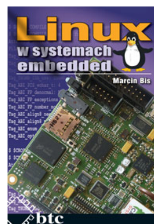
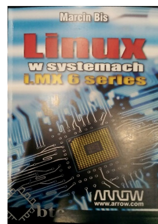
Jak Linux widzi sprzęt?

Marcin Bis

2024.10.27

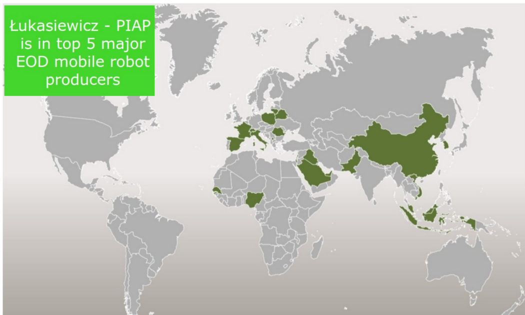
- Embedded Linux, **Yocto Project**, **Linux Kernel**, Real-Time Systems, C/C++
- `marcin@bis-linux.com`
- Szkolenia (`http://bis-linux.com`), wsparcie techniczne, projekty.





Sieć Badawcza Łukasiewicz - **Przemysłowy Instytut Automatyki i Pomiarów PIAP**
https://piap.lukasiewicz.gov.pl

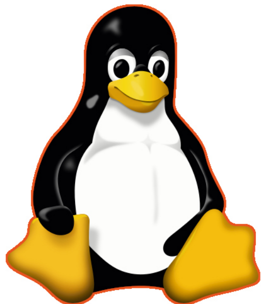
- marcin.bis@piap.lukasiewicz.gov.pl
- https://www.antiterrorism.eu

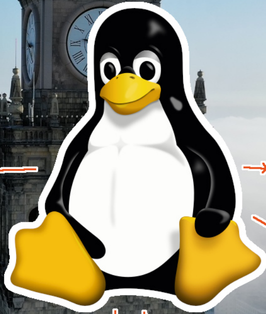


Przezentacja zawiera opinie i poglądy autora, które nie muszą być zgodne z opiniami i polityką organizacji, które reprezentuje.

Roboty są jak supersamochody…

. . . z Linuksem




Cloud
Cloud
Cloud
Cloud
(probably)
Embedded

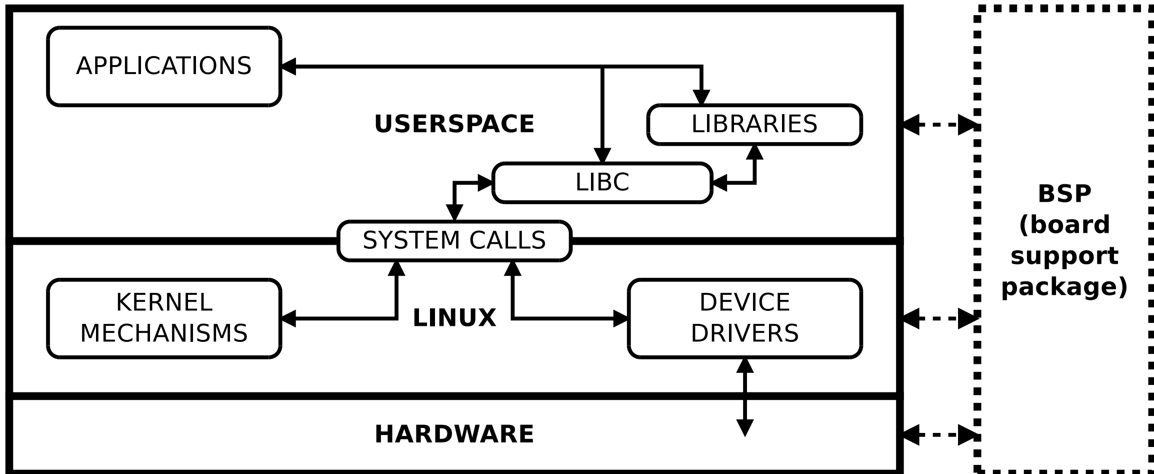
(Embedded) Linux - Architektura

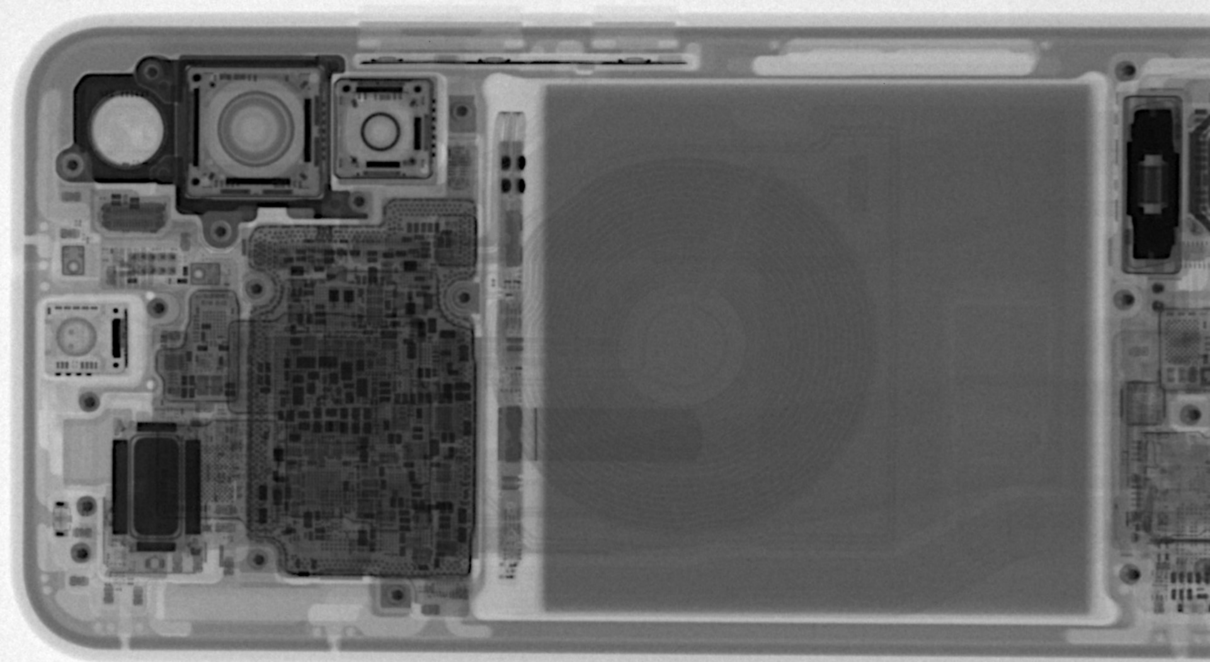
Architektura systemu

System operacyjny zbudowany jest z kilku warstw:

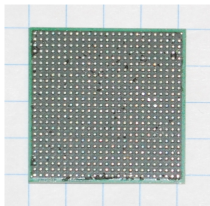
- `Przestrzeń użytkownika` - programy działające pod kontrolą jądra. Implementują logikę.
  - środowisko zgodne z POSIX (UNIX) - dystrybucja Linuksa.
  - inne środowisko - Android.
- `Linux` - jądro systemu
  - specjalny program, działa bezpośrednio na sprzęcie;
  - uruchamia programy - procesy działające w przestrzeni użytkownika;
  - zapewnia im abstrakcyjne sposoby dostępu do sprzętu (i współdzielenia go);
  - dostarcza mechanizmów.
- `Sprzęt` - na nim działa system.

Architektura systemu

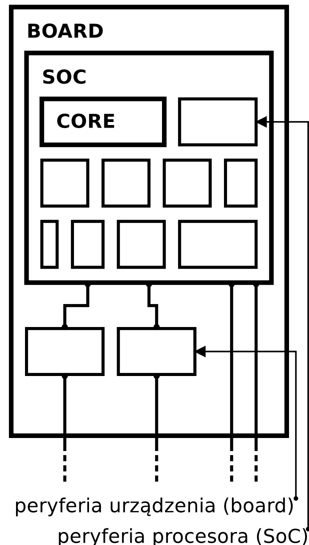




- CORE - zaprojektowany przez firmę ARM, określa architekturę i jej wariant.
- SoC - *System-on-chip* - zawiera rdzeń i inne urządzenia (*IP Core*). Ze względu na ograniczoną ilość wyprowadzeń - nie wszystkie urządzenia mogą działać równocześnie.

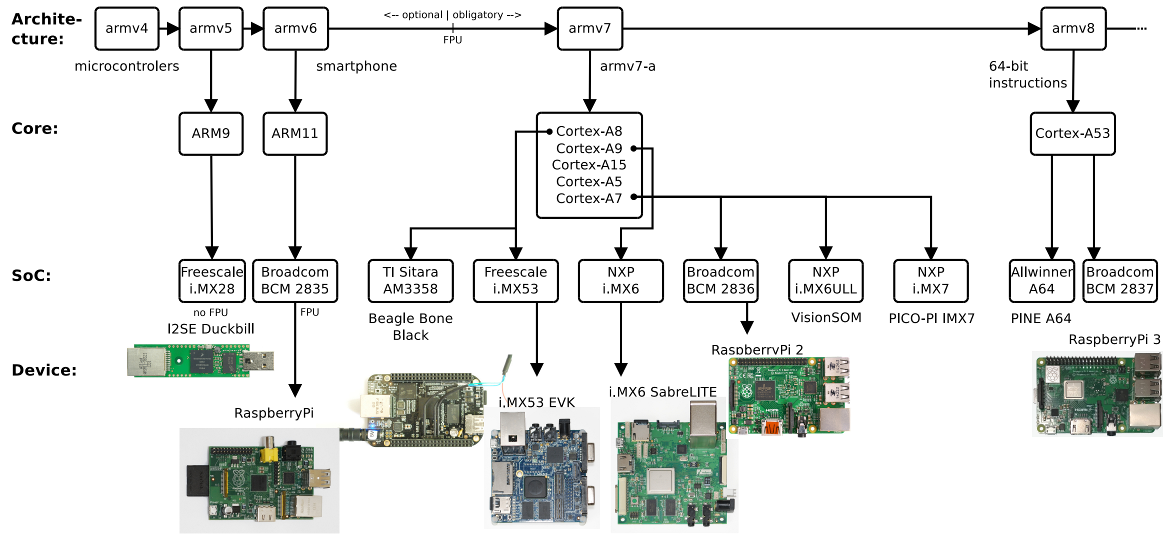


- BOARD - urządzenie zbudowane na bazie SoC-a, w którym wyprowadzono/wykorzystano niektóre jego peryferia i dodatkowe urządzenia. Może być zrealizowane jako SoM - *System-on-Module*.



- **ARM** is a family of reduced instruction set computing (RISC) architectures for computer processors, configured for various environments.
- British company ARM Holdings develops the architecture and licenses it to other companies, who design their own products that implement systems-on-chips (SoC) and systems-on-modules (SoM).
  It also designs cores that implement this instruction set and licenses these designs to a number of companies that incorporate those core designs into their own products.

1. `DEN0013D_cortex_a_series_PG.pdf`
   ARM Cortex-A Series Programmer's Guide
2. `DDI0500F_cortex_a53_r0p4_trm.pdf`
   ARM Cortex-A53 MPCore Processor Technical Reference Manual


Archite-cture:
armv4
armv5
armv6
<-- optional | obligatory -->
FPU
armv7
armv8
microcontrolers
smartphone
armv7-a
64-bit instructions
Core:
ARM9
ARM11
Cortex-A8
Cortex-A9
Cortex-A15
Cortex-A5
Cortex-A7
Cortex-A53
SoC:
Freescale i.MX28
Broadcom BCM 2835
TI Sitara AM3358
Freescale i.MX53
NXP i.MX6
Broadcom BCM 2836
NXP i.MX6ULL
NXP i.MX7
Allwinner A64
Broadcom BCM 2837
no FPU
FPU
I2SE Duckbill
Beagle Bone Black
VisionSOM
PICO-PI IMX7
PINE A64
RaspberryPi 3
RaspberryPi 2
Device:
i.MX53 EVK
i.MX6 SabreLITE
RaspberryPi

ARM Cortex-A Series Programmer's Guide

- Ch.3 *ARM Processor Modes and Registers*
  Privilege levels (PL0 - PL2) and processor modes.
- Ch.11 *Exception Handling*
  Types of exceptions; exception generatig instructions.
- Ch.9 *The Memory Management Unit*

Modes

2.2 Arc… 25
2.3 Pro… 30
2.4 Cor… 32
2.5 Ke… 38
3: ARM P… 39
3.1 Re… 44
4: Introd… 52
5: ARM/T… 65
6: Floatin… 84
7: Introd… 95
8: Caches 107
9: The M… 130
10: Mem… 149
11: Exce… 160
11.1 Ty… 162
11.1.1 … 165
11.2 E… 169
11.3 O… 171
11.3… 171
11.3… 171
11.3… 171
11.4 Li… 173
12: Interr… 174
13: Boot … 183
14: Porting 194
15: Appli… 207
16: Profili… 217
17: Opti… 224
18: Multi- 240

PL0 The privilege level of application software, that executes in User mode. Software executed in User mode is described as unprivileged software. This software cannot access some features of the architecture. In particular, it cannot change many of the configuration settings.

Software executing at PL0 can make only unprivileged memory accesses.

PL1 Software execution in all modes other than User mode and Hyp mode is at PL1. Normally, operating system software executes at PL1.

The PL1 modes refers to all the modes other than User mode and Hyp mode.

An Operating System is expected to execute across all PL1 modes and its applications executing in PL0 (User Mode).

PL2 Hyp mode is normally used by a hypervisor, that controls, and can switch between Guest Operating Systems that execute at PL1.

If Virtualization Extensions are implemented, a hypervisor will execute at PL2 (Hyp mode). A hypervisor will control and enable multiple Operating Systems to co-exist and execute on the same processor system.

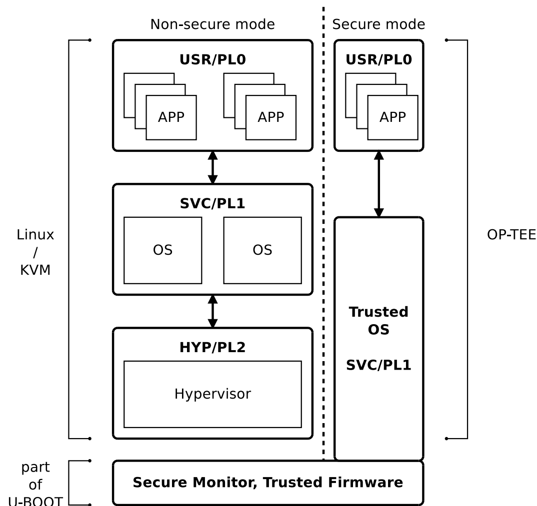
These privilege levels are separate from the TrustZone Secure and Normal (Non-secure) settings.

ARM CPU Modes

ARMv7 CPU boots in the highest privilege mode, in the secure SVC mode.

```
$ dmesg | grep SVC
CPU: All CPU(s) started in SVC mode.
```

Linux kernel cannot switch to non-secure mode itself (by design). It is role of the bootloader either to switch, or load secure firmware. Provided by U-BOOT or `https://github.com/ARM-software/arm-trusted-firmware`.



Exceptions

42 of 421

ARM Cortex-A Series Programmer's Guide
DEN0013D_cortex_a_series_PG.pdf

117,0%

2.2 Arc... 25
2.3 Pro... 30
2.4 Cor... 32
2.5 Ke... 38
3: ARM P... 39
3.1 Re... 44
4: Introd... 52
5: ARM/T... 65
6: Floatin... 84
7: Introd... 95
8: Caches 107
9: The M... 130
10: Mem... 149
11: Exce... 160
11.1 Ty... 162
11.1.1 ... 165
11.2 E... 169
11.3 O... 171
11.3... 171
11.3... 171
11.3... 171
11.4 Li... 173
12: Interr... 174
13: Boot ... 183
14: Porting 194
15: Appli... 207
16: Profili... 217
17: Opti... 224
18: Multi-... 240
19: Parall... 259
20: Powe... 272

Table 3-2 ARMv7 processor modes

Mode	Encoding	Function	Security state	Privilege level
User (USR)	10000	Unprivileged mode in which most applications run	Both	PL0
FIQ	10001	Entered on an FIQ interrupt exception	Both	PL1
IRQ	10010	Entered on an IRQ interrupt exception	Both	PL1
Supervisor (SVC)	10011	Entered on reset or when a Supervisor Call instruction (SVC) is executed	Both	PL1
Monitor (MON)	10110	Implemented with Security Extensions. See Chapter 21	Secure only	PL1
Abort (ABT)	10111	Entered on a memory access exception	Both	PL1
Hyp (HYP)	11010	Implemented with Virtualization Extensions. See Chapter 22	Non-secure	PL2
Undef (UND)	11011	Entered when an undefined instruction executed	Both	PL1
System (SYS)	11111	Privileged mode, sharing the register view with User mode	Both	PL1

A general purpose Operating System, such as Linux, and its applications, are expected to run in

Exceptions

2.2 Arc… 25
2.3 Pro… 30
2.4 Cor… 32
2.5 Ke… 38
3: ARM P… 39
3.1 Re… 44
3.1.… 46
3.1.… 46
3.1.… 47
3.1.… 50
4: Introd… 52
4.1 Co… 53
4.2 Th… 54
4.3 Int… 56
4.4 AR… 60
4.5 Int… 62
4.6 Ide… 63
4.7 Co… 64
5: ARM/T… 65
6: Floatin… 84
7: Introd… 95
8: Caches 107
9: The M… 130
10: Mem… 149
11: Exce… 160
11.1 Ty… 162
11.1.1 … 165
11.1.… 166
11.1 166

Table 11-1 Summary of exception behavior

Normal Vector offset	High vector address	Non-secure	Secure	Hypervisor[a]	Monitor
`0x0`	0xFFFF0000	Not used	Reset	Reset	Not used
`0x4`	0xFFFF0004	UNDEFINED instruction	UNDEFINED instruction	UNDEFINED instruction from Hyp mode.	Not used
`0x8`	0xFFFF0008	Supervisor Call	Supervisor Call	Secure Monitor Call	Secure Monitor Call
`0xC`	0xFFFF000C	Prefetch Abort	Prefetch Abort	Prefetch Abort from Hyp mode.	Prefetch Abort

Copyright © 2011 – 2013 ARM. All rights reserved.
Non-Confidential

Exceptions

2.2 Arc… 25
2.3 Pro… 30
2.4 Cor… 32
2.5 Ke… 38
3: ARM P… 39
3.1 Re… 44
3.1.… 46
3.1.… 46
3.1.… 47
3.1.… 50
4: Introd… 52
4.1 Co… 53
4.2 Th… 54
4.3 Int… 56
4.4 AR… 60
4.5 Int… 62
4.6 Ide… 63
4.7 Co… 64
5: ARM/T… 65
6: Floatin… 84
7: Introd… 95
8: Caches 107
9: The M… 130
10: Mem… 149
11: Exce… 160
11.1 Ty… 162
11.1.1 … 165
11.1.… 166
11.1 166

Table 11-1 Summary of exception behavior (continued)

Normal Vector offset	High vector address	Non-secure	Secure	Hypervisor[a]	Monitor
`0x10`	0xFFFF0010	Data Abort	Data Abort	Data Abort from Hyp mode,	Data Abort
`0x14`	0xFFFF0014	Not used	Not used	Hyp mode entry	Not used
`0x18`	0xFFFF0018	IRQ interrupt	IRQ interrupt	IRQ interrupt	IRQ interrupt
`0x1C`	0xFFFF001C	FIQ interrupt	FIQ interrupt	FIQ interrupt	FIQ interrupt

a. Hypervisor entry exception (described in Chapter 22 *Virtualization*) is available only in cores that support Virtualization Extensions and is unused in other cores.

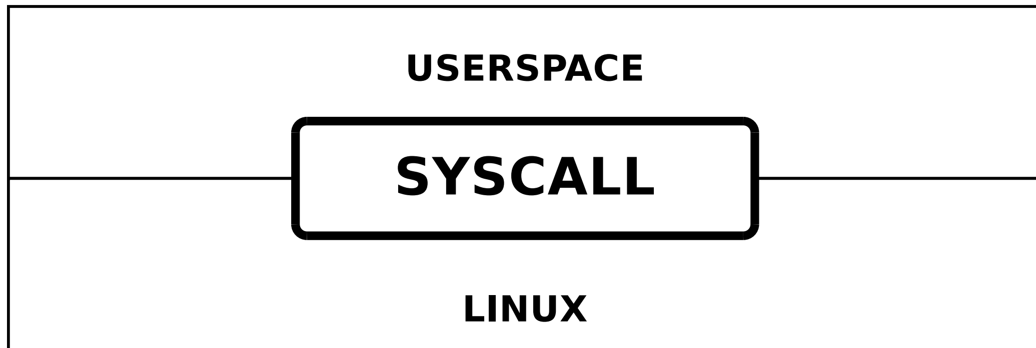
11.1.2 Exception mode summary

Table 11-2 lists the state of the interrupt disabling I and F bits of the CPSR on entering an exception handler.

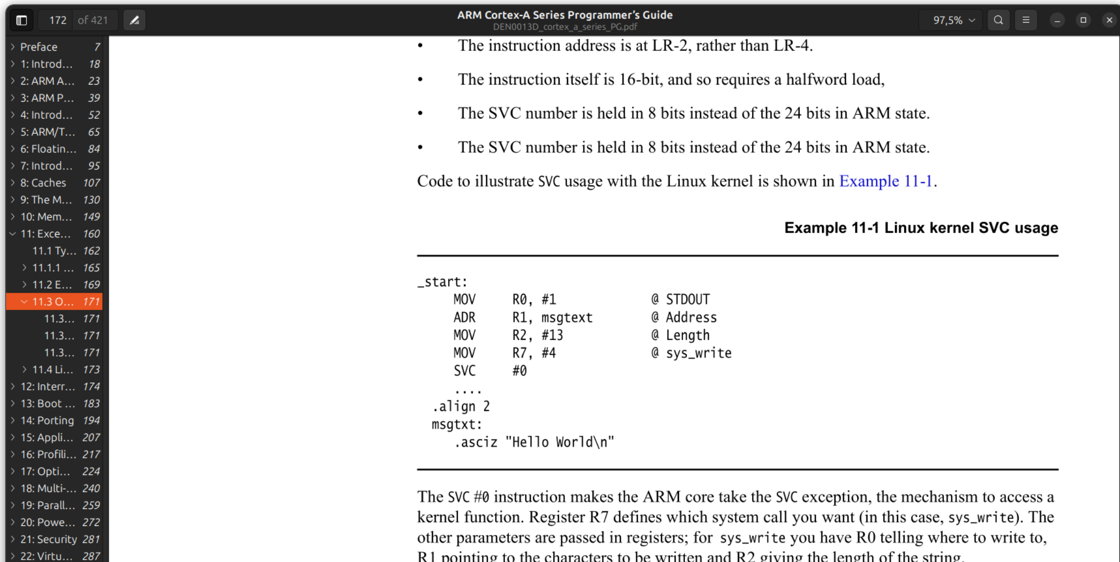
Table 11-2 CPSR behavior

Widok z góry

Wywołania systemowe



Syscall



strace

System calls

Standard C library encapsulates system calls provided by kernel.

- System call is not an ordinary function - it must be called in a special way (CPU enters service/superviser mode).
  So system calls are defined using special macros and has unique numbers.
- Libc needs kernel headers to compile.
  System call numbers are defined in `arch/arm/include/uapi/asm/unistd.h`.
- Kernel API and ABI for userspace rarely changes - new system calls are added.

Toolchain does not have to use headers from same version of kernel, the programs will be running.

- New kernels are backwards compatible.
- Library compensates lack of some calls while running on older kernel versions.

Documentation in kernel sources: `Documentation/process/adding-syscalls.rst`

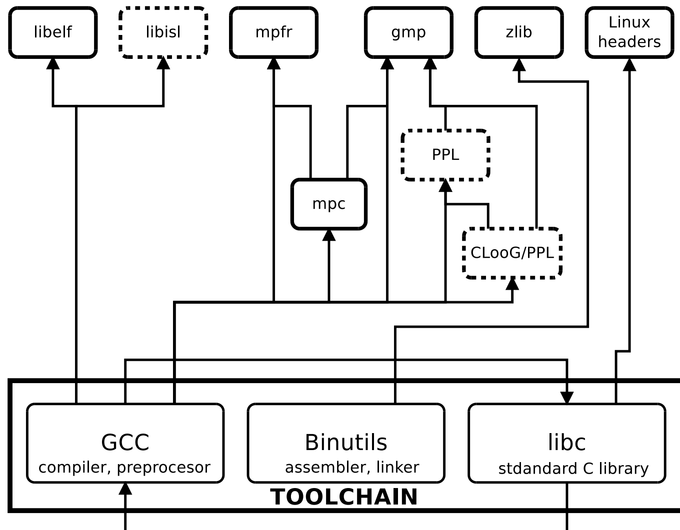
Syscall

unistd.h « asm « uapi « include «

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/arm/include/uapi/asm/unistd.h?h=v4.0

```
28  #define __NR_restart_syscall          (__NR_SYSCALL_BASE+  0)
29  #define __NR_exit                     (__NR_SYSCALL_BASE+  1)
30  #define __NR_fork                     (__NR_SYSCALL_BASE+  2)
31  #define __NR_read                     (__NR_SYSCALL_BASE+  3)
32  #define __NR_write                    (__NR_SYSCALL_BASE+  4)
33  #define __NR_open                     (__NR_SYSCALL_BASE+  5)
34  #define __NR_close                    (__NR_SYSCALL_BASE+  6)
35                                        /* 7 was sys_waitpid */
36  #define __NR_creat                    (__NR_SYSCALL_BASE+  8)
37  #define __NR_link                     (__NR_SYSCALL_BASE+  9)
38  #define __NR_unlink                   (__NR_SYSCALL_BASE+ 10)
39  #define __NR_execve                   (__NR_SYSCALL_BASE+ 11)
40  #define __NR_chdir                    (__NR_SYSCALL_BASE+ 12)
41  #define __NR_time                     (__NR_SYSCALL_BASE+ 13)
42  #define __NR_mknod                    (__NR_SYSCALL_BASE+ 14)
43  #define __NR_chmod                    (__NR_SYSCALL_BASE+ 15)
44  #define __NR_lchown                   (__NR_SYSCALL_BASE+ 16)
45                                        /* 17 was sys_break */
```

Toolchain - building blocks



file

- `file` - shows file information based in its contents.

```
$ file hello
```

architektura

```
hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux-armhf.so.3, for GNU/Linux 3.2.0, BuildID[sha1]=0d..c, stripped

hello: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynamically linked,
interpreter /lib/ld-linux-aarch64.so.1, for GNU/Linux 3.7.0, BuildID[sha1]=83..7, stripped
```

interpreter minimalna wersja jądra symbole debuggera

- ELF dynamic section - shared libraries needed by programme:

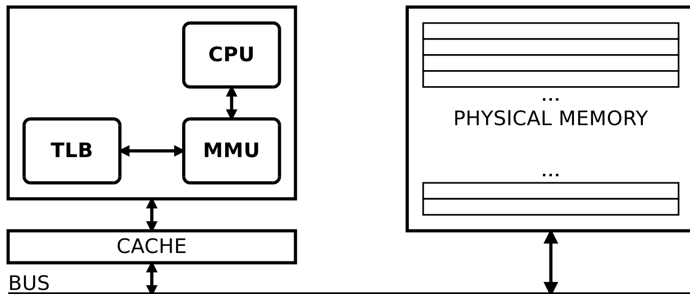
```
$ readelf -d hello | grep NEEDED
```

For a native platform, `ldd` may be used (it works recursively - shows dependencies of libraries)

- Sekcja atrybutów dla ARM:

```
$ readelf -a sqrt
Attribute Section: aeabi | Tag_CPU_name: "7-A" | Tag_THUMB_ISA_use: Thumb-2
File Attributes          | Tag_CPU_arch: v7    | Tag_FP_arch: VFPv3
```

MMU - ARMv7



- MMU kontroluje dostęp do fizycznej przestrzeni adresowej i znajdującej się w niej pamięci.
- ARM Architecture Reference Manual - Virtual Memory System Architecture version 7 (VMSAv7)
- MMU obsługiwany jest jako koprocesor CP15.

```
MRC p15, 0, R1, c1, C0, 0          ;Read control register
ORR R1, #0x1                       ;Set M bit
MCR p15, 0,R1,C1, C0,0             ;Write control register and enable MMU
```

TLB

Co się dzieje kiedy procesor generuje żądanie do adresu w pamięci?

MMU: przeszukuje TLB - `Translation Lookaside Buffer`

- Szuka adresu, aktualnego ASID (stanu) i uprawnień (czy dozwolone) w `micro TLB` (1 cykl) - odpowiednio: dla instrukcji lub danych;
  micro TLB - 32 lub 64 wpisy (zależy od sprzętu) dla instrukcji i 32 dla danych.
- Szuka w głównym TLB (co kosztuje zmienną ilość cykli);
  main TLB - 64 - 512 wpisów.
- Trawersuje tabele z opisem mapowania znajdujące się w pamięci RAM.

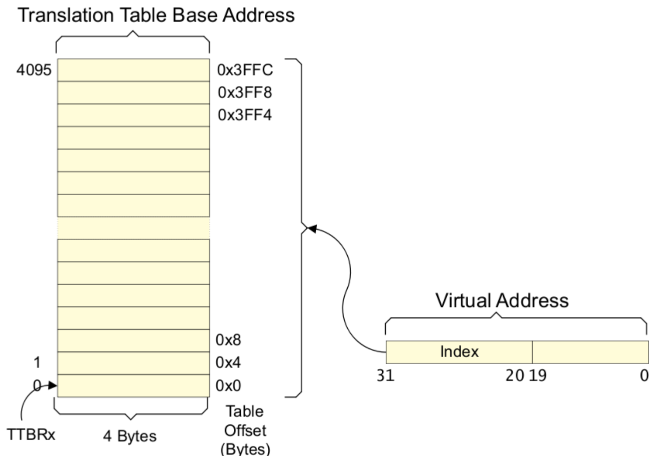
Tabele - struktury danych definiowane przez architekturę (sprzęt).

- Dwa poziomy tablic: L1 i L2
- Trawersowanie tabel jest dość kosztowne:
  dostęp do adresu w pamięci może powodować 3 odwołania do niej: do L1, L2 i właściwego adresu.
- Dlatego MMU buforuje zapytania w TLB.

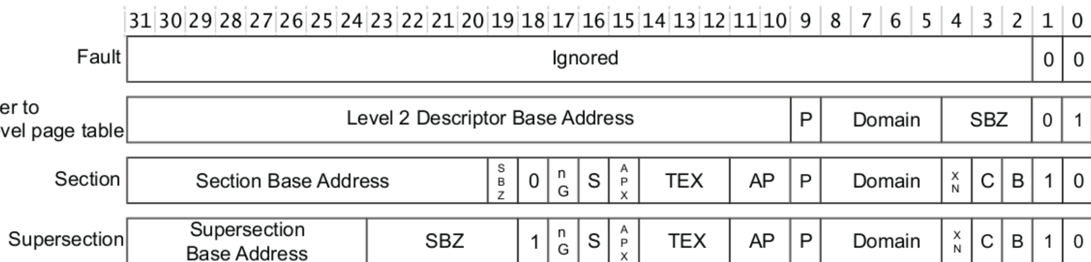
TLB - L1

- Adres L1 jest przechowywany w rejestrze `c2 CP15` (align: 16kB).
- Przestrzeń adresowa (2^{32} = 4GB) jest mapowana w tabeli L1 na 4096 równych sekcji (po 1MB każda).
- MMU używa początkowych 12 bitów adresu witrualnego jako indeksu w L1.
- Każdy wpis w L1 przechowuje adresy i flagi:
  - adres sekcji - 1MB
  - adres supersekcji - 16MB
  - adres struktury L2,
  - nic - przy próbie dosępu generowany jest wyjątek.

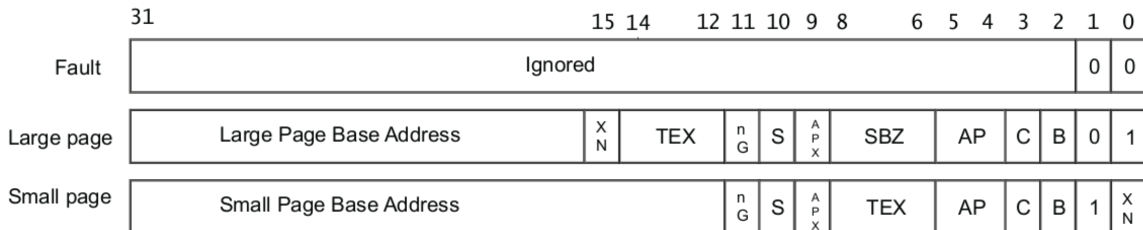
`Cortex-A Series Programmer's Guide` p. 138



L1:

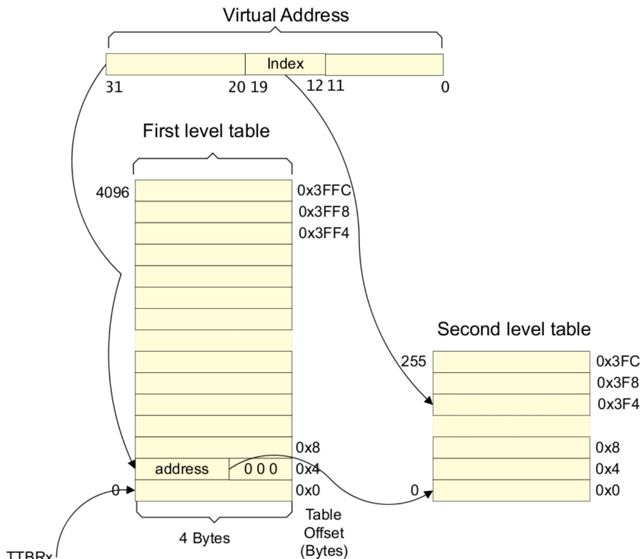


L2:



TLB - L2

- 256 4-ro bajtowych wpisów.
- Każdy wpis mapuje:
  - 4kB pamięci (używane jako strona w Linuksie).
  - 64kB,
  - nic (błąd strony)



MMU ARMv8

ARMv8 **AArch32** - same as ARMv7
ARMv8 **AArch64**:

- Maximum supported physical address size - 40 bits.
- 3 or 4 levels of TLB tables.

```
AArch64 Linux memory layout with 4KB pages + 3 levels:
Start                   End                     Size         Use
-----------------------------------------------------------------------
0000000000000000        0000007fffffffff        512GB        user
ffffff8000000000        ffffffffffffffff        512GB        kernel

AArch64 Linux memory layout with 4KB pages + 4 levels:
Start                   End                     Size         Use
-----------------------------------------------------------------------
0000000000000000        0000ffffffffffff        256TB        user
ffff000000000000        ffffffffffffffff        256TB        kernel
```

MMU - implementaja w Linuksie

Kod niezależny od sprzętu.

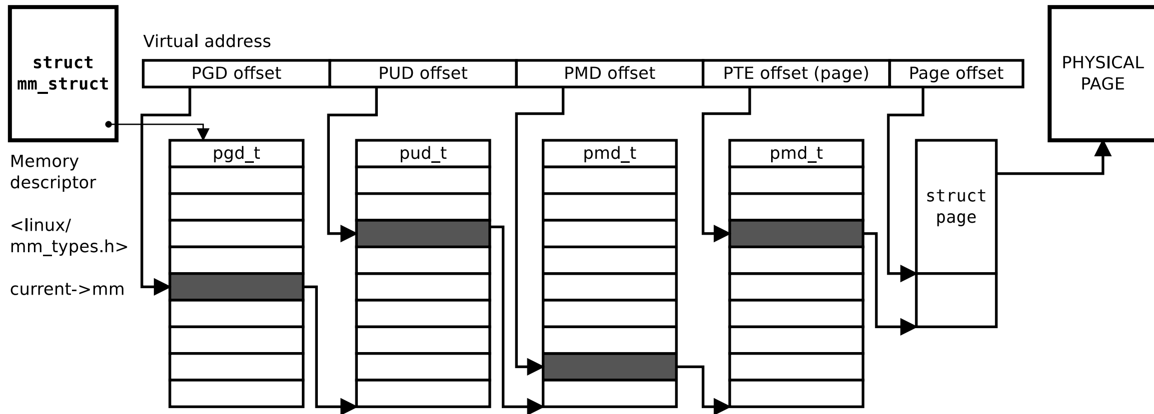
- Procesy są opisywane przez `struct task_struct`
- Pole `mm` zawiera wskaźnik do `struct mm_struct`.
- `struct mm_struct` opisuje **czteropoziomowy** schemat stronicowania (od 2.6.11, wcześniej 3 poziomy).
  Każdy adres wirtualny jest dzielony na **5** pól (4 adresują stronę, ostatni offset w ramach strony).
  1. *(PDG) Page Global Directory*
  2. *(PUD) Page Unit Directory*
  3. *(PMD) Page Middle Directory*
  4. *(PTE) Page Table Entry*

Poza tym

- Żadna część jądra nie podlega stronicowaniu.
- To samo - struktury danych używane do zarządzania mapowaniem pamięci.
- Niektóre architektury upraszczają schemat (x86, ARMv7 - 2 poziomy).
  Inne emulują. . .

Adresowanie - Linux

Każdy proces ma wskaźnik do swojej własnej PGD (`current->mm_struct->pgd`).



W momencie wystąpienia wyjątku MMU (błędu strony) - struktura jest trawersowana i wypełniana jest właściwa tablica TLB - (zależna od architektury).

`PTE` - przechowuje informacje o tym, czy strona została zeswapowana itp.

Przykłady funkcji - Linux

Funkcje operujące na TLB:

- Wszystkie procesory, po modyfikacji globalnej tablicy (np. wywołaniu vfree):

```
void flush_tlb_all(void)
```

- Tylko userspace (do PAGE_OFFSET):

```
void flush_tlb_mm(struct mm_struct *mm)
```

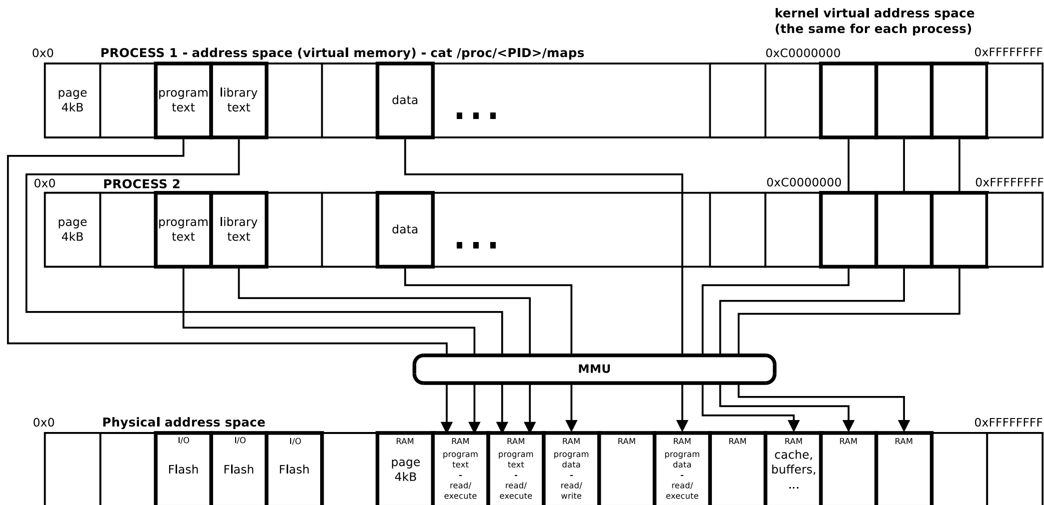
- Tylko userspace (podany zakres):

```
void flush_tlb_range(struct mm_struct *mm,
    unsigned long start, unsigned long end)
```

- Jedna strona:

```
void flush_tlb_page(struct vm_area_struct *vma, unsigned long addr)
```

Od 2.6.25 program w przestrzeni użytkownika może dostać się do swoich map przez
/proc/<PID>/pagemap>
Documentation/vm/pagemap.txt


kernel virtual address space
(the same for each process)
0x0
PROCESS 1 - address space (virtual memory) - cat /proc/<PID>/maps
0xC0000000
0xFFFFFFFF
page 4kB
program text
library text
data
. . .
0x0
PROCESS 2
0xC0000000
0xFFFFFFFF
page 4kB
program text
library text
data
. . .
MMU
0x0
Physical address space
0xFFFFFFFF
I/O Flash
I/O Flash
I/O Flash
RAM page 4kB
RAM program text - read/ execute
RAM program text - read/ execute
RAM program data - read/ write
RAM
RAM program data - read/ execute
RAM
RAM cache, buffers, ...
RAM
RAM

Process virtual address space - shell (ARM 32bit)

- `$ cat /proc/$$/maps`

```
address           perms offset   dev   inode pathname
00008000-000c5000 r-xp 00000000 b3:03 1891 /bin/bash
000cc000-000d1000 rw-p 000bc000 b3:03 1891 /bin/bash
000d1000-000d6000 rw-p 00000000 00:00 0
00142000-0017b000 rw-p 00000000 00:00 0    [heap]
40019000-40036000 r-xp 00000000 b3:03 5575
                                   /lib/ld-2.11.2.so
400c2000-400c4000 r-xp 00000000 b3:03 5552
/* ... */                       /lib/libdl-2.11.2.so
be8bb000-be8dc000 rw-p 00000000 00:00 0    [stack]
ffff0000-ffff1000 r-xp 00000000 00:00 0    [vectors]
```

- Detailed data can be read from `smaps`.
- Trying to access unmapped area - causes *Segmentation fault*.
- This is only mapping, data may not be here, kernel loads them on first access (on `page_fault`).
- Some functions modifying map: `brk()`, `mmap()`, `unmap()`, `malloc()`, `realloc()`.

Process virtual address space - explained

- **address** - virtual addres the mapping occupies
- **perms** - permissions: **r**ead, **w**rite, e**x**ecutable, **s**hared, **p**rivate (copy-on-write).
- **offest** - a file offset,
- **dev** - device (major:minor number),
- **inode** - inode number on a device,
- **path** - a file backing the mapping (if applicable). Some special entries ar also shown:
  - `[stack]` - initial process (main thread) stack,
  - `[stack:<tid>]` - a thread stack (thread id),
  - `[vdso]` - Virtual Dynamically Linked Shared Object,
  - `[heap]` - the processe's heap.

If file is backing the mapping -path is shown and offset points to specific ELF section (see `readfelf -l`).

- vDSO mechanism is an optimization provided by the kernel to reduce the cost of certain frequently used system calls.
- It is a small region of kernel-provided memory that is normally mapped into the address space of every user-space process;
- it contains implementations of system calls that operate on a user-space context.
- That allows the caller to avoid the cost of a context switch into kernel mode.

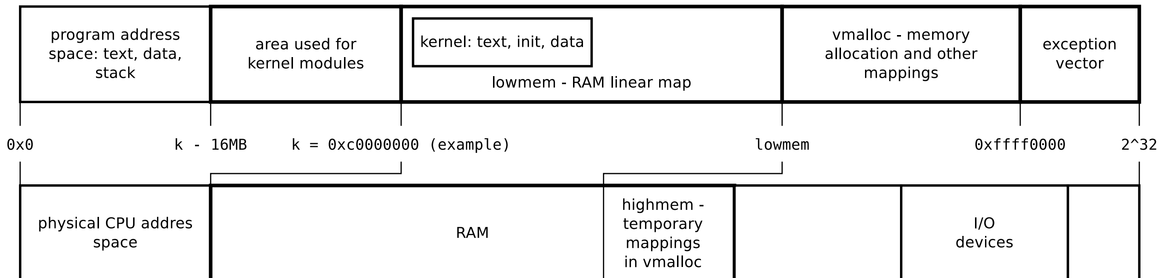
Eg. used for implementing system calls related to timekeeping (gettimeofday()).

Kernel virtual memory layout - ARM

Printed in kernel message buffer during boot:

```
Memory: 128MB = 128MB total
Memory: 86752k/86752k available, 44320k reserved, 0K highmem
Virtual kernel memory layout:
    vector  : 0xffff0000 - 0xffff1000   (   4 kB)
    fixmap  : 0xfff00000 - 0xfffe0000   ( 896 kB)
    DMA     : 0xffc00000 - 0xffe00000   (   2 MB)
    vmalloc : 0xc8800000 - 0xf8000000   ( 760 MB)
    lowmem  : 0xc0000000 - 0xc8000000   ( 128 MB)
    modules : 0xbf000000 - 0xc0000000   (  16 MB)
      .init : 0xc0008000 - 0xc0028000   ( 128 kB)
      .text : 0xc0028000 - 0xc033c380   (3153 kB)
      .data : 0xc033e000 - 0xc035af40   ( 116 kB)
```

Kernel virtual address space



Widok z dołu

```
$ cat /proc/ioports
0000-03af : PCI Bus 0000:00
  0000-001f : dma1
  0020-0021 : pic1
  0040-0043 : timer0
  0050-0053 : timer1
  0060-0060 : keyboard
  0064-0064 : keyboard
  0070-0071 : rtc0
  0080-008f : dma page reg
  00c0-00df : dma2
  00f0-00ff : fpu
03b0-03df : PCI Bus 0000:00
...
```

x86:

- Urządzenia PCI widoczne jako PIO i MMIO.

```
$ cat /proc/iomem
00000000-00000fff : reserved
00001000-0009fbff : System RAM
0009fc00-0009ffff : reserved
000c0000-000dffff : PCI Bus 0000:00
  000c0000-000cf9ff : Video ROM
000e0000-000fffff : reserved
  000f0000-000fffff : System ROM
00100000-9e4a4fff : System RAM
  01000000-01735e33 : Kernel code
  01735e34-01d1e37f : Kernel data
  01e77000-01fdffff : Kernel bss
  94000000-97ffffff : GART
9e4e3000-9e4effff : ACPI Tables
...
c0000000-ffffffff : PCI Bus 0000:00
  c0000000-cfffffff : 0000:00:01.0
  d0000000-d00fffff : PCI Bus 0000:01
    d0000000-d0003fff : 0000:01:00.0
      d0000000-d0003fff : r8169
```

Fizyczna przestrzeń adresowa

MMIO (*Memory-Mapped I/O*):

- Dostęp bezpośrednio w przestrzeni adresowej procesora.
- Ta sama magistrala dla dostępu do urządzeń i do pamięci (dotyczy to wewnętrznej budowy układu - magistrala adresowa nie jest bezpośrednio wyprowadzana z SoC-a).
- Do dostępu do urządzeń służą te same instrukcje, co w przypadku pamięci RAM (ARM: `ldr`, `str`).

PIO (*Port I/O*/*Isolated I/O*):

- Różne przestrzenie adresowe dla pamięci i urządzeń (np. adresy 16-bitowe w PC).
- Oddzielna magistrala - więcej wyprowadzeń (więcej miejsca na krzemie, większe zużycie energii).
- Dedykowane instrukcje assemblera (x86: `IN`, `OUT`).

W ARM stosowane jest przede wszystkim **MMIO**.

- Układ przestrzeni adresowej określany jest na etapie produkcji SoC.
- Dokumentacja dla i.MX6 [RM, 2.2].

175 of 4127

i.MX 6ULL Applications Processor Reference Manual
IMX6ULLRM.pdf

91,5%

171

172

173

174

Table 2-1. System memory map

Start address	End address	Size	Description
8000_0000	FFFF_FFFF	2048 MB	MMDC—x16 DDR Controller.
7000_0000	7FFF_FFFF	256 MB	Reserved
6000_0000	6FFF_FFFF	256 MB	QSPI1 Memory
5800_0000	5FFF_FFFF	128 MB	EIM Aliased
5000_0000	57FF_FFFF	128 MB	EIM (NOR/SRAM)
1000_0000	4FFF_FFFF	1024 MB	Reserved
0E00_0000	0FFF_FFFF	32 MB	Reserved
0C00_0000	0DFF_FFFF	32 MB	QSPI1 Rx Buffer
0900_0000	0BFF_FFFF	48 MB	Reserved
0800_0000	08FF_FFFF	16 MB	Reserved
02C0_0000	07FF_FFFF	84 MB	Reserved
0230_0000	02BF_FFFF	9 MB	Reserved
0220_0000	022F_FFFF	1 MB	Table 2-4 AIPS-3. See IP listing on the separate map.
0210_0000	021F_FFFF	1 MB	Table 2-3 AIPS-2. See the IP listing on the separate map.
0200_0000	020F_FFFF	1 MB	Table 2-2 AIPS-1. See the IP listing on the separate map.
0181_0000	01FF_FFFF	8128 KB	Reserved
0180_C000	0180_FFFF	16 KB	Reserved

MMIO

176 of 4127 — i.MX 6ULL Applications Processor Reference Manual — IMX6ULLRM.pdf — 91,5%

Table 2-1. System memory map (continued)

Start address	End address	Size	Description
0180_8000	0180_BFFF	16 KB	BCH
0180_6000	0180_7FFF	8 KB	GPMI
0180_4000	0180_5FFF	32 KB	APBH DMA
0180_0000	0180_3FFF	16 KB	Reserved
0120_0000	017F_FFFF	6 MB	Reserved
0110_0000	011F_FFFF	1 MB	Reserved
0100_0000	010F_FFFF	1 MB	Reserved
00F0_0000	00FF_FFFF	1 MB	Reserved
00E0_0000	00EF_FFFF	1 MB	(per_m) configuration port
00D0_0000	00DF_FFFF	1 MB	(cpu) configuration port
00C0_0000	00CF_FFFF	1 MB	GPV_1 PL301
00B0_0000	00BF_FFFF	1 MB	GPV_0 PL301 configuration port
00A0_8000	00AF_FFFF	992 KB	Reserved
00A0_0000	00A0_7FFF	32 KB	ARM Peripherals: GIC400 Only visible to ARM core(s)
009C_0000	009F_FFFF	256 KB	Reserved
0098_0000	009B_FFFF	256 KB	Reserved
0092_0000	0097_FFFF	384 KB	OCRAM aliased
0090_0000	0091_FFFF	128 KB	OCRAM 128 KB

MMIO

i.MX 6ULL Applications Processor Reference Manual
IMX6ULLRM.pdf

177 of 4127

Chapter 2 Memory Maps

Table 2-2. AIPS-1 memory map (continued)

Start Address	End Address	Region	NIC Port	Size
020D_8000	020D_BFFF		SRC	16 KB
020D_4000	020D_7FFF		EPIT2	16 KB
020D_0000	020D_3FFF		EPIT1	16 KB
020C_C000	020C_FFFF		SNVS_HP	16 KB
020C_8000	020C_8FFF		ANALOG_DIG	16 KB
020C_4000	020C_7FFF		CCM	16 KB
020C_0000	020C_3FFF		WDOG2	16 KB
020B_C000	020B_FFFF		WDOG1	16 KB
020B_8000	020B_BFFF		KPP	16 KB
020B_4000	020B_7FFF		ENET2	16 KB
020B_0000	020B_3FFF		SNVS_LP	16 KB
020A_C000	020A_FFFF		GPIO5	16 KB
020A_8000	020A_BFFF		GPIO4	16 KB
020A_4000	020A_7FFF		GPIO3	16 KB
020A_0000	020A_3FFF		GPIO2	16 KB
0209_C000	0209_FFFF		GPIO1	16 KB
0209_8000	0209_BFFF		GPT1	16 KB
0209_4000	0209_7FFF		CAN2	16 KB

Alokacja zasobów

Do alokacji (MMIO i PIO) służy struktura `struct resource`, rejestrowana przy pomocy:

```
int request_resource(struct resource *root,
                     struct resource *new);
```

Informacje o urządzeniach pochodzą z:

- **x86**
  - określone w specyfikacji architektury, przykład: `drivers/char/rtc.c`:
    `935: r = request_region(RTC_PORT(0), size, "rtc");`
  - odczytane przez odpowiednie sterowniki (BIOS, tablice ACPI, kontroler PCI)
- **ARM**
  - kodu C (inicjalizacja platformy) - starsze wersje jądra
  - Device-Tree - nowsze

W obu przypadkach są to dane przepisane z dokumentacji SoC-a.

MMIO - dostęp do urządzeń

- W kodzie sterownika nie można posłużyć się bezpośrednim adresem rejestru urządzenia
- konieczne jest zaprogramowanie odpowiedniego mapowania w MMU:

```
#include <linux/io.h>
void __iomem *ioremap(phys_addr_t offset, unsigned long size)
```

  Funkcja przyjmuje adres fizyczny i rozmiar żądanego obszaru, zwraca wskaźnik do pamięci (adres wirtualny) lub `NULL` jeżeli nie udało się uzyskać dostępu (należy sprawdzać wynik).

- czyszczenie mapowania:

```
void iounmap(void *address);
```

- wariant pozwalający powiązać zasób ze sterownikiem:

```
void __iomem *devm_ioremap(struct device *dev,
                           resource_size_t offset,
                           resource_size_t size);
void devm_iounmap(struct device *dev, void __iomem *addr);
```

- Wariant `ioremap_nocache` wyłącza cache dla podanego obszaru.

Bezpieczny dostęp do pamięci I/O

- Aby dostać się do rejestru urządzenia, wystarczy wykonać dereferencję wskaźnika uzyskanego przez `ioremap`. Nie jest to jednak bezpieczne we wszystkich architekturach (cache, synchronizacja).
- Jądro dostarcza specjalnych funkcji:

```
unsigned int ioread8(void *addr);
void iowrite8(u8 value, void *addr);
```

- Inne warianty: `ioread16()`, `ioread32()`, `iowrite16()`, `iowrite32()`
- Zmiana kolejności bajtów w słowie: `ioread16be()`, `ioread32be()`, `iowrite16be()`, `owrite32be()`.
- Więcej wartości na raz: `ioread8_rep()`, `iowrite8_rep()`
- Inne przydatne funkcje: `memset_io()`, `memcpy_fromio()`, `memcpy_toio()`

Funkcje wymuszają synchronizacje danych na poziomie cache-u procesora.

/dev/mem

- Plik urządzenia `/dev/mem` jest używany do udostępnienia zawartości fizycznej przestrzeni adresowej.
- Proces:
  1. otwiera `/dev/mem` (`open()`),
  2. mapuje fragment pliku do swojej przestrzeni adresowej (`mmap()`) - offset w pliku odpowiada adresowi fizycznemu,
  3. czyta lub pisze z/do uzyskanego w ten sposób adresu.
- Z `/dev/mem`, korzystają aplikacje takie jak X.org - do implementacji sterowników w przestrzeni użytkownika.
- Busybox zawiera program **devmem**. Dostępny jest też samodzielny **devmem2**.
- Mechanizm nie pozwala tylko na rejestrowanie procedur obsługi przerwań (jest to możliwe przy zastosowaniu **UIO** - *Userspace I/O*).
- Ze względów bezpieczeństwa, zabroniony jest dostęp do adresów określających pamięć RAM, chyba że skonfigurowano inaczej (`CONFIG_STRICT_DEVMEM`).
- Od 4.0 sterownik jest opcjonalny (`CONFIG_DEVMEM`).

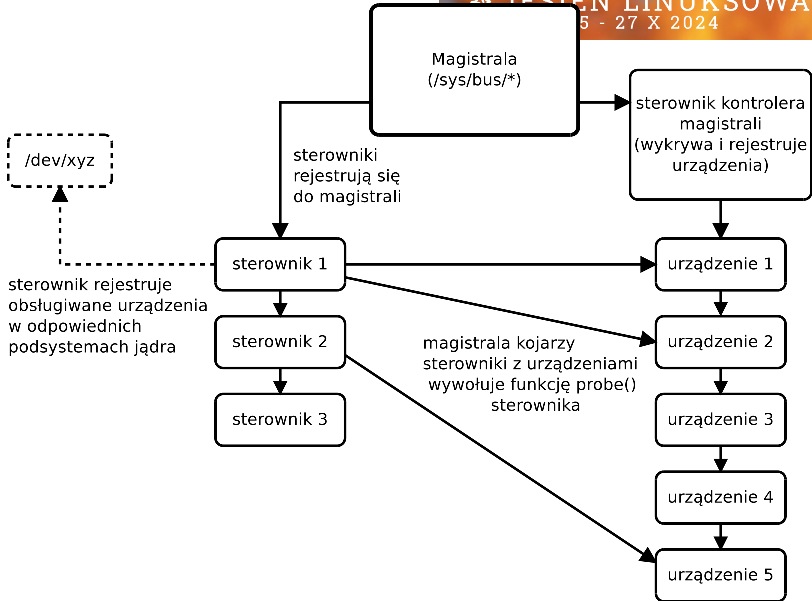
W środku

Model sterowników urządzeń

- Wprowadzony w jądrach serii 2.6.
- Wprowadza uniwersalny sposób organizacji urządzeń w magistrale.
- W jednym miejscu zbiera funkcje (callback-i), które pozwalają na unifikację sposobu tworzenia sterowników, kojarzenia ich z urządzeniami i przekazywania im informacji o zasobach.
- Działanie modelu odpowiada działaniu współczesnych urządzeń. Pozwala: śledzić odwołania do urządzeń, zarządzać energią, reagować na zmianę stanu - wykrycie lub usunięcie urządzenia z systemu.

Magistrala - sterownik - urządzenie

- W systemie jest tylko jedna (abstrakcyjna) magistrala danego typu: `PCI`, `USB`, `MMC`, `ISA`, `SPI`, `AMBA`, `1-wire`.
  - Magistrale znacznie różnią się funkcjonalnością: niektóre z nich posiadają (fizyczne) kontrolery zdolne do wykrywania urządzeń (`PCI`, `USB`, `1-wire`) i przydzielania im zasobów (`PCI`), inne są deklaratywne (`I2C`, `SPI`).
  - Dla urządzeń podłączonych bezpośrednio do procesora (kontroler klawiatury w x86, prawie wszystkie w ARM) przewidziana jest magistrala **platform**.
- **Sterownik magistrali** rejestruje magistralę w systemie i. . .
  - umożliwia rejestrowanie się **sterownikom kontrolerów** - zajmują się wykrywaniem urządzeń podłączonych do magistrali, konfigurowaniem ich zasobów i rejestrowaniem ich w magistrali.
    W przypadku magistrali platform - sterownik kontrolera nie ma możliwości wykrycia urządzeń - są one zadeklarowane:
    - w kodzie C (stary sposób),
    - w pliku Device-Tree ładowanym razem z jądrem.
  - umożliwia rejestrowanie się **sterownikom urządzeń**
  - kojarzy ze sobą sterowniki i urządzenia.


Magistrala
(/sys/bus/*)
sterownik kontrolera magistrali (wykrywa i rejestruje urządzenia)
/dev/xyz
sterowniki rejestrują się do magistrali
sterownik 1
sterownik 2
sterownik 3
urządzenie 1
urządzenie 2
urządzenie 3
urządzenie 4
urządzenie 5
sterownik rejestruje obsługiwane urządzenia w odpowiednich podsystemach jądra
magistrala kojarzy sterowniki z urządzeniami wywołuje funkcję probe() sterownika

sysfs

- Magistrale, sterowniki i urządzenia funkcjonujące w jądrze, prezentowane są systemie plików `sysfs` mountowanum w katalogu `/sys`.
- Powiązania pomiędzy obiektami, reprezentowane są przy pomocy dowiązań symbolicznych, plików i katalogów:
  - `/sys/bus` - lista magistral,
  - `/sys/devices` - wszystkie urządzenia,
  - `/sys/bus/<magistrala>/devices` - urządzenia danej magistrali,
  - `/sys/bus/<magistrala>/drivers` - sterowniki w danej magistrali,
  - `/sys/class` - urządzenia pogrupowane według klas (frameworków): `net`, `input`, `block`, itp.
  - `/sys/bus/<magistrala>/devices/<urządzenie>/driver` - sterownik powiązany z danym urządzeniem; analogicznie w katalogu sterownika znajduje się dowiązanie do urządzenia(-ń).
- Z danych korzystają programy takie jak `udev` lub `mdev`.

Przykład


```
$ ls -l /sys/bus/pci/drivers/ahci/
razem 0
lrwxrwxrwx 1 root root 0 02-26 23:21 0000:00:1f.2 ->
         ../../../../devices/pci0000:00/0000:00:1f.2
--w------- 1 root root 4096 02-26 23:21 bind
lrwxrwxrwx 1 root root    0 02-26 23:21 module ->
                             ../../../../module/ahci
--w------- 1 root root 4096 02-26 23:21 new_id
--w------- 1 root root 4096 02-26 23:21 remove_id
--w------- 1 root root 4096 02-26 23:21 uevent
--w------- 1 root root 4096 02-26 23:21 unbind
```

- sterownik `ahci` obsługuje urządzenie PCI: `0000:00:1f.2`
- sterownik został zarejestrowany przez moduł o nazwie `ahci`
- moduł - w sensie organizacji kodu (nie ma różnicy czy kod jest wkompilowany na stałe, czy do pliku `.ko`)

Identyfikator urządzenia

- Magistrala kojarzy urządzenie ze sterownikiem.
- Metoda zależy od konkretnej magistrali, na przykład:
  - `PCI` - unikalne identyfikatory: *Vendor*, *Device* a także wersja i klasa urządzenia.
  - `USB` - *Vendor*, *Product*, *Class*
  - `I2C` - nazwy: sterownika i urządzenia (tu ewentualnie jeszcze numer)
  - `platform` - nazwy, lub pole `compatible=` w Dvice-Tree.
- W przypadku nieznalezienia w magistrali sterownika pasującego do urządzenia, jądro uruchomi program `modprobe` z identyfikatorem urządzenia jako argumentem.

Funkcjonalność ta może nie być dostępna w wersji `modprobe` z Busybox-a

Identyfikator urządzenia - przykład PCI


```
$ modinfo 8139too
alias:          pci:v00001113d00001211sv*sd*bc*sc*i*
alias:          pci:v000010ECd00008139sv*sd*bc*sc*i*
```

W kodzie `drivers/net/ethernet/realtek/8139too.c` znajdują się makra:

```
static const struct pci_device_id rtl8139_pci_tbl[] = {
  {0x10ec, 0x8139, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
  {0x10ec, 0x8138, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
  {0x1113, 0x1211, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
  {0x1500, 0x1360, PCI_ANY_ID, PCI_ANY_ID, 0, 0, RTL8139 },
  /* ... */
  {PCI_ANY_ID, 0x8139, 0x13d1, 0xab06, 0, 0, RTL8139 },
  {0,}
};
MODULE_DEVICE_TABLE (pci, rtl8139_pci_tbl);
```

Odpowiadają za wpisy w module i jego automatyczne ładowanie w przypadku znalezienia kompatybilnego urządzenia.

Sterownik urządzenia

Moduł rejestruje strukturę opisująca sterownik danej magistrali: `pci_driver`, `usb_driver`, `platform_driver` itp:

```
static struct pci_driver rtl8139_pci_driver = {
        .name           = DRV_NAME,
        .id_table       = rtl8139_pci_tbl,
        .probe          = rtl8139_init_one,
        .remove         = rtl8139_remove_one,
#ifdef CONFIG_PM
        .suspend        = rtl8139_suspend,
        .resume         = rtl8139_resume,
#endif /* CONFIG_PM */
};

static int __init rtl8139_init_module (void)
{ /* ... */
   return pci_register_driver(&rtl8139_pci_driver);
}
```

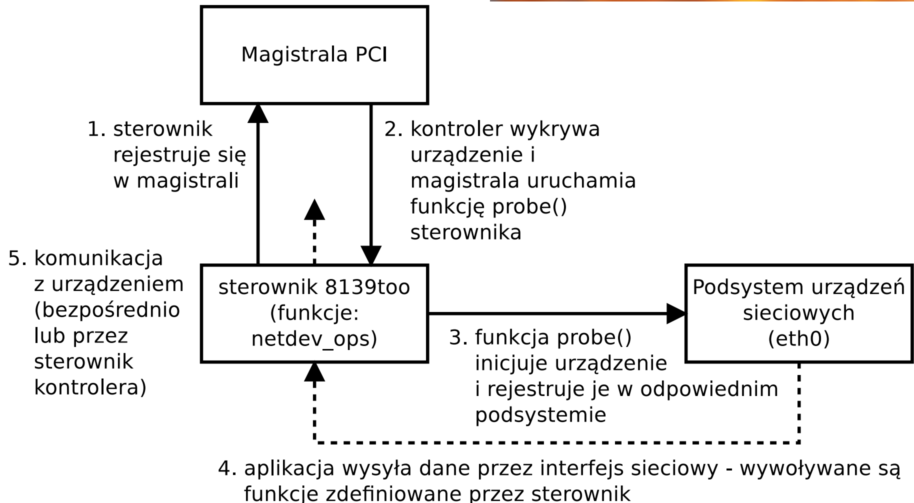
Analogicznie - sterownik jest wyrejestrowywany przy wyładowywaniu.

probe()

Sterownik implementuje funkcję, do której wskaźnik ustawiany jest w polu `.probe`.

- Jest ona wykonywana, jeżeli magistrala wykryje, że jeden z identyfikatorów w zarejestrowanym sterowniku pasuje do jakiegoś urządzenia.
- Jako argument przekazywany jest wskaźnik na strukturę opisująca urządzenie (jego zasoby, adresy, parametry) - specyficzną dla danej magistrali (`pci_dev`, `usb_dev`, `platform_device`).
- Uruchamia urządzenie, inicjuje rejestry, rezerwuje pamięć na bufory wejścia - wyjścia, rejestruje funkcje obsługi przerwań, timery itp.
- Rejestruje urządzenie odpowiedniej klasy. Przy pomocy funkcji danego frameworku, rejestruje w nim abstrakcyjne urządzenie (np. sieciowe, `misc`, `input`).

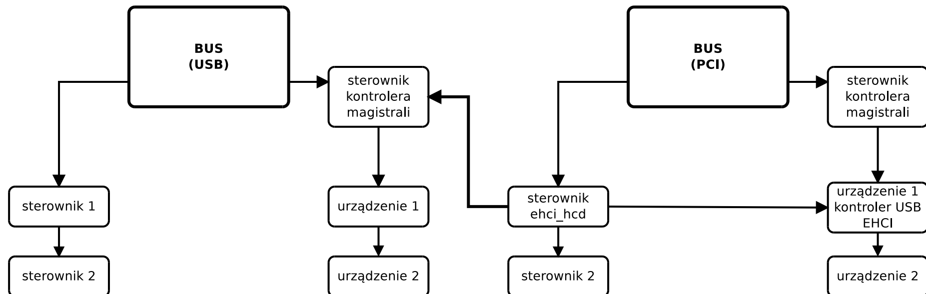
Przykład: sterownik karty sieciowej staje się pomostem pomiędzy odpowiednim interfejsem (urządzeń sieciowych) a konkretnym urządzeniem PCI. Zadaniem funkcji `probe()` jest zdefiniowanie tego pomostu.


Magistrala PCI
1. sterownik rejestruje się w magistrali
2. kontroler wykrywa urządzenie i magistrala uruchamia funkcję probe() sterownika
5. komunikacja z urządzeniem (bezpośrednio lub przez sterownik kontrolera)
sterownik 8139too (funkcje: netdev_ops)
Podsystem urządzeń sieciowych (eth0)
3. funkcja probe() inicjuje urządzenie i rejestruje je w odpowiednim podsystemie
4. aplikacja wysyła dane przez interfejs sieciowy - wywoływane są funkcje zdefiniowane przez sterownik

Przykład 8139too.c


```
static int rtl8139_init_one (struct pci_dev *pdev,
                             const struct pci_device_id *ent)
{
  struct net_device *dev = NULL;
  dev = rtl8139_init_board (pdev); /* Wywyołanie funkcji. */
    /* Alokacja i zainicjowanie struktury */
    struct net_device *dev;
    dev = alloc_etherdev (sizeof (*tp));
    /* Uruchomienie urządzenia, rezerwowanie zasobów */
    rc = pci_enable_device (pdev);
    rc = pci_request_regions (pdev, DRV_NAME);
    ioaddr = pci_iomap(pdev, bar, 0);
    rtl8139_chip_reset (ioaddr);
    return dev;
  /* Struktury ze wskaźnikami na funkcje obsługi urządzenia */
  dev->netdev_ops = &rtl8139_netdev_ops;
  dev->ethtool_ops = &rtl8139_ethtool_ops;
  /* Rejestrowanie urządzenia */
  i = register_netdev (dev);
}
```

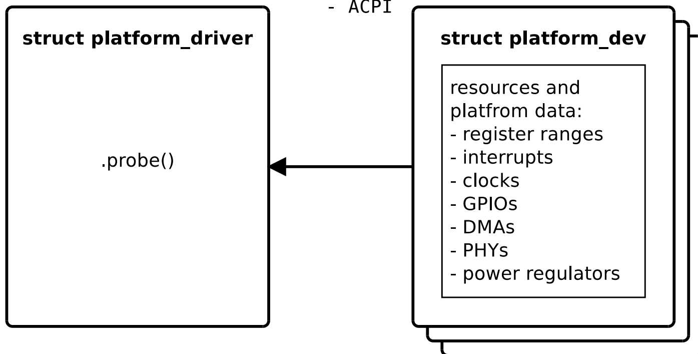
- Model urządzeń może być rekurencyjny - sterownik urządzenia jednej magistrali, może dostarczać sterownik kontrolera w innej.
  Na przykład kontroler USB podłączony do PCI.
  Kontroler I2C, podłączony do USB, którego kontroler podłączony jest do PCI, itp.



Platform bus

platform_driver_register()

```
- Device-Tree:
    of_platform_default_populate_init()
    (arch_initcall_sync())
- Board file code: init_machine()
- ACPI
```



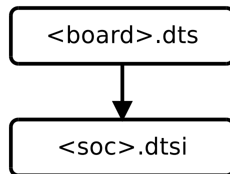
Device-Tree

- **Device-Tree** to plik tekstowy opisujący sprzęt.
  - Ma strukturę drzewiastą.
  - Nie ogranicza się tylko do Linuksa, jest dość starym standardem.
  - Ma ściśle określoną gramatykę.
- Podczas startu systemu, Linux nie zwraca już uwagi na *machine number* (rejestr `r1`). W `r2` jest przekazywany adres w pamięci, do którego zostało załadowane skompilowane Device-Tree.

  W przypadku starszego bootloadera, który nie będzie w stanie załadować Device-Tree, można skorzystać z opcji doklejenia danych do obrazu jądra (opcja: `CONFIG_ARM_APPEND_DTB`). Ładujemy wtedy:

```
cat arch/arm/boot/zImage arch/arm/boot/dts/moj.dtb > moj_zImage
```

Obsługa Device-Tree



- W Linuksie Device-Tree używane jest dla architektur: ARM i PowerPC.
- Dla ARM pliki znajdują się w: `arch/arm/boot/dts`:
  - **`.dts`** - pliki dla konkretnych urządzeń,
  - **`.dtsi`** - „biblioteki” dla SoC-ów, SoM-ów, itp.
- Kompilator (`scripts/dtc/`**`dtc`**) tworzy z nich wersję binarną - **`.dtb`**
- Operacja kompilacji jest odwracalna.
- Od wersji 3.13 używany jest preprocesor C (symbole, pliki **`.h`**)

Kompilacja:

```
$ make ARCH=arm dtbs
```

Aby zbudować zewnętrzny plik, należy skopiować go do `arch/arm/boot/dts`, i uruchomić:

```
$ make ARCH=arm <board>.dtb
```

Korzystanie z dtc

DTC - *Device Tree Compiler*

- Budowany razem z Linuksem (preferowany - na pewno obsługuje odpowiednią składnię):

  ```
  ./scripts/dtc/dtc
  ```

  Może być również zainstalowany jako pakiet dystrybucji lub zawarty w BSP.

- Użycie:

  ```
  dtc -I <input format> -O <output format> <plik>
  ```

  Formaty wejściowe: `dts`, `dtb`, `fs` (struktura katalogów z `/proc/device-tree`).
  Formaty wyjściowe: `dts`, `dtb`, `asm` (tablica w assemblerze).

- Dekompilacja pliku dtb do czytelnej postaci (dts):

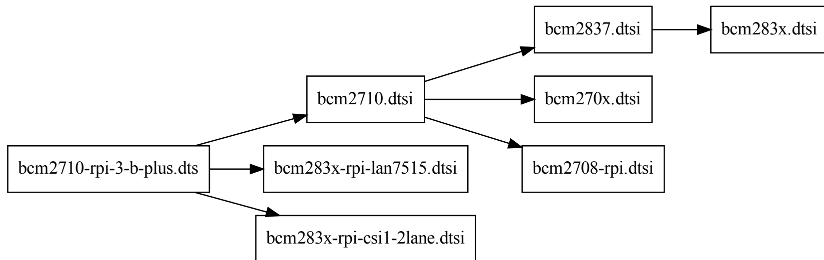
  ```
  dtc -I dtb -O dts plik.dtb | less
  ```

- Kompilacja plików dostarczanych z jądrem - szczegóły:

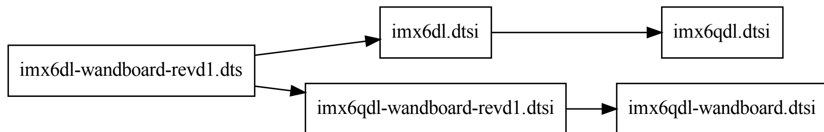
  ```
  $ <pref>-cpp -nostdinc -undef -D__DTS__ -x assembler-with-cpp \
     -Iarch/arm/boot/dts -Iarch/arm/boot/dts/include -o t.tmp t.dts
  $ scripts/dtc/dtc -O dtb -o t.dtb -i arch/arm/boot/dts t.tmp
  ```

Pliki dts dla urządzeń:

- Raspberry Pi:



- Wandboard:



Każde urządzenie opisywane jest przez jedno drzewo. Zaczyna się ono od korzenia (`skeleton.dtsi`):

```
 7 / {
 8       #address-cells = <1>;
 9       #size-cells = <1>;
/* Sekcja chosen służy do przekazywania ogólnych ustawień, na przykład parametrów j
10       chosen { };
/* Alternatywne nazwy urządzeń */
11       aliases { };
/* Informacje o ilości pamięci operacyjnej i jej fizycznych adresach */
12       memory { device_type = "memory"; reg = <0 0>; };
13 };
```

Minimalna zawartość drzewa:
`Documentation/devicetree/booting-without-of.txt`.
Plik jest załączany przez `imx6qdl.dtsi` który rozszerza wpisy na dwa sposoby:

1. Re-define same path to extend or modify entries:

```
/ {           <-- Device-Tree root node
  memory {
                device_type = "memory";
                reg = <0x80000000 0x10000000>; /* 256 MB */
        };
};
```

2. Find an element by alias:

```
&mmc1 {
        vmmc-supply = <&vmmcsd_fixed>;
};
```

`mmc1` is an alias. It can even be defined later!

```
/ {
  ocp {
    mmc1: mmc@48060000 {
      /* ... */
    };
  };
};
```

`imx6qdl.dtsi` definiuje peryferia SoC-a:

```
 16 / { /*77*/     soc {
732         aips-bus@02100000 { /* AIPS2 */
899             i2c3: i2c@021a8000 {
```

Wpisy z # określają ilość wartości w polu `reg`

```
900                 #address-cells = <1>;
901                 #size-cells = <0>;
```

Linia `compatible` jest używana do parowania ze sterownikami.

```
902               compatible = "fsl,imx6q-i2c", "fsl,imx21-i2c";
```

Fizyczny adres i rozmiar obszaru rejestrów w przestrzeni adresowej:

```
903                 reg = <0x021a8000 0x4000>;
```

Przerwanie: `<kontroler, numer irq, flagi>`, zegary:

```
904                 interrupts = <0 38 0x04>;
905                 clocks = <&clks 127>;
```

Urządzenie jest nieaktywne.

Board definition

Device-Tree dla urządzenia:

- `imx6q-sabresd.dts` załącza `imx6qdl-sabresd.dtsi`.

Dodatkowe parametry sterownika (zależne od układów peryferyjnych):

```
308 &i2c3 {
309         clock-frequency = <100000>;
```

Odwołanie do sekcji sterownika multipleksacji:

```
310         pinctrl-names = "default";
311         pinctrl-0 = <&pinctrl_i2c3>;
```

Włączenie urządzenia:

```
312         status = "okay";
```

Dodatkowe węzły (wykorzstywane przez sterownik):

```
314         egalax_ts@04 {
315                 compatible = "eeti,egalax_ts";
/*...*/
320 }; };
```

```
1045 &iomuxc {
...
1329    i2c3 {
```

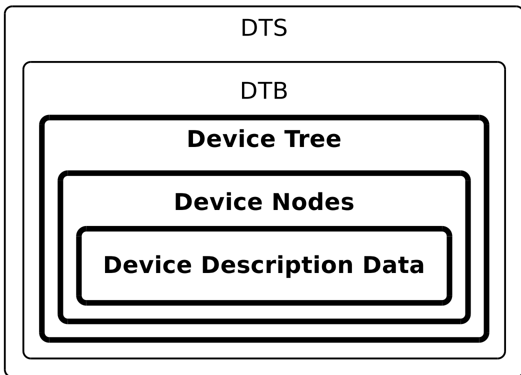
Alternatywne pin-y sygnałów magistrali i2c3 Wartości określają liczby, którymi zaprogramowany zostanie kontroler PINMUX - wybór opcji pin-u i jego ustawienia (np. pull-up).

```
1330        pinctrl_i2c3_1: i2c3grp-1 {
...
1337        pinctrl_i2c3_2: i2c3grp-2 {
...
1344        pinctrl_i2c3_3: i2c3grp-3 {
1345            fsl,pins = <
1346                MX6QDL_PAD_GPIO_5__I2C3_SCL  0x4001b8b1
1347                MX6QDL_PAD_GPIO_16__I2C3_SDA 0x4001b8b1
1348            >;
1349        };
1350
1351        pinctrl_i2c3_4: i2c3grp-4 {
...
1356    }; };
```

ACPI example


```
_DSD (Device Specific Data) introduced in ACPI 5.1
Name (_DSD, Package () {
        ToUUID("<UUID string>"), // Format identifier
        Package () {
                ...         // Device data in the given format
        }
}

Device Properties UUID: daffd814-6eba-4d8c-8a91-bc9bbf4aa301
Name (_DSD, Package () {
  ToUUID("daffd814-6eba-4d8c-8a91-bc9bbf4aa301"),
  Package () {
    Package {"a-string-property", "A string"},
    Package {"a-string-list-property", Package {"first string",
                                                "second string"}};
    Package {"a-cell-property", Package {1, 2, 3, 4}};
  }
}
```



ASL (ACPI Source Language)

AML (ACPI Machine Language)

ACPI Tables

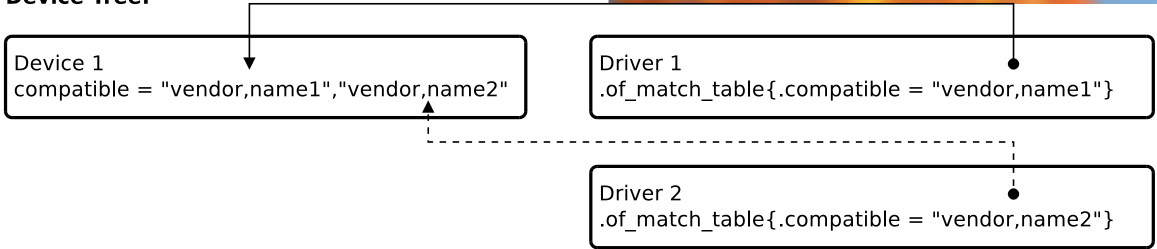
ACPI Definition Blocks

ACPI Namespace

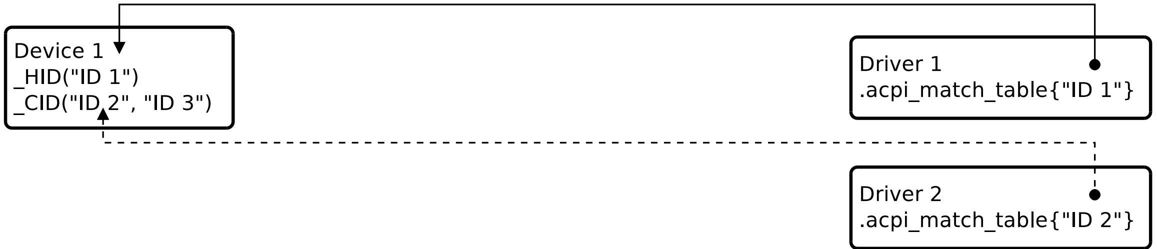
ACPI Objects

Device Description Data

Device Tree:



ACPI:

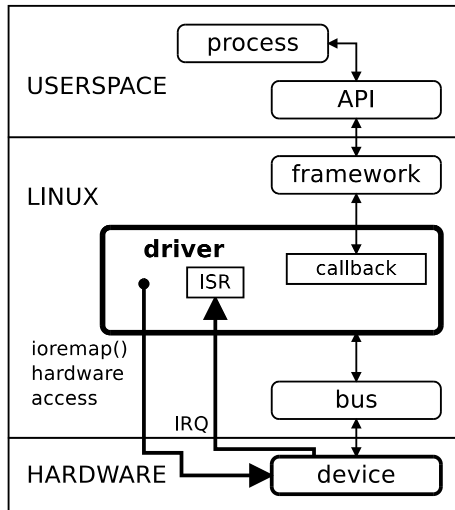


Interrupt

Linux kernel performs operations in two contexts:

- process context - on request from userspace process (synchronously).
- interrupt context - on request from hardware - interrupt controller (asynchronously).

Interrupts are scheduled by CPU and run asynchronously. Kernel could be in any state, when interrupt occurs. So interrupt context can not access user buffers, and can not sleep. Memory must be allocated witch `GFP_ATOMIC` flag.



Registering an interrupt handler

Architecture-independent functions defined in `<linux/interrupt.h>`

```
int request_irq(              /* returns 0 on success */
  unsigned int irq,           /* irq number */
  irq_handler_t handler,      /* points to handler */
  unsigned long irq_flags,    /* flags */
  const char * devname,       /* will be in /proc/interrupts */
  void *dev_id);              /* handler arguents (not null) */
```

***I**nterrupr **S**ervice **R**equest*:

```
irqreturn_t (*handler) (unsigned int irq, void *dev_id);
```

Returns:

- `IRQ_HANDLED` - request recognized and handled (irq line freed), system ready for next request;
- `IRQ_NONE` - request not handled (device does not signal it).
  Kernel will try to run next handler on current line. If none available - error occurs (*spurious interrupt*) and line is blocked.

Flags (may be coupled with | operator):

- `IRQF_SHARED` - allows sharing interrupt line between devices (common in PC),
- `IRQF_NO_THREAD` - avoids switching interrupt handler to thread,
- `IRQF_ONESHOT` - does not re-activate interrupt line after handling it. Used if interrupt will be re-enabled in some other place,
- `IRQF_NO_SUSPEND` - interrupt is not masked on system suspend (can be used to wake-up system),
- `IRQF_TRIGGER_RISING`, `IRQF_TRIGGER_FALLING`, `IRQF_TRIGGER_HIGH`, `IRQF_TRIGGER_LOW` - interrupt triggered by slope or level (defined by hardware).

Unregistering interrupt handler:

```
void free_irq(unsigned int irq, void *dev_id);
```

Registered handlers (x86)


```
$ cat /proc/interrupts
           CPU0       CPU1
  0:         45          0 IO-APIC-edge      timer
  1:    1686893        413 IO-APIC-edge      i8042
  8:          0          1 IO-APIC-edge      rtc0
  9:     251756        993 IO-APIC-fasteoi   acpi
 16:   16018752      47476 IO-APIC-fasteoi   ath9k, snd_hda_intel
 17:    2624190         20 IO-APIC-fasteoi   ehci_hcd:usb1,
                                                     ehci_hcd:usb2
 40:          0          0 PCI-MSI-edge      xhci_hcd
 44:    1940203      17322 PCI-MSI-edge      ahci
NMI:       4486       4740 Non-maskable interrupts
LOC:   43535024   39106389 Local timer interrupts
SPU:          0          0 Spurious interrupts
TLB:    1784029    1865609 TLB shootdowns
TRM:          0          0 Thermal event interrupts
THR:          0          0 Threshold APIC interrupts
MCE:          0          0 Machine check exceptions
MCP:       2050       2038 Machine check polls
ERR:          1
```

Registered handlers (ARM)

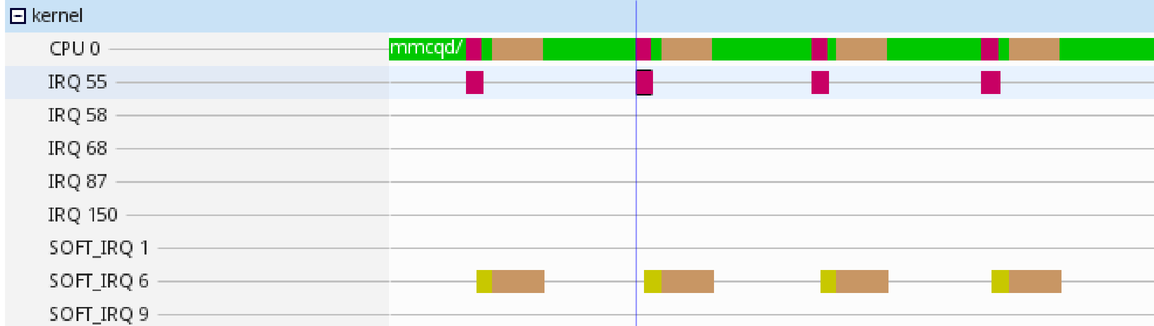

```
# cat /proc/interrupts
           CPU0
  4:       1165        VIC  Versatile Timer Tick
 12:        470        VIC  uart-pl011
 25:       2060        VIC  eth0
 35:          7        SIC  kmi-pl050
 36:         89        SIC  kmi-pl050
Err:          0
```

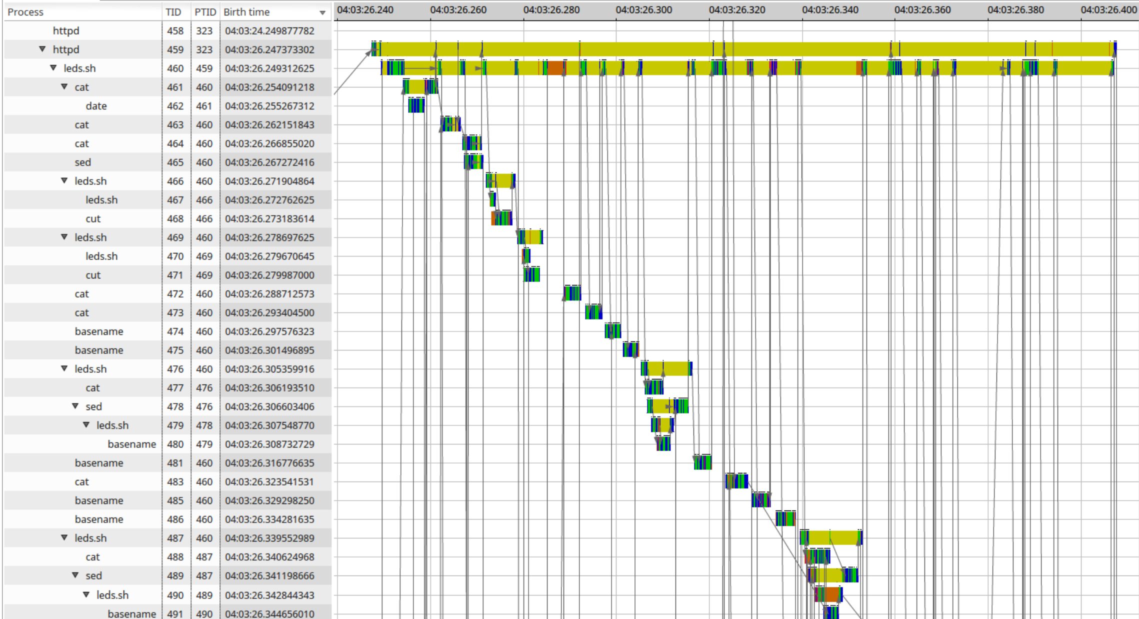
Counters are also available in:

```
$ cat /proc/stat | grep intr
intr 397262642 12071530 18 0 0 0 0 0 0 1 0 0 0 28
                             0 0 0 10252120 0 ...
```

Better for automatic parsing.

Each field - one interrupt number.


Resources
2016 Mar 06
16:22:06.837400
16:22:06.837450
16:22:06.837500
kernel
CPU 0
mmcqd/
IRQ 55
IRQ 58
IRQ 68
IRQ 87
IRQ 150
SOFT_IRQ 1
SOFT_IRQ 6
SOFT_IRQ 9


Control Flow
Process
TID
PTID
Birth time
04:03:26.240
04:03:26.260
04:03:26.280
04:03:26.300
04:03:26.320
04:03:26.340
04:03:26.360
04:03:26.380
04:03:26.400
httpd 458 323 04:03:24.249877782
httpd 459 323 04:03:26.247373302
leds.sh 460 459 04:03:26.249312625
cat 461 460 04:03:26.254091218
date 462 461 04:03:26.255267312
cat 463 460 04:03:26.262151843
cat 464 460 04:03:26.266855020
sed 465 460 04:03:26.267272416
leds.sh 466 460 04:03:26.271904864
leds.sh 467 466 04:03:26.272762625
cut 468 466 04:03:26.273183614
leds.sh 469 460 04:03:26.278697625
leds.sh 470 469 04:03:26.279670645
cut 471 469 04:03:26.279987000
cat 472 460 04:03:26.288712573
cat 473 460 04:03:26.293404500
basename 474 460 04:03:26.297576323
basename 475 460 04:03:26.301496895
leds.sh 476 460 04:03:26.305359916
cat 477 476 04:03:26.306193510
sed 478 476 04:03:26.306603406
leds.sh 479 478 04:03:26.307548770
basename 480 479 04:03:26.308732729
basename 481 460 04:03:26.316776635
cat 483 460 04:03:26.323541531
basename 485 460 04:03:26.329298250
basename 486 460 04:03:26.334281635
leds.sh 487 460 04:03:26.339552989
cat 488 487 04:03:26.340624968
sed 489 487 04:03:26.341198666
leds.sh 490 489 04:03:26.342844343
basename 491 490 04:03:26.344656010

Komentarze/pytania

https://bis-linux.com/jesien_linuksowa_2024

/* */ || ?