Linux Device Drivers – Data Types in Kernel

Jernej Vičič

Overview

1. Introduction
   - Standard C Types
   - Explicit size data types
   - Interface-Specific Types
2. jiffy
3. Other portability issues
   - Page size
   - Byte order
   - Data alignment
   - Linked lists

Introduction

- portability issues,
- kernels are "highly portable",
- drivers should be also,
- and they are ... the good ones ...,
- data types,
- 3 basic groups:
  - *standard C types* – basic types,
  - *explicitly sized types* – types with explicit size,
  - *types used for specific kernel objects* – special kernel data types.

Standard C Types

- basic types,
- examples:
  - int, long,
- can be changed on different architectures

Standard C Types

arch	Size:	char	short	int	long	ptr	long-long	u8	u16	u32	u64
i386		1	2	4	4	4	8	1	2	4	8
alpha		1	2	4	8	8	8	1	2	4	8
armv4l		1	2	4	4	4	8	1	2	4	8
ia64		1	2	4	8	8	8	1	2	4	8
m68k		1	2	4	4	4	8	1	2	4	8
mips		1	2	4	4	4	8	1	2	4	8
ppc		1	2	4	4	4	8	1	2	4	8
sparc		1	2	4	4	4	8	1	2	4	8
sparc64		1	2	4	4	4	8	1	2	4	8
x86_64		1	2	4	8	8	8	1	2	4	8

Figure: Types.

Standard C Types

```
kernel: arch    Size:  char short int long  ptr long-long u8 u16 u32 u64
kernel: sparc64          1    2    4    8    8      8       1   2   4   8
```

Figure: Sparc64.

- interesting example:
  - *SPARC 64* uses (used:)) 32-bit user space,
  - pointers 32 bit,
  - in kernel pointers are 64 bit,
- nnot the same on all architectures.

Explicit size data types

- Sometimes we need data types of a specified size,
- examples:
  - correspond to specific binary structures,
  - communication with user space,
  - alignment of data in structures (adding "padding fields"),
- The kernel allows to use data types where you know the exact size.

Explicit size data types

```
<asm/types.h>
<linux/types.h>

u8; /* unsigned byte (8 bits) */
u16; /* unsigned word (16 bits) */
u32; /* unsigned 32-bit value */
u64; /* unsigned 64-bit value */
s8; /* signed byte (8 bits) */
s16; /* signed word (16 bits) */
s32; /* signed 32-bit value */
s64; /* signed 64-bit value */
```

- *signed* rarely used,
- user space should use __ prefix,
- example: *__u8* prefix.

Explicit size data types

- these types are specific to Linux,
- using these types makes it hard to port the driver to other UNIX sistems,
- new (already old) compilers support *C99* standard types:
  - *uint8_t*,
  - *uint32_t*
- use these types to preserve portability,
- could run into problems with compilers (hopefully not anymore).

Interface-Specific Types

- *interface-specific* – type defined by the library that is the interface for a data strucutre,
- some types have own *typedef*,
- these type should have no problems with portability,
- examples:
  - *pid_t* – process identifier (instead of int),
  - use *pid_t* masks possible problems.

Interface-Specific Types

- most special types are defined in <linux/types.h>,
- they are tagged with *_t*,
- driver needs functions that must have special types,
- if not used properly the compiler will complain,
- *-Wall* – This enables all the warnings,
- if there are no warnings, the code will be portable.

Interface-Specific Types

- problems when printing,
- example: *size_t*:
  - can be unsigned long,
  - can be unsigned int,
  - depends on architecture,
- partial solution is casting in the biggest possible type,
- this is usually long or unsigned long,
- displaying this type is easy with the printk and printf functions,

jiffy

- Time Intervals
- *Jiffy* is unformal time measure,
- defines short time (not exactly defined),
- "I'll be back in a jiffy",
- 'jiffies' OR 'ticks' is the finest time resolution that we can get on Linux system,
- proces blocks for at least one *jiffy*,
- *jiffy* is defined with 'HZ'.

jiffy

- *HZ* determines how many times in a second the timer interrupt occurs,
- initially this value was set to 100, and later to 250, then 1000,
- now is the kernel parameter, (100, 250, 300 (for video), 1000).

jiffy

- The accuracy of many system calls and timestamps is limited by the resolution of the software clock,
- clock maintained by the kernel which measures time in jiffies.
- The size of a jiffy is determined by the value of the kernel constant HZ.
- The value of HZ varies across kernel versions and hardware platforms.

jiffy

- On i386 the situation is as follows: on kernels up to and including 2.4.x, HZ was 100, giving a jiffy value of 0.01 seconds,
- starting with 2.6.0, HZ was raised to 1000, giving a jiffy of 0.001 seconds,
- since kernel 2.6.13, the HZ value is a kernel configuration parameter and can be 100, 250 (the default) or 1000, yielding a jiffies value of, respectively, 0.01, 0.004, or 0.001 seconds.

converting into jiffys and back

```
#define MAX_JIFFY_OFFSET ((~0UL >> 1)-1)

    static __inline__ unsigned long
    timespec_to_jiffies(struct timespec *value)
    {
            unsigned long sec = value->tv_sec;
            long nsec = value->tv_nsec;

            if (sec >= (MAX_JIFFY_OFFSET / HZ))
                    return MAX_JIFFY_OFFSET;
            nsec += 1000000000L / HZ - 1;
            nsec /= 1000000000L / HZ;
            return HZ * sec + nsec;
    }

    static __inline__ void
    jiffies_to_timespec(unsigned long jiffies, struct timespec *value)
    {
            value->tv_nsec = (jiffies % HZ) * (1000000000L / HZ);
            value->tv_sec = jiffies / HZ;
    }
```

Page size

- page sizeis *PAGE_SIZE*,
- can be 4 KB!,
- depends on architecture,
- from 4 KB to 64 KB.

Page size

- macros:
  - *PAGE_SIZE* – page size,
  - *PAGE_SHIFT* – number of bits, that need to be shifted for an address to get page number,
  - defined in $< asm/page.h >$.
- user space programs: function
- getpagesize,

Explicit size data types

```
#include <asm/page.h>
int order = get_order(16*1024);
buf = get_free_pages(GFP_KERNEL, order);
```

- driver needs 16 KB of temporary data,
- the argument of the *get_order* function must be a 2-based potency.

Byte order

- PC saves multibyte values by low-byte first,
- little end first,
- little-endian,
- example (big-endian): PowerPC,
- your code should not be dependent on this.

Byte order

- $< asm/byteorder.h >$ defines $\_\_LITTLE\_ENDIAN$ and $\_\_BIG\_ENDIAN$,
- depends on processor,
- in code we can use:

Byte order

```
#ifdef __LITTLE_ENDIAN
```

- we can use a set of macros,
- serve as conversions between multibyte types.

Byte order

```
u32 cpu_to_le32 (u32);
u32 le32_to_cpu (u32);
```

- convert to and from 32 bit unsigned.

Data alignment

- example: how to read the 4-byte size value,
- is stored at a title other than a multiple of 4,
- for outstanding data use macros:

```
#include <asm/unaligned.h>

get_unaligned(ptr);
put_unaligned(val, ptr);
```

Linked lists

- commonly used data structure,
- every programmer had its own implementation,
- is now in the kernel:
  - double-linked,
  - circular,
  - list,
- functions are lock-free,
- difficult to detect errors.

Linked lists

```
<linux/list.h>

struct list_head {
  struct list_head *next, *prev;
};
```

- *list_head* add to our structure:

```
struct todo_struct {
  struct list_head list;
  int priority; /* driver specific */
  /* ... add other driver-specific fields */
};
```

Linked lists

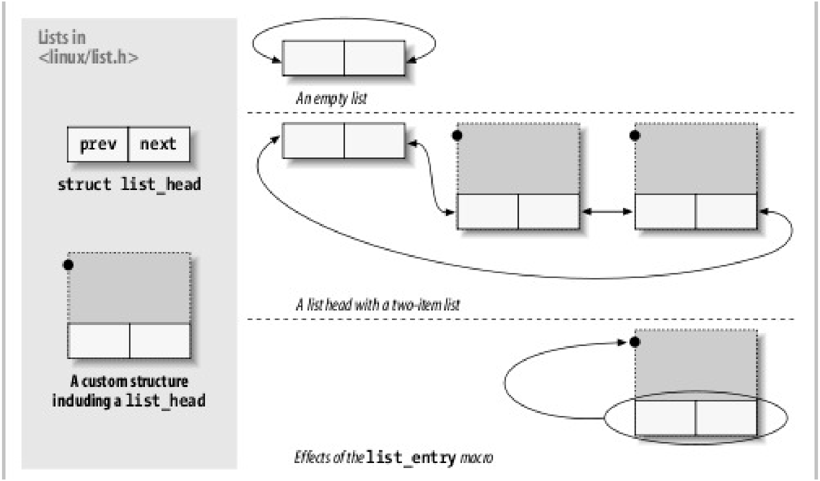

Figure: Linked list

Linked lists

list_add(struct list_head \asterisk new, struct list_head \asterisk head); - doda nov vnos po glavi,

list_add_tail(struct list_head *new, struct list_head *head); - doda nov vnos pred podanim elementom,

list_del(struct list_head *entry); - zbriše element,

list_move(struct list_head *entry, struct list_head *head);} -

list_move_tail(struct list_head *entry, struct list_head *head); - element je premaknjen na začetek head,

list_empty(struct list_head *head); - vrne neničelno rešitev, če je prazen seznam,

list_splice(struct list_head *list, struct list_head *head); - združi dva seznama, list za head.

Linked lists

```
void todo_add_entry(struct todo_struct *new)
{
  struct list_head *ptr;
  struct todo_struct *entry;
  list_for_each(ptr, &todo_list) {
    entry = list_entry(ptr, struct todo_struct, list);
    if (entry->priority < new->priority) {
      list_add_tail(&new->list, ptr);
      return;
    }
  }
  list_add_tail(&new->list, &todo_struct)
}
```