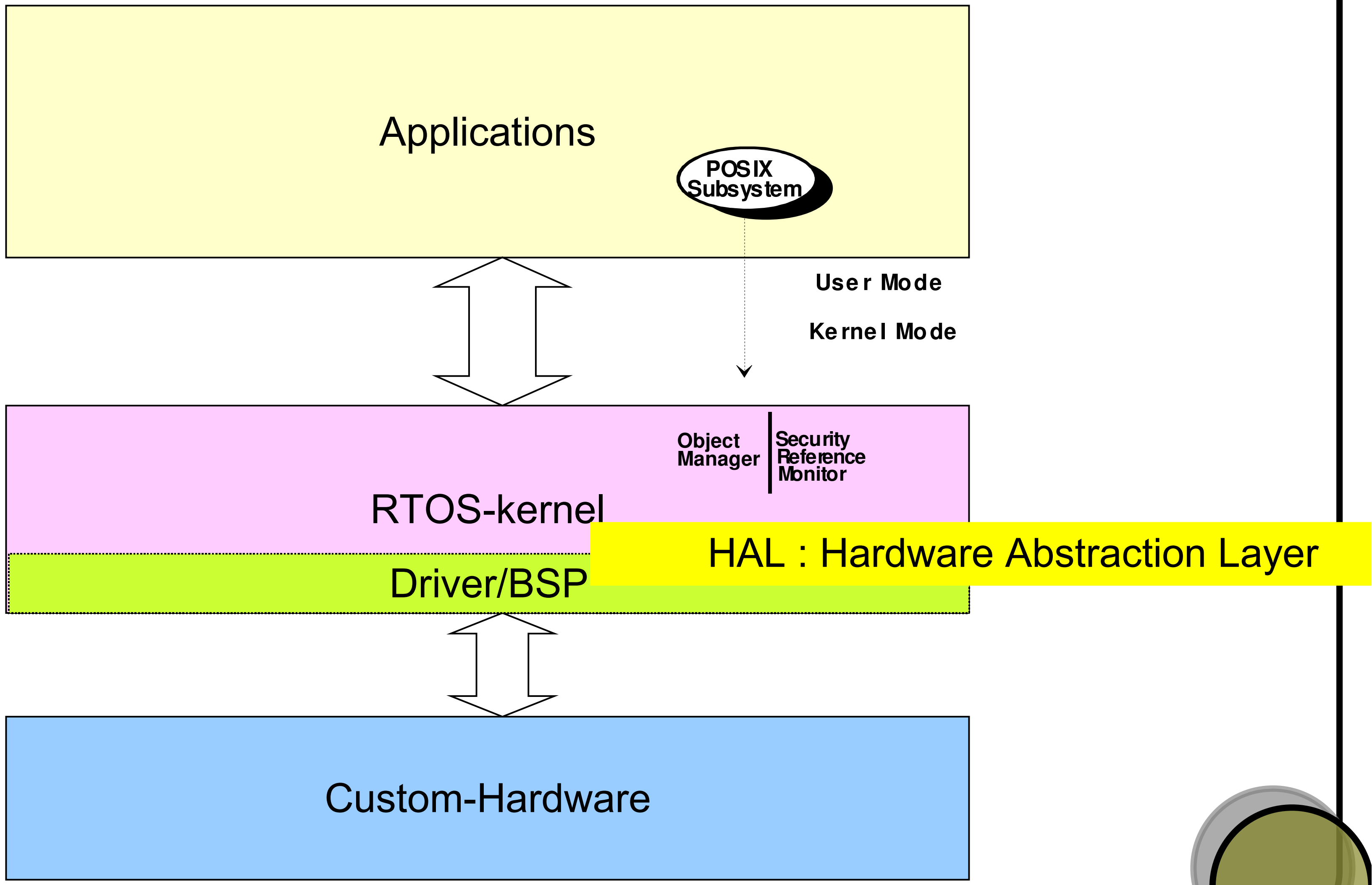
- Small microcontroller (microprocessor + I/O)
  - 8- and 16-bit processors
  - Typical memory size
    - 1 – 100 k Flash ROM for code
    - 0.1 – 10 k RAM for data
  - Typical speed
    - 1 – 10 MHz
- 8051, AVR, MSP430, Z80, 6502, ARM, ...



設計自己的 RTOS

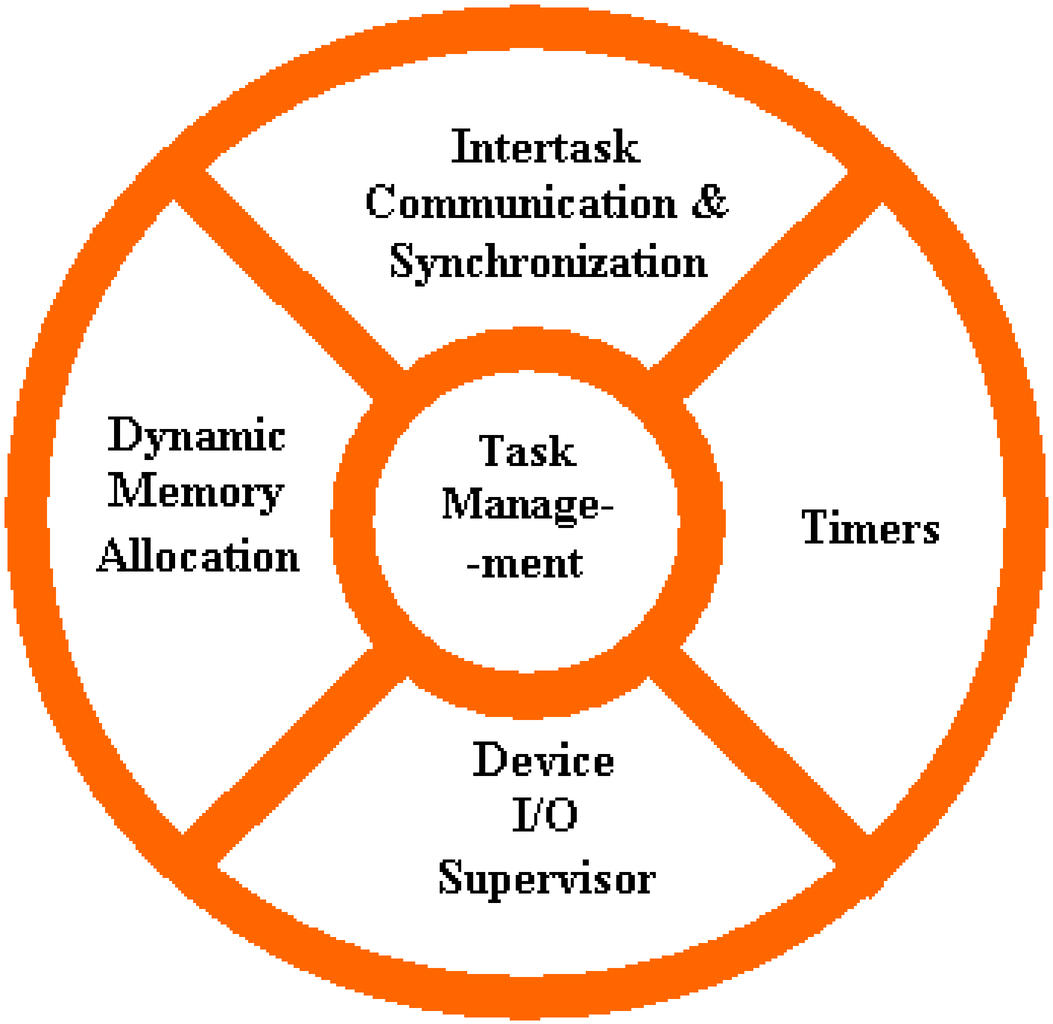
- Jamei = **J**ust **A**nother **M**icroprocessor **E**mbedded **I**nfrastructure
  - 輕巧並可調整組態
  - 仿造 Linux kernel 部份設計
    - arch(平台相依實做)
    - device driver model
    - vfs
  - 部份 POSIX Thread
  - 部份 Realtime API (IEEE 1003.1b)
  - New BSD License 與 GNU GPLv2(部份)
  - 支援 i386 與 ARM9

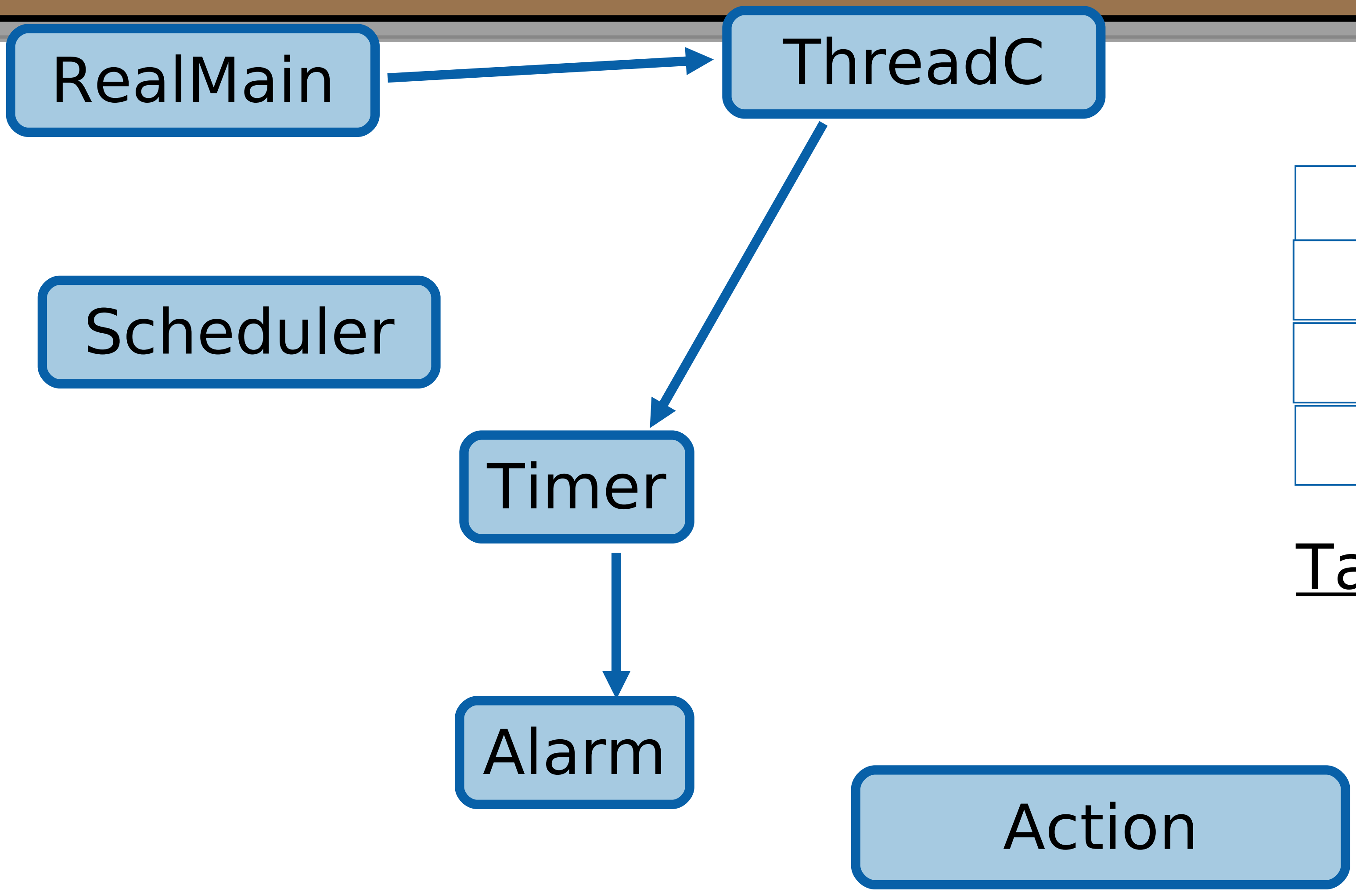
RTOS 結構



Applications
POSIX Subsystem
User Mode
Kernel Mode
Object Manager
Security Reference Monitor
RTOS-kernel
HAL : Hardware Abstraction Layer
Driver/BSP
Custom-Hardware

RTOS Kernel


Intertask Communication & Synchronization
Dynamic Memory Allocation
Task Manage- -ment
Timers
Device I/O Supervisor



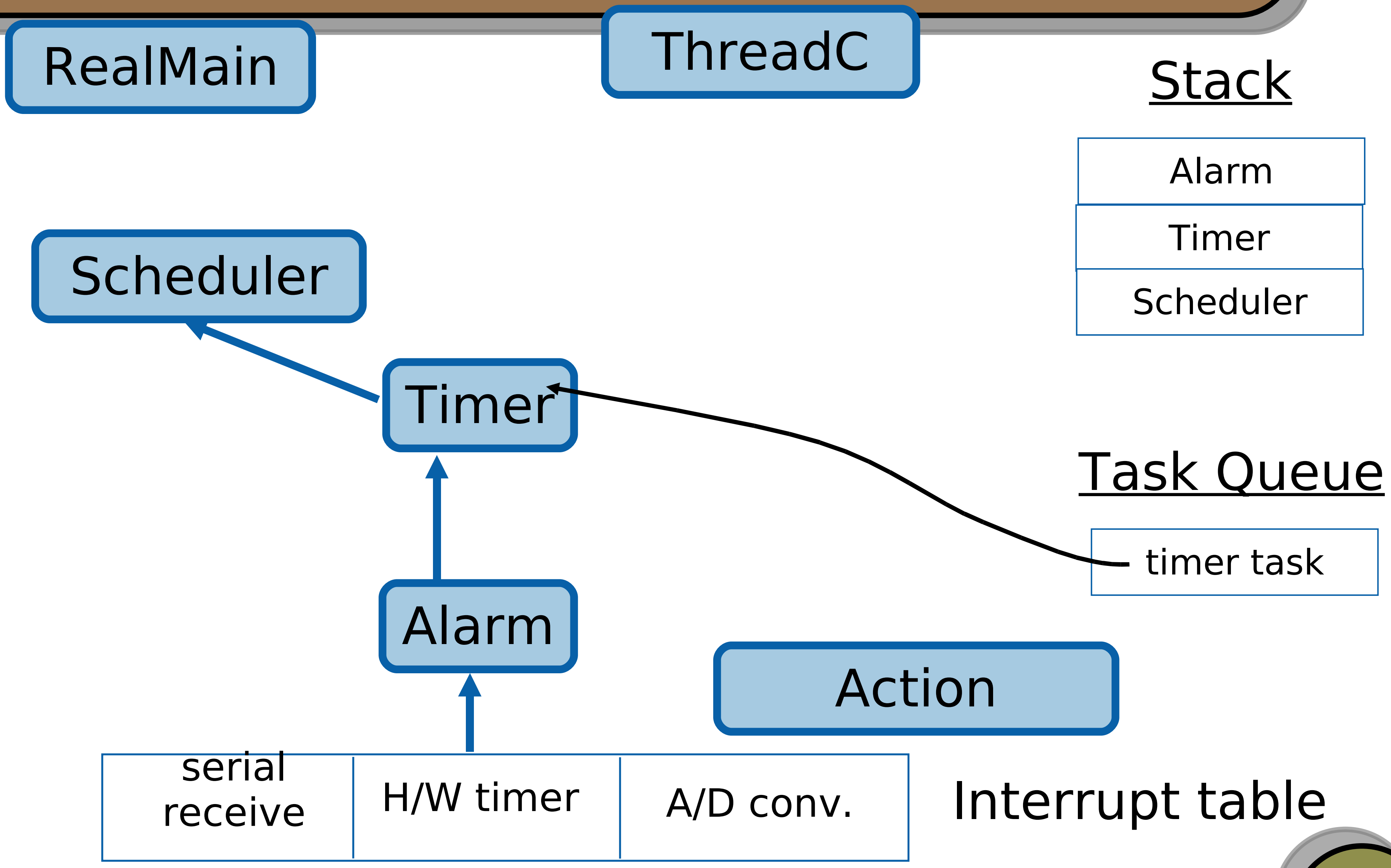
Stack

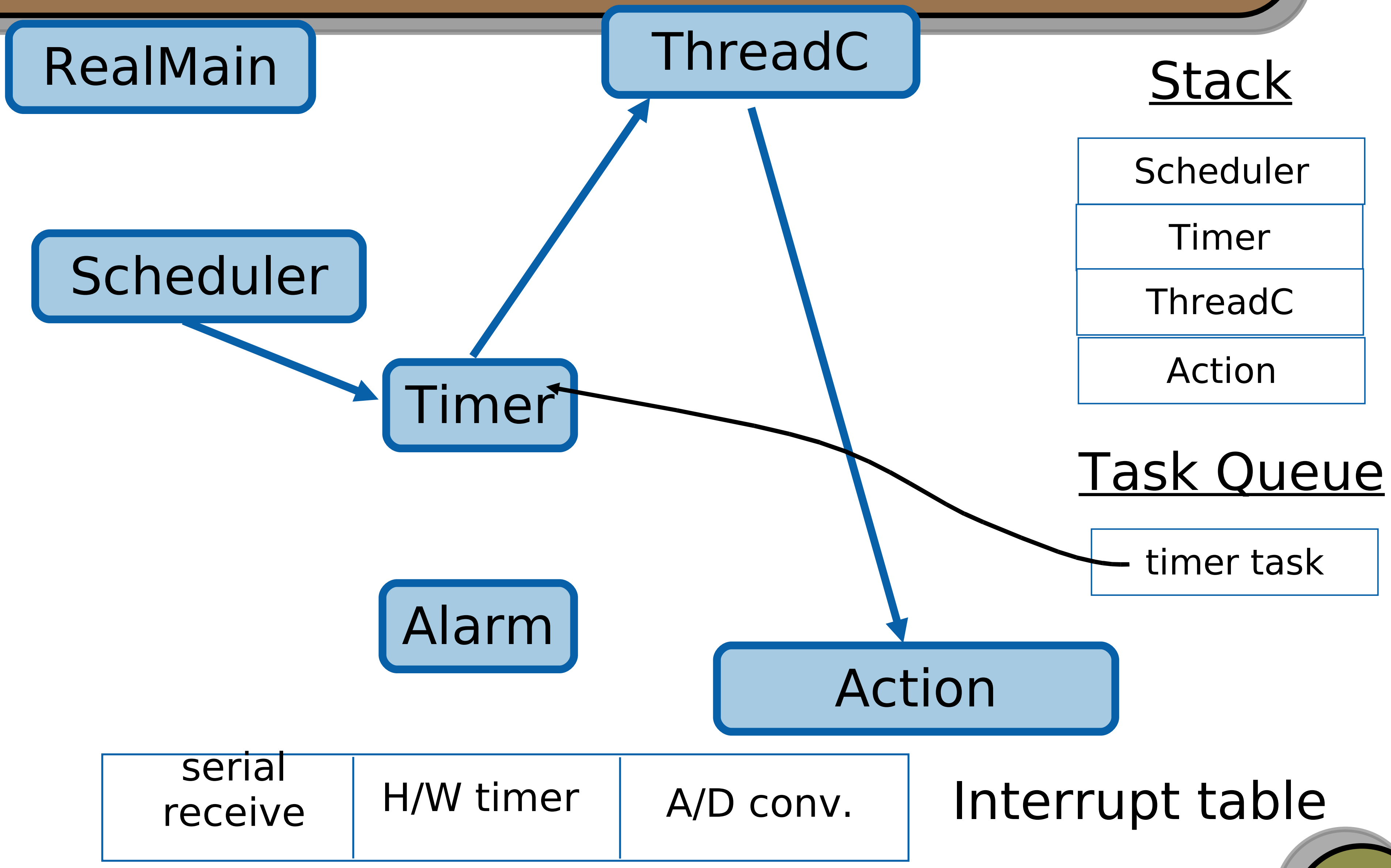
RealMain
ThreadC
Timer
Alarm

Task Queue

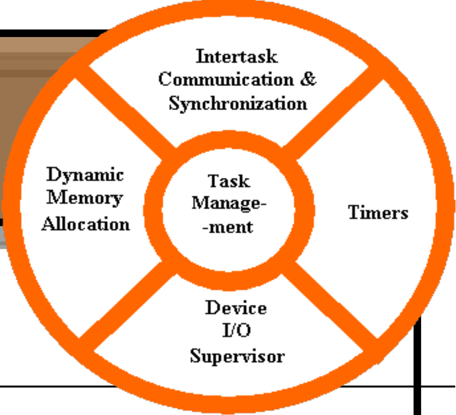
serial receive	H/W timer	A/D conv.

Interrupt table


RealMain
ThreadC
Stack
Alarm
Timer
Scheduler
Scheduler
Timer
Task Queue
timer task
Alarm
Action
serial receive
H/W timer
A/D conv.
Interrupt table


RealMain
ThreadC
Scheduler
Timer
Alarm
Action
Stack
Scheduler
Timer
ThreadC
Action
Task Queue
timer task
serial receive
H/W timer
A/D conv.
Interrupt table

RTOS Kernel::Tasks


Intertask Communication & Synchronization
Dynamic Memory Allocation
Task Manage--ment
Timers
Device I/O Supervisor

Task 是 RTOS 最重要的項目

RTOS scheduler 必須 determinstic

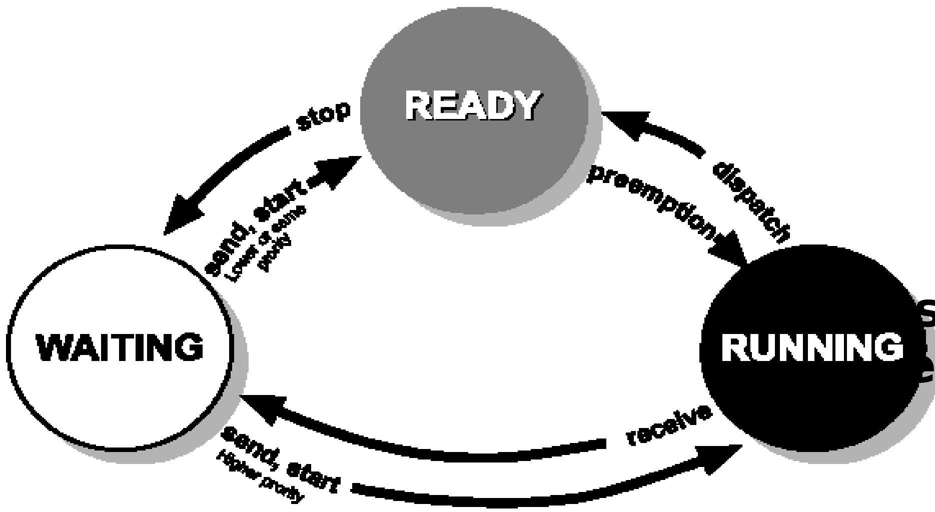
O(1) or O(n)

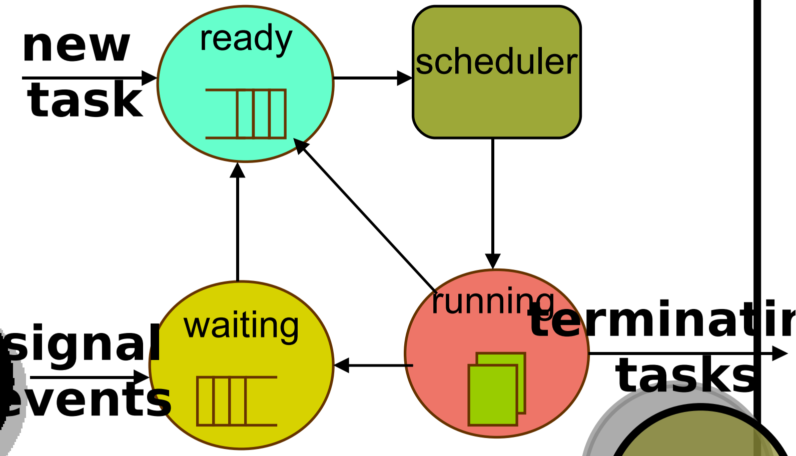
Scheduling policy

Clock driven

Priority driven (RMS & EDF)

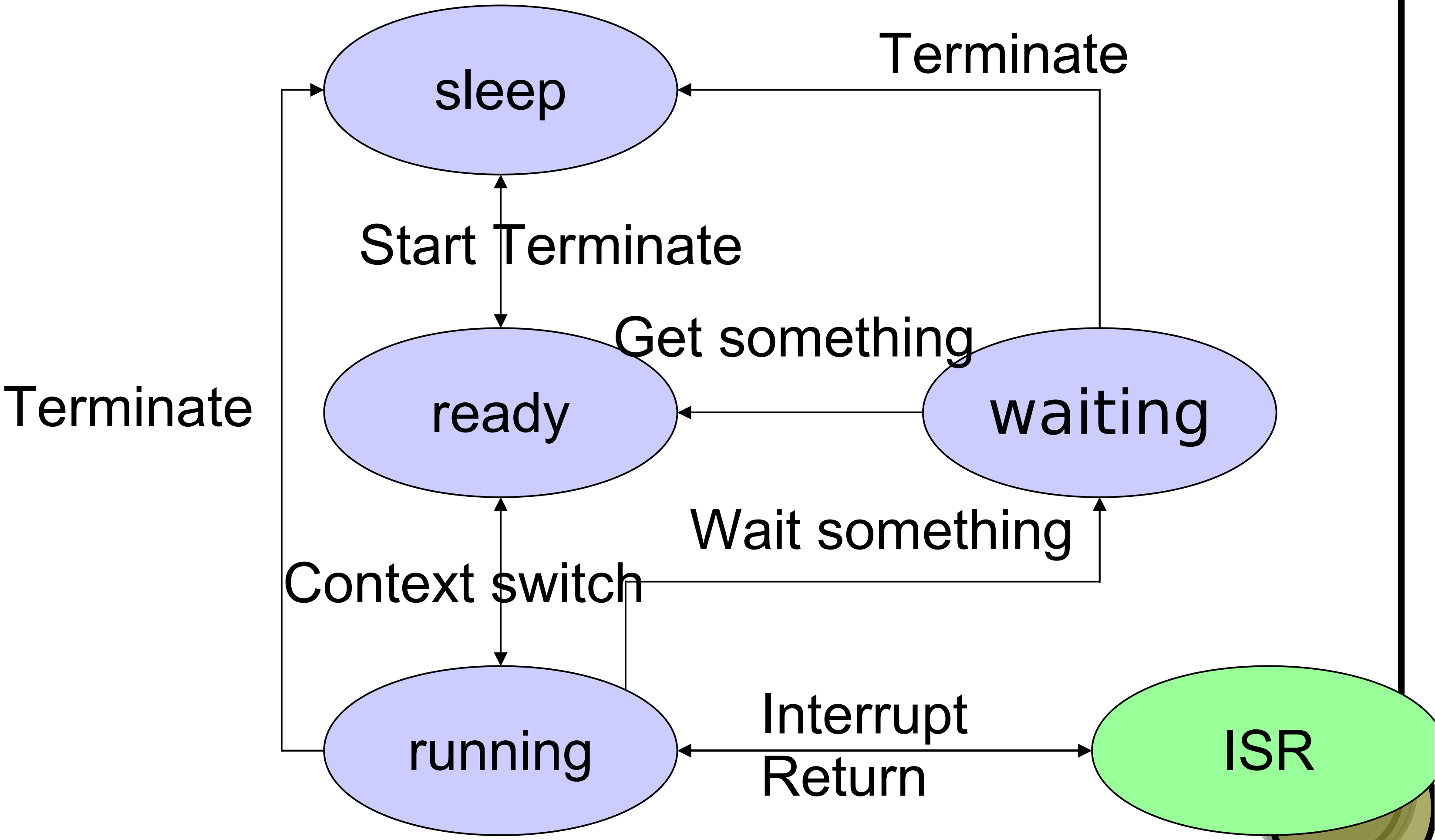
笨蛋，問題在 scheduling ！





RTOS Kernel::Tasks

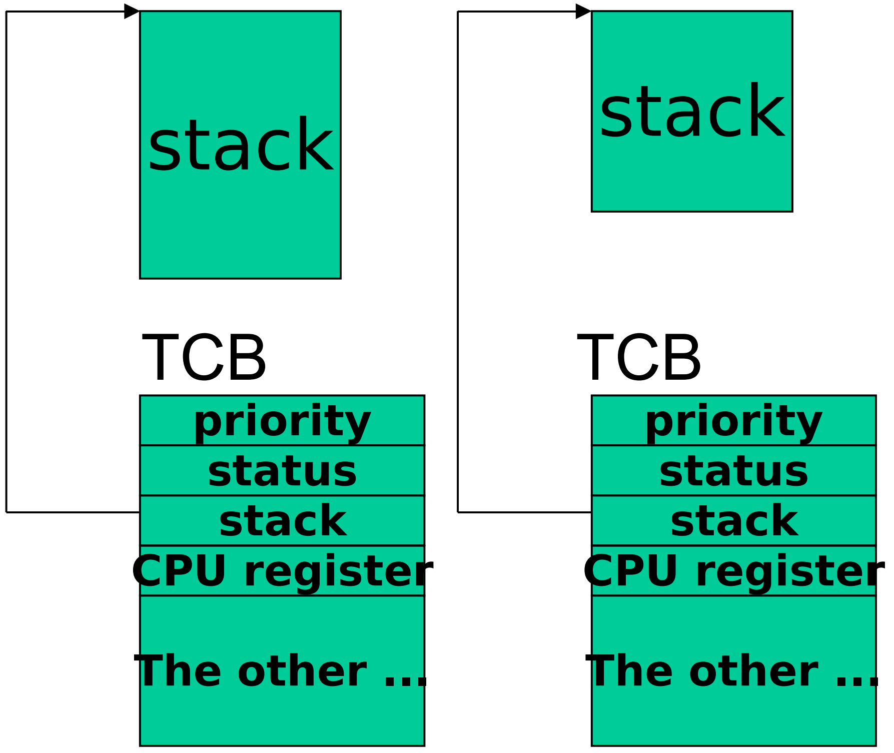
Task 的狀態移轉

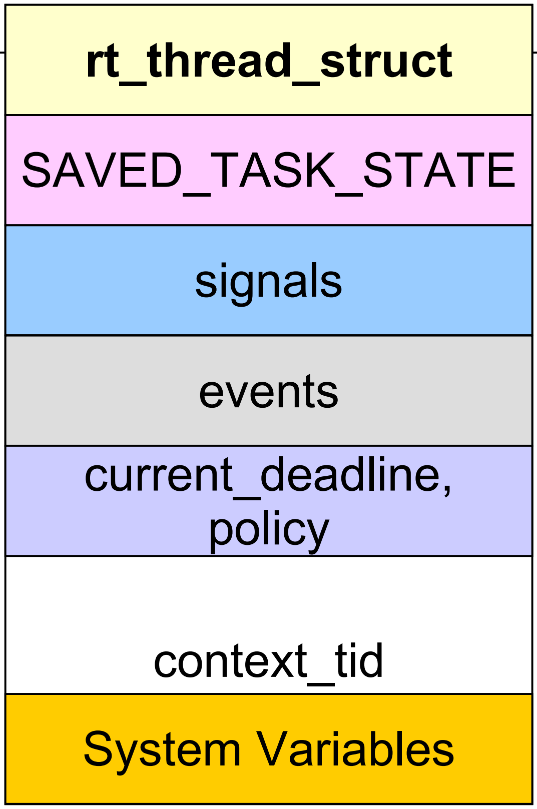


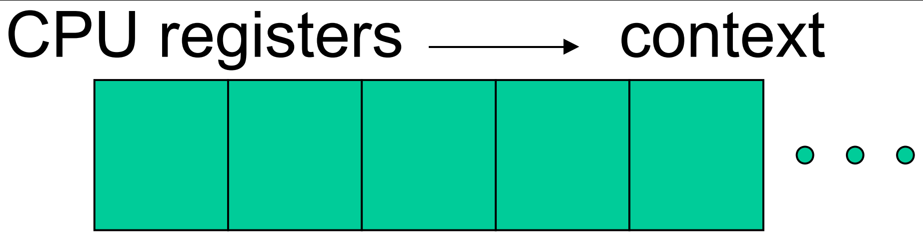
RTOS Kernel::Tasks

TCB (Task Control Block)

- Introduction
- Structure
- RTOS Kernel
- Tasks
- Memory
- Timers
- I/O
- IPCs
- Device Driver
- In an Action







RTOS Kernel::Tasks

Jamei 中 thread 是 Task 的單元

```
typedef struct rt_thread_struct *pthread_t;
pthread_create
pthread_exit
pthread_kill
pthread_wakeup_np
pthread_suspend_np
pthread_wait_np
...
```

- 試圖滿足 POSIX Thread 語意
- _np]non-POSIX

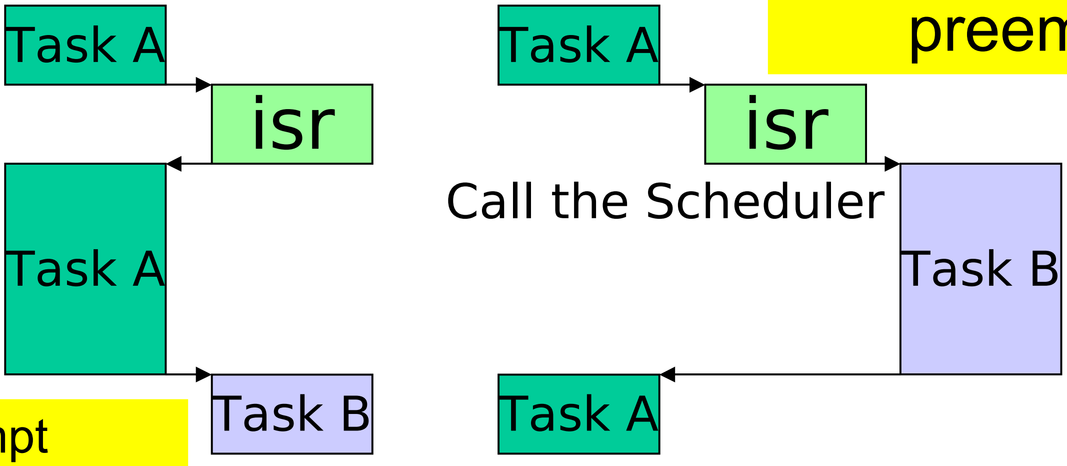
RTOS Kernel::Tasks

Jamei 中 thread 是 Task 的單元

- Introduction
- Structure
- RTOS Kernel
- Tasks
- Memory
- Timers
- I/O
- IPCs
- Device Driver
- In an Action

non-preempt

preempt



preemptive scheduling

```
int rt_schedule(void)
{
  struct rt_thread_struct *preemptor = 0;
  ...

  if ((s->clock->mode == RT_CLOCK_MODE_ONESHOT)) {
    if ((preemptor = find_preemptor(s,new_task)))

    {
      (s->clock)->settimer(s->clock, preemptor->resume_time - now);
    } else {
      (s->clock)->settimer(s->clock, (HRTICKS_PER_SEC / HZ) / 2 );
    }
    set_bit (RT_SCHED_TIMER_OK, &s->sched_flags);
  }
```

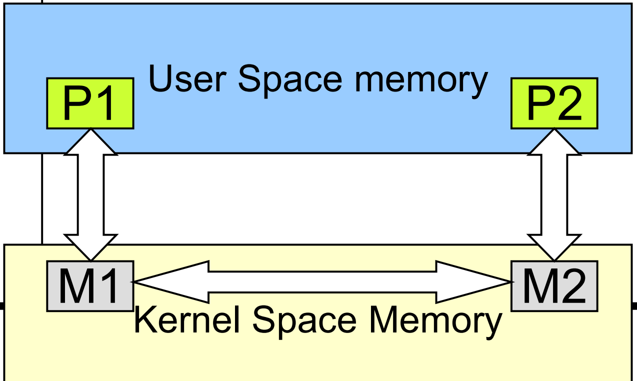
優先權 Task A < Task B

RTOS Kernel::Memory

在 i386 硬體架構下， Jamei 可支援 MMU (virtual memory) 與 MPU (memory protection)

區分 user-space 與 kernel-space memory

```
void start_kernel(void)
{
  /* architecture-dependent */
   init_arch();
  /* NOTE:
       - Mask all interrupts.
       - Interrupts are mapped in 0x20-0x30 IDT entries
  */

#if CONFIG_KERNEL_MEMORYPROT
 init_page();
#endif

#if CONFIG_CONTEXT_MEMORYPROT
 init_context();
#endif
...
```



RTOS Kernel::Memory

Jamei 仿效 POSIX/Linux 的處理

APIs
- kmalloc / kfree
- mmap
- shm (shared memory area)

有彈性的記憶體管理機制
- 放任 (no protected memory)
- 最低限度記憶體保護機制
  - 不可寫入 RT executive 記憶體
- 保護 context 執行單元記憶體

RTOS Kernel::Timer

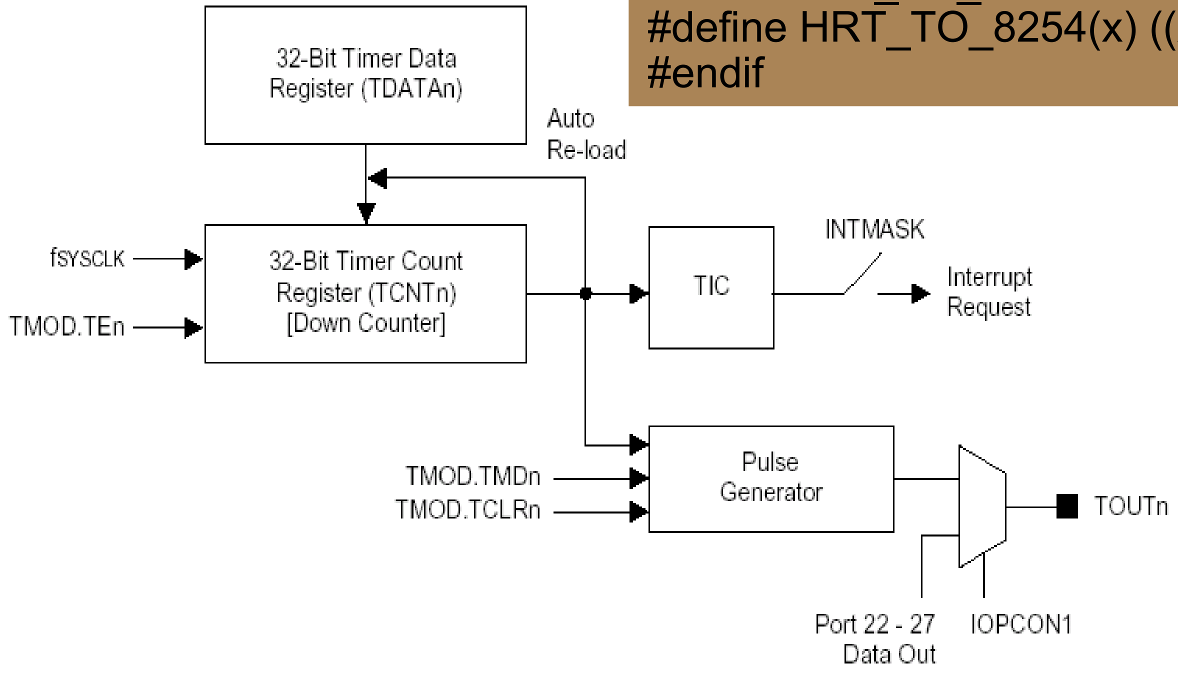
Timer 本質上是硬體時鐘的軟體呈現

型態

watchdog timer

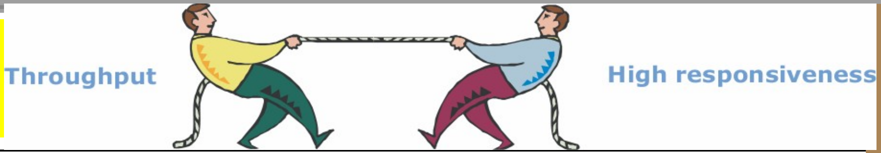
programmable timer

```
#define HRT_FROM_NS(x) (x)
#define HRTICKS_PER_SEC 1000000000

#ifndef HRT_FROM_8254
#ifndef HRT_FROM_8254
#define HRT_FROM_8254(x) ((x) * 838)
#endif

#ifndef HRT_TO_8254
#define HRT_TO_8254(x) ((x) / 838)
#endif
```



RTOS Kernel::Timer

萬惡 HZ



實做 HRT (High Resolution Timer)

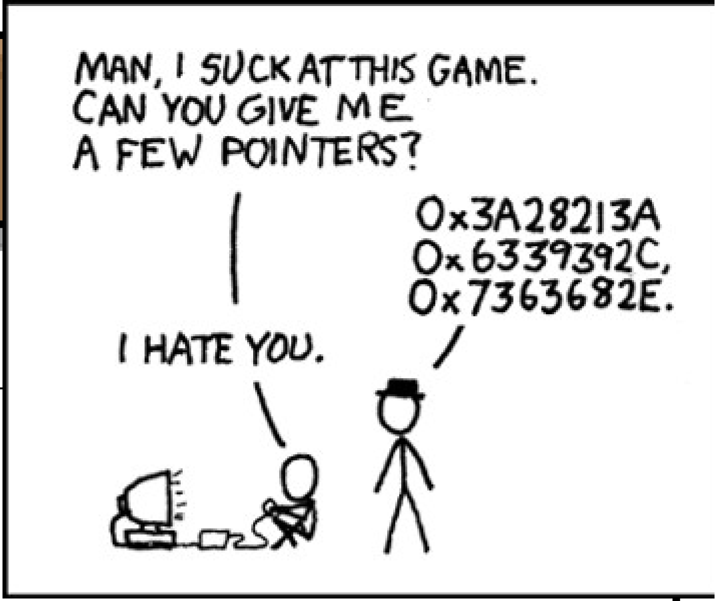
用於 sched, sync 等系統實作

```
struct rt_clock {
        ...
        int (*sethrtime)(
                struct rt_clock *,
                hrtime_t t);

        int (*settimer)(
                struct rt_clock *,
                hrtime_t interval);
        ...
        hrtime_t resolution;
        hrtime_t value;
        hrtime_t delta;
        pthread_spinlock_t lock;
        struct rt_clock_arch arch;
};
```

```
int rt_schedule(void)
{
...
  if ((s->clock->mode == RT_CLOCK_MODE_ONESHOT)) {
    if ((preemptor = find_preemptor(s,new_task)))

    {
      (s->clock)->settimer(s->clock, preemptor->resume_time - now);
    } else {
      (s->clock)->settimer(s->clock, (HRTICKS_PER_SEC / HZ) / 2 );
    }
...
```

RTOS Kernel::I/O

everything is file


MAN, I SUCK AT THIS GAME. CAN YOU GIVE ME A FEW POINTERS?
0x3A28213A 0x6339392C, 0x7363682E.
I HATE YOU.

Interrupt-driven
- polling / DMA

I/O mapping
- memory space 與 I/O 操作的對應

APIs
- open / close
- read / write
- mmap
- register_rtdev / unregister_rtdev

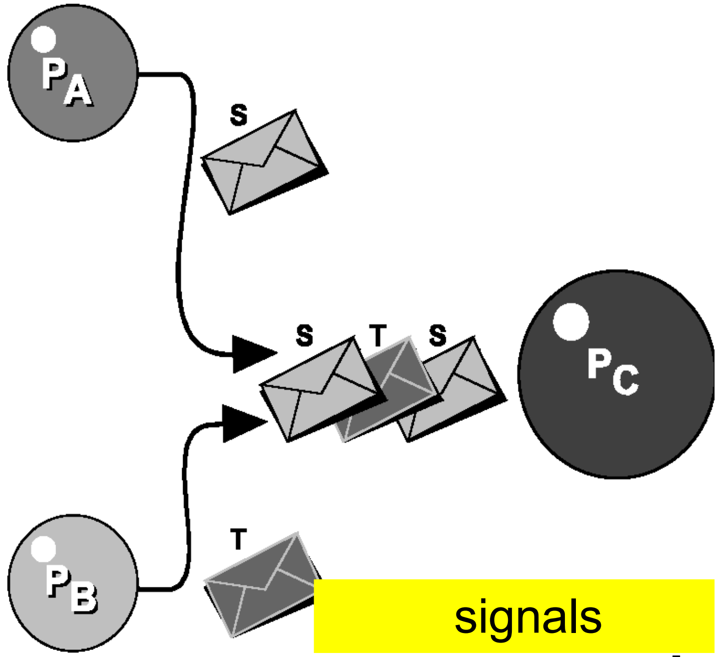
RTOS Kernel::IPC

其實是 Inter-Task Communications

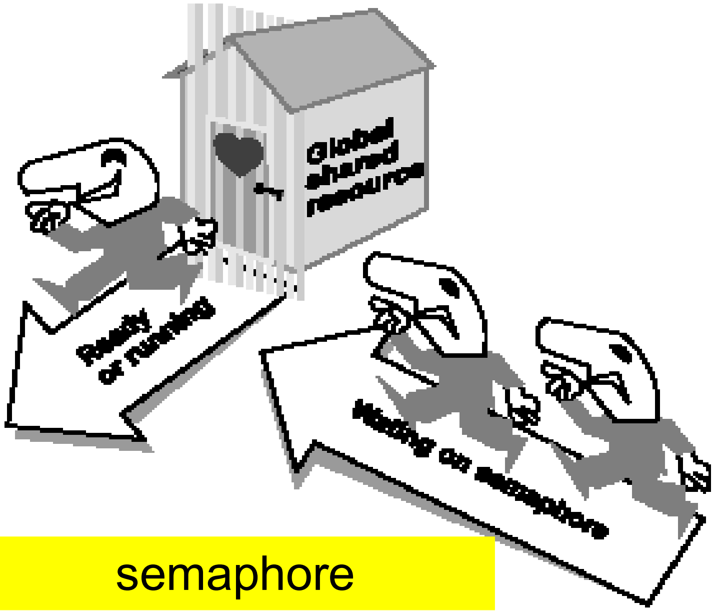
可配置的組態

Signals

Semaphore

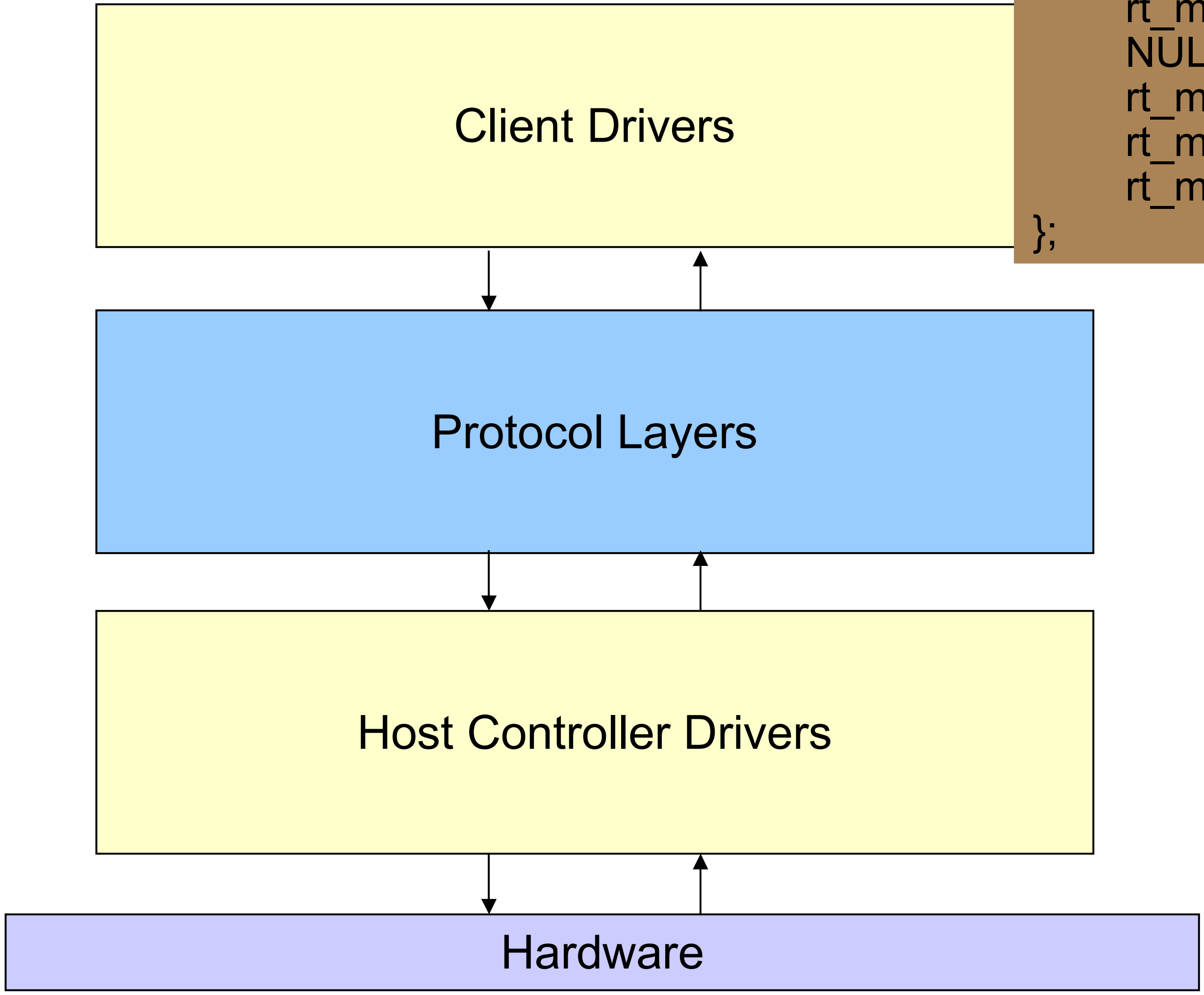


signals



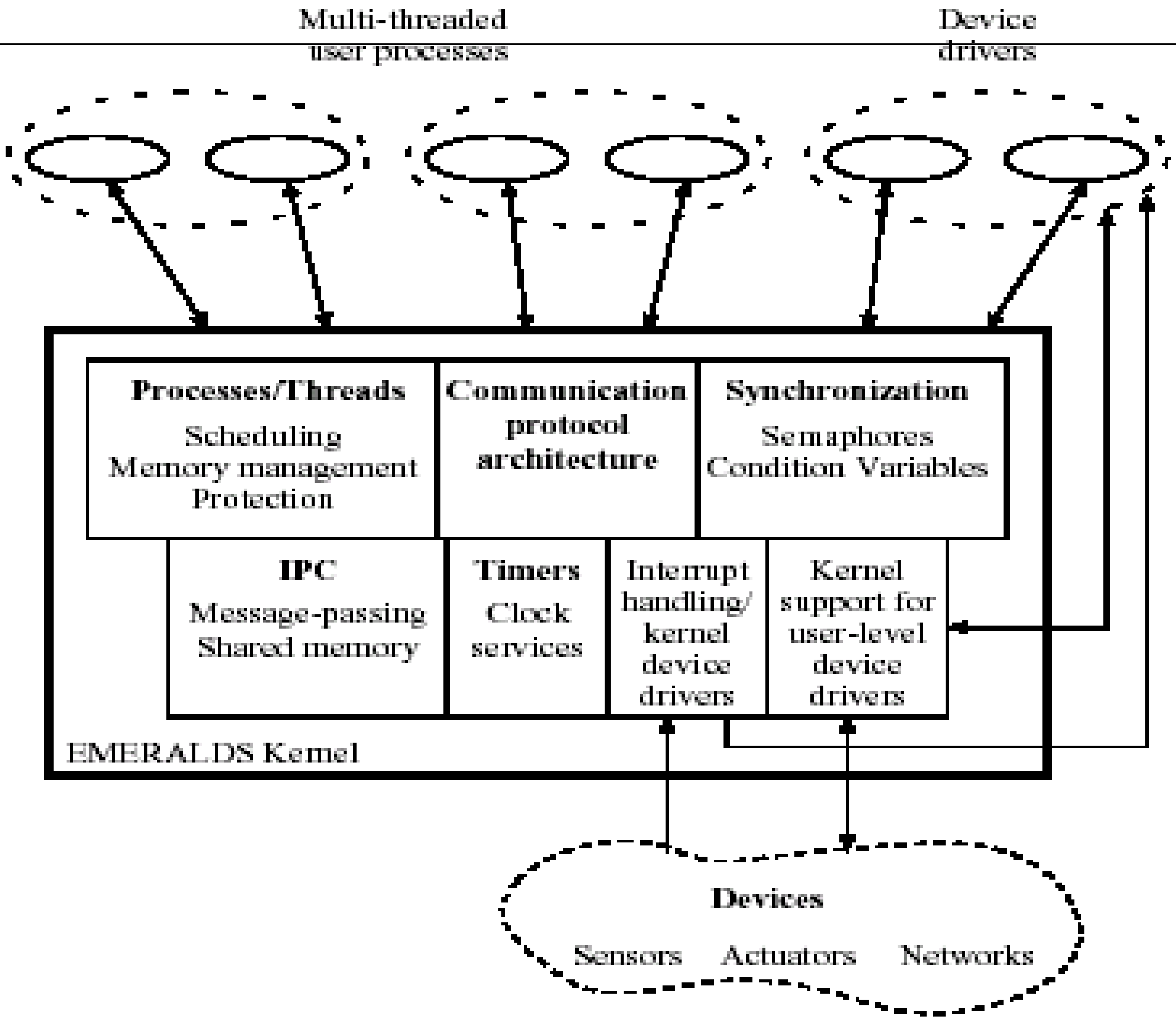
semaphore

RTOS Kernel::Device Driver



```
static struct rt_file_operations
rt_mem_fops = {
        rt_mem_llseek,
        rt_mem_read,
        rt_mem_write,
        NULL,
        rt_mem_mmap,
        rt_mem_open,
        rt_mem_release
};
```

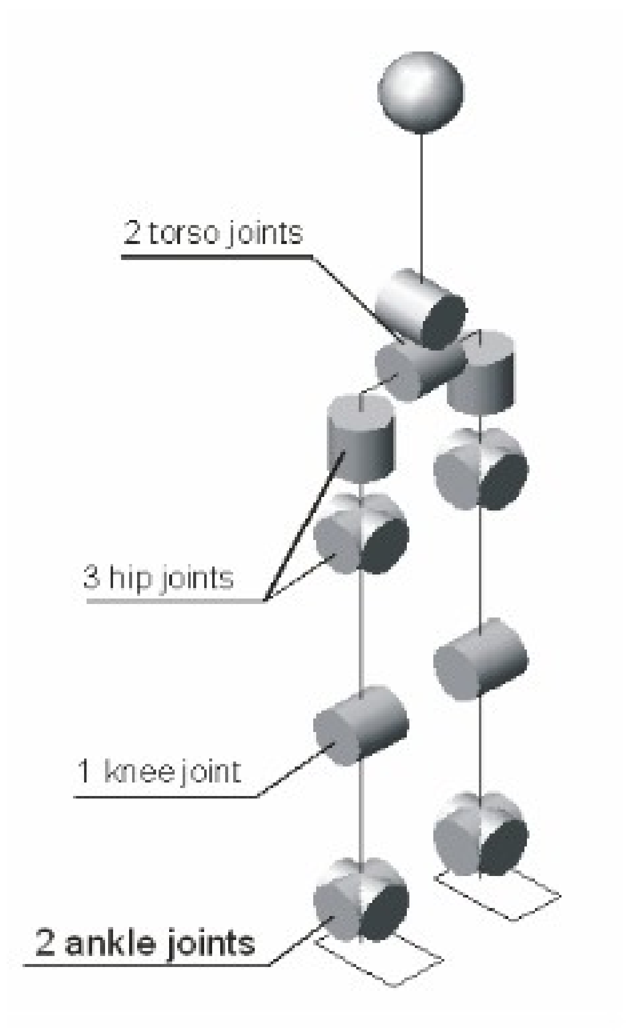
In an Action



Multi-threaded user processes
Device drivers
Processes/Threads
Scheduling
Memory management
Protection
Communication protocol architecture
Synchronization
Semaphores
Condition Variables
IPC
Message-passing
Shared memory
Timers
Clock services
Interrupt handling/ kernel device drivers
Kernel support for user-level device drivers
EMERALDS Kernel
Devices
Sensors
Actuators
Networks

機器人硬體機構概況

硬體：就是電腦系統中，可以讓你踢一腳的地方



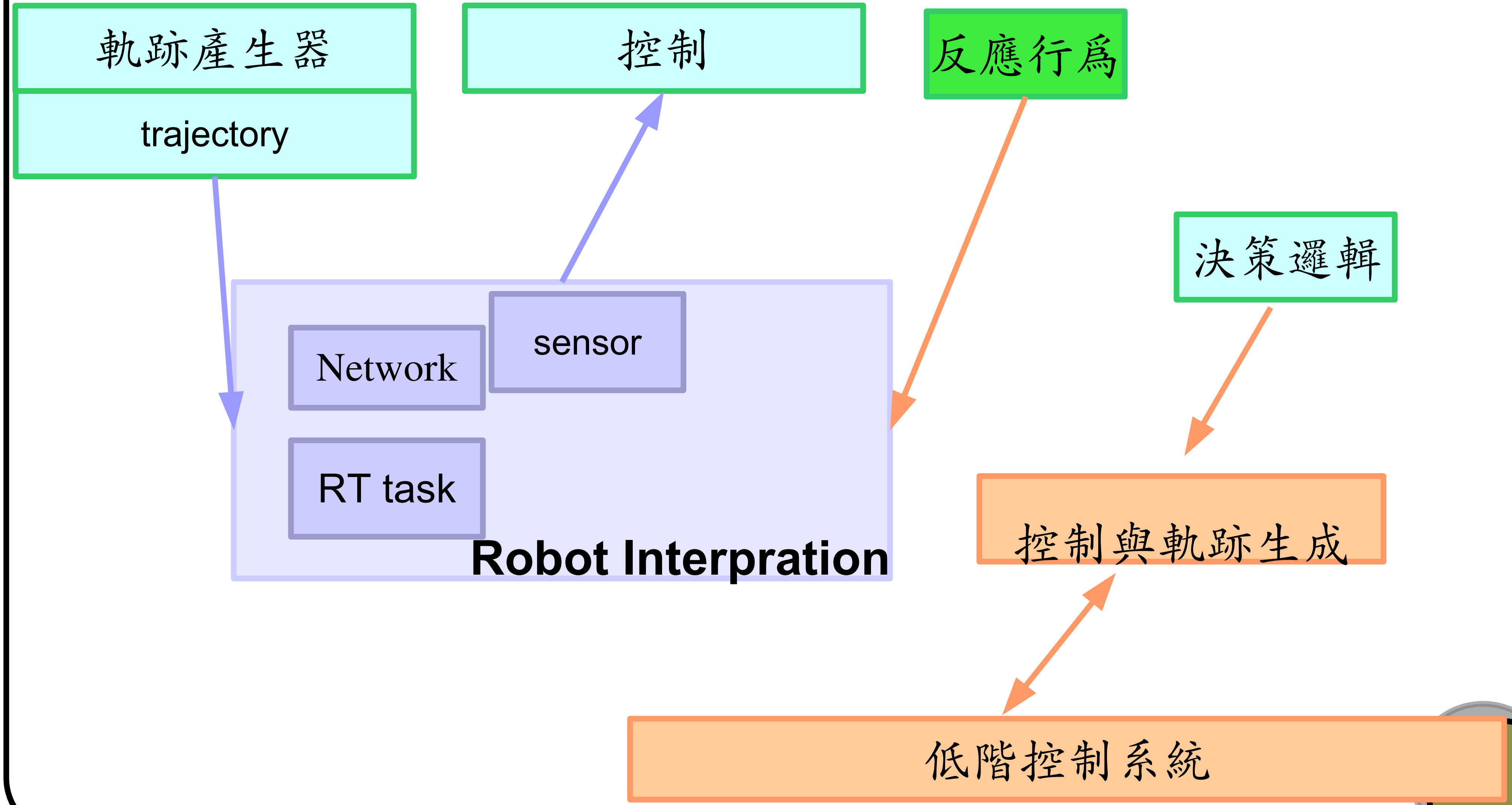
Programming Embedded Systems with C and GNU Development Tools

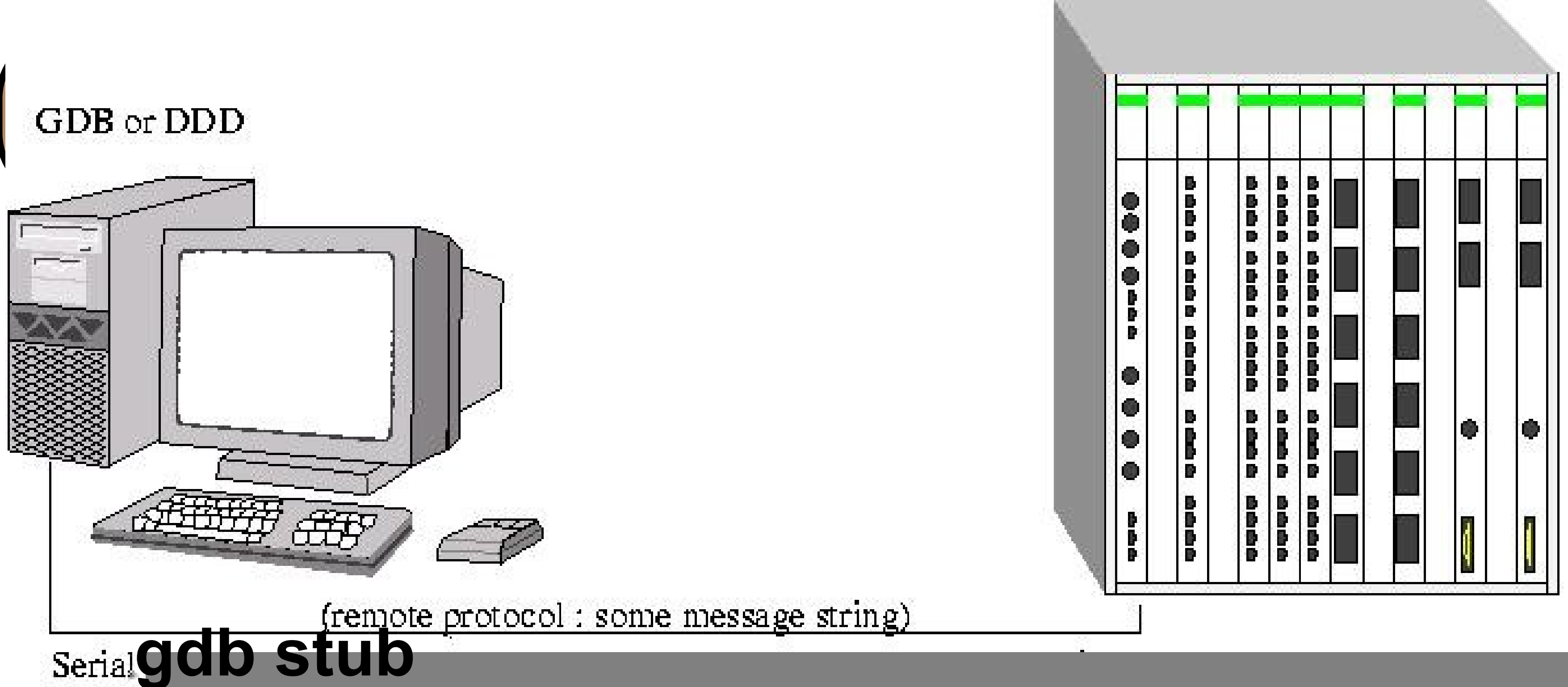
嵌入式系統

— 使用 C 和 GNU 開發工具

O'REILLY

控制系統

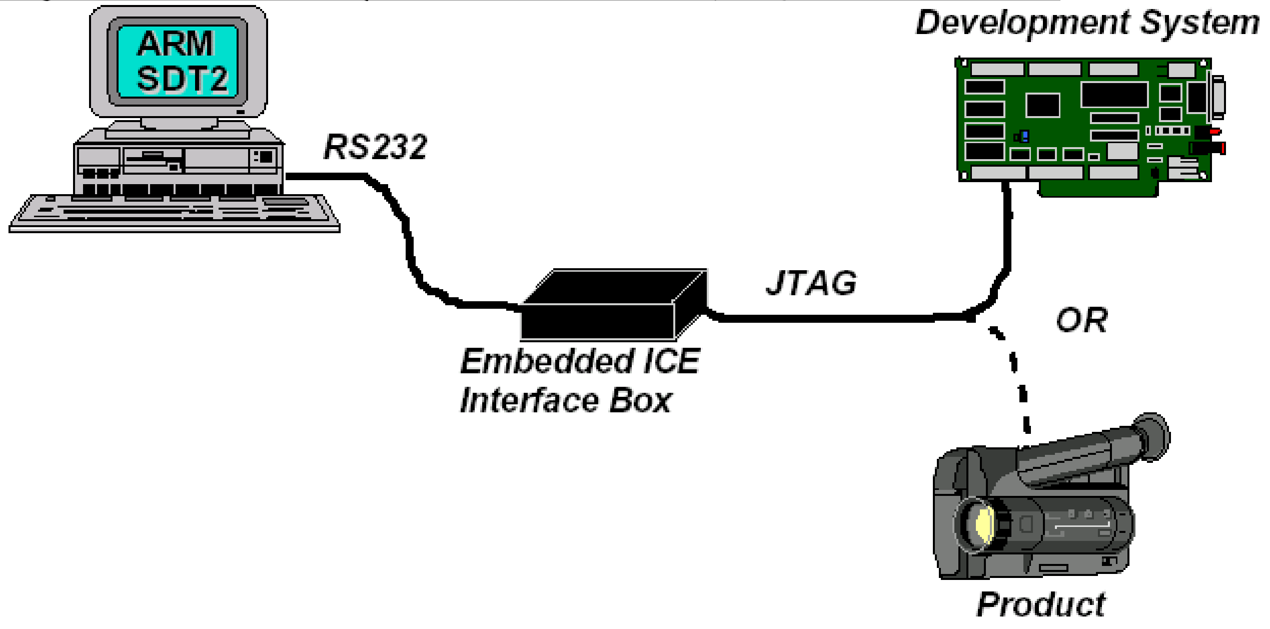

軌跡產生器
trajectory
控制
反應行爲
決策邏輯
Network
sensor
RT task
Robot Interpration
控制與軌跡生成
低階控制系統



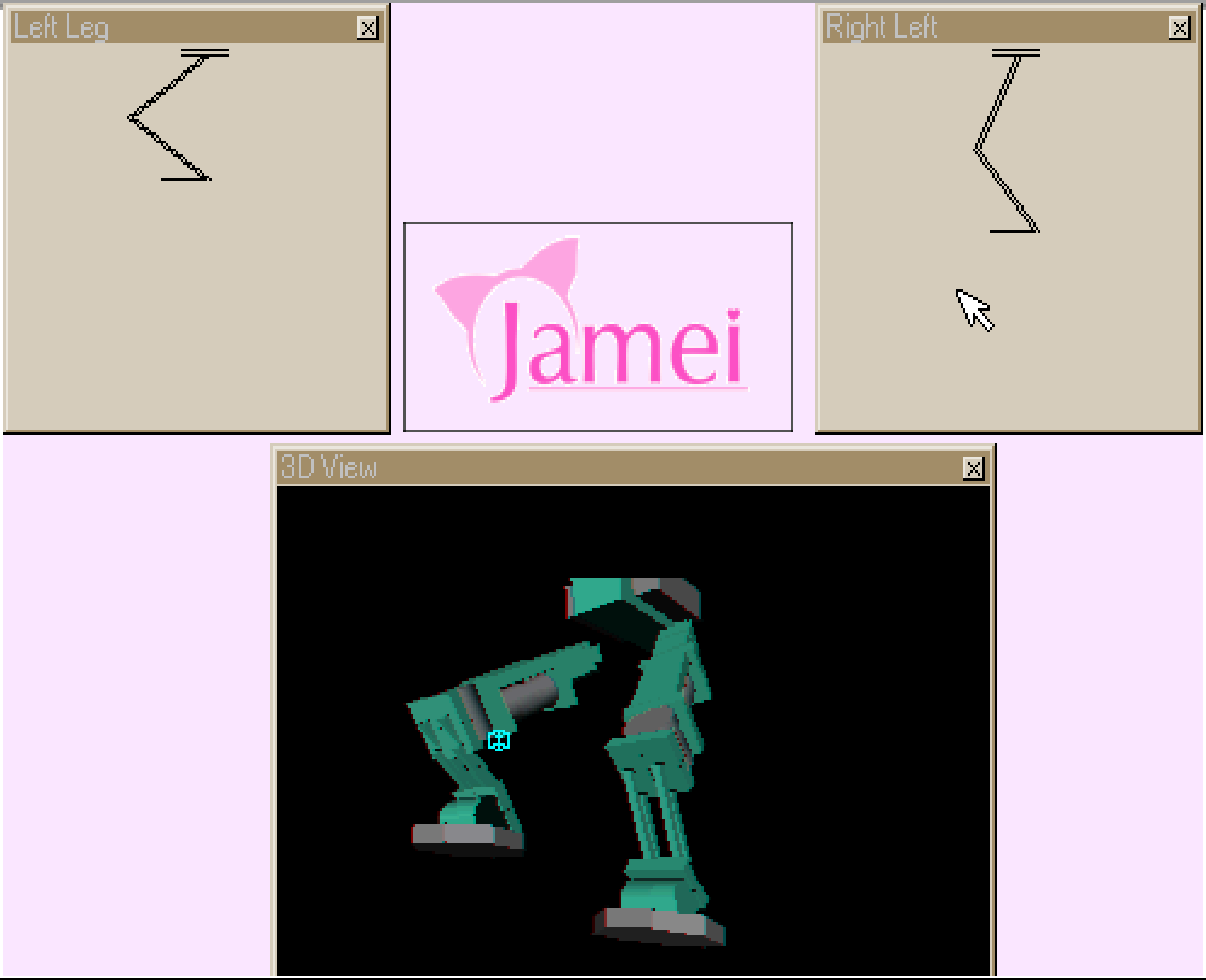
gdb stub

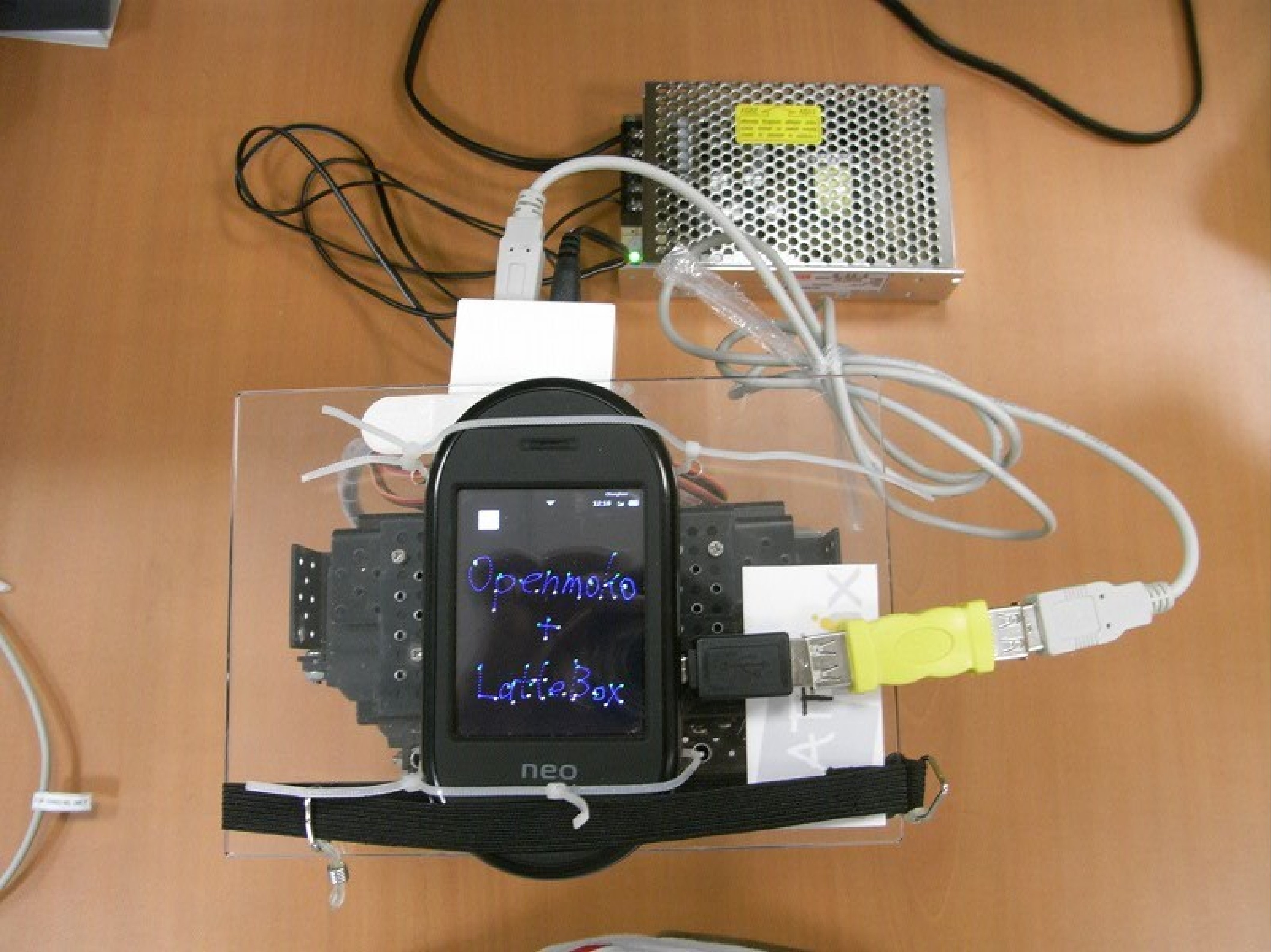
考慮在 emulation/target 模式下，該如何喚起 gdb ？

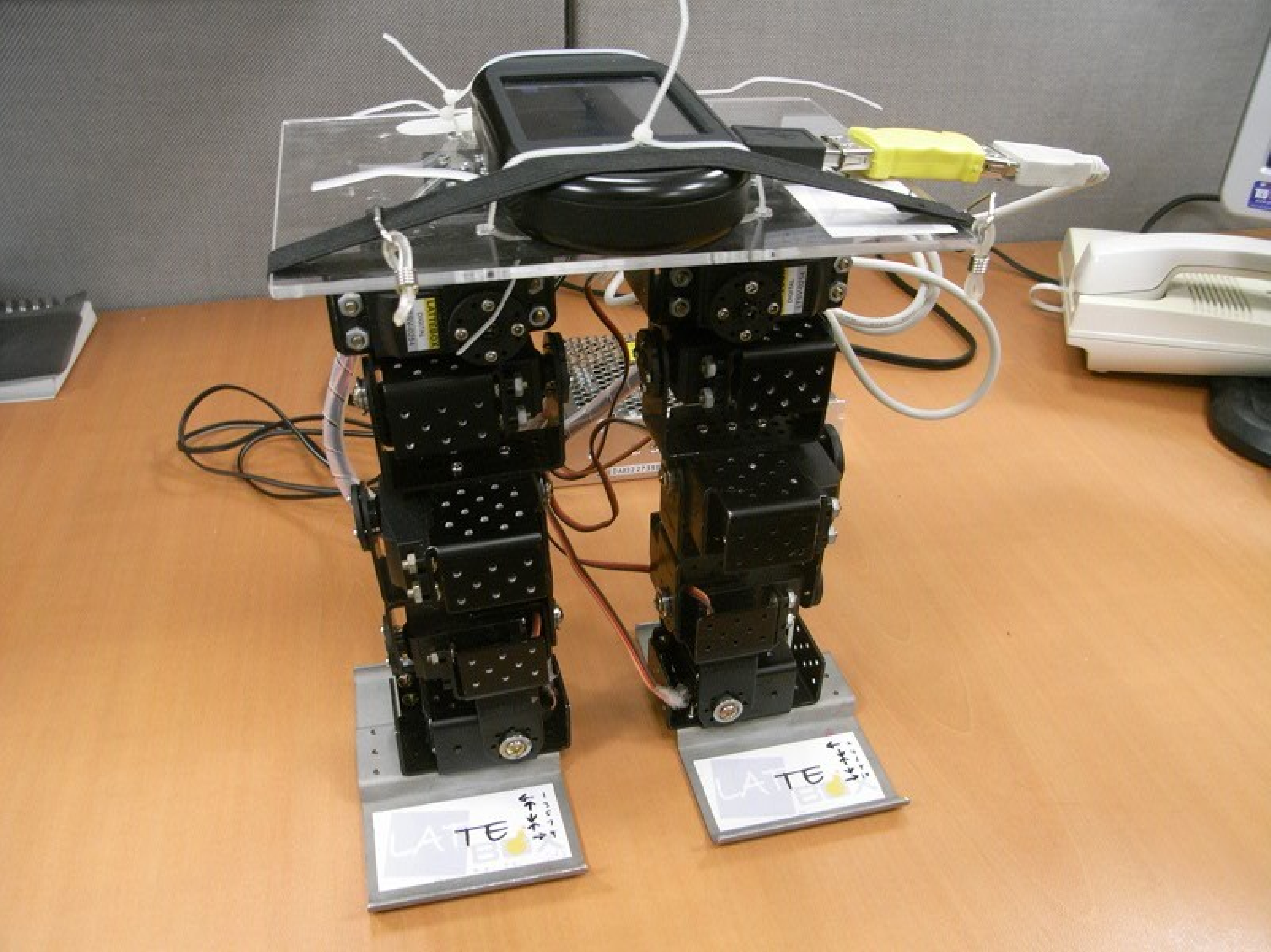
Remote Debugging ：透過 serial 或 TCP/IP 進行遠端除錯



Show me the Robot




Openmoko
+
LatteBox
neo



實現 TCP/IP 的難題

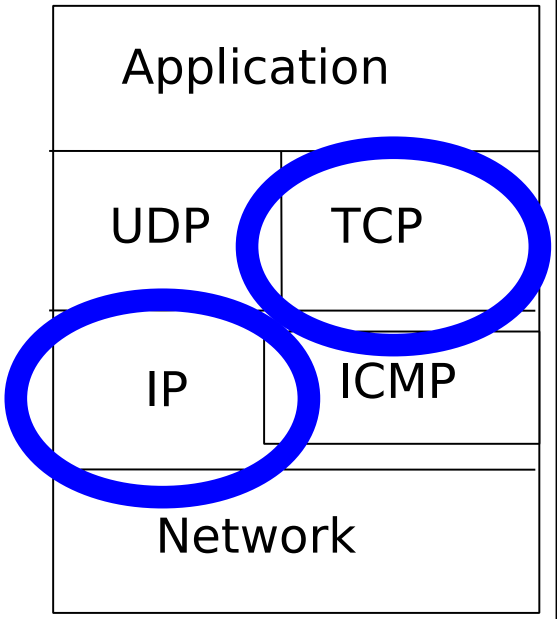
IP Network

- UDP – best-effort datagrams
- TCP – connection oriented, reliable byte-stream, full-duplex
  - Flow control, congestion control, etc
- IP – best-effort packet delivery
  - Forwarding, fragmentation

並未對資源消耗作優化

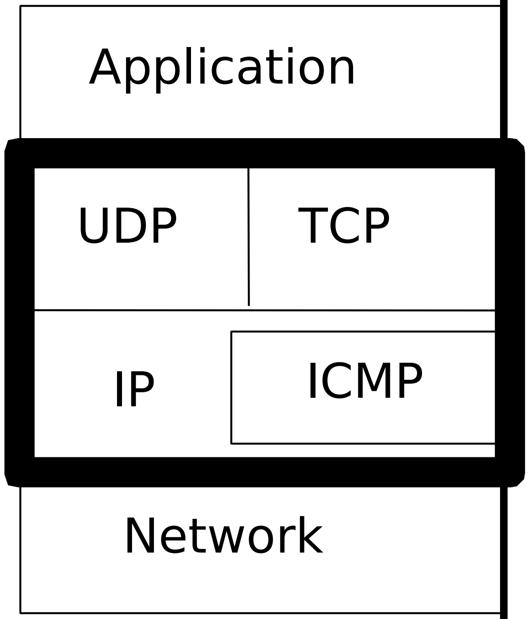
相對來說，是可行的解決方案

共通性



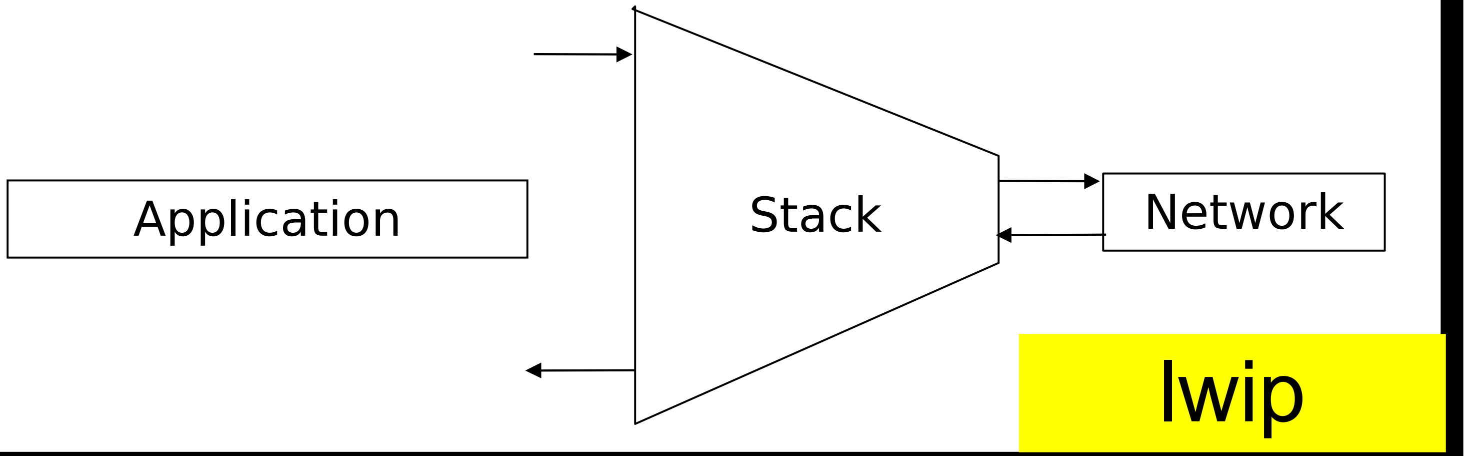
TCP/IP stack design

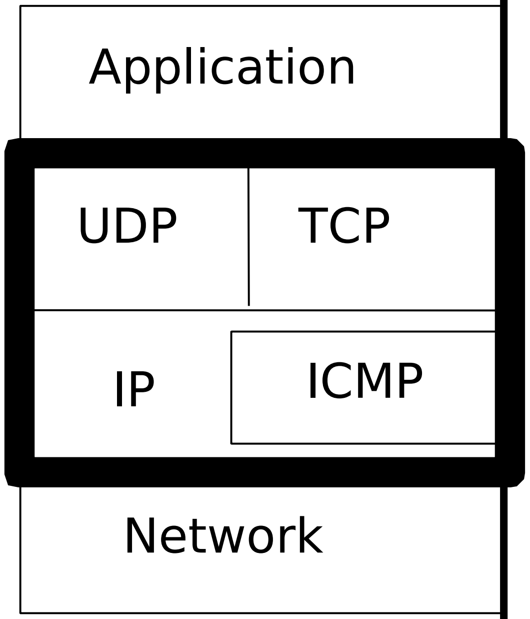
- lwIP – lightweight IP
  - “Application driven” : larger
- µIP – micro IP
  - “Network driven” : smaller



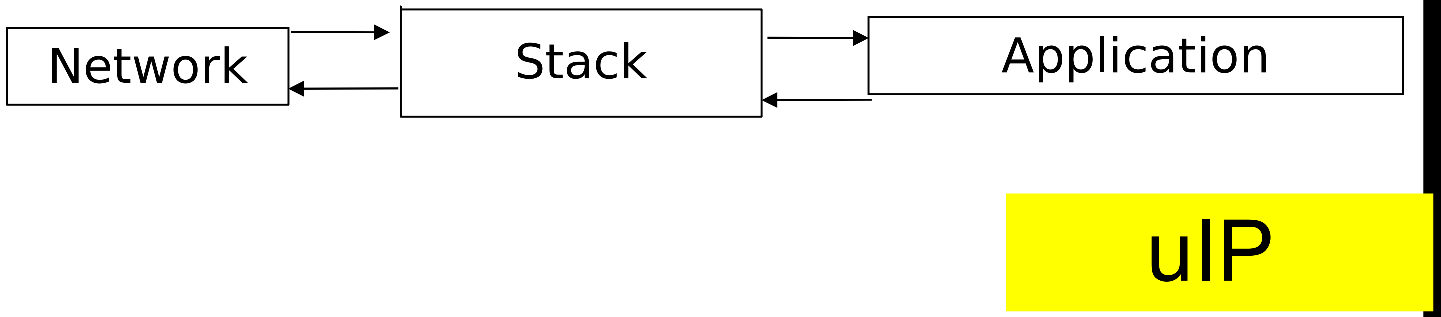
TCP/IP stack design

Application driven



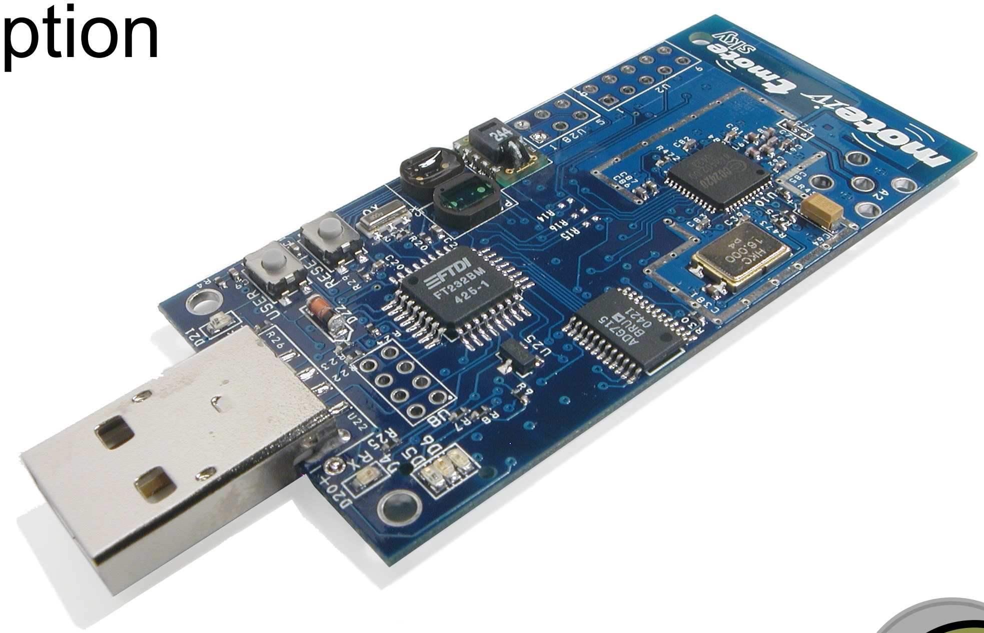


Network driven



資源有限、慾望無窮

- 成本、實體空間
- Memory
  - ~10 k RAM, ~100 k ROM
- Energy, power consumption
  - Batteries, ~10 mW
- Bandwidth
  - ~100 kbits/second



資源限制

memory

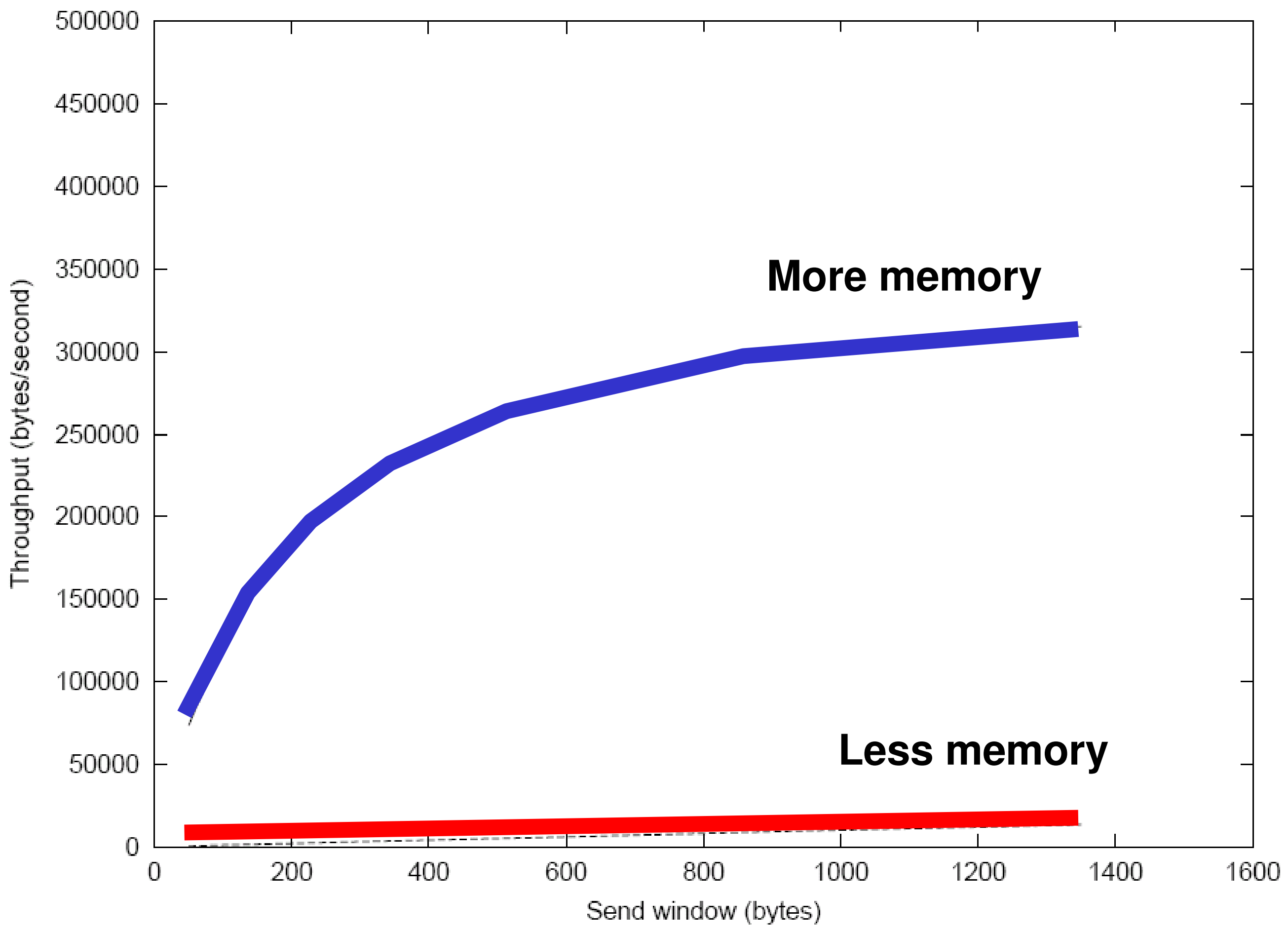
energy

bandwidth

資源限制： Memory+TCP/IP

RAM vs throughput

基本上是兩難



資源限制

memory

energy

bandwidth

資源限制：功耗

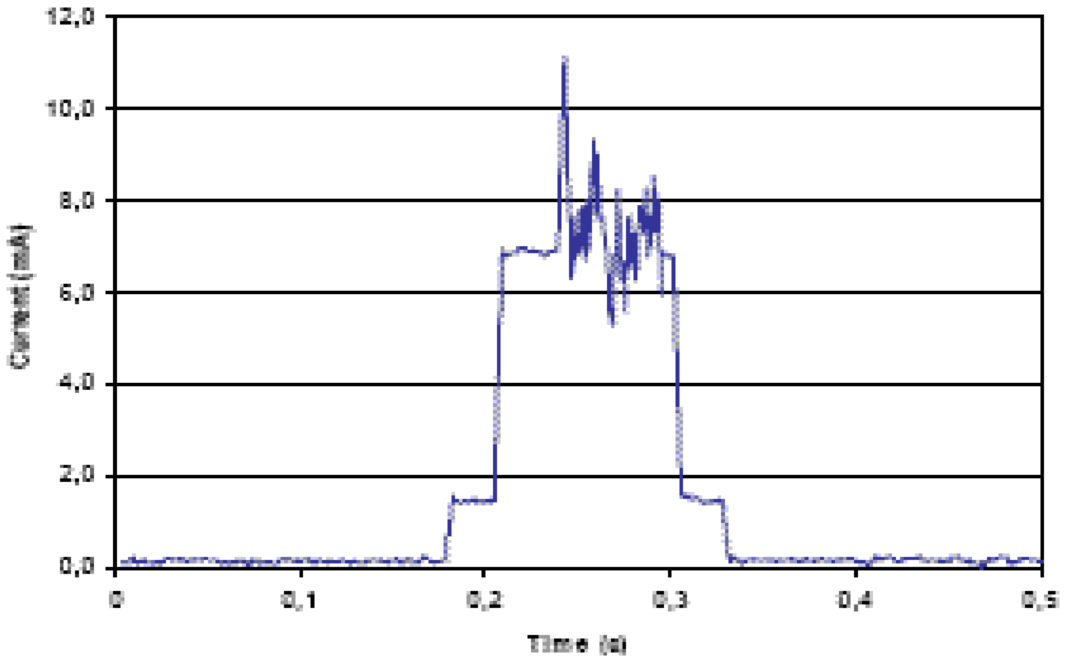


Figure 6. Power consumption: sending

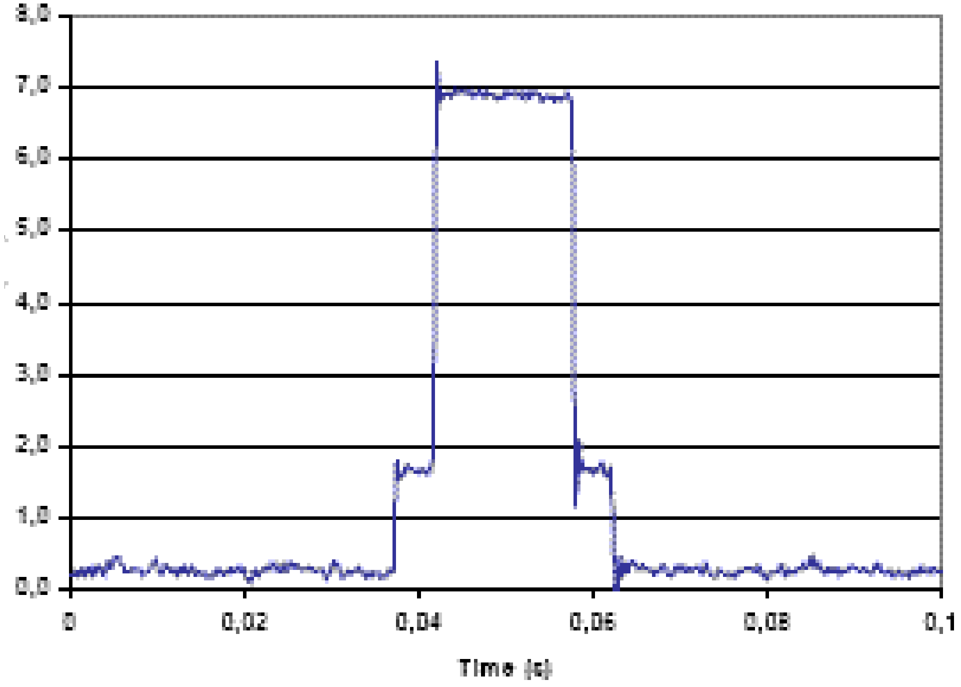
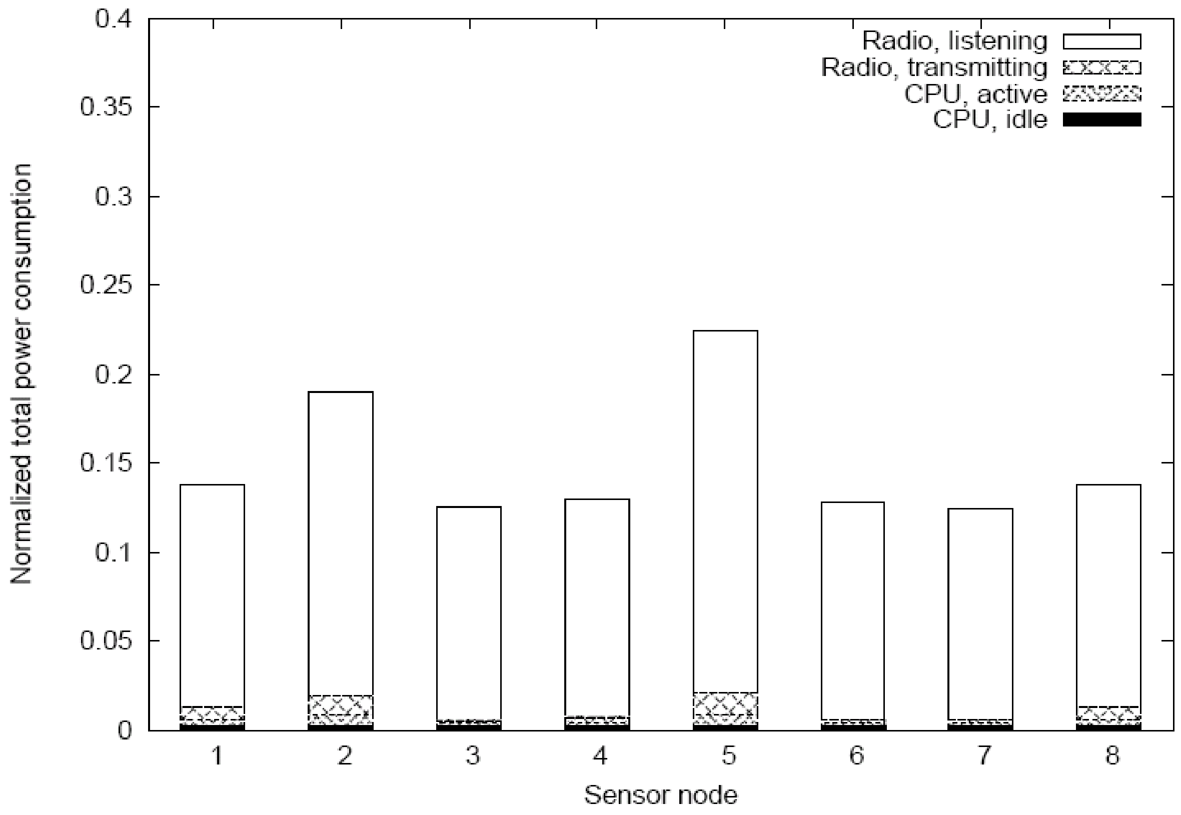


Figure 7. Power consumption: receiving

listening 與 receiving 一樣耗電

資源限制：功耗

- listening 與 receiving 一樣耗電，需透過軟體解決
- Solution 1: don’t listen (ZigBee)
  - Listening nodes have a lot of energy (mains powered)
  - Problem: makes mesh networking difficult
- Solution 2: be smart about listening and energy
  - Power-saving radio mechanisms



資源限制

memory

energy

bandwidth

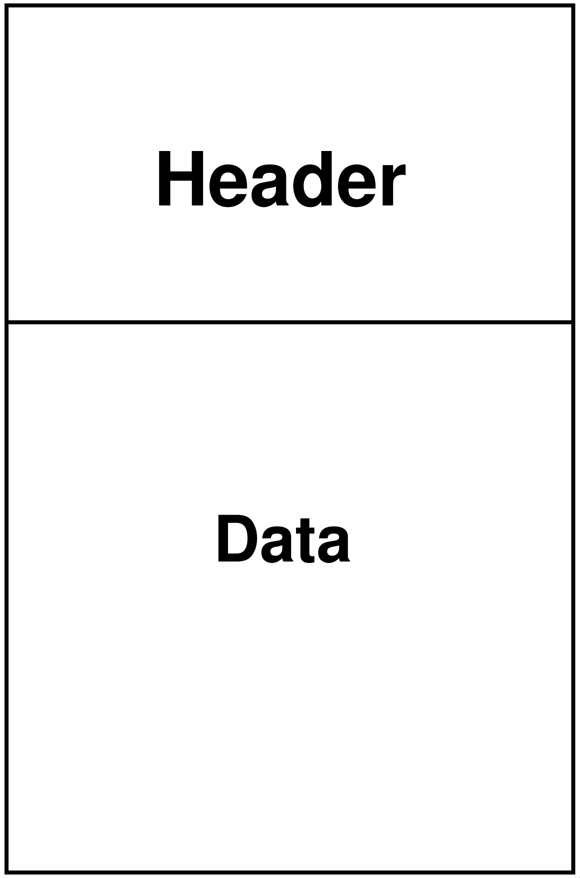
資源限制： TCP/IP bandwidth

~100 kbit/second

Messages are small

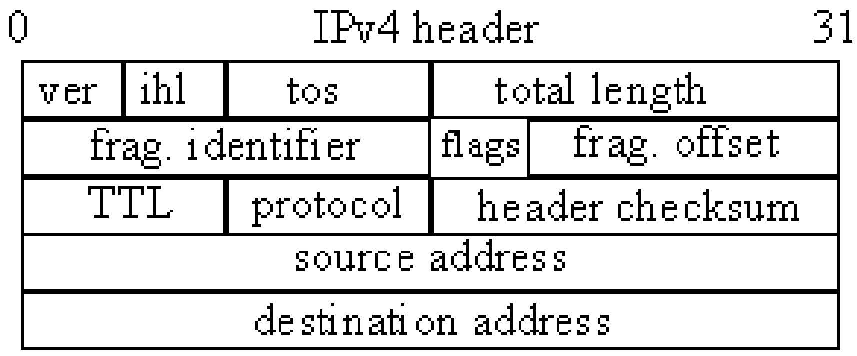
- 802.15.4 max size 128 bytes

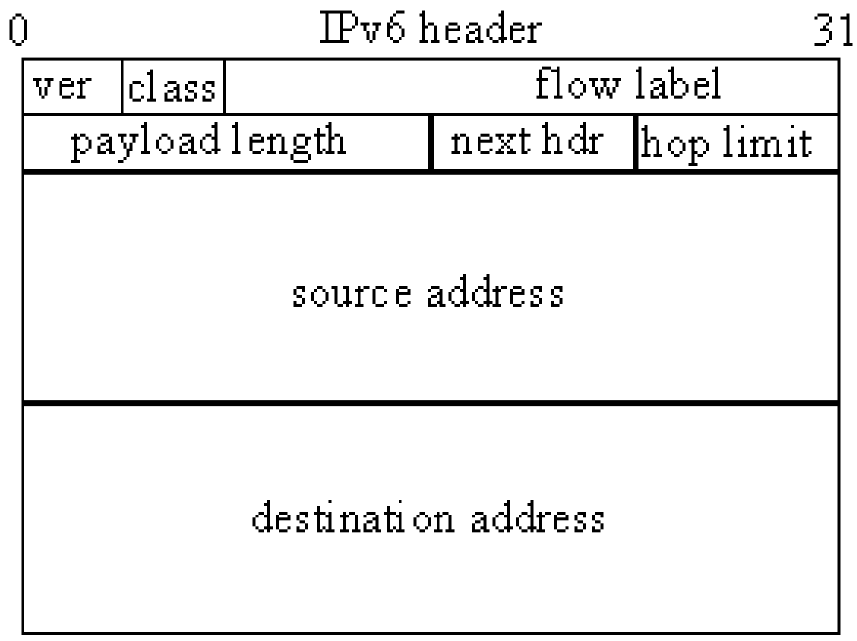
TCP/IP headers are large



TCP/IP header is too long

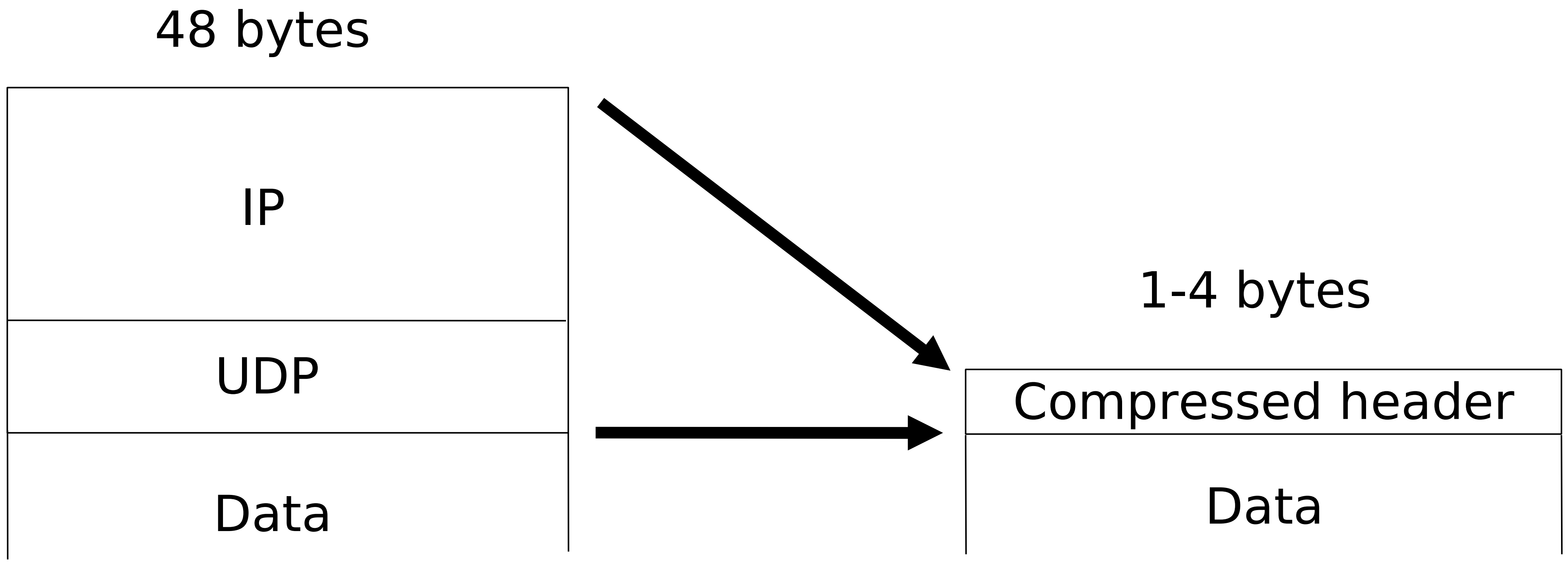
- TCP/IP headers 40 bytes
- UDP/IP headers 28 bytes
- 802.15.4 frame size 128 bytes
  - 27% – 36% overhead
- IPv6 adds 20 bytes
  - 42% – 52% overhead





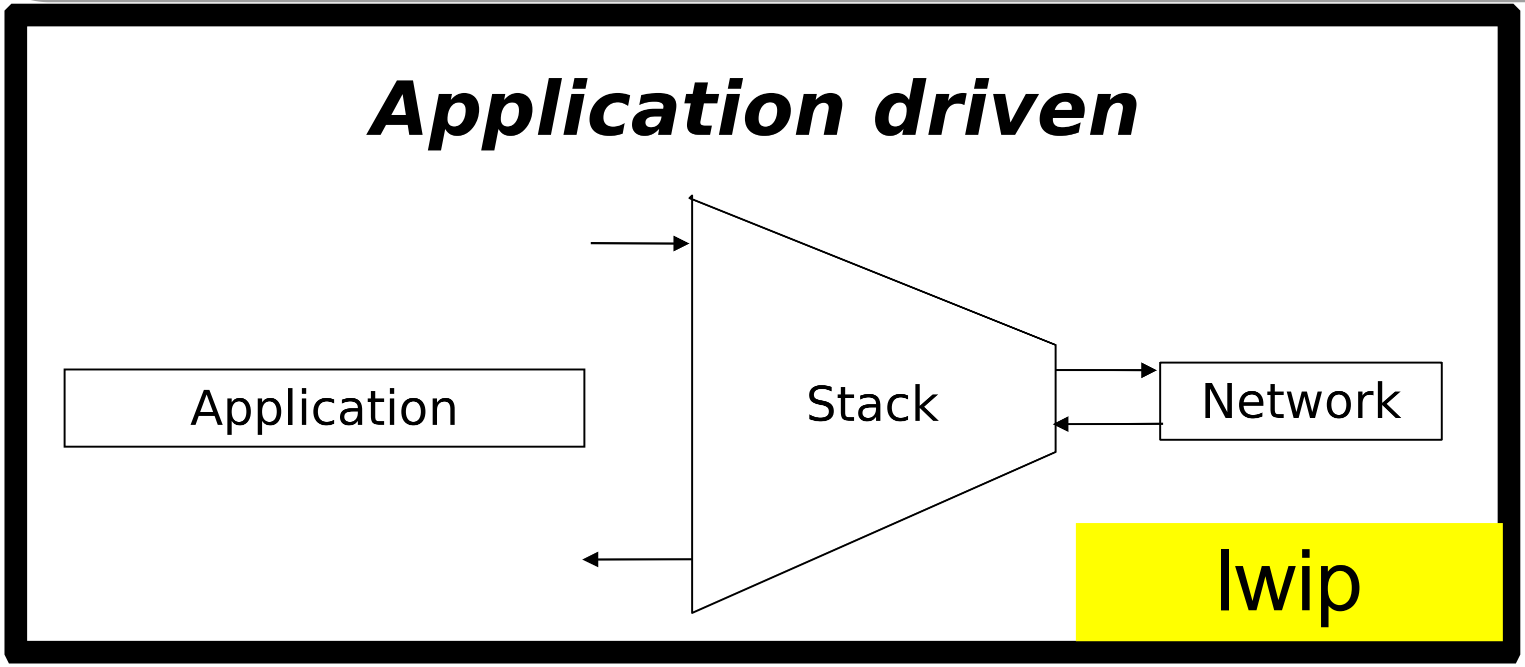
Header compression

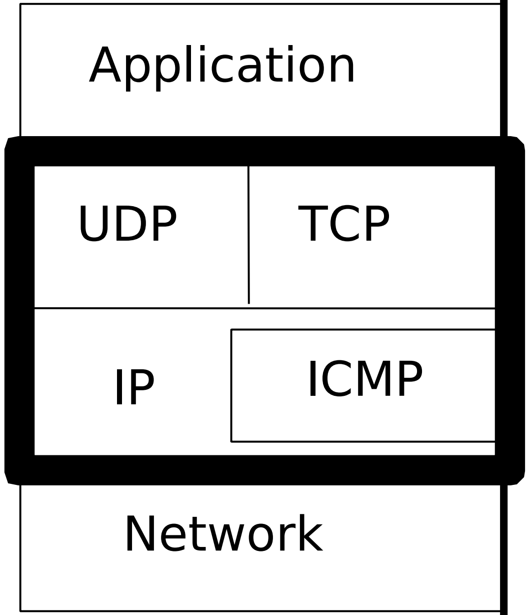
僅傳遞重要資訊

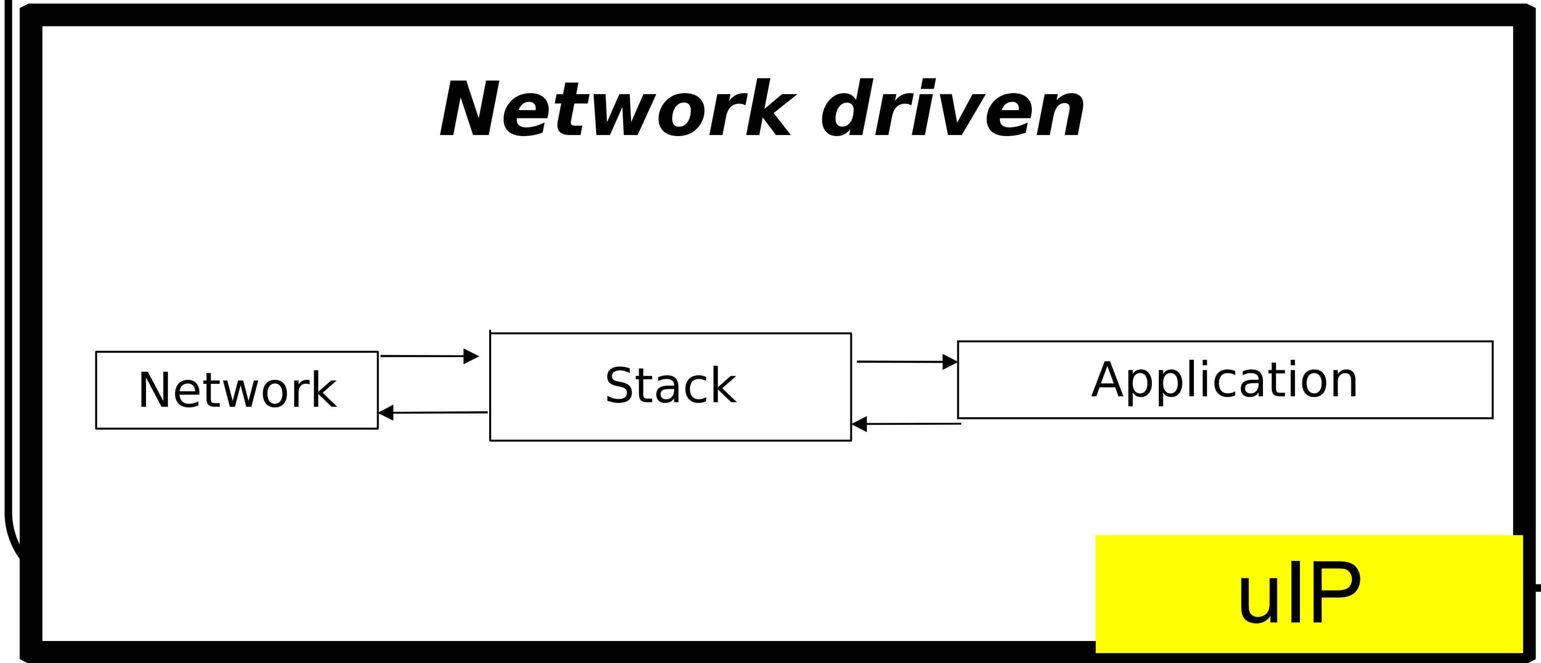


回歸現實：剪裁與調適

TCP/IP stack design

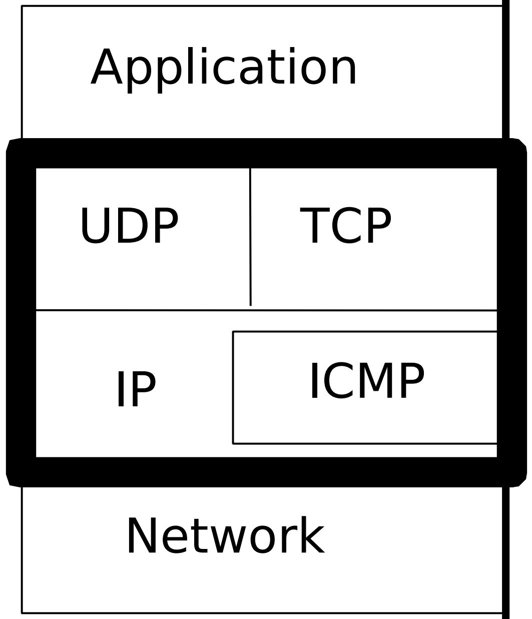


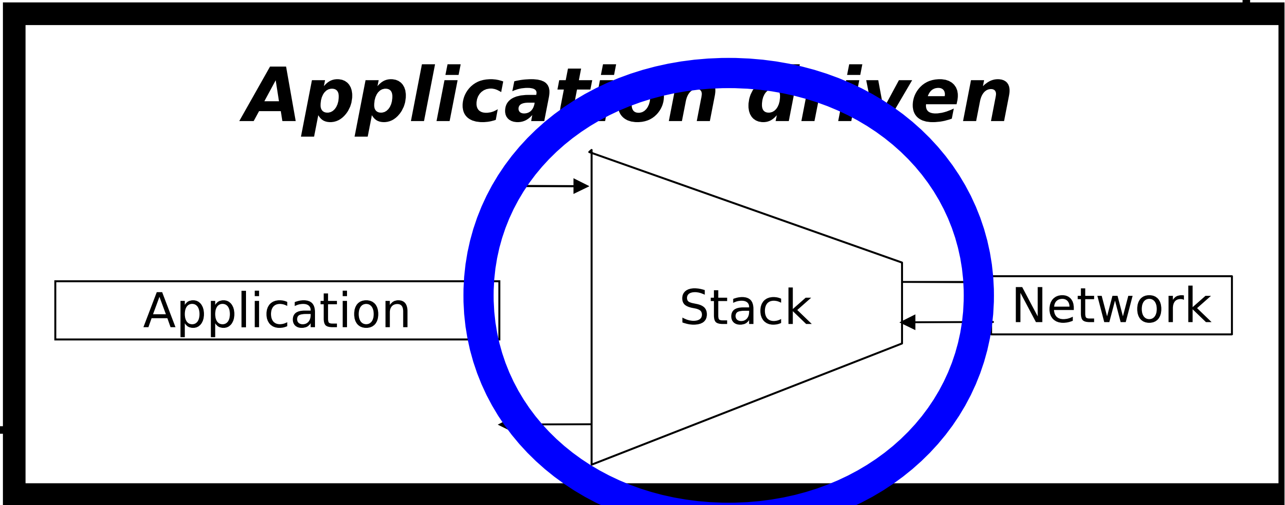




lwIP – Application driven

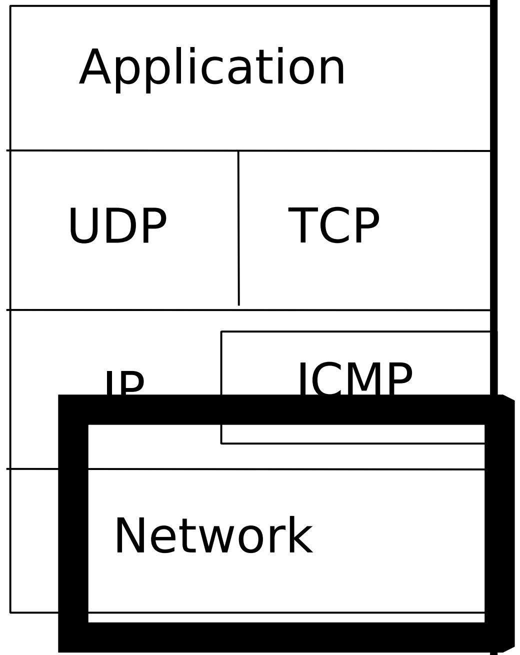
- http://savannah.nongnu.org/projects/lwip/
- lwIP – lightweight IP
- First release late 2000
- 40k code, 40k RAM
- Application driven design
  - Similar to Linux, BSD stacks
- Middle-end

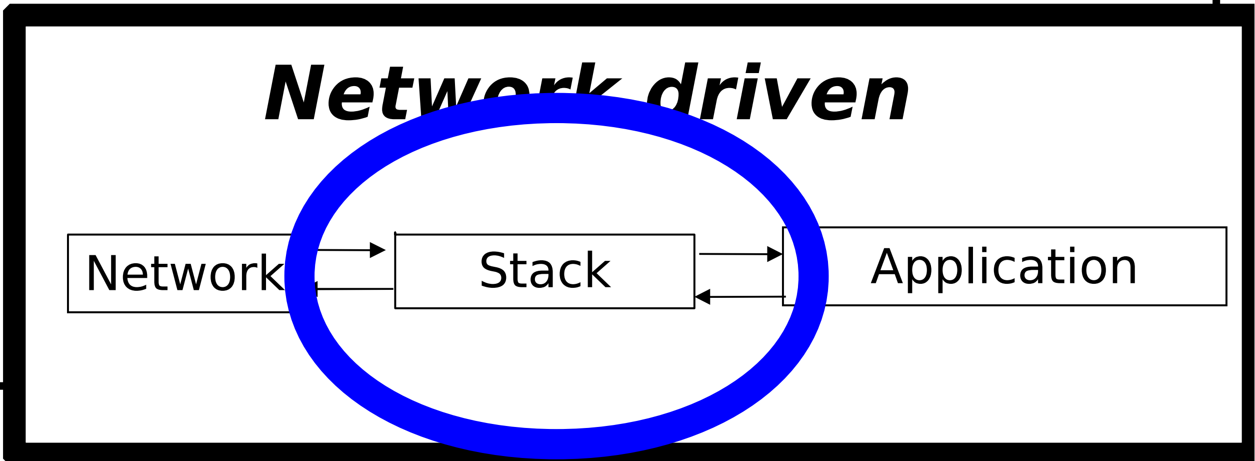




µIP – smallest "full" TCP/IP stack

- http://www.sics.se/~adam/uip/
- First release 2001
- 3 - 5k code, 100 bytes - 2k RAM
- "Full" TCP/IP (RFC compliant)
- Used in Contiki OS



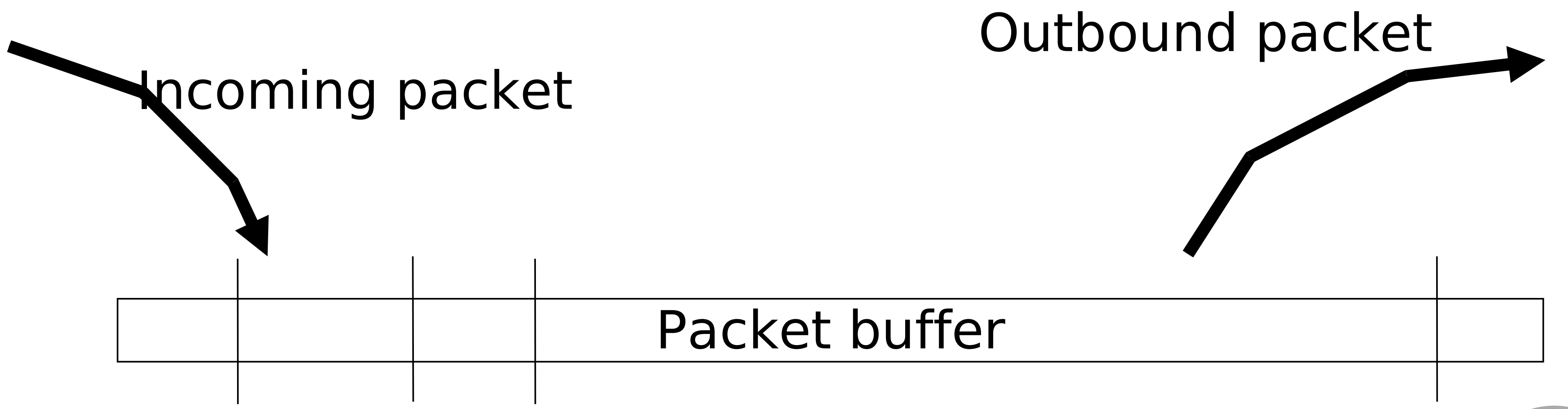


µIP 技巧

- Shared packet buffer
- Lower throughput
- Event-driven APIs

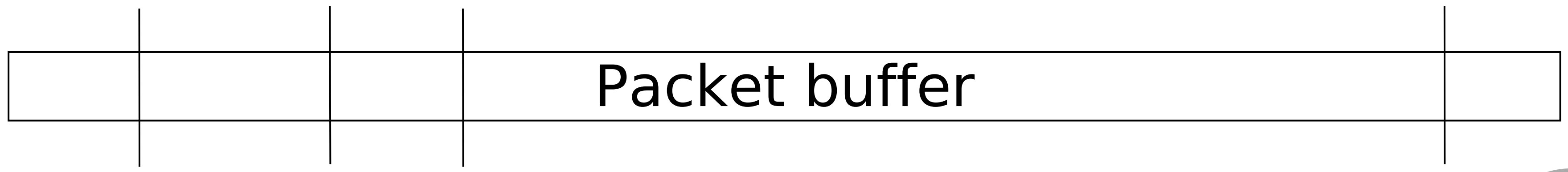
Shared packet buffer

- **所有**封包 (outbound + inbound) 使用同一份緩衝區
  - 緩衝區的大小決定 throughput



Shared packet buffer

- Implicit locking: single-threaded access
  1) Grab packet from network – put into buffer
  2) Process packet
     - Put reply packet in the same buffer
  3) Send reply packet into network



Lower Throughput

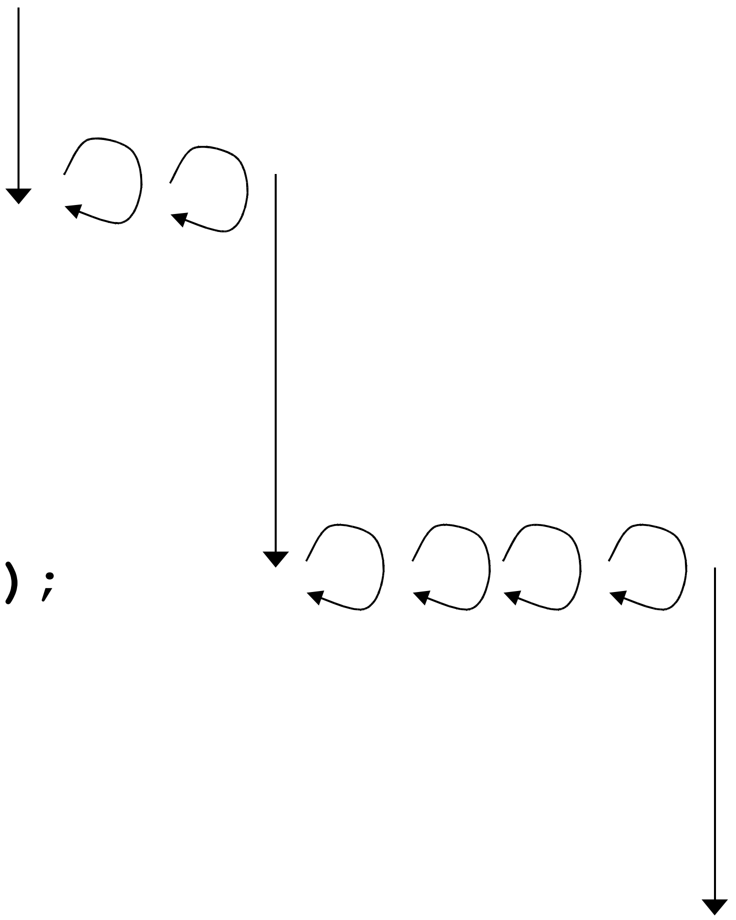
- µIP trades throughput for RAM
  - Low RAM usage = low throughput
- 小的系統通常不會有太多資料
- **「具通訊的能力」會比 throughput 多寡來得重要**

Event-driven APIs

- µIP 沒有 BSD sockets
  - BSD sockets 建構於 threads 之上
  - Threads 引來 overhead (RAM)
- 因此，透過特製的 event-driven API
- co-routine
  - Applications are called by µIP, call must `return`
- Protosockets – BSD socket-like API based on protothreads

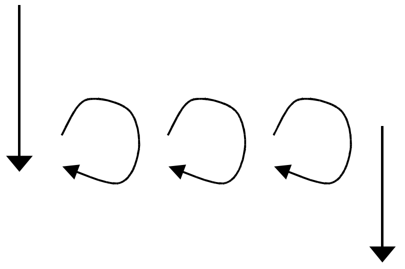
Protothread

```
int a_protothread(struct pt *pt) {
  PT_BEGIN(pt);

   /* … */

  PT_WAIT_UNTIL(pt, condition1);

   /* … */

  if(something) {

     /* … */

    PT_WAIT_UNTIL(pt, condition2);

     /* … */

  }

  PT_END(pt);
}
```

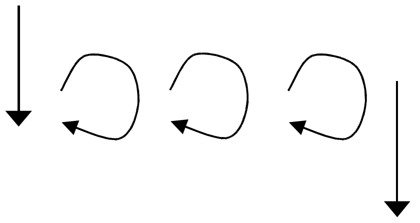


Hierarchical Protothreads

```
int a_protothread(struct pt *pt) {
  static struct pt child_pt;

  PT_BEGIN(pt);

  PT_INIT(&child_pt);
  PT_WAIT_UNTIL(pt2(&child_pt) != 0);

  PT_END(pt);
}
```



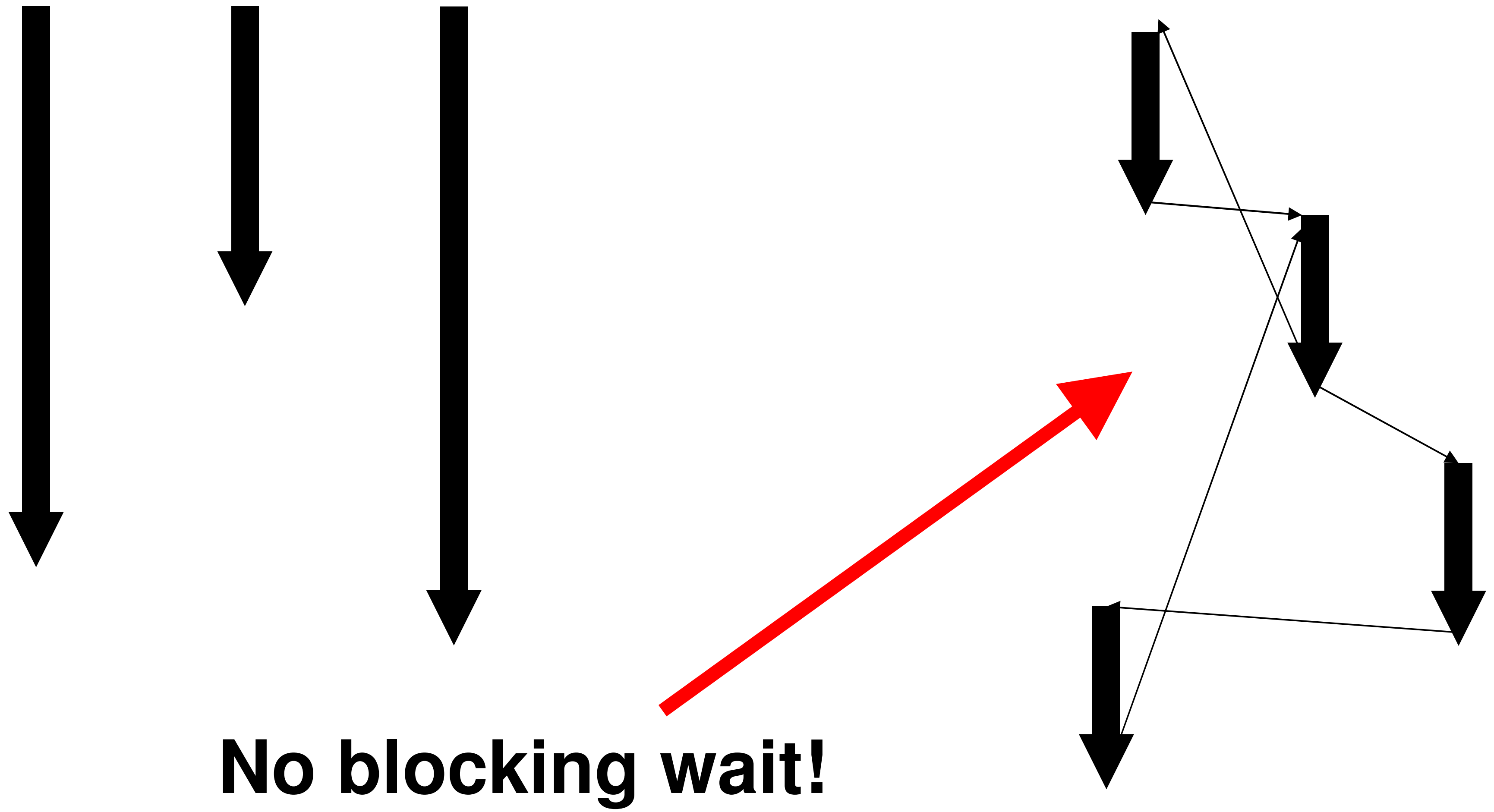
```
int pt2(struct pt *pt) {
  PT_BEGIN(pt);

  PT_WAIT_UNTIL(pt, condition);

  PT_END(pt);
}
```



Threads vs. Events

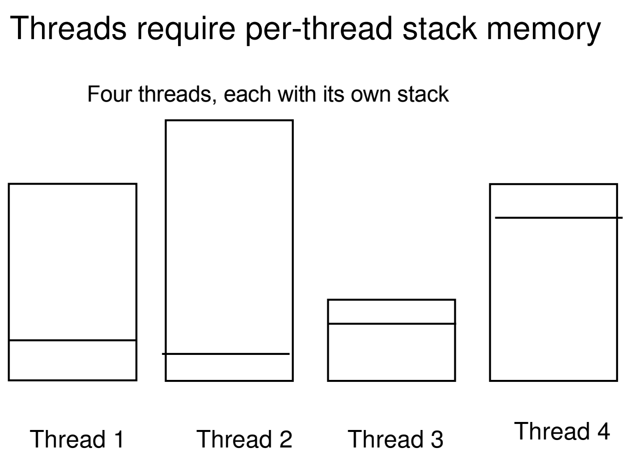
Threads: sequential code flow

Events: unstructured code flow



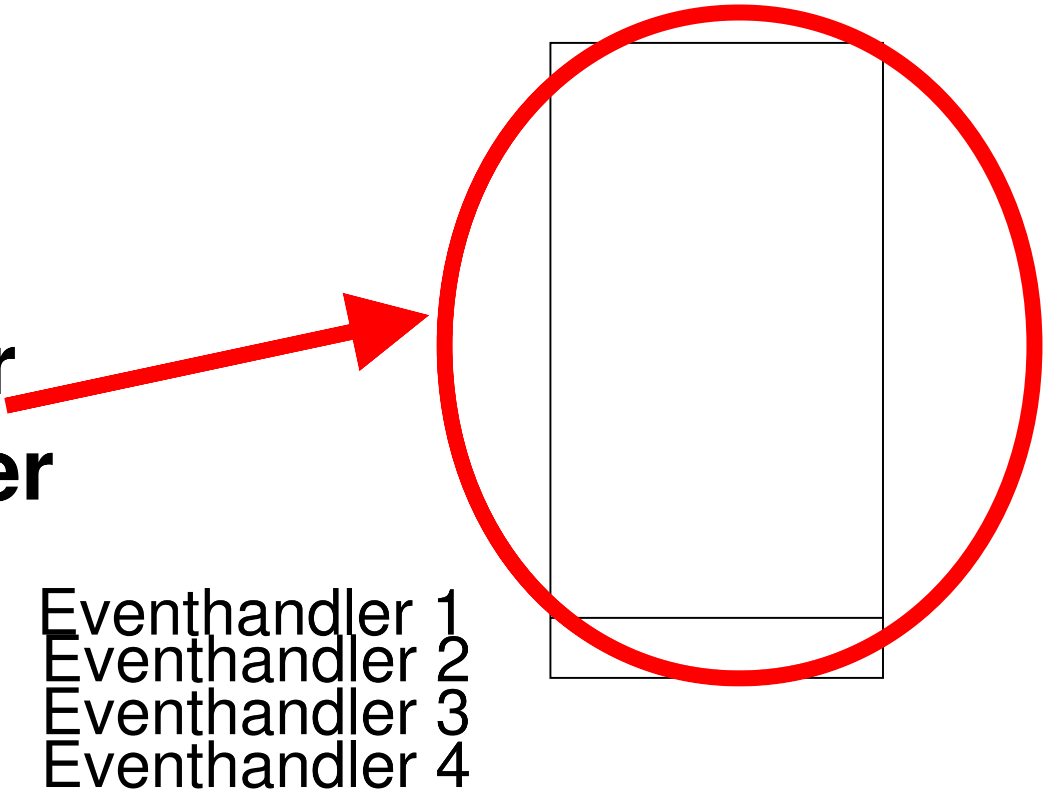
No blocking wait!

Events require one stack


Threads require per-thread stack memory
Four threads, each with its own stack
Thread 1
Thread 2
Thread 3
Thread 4

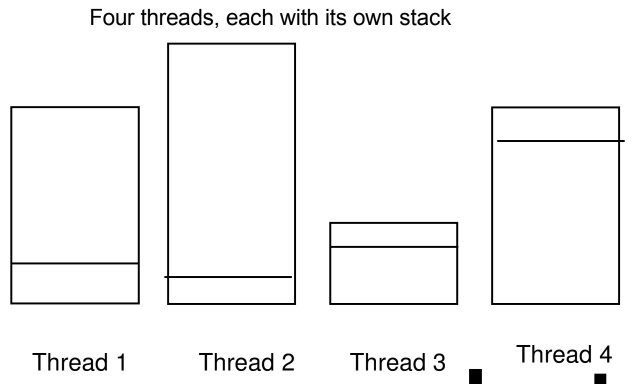
- Four event handlers, one stack

Stack is reused for every event handler



Protothreads require one stack

Threads require per-thread stack memory

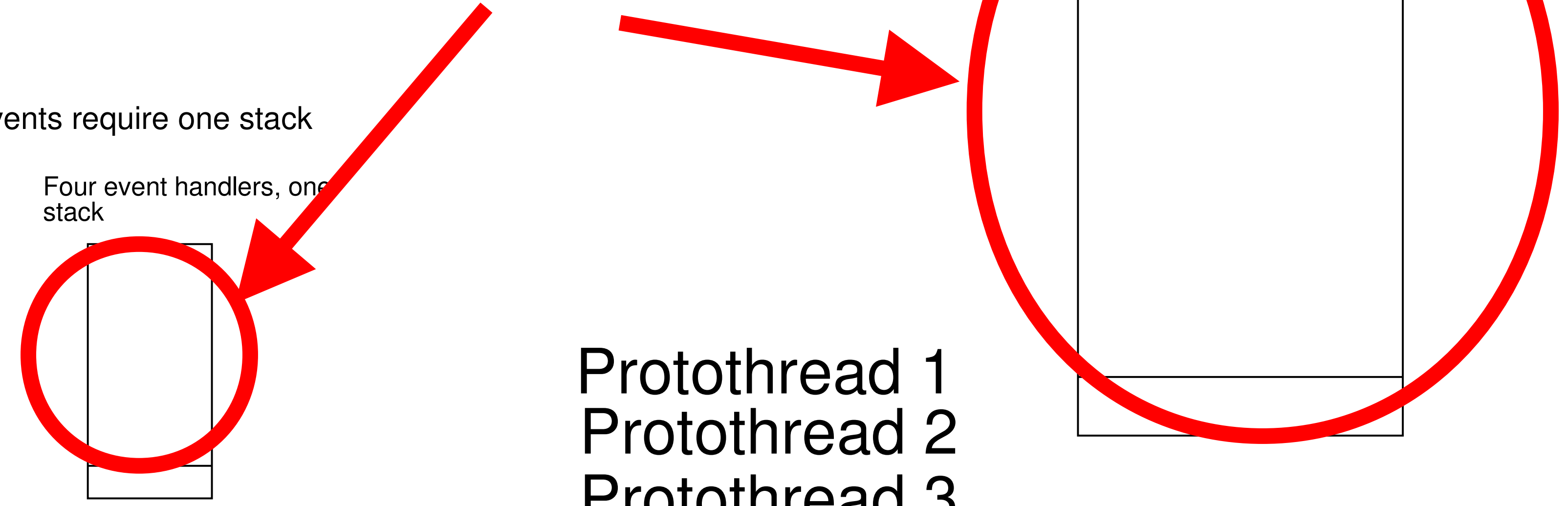


- Four protothreads, one stack

Just like events

Events require one stack

- Four event handlers, one stack



Protothread 1
Protothread 2
Protothread 3
Protothread 4

Event-driven APIs

```
void example2_app(void) {
   struct example2_state *s =
     (struct example2_state *)uip_conn->appstate;

   if(uip_connected()) {
      s->state = WELCOME_SENT;
      uip_send("Welcome!\n", 9);
      return;
   }

   if(uip_acked() &&
      s->state == WELCOME_SENT) {
      s->state = WELCOME_ACKED;
   }

   if(uip_newdata()) {
      uip_send("ok\n", 3);
   }

   if(uip_rexmit()) {
      switch(s->state) {
      case WELCOME_SENT:
         uip_send("Welcome!\n", 9);
         break;
      case WELCOME_ACKED:
         uip_send("ok\n", 3);
         break;
      }
   }
}
```

Event-driven APIs

- Event-driven API 不見得適用所有程式
- Protosockets: sockets-like API using protothreads
  - Extremely lightweight stackless threads
  - 2 bytes per-thread state, no stack
- Protothreads 允許“ blocking”，本質是循序的
- overhead 遠小於眞實的 thread

Event-driven APIs

```
PT_THREAD(smtp_protothread(void))
{
  PSOCK_BEGIN(s);

  PSOCK_READTO(s, '\n');

  if(strncmp(inputbuffer, “220”, 3) != 0) {
    PSOCK_CLOSE(s);
    PSOCK_EXIT(s);
  }

  PSOCK_SEND(s, “HELO ”, 5);
  PSOCK_SEND(s, hostname, strlen(hostname));
  PSOCK_SEND(s, “\r\n”, 2);

  PSOCK_READTO(s, '\n');

  if(inputbuffer[0] != '2') {
    PSOCK_CLOSE(s);
    PSOCK_EXIT(s);
  }
```

Proof-of-concept TCP/IP stacks

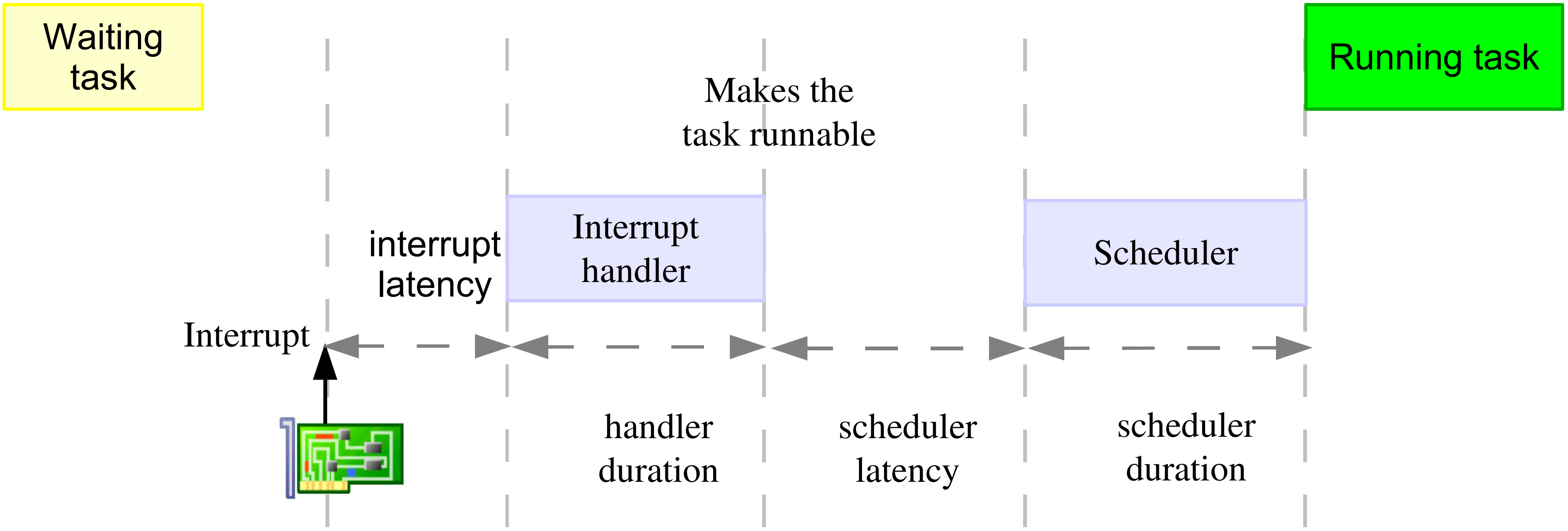
- phpstack – TCP/IP stack, webserver written in PHP
  `http://www.sics.se/~adam/phpstack/`
- miniweb – TCP/IP stack, webserver using 30 bytes of RAM
  `http://www.sics.se/~adam/miniweb/`

技術討論

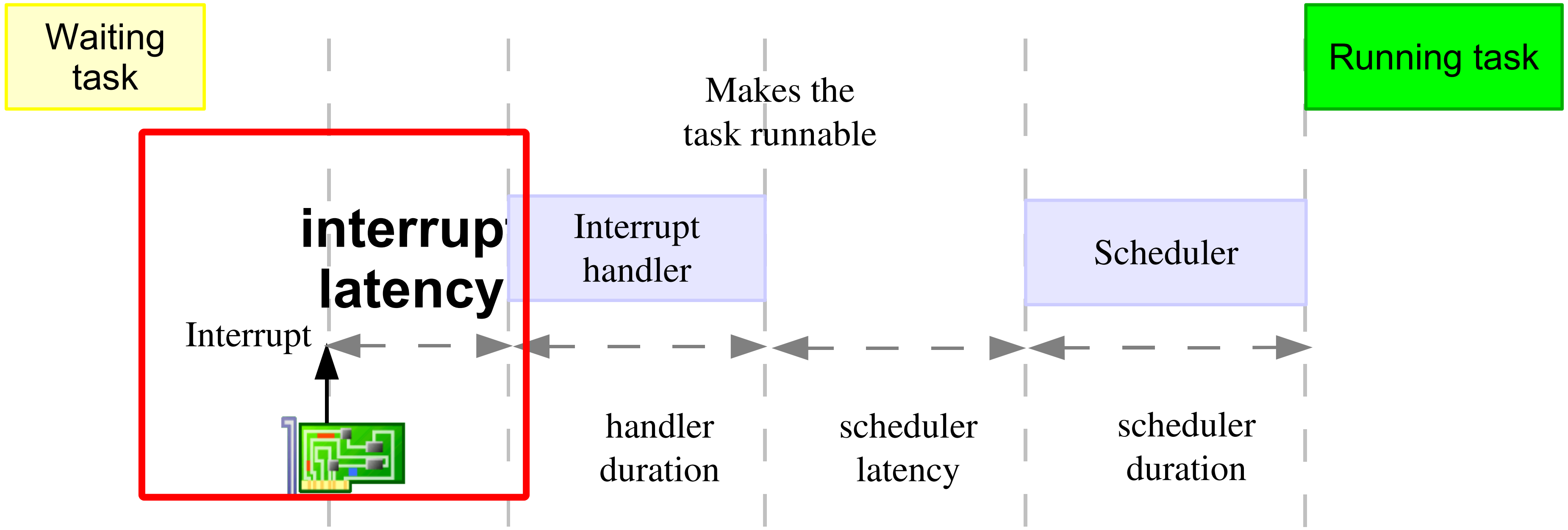
討論

- μIP 適用於低階的環境
- 中階或網路需求較高者，應用 lwIP 或 BSD socket
- Thread preemption

情境： Process 正等待 Device I/O(由中斷觸發) 的結束，方可繼續執行



kernel latency = **interrupt latency +**
handler duration +
scheduler latency +
scheduler duration

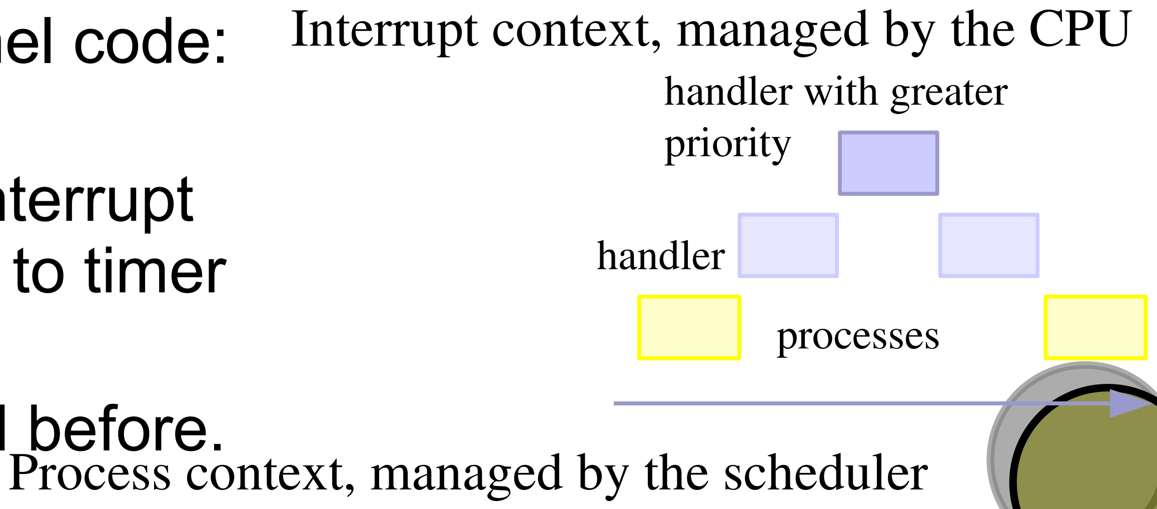


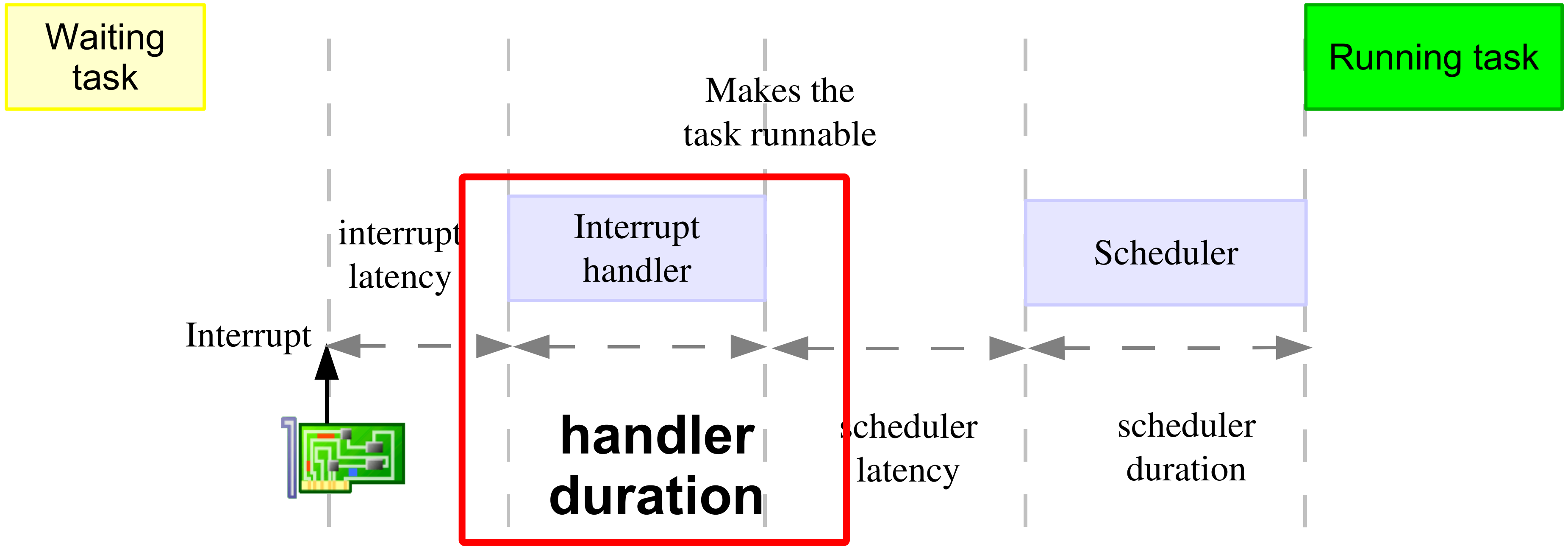
來源

Interrupts disabled by kernel code: spinlocks, driver code...

Bad driver using the fast interrupt mode (should be reserved to timer interrupts).

Other interrupts processed before.


Interrupt context, managed by the CPU
handler with greater priority
handler
processes
Process context, managed by the scheduler



執行 interrupt handler 的時間，在以下情況會更嚴重：

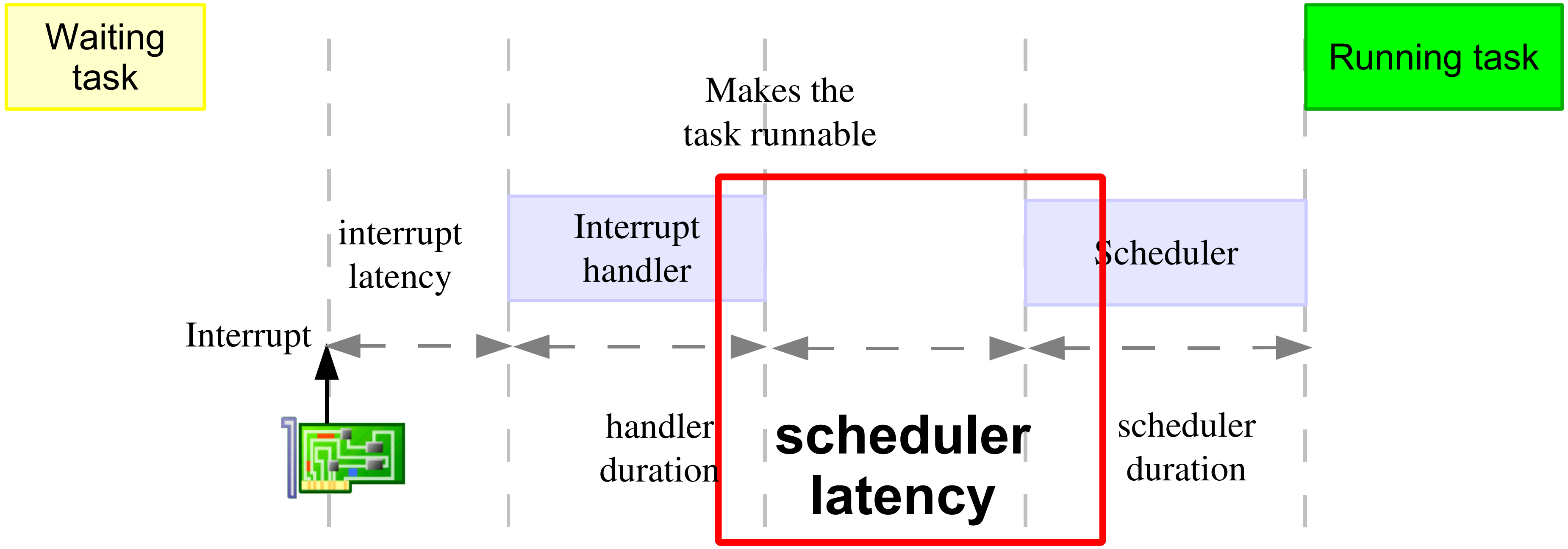
Preemption by other interrupts.

Interrupt handler with a softirq component

Interrupts disabled by kernel code

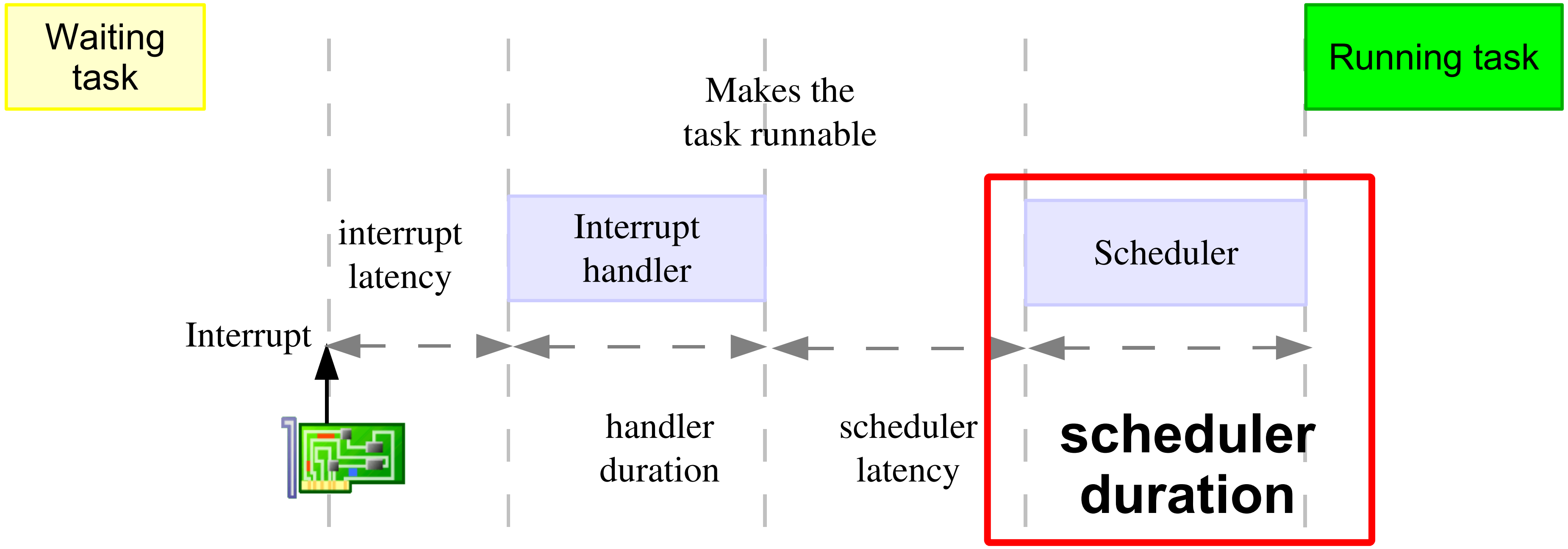
Bad driver using the fast interrupt mode

Other interrupts processed before.



用於 scheduler 的時間，原因：

- Kernel code not preemptible.
  - Need to wait for returning from interrupts or from system calls.
- Interrupts disabled for a long time in driver code.
  - Can cause a timer interrupt to be missed.



- 執行 scheduler 並切換到新的 task 所耗費的時間
  - 執行 scheduler 的時間為常數
  - context switching time
    - Restoring processor registers and stack for the new process.
    - Restoring the address space of the new process (except for threads sharing the same address space).

結論

- 無所不在的嵌入式系統
  - Invisible Computer
- 機電整合與自由軟體的機會
- 作中學