БУДУЩЕЕ
В НАШИХ
РУКАХ

Драйвер для Memory Extender

Дмитрий Точанский

Группа разработки системного ПО (BSP)

d.tochanskiy@yadro.com

О чем этот рассказ:

1. Расскажу о том, как возник Memory Extender и что это такое
2. Расскажу о том, как развивался драйвер для него
3. Опишу несколько случаев его использования

Описание проблемы

1. Система-на-кристале(СнК) состоит из множества блоков
2. Не всегда есть возможность найти подходящие блоки
3. Предпочтительно использовать знакомые проверенные блоки
4. У нас как раз есть такие блоки

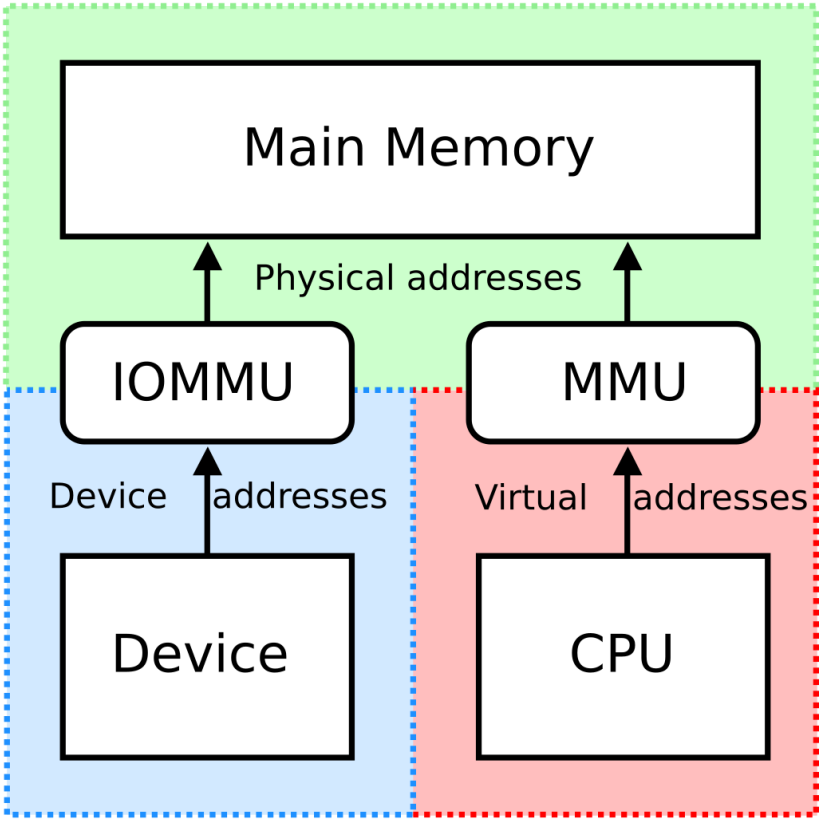
Direct Memory Access (DMA)

1. Отдельный блок устройства
2. Освобождает процессор от несвойственной работы, как, например, при *memcpy()*
3. Позволяет более гибко работать внешним устройствам, например, одновременно
4. Простейший пример:

SOURCE	0x4A000000
DESTINATION	0x5C000000
COUNT	1024

Input/output memory management unit (IOMMU)

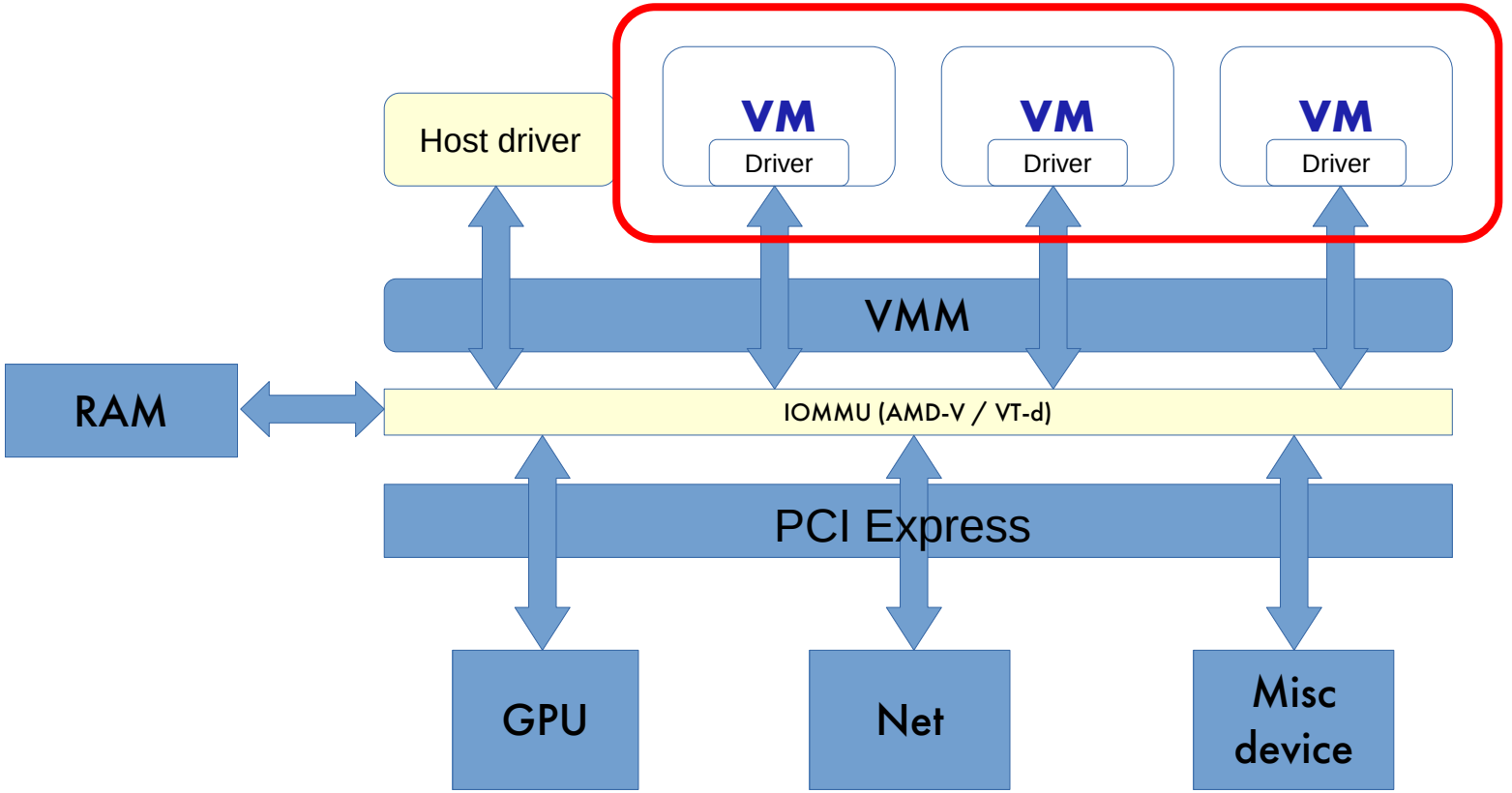
1. Блок управления памятью (MMU) для операций ввода-вывода
2. Позволяет аппаратуре работать с логическими адресами
3. Вносит дополнительные накладные расходы на поддержку таблицы трансляции
4. Широко используется в виртуализации
5. Обеспечивает изоляцию ресурсов



* источник картинки: Wikipedia

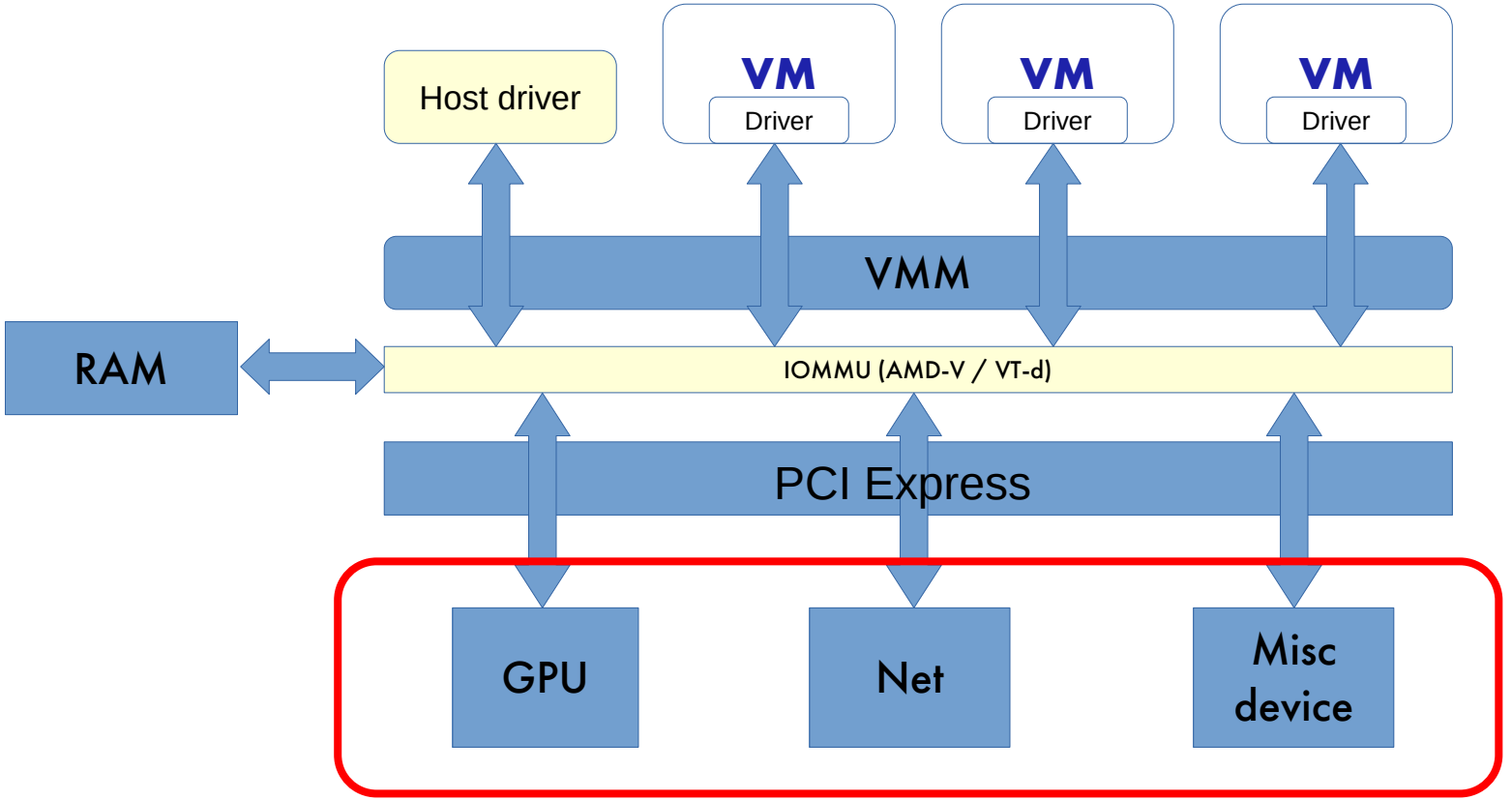
Аппаратные решение

IOMMU в виртаулизации (PCIe Passthrough)



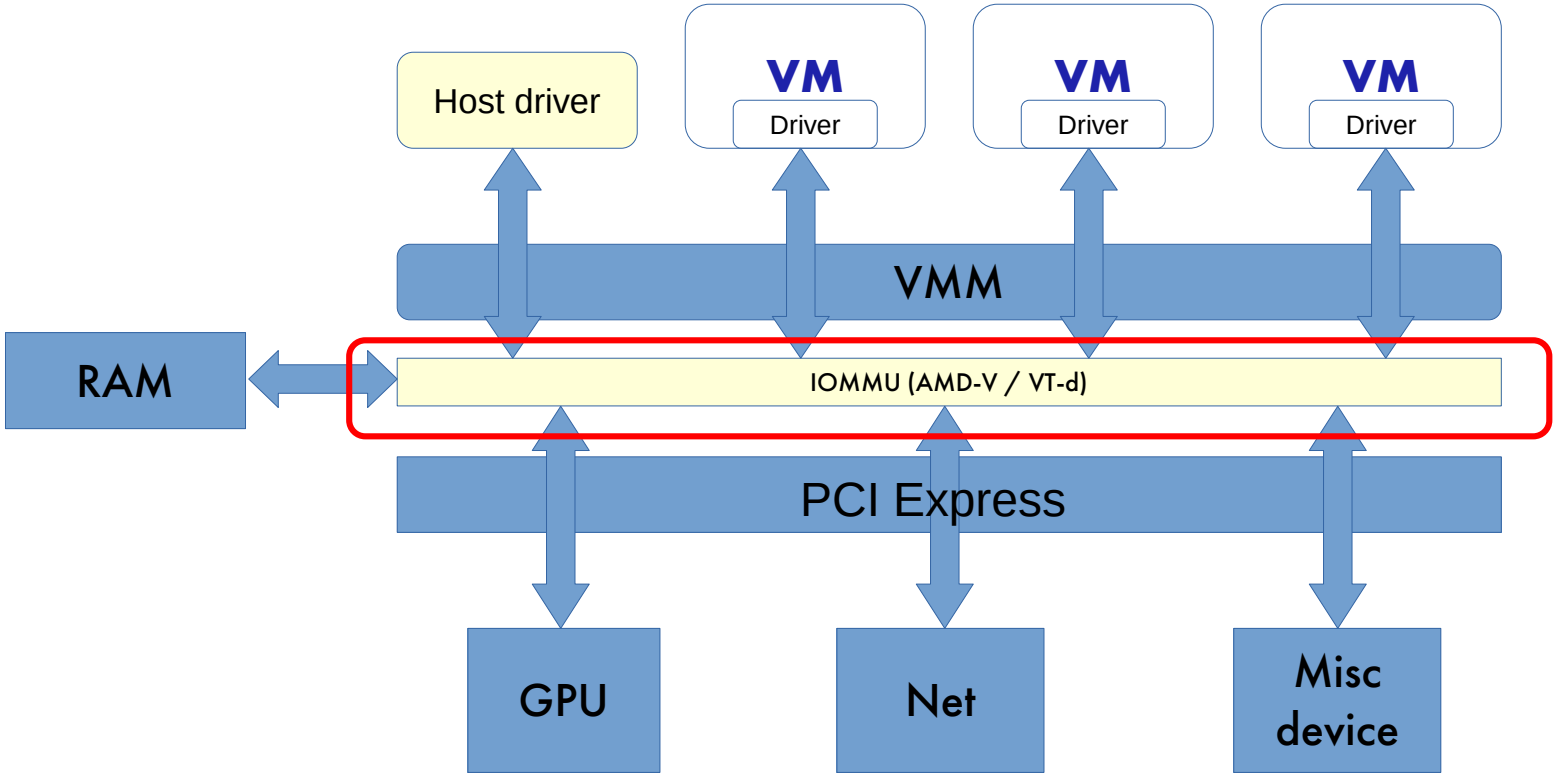
Аппаратные решение

IOMMU в виртаулизации (PCIe Passthrough)



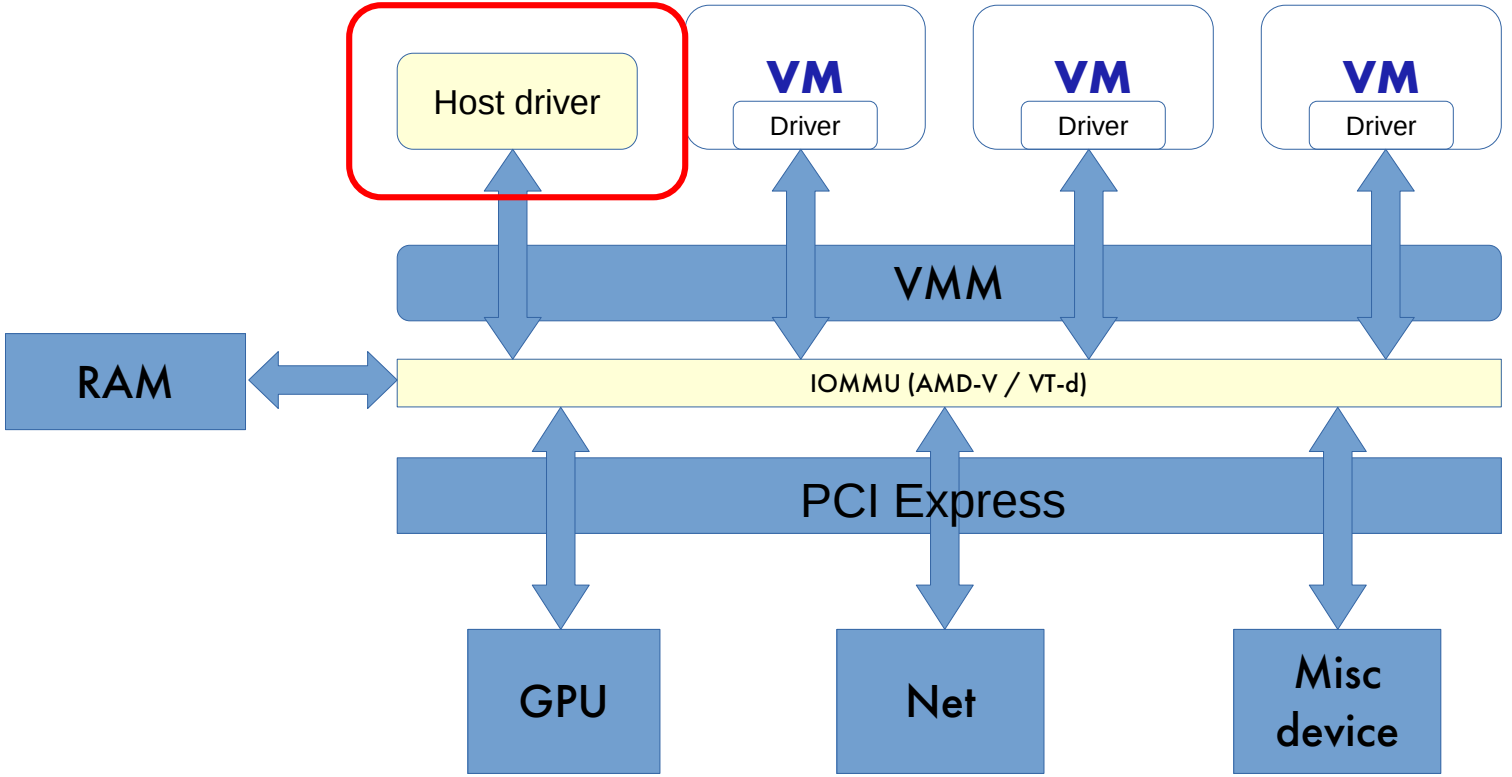
Аппаратные решение

IOMMU в виртаулизации (PCIe Passthrough)



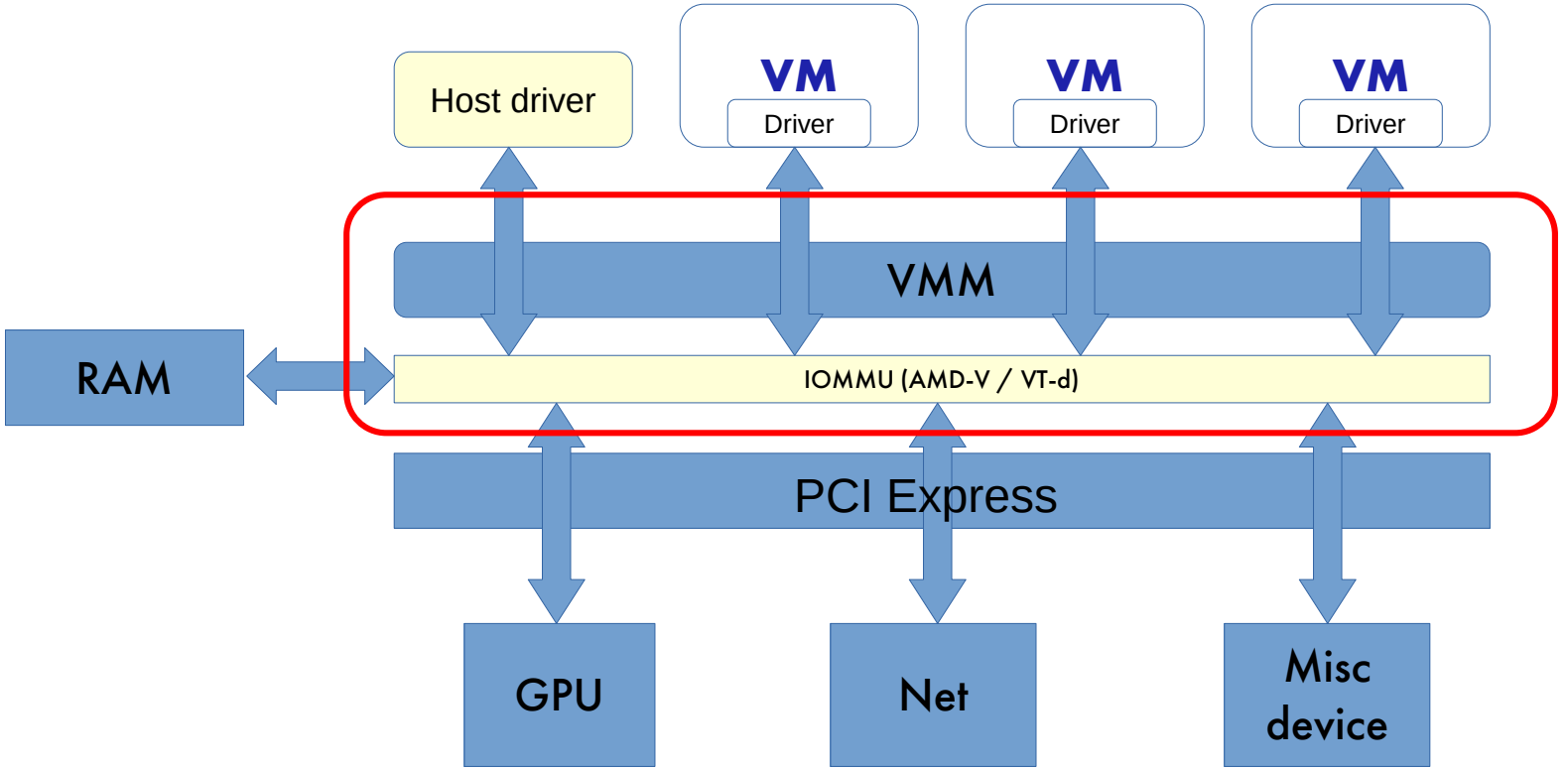
Аппаратные решение

IOMMU в виртаулизации (PCIe Passthrough)



Аппаратные решение

IOMMU в виртаулизации (PCIe Passthrough)

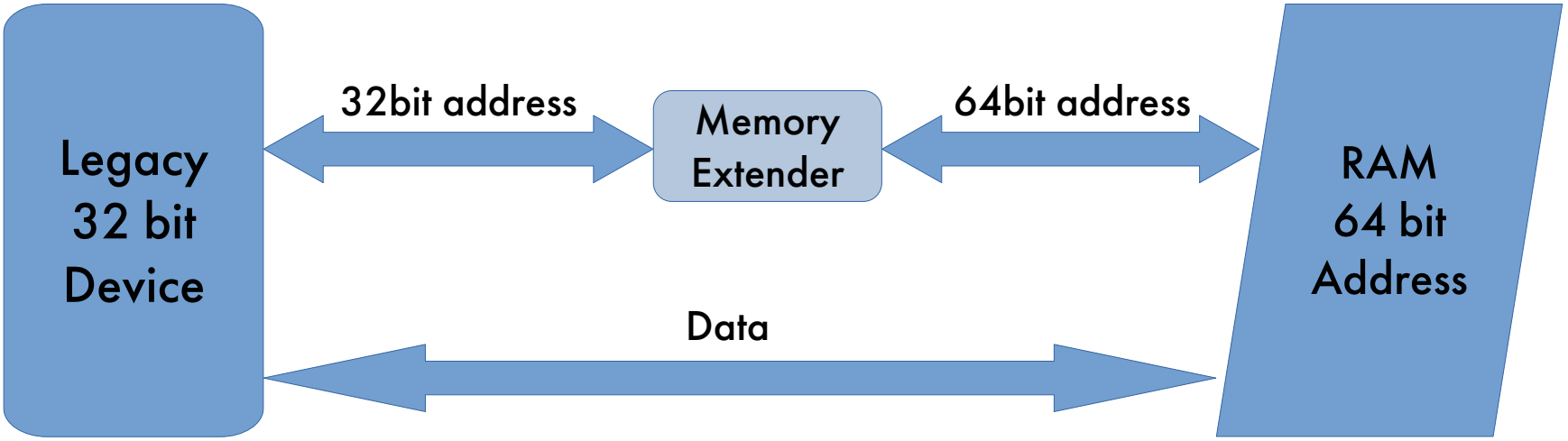


Наше решение

1. Что такое *Memory Extender*
2. Что он делает
3. Как устроен внутри

Аппаратные решение

Схема



Регистры

Таблица дешифрации

000	011000
001	011100
010	011110
011	111000
100	111001
101	111010
110	111100
111	111101

Регистры

32 Bit

01110101...01100

Таблица дешифрации

000	011000
001	011100
010	011110
011	111000
100	111001
101	111010
110	111100
111	111101

Аппаратные решение

Регистры

32 Bit

01110101...01100

Таблица дешифрации

000	011000
001	011100
010	011110
011	111000
100	111001
101	111010
110	111100
111	111101

Регистры

32 Bit

01110101...01100

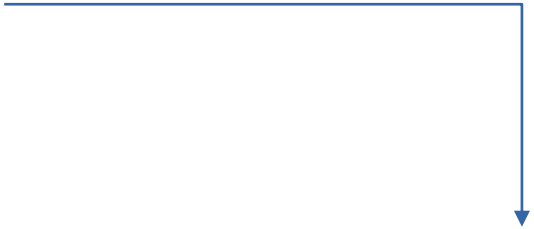


Таблица дешифрации

000	011000
001	011100
010	011110
011	111000
100	111001
101	111010
110	111100
111	111101

Регистры

32 Bit

01110101...01100

Таблица дешифрации

000	011000
001	011100
010	011110
011	111000
100	111001
101	111010
110	111100
111	111101

64 Bit

11100010101...01100

Пример

- 0x1FFFFFFF

Пример

- 0x1FFFFFFF -> 0b00011111111111111111111111111111

Пример

- 0x1FFFFFFF -> 0b00011111111111111111111111111111

Пример

- 0x1FFFFFFF -> 0b00011111111111111111111111111111
- 0x3AA

Пример

- 0x1FFFFFFF -> 0b00011111111111111111111111111111
- 0x3AA → 0b1110101010

Пример

- 0x1FFFFFFF -> 0b00011111111111111111111111111111
- 0x3AA → 0b1110101010
- 0b11101010101111111111111111111111111111

Пример

- 0x1FFFFFFF -> 0b00011111111111111111111111111111
- 0x3AA → 0b1110101010
- 0b11101010101111111111111111111111111111 → 0x755FFFFFF

В режиме baremetal все просто...

```
proc programm_legdev0_mx {}
{
        programm_mx 0x20603000
        programm_mx 0x20604000
        programm_mx 0x20605000
}
```

Настройки memory extender прописываются статически при загрузке системы средствами JTAG

Программное решение

Описание linux DMA

1. Широкий API на все случаи жизни

2. Все бежит, все меняется(например, GFP_DMA32)

```
/* let the implementation decide on the zone to allocate from: */
flag &= ~(__GFP_DMA | __GFP_DMA32 | __GFP_HIGHMEM);
```

Программное решение

Внутреннее устройство драйвера

1. Представляемся iommu (таков путь!)
2. Подсказка: **iommu_setup_dma_ops()**
3. Подготавливаем функции для dma_ops
4. На этапе **.probe_device()** прописываем себя как iommu для устройств, перечисленных в device-tree, а в **.of_xlate()** реально подменяем dma_ops

```
mx_iommu: mx_iommu {
 compatible = "ymp,mx";
 ...
};
```

```
video: video {
 compatible = "ymp,dev";
 ...
 iommus = <&mx_iommu 10>;
 ...
};
```

Предстваляемся iommu

```
struct mx_drv_data {
    struct iommu_device iommu;
    ...
} mx_root;

static struct iommu_ops mx_iommu_ops = {
    .probe_device = mx_probe_device,
    .release_device = mx_release_device,
    .of_xlate = mx_of_xlate,
    ...
};

ret = iommu_device_register(&mx_root→iommu,
        &mx_iommu_ops, dev);
```

Внутренная структура, где будем хранить информацию для себя

Представляемся iommu

```
struct mx_drv_data {
    struct iommu_device iommu;
    ...
} mx_root;

static struct iommu_ops mx_iommu_ops = {
    .probe_device = mx_probe_device,
    .release_device = mx_release_device,
    .of_xlate = mx_of_xlate,
    ...
};

ret = iommu_device_register(&mx_root→iommu,
        &mx_iommu_ops, dev);
```

Внутренная структура, где будем хранить информацию для себя

Джентельменский набор функций, необходимых для работы

Представляемся iommu

```
struct mx_drv_data {
    struct iommu_device iommu;
    ...
} mx_root;

static struct iommu_ops mx_iommu_ops = {
    .probe_device = mx_probe_device,
    .release_device = mx_release_device,
    .of_xlate = mx_of_xlate,
    ...
};

ret = iommu_device_register(&mx_root→iommu,
        &mx_iommu_ops, dev);
```

Внутренная структура, где будем хранить информацию для себя

Джентельменский набор функций, необходимых для работы

Теперь ты ~~в армии~~ iommu, сынок

Программное решение

Теперь все DMA только через нас

```
static struct dma_map_ops mx_dma_ops = {
        .alloc = mx_dma_alloc,
        .free = mx_dma_free,
        ...
};

static int mx_of_xlate(
        struct device *dev,
        struct of_phandle_args *args)
{
                ...
                set_dma_ops(dev, &mx_dma_map_ops);
                ...
}
```

Именно эта функция отвечает за связь нашего драйвера с устройством

Программное решение

Теперь все DMA только через нас

```
static struct dma_map_ops mx_dma_ops = {
        .alloc = mx_dma_alloc,
        .free = mx_dma_free,
        ...
};

static int mx_of_xlate(
        struct device *dev,
        struct of_phandle_args *args)
{
                ...
                set_dma_ops(dev, &mx_dma_map_ops);
                ...
}
```

Структура, содержащая указатели на “перегруженные” функции

Именно эта функция отвечает за связь нашего драйвера с устройством

Простой случай

1. Не было каких-то сложных манипуляций с буферами
2. Адреса просто урезались, в регистры попадало только 32 бита
3. Поставочный драйвер отлично справляется

Простой случай

1. Не было каких-то сложных манипуляций с буферами
2. Адреса просто урезались, в регистры попадало только 32 бита
3. Поставочный драйвер отлично справляется
   - Получили адрес 0x4A0000000 и 0x4B0000000

Простой случай

1. Не было каких-то сложных манипуляций с буферами
2. Адреса просто урезались, в регистры попадало только 32 бита
3. Поставочный драйвер отлично справляется
   - Получили адрес 0x4A0000000 и 0x4B0000000
   - В регистры попало 0xA0000000 и 0xB0000000

Простой случай

1. Не было каких-то сложных манипуляций с буферами
2. Адреса просто урезались, в регистры попадало только 32 бита
3. Поставочный драйвер отлично справляется
   - Получили адрес 0x4A0000000 и 0x4B0000000
   - В регистры попало 0xA0000000 и 0xB0000000
   - Проходя через MX адрес успешно дополняется

SOURCE	0xA0000000
DESTINATION	0xB0000000
COUNT	1024

Случай на производстве

Непростой случай

1. Драйвер появился в апстриме, как часть большой подсистемы
2. Использует **dma_set_mask_and_coherent(dev, DMA_BIT_MASK(32))**
3. Используется арифметика указателей...

Непростой случай

- Получили от ОС указатели 0x4A0000000 и 0x4B0000000

SOURCE	
DESTINATION	
COUNT	
LAST DONE	

Непростой случай

- Получили от ОС указатели 0x4A0000000 и 0x4B0000000
- Уложили в регистры усеченные значения 0xA0000000 и 0xB0000000

SOURCE	0xA0000000
DESTINATION	0xB0000000
COUNT	1024
LAST DONE	

Случай на производстве

Непростой случай

- Получили от ОС указатели 0x4A0000000 и 0x4B0000000
- Уложили в регистры усеченные значения 0xA0000000 и 0xB0000000
- В процессе работы происходит опрос регистров для оценки выполненной работы:

0xA0000100 – 0x4A0000000 = 0xFFFFFFFC00000100

SOURCE	0xA0000000
DESTINATION	0xB0000000
COUNT	1024
LAST DONE	0xA0000100

Случай на производстве

Непростой случай

- Получили от ОС указатели 0x4A0000000 и 0x4B0000000
- Уложили в регистры усеченные значения 0xA0000000 и 0xB0000000
- В процессе работы происходит опрос регистров для оценки выполненной работы:

0xA0000100 – 0x4A0000000 = 0xFFFFFFFC00000100

SOURCE	0xA0000000
DESTINATION	0xB0000000
COUNT	1024
LAST DONE	0xA0000100

У нас появилась поддержка dma_set_mask_and_coherent() :-)

Запасной парашют

Скрытые возможности (надеюсь, не пригодится :)):

1. Средствами device-tree можно заставить драйвер выставить необходимую ему маску dma
2. Используя reserved-memory можно настраивать память для устройства аналогично тому, как это делается в простом режиме

Выводы

1. Удалось создать модуль ядра, для программной эмуляции IOMMU
2. Сэкономили на внедрении и отладке блока IOMMU
3. Интересно провели время, изучая разные подходы в работе драйверов