Resource Table Version 1.0

Previous version:

N.A

Chair

TBD

Technical writers

TBD

Technical reviewers

TBD

Abstract

This document describes the specifications of the resource table used by a main and a secondary device (e.g. a main and a secondary processor) to exchange initial information necessary to interact and coexist.

The purpose of the resource table and this specification is to provide generic structures allowing a secondary device to expose to a main device its needs and supported feature.

Status

First Draft which is mainly a cherry-picking for the Linux kernel source code and documentation.

Revision History

Revision	Date	Editor	Changes Made
V1	???		Initial revision base on Linux kernel and OpenAMP library implementation.

Licensing (To be completed)

Table of Contents

Table of Contents

1. Introduction

The resource table contains system resources that a secondary device requires before it should be powered on, such as allocation of physically contiguous memory, or IOMMU mapping of certain on-chip peripherals. In addition to system resources, the resource table may also contain resource entries that publish the existence of supported features or configurations by the secondary device, such as trace buffers and supported virtio devices.

1.1. Normative References

[RFC2119] Bradner S., “Key words for use in RFCs to Indicate Requirement Levels”, BCP 14, RFC 2119, March 1997.

http://www.ietf.org/rfc/rfc2119.txt1.3

[kernel.org] Linux source code

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/

[VirtIO] Virtio specification version 1.1

http://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.pdf

1.2. Terminology

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in [RFC2119].

Definitions

- **Device Address (DA)**: Physical address mapped in the secondary device.
- **Main device** : Device in charge of the system resource allocations. The main device interprets the resource table provided by the secondary device and update it in consequence.

  Is device the good term? Should it be context, entity, processor, else?
- **Physical Address (PA)**: Physical address mapped in the main device.

- **Secondary device**: Device that expose its system resource requirement and supported features through the resource table.
- **VirtIO**: The Virtual Input/output an abstraction layer originally developed for para-virtualization that expose a virtual device.
- **VRING** : VirtIO buffer management component which is a ring data structure to manage buffer descriptors located in shared memory.

1.3. Structure Specifications (get from Virtio spec 1.1)

Many in-memory structure layouts are documented using the C struct syntax. All structures are assumed to be without additional padding. To stress this, cases where common C compilers are known to insert extra padding within structures are tagged using the GNU C __attribute__((packed)) syntax.

For the integer data types used in the structure definitions, the following conventions are used:

u8, u16, u32, u64
An unsigned integer of the specified length in bits.

le16, le32, le64
An unsigned integer of the specified length in bits, in little-endian byte order.

be16, be32, be64
An unsigned integer of the specified length in bits, in big-endian byte order.

Some of the fields to be defined in this specification do not start or do not end on a byte boundary. Such fields are called bit-fields. A set of bit-fields is always a sub-division of an integer typed field.

Bit-fields within integer fields are always listed in order, from the least significant to the most significant bit. The bit-fields are considered unsigned integers of the specified width with the next in significance relationship of the bits preserved.

For example:

```
struct S {
	be16 {
		A : 15;
		B : 1;
	} x;
	be16y;
};
```

documents the value A stored in the low 15 bit of *x* and the value B stored in the high bit of *x*, the 16-bit integer *x* in turn stored using the big-endian byte order at the beginning of the structure S, and being followed immediately by an unsigned integer *y* stored in big-endian byte order at an offset of 2 bytes (16 bits) from the beginning of the structure.

Note that this notation somewhat resembles the C bitfield syntax but should not be naively converted to a bitfield notation for portable code: it matches the way bitfields are packed by C compilers on little-endian architectures but not the way bitfields are packed by C compilers on big-endian architectures.

Assuming that CPU_TO_BE16 converts a 16-bit integer from a native CPU to the big-endian byte order, the following is the equivalent portable C code to generate a value to be stored into *x*:

```
CPU_TO_BE16(B << 15 | A)
```

2. Resource table structure (get from kernel.org)

2.1. Resource table header

The resource table structure MUST start with a header structure. This structure contains:

- **ver**: a version number that is the first element of the resource table reflecting the resource table revision,
- **num**: the number of resource table entries,
- **offset**: a table containing address offsets that point to the structure describing the resource entry.

```
struct resource_table {
        u32 ver;
        u32 num;
        u32 reserved[2];
        u32 offset[0];
 } __packed;
```

2.2. Resource table entries

Immediately following the header are the resource entries themselves, each of which begins with the following resource entry header:

- The resource type,
- The resource data according to the resource type.

```
struct fw_rsc_hdr {
        u32 type;
        u8 data[0];
 } __packed;
```

Some resources entries are mere announcements, where the main device is informed of specific secondary device configuration. Other entries require the main device to do something (e.g. allocate a system resource). Sometimes a negotiation is expected, where the firmware requests a resource, and once allocated,

The main device should provide back its details (e.g. address of an allocated memory region).

Various resource types that are currently supported:

Resource name	Description	value
RSC_CARVEOUT	Request for allocation of a physically contiguous memory region.	0
RSC_DEVMEM	Request to iommu_map a memory-based peripheral.	1
RSC_TRACE	Announces the availability of a trace buffer into which the secondary device will be writing logs.	2
RSC_VDEV	Declare support for a virtio device and serve as its virtio header.	3
RSC_VENDOR_START	Start of the vendor specific resource types range.	128
RSC_VENDOR_END	End of the vendor specific resource types range.	512

Please note that these values are used as indices to the rproc_handle_rsc lookup table, an RSC_LAST value is used to check the validity of an index before the lookup table is accessed.

```
enum fw_resource_type {
        RSC_CARVEOUT        = 0,
        RSC_DEVMEM          = 1,
        RSC_TRACE           = 2,
        RSC_VDEV            = 3,
        RSC_LAST            = 4,
        RSC_VENDOR_START= 128,
        RSC_VENDOR_END    = 512,
 };
```

2.2.1. Carveout resource

This resource entry requests the main device to allocate a physically contiguous memory region.

These request entries should precede other firmware resource entries, as other entries might request placing other data objects inside these memory regions (e.g. data/code segments, trace resource entries, ...).

Allocating memory this way helps utilizing the reserved physical memory (e.g. CMA) more efficiently, and also minimizes the number of TLB entries needed to map it (in case the system is using an IOMMU). Reducing the TLB pressure is important; it may have a substantial impact on performance.

If the firmware is compiled with static addresses, then **da** should specify the expected device address of this memory region. If **da** is set to FW_RSC_ADDR_ANY (-1), then the main device will dynamically allocate it, and then overwrite **da** with the dynamically allocated address.

We will always use **da** to negotiate the device addresses, even if it isn't using an IOMMU . In that case, though, it will obviously contain physical addresses.

Some secondary devices need to know the allocated physical address even if they do use an IOMMU. This is needed, e.g., if they control hardware accelerators which access the physical memory directly (this is the case with OMAP4 for instance). In that case, the main device will overwrite **pa** with the dynamically allocated physical address. Generally, we don't want to expose physical addresses if we don't have to (secondary devices are generally _not_ trusted), so we might want to change this to happen _only_ when explicitly required by the hardware.

flags is used to provide IOMMU protection flags, and **name** should (optionally) contain a human readable name of this carveout region (mainly for debugging purposes).

```
struct fw_rsc_carveout {
        u32 da;
        u32 pa;
        u32 len;
        u32 flags;
        u32 reserved;
        u8 name[32];
} __packed;
```

2.2.2. Device memory resource

This resource entry requests the main device to IOMMU map a physically contiguous memory region. This is needed in case the secondary device requires access to certain memory-based peripherals; _never_ use it to access regular memory.

This is obviously only needed if the secondary device is accessing memory via an IOMMU.

- **da:** should specify the required device address,
- **pa:** should specify the physical address we want to map,
- **len:** should specify the size of the mapping,
- **flags:** is the IOMMU protection flags.
- **Name:** may (optionally) contain a human readable name of this mapping (mainly for debugging purposes).

Note: at this point we just "trust" those devmem entries to contain valid physical addresses, but this isn't safe and will be changed: eventually we want main device implementations to provide us ranges of physical addresses the firmware is allowed to request, and not allow firmware to request access to physical addresses that are outside those ranges.

```
struct fw_rsc_devmem {
        u32 da;
        u32 pa;
        u32 len;
        u32 flags;
        u32 reserved;
        u8 name[32];
} __packed;
```

2.2.3. Trace resource

This resource entry provides the main device information about a trace buffer into which the secondary device will write log messages.

- **da:** specifies the device address of the buffer,
- **len:** specifies its size,
- **name:** may contain a human readable name of the trace buffer.

```
struct fw_rsc_trace {
        u32 da;
        u32 len;
        u32 reserved;
        u8 name[32];
} __packed;
```

2.2.4. Virtio device resource

This resource is a virtio device header: it provides information about the virtio device and is then used by the main device and its peer secondary devices to negotiate and share certain virtio properties.

By providing this resource entry, the firmware essentially asks the host to allocate a virtio device. unlike virtualization systems, the term 'host' here means the main device side which is controlling the secondary devices.

The name '**gfeatures**' is used to comply with virtio's terms, though there is not really any virtualized guest OS here: it's the host which is responsible for negotiating the final features.

- **id**: virtio device id as defined in [VirtIO],
- **notifyid**: is a unique notify index for this vdev. This notify index is used when kicking a secondary device, to let it know that the status/features of this vdev have changes,
- **dfeatures**: specifies the virtio device features supported by the secondary device,
- **gfeatures**: is a place holder used by the host to write back the negotiated features that are supported by both sides,
- **config_len**: is the size of the virtio config space of this vdev. The config space lies in the resource table immediate after this fw_rsc_vdev structure. For more information, read [VirtIO] specification,
- **status**: is a place holder where the host will indicate its virtio progress,
- **num_of_vrings**: indicates how many vrings are described in this vdev structure.

```
struct fw_rsc_vdev {
        u32 id;
        u32 notifyid;
        u32 dfeatures;
        u32 gfeatures;
        u32 config_len;
        u8 status;
        u8 num_of_vrings;
        u8 reserved[2];
        struct fw_rsc_vdev_vring vring[]; ( Is it ANSI-C?)
} __packed;
```

2.2.5. virtio ring resource

This descriptor is not a resource entry by itself; it is part of the vdev resource type (see below).

Note that da should either contain the device address where the secondary device is expecting the vring, or indicate (if set to FW_RSC_ADDR_ANY) that dynamically allocation of the vring's device address is supported.

- **da**: device address,
- **align**: the alignment between the consumer and producer parts of the vring,
- **num**: num of buffers supported by this vring (must be power of two),
- **notifyid** is a unique notify index for this vring. This notify index is used when kicking a secondary device, to let it know that this vring is triggered.

```
struct fw_rsc_vdev_vring {
        u32 da;
        u32 align;
        u32 num;
        u32 notifyid;
        u32 pa;
} __packed;
```