Linux Powerpc Documentation

The kernel development community

Jul 14, 2020

CONTENTS

CHAPTER

ONE

THE POWERPC BOOT WRAPPER

PowerPC image targets compresses and wraps the kernel image (vmlinux) with a boot wrapper to make it usable by the system firmware. There is no standard PowerPC firmware interface, so the boot wrapper is designed to be adaptable for each kind of image that needs to be built.

The boot wrapper can be found in the arch/powerpc/boot/ directory. The Makefile in that directory has targets for all the available image types. The different image types are used to support all of the various firmware interfaces found on PowerPC platforms. OpenFirmware is the most commonly used firmware type on general purpose PowerPC systems from Apple, IBM and others. U-Boot is typically found on embedded PowerPC hardware, but there are a handful of other firmware implementations which are also popular. Each firmware interface requires a different image format.

The boot wrapper is built from the makefile in arch/powerpc/boot/Makefile and it uses the wrapper script (arch/powerpc/boot/wrapper) to generate target image. The details of the build system is discussed in the next section. Currently, the following image format targets exist:

cuImage.%	Backwards compatible uImage for older version of U-Boot (for versions that don't understand the device tree). This image embeds a device tree blob inside the image. The boot wrapper, kernel and device tree are all embedded inside the U-Boot uImage file format with boot wrapper code that extracts data from the old bd_info structure and loads the data into the device tree before jumping into the kernel. Because of the series of #ifdefs found in the bd_info structure used in the old U-Boot interfaces, cuImages are platform specific. Each specific U-Boot platform has a different platform init file which populates the embedded device tree with data from the platform specific bd_info file. The platform specific cuImage platform init code can be found in arch/powerpc/boot/cuboot.*.c. Selection of the correct cuImage init code for a specific board can be found in the wrapper structure.
dtbImage.%	Similar to zImage, except device tree blob is embedded inside the image instead of provided by firmware. The output image file can be either an elf file or a flat binary depending on the platform. dtbImages are used on systems which do not have an interface for passing a device tree directly. dtbImages are similar to simpleImages except that dtbImages have platform specific code for extracting data from the board firmware, but simpleImages do not talk to the firmware at all. PlayStation 3 support uses dtbImage. So do Embedded Planet boards using the PlanetCore firmware. Board specific initialization code is typically found in a file named arch/powerpc/boot/<platform>.c; but this can be overridden by the wrapper script.
simpleImage.%	Firmware independent compressed image that does not depend on any particular firmware interface and embeds a device tree blob. This image is a flat binary that can be loaded to any location in RAM and jumped to. Firmware cannot pass any configuration data to the kernel with this image type and it depends entirely on the embedded device tree for all information.
treeImage.%	Image format for used with OpenBIOS firmware found on some ppc4xx hardware. This image embeds a device tree blob inside the image.
uImage:	Native image format used by U-Boot. The uImage target does not add any boot code. It just wraps a compressed vmlinux in the uImage data structure. This image requires a version of U-Boot that is able to pass a device tree to the kernel at boot. If using an older version of U-Boot, then you need to use a cuImage instead.
zImage.%	Image format which does not embed a device tree. Used by OpenFirmware and other firmware interfaces which are able to supply a device tree. This image expects firmware to provide the device tree at boot. Typically, if you have general purpose PowerPC hardware then you want this image format.

Image types which embed a device tree blob (simpleImage, dtbImage, treeImage, and cuImage) all generate the device tree blob from a file in the arch/powerpc/boot/dts/ directory. The Makefile selects the correct device tree source based on the name of the target. Therefore, if the kernel is built with 'make treeImage.walnut', then the build system will use arch/powerpc/boot/dts/walnut.dts to build treeImage.walnut.

Two special targets called 'zImage' and 'zImage.initrd' also exist. These targets build all the default images as selected by the kernel configuration. Default images are selected by the boot wrapper Makefile (arch/powerpc/boot/Makefile) by adding targets to the $image-y variable. Look at the Makefile to see which default image targets are available.

1.1 How it is built

arch/powerpc is designed to support multiplatform kernels, which means that a single vmlinux image can be booted on many different target boards. It also means that the boot wrapper must be able to wrap for many kinds of images on a single build. The design decision was made to not use any conditional compilation code (#ifdef, etc) in the boot wrapper source code. All of the boot wrapper pieces are buildable at any time regardless of the kernel configuration. Building all the wrapper bits on every kernel build also ensures that obscure parts of the wrapper are at the very least compile tested in a large variety of environments.

The wrapper is adapted for different image types at link time by linking in just the wrapper bits that are appropriate for the image type. The 'wrapper script' (found in arch/powerpc/boot/wrapper) is called by the Makefile and is responsible for selecting the correct wrapper bits for the image type. The arguments are well documented in the script's comment block, so they are not repeated here. However, it is worth mentioning that the script uses the -p (platform) argument as the main method of deciding which wrapper bits to compile in. Look for the large 'case "$platform" in' block in the middle of the script. This is also the place where platform specific fixups can be selected by changing the link order.

In particular, care should be taken when working with cuImages. cuImage wrapper bits are very board specific and care should be taken to make sure the target you are trying to build is supported by the wrapper bits.

CHAPTER

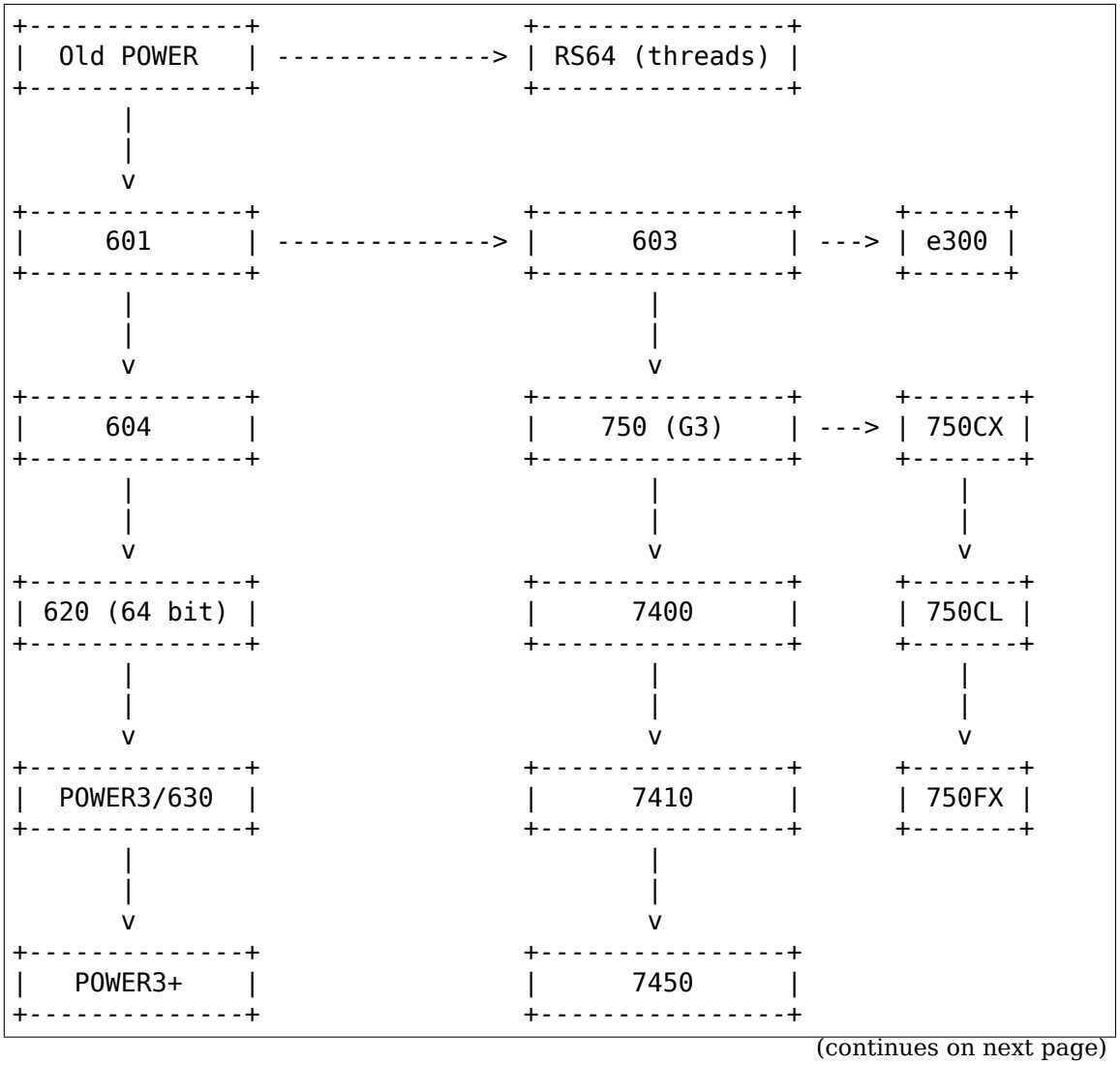
TWO

CPU FAMILIES

This document tries to summarise some of the different cpu families that exist and are supported by arch/powerpc.

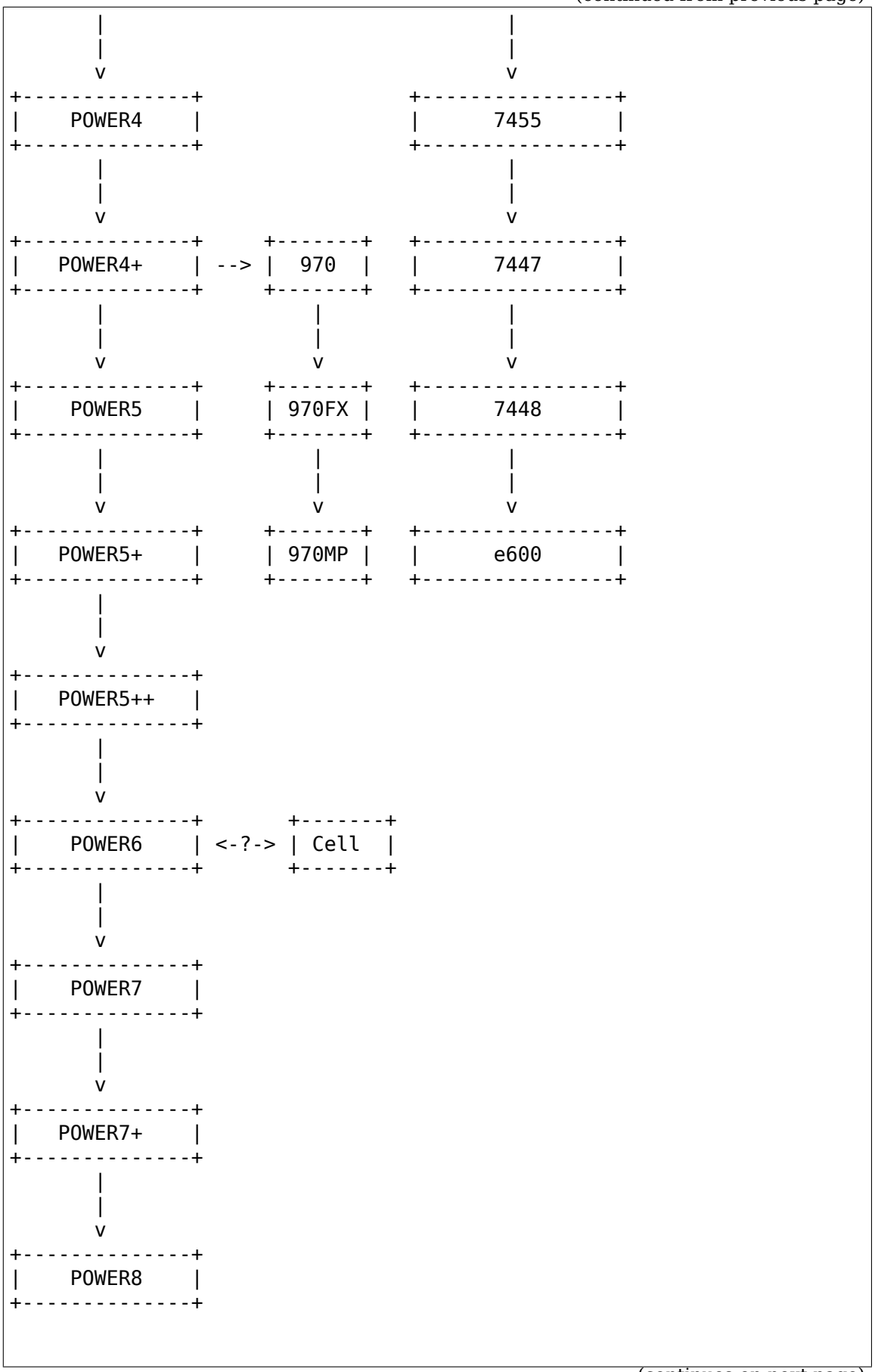
2.1 Book3S (aka sPAPR)

- Hash MMU
- Mix of 32 & 64 bit:



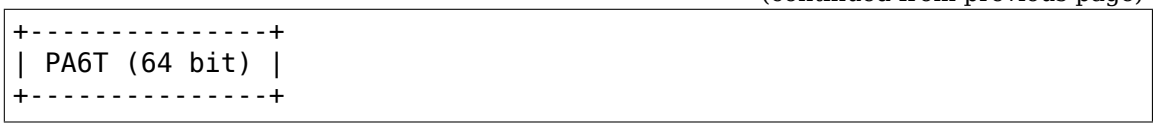
(continues on next page)

(continued from previous page)



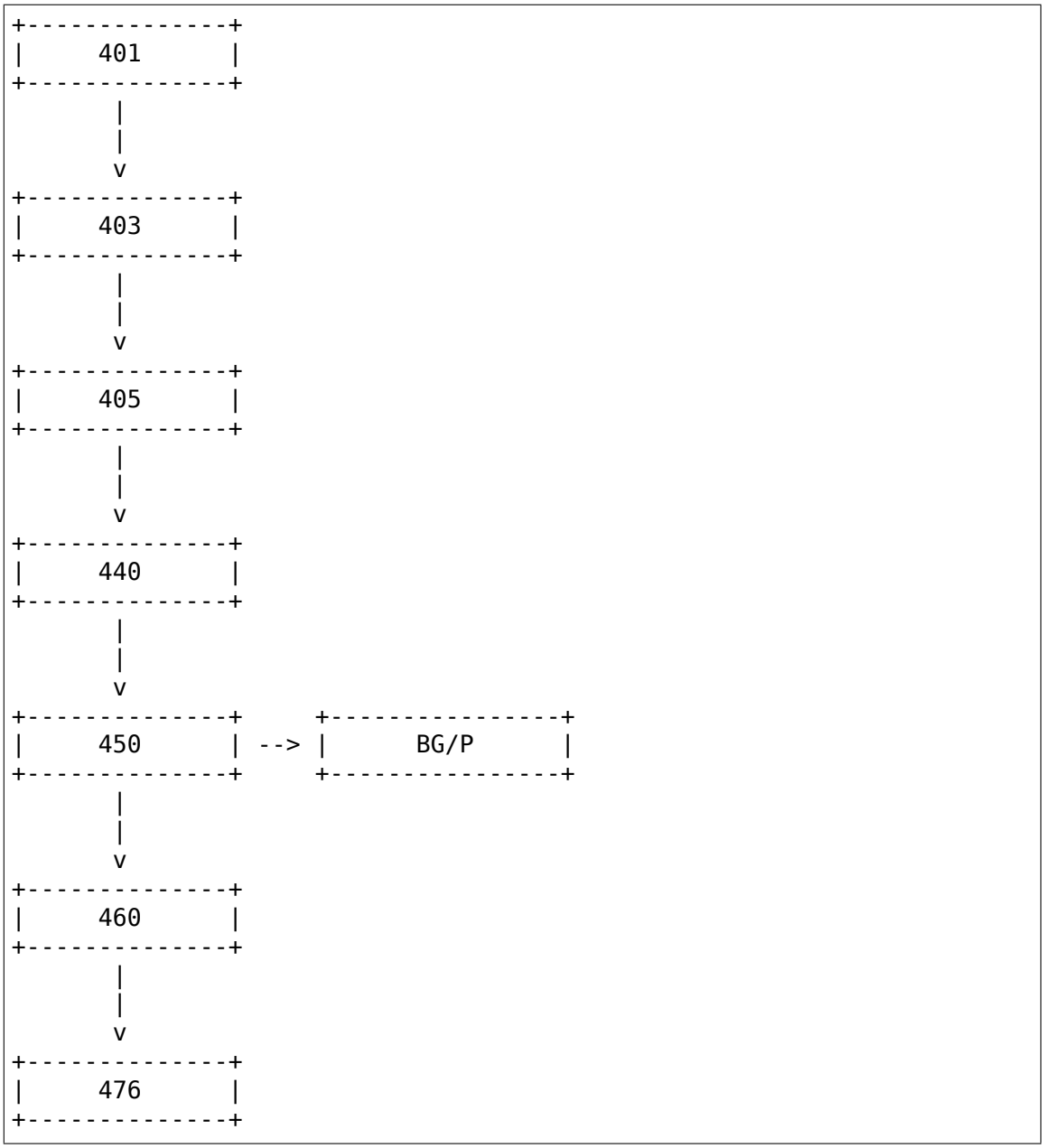
(continues on next page)

(continued from previous page)



2.2 IBM BookE

- Software loaded TLB.
- All 32 bit:

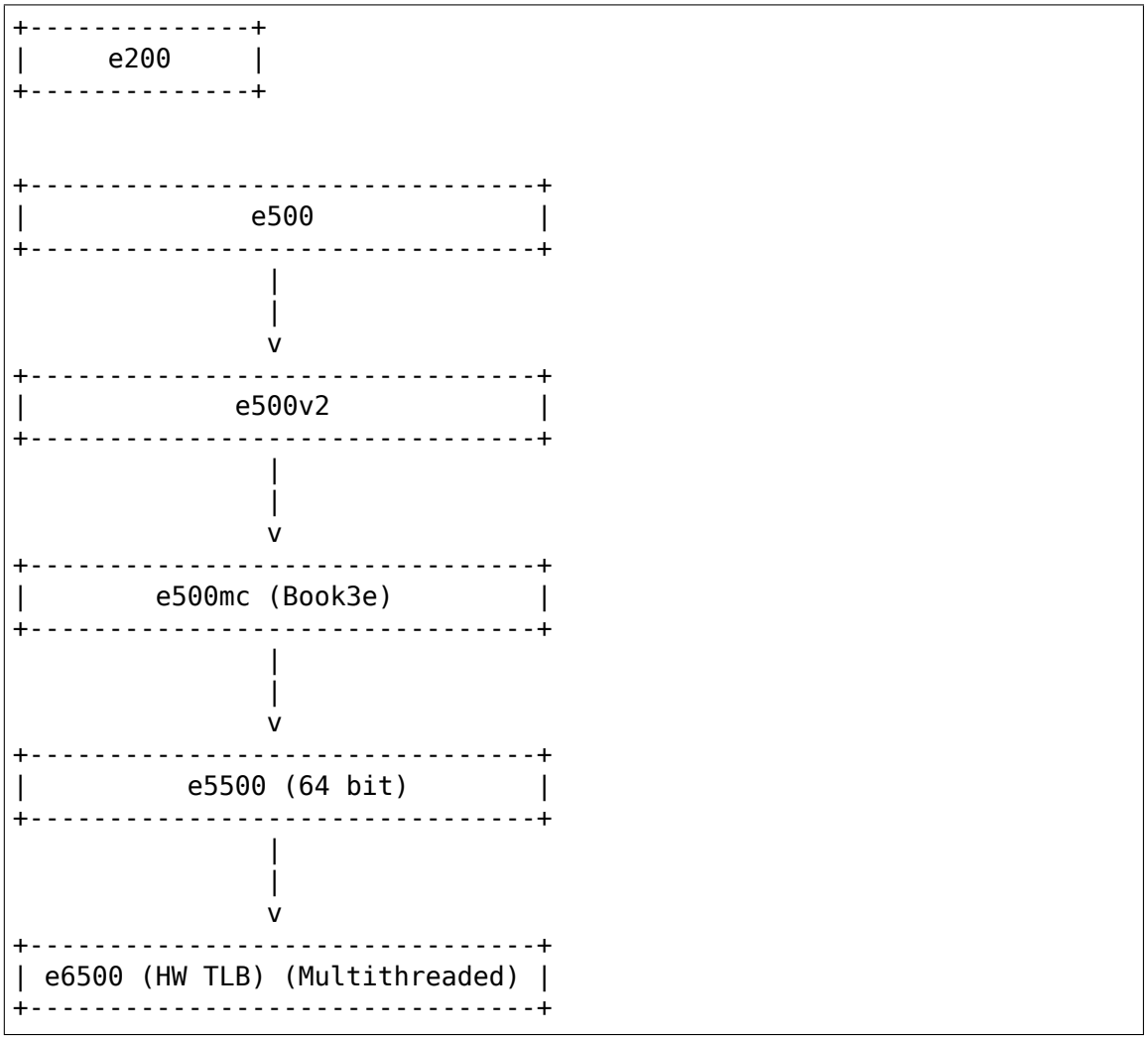


2.3 Motorola/Freescale 8xx

- Software loaded with hardware assist.
- All 32 bit:

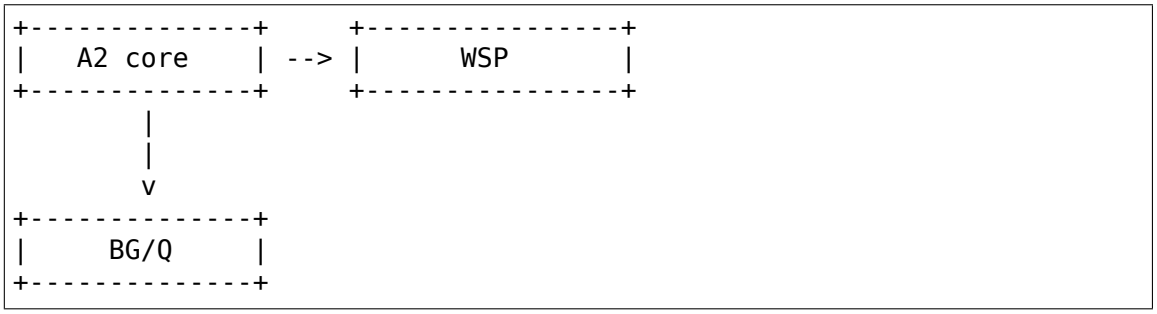
2.4 Freescale BookE

- Software loaded TLB.
- e6500 adds HW loaded indirect TLB entries.
- Mix of 32 & 64 bit:



2.5 IBM A2 core

- Book3E, software loaded TLB + HW loaded indirect TLB entries.
- 64 bit:



CHAPTER

THREE

CPU FEATURES

Hollis Blanchard <hollis@austin.ibm.com> 5 Jun 2002

This document describes the system (including self-modifying code) used in the PPC Linux kernel to support a variety of PowerPC CPUs without requiring compile-time selection.

Early in the boot process the ppc32 kernel detects the current CPU type and chooses a set of features accordingly. Some examples include Altivec support, split instruction and data caches, and if the CPU supports the DOZE and NAP sleep modes.

Detection of the feature set is simple. A list of processors can be found in arch/powerpc/kernel/cputable.c. The PVR register is masked and compared with each value in the list. If a match is found, the cpu_features of cur_cpu_spec is assigned to the feature bitmask for this processor and a __setup_cpu function is called.

C code may test 'cur_cpu_spec[smp_processor_id()]->cpu_features'for a particular feature bit. This is done in quite a few places, for example in ppc_setup_l2cr().

Implementing cpufeatures in assembly is a little more involved. There are several paths that are performance-critical and would suffer if an array index, structure dereference, and conditional branch were added. To avoid the performance penalty but still allow for runtime (rather than compile-time) CPU selection, unused code is replaced by 'nop' instructions. This nop'ing is based on CPU 0's capabilities, so a multi-processor system with non-identical processors will not work (but such a system would likely have other problems anyways).

After detecting the processor type, the kernel patches out sections of code that shouldn't be used by writing nop's over it. Using cpufeatures requires just 2 macros (found in arch/powerpc/include/asm/cputable.h), as seen in head.S transfer_to_handler:

```
#ifdef CONFIG_ALTIVEC
BEGIN_FTR_SECTION
        mfspr   r22,SPRN_VRSAVE         /* if G4, save vrsave register␣
↪value */
        stw     r22,THREAD_VRSAVE(r23)
END_FTR_SECTION_IFSET(CPU_FTR_ALTIVEC)
#endif /* CONFIG_ALTIVEC */
```

If CPU 0 supports Altivec, the code is left untouched. If it doesn't, both instructions are replaced with nop's.

The END_FTR_SECTION macro has two simpler variations: END_FTR_SECTION_IFSET and END_FTR_SECTION_IFCLR. These simply test if a flag is set (in cur_cpu_spec[0]->cpu_features) or is cleared, respectively. These two macros should be used in the majority of cases.

The END_FTR_SECTION macros are implemented by storing information about this code in the '__ftr_fixup' ELF section. When do_cpu_ftr_fixups (arch/powerpc/kernel/misc.S) is invoked, it will iterate over the records in __ftr_fixup, and if the required feature is not present it will loop writing nop's from each BEGIN_FTR_SECTION to END_FTR_SECTION.

CHAPTER

FOUR

COHERENT ACCELERATOR INTERFACE (CXL)

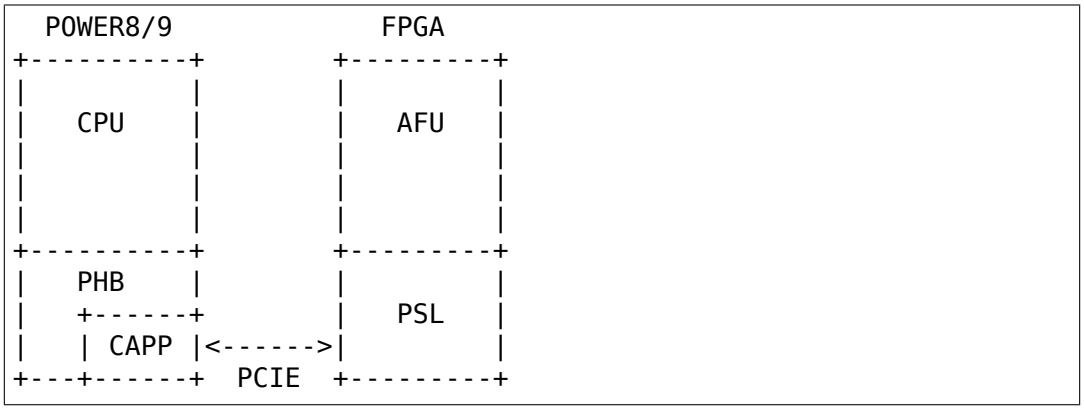
4.1 Introduction

The coherent accelerator interface is designed to allow the coherent connection of accelerators (FPGAs and other devices) to a POWER system. These devices need to adhere to the Coherent Accelerator Interface Architecture (CAIA).

IBM refers to this as the Coherent Accelerator Processor Interface or CAPI. In the kernel it's referred to by the name CXL to avoid confusion with the ISDN CAPI subsystem.

Coherent in this context means that the accelerator and CPUs can both access system memory directly and with the same effective addresses.

4.2 Hardware overview



The POWER8/9 chip has a Coherently Attached Processor Proxy (CAPP) unit which is part of the PCIe Host Bridge (PHB). This is managed by Linux by calls into OPAL. Linux doesn't directly program the CAPP.

The FPGA (or coherently attached device) consists of two parts. The POWER Service Layer (PSL) and the Accelerator Function Unit (AFU). The AFU is used to implement specific functionality behind the PSL. The PSL, among other things, provides memory address translation services to allow each AFU direct access to userspace memory.

The AFU is the core part of the accelerator (eg. the compression, crypto etc function). The kernel has no knowledge of the function of the AFU. Only userspace interacts directly with the AFU.

The PSL provides the translation and interrupt services that the AFU needs. This is what the kernel interacts with. For example, if the AFU needs to read a particular effective address, it sends that address to the PSL, the PSL then translates it, fetches the data from memory and returns it to the AFU. If the PSL has a translation miss, it interrupts the kernel and the kernel services the fault. The context to which this fault is serviced is based on who owns that acceleration function.

- POWER8 and PSL Version 8 are compliant to the CAIA Version 1.0.
- POWER9 and PSL Version 9 are compliant to the CAIA Version 2.0.

This PSL Version 9 provides new features such as:

- Interaction with the nest MMU on the P9 chip.
- Native DMA support.
- Supports sending ASB_Notify messages for host thread wakeup.
- Supports Atomic operations.
- etc.

Cards with a PSL9 won't work on a POWER8 system and cards with a PSL8 won't work on a POWER9 system.

4.3 AFU Modes

There are two programming modes supported by the AFU. Dedicated and AFU directed. AFU may support one or both modes.

When using dedicated mode only one MMU context is supported. In this mode, only one userspace process can use the accelerator at time.

When using AFU directed mode, up to 16K simultaneous contexts can be supported. This means up to 16K simultaneous userspace applications may use the accelerator (although specific AFUs may support fewer). In this mode, the AFU sends a 16 bit context ID with each of its requests. This tells the PSL which context is associated with each operation. If the PSL can't translate an operation, the ID can also be accessed by the kernel so it can determine the userspace context associated with an operation.

4.4 MMIO space

A portion of the accelerator MMIO space can be directly mapped from the AFU to userspace. Either the whole space can be mapped or just a per context portion. The hardware is self describing, hence the kernel can determine the offset and size of the per context portion.

4.5 Interrupts

AFUs may generate interrupts that are destined for userspace. These are received by the kernel as hardware interrupts and passed onto userspace by a read syscall documented below.

Data storage faults and error interrupts are handled by the kernel driver.

4.6 Work Element Descriptor (WED)

The WED is a 64-bit parameter passed to the AFU when a context is started. Its format is up to the AFU hence the kernel has no knowledge of what it represents. Typically it will be the effective address of a work queue or status block where the AFU and userspace can share control and status information.

4.7 User API

4.7.1 1. AFU character devices

For AFUs operating in AFU directed mode, two character device files will be created. /dev/cxl/afu0.0m will correspond to a master context and /dev/cxl/afu0.0s will correspond to a slave context. Master contexts have access to the full MMIO space an AFU provides. Slave contexts have access to only the per process MMIO space an AFU provides.

For AFUs operating in dedicated process mode, the driver will only create a single character device per AFU called /dev/cxl/afu0.0d. This will have access to the entire MMIO space that the AFU provides (like master contexts in AFU directed).

The types described below are defined in include/uapi/misc/cxl.h

The following file operations are supported on both slave and master devices.

A userspace library libcxl is available here:

https://github.com/ibm-capi/libcxl

This provides a C interface to this kernel API.

open

Opens the device and allocates a file descriptor to be used with the rest of the API.

A dedicated mode AFU only has one context and only allows the device to be opened once.

An AFU directed mode AFU can have many contexts, the device can be opened once for each context that is available.

When all available contexts are allocated the open call will fail and return -ENOSPC.

Note: IRQs need to be allocated for each context, which may limit the number of contexts that can be created, and therefore how many times the device can be opened. The POWER8 CAPP supports 2040 IRQs and 3 are used by the kernel, so 2037 are left. If 1 IRQ is needed per context, then only 2037 contexts can be allocated. If 4 IRQs are needed per context, then only 2037/4 = 509 contexts can be allocated.

ioctl

CXL_IOCTL_START_WORK: Starts the AFU context and associates it with the current process. Once this ioctl is successfully executed, all memory mapped into this process is accessible to this AFU context using the same effective addresses. No additional calls are required to map/unmap memory. The AFU memory context will be updated as userspace allocates and frees memory. This ioctl returns once the AFU context is started.

Takes a pointer to a struct cxl_ioctl_start_work

```
struct cxl_ioctl_start_work {
        __u64 flags;
        __u64 work_element_descriptor;
        __u64 amr;
        __s16 num_interrupts;
        __s16 reserved1;
        __s32 reserved2;
        __u64 reserved3;
        __u64 reserved4;
        __u64 reserved5;
        __u64 reserved6;
};
```

flags: Indicates which optional fields in the structure are valid.

work_element_descriptor: The Work Element Descriptor (WED) is a 64-bit argument defined by the AFU. Typically this is an effective address pointing to an AFU specific structure describing what work to perform.

amr: Authority Mask Register (AMR), same as the powerpc AMR. This field is only used by the kernel when the corresponding CXL_START_WORK_AMR value is specified in flags. If not specified the kernel will use a default value of 0.

num_interrupts: Number of userspace interrupts to request. This field is only used by the kernel when the corresponding CXL_START_WORK_NUM_IRQS value is specified in flags. If not specified the minimum number required by the AFU will be allocated. The min and max number can be obtained from sysfs.

reserved fields: For ABI padding and future extensions

CXL_IOCTL_GET_PROCESS_ELEMENT: Get the current context id, also known as the process element. The value is returned from the kernel as a __u32.

mmap

An AFU may have an MMIO space to facilitate communication with the AFU. If it does, the MMIO space can be accessed via mmap. The size and contents of this area are specific to the particular AFU. The size can be discovered via sysfs.

In AFU directed mode, master contexts are allowed to map all of the MMIO space and slave contexts are allowed to only map the per process MMIO space associated with the context. In dedicated process mode the entire MMIO space can always be mapped.

This mmap call must be done after the START_WORK ioctl.

Care should be taken when accessing MMIO space. Only 32 and 64-bit accesses are supported by POWER8. Also, the AFU will be designed with a specific endianness, so all MMIO accesses should consider endianness (recommend endian(3) variants like: le64toh(), be64toh() etc). These endian issues equally apply to shared memory queues the WED may describe.

read

Reads events from the AFU. Blocks if no events are pending (unless O_NONBLOCK is supplied). Returns -EIO in the case of an unrecoverable error or if the card is removed.

read() will always return an integral number of events.

The buffer passed to read() must be at least 4K bytes.

The result of the read will be a buffer of one or more events, each event is of type struct cxl_event, of varying size:

```
struct cxl_event {
        struct cxl_event_header header;
        union {
                struct cxl_event_afu_interrupt irq;
                struct cxl_event_data_storage fault;
                struct cxl_event_afu_error afu_error;
        };
};
```

The struct cxl_event_header is defined as

```
struct cxl_event_header {
        __u16 type;
        __u16 size;
        __u16 process_element;
        __u16 reserved1;
};
```

type: This defines the type of event. The type determines how the rest of the event is structured. These types are described below and defined by enum cxl_event_type.

size: This is the size of the event in bytes including the struct cxl_event_header. The start of the next event can be found at this offset from the start of the current event.

process_element: Context ID of the event.

reserved field: For future extensions and padding.

If the event type is CXL_EVENT_AFU_INTERRUPT then the event structure is defined as

```
struct cxl_event_afu_interrupt {
        __u16 flags;
        __u16 irq; /* Raised AFU interrupt number */
        __u32 reserved1;
};
```

flags: These flags indicate which optional fields are present in this struct. Currently all fields are mandatory.

irq: The IRQ number sent by the AFU.

reserved field: For future extensions and padding.

If the event type is CXL_EVENT_DATA_STORAGE then the event structure is defined as

```
struct cxl_event_data_storage {
        __u16 flags;
        __u16 reserved1;
        __u32 reserved2;
        __u64 addr;
        __u64 dsisr;
        __u64 reserved3;
};
```

flags: These flags indicate which optional fields are present in this struct. Currently all fields are mandatory.

address: The address that the AFU unsuccessfully attempted to access. Valid accesses will be handled transparently by the kernel but invalid accesses will generate this event.

dsisr: This field gives information on the type of fault. It is a copy of the DSISR from the PSL hardware when the address fault occurred. The form of the DSISR is as defined in the CAIA.

reserved fields: For future extensions

If the event type is CXL_EVENT_AFU_ERROR then the event structure is defined as

```
struct cxl_event_afu_error {
        __u16 flags;
        __u16 reserved1;
        __u32 reserved2;
        __u64 error;
};
```

flags: These flags indicate which optional fields are present in this struct. Currently all fields are Mandatory.

error: Error status from the AFU. Defined by the AFU.

reserved fields: For future extensions and padding

4.7.2 2. Card character device (powerVM guest only)

In a powerVM guest, an extra character device is created for the card. The device is only used to write (flash) a new image on the FPGA accelerator. Once the image is written and verified, the device tree is updated and the card is reset to reload the updated image.

open

Opens the device and allocates a file descriptor to be used with the rest of the API. The device can only be opened once.

ioctl

CXL_IOCTL_DOWNLOAD_IMAGE / CXL_IOCTL_VALIDATE_IMAGE: Starts and controls flashing a new FPGA image. Partial reconfiguration is not supported (yet), so the image must contain a copy of the PSL and AFU(s). Since an image can be quite large, the caller may have to iterate, splitting the image in smaller chunks.

Takes a pointer to a struct cxl_adapter_image:

```
struct cxl_adapter_image {
    __u64 flags;
    __u64 data;
    __u64 len_data;
    __u64 len_image;
    __u64 reserved1;
    __u64 reserved2;
    __u64 reserved3;
    __u64 reserved4;
};
```

flags: These flags indicate which optional fields are present in this struct. Currently all fields are mandatory.

data: Pointer to a buffer with part of the image to write to the card.

len_data: Size of the buffer pointed to by data.

len_image: Full size of the image.

4.8 Sysfs Class

A cxl sysfs class is added under /sys/class/cxl to facilitate enumeration and tuning of the accelerators. Its layout is described in Documentation/ABI/testing/sysfs-class-cxl

4.9 Udev rules

The following udev rules could be used to create a symlink to the most logical chardev to use in any programming mode (afuX.Yd for dedicated, afuX.Ys for afu directed), since the API is virtually identical for each:

```
SUBSYSTEM=="cxl", ATTRS{mode}=="dedicated_process", SYMLINK="cxl/
↪%b"
SUBSYSTEM=="cxl", ATTRS{mode}=="afu_directed", \
                  KERNEL=="afu[0-9]*.[0-9]*s", SYMLINK="cxl/%b"
```

CHAPTER

FIVE

COHERENT ACCELERATOR (CXL) FLASH

5.1 Introduction

The IBM Power architecture provides support for CAPI (Coherent Accelerator Power Interface), which is available to certain PCIe slots on Power 8 systems. CAPI can be thought of as a special tunneling protocol through PCIe that allow PCIe adapters to look like special purpose co-processors which can read or write an application's memory and generate page faults. As a result, the host interface to an adapter running in CAPI mode does not require the data buffers to be mapped to the device's memory (IOMMU bypass) nor does it require memory to be pinned.

On Linux, Coherent Accelerator (CXL) kernel services present CAPI devices as a PCI device by implementing a virtual PCI host bridge. This abstraction simplifies the infrastructure and programming model, allowing for drivers to look similar to other native PCI device drivers.

CXL provides a mechanism by which user space applications can directly talk to a device (network or storage) bypassing the typical kernel/device driver stack. The CXL Flash Adapter Driver enables a user space application direct access to Flash storage.

The CXL Flash Adapter Driver is a kernel module that sits in the SCSI stack as a low level device driver (below the SCSI disk and protocol drivers) for the IBM CXL Flash Adapter. This driver is responsible for the initialization of the adapter, setting up the special path for user space access, and performing error recovery. It communicates directly the Flash Accelerator Functional Unit (AFU) as described in Documentation/powerpc/cxl.rst.

The cxlflash driver supports two, mutually exclusive, modes of operation at the device (LUN) level:

- Any flash device (LUN) can be configured to be accessed as a regular disk device (i.e.: /dev/sdc). This is the default mode.
- Any flash device (LUN) can be configured to be accessed from user space with a special block library. This mode further specifies the means of accessing the device and provides for either raw access to the entire LUN (referred to as direct or physical LUN access) or access to a kernel/AFU-mediated partition of the LUN (referred to as virtual LUN access). The segmentation of a disk device into

virtual LUNs is assisted by special translation services provided by the Flash AFU.

5.2 Overview

The Coherent Accelerator Interface Architecture (CAIA) introduces a concept of a master context. A master typically has special privileges granted to it by the kernel or hypervisor allowing it to perform AFU wide management and control. The master may or may not be involved directly in each user I/O, but at the minimum is involved in the initial setup before the user application is allowed to send requests directly to the AFU.

The CXL Flash Adapter Driver establishes a master context with the AFU. It uses memory mapped I/O (MMIO) for this control and setup. The Adapter Problem Space Memory Map looks like this:

```
+------------------------------+
|    512 * 64 KB User MMIO     |
|        (per context)         |
|       User Accessible        |
+------------------------------+
|    512 * 128 B per context   |
|    Provisioning and Control  |
|   Trusted Process accessible |
+------------------------------+
|         64 KB Global         |
|   Trusted Process accessible |
+------------------------------+
```

This driver configures itself into the SCSI software stack as an adapter driver. The driver is the only entity that is considered a Trusted Process to program the Provisioning and Control and Global areas in the MMIO Space shown above. The master context driver discovers all LUNs attached to the CXL Flash adapter and instantiates scsi block devices (/dev/sdb, /dev/sdc etc.) for each unique LUN seen from each path.

Once these scsi block devices are instantiated, an application written to a specification provided by the block library may get access to the Flash from user space (without requiring a system call).

This master context driver also provides a series of ioctls for this block library to enable this user space access. The driver supports two modes for accessing the block device.

The first mode is called a virtual mode. In this mode a single scsi block device (/dev/sdb) may be carved up into any number of distinct virtual LUNs. The virtual LUNs may be resized as long as the sum of the sizes of all the virtual LUNs, along with the meta-data associated with it does not exceed the physical capacity.

The second mode is called the physical mode. In this mode a single block device (/dev/sdb) may be opened directly by the block library and the entire space for the LUN is available to the application.

Only the physical mode provides persistence of the data. i.e. The data written to the block device will survive application exit and restart and also reboot. The virtual LUNs do not persist (i.e. do not survive after the application terminates or the system reboots).

5.3 Block library API

Applications intending to get access to the CXL Flash from user space should use the block library, as it abstracts the details of interfacing directly with the cxlflash driver that are necessary for performing administrative actions (i.e.: setup, tear down, resize). The block library can be thought of as a 'user' of services, implemented as IOCTLs, that are provided by the cxlflash driver specifically for devices (LUNs) operating in user space access mode. While it is not a requirement that applications understand the interface between the block library and the cxlflash driver, a high-level overview of each supported service (IOCTL) is provided below.

The block library can be found on GitHub: http://github.com/open-power/capiflash

5.4 CXL Flash Driver LUN IOCTLs

Users, such as the block library, that wish to interface with a flash device (LUN) via user space access need to use the services provided by the cxlflash driver. As these services are implemented as ioctls, a file descriptor handle must first be obtained in order to establish the communication channel between a user and the kernel. This file descriptor is obtained by opening the device special file associated with the scsi disk device (/dev/sdb) that was created during LUN discovery. As per the location of the cxlflash driver within the SCSI protocol stack, this open is actually not seen by the cxlflash driver. Upon successful open, the user receives a file descriptor (herein referred to as fd1) that should be used for issuing the subsequent ioctls listed below.

The structure definitions for these IOCTLs are available in: uapi/scsi/cxlflash_ioctl.h

5.4.1 DK_CXLFLASH_ATTACH

This ioctl obtains, initializes, and starts a context using the CXL kernel services. These services specify a context id (u16) by which to uniquely identify the context and its allocated resources. The services additionally provide a second file descriptor (herein referred to as fd2) that is used by the block library to initiate memory mapped I/O (via mmap()) to the CXL flash device and poll for completion events. This file descriptor is intentionally installed by this driver and not the CXL kernel services to

allow for intermediary notification and access in the event of a non-user-initiated close(), such as a killed process. This design point is described in further detail in the description for the DK_CXLFLASH_DETACH ioctl.

There are a few important aspects regarding the "tokens" (context id and fd2) that are provided back to the user:

- These tokens are only valid for the process under which they were created. The child of a forked process cannot continue to use the context id or file descriptor created by its parent (see DK_CXLFLASH_VLUN_CLONE for further details).
- These tokens are only valid for the lifetime of the context and the process under which they were created. Once either is destroyed, the tokens are to be considered stale and subsequent usage will result in errors.
- A valid adapter file descriptor (fd2 >= 0) is only returned on the initial attach for a context. Subsequent attaches to an existing context (DK_CXLFLASH_ATTACH_REUSE_CONTEXT flag present) do not provide the adapter file descriptor as it was previously made known to the application.
- When a context is no longer needed, the user shall detach from the context via the DK_CXLFLASH_DETACH ioctl. When this ioctl returns with a valid adapter file descriptor and the return flag DK_CXLFLASH_APP_CLOSE_ADAP_FD is present, the application _must_ close the adapter file descriptor following a successful detach.
- When this ioctl returns with a valid fd2 and the return flag DK_CXLFLASH_APP_CLOSE_ADAP_FD is present, the application _must_ close fd2 in the following circumstances:
  - Following a successful detach of the last user of the context
  - Following a successful recovery on the context's original fd2
  - In the child process of a fork(), following a clone ioctl, on the fd2 associated with the source context
- At any time, a close on fd2 will invalidate the tokens. Applications should exercise caution to only close fd2 when appropriate (outlined in the previous bullet) to avoid premature loss of I/O.

5.4.2 DK_CXLFLASH_USER_DIRECT

This ioctl is responsible for transitioning the LUN to direct (physical) mode access and configuring the AFU for direct access from user space on a per-context basis. Additionally, the block size and last logical block address (LBA) are returned to the user.

As mentioned previously, when operating in user space access mode, LUNs may be accessed in whole or in part. Only one mode is allowed at a time and if one mode is active (outstanding references exist), requests to use the LUN in a different mode are denied.

The AFU is configured for direct access from user space by adding an entry to the AFU's resource handle table. The index of the entry is treated as a resource handle that is returned to the user. The user is then able to use the handle to reference the LUN during I/O.

5.4.3 DK_CXLFLASH_USER_VIRTUAL

This ioctl is responsible for transitioning the LUN to virtual mode of access and configuring the AFU for virtual access from user space on a per-context basis. Additionally, the block size and last logical block address (LBA) are returned to the user.

As mentioned previously, when operating in user space access mode, LUNs may be accessed in whole or in part. Only one mode is allowed at a time and if one mode is active (outstanding references exist), requests to use the LUN in a different mode are denied.

The AFU is configured for virtual access from user space by adding an entry to the AFU's resource handle table. The index of the entry is treated as a resource handle that is returned to the user. The user is then able to use the handle to reference the LUN during I/O.

By default, the virtual LUN is created with a size of 0. The user would need to use the DK_CXLFLASH_VLUN_RESIZE ioctl to adjust the grow the virtual LUN to a desired size. To avoid having to perform this resize for the initial creation of the virtual LUN, the user has the option of specifying a size as part of the DK_CXLFLASH_USER_VIRTUAL ioctl, such that when success is returned to the user, the resource handle that is provided is already referencing provisioned storage. This is reflected by the last LBA being a non-zero value.

When a LUN is accessible from more than one port, this ioctl will return with the DK_CXLFLASH_ALL_PORTS_ACTIVE return flag set. This provides the user with a hint that I/O can be retried in the event of an I/O error as the LUN can be reached over multiple paths.

5.4.4 DK_CXLFLASH_VLUN_RESIZE

This ioctl is responsible for resizing a previously created virtual LUN and will fail if invoked upon a LUN that is not in virtual mode. Upon success, an updated last LBA is returned to the user indicating the new size of the virtual LUN associated with the resource handle.

The partitioning of virtual LUNs is jointly mediated by the cxlflash driver and the AFU. An allocation table is kept for each LUN that is operating in the virtual mode and used to program a LUN translation table that the AFU references when provided with a resource handle.

This ioctl can return -EAGAIN if an AFU sync operation takes too long. In addition to returning a failure to user, cxlflash will also schedule an asynchronous AFU reset. Should the user choose to retry the operation, it is expected to succeed. If this ioctl fails with -EAGAIN, the user can either retry the operation or treat it as a failure.

5.4.5 DK_CXLFLASH_RELEASE

This ioctl is responsible for releasing a previously obtained reference to either a physical or virtual LUN. This can be thought of as the inverse of the DK_CXLFLASH_USER_DIRECT or DK_CXLFLASH_USER_VIRTUAL ioctls. Upon success, the resource handle is no longer valid and the entry in the resource handle table is made available to be used again.

As part of the release process for virtual LUNs, the virtual LUN is first resized to 0 to clear out and free the translation tables associated with the virtual LUN reference.

5.4.6 DK_CXLFLASH_DETACH

This ioctl is responsible for unregistering a context with the cxlflash driver and release outstanding resources that were not explicitly released via the DK_CXLFLASH_RELEASE ioctl. Upon success, all "tokens" which had been provided to the user from the DK_CXLFLASH_ATTACH onward are no longer valid.

When the DK_CXLFLASH_APP_CLOSE_ADAP_FD flag was returned on a successful attach, the application _must_ close the fd2 associated with the context following the detach of the final user of the context.

5.4.7 DK_CXLFLASH_VLUN_CLONE

This ioctl is responsible for cloning a previously created context to a more recently created context. It exists solely to support maintaining user space access to storage after a process forks. Upon success, the child process (which invoked the ioctl) will have access to the same LUNs via the same resource handle(s) as the parent, but under a different context.

Context sharing across processes is not supported with CXL and therefore each fork must be met with establishing a new context for the child process. This ioctl simplifies the state management and playback required by a user in such a scenario. When a process forks, child process can clone the parents context by first creating a context (via DK_CXLFLASH_ATTACH) and then using this ioctl to perform the clone from the parent to the child.

The clone itself is fairly simple. The resource handle and lun translation tables are copied from the parent context to the child' s and then synced with the AFU.

When the DK_CXLFLASH_APP_CLOSE_ADAP_FD flag was returned on a successful attach, the application _must_ close the fd2 associated with the source context (still resident/accessible in the parent process) following the clone. This is to avoid a stale entry in the file descriptor table of the child process.

This ioctl can return -EAGAIN if an AFU sync operation takes too long. In addition to returning a failure to user, cxlflash will also schedule an

asynchronous AFU reset. Should the user choose to retry the operation, it is expected to succeed. If this ioctl fails with -EAGAIN, the user can either retry the operation or treat it as a failure.

5.4.8 DK_CXLFLASH_VERIFY

This ioctl is used to detect various changes such as the capacity of the disk changing, the number of LUNs visible changing, etc. In cases where the changes affect the application (such as a LUN resize), the cxlflash driver will report the changed state to the application.

The user calls in when they want to validate that a LUN hasn't been changed in response to a check condition. As the user is operating out of band from the kernel, they will see these types of events without the kernel's knowledge. When encountered, the user's architected behavior is to call in to this ioctl, indicating what they want to verify and passing along any appropriate information. For now, only verifying a LUN change (ie: size different) with sense data is supported.

5.4.9 DK_CXLFLASH_RECOVER_AFU

This ioctl is used to drive recovery (if such an action is warranted) of a specified user context. Any state associated with the user context is re-established upon successful recovery.

User contexts are put into an error condition when the device needs to be reset or is terminating. Users are notified of this error condition by seeing all 0xF's on an MMIO read. Upon encountering this, the architected behavior for a user is to call into this ioctl to recover their context. A user may also call into this ioctl at any time to check if the device is operating normally. If a failure is returned from this ioctl, the user is expected to gracefully clean up their context via release/detach ioctls. Until they do, the context they hold is not relinquished. The user may also optionally exit the process at which time the context/resources they held will be freed as part of the release fop.

When the DK_CXLFLASH_APP_CLOSE_ADAP_FD flag was returned on a successful attach, the application _must_ unmap and close the fd2 associated with the original context following this ioctl returning success and indicating that the context was recovered (DK_CXLFLASH_RECOVER_AFU_CONTEXT_RESET).

5.4.10 DK_CXLFLASH_MANAGE_LUN

This ioctl is used to switch a LUN from a mode where it is available for file-system access (legacy), to a mode where it is set aside for exclusive user space access (superpipe). In case a LUN is visible across multiple ports and adapters, this ioctl is used to uniquely identify each LUN by its World Wide Node Name (WWNN).

5.5 CXL Flash Driver Host IOCTLs

Each host adapter instance that is supported by the cxlflash driver has a special character device associated with it to enable a set of host management function. These character devices are hosted in a class dedicated for cxlflash and can be accessed via /dev/cxlflash/*.

Applications can be written to perform various functions using the host ioctl APIs below.

The structure definitions for these IOCTLs are available in: uapi/scsi/cxlflash_ioctl.h

5.5.1 HT_CXLFLASH_LUN_PROVISION

This ioctl is used to create and delete persistent LUNs on cxlflash devices that lack an external LUN management interface. It is only valid when used with AFUs that support the LUN provision capability.

When sufficient space is available, LUNs can be created by specifying the target port to host the LUN and a desired size in 4K blocks. Upon success, the LUN ID and WWID of the created LUN will be returned and the SCSI bus can be scanned to detect the change in LUN topology. Note that partial allocations are not supported. Should a creation fail due to a space issue, the target port can be queried for its current LUN geometry.

To remove a LUN, the device must first be disassociated from the Linux SCSI subsystem. The LUN deletion can then be initiated by specifying a target port and LUN ID. Upon success, the LUN geometry associated with the port will be updated to reflect new number of provisioned LUNs and available capacity.

To query the LUN geometry of a port, the target port is specified and upon success, the following information is presented:

- Maximum number of provisioned LUNs allowed for the port
- Current number of provisioned LUNs for the port
- Maximum total capacity of provisioned LUNs for the port (4K blocks)
- Current total capacity of provisioned LUNs for the port (4K blocks)

With this information, the number of available LUNs and capacity can be can be calculated.

5.5.2 HT_CXLFLASH_AFU_DEBUG

This ioctl is used to debug AFUs by supporting a command pass-through interface. It is only valid when used with AFUs that support the AFU debug capability.

With exception of buffer management, AFU debug commands are opaque to cxlflash and treated as pass-through. For debug commands that do require data transfer, the user supplies an adequately sized data buffer and must specify the data transfer direction with respect to the host. There is a maximum transfer size of 256K imposed. Note that partial read completions are not supported - when errors are experienced with a host read data transfer, the data buffer is not copied back to the user.

CHAPTER

SIX

DAWR ISSUES ON POWER9

On POWER9 the Data Address Watchpoint Register (DAWR) can cause a checkstop if it points to cache inhibited (CI) memory. Currently Linux has no way to disinguish CI memory when configuring the DAWR, so (for now) the DAWR is disabled by this commit:

```
commit 9654153158d3e0684a1bdb76dbababdb7111d5a0
Author: Michael Neuling <mikey@neuling.org>
Date:   Tue Mar 27 15:37:24 2018 +1100
powerpc: Disable DAWR in the base POWER9 CPU features
```

6.1 Technical Details:

DAWR has 6 different ways of being set. 1) ptrace 2) h_set_mode(DAWR) 3) h_set_dabr() 4) kvmppc_set_one_reg() 5) xmon

For ptrace, we now advertise zero breakpoints on POWER9 via the PPC_PTRACE_GETHWDBGINFO call. This results in GDB falling back to software emulation of the watchpoint (which is slow).

h_set_mode(DAWR) and h_set_dabr() will now return an error to the guest on a POWER9 host. Current Linux guests ignore this error, so they will silently not get the DAWR.

kvmppc_set_one_reg() will store the value in the vcpu but won't actually set it on POWER9 hardware. This is done so we don't break migration from POWER8 to POWER9, at the cost of silently losing the DAWR on the migration.

For xmon, the 'bd' command will return an error on P9.

6.2 Consequences for users

For GDB watchpoints (ie 'watch' command) on POWER9 bare metal , GDB will accept the command. Unfortunately since there is no hardware support for the watchpoint, GDB will software emulate the watchpoint making it run very slowly.

The same will also be true for any guests started on a POWER9 host. The watchpoint will fail and GDB will fall back to software emulation.

If a guest is started on a POWER8 host, GDB will accept the watchpoint and configure the hardware to use the DAWR. This will run at full speed since it can use the

hardware emulation. Unfortunately if this guest is migrated to a POWER9 host, the watchpoint will be lost on the POWER9. Loads and stores to the watchpoint locations will not be trapped in GDB. The watchpoint is remembered, so if the guest is migrated back to the POWER8 host, it will start working again.

6.3 Force enabling the DAWR

Kernels (since ~v5.2) have an option to force enable the DAWR via:

```
echo Y > /sys/kernel/debug/powerpc/dawr_enable_dangerous
```

This enables the DAWR even on POWER9.

This is a dangerous setting, USE AT YOUR OWN RISK.

Some users may not care about a bad user crashing their box (ie. single user/desktop systems) and really want the DAWR. This allows them to force enable DAWR.

This flag can also be used to disable DAWR access. Once this is cleared, all DAWR access should be cleared immediately and your machine once again safe from crashing.

Userspace may get confused by toggling this. If DAWR is force enabled/disabled between getting the number of breakpoints (via PTRACE_GETHWDBGINFO) and setting the breakpoint, userspace will get an inconsistent view of what's available. Similarly for guests.

For the DAWR to be enabled in a KVM guest, the DAWR needs to be force enabled in the host AND the guest. For this reason, this won't work on POWERVM as it doesn't allow the HCALL to work. Writes of 'Y' to the dawr_enable_dangerous file will fail if the hypervisor doesn't support writing the DAWR.

To double check the DAWR is working, run this kernel selftest:

    tools/testing/selftests/powerpc/ptrace/ptrace-hwbreak.c

Any errors/failures/skips mean something is wrong.

CHAPTER

SEVEN

DSCR (DATA STREAM CONTROL REGISTER)

DSCR register in powerpc allows user to have some control of prefetch of data stream in the processor. Please refer to the ISA documents or related manual for more detailed information regarding how to use this DSCR to attain this control of the prefetches . This document here provides an overview of kernel support for DSCR, related kernel objects, it's functionalities and exported user interface.

(A) Data Structures:

(1) thread_struct:

```
dscr            /* Thread DSCR value */
dscr_inherit    /* Thread has changed default DSCR */
```

(2) PACA:

```
dscr_default    /* per-CPU DSCR default value */
```

(3) sysfs.c:

```
dscr_default    /* System DSCR default value */
```

(B) Scheduler Changes:

Scheduler will write the per-CPU DSCR default which is stored in the CPU's PACA value into the register if the thread has dscr_inherit value cleared which means that it has not changed the default DSCR till now. If the dscr_inherit value is set which means that it has changed the default DSCR value, scheduler will write the changed value which will now be contained in thread struct's dscr into the register instead of the per-CPU default PACA based DSCR value.

NOTE: Please note here that the system wide global DSCR value never gets used directly in the scheduler process context switch at all.

(C) SYSFS Interface:

- Global DSCR default: /sys/devices/system/cpu/dscr_default
- CPU specific DSCR default: /sys/devices/system/cpu/cpuN/dscr

Changing the global DSCR default in the sysfs will change all the CPU specific DSCR defaults immediately in their PACA structures. Again if the current process has the dscr_inherit clear, it also writes the

new value into every CPU's DSCR register right away and updates the current thread's DSCR value as well.

Changing the CPU specific DSCR default value in the sysfs does exactly the same thing as above but unlike the global one above, it just changes stuff for that particular CPU instead for all the CPUs on the system.

(D) User Space Instructions:

The DSCR register can be accessed in the user space using any of these two SPR numbers available for that purpose.

(1) Problem state SPR: 0x03 (Un-privileged, POWER8 only)

(2) Privileged state SPR: 0x11 (Privileged)

Accessing DSCR through privileged SPR number (0x11) from user space works, as it is emulated following an illegal instruction exception inside the kernel. Both mfspr and mtspr instructions are emulated.

Accessing DSCR through user level SPR (0x03) from user space will first create a facility unavailable exception. Inside this exception handler all mfspr instruction based read attempts will get emulated and returned where as the first mtspr instruction based write attempts will enable the DSCR facility for the next time around (both for read and write) by setting DSCR facility in the FSCR register.

(E) Specifics about 'dscr_inherit':

The thread struct element 'dscr_inherit' represents whether the thread in question has attempted and changed the DSCR itself using any of the following methods. This element signifies whether the thread wants to use the CPU default DSCR value or its own changed DSCR value in the kernel.

(1) mtspr instruction (SPR number 0x03)

(2) mtspr instruction (SPR number 0x11)

(3) ptrace interface (Explicitly set user DSCR value)

Any child of the process created after this event in the process inherits this same behaviour as well.

CHAPTER

EIGHT

PCI BUS EEH ERROR RECOVERY

Linas Vepstas <linas@austin.ibm.com>

12 January 2005

8.1 Overview:

The IBM POWER-based pSeries and iSeries computers include PCI bus controller chips that have extended capabilities for detecting and reporting a large variety of PCI bus error conditions. These features go under the name of "EEH", for "Enhanced Error Handling". The EEH hardware features allow PCI bus errors to be cleared and a PCI card to be "rebooted", without also having to reboot the operating system.

This is in contrast to traditional PCI error handling, where the PCI chip is wired directly to the CPU, and an error would cause a CPU machine-check/check-stop condition, halting the CPU entirely. Another "traditional" technique is to ignore such errors, which can lead to data corruption, both of user data or of kernel data, hung/unresponsive adapters, or system crashes/lockups. Thus, the idea behind EEH is that the operating system can become more reliable and robust by protecting it from PCI errors, and giving the OS the ability to "reboot"/recover individual PCI devices.

Future systems from other vendors, based on the PCI-E specification, may contain similar features.

8.2 Causes of EEH Errors

EEH was originally designed to guard against hardware failure, such as PCI cards dying from heat, humidity, dust, vibration and bad electrical connections. The vast majority of EEH errors seen in "real life" are due to either poorly seated PCI cards, or, unfortunately quite commonly, due to device driver bugs, device firmware bugs, and sometimes PCI card hardware bugs.

The most common software bug, is one that causes the device to attempt to DMA to a location in system memory that has not been reserved for DMA access for that card. This is a powerful feature, as it prevents what; otherwise, would have been silent memory corruption caused by the bad DMA. A number of device driver

bugs have been found and fixed in this way over the past few years. Other possible causes of EEH errors include data or address line parity errors (for example, due to poor electrical connectivity due to a poorly seated card), and PCI-X split-completion errors (due to software, device firmware, or device PCI hardware bugs). The vast majority of "true hardware failures" can be cured by physically removing and re-seating the PCI card.

8.3 Detection and Recovery

In the following discussion, a generic overview of how to detect and recover from EEH errors will be presented. This is followed by an overview of how the current implementation in the Linux kernel does it. The actual implementation is subject to change, and some of the finer points are still being debated. These may in turn be swayed if or when other architectures implement similar functionality.

When a PCI Host Bridge (PHB, the bus controller connecting the PCI bus to the system CPU electronics complex) detects a PCI error condition, it will "isolate" the affected PCI card. Isolation will block all writes (either to the card from the system, or from the card to the system), and it will cause all reads to return all-ff's (0xff, 0xffff, 0xffffffff for 8/16/32-bit reads). This value was chosen because it is the same value you would get if the device was physically unplugged from the slot. This includes access to PCI memory, I/O space, and PCI config space. Interrupts; however, will continued to be delivered.

Detection and recovery are performed with the aid of ppc64 firmware. The programming interfaces in the Linux kernel into the firmware are referred to as RTAS (Run-Time Abstraction Services). The Linux kernel does not (should not) access the EEH function in the PCI chipsets directly, primarily because there are a number of different chipsets out there, each with different interfaces and quirks. The firmware provides a uniform abstraction layer that will work with all pSeries and iSeries hardware (and be forwards-compatible).

If the OS or device driver suspects that a PCI slot has been EEH-isolated, there is a firmware call it can make to determine if this is the case. If so, then the device driver should put itself into a consistent state (given that it won't be able to complete any pending work) and start recovery of the card. Recovery normally would consist of resetting the PCI device (holding the PCI #RST line high for two seconds), followed by setting up the device config space (the base address registers (BAR's), latency timer, cache line size, interrupt line, and so on). This is followed by a reinitialization of the device driver. In a worst-case scenario, the power to the card can be toggled, at least on hot-plug-capable slots. In principle, layers far above the device driver probably do not need to know that the PCI card has been "rebooted" in this way; ideally, there should be at most a pause in Ethernet/disk/USB I/O while the card is being reset.

If the card cannot be recovered after three or four resets, the kernel/device driver should assume the worst-case scenario, that the card has died completely, and report this error to the sysadmin. In addition, error messages are reported through RTAS and also through syslogd (/var/log/messages) to alert the sysadmin of PCI resets. The correct way to deal with failed adapters is to use the standard PCI hotplug tools to remove and replace the dead card.

8.4 Current PPC64 Linux EEH Implementation

At this time, a generic EEH recovery mechanism has been implemented, so that individual device drivers do not need to be modified to support EEH recovery. This generic mechanism piggy-backs on the PCI hotplug infrastructure, and percolates events up through the userspace/udev infrastructure. Following is a detailed description of how this is accomplished.

EEH must be enabled in the PHB's very early during the boot process, and if a PCI slot is hot-plugged. The former is performed by eeh_init() in arch/powerpc/platforms/pseries/eeh.c, and the later by drivers/pci/hotplug/pSeries_pci.c calling in to the eeh.c code. EEH must be enabled before a PCI scan of the device can proceed. Current Power5 hardware will not work unless EEH is enabled; although older Power4 can run with it disabled. Effectively, EEH can no longer be turned off. PCI devices must be registered with the EEH code; the EEH code needs to know about the I/O address ranges of the PCI device in order to detect an error. Given an arbitrary address, the routine pci_get_device_by_addr() will find the pci device associated with that address (if any).

The default arch/powerpc/include/asm/io.h macros readb(), inb(), insb(), etc. include a check to see if the i/o read returned all-0xff's. If so, these make a call to eeh_dn_check_failure(), which in turn asks the firmware if the all-ff's value is the sign of a true EEH error. If it is not, processing continues as normal. The grand total number of these false alarms or "false positives" can be seen in /proc/ppc64/eeh (subject to change). Normally, almost all of these occur during boot, when the PCI bus is scanned, where a large number of 0xff reads are part of the bus scan procedure.

If a frozen slot is detected, code in arch/powerpc/platforms/pseries/eeh.c will print a stack trace to syslog (/var/log/messages). This stack trace has proven to be very useful to device-driver authors for finding out at what point the EEH error was detected, as the error itself usually occurs slightly beforehand.

Next, it uses the Linux kernel notifier chain/work queue mechanism to allow any interested parties to find out about the failure. Device drivers, or other parts of the kernel, can use eeh_register_notifier(struct notifier_block *) to find out about EEH events. The event will include a pointer to the pci device, the device node and some state info. Receivers of the event can "do as they wish"; the default handler will be described further in this section.

To assist in the recovery of the device, eeh.c exports the following functions:

rtas_set_slot_reset() assert the PCI #RST line for 1/8th of a second

rtas_configure_bridge() ask firmware to configure any PCI bridges located topologically under the pci slot.

eeh_save_bars() and eeh_restore_bars(): save and restore the PCI config-space info for a device and any devices under it.

A handler for the EEH notifier_block events is implemented in drivers/pci/hotplug/pSeries_pci.c, called handle_eeh_events(). It saves the device BAR's and then calls rpaphp_unconfig_pci_adapter(). This last call causes the device driver for the card to be stopped, which causes uevents to go out to

user space. This triggers user-space scripts that might issue commands such as "ifdown eth0" for ethernet cards, and so on. This handler then sleeps for 5 seconds, hoping to give the user-space scripts enough time to complete. It then resets the PCI card, reconfigures the device BAR's, and any bridges underneath. It then calls rpaphp_enable_pci_slot(), which restarts the device driver and triggers more user-space events (for example, calling "ifup eth0" for ethernet cards).

8.5 Device Shutdown and User-Space Events

This section documents what happens when a pci slot is unconfigured, focusing on how the device driver gets shut down, and on how the events get delivered to user-space scripts.

Following is an example sequence of events that cause a device driver close function to be called during the first phase of an EEH reset. The following sequence is an example of the pcnet32 device driver:

```
rpa_php_unconfig_pci_adapter (struct slot *)  // in rpaphp_pci.c
{
  calls
  pci_remove_bus_device (struct pci_dev *) // in /drivers/pci/remove.c
  {
    calls
    pci_destroy_dev (struct pci_dev *)
    {
      calls
      device_unregister (&dev->dev) // in /drivers/base/core.c
      {
        calls
        device_del (struct device *)
        {
          calls
          bus_remove_device() // in /drivers/base/bus.c
          {
            calls
            device_release_driver()
            {
              calls
              struct device_driver->remove() which is just
              pci_device_remove()  // in /drivers/pci/pci_driver.c
              {
                calls
                struct pci_driver->remove() which is just
                pcnet32_remove_one() // in /drivers/net/pcnet32.c
                {
                  calls
                  unregister_netdev() // in /net/core/dev.c
                  {
                    calls
                    dev_close()  // in /net/core/dev.c
                    {
                       calls dev->stop();
                       which is just pcnet32_close() // in pcnet32.c
                       {
```

(continues on next page)

(continued from previous page)

```
                        which does what you wanted
                        to stop the device
                      }
                    }
                  }
                which
                frees pcnet32 device driver memory
              }
 }}}}}}
```

in drivers/pci/pci_driver.c, struct device_driver->remove() is just pci_device_remove() which calls struct pci_driver->remove() which is pcnet32_remove_one() which calls unregister_netdev() (in net/core/dev.c) which calls dev_close() (in net/core/dev.c) which calls dev->stop() which is pcnet32_close() which then does the appropriate shutdown.

—

Following is the analogous stack trace for events sent to user-space when the pci device is unconfigured:

```
rpa_php_unconfig_pci_adapter() {              // in rpaphp_pci.c
  calls
  pci_remove_bus_device (struct pci_dev *) { // in /drivers/pci/remove.c
    calls
    pci_destroy_dev (struct pci_dev *) {
      calls
      device_unregister (&dev->dev) {         // in /drivers/base/core.c
        calls
        device_del(struct device * dev) {     // in /drivers/base/core.c
          calls
          kobject_del() {                     //in /libs/kobject.c
            calls
            kobject_uevent() {                // in /libs/kobject.c
              calls
              kset_uevent() {                 // in /lib/kobject.c
                calls
                kset->uevent_ops->uevent()    // which is really just
                a call to
                dev_uevent() {                // in /drivers/base/core.c
                  calls
                  dev->bus->uevent() which is really just a call to
                  pci_uevent () {             // in drivers/pci/hotplug.c
                    which prints device name, etc....
                 }
               }
               then kobject_uevent() sends a netlink uevent to userspace
               --> userspace uevent
               (during early boot, nobody listens to netlink events and
               kobject_uevent() executes uevent_helper[], which runs the
               event process /sbin/hotplug)
           }
         }
         kobject_del() then calls sysfs_remove_dir(), which would
         trigger any user-space daemon that was watching /sysfs,
         and notice the delete event.
```

8.6 Pro's and Con's of the Current Design

There are several issues with the current EEH software recovery design, which may be addressed in future revisions. But first, note that the big plus of the current design is that no changes need to be made to individual device drivers, so that the current design throws a wide net. The biggest negative of the design is that it potentially disturbs network daemons and file systems that didn't need to be disturbed.

- A minor complaint is that resetting the network card causes user-space back-to-back ifdown/ifup burps that potentially disturb network daemons, that didn't need to even know that the pci card was being rebooted.
- A more serious concern is that the same reset, for SCSI devices, causes havoc to mounted file systems. Scripts cannot post-facto unmount a file system without flushing pending buffers, but this is impossible, because I/O has already been stopped. Thus, ideally, the reset should happen at or below the block layer, so that the file systems are not disturbed.

  Reiserfs does not tolerate errors returned from the block device. Ext3fs seems to be tolerant, retrying reads/writes until it does succeed. Both have been only lightly tested in this scenario.

  The SCSI-generic subsystem already has built-in code for performing SCSI device resets, SCSI bus resets, and SCSI host-bus-adapter (HBA) resets. These are cascaded into a chain of attempted resets if a SCSI command fails. These are completely hidden from the block layer. It would be very natural to add an EEH reset into this chain of events.
- If a SCSI error occurs for the root device, all is lost unless the sysadmin had the foresight to run /bin, /sbin, /etc, /var and so on, out of ramdisk/tmpfs.

8.7 Conclusions

There's forward progress ⋯

CHAPTER

NINE

ELF NOTE POWERPC NAMESPACE

The PowerPC namespace in an ELF Note of the kernel binary is used to store capabilities and information which can be used by a bootloader or userland.

9.1 Types and Descriptors

The types to be used with the "PowerPC" namesapce are defined in[1].

1) PPC_ELFNOTE_CAPABILITIES

Define the capabilities supported/required by the kernel. This type uses a bitmap as "descriptor" field. Each bit is described below:

- Ultravisor-capable bit (PowerNV only).

```
#define PPCCAP_ULTRAVISOR_BIT (1 << 0)
```

Indicate that the powerpc kernel binary knows how to run in an ultravisor-enabled system.

In an ultravisor-enabled system, some machine resources are now controlled by the ultravisor. If the kernel is not ultravisor-capable, but it ends up being run on a machine with ultravisor, the kernel will probably crash trying to access ultravisor resources. For instance, it may crash in early boot trying to set the partition table entry 0.

In an ultravisor-enabled system, a bootloader could warn the user or prevent the kernel from being run if the PowerPC ultravisor capability doesn't exist or the Ultravisor-capable bit is not set.

9.2 References

[1] arch/powerpc/include/asm/elfnote.h

CHAPTER

TEN

FIRMWARE-ASSISTED DUMP

July 2011

The goal of firmware-assisted dump is to enable the dump of a crashed system, and to do so from a fully-reset system, and to minimize the total elapsed time until the system is back in production use.

- Firmware-Assisted Dump (FADump) infrastructure is intended to replace the existing phyp assisted dump.
- Fadump uses the same firmware interfaces and memory reservation model as phyp assisted dump.
- Unlike phyp dump, FADump exports the memory dump through /proc/vmcore in the ELF format in the same way as kdump. This helps us reuse the kdump infrastructure for dump capture and filtering.
- Unlike phyp dump, userspace tool does not need to refer any sysfs interface while reading /proc/vmcore.
- Unlike phyp dump, FADump allows user to release all the memory reserved for dump, with a single operation of echo 1 > /sys/kernel/fadump_release_mem.
- Once enabled through kernel boot parameter, FADump can be started/stopped through /sys/kernel/fadump_registered interface (see sysfs files section below) and can be easily integrated with kdump service start/stop init scripts.

Comparing with kdump or other strategies, firmware-assisted dump offers several strong, practical advantages:

- Unlike kdump, the system has been reset, and loaded with a fresh copy of the kernel. In particular, PCI and I/O devices have been reinitialized and are in a clean, consistent state.
- Once the dump is copied out, the memory that held the dump is immediately available to the running kernel. And therefore, unlike kdump, FADump doesn' t need a 2nd reboot to get back the system to the production configuration.

The above can only be accomplished by coordination with, and assistance from the Power firmware. The procedure is as follows:

- The first kernel registers the sections of memory with the Power firmware for dump preservation during OS initialization. These registered sections of memory are reserved by the first kernel during early boot.

- When system crashes, the Power firmware will copy the registered low memory regions (boot memory) from source to destination area. It will also save hardware PTE's.

  NOTE: The term 'boot memory' means size of the low memory chunk that is required for a kernel to boot successfully when booted with restricted memory. By default, the boot memory size will be the larger of 5% of system RAM or 256MB. Alternatively, user can also specify boot memory size through boot parameter 'crashkernel=' which will override the default calculated size. Use this option if default boot memory size is not sufficient for second kernel to boot successfully. For syntax of crashkernel= parameter, refer to Documentation/admin-guide/kdump/kdump.rst. If any offset is provided in crashkernel= parameter, it will be ignored as FADump uses a predefined offset to reserve memory for boot memory dump preservation in case of a crash.

- After the low memory (boot memory) area has been saved, the firmware will reset PCI and other hardware state. It will not clear the RAM. It will then launch the bootloader, as normal.

- The freshly booted kernel will notice that there is a new node (rtas/ibm,kernel-dump on pSeries or ibm,opal/dump/mpipl-boot on OPAL platform) in the device tree, indicating that there is crash data available from a previous boot. During the early boot OS will reserve rest of the memory above boot memory size effectively booting with restricted memory size. This will make sure that this kernel (also, referred to as second kernel or capture kernel) will not touch any of the dump memory area.

- User-space tools will read /proc/vmcore to obtain the contents of memory, which holds the previous crashed kernel dump in ELF format. The userspace tools may copy this info to disk, or network, nas, san, iscsi, etc. as desired.

- Once the userspace tool is done saving dump, it will echo '1' to /sys/kernel/fadump_release_mem to release the reserved memory back to general use, except the memory required for next firmware-assisted dump registration.

  e.g.:

```
# echo 1 > /sys/kernel/fadump_release_mem
```

Please note that the firmware-assisted dump feature is only available on POWER6 and above systems on pSeries (PowerVM) platform and POWER9 and above systems with OP940 or later firmware versions on PowerNV (OPAL) platform. Note that, OPAL firmware exports ibm,opal/dump node when FADump is supported on PowerNV platform.

On OPAL based machines, system first boots into an intermittent kernel (referred to as petitboot kernel) before booting into the capture kernel. This kernel would have minimal kernel and/or userspace support to process crash data. Such kernel needs to preserve previously crash'ed kernel's memory for the subsequent capture kernel boot to process this crash data. Kernel config option CONFIG_PRESERVE_FA_DUMP has to be enabled on such kernel to ensure that crash data is preserved to process later.

– On OPAL based machines (PowerNV), if the kernel is build with
CONFIG_OPAL_CORE=y, OPAL memory at the time of crash is also exported as /sys/firmware/opal/mpipl/core file. This procfs file is helpful in debugging OPAL crashes with GDB. The kernel memory used for exporting this procfs file can be released by echo'ing '1' to /sys/firmware/opal/mpipl/release_core node.

e.g. # echo 1 > /sys/firmware/opal/mpipl/release_core

10.1 Implementation details:

During boot, a check is made to see if firmware supports this feature on that particular machine. If it does, then we check to see if an active dump is waiting for us. If yes then everything but boot memory size of RAM is reserved during early boot (See Fig. 2). This area is released once we finish collecting the dump from user land scripts (e.g. kdump scripts) that are run. If there is dump data, then the /sys/kernel/fadump_release_mem file is created, and the reserved memory is held.

If there is no waiting dump data, then only the memory required to hold CPU state, HPTE region, boot memory dump, FADump header and elfcore header, is usually reserved at an offset greater than boot memory size (see Fig. 1). This area is not released: this region will be kept permanently reserved, so that it can act as a receptacle for a copy of the boot memory content in addition to CPU state and HPTE region, in the case a crash does occur.

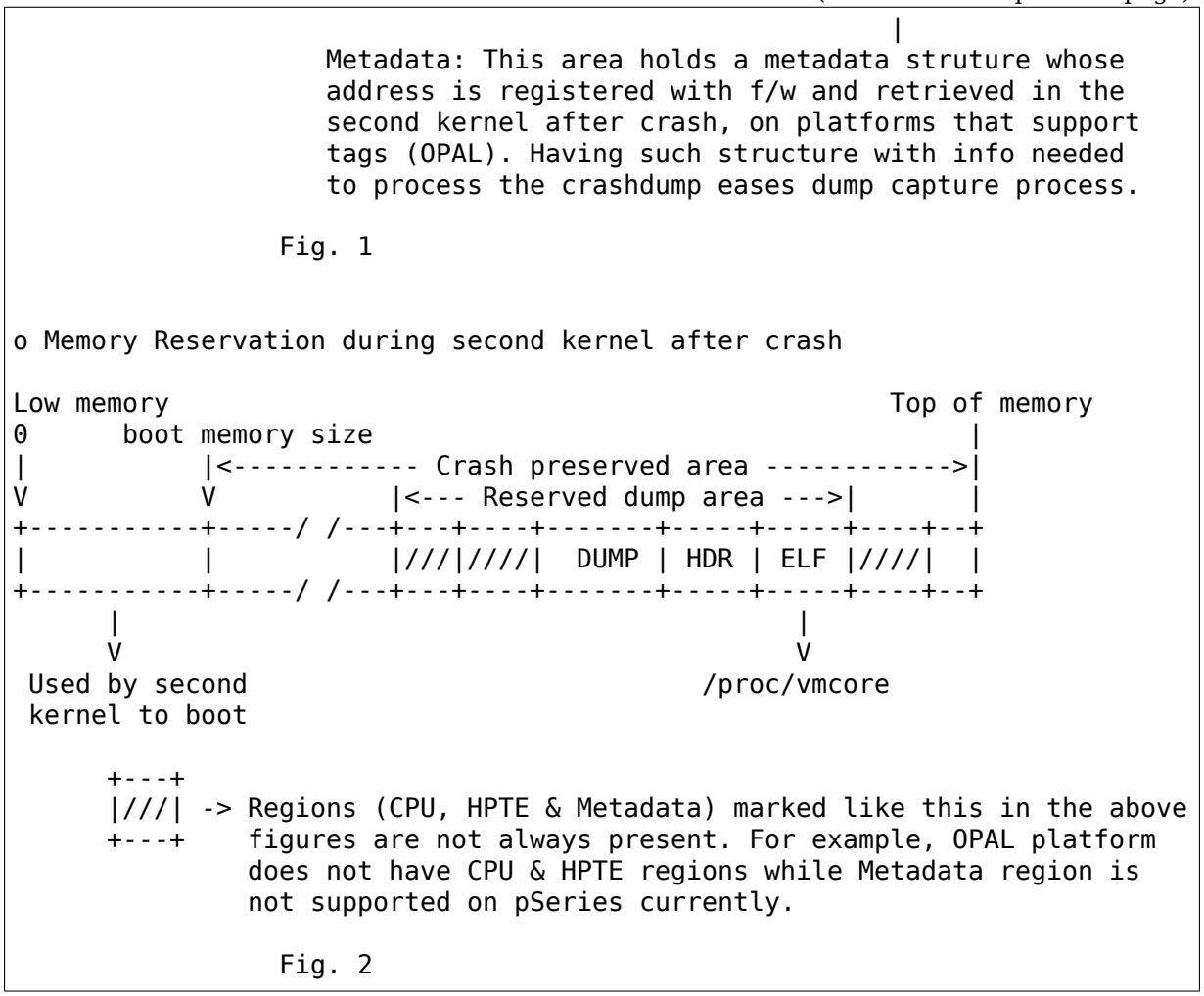
Since this reserved memory area is used only after the system crash, there is no point in blocking this significant chunk of memory from production kernel. Hence, the implementation uses the Linux kernel's Contiguous Memory Allocator (CMA) for memory reservation if CMA is configured for kernel. With CMA reservation this memory will be available for applications to use it, while kernel is prevented from using it. With this FADump will still be able to capture all of the kernel memory and most of the user space memory except the user pages that were present in CMA region:

```
o Memory Reservation during first kernel

Low memory                                                  Top of memory
0    boot memory size   |<--- Reserved dump area --->|       |
|           |           |    Permanent Reservation   |       |
V           V           |                            |       V
+-----------+-----/ /---+---+----+-------+-----+-----+----+--+
|           |           |///|////|  DUMP | HDR | ELF |////|  |
+-----------+-----/ /---+---+----+-------+-----+-----+----+--+
      |                    ^    ^     ^      ^           ^
      |                    |    |     |      |           |
      \                   CPU  HPTE   /      |           |
       ------------------------------        |           |
    Boot memory content gets transferred     |           |
    to reserved area by firmware at the      |           |
    time of crash.                           |           |
                                             |           |
                                        FADump Header    |
                                         (meta area)     |
                                                         |
```

(continues on next page)

(continued from previous page)



Currently the dump will be copied from /proc/vmcore to a new file upon user intervention. The dump data available through /proc/vmcore will be in ELF format. Hence the existing kdump infrastructure (kdump scripts) to save the dump works fine with minor modifications. KDump scripts on major Distro releases have already been modified to work seemlessly (no user intervention in saving the dump) when FADump is used, instead of KDump, as dump mechanism.

The tools to examine the dump will be same as the ones used for kdump.

10.2 How to enable firmware-assisted dump (FADump):

1. Set config option CONFIG_FA_DUMP=y and build kernel.
2. Boot into linux kernel with ‘fadump=on’ kernel cmdline option. By default, FADump reserved memory will be initialized as CMA area. Alternatively, user can boot linux kernel with ‘fadump=nocma’ to prevent FADump to use CMA.
3. Optionally, user can also set ‘crashkernel=’ kernel cmdline to specify size of the memory to reserve for boot memory dump preservation.

NOTE:

1. ‘fadump_reserve_mem=’ parameter has been deprecated. Instead use ‘crashkernel=’ to specify size of the memory to reserve for boot memory dump preservation.
2. If firmware-assisted dump fails to reserve memory then it will fallback to existing kdump mechanism if ‘crashkernel=’ option is set at kernel cmdline.
3. if user wants to capture all of user space memory and ok with reserved memory not available to production system, then ‘fadump=nocma’ kernel parameter can be used to fallback to old behaviour.

10.3 Sysfs/debugfs files:

Firmware-assisted dump feature uses sysfs file system to hold the control files and debugfs file to display memory reserved region.

Here is the list of files under kernel sysfs:

/sys/kernel/fadump_enabled This is used to display the FADump status.

- 0 = FADump is disabled
- 1 = FADump is enabled

This interface can be used by kdump init scripts to identify if FADump is enabled in the kernel and act accordingly.

/sys/kernel/fadump_registered This is used to display the FADump registration status as well as to control (start/stop) the FADump registration.

- 0 = FADump is not registered.
- 1 = FADump is registered and ready to handle system crash.

To register FADump echo 1 > /sys/kernel/fadump_registered and echo 0 > /sys/kernel/fadump_registered for un-register and stop the FADump. Once the FADump is un-registered, the system crash will not be handled and vmcore will not be captured. This interface can be easily integrated with kdump service start/stop.

/sys/kernel/fadump/mem_reserved

This is used to display the memory reserved by FADump for saving the crash dump.

/sys/kernel/fadump_release_mem This file is available only when FADump is active during second kernel. This is used to release the reserved memory region that are held for saving crash dump. To release the reserved memory echo 1 to it:

```
echo 1  > /sys/kernel/fadump_release_mem
```

After echo 1, the content of the /sys/kernel/debug/powerpc/fadump_region file will change to reflect the new memory reservations.

The existing userspace tools (kdump infrastructure) can be easily enhanced to use this interface to release the memory reserved for dump and continue without 2nd reboot.

Note: /sys/kernel/fadump_release_opalcore sysfs has moved to

/sys/firmware/opal/mpipl/release_core

/sys/firmware/opal/mpipl/release_core

This file is available only on OPAL based machines when FADump is active during capture kernel. This is used to release the memory used by the kernel to export /sys/firmware/opal/mpipl/core file. To release this memory, echo '1' to it:

echo 1 > /sys/firmware/opal/mpipl/release_core

Note: The following FADump sysfs files are deprecated.

Deprecated	Alternative
/sys/kernel/fadump_enabled	/sys/kernel/fadump/enabled
/sys/kernel/fadump_registered	/sys/kernel/fadump/registered
/sys/kernel/fadump_release_mem	/sys/kernel/fadump/release_mem

Here is the list of files under powerpc debugfs: (Assuming debugfs is mounted on /sys/kernel/debug directory.)

/sys/kernel/debug/powerpc/fadump_region This file shows the reserved memory regions if FADump is enabled otherwise this file is empty. The output format is:

```
<region>: [<start>-<end>] <reserved-size> bytes, Dumped:
↪<dump-size>
```

and for kernel DUMP region is:

DUMP: Src: <src-addr>, Dest: <dest-addr>, Size: <size>, Dumped: # bytes

e.g. Contents when FADump is registered during first kernel:

```
# cat /sys/kernel/debug/powerpc/fadump_region
CPU : [0x0000006ffb0000-0x0000006fff001f] 0x40020 bytes,␣
↪Dumped: 0x0
HPTE: [0x0000006fff0020-0x0000006fff101f] 0x1000 bytes,␣
↪Dumped: 0x0
DUMP: [0x0000006fff1020-0x0000007fff101f] 0x10000000 bytes,␣
↪Dumped: 0x0
```

Contents when FADump is active during second kernel:

```
# cat /sys/kernel/debug/powerpc/fadump_region
CPU : [0x0000006ffb0000-0x0000006fff001f] 0x40020 bytes,␣
↪Dumped: 0x40020
HPTE: [0x0000006fff0020-0x0000006fff101f] 0x1000 bytes,␣
↪Dumped: 0x1000
```

(continues on next page)

(continued from previous page)

```
DUMP: [0x0000006fff1020-0x0000007fff101f] 0x10000000 bytes,␣
↪Dumped: 0x10000000
    : [0x00000010000000-0x0000006ffaffff] 0x5ffb0000 bytes,␣
↪Dumped: 0x5ffb0000
```

NOTE: Please refer to Documentation/filesystems/debugfs.rst on how to mount the debugfs filesystem.

10.4 TODO:

- Need to come up with the better approach to find out more accurate boot memory size that is required for a kernel to boot successfully when booted with restricted memory.
- The FADump implementation introduces a FADump crash info structure in the scratch area before the ELF core header. The idea of introducing this structure is to pass some important crash info data to the second kernel which will help second kernel to populate ELF core header with correct data before it gets exported through /proc/vmcore. The current design implementation does not address a possibility of introducing additional fields (in future) to this structure without affecting compatibility. Need to come up with the better approach to address this.

  The possible approaches are:

  1. Introduce version field for version tracking, bump up the version whenever a new field is added to the structure in future. The version field can be used to find out what fields are valid for the current version of the structure. 2. Reserve the area of predefined size (say PAGE_SIZE) for this structure and have unused area as reserved (initialized to zero) for future field additions.

  The advantage of approach 1 over 2 is we don't need to reserve extra space.

Author: Mahesh Salgaonkar <mahesh@linux.vnet.ibm.com>

This document is based on the original documentation written for phyp

assisted dump by Linas Vepstas and Manish Ahuja.

CHAPTER

ELEVEN

HVCS IBM “HYPERVISOR VIRTUAL CONSOLE SERVER” INSTALLATION GUIDE

for Linux Kernel 2.6.4+

Author(s): Ryan S. Arnold <rsa@us.ibm.com>

Date Created: March, 02, 2004 Last Changed: August, 24, 2004

11.1 1. Driver Introduction:

This is the device driver for the IBM Hypervisor Virtual Console Server, “hvcs”. The IBM hvcs provides a tty driver interface to allow Linux user space applications access to the system consoles of logically partitioned operating systems (Linux and AIX) running on the same partitioned Power5 ppc64 system. Physical hardware consoles per partition are not practical on this hardware so system consoles are accessed by this driver using firmware interfaces to virtual terminal devices.

11.2 2. System Requirements:

This device driver was written using 2.6.4 Linux kernel APIs and will only build and run on kernels of this version or later.

This driver was written to operate solely on IBM Power5 ppc64 hardware though some care was taken to abstract the architecture dependent firmware calls from the driver code.

Sysfs must be mounted on the system so that the user can determine which major and minor numbers are associated with each vty-server. Directions for sysfs mounting are outside the scope of this document.

11.3 3. Build Options:

The hvcs driver registers itself as a tty driver. The tty layer dynamically allocates a block of major and minor numbers in a quantity requested by the registering driver. The hvcs driver asks the tty layer for 64 of these major/minor numbers by default to use for hvcs device node entries.

If the default number of device entries is adequate then this driver can be built into the kernel. If not, the default can be over-ridden by inserting the driver as a module with insmod parameters.

11.3.1 3.1 Built-in:

The following menuconfig example demonstrates selecting to build this driver into the kernel:

```
Device Drivers  --->
        Character devices  --->
                <*> IBM Hypervisor Virtual Console Server Support
```

Begin the kernel make process.

11.3.2 3.2 Module:

The following menuconfig example demonstrates selecting to build this driver as a kernel module:

```
Device Drivers  --->
        Character devices  --->
                <M> IBM Hypervisor Virtual Console Server Support
```

The make process will build the following kernel modules:

- hvcs.ko
- hvcserver.ko

To insert the module with the default allocation execute the following commands in the order they appear:

```
insmod hvcserver.ko
insmod hvcs.ko
```

The hvcserver module contains architecture specific firmware calls and must be inserted first, otherwise the hvcs module will not find some of the symbols it expects.

To override the default use an insmod parameter as follows (requesting 4 tty devices as an example):

```
insmod hvcs.ko hvcs_parm_num_devs=4
```

There is a maximum number of dev entries that can be specified on insmod. We think that 1024 is currently a decent maximum number of server adapters to allow.

This can always be changed by modifying the constant in the source file before building.

NOTE: The length of time it takes to insmod the driver seems to be related to the number of tty interfaces the registering driver requests.

In order to remove the driver module execute the following command:

```
rmmod hvcs.ko
```

The recommended method for installing hvcs as a module is to use depmod to build a current modules.dep file in /lib/modules/uname -r and then execute:

```
modprobe hvcs hvcs_parm_num_devs=4
```

The modules.dep file indicates that hvcserver.ko needs to be inserted before hvcs.ko and modprobe uses this file to smartly insert the modules in the proper order.

The following modprobe command is used to remove hvcs and hvcserver in the proper order:

```
modprobe -r hvcs
```

11.4 4. Installation:

The tty layer creates sysfs entries which contain the major and minor numbers allocated for the hvcs driver. The following snippet of "tree" output of the sysfs directory shows where these numbers are presented:

```
sys/
|-- *other sysfs base dirs*
|
|-- class
|   |-- *other classes of devices*
|   |
|   `-- tty
|       |-- *other tty devices*
|       |
|       |-- hvcs0
|       |   `-- dev
|       |-- hvcs1
|       |   `-- dev
|       |-- hvcs2
|       |   `-- dev
|       |-- hvcs3
|       |   `-- dev
|       |
|       |-- *other tty devices*
|
|-- *other sysfs base dirs*
```

For the above examples the following output is a result of cat'ing the "dev" entry in the hvcs directory:

```
Pow5:/sys/class/tty/hvcs0/ # cat dev
254:0

Pow5:/sys/class/tty/hvcs1/ # cat dev
254:1

Pow5:/sys/class/tty/hvcs2/ # cat dev
254:2

Pow5:/sys/class/tty/hvcs3/ # cat dev
254:3
```

The output from reading the "dev" attribute is the char device major and minor numbers that the tty layer has allocated for this driver's use. Most systems running hvcs will already have the device entries created or udev will do it automatically.

Given the example output above, to manually create a /dev/hvcs* node entry mknod can be used as follows:

```
mknod /dev/hvcs0 c 254 0
mknod /dev/hvcs1 c 254 1
mknod /dev/hvcs2 c 254 2
mknod /dev/hvcs3 c 254 3
```

Using mknod to manually create the device entries makes these device nodes persistent. Once created they will exist prior to the driver insmod.

Attempting to connect an application to /dev/hvcs* prior to insertion of the hvcs module will result in an error message similar to the following:

```
"/dev/hvcs*: No such device".
```

NOTE: Just because there is a device node present doesn't mean that there is a vty-server device configured for that node.

11.5 5. Connection

Since this driver controls devices that provide a tty interface a user can interact with the device node entries using any standard tty-interactive method (e.g. "cat", "dd", "echo"). The intent of this driver however, is to provide real time console interaction with a Linux partition's console, which requires the use of applications that provide bi-directional, interactive I/O with a tty device.

Applications (e.g. "minicom" and "screen") that act as terminal emulators or perform terminal type control sequence conversion on the data being passed through them are NOT acceptable for providing interactive console I/O. These programs often emulate antiquated terminal types (vt100 and ANSI) and expect inbound data to take the form of one of these supported terminal types but they either do not convert, or do not _adequately_ convert, outbound data into the terminal type of the terminal which invoked them (though screen makes an attempt and can apparently be configured with much termcap wrestling.)

For this reason kermit and cu are two of the recommended applications for interacting with a Linux console via an hvcs device. These programs simply act as a

conduit for data transfer to and from the tty device. They do not require inbound data to take the form of a particular terminal type, nor do they cook outbound data to a particular terminal type.

In order to ensure proper functioning of console applications one must make sure that once connected to a /dev/hvcs console that the console's $TERM env variable is set to the exact terminal type of the terminal emulator used to launch the interactive I/O application. If one is using xterm and kermit to connect to /dev/hvcs0 when the console prompt becomes available one should "export TERM=xterm" on the console. This tells ncurses applications that are invoked from the console that they should output control sequences that xterm can understand.

As a precautionary measure an hvcs user should always "exit" from their session before disconnecting an application such as kermit from the device node. If this is not done, the next user to connect to the console will continue using the previous user's logged in session which includes using the $TERM variable that the previous user supplied.

Hotplug add and remove of vty-server adapters affects which /dev/hvcs* node is used to connect to each vty-server adapter. In order to determine which vty-server adapter is associated with which /dev/hvcs* node a special sysfs attribute has been added to each vty-server sysfs entry. This entry is called "index" and showing it reveals an integer that refers to the /dev/hvcs* entry to use to connect to that device. For instance cating the index attribute of vty-server adapter 30000004 shows the following:

```
Pow5:/sys/bus/vio/drivers/hvcs/30000004 # cat index
2
```

This index of '2' means that in order to connect to vty-server adapter 30000004 the user should interact with /dev/hvcs2.

It should be noted that due to the system hotplug I/O capabilities of a system the /dev/hvcs* entry that interacts with a particular vty-server adapter is not guaranteed to remain the same across system reboots. Look in the Q & A section for more on this issue.

11.6 6. Disconnection

As a security feature to prevent the delivery of stale data to an unintended target the Power5 system firmware disables the fetching of data and discards that data when a connection between a vty-server and a vty has been severed. As an example, when a vty-server is immediately disconnected from a vty following output of data to the vty the vty adapter may not have enough time between when it received the data interrupt and when the connection was severed to fetch the data from firmware before the fetch is disabled by firmware.

When hvcs is being used to serve consoles this behavior is not a huge issue because the adapter stays connected for large amounts of time following almost all data writes. When hvcs is being used as a tty conduit to tunnel data between two partitions [see Q & A below] this is a huge problem because the standard Linux behavior when cat'ing or dd'ing data to a device is to open the tty, send the data, and then close the tty. If this driver manually terminated vty-server connections

on tty close this would close the vty-server and vty connection before the target vty has had a chance to fetch the data.

Additionally, disconnecting a vty-server and vty only on module removal or adapter removal is impractical because other vty-servers in other partitions may require the usage of the target vty at any time.

Due to this behavioral restriction disconnection of vty-servers from the connected vty is a manual procedure using a write to a sysfs attribute outlined below, on the other hand the initial vty-server connection to a vty is established automatically by this driver. Manual vty-server connection is never required.

In order to terminate the connection between a vty-server and vty the "vterm_state" sysfs attribute within each vty-server's sysfs entry is used. Reading this attribute reveals the current connection state of the vty-server adapter. A zero means that the vty-server is not connected to a vty. A one indicates that a connection is active.

Writing a '0' (zero) to the vterm_state attribute will disconnect the VTERM connection between the vty-server and target vty ONLY if the vterm_state previously read '1'. The write directive is ignored if the vterm_state read '0' or if any value other than '0' was written to the vterm_state attribute. The following example will show the method used for verifying the vty-server connection status and disconnecting a vty-server connection:

```
Pow5:/sys/bus/vio/drivers/hvcs/30000004 # cat vterm_state
1

Pow5:/sys/bus/vio/drivers/hvcs/30000004 # echo 0 > vterm_state

Pow5:/sys/bus/vio/drivers/hvcs/30000004 # cat vterm_state
0
```

All vty-server connections are automatically terminated when the device is hotplug removed and when the module is removed.

11.7 7. Configuration

Each vty-server has a sysfs entry in the /sys/devices/vio directory, which is symlinked in several other sysfs tree directories, notably under the hvcs driver entry, which looks like the following example:

```
Pow5:/sys/bus/vio/drivers/hvcs # ls
.  ..  30000003  30000004  rescan
```

By design, firmware notifies the hvcs driver of vty-server lifetimes and partner vty removals but not the addition of partner vtys. Since an HMC Super Admin can add partner info dynamically we have provided the hvcs driver sysfs directory with the "rescan" update attribute which will query firmware and update the partner info for all the vty-servers that this driver manages. Writing a '1' to the attribute triggers the update. An explicit example follows:

```
Pow5:/sys/bus/vio/drivers/hvcs # echo 1 > rescan
```

Reading the attribute will indicate a state of ‘1’ or ‘0’. A one indicates that an update is in process. A zero indicates that an update has completed or was never executed.

Vty-server entries in this directory are a 32 bit partition unique unit address that is created by firmware. An example vty-server sysfs entry looks like the following:

```
Pow5:/sys/bus/vio/drivers/hvcs/30000004 # ls
.   current_vty   devspec       name          partner_vtys
..  index         partner_clcs  vterm_state
```

Each entry is provided, by default with a “name” attribute. Reading the “name” attribute will reveal the device type as shown in the following example:

```
Pow5:/sys/bus/vio/drivers/hvcs/30000003 # cat name
vty-server
```

Each entry is also provided, by default, with a “devspec” attribute which reveals the full device specification when read, as shown in the following example:

```
Pow5:/sys/bus/vio/drivers/hvcs/30000004 # cat devspec
/vdevice/vty-server@30000004
```

Each vty-server sysfs dir is provided with two read-only attributes that provide lists of easily parsed partner vty data: “partner_vtys” and “partner_clcs”:

```
Pow5:/sys/bus/vio/drivers/hvcs/30000004 # cat partner_vtys
30000000
30000001
30000002
30000000
30000000

Pow5:/sys/bus/vio/drivers/hvcs/30000004 # cat partner_clcs
U5112.428.103048A-V3-C0
U5112.428.103048A-V3-C2
U5112.428.103048A-V3-C3
U5112.428.103048A-V4-C0
U5112.428.103048A-V5-C0
```

Reading partner_vtys returns a list of partner vtys. Vty unit address numbering is only per-partition-unique so entries will frequently repeat.

Reading partner_clcs returns a list of “converged location codes” which are composed of a system serial number followed by “-V*”, where the ‘*’ is the target partition number, and “-C*”, where the ‘*’ is the slot of the adapter. The first vty partner corresponds to the first clc item, the second vty partner to the second clc item, etc.

A vty-server can only be connected to a single vty at a time. The entry, “current_vty” prints the clc of the currently selected partner vty when read.

The current_vty can be changed by writing a valid partner clc to the entry as in the following example:

```
Pow5:/sys/bus/vio/drivers/hvcs/30000004 # echo U5112.428.10304
8A-V4-C0 > current_vty
```

Changing the current_vty when a vty-server is already connected to a vty does not affect the current connection. The change takes effect when the currently open connection is freed.

Information on the "vterm_state" attribute was covered earlier on the chapter entitled "disconnection".

11.8 8. Questions & Answers:

Q: What are the security concerns involving hvcs?

A: There are three main security concerns:

1. The creator of the /dev/hvcs* nodes has the ability to restrict the access of the device entries to certain users or groups. It may be best to create a special hvcs group privilege for providing access to system consoles.

2. To provide network security when grabbing the console it is suggested that the user connect to the console hosting partition using a secure method, such as SSH or sit at a hardware console.

3. Make sure to exit the user session when done with a console or the next vty-server connection (which may be from another partition) will experience the previously logged in session.

Q: How do I multiplex a console that I grab through hvcs so that other people can see it:

A: You can use "screen" to directly connect to the /dev/hvcs* device and setup a session on your machine with the console group privileges. As pointed out earlier by default screen doesn't provide the termcap settings for most terminal emulators to provide adequate character conversion from term type "screen" to others. This means that curses based programs may not display properly in screen sessions.

Q: Why are the colors all messed up? Q: Why are the control characters acting strange or not working? Q: Why is the console output all strange and unintelligible?

A: Please see the preceding section on "Connection" for a discussion of how applications can affect the display of character control sequences. Additionally, just because you logged into the console using and xterm doesn't mean someone else didn't log into the console with the HMC console (vt320) before you and leave the session logged in. The best thing to do is to export TERM to the terminal type of your terminal emulator when you get the console. Additionally make sure to "exit" the console before you disconnect from the console. This will ensure that the next user gets their own TERM type set when they login.

Q: When I try to CONNECT kermit to an hvcs device I get: "Sorry, can't open connection: /dev/hvcs*" What is happening?

A: Some other Power5 console mechanism has a connection to the vty and isn't giving it up. You can try to force disconnect the consoles from the HMC by right clicking on the partition and then selecting "close terminal". Otherwise you have to hunt down the people who have console authority. It is possible that you already have the console open using another kermit session and just forgot about it. Please review the console options for Power5 systems to determine the many ways a system console can be held.

OR

A: Another user may not have a connectivity method currently attached to a /dev/hvcs device but the vterm_state may reveal that they still have the vty-server connection established. They need to free this using the method outlined in the section on "Disconnection" in order for others to connect to the target vty.

OR

A: The user profile you are using to execute kermit probably doesn't have permissions to use the /dev/hvcs* device.

OR

A: You probably haven't inserted the hvcs.ko module yet but the /dev/hvcs* entry still exists (on systems without udev).

OR

A: There is not a corresponding vty-server device that maps to an existing /dev/hvcs* entry.

Q: When I try to CONNECT kermit to an hvcs device I get: "Sorry, write access to UUCP lockfile directory denied."

A: The /dev/hvcs* entry you have specified doesn't exist where you said it does? Maybe you haven't inserted the module (on systems with udev).

Q: If I already have one Linux partition installed can I use hvcs on said partition to provide the console for the install of a second Linux partition?

A: Yes granted that your are connected to the /dev/hvcs* device using kermit or cu or some other program that doesn't provide terminal emulation.

Q: Can I connect to more than one partition's console at a time using this driver?

A: Yes. Of course this means that there must be more than one vty-server configured for this partition and each must point to a disconnected vty.

Q: Does the hvcs driver support dynamic (hotplug) addition of devices?

A: Yes, if you have dlpar and hotplug enabled for your system and it has been built into the kernel the hvcs drivers is configured to dynamically handle additions of new devices and removals of unused devices.

Q: For some reason /dev/hvcs* doesn't map to the same vty-server adapter after a reboot. What happened?

A: Assignment of vty-server adapters to /dev/hvcs* entries is always done in the order that the adapters are exposed. Due to hotplug capabilities of this driver assignment of hotplug added vty-servers may be in a different order than how they would be exposed on module load. Rebooting or reloading the module after dynamic addition may result in the /dev/hvcs* and vty-server coupling changing if a vty-server adapter was added in a slot between two other vty-server adapters. Refer to the section above on how to determine which vty-server goes with which /dev/hvcs* node. Hint; look at the sysfs "index" attribute for the vty-server.

Q: Can I use /dev/hvcs* as a conduit to another partition and use a tty device on that partition as the other end of the pipe?

A: Yes, on Power5 platforms the hvc_console driver provides a tty interface for extra /dev/hvc* devices (where /dev/hvc0 is most likely the console). In order to get a tty conduit working between the two partitions the HMC Super Admin must create an additional "serial server" for the target partition with the HMC gui which will show up as /dev/hvc* when the target partition is rebooted.

The HMC Super Admin then creates an additional "serial client" for the current partition and points this at the target partition's newly created "serial server" adapter (remember the slot). This shows up as an additional /dev/hvcs* device.

Now a program on the target system can be configured to read or write to /dev/hvc* and another program on the current partition can be configured to read or write to /dev/hvcs*. Now you have a tty conduit between two partitions.

11.9 9. Reporting Bugs:

The proper channel for reporting bugs is either through the Linux OS distribution company that provided your OS or by posting issues to the PowerPC development mailing list at:

linuxppc-dev@lists.ozlabs.org

This request is to provide a documented and searchable public exchange of the problems and solutions surrounding this driver for the benefit of all users.

CHAPTER

TWELVE

IMC (IN-MEMORY COLLECTION COUNTERS)

Anju T Sudhakar, 10 May 2019

Contents

12.1 Basic overview

IMC (In-Memory collection counters) is a hardware monitoring facility that collects large numbers of hardware performance events at Nest level (these are on-chip but off-core), Core level and Thread level.

The Nest PMU counters are handled by a Nest IMC microcode which runs in the OCC (On-Chip Controller) complex. The microcode collects the counter data and moves the nest IMC counter data to memory.

The Core and Thread IMC PMU counters are handled in the core. Core level PMU counters give us the IMC counters' data per core and thread level PMU counters give us the IMC counters' data per CPU thread.

OPAL obtains the IMC PMU and supported events information from the IMC Catalog and passes on to the kernel via the device tree. The event' s information contains:

- Event name
- Event Offset
- Event description

and possibly also:

- Event scale
- Event unit

Some PMUs may have a common scale and unit values for all their supported events. For those cases, the scale and unit properties for those events must be inherited from the PMU.

The event offset in the memory is where the counter data gets accumulated.

IMC catalog is available at: https://github.com/open-power/ima-catalog

The kernel discovers the IMC counters information in the device tree at the imc-counters device node which has a compatible field ibm,opal-in-memory-counters. From the device tree, the kernel parses the PMUs and their event' s information and register the PMU and its attributes in the kernel.

12.2 IMC example usage

```
# perf list
[...]
nest_mcs01/PM_MCS01_64B_RD_DISP_PORT01/                 [Kernel PMU event]
nest_mcs01/PM_MCS01_64B_RD_DISP_PORT23/                 [Kernel PMU event]
[...]
core_imc/CPM_0THRD_NON_IDLE_PCYC/                       [Kernel PMU event]
core_imc/CPM_1THRD_NON_IDLE_INST/                       [Kernel PMU event]
[...]
thread_imc/CPM_0THRD_NON_IDLE_PCYC/                     [Kernel PMU event]
thread_imc/CPM_1THRD_NON_IDLE_INST/                     [Kernel PMU event]
```

To see per chip data for nest_mcs0/PM_MCS_DOWN_128B_DATA_XFER_MC0/:

```
# ./perf stat -e "nest_mcs01/PM_MCS01_64B_WR_DISP_PORT01/" -a --per-socket
```

To see non-idle instructions for core 0:

```
# ./perf stat -e "core_imc/CPM_NON_IDLE_INST/" -C 0 -I 1000
```

To see non-idle instructions for a "make":

```
# ./perf stat -e "thread_imc/CPM_NON_IDLE_PCYC/" make
```

12.3 IMC Trace-mode

POWER9 supports two modes for IMC which are the Accumulation mode and Trace mode. In Accumulation mode, event counts are accumulated in system Memory. Hypervisor then reads the posted counts periodically or when requested. In IMC Trace mode, the 64 bit trace SCOM value is initialized with the event information. The CPMCxSEL and CPMC_LOAD in the trace SCOM, specifies the event to be monitored and the sampling duration. On each overflow in the CPMCxSEL,

hardware snapshots the program counter along with event counts and writes into memory pointed by LDBAR.

LDBAR is a 64 bit special purpose per thread register, it has bits to indicate whether hardware is configured for accumulation or trace mode.

12.3.1 LDBAR Register Layout

0	Enable/Disable
1	0: Accumulation Mode
	1: Trace Mode
2:3	Reserved
4-6	PB scope
7	Reserved
8:50	Counter Address
51:63	Reserved

12.3.2 TRACE_IMC_SCOM bit representation

0:1	SAMPSEL
2:33	CPMC_LOAD
34:40	CPMC1SEL
41:47	CPMC2SEL
48:50	BUFFERSIZE
51:63	RESERVED

CPMC_LOAD contains the sampling duration. SAMPSEL and CPMCxSEL determines the event to count. BUFFERSIZE indicates the memory range. On each overflow, hardware snapshots the program counter along with event counts and updates the memory and reloads the CMPC_LOAD value for the next sampling duration. IMC hardware does not support exceptions, so it quietly wraps around if memory buffer reaches the end.

Currently the event monitored for trace-mode is fixed as cycle.

12.4 Trace IMC example usage

```
# perf list
[....]
trace_imc/trace_cycles/                                 [Kernel PMU event]
```

To record an application/process with trace-imc event:

```
# perf record -e trace_imc/trace_cycles/ yes > /dev/null
[ perf record: Woken up 1 times to write data ]
[ perf record: Captured and wrote 0.012 MB perf.data (21 samples) ]
```

The perf.data generated, can be read using perf report.

12.5 Benefits of using IMC trace-mode

PMI (Performance Monitoring Interrupts) interrupt handling is avoided, since IMC trace mode snapshots the program counter and updates to the memory. And this also provide a way for the operating system to do instruction sampling in real time without PMI processing overhead.

Performance data using perf top with and without trace-imc event.

PMI interrupts count when perf top command is executed without trace-imc event.

```
# grep PMI /proc/interrupts
PMI:          0          0          0          0   Performance monitoring␣
↪interrupts
# ./perf top
...
# grep PMI /proc/interrupts
PMI:      39735       8710      17338      17801   Performance monitoring␣
↪interrupts
# ./perf top -e trace_imc/trace_cycles/
...
# grep PMI /proc/interrupts
PMI:      39735       8710      17338      17801   Performance monitoring␣
↪interrupts
```

That is, the PMI interrupt counts do not increment when using the trace_imc event.

CHAPTER

THIRTEEN

CPU TO ISA VERSION MAPPING

Mapping of some CPU versions to relevant ISA versions.

CPU	Architecture version
Power9	Power ISA v3.0B
Power8	Power ISA v2.07
Power7	Power ISA v2.06
Power6	Power ISA v2.05
PA6T	Power ISA v2.04
Cell PPU	• Power ISA v2.02 with some minor exceptions • Plus Altivec/VMX ~= 2.03
Power5++	Power ISA v2.04 (no VMX)
Power5+	Power ISA v2.03
Power5	• PowerPC User Instruction Set Architecture Book I v2.02 • PowerPC Virtual Environment Architecture Book II v2.02 • PowerPC Operating Environment Architecture Book III v2.02
PPC970	• PowerPC User Instruction Set Architecture Book I v2.01 • PowerPC Virtual Environment Architecture Book II v2.01 • PowerPC Operating Environment Architecture Book III v2.01 • Plus Altivec/VMX ~= 2.03

13.1 Key Features

CPU	VMX (aka. Altivec)
Power9	Yes
Power8	Yes
Power7	Yes
Power6	Yes
PA6T	Yes
Cell PPU	Yes
Power5++	No
Power5+	No
Power5	No
PPC970	Yes

CPU	VSX
Power9	Yes
Power8	Yes
Power7	Yes
Power6	No
PA6T	No
Cell PPU	No
Power5++	No
Power5+	No
Power5	No
PPC970	No

CPU	Transactional Memory
Power9	Yes (* see transactional_memory.txt)
Power8	Yes
Power7	No
Power6	No
PA6T	No
Cell PPU	No
Power5++	No
Power5+	No
Power5	No
PPC970	No

CHAPTER

FOURTEEN

KASLR FOR FREESCALE BOOKE32

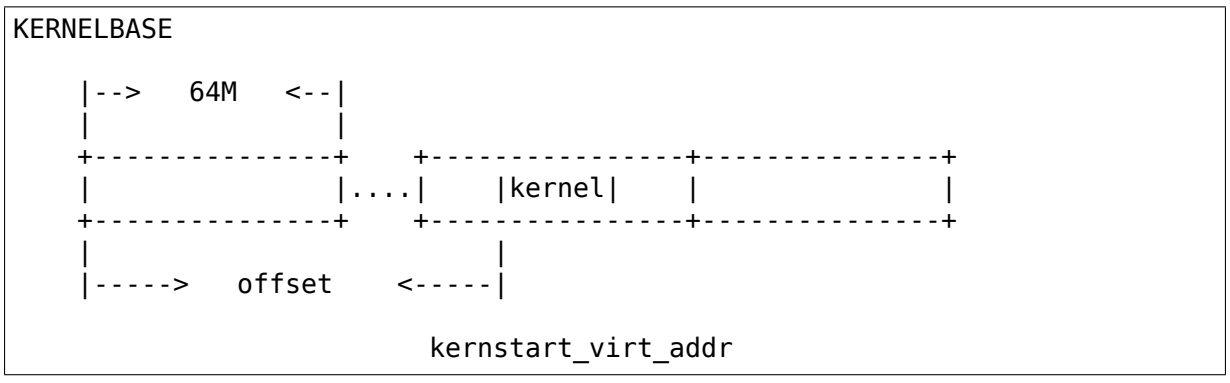
The word KASLR stands for Kernel Address Space Layout Randomization.

This document tries to explain the implementation of the KASLR for Freescale BookE32. KASLR is a security feature that deters exploit attempts relying on knowledge of the location of kernel internals.

Since CONFIG_RELOCATABLE has already supported, what we need to do is map or copy kernel to a proper place and relocate. Freescale Book-E parts expect lowmem to be mapped by fixed TLB entries(TLB1). The TLB1 entries are not suitable to map the kernel directly in a randomized region, so we chose to copy the kernel to a proper place and restart to relocate.

Entropy is derived from the banner and timer base, which will change every build and boot. This not so much safe so additionally the bootloader may pass entropy via the /chosen/kaslr-seed node in device tree.

We will use the first 512M of the low memory to randomize the kernel image. The memory will be split in 64M zones. We will use the lower 8 bit of the entropy to decide the index of the 64M zone. Then we chose a 16K aligned offset inside the 64M zone to put the kernel in:



To enable KASLR, set CONFIG_RANDOMIZE_BASE = y. If KASLR is enable and you want to disable it at runtime, add “nokaslr” to the kernel cmdline.

CHAPTER

FIFTEEN

LINUX 2.6.X ON MPC52XX FAMILY

For the latest info, go to http://www.246tNt.com/mpc52xx/

To compile/use :

- U-Boot:

```
# <edit Makefile to set ARCH=ppc & CROSS_COMPILE=... ( also␣
↪EXTRAVERSION
   if you wish to ).
# make lite5200_defconfig
# make uImage

then, on U-boot:
=> tftpboot 200000 uImage
=> tftpboot 400000 pRamdisk
=> bootm 200000 400000
```

- DBug:

```
# <edit Makefile to set ARCH=ppc & CROSS_COMPILE=... ( also␣
↪EXTRAVERSION
   if you wish to ).
# make lite5200_defconfig
# cp your_initrd.gz arch/ppc/boot/images/ramdisk.image.gz
# make zImage.initrd
# make

then in DBug:
DBug> dn -i zImage.initrd.lite5200
```

Some remarks:

- The port is named mpc52xxx, and config options are PPC_MPC52xx. The MGT5100 is not supported, and I'm not sure anyone is interesting in working on it so. I didn't took 5xxx because there's apparently a lot of 5xxx that have nothing to do with the MPC5200. I also included the 'MPC' for the same reason.
- Of course, I inspired myself from the 2.4 port. If you think I forgot to mention you/your company in the copyright of some code, I'll correct it ASAP.

CHAPTER

SIXTEEN

HYPERCALL OP-CODES (HCALLS)

16.1 Overview

Virtualization on 64-bit Power Book3S Platforms is based on the PAPR specification[1] which describes the run-time environment for a guest operating system and how it should interact with the hypervisor for privileged operations. Currently there are two PAPR compliant hypervisors:

- **IBM PowerVM (PHYP)**: IBM' s proprietary hypervisor that supports AIX, IBM-i and Linux as supported guests (termed as Logical Partitions or LPARS). It supports the full PAPR specification.
- **Qemu/KVM**: Supports PPC64 linux guests running on a PPC64 linux host. Though it only implements a subset of PAPR specification called LoPAPR[2].

On PPC64 arch a guest kernel running on top of a PAPR hypervisor is called a pSeries guest. A pseries guest runs in a supervisor mode (HV=0) and must issue hypercalls to the hypervisor whenever it needs to perform an action that is hypervisor priviledged[3] or for other services managed by the hypervisor.

Hence a Hypercall (hcall) is essentially a request by the pseries guest asking hypervisor to perform a privileged operation on behalf of the guest. The guest issues a with necessary input operands. The hypervisor after performing the privilege operation returns a status code and output operands back to the guest.

16.2 HCALL ABI

The ABI specification for a hcall between a pseries guest and PAPR hypervisor is covered in section 14.5.3 of ref[2]. Switch to the Hypervisor context is done via the instruction **HVCS** that expects the Opcode for hcall is set in r3 and any in-arguments for the hcall are provided in registers r4-r12. If values have to be passed through a memory buffer, the data stored in that buffer should be in Big-endian byte order.

[1] "Power Architecture Platform Reference" https://en.wikipedia.org/wiki/Power_Architecture_Platform_Reference

[2] "Linux on Power Architecture Platform Reference" https://members.openpowerfoundation.org/document/dl/469

[3] "Definitions and Notation" Book III-Section 14.5.3 https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0

Once control is returns back to the guest after hypervisor has serviced the 'HVCS' instruction the return value of the hcall is available in r3 and any out values are returned in registers r4-r12. Again like in case of in-arguments, any out values stored in a memory buffer will be in Big-endian byte order.

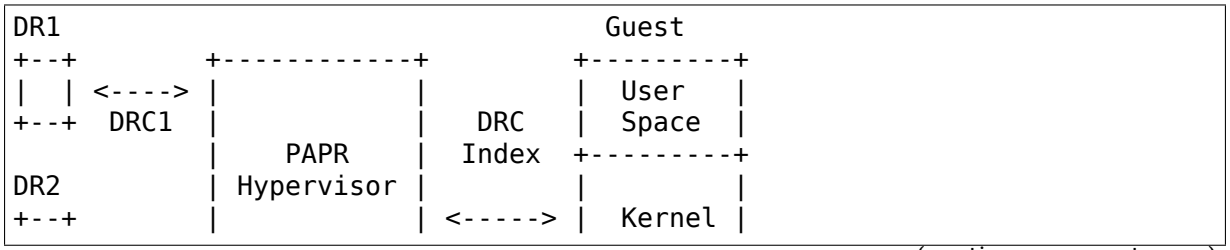
Powerpc arch code provides convenient wrappers named **plpar_hcall_xxx** defined in a arch specific header[4] to issue hcalls from the linux kernel running as pseries guest.

16.3 Register Conventions

Any hcall should follow same register convention as described in section 2.2.1.1 of "64-Bit ELF V2 ABI Specification: Power Architecture"[5]. Table below summarizes these conventions:

Register Range	Volatile (Y/N)	Purpose
r0	Y	Optional-usage
r1	N	Stack Pointer
r2	N	TOC
r3	Y	hcall opcode/return value
r4-r10	Y	in and out values
r11	Y	Optional-usage/Environmental pointer
r12	Y	Optional-usage/Function entry address at global entry point
r13	N	Thread-Pointer
r14-r31	N	Local Variables
LR	Y	Link Register
CTR	Y	Loop Counter
XER	Y	Fixed-point exception register.
CR0-1	Y	Condition register fields.
CR2-4	N	Condition register fields.
CR5-7	Y	Condition register fields.
Others	N	

16.4 DRC & DRC Indexes



(continues on next page)

[4] arch/powerpc/include/asm/hvcall.h

[5] "64-Bit ELF V2 ABI Specification: Power Architecture" https://openpowerfoundation.org/?resource_lib=64-bit-elf-v2-abi-specification-power-architecture

(continued from previous page)

```
|  | <----> |            |  Hcall  |         |
+--+  DRC2  +------------+         +---------+
```

PAPR hypervisor terms shared hardware resources like PCI devices, NVDIMMs etc available for use by LPARs as Dynamic Resource (DR). When a DR is allocated to an LPAR, PHYP creates a data-structure called Dynamic Resource Connector (DRC) to manage LPAR access. An LPAR refers to a DRC via an opaque 32-bit number called DRC-Index. The DRC-index value is provided to the LPAR via device-tree where its present as an attribute in the device tree node associated with the DR.

16.5 HCALL Return-values

After servicing the hcall, hypervisor sets the return-value in r3 indicating success or failure of the hcall. In case of a failure an error code indicates the cause for error. These codes are defined and documented in arch specific header[4].

In some cases a hcall can potentially take a long time and need to be issued multiple times in order to be completely serviced. These hcalls will usually accept an opaque value continue-token within there argument list and a return value of H_CONTINUE indicates that hypervisor hasn't still finished servicing the hcall yet.

To make such hcalls the guest need to set continue-token == 0 for the initial call and use the hypervisor returned value of continue-token for each subsequent hcall until hypervisor returns a non H_CONTINUE return value.

16.6 HCALL Op-codes

Below is a partial list of HCALLs that are supported by PHYP. For the corresponding opcode values please look into the arch specific header[4]:

H_SCM_READ_METADATA

Input: drcIndex, offset, buffer-address, numBytesToRead
Out: numBytesRead
Return Value: H_Success, H_Parameter, H_P2, H_P3, H_Hardware

Given a DRC Index of an NVDIMM, read N-bytes from the the metadata area associated with it, at a specified offset and copy it to provided buffer. The metadata area stores configuration information such as label information, bad-blocks etc. The metadata area is located out-of-band of NVDIMM storage area hence a separate access semantics is provided.

H_SCM_WRITE_METADATA

Input: drcIndex, offset, data, numBytesToWrite

Out: None
Return Value: H_Success, H_Parameter, H_P2, H_P4, H_Hardware

Given a DRC Index of an NVDIMM, write N-bytes to the metadata area associated with it, at the specified offset and from the provided buffer.

H_SCM_BIND_MEM

Input: drcIndex, startingScmBlockIndex, numScmBlocksToBind,
targetLogicalMemoryAddress, continue-token
Out: continue-token, targetLogicalMemoryAddress, numScmBlocksToBound
Return Value: H_Success, H_Parameter, H_P2, H_P3, H_P4, H_Overlap,
H_Too_Big, H_P5, H_Busy

Given a DRC-Index of an NVDIMM, map a continuous SCM blocks range (startingScmBlockIndex, startingScmBlockIndex+numScmBlocksToBind) to the guest at targetLogicalMemoryAddress within guest physical address space. In case targetLogicalMemoryAddress == 0xFFFFFFFF_FFFFFFFF then hypervisor assigns a target address to the guest. The HCALL can fail if the Guest has an active PTE entry to the SCM block being bound.

H_SCM_UNBIND_MEM | Input: drcIndex, startingScmLogicalMemoryAddress, numScmBlocksToUnbind | Out: numScmBlocksUnbound | Return Value: H_Success, H_Parameter, H_P2, H_P3, H_In_Use, H_Overlap, | H_Busy, H_LongBusyOrder1mSec, H_LongBusyOrder10mSec

Given a DRC-Index of an NVDimm, unmap numScmBlocksToUnbind SCM blocks starting at startingScmLogicalMemoryAddress from guest physical address space. The HCALL can fail if the Guest has an active PTE entry to the SCM block being unbound.

H_SCM_QUERY_BLOCK_MEM_BINDING

Input: drcIndex, scmBlockIndex
Out: Guest-Physical-Address
Return Value: H_Success, H_Parameter, H_P2, H_NotFound

Given a DRC-Index and an SCM Block index return the guest physical address to which the SCM block is mapped to.

H_SCM_QUERY_LOGICAL_MEM_BINDING

Input: Guest-Physical-Address
Out: drcIndex, scmBlockIndex
Return Value: H_Success, H_Parameter, H_P2, H_NotFound

Given a guest physical address return which DRC Index and SCM block is mapped to that address.

H_SCM_UNBIND_ALL

Input: scmTargetScope, drcIndex
Out: None
Return Value: H_Success, H_Parameter, H_P2, H_P3, H_In_Use, H_Busy, H_LongBusyOrder1mSec, H_LongBusyOrder10mSec

Depending on the Target scope unmap all SCM blocks belonging to all NVDIMMs or all SCM blocks belonging to a single NVDIMM identified by its drcIndex from the LPAR memory.

H_SCM_HEALTH

Input: drcIndex
Out: health-bitmap (r4), health-bit-valid-bitmap (r5)
Return Value: H_Success, H_Parameter, H_Hardware

Given a DRC Index return the info on predictive failure and overall health of the PMEM device. The asserted bits in the health-bitmap indicate one or more states (described in table below) of the PMEM device and health-bit-valid-bitmap indicate which bits in health-bitmap are valid. The bits are reported in reverse bit ordering for example a value of 0xC400000000000000 indicates bits 0, 1, and 5 are valid.

Health Bitmap Flags:

Bit	Definition
00	PMEM device is unable to persist memory contents. If the system is powered down, nothing will be saved.
01	PMEM device failed to persist memory contents. Either contents were not saved successfully on power down or were not restored properly on power up.
02	PMEM device contents are persisted from previous IPL. The data from the last boot were successfully restored.
03	PMEM device contents are not persisted from previous IPL. There was no data to restore from the last boot.
04	PMEM device memory life remaining is critically low
05	PMEM device will be garded off next IPL due to failure
06	PMEM device contents cannot persist due to current platform health status. A hardware failure may prevent data from being saved or restored.
07	PMEM device is unable to persist memory contents in certain conditions
08	PMEM device is encrypted
09	PMEM device has successfully completed a requested erase or secure erase procedure.
10:63	Reserved / Unused

H_SCM_PERFORMANCE_STATS

Input: drcIndex, resultBuffer Addr
Out: None
Return Value: H_Success, H_Parameter, H_Unsupported, H_Hardware, H_Authority, H_Privilege

Given a DRC Index collect the performance statistics for NVDIMM and copy them to the resultBuffer.

16.7 References

CHAPTER

SEVENTEEN

PCI EXPRESS I/O VIRTUALIZATION RESOURCE ON POWERENV

Wei Yang <weiyang@linux.vnet.ibm.com>

Benjamin Herrenschmidt <benh@au1.ibm.com>

Bjorn Helgaas <bhelgaas@google.com>

26 Aug 2014

This document describes the requirement from hardware for PCI MMIO resource sizing and assignment on PowerKVM and how generic PCI code handles this requirement. The first two sections describe the concepts of Partitionable Endpoints and the implementation on P8 (IODA2). The next two sections talks about considerations on enabling SRIOV on IODA2.

17.1 1. Introduction to Partitionable Endpoints

A Partitionable Endpoint (PE) is a way to group the various resources associated with a device or a set of devices to provide isolation between partitions (i.e., filtering of DMA, MSIs etc.) and to provide a mechanism to freeze a device that is causing errors in order to limit the possibility of propagation of bad data.

There is thus, in HW, a table of PE states that contains a pair of "frozen" state bits (one for MMIO and one for DMA, they get set together but can be cleared independently) for each PE.

When a PE is frozen, all stores in any direction are dropped and all loads return all 1's value. MSIs are also blocked. There's a bit more state that captures things like the details of the error that caused the freeze etc., but that's not critical.

The interesting part is how the various PCIe transactions (MMIO, DMA, ⋯) are matched to their corresponding PEs.

The following section provides a rough description of what we have on P8 (IODA2). Keep in mind that this is all per PHB (PCI host bridge). Each PHB is a completely separate HW entity that replicates the entire logic, so has its own set of PEs, etc.

17.2 2. Implementation of Partitionable Endpoints on P8 (IODA2)

P8 supports up to 256 Partitionable Endpoints per PHB.

- Inbound

  For DMA, MSIs and inbound PCIe error messages, we have a table (in memory but accessed in HW by the chip) that provides a direct correspondence between a PCIe RID (bus/dev/fn) with a PE number. We call this the RTT.

  - For DMA we then provide an entire address space for each PE that can contain two "windows", depending on the value of PCI address bit 59. Each window can be configured to be remapped via a "TCE table" (IOMMU translation table), which has various configurable characteristics not described here.
  - For MSIs, we have two windows in the address space (one at the top of the 32-bit space and one much higher) which, via a combination of the address and MSI value, will result in one of the 2048 interrupts per bridge being triggered. There's a PE# in the interrupt controller descriptor table as well which is compared with the PE# obtained from the RTT to "authorize" the device to emit that specific interrupt.
  - Error messages just use the RTT.

- Outbound. That's where the tricky part is.

  Like other PCI host bridges, the Power8 IODA2 PHB supports "windows" from the CPU address space to the PCI address space. There is one M32 window and sixteen M64 windows. They have different characteristics. First what they have in common: they forward a configurable portion of the CPU address space to the PCIe bus and must be naturally aligned power of two in size. The rest is different:

  - The M32 window:
    * Is limited to 4GB in size.
    * Drops the top bits of the address (above the size) and replaces them with a configurable value. This is typically used to generate 32-bit PCIe accesses. We configure that window at boot from FW and don't touch it from Linux; it's usually set to forward a 2GB portion of address space from the CPU to PCIe 0x8000_0000..0xffff_ffff. (Note: The top 64KB are actually reserved for MSIs but this is not a problem at this point; we just need to ensure Linux doesn't assign anything there, the M32 logic ignores that however and will forward in that space if we try).
    * It is divided into 256 segments of equal size. A table in the chip maps each segment to a PE#. That allows portions of the MMIO space to be assigned to PEs on a segment granularity. For a 2GB window, the segment granularity is 2GB/256 = 8MB.

  Now, this is the "main" window we use in Linux today (excluding SR-IOV). We basically use the trick of forcing the bridge MMIO windows onto a segment

alignment/granularity so that the space behind a bridge can be assigned to a PE.

Ideally we would like to be able to have individual functions in PEs but that would mean using a completely different address allocation scheme where individual function BARs can be "grouped" to fit in one or more segments.

- The M64 windows:
  - Must be at least 256MB in size.
  - Do not translate addresses (the address on PCIe is the same as the address on the PowerBus). There is a way to also set the top 14 bits which are not conveyed by PowerBus but we don't use this.
  - Can be configured to be segmented. When not segmented, we can specify the PE# for the entire window. When segmented, a window has 256 segments; however, there is no table for mapping a segment to a PE#. The segment number is the PE#.
  - Support overlaps. If an address is covered by multiple windows, there's a defined ordering for which window applies.

We have code (fairly new compared to the M32 stuff) that exploits that for large BARs in 64-bit space:

We configure an M64 window to cover the entire region of address space that has been assigned by FW for the PHB (about 64GB, ignore the space for the M32, it comes out of a different "reserve"). We configure it as segmented.

Then we do the same thing as with M32, using the bridge alignment trick, to match to those giant segments.

Since we cannot remap, we have two additional constraints:

- We do the PE# allocation after the 64-bit space has been assigned because the addresses we use directly determine the PE#. We then update the M32 PE# for the devices that use both 32-bit and 64-bit spaces or assign the remaining PE# to 32-bit only devices.
- We cannot "group" segments in HW, so if a device ends up using more than one segment, we end up with more than one PE#. There is a HW mechanism to make the freeze state cascade to "companion" PEs but that only works for PCIe error messages (typically used so that if you freeze a switch, it freezes all its children). So we do it in SW. We lose a bit of effectiveness of EEH in that case, but that's the best we found. So when any of the PEs freezes, we freeze the other ones for that "domain". We thus introduce the concept of "master PE" which is the one used for DMA, MSIs, etc., and "secondary PEs" that are used for the remaining M64 segments.

We would like to investigate using additional M64 windows in "single PE" mode to overlay over specific BARs to work around some of that, for example for devices with very large BARs, e.g., GPUs. It would make sense, but we haven't done it yet.

17.3 3. Considerations for SR-IOV on PowerKVM

- SR-IOV Background

  The PCIe SR-IOV feature allows a single Physical Function (PF) to support several Virtual Functions (VFs). Registers in the PF's SR-IOV Capability control the number of VFs and whether they are enabled.

  When VFs are enabled, they appear in Configuration Space like normal PCI devices, but the BARs in VF config space headers are unusual. For a non-VF device, software uses BARs in the config space header to discover the BAR sizes and assign addresses for them. For VF devices, software uses VF BAR registers in the PF SR-IOV Capability to discover sizes and assign addresses. The BARs in the VF's config space header are read-only zeros.

  When a VF BAR in the PF SR-IOV Capability is programmed, it sets the base address for all the corresponding VF(n) BARs. For example, if the PF SR-IOV Capability is programmed to enable eight VFs, and it has a 1MB VF BAR0, the address in that VF BAR sets the base of an 8MB region. This region is divided into eight contiguous 1MB regions, each of which is a BAR0 for one of the VFs. Note that even though the VF BAR describes an 8MB region, the alignment requirement is for a single VF, i.e., 1MB in this example.

There are several strategies for isolating VFs in PEs:

- M32 window: There's one M32 window, and it is split into 256 equally-sized segments. The finest granularity possible is a 256MB window with 1MB segments. VF BARs that are 1MB or larger could be mapped to separate PEs in this window. Each segment can be individually mapped to a PE via the lookup table, so this is quite flexible, but it works best when all the VF BARs are the same size. If they are different sizes, the entire window has to be small enough that the segment size matches the smallest VF BAR, which means larger VF BARs span several segments.
- Non-segmented M64 window: A non-segmented M64 window is mapped entirely to a single PE, so it could only isolate one VF.
- Single segmented M64 windows: A segmented M64 window could be used just like the M32 window, but the segments can't be individually mapped to PEs (the segment number is the PE#), so there isn't as much flexibility. A VF with multiple BARs would have to be in a "domain" of multiple PEs, which is not as well isolated as a single PE.
- Multiple segmented M64 windows: As usual, each window is split into 256 equally-sized segments, and the segment number is the PE#. But if we use several M64 windows, they can be set to different base addresses and different segment sizes. If we have VFs that each have a 1MB BAR and a 32MB BAR, we could use one M64 window to assign 1MB segments and another M64 window to assign 32MB segments.

Finally, the plan to use M64 windows for SR-IOV, which will be described more in the next two sections. For a given VF BAR, we need to effectively reserve the entire 256 segments (256 * VF BAR size) and position the VF BAR to start at the beginning of a free range of segments/PEs inside that M64 window.

The goal is of course to be able to give a separate PE for each VF.

The IODA2 platform has 16 M64 windows, which are used to map MMIO range to PE#. Each M64 window defines one MMIO range and this range is divided into 256 segments, with each segment corresponding to one PE.

We decide to leverage this M64 window to map VFs to individual PEs, since SR-IOV VF BARs are all the same size.

But doing so introduces another problem: total_VFs is usually smaller than the number of M64 window segments, so if we map one VF BAR directly to one M64 window, some part of the M64 window will map to another device's MMIO range.

IODA supports 256 PEs, so segmented windows contain 256 segments, so if total_VFs is less than 256, we have the situation in Figure 1.0, where segments [total_VFs, 255] of the M64 window may map to some MMIO range on other devices:

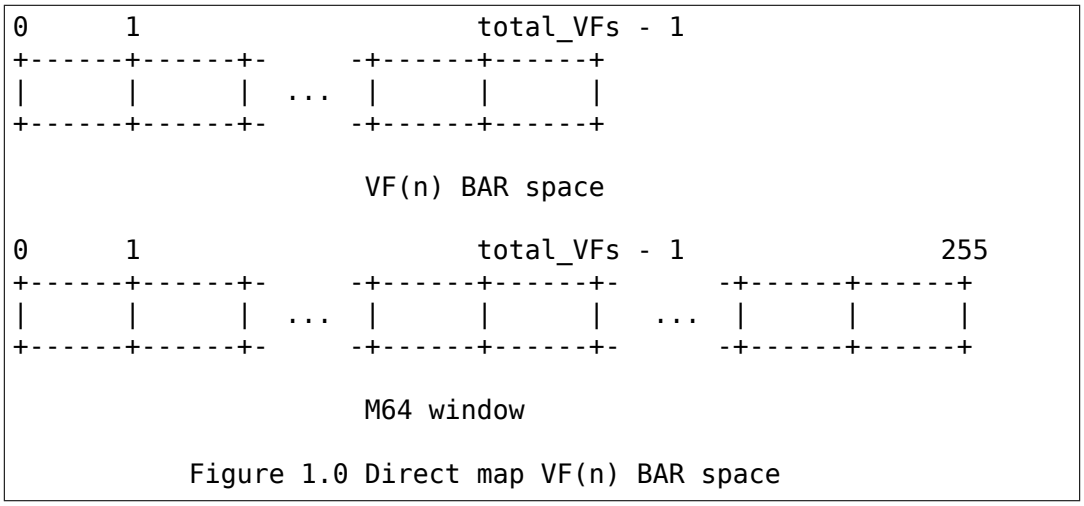
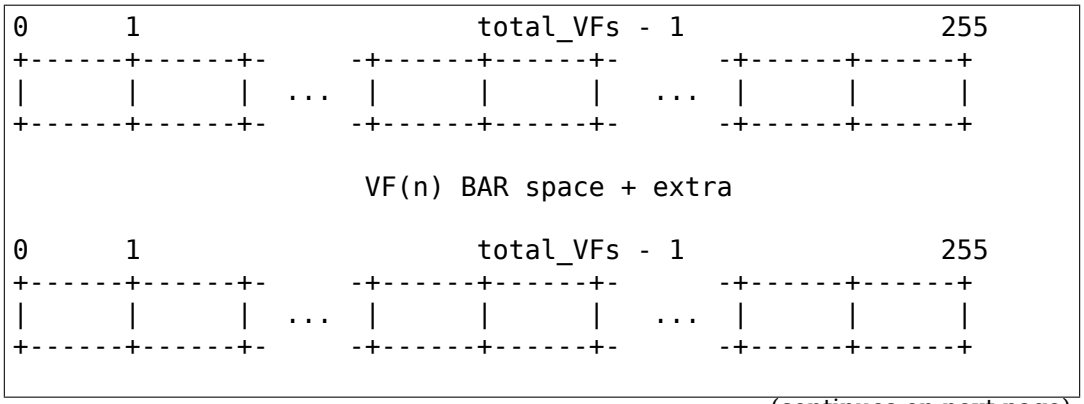


Figure 1.0 Direct map VF(n) BAR space

Our current solution is to allocate 256 segments even if the VF(n) BAR space doesn't need that much, as shown in Figure 1.1:



(continues on next page)

(continued from previous page)

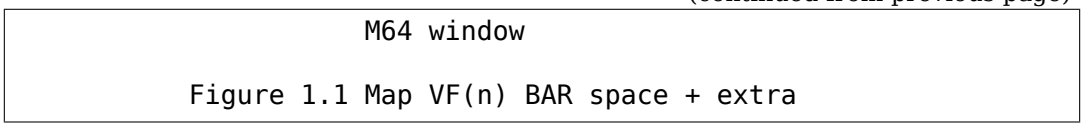


Figure 1.1 Map VF(n) BAR space + extra

Allocating the extra space ensures that the entire M64 window will be assigned to this one SR-IOV device and none of the space will be available for other devices. Note that this only expands the space reserved in software; there are still only total_VFs VFs, and they only respond to segments [0, total_VFs - 1]. There's nothing in hardware that responds to segments [total_VFs, 255].

17.4 4. Implications for the Generic PCI Code

The PCIe SR-IOV spec requires that the base of the VF(n) BAR space be aligned to the size of an individual VF BAR.

In IODA2, the MMIO address determines the PE#. If the address is in an M32 window, we can set the PE# by updating the table that translates segments to PE#s. Similarly, if the address is in an unsegmented M64 window, we can set the PE# for the window. But if it's in a segmented M64 window, the segment number is the PE#.

Therefore, the only way to control the PE# for a VF is to change the base of the VF(n) BAR space in the VF BAR. If the PCI core allocates the exact amount of space required for the VF(n) BAR space, the VF BAR value is fixed and cannot be changed.

On the other hand, if the PCI core allocates additional space, the VF BAR value can be changed as long as the entire VF(n) BAR space remains inside the space allocated by the core.

Ideally the segment size will be the same as an individual VF BAR size. Then each VF will be in its own PE. The VF BARs (and therefore the PE#s) are contiguous. If VF0 is in PE(x), then VF(n) is in PE(x+n). If we allocate 256 segments, there are (256 - numVFs) choices for the PE# of VF0.

If the segment size is smaller than the VF BAR size, it will take several segments to cover a VF BAR, and a VF will be in several PEs. This is possible, but the isolation isn't as good, and it reduces the number of PE# choices because instead of consuming only numVFs segments, the VF(n) BAR space will consume (numVFs * n) segments. That means there aren't as many available segments for adjusting base of the VF(n) BAR space.

CHAPTER

EIGHTEEN

PMU EVENT BASED BRANCHES

Event Based Branches (EBBs) are a feature which allows the hardware to branch directly to a specified user space address when certain events occur.

The full specification is available in Power ISA v2.07:

https://www.power.org/documentation/power-isa-version-2-07/

One type of event for which EBBs can be configured is PMU exceptions. This document describes the API for configuring the Power PMU to generate EBBs, using the Linux perf_events API.

18.1 Terminology

Throughout this document we will refer to an "EBB event" or "EBB events". This just refers to a struct perf_event which has set the "EBB" flag in its attr.config. All events which can be configured on the hardware PMU are possible "EBB events".

18.2 Background

When a PMU EBB occurs it is delivered to the currently running process. As such EBBs can only sensibly be used by programs for self-monitoring.

It is a feature of the perf_events API that events can be created on other processes, subject to standard permission checks. This is also true of EBB events, however unless the target process enables EBBs (via mtspr(BESCR)) no EBBs will ever be delivered.

This makes it possible for a process to enable EBBs for itself, but not actually configure any events. At a later time another process can come along and attach an EBB event to the process, which will then cause EBBs to be delivered to the first process. It's not clear if this is actually useful.

When the PMU is configured for EBBs, all PMU interrupts are delivered to the user process. This means once an EBB event is scheduled on the PMU, no non-EBB events can be configured. This means that EBB events can not be run concurrently with regular 'perf' commands, or any other perf events.

It is however safe to run ‘perf’ commands on a process which is using EBBs. The kernel will in general schedule the EBB event, and perf will be notified that its events could not run.

The exclusion between EBB events and regular events is implemented using the existing “pinned” and “exclusive” attributes of perf_events. This means EBB events will be given priority over other events, unless they are also pinned. If an EBB event and a regular event are both pinned, then whichever is enabled first will be scheduled and the other will be put in error state. See the section below titled “Enabling an EBB event” for more information.

18.3 Creating an EBB event

To request that an event is counted using EBB, the event code should have bit 63 set.

EBB events must be created with a particular, and restrictive, set of attributes - this is so that they interoperate correctly with the rest of the perf_events subsystem.

An EBB event must be created with the “pinned” and “exclusive” attributes set. Note that if you are creating a group of EBB events, only the leader can have these attributes set.

An EBB event must NOT set any of the “inherit”, “sample_period”, “freq” or “enable_on_exec” attributes.

An EBB event must be attached to a task. This is specified to perf_event_open() by passing a pid value, typically 0 indicating the current task.

All events in a group must agree on whether they want EBB. That is all events must request EBB, or none may request EBB.

EBB events must specify the PMC they are to be counted on. This ensures userspace is able to reliably determine which PMC the event is scheduled on.

18.4 Enabling an EBB event

Once an EBB event has been successfully opened, it must be enabled with the perf_events API. This can be achieved either via the ioctl() interface, or the prctl() interface.

However, due to the design of the perf_events API, enabling an event does not guarantee that it has been scheduled on the PMU. To ensure that the EBB event has been scheduled on the PMU, you must perform a read() on the event. If the read() returns EOF, then the event has not been scheduled and EBBs are not enabled.

This behaviour occurs because the EBB event is pinned and exclusive. When the EBB event is enabled it will force all other non-pinned events off the PMU. In this case the enable will be successful. However if there is already an event pinned on the PMU then the enable will not be successful.

18.5 Reading an EBB event

It is possible to read() from an EBB event. However the results are meaningless. Because interrupts are being delivered to the user process the kernel is not able to count the event, and so will return a junk value.

18.6 Closing an EBB event

When an EBB event is finished with, you can close it using close() as for any regular event. If this is the last EBB event the PMU will be deconfigured and no further PMU EBBs will be delivered.

18.7 EBB Handler

The EBB handler is just regular userspace code, however it must be written in the style of an interrupt handler. When the handler is entered all registers are live (possibly) and so must be saved somehow before the handler can invoke other code.

It’ s up to the program how to handle this. For C programs a relatively simple option is to create an interrupt frame on the stack and save registers there.

18.8 Fork

EBB events are not inherited across fork. If the child process wishes to use EBBs it should open a new event for itself. Similarly the EBB state in BESCR/EBBHR/EBBRR is cleared across fork().

CHAPTER

NINETEEN

PTRACE

GDB intends to support the following hardware debug features of BookE processors:

4 hardware breakpoints (IAC) 2 hardware watchpoints (read, write and read-write) (DAC) 2 value conditions for the hardware watchpoints (DVC)

For that, we need to extend ptrace so that GDB can query and set these resources. Since we're extending, we're trying to create an interface that's extendable and that covers both BookE and server processors, so that GDB doesn't need to special-case each of them. We added the following 3 new ptrace requests.

19.1 1. PTRACE_PPC_GETHWDEBUGINFO

Query for GDB to discover the hardware debug features. The main info to be returned here is the minimum alignment for the hardware watchpoints. BookE processors don't have restrictions here, but server processors have an 8-byte alignment restriction for hardware watchpoints. We'd like to avoid adding special cases to GDB based on what it sees in AUXV.

Since we're at it, we added other useful info that the kernel can return to GDB: this query will return the number of hardware breakpoints, hardware watchpoints and whether it supports a range of addresses and a condition. The query will fill the following structure provided by the requesting process:

```
struct ppc_debug_info {
     unit32_t version;
     unit32_t num_instruction_bps;
     unit32_t num_data_bps;
     unit32_t num_condition_regs;
     unit32_t data_bp_alignment;
     unit32_t sizeof_condition; /* size of the DVC register */
     uint64_t features; /* bitmask of the individual flags */
};
```

features will have bits indicating whether there is support for:

```
#define PPC_DEBUG_FEATURE_INSN_BP_RANGE                0x1
#define PPC_DEBUG_FEATURE_INSN_BP_MASK                 0x2
#define PPC_DEBUG_FEATURE_DATA_BP_RANGE                0x4
#define PPC_DEBUG_FEATURE_DATA_BP_MASK                 0x8
#define PPC_DEBUG_FEATURE_DATA_BP_DAWR                 0x10
```

2. PTRACE_SETHWDEBUG

Sets a hardware breakpoint or watchpoint, according to the provided structure:

```
 struct ppc_hw_breakpoint {
       uint32_t version;
 #define PPC_BREAKPOINT_TRIGGER_EXECUTE  0x1
 #define PPC_BREAKPOINT_TRIGGER_READ     0x2
#define PPC_BREAKPOINT_TRIGGER_WRITE    0x4
       uint32_t trigger_type;       /* only some combinations allowed */
 #define PPC_BREAKPOINT_MODE_EXACT               0x0
 #define PPC_BREAKPOINT_MODE_RANGE_INCLUSIVE     0x1
 #define PPC_BREAKPOINT_MODE_RANGE_EXCLUSIVE     0x2
 #define PPC_BREAKPOINT_MODE_MASK                0x3
       uint32_t addr_mode;          /* address match mode */

 #define PPC_BREAKPOINT_CONDITION_MODE   0x3
 #define PPC_BREAKPOINT_CONDITION_NONE   0x0
 #define PPC_BREAKPOINT_CONDITION_AND    0x1
 #define PPC_BREAKPOINT_CONDITION_EXACT  0x1   /* different name for the␣
↪same thing as above */
 #define PPC_BREAKPOINT_CONDITION_OR     0x2
 #define PPC_BREAKPOINT_CONDITION_AND_OR 0x3
 #define PPC_BREAKPOINT_CONDITION_BE_ALL 0x00ff0000    /* byte enable bits␣
↪*/
 #define PPC_BREAKPOINT_CONDITION_BE(n)  (1<<((n)+16))
       uint32_t condition_mode;     /* break/watchpoint condition flags */

       uint64_t addr;
       uint64_t addr2;
       uint64_t condition_value;
 };
```

A request specifies one event, not necessarily just one register to be set. For instance, if the request is for a watchpoint with a condition, both the DAC and DVC registers will be set in the same request.

With this GDB can ask for all kinds of hardware breakpoints and watchpoints that the BookE supports. COMEFROM breakpoints available in server processors are not contemplated, but that is out of the scope of this work.

ptrace will return an integer (handle) uniquely identifying the breakpoint or watchpoint just created. This integer will be used in the PTRACE_DELHWDEBUG request to ask for its removal. Return -ENOSPC if the requested breakpoint can't be allocated on the registers.

Some examples of using the structure to:

- set a breakpoint in the first breakpoint register:

```
p.version         = PPC_DEBUG_CURRENT_VERSION;
p.trigger_type    = PPC_BREAKPOINT_TRIGGER_EXECUTE;
p.addr_mode       = PPC_BREAKPOINT_MODE_EXACT;
p.condition_mode  = PPC_BREAKPOINT_CONDITION_NONE;
p.addr            = (uint64_t) address;
p.addr2           = 0;
p.condition_value = 0;
```

- set a watchpoint which triggers on reads in the second watchpoint register:

```
p.version         = PPC_DEBUG_CURRENT_VERSION;
p.trigger_type    = PPC_BREAKPOINT_TRIGGER_READ;
p.addr_mode       = PPC_BREAKPOINT_MODE_EXACT;
p.condition_mode  = PPC_BREAKPOINT_CONDITION_NONE;
p.addr            = (uint64_t) address;
p.addr2           = 0;
p.condition_value = 0;
```

- set a watchpoint which triggers only with a specific value:

```
p.version         = PPC_DEBUG_CURRENT_VERSION;
p.trigger_type    = PPC_BREAKPOINT_TRIGGER_READ;
p.addr_mode       = PPC_BREAKPOINT_MODE_EXACT;
p.condition_mode  = PPC_BREAKPOINT_CONDITION_AND | PPC_BREAKPOINT_
↪CONDITION_BE_ALL;
p.addr            = (uint64_t) address;
p.addr2           = 0;
p.condition_value = (uint64_t) condition;
```

- set a ranged hardware breakpoint:

```
p.version         = PPC_DEBUG_CURRENT_VERSION;
p.trigger_type    = PPC_BREAKPOINT_TRIGGER_EXECUTE;
p.addr_mode       = PPC_BREAKPOINT_MODE_RANGE_INCLUSIVE;
p.condition_mode  = PPC_BREAKPOINT_CONDITION_NONE;
p.addr            = (uint64_t) begin_range;
p.addr2           = (uint64_t) end_range;
p.condition_value = 0;
```

- set a watchpoint in server processors (BookS):

```
p.version         = 1;
p.trigger_type    = PPC_BREAKPOINT_TRIGGER_RW;
p.addr_mode       = PPC_BREAKPOINT_MODE_RANGE_INCLUSIVE;
or
p.addr_mode       = PPC_BREAKPOINT_MODE_EXACT;

p.condition_mode  = PPC_BREAKPOINT_CONDITION_NONE;
p.addr            = (uint64_t) begin_range;
/* For PPC_BREAKPOINT_MODE_RANGE_INCLUSIVE addr2 needs to be␣
↪specified, where
 * addr2 - addr <= 8 Bytes.
 */
p.addr2           = (uint64_t) end_range;
p.condition_value = 0;
```

3. PTRACE_DELHWDEBUG

Takes an integer which identifies an existing breakpoint or watchpoint (i.e., the value returned from PTRACE_SETHWDEBUG), and deletes the corresponding breakpoint or watchpoint..

CHAPTER

TWENTY

FREESCALE QUICC ENGINE FIRMWARE UPLOADING

20.1 Revision Information

November 30, 2007: Rev 1.0 - Initial version

20.2 I - Software License for Firmware

Each firmware file comes with its own software license. For information on the particular license, please see the license text that is distributed with the firmware.

20.3 II - Microcode Availability

Firmware files are distributed through various channels. Some are available on http://opensource.freescale.com. For other firmware files, please contact your Freescale representative or your operating system vendor.

20.4 III - Description and Terminology

In this document, the term ‘microcode’ refers to the sequence of 32-bit integers that compose the actual QE microcode.

The term ‘firmware’ refers to a binary blob that contains the microcode as well as other data that

1) describes the microcode’s purpose
2) describes how and where to upload the microcode
3) specifies the values of various registers
4) includes additional data for use by specific device drivers

Firmware files are binary files that contain only a firmware.

20.5 IV - Microcode Programming Details

The QE architecture allows for only one microcode present in I-RAM for each RISC processor. To replace any current microcode, a full QE reset (which disables the microcode) must be performed first.

QE microcode is uploaded using the following procedure:

1) The microcode is placed into I-RAM at a specific location, using the IRAM.IADD and IRAM.IDATA registers.
2) The CERCR.CIR bit is set to 0 or 1, depending on whether the firmware needs split I-RAM. Split I-RAM is only meaningful for SOCs that have QEs with multiple RISC processors, such as the 8360. Splitting the I-RAM allows each processor to run a different microcode, effectively creating an asymmetric multiprocessing (AMP) system.
3) The TIBCR trap registers are loaded with the addresses of the trap handlers in the microcode.
4) The RSP.ECCR register is programmed with the value provided.
5) If necessary, device drivers that need the virtual traps and extended mode data will use them.

Virtual Microcode Traps

These virtual traps are conditional branches in the microcode. These are “soft”provisional introduced in the ROMcode in order to enable higher flexibility and save h/w traps If new features are activated or an issue is being fixed in the RAM package utilizing they should be activated. This data structure signals the microcode which of these virtual traps is active.

This structure contains 6 words that the application should copy to some specific been defined. This table describes the structure:

Offset in array	Protocol	Destination Offset within PRAM	Size of Operand
0	Ethernet interworking	0xF8	4 bytes
4	ATM interworking	0xF8	4 bytes
8	PPP interworking	0xF8	4 bytes
12	Ethernet RX Distributor Page	0x22	1 byte
16	ATM Globtal Params Table	0x28	1 byte
20	Insert Frame	0xF8	4 bytes

Extended Modes

This is a double word bit array (64 bits) that defines special functionality which has an impact on the software drivers. Each bit has its own impact and has special instructions for the s/w associated with it. This structure is described in this table:

Bit #	Name	Description
0	General push command	Indicates that prior to each host command given by the application, the software must assert a special host command (push command) CECDR = 0x00800000. CECR = 0x01c1000f.
1	UCC ATM RX INIT push command	Indicates that after issuing ATM RX INIT command, the host must issue another special command (push command) and immediately following that re-issue the ATM RX INIT command. (This makes the sequence of initializing the ATM receiver a sequence of three host commands) CECDR = 0x00800000. CECR = 0x01c1000f.
2	Add/remove command validation	Indicates that following the specific host command: "Add/Remove entry in Hash Lookup Table" used in Interworking setup, the user must issue another command. CECDR = 0xce000003. CECR = 0x01c10f58.
3	General push command	Indicates that the s/w has to initialize some pointers in the Ethernet thread pages which are used when Header Compression is activated. The full details of these pointers is located in the software drivers.
4	General push command	Indicates that after issuing Ethernet TX INIT command, user must issue this command for each SNUM of Ethernet TX thread. CECDR = 0x00800003. CECR = 0x7'b{0}, 8'b{Enet TX thread SNUM}, 1'b{1}, 12'b{0}, 4'b{1}
5 - 31	N/A	Reserved, set to zero.

20.6 V - Firmware Structure Layout

QE microcode from Freescale is typically provided as a header file. This header file contains macros that define the microcode binary itself as well as some other data used in uploading that microcode. The format of these files do not lend themselves to simple inclusion into other code. Hence, the need for a more portable format. This section defines that format.

Instead of distributing a header file, the microcode and related data are embedded into a binary blob. This blob is passed to the qe_upload_firmware() function, which parses the blob and performs everything necessary to upload the microcode.

All integers are big-endian. See the comments for function qe_upload_firmware() for up-to-date implementation information.

This structure supports versioning, where the version of the structure is embedded into the structure itself. To ensure forward and backwards compatibility, all versions of the structure must use the same 'qe_header' structure at the beginning.

'header' (type: struct qe_header): The 'length' field is the size, in bytes, of the entire structure, including all the microcode embedded in it, as well as the CRC (if present).

The 'magic' field is an array of three bytes that contains the letters 'Q', 'E', and 'F'. This is an identifier that indicates that this structure is a QE Firmware structure.

The 'version' field is a single byte that indicates the version of this structure. If the layout of the structure should ever need to be changed to add support for additional types of microcode, then the version number should also be changed.

The 'id' field is a null-terminated string(suitable for printing) that identifies the firmware.

The 'count' field indicates the number of 'microcode' structures. There must be one and only one 'microcode' structure for each RISC processor. Therefore, this field also represents the number of RISC processors for this SOC.

The 'soc' structure contains the SOC numbers and revisions used to match the microcode to the SOC itself. Normally, the microcode loader should check the data in this structure with the SOC number and revisions, and only upload the microcode if there's a match. However, this check is not made on all platforms.

Although it is not recommended, you can specify '0' in the soc.model field to skip matching SOCs altogether.

The 'model' field is a 16-bit number that matches the actual SOC. The 'major' and 'minor' fields are the major and minor revision numbers, respectively, of the SOC.

For example, to match the 8323, revision 1.0:

```
soc.model = 8323
soc.major = 1
soc.minor = 0
```

‘padding’ is necessary for structure alignment. This field ensures that the ‘extended_modes’ field is aligned on a 64-bit boundary.

‘extended_modes’is a bitfield that defines special functionality which has an impact on the device drivers. Each bit has its own impact and has special instructions for the driver associated with it. This field is stored in the QE library and available to any driver that calles qe_get_firmware_info().

‘vtraps’ is an array of 8 words that contain virtual trap values for each virtual traps. As with ‘extended_modes’, this field is stored in the QE library and available to any driver that calles qe_get_firmware_info().

‘**microcode**’ **(type: struct qe_microcode):** For each RISC processor there is one ‘microcode’ structure. The first ‘microcode’ structure is for the first RISC, and so on.

The ‘id’ field is a null-terminated string suitable for printing that identifies this particular microcode.

‘traps’ is an array of 16 words that contain hardware trap values for each of the 16 traps. If trap[i] is 0, then this particular trap is to be ignored (i.e. not written to TIBCR[i]). The entire value is written as-is to the TIBCR[i] register, so be sure to set the EN and T_IBP bits if necessary.

‘eccr’ is the value to program into the ECCR register.

‘iram_offset’ is the offset into IRAM to start writing the microcode.

‘count’ is the number of 32-bit words in the microcode.

‘code_offset’ is the offset, in bytes, from the beginning of this structure where the microcode itself can be found. The first microcode binary should be located immediately after the ‘microcode’ array.

‘major’, ‘minor’, and ‘revision’ are the major, minor, and revision version numbers, respectively, of the microcode. If all values are 0, then these fields are ignored.

‘reserved’ is necessary for structure alignment. Since ‘microcode’ is an array, the 64-bit ‘extended_modes’ field needs to be aligned on a 64-bit boundary, and this can only happen if the size of ‘microcode’ is a multiple of 8 bytes. To ensure that, we add ‘reserved’.

After the last microcode is a 32-bit CRC. It can be calculated using this algorithm:

```
u32 crc32(const u8 *p, unsigned int len)
{
      unsigned int i;
      u32 crc = 0;

      while (len--) {
         crc ^= *p++;
         for (i = 0; i < 8; i++)
                  crc = (crc >> 1) ^ ((crc & 1) ? 0xedb88320 : 0);
      }
      return crc;
}
```

20.7 VI - Sample Code for Creating Firmware Files

A Python program that creates firmware binaries from the header files normally distributed by Freescale can be found on http://opensource.freescale.com.

CHAPTER

TWENTYONE

POWER ARCHITECTURE 64-BIT LINUX SYSTEM CALL ABI

21.1 syscall

syscall calling sequence[1] matches the Power Architecture 64-bit ELF ABI specification C function calling sequence, including register preservation rules, with the following differences.

21.1.1 Parameters and return value

The system call number is specified in r0.

There is a maximum of 6 integer parameters to a syscall, passed in r3-r8.

Both a return value and a return error code are returned. cr0.SO is the return error code, and r3 is the return value or error code. When cr0.SO is clear, the syscall succeeded and r3 is the return value. When cr0.SO is set, the syscall failed and r3 is the error code that generally corresponds to errno.

21.1.2 Stack

System calls do not modify the caller's stack frame. For example, the caller's stack frame LR and CR save fields are not used.

21.1.3 Register preservation rules

Register preservation rules match the ELF ABI calling sequence with the following differences:

r0	Volatile	(System call number.)
r3	Volatile	(Parameter 1, and return value.)
r4-r8	Volatile	(Parameters 2-6.)
cr0	Volatile	(cr0.SO is the return error condition)
cr1, cr5-7	Nonvolatile	
lr	Nonvolatile	

[1] Some syscalls (typically low-level management functions) may have different calling sequences (e.g., rt_sigreturn).

All floating point and vector data registers as well as control and status registers are nonvolatile.

21.1.4 Invocation

The syscall is performed with the sc instruction, and returns with execution continuing at the instruction following the sc instruction.

21.1.5 Transactional Memory

Syscall behavior can change if the processor is in transactional or suspended transaction state, and the syscall can affect the behavior of the transaction.

If the processor is in suspended state when a syscall is made, the syscall will be performed as normal, and will return as normal. The syscall will be performed in suspended state, so its side effects will be persistent according to the usual transactional memory semantics. A syscall may or may not result in the transaction being doomed by hardware.

If the processor is in transactional state when a syscall is made, then the behavior depends on the presence of PPC_FEATURE2_HTM_NOSC in the AT_HWCAP2 ELF auxiliary vector.

- If present, which is the case for newer kernels, then the syscall will not be performed and the transaction will be doomed by the kernel with the failure code TM_CAUSE_SYSCALL | TM_CAUSE_PERSISTENT in the TEXASR SPR.
- If not present (older kernels), then the kernel will suspend the transactional state and the syscall will proceed as in the case of a suspended state syscall, and will resume the transactional state before returning to the caller. This case is not well defined or supported, so this behavior should not be relied upon.

21.2 vsyscall

vsyscall calling sequence matches the syscall calling sequence, with the following differences. Some vsyscalls may have different calling sequences.

21.2.1 Parameters and return value

r0 is not used as an input. The vsyscall is selected by its address.

21.2.2 Stack

The vsyscall may or may not use the caller’ s stack frame save areas.

21.2.3 Register preservation rules

r0	Volatile
cr1, cr5-7	Volatile
lr	Volatile

21.2.4 Invocation

The vsyscall is performed with a branch-with-link instruction to the vsyscall function address.

21.2.5 Transactional Memory

vsyscalls will run in the same transactional state as the caller. A vsyscall may or may not result in the transaction being doomed by hardware.

CHAPTER

TWENTYTWO

TRANSACTIONAL MEMORY SUPPORT

POWER kernel support for this feature is currently limited to supporting its use by user programs. It is not currently used by the kernel itself.

This file aims to sum up how it is supported by Linux and what behaviour you can expect from your user programs.

22.1 Basic overview

Hardware Transactional Memory is supported on POWER8 processors, and is a feature that enables a different form of atomic memory access. Several new instructions are presented to delimit transactions; transactions are guaranteed to either complete atomically or roll back and undo any partial changes.

A simple transaction looks like this:

```
begin_move_money:
  tbegin
  beq   abort_handler

  ld    r4, SAVINGS_ACCT(r3)
  ld    r5, CURRENT_ACCT(r3)
  subi  r5, r5, 1
  addi  r4, r4, 1
  std   r4, SAVINGS_ACCT(r3)
  std   r5, CURRENT_ACCT(r3)

  tend

  b     continue

abort_handler:
  ... test for odd failures ...

  /* Retry the transaction if it failed because it conflicted with
   * someone else: */
  b     begin_move_money
```

The ‘tbegin’ instruction denotes the start point, and ‘tend’ the end point. Between these points the processor is in ‘Transactional’ state; any memory references will complete in one go if there are no conflicts with other transactional or non-transactional accesses within the system. In this example, the transaction completes as though it were normal straight-line code IF no other proces-

sor has touched SAVINGS_ACCT(r3) or CURRENT_ACCT(r3); an atomic move of money from the current account to the savings account has been performed. Even though the normal ld/std instructions are used (note no lwarx/stwcx), either both SAVINGS_ACCT(r3) and CURRENT_ACCT(r3) will be updated, or neither will be updated.

If, in the meantime, there is a conflict with the locations accessed by the transaction, the transaction will be aborted by the CPU. Register and memory state will roll back to that at the 'tbegin', and control will continue from 'tbegin+4'. The branch to abort_handler will be taken this second time; the abort handler can check the cause of the failure, and retry.

Checkpointed registers include all GPRs, FPRs, VRs/VSRs, LR, CCR/CR, CTR, FPCSR and a few other status/flag regs; see the ISA for details.

22.2 Causes of transaction aborts

- Conflicts with cache lines used by other processors
- Signals
- Context switches
- See the ISA for full documentation of everything that will abort transactions.

22.3 Syscalls

Syscalls made from within an active transaction will not be performed and the transaction will be doomed by the kernel with the failure code TM_CAUSE_SYSCALL | TM_CAUSE_PERSISTENT.

Syscalls made from within a suspended transaction are performed as normal and the transaction is not explicitly doomed by the kernel. However, what the kernel does to perform the syscall may result in the transaction being doomed by the hardware. The syscall is performed in suspended mode so any side effects will be persistent, independent of transaction success or failure. No guarantees are provided by the kernel about which syscalls will affect transaction success.

Care must be taken when relying on syscalls to abort during active transactions if the calls are made via a library. Libraries may cache values (which may give the appearance of success) or perform operations that cause transaction failure before entering the kernel (which may produce different failure codes). Examples are glibc's getpid() and lazy symbol resolution.

22.4 Signals

Delivery of signals (both sync and async) during transactions provides a second thread state (ucontext/mcontext) to represent the second transactional register state. Signal delivery 'treclaim's to capture both register states, so signals abort transactions. The usual ucontext_t passed to the signal handler represents the checkpointed/original register state; the signal appears to have arisen at 'tbegin+4'.

If the sighandler ucontext has uc_link set, a second ucontext has been delivered. For future compatibility the MSR.TS field should be checked to determine the transactional state – if so, the second ucontext in uc->uc_link represents the active transactional registers at the point of the signal.

For 64-bit processes, uc->uc_mcontext.regs->msr is a full 64-bit MSR and its TS field shows the transactional mode.

For 32-bit processes, the mcontext's MSR register is only 32 bits; the top 32 bits are stored in the MSR of the second ucontext, i.e. in uc->uc_link->uc_mcontext.regs->msr. The top word contains the transactional state TS.

However, basic signal handlers don't need to be aware of transactions and simply returning from the handler will deal with things correctly:

Transaction-aware signal handlers can read the transactional register state from the second ucontext. This will be necessary for crash handlers to determine, for example, the address of the instruction causing the SIGSEGV.

Example signal handler:

```
  void crash_handler(int sig, siginfo_t *si, void *uc)
  {
    ucontext_t *ucp = uc;
    ucontext_t *transactional_ucp = ucp->uc_link;

    if (ucp_link) {
      u64 msr = ucp->uc_mcontext.regs->msr;
      /* May have transactional ucontext! */
#ifndef __powerpc64__
      msr |= ((u64)transactional_ucp->uc_mcontext.regs->msr) << 32;
#endif
      if (MSR_TM_ACTIVE(msr)) {
         /* Yes, we crashed during a transaction.  Oops. */
 fprintf(stderr, "Transaction to be restarted at 0x%llx, but "
                          "crashy instruction was at 0x%llx\n",
                          ucp->uc_mcontext.regs->nip,
                          transactional_ucp->uc_mcontext.regs->nip);
      }
    }

    fix_the_problem(ucp->dar);
  }
```

When in an active transaction that takes a signal, we need to be careful with the stack. It's possible that the stack has moved back up after the tbegin. The obvious case here is when the tbegin is called inside a function that returns before a tend. In this case, the stack is part of the checkpointed transactional memory state. If

we write over this non transactionally or in suspend, we are in trouble because if we get a tm abort, the program counter and stack pointer will be back at the tbegin but our in memory stack won't be valid anymore.

To avoid this, when taking a signal in an active transaction, we need to use the stack pointer from the checkpointed state, rather than the speculated state. This ensures that the signal context (written tm suspended) will be written below the stack required for the rollback. The transaction is aborted because of the treclaim, so any memory written between the tbegin and the signal will be rolled back anyway.

For signals taken in non-TM or suspended mode, we use the normal/non-checkpointed stack pointer.

Any transaction initiated inside a sighandler and suspended on return from the sighandler to the kernel will get reclaimed and discarded.

22.5 Failure cause codes used by kernel

These are defined in <asm/reg.h>, and distinguish different reasons why the kernel aborted a transaction:

TM_CAUSE_RESCHED	Thread was rescheduled.
TM_CAUSE_TLBI	Software TLB invalid.
TM_CAUSE_FAC_UNAV	FP/VEC/VSX unavailable trap.
TM_CAUSE_SYSCALL	Syscall from active transaction.
TM_CAUSE_SIGNAL	Signal delivered.
TM_CAUSE_MISC	Currently unused.
TM_CAUSE_ALIGNMENT	Alignment fault.
TM_CAUSE_EMULATE	Emulation that touched memory.

These can be checked by the user program's abort handler as TEXASR[0:7]. If bit 7 is set, it indicates that the error is consider persistent. For example a TM_CAUSE_ALIGNMENT will be persistent while a TM_CAUSE_RESCHED will not.

22.6 GDB

GDB and ptrace are not currently TM-aware. If one stops during a transaction, it looks like the transaction has just started (the checkpointed state is presented). The transaction cannot then be continued and will take the failure handler route. Furthermore, the transactional 2nd register state will be inaccessible. GDB can currently be used on programs using TM, but not sensibly in parts within transactions.

22.7 POWER9

TM on POWER9 has issues with storing the complete register state. This is described in this commit:

```
commit 4bb3c7a0208fc13ca70598efd109901a7cd45ae7
Author: Paul Mackerras <paulus@ozlabs.org>
Date:   Wed Mar 21 21:32:01 2018 +1100
KVM: PPC: Book3S HV: Work around transactional memory bugs in POWER9
```

To account for this different POWER9 chips have TM enabled in different ways.

On POWER9N DD2.01 and below, TM is disabled. ie HWCAP2[PPC_FEATURE2_HTM] is not set.

On POWER9N DD2.1 TM is configured by firmware to always abort a transaction when tm suspend occurs. So tsuspend will cause a transaction to be aborted and rolled back. Kernel exceptions will also cause the transaction to be aborted and rolled back and the exception will not occur. If userspace constructs a sigcontext that enables TM suspend, the sigcontext will be rejected by the kernel. This mode is advertised to users with HWCAP2[PPC_FEATURE2_HTM_NO_SUSPEND] set. HWCAP2[PPC_FEATURE2_HTM] is not set in this mode.

On POWER9N DD2.2 and above, KVM and POWERVM emulate TM for guests (as described in commit 4bb3c7a0208f), hence TM is enabled for guests ie. HWCAP2[PPC_FEATURE2_HTM] is set for guest userspace. Guests that makes heavy use of TM suspend (tsuspend or kernel suspend) will result in traps into the hypervisor and hence will suffer a performance degradation. Host userspace has TM disabled ie. HWCAP2[PPC_FEATURE2_HTM] is not set. (although we make enable it at some point in the future if we bring the emulation into host userspace context switching).

POWER9C DD1.2 and above are only available with POWERVM and hence Linux only runs as a guest. On these systems TM is emulated like on POWER9N DD2.2.

Guest migration from POWER8 to POWER9 will work with POWER9N DD2.2 and POWER9C DD1.2. Since earlier POWER9 processors don't support TM emulation, migration from POWER8 to POWER9 is not supported there.

22.8 Kernel implementation

22.8.1 h/rfid mtmsrd quirk

As defined in the ISA, rfid has a quirk which is useful in early exception handling. When in a userspace transaction and we enter the kernel via some exception, MSR will end up as TM=0 and TS=01 (ie. TM off but TM suspended). Regularly the kernel will want change bits in the MSR and will perform an rfid to do this. In this case rfid can have SRR0 TM = 0 and TS = 00 (ie. TM off and non transaction) and the resulting MSR will retain TM = 0 and TS=01 from before (ie. stay in suspend). This is a quirk in the architecture as this would normally be a transition from TS=01 to TS=00 (ie. suspend -> non transactional) which is an illegal transition.

This quirk is described the architecture in the definition of rfid with these lines:

if (MSR 29:31 ¬ = 0b010 | SRR1 29:31 ¬ = 0b000) then MSR 29:31 <- SRR1 29:31

hrfid and mtmsrd have the same quirk.

The Linux kernel uses this quirk in it’ s early exception handling.

CHAPTER

TWENTYTHREE

PROTECTED EXECUTION FACILITY

Contents

23.1 Introduction

Protected Execution Facility (PEF) is an architectural change for POWER 9 that enables Secure Virtual Machines (SVMs). DD2.3 chips (PVR=0x004e1203) or greater will be PEF-capable. A new ISA release will include the PEF RFC02487 changes.

When enabled, PEF adds a new higher privileged mode, called Ultravisor mode, to POWER architecture. Along with the new mode there is new firmware called the Protected Execution Ultravisor (or Ultravisor for short). Ultravisor mode is the highest privileged mode in POWER architecture.

Privilege States
Problem
Supervisor
Hypervisor
Ultravisor

PEF protects SVMs from the hypervisor, privileged users, and other VMs in the system. SVMs are protected while at rest and can only be executed by an authorized machine. All virtual machines utilize hypervisor services. The Ultravisor filters calls between the SVMs and the hypervisor to assure that information does not accidentally leak. All hypercalls except H_RANDOM are reflected to the hypervisor. H_RANDOM is not reflected to prevent the hypervisor from influencing random values in the SVM.

To support this there is a refactoring of the ownership of resources in the CPU. Some of the resources which were previously hypervisor privileged are now ultravisor privileged.

23.1.1 Hardware

The hardware changes include the following:

- There is a new bit in the MSR that determines whether the current process is running in secure mode, MSR(S) bit 41. MSR(S)=1, process is in secure mode, MSR(s)=0 process is in normal mode.
- The MSR(S) bit can only be set by the Ultravisor.
- HRFID cannot be used to set the MSR(S) bit. If the hypervisor needs to return to a SVM it must use an ultracall. It can determine if the VM it is returning to is secure.
- There is a new Ultravisor privileged register, SMFCTRL, which has an enable/disable bit SMFCTRL(E).
- The privilege of a process is now determined by three MSR bits, MSR(S, HV, PR). In each of the tables below the modes are listed from least privilege to highest privilege. The higher privilege modes can access all the resources of the lower privilege modes.

  Secure Mode MSR Settings

S	HV	PR	Privilege
1	0	1	Problem
1	0	0	Privileged(OS)
1	1	0	Ultravisor
1	1	1	Reserved

  Normal Mode MSR Settings

S	HV	PR	Privilege
0	0	1	Problem
0	0	0	Privileged(OS)
0	1	0	Hypervisor
0	1	1	Problem (Host)

- Memory is partitioned into secure and normal memory. Only processes that are running in secure mode can access secure memory.

- The hardware does not allow anything that is not running secure to access secure memory. This means that the Hypervisor cannot access the memory of the SVM without using an ultracall (asking the Ultravisor). The Ultravisor will only allow the hypervisor to see the SVM memory encrypted.
- I/O systems are not allowed to directly address secure memory. This limits the SVMs to virtual I/O only.
- The architecture allows the SVM to share pages of memory with the hypervisor that are not protected with encryption. However, this sharing must be initiated by the SVM.
- When a process is running in secure mode all hypercalls (syscall lev=1) go to the Ultravisor.
- When a process is in secure mode all interrupts go to the Ultravisor.
- The following resources have become Ultravisor privileged and require an Ultravisor interface to manipulate:
  - Processor configurations registers (SCOMs).
  - Stop state information.
  - The debug registers CIABR, DAWR, and DAWRX when SMFCTRL(D) is set. If SMFCTRL(D) is not set they do not work in secure mode. When set, reading and writing requires an Ultravisor call, otherwise that will cause a Hypervisor Emulation Assistance interrupt.
  - PTCR and partition table entries (partition table is in secure memory). An attempt to write to PTCR will cause a Hypervisor Emulation Assitance interrupt.
  - LDBAR (LD Base Address Register) and IMC (In-Memory Collection) non-architected registers. An attempt to write to them will cause a Hypervisor Emulation Assistance interrupt.
  - Paging for an SVM, sharing of memory with Hypervisor for an SVM. (Including Virtual Processor Area (VPA) and virtual I/O).

23.1.2 Software/Microcode

The software changes include:

- SVMs are created from normal VM using (open source) tooling supplied by IBM.
- All SVMs start as normal VMs and utilize an ultracall, UV_ESM (Enter Secure Mode), to make the transition.
- When the UV_ESM ultracall is made the Ultravisor copies the VM into secure memory, decrypts the verification information, and checks the integrity of the SVM. If the integrity check passes the Ultravisor passes control in secure mode.

- The verification information includes the pass phrase for the encrypted disk associated with the SVM. This pass phrase is given to the SVM when requested.
- The Ultravisor is not involved in protecting the encrypted disk of the SVM while at rest.
- For external interrupts the Ultravisor saves the state of the SVM, and reflects the interrupt to the hypervisor for processing. For hypercalls, the Ultravisor inserts neutral state into all registers not needed for the hypercall then reflects the call to the hypervisor for processing. The H_RANDOM hypercall is performed by the Ultravisor and not reflected.
- For virtual I/O to work bounce buffering must be done.
- The Ultravisor uses AES (IAPM) for protection of SVM memory. IAPM is a mode of AES that provides integrity and secrecy concurrently.
- The movement of data between normal and secure pages is coordinated with the Ultravisor by a new HMM plug-in in the Hypervisor.

The Ultravisor offers new services to the hypervisor and SVMs. These are accessed through ultracalls.

23.1.3 Terminology

- Hypercalls: special system calls used to request services from Hypervisor.
- Normal memory: Memory that is accessible to Hypervisor.
- Normal page: Page backed by normal memory and available to Hypervisor.
- Shared page: A page backed by normal memory and available to both the Hypervisor/QEMU and the SVM (i.e page has mappings in SVM and Hypervisor/QEMU).
- Secure memory: Memory that is accessible only to Ultravisor and SVMs.
- Secure page: Page backed by secure memory and only available to Ultravisor and SVM.
- SVM: Secure Virtual Machine.
- Ultracalls: special system calls used to request services from Ultravisor.

23.2 Ultravisor calls API

This section describes Ultravisor calls (ultracalls) needed to support Secure Virtual Machines (SVM)s and Paravirtualized KVM. The ultracalls allow the SVMs and Hypervisor to request services from the Ultravisor such as accessing a register or memory region that can only be accessed when running in Ultravisor-privileged mode.

The specific service needed from an ultracall is specified in register R3 (the first parameter to the ultracall). Other parameters to the ultracall, if any, are specified in registers R4 through R12.

Return value of all ultracalls is in register R3. Other output values from the ultracall, if any, are returned in registers R4 through R12. The only exception to this register usage is the `UV_RETURN` ultracall described below.

Each ultracall returns specific error codes, applicable in the context of the ultracall. However, like with the PowerPC Architecture Platform Reference (PAPR), if no specific error code is defined for a particular situation, then the ultracall will fallback to an erroneous parameter-position based code. i.e U_PARAMETER, U_P2, U_P3 etc depending on the ultracall parameter that may have caused the error.

Some ultracalls involve transferring a page of data between Ultravisor and Hypervisor. Secure pages that are transferred from secure memory to normal memory may be encrypted using dynamically generated keys. When the secure pages are transferred back to secure memory, they may be decrypted using the same dynamically generated keys. Generation and management of these keys will be covered in a separate document.

For now this only covers ultracalls currently implemented and being used by Hypervisor and SVMs but others can be added here when it makes sense.

The full specification for all hypercalls/ultracalls will eventually be made available in the public/OpenPower version of the PAPR specification.

Note: If PEF is not enabled, the ultracalls will be redirected to the Hypervisor which must handle/fail the calls.

23.2.1 Ultracalls used by Hypervisor

This section describes the virtual memory management ultracalls used by the Hypervisor to manage SVMs.

UV_PAGE_OUT

Encrypt and move the contents of a page from secure memory to normal memory.

Syntax

```
uint64_t ultracall(const uint64_t UV_PAGE_OUT,
        uint16_t lpid,          /* LPAR ID */
        uint64_t dest_ra,       /* real address of destination page */
        uint64_t src_gpa,       /* source guest-physical-address */
        uint8_t  flags,         /* flags */
        uint64_t order)         /* page size order */
```

Return values

One of the following values:

- U_SUCCESS on success.
- U_PARAMETER if `lpid` is invalid.
- U_P2 if `dest_ra` is invalid.
- U_P3 if the `src_gpa` address is invalid.
- U_P4 if any bit in the `flags` is unrecognized
- U_P5 if the `order` parameter is unsupported.
- U_FUNCTION if functionality is not supported.
- U_BUSY if page cannot be currently paged-out.

Description

Encrypt the contents of a secure-page and make it available to Hypervisor in a normal page.

By default, the source page is unmapped from the SVM's partition-scoped page table. But the Hypervisor can provide a hint to the Ultravisor to retain the page mapping by setting the `UV_SNAPSHOT` flag in `flags` parameter.

If the source page is already a shared page the call returns U_SUCCESS, without doing anything.

Use cases

1. QEMU attempts to access an address belonging to the SVM but the page frame for that address is not mapped into QEMU's address space. In this case, the Hypervisor will allocate a page frame, map it into QEMU's address space and issue the `UV_PAGE_OUT` call to retrieve the encrypted contents of the page.
2. When Ultravisor runs low on secure memory and it needs to page-out an LRU page. In this case, Ultravisor will issue the `H_SVM_PAGE_OUT` hypercall to the Hypervisor. The Hypervisor will then allocate a normal page and issue the `UV_PAGE_OUT` ultracall and the Ultravisor will encrypt and move the contents of the secure page into the normal page.
3. When Hypervisor accesses SVM data, the Hypervisor requests the Ultravisor to transfer the corresponding page into a insecure page, which the Hypervisor can access. The data in the normal page will be encrypted though.

UV_PAGE_IN

Move the contents of a page from normal memory to secure memory.

Syntax

```
uint64_t ultracall(const uint64_t UV_PAGE_IN,
        uint16_t lpid,          /* the LPAR ID */
        uint64_t src_ra,        /* source real address of page */
        uint64_t dest_gpa,      /* destination guest physical address */
        uint64_t flags,         /* flags */
        uint64_t order)         /* page size order */
```

Return values

One of the following values:

- U_SUCCESS on success.
- U_BUSY if page cannot be currently paged-in.
- U_FUNCTION if functionality is not supported
- U_PARAMETER if `lpid` is invalid.
- U_P2 if `src_ra` is invalid.
- U_P3 if the `dest_gpa` address is invalid.
- U_P4 if any bit in the `flags` is unrecognized
- U_P5 if the `order` parameter is unsupported.

Description

Move the contents of the page identified by `src_ra` from normal memory to secure memory and map it to the guest physical address `dest_gpa`.

If dest_gpa refers to a shared address, map the page into the partition-scoped page-table of the SVM. If dest_gpa is not shared, copy the contents of the page into the corresponding secure page. Depending on the context, decrypt the page before being copied.

The caller provides the attributes of the page through the `flags` parameter. Valid values for `flags` are:

- CACHE_INHIBITED
- CACHE_ENABLED
- WRITE_PROTECTION

The Hypervisor must pin the page in memory before making `UV_PAGE_IN` ultracall.

Use cases

1. When a normal VM switches to secure mode, all its pages residing in normal memory, are moved into secure memory.
2. When an SVM requests to share a page with Hypervisor the Hypervisor allocates a page and informs the Ultravisor.
3. When an SVM accesses a secure page that has been paged-out, Ultravisor invokes the Hypervisor to locate the page. After locating the page, the Hypervisor uses UV_PAGE_IN to make the page available to Ultravisor.

UV_PAGE_INVAL

Invalidate the Ultravisor mapping of a page.

Syntax

```
uint64_t ultracall(const uint64_t UV_PAGE_INVAL,
        uint16_t lpid,          /* the LPAR ID */
        uint64_t guest_pa,      /* destination guest-physical-address */
        uint64_t order)         /* page size order */
```

Return values

One of the following values:

- U_SUCCESS on success.
- U_PARAMETER if `lpid` is invalid.
- **U_P2 if guest_pa is invalid (or corresponds to a secure** page mapping).
- U_P3 if the `order` is invalid.
- U_FUNCTION if functionality is not supported.
- U_BUSY if page cannot be currently invalidated.

Description

This ultracall informs Ultravisor that the page mapping in Hypervisor corresponding to the given guest physical address has been invalidated and that the Ultravisor should not access the page. If the specified `guest_pa` corresponds to a secure page, Ultravisor will ignore the attempt to invalidate the page and return U_P2.

Use cases

1. When a shared page is unmapped from the QEMU' s page table, possibly because it is paged-out to disk, Ultravisor needs to know that the page should not be accessed from its side too.

UV_WRITE_PATE

Validate and write the partition table entry (PATE) for a given partition.

Syntax

```
uint64_t ultracall(const uint64_t UV_WRITE_PATE,
        uint32_t lpid,          /* the LPAR ID */
        uint64_t dw0            /* the first double word to write */
        uint64_t dw1)           /* the second double word to write */
```

Return values

One of the following values:

- U_SUCCESS on success.
- U_BUSY if PATE cannot be currently written to.
- U_FUNCTION if functionality is not supported.
- U_PARAMETER if `lpid` is invalid.
- U_P2 if `dw0` is invalid.
- U_P3 if the `dw1` address is invalid.
- **U_PERMISSION if the Hypervisor is attempting to change the PATE** of a secure virtual machine or if called from a context other than Hypervisor.

Description

Validate and write a LPID and its partition-table-entry for the given LPID. If the LPID is already allocated and initialized, this call results in changing the partition table entry.

Use cases

1. The Partition table resides in Secure memory and its entries, called PATE (Partition Table Entries), point to the partition- scoped page tables for the Hypervisor as well as each of the virtual machines (both secure and normal). The Hypervisor operates in partition 0 and its partition-scoped page tables reside in normal memory.
2. This ultracall allows the Hypervisor to register the partition- scoped and process-scoped page table entries for the Hypervisor and other partitions (virtual machines) with the Ultravisor.
3. If the value of the PATE for an existing partition (VM) changes, the TLB cache for the partition is flushed.
4. The Hypervisor is responsible for allocating LPID. The LPID and its PATE entry are registered together. The Hypervisor manages the PATE entries for a normal VM and can change the PATE entry anytime. Ultravisor manages the PATE entries for an SVM and Hypervisor is not allowed to modify them.

UV_RETURN

Return control from the Hypervisor back to the Ultravisor after processing an hypercall or interrupt that was forwarded (aka reflected) to the Hypervisor.

Syntax

```
uint64_t ultracall(const uint64_t UV_RETURN)
```

Return values

This call never returns to Hypervisor on success. It returns U_INVALID if ultracall is not made from a Hypervisor context.

Description

When an SVM makes an hypercall or incurs some other exception, the Ultravisor usually forwards (aka reflects) the exceptions to the Hypervisor. After processing the exception, Hypervisor uses the `UV_RETURN` ultracall to return control back to the SVM.

The expected register state on entry to this ultracall is:

- Non-volatile registers are restored to their original values.
- If returning from an hypercall, register R0 contains the return value (**unlike other ultracalls**) and, registers R4 through R12 contain any output values of the hypercall.
- R3 contains the ultracall number, i.e UV_RETURN.
- If returning with a synthesized interrupt, R2 contains the synthesized interrupt number.

Use cases

1. Ultravisor relies on the Hypervisor to provide several services to the SVM such as processing hypercall and other exceptions. After processing the exception, Hypervisor uses UV_RETURN to return control back to the Ultravisor.
2. Hypervisor has to use this ultracall to return control to the SVM.

UV_REGISTER_MEM_SLOT

Register an SVM address-range with specified properties.

Syntax

```
uint64_t ultracall(const uint64_t UV_REGISTER_MEM_SLOT,
        uint64_t lpid,          /* LPAR ID of the SVM */
        uint64_t start_gpa,     /* start guest physical address */
        uint64_t size,          /* size of address range in bytes */
        uint64_t flags          /* reserved for future expansion */
        uint16_t slotid)        /* slot identifier */
```

Return values

One of the following values:

- U_SUCCESS on success.
- U_PARAMETER if `lpid` is invalid.
- U_P2 if `start_gpa` is invalid.
- U_P3 if `size` is invalid.
- U_P4 if any bit in the `flags` is unrecognized.
- U_P5 if the `slotid` parameter is unsupported.
- U_PERMISSION if called from context other than Hypervisor.
- U_FUNCTION if functionality is not supported.

Description

Register a memory range for an SVM. The memory range starts at the guest physical address `start_gpa` and is `size` bytes long.

Use cases

1. When a virtual machine goes secure, all the memory slots managed by the Hypervisor move into secure memory. The Hypervisor iterates through each of memory slots, and registers the slot with Ultravisor. Hypervisor may discard some slots such as those used for firmware (SLOF).
2. When new memory is hot-plugged, a new memory slot gets registered.

UV_UNREGISTER_MEM_SLOT

Unregister an SVM address-range that was previously registered using UV_REGISTER_MEM_SLOT.

Syntax

```
uint64_t ultracall(const uint64_t UV_UNREGISTER_MEM_SLOT,
        uint64_t lpid,          /* LPAR ID of the SVM */
        uint64_t slotid)        /* reservation slotid */
```

Return values

One of the following values:

- U_SUCCESS on success.
- U_FUNCTION if functionality is not supported.
- U_PARAMETER if `lpid` is invalid.
- U_P2 if `slotid` is invalid.
- U_PERMISSION if called from context other than Hypervisor.

Description

Release the memory slot identified by `slotid` and free any resources allocated towards the reservation.

Use cases

1. Memory hot-remove.

UV_SVM_TERMINATE

Terminate an SVM and release its resources.

Syntax

```
uint64_t ultracall(const uint64_t UV_SVM_TERMINATE,
        uint64_t lpid,          /* LPAR ID of the SVM */)
```

Return values

One of the following values:

- U_SUCCESS on success.
- U_FUNCTION if functionality is not supported.
- U_PARAMETER if `lpid` is invalid.
- U_INVALID if VM is not secure.
- U_PERMISSION if not called from a Hypervisor context.

Description

Terminate an SVM and release all its resources.

Use cases

1. Called by Hypervisor when terminating an SVM.

23.2.2 Ultracalls used by SVM

UV_SHARE_PAGE

Share a set of guest physical pages with the Hypervisor.

Syntax

```
uint64_t ultracall(const uint64_t UV_SHARE_PAGE,
        uint64_t gfn,   /* guest page frame number */
        uint64_t num)   /* number of pages of size PAGE_SIZE */
```

Return values

One of the following values:

- U_SUCCESS on success.
- U_FUNCTION if functionality is not supported.
- U_INVALID if the VM is not secure.
- U_PARAMETER if `gfn` is invalid.
- U_P2 if `num` is invalid.

Description

Share the num pages starting at guest physical frame number gfn with the Hypervisor. Assume page size is PAGE_SIZE bytes. Zero the pages before returning.

If the address is already backed by a secure page, unmap the page and back it with an insecure page, with the help of the Hypervisor. If it is not backed by any page yet, mark the PTE as insecure and back it with an insecure page when the address is accessed. If it is already backed by an insecure page, zero the page and return.

Use cases

1. The Hypervisor cannot access the SVM pages since they are backed by secure pages. Hence an SVM must explicitly request Ultravisor for pages it can share with Hypervisor.
2. Shared pages are needed to support virtio and Virtual Processor Area (VPA) in SVMs.

UV_UNSHARE_PAGE

Restore a shared SVM page to its initial state.

Syntax

```
uint64_t ultracall(const uint64_t UV_UNSHARE_PAGE,
        uint64_t gfn,   /* guest page frame number */
        uint73 num)     /* number of pages of size PAGE_SIZE*/
```

Return values

One of the following values:

- U_SUCCESS on success.
- U_FUNCTION if functionality is not supported.
- U_INVALID if VM is not secure.
- U_PARAMETER if gfn is invalid.
- U_P2 if num is invalid.

Description

Stop sharing `num` pages starting at `gfn` with the Hypervisor. Assume that the page size is PAGE_SIZE. Zero the pages before returning.

If the address is already backed by an insecure page, unmap the page and back it with a secure page. Inform the Hypervisor to release reference to its shared page. If the address is not backed by a page yet, mark the PTE as secure and back it with a secure page when that address is accessed. If it is already backed by an secure page zero the page and return.

Use cases

1. The SVM may decide to unshare a page from the Hypervisor.

UV_UNSHARE_ALL_PAGES

Unshare all pages the SVM has shared with Hypervisor.

Syntax

```
uint64_t ultracall(const uint64_t UV_UNSHARE_ALL_PAGES)
```

Return values

One of the following values:

- U_SUCCESS on success.
- U_FUNCTION if functionality is not supported.
- U_INVAL if VM is not secure.

Description

Unshare all shared pages from the Hypervisor. All unshared pages are zeroed on return. Only pages explicitly shared by the SVM with the Hypervisor (using UV_SHARE_PAGE ultracall) are unshared. Ultravisor may internally share some pages with the Hypervisor without explicit request from the SVM. These pages will not be unshared by this ultracall.

Use cases

1. This call is needed when `kexec` is used to boot a different kernel. It may also be needed during SVM reset.

UV_ESM

Secure the virtual machine (enter secure mode).

Syntax

```
uint64_t ultracall(const uint64_t UV_ESM,
        uint64_t esm_blob_addr, /* location of the ESM blob */
        unint64_t fdt)          /* Flattened device tree */
```

Return values

One of the following values:

- U_SUCCESS on success (including if VM is already secure).
- U_FUNCTION if functionality is not supported.
- U_INVALID if VM is not secure.
- U_PARAMETER if `esm_blob_addr` is invalid.
- U_P2 if `fdt` is invalid.
- U_PERMISSION if any integrity checks fail.
- U_RETRY insufficient memory to create SVM.
- U_NO_KEY symmetric key unavailable.

Description

Secure the virtual machine. On successful completion, return control to the virtual machine at the address specified in the ESM blob.

Use cases

1. A normal virtual machine can choose to switch to a secure mode.

23.3 Hypervisor Calls API

This document describes the Hypervisor calls (hypercalls) that are needed to support the Ultravisor. Hypercalls are services provided by the Hypervisor to virtual machines and Ultravisor.

Register usage for these hypercalls is identical to that of the other hypercalls defined in the Power Architecture Platform Reference (PAPR) document. i.e on input, register R3 identifies the specific service that is being requested and registers R4 through R11 contain additional parameters to the hypercall, if any. On output, register R3 contains the return value and registers R4 through R9 contain any other output values from the hypercall.

This document only covers hypercalls currently implemented/planned for Ultravisor usage but others can be added here when it makes sense.

The full specification for all hypercalls/ultracalls will eventually be made available in the public/OpenPower version of the PAPR specification.

23.3.1 Hypervisor calls to support Ultravisor

Following are the set of hypercalls needed to support Ultravisor.

H_SVM_INIT_START

Begin the process of converting a normal virtual machine into an SVM.

Syntax

```
uint64_t hypercall(const uint64_t H_SVM_INIT_START)
```

Return values

One of the following values:

- H_SUCCESS on success.

Description

Initiate the process of securing a virtual machine. This involves coordinating with the Ultravisor, using ultracalls, to allocate resources in the Ultravisor for the new SVM, transferring the VM’ s pages from normal to secure memory etc. When the process is completed, Ultravisor issues the H_SVM_INIT_DONE hypercall.

Use cases

1. Ultravisor uses this hypercall to inform Hypervisor that a VM has initiated the process of switching to secure mode.

H_SVM_INIT_DONE

Complete the process of securing an SVM.

Syntax

```
uint64_t hypercall(const uint64_t H_SVM_INIT_DONE)
```

Return values

One of the following values:

- H_SUCCESS on success.
- **H_UNSUPPORTED if called from the wrong context (e.g.** from an SVM or before an H_SVM_INIT_START hypercall).

Description

Complete the process of securing a virtual machine. This call must be made after a prior call to `H_SVM_INIT_START` hypercall.

Use cases

On successfully securing a virtual machine, the Ultravisor informs Hypervisor about it. Hypervisor can use this call to finish setting up its internal state for this virtual machine.

H_SVM_INIT_ABORT

Abort the process of securing an SVM.

Syntax

```
uint64_t hypercall(const uint64_t H_SVM_INIT_ABORT)
```

Return values

One of the following values:

- **H_PARAMETER on successfully cleaning up the state,** Hypervisor will return this value to the **guest**, to indicate that the underlying UV_ESM ultracall failed.
- **H_STATE if called after a VM has gone secure (i.e** H_SVM_INIT_DONE hypercall was successful).
- **H_UNSUPPORTED if called from a wrong context (e.g. from a** normal VM).

Description

Abort the process of securing a virtual machine. This call must be made after a prior call to `H_SVM_INIT_START` hypercall and before a call to `H_SVM_INIT_DONE`.

On entry into this hypercall the non-volatile GPRs and FPRs are expected to contain the values they had at the time the VM issued the UV_ESM ultracall. Further `SRR0` is expected to contain the address of the instruction after the `UV_ESM` ultracall and `SRR1` the MSR value with which to return to the VM.

This hypercall will cleanup any partial state that was established for the VM since the prior `H_SVM_INIT_START` hypercall, including paging out pages that were paged-into secure memory, and issue the `UV_SVM_TERMINATE` ultracall to terminate the VM.

After the partial state is cleaned up, control returns to the VM (**not Ultravisor**), at the address specified in `SRR0` with the MSR values set to the value in `SRR1`.

Use cases

If after a successful call to `H_SVM_INIT_START`, the Ultravisor encounters an error while securing a virtual machine, either due to lack of resources or because the VM' s security information could not be validated, Ultravisor informs the Hypervisor about it. Hypervisor should use this call to clean up any internal state for this virtual machine and return to the VM.

H_SVM_PAGE_IN

Move the contents of a page from normal memory to secure memory.

Syntax

```
uint64_t hypercall(const uint64_t H_SVM_PAGE_IN,
        uint64_t guest_pa,      /* guest-physical-address */
        uint64_t flags,         /* flags */
        uint64_t order)         /* page size order */
```

Return values

One of the following values:

- H_SUCCESS on success.
- H_PARAMETER if `guest_pa` is invalid.
- H_P2 if `flags` is invalid.
- H_P3 if `order` of page is invalid.

Description

Retrieve the content of the page, belonging to the VM at the specified guest physical address.

Only valid value(s) in `flags` are:

- H_PAGE_IN_SHARED which indicates that the page is to be shared with the Ultravisor.
- H_PAGE_IN_NONSHARED indicates that the UV is not anymore interested in the page. Applicable if the page is a shared page.

The `order` parameter must correspond to the configured page size.

Use cases

1. When a normal VM becomes a secure VM (using the UV_ESM ultracall), the Ultravisor uses this hypercall to move contents of each page of the VM from normal memory to secure memory.
2. Ultravisor uses this hypercall to ask Hypervisor to provide a page in normal memory that can be shared between the SVM and Hypervisor.
3. Ultravisor uses this hypercall to page-in a paged-out page. This can happen when the SVM touches a paged-out page.
4. If SVM wants to disable sharing of pages with Hypervisor, it can inform Ultravisor to do so. Ultravisor will then use this hypercall and inform Hypervisor that it has released access to the normal page.

H_SVM_PAGE_OUT

Move the contents of the page to normal memory.

Syntax

```
uint64_t hypercall(const uint64_t H_SVM_PAGE_OUT,
        uint64_t guest_pa,      /* guest-physical-address */
        uint64_t flags,         /* flags (currently none) */
        uint64_t order)         /* page size order */
```

Return values

One of the following values:

- H_SUCCESS on success.
- H_PARAMETER if `guest_pa` is invalid.
- H_P2 if `flags` is invalid.
- H_P3 if `order` is invalid.

Description

Move the contents of the page identified by `guest_pa` to normal memory.

Currently `flags` is unused and must be set to 0. The `order` parameter must correspond to the configured page size.

Use cases

1. If Ultravisor is running low on secure pages, it can move the contents of some secure pages, into normal pages using this hypercall. The content will be encrypted.

23.4 References

- Supporting Protected Computing on IBM Power Architecture

CHAPTER

TWENTYFOUR

VIRTUAL ACCELERATOR SWITCHBOARD (VAS) USERSPACE API

24.1 Introduction

Power9 processor introduced Virtual Accelerator Switchboard (VAS) which allows both userspace and kernel communicate to co-processor (hardware accelerator) referred to as the Nest Accelerator (NX). The NX unit comprises of one or more hardware engines or co-processor types such as 842 compression, GZIP compression and encryption. On power9, userspace applications will have access to only GZIP Compression engine which supports ZLIB and GZIP compression algorithms in the hardware.

To communicate with NX, kernel has to establish a channel or window and then requests can be submitted directly without kernel involvement. Requests to the GZIP engine must be formatted as a co-processor Request Block (CRB) and these CRBs must be submitted to the NX using COPY/PASTE instructions to paste the CRB to hardware address that is associated with the engine's request queue.

The GZIP engine provides two priority levels of requests: Normal and High. Only Normal requests are supported from userspace right now.

This document explains userspace API that is used to interact with kernel to setup channel / window which can be used to send compression requests directly to NX accelerator.

24.2 Overview

Application access to the GZIP engine is provided through /dev/crypto/nx-gzip device node implemented by the VAS/NX device driver. An application must open the /dev/crypto/nx-gzip device to obtain a file descriptor (fd). Then should issue VAS_TX_WIN_OPEN ioctl with this fd to establish connection to the engine. It means send window is opened on GZIP engine for this process. Once a connection is established, the application should use the mmap() system call to map the hardware address of engine's request queue into the application's virtual address space.

The application can then submit one or more requests to the the engine by using copy/paste instructions and pasting the CRBs to the virtual address (aka

paste_address) returned by mmap(). User space can close the established connection or send window by closing the file descriptior (close(fd)) or upon the process exit.

Note that applications can send several requests with the same window or can establish multiple windows, but one window for each file descriptor.

Following sections provide additional details and references about the individual steps.

24.3 NX-GZIP Device Node

There is one /dev/crypto/nx-gzip node in the system and it provides access to all GZIP engines in the system. The only valid operations on /dev/crypto/nx-gzip are:

- open() the device for read and write.
- issue VAS_TX_WIN_OPEN ioctl
- mmap() the engine’s request queue into application’s virtual address space (i.e. get a paste_address for the co-processor engine).
- close the device node.

Other file operations on this device node are undefined.

Note that the copy and paste operations go directly to the hardware and do not go through this device. Refer COPY/PASTE document for more details.

Although a system may have several instances of the NX co-processor engines (typically, one per P9 chip) there is just one /dev/crypto/nx-gzip device node in the system. When the nx-gzip device node is opened, Kernel opens send window on a suitable instance of NX accelerator. It finds CPU on which the user process is executing and determine the NX instance for the corresponding chip on which this CPU belongs.

Applications may chose a specific instance of the NX co-processor using the vas_id field in the VAS_TX_WIN_OPEN ioctl as detailed below.

A userspace library libnxz is available here but still in development:
https://github.com/abalib/power-gzip

Applications that use inflate / deflate calls can link with libnxz instead of libz and use NX GZIP compression without any modification.

24.4 Open /dev/crypto/nx-gzip

The nx-gzip device should be opened for read and write. No special privileges are needed to open the device. Each window corresponds to one file descriptor. So if the userspace process needs multiple windows, several open calls have to be issued.

See open(2) system call man pages for other details such as return values, error codes and restrictions.

24.5 VAS_TX_WIN_OPEN ioctl

Applications should use the VAS_TX_WIN_OPEN ioctl as follows to establish a connection with NX co-processor engine:

::

struct vas_tx_win_open_attr { __u32 version; __s16 vas_id; /* specific instance of vas or -1

for default */

__u16 reserved1; __u64 flags; /* For future use */ __u64 reserved2[6];

};

version: The version field must be currently set to 1. vas_id: If '-1' is passed, kernel will make a best-effort attempt

to assign an optimal instance of NX for the process. To select the specific VAS instance, refer "Discovery of available VAS engines" section below.

flags, reserved1 and reserved2[6] fields are for future extension and must be set to 0.

The attributes attr for the VAS_TX_WIN_OPEN ioctl are defined as follows:

#define VAS_MAGIC 'v' #define VAS_TX_WIN_OPEN _IOW(VAS_MAGIC, 1,

struct vas_tx_win_open_attr)

struct vas_tx_win_open_attr attr; rc = ioctl(fd, VAS_TX_WIN_OPEN, &attr);

The VAS_TX_WIN_OPEN ioctl returns 0 on success. On errors, it returns -1 and sets the errno variable to indicate the error.

Error conditions: EINVAL fd does not refer to a valid VAS device. EINVAL Invalid vas ID EINVAL version is not set with proper value EEXIST Window is already opened for the given fd ENOMEM Memory is not available to allocate window ENOSPC System has too many active windows (connections)

opened

EINVAL reserved fields are not set to 0.

See the ioctl(2) man page for more details, error codes and restrictions.

24.6 mmap() NX-GZIP device

The mmap() system call for a NX-GZIP device fd returns a paste_address that the application can use to copy/paste its CRB to the hardware engines.

```
paste_addr = mmap(addr, size, prot, flags, fd, offset);
```

Only restrictions on mmap for a NX-GZIP device fd are:

- size should be PAGE_SIZE
- offset parameter should be 0ULL

Refer to mmap(2) man page for additional details/restrictions. In addition to the error conditions listed on the mmap(2) man page, can also fail with one of the following error codes:

EINVAL fd is not associated with an open window
(i.e mmap() does not follow a successful call to the VAS_TX_WIN_OPEN ioctl).

EINVAL offset field is not 0ULL.

24.7 Discovery of available VAS engines

Each available VAS instance in the system will have a device tree node like /proc/device-tree/vas@* or /proc/device-tree/xscom@*/vas@*. Determine the chip or VAS instance and use the corresponding ibm,vas-id property value in this node to select specific VAS instance.

24.8 Copy/Paste operations

Applications should use the copy and paste instructions to send CRB to NX. Refer section 4.4 in PowerISA for Copy/Paste instructions: https://openpowerfoundation.org/?resource_lib=power-isa-version-3-0

24.9 CRB Specification and use NX

Applications should format requests to the co-processor using the co-processor Request Block (CRBs). Refer NX-GZIP user's manual for the format of CRB and use NX from userspace such as sending requests and checking request status.

24.10 NX Fault handling

Applications send requests to NX and wait for the status by polling on co-processor Status Block (CSB) flags. NX updates status in CSB after each request is processed. Refer NX-GZIP user's manual for the format of CSB and status flags.

In case if NX encounters translation error (called NX page fault) on CSB address or any request buffer, raises an interrupt on the CPU to handle the fault. Page fault can happen if an application passes invalid addresses or request buffers are not in memory. The operating system handles the fault by updating CSB with the following data:

> csb.flags = CSB_V; csb.cc = CSB_CC_TRANSLATION; csb.ce = CSB_CE_TERMINATION; csb.address = fault_address;

When an application receives translation error, it can touch or access the page that has a fault address so that this page will be in memory. Then the application can resend this request to NX.

If the OS can not update CSB due to invalid CSB address, sends SEGV signal to the process who opened the send window on which the original request was issued. This signal returns with the following siginfo struct:

> siginfo.si_signo = SIGSEGV; siginfo.si_errno = EFAULT; siginfo.si_code = SEGV_MAPERR; siginfo.si_addr = CSB adress;

In the case of multi-thread applications, NX send windows can be shared across all threads. For example, a child thread can open a send window, but other threads can send requests to NX using this window. These requests will be successful even in the case of OS handling faults as long as CSB address is valid. If the NX request contains an invalid CSB address, the signal will be sent to the child thread that opened the window. But if the thread is exited without closing the window and the request is issued using this window. the signal will be issued to the thread group leader (tgid). It is up to the application whether to ignore or handle these signals.

NX-GZIP User's Manual: https://github.com/libnxz/power-gzip/blob/master/power_nx_gzip_um.pdf

24.11 Simple example

```
:: int use_nx_gzip() {
        int rc, fd; void *addr; struct vas_setup_attr txattr;
        fd = open( "/dev/crypto/nx-gzip" , O_RDWR); if (fd < 0) {
            fprintf(stderr, "open nx-gzip failedn" ); return -1;
        } memset(&txattr, 0, sizeof(txattr)); txattr.version = 1; tx-
        attr.vas_id = -1 rc = ioctl(fd, VAS_TX_WIN_OPEN,
            (unsigned long)&txattr);
        if (rc < 0) {
```

fprintf(stderr, "ioctl() n %d, error %dn", rc, errno);

return rc;

} addr = mmap(NULL, 4096, PROT_READ|PROT_WRITE,

MAP_SHARED, fd, 0ULL);

if (addr == MAP_FAILED) {

fprintf(stderr, "mmap() failed, errno %dn", errno);

return -errno;

} do {

//Format CRB request with compression or //uncompression // Refer tests for vas_copy/vas_paste vas_copy((&crb, 0, 1); vas_paste(addr, 0, 1); // Poll on csb.flags with timeout // csb address is listed in CRB

} while (true) close(fd) or window can be closed upon process exit

}

Refer https://github.com/abalib/power-gzip for tests or more use cases.