Linux centered heterogeneous multi-core architectures

A. Paccoia, M. Rodolfi, C. Salati, M. Sartori

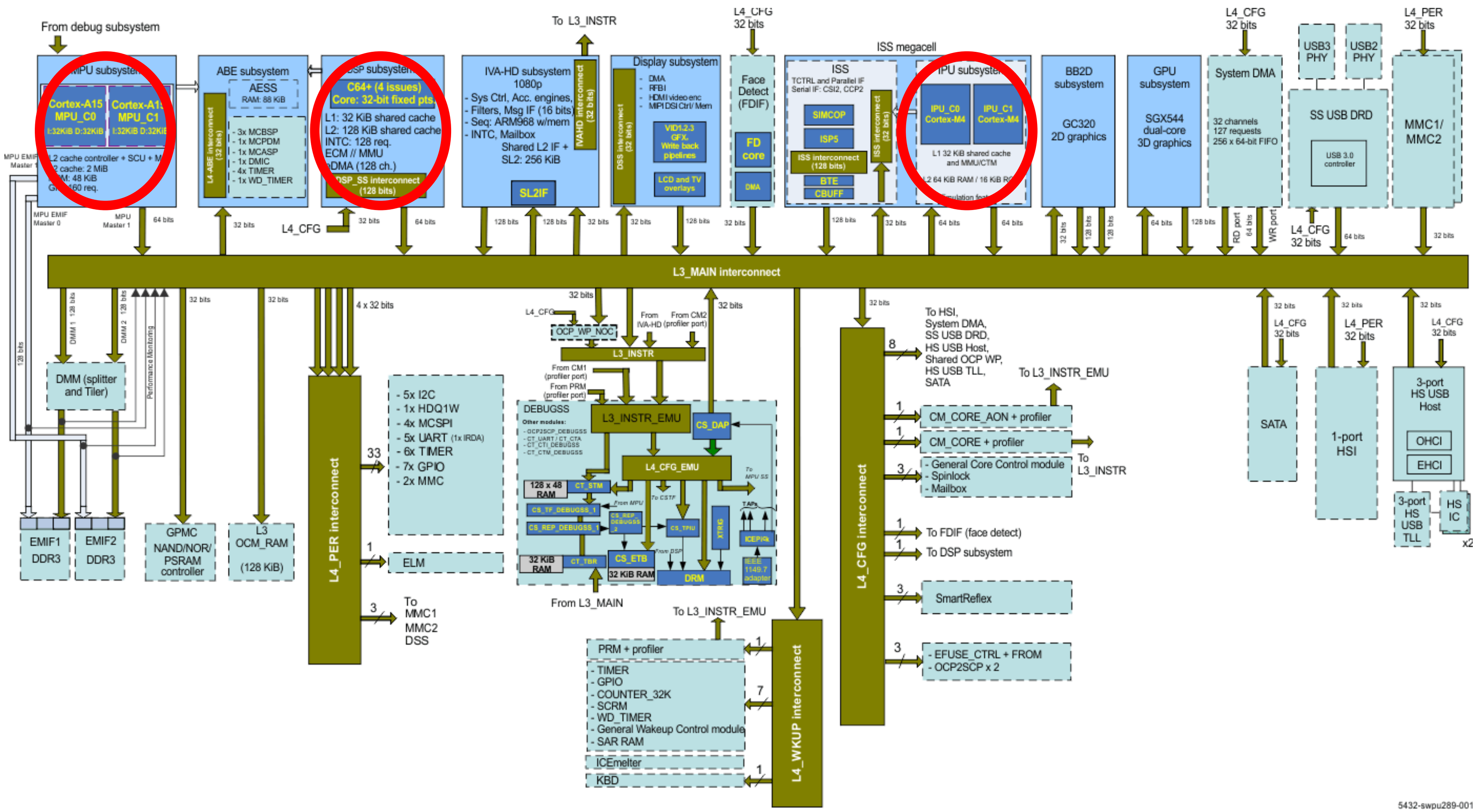
T3LAB, Bologna, Italy

Ordine degli Ingegneri, Bologna, novembre 2016

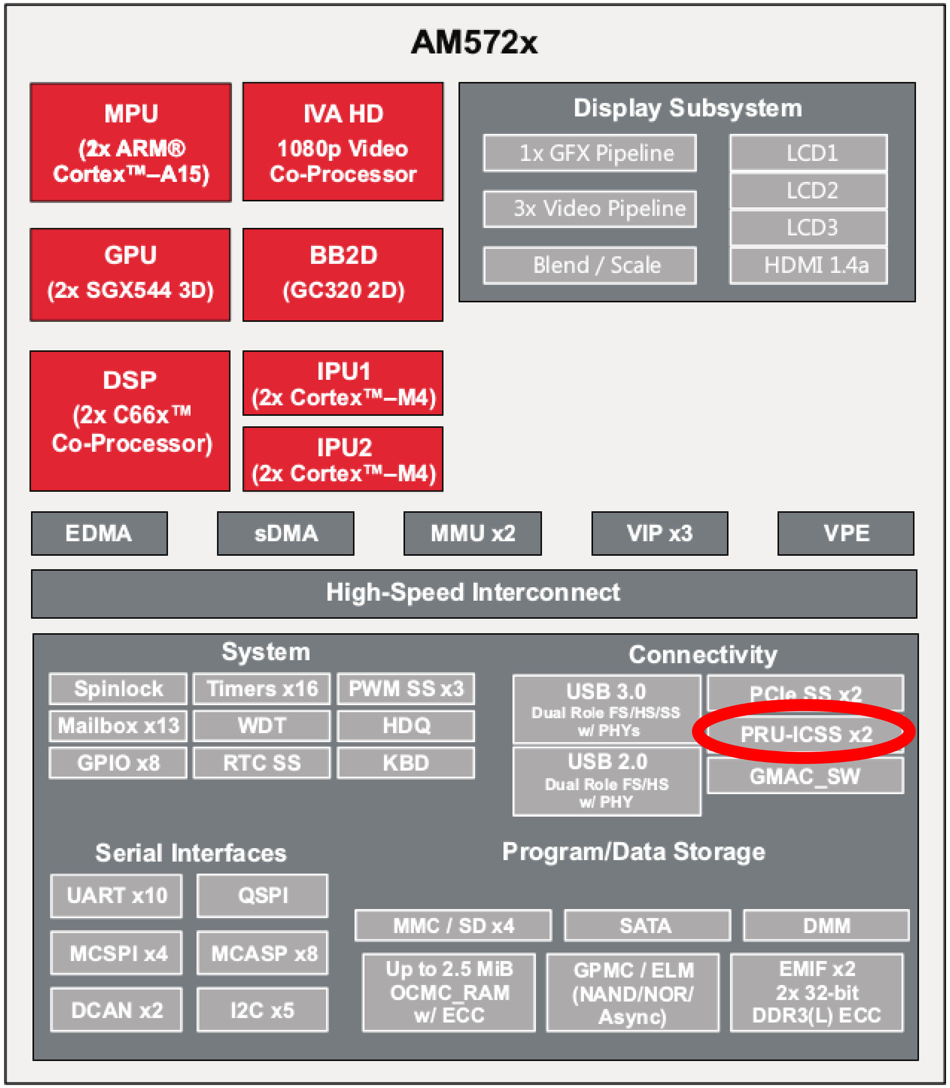
List of contents

- Evolution of HW platforms
- Rationale for heterogeneous multi-core architectures
- Complex SW platforms: problems and requirements
- What’s available: remoteproc and rpmsg
- An rpsmsg based inter-kernels IPC service and its socket API
- Exporting the rpmsg bus interface to the user space
- Linux topics
    - Device model
    - Device Tree
    - Device driver API
    - Sockets
    - Platform and misc drivers

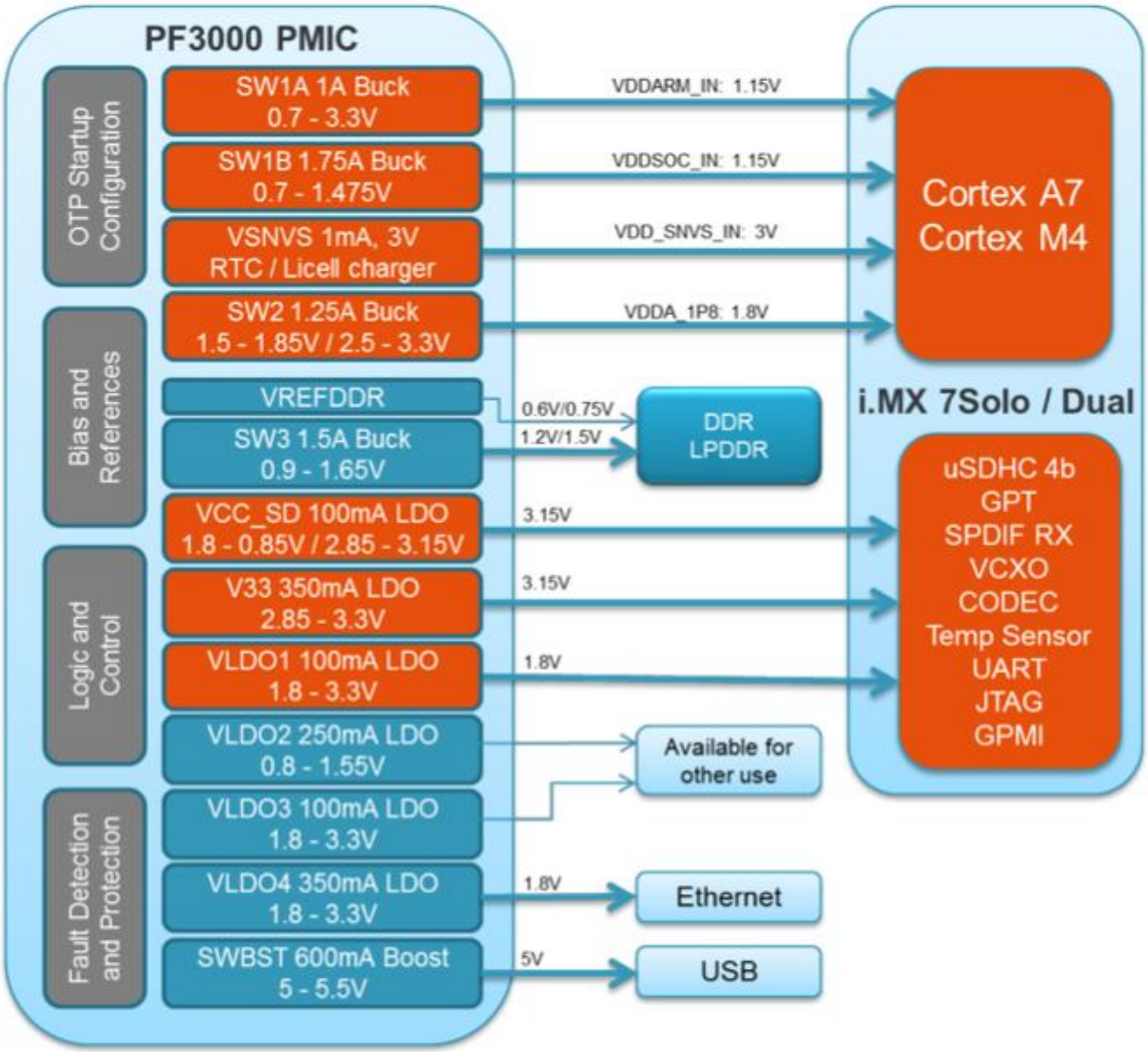
HW platforms: TI OMAP 5



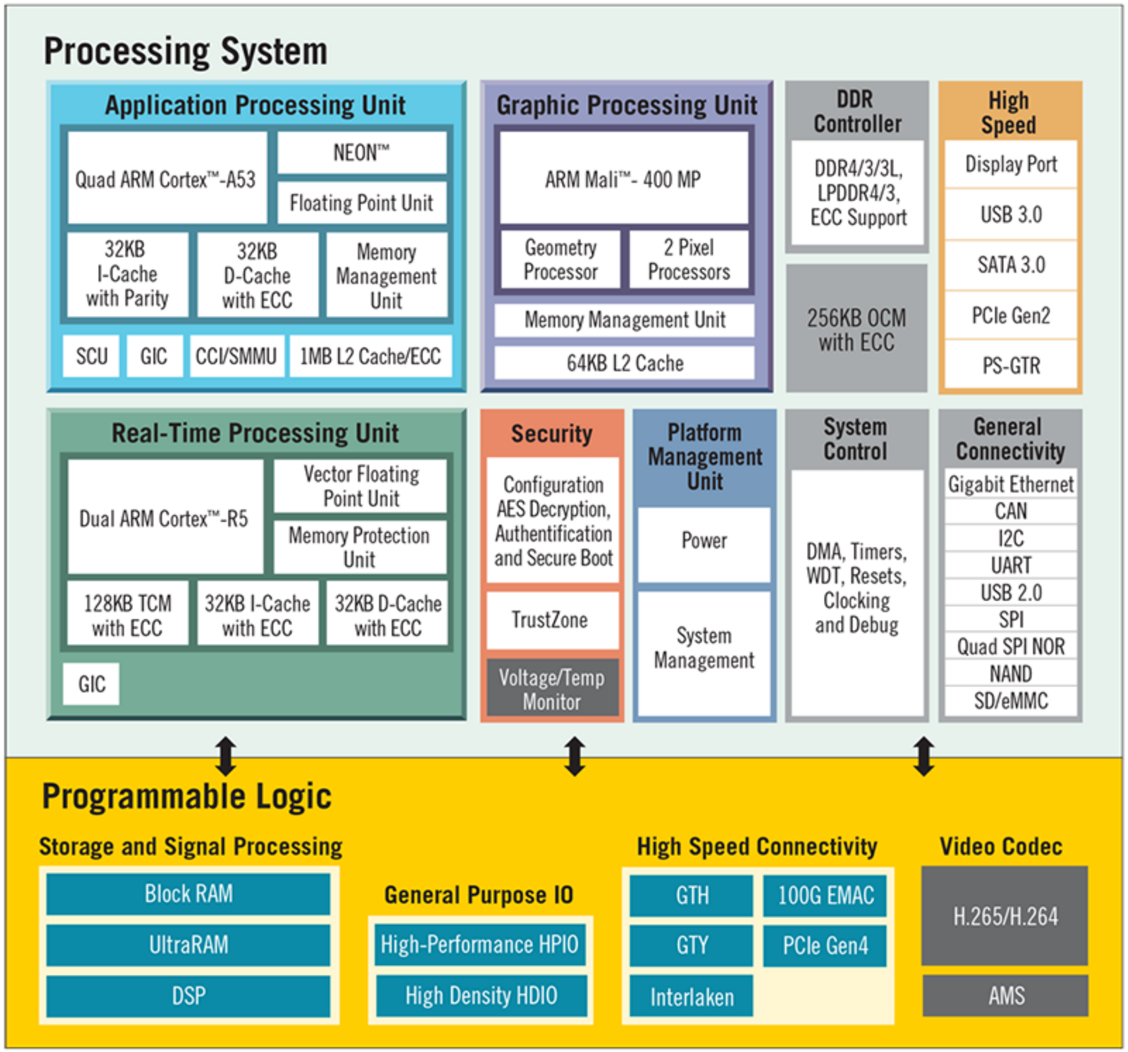
HW platforms: TI Sitara AM572x



HW platforms: NXP i.MX 7Solo/Dual

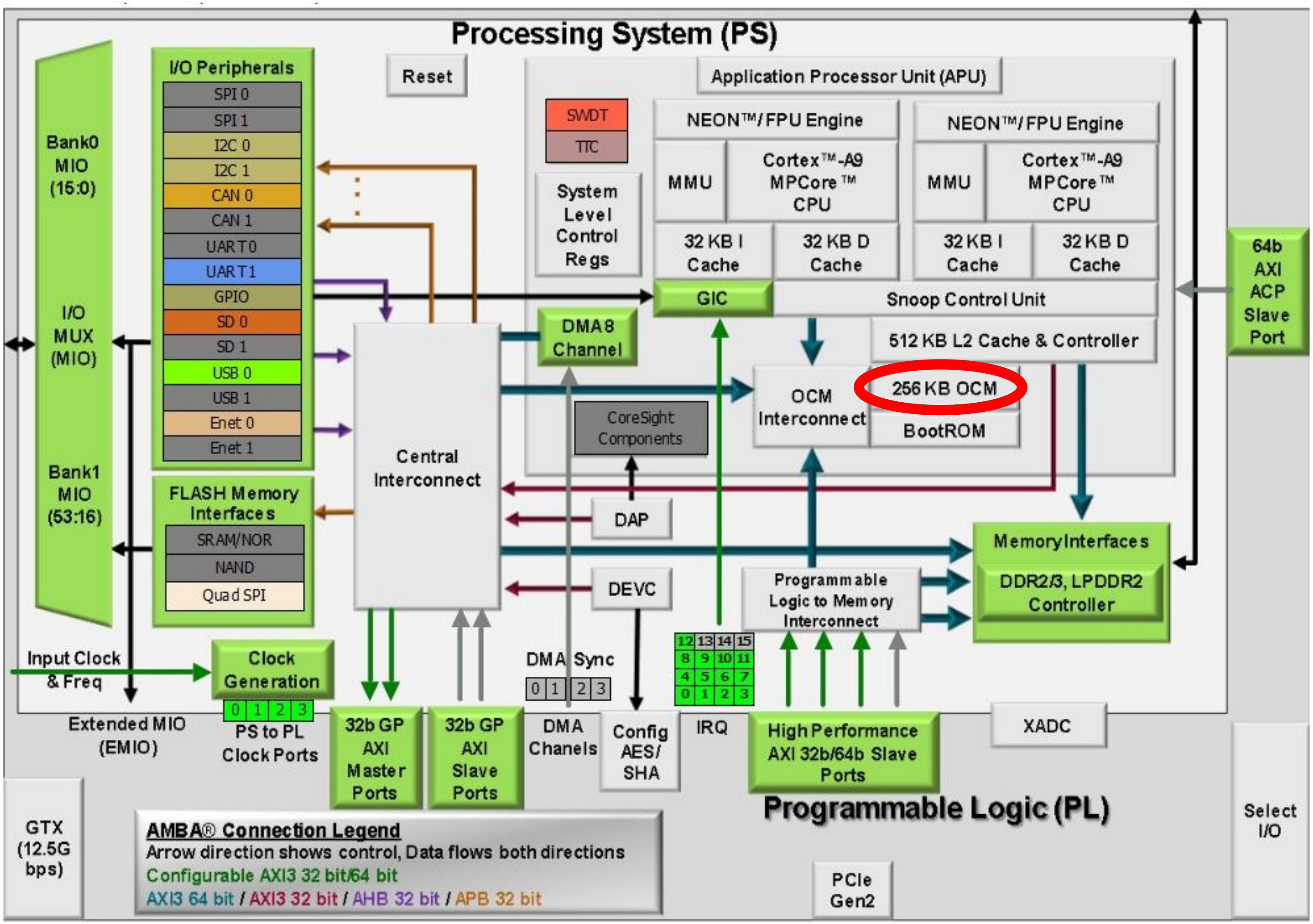


HW platforms:
Xilinx UltraScale MPSoC



Note: Illustration not drawn to scale.

HW platforms: Xilinx Zynq and SW generated heterogeneity



Why AMP?
(Asymmetric Multi-Processing)

- Consolidation of applications
  - Reuse
  - Space / weight / power reduction
  - "A growing number of embedded use cases require concurrent execution of isolated SW environments within the system" (F. Baum, Mentor Graphics)
- Robustness / security
- Boot time
- Heterogeneous functional and performance requirements
  - Real time

Heterogeneous requirements: Computing vs. controlling

- **Computing**
  - ➢ Large and complex applications
  - ➢ Heavy computational requirements
  - ➢ Real time / high throughput
  - ➢ Complex arithmetic
  - ➢ Large data movements
- **Controlling**
  - ➢ Real time / determinism
  - ➢ Minimum latency

Addressing design challenges in heterogeneous multicore embedded systems (W. Kurisu, Mentor Graphics)

1. **Each device runs its own operating system or operating environment**
2. **Each device runs on its own discrete processor and those processors are typically different**
   the type of application drives the processor selection, ranging from low-end microcontrollers to high-end application processors;
   each component of the system has full ownership of all the hardware available to the component. Examples of that hardware include the processors, graphic processing units, memory, I/O, cache, etc.
3. **The discrete components of this system are typically loosely connected**
   each component boots independently (?) and communicates with each other through messages over some physical connection.

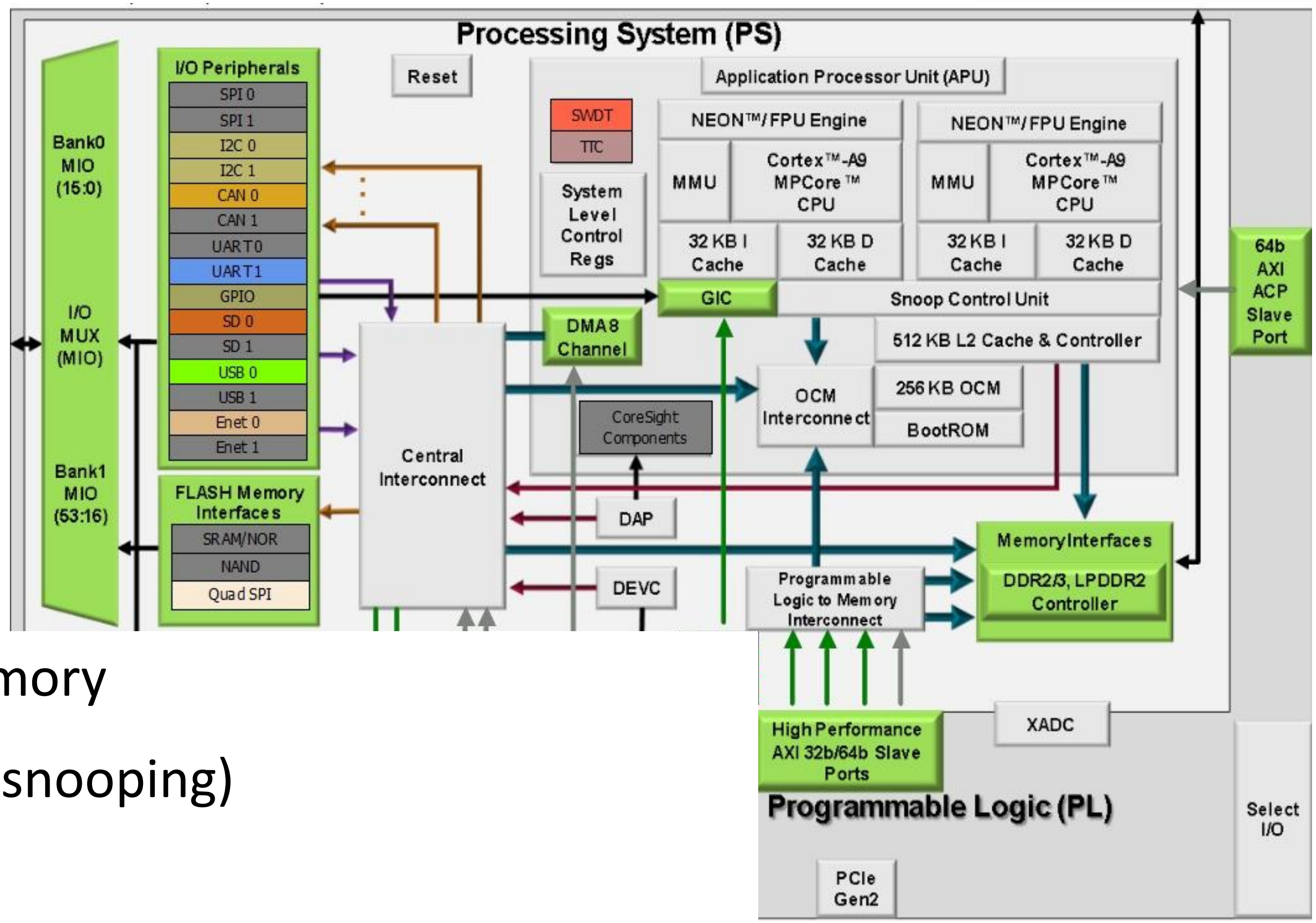
Complex SW platforms

- Multiple kernels and multiple independent instances of a kernel on the same chip
  - Linux
  - RTOS (e.g. FreeRTOS is not multicore!)
- Partitioning of resources
- Boot & life cycle of processing cores and kernels
- Interprocessor/intercore communications
- Interprocess communications (IPC)
- Programming model

Why Linux centered?

- Because of silicon vendors’ support (of the main processor of the chip)
- Because of Linux support of “computing” requirements
- Because Linux already supports, to some extent, heterogeneous architectures

Partitioning of resources



- Central memory
- Cache (and snooping)
- Peripherals
  - ➢ Interrupts (and handling of PIC)
  - ➢ Virtual I/O

Boot

- Pin mux
  - ➢ Consistency with partitioning of resources
  - ➢ Implications on SW factory
- Loading of executable images and coordination with life cycle management
  - ➢ MMU to match relocation address of RTOS-based executable images

Life cycle

- Coordinated start/stop of different kernel instances
- **remoteproc**
  - Developed by TI
  - Master-slave architecture
  - Integrated with rpmsg support of interprocessor/intercore communications
    - ➢ Allocation and initialization of shared memory communication resources
    - ➢ 2 cores can communicate via rpmsg only if one is the remoteproc master of the other
  - Integrated in Linux main branch (master role only)
  - Implemented as a platform driver

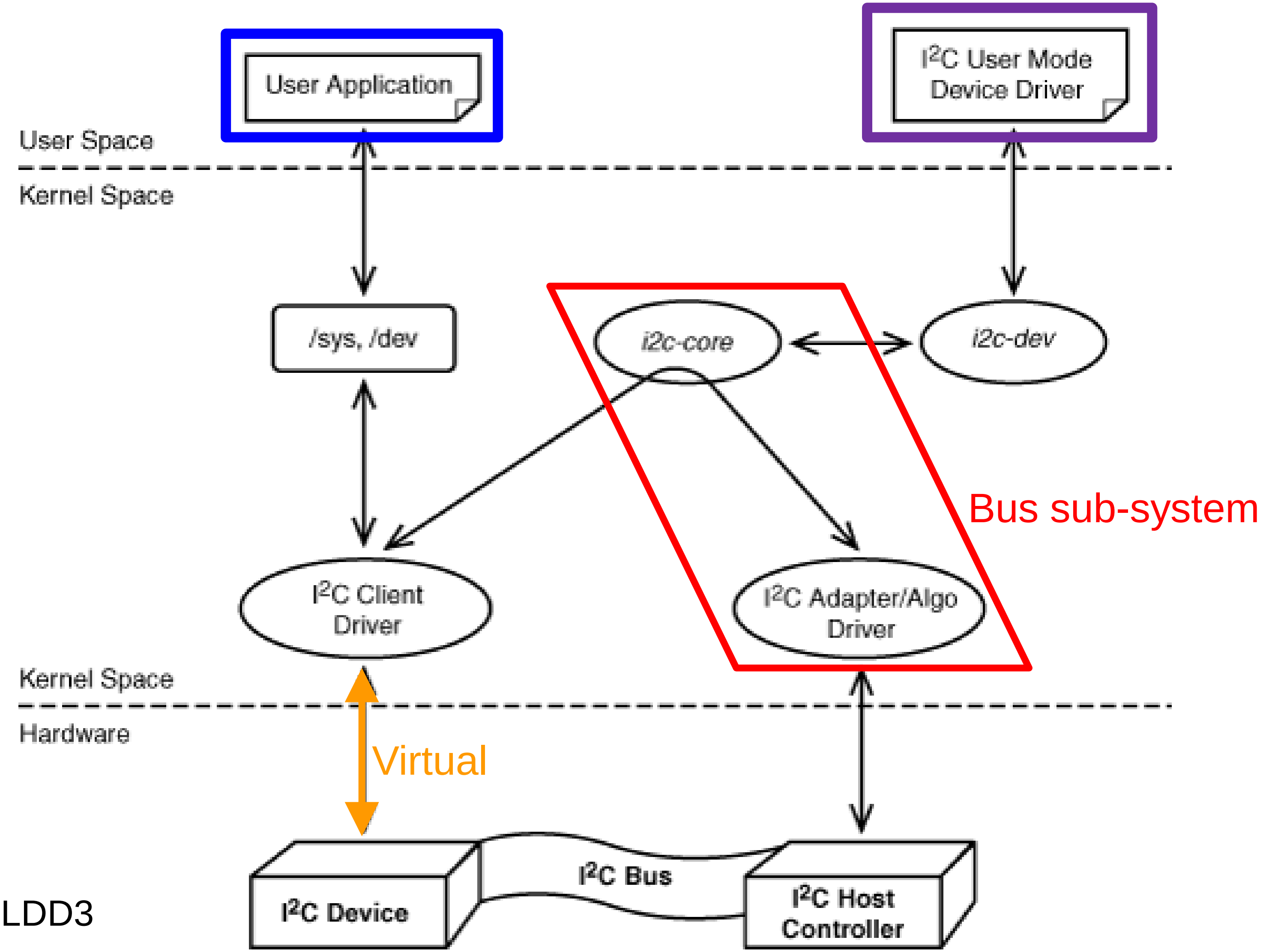
Interprocessor/intercore communications: rpmsg

- Developed by TI
- Based on standard Linux components (virtio)
- Point-to-point architecture between remoteproc master and its remotes (host-device pattern)
- Message style communications, based on circular buffers in shared memory (2 uni-directional vrings per point-to-point connection)
- Cache configuration must guarantee that communicating cores have a coherent view of shared memory
- Integrated in Linux as a bus driver (an I/O subsystem)
  - The API offered by the rpmsg bus driver in Linux is in kernel space!
  - Several client drivers (network or character drivers) can support different transport/application dialogues on the same rpmsg bus

rpmsg communication topology

- Constrained by connection with remoteproc
- Allows only communications between a remoteproc master and each of its remotes
  - 2 uni-directional vrings are created for communications between the remoteproc master and each remote
- No support for routing (e.g. by remoteproc/rpmsg master)
- Supported communication topologies
  - Star
  - Tree (restricted to directly linked nodes)
- Linux support limited to rpmsg master side (center of star)

Linux I/O subsystem (device model)

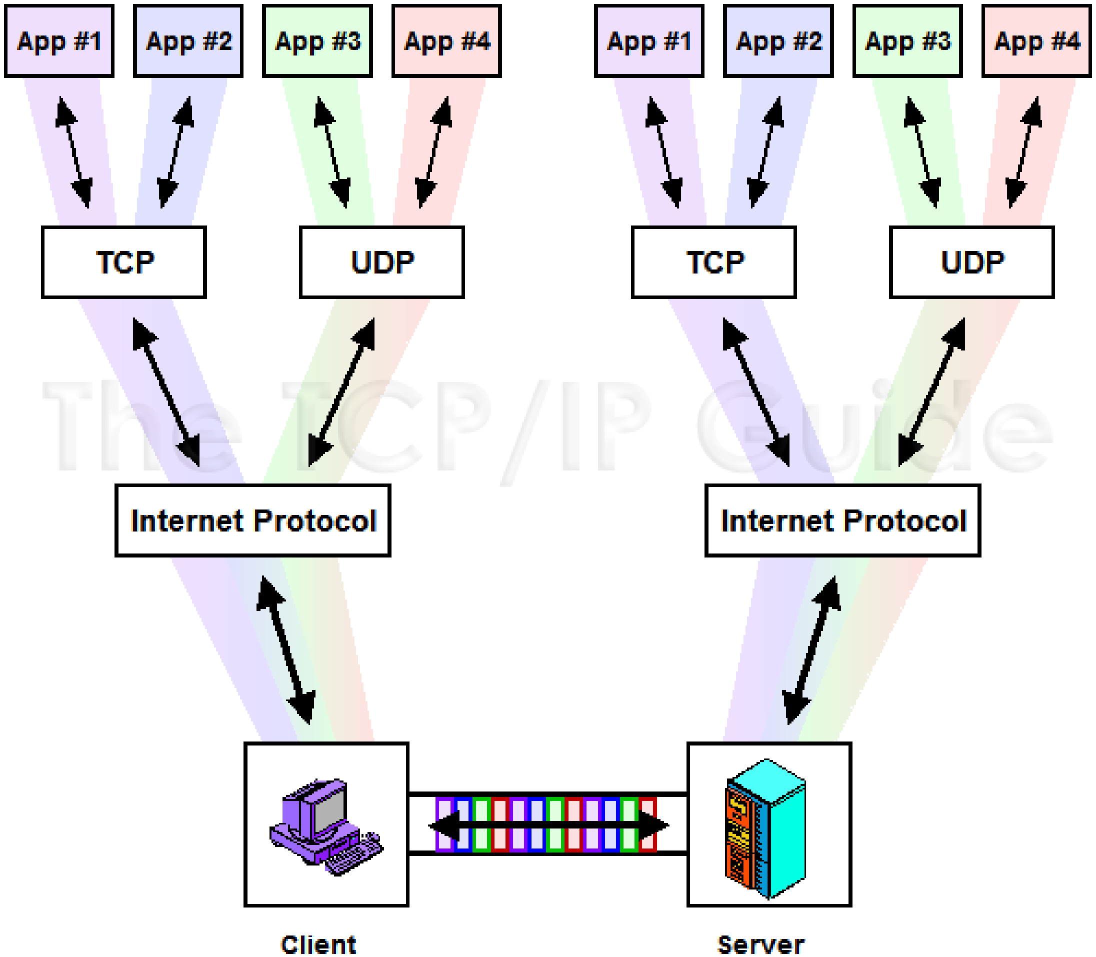


from LDD3

rpmsg service

- Analogous to a Data Link layer service
- De/multiplexing of higher layer services (of rpmsg channels)
  - A remote creates an rmpsg channel by binding to a host provided service (identified by a string, the name of the channel)
  - The channel is then identified in the 2 directions by dynamically created numerical endpoints
  - The channel (host side) identifies also the remote we are communicating with
  - There may multiple active channels on a same vring pair
- Tx side provides reliable/flow-controlled and unreliable/best-effort services
- Rx side expects that when a message is received it is immediately extracted from the circular buffer (is dealt with by higher layer SW)
- Reliability of an rpmsg based transport (channel) service depends on the support of flow-control by the transport protocol!

De/multiplexing



rpmsg service

```
struct rpmsg_driver {
    struct device_driver drv;
    const struct rpmsg_device_id *id_table;
    int (*probe)(struct rpmsg_channel *dev);
    void (*remove)(struct rpmsg_channel *dev);
    void (*callback)(struct rpmsg_channel *, void *data,
                     int len, void *priv, u32 src);
};

static struct rpmsg_device_id rpmsg_ipcproto_id_table[] = {
    { .name    = RPMSG_PROTO_CHANNEL_ID }, // e.g. "rpmsg-ipcproto"
    { },
};
MODULE_DEVICE_TABLE(rpmsg, rpmsg_ipcproto_id_table);

static struct rpmsg_driver rpmsg_ipcproto_driver = {
    .drv.name    = KBUILD_MODNAME,
    .id_table    = rpmsg_ipcproto_id_table,
    .probe       = rpmsg_ipcproto_probe,     // when channel created
    .callback    = rpmsg_ipcproto_cb,        // when data received
    .remove      = rpmsg_ipcproto_remove,
};
```

rpmsg service

```
int register_rpmsg_driver(struct rpmsg_driver
                                   *rpdrv);
```

- Registers an rpmsg driver with the rpmsg bus.
- User should provide a pointer to an `rpmsg_driver` struct, which contains
  - the driver's `->probe()` and `->remove()` functions,
  - an rx callback, and
  - an `id_table` specifying the names of the channels this driver is interested to be probed with (e.g. "`rpmsg-ipcproto`").

rpmsg service

```
int rpmsg_send(struct rpmsg_channel *rpdev,
               void *data, int len);
```

- Sends a message across to the remote processor on a given channel.
- The caller should specify the channel, the data it wants to send and its length (in bytes).
- The message will be sent on the specified channel, i.e. its source and destination address fields will be set to the channel's src and dst addresses (endpoints).
- In case there are no TX buffers available, the function will block until one becomes available (i.e. until the remote processor consumes a tx buffer and puts it back on virtio's used descriptor ring), or a timeout of 15 seconds elapses.
- When the latter happens, `-ERESTARTSYS` is returned.

rpmsg service

```
int rpmsg_trysend(struct rpmsg_channel *rpdev,
                  void *data, int len);
```

- Sends a message across to the remote processor on a given channel.
- The caller should specify the channel, the data it wants to send, and its length (in bytes).
- The message will be sent on the specified channel, i.e. its source and destination address fields will be set to the channel's src and dst addresses.
- In case there are no TX buffers available, the function will immediately return `-ENOMEM` without waiting until one becomes available.

rpmsg service

```
void (*callback)(struct rpmsg_channel *rpdev,
                 void *data, int len, void *priv,
                 u32 src);
```

- Passes over to the client driver associated to the channel `rpdev` the data that have been received by the bus driver, along with opaque structure `priv`.
- The callback function must dispatch `data` so that following messages in the vring can be processed.

Message style IPC service

- Analogous to UDP
  - Users can create their own transport communication endpoints (service access points, analogous to UDP ports)
  - Service access point identified by a numbered port
  - Best-effort
  - Transport protocol `rpmsg_dgproto` similar to UDP
- Implemented by a (client driver) protocol module
- On rpmsg channel "`rpmsg-ipcproto`"
- Interfaced via the socket (system call) API
  - Address family `AF_RPMSG`
  - Socket type `SOCK_DGRAM`

Message style IPC service

Socket layer		
AF_INET protocol stack	AF_SYSTEM protocol stack	AF_RPMSG protocol stack
TCP, UDP, … **IP**		**IPC messaging** rpmsg_dgproto
subnetworks		**rpmsg**

```
struct sockaddr_rpmsg {
    short               srpmsg_family;   // AF_RPMSG
    unsigned short      srpmsg_port;
    unsigned long       srpmsg_addr;
    char                srpmsg_zero[8];
};
```

Why an additional transport layer?

- Multiple application dialogues between two cores
- But why not simply using different rpmsg channels?
  - Users must be able to create their communication end-points
  - Different dialogues may have different requirements: e.g. tcp supports reliable, stream based communications while udp supports best-effort, message based communications
  - Without an additional transport layer protocol rpmsg supports only best-effort, message based communications
  - If we want a `SOCK_SEQPACKET` semantic we need a complex connection oriented protocol that implements flow control

rpmsg IPC: core address

- In our IPC autonomous cores on a chip are addressed via a integer identifier
- When the remote requests the creation of a channel the client driver gets a reference to the channel, and the channel allows to locate the description of the remote in the Device Tree
- The definition of an alias in the Device Tree allows us to associate an rpmsg core address to a remote
- The transport protocol `rpmsg_dgproto` keeps the association rpmsg channel $\leftrightarrow$ core address
  - Tx side: core address $\rightarrow$ rpmsg channel
  - Rx side: rpmsg channel $\rightarrow$ core address

Device Tree

```
{
...
aliases {
        ethernet0 = &gem0;
        serial0 = &uart1;
        spi0 = &qspi;
        rproc1 = &remoteproc1;      // remote core #1
    };
...
remoteproc1: remoteproc@1F000000 {
        compatible = "xlnx,zynq_remoteproc";
        reg = < 0x1F000000 0x1000000 >;
        interrupt-parent = <&intc>;
        interrupts = < 0 37 0 0 38 0 >;
        firmware = "firmware.elf";
        ipino = <0>;
        vring0 = <15>;
        vring1 = <14>;
    };
...
};
```

rpmsg IPC messaging: open issues

- Performance / functionalities
  - Reliable communications
  - Priority and guaranteed bandwidth
  - Max size of messages
  - 0-copy (now 2 copies Linux side)
  - OCM vs. DDR (what shared memory?)
  - Caching?
  - Routing
- API
  - Implementing `SOCK_SEQPACKET` communications
  - Extended semantics

rpmsg on RTOSs

- Focus on FreeRTOS but other RTOSs may be relevant
  - SYSBIOS for TI chips
- Portable implementation provided by OpenAMP
  - Port available for FreeRTOS Xilinx/Freescale
  - Only rpmsg bus
  - No link with upstream Linux community
    (port of bus driver on Linux is in user space!)
- How can we work upstream for FreeRTOS?

Multicore Association

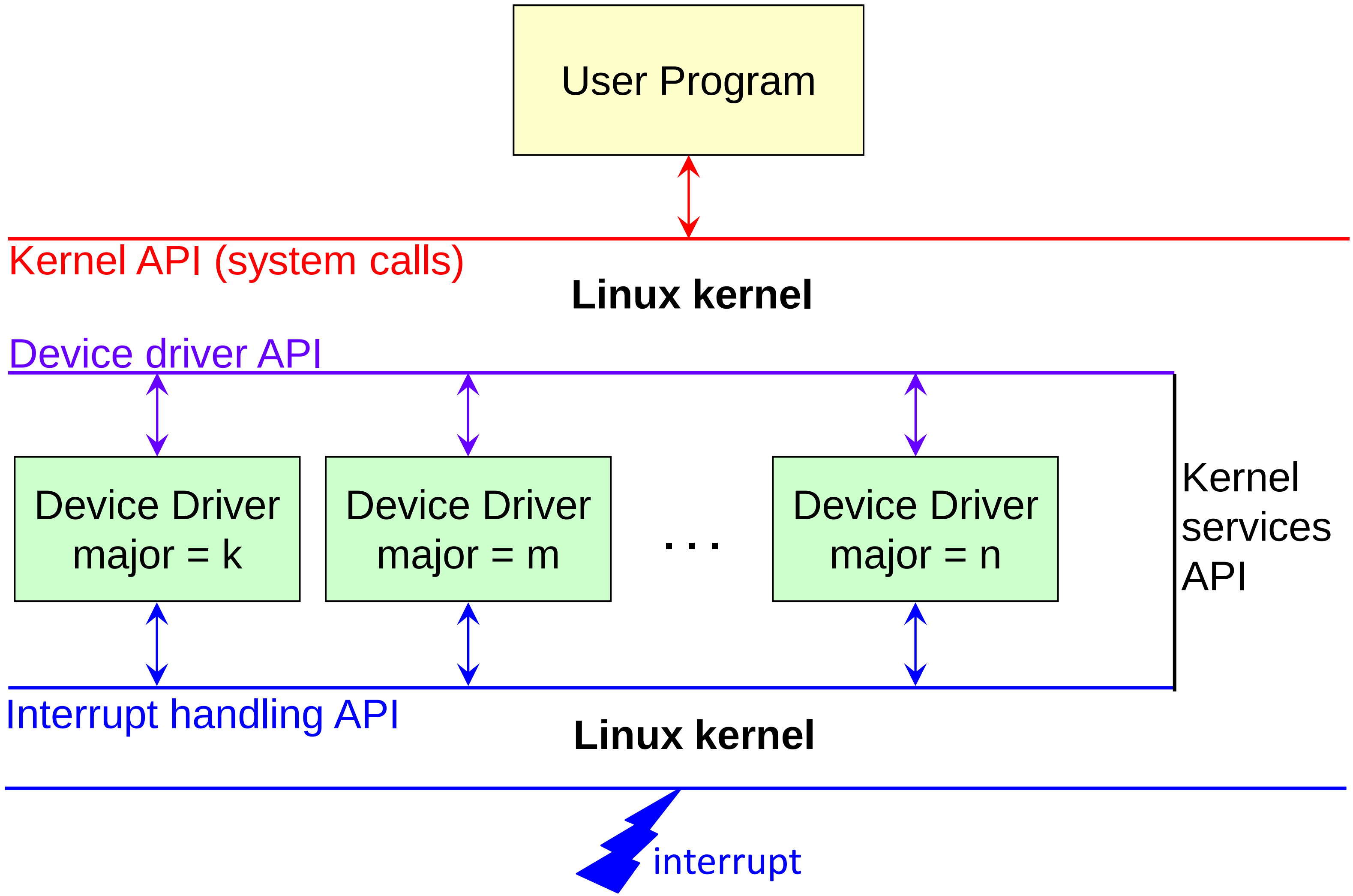
t3lab TECHNOLOGY TRANSFER TEAM

- http://www.multicore-association.org/index.php
- **MCAPI**: the Multicore Communications API specification defines an API and a semantic for communication and synchronization between processing cores in embedded systems
- **OpenAMP**: an open source framework that allows operating systems to interact within a broad range of complex homogeneous and heterogeneous architectures and allows asymmetric multiprocessing applications to leverage parallelism offered by the multicore configuration

rpmsg API in user space

- Master side
- Via a character driver
- 1 major number and N minor numbers
- Each minor number associated to a pair [remote core, rpmsg service]
- Pair [remote core, rpmsg service] associated to a filename in /dev
- Service is best effort

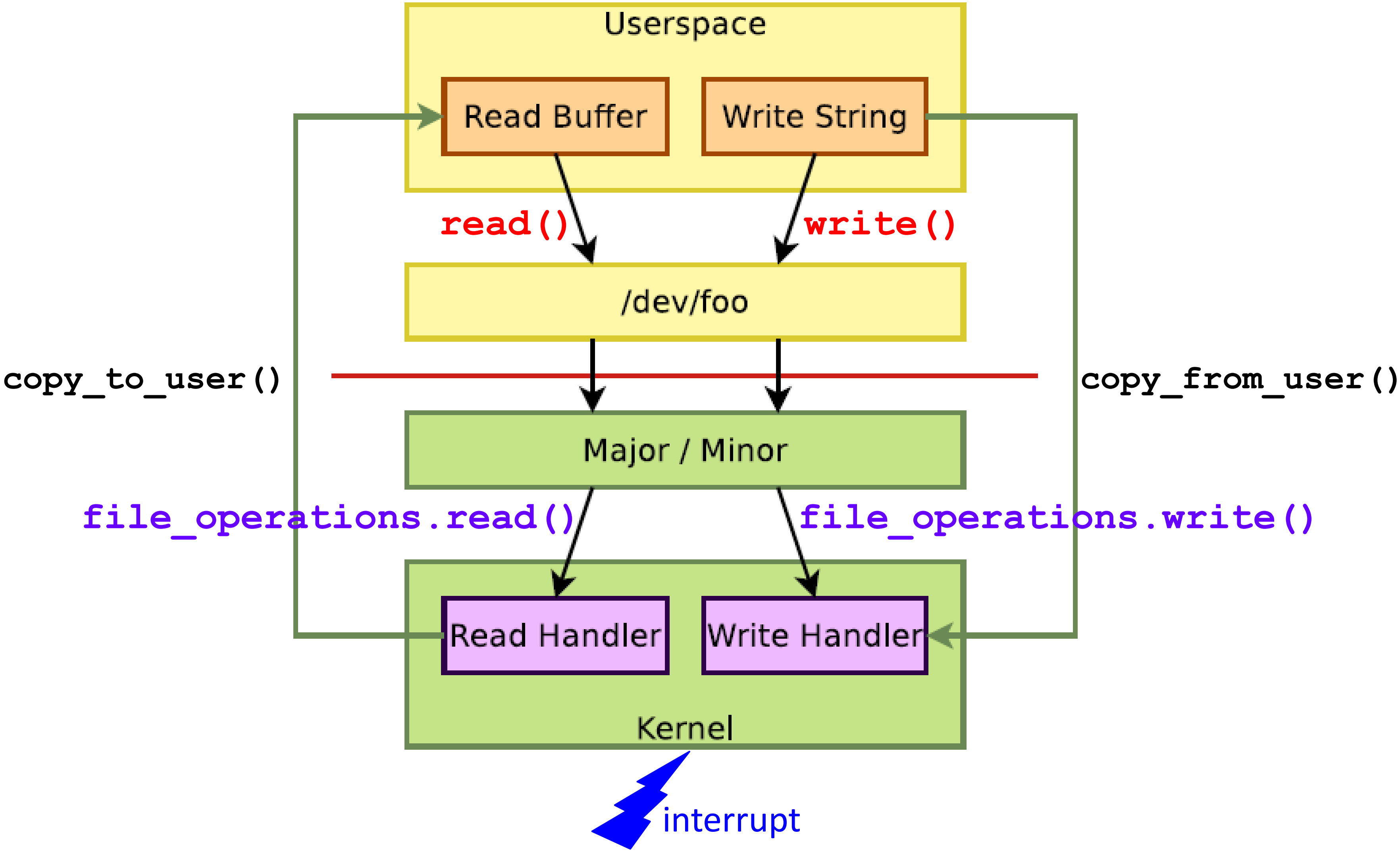
User space, kernel & drivers



Character drivers

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    int (*iterate) (struct file *, struct dir_context *);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
    long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
    int (*mmap) (struct file *, struct vm_area_struct *);
    int (*open) (struct inode *, struct file *);
    int (*flush) (struct file *, fl_owner_t id);
    int (*release) (struct inode *, struct file *);
    int (*fsync) (struct file *, loff_t, loff_t, int datasync);
    int (*aio_fsync) (struct kiocb *, int datasync);
    int (*fasync) (int, struct file *, int);
    int (*lock) (struct file *, int, struct file_lock *);
    ssize_t (*sendpage) (struct file *, struct page *, int, size_t, loff_t *, int);
    unsigned long (*get_unmapped_area)(struct file *, unsigned long, unsigned long,
                                       unsigned long, unsigned long);
    int (*check_flags)(int);
    int (*flock) (struct file *, int, struct file_lock *);
    ssize_t (*splice_write)(struct pipe_inode_info *, struct file *, loff_t *, size_t,
                            unsigned int);
    ssize_t (*splice_read)(struct file *, loff_t *, struct pipe_inode_info *, size_t,
                           unsigned int);
    int (*setlease)(struct file *, long, struct file_lock **);
    long (*fallocate)(struct file *file, int mode, loff_t offset, loff_t len);
    int (*show_fdinfo)(struct seq_file *m, struct file *f);
};
```

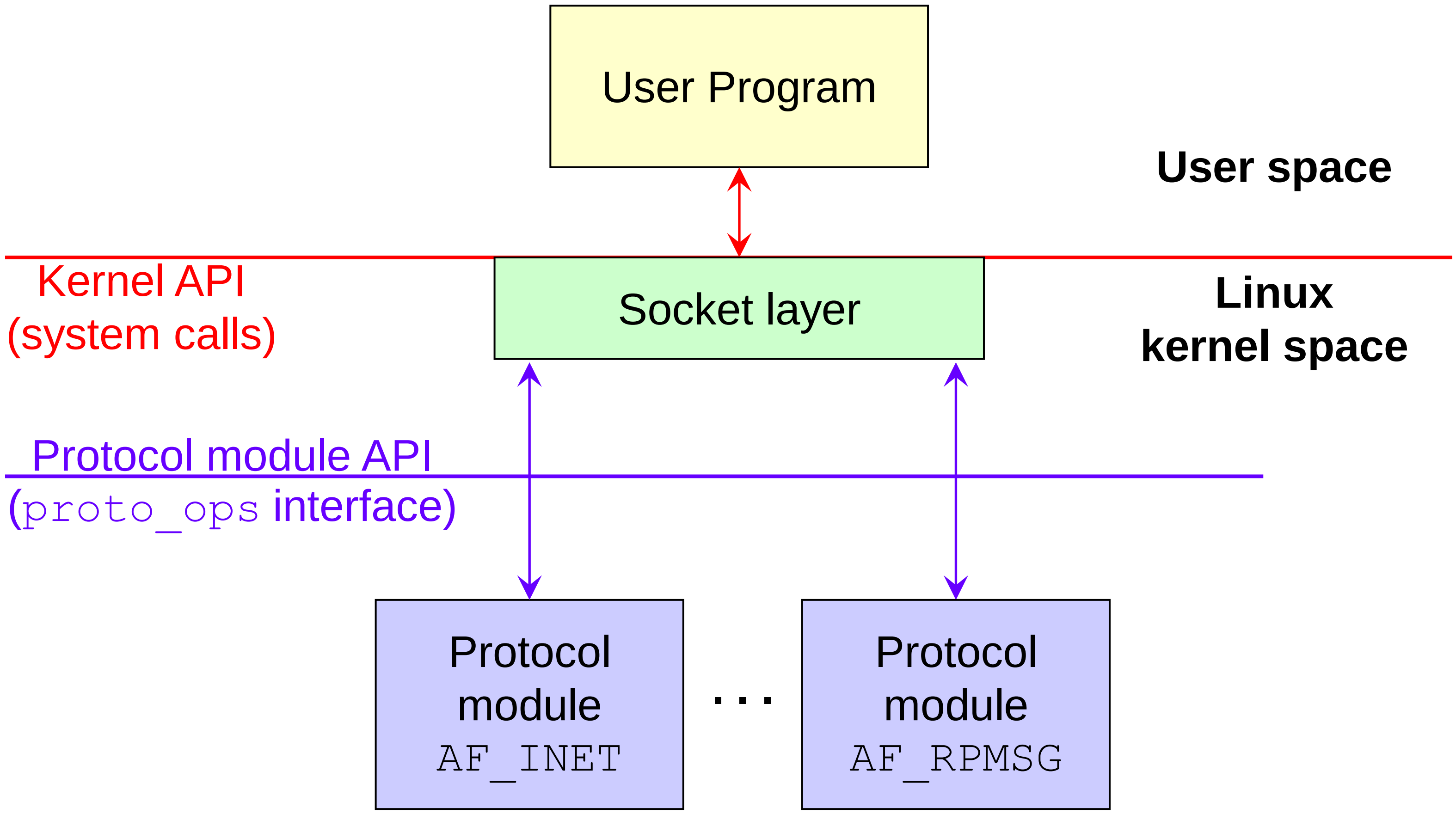
From user space to char drivers



Network protocol

```
static const struct proto_ops rpmsg_sock_ops = {
    .family         = PF_RPMSG,
    .owner          = THIS_MODULE,
    .release        = rpmsg_sock_release,
    .connect        = rpmsg_sock_connect,
    .getname        = rpmsg_sock_getname,
    .sendmsg        = rpmsg_sock_sendmsg,
    .recvmsg        = rpmsg_sock_recvmsg,
    .bind           = rpmsg_sock_bind,
    .poll           = sock_no_poll,
    .listen         = sock_no_listen,
    .accept         = sock_no_accept,
    .ioctl          = sock_no_ioctl,
    .mmap           = sock_no_mmap,
    .socketpair     = sock_no_socketpair,
    .shutdown       = sock_no_shutdown,
    .setsockopt     = sock_no_setsockopt,
    .getsockopt     = sock_no_getsockopt
};
```

Network protocol



Programming model

- Pthreads
  - E.g. manager-worker model:
    manager on Linux, workers on RTOSs
  - `pthread_attr_setaffinity_np()` allows to control on what core a thread is created/run
  - How can we share data between threads working on different cores?
- RPC
- OpenMP

Platform driver / device

- On embedded systems, devices are often not connected through a bus allowing enumeration, hotplugging, and providing unique identifiers for devices.
- However, we still want the devices to be part of the device model.
- The solution to this is the platform driver / platform device infrastructure.
- The platform devices are the devices that are directly connected to the CPU (e.g. memory mapped devices), without any kind of bus.
- https://www.kernel.org/doc/Documentation/driver-model/platform.txt

Platform driver / device

```
struct platform_device {
    const char       *name;
    u32              id;
    struct device    dev;
    u32              num_resources;
    struct resource  *resource;
};

struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*suspend_late)(struct platform_device *,
                        pm_message_t state);
    int (*resume_early)(struct platform_device *);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
};
```

Example .1/9

- From S. A. Edwards, Device Drivers, Columbia University, http://www.cs.columbia.edu/~sedwards/classes/2014/4840/device-drivers.pdf

- **<u>Module's API:</u>**

```
#ifndef _VGA_LED_H
#define _VGA_LED_H

#include <linux/ioctl.h>

#define VGA_LED_DIGITS 8

typedef struct {
  unsigned char digit;       // 0, 1, .. , VGA_LED_DIGITS-1
  unsigned char segments;    // LSB: segment a; MSB: decimal point
} vga_led_arg_t;

#define VGA_LED_MAGIC 'q'

// ioctls and their arguments
#define VGA_LED_WRITE_DIGIT _IOW(VGA_LED_MAGIC, 1, vga_led_arg_t*)
#define VGA_LED_READ_DIGIT _IOWR(VGA_LED_MAGIC, 2, vga_led_arg_t*)

#endif
```

Example .2/9

- **Excerpt of the Device Tree:**

```
lightweight_bridge: bridge@0xff200000 {
    compatible = "simplebus";
    #address-cells = <1>;
    #size-cells = <1>;
    ranges = < 0x0 0xff200000 0x200000 >;
    vga_led: vga_led@0 {
        compatible = "altr,vga_led";
        reg = <0x0 0x8>;
    };
};
```

Example .3/9

- **Driver source code – part 1:**

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_led.h"

#define DRIVER_NAME "vga_led"

struct vga_led_dev {
    struct resource res;          // Resource: our registers
    void __iomem *virtbase;       // Pointer to registers
    u8 segments[VGA_LED_DIGITS];
} dev;

static void write_digit(int digit, u8 segments) {
    iowrite8(segments, dev.virtbase + digit);
    dev.segments[digit] = segments;
}
```

Example .4/9

- **Driver source code – part 2:**

```
static long vga_led_ioctl(struct file *f, unsigned int cmd,
                          unsigned long arg) {
  vga_led_arg_t vla;
  switch (cmd) {
    case VGA_LED_WRITE_DIGIT:
      if (copy_from_user(&vla, (vga_led_arg_t *) arg,
                         sizeof(vga_led_arg_t))) return -EACCES;
      if (vla.digit > 8) return -EINVAL;
      write_digit(vla.digit, vla.segments);
      break;
    case VGA_LED_READ_DIGIT:
      if (copy_from_user(&vla, (vga_led_arg_t *) arg,
                         sizeof(vga_led_arg_t))) return -EACCES;
      if (vla.digit > 8) return -EINVAL;
      vla.segments = dev.segments[vla.digit];
      if (copy_to_user((vga_led_arg_t *) arg, &vla,
                       sizeof(vga_led_arg_t))) return -EACCES;
      break;
    default: return EINVAL;
  }
  return 0;
}
```

Example .5/9

- **Driver source code – part 3:**

```
static const struct file_operations vga_led_fops = {
    .owner = THIS_MODULE,
    .unlocked_ioctl = vga_led_ioctl,
};

// we define our module as a misc device, with its minor
// number dynamically assigned
static struct miscdevice vga_led_misc_device = {
    .minor = MISC_DYNAMIC_MINOR,
    .name = DRIVER_NAME,
    .fops = &vga_led_fops,
};

static int vga_led_remove(struct platform_device *pdev) {
  iounmap(dev.virtbase);
  release_mem_region(dev.res.start, resource_size(&dev.res));
  misc_deregister(&vga_led_misc_device);
  return 0;
}
```

Example .6/9

- **Driver source code – part 4:**

```
static int __init vga_led_probe(struct platform_device *pdev) {
  static unsigned char welcome_message[VGA_LED_DIGITS] = {
              0x3E, 0x7D, 0x77, 0x08, 0x38, 0x79, 0x5E, 0x00};
  int i, ret;
  // Register ourselves as a misc device: creates /dev/vga_led
  ret = misc_register(&vga_led_misc_device);
  // Find our registers in device tree; verify availability
  ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
  if (ret) {
    ret = -ENOENT;
    goto out_deregister;
  }
  if (request_mem_region(dev.res.start, resource_size(&dev.res),
                       DRIVER_NAME) == NULL) {
    ret = EBUSY;
    goto out_deregister;
  }
  // vga_led_probe() continues
```

Example .7/9

- **Driver source code – part 5:**

```
  // Arrange access to our registers (calls ioremap)
  dev.virtbase = of_iomap(pdev->dev.of_node, 0);
  if (dev.virtbase == NULL) {
    ret = -ENOMEM;
    goto out_release_mem_region;
  }
  // Display a welcome message
  for (i = 0; i < VGA_LED_DIGITS; i++) {
    write_digit(i, welcome_message[i]);
  }
  return 0;
out_release_mem_region:
  release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
  misc_deregister(&vga_led_misc_device);
  return ret;
}
```

Example .8/9

- **Driver source code – part 6:**

```
static const struct of_device_id vga_led_of_match[] = {
    { .compatible = "altr,vga_led" },
    {},
};
MODULE_DEVICE_TABLE(of, vga_led_of_match);
static struct platform_driver vga_led_driver = {
    .driver = {
                .name = DRIVER_NAME,
                .owner = THIS_MODULE,
                .of_match_table = of_match_ptr(vga_led_of_match),
    },
    .probe = vga_led_probe,
    .remove = __exit_p(vga_led_remove),
};
static int __init vga_led_init(void) {
  pr_info(DRIVER_NAME ": init\n");
  return return platform_driver_register(&vga_led_driver);
}
static void __exit vga_led_exit(void) {
  platform_driver_unregister(&vga_led_driver);
  pr_info(DRIVER_NAME ": exit\n");
}
```

Example .9/9

- **Driver source code – part 7:**

```
module_init(vga_led_init);
module_exit(vga_led_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");
MODULE_DESCRIPTION("VGA 7-segment LED Emulator");
```

- **notes:**
  1. The init function of the module is `vga_led_init()`.
  2. Function `vga_led_init()` registers a platform driver with `.compatible=="altr,vga_led"` and probe function `vga_led_probe()`.
  3. Because a `compatible` match is found with device tree node `vga_led`, function `vga_led_probe()` is activated (a `platform_device` struct is filled with data extracted from the device tree and passed to the probe function).
  4. Function `vga_led_probe()` registers a miscDevice with the `misc` subsystem (by invoking `misc_register()`), thus completing the initialization of the `vga_led` driver module.
  5. The only file operation supported by the `vga_led` module is `ioctl()`.

Miscellaneous Character Drivers

- MiscDevice is a thin layer around character devices.
- The `misc` driver and all miscDevices are assigned major number 10.
- Minor numbers may be assigned dynamically to miscDevices.
- The `misc` subsystem automatically creates the special file representing a miscDevice in `/dev` directory.
- The `misc` driver exports two functions for user modules to register and unregister (minor numbers must be unique among miscDevices):

```
#include <linux/miscdevice.h>
struct miscdevice {
  int minor;          // MISC_DYNAMIC_MINOR assigns it dynamically
  const char *name;   // for humans, will appear in /proc/misc file
  const struct file_operations *fops;
  struct miscdevice *next, *prev;
                      // must be cleared before registering
};
int misc_register(struct miscdevice *misc);
int misc_deregister(struct miscdevice *misc);
```

remoteproc: device tree excerpt

```
…
remoteproc1: remoteproc@1 {
        compatible = "xlnx,zynq_remoteproc";
        reg = < 0x1F000000 0x1000000 >;
        //reg = < 0x18000000 0x08000000 >;
        interrupt-parent = <&intc>;
        interrupts = < 0 37 0 0 38 0 >;
        firmware = "firmware.elf";
        //status = "disabled";
        ipino = <0>;
        vring0 = <15>;
        vring1 = <14>;
};
…
```

remoteproc: driver excerpt .1/4

```
/*
 * Zynq Remote Processor driver
 *
 * Copyright (C) 2012 Michal Simek <monstr@monstr.eu>
 * Copyright (C) 2012 PetaLogix
 *
 * Based on origin OMAP Remote Processor driver
 *
 * Copyright (C) 2011 Texas Instruments, Inc.
 * Copyright (C) 2011 Google, Inc.
 *
 * This program is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public License
 * version 2 as published by the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */
```

remoteproc: driver excerpt .2/4

```
. . .
/* Match table for OF platform binding */
static const struct of_device_id zynq_remoteproc_match[] = {
             { .compatible = "xlnx,zynq_remoteproc", },
             { /* end of list */ },
};
MODULE_DEVICE_TABLE(of, zynq_remoteproc_match);

static struct platform_driver zynq_remoteproc_driver = {
             .probe = zynq_remoteproc_probe,
             .remove = zynq_remoteproc_remove,
             .driver = {
               .name = "zynq_remoteproc",
               .of_match_table = zynq_remoteproc_match,
             },
};
. . .
```

remoteproc: driver excerpt .3/4

```
. . .
static int zynq_remoteproc_probe(struct platform_device *pdev) {
    . . .
};
. . .
static int zynq_remoteproc_remove(struct platform_device *pdev) {
    . . .
};
. . .
```

remoteproc: driver excerpt .3/4

```
. . .
module_platform_driver(zynq_remoteproc_driver);   // espande:
// module_driver(zynq_remoteproc_driver, platform_driver_register,
//               platform_driver_unregister);     // espande:
// static int __init zynq_remoteproc_driver_init(void) {
//     return platform_driver_register(&zynq_remoteproc_driver);
// }
// module_init(zynq_remoteproc_driver_init);
//
// static void __exit zynq_remoteproc_driver_exit(void) {
//     platform_driver_unregister(&zynq_remoteproc_driver);
// }
// module_exit(zynq_remoteproc_driver_exit);

module_param(firmware, charp, 0);
MODULE_PARM_DESC(firmware, "Override the firmware image name.
Default value in DTS.");

MODULE_AUTHOR("Michal Simek <monstr@monstr.eu");
MODULE_LICENSE("GPL v2");
MODULE_DESCRIPTION("Zynq remote processor control driver");
```

Acknowledgements

- OpenNext is a project co-funded by

  Regione Emilia-Romagna

  thanks to POR FESR funds

- alessio.paccoia@t3lab.it
  michele.rodolfi@t3lab.it
  claudio.salati@t3lab.it
  matteo.sartori@t3lab.it