Ramoops oops/panic logger

Sergiu Iordache <sergiu@chromium.org>

Updated: 10 Feb 2021

Introduction

Ramoops is an oops/panic logger that writes its logs to RAM before the system crashes. It works by logging oopses and panics in a circular buffer. Ramoops needs a system with persistent RAM so that the content of that area can survive after a restart.

Ramoops concepts

Ramoops uses a predefined memory area to store the dump. The start and size and type of the memory area are set using three variables:

- `mem_address` for the start
- `mem_size` for the size. The memory size will be rounded down to a power of two.
- `mem_type` to specifiy if the memory type (default is pgprot_writecombine).

Typically the default value of `mem_type=0` should be used as that sets the pstore mapping to pgprot_writecombine. Setting `mem_type=1` attempts to use `pgprot_noncached`, which only works on some platforms. This is because pstore depends on atomic operations. At least on ARM, pgprot_noncached causes the memory to be mapped strongly ordered, and atomic operations on strongly ordered memory are implementation defined, and won't work on many ARMs such as omaps. Setting `mem_type=2` attempts to treat the memory region as normal memory, which enables full cache on it. This can improve the performance.

The memory area is divided into `record_size` chunks (also rounded down to power of two) and each kmesg dump writes a `record_size` chunk of information.

Limiting which kinds of kmsg dumps are stored can be controlled via the `max_reason` value, as defined in include/linux/kmsg_dump.h's `enum kmsg_dump_reason`. For example, to store both Oopses and Panics, `max_reason` should be set to 2 (KMSG_DUMP_OOPS), to store only Panics `max_reason` should be set to 1 (KMSG_DUMP_PANIC). Setting this to 0 (KMSG_DUMP_UNDEF), means the reason filtering will be controlled by the `printk.always_kmsg_dump` boot param: if unset, it'll be KMSG_DUMP_OOPS, otherwise KMSG_DUMP_MAX.

The module uses a counter to record multiple dumps but the counter gets reset on restart (i.e. new dumps after the restart will overwrite old ones).

Ramoops also supports software ECC protection of persistent memory regions. This might be useful when a hardware reset was used to bring the machine back to life (i.e. a watchdog triggered). In such cases, RAM may be somewhat corrupt, but usually it is restorable.

Setting the parameters

Setting the ramoops parameters can be done in several different manners:

A. Use the module parameters (which have the names of the variables described as before). For quick debugging, you can also reserve parts of memory during boot and then use the reserved memory for ramoops. For example, assuming a machine with > 128 MB of memory, the following kernel command line will tell the kernel to use only the first 128 MB of memory, and place ECC-protected ramoops region at 128 MB boundary:

```
mem=128M ramoops.mem_address=0x8000000 ramoops.ecc=1
```

B. Use Device Tree bindings, as described in `Documentation/devicetree/bindings/reserved-memory/ramoops.yaml`. For example:

```
reserved-memory {
        #address-cells = <2>;
        #size-cells = <2>;
        ranges;

        ramoops@8f000000 {
                compatible = "ramoops";
                reg = <0 0x8f000000 0 0x100000>;
                record-size = <0x4000>;
                console-size = <0x4000>;
        };
};
```

C. Use a platform device and set the platform data. The parameters can then be set through that platform data. An example of doing that is:

```
#include <linux/pstore_ram.h>
```

```
[...]

static struct ramoops_platform_data ramoops_data = {
      .mem_size               = <...>,
      .mem_address            = <...>,
      .mem_type               = <...>,
      .record_size            = <...>,
      .max_reason             = <...>,
      .ecc                    = <...>,
};

static struct platform_device ramoops_dev = {
      .name = "ramoops",
      .dev = {
              .platform_data = &ramoops_data,
      },
};

[... inside a function ...]
int ret;

ret = platform_device_register(&ramoops_dev);
if (ret) {
      printk(KERN_ERR "unable to register platform device\n");
      return ret;
}
```

You can specify either RAM memory or peripheral devices' memory. However, when specifying RAM, be sure to reserve the memory by issuing memblock_reserve() very early in the architecture code, e.g.:

```
#include <linux/memblock.h>

memblock_reserve(ramoops_data.mem_address, ramoops_data.mem_size);
```

Dump format

The data dump begins with a header, currently defined as `====` followed by a timestamp and a new line. The dump then continues with the actual data.

Reading the data

The dump data can be read from the pstore filesystem. The format for these files is `dmesg-ramoops-N`, where N is the record number in memory. To delete a stored record from RAM, simply unlink the respective pstore file.

Persistent function tracing

Persistent function tracing might be useful for debugging software or hardware related hangs. The functions call chain log is stored in a `ftrace-ramoops` file. Here is an example of usage:

```
# mount -t debugfs debugfs /sys/kernel/debug/
# echo 1 > /sys/kernel/debug/pstore/record_ftrace
# reboot -f
[...]
# mount -t pstore pstore /mnt/
# tail /mnt/ftrace-ramoops
0 ffffffff8101ea64  ffffffff8101bcda  native_apic_mem_read <- disconnect_bsp_APIC+0x6a/0xc0
0 ffffffff8101ea44  ffffffff8101bcf6  native_apic_mem_write <- disconnect_bsp_APIC+0x86/0xc0
0 ffffffff81020084  ffffffff8101a4b5  hpet_disable <- native_machine_shutdown+0x75/0x90
0 ffffffff81005f94  ffffffff8101a4bb  iommu_shutdown_noop <- native_machine_shutdown+0x7b/0x90
0 ffffffff8101a6a1  ffffffff8101a437  native_machine_emergency_restart <- native_machine_restart+0x37/0x
0 ffffffff811f9876  ffffffff8101a73a  acpi_reboot <- native_machine_emergency_restart+0xaa/0x1e0
0 ffffffff8101a514  ffffffff8101a772  mach_reboot_fixups <- native_machine_emergency_restart+0xe2/0x1e0
0 ffffffff811d9c54  ffffffff8101a7a0  __const_udelay <- native_machine_emergency_restart+0x110/0x1e0
0 ffffffff811d9c34  ffffffff811d9c80  __delay <- __const_udelay+0x30/0x40
0 ffffffff811d9d14  ffffffff811d9c3f  delay_tsc <- __delay+0xf/0x20
```