Linux Arch Documentation

The kernel development community

Jun 10, 2024

CONTENTS

These books provide programming details about architecture-specific implementation.

CHAPTER

ONE

ARC ARCHITECTURE

1.1 Linux kernel for ARC processors

1.1.1 Other sources of information

Below are some resources where more information can be found on ARC processors and relevant open source projects.

- https://embarc.org - Community portal for open source on ARC. Good place to start to find relevant FOSS projects, toolchain releases, news items and more.
- https://github.com/foss-for-synopsys-dwc-arc-processors - Home for all development activities regarding open source projects for ARC processors. Some of the projects are forks of various upstream projects, where "work in progress" is hosted prior to submission to upstream projects. Other projects are developed by Synopsys and made available to community as open source for use on ARC Processors.
- Official Synopsys ARC Processors website - location, with access to some IP documentation (Programmer's Reference Manual, AKA PRM for ARC HS processors) and free versions of some commercial tools (Free nSIM and MetaWare Light Edition). Please note though, registration is required to access both the documentation and the tools.

1.1.2 Important note on ARC processors configurability

ARC processors are highly configurable and several configurable options are supported in Linux. Some options are transparent to software (i.e cache geometries, some can be detected at runtime and configured and used accordingly, while some need to be explicitly selected or configured in the kernel's configuration utility (AKA "make menuconfig").

However not all configurable options are supported when an ARC processor is to run Linux. SoC design teams should refer to "Appendix E: Configuration for ARC Linux" in the ARC HS Databook for configurability guidelines.

Following these guidelines and selecting valid configuration options up front is critical to help prevent any unwanted issues during SoC bringup and software development in general.

1.1.3 Building the Linux kernel for ARC processors

The process of kernel building for ARC processors is the same as for any other architecture and could be done in 2 ways:

- Cross-compilation: process of compiling for ARC targets on a development host with a different processor architecture (generally x86_64/amd64).
- Native compilation: process of compiling for ARC on a ARC platform (hardware board or a simulator like QEMU) with complete development environment (GNU toolchain, dtc, make etc) installed on the platform.

In both cases, up-to-date GNU toolchain for ARC for the host is needed. Synopsys offers prebuilt toolchain releases which can be used for this purpose, available from:

- Synopsys GNU toolchain releases: https://github.com/foss-for-synopsys-dwc-arc-processors/toolchain/releases
- Linux kernel compilers collection: https://mirrors.edge.kernel.org/pub/tools/crosstool
- Bootlin's toolchain collection: https://toolchains.bootlin.com

Once the toolchain is installed in the system, make sure its "bin" folder is added in your `PATH` environment variable. Then set `ARCH=arc` & `CROSS_COMPILE=arc-linux` (or whatever matches installed ARC toolchain prefix) and then as usual `make defconfig && make`.

This will produce "vmlinux" file in the root of the kernel source tree usable for loading on the target system via JTAG. If you need to get an image usable with U-Boot bootloader, type `make uImage` and `uImage` will be produced in `arch/arc/boot` folder.

1.2 Feature status on arc architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	TODO
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok
core	thread-info-in-task	THREAD_INFO_IN_TASK	TODO
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	ok
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	TODO
debug	KASAN	HAVE_ARCH_KASAN	TODO
debug	kcov	ARCH_HAS_KCOV	TODO
debug	kgdb	HAVE_ARCH_KGDB	ok
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok
debug	kprobes	HAVE_KPROBES	ok
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	TODO
debug	kretprobes	HAVE_KRETPROBES	ok
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	TODO
debug	uprobes	ARCH_SUPPORTS_UPROBES	TODO
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO

Table 1 – continued fron

Subsystem	Feature	Kconfig	Status
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	TODO
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	TODO
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	TODO
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok
perf	perf-regs	HAVE_PERF_REGS	TODO
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	TODO
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	TODO
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	---
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	TODO
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	TODO
time	clockevents	!LEGACY_TIMER_TICK	ok
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	TODO
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	TODO
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	TODO
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	TODO
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	TODO
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO
vm	ioremap_prot	HAVE_IOREMAP_PROT	ok
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	ok

CHAPTER

TWO

ARM ARCHITECTURE

2.1 ARM Linux 2.6 and upper

Please check <ftp://ftp.arm.linux.org.uk/pub/armlinux> for updates.

2.1.1 Compilation of kernel

In order to compile ARM Linux, you will need a compiler capable of generating ARM ELF code with GNU extensions. GCC 3.3 is known to be a good compiler. Fortunately, you needn't guess. The kernel will report an error if your compiler is a recognized offender.

To build ARM Linux natively, you shouldn't have to alter the ARCH = line in the top level Makefile. However, if you don't have the ARM Linux ELF tools installed as default, then you should change the CROSS_COMPILE line as detailed below.

If you wish to cross-compile, then alter the following lines in the top level make file:

```
ARCH = <whatever>
```

with:

```
ARCH = arm
```

and:

```
CROSS_COMPILE=
```

to:

```
CROSS_COMPILE=<your-path-to-your-compiler-without-gcc>
```

eg.:

```
CROSS_COMPILE=arm-linux-
```

Do a 'make config', followed by 'make Image' to build the kernel (arch/arm/boot/Image). A compressed image can be built by doing a 'make zImage' instead of 'make Image'.

2.1.2 Bug reports etc

Please send patches to the patch system. For more information, see http://www.arm.linux.org.uk/developer/patches/info.php Always include some explanation as to what the patch does and why it is needed.

Bug reports should be sent to linux-arm-kernel@lists.arm.linux.org.uk, or submitted through the web form at http://www.arm.linux.org.uk/developer/

When sending bug reports, please ensure that they contain all relevant information, eg. the kernel messages that were printed before/during the problem, what you were doing, etc.

2.1.3 Include files

Several new include directories have been created under include/asm-arm, which are there to reduce the clutter in the top-level directory. These directories, and their purpose is listed below:

*arch-**	machine/platform specific header files
hardware	driver-internal ARM specific data structures/definitions
mach	descriptions of generic ARM to specific machine interfaces
*proc-**	processor dependent header files (currently only two categories)

2.1.4 Machine/Platform support

The ARM tree contains support for a lot of different machine types. To continue supporting these differences, it has become necessary to split machine-specific parts by directory. For this, the machine category is used to select which directories and files get included (we will use $(MACHINE) to refer to the category)

To this end, we now have arch/arm/mach-$(MACHINE) directories which are designed to house the non-driver files for a particular machine (eg, PCI, memory management, architecture definitions etc). For all future machines, there should be a corresponding arch/arm/mach-$(MACHINE)/include/mach directory.

2.1.5 Modules

Although modularisation is supported (and required for the FP emulator), each module on an ARM2/ARM250/ARM3 machine when is loaded will take memory up to the next 32k boundary due to the size of the pages. Therefore, is modularisation on these machines really worth it?

However, ARM6 and up machines allow modules to take multiples of 4k, and as such Acorn RiscPCs and other architectures using these processors can make good use of modularisation.

2.1.6 ADFS Image files

You can access image files on your ADFS partitions by mounting the ADFS partition, and then using the loopback device driver. You must have losetup installed.

Please note that the PCEmulator DOS partitions have a partition table at the start, and as such, you will have to give '-o offset' to losetup.

2.1.7 Request to developers

When writing device drivers which include a separate assembler file, please include it in with the C file, and not the arch/arm/lib directory. This allows the driver to be compiled as a loadable module without requiring half the code to be compiled into the kernel image.

In general, try to avoid using assembler unless it is really necessary. It makes drivers far less easy to port to other hardware.

2.1.8 ST506 hard drives

The ST506 hard drive controllers seem to be working fine (if a little slowly). At the moment they will only work off the controllers on an A4x0's motherboard, but for it to work off a Podule just requires someone with a podule to add the addresses for the IRQ mask and the HDC base to the source.

As of 31/3/96 it works with two drives (you should get the ADFS **configure* harddrive set to 2). I've got an internal 20MB and a great big external 5.25" FH 64MB drive (who could ever want more :-)).

I've just got 240K/s off it (a dd with bs=128k); that's about half of what RiscOS gets; but it's a heck of a lot better than the 50K/s I was getting last week :-)

Known bug: Drive data errors can cause a hang; including cases where the controller has fixed the error using ECC. (Possibly ONLY in that case…hmm).

2.1.9 1772 Floppy

This also seems to work OK, but hasn't been stressed much lately. It hasn't got any code for disc change detection in there at the moment which could be a bit of a problem! Suggestions on the correct way to do this are welcome.

2.1.10 *CONFIG_MACH_* and *CONFIG_ARCH_*

A change was made in 2003 to the macro names for new machines. Historically, *CONFIG_ARCH_* was used for the bonafide architecture, e.g. SA1100, as well as implementations of the architecture, e.g. Assabet. It was decided to change the implementation macros to read *CONFIG_MACH_* for clarity. Moreover, a retroactive fixup has not been made because it would complicate patching.

Previous registrations may be found online.

> <http://www.arm.linux.org.uk/developer/machines/>

2.1.11 Kernel entry (head.S)

The initial entry into the kernel is via head.S, which uses machine independent code. The machine is selected by the value of 'r1' on entry, which must be kept unique.

Due to the large number of machines which the ARM port of Linux provides for, we have a method to manage this which ensures that we don't end up duplicating large amounts of code.

We group machine (or platform) support code into machine classes. A class typically based around one or more system on a chip devices, and acts as a natural container around the actual implementations. These classes are given directories - arch/arm/mach-<class> - which contain the source files and include/mach/ to support the machine class.

For example, the SA1100 class is based upon the SA1100 and SA1110 SoC devices, and contains the code to support the way the on-board and off- board devices are used, or the device is setup, and provides that machine specific "personality."

For platforms that support device tree (DT), the machine selection is controlled at runtime by passing the device tree blob to the kernel. At compile-time, support for the machine type must be selected. This allows for a single multiplatform kernel build to be used for several machine types.

For platforms that do not use device tree, this machine selection is controlled by the machine type ID, which acts both as a run-time and a compile-time code selection method. You can register a new machine via the web site at:

> <http://www.arm.linux.org.uk/developer/machines/>

Note: Please do not register a machine type for DT-only platforms. If your platform is DT-only, you do not need a registered machine type.

Russell King (15/03/2004)

2.2 Booting ARM Linux

Author: Russell King

Date : 18 May 2002

The following documentation is relevant to 2.4.18-rmk6 and beyond.

In order to boot ARM Linux, you require a boot loader, which is a small program that runs before the main kernel. The boot loader is expected to initialise various devices, and eventually call the Linux kernel, passing information to the kernel.

Essentially, the boot loader should provide (as a minimum) the following:

1. Setup and initialise the RAM.
2. Initialise one serial port.
3. Detect the machine type.
4. Setup the kernel tagged list.

5. Load initramfs.
6. Call the kernel image.

2.2.1 1. Setup and initialise RAM

Existing boot loaders:
MANDATORY

New boot loaders:
MANDATORY

The boot loader is expected to find and initialise all RAM that the kernel will use for volatile data storage in the system. It performs this in a machine dependent manner. (It may use internal algorithms to automatically locate and size all RAM, or it may use knowledge of the RAM in the machine, or any other method the boot loader designer sees fit.)

2.2.2 2. Initialise one serial port

Existing boot loaders:
OPTIONAL, RECOMMENDED

New boot loaders:
OPTIONAL, RECOMMENDED

The boot loader should initialise and enable one serial port on the target. This allows the kernel serial driver to automatically detect which serial port it should use for the kernel console (generally used for debugging purposes, or communication with the target.)

As an alternative, the boot loader can pass the relevant 'console=' option to the kernel via the tagged lists specifying the port, and serial format options as described in

Documentation/admin-guide/kernel-parameters.rst.

2.2.3 3. Detect the machine type

Existing boot loaders:
OPTIONAL

New boot loaders:
MANDATORY except for DT-only platforms

The boot loader should detect the machine type its running on by some method. Whether this is a hard coded value or some algorithm that looks at the connected hardware is beyond the scope of this document. The boot loader must ultimately be able to provide a MACH_TYPE_xxx value to the kernel. (see linux/arch/arm/tools/mach-types). This should be passed to the kernel in register r1.

For DT-only platforms, the machine type will be determined by device tree. set the machine type to all ones (~0). This is not strictly necessary, but assures that it will not match any existing types.

2.2.4 4. Setup boot data

Existing boot loaders:
OPTIONAL, HIGHLY RECOMMENDED

New boot loaders:
MANDATORY

The boot loader must provide either a tagged list or a dtb image for passing configuration data to the kernel. The physical address of the boot data is passed to the kernel in register r2.

2.2.5 4a. Setup the kernel tagged list

The boot loader must create and initialise the kernel tagged list. A valid tagged list starts with ATAG_CORE and ends with ATAG_NONE. The ATAG_CORE tag may or may not be empty. An empty ATAG_CORE tag has the size field set to '2' (0x00000002). The ATAG_NONE must set the size field to zero.

Any number of tags can be placed in the list. It is undefined whether a repeated tag appends to the information carried by the previous tag, or whether it replaces the information in its entirety; some tags behave as the former, others the latter.

The boot loader must pass at a minimum the size and location of the system memory, and root filesystem location. Therefore, the minimum tagged list should look:

```
              +-----------+
base ->       | ATAG_CORE |  |
              +-----------+  |
              | ATAG_MEM  |  | increasing address
              +-----------+  |
              | ATAG_NONE |  |
              +-----------+  v
```

The tagged list should be stored in system RAM.

The tagged list must be placed in a region of memory where neither the kernel decompressor nor initrd 'bootp' program will overwrite it. The recommended placement is in the first 16KiB of RAM.

2.2.6 4b. Setup the device tree

The boot loader must load a device tree image (dtb) into system ram at a 64bit aligned address and initialize it with the boot data. The dtb format is documented at https://www.devicetree.org/specifications/. The kernel will look for the dtb magic value of 0xd00dfeed at the dtb physical address to determine if a dtb has been passed instead of a tagged list.

The boot loader must pass at a minimum the size and location of the system memory, and the root filesystem location. The dtb must be placed in a region of memory where the kernel decompressor will not overwrite it, while remaining within the region which will be covered by the kernel's low-memory mapping.

A safe location is just above the 128MiB boundary from start of RAM.

2.2.7 5. Load initramfs.

Existing boot loaders:
OPTIONAL

New boot loaders:
OPTIONAL

If an initramfs is in use then, as with the dtb, it must be placed in a region of memory where the kernel decompressor will not overwrite it while also with the region which will be covered by the kernel's low-memory mapping.

A safe location is just above the device tree blob which itself will be loaded just above the 128MiB boundary from the start of RAM as recommended above.

2.2.8 6. Calling the kernel image

Existing boot loaders:
MANDATORY

New boot loaders:
MANDATORY

There are two options for calling the kernel zImage. If the zImage is stored in flash, and is linked correctly to be run from flash, then it is legal for the boot loader to call the zImage in flash directly.

The zImage may also be placed in system RAM and called there. The kernel should be placed in the first 128MiB of RAM. It is recommended that it is loaded above 32MiB in order to avoid the need to relocate prior to decompression, which will make the boot process slightly faster.

When booting a raw (non-zImage) kernel the constraints are tighter. In this case the kernel must be loaded at an offset into system equal to TEXT_OFFSET - PAGE_OFFSET.

In any case, the following conditions must be met:

- Quiesce all DMA capable devices so that memory does not get corrupted by bogus network packets or disk data. This will save you many hours of debug.
- CPU register settings
  - r0 = 0,
  - r1 = machine type number discovered in (3) above.
  - r2 = physical address of tagged list in system RAM, or physical address of device tree block (dtb) in system RAM
- CPU mode

  All forms of interrupts must be disabled (IRQs and FIQs)

  For CPUs which do not include the ARM virtualization extensions, the CPU must be in SVC mode. (A special exception exists for Angel)

  CPUs which include support for the virtualization extensions can be entered in HYP mode in order to enable the kernel to make full use of these extensions. This is the recommended boot method for such CPUs, unless the virtualisations are already in use by a pre-installed hypervisor.

  If the kernel is not entered in HYP mode for any reason, it must be entered in SVC mode.

- Caches, MMUs

  The MMU must be off.

  Instruction cache may be on or off.

  Data cache must be off.

  If the kernel is entered in HYP mode, the above requirements apply to the HYP mode configuration in addition to the ordinary PL1 (privileged kernel modes) configuration. In addition, all traps into the hypervisor must be disabled, and PL1 access must be granted for all peripherals and CPU resources for which this is architecturally possible. Except for entering in HYP mode, the system configuration should be such that a kernel which does not include support for the virtualization extensions can boot correctly without extra help.

- The boot loader is expected to call the kernel image by jumping directly to the first instruction of the kernel image.

  On CPUs supporting the ARM instruction set, the entry must be made in ARM state, even for a Thumb-2 kernel.

  On CPUs supporting only the Thumb instruction set such as Cortex-M class CPUs, the entry must be made in Thumb state.

2.3 Cluster-wide Power-up/power-down race avoidance algorithm

This file documents the algorithm which is used to coordinate CPU and cluster setup and teardown operations and to manage hardware coherency controls safely.

The section "Rationale" explains what the algorithm is for and why it is needed. "Basic model" explains general concepts using a simplified view of the system. The other sections explain the actual details of the algorithm in use.

2.3.1 Rationale

In a system containing multiple CPUs, it is desirable to have the ability to turn off individual CPUs when the system is idle, reducing power consumption and thermal dissipation.

In a system containing multiple clusters of CPUs, it is also desirable to have the ability to turn off entire clusters.

Turning entire clusters off and on is a risky business, because it involves performing potentially destructive operations affecting a group of independently running CPUs, while the OS continues to run. This means that we need some coordination in order to ensure that critical cluster-level operations are only performed when it is truly safe to do so.

Simple locking may not be sufficient to solve this problem, because mechanisms like Linux spinlocks may rely on coherency mechanisms which are not immediately enabled when a cluster powers up. Since enabling or disabling those mechanisms may itself be a non-atomic operation (such as writing some hardware registers and invalidating large caches), other methods of coordination are required in order to guarantee safe power-down and power-up at the cluster level.

The mechanism presented in this document describes a coherent memory based protocol for performing the needed coordination. It aims to be as lightweight as possible, while providing the required safety properties.

2.3.2 Basic model

Each cluster and CPU is assigned a state, as follows:

- DOWN
- COMING_UP
- UP
- GOING_DOWN

```
    +---------> UP ----------+
    |                        v

COMING_UP                GOING_DOWN

    ^                        |
    +--------- DOWN <--------+
```

DOWN:
: The CPU or cluster is not coherent, and is either powered off or suspended, or is ready to be powered off or suspended.

COMING_UP:
: The CPU or cluster has committed to moving to the UP state. It may be part way through the process of initialisation and enabling coherency.

UP:
: The CPU or cluster is active and coherent at the hardware level. A CPU in this state is not necessarily being used actively by the kernel.

GOING_DOWN:
: The CPU or cluster has committed to moving to the DOWN state. It may be part way through the process of teardown and coherency exit.

Each CPU has one of these states assigned to it at any point in time. The CPU states are described in the "CPU state" section, below.

Each cluster is also assigned a state, but it is necessary to split the state value into two parts (the "cluster" state and "inbound" state) and to introduce additional states in order to avoid races between different CPUs in the cluster simultaneously modifying the state. The cluster-level states are described in the "Cluster state" section.

To help distinguish the CPU states from cluster states in this discussion, the state names are given a *CPU_* prefix for the CPU states, and a *CLUSTER_* or *INBOUND_* prefix for the cluster states.

2.3.3 CPU state

In this algorithm, each individual core in a multi-core processor is referred to as a "CPU". CPUs are assumed to be single-threaded: therefore, a CPU can only be doing one thing at a single point in time.

This means that CPUs fit the basic model closely.

The algorithm defines the following states for each CPU in the system:

- CPU_DOWN
- CPU_COMING_UP
- CPU_UP
- CPU_GOING_DOWN

```
 cluster setup and
CPU setup complete          policy decision
      +-----------> CPU_UP ------------+
      |                                v

CPU_COMING_UP                   CPU_GOING_DOWN

      ^                                |
      +----------- CPU_DOWN <----------+
 policy decision           CPU teardown complete
or hardware event
```

The definitions of the four states correspond closely to the states of the basic model.

Transitions between states occur as follows.

A trigger event (spontaneous) means that the CPU can transition to the next state as a result of making local progress only, with no requirement for any external event to happen.

CPU_DOWN:
A CPU reaches the CPU_DOWN state when it is ready for power-down. On reaching this state, the CPU will typically power itself down or suspend itself, via a WFI instruction or a firmware call.

Next state:
CPU_COMING_UP

Conditions:
none

Trigger events:

a) an explicit hardware power-up operation, resulting from a policy decision on another CPU;

b) a hardware event, such as an interrupt.

CPU_COMING_UP:
A CPU cannot start participating in hardware coherency until the cluster is set up and coherent. If the cluster is not ready, then the CPU will wait in the CPU_COMING_UP state until the cluster has been set up.

Next state:
CPU_UP

Conditions:
The CPU's parent cluster must be in CLUSTER_UP.

Trigger events:
Transition of the parent cluster to CLUSTER_UP.

Refer to the "Cluster state" section for a description of the CLUSTER_UP state.

CPU_UP:
When a CPU reaches the CPU_UP state, it is safe for the CPU to start participating in local coherency.

This is done by jumping to the kernel's CPU resume code.

Note that the definition of this state is slightly different from the basic model definition: CPU_UP does not mean that the CPU is coherent yet, but it does mean that it is safe to resume the kernel. The kernel handles the rest of the resume procedure, so the remaining steps are not visible as part of the race avoidance algorithm.

The CPU remains in this state until an explicit policy decision is made to shut down or suspend the CPU.

Next state:
CPU_GOING_DOWN

Conditions:
none

Trigger events:
explicit policy decision

CPU_GOING_DOWN:
While in this state, the CPU exits coherency, including any operations required to achieve this (such as cleaning data caches).

Next state:
CPU_DOWN

Conditions:
local CPU teardown complete

Trigger events:
(spontaneous)

2.3.4 Cluster state

A cluster is a group of connected CPUs with some common resources. Because a cluster contains multiple CPUs, it can be doing multiple things at the same time. This has some implications. In particular, a CPU can start up while another CPU is tearing the cluster down.

In this discussion, the "outbound side" is the view of the cluster state as seen by a CPU tearing the cluster down. The "inbound side" is the view of the cluster state as seen by a CPU setting the CPU up.

In order to enable safe coordination in such situations, it is important that a CPU which is setting up the cluster can advertise its state independently of the CPU which is tearing down the cluster. For this reason, the cluster state is split into two parts:

"cluster" state: The global state of the cluster; or the state on the outbound side:

- CLUSTER_DOWN
- CLUSTER_UP
- CLUSTER_GOING_DOWN

"inbound" state: The state of the cluster on the inbound side.

- INBOUND_NOT_COMING_UP
- INBOUND_COMING_UP

The different pairings of these states results in six possible states for the cluster as a whole:

```
                          CLUSTER_UP
        +==========> INBOUND_NOT_COMING_UP -------------+
        #                                               |
                                                        |
   CLUSTER_UP     <----+                                |
INBOUND_COMING_UP      |                                v

        ^             CLUSTER_GOING_DOWN       CLUSTER_GOING_DOWN
        #              INBOUND_COMING_UP <=== INBOUND_NOT_COMING_UP

  CLUSTER_DOWN         |                                |
INBOUND_COMING_UP <----+                                |
                                                        |
        ^                                               |
        +===========     CLUSTER_DOWN      <------------+
                     INBOUND_NOT_COMING_UP
```

Transitions -----> can only be made by the outbound CPU, and only involve changes to the "cluster" state.

Transitions ===##> can only be made by the inbound CPU, and only involve changes to the "inbound" state, except where there is no further transition possible on the outbound side (i.e., the outbound CPU has put the cluster into the CLUSTER_DOWN state).

The race avoidance algorithm does not provide a way to determine which exact CPUs within the cluster play these roles. This must be decided in advance by some other means. Refer to the section "Last man and first man selection" for more explanation.

CLUSTER_DOWN/INBOUND_NOT_COMING_UP is the only state where the cluster can actually be powered down.

The parallelism of the inbound and outbound CPUs is observed by the existence of two different paths from CLUSTER_GOING_DOWN/ INBOUND_NOT_COMING_UP (corresponding to GOING_DOWN in the basic model) to CLUSTER_DOWN/INBOUND_COMING_UP (corresponding to COMING_UP in the basic model). The second path avoids cluster teardown completely.

CLUSTER_UP/INBOUND_COMING_UP is equivalent to UP in the basic model. The final transition to CLUSTER_UP/INBOUND_NOT_COMING_UP is trivial and merely resets the state machine ready for the next cycle.

Details of the allowable transitions follow.

The next state in each case is notated

> <cluster state>/<inbound state> (<transitioner>)

where the <transitioner> is the side on which the transition can occur; either the inbound or the outbound side.

CLUSTER_DOWN/INBOUND_NOT_COMING_UP:

Next state:
CLUSTER_DOWN/INBOUND_COMING_UP (inbound)

Conditions:
none

Trigger events:

a) an explicit hardware power-up operation, resulting from a policy decision on another CPU;

b) a hardware event, such as an interrupt.

CLUSTER_DOWN/INBOUND_COMING_UP:

In this state, an inbound CPU sets up the cluster, including enabling of hardware coherency at the cluster level and any other operations (such as cache invalidation) which are required in order to achieve this.

The purpose of this state is to do sufficient cluster-level setup to enable other CPUs in the cluster to enter coherency safely.

Next state:
CLUSTER_UP/INBOUND_COMING_UP (inbound)

Conditions:
cluster-level setup and hardware coherency complete

Trigger events:
(spontaneous)

CLUSTER_UP/INBOUND_COMING_UP:

Cluster-level setup is complete and hardware coherency is enabled for the cluster. Other CPUs in the cluster can safely enter coherency.

This is a transient state, leading immediately to CLUSTER_UP/INBOUND_NOT_COMING_UP. All other CPUs on the cluster should consider treat these two states as equivalent.

Next state:
CLUSTER_UP/INBOUND_NOT_COMING_UP (inbound)

Conditions:
none

Trigger events:
(spontaneous)

CLUSTER_UP/INBOUND_NOT_COMING_UP:

Cluster-level setup is complete and hardware coherency is enabled for the cluster. Other CPUs in the cluster can safely enter coherency.

The cluster will remain in this state until a policy decision is made to power the cluster down.

Next state:
CLUSTER_GOING_DOWN/INBOUND_NOT_COMING_UP (outbound)

Conditions:
none

Trigger events:
policy decision to power down the cluster

CLUSTER_GOING_DOWN/INBOUND_NOT_COMING_UP:

An outbound CPU is tearing the cluster down. The selected CPU must wait in this state until all CPUs in the cluster are in the CPU_DOWN state.

When all CPUs are in the CPU_DOWN state, the cluster can be torn down, for example by cleaning data caches and exiting cluster-level coherency.

To avoid wasteful unnecessary teardown operations, the outbound should check the inbound cluster state for asynchronous transitions to INBOUND_COMING_UP. Alternatively, individual CPUs can be checked for entry into CPU_COMING_UP or CPU_UP.

Next states:

CLUSTER_DOWN/INBOUND_NOT_COMING_UP (outbound)

Conditions:
cluster torn down and ready to power off

Trigger events:
(spontaneous)

CLUSTER_GOING_DOWN/INBOUND_COMING_UP (inbound)

Conditions:
none

Trigger events:

a) an explicit hardware power-up operation, resulting from a policy decision on another CPU;

b) a hardware event, such as an interrupt.

CLUSTER_GOING_DOWN/INBOUND_COMING_UP:

The cluster is (or was) being torn down, but another CPU has come online in the meantime and is trying to set up the cluster again.

If the outbound CPU observes this state, it has two choices:

a) back out of teardown, restoring the cluster to the CLUSTER_UP state;

b) finish tearing the cluster down and put the cluster in the CLUSTER_DOWN state; the inbound CPU will set up the cluster again from there.

Choice (a) permits the removal of some latency by avoiding unnecessary teardown and setup operations in situations where the cluster is not really going to be powered down.

Next states:

CLUSTER_UP/INBOUND_COMING_UP (outbound)

Conditions:
cluster-level setup and hardware coherency complete

Trigger events:
(spontaneous)

CLUSTER_DOWN/INBOUND_COMING_UP (outbound)

Conditions:
cluster torn down and ready to power off

Trigger events:
(spontaneous)

2.3.5 Last man and First man selection

The CPU which performs cluster tear-down operations on the outbound side is commonly referred to as the "last man".

The CPU which performs cluster setup on the inbound side is commonly referred to as the "first man".

The race avoidance algorithm documented above does not provide a mechanism to choose which CPUs should play these roles.

Last man:

When shutting down the cluster, all the CPUs involved are initially executing Linux and hence coherent. Therefore, ordinary spinlocks can be used to select a last man safely, before the CPUs become non-coherent.

First man:

Because CPUs may power up asynchronously in response to external wake-up events, a dynamic mechanism is needed to make sure that only one CPU attempts to play the first man role and do the cluster-level initialisation: any other CPUs must wait for this to complete before proceeding.

Cluster-level initialisation may involve actions such as configuring coherency controls in the bus fabric.

The current implementation in mcpm_head.S uses a separate mutual exclusion mechanism to do this arbitration. This mechanism is documented in detail in vlocks.txt.

2.3.6 Features and Limitations

Implementation:

> The current ARM-based implementation is split between arch/arm/common/mcpm_head.S (low-level inbound CPU operations) and arch/arm/common/mcpm_entry.c (everything else):
>
> __mcpm_cpu_going_down() signals the transition of a CPU to the CPU_GOING_DOWN state.
>
> __mcpm_cpu_down() signals the transition of a CPU to the CPU_DOWN state.
>
> A CPU transitions to CPU_COMING_UP and then to CPU_UP via the low-level power-up code in mcpm_head.S. This could involve CPU-specific setup code, but in the current implementation it does not.
>
> __mcpm_outbound_enter_critical() and __mcpm_outbound_leave_critical() handle transitions from CLUSTER_UP to CLUSTER_GOING_DOWN and from there to CLUSTER_DOWN or back to CLUSTER_UP (in the case of an aborted cluster power-down).
>
> These functions are more complex than the __mcpm_cpu_*() functions due to the extra inter-CPU coordination which is needed for safe transitions at the cluster level.
>
> A cluster transitions from CLUSTER_DOWN back to CLUSTER_UP via the low-level power-up code in mcpm_head.S. This typically involves platform-specific setup code, provided by the platform-specific power_up_setup function registered via mcpm_sync_init.

Deep topologies:

> As currently described and implemented, the algorithm does not support CPU topologies involving more than two levels (i.e., clusters of clusters are not supported). The algorithm could be extended by replicating the cluster-level states for the additional topological levels, and modifying the transition rules for the intermediate (non-outermost) cluster levels.

2.3.7 Colophon

Originally created and documented by Dave Martin for Linaro Limited, in collaboration with Nicolas Pitre and Achin Gupta.

2.4 Interface for registering and calling firmware-specific operations for ARM

Written by Tomasz Figa <t.figa@samsung.com>

Some boards are running with secure firmware running in TrustZone secure world, which changes the way some things have to be initialized. This makes a need to provide an interface for such platforms to specify available firmware operations and call them when needed.

Firmware operations can be specified by filling in a struct firmware_ops with appropriate callbacks and then registering it with register_firmware_ops() function:

```
void register_firmware_ops(const struct firmware_ops *ops)
```

The ops pointer must be non-NULL. More information about struct firmware_ops and its members can be found in arch/arm/include/asm/firmware.h header.

There is a default, empty set of operations provided, so there is no need to set anything if platform does not require firmware operations.

To call a firmware operation, a helper macro is provided:

```
#define call_firmware_op(op, ...)                               \
        ((firmware_ops->op) ? firmware_ops->op(__VA_ARGS__) : (-ENOSYS))
```

the macro checks if the operation is provided and calls it or otherwise returns -ENOSYS to signal that given operation is not available (for example, to allow fallback to legacy operation).

Example of registering firmware operations:

```
/* board file */

static int platformX_do_idle(void)
{
        /* tell platformX firmware to enter idle */
        return 0;
}

static int platformX_cpu_boot(int i)
{
        /* tell platformX firmware to boot CPU i */
        return 0;
}

static const struct firmware_ops platformX_firmware_ops = {
        .do_idle        = exynos_do_idle,
        .cpu_boot       = exynos_cpu_boot,
        /* other operations not available on platformX */
};

/* init_early callback of machine descriptor */
static void __init board_init_early(void)
{
        register_firmware_ops(&platformX_firmware_ops);
}
```

Example of using a firmware operation:

```
/* some platform code, e.g. SMP initialization */

__raw_writel(__pa_symbol(exynos4_secondary_startup),
        CPU1_BOOT_REG);
```

```
/* Call Exynos specific smc call */
if (call_firmware_op(cpu_boot, cpu) == -ENOSYS)
        cpu_boot_legacy(...); /* Try legacy way */

gic_raise_softirq(cpumask_of(cpu), 1);
```

2.5 Interrupts

2.5.2-rmk5:

This is the first kernel that contains a major shake up of some of the major architecture-specific subsystems.

Firstly, it contains some pretty major changes to the way we handle the MMU TLB. Each MMU TLB variant is now handled completely separately - we have TLB v3, TLB v4 (without write buffer), TLB v4 (with write buffer), and finally TLB v4 (with write buffer, with I TLB invalidate entry). There is more assembly code inside each of these functions, mainly to allow more flexible TLB handling for the future.

Secondly, the IRQ subsystem.

The 2.5 kernels will be having major changes to the way IRQs are handled. Unfortunately, this means that machine types that touch the irq_desc[] array (basically all machine types) will break, and this means every machine type that we currently have.

Lets take an example. On the Assabet with Neponset, we have:

```
          GPIO25                 IRR:2
SA1100 ------------> Neponset -----------> SA1111
                                 IIR:1
                              -----------> USAR
                                 IIR:0
                              -----------> SMC9196
```

The way stuff currently works, all SA1111 interrupts are mutually exclusive of each other - if you're processing one interrupt from the SA1111 and another comes in, you have to wait for that interrupt to finish processing before you can service the new interrupt. Eg, an IDE PIO-based interrupt on the SA1111 excludes all other SA1111 and SMC9196 interrupts until it has finished transferring its multi-sector data, which can be a long time. Note also that since we loop in the SA1111 IRQ handler, SA1111 IRQs can hold off SMC9196 IRQs indefinitely.

The new approach brings several new ideas…

We introduce the concept of a "parent" and a "child". For example, to the Neponset handler, the "parent" is GPIO25, and the "children"d are SA1111, SMC9196 and USAR.

We also bring the idea of an IRQ "chip" (mainly to reduce the size of the irqdesc array). This doesn't have to be a real "IC"; indeed the SA11x0 IRQs are handled by two separate "chip" structures, one for GPIO0-10, and another for all the rest. It is just a container for the various operations (maybe this'll change to a better name). This structure has the following operations:

```
struct irqchip {
        /*
         * Acknowledge the IRQ.
         * If this is a level-based IRQ, then it is expected to mask the IRQ
         * as well.
         */
        void (*ack)(unsigned int irq);
        /*
         * Mask the IRQ in hardware.
         */
        void (*mask)(unsigned int irq);
        /*
         * Unmask the IRQ in hardware.
         */
        void (*unmask)(unsigned int irq);
        /*
         * Re-run the IRQ
         */
        void (*rerun)(unsigned int irq);
        /*
         * Set the type of the IRQ.
         */
        int (*type)(unsigned int irq, unsigned int, type);
};
```

ack

- required. May be the same function as mask for IRQs handled by do_level_IRQ.

mask

- required.

unmask

- required.

rerun

- optional. Not required if you're using do_level_IRQ for all IRQs that use this 'irqchip'. Generally expected to re-trigger the hardware IRQ if possible. If not, may call the handler directly.

type

- optional. If you don't support changing the type of an IRQ, it should be null so people can detect if they are unable to set the IRQ type.

For each IRQ, we keep the following information:

- "disable" depth (number of disable_irq()s without enable_irq()s)
- flags indicating what we can do with this IRQ (valid, probe, noautounmask) as before
- status of the IRQ (probing, enable, etc)
- chip
- per-IRQ handler
- irqaction structure list

The handler can be one of the 3 standard handlers - “level”, “edge” and “simple”, or your own specific handler if you need to do something special.

The “level” handler is what we currently have - its pretty simple. “edge” knows about the brokenness of such IRQ implementations - that you need to leave the hardware IRQ enabled while processing it, and queueing further IRQ events should the IRQ happen again while processing. The “simple” handler is very basic, and does not perform any hardware manipulation, nor state tracking. This is useful for things like the SMC9196 and USAR above.

2.5.1 So, what’s changed?

1. Machine implementations must not write to the irqdesc array.
2. New functions to manipulate the irqdesc array. The first 4 are expected to be useful only to machine specific code. The last is recommended to only be used by machine specific code, but may be used in drivers if absolutely necessary.

   set_irq_chip(irq,chip)
   Set the mask/unmask methods for handling this IRQ

   set_irq_handler(irq,handler)
   Set the handler for this IRQ (level, edge, simple)

   set_irq_chained_handler(irq,handler)
   Set a “chained” handler for this IRQ - automatically enables this IRQ (eg, Neponset and SA1111 handlers).

   set_irq_flags(irq,flags)
   Set the valid/probe/noautoenable flags.

   set_irq_type(irq,type)
   Set active the IRQ edge(s)/level. This replaces the SA1111 INTPOL manipulation, and the set_GPIO_IRQ_edge() function. Type should be one of IRQ_TYPE_xxx defined in <linux/irq.h>

3. set_GPIO_IRQ_edge() is obsolete, and should be replaced by set_irq_type.
4. Direct access to SA1111 INTPOL is deprecated. Use set_irq_type instead.
5. A handler is expected to perform any necessary acknowledgement of the parent IRQ via the correct chip specific function. For instance, if the SA1111 is directly connected to a SA1110 GPIO, then you should acknowledge the SA1110 IRQ each time you re-read the SA1111 IRQ status.
6. For any child which doesn’t have its own IRQ enable/disable controls (eg, SMC9196), the handler must mask or acknowledge the parent IRQ while the child handler is called, and the child handler should be the “simple” handler (not “edge” nor “level”). After the handler completes, the parent IRQ should be unmasked, and the status of all children must be re-checked for pending events. (see the Neponset IRQ handler for details).
7. fixup_irq() is gone, as is *arch/arm/mach-*/include/mach/irq.h*

Please note that this will not solve all problems - some of them are hardware based. Mixing level-based and edge-based IRQs on the same parent signal (eg neponset) is one such area where a software based solution can’t provide the full answer to low IRQ latency.

2.6 Kernel mode NEON

2.6.1 TL;DR summary

- Use only NEON instructions, or VFP instructions that don't rely on support code
- Isolate your NEON code in a separate compilation unit, and compile it with '-march=armv7-a -mfpu=neon -mfloat-abi=softfp'
- Put kernel_neon_begin() and kernel_neon_end() calls around the calls into your NEON code
- Don't sleep in your NEON code, and be aware that it will be executed with preemption disabled

2.6.2 Introduction

It is possible to use NEON instructions (and in some cases, VFP instructions) in code that runs in kernel mode. However, for performance reasons, the NEON/VFP register file is not preserved and restored at every context switch or taken exception like the normal register file is, so some manual intervention is required. Furthermore, special care is required for code that may sleep [i.e., may call schedule()], as NEON or VFP instructions will be executed in a non-preemptible section for reasons outlined below.

2.6.3 Lazy preserve and restore

The NEON/VFP register file is managed using lazy preserve (on UP systems) and lazy restore (on both SMP and UP systems). This means that the register file is kept 'live', and is only preserved and restored when multiple tasks are contending for the NEON/VFP unit (or, in the SMP case, when a task migrates to another core). Lazy restore is implemented by disabling the NEON/VFP unit after every context switch, resulting in a trap when subsequently a NEON/VFP instruction is issued, allowing the kernel to step in and perform the restore if necessary.

Any use of the NEON/VFP unit in kernel mode should not interfere with this, so it is required to do an 'eager' preserve of the NEON/VFP register file, and enable the NEON/VFP unit explicitly so no exceptions are generated on first subsequent use. This is handled by the function kernel_neon_begin(), which should be called before any kernel mode NEON or VFP instructions are issued. Likewise, the NEON/VFP unit should be disabled again after use to make sure user mode will hit the lazy restore trap upon next use. This is handled by the function kernel_neon_end().

2.6.4 Interruptions in kernel mode

For reasons of performance and simplicity, it was decided that there shall be no preserve/restore mechanism for the kernel mode NEON/VFP register contents. This implies that interruptions of a kernel mode NEON section can only be allowed if they are guaranteed not to touch the NEON/VFP registers. For this reason, the following rules and restrictions apply in the kernel: * NEON/VFP code is not allowed in interrupt context; * NEON/VFP code is not allowed to sleep; * NEON/VFP code is executed with preemption disabled.

If latency is a concern, it is possible to put back to back calls to kernel_neon_end() and kernel_neon_begin() in places in your code where none of the NEON registers are live. (Additional calls to kernel_neon_begin() should be reasonably cheap if no context switch occurred in the meantime)

2.6.5 VFP and support code

Earlier versions of VFP (prior to version 3) rely on software support for things like IEEE-754 compliant underflow handling etc. When the VFP unit needs such software assistance, it signals the kernel by raising an undefined instruction exception. The kernel responds by inspecting the VFP control registers and the current instruction and arguments, and emulates the instruction in software.

Such software assistance is currently not implemented for VFP instructions executed in kernel mode. If such a condition is encountered, the kernel will fail and generate an OOPS.

2.6.6 Separating NEON code from ordinary code

The compiler is not aware of the special significance of kernel_neon_begin() and kernel_neon_end(), i.e., that it is only allowed to issue NEON/VFP instructions between calls to these respective functions. Furthermore, GCC may generate NEON instructions of its own at -O3 level if -mfpu=neon is selected, and even if the kernel is currently compiled at -O2, future changes may result in NEON/VFP instructions appearing in unexpected places if no special care is taken.

Therefore, the recommended and only supported way of using NEON/VFP in the kernel is by adhering to the following rules:

- isolate the NEON code in a separate compilation unit and compile it with '-march=armv7-a -mfpu=neon -mfloat-abi=softfp';
- issue the calls to kernel_neon_begin(), kernel_neon_end() as well as the calls into the unit containing the NEON code from a compilation unit which is *not* built with the GCC flag '-mfpu=neon' set.

As the kernel is compiled with '-msoft-float', the above will guarantee that both NEON and VFP instructions will only ever appear in designated compilation units at any optimization level.

2.6.7 NEON assembler

NEON assembler is supported with no additional caveats as long as the rules above are followed.

2.6.8 NEON code generated by GCC

The GCC option -ftree-vectorize (implied by -O3) tries to exploit implicit parallelism, and generates NEON code from ordinary C source code. This is fully supported as long as the rules above are followed.

2.6.9 NEON intrinsics

NEON intrinsics are also supported. However, as code using NEON intrinsics relies on the GCC header <arm_neon.h>, (which #includes <stdint.h>), you should observe the following in addition to the rules above:

- Compile the unit containing the NEON intrinsics with '-ffreestanding' so GCC uses its builtin version of <stdint.h> (this is a C99 header which the kernel does not supply);
- Include <arm_neon.h> last, or at least after <linux/types.h>

2.7 Kernel-provided User Helpers

These are segment of kernel provided user code reachable from user space at a fixed address in kernel memory. This is used to provide user space with some operations which require kernel help because of unimplemented native feature and/or instructions in many ARM CPUs. The idea is for this code to be executed directly in user mode for best efficiency but which is too intimate with the kernel counter part to be left to user libraries. In fact this code might even differ from one CPU to another depending on the available instruction set, or whether it is a SMP systems. In other words, the kernel reserves the right to change this code as needed without warning. Only the entry points and their results as documented here are guaranteed to be stable.

This is different from (but doesn't preclude) a full blown VDSO implementation, however a VDSO would prevent some assembly tricks with constants that allows for efficient branching to those code segments. And since those code segments only use a few cycles before returning to user code, the overhead of a VDSO indirect far call would add a measurable overhead to such minimalistic operations.

User space is expected to bypass those helpers and implement those things inline (either in the code emitted directly by the compiler, or part of the implementation of a library call) when optimizing for a recent enough processor that has the necessary native support, but only if resulting binaries are already to be incompatible with earlier ARM processors due to usage of similar native instructions for other things. In other words don't make binaries unable to run on earlier processors just for the sake of not using these kernel helpers if your compiled code is not going to use new instructions for other purpose.

New helpers may be added over time, so an older kernel may be missing some helpers present in a newer kernel. For this reason, programs must check the value of __kuser_helper_version (see below) before assuming that it is safe to call any particular helper. This check should ideally be performed only once at process startup time, and execution aborted early if the required helpers are not provided by the kernel version that process is running on.

2.7.1 kuser_helper_version

Location: 0xffff0ffc

Reference declaration:

```
extern int32_t __kuser_helper_version;
```

Definition:

This field contains the number of helpers being implemented by the running kernel. User space may read this to determine the availability of a particular helper.

Usage example:

```
#define __kuser_helper_version (*(int32_t *)0xffff0ffc)

void check_kuser_version(void)
{
      if (__kuser_helper_version < 2) {
              fprintf(stderr, "can't do atomic operations, kernel too old\n");
              abort();
      }
}
```

Notes:

User space may assume that the value of this field never changes during the lifetime of any single process. This means that this field can be read once during the initialisation of a library or startup phase of a program.

2.7.2 kuser_get_tls

Location: 0xffff0fe0

Reference prototype:

```
void * __kuser_get_tls(void);
```

Input:

lr = return address

Output:

r0 = TLS value

Clobbered registers:

none

Definition:

Get the TLS value as previously set via the __ARM_NR_set_tls syscall.

Usage example:

```
typedef void * (__kuser_get_tls_t)(void);
#define __kuser_get_tls (*(__kuser_get_tls_t *)0xffff0fe0)

void foo()
{
      void *tls = __kuser_get_tls();
      printf("TLS = %p\n", tls);
}
```

Notes:

- Valid only if __kuser_helper_version >= 1 (from kernel version 2.6.12).

2.7.3 kuser_cmpxchg

Location: 0xffff0fc0

Reference prototype:

```
int __kuser_cmpxchg(int32_t oldval, int32_t newval, volatile int32_t *ptr);
```

Input:

r0 = oldval r1 = newval r2 = ptr lr = return address

Output:

r0 = success code (zero or non-zero) C flag = set if r0 == 0, clear if r0 != 0

Clobbered registers:

r3, ip, flags

Definition:

Atomically store newval in **ptr* only if **ptr* is equal to oldval. Return zero if **ptr* was changed or non-zero if no exchange happened. The C flag is also set if **ptr* was changed to allow for assembly optimization in the calling code.

Usage example:

```
typedef int (__kuser_cmpxchg_t)(int oldval, int newval, volatile int *ptr);
#define __kuser_cmpxchg (*(__kuser_cmpxchg_t *)0xffff0fc0)

int atomic_add(volatile int *ptr, int val)
{
      int old, new;

      do {
              old = *ptr;
              new = old + val;
      } while(__kuser_cmpxchg(old, new, ptr));

      return new;
}
```

Notes:

- This routine already includes memory barriers as needed.
- Valid only if __kuser_helper_version >= 2 (from kernel version 2.6.12).

2.7.4 kuser_memory_barrier

Location: 0xffff0fa0

Reference prototype:

```
void __kuser_memory_barrier(void);
```

Input:

lr = return address

Output:

none

Clobbered registers:

none

Definition:

Apply any needed memory barrier to preserve consistency with data modified manually and __kuser_cmpxchg usage.

Usage example:

```
typedef void (__kuser_dmb_t)(void);
#define __kuser_dmb (*(__kuser_dmb_t *)0xffff0fa0)
```

Notes:

- Valid only if __kuser_helper_version >= 3 (from kernel version 2.6.15).

2.7.5 kuser_cmpxchg64

Location: 0xffff0f60

Reference prototype:

```
int __kuser_cmpxchg64(const int64_t *oldval,
                      const int64_t *newval,
                      volatile int64_t *ptr);
```

Input:

r0 = pointer to oldval r1 = pointer to newval r2 = pointer to target value lr = return address

Output:

r0 = success code (zero or non-zero) C flag = set if r0 == 0, clear if r0 != 0

Clobbered registers:

r3, lr, flags

Definition:

Atomically store the 64-bit value pointed by **newval* in **ptr* only if **ptr* is equal to the 64-bit value pointed by **oldval*. Return zero if **ptr* was changed or non-zero if no exchange happened.

The C flag is also set if **ptr* was changed to allow for assembly optimization in the calling code.

Usage example:

```
typedef int (__kuser_cmpxchg64_t)(const int64_t *oldval,
                                  const int64_t *newval,
                                  volatile int64_t *ptr);
#define __kuser_cmpxchg64 (*(__kuser_cmpxchg64_t *)0xffff0f60)

int64_t atomic_add64(volatile int64_t *ptr, int64_t val)
{
      int64_t old, new;

      do {
              old = *ptr;
              new = old + val;
      } while(__kuser_cmpxchg64(&old, &new, ptr));

      return new;
}
```

Notes:

- This routine already includes memory barriers as needed.
- Due to the length of this sequence, this spans 2 conventional kuser "slots", therefore 0xffff0f80 is not used as a valid entry point.
- Valid only if __kuser_helper_version >= 5 (from kernel version 3.1).

2.8 Kernel Memory Layout on ARM Linux

Russell King <rmk@arm.linux.org.uk>

November 17, 2005 (2.6.15)

This document describes the virtual memory layout which the Linux kernel uses for ARM processors. It indicates which regions are free for platforms to use, and which are used by generic code.

The ARM CPU is capable of addressing a maximum of 4GB virtual memory space, and this must be shared between user space processes, the kernel, and hardware devices.

As the ARM architecture matures, it becomes necessary to reserve certain regions of VM space for use for new facilities; therefore this document may reserve more VM space over time.

Start	End	Use
ffff8000	ffffffff	copy_user_page / clear_user_page use. For SA11xx and Xscale, this is used to setup a minicache mapping.
ffff4000	ffffffff	cache aliasing on ARMv6 and later CPUs.
ffff1000	ffff7fff	Reserved. Platforms must not use this address range.
ffff0000	ffff0fff	CPU vector page. The CPU vectors are mapped here if the CPU supports vector relocation (control register V bit.)
fffe0000	fffeffff	XScale cache flush area. This is used in proc-xscale.S to flush the whole data cache. (XScale does not have TCM.)
fffe8000	fffeffff	DTCM mapping area for platforms with DTCM mounted inside the CPU.
fffe0000	fffe7fff	ITCM mapping area for platforms with ITCM mounted inside the CPU.
ffc80000	ffefffff	Fixmap mapping region. Addresses provided by fix_to_virt() will be located here.
ffc00000	ffc7ffff	Guard region
ff800000	ffbfffff	Permanent, fixed read-only mapping of the firmware provided DT blob
fee00000	feffffff	Mapping of PCI I/O space. This is a static mapping within the vmalloc space.
VMALLOC_START	VMALLOC_END-1	vmalloc() / ioremap() space. Memory returned by vmalloc/ioremap will be dynamically placed in this region. Machine specific static mappings are also located here through iotable_init(). VMALLOC_START is based upon the value of the high_memory variable, and VMALLOC_END is equal to 0xff800000.
PAGE_OFFSET	high_memory-1	Kernel direct-mapped RAM region. This maps the platforms RAM, and typically maps all platform RAM in a 1:1 relationship.
PKMAP_BASE	PAGE_OFFSET-1	Permanent kernel mappings One way of mapping HIGHMEM pages into kernel space.
MODULES_VADDR	MODULES_END-1	Kernel module space Kernel modules inserted via insmod are placed here using dynamic mappings.
TASK_SIZE	MODULES_VADDR-1	KASAn shadow memory when KASan is in use. The range from MODULES_VADDR to the top of the memory is shadowed here with 1 bit per byte of memory.
00001000	TASK_SIZE-1	User space mappings Per-thread mappings are placed here via the mmap() system call.
00000000	00000fff	CPU vector page / null pointer trap CPUs which do not support vector remapping place their vector page here. NULL pointer dereferences by both the kernel and user space are also caught via this mapping.

Please note that mappings which collide with the above areas may result in a non-bootable kernel, or may cause the kernel to (eventually) panic at run time.

Since future CPUs may impact the kernel mapping layout, user programs must not access any memory which is not mapped inside their 0x0001000 to TASK_SIZE address range. If they wish to access these areas, they must set up their own mappings using open() and mmap().

2.9 Memory alignment

Too many problems popped up because of unnoticed misaligned memory access in kernel code lately. Therefore the alignment fixup is now unconditionally configured in for SA11x0 based targets. According to Alan Cox, this is a bad idea to configure it out, but Russell King has some good reasons for doing so on some f***ed up ARM architectures like the EBSA110. However this is not the case on many design I'm aware of, like all SA11x0 based ones.

Of course this is a bad idea to rely on the alignment trap to perform unaligned memory access in general. If those access are predictable, you are better to use the macros provided by include/asm/unaligned.h. The alignment trap can fixup misaligned access for the exception cases, but at a high performance cost. It better be rare.

Now for user space applications, it is possible to configure the alignment trap to SIGBUS any code performing unaligned access (good for debugging bad code), or even fixup the access by software like for kernel code. The later mode isn't recommended for performance reasons (just think about the floating point emulation that works about the same way). Fix your code instead!

Please note that randomly changing the behaviour without good thought is real bad - it changes the behaviour of all unaligned instructions in user space, and might cause programs to fail unexpectedly.

To change the alignment trap behavior, simply echo a number into /proc/cpu/alignment. The number is made up from various bits:

bit	behavior when set
0	A user process performing an unaligned memory access will cause the kernel to print a message indicating process name, pid, pc, instruction, address, and the fault code.
1	The kernel will attempt to fix up the user process performing the unaligned access. This is of course slow (think about the floating point emulator) and not recommended for production use.
2	The kernel will send a SIGBUS signal to the user process performing the unaligned access.

Note that not all combinations are supported - only values 0 through 5. (6 and 7 don't make sense).

For example, the following will turn on the warnings, but without fixing up or sending SIGBUS signals:

```
echo 1 > /proc/cpu/alignment
```

You can also read the content of the same file to get statistical information on unaligned access occurrences plus the current mode of operation for user space code.

Nicolas Pitre, Mar 13, 2001. Modified Russell King, Nov 30, 2001.

2.10 ARM TCM (Tightly-Coupled Memory) handling in Linux

Written by Linus Walleij <linus.walleij@stericsson.com>

Some ARM SoCs have a so-called TCM (Tightly-Coupled Memory). This is usually just a few (4-64) KiB of RAM inside the ARM processor.

Due to being embedded inside the CPU, the TCM has a Harvard-architecture, so there is an ITCM (instruction TCM) and a DTCM (data TCM). The DTCM can not contain any instructions, but the ITCM can actually contain data. The size of DTCM or ITCM is minimum 4KiB so the typical minimum configuration is 4KiB ITCM and 4KiB DTCM.

ARM CPUs have special registers to read out status, physical location and size of TCM memories. arch/arm/include/asm/cputype.h defines a CPUID_TCM register that you can read out from the system control coprocessor. Documentation from ARM can be found at http://infocenter.arm.com, search for "TCM Status Register" to see documents for all CPUs. Reading this register you can determine if ITCM (bits 1-0) and/or DTCM (bit 17-16) is present in the machine.

There is further a TCM region register (search for "TCM Region Registers" at the ARM site) that can report and modify the location size of TCM memories at runtime. This is used to read out and modify TCM location and size. Notice that this is not a MMU table: you actually move the physical location of the TCM around. At the place you put it, it will mask any underlying RAM from the CPU so it is usually wise not to overlap any physical RAM with the TCM.

The TCM memory can then be remapped to another address again using the MMU, but notice that the TCM is often used in situations where the MMU is turned off. To avoid confusion the current Linux implementation will map the TCM 1 to 1 from physical to virtual memory in the location specified by the kernel. Currently Linux will map ITCM to 0xfffe0000 and on, and DTCM to 0xfffe8000 and on, supporting a maximum of 32KiB of ITCM and 32KiB of DTCM.

Newer versions of the region registers also support dividing these TCMs in two separate banks, so for example an 8KiB ITCM is divided into two 4KiB banks with its own control registers. The idea is to be able to lock and hide one of the banks for use by the secure world (TrustZone).

TCM is used for a few things:

- FIQ and other interrupt handlers that need deterministic timing and cannot wait for cache misses.
- Idle loops where all external RAM is set to self-refresh retention mode, so only on-chip RAM is accessible by the CPU and then we hang inside ITCM waiting for an interrupt.
- Other operations which implies shutting off or reconfiguring the external RAM controller.

There is an interface for using TCM on the ARM architecture in <asm/tcm.h>. Using this interface it is possible to:

- Define the physical address and size of ITCM and DTCM.
- Tag functions to be compiled into ITCM.
- Tag data and constants to be allocated to DTCM and ITCM.

- Have the remaining TCM RAM added to a special allocation pool with gen_pool_create() and gen_pool_add() and provide tcm_alloc() and tcm_free() for this memory. Such a heap is great for things like saving device state when shutting off device power domains.

A machine that has TCM memory shall select HAVE_TCM from arch/arm/Kconfig for itself. Code that needs to use TCM shall #include <asm/tcm.h>

Functions to go into itcm can be tagged like this: int __tcmfunc foo(int bar);

Since these are marked to become long_calls and you may want to have functions called locally inside the TCM without wasting space, there is also the __tcmlocalfunc prefix that will make the call relative.

Variables to go into dtcm can be tagged like this:

```
int __tcmdata foo;
```

Constants can be tagged like this:

```
int __tcmconst foo;
```

To put assembler into TCM just use:

```
.section ".tcm.text" or .section ".tcm.data"
```

respectively.

Example code:

```
#include <asm/tcm.h>

/* Uninitialized data */
static u32 __tcmdata tcmvar;
/* Initialized data */
static u32 __tcmdata tcmassigned = 0x2BADBABEU;
/* Constant */
static const u32 __tcmconst tcmconst = 0xCAFEBABEU;

static void __tcmlocalfunc tcm_to_tcm(void)
{
      int i;
      for (i = 0; i < 100; i++)
              tcmvar ++;
}

static void __tcmfunc hello_tcm(void)
{
      /* Some abstract code that runs in ITCM */
      int i;
      for (i = 0; i < 100; i++) {
              tcmvar ++;
      }
      tcm_to_tcm();
}
```

```
static void __init test_tcm(void)
{
	u32 *tcmem;
	int i;

	hello_tcm();
	printk("Hello TCM executed from ITCM RAM\n");

	printk("TCM variable from testrun: %u @ %p\n", tcmvar, &tcmvar);
	tcmvar = 0xDEADBEEFU;
	printk("TCM variable: 0x%x @ %p\n", tcmvar, &tcmvar);

	printk("TCM assigned variable: 0x%x @ %p\n", tcmassigned, &tcmassigned);

	printk("TCM constant: 0x%x @ %p\n", tcmconst, &tcmconst);

	/* Allocate some TCM memory from the pool */
	tcmem = tcm_alloc(20);
	if (tcmem) {
		printk("TCM Allocated 20 bytes of TCM @ %p\n", tcmem);
		tcmem[0] = 0xDEADBEEFU;
		tcmem[1] = 0x2BADBABEU;
		tcmem[2] = 0xCAFEBABEU;
		tcmem[3] = 0xDEADBEEFU;
		tcmem[4] = 0x2BADBABEU;
		for (i = 0; i < 5; i++)
			printk("TCM tcmem[%d] = %08x\n", i, tcmem[i]);
		tcm_free(tcmem, 20);
	}
}
```

2.11 Kernel initialisation parameters on ARM Linux

The following document describes the kernel initialisation parameter structure, otherwise known as 'struct param_struct' which is used for most ARM Linux architectures.

This structure is used to pass initialisation parameters from the kernel loader to the Linux kernel proper, and may be short lived through the kernel initialisation process. As a general rule, it should not be referenced outside of arch/arm/kernel/setup.c:setup_arch().

There are a lot of parameters listed in there, and they are described below:

page_size
This parameter must be set to the page size of the machine, and will be checked by the kernel.

nr_pages
This is the total number of pages of memory in the system. If the memory is banked, then this should contain the total number of pages in the system.

If the system contains separate VRAM, this value should not include this information.

ramdisk_size
This is now obsolete, and should not be used.

flags
Various kernel flags, including:

bit 0	1 = mount root read only
bit 1	unused
bit 2	0 = load ramdisk
bit 3	0 = prompt for ramdisk

rootdev
major/minor number pair of device to mount as the root filesystem.

video_num_cols / video_num_rows
These two together describe the character size of the dummy console, or VGA console character size. They should not be used for any other purpose.

It's generally a good idea to set these to be either standard VGA, or the equivalent character size of your fbcon display. This then allows all the bootup messages to be displayed correctly.

video_x / video_y
This describes the character position of cursor on VGA console, and is otherwise unused. (should not be used for other console types, and should not be used for other purposes).

memc_control_reg
MEMC chip control register for Acorn Archimedes and Acorn A5000 based machines. May be used differently by different architectures.

sounddefault
Default sound setting on Acorn machines. May be used differently by different architectures.

adfsdrives
Number of ADFS/MFM disks. May be used differently by different architectures.

bytes_per_char_h / bytes_per_char_v
These are now obsolete, and should not be used.

pages_in_bank[4]
Number of pages in each bank of the systems memory (used for RiscPC). This is intended to be used on systems where the physical memory is non-contiguous from the processors point of view.

pages_in_vram
Number of pages in VRAM (used on Acorn RiscPC). This value may also be used by loaders if the size of the video RAM can't be obtained from the hardware.

initrd_start / initrd_size
This describes the kernel virtual start address and size of the initial ramdisk.

rd_start
Start address in sectors of the ramdisk image on a floppy disk.

system_rev
system revision number.

system_serial_low / system_serial_high
system 64-bit serial number

mem_fclk_21285
The speed of the external oscillator to the 21285 (footbridge), which control's the speed of the memory bus, timer & serial port. Depending upon the speed of the cpu its value can be between 0-66 MHz. If no params are passed or a value of zero is passed, then a value of 50 Mhz is the default on 21285 architectures.

paths[8][128]
These are now obsolete, and should not be used.

commandline
Kernel command line parameters. Details can be found elsewhere.

2.12 Software emulation of deprecated SWP instruction (CONFIG_SWP_EMULATE)

ARMv6 architecture deprecates use of the SWP/SWPB instructions, and recommends moving to the load-locked/store-conditional instructions LDREX and STREX.

ARMv7 multiprocessing extensions introduce the ability to disable these instructions, triggering an undefined instruction exception when executed. Trapped instructions are emulated using an LDREX/STREX or LDREXB/STREXB sequence. If a memory access fault (an abort) occurs, a segmentation fault is signalled to the triggering process.

/proc/cpu/swp_emulation holds some statistics/information, including the PID of the last process to trigger the emulation to be invocated. For example:

```
Emulated SWP:         12
Emulated SWPB:                0
Aborted SWP{B}:               1
Last process:         314
```

NOTE:
when accessing uncached shared regions, LDREX/STREX rely on an external transaction monitoring block called a global monitor to maintain update atomicity. If your system does not implement a global monitor, this option can cause programs that perform SWP operations to uncached memory to deadlock, as the STREX operation will always fail.

2.13 The Unified Extensible Firmware Interface (UEFI)

UEFI, the Unified Extensible Firmware Interface, is a specification governing the behaviours of compatible firmware interfaces. It is maintained by the UEFI Forum - http://www.uefi.org/.

UEFI is an evolution of its predecessor 'EFI', so the terms EFI and UEFI are used somewhat interchangeably in this document and associated source code. As a rule, anything new uses 'UEFI', whereas 'EFI' refers to legacy code or specifications.

2.13.1 UEFI support in Linux

Booting on a platform with firmware compliant with the UEFI specification makes it possible for the kernel to support additional features:

- UEFI Runtime Services
- Retrieving various configuration information through the standardised interface of UEFI configuration tables. (ACPI, SMBIOS, ...)

For actually enabling [U]EFI support, enable:

- CONFIG_EFI=y
- CONFIG_EFIVAR_FS=y or m

The implementation depends on receiving information about the UEFI environment in a Flattened Device Tree (FDT) - so is only available with CONFIG_OF.

2.13.2 UEFI stub

The "stub" is a feature that extends the Image/zImage into a valid UEFI PE/COFF executable, including a loader application that makes it possible to load the kernel directly from the UEFI shell, boot menu, or one of the lightweight bootloaders like Gummiboot or rEFInd.

The kernel image built with stub support remains a valid kernel image for booting in non-UEFI environments.

2.13.3 UEFI kernel support on ARM

UEFI kernel support on the ARM architectures (arm and arm64) is only available when boot is performed through the stub.

When booting in UEFI mode, the stub deletes any memory nodes from a provided DT. Instead, the kernel reads the UEFI memory map.

The stub populates the FDT /chosen node with (and the kernel scans for) the following parameters:

Name	Type	Description
linux,uefi-system-table	64-bit	Physical address of the UEFI System Table.
linux,uefi-mmap-start	64-bit	Physical address of the UEFI memory map, populated by the UEFI GetMemoryMap() call.
linux,uefi-mmap-size	32-bit	Size in bytes of the UEFI memory map pointed to in previous entry.
linux,uefi-mmap-desc-size	32-bit	Size in bytes of each entry in the UEFI memory map.
linux,uefi-mmap-desc-ver	32-bit	Version of the mmap descriptor format.
kaslr-seed	64-bit	Entropy used to randomize the kernel image base address location.
bootargs	String	Kernel command line

2.14 vlocks for Bare-Metal Mutual Exclusion

Voting Locks, or "vlocks" provide a simple low-level mutual exclusion mechanism, with reasonable but minimal requirements on the memory system.

These are intended to be used to coordinate critical activity among CPUs which are otherwise non-coherent, in situations where the hardware provides no other mechanism to support this and ordinary spinlocks cannot be used.

vlocks make use of the atomicity provided by the memory system for writes to a single memory location. To arbitrate, every CPU "votes for itself", by storing a unique number to a common memory location. The final value seen in that memory location when all the votes have been cast identifies the winner.

In order to make sure that the election produces an unambiguous result in finite time, a CPU will only enter the election in the first place if no winner has been chosen and the election does not appear to have started yet.

2.14.1 Algorithm

The easiest way to explain the vlocks algorithm is with some pseudo-code:

```
int currently_voting[NR_CPUS] = { 0, };
int last_vote = -1; /* no votes yet */

bool vlock_trylock(int this_cpu)
{
        /* signal our desire to vote */
        currently_voting[this_cpu] = 1;
        if (last_vote != -1) {
                /* someone already volunteered himself */
                currently_voting[this_cpu] = 0;
                return false; /* not ourself */
        }

        /* let's suggest ourself */
        last_vote = this_cpu;
        currently_voting[this_cpu] = 0;

        /* then wait until everyone else is done voting */
        for_each_cpu(i) {
                while (currently_voting[i] != 0)
                        /* wait */;
        }

        /* result */
        if (last_vote == this_cpu)
                return true; /* we won */
        return false;
}

bool vlock_unlock(void)
{
        last_vote = -1;
}
```

The currently_voting[] array provides a way for the CPUs to determine whether an election is in progress, and plays a role analogous to the "entering" array in Lamport's bakery algorithm [1].

However, once the election has started, the underlying memory system atomicity is used to pick the winner. This avoids the need for a static priority rule to act as a tie-breaker, or any counters which could overflow.

As long as the last_vote variable is globally visible to all CPUs, it will contain only one value that won't change once every CPU has cleared its currently_voting flag.

2.14.2 Features and limitations

- vlocks are not intended to be fair. In the contended case, it is the _last_ CPU which attempts to get the lock which will be most likely to win.

  vlocks are therefore best suited to situations where it is necessary to pick a unique winner, but it does not matter which CPU actually wins.

- Like other similar mechanisms, vlocks will not scale well to a large number of CPUs.

  vlocks can be cascaded in a voting hierarchy to permit better scaling if necessary, as in the following hypothetical example for 4096 CPUs:

  ```
  /* first level: local election */
  my_town = towns[(this_cpu >> 4) & 0xf];
  I_won = vlock_trylock(my_town, this_cpu & 0xf);
  if (I_won) {
          /* we won the town election, let's go for the state */
          my_state = states[(this_cpu >> 8) & 0xf];
          I_won = vlock_lock(my_state, this_cpu & 0xf));
          if (I_won) {
                  /* and so on */
                  I_won = vlock_lock(the_whole_country, this_cpu & 0xf];
  ```

```
                if (I_won) {
                        /* ... */
                }
                vlock_unlock(the_whole_country);
        }
        vlock_unlock(my_state);
}
vlock_unlock(my_town);
```

2.14.3 ARM implementation

The current ARM implementation [2] contains some optimisations beyond the basic algorithm:

- By packing the members of the currently_voting array close together, we can read the whole array in one transaction (providing the number of CPUs potentially contending the lock is small enough). This reduces the number of round-trips required to external memory.

  In the ARM implementation, this means that we can use a single load and comparison:

  ```
  LDR     Rt, [Rn]
  CMP     Rt, #0
  ```

  ...in place of code equivalent to:

  ```
  LDRB    Rt, [Rn]
  CMP     Rt, #0
  LDRBEQ  Rt, [Rn, #1]
  CMPEQ   Rt, #0
  LDRBEQ  Rt, [Rn, #2]
  CMPEQ   Rt, #0
  LDRBEQ  Rt, [Rn, #3]
  CMPEQ   Rt, #0
  ```

  This cuts down on the fast-path latency, as well as potentially reducing bus contention in contended cases.

  The optimisation relies on the fact that the ARM memory system guarantees coherency between overlapping memory accesses of different sizes, similarly to many other architectures. Note that we do not care which element of currently_voting appears in which bits of Rt, so there is no need to worry about endianness in this optimisation.

  If there are too many CPUs to read the currently_voting array in one transaction then multiple transactions are still required. The implementation uses a simple loop of word-sized loads for this case. The number of transactions is still fewer than would be required if bytes were loaded individually.

  In principle, we could aggregate further by using LDRD or LDM, but to keep the code simple this was not attempted in the initial implementation.

- vlocks are currently only used to coordinate between CPUs which are unable to enable their caches yet. This means that the implementation removes many of the barriers which would be required when executing the algorithm in cached memory.

packing of the currently_voting array does not work with cached memory unless all CPUs contending the lock are cache-coherent, due to cache writebacks from one CPU clobbering values written by other CPUs. (Though if all the CPUs are cache-coherent, you should be probably be using proper spinlocks instead anyway).

- The "no votes yet" value used for the last_vote variable is 0 (not -1 as in the pseudocode). This allows statically-allocated vlocks to be implicitly initialised to an unlocked state simply by putting them in .bss.

  An offset is added to each CPU's ID for the purpose of setting this variable, so that no CPU uses the value 0 for its ID.

2.14.4 Colophon

Originally created and documented by Dave Martin for Linaro Limited, for use in ARM-based big.LITTLE platforms, with review and input gratefully received from Nicolas Pitre and Achin Gupta. Thanks to Nicolas for grabbing most of this text out of the relevant mail thread and writing up the pseudocode.

2.14.5 References

[1] Lamport, L. "A New Solution of Dijkstra's Concurrent Programming
Problem", Communications of the ACM 17, 8 (August 1974), 453-455.

https://en.wikipedia.org/wiki/Lamport%27s_bakery_algorithm

[2] linux/arch/arm/common/vlock.S, www.kernel.org.

2.15 Porting

Taken from list archive at http://lists.arm.linux.org.uk/pipermail/linux-arm-kernel/2001-July/004064.html

2.15.1 Initial definitions

The following symbol definitions rely on you knowing the translation that __virt_to_phys() does for your machine. This macro converts the passed virtual address to a physical address. Normally, it is simply:

phys = virt - PAGE_OFFSET + PHYS_OFFSET

2.15.2 Decompressor Symbols

ZTEXTADDR
Start address of decompressor. There's no point in talking about virtual or physical addresses here, since the MMU will be off at the time when you call the decompressor code. You normally call the kernel at this address to start it booting. This doesn't have to be located in RAM, it can be in flash or other read-only or read-write addressable medium.

ZBSSADDR
Start address of zero-initialised work area for the decompressor. This must be pointing at RAM. The decompressor will zero initialise this for you. Again, the MMU will be off.

ZRELADDR
This is the address where the decompressed kernel will be written, and eventually executed. The following constraint must be valid:

__virt_to_phys(TEXTADDR) == ZRELADDR

The initial part of the kernel is carefully coded to be position independent.

INITRD_PHYS
Physical address to place the initial RAM disk. Only relevant if you are using the bootpImage stuff (which only works on the old struct param_struct).

INITRD_VIRT
Virtual address of the initial RAM disk. The following constraint must be valid:

__virt_to_phys(INITRD_VIRT) == INITRD_PHYS

PARAMS_PHYS
Physical address of the struct param_struct or tag list, giving the kernel various parameters about its execution environment.

2.15.3 Kernel Symbols

PHYS_OFFSET
Physical start address of the first bank of RAM.

PAGE_OFFSET
Virtual start address of the first bank of RAM. During the kernel boot phase, virtual address PAGE_OFFSET will be mapped to physical address PHYS_OFFSET, along with any other mappings you supply. This should be the same value as TASK_SIZE.

TASK_SIZE
The maximum size of a user process in bytes. Since user space always starts at zero, this is the maximum address that a user process can access+1. The user space stack grows down from this address.

Any virtual address below TASK_SIZE is deemed to be user process area, and therefore managed dynamically on a process by process basis by the kernel. I'll call this the user segment.

Anything above TASK_SIZE is common to all processes. I'll call this the kernel segment.

(In other words, you can't put IO mappings below TASK_SIZE, and hence PAGE_OFFSET).

TEXTADDR
Virtual start address of kernel, normally PAGE_OFFSET + 0x8000. This is where the kernel image ends up. With the latest kernels, it must be located at 32768 bytes into a 128MB region. Previous kernels placed a restriction of 256MB here.

DATAADDR
Virtual address for the kernel data segment. Must not be defined when using the decompressor.

VMALLOC_START / VMALLOC_END
Virtual addresses bounding the vmalloc() area. There must not be any static mappings in this area; vmalloc will overwrite them. The addresses must also be in the kernel segment (see above). Normally, the vmalloc() area starts VMALLOC_OFFSET bytes above the last virtual RAM address (found using variable high_memory).

VMALLOC_OFFSET
Offset normally incorporated into VMALLOC_START to provide a hole between virtual RAM and the vmalloc area. We do this to allow out of bounds memory accesses (eg, something writing off the end of the mapped memory map) to be caught. Normally set to 8MB.

2.15.4 Architecture Specific Macros

BOOT_MEM(pram,pio,vio)
pram specifies the physical start address of RAM. Must always be present, and should be the same as PHYS_OFFSET.

pio is the physical address of an 8MB region containing IO for use with the debugging macros in arch/arm/kernel/debug-armv.S.

vio is the virtual address of the 8MB debugging region.

It is expected that the debugging region will be re-initialised by the architecture specific code later in the code (via the MAPIO function).

BOOT_PARAMS
Same as, and see PARAMS_PHYS.

FIXUP(func)
Machine specific fixups, run before memory subsystems have been initialised.

MAPIO(func)
Machine specific function to map IO areas (including the debug region above).

INITIRQ(func)
Machine specific function to initialise interrupts.

2.16 Feature status on arm architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	ok
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok
core	thread-info-in-task	THREAD_INFO_IN_TASK	ok
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	TODO
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	ok
debug	KASAN	HAVE_ARCH_KASAN	ok
debug	kcov	ARCH_HAS_KCOV	ok
debug	kgdb	HAVE_ARCH_KGDB	ok
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok
debug	kprobes	HAVE_KPROBES	ok
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	TODO
debug	kretprobes	HAVE_KRETPROBES	ok
debug	optprobes	HAVE_OPTPROBES	ok
debug	stackprotector	HAVE_STACKPROTECTOR	ok
debug	uprobes	ARCH_SUPPORTS_UPROBES	ok
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	ok
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	TODO
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	TODO
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok
perf	perf-regs	HAVE_PERF_REGS	ok
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	ok
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	ok
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	---
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	ok
time	clockevents	!LEGACY_TIMER_TICK	TODO
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	ok
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	ok
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	ok
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	TODO
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	ok
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO
vm	ioremap_prot	HAVE_IOREMAP_PROT	TODO
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	ok

2.17 SoC-specific documents

2.17.1 Chromebook Boot Flow

Most recent Chromebooks that use device tree are using the opensource depthcharge bootloader. Depthcharge expects the OS to be packaged as a FIT Image which contains an OS image as well as a collection of device trees. It is up to depthcharge to pick the right device tree from the FIT Image and provide it to the OS.

The scheme that depthcharge uses to pick the device tree takes into account three variables:

- Board name, specified at depthcharge compile time. This is $(BOARD) below.
- Board revision number, determined at runtime (perhaps by reading GPIO strappings, perhaps via some other method). This is $(REV) below.
- SKU number, read from GPIO strappings at boot time. This is $(SKU) below.

For recent Chromebooks, depthcharge creates a match list that looks like this:

- google,$(BOARD)-rev$(REV)-sku$(SKU)
- google,$(BOARD)-rev$(REV)
- google,$(BOARD)-sku$(SKU)
- google,$(BOARD)

Note that some older Chromebooks use a slightly different list that may not include SKU matching or may prioritize SKU/rev differently.

Note that for some boards there may be extra board-specific logic to inject extra compatibles into the list, but this is uncommon.

Depthcharge will look through all device trees in the FIT Image trying to find one that matches the most specific compatible. It will then look through all device trees in the FIT Image trying to find the one that matches the *second most* specific compatible, etc.

When searching for a device tree, depthcharge doesn't care where the compatible string falls within a device tree's root compatible string array. As an example, if we're on board "lazor", rev 4, SKU 0 and we have two device trees:

- "google,lazor-rev5-sku0", "google,lazor-rev4-sku0", "qcom,sc7180"
- "google,lazor", "qcom,sc7180"

Then depthcharge will pick the first device tree even though "google,lazor-rev4-sku0" was the second compatible listed in that device tree. This is because it is a more specific compatible than "google,lazor".

It should be noted that depthcharge does not have any smarts to try to match board or SKU revisions that are "close by". That is to say that if depthcharge knows it's on "rev4" of a board but there is no "rev4" device tree then depthcharge *won't* look for a "rev3" device tree.

In general when any significant changes are made to a board the board revision number is increased even if none of those changes need to be reflected in the device tree. Thus it's fairly common to see device trees with multiple revisions.

It should be noted that, taking into account the above system that depthcharge has, the most flexibility is achieved if the device tree supporting the newest revision(s) of a board omits the

"-rev{REV}" compatible strings. When this is done then if you get a new board revision and try to run old software on it then we'll at pick the newest device tree we know about.

2.17.2 Release Notes for Linux on Intel's IXP4xx Network Processor

Maintained by Deepak Saxena <dsaxena@plexity.net>

1. Overview

Intel's IXP4xx network processor is a highly integrated SOC that is targeted for network applications, though it has become popular in industrial control and other areas due to low cost and power consumption. The IXP4xx family currently consists of several processors that support different network offload functions such as encryption, routing, firewalling, etc. The IXP46x family is an updated version which supports faster speeds, new memory and flash configurations, and more integration such as an on-chip I2C controller.

For more information on the various versions of the CPU, see:

> http://developer.intel.com/design/network/products/npfamily/ixp4xx.htm

Intel also made the IXCP1100 CPU for sometime which is an IXP4xx stripped of much of the network intelligence.

2. Linux Support

Linux currently supports the following features on the IXP4xx chips:

- Dual serial ports
- PCI interface
- Flash access (MTD/JFFS)
- I2C through GPIO on IXP42x
- GPIO for input/output/interrupts See arch/arm/mach-ixp4xx/include/mach/platform.h for access functions.
- Timers (watchdog, OS)

The following components of the chips are not supported by Linux and require the use of Intel's proprietary CSR software:

- USB device interface
- Network interfaces (HSS, Utopia, NPEs, etc)
- Network offload functionality

If you need to use any of the above, you need to download Intel's software from:

> http://developer.intel.com/design/network/products/npfamily/ixp425.htm

DO NOT POST QUESTIONS TO THE LINUX MAILING LISTS REGARDING THE PROPRIETARY SOFTWARE.

There are several websites that provide directions/pointers on using Intel's software:

- http://sourceforge.net/projects/ixp4xx-osdg/ Open Source Developer's Guide for using uClinux and the Intel libraries

- http://gatewaymaker.sourceforge.net/ Simple one page summary of building a gateway using an IXP425 and Linux
- http://ixp425.sourceforge.net/ ATM device driver for IXP425 that relies on Intel's libraries

3. Known Issues/Limitations

3a. Limited inbound PCI window

The IXP4xx family allows for up to 256MB of memory but the PCI interface can only expose 64MB of that memory to the PCI bus. This means that if you are running with > 64MB, all PCI buffers outside of the accessible range will be bounced using the routines in arch/arm/common/dmabounce.c.

3b. Limited outbound PCI window

IXP4xx provides two methods of accessing PCI memory space:

1) A direct mapped window from 0x48000000 to 0x4bffffff (64MB). To access PCI via this space, we simply ioremap() the BAR into the kernel and we can use the standard read[bwl]/write[bwl] macros. This is the preferred method due to speed but it limits the system to just 64MB of PCI memory. This can be problematic if using video cards and other memory-heavy devices.
2) If > 64MB of memory space is required, the IXP4xx can be configured to use indirect registers to access PCI This allows for up to 128MB (0x48000000 to 0x4fffffff) of memory on the bus. The disadvantage of this is that every PCI access requires three local register accesses plus a spinlock, but in some cases the performance hit is acceptable. In addition, you cannot mmap() PCI devices in this case due to the indirect nature of the PCI window.

By default, the direct method is used for performance reasons. If you need more PCI memory, enable the IXP4XX_INDIRECT_PCI config option.

3c. GPIO as Interrupts

Currently the code only handles level-sensitive GPIO interrupts

4. Supported platforms

ADI Engineering Coyote Gateway Reference Platform http://www.adiengineering.com/productsCoyote.html

> The ADI Coyote platform is reference design for those building small residential/office gateways. One NPE is connected to a 10/100 interface, one to 4-port 10/100 switch, and the third to and ADSL interface. In addition, it also supports to POTs interfaces connected via SLICs. Note that those are not supported by Linux ATM. Finally, the platform has two mini-PCI slots used for 802.11[bga] cards. Finally, there is an IDE port hanging off the expansion bus.

Gateworks Avila Network Platform http://www.gateworks.com/support/overview.php

> The Avila platform is basically and IXDP425 with the 4 PCI slots replaced with mini-PCI slots and a CF IDE interface hanging off the expansion bus.

Intel IXDP425 Development Platform http://www.intel.com/design/network/products/npfamily/ixdpg425.htm

> This is Intel's standard reference platform for the IXDP425 and is also known as the Richfield board. It contains 4 PCI slots, 16MB of flash, two 10/100 ports and one ADSL port.

Intel IXDP465 Development Platform http://www.intel.com/design/network/products/npfamily/ixdp465.htm

> This is basically an IXDP425 with an IXP465 and 32M of flash instead of just 16.

Intel IXDPG425 Development Platform

> This is basically and ADI Coyote board with a NEC EHCI controller added. One issue with this board is that the mini-PCI slots only have the 3.3v line connected, so you can't use a PCI to mini-PCI adapter with an E100 card. So to NFS root you need to use either the CSR or a WiFi card and a ramdisk that BOOTPs and then does a pivot_root to NFS.

Motorola PrPMC1100 Processor Mezanine Card http://www.fountainsys.com

> The PrPMC1100 is based on the IXCP1100 and is meant to plug into and IXP2400/2800 system to act as the system controller. It simply contains a CPU and 16MB of flash on the board and needs to be plugged into a carrier board to function. Currently Linux only supports the Motorola PrPMC carrier board for this platform.

5. TODO LIST

- Add support for Coyote IDE
- Add support for edge-based GPIO interrupts
- Add support for CF IDE on expansion bus

6. Thanks

The IXP4xx work has been funded by Intel Corp. and MontaVista Software, Inc.

The following people have contributed patches/comments/etc:

- Lennerty Buytenhek
- Lutz Jaenicke
- Justin Mayfield
- Robert E. Ranslam

[I know I've forgotten others, please email me to be added]

Last Update: 01/04/2005

2.17.3 ARM Marvell SoCs

This document lists all the ARM Marvell SoCs that are currently supported in mainline by the Linux kernel. As the Marvell families of SoCs are large and complex, it is hard to understand where the support for a particular SoC is available in the Linux kernel. This document tries to help in understanding where those SoCs are supported, and to match them with their corresponding public datasheet, when available.

Orion family

Flavors:

- 88F5082
- 88F5181 a.k.a Orion-1
- 88F5181L a.k.a Orion-VoIP
- 88F5182 a.k.a Orion-NAS
  - Datasheet: https://web.archive.org/web/20210124231420/http://csclub.uwaterloo.ca/~board/ts7800/MV88F5182-datasheet.pdf
  - Programmer's User Guide: https://web.archive.org/web/20210124231536/http://csclub.uwaterloo.ca/~board/ts7800/MV88F5182-opensource-manual.pdf
  - User Manual: https://web.archive.org/web/20210124231631/http://csclub.uwaterloo.ca/~board/ts7800/MV88F5182-usermanual.pdf
  - Functional Errata: https://web.archive.org/web/20210704165540/https://www.digriz.org.uk/ts78xx/88F5182_Functional_Errata.pdf
- 88F5281 a.k.a Orion-2
  - Datasheet: https://web.archive.org/web/20131028144728/http://www.ocmodshop.com/images/reviews/networking/qnap_ts409u/marvel_88f5281_data_sheet.pdf
- 88F6183 a.k.a Orion-1-90

Homepage:
https://web.archive.org/web/20080607215437/http://www.marvell.com/products/media/index.jsp

Core:
Feroceon 88fr331 (88f51xx) or 88fr531-vd (88f52xx) ARMv5 compatible

Linux kernel mach directory:
arch/arm/mach-orion5x

Linux kernel plat directory:
arch/arm/plat-orion

Kirkwood family

Flavors:

- 88F6282 a.k.a Armada 300
  - Product Brief : https://web.archive.org/web/20111027032509/http://www.marvell.com/embedded-processors/armada-300/assets/armada_310.pdf
- 88F6283 a.k.a Armada 310

  - Product Brief : https://web.archive.org/web/20111027032509/http://www.marvell.com/embedded-processors/armada-300/assets/armada_310.pdf

- 88F6190
  - Product Brief : https://web.archive.org/web/20130730072715/http://www.marvell.com/embedded-processors/kirkwood/assets/88F6190-003_WEB.pdf
  - Hardware Spec : https://web.archive.org/web/20121021182835/http://www.marvell.com/embedded-processors/kirkwood/assets/HW_88F619x_OpenSource.pdf
  - Functional Spec: https://web.archive.org/web/20130730091033/http://www.marvell.com/embedded-processors/kirkwood/assets/FS_88F6180_9x_6281_OpenSource.pdf
- 88F6192
  - Product Brief : https://web.archive.org/web/20131113121446/http://www.marvell.com/embedded-processors/kirkwood/assets/88F6192-003_ver1.pdf
  - Hardware Spec : https://web.archive.org/web/20121021182835/http://www.marvell.com/embedded-processors/kirkwood/assets/HW_88F619x_OpenSource.pdf
  - Functional Spec: https://web.archive.org/web/20130730091033/http://www.marvell.com/embedded-processors/kirkwood/assets/FS_88F6180_9x_6281_OpenSource.pdf
- 88F6182
- 88F6180
  - Product Brief : https://web.archive.org/web/20120616201621/http://www.marvell.com/embedded-processors/kirkwood/assets/88F6180-003_ver1.pdf
  - Hardware Spec : https://web.archive.org/web/20130730091654/http://www.marvell.com/embedded-processors/kirkwood/assets/HW_88F6180_OpenSource.pdf
  - Functional Spec: https://web.archive.org/web/20130730091033/http://www.marvell.com/embedded-processors/kirkwood/assets/FS_88F6180_9x_6281_OpenSource.pdf
- 88F6280
  - Product Brief : https://web.archive.org/web/20130730091058/http://www.marvell.com/embedded-processors/kirkwood/assets/88F6280_SoC_PB-001.pdf
- 88F6281
  - Product Brief : https://web.archive.org/web/20120131133709/http://www.marvell.com/embedded-processors/kirkwood/assets/88F6281-004_ver1.pdf

    - Hardware Spec : https://web.archive.org/web/20120620073511/http://www.marvell.com/embedded-processors/kirkwood/assets/HW_88F6281_OpenSource.pdf
    - Functional Spec: https://web.archive.org/web/20130730091033/http://www.marvell.com/embedded-processors/kirkwood/assets/FS_88F6180_9x_6281_OpenSource.pdf
- 88F6321
- 88F6322
- 88F6323
    - Product Brief : https://web.archive.org/web/20120616201639/http://www.marvell.com/embedded-processors/kirkwood/assets/88f632x_pb.pdf

Homepage:
https://web.archive.org/web/20160513194943/http://www.marvell.com/embedded-processors/kirkwood/

Core:
Feroceon 88fr131 ARMv5 compatible

Linux kernel mach directory:
arch/arm/mach-mvebu

Linux kernel plat directory:
none

Discovery family

Flavors:

- MV78100
    - Product Brief : https://web.archive.org/web/20120616194711/http://www.marvell.com/embedded-processors/discovery-innovation/assets/MV78100-003_WEB.pdf
    - Hardware Spec : https://web.archive.org/web/20141005120451/http://www.marvell.com/embedded-processors/discovery-innovation/assets/HW_MV78100_OpenSource.pdf
    - Functional Spec: https://web.archive.org/web/20111110081125/http://www.marvell.com/embedded-processors/discovery-innovation/assets/FS_MV76100_78100_78200_OpenSource.pdf
- MV78200
    - Product Brief : https://web.archive.org/web/20140801121623/http://www.marvell.com/embedded-processors/discovery-innovation/assets/MV78200-002_WEB.pdf
    - Hardware Spec : https://web.archive.org/web/20141005120458/http://www.marvell.com/embedded-processors/discovery-innovation/assets/HW_MV78200_OpenSource.pdf

  - Functional Spec: https://web.archive.org/web/20111110081125/http://www.marvell.com/embedded-processors/discovery-innovation/assets/FS_MV76100_78100_78200_OpenSource.pdf

- MV76100
  - Product Brief : https://web.archive.org/web/20140722064429/http://www.marvell.com/embedded-processors/discovery-innovation/assets/MV76100-002_WEB.pdf
  - Hardware Spec : https://web.archive.org/web/20140722064425/http://www.marvell.com/embedded-processors/discovery-innovation/assets/HW_MV76100_OpenSource.pdf
  - Functional Spec: https://web.archive.org/web/20111110081125/http://www.marvell.com/embedded-processors/discovery-innovation/assets/FS_MV76100_78100_78200_OpenSource.pdf

  Not supported by the Linux kernel.

Homepage:
https://web.archive.org/web/20110924171043/http://www.marvell.com/embedded-processors/discovery-innovation/

Core:
Feroceon 88fr571-vd ARMv5 compatible

Linux kernel mach directory:
arch/arm/mach-mv78xx0

Linux kernel plat directory:
arch/arm/plat-orion

EBU Armada family

Armada 370 Flavors:

- 88F6710
- 88F6707
- 88F6W11
- Product infos: https://web.archive.org/web/20141002083258/http://www.marvell.com/embedded-processors/armada-370/
- Product Brief: https://web.archive.org/web/20121115063038/http://www.marvell.com/embedded-processors/armada-300/assets/Marvell_ARMADA_370_SoC.pdf
- Hardware Spec: https://web.archive.org/web/20140617183747/http://www.marvell.com/embedded-processors/armada-300/assets/ARMADA370-datasheet.pdf
- Functional Spec: https://web.archive.org/web/20140617183701/http://www.marvell.com/embedded-processors/armada-300/assets/ARMADA370-FunctionalSpec-datasheet.pdf

Core:
Sheeva ARMv7 compatible PJ4B

Armada XP Flavors:

- MV78230
- MV78260
- MV78460

NOTE:
not to be confused with the non-SMP 78xx0 SoCs

- Product infos: https://web.archive.org/web/20150101215721/http://www.marvell.com/embedded-processors/armada-xp/
- Product Brief: https://web.archive.org/web/20121021173528/http://www.marvell.com/embedded-processors/armada-xp/assets/Marvell-ArmadaXP-SoC-product%20brief.pdf
- Functional Spec: https://web.archive.org/web/20180829171131/http://www.marvell.com/embedded-processors/armada-xp/assets/ARMADA-XP-Functional-SpecDatasheet.pdf
- **Hardware Specs:**
  - https://web.archive.org/web/20141127013651/http://www.marvell.com/embedded-processors/armada-xp/assets/HW_MV78230_OS.PDF
  - https://web.archive.org/web/20141222000224/http://www.marvell.com/embedded-processors/armada-xp/assets/HW_MV78260_OS.PDF
  - https://web.archive.org/web/20141222000230/http://www.marvell.com/embedded-processors/armada-xp/assets/HW_MV78460_OS.PDF

Core:
Sheeva ARMv7 compatible Dual-core or Quad-core PJ4B-MP

Armada 375 Flavors:

- 88F6720
- Product infos: https://web.archive.org/web/20140108032402/http://www.marvell.com/embedded-processors/armada-375/
- Product Brief: https://web.archive.org/web/20131216023516/http://www.marvell.com/embedded-processors/armada-300/assets/ARMADA_375_SoC-01_product_brief.pdf

Core:
ARM Cortex-A9

Armada 38x Flavors:

- 88F6810 Armada 380
- 88F6811 Armada 381
- 88F6821 Armada 382
- 88F6W21 Armada 383

- 88F6820 Armada 385
- 88F6825
- 88F6828 Armada 388
- Product infos: https://web.archive.org/web/20181006144616/http://www.marvell.com/embedded-processors/armada-38x/
- Functional Spec: https://web.archive.org/web/20200420191927/https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-embedded-processors-armada-38x-functional-specificatio pdf
- Hardware Spec: https://web.archive.org/web/20180713105318/https://www.marvell.com/docs/embedded-processors/assets/marvell-embedded-processors-armada-38x-hardware-specifications-2017-03.pdf
- Design guide: https://web.archive.org/web/20180712231737/https://www.marvell.com/docs/embedded-processors/assets/marvell-embedded-processors-armada-38x-hardware-design-guide-2017-08.pdf

Core:
ARM Cortex-A9

Armada 39x Flavors:

- 88F6920 Armada 390
- 88F6925 Armada 395
- 88F6928 Armada 398
- Product infos: https://web.archive.org/web/20181020222559/http://www.marvell.com/embedded-processors/armada-39x/

Core:
ARM Cortex-A9

Linux kernel mach directory:
arch/arm/mach-mvebu

Linux kernel plat directory:
none

EBU Armada family ARMv8

Armada 3710/3720 Flavors:

- 88F3710
- 88F3720

Core:
ARM Cortex A53 (ARMv8)

Homepage:
https://web.archive.org/web/20181103003602/http://www.marvell.com/embedded-processors/armada-3700/

Product Brief:
https://web.archive.org/web/20210121194810/https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-embedded-processors-armada-37xx-product-brief-2016-01.pdf

Hardware Spec:
https://web.archive.org/web/20210202162011/http://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-embedded-processors-armada-37xx-hardware-specifications-2019-09.pdf

Device tree files:
arch/arm64/boot/dts/marvell/armada-37*

Armada 7K Flavors:

- 88F6040 (AP806 Quad 600 MHz + one CP110)
- 88F7020 (AP806 Dual + one CP110)
- 88F7040 (AP806 Quad + one CP110)

Core: ARM Cortex A72

Homepage:
https://web.archive.org/web/20181020222606/http://www.marvell.com/embedded-processors/armada-70xx/

Product Brief:

- https://web.archive.org/web/20161010105541/http://www.marvell.com/embedded-processors/assets/Armada7020PB-Jan2016.pdf
- https://web.archive.org/web/20160928154533/http://www.marvell.com/embedded-processors/assets/Armada7040PB-Jan2016.pdf

Device tree files:
arch/arm64/boot/dts/marvell/armada-70*

Armada 8K Flavors:

- 88F8020 (AP806 Dual + two CP110)
- 88F8040 (AP806 Quad + two CP110)

Core:
ARM Cortex A72

Homepage:
https://web.archive.org/web/20181022004830/http://www.marvell.com/embedded-processors/armada-80xx/

Product Brief:

- https://web.archive.org/web/20210124233728/https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-embedded-processors-armada-8020-product-brief-2017-12.pdf

- https://web.archive.org/web/20161010105532/http://www.marvell.com/embedded-processors/assets/Armada8040PB-Jan2016.pdf

Device tree files:
arch/arm64/boot/dts/marvell/armada-80*

Octeon TX2 CN913x Flavors:

- CN9130 (AP807 Quad + one internal CP115)
- CN9131 (AP807 Quad + one internal CP115 + one external CP115 / 88F8215)
- CN9132 (AP807 Quad + one internal CP115 + two external CP115 / 88F8215)

Core:
ARM Cortex A72

Homepage:
https://web.archive.org/web/20200803150818/https://www.marvell.com/products/infrastructure-processors/multi-core-processors/octeon-tx2/octeon-tx2-cn9130.html

Product Brief:
https://web.archive.org/web/20200803150818/https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-infrastructure-processors-octeon-tx2-cn913x-product-brief-2020-02.pdf

Device tree files:
arch/arm64/boot/dts/marvell/cn913*

Avanta family

Flavors:

- 88F6500
- 88F6510
- 88F6530P
- 88F6550
- 88F6560
- 88F6601

Homepage:
https://web.archive.org/web/20181005145041/http://www.marvell.com/broadband/

Product Brief:
https://web.archive.org/web/20180829171057/http://www.marvell.com/broadband/assets/Marvell_Avanta_88F6510_305_060-001_product_brief.pdf

No public datasheet available.

Core:
ARMv5 compatible

Linux kernel mach directory:
no code in mainline yet, planned for the future

Linux kernel plat directory:
no code in mainline yet, planned for the future

Storage family

Armada SP:

- 88RC1580

Product infos:
https://web.archive.org/web/20191129073953/http://www.marvell.com/storage/armada-sp/

Core:
Sheeva ARMv7 compatible Quad-core PJ4C

(not supported in upstream Linux kernel)

Dove family (application processor)

Flavors:

- 88AP510 a.k.a Armada 510

  Product Brief:
  https://web.archive.org/web/20111102020643/http://www.marvell.com/application-processors/armada-500/assets/Marvell_Armada510_SoC.pdf

  Hardware Spec:
  https://web.archive.org/web/20160428160231/http://www.marvell.com/application-processors/armada-500/assets/Armada-510-Hardware-Spec.pdf

Functional Spec:
https://web.archive.org/web/20120130172443/http://www.marvell.com/application-processors/armada-500/assets/Armada-510-Functional-Spec.pdf

Homepage:
https://web.archive.org/web/20160822232651/http://www.marvell.com/application-processors/armada-500/

Core:
ARMv7 compatible

Directory:

- arch/arm/mach-mvebu (DT enabled platforms)
- arch/arm/mach-dove (non-DT enabled platforms)

PXA 2xx/3xx/93x/95x family

Flavors:

- **PXA21x, PXA25x, PXA26x**
  - Application processor only
  - Core: ARMv5 XScale1 core
- **PXA270, PXA271, PXA272**
  - Product Brief : https://web.archive.org/web/20150927135510/http://www.marvell.com/application-processors/pxa-family/assets/pxa_27x_pb.pdf
  - Design guide : https://web.archive.org/web/20120111181937/http://www.marvell.com/application-processors/pxa-family/assets/pxa_27x_design_guide.pdf
  - Developers manual : https://web.archive.org/web/20150927164805/http://www.marvell.com/application-processors/pxa-family/assets/pxa_27x_dev_man.pdf
  - Specification : https://web.archive.org/web/20140211221535/http://www.marvell.com/application-processors/pxa-family/assets/pxa_27x_emts.pdf
  - Specification update : https://web.archive.org/web/20120111104906/http://www.marvell.com/application-processors/pxa-family/assets/pxa_27x_spec_update.pdf
  - Application processor only
  - Core: ARMv5 XScale2 core
- **PXA300, PXA310, PXA320**
  - PXA 300 Product Brief : https://web.archive.org/web/20120111121203/http://www.marvell.com/application-processors/pxa-family/assets/PXA300_PB_R4.pdf
  - PXA 310 Product Brief : https://web.archive.org/web/20120111104515/http://www.marvell.com/application-processors/pxa-family/assets/PXA310_PB_R4.pdf
  - PXA 320 Product Brief : https://web.archive.org/web/20121021182826/http://www.marvell.com/application-processors/pxa-family/assets/PXA320_PB_R4.pdf
  - Design guide : https://web.archive.org/web/20130727144625/http://www.marvell.com/application-processors/pxa-family/assets/PXA3xx_Design_Guide.pdf
  - Developers manual : https://web.archive.org/web/20130727144605/http://www.marvell.com/application-processors/pxa-family/assets/PXA3xx_Developers_Manual.zip

  - Specifications : https://web.archive.org/web/20130727144559/http://www.marvell.com/application-processors/pxa-family/assets/PXA3xx_EMTS.pdf
  - Specification Update : https://web.archive.org/web/20150927183411/http://www.marvell.com/application-processors/pxa-family/assets/PXA3xx_Spec_Update.zip
  - Reference Manual : https://web.archive.org/web/20120111103844/http://www.marvell.com/application-processors/pxa-family/assets/PXA3xx_TavorP_BootROM_Ref_Manual.pdf
  - Application processor only
  - Core: ARMv5 XScale3 core
- **PXA930, PXA935**
  - Application processor with Communication processor
  - Core: ARMv5 XScale3 core
- **PXA955**
  - Application processor with Communication processor
  - Core: ARMv7 compatible Sheeva PJ4 core

Comments:

- This line of SoCs originates from the XScale family developed by Intel and acquired by Marvell in ~2006. The PXA21x, PXA25x, PXA26x, PXA27x, PXA3xx and PXA93x were developed by Intel, while the later PXA95x were developed by Marvell.
- Due to their XScale origin, these SoCs have virtually nothing in common with the other (Kirkwood, Dove, etc.) families of Marvell SoCs, except with the MMP/MMP2 family of SoCs.

Linux kernel mach directory:
arch/arm/mach-pxa

MMP/MMP2/MMP3 family (communication processor)

Flavors:

- **PXA168, a.k.a Armada 168**
  - Homepage : https://web.archive.org/web/20110926014256/http://www.marvell.com/application-processors/armada-100/armada-168.jsp
  - Product brief : https://web.archive.org/web/20111102030100/http://www.marvell.com/application-processors/armada-100/assets/pxa_168_pb.pdf
  - Hardware manual : https://web.archive.org/web/20160428165359/http://www.marvell.com/application-processors/armada-100/assets/armada_16x_datasheet.pdf

  - Software manual : https://web.archive.org/web/20160428154454/http://www.marvell.com/application-processors/armada-100/assets/armada_16x_software_manual.pdf
  - Specification update : https://web.archive.org/web/20150927160338/http://www.marvell.com/application-processors/armada-100/assets/ARMADA16x_Spec_update.pdf
  - Boot ROM manual : https://web.archive.org/web/20130727205559/http://www.marvell.com/application-processors/armada-100/assets/armada_16x_ref_manual.pdf
  - App node package : https://web.archive.org/web/20141005090706/http://www.marvell.com/application-processors/armada-100/assets/armada_16x_app_note_package.pdf
  - Application processor only
  - Core: ARMv5 compatible Marvell PJ1 88sv331 (Mohawk)

- **PXA910/PXA920**
  - Homepage : https://web.archive.org/web/20150928121236/http://www.marvell.com/communication-processors/pxa910/
  - Product Brief : https://archive.org/download/marvell-pxa910-pb/Marvell_PXA910_Platform-001_PB.pdf
  - Application processor with Communication processor
  - Core: ARMv5 compatible Marvell PJ1 88sv331 (Mohawk)
- **PXA688, a.k.a. MMP2, a.k.a Armada 610 (OLPC XO-1.75)**
  - Product Brief : https://web.archive.org/web/20111102023255/http://www.marvell.com/application-processors/armada-600/assets/armada610_pb.pdf
  - Application processor only
  - Core: ARMv7 compatible Sheeva PJ4 88sv581x core
- **PXA2128, a.k.a. MMP3, a.k.a Armada 620 (OLPC XO-4)**
  - Product Brief : https://web.archive.org/web/20120824055155/http://www.marvell.com/application-processors/armada/pxa2128/assets/Marvell-ARMADA-PXA2128-SoC-PB.pdf
  - Application processor only
  - Core: Dual-core ARMv7 compatible Sheeva PJ4C core
- **PXA960/PXA968/PXA978 (Linux support not upstream)**
  - Application processor with Communication Processor
  - Core: ARMv7 compatible Sheeva PJ4 core
- **PXA986/PXA988 (Linux support not upstream)**
  - Application processor with Communication Processor
  - Core: Dual-core ARMv7 compatible Sheeva PJ4B-MP core

  - **PXA1088/PXA1920 (Linux support not upstream)**
    - Application processor with Communication Processor
    - Core: quad-core ARMv7 Cortex-A7
  - **PXA1908/PXA1928/PXA1936**
    - Application processor with Communication Processor
    - Core: multi-core ARMv8 Cortex-A53

Comments:

- This line of SoCs originates from the XScale family developed by Intel and acquired by Marvell in ~2006. All the processors of this MMP/MMP2 family were developed by Marvell.
- Due to their XScale origin, these SoCs have virtually nothing in common with the other (Kirkwood, Dove, etc.) families of Marvell SoCs, except with the PXA family of SoCs listed above.

Linux kernel mach directory:
arch/arm/mach-mmp

Berlin family (Multimedia Solutions)

- **Flavors:**
  - **88DE3010, Armada 1000 (no Linux support)**
    - Core: Marvell PJ1 (ARMv5TE), Dual-core
    - Product Brief: https://web.archive.org/web/20131103162620/http://www.marvell.com/digital-entertainment/assets/armada_1000_pb.pdf
  - **88DE3005, Armada 1500 Mini**
    - Design name: BG2CD
    - Core: ARM Cortex-A9, PL310 L2CC
  - **88DE3006, Armada 1500 Mini Plus**
    - Design name: BG2CDP
    - Core: Dual Core ARM Cortex-A7
  - **88DE3100, Armada 1500**
    - Design name: BG2
    - Core: Marvell PJ4B-MP (ARMv7), Tauros3 L2CC
  - **88DE3114, Armada 1500 Pro**
    - Design name: BG2Q
    - Core: Quad Core ARM Cortex-A9, PL310 L2CC
  - **88DE3214, Armada 1500 Pro 4K**
    - Design name: BG3

    * Core: ARM Cortex-A15, CA15 integrated L2CC

  - **88DE3218, ARMADA 1500 Ultra**

    * Core: ARM Cortex-A53

Homepage: https://www.synaptics.com/products/multimedia-solutions Directory: arch/arm/mach-berlin

Comments:

- This line of SoCs is based on Marvell Sheeva or ARM Cortex CPUs with Synopsys DesignWare (IRQ, GPIO, Timers, ...) and PXA IP (SDHCI, USB, ETH, ...).
- The Berlin family was acquired by Synaptics from Marvell in 2017.

CPU Cores

The XScale cores were designed by Intel, and shipped by Marvell in the older PXA processors. Feroceon is a Marvell designed core that developed in-house, and that evolved into Sheeva. The XScale and Feroceon cores were phased out over time and replaced with Sheeva cores in later products, which subsequently got replaced with licensed ARM Cortex-A cores.

XScale 1
CPUID 0x69052xxx ARMv5, iWMMXt

XScale 2
CPUID 0x69054xxx ARMv5, iWMMXt

XScale 3
CPUID 0x69056xxx or 0x69056xxx ARMv5, iWMMXt

Feroceon-1850 88fr331 "Mohawk"
CPUID 0x5615331x or 0x41xx926x ARMv5TE, single issue

Feroceon-2850 88fr531-vd "Jolteon"
CPUID 0x5605531x or 0x41xx926x ARMv5TE, VFP, dual-issue

Feroceon 88fr571-vd "Jolteon"
CPUID 0x5615571x ARMv5TE, VFP, dual-issue

Feroceon 88fr131 "Mohawk-D"
CPUID 0x5625131x ARMv5TE, single-issue in-order

Sheeva PJ1 88sv331 "Mohawk"
CPUID 0x561584xx ARMv5, single-issue iWMMXt v2

Sheeva PJ4 88sv581x "Flareon"
CPUID 0x560f581x ARMv7, idivt, optional iWMMXt v2

Sheeva PJ4B 88sv581x
CPUID 0x561f581x ARMv7, idivt, optional iWMMXt v2

Sheeva PJ4B-MP / PJ4C
CPUID 0x562f584x ARMv7, idivt/idiva, LPAE, optional iWMMXt v2 and/or NEON

Long-term plans

- Unify the mach-dove/, mach-mv78xx0/, mach-orion5x/ into the mach-mvebu/ to support all SoCs from the Marvell EBU (Engineering Business Unit) in a single mach-<foo> directory. The plat-orion/ would therefore disappear.

Credits

- Maen Suleiman <maen@marvell.com>
- Lior Amsalem <alior@marvell.com>
- Thomas Petazzoni <thomas.petazzoni@free-electrons.com>
- Andrew Lunn <andrew@lunn.ch>
- Nicolas Pitre <nico@fluxnic.net>
- Eric Miao <eric.y.miao@gmail.com>

2.17.4 ARM Microchip SoCs (aka AT91)

Introduction

This document gives useful information about the ARM Microchip SoCs that are currently supported in Linux Mainline (you know, the one on kernel.org).

It is important to note that the Microchip (previously Atmel) ARM-based MPU product line is historically named "AT91" or "at91" throughout the Linux kernel development process even if this product prefix has completely disappeared from the official Microchip product name. Anyway, files, directories, git trees, git branches/tags and email subject always contain this "at91" sub-string.

AT91 SoCs

Documentation and detailed datasheet for each product are available on the Microchip website: http://www.microchip.com.

Flavors:

- ARM 920 based SoC - at91rm9200
  - Datasheet

    http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-1768-32-bit-ARM920T-Embedded-Microprocessor-AT91RM9200_Datasheet.pdf
- ARM 926 based SoCs - at91sam9260
  - Datasheet

    http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-6221-32-bit-ARM926EJ-S-Embedded-Microprocessor-SAM9260_Datasheet.pdf

- at91sam9xe
  - Datasheet

    http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-6254-32-bit-ARM926EJ-S-Embedded-Microprocessor-SAM9XE_Datasheet.pdf
- at91sam9261
  - Datasheet

    http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-6062-ARM926EJ-S-Microprocessor-SAM9261_Datasheet.pdf
- at91sam9263
  - Datasheet

    http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-6249-32-bit-ARM926EJ-S-Embedded-Microprocessor-SAM9263_Datasheet.pdf
- at91sam9rl
  - Datasheet

    http://ww1.microchip.com/downloads/en/DeviceDoc/doc6289.pdf
- at91sam9g20
  - Datasheet

    http://ww1.microchip.com/downloads/en/DeviceDoc/DS60001516A.pdf
- at91sam9g45 family - at91sam9g45 - at91sam9g46 - at91sam9m10 - at91sam9m11 (device superset)
  - Datasheet

    http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-6437-32-bit-ARM926-Embedded-Microprocessor-SAM9M11_Datasheet.pdf
- at91sam9x5 family (aka "The 5 series") - at91sam9g15 - at91sam9g25 - at91sam9g35 - at91sam9x25 - at91sam9x35
  - Datasheet (can be considered as covering the whole family)

    http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11055-32-bit-ARM926EJ-S-Microcontroller-SAM9X35_Datasheet.pdf
- at91sam9n12
  - Datasheet

    http://ww1.microchip.com/downloads/en/DeviceDoc/DS60001517A.pdf
- sam9x60

    * Datasheet

      http://ww1.microchip.com/downloads/en/DeviceDoc/SAM9X60-Data-Sheet-DS60001579A.pdf

- ARM Cortex-A5 based SoCs - sama5d3 family
  - sama5d31
  - sama5d33
  - sama5d34
  - sama5d35
  - sama5d36 (device superset)
    * Datasheet

    http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-11121-32-bit-Cortex-A5-Microcontroller-SAMA5D3_Datasheet_B.pdf
- ARM Cortex-A5 + NEON based SoCs - sama5d4 family
  - sama5d41
  - sama5d42
  - sama5d43
  - sama5d44 (device superset)
    * Datasheet

    http://ww1.microchip.com/downloads/en/DeviceDoc/60001525A.pdf
  - sama5d2 family
    * sama5d21
    * sama5d22
    * sama5d23
    * sama5d24
    * sama5d26
    * sama5d27 (device superset)
    * sama5d28 (device superset + environmental monitors)
      · Datasheet

      http://ww1.microchip.com/downloads/en/DeviceDoc/DS60001476B.pdf
- ARM Cortex-A7 based SoCs - sama7g5 family
  - sama7g51
  - sama7g52
  - sama7g53
  - sama7g54 (device superset)

        * Datasheet

          Coming soon

    - lan966 family - lan9662 - lan9668

        * Datasheet

          Coming soon

- ARM Cortex-M7 MCUs - sams70 family

    - sams70j19
    - sams70j20
    - sams70j21
    - sams70n19
    - sams70n20
    - sams70n21
    - sams70q19
    - sams70q20
    - sams70q21
    - samv70 family
        * samv70j19
        * samv70j20
        * samv70n19
        * samv70n20
        * samv70q19
        * samv70q20
    - samv71 family
        * samv71j19
        * samv71j20
        * samv71j21
        * samv71n19
        * samv71n20
        * samv71n21
        * samv71q19
        * samv71q20
        * samv71q21
            · Datasheet

              http://ww1.microchip.com/downloads/en/DeviceDoc/SAM-E70-S70-V70-V71-Family-Data-Sheet-DS60001527D.pdf

Linux kernel information

Linux kernel mach directory: arch/arm/mach-at91 MAINTAINERS entry is: “ARM/Microchip (AT91) SoC support”

Device Tree for AT91 SoCs and boards

All AT91 SoCs are converted to Device Tree. Since Linux 3.19, these products must use this method to boot the Linux kernel.

Work In Progress statement: Device Tree files and Device Tree bindings that apply to AT91 SoCs and boards are considered as “Unstable”. To be completely clear, any at91 binding can change at any time. So, be sure to use a Device Tree Binary and a Kernel Image generated from the same source tree. Please refer to the Documentation/devicetree/bindings/ABI.rst file for a definition of a “Stable” binding/ABI. This statement will be removed by AT91 MAINTAINERS when appropriate.

Naming conventions and best practice:

- SoCs Device Tree Source Include files are named after the official name of the product (at91sam9g20.dtsi or sama5d33.dtsi for instance).
- Device Tree Source Include files (.dtsi) are used to collect common nodes that can be shared across SoCs or boards (sama5d3.dtsi or at91sam9x5cm.dtsi for instance). When collecting nodes for a particular peripheral or topic, the identifier have to be placed at the end of the file name, separated with a “_” (at91sam9x5_can.dtsi or sama5d3_gmac.dtsi for example).
- board Device Tree Source files (.dts) are prefixed by the string “at91-” so that they can be identified easily. Note that some files are historical exceptions to this rule (sama5d3[13456]ek.dts, usb_a9g20.dts or animeo_ip.dts for example).

2.17.5 NetWinder specific documentation

The NetWinder is a small low-power computer, primarily designed to run Linux. It is based around the StrongARM RISC processor, DC21285 PCI bridge, with PC-type hardware glued around it.

Port usage

Min	Max	Description
0x0000	0x000f	DMA1
0x0020	0x0021	PIC1
0x0060	0x006f	Keyboard
0x0070	0x007f	RTC
0x0080	0x0087	DMA1
0x0088	0x008f	DMA2
0x00a0	0x00a3	PIC2
0x00c0	0x00df	DMA2
0x0180	0x0187	IRDA
0x01f0	0x01f6	ide0
0x0201		Game port
0x0203		RWA010 configuration read
0x0220	?	SoundBlaster
0x0250	?	WaveArtist
0x0279		RWA010 configuration index
0x02f8	0x02ff	Serial ttyS1
0x0300	0x031f	Ether10
0x0338		GPIO1
0x033a		GPIO2
0x0370	0x0371	W83977F configuration registers
0x0388	?	AdLib
0x03c0	0x03df	VGA
0x03f6		ide0
0x03f8	0x03ff	Serial ttyS0
0x0400	0x0408	DC21143
0x0480	0x0487	DMA1
0x0488	0x048f	DMA2
0x0a79		RWA010 configuration write
0xe800	0xe80f	ide0/ide1 BM DMA

Interrupt usage

IRQ	type	Description
0	ISA	100Hz timer
1	ISA	Keyboard
2	ISA	cascade
3	ISA	Serial ttyS1
4	ISA	Serial ttyS0
5	ISA	PS/2 mouse
6	ISA	IRDA
7	ISA	Printer
8	ISA	RTC alarm
9	ISA	
10	ISA	GP10 (Orange reset button)
11	ISA	
12	ISA	WaveArtist
13	ISA	
14	ISA	hda1
15	ISA	

DMA usage

DMA	type	Description
0	ISA	IRDA
1	ISA	
2	ISA	cascade
3	ISA	WaveArtist
4	ISA	
5	ISA	
6	ISA	
7	ISA	WaveArtist

2.17.6 NetWinder's floating point emulator

Introduction

This directory contains the version 0.92 test release of the NetWinder Floating Point Emulator.

The majority of the code was written by me, Scott Bambrough It is written in C, with a small number of routines in inline assembler where required. It was written quickly, with a goal of implementing a working version of all the floating point instructions the compiler emits as the first target. I have attempted to be as optimal as possible, but there remains much room for improvement.

I have attempted to make the emulator as portable as possible. One of the problems is with leading underscores on kernel symbols. Elf kernels have no leading underscores, a.out compiled kernels do. I have attempted to use the C_SYMBOL_NAME macro wherever this may be important.

Another choice I made was in the file structure. I have attempted to contain all operating system specific code in one module (fpmodule.*). All the other files contain emulator specific code. This should allow others to port the emulator to NetBSD for instance relatively easily.

The floating point operations are based on SoftFloat Release 2, by John Hauser. SoftFloat is a software implementation of floating-point that conforms to the IEC/IEEE Standard for Binary Floating-point Arithmetic. As many as four formats are supported: single precision, double precision, extended double precision, and quadruple precision. All operations required by the standard are implemented, except for conversions to and from decimal. We use only the single precision, double precision and extended double precision formats. The port of SoftFloat to the ARM was done by Phil Blundell, based on an earlier port of SoftFloat version 1 by Neil Carson for NetBSD/arm32.

The file README.FPE contains a description of what has been implemented so far in the emulator. The file TODO contains a information on what remains to be done, and other ideas for the emulator.

Bug reports, comments, suggestions should be directed to me at <scottb@netwinder.org>. General reports of "this program doesn't work correctly when your emulator is installed" are useful for determining that bugs still exist; but are virtually useless when attempting to isolate the problem. Please report them, but don't expect quick action. Bugs still exist. The problem remains in isolating which instruction contains the bug. Small programs illustrating a specific problem are a godsend.

Legal Notices

The NetWinder Floating Point Emulator is free software. Everything Rebel.com has written is provided under the GNU GPL. See the file COPYING for copying conditions. Excluded from the above is the SoftFloat code. John Hauser's legal notice for SoftFloat is included below.

SoftFloat Legal Notice

SoftFloat was written by John R. Hauser. This work was made possible in part by the International Computer Science Institute, located at Suite 600, 1947 Center Street, Berkeley, California 94704. Funding was partially provided by the National Science Foundation under grant MIP-9311980. The original version of this code was written as part of a project to build a fixed-point vector processor in collaboration with the University of California at Berkeley, overseen by Profs. Nelson Morgan and John Wawrzynek.

THIS SOFTWARE IS DISTRIBUTED AS IS, FOR FREE. Although reasonable effort has been made to avoid it, THIS SOFTWARE MAY CONTAIN FAULTS THAT WILL AT TIMES RESULT IN INCORRECT BEHAVIOR. USE OF THIS SOFTWARE IS RESTRICTED TO PERSONS AND ORGANIZATIONS WHO CAN AND WILL TAKE FULL RESPONSIBILITY FOR ANY AND ALL LOSSES, COSTS, OR OTHER PROBLEMS ARISING FROM ITS USE. --

Current State

The following describes the current state of the NetWinder's floating point emulator.

In the following nomenclature is used to describe the floating point instructions. It follows the conventions in the ARM manual.

```
<S|D|E> = <single|double|extended>, no default
{P|M|Z} = {round to +infinity,round to -infinity,round to zero},
          default = round to nearest
```

Note: items enclosed in {} are optional.

Floating Point Coprocessor Data Transfer Instructions (CPDT)

LDF/STF - load and store floating

<LDF|STF>{cond}<S|D|E> Fd, Rn <LDF|STF>{cond}<S|D|E> Fd, [Rn, #<expression>]{!} <LDF|STF>{cond}<S|D|E> Fd, [Rn], #<expression>

These instructions are fully implemented.

LFM/SFM - load and store multiple floating

Form 1 syntax: <LFM|SFM>{cond}<S|D|E> Fd, <count>, [Rn] <LFM|SFM>{cond}<S|D|E> Fd, <count>, [Rn, #<expression>]{!} <LFM|SFM>{cond}<S|D|E> Fd, <count>, [Rn], #<expression>

Form 2 syntax: <LFM|SFM>{cond}<FD,EA> Fd, <count>, [Rn]{!}

These instructions are fully implemented. They store/load three words for each floating point register into the memory location given in the instruction. The format in memory is unlikely to be compatible with other implementations, in particular the actual hardware. Specific mention of this is made in the ARM manuals.

Floating Point Coprocessor Register Transfer Instructions (CPRT)

Conversions, read/write status/control register instructions

FLT{cond}<S,D,E>{P,M,Z} Fn, Rd Convert integer to floating point FIX{cond}{P,M,Z} Rd, Fn Convert floating point to integer WFS{cond} Rd Write floating point status register RFS{cond} Rd Read floating point status register WFC{cond} Rd Write floating point control register RFC{cond} Rd Read floating point control register

FLT/FIX are fully implemented.

RFS/WFS are fully implemented.

RFC/WFC are fully implemented. RFC/WFC are supervisor only instructions, and presently check the CPU mode, and do an invalid instruction trap if not called from supervisor mode.

Compare instructions

CMF{cond} Fn, Fm Compare floating CMFE{cond} Fn, Fm Compare floating with exception CNF{cond} Fn, Fm Compare negated floating CNFE{cond} Fn, Fm Compare negated floating with exception

These are fully implemented.

Floating Point Coprocessor Data Instructions (CPDT)

Dyadic operations:

ADF{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - add SUF{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - subtract RSF{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - reverse subtract MUF{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - multiply DVF{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - divide RDV{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - reverse divide

These are fully implemented.

FML{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - fast multiply FDV{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - fast divide FRD{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - fast reverse divide

These are fully implemented as well. They use the same algorithm as the non-fast versions. Hence, in this implementation their performance is equivalent to the MUF/DVF/RDV instructions. This is acceptable according to the ARM manual. The manual notes these are defined only for single operands, on the actual FPA11 hardware they do not work for double or extended precision operands. The emulator currently does not check the requested permissions conditions, and performs the requested operation.

RMF{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - IEEE remainder

This is fully implemented.

Monadic operations:

MVF{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - move MNF{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - move negated

These are fully implemented.

ABS{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - absolute value SQT{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - square root RND{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - round

These are fully implemented.

URD{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - unnormalized round NRM{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - normalize

These are implemented. URD is implemented using the same code as the RND instruction. Since URD cannot return a unnormalized number, NRM becomes a NOP.

Library calls:

POW{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - power RPW{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - reverse power POL{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - polar angle (arctan2)

LOG{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - logarithm to base 10 LGN{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - logarithm to base e EXP{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - exponent SIN{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - sine COS{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - cosine TAN{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - tangent ASN{cond}<S|D|E>{P,M,Z}

Fd, <Fm,#value> - arcsine ACS{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - arccosine ATN{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - arctangent

These are not implemented. They are not currently issued by the compiler, and are handled by routines in libc. These are not implemented by the FPA11 hardware, but are handled by the floating point support code. They should be implemented in future versions.

Signalling:

Signals are implemented. However current ELF kernels produced by Rebel.com have a bug in them that prevents the module from generating a SIGFPE. This is caused by a failure to alias fp_current to the kernel variable current_set[0] correctly.

The kernel provided with this distribution (vmlinux-nwfpe-0.93) contains a fix for this problem and also incorporates the current version of the emulator directly. It is possible to run with no floating point module loaded with this kernel. It is provided as a demonstration of the technology and for those who want to do floating point work that depends on signals. It is not strictly necessary to use the module.

A module (either the one provided by Russell King, or the one in this distribution) can be loaded to replace the functionality of the emulator built into the kernel.

Notes

There seems to be a problem with exp(double) and our emulator. I haven't been able to track it down yet. This does not occur with the emulator supplied by Russell King.

I also found one oddity in the emulator. I don't think it is serious but will point it out. The ARM calling conventions require floating point registers f4-f7 to be preserved over a function call. The compiler quite often uses an stfe instruction to save f4 on the stack upon entry to a function, and an ldfe instruction to restore it before returning.

I was looking at some code, that calculated a double result, stored it in f4 then made a function call. Upon return from the function call the number in f4 had been converted to an extended value in the emulator.

This is a side effect of the stfe instruction. The double in f4 had to be converted to extended, then stored. If an lfm/sfm combination had been used, then no conversion would occur. This has performance considerations. The result from the function call and f4 were used in a multiplication. If the emulator sees a multiply of a double and extended, it promotes the double to extended, then does the multiply in extended precision.

This code will cause this problem:

double x, y, z; z = log(x)/log(y);

The result of log(x) (a double) will be calculated, returned in f0, then moved to f4 to preserve it over the log(y) call. The division will be done in extended precision, due to the stfe instruction used to save f4 in log(y).

TODO LIST

```
POW{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - power
RPW{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - reverse power
POL{cond}<S|D|E>{P,M,Z} Fd, Fn, <Fm,#value> - polar angle (arctan2)

LOG{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - logarithm to base 10
LGN{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - logarithm to base e
EXP{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - exponent
SIN{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - sine
COS{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - cosine
TAN{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - tangent
ASN{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - arcsine
ACS{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - arccosine
ATN{cond}<S|D|E>{P,M,Z} Fd, <Fm,#value> - arctangent
```

These are not implemented. They are not currently issued by the compiler, and are handled by routines in libc. These are not implemented by the FPA11 hardware, but are handled by the floating point support code. They should be implemented in future versions.

There are a couple of ways to approach the implementation of these. One method would be to use accurate table methods for these routines. I have a couple of papers by S. Gal from IBM's research labs in Haifa, Israel that seem to promise extreme accuracy (in the order of 99.8%) and reasonable speed. These methods are used in GLIBC for some of the transcendental functions.

Another approach, which I know little about is CORDIC. This stands for Coordinate Rotation Digital Computer, and is a method of computing transcendental functions using mostly shifts and adds and a few multiplications and divisions. The ARM excels at shifts and adds, so such a method could be promising, but requires more research to determine if it is feasible.

Rounding Methods

The IEEE standard defines 4 rounding modes. Round to nearest is the default, but rounding to + or - infinity or round to zero are also allowed. Many architectures allow the rounding mode to be specified by modifying bits in a control register. Not so with the ARM FPA11 architecture. To change the rounding mode one must specify it with each instruction.

This has made porting some benchmarks difficult. It is possible to introduce such a capability into the emulator. The FPCR contains bits describing the rounding mode. The emulator could be altered to examine a flag, which if set forced it to ignore the rounding mode in the instruction, and use the mode specified in the bits in the FPCR.

This would require a method of getting/setting the flag, and the bits in the FPCR. This requires a kernel call in ArmLinux, as WFC/RFC are supervisor only instructions. If anyone has any ideas or comments I would like to hear them.

NOTE:

> pulled out from some docs on ARM floating point, specifically for the Acorn FPE, but not limited to it:
>
> The floating point control register (FPCR) may only be present in some implementations: it is there to control the hardware in an implementation- specific manner, for example to disable the floating point system. The user mode of the ARM is not permitted to use this

register (since the right is reserved to alter it between implementations) and the WFC and RFC instructions will trap if tried in user mode.

Hence, the answer is yes, you could do this, but then you will run a high risk of becoming isolated if and when hardware FP emulation comes out

-- Russell.

2.17.7 TI Keystone Linux Overview

Introduction

Keystone range of SoCs are based on ARM Cortex-A15 MPCore Processors and c66x DSP cores. This document describes essential information required for users to run Linux on Keystone based EVMs from Texas Instruments.

Following SoCs & EVMs are currently supported:-

K2HK SoC and EVM

a.k.a Keystone 2 Hawking/Kepler SoC TCI6636K2H & TCI6636K2K: See documentation at

http://www.ti.com/product/tci6638k2k http://www.ti.com/product/tci6638k2h

EVM:
http://www.advantech.com/Support/TI-EVM/EVMK2HX_sd.aspx

K2E SoC and EVM

a.k.a Keystone 2 Edison SoC

K2E - 66AK2E05:

See documentation at

http://www.ti.com/product/66AK2E05/technicaldocuments

EVM:
https://www.einfochips.com/index.php/partnerships/texas-instruments/k2e-evm.html

K2L SoC and EVM

a.k.a Keystone 2 Lamarr SoC

K2L - TCI6630K2L:

See documentation at
http://www.ti.com/product/TCI6630K2L/technicaldocuments

EVM:
https://www.einfochips.com/index.php/partnerships/texas-instruments/k2l-evm.html

Configuration

All of the K2 SoCs/EVMs share a common defconfig, keystone_defconfig and same image is used to boot on individual EVMs. The platform configuration is specified through DTS. Following are the DTS used:

K2HK EVM:
k2hk-evm.dts

K2E EVM:
k2e-evm.dts

K2L EVM:
k2l-evm.dts

The device tree documentation for the keystone machines are located at

Documentation/devicetree/bindings/arm/keystone/keystone.txt

Document Author

Murali Karicheri <m-karicheri2@ti.com>

Copyright 2015 Texas Instruments

2.17.8 Texas Instruments Keystone Navigator Queue Management SubSystem driver

Driver source code path
drivers/soc/ti/knav_qmss.c drivers/soc/ti/knav_qmss_acc.c

The QMSS (Queue Manager Sub System) found on Keystone SOCs is one of the main hardware sub system which forms the backbone of the Keystone multi-core Navigator. QMSS consist of queue managers, packed-data structure processors(PDSP), linking RAM, descriptor pools and infrastructure Packet DMA. The Queue Manager is a hardware module that is responsible for accelerating management of the packet queues. Packets are queued/de-queued by writing or reading descriptor address to a particular memory mapped location. The PDSPs perform QMSS related functions like accumulation, QoS, or event management. Linking RAM registers are used to link the descriptors which are stored in descriptor RAM. Descriptor RAM is configurable as internal or external memory. The QMSS driver manages the PDSP setups, linking RAM regions, queue pool management (allocation, push, pop and notify) and descriptor pool management.

knav qmss driver provides a set of APIs to drivers to open/close qmss queues, allocate descriptor pools, map the descriptors, push/pop to queues etc. For details of the available APIs, please refers to include/linux/soc/ti/knav_qmss.h

DT documentation is available at Documentation/devicetree/bindings/soc/ti/keystone-navigator-qmss.txt

Accumulator QMSS queues using PDSP firmware

The QMSS PDSP firmware support accumulator channel that can monitor a single queue or multiple contiguous queues. drivers/soc/ti/knav_qmss_acc.c is the driver that interface with the accumulator PDSP. This configures accumulator channels defined in DTS (example in DT documentation) to monitor 1 or 32 queues per channel. More description on the firmware is available in CPPI/QMSS Low Level Driver document (docs/CPPI_QMSS_LLD_SDS.pdf) at

> git://git.ti.com/keystone-rtos/qmss-lld.git

k2_qmss_pdsp_acc48_k2_le_1_0_0_9.bin firmware supports upto 48 accumulator channels. This firmware is available under ti-keystone folder of firmware.git at

> git://git.kernel.org/pub/scm/linux/kernel/git/firmware/linux-firmware.git

To use copy the firmware image to lib/firmware folder of the initramfs or ubifs file system and provide a sym link to k2_qmss_pdsp_acc48_k2_le_1_0_0_9.bin in the file system and boot up the kernel. User would see

> "firmware file ks2_qmss_pdsp_acc48.bin downloaded for PDSP"

in the boot up log if loading of firmware to PDSP is successful.

Use of accumulated queues requires the firmware image to be present in the file system. The driver doesn't acc queues to the supported queue range if PDSP is not running in the SoC. The API call fails if there is a queue open request to an acc queue and PDSP is not running. So make sure to copy firmware to file system before using these queue types.

2.17.9 TI OMAP

OMAP history

This file contains documentation for running mainline kernel on omaps.

KERNEL	NEW DEPENDENCIES
v4.3+	Update is needed for custom .config files to make sure CONFIG_REGULATOR_PBIAS is enabled for MMC1 to work properly.
v4.18+	Update is needed for custom .config files to make sure CONFIG_MMC_SDHCI_OMAP is enabled for all MMC instances to work in DRA7 and K2G based boards.

The OMAP PM interface

This document describes the temporary OMAP PM interface. Driver authors use these functions to communicate minimum latency or throughput constraints to the kernel power management code. Over time, the intention is to merge features from the OMAP PM interface into the Linux PM QoS code.

Drivers need to express PM parameters which:

- support the range of power management parameters present in the TI SRF;

- separate the drivers from the underlying PM parameter implementation, whether it is the TI SRF or Linux PM QoS or Linux latency framework or something else;
- specify PM parameters in terms of fundamental units, such as latency and throughput, rather than units which are specific to OMAP or to particular OMAP variants;
- allow drivers which are shared with other architectures (e.g., DaVinci) to add these constraints in a way which won't affect non-OMAP systems,
- can be implemented immediately with minimal disruption of other architectures.

This document proposes the OMAP PM interface, including the following five power management functions for driver code:

1. Set the maximum MPU wakeup latency:

   ```
   (*pdata->set_max_mpu_wakeup_lat)(struct device *dev, unsigned long t)
   ```

2. Set the maximum device wakeup latency:

   ```
   (*pdata->set_max_dev_wakeup_lat)(struct device *dev, unsigned long t)
   ```

3. Set the maximum system DMA transfer start latency (CORE pwrdm):

   ```
   (*pdata->set_max_sdma_lat)(struct device *dev, long t)
   ```

4. Set the minimum bus throughput needed by a device:

   ```
   (*pdata->set_min_bus_tput)(struct device *dev, u8 agent_id, unsigned long␣
   ↪r)
   ```

5. Return the number of times the device has lost context:

   ```
   (*pdata->get_dev_context_loss_count)(struct device *dev)
   ```

Further documentation for all OMAP PM interface functions can be found in arch/arm/plat-omap/include/mach/omap-pm.h.

The OMAP PM layer is intended to be temporary

The intention is that eventually the Linux PM QoS layer should support the range of power management features present in OMAP3. As this happens, existing drivers using the OMAP PM interface can be modified to use the Linux PM QoS code; and the OMAP PM interface can disappear.

Driver usage of the OMAP PM functions

As the 'pdata' in the above examples indicates, these functions are exposed to drivers through function pointers in driver .platform_data structures. The function pointers are initialized by the *board-*.c* files to point to the corresponding OMAP PM functions:

- set_max_dev_wakeup_lat will point to omap_pm_set_max_dev_wakeup_lat(), etc. Other architectures which do not support these functions should leave these function pointers set to NULL. Drivers should use the following idiom:

  ```
  if (pdata->set_max_dev_wakeup_lat)
      (*pdata->set_max_dev_wakeup_lat)(dev, t);
  ```

The most common usage of these functions will probably be to specify the maximum time from when an interrupt occurs, to when the device becomes accessible. To accomplish this, driver writers should use the set_max_mpu_wakeup_lat() function to constrain the MPU wakeup latency, and the set_max_dev_wakeup_lat() function to constrain the device wakeup latency (from clk_enable() to accessibility). For example:

```
/* Limit MPU wakeup latency */
if (pdata->set_max_mpu_wakeup_lat)
    (*pdata->set_max_mpu_wakeup_lat)(dev, tc);

/* Limit device powerdomain wakeup latency */
if (pdata->set_max_dev_wakeup_lat)
    (*pdata->set_max_dev_wakeup_lat)(dev, td);

/* total wakeup latency in this example: (tc + td) */
```

The PM parameters can be overwritten by calling the function again with the new value. The settings can be removed by calling the function with a t argument of -1 (except in the case of set_max_bus_tput(), which should be called with an r argument of 0).

The fifth function above, omap_pm_get_dev_context_loss_count(), is intended as an optimization to allow drivers to determine whether the device has lost its internal context. If context has been lost, the driver must restore its internal context before proceeding.

Other specialized interface functions

The five functions listed above are intended to be usable by any device driver. DSPBridge and CPUFreq have a few special requirements. DSPBridge expresses target DSP performance levels in terms of OPP IDs. CPUFreq expresses target MPU performance levels in terms of MPU frequency. The OMAP PM interface contains functions for these specialized cases to convert that input information (OPPs/MPU frequency) into the form that the underlying power management implementation needs:

6. *(*pdata->dsp_get_opp_table)(void)*
7. *(*pdata->dsp_set_min_opp)(u8 opp_id)*
8. *(*pdata->dsp_get_opp)(void)*
9. *(*pdata->cpu_get_freq_table)(void)*

10. *(*pdata->cpu_set_freq)(unsigned long f)*
11. *(*pdata->cpu_get_freq)(void)*

Customizing OPP for platform

Defining CONFIG_PM should enable OPP layer for the silicon and the registration of OPP table should take place automatically. However, in special cases, the default OPP table may need to be tweaked, for e.g.:

- enable default OPPs which are disabled by default, but which could be enabled on a platform
- Disable an unsupported OPP on the platform
- Define and add a custom opp table entry in these cases, the board file needs to do additional steps as follows:

arch/arm/mach-omapx/board-xyz.c:

```
#include "pm.h"
....
static void __init omap_xyz_init_irq(void)
{
        ....
        /* Initialize the default table */
        omapx_opp_init();
        /* Do customization to the defaults */
        ....
}
```

NOTE:
omapx_opp_init will be omap3_opp_init or as required based on the omap family.

OMAP2/3 Display Subsystem

This is an almost total rewrite of the OMAP FB driver in drivers/video/omap (let's call it DSS1). The main differences between DSS1 and DSS2 are DSI, TV-out and multiple display support, but there are lots of small improvements also.

The DSS2 driver (omapdss module) is in arch/arm/plat-omap/dss/, and the FB, panel and controller drivers are in drivers/video/omap2/. DSS1 and DSS2 live currently side by side, you can choose which one to use.

Features

Working and tested features include:

- MIPI DPI (parallel) output
- MIPI DSI output in command mode
- MIPI DBI (RFBI) output
- SDI output
- TV output
- All pieces can be compiled as a module or inside kernel
- Use DISPC to update any of the outputs
- Use CPU to update RFBI or DSI output
- OMAP DISPC planes
- RGB16, RGB24 packed, RGB24 unpacked
- YUV2, UYVY
- Scaling
- Adjusting DSS FCK to find a good pixel clock
- Use DSI DPLL to create DSS FCK

Tested boards include: - OMAP3 SDP board - Beagle board - N810

omapdss driver

The DSS driver does not itself have any support for Linux framebuffer, V4L or such like the current ones, but it has an internal kernel API that upper level drivers can use.

The DSS driver models OMAP's overlays, overlay managers and displays in a flexible way to enable non-common multi-display configuration. In addition to modelling the hardware overlays, omapdss supports virtual overlays and overlay managers. These can be used when updating a display with CPU or system DMA.

omapdss driver support for audio

There exist several display technologies and standards that support audio as well. Hence, it is relevant to update the DSS device driver to provide an audio interface that may be used by an audio driver or any other driver interested in the functionality.

The audio_enable function is intended to prepare the relevant IP for playback (e.g., enabling an audio FIFO, taking in/out of reset some IP, enabling companion chips, etc). It is intended to be called before audio_start. The audio_disable function performs the reverse operation and is intended to be called after audio_stop.

While a given DSS device driver may support audio, it is possible that for certain configurations audio is not supported (e.g., an HDMI display using a VESA video timing). The audio_supported function is intended to query whether the current configuration of the display supports audio.

The audio_config function is intended to configure all the relevant audio parameters of the display. In order to make the function independent of any specific DSS device driver, a struct omap_dss_audio is defined. Its purpose is to contain all the required parameters for audio configuration. At the moment, such structure contains pointers to IEC-60958 channel status word and CEA-861 audio infoframe structures. This should be enough to support HDMI and DisplayPort, as both are based on CEA-861 and IEC-60958.

The audio_enable/disable, audio_config and audio_supported functions could be implemented as functions that may sleep. Hence, they should not be called while holding a spinlock or a readlock.

The audio_start/audio_stop function is intended to effectively start/stop audio playback after the configuration has taken place. These functions are designed to be used in an atomic context. Hence, audio_start should return quickly and be called only after all the needed resources for audio playback (audio FIFOs, DMA channels, companion chips, etc) have been enabled to begin data transfers. audio_stop is designed to only stop the audio transfers. The resources used for playback are released using audio_disable.

The enum omap_dss_audio_state may be used to help the implementations of the interface to keep track of the audio state. The initial state is _DISABLED; then, the state transitions to _CONFIGURED, and then, when it is ready to play audio, to _ENABLED. The state _PLAYING is used when the audio is being rendered.

Panel and controller drivers

The drivers implement panel or controller specific functionality and are not usually visible to users except through omapfb driver. They register themselves to the DSS driver.

omapfb driver

The omapfb driver implements arbitrary number of standard linux framebuffers. These framebuffers can be routed flexibly to any overlays, thus allowing very dynamic display architecture.

The driver exports some omapfb specific ioctls, which are compatible with the ioctls in the old driver.

The rest of the non standard features are exported via sysfs. Whether the final implementation will use sysfs, or ioctls, is still open.

V4L2 drivers

V4L2 is being implemented in TI.

From omapdss point of view the V4L2 drivers should be similar to framebuffer driver.

Architecture

Some clarification what the different components do:

- Framebuffer is a memory area inside OMAP's SRAM/SDRAM that contains the pixel data for the image. Framebuffer has width and height and color depth.
- Overlay defines where the pixels are read from and where they go on the screen. The overlay may be smaller than framebuffer, thus displaying only part of the framebuffer. The position of the overlay may be changed if the overlay is smaller than the display.
- Overlay manager combines the overlays in to one image and feeds them to display.
- Display is the actual physical display device.

A framebuffer can be connected to multiple overlays to show the same pixel data on all of the overlays. Note that in this case the overlay input sizes must be the same, but, in case of video overlays, the output size can be different. Any framebuffer can be connected to any overlay.

An overlay can be connected to one overlay manager. Also DISPC overlays can be connected only to DISPC overlay managers, and virtual overlays can be only connected to virtual overlays.

An overlay manager can be connected to one display. There are certain restrictions which kinds of displays an overlay manager can be connected:

- DISPC TV overlay manager can be only connected to TV display.
- Virtual overlay managers can only be connected to DBI or DSI displays.
- DISPC LCD overlay manager can be connected to all displays, except TV display.

Sysfs

The sysfs interface is mainly used for testing. I don't think sysfs interface is the best for this in the final version, but I don't quite know what would be the best interfaces for these things.

The sysfs interface is divided to two parts: DSS and FB.

/sys/class/graphics/fb? directory: mirror 0=off, 1=on rotate Rotation 0-3 for 0, 90, 180, 270 degrees rotate_type 0 = DMA rotation, 1 = VRFB rotation overlays List of overlay numbers to which framebuffer pixels go phys_addr Physical address of the framebuffer virt_addr Virtual address of the framebuffer size Size of the framebuffer

/sys/devices/platform/omapdss/overlay? directory: enabled 0=off, 1=on input_size width,height (ie. the framebuffer size) manager Destination overlay manager name name output_size width,height position x,y screen_width width global_alpha global alpha 0-255 0=transparent 255=opaque

/sys/devices/platform/omapdss/manager? directory: display Destination display name alpha_blending_enabled 0=off, 1=on trans_key_enabled 0=off, 1=on trans_key_type gfx-destination, video-source trans_key_value transparency color key (RGB24) default_color default background color (RGB24)

/sys/devices/platform/omapdss/display? directory:

ctrl_name	Controller name
mirror	0=off, 1=on
update_mode	0=off, 1=auto, 2=manual
enabled	0=off, 1=on
name	
rotate	Rotation 0-3 for 0, 90, 180, 270 degrees
timings	Display timings (pixclock,xres/hfp/hbp/hsw,yres/vfp/vbp/vsw) When writing, two special timings are accepted for tv-out: "pal" and "ntsc"
panel_name	
tear_elim	Tearing elimination 0=off, 1=on
output_type	Output type (video encoder only): "composite" or "svideo"

There are also some debugfs files at <debugfs>/omapdss/ which show information about clocks and registers.

Examples

The following definitions have been made for the examples below:

```
ovl0=/sys/devices/platform/omapdss/overlay0
ovl1=/sys/devices/platform/omapdss/overlay1
ovl2=/sys/devices/platform/omapdss/overlay2

mgr0=/sys/devices/platform/omapdss/manager0
mgr1=/sys/devices/platform/omapdss/manager1

lcd=/sys/devices/platform/omapdss/display0
dvi=/sys/devices/platform/omapdss/display1
tv=/sys/devices/platform/omapdss/display2

fb0=/sys/class/graphics/fb0
fb1=/sys/class/graphics/fb1
fb2=/sys/class/graphics/fb2
```

Default setup on OMAP3 SDP

Here's the default setup on OMAP3 SDP board. All planes go to LCD. DVI and TV-out are not in use. The columns from left to right are: framebuffers, overlays, overlay managers, displays. Framebuffers are handled by omapfb, and the rest by the DSS:

```
FB0 --- GFX  -\            DVI
FB1 --- VID1 --+- LCD ---- LCD
FB2 --- VID2 -/   TV ----- TV
```

Example: Switch from LCD to DVI

```
w=`cat $dvi/timings | cut -d "," -f 2 | cut -d "/" -f 1`
h=`cat $dvi/timings | cut -d "," -f 3 | cut -d "/" -f 1`

echo "0" > $lcd/enabled
echo "" > $mgr0/display
fbset -fb /dev/fb0 -xres $w -yres $h -vxres $w -vyres $h
# at this point you have to switch the dvi/lcd dip-switch from the omap board
echo "dvi" > $mgr0/display
echo "1" > $dvi/enabled
```

After this the configuration looks like::

```
FB0 --- GFX  -\         -- DVI
FB1 --- VID1 --+- LCD -/   LCD
FB2 --- VID2 -/   TV ----- TV
```

Example: Clone GFX overlay to LCD and TV

```
w=`cat $tv/timings | cut -d "," -f 2 | cut -d "/" -f 1`
h=`cat $tv/timings | cut -d "," -f 3 | cut -d "/" -f 1`

echo "0" > $ovl0/enabled
echo "0" > $ovl1/enabled

echo "" > $fb1/overlays
echo "0,1" > $fb0/overlays

echo "$w,$h" > $ovl1/output_size
echo "tv" > $ovl1/manager

echo "1" > $ovl0/enabled
echo "1" > $ovl1/enabled

echo "1" > $tv/enabled
```

After this the configuration looks like (only relevant parts shown):

```
FB0 +-- GFX  ---- LCD ---- LCD
\- VID1 ---- TV  ---- TV
```

Misc notes

OMAP FB allocates the framebuffer memory using the standard dma allocator. You can enable Contiguous Memory Allocator (CONFIG_CMA) to improve the dma allocator, and if CMA is enabled, you use "cma=" kernel parameter to increase the global memory area for CMA.

Using DSI DPLL to generate pixel clock it is possible produce the pixel clock of 86.5MHz (max possible), and with that you get 1280x1024@57 output from DVI.

Rotation and mirroring currently only supports RGB565 and RGB8888 modes. VRFB does not support mirroring.

VRFB rotation requires much more memory than non-rotated framebuffer, so you probably need to increase your vram setting before using VRFB rotation. Also, many applications may not work with VRFB if they do not pay attention to all framebuffer parameters.

Kernel boot arguments

omapfb.mode=<display>:<mode>[,...]

- Default video mode for specified displays. For example, "dvi:800x400MR-24@60". See drivers/video/modedb.c. There are also two special modes: "pal" and "ntsc" that can be used to tv out.

omapfb.vram=<fbnum>:<size>[@<physaddr>][,...]

- VRAM allocated for a framebuffer. Normally omapfb allocates vram depending on the display size. With this you can manually allocate more or define the physical address of each framebuffer. For example, "1:4M" to allocate 4M for fb1.

omapfb.debug=<y|n>

- Enable debug printing. You have to have OMAPFB debug support enabled in kernel config.

omapfb.test=<y|n>

- Draw test pattern to framebuffer whenever framebuffer settings change. You need to have OMAPFB debug support enabled in kernel config.

omapfb.vrfb=<y|n>

- Use VRFB rotation for all framebuffers.

omapfb.rotate=<angle>

- Default rotation applied to all framebuffers. 0 - 0 degree rotation 1 - 90 degree rotation 2 - 180 degree rotation 3 - 270 degree rotation

omapfb.mirror=<y|n>

- Default mirror for all framebuffers. Only works with DMA rotation.

omapdss.def_disp=<display>

- Name of default display, to which all overlays will be connected. Common examples are "lcd" or "tv".

omapdss.debug=<y|n>

- Enable debug printing. You have to have DSS debug support enabled in kernel config.

TODO

DSS locking

Error checking

- Lots of checks are missing or implemented just as BUG()

System DMA update for DSI

- Can be used for RGB16 and RGB24P modes. Probably not for RGB24U (how to skip the empty byte?)

OMAP1 support

- Not sure if needed

2.17.10 MFP Configuration for PXA2xx/PXA3xx Processors

Eric Miao <eric.miao@marvell.com>

MFP stands for Multi-Function Pin, which is the pin-mux logic on PXA3xx and later PXA series processors. This document describes the existing MFP API, and how board/platform driver authors could make use of it.

Basic Concept

Unlike the GPIO alternate function settings on PXA25x and PXA27x, a new MFP mechanism is introduced from PXA3xx to completely move the pin-mux functions out of the GPIO controller. In addition to pin-mux configurations, the MFP also controls the low power state, driving strength, pull-up/down and event detection of each pin. Below is a diagram of internal connections between the MFP logic and the remaining SoC peripherals:

```
+--------+
|        |--(GPIO19)--+
|  GPIO  |            |
|        |--(GPIO...) |
+--------+            |
                      |       +---------+
+--------+            +------>|         |
|  PWM2  |--(PWM_OUT)-------->|   MFP   |
+--------+            +------>|         |-------> to external PAD
                      | +---->|         |
+--------+            | | +-->|         |
|  SSP2  |---(TXD)----+ | |   +---------+
+--------+              | |
                        | |
+--------+              | |
| Keypad |--(MKOUT4)----+ |
+--------+                |
                          |
+--------+                |
|  UART2 |---(TXD)--------+
+--------+
```

NOTE: the external pad is named as MFP_PIN_GPIO19, it doesn't necessarily mean it's dedicated for GPIO19, only as a hint that internally this pin can be routed from GPIO19 of the GPIO controller.

To better understand the change from PXA25x/PXA27x GPIO alternate function to this new MFP mechanism, here are several key points:

1. GPIO controller on PXA3xx is now a dedicated controller, same as other internal controllers like PWM, SSP and UART, with 128 internal signals which can be routed to external through one or more MFPs (e.g. GPIO<0> can be routed through either MFP_PIN_GPIO0 as well as MFP_PIN_GPIO0_2, see arch/arm/mach-pxa/mfp-pxa300.h)
2. Alternate function configuration is removed from this GPIO controller, the remaining functions are pure GPIO-specific, i.e.
   - GPIO signal level control
   - GPIO direction control
   - GPIO level change detection
3. Low power state for each pin is now controlled by MFP, this means the PGSRx registers on PXA2xx are now useless on PXA3xx
4. Wakeup detection is now controlled by MFP, PWER does not control the wakeup from GPIO(s) any more, depending on the sleeping state, ADxER (as defined in pxa3xx-regs.h) controls the wakeup from MFP

NOTE: with such a clear separation of MFP and GPIO, by GPIO<xx> we normally mean it is a GPIO signal, and by MFP<xxx> or pin xxx, we mean a physical pad (or ball).

MFP API Usage

For board code writers, here are some guidelines:

1. include ONE of the following header files in your <board>.c:
   - #include "mfp-pxa25x.h"
   - #include "mfp-pxa27x.h"
   - #include "mfp-pxa300.h"
   - #include "mfp-pxa320.h"
   - #include "mfp-pxa930.h"

   NOTE: only one file in your <board>.c, depending on the processors used, because pin configuration definitions may conflict in these file (i.e. same name, different meaning and settings on different processors). E.g. for zylonite platform, which support both PXA300/PXA310 and PXA320, two separate files are introduced: zylonite_pxa300.c and zylonite_pxa320.c (in addition to handle MFP configuration differences, they also handle the other differences between the two combinations).

NOTE: PXA300 and PXA310 are almost identical in pin configurations (with PXA310 supporting some additional ones), thus the difference is actually covered in a single mfp-pxa300.h.

2. prepare an array for the initial pin configurations, e.g.:

```
static unsigned long mainstone_pin_config[] __initdata = {
   /* Chip Select */
   GPIO15_nCS_1,

   /* LCD - 16bpp Active TFT */
   GPIOxx_TFT_LCD_16BPP,
   GPIO16_PWM0_OUT,        /* Backlight */

   /* MMC */
   GPIO32_MMC_CLK,
   GPIO112_MMC_CMD,
   GPIO92_MMC_DAT_0,
   GPIO109_MMC_DAT_1,
   GPIO110_MMC_DAT_2,
   GPIO111_MMC_DAT_3,

   ...

   /* GPIO */
   GPIO1_GPIO | WAKEUP_ON_EDGE_BOTH,
};
```

a) once the pin configurations are passed to pxa{2xx,3xx}_mfp_config(), and written to the actual registers, they are useless and may discard, adding '__initdata' will help save some additional bytes here.

b) when there is only one possible pin configurations for a component, some simplified definitions can be used, e.g. GPIOxx_TFT_LCD_16BPP on PXA25x and PXA27x processors

c) if by board design, a pin can be configured to wake up the system from low power state, it can be 'OR'ed with any of:

> WAKEUP_ON_EDGE_BOTH WAKEUP_ON_EDGE_RISE WAKEUP_ON_EDGE_FALL WAKEUP_ON_LEVEL_HIGH - specifically for enabling of keypad GPIOs,

to indicate that this pin has the capability of wake-up the system, and on which edge(s). This, however, doesn't necessarily mean the pin _will_ wakeup the system, it will only when set_irq_wake() is invoked with the corresponding GPIO IRQ (GPIO_IRQ(xx) or gpio_to_irq()) and eventually calls gpio_set_wake() for the actual register setting.

d) although PXA3xx MFP supports edge detection on each pin, the internal logic will only wakeup the system when those specific bits in ADxER registers are set, which can be well mapped to the corresponding peripheral, thus set_irq_wake() can be called with the peripheral IRQ to enable the wakeup.

MFP on PXA3xx

Every external I/O pad on PXA3xx (excluding those for special purpose) has one MFP logic associated, and is controlled by one MFP register (MFPR).

The MFPR has the following bit definitions (for PXA300/PXA310/PXA320):

```
31                         16 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1  0
 +-------------------------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |         RESERVED        |PS|PU|PD|  DRIVE |SS|SD|SO|EC|EF|ER|--| AF_SEL |
 +-------------------------+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+

 Bit 3:   RESERVED
 Bit 4:   EDGE_RISE_EN - enable detection of rising edge on this pin
 Bit 5:   EDGE_FALL_EN - enable detection of falling edge on this pin
 Bit 6:   EDGE_CLEAR   - disable edge detection on this pin
 Bit 7:   SLEEP_OE_N   - enable outputs during low power modes
 Bit 8:   SLEEP_DATA   - output data on the pin during low power modes
 Bit 9:   SLEEP_SEL    - selection control for low power modes signals
 Bit 13:  PULLDOWN_EN  - enable the internal pull-down resistor on this pin
 Bit 14:  PULLUP_EN    - enable the internal pull-up resistor on this pin
 Bit 15:  PULL_SEL     - pull state controlled by selected alternate function
                         (0) or by PULL{UP,DOWN}_EN bits (1)

 Bit 0 - 2: AF_SEL - alternate function selection, 8 possibilities, from 0-7
 Bit 10-12: DRIVE  - drive strength and slew rate
                       0b000 - fast 1mA
                       0b001 - fast 2mA
                       0b002 - fast 3mA
                       0b003 - fast 4mA
                       0b004 - slow 6mA
                       0b005 - fast 6mA
                       0b006 - slow 10mA
                       0b007 - fast 10mA
```

MFP Design for PXA2xx/PXA3xx

Due to the difference of pin-mux handling between PXA2xx and PXA3xx, a unified MFP API is introduced to cover both series of processors.

The basic idea of this design is to introduce definitions for all possible pin configurations, these definitions are processor and platform independent, and the actual API invoked to convert these definitions into register settings and make them effective there-after.

Files Involved

- arch/arm/mach-pxa/include/mach/mfp.h

for

1. Unified pin definitions - enum constants for all configurable pins
2. processor-neutral bit definitions for a possible MFP configuration

- arch/arm/mach-pxa/mfp-pxa3xx.h

for PXA3xx specific MFPR register bit definitions and PXA3xx common pin configurations

- arch/arm/mach-pxa/mfp-pxa2xx.h

for PXA2xx specific definitions and PXA25x/PXA27x common pin configurations

- arch/arm/mach-pxa/mfp-pxa25x.h arch/arm/mach-pxa/mfp-pxa27x.h arch/arm/mach-pxa/mfp-pxa300.h arch/arm/mach-pxa/mfp-pxa320.h arch/arm/mach-pxa/mfp-pxa930.h

for processor specific definitions

- arch/arm/mach-pxa/mfp-pxa3xx.c
- arch/arm/mach-pxa/mfp-pxa2xx.c

for implementation of the pin configuration to take effect for the actual processor.

Pin Configuration

The following comments are copied from mfp.h (see the actual source code for most updated info):

```
/*
 * a possible MFP configuration is represented by a 32-bit integer
 *
 * bit  0.. 9 - MFP Pin Number (1024 Pins Maximum)
 * bit 10..12 - Alternate Function Selection
 * bit 13..15 - Drive Strength
 * bit 16..18 - Low Power Mode State
 * bit 19..20 - Low Power Mode Edge Detection
 * bit 21..22 - Run Mode Pull State
 *
 * to facilitate the definition, the following macros are provided
 *
 * MFP_CFG_DEFAULT - default MFP configuration value, with
 *             alternate function = 0,
 *             drive strength = fast 3mA (MFP_DS03X)
 *             low power mode = default
 *             edge detection = none
 *
 * MFP_CFG  - default MFPR value with alternate function
 * MFP_CFG_DRV      - default MFPR value with alternate function and
```

```
  *             pin drive strength
  * MFP_CFG_LPM      - default MFPR value with alternate function and
  *             low power mode
  * MFP_CFG_X        - default MFPR value with alternate function,
  *             pin drive strength and low power mode
  */

Examples of pin configurations are::

  #define GPIO94_SSP3_RXD            MFP_CFG_X(GPIO94, AF1, DS08X,␣
↪FLOAT)

which reads GPIO94 can be configured as SSP3_RXD, with alternate␣
↪function
selection of 1, driving strength of 0b101, and a float state in low␣
↪power
modes.

NOTE: this is the default setting of this pin being configured as SSP3_
↪RXD
which can be modified a bit in board code, though it is not␣
↪recommended to
do so, simply because this default setting is usually carefully␣
↪encoded,
and is supposed to work in most cases.
```

Register Settings

Register settings on PXA3xx for a pin configuration is actually very straight-forward, most bits can be converted directly into MFPR value in a easier way. Two sets of MFPR values are calculated: the run-time ones and the low power mode ones, to allow different settings.

The conversion from a generic pin configuration to the actual register settings on PXA2xx is a bit complicated: many registers are involved, including GAFRx, GPDRx, PGSRx, PWER, PKWR, PFER and PRER. Please see mfp-pxa2xx.c for how the conversion is made.

2.17.11 Intel StrongARM 1100

The Intel Assabet (SA-1110 evaluation) board

Please see: http://developer.intel.com

Also some notes from John G Dorsey <jd5q@andrew.cmu.edu>: http://www.cs.cmu.edu/~wearable/software/assabet.html

Building the kernel

To build the kernel with current defaults:

```
make assabet_defconfig
make oldconfig
make zImage
```

The resulting kernel image should be available in linux/arch/arm/boot/zImage.

Installing a bootloader

A couple of bootloaders able to boot Linux on Assabet are available:

BLOB (http://www.lartmaker.nl/lartware/blob/)

> BLOB is a bootloader used within the LART project. Some contributed patches were merged into BLOB to add support for Assabet.

Compaq's Bootldr + John Dorsey's patch for Assabet support (http://www.handhelds.org/Compaq/bootldr.html) (http://www.wearablegroup.org/software/bootldr/)

> Bootldr is the bootloader developed by Compaq for the iPAQ Pocket PC. John Dorsey has produced add-on patches to add support for Assabet and the JFFS filesystem.

RedBoot (http://sources.redhat.com/redboot/)

> RedBoot is a bootloader developed by Red Hat based on the eCos RTOS hardware abstraction layer. It supports Assabet amongst many other hardware platforms.

RedBoot is currently the recommended choice since it's the only one to have networking support, and is the most actively maintained.

Brief examples on how to boot Linux with RedBoot are shown below. But first you need to have RedBoot installed in your flash memory. A known to work precompiled RedBoot binary is available from the following location:

- ftp://ftp.netwinder.org/users/n/nico/
- ftp://ftp.arm.linux.org.uk/pub/linux/arm/people/nico/
- ftp://ftp.handhelds.org/pub/linux/arm/sa-1100-patches/

Look for redboot-assabet*.tgz. Some installation infos are provided in redboot-assabet*.txt.

Initial RedBoot configuration

The commands used here are explained in The RedBoot User's Guide available on-line at http://sources.redhat.com/ecos/docs.html. Please refer to it for explanations.

If you have a CF network card (my Assabet kit contained a CF+ LP-E from Socket Communications Inc.), you should strongly consider using it for TFTP file transfers. You must insert it before RedBoot runs since it can't detect it dynamically.

To initialize the flash directory:

```
fis init -f
```

To initialize the non-volatile settings, like whether you want to use BOOTP or a static IP address, etc, use this command:

```
fconfig -i
```

Writing a kernel image into flash

First, the kernel image must be loaded into RAM. If you have the zImage file available on a TFTP server:

```
load zImage -r -b 0x100000
```

If you rather want to use Y-Modem upload over the serial port:

```
load -m ymodem -r -b 0x100000
```

To write it to flash:

```
fis create "Linux kernel" -b 0x100000 -l 0xc0000
```

Booting the kernel

The kernel still requires a filesystem to boot. A ramdisk image can be loaded as follows:

```
load ramdisk_image.gz -r -b 0x800000
```

Again, Y-Modem upload can be used instead of TFTP by replacing the file name by '-y ymodem'.

Now the kernel can be retrieved from flash like this:

```
fis load "Linux kernel"
```

or loaded as described previously. To boot the kernel:

```
exec -b 0x100000 -l 0xc0000
```

The ramdisk image could be stored into flash as well, but there are better solutions for on-flash filesystems as mentioned below.

Using JFFS2

Using JFFS2 (the Second Journalling Flash File System) is probably the most convenient way to store a writable filesystem into flash. JFFS2 is used in conjunction with the MTD layer which is responsible for low-level flash management. More information on the Linux MTD can be found on-line at: http://www.linux-mtd.infradead.org/. A JFFS howto with some infos about creating JFFS/JFFS2 images is available from the same site.

For instance, a sample JFFS2 image can be retrieved from the same FTP sites mentioned below for the precompiled RedBoot image.

To load this file:

```
load sample_img.jffs2 -r -b 0x100000
```

The result should look like:

```
RedBoot> load sample_img.jffs2 -r -b 0x100000
Raw file loaded 0x00100000-0x00377424
```

Now we must know the size of the unallocated flash:

```
fis free
```

Result:

```
RedBoot> fis free
  0x500E0000 .. 0x503C0000
```

The values above may be different depending on the size of the filesystem and the type of flash. See their usage below as an example and take care of substituting yours appropriately.

We must determine some values:

```
size of unallocated flash:      0x503c0000 - 0x500e0000 = 0x2e0000
size of the filesystem image:   0x00377424 - 0x00100000 = 0x277424
```

We want to fit the filesystem image of course, but we also want to give it all the remaining flash space as well. To write it:

```
fis unlock -f 0x500E0000 -l 0x2e0000
fis erase -f 0x500E0000 -l 0x2e0000
fis write -b 0x100000 -l 0x277424 -f 0x500E0000
fis create "JFFS2" -n -f 0x500E0000 -l 0x2e0000
```

Now the filesystem is associated to a MTD "partition" once Linux has discovered what they are in the boot process. From Redboot, the 'fis list' command displays them:

```
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
RedBoot           0x50000000  0x50000000  0x00020000  0x00000000
RedBoot config    0x503C0000  0x503C0000  0x00020000  0x00000000
FIS directory     0x503E0000  0x503E0000  0x00020000  0x00000000
```

```
Linux kernel      0x50020000  0x00100000  0x000C0000  0x00000000
JFFS2             0x500E0000  0x500E0000  0x002E0000  0x00000000
```

However Linux should display something like:

```
SA1100 flash: probing 32-bit flash bus
SA1100 flash: Found 2 x16 devices at 0x0 in 32-bit mode
Using RedBoot partition definition
Creating 5 MTD partitions on "SA1100 flash":
0x00000000-0x00020000 : "RedBoot"
0x00020000-0x000e0000 : "Linux kernel"
0x000e0000-0x003c0000 : "JFFS2"
0x003c0000-0x003e0000 : "RedBoot config"
0x003e0000-0x00400000 : "FIS directory"
```

What's important here is the position of the partition we are interested in, which is the third one. Within Linux, this correspond to /dev/mtdblock2. Therefore to boot Linux with the kernel and its root filesystem in flash, we need this RedBoot command:

```
fis load "Linux kernel"
exec -b 0x100000 -l 0xc0000 -c "root=/dev/mtdblock2"
```

Of course other filesystems than JFFS might be used, like cramfs for example. You might want to boot with a root filesystem over NFS, etc. It is also possible, and sometimes more convenient, to flash a filesystem directly from within Linux while booted from a ramdisk or NFS. The Linux MTD repository has many tools to deal with flash memory as well, to erase it for example. JFFS2 can then be mounted directly on a freshly erased partition and files can be copied over directly. Etc...

RedBoot scripting

All the commands above aren't so useful if they have to be typed in every time the Assabet is rebooted. Therefore it's possible to automate the boot process using RedBoot's scripting capability.

For example, I use this to boot Linux with both the kernel and the ramdisk images retrieved from a TFTP server on the network:

```
RedBoot> fconfig
Run script at boot: false true
Boot script:
Enter script, terminate with empty line
>> load zImage -r -b 0x100000
>> load ramdisk_ks.gz -r -b 0x800000
>> exec -b 0x100000 -l 0xc0000
>>
Boot script timeout (1000ms resolution): 3
Use BOOTP for network configuration: true
GDB connection port: 9000
Network debug at boot time: false
Update RedBoot non-volatile configuration - are you sure (y/n)? y
```

Then, rebooting the Assabet is just a matter of waiting for the login prompt.

Nicolas Pitre nico@fluxnic.net

June 12, 2001

Status of peripherals in -rmk tree (updated 14/10/2001)

Assabet:

Serial ports:

Radio: TX, RX, CTS, DSR, DCD, RI

- PM: Not tested.
- COM: TX, RX, CTS, DSR, DCD, RTS, DTR, PM
- PM: Not tested.
- I2C: Implemented, not fully tested.
- L3: Fully tested, pass.
- PM: Not tested.

Video:

- LCD: Fully tested. PM

  (LCD doesn't like being blanked with neponset connected)
- Video out: Not fully

Audio:

UDA1341: - Playback: Fully tested, pass. - Record: Implemented, not tested. - PM: Not tested.

UCB1200: - Audio play: Implemented, not heavily tested. - Audio rec: Implemented, not heavily tested. - Telco audio play: Implemented, not heavily tested. - Telco audio rec: Implemented, not heavily tested. - POTS control: No - Touchscreen: Yes - PM: Not tested.

Other:

- PCMCIA:
- LPE: Fully tested, pass.
- USB: No
- IRDA:
- SIR: Fully tested, pass.
- FIR: Fully tested, pass.
- PM: Not tested.

Neponset:

Serial ports:

- COM1,2: TX, RX, CTS, DSR, DCD, RTS, DTR

- PM: Not tested.
- USB: Implemented, not heavily tested.
- PCMCIA: Implemented, not heavily tested.
- CF: Implemented, not heavily tested.
- PM: Not tested.

More stuff can be found in the -np (Nicolas Pitre's) tree.

CerfBoard/Cube

*** The StrongARM version of the CerfBoard/Cube has been discontinued ***

The Intrinsyc CerfBoard is a StrongARM 1110-based computer on a board that measures approximately 2" square. It includes an Ethernet controller, an RS232-compatible serial port, a USB function port, and one CompactFlash+ slot on the back. Pictures can be found at the Intrinsyc website, http://www.intrinsyc.com.

This document describes the support in the Linux kernel for the Intrinsyc CerfBoard.

Supported in this version

- CompactFlash+ slot (select PCMCIA in General Setup and any options that may be required)
- Onboard Crystal CS8900 Ethernet controller (Cerf CS8900A support in Network Devices)
- Serial ports with a serial console (hardcoded to 38400 8N1)

In order to get this kernel onto your Cerf, you need a server that runs both BOOTP and TFTP. Detailed instructions should have come with your evaluation kit on how to use the bootloader. This series of commands will suffice:

```
make ARCH=arm CROSS_COMPILE=arm-linux- cerfcube_defconfig
make ARCH=arm CROSS_COMPILE=arm-linux- zImage
make ARCH=arm CROSS_COMPILE=arm-linux- modules
cp arch/arm/boot/zImage <TFTP directory>
```

support@intrinsyc.com

Linux Advanced Radio Terminal (LART)

The LART is a small (7.5 x 10cm) SA-1100 board, designed for embedded applications. It has 32 MB DRAM, 4MB Flash ROM, double RS232 and all other StrongARM-gadgets. Almost all SA signals are directly accessible through a number of connectors. The powersupply accepts voltages between 3.5V and 16V and is overdimensioned to support a range of daughterboards. A quad Ethernet / IDE / PS2 / sound daughterboard is under development, with plenty of others in different stages of planning.

The hardware designs for this board have been released under an open license; see the LART page at http://www.lartmaker.nl/ for more information.

SA1100 serial port

The SA1100 serial port had its major/minor numbers officially assigned:

```
> Date: Sun, 24 Sep 2000 21:40:27 -0700
> From: H. Peter Anvin <hpa@transmeta.com>
> To: Nicolas Pitre <nico@CAM.ORG>
> Cc: Device List Maintainer <device@lanana.org>
> Subject: Re: device
>
> Okay.  Note that device numbers 204 and 205 are used for "low density
> serial devices", so you will have a range of minors on those majors (the
> tty device layer handles this just fine, so you don't have to worry about
> doing anything special.)
>
> So your assignments are:
>
> 204 char        Low-density serial ports
>                   5 = /dev/ttySA0               SA1100 builtin serial port 0
>                   6 = /dev/ttySA1               SA1100 builtin serial port 1
>                   7 = /dev/ttySA2               SA1100 builtin serial port 2
>
> 205 char        Low-density serial ports (alternate device)
>                   5 = /dev/cusa0                Callout device for ttySA0
>                   6 = /dev/cusa1                Callout device for ttySA1
>                   7 = /dev/cusa2                Callout device for ttySA2
>
```

You must create those inodes in /dev on the root filesystem used by your SA1100-based device:

```
mknod ttySA0 c 204 5
mknod ttySA1 c 204 6
mknod ttySA2 c 204 7
mknod cusa0 c 205 5
mknod cusa1 c 205 6
mknod cusa2 c 205 7
```

In addition to the creation of the appropriate device nodes above, you must ensure your user space applications make use of the correct device name. The classic example is the content of the /etc/inittab file where you might have a getty process started on ttyS0.

In this case:

- replace occurrences of ttyS0 with ttySA0, ttyS1 with ttySA1, etc.
- don't forget to add 'ttySA0', 'console', or the appropriate tty name in /etc/securetty for root to be allowed to login as well.

2.17.12 STM32F746 Overview

Introduction

The STM32F746 is a Cortex-M7 MCU aimed at various applications. It features:

- Cortex-M7 core running up to @216MHz
- 1MB internal flash, 320KBytes internal RAM (+4KB of backup SRAM)
- FMC controller to connect SDRAM, NOR and NAND memories
- Dual mode QSPI
- SD/MMC/SDIO support
- Ethernet controller
- USB OTFG FS & HS controllers
- I2C, SPI, CAN busses support
- Several 16 & 32 bits general purpose timers
- Serial Audio interface
- LCD controller
- HDMI-CEC
- SPDIFRX

Resources

Datasheet and reference manual are publicly available on ST website (STM32F746).

Authors
Alexandre Torgue <alexandre.torgue@st.com>

2.17.13 STM32 ARM Linux Overview

Introduction

The STMicroelectronics STM32 family of Cortex-A microprocessors (MPUs) and Cortex-M microcontrollers (MCUs) are supported by the 'STM32' platform of ARM Linux.

Configuration

For MCUs, use the provided default configuration:
make stm32_defconfig

For MPUs, use multi_v7 configuration:
make multi_v7_defconfig

Layout

All the files for multiple machine families are located in the platform code contained in arch/arm/mach-stm32

There is a generic board board-dt.c in the mach folder which support Flattened Device Tree, which means, it works with any compatible board with Device Trees.

Authors

- Maxime Coquelin <mcoquelin.stm32@gmail.com>
- Ludovic Barre <ludovic.barre@st.com>
- Gerald Baeza <gerald.baeza@st.com>

2.17.14 STM32H743 Overview

Introduction

The STM32H743 is a Cortex-M7 MCU aimed at various applications. It features:

- Cortex-M7 core running up to @400MHz
- 2MB internal flash, 1MBytes internal RAM
- FMC controller to connect SDRAM, NOR and NAND memories
- Dual mode QSPI
- SD/MMC/SDIO support
- Ethernet controller
- USB OTFG FS & HS controllers
- I2C, SPI, CAN busses support
- Several 16 & 32 bits general purpose timers
- Serial Audio interface
- LCD controller
- HDMI-CEC
- SPDIFRX
- DFSDM

Resources

Datasheet and reference manual are publicly available on ST website (STM32H743).

Authors
Alexandre Torgue <alexandre.torgue@st.com>

2.17.15 STM32H750 Overview

Introduction

The STM32H750 is a Cortex-M7 MCU aimed at various applications. It features:

- Cortex-M7 core running up to @480MHz
- 128K internal flash, 1MBytes internal RAM
- FMC controller to connect SDRAM, NOR and NAND memories
- Dual mode QSPI
- SD/MMC/SDIO support
- Ethernet controller
- USB OTFG FS & HS controllers
- I2C, SPI, CAN busses support
- Several 16 & 32 bits general purpose timers
- Serial Audio interface
- LCD controller
- HDMI-CEC
- SPDIFRX
- DFSDM

Resources

Datasheet and reference manual are publicly available on ST website (STM32H750).

Authors
Dillon Min <dillon.minfei@gmail.com>

2.17.16 STM32F769 Overview

Introduction

The STM32F769 is a Cortex-M7 MCU aimed at various applications. It features:

- Cortex-M7 core running up to @216MHz
- 2MB internal flash, 512KBytes internal RAM (+4KB of backup SRAM)
- FMC controller to connect SDRAM, NOR and NAND memories
- Dual mode QSPI
- SD/MMC/SDIO support*2
- Ethernet controller
- USB OTFG FS & HS controllers

- I2C*4, SPI*6, CAN*3 busses support
- Several 16 & 32 bits general purpose timers
- Serial Audio interface*2
- LCD controller
- HDMI-CEC
- DSI
- SPDIFRX
- MDIO salave interface

Resources

Datasheet and reference manual are publicly available on ST website (STM32F769).

Authors
Alexandre Torgue <alexandre.torgue@st.com>

2.17.17 STM32F429 Overview

Introduction

The STM32F429 is a Cortex-M4 MCU aimed at various applications. It features:

- ARM Cortex-M4 up to 180MHz with FPU
- 2MB internal Flash Memory
- External memory support through FMC controller (PSRAM, SDRAM, NOR, NAND)
- I2C, SPI, SAI, CAN, USB OTG, Ethernet controllers
- LCD controller & Camera interface
- Cryptographic processor

Resources

Datasheet and reference manual are publicly available on ST website (STM32F429).

Authors
Maxime Coquelin <mcoquelin.stm32@gmail.com>

2.17.18 STM32MP13 Overview

Introduction

The STM32MP131/STM32MP133/STM32MP135 are Cortex-A MPU aimed at various applications. They feature:

- One Cortex-A7 application core
- Standard memories interface support
- Standard connectivity, widely inherited from the STM32 MCU family
- Comprehensive security support

More details:

- Cortex-A7 core running up to @900MHz
- FMC controller to connect SDRAM, NOR and NAND memories
- QSPI
- SD/MMC/SDIO support
- 2*Ethernet controller
- CAN
- ADC/DAC
- USB EHCI/OHCI controllers
- USB OTG
- I2C, SPI, CAN busses support
- Several general purpose timers
- Serial Audio interface
- LCD controller
- DCMIPP
- SPDIFRX
- DFSDM

Authors

- Alexandre Torgue <alexandre.torgue@foss.st.com>

2.17.19 STM32MP151 Overview

Introduction

The STM32MP151 is a Cortex-A MPU aimed at various applications. It features:

- Single Cortex-A7 application core
- Standard memories interface support
- Standard connectivity, widely inherited from the STM32 MCU family
- Comprehensive security support

More details:

- Cortex-A7 core running up to @800MHz
- FMC controller to connect SDRAM, NOR and NAND memories
- QSPI
- SD/MMC/SDIO support
- Ethernet controller
- ADC/DAC
- USB EHCI/OHCI controllers
- USB OTG
- I2C, SPI busses support
- Several general purpose timers
- Serial Audio interface
- LCD-TFT controller
- DCMIPP
- SPDIFRX
- DFSDM

Authors

- Roan van Dijk <roan@protonic.nl>

2.17.20 STM32MP157 Overview

Introduction

The STM32MP157 is a Cortex-A MPU aimed at various applications. It features:

- Dual core Cortex-A7 application core
- 2D/3D image composition with GPU
- Standard memories interface support
- Standard connectivity, widely inherited from the STM32 MCU family

- Comprehensive security support

Authors

- Ludovic Barre <ludovic.barre@st.com>
- Gerald Baeza <gerald.baeza@st.com>

2.17.21 STM32 DMA-MDMA chaining

Introduction

This document describes the STM32 DMA-MDMA chaining feature. But before going further, let's introduce the peripherals involved.

To offload data transfers from the CPU, STM32 microprocessors (MPUs) embed direct memory access controllers (DMA).

STM32MP1 SoCs embed both STM32 DMA and STM32 MDMA controllers. STM32 DMA request routing capabilities are enhanced by a DMA request multiplexer (STM32 DMAMUX).

STM32 DMAMUX

STM32 DMAMUX routes any DMA request from a given peripheral to any STM32 DMA controller (STM32MP1 counts two STM32 DMA controllers) channels.

STM32 DMA

STM32 DMA is mainly used to implement central data buffer storage (usually in the system SRAM) for different peripheral. It can access external RAMs but without the ability to generate convenient burst transfer ensuring the best load of the AXI.

STM32 MDMA

STM32 MDMA (Master DMA) is mainly used to manage direct data transfers between RAM data buffers without CPU intervention. It can also be used in a hierarchical structure that uses STM32 DMA as first level data buffer interfaces for AHB peripherals, while the STM32 MDMA acts as a second level DMA with better performance. As a AXI/AHB master, STM32 MDMA can take control of the AXI/AHB bus.

Principles

STM32 DMA-MDMA chaining feature relies on the strengths of STM32 DMA and STM32 MDMA controllers.

STM32 DMA has a circular Double Buffer Mode (DBM). At each end of transaction (when DMA data counter - DMA_SxNDTR - reaches 0), the memory pointers (configured with DMA_SxSM0AR and DMA_SxM1AR) are swapped and the DMA data counter is automatically reloaded. This allows the SW or the STM32 MDMA to process one memory area while the second memory area is being filled/used by the STM32 DMA transfer.

With STM32 MDMA linked-list mode, a single request initiates the data array (collection of nodes) to be transferred until the linked-list pointer for the channel is null. The channel transfer complete of the last node is the end of transfer, unless first and last nodes are linked to each other, in such a case, the linked-list loops on to create a circular MDMA transfer.

STM32 MDMA has direct connections with STM32 DMA. This enables autonomous communication and synchronization between peripherals, thus saving CPU resources and bus congestion. Transfer Complete signal of STM32 DMA channel can triggers STM32 MDMA transfer. STM32 MDMA can clear the request generated by the STM32 DMA by writing to its Interrupt Clear register (whose address is stored in MDMA_CxMAR, and bit mask in MDMA_CxMDR).

Table 2: STM32 MDMA interconnect table with STM32 DMA

STM32 DMAMUX channels	STM32 DMA channels	STM32 DMA Transfer complete signal	STM32 MDMA request
Channel *0*	DMA1 channel 0	dma1_tcf0	*0x00*
Channel *1*	DMA1 channel 1	dma1_tcf1	*0x01*
Channel *2*	DMA1 channel 2	dma1_tcf2	*0x02*
Channel *3*	DMA1 channel 3	dma1_tcf3	*0x03*
Channel *4*	DMA1 channel 4	dma1_tcf4	*0x04*
Channel *5*	DMA1 channel 5	dma1_tcf5	*0x05*
Channel *6*	DMA1 channel 6	dma1_tcf6	*0x06*
Channel *7*	DMA1 channel 7	dma1_tcf7	*0x07*
Channel *8*	DMA2 channel 0	dma2_tcf0	*0x08*
Channel *9*	DMA2 channel 1	dma2_tcf1	*0x09*
Channel *10*	DMA2 channel 2	dma2_tcf2	*0x0A*
Channel *11*	DMA2 channel 3	dma2_tcf3	*0x0B*
Channel *12*	DMA2 channel 4	dma2_tcf4	*0x0C*
Channel *13*	DMA2 channel 5	dma2_tcf5	*0x0D*
Channel *14*	DMA2 channel 6	dma2_tcf6	*0x0E*
Channel *15*	DMA2 channel 7	dma2_tcf7	*0x0F*

STM32 DMA-MDMA chaining feature then uses a SRAM buffer. STM32MP1 SoCs embed three fast access static internal RAMs of various size, used for data storage. Due to STM32 DMA legacy (within microcontrollers), STM32 DMA performances are bad with DDR, while they are optimal with SRAM. Hence the SRAM buffer used between STM32 DMA and STM32 MDMA. This buffer is split in two equal periods and STM32 DMA uses one period while STM32 MDMA uses the other period simultaneously.

```
               dma[1:2]-tcf[0:7]
              .-----------------.
 ____________ '    _________     V____________
| STM32 DMA  |    /  __|>_  \    | STM32 MDMA |
|------------|   |  /     \  |   |------------|
| DMA_SxM0AR |<=>| | SRAM  | |<=>| []-[]...[] |
| DMA_SxM1AR |   |  \_____/  |   |            |
|____________|    \___<|____/    |____________|
```

STM32 DMA-MDMA chaining uses (struct dma_slave_config).peripheral_config to exchange the parameters needed to configure MDMA. These parameters are gathered into a u32 array with three values:

- the STM32 MDMA request (which is actually the DMAMUX channel ID),
- the address of the STM32 DMA register to clear the Transfer Complete interrupt flag,
- the mask of the Transfer Complete interrupt flag of the STM32 DMA channel.

Device Tree updates for STM32 DMA-MDMA chaining support

1. **Allocate a SRAM buffer**

   SRAM device tree node is defined in SoC device tree. You can refer to it in your board device tree to define your SRAM pool.

   ```
   &sram {
           my_foo_device_dma_pool: dma-sram@0 {
                   reg = <0x0 0x1000>;
           };
   };
   ```

   Be careful of the start index, in case there are other SRAM consumers. Define your pool size strategically: to optimise chaining, the idea is that STM32 DMA and STM32 MDMA can work simultaneously, on each buffer of the SRAM. If the SRAM period is greater than the expected DMA transfer, then STM32 DMA and STM32 MDMA will work sequentially instead of simultaneously. It is not a functional issue but it is not optimal.

   Don't forget to refer to your SRAM pool in your device node. You need to define a new property.

   ```
   &my_foo_device {
           ...
           my_dma_pool = &my_foo_device_dma_pool;
   };
   ```

   Then get this SRAM pool in your foo driver and allocate your SRAM buffer.

2. **Allocate a STM32 DMA channel and a STM32 MDMA channel**

   You need to define an extra channel in your device tree node, in addition to the one you should already have for "classic" DMA operation.

   This new channel must be taken from STM32 MDMA channels, so, the phandle of the DMA controller to use is the MDMA controller's one.

   ```
   &my_foo_device {
           [...]
           my_dma_pool = &my_foo_device_dma_pool;
           dmas = <&dmamux1 ...>,                // STM32 DMA␣
    ↪channel
                  <&mdma1 0 0x3 0x1200000a 0 0>; // + STM32 MDMA␣
    ↪channel
   };
   ```

Concerning STM32 MDMA bindings:

1. The request line number : whatever the value here, it will be overwritten by MDMA driver with the STM32 DMAMUX channel ID passed through (struct dma_slave_config).peripheral_config

2. The priority level : choose Very High (0x3) so that your channel will take priority other the other during request arbitration

3. A 32bit mask specifying the DMA channel configuration : source and destination address increment, block transfer with 128 bytes per single transfer

4. The 32bit value specifying the register to be used to acknowledge the request: it will be overwritten by MDMA driver, with the DMA channel interrupt flag clear register address passed through (struct dma_slave_config).peripheral_config

5. The 32bit mask specifying the value to be written to acknowledge the request: it will be overwritten by MDMA driver, with the DMA channel Transfer Complete flag passed through (struct dma_slave_config).peripheral_config

Driver updates for STM32 DMA-MDMA chaining support in foo driver

0. (optional) Refactor the original sg_table if dmaengine_prep_slave_sg()

In case of dmaengine_prep_slave_sg(), the original sg_table can't be used as is. Two new sg_tables must be created from the original one. One for STM32 DMA transfer (where memory address targets now the SRAM buffer instead of DDR buffer) and one for STM32 MDMA transfer (where memory address targets the DDR buffer).

The new sg_list items must fit SRAM period length. Here is an example for DMA_DEV_TO_MEM:

```
/*
  * Assuming sgl and nents, respectively the initial␣
↪scatterlist and its
  * length.
  * Assuming sram_dma_buf and sram_period, respectively the␣
↪memory
  * allocated from the pool for DMA usage, and the length of␣
↪the period,
  * which is half of the sram_buf size.
  */
struct sg_table new_dma_sgt, new_mdma_sgt;
struct scatterlist *s, *_sgl;
dma_addr_t ddr_dma_buf;
u32 new_nents = 0, len;
int i;

/* Count the number of entries needed */
for_each_sg(sgl, s, nents, i)
        if (sg_dma_len(s) > sram_period)
                new_nents += DIV_ROUND_UP(sg_dma_len(s), sram_
↪period);
        else
                new_nents++;

/* Create sg table for STM32 DMA channel */
ret = sg_alloc_table(&new_dma_sgt, new_nents, GFP_ATOMIC);
if (ret)
        dev_err(dev, "DMA sg table alloc failed\n");

for_each_sg(new_dma_sgt.sgl, s, new_dma_sgt.nents, i) {
        _sgl = sgl;
        sg_dma_len(s) = min(sg_dma_len(_sgl), sram_period);
        /* Targets the beginning = first half of the sram_buf */
        s->dma_address = sram_buf;
        /*
          * Targets the second half of the sram_buf
          * for odd indexes of the item of the sg_list
          */
        if (i & 1)
                s->dma_address += sram_period;
}

/* Create sg table for STM32 MDMA channel */
ret = sg_alloc_table(&new_mdma_sgt, new_nents, GFP_ATOMIC);
if (ret)
        dev_err(dev, "MDMA sg_table alloc failed\n");

_sgl = sgl;
len = sg_dma_len(sgl);
ddr_dma_buf = sg_dma_address(sgl);
for_each_sg(mdma_sgt.sgl, s, mdma_sgt.nents, i) {
        size_t bytes = min_t(size_t, len, sram_period);

        sg_dma_len(s) = bytes;
        sg_dma_address(s) = ddr_dma_buf;
        len -= bytes;

        if (!len && sg_next(_sgl)) {
                _sgl = sg_next(_sgl);
                len = sg_dma_len(_sgl);
                ddr_dma_buf = sg_dma_address(_sgl);
        } else {
                ddr_dma_buf += bytes;
        }
}
```

Don't forget to release these new sg_tables after getting the descriptors with dmaengine_prep_slave_sg().

1. **Set controller specific parameters**

   First, use dmaengine_slave_config() with a struct dma_slave_config to configure STM32 DMA channel. You just have to take care of DMA addresses, the memory address (depending on the transfer direction) must point on your SRAM buffer, and set (struct dma_slave_config).peripheral_size != 0.

STM32 DMA driver will check (struct dma_slave_config).peripheral_size to determine if chaining is being used or not. If it is used, then STM32 DMA driver fills (struct dma_slave_config).peripheral_config with an array of three u32 : the first one containing STM32 DMAMUX channel ID, the second one the channel interrupt flag clear register address, and the third one the channel Transfer Complete flag mask.

Then, use dmaengine_slave_config with another struct dma_slave_config to configure STM32 MDMA channel. Take care of DMA addresses, the device address (depending on the transfer direction) must point on your SRAM buffer, and the memory address must point to the buffer originally used for "classic" DMA operation. Use the previous (struct dma_slave_config).peripheral_size and .peripheral_config that have been updated by STM32 DMA driver, to set (struct dma_slave_config).peripheral_size and .peripheral_config of the struct dma_slave_config to configure STM32 MDMA channel.

```
struct dma_slave_config dma_conf;
struct dma_slave_config mdma_conf;

memset(&dma_conf, 0, sizeof(dma_conf));
[...]
config.direction = DMA_DEV_TO_MEM;
config.dst_addr = sram_dma_buf;        // SRAM buffer
config.peripheral_size = 1;            // peripheral_size != 0␣
↪=> chaining

dmaengine_slave_config(dma_chan, &dma_config);

memset(&mdma_conf, 0, sizeof(mdma_conf));
config.direction = DMA_DEV_TO_MEM;
mdma_conf.src_addr = sram_dma_buf;     // SRAM buffer
mdma_conf.dst_addr = rx_dma_buf;       // original memory buffer
mdma_conf.peripheral_size = dma_conf.peripheral_size;       //
↪<- dma_conf
mdma_conf.peripheral_config = dma_config.peripheral_config; //
↪<- dma_conf

dmaengine_slave_config(mdma_chan, &mdma_conf);
```

2. **Get a descriptor for STM32 DMA channel transaction**

   In the same way you get your descriptor for your "classic" DMA operation, you just have to replace the original sg_list (in case of dmaengine_prep_slave_sg()) with the new sg_list using SRAM buffer, or to replace the original buffer address, length and period (in case of dmaengine_prep_dma_cyclic()) with the new SRAM buffer.

3. **Get a descriptor for STM32 MDMA channel transaction**

If you previously get descriptor (for STM32 DMA) with

- dmaengine_prep_slave_sg(), then use dmaengine_prep_slave_sg() for STM32 MDMA;
- dmaengine_prep_dma_cyclic(), then use dmaengine_prep_dma_cyclic() for STM32 MDMA.

Use the new sg_list using SRAM buffer (in case of dmaengine_prep_slave_sg()) or, depending on the transfer direction, either the original DDR buffer (in case of DMA_DEV_TO_MEM) or the SRAM buffer (in case of DMA_MEM_TO_DEV), the source address being previously set with dmaengine_slave_config().

4. **Submit both transactions**

Before submitting your transactions, you may need to define on which descriptor you want a callback to be called at the end of the transfer (dmaengine_prep_slave_sg()) or the period (dmaengine_prep_dma_cyclic()). Depending on the direction, set the callback on the descriptor that finishes the overal transfer:

- DMA_DEV_TO_MEM: set the callback on the "MDMA" descriptor
- DMA_MEM_TO_DEV: set the callback on the "DMA" descriptor

Then, submit the descriptors whatever the order, with dmaengine_tx_submit().

5. **Issue pending requests (and wait for callback notification)**

As STM32 MDMA channel transfer is triggered by STM32 DMA, you must issue STM32 MDMA channel before STM32 DMA channel.

If any, your callback will be called to warn you about the end of the overal transfer or the period completion.

Don't forget to terminate both channels. STM32 DMA channel is configured in cyclic Double-Buffer mode so it won't be disabled by HW, you need to terminate it. STM32 MDMA channel will be stopped by HW in case of sg transfer, but not in case of cyclic transfer. You can terminate it whatever the kind of transfer.

STM32 DMA-MDMA chaining DMA_MEM_TO_DEV special case

STM32 DMA-MDMA chaining in DMA_MEM_TO_DEV is a special case. Indeed, the STM32 MDMA feeds the SRAM buffer with the DDR data, and the STM32 DMA reads data from SRAM buffer. So some data (the first period) have to be copied in SRAM buffer when the STM32 DMA starts to read.

A trick could be pausing the STM32 DMA channel (that will raise a Transfer Complete signal, triggering the STM32 MDMA channel), but the first data read by the STM32 DMA could be "wrong". The proper way is to prepare the first SRAM period with dmaengine_prep_dma_memcpy(). Then this first period should be "removed" from the sg or the cyclic transfer.

Due to this complexity, rather use the STM32 DMA-MDMA chaining for DMA_DEV_TO_MEM and keep the "classic" DMA usage for DMA_MEM_TO_DEV, unless you're not afraid.

Resources

Application note, datasheet and reference manual are available on ST website (STM32MP1).

Dedicated focus on three application notes (AN5224, AN4031 & AN5001) dealing with STM32 DMAMUX, STM32 DMA and STM32 MDMA.

Authors

- Amelie Delaunay <amelie.delaunay@foss.st.com>

2.17.22 ARM Allwinner SoCs

This document lists all the ARM Allwinner SoCs that are currently supported in mainline by the Linux kernel. This document will also provide links to documentation and/or datasheet for these SoCs.

SunXi family

Linux kernel mach directory: arch/arm/mach-sunxi

Flavors:

- ARM926 based SoCs - Allwinner F20 (sun3i)
  - Not Supported
- ARM Cortex-A8 based SoCs - Allwinner A10 (sun4i)
  - Datasheet

    http://dl.linux-sunxi.org/A10/A10%20Datasheet%20-%20v1.21%20%282012-04-06%29.pdf
  - User Manual

    http://dl.linux-sunxi.org/A10/A10%20User%20Manual%20-%20v1.20%20%282012-04-09%2c%20DECRYPTED%29.pdf
  - Allwinner A10s (sun5i)
    - Datasheet

      http://dl.linux-sunxi.org/A10s/A10s%20Datasheet%20-%20v1.20%20%282012-03-27%29.pdf
  - Allwinner A13 / R8 (sun5i)
    - Datasheet

      http://dl.linux-sunxi.org/A13/A13%20Datasheet%20-%20v1.12%20%282012-03-29%29.pdf
    - User Manual

      http://dl.linux-sunxi.org/A13/A13%20User%20Manual%20-%20v1.2%20%282013-01-08%29.pdf
  - Next Thing Co GR8 (sun5i)

- Single ARM Cortex-A7 based SoCs - Allwinner V3s (sun8i)
  - Datasheet

    http://linux-sunxi.org/File:Allwinner_V3s_Datasheet_V1.0.pdf
- Dual ARM Cortex-A7 based SoCs - Allwinner A20 (sun7i)
  - User Manual

    http://dl.linux-sunxi.org/A20/A20%20User%20Manual%202013-03-22.pdf
  - Allwinner A23 (sun8i)
    - Datasheet

      http://dl.linux-sunxi.org/A23/A23%20Datasheet%20V1.0%2020130830.pdf
    - User Manual

      http://dl.linux-sunxi.org/A23/A23%20User%20Manual%20V1.0%2020130830.pdf
- Quad ARM Cortex-A7 based SoCs - Allwinner A31 (sun6i)
  - Datasheet

    http://dl.linux-sunxi.org/A31/A3x_release_document/A31/IC/A31%20datasheet%20V1.3%2020131106.pdf
  - User Manual

    http://dl.linux-sunxi.org/A31/A3x_release_document/A31/IC/A31%20user%20manual%20V1.1%2020130630.pdf
  - Allwinner A31s (sun6i)
    - Datasheet

      http://dl.linux-sunxi.org/A31/A3x_release_document/A31s/IC/A31s%20datasheet%20V1.3%2020131106.pdf
    - User Manual

      http://dl.linux-sunxi.org/A31/A3x_release_document/A31s/IC/A31s%20User%20Manual%20%20V1.0%2020130322.pdf
  - Allwinner A33 (sun8i)
    - Datasheet

      http://dl.linux-sunxi.org/A33/A33%20Datasheet%20release%201.1.pdf
    - User Manual

      http://dl.linux-sunxi.org/A33/A33%20user%20manual%20release%201.1.pdf
  - Allwinner H2+ (sun8i)
    - No document available now, but is known to be working properly with H3 drivers and memory map.
  - Allwinner H3 (sun8i)

    * Datasheet

      https://linux-sunxi.org/images/4/4b/Allwinner_H3_Datasheet_V1.2.pdf

  - Allwinner R40 (sun8i)

    * Datasheet

      https://github.com/tinalinux/docs/raw/r40-v1.y/R40_Datasheet_V1.0.pdf

    * User Manual

      https://github.com/tinalinux/docs/raw/r40-v1.y/Allwinner_R40_User_Manual_V1.0.pdf

- Quad ARM Cortex-A15, Quad ARM Cortex-A7 based SoCs - Allwinner A80

  - Datasheet

    http://dl.linux-sunxi.org/A80/A80_Datasheet_Revision_1.0_0404.pdf

- Octa ARM Cortex-A7 based SoCs - Allwinner A83T

  - Datasheet

    https://github.com/allwinner-zh/documents/raw/master/A83T/A83T_Datasheet_v1.3_20150510.pdf

  - User Manual

    https://github.com/allwinner-zh/documents/raw/master/A83T/A83T_User_Manual_v1.5.1_20150513.pdf

- Quad ARM Cortex-A53 based SoCs - Allwinner A64

  - Datasheet

    http://dl.linux-sunxi.org/A64/A64_Datasheet_V1.1.pdf

  - User Manual

    http://dl.linux-sunxi.org/A64/Allwinner%20A64%20User%20Manual%20v1.0.pdf

  - Allwinner H6

    * Datasheet

      https://linux-sunxi.org/images/5/5c/Allwinner_H6_V200_Datasheet_V1.1.pdf

    * User Manual

      https://linux-sunxi.org/images/4/46/Allwinner_H6_V200_User_Manual_V1.1.pdf

  - Allwinner H616

    * Datasheet

      https://linux-sunxi.org/images/b/b9/H616_Datasheet_V1.0_cleaned.pdf

    * User Manual

      https://linux-sunxi.org/images/2/24/H616_User_Manual_V1.0_cleaned.pdf

2.17.23 Samsung SoC

Samsung GPIO implementation

Introduction

This outlines the Samsung GPIO implementation and the architecture specific calls provided alongside the drivers/gpio core.

GPIOLIB integration

The gpio implementation uses gpiolib as much as possible, only providing specific calls for the items that require Samsung specific handling, such as pin special-function or pull resistor control.

GPIO numbering is synchronised between the Samsung and gpiolib system.

PIN configuration

Pin configuration is specific to the Samsung architecture, with each SoC registering the necessary information for the core gpio configuration implementation to configure pins as necessary.

The s3c_gpio_cfgpin() and s3c_gpio_setpull() provide the means for a driver or machine to change gpio configuration.

See arch/arm/mach-s3c/gpio-cfg.h for more information on these functions.

Interface between kernel and boot loaders on Exynos boards

Author: Krzysztof Kozlowski

Date : 6 June 2015

The document tries to describe currently used interface between Linux kernel and boot loaders on Samsung Exynos based boards. This is not a definition of interface but rather a description of existing state, a reference for information purpose only.

In the document "boot loader" means any of following: U-boot, proprietary SBOOT or any other firmware for ARMv7 and ARMv8 initializing the board before executing kernel.

1. Non-Secure mode

Address: sysram_ns_base_addr

Offset	Value	Purpose
0x08	exynos_cpu_resume_ns, mcpm_entry_point	System suspend
0x0c	0x00000bad (Magic cookie)	System suspend
0x1c	exynos4_secondary_startup	Secondary CPU boot
0x1c + 4*cpu	exynos4_secondary_startup (Exynos4412)	Secondary CPU boot
0x20	0xfcba0d10 (Magic cookie)	AFTR
0x24	exynos_cpu_resume_ns	AFTR
0x28 + 4*cpu	0x8 (Magic cookie, Exynos3250)	AFTR
0x28	0x0 or last value during resume (Exynos542x)	System suspend

2. Secure mode

Address: sysram_base_addr

Offset	Value	Purpose
0x00	exynos4_secondary_startup	Secondary CPU boot
0x04	exynos4_secondary_startup (Exynos542x)	Secondary CPU boot
4*cpu	exynos4_secondary_startup (Exynos4412)	Secondary CPU boot
0x20	exynos_cpu_resume (Exynos4210 r1.0)	AFTR
0x24	0xfcba0d10 (Magic cookie, Exynos4210 r1.0)	AFTR

Address: pmu_base_addr

Offset	Value	Purpose
0x0800	exynos_cpu_resume	AFTR, suspend
0x0800	mcpm_entry_point (Exynos542x with MCPM)	AFTR, suspend
0x0804	0xfcba0d10 (Magic cookie)	AFTR
0x0804	0x00000bad (Magic cookie)	System suspend
0x0814	exynos4_secondary_startup (Exynos4210 r1.1)	Secondary CPU boot
0x0818	0xfcba0d10 (Magic cookie, Exynos4210 r1.1)	AFTR
0x081C	exynos_cpu_resume (Exynos4210 r1.1)	AFTR

3. Other (regardless of secure/non-secure mode)

Address: pmu_base_addr

Offset	Value	Purpose
0x0908	Non-zero	Secondary CPU boot up indicator on Exynos3250 and Exynos542x

4. Glossary

AFTR - ARM Off Top Running, a low power mode, Cortex cores and many other modules are power gated, except the TOP modules MCPM - Multi-Cluster Power Management

Samsung ARM Linux Overview

Introduction

The Samsung range of ARM SoCs spans many similar devices, from the initial ARM9 through to the newest ARM cores. This document shows an overview of the current kernel support, how to use it and where to find the code that supports this.

The currently supported SoCs are:

- S3C64XX: S3C6400 and S3C6410
- S5PC110 / S5PV210

Configuration

A number of configurations are supplied, as there is no current way of unifying all the SoCs into one kernel.

s5pc110_defconfig

- S5PC110 specific default configuration

s5pv210_defconfig

- S5PV210 specific default configuration

Layout

The directory layout is currently being restructured, and consists of several platform directories and then the machine specific directories of the CPUs being built for.

plat-samsung provides the base for all the implementations, and is the last in the line of include directories that are processed for the build specific information. It contains the base clock, GPIO and device definitions to get the system running.

plat-s5p is for s5p specific builds, and contains common support for the S5P specific systems. Not all S5Ps use all the features in this directory due to differences in the hardware.

Layout changes

The old plat-s3c and plat-s5pc1xx directories have been removed, with support moved to either plat-samsung or plat-s5p as necessary. These moves where to simplify the include and dependency issues involved with having so many different platform directories.

Port Contributors

Ben Dooks (BJD) Vincent Sanders Herbert Potzl Arnaud Patard (RTP) Roc Wu Klaus Fetscher Dimitry Andric Shannon Holland Guillaume Gourat (NexVision) Christer Weinigel (wingel) (Acer N30) Lucas Correia Villa Real (S3C2400 port)

Document Author

Copyright 2009-2010 Ben Dooks <ben-linux@fluff.org>

2.17.24 Frequently asked questions about the sunxi clock system

This document contains useful bits of information that people tend to ask about the sunxi clock system, as well as accompanying ASCII art when adequate.

Q: Why is the main 24MHz oscillator gateable? Wouldn't that break the
system?

A: The 24MHz oscillator allows gating to save power. Indeed, if gated
carelessly the system would stop functioning, but with the right steps, one can gate it and keep the system running. Consider this simplified suspend example:

While the system is operational, you would see something like:

```
24MHz         32kHz
 |
PLL1
 \
  \_ CPU Mux
       |
     [CPU]
```

When you are about to suspend, you switch the CPU Mux to the 32kHz oscillator:

```
  24Mhz         32kHz
   |              |
  PLL1            |
                 /
       CPU Mux _/
         |
       [CPU]

Finally you can gate the main oscillator::

                32kHz
                  |
                  |
                 /
       CPU Mux _/
         |
       [CPU]
```

Q: Were can I learn more about the sunxi clocks?

A: The linux-sunxi wiki contains a page documenting the clock registers,
you can find it at

http://linux-sunxi.org/A10/CCM

The authoritative source for information at this time is the ccmu driver released by Allwinner, you can find it at

https://github.com/linux-sunxi/linux-sunxi/tree/sunxi-3.0/arch/arm/mach-sun4i/clock/ccmu

2.17.25 SPEAr ARM Linux Overview

Introduction

SPEAr (Structured Processor Enhanced Architecture). weblink : http://www.st.com/spear

The ST Microelectronics SPEAr range of ARM9/CortexA9 System-on-Chip CPUs are supported by the 'spear' platform of ARM Linux. Currently SPEAr1310, SPEAr1340, SPEAr300, SPEAr310, SPEAr320 and SPEAr600 SOCs are supported.

Hierarchy in SPEAr is as follows:

SPEAr (Platform)

- **SPEAr3XX (3XX SOC series, based on ARM9)**
  - **SPEAr300 (SOC)**
    - SPEAr300 Evaluation Board
  - **SPEAr310 (SOC)**
    - SPEAr310 Evaluation Board
  - **SPEAr320 (SOC)**
    - SPEAr320 Evaluation Board
- **SPEAr6XX (6XX SOC series, based on ARM9)**
  - **SPEAr600 (SOC)**
    - SPEAr600 Evaluation Board
- **SPEAr13XX (13XX SOC series, based on ARM CORTEXA9)**
  - **SPEAr1310 (SOC)**
    - SPEAr1310 Evaluation Board
  - **SPEAr1340 (SOC)**
    - SPEAr1340 Evaluation Board

Configuration

A generic configuration is provided for each machine, and can be used as the default by:

```
make spear13xx_defconfig
make spear3xx_defconfig
make spear6xx_defconfig
```

Layout

The common files for multiple machine families (SPEAr3xx, SPEAr6xx and SPEAr13xx) are located in the platform code contained in arch/arm/plat-spear with headers in plat/.

Each machine series have a directory with name arch/arm/mach-spear followed by series name. Like mach-spear3xx, mach-spear6xx and mach-spear13xx.

Common file for machines of spear3xx family is mach-spear3xx/spear3xx.c, for spear6xx is mach-spear6xx/spear6xx.c and for spear13xx family is mach-spear13xx/spear13xx.c. mach-spear* also contain soc/machine specific files, like spear1310.c, spear1340.c spear300.c, spear310.c, spear320.c and spear600.c. mach-spear* doesn't contains board specific files as they fully support Flattened Device Tree.

Document Author

Viresh Kumar <vireshk@kernel.org>, (c) 2010-2012 ST Microelectronics

2.17.26 STiH407 Overview

Introduction

The STiH407 is the new generation of SoC for Multi-HD, AVC set-top boxes and server/connected client application for satellite, cable, terrestrial and IP-STB markets.

Features - ARM Cortex-A9 1.5 GHz dual core CPU (28nm) - SATA2, USB 3.0, PCIe, Gbit Ethernet

Document Author

Maxime Coquelin <maxime.coquelin@st.com>, (c) 2014 ST Microelectronics

2.17.27 STiH418 Overview

Introduction

The STiH418 is the new generation of SoC for UHDp60 set-top boxes and server/connected client application for satellite, cable, terrestrial and IP-STB markets.

Features - ARM Cortex-A9 1.5 GHz quad core CPU (28nm) - SATA2, USB 3.0, PCIe, Gbit Ethernet - HEVC L5.1 Main 10 - VP9

Document Author

Maxime Coquelin <maxime.coquelin@st.com>, (c) 2015 ST Microelectronics

2.17.28 STi ARM Linux Overview

Introduction

The ST Microelectronics Multimedia and Application Processors range of CortexA9 System-on-Chip are supported by the 'STi' platform of ARM Linux. Currently STiH407, STiH410 and STiH418 are supported.

configuration

The configuration for the STi platform is supported via the multi_v7_defconfig.

Layout

All the files for multiple machine families (STiH407, STiH410, and STiH418) are located in the platform code contained in arch/arm/mach-sti

There is a generic board board-dt.c in the mach folder which support Flattened Device Tree, which means, It works with any compatible board with Device Trees.

Document Author

Srinivas Kandagatla <srinivas.kandagatla@st.com>, (c) 2013 ST Microelectronics

2.17.29 Release notes for Linux Kernel VFP support code

Date: 20 May 2004

Author: Russell King

This is the first release of the Linux Kernel VFP support code. It provides support for the exceptions bounced from VFP hardware found on ARM926EJ-S.

This release has been validated against the SoftFloat-2b library by John R. Hauser using the TestFloat-2a test suite. Details of this library and test suite can be found at:

http://www.jhauser.us/arithmetic/SoftFloat.html

The operations which have been tested with this package are:

- fdiv
- fsub
- fadd
- fmul
- fcmp
- fcmpe
- fcvtd
- fcvts
- fsito
- ftosi
- fsqrt

All the above pass softfloat tests with the following exceptions:

- fadd/fsub shows some differences in the handling of +0 / -0 results when input operands differ in signs.
- the handling of underflow exceptions is slightly different. If a result underflows before rounding, but becomes a normalised number after rounding, we do not signal an underflow exception.

Other operations which have been tested by basic assembly-only tests are:

- fcpy
- fabs
- fneg
- ftoui
- ftosiz
- ftouiz

The combination operations have not been tested:

- fmac
- fnmac
- fmsc
- fnmsc
- fnmul

CHAPTER

THREE

ARM64 ARCHITECTURE

3.1 ACPI Tables

The expectations of individual ACPI tables are discussed in the list that follows.

If a section number is used, it refers to a section number in the ACPI specification where the object is defined. If "Signature Reserved" is used, the table signature (the first four bytes of the table) is the only portion of the table recognized by the specification, and the actual table is defined outside of the UEFI Forum (see Section 5.2.6 of the specification).

For ACPI on arm64, tables also fall into the following categories:

- Required: DSDT, FADT, GTDT, MADT, MCFG, RSDP, SPCR, XSDT
- Recommended: BERT, EINJ, ERST, HEST, PCCT, SSDT
- Optional: AGDI, BGRT, CEDT, CPEP, CSRT, DBG2, DRTM, ECDT, FACS, FPDT, HMAT, IBFT, IORT, MCHI, MPAM, MPST, MSCT, NFIT, PMTT, PPTT, RASF, SBST, SDEI, SLIT, SPMI, SRAT, STAO, TCPA, TPM2, UEFI, XENV
- Not supported: AEST, APMT, BOOT, DBGP, DMAR, ETDT, HPET, IVRS, LPIT, MSDM, OEMx, PDTT, PSDT, RAS2, RSDT, SLIC, WAET, WDAT, WDRT, WPBT

Table	Usage for ARMv8 Linux
AEST	Signature Reserved (signature == "AEST") **Arm Error Source Table** This table informs the OS of any error nodes in the system that are compliant with the Arm RAS architecture.
AGDI	Signature Reserved (signature == "AGDI") **Arm Generic diagnostic Dump and Reset Device Interface Table** This table describes a non-maskable event, that is used by the platform firmware, to request the OS to generate a diagnostic dump and reset the device.
APMT	Signature Reserved (signature == "APMT") **Arm Performance Monitoring Table** This table describes the properties of PMU support implemented by components in the system.

continues on next page

Table 1 – continued from previous page

Table	Usage for ARMv8 Linux
BERT	Section 18.3 (signature == "BERT") **Boot Error Record Table** Must be supplied if RAS support is provided by the platform. It is recommended this table be supplied.
BOOT	Signature Reserved (signature == "BOOT") **simple BOOT flag table** Microsoft only table, will not be supported.
BGRT	Section 5.2.22 (signature == "BGRT") **Boot Graphics Resource Table** Optional, not currently supported, with no real use-case for an ARM server.
CEDT	Signature Reserved (signature == "CEDT") **CXL Early Discovery Table** This table allows the OS to discover any CXL Host Bridges and the Host Bridge registers.
CPEP	Section 5.2.18 (signature == "CPEP") **Corrected Platform Error Polling table** Optional, not currently supported, and not recommended until such time as ARM-compatible hardware is available, and the specification suitably modified.
CSRT	Signature Reserved (signature == "CSRT") **Core System Resources Table** Optional, not currently supported.
DBG2	Signature Reserved (signature == "DBG2") **DeBuG port table 2** License has changed and should be usable. Optional if used instead of earlycon=<device> on the command line.
DBGP	Signature Reserved (signature == "DBGP") **DeBuG Port table** Microsoft only table, will not be supported.
DSDT	Section 5.2.11.1 (signature == "DSDT") **Differentiated System Description Table** A DSDT is required; see also SSDT. ACPI tables contain only one DSDT but can contain one or more SSDTs, which are optional. Each SSDT can only add to the ACPI namespace, but cannot modify or replace anything in the DSDT.
DMAR	Signature Reserved (signature == "DMAR") **DMA Remapping table** x86 only table, will not be supported.
DRTM	Signature Reserved (signature == "DRTM") **Dynamic Root of Trust for Measurement table** Optional, not currently supported.

continues on next page

Table 1 – continued from previous page

Table	Usage for ARMv8 Linux
ECDT	Section 5.2.16 (signature == "ECDT") **Embedded Controller Description Table** Optional, not currently supported, but could be used on ARM if and only if one uses the GPE_BIT field to represent an IRQ number, since there are no GPE blocks defined in hardware reduced mode. This would need to be modified in the ACPI specification.
EINJ	Section 18.6 (signature == "EINJ") **Error Injection table** This table is very useful for testing platform response to error conditions; it allows one to inject an error into the system as if it had actually occurred. However, this table should not be shipped with a production system; it should be dynamically loaded and executed with the ACPICA tools only during testing.
ERST	Section 18.5 (signature == "ERST") **Error Record Serialization Table** On a platform supports RAS, this table must be supplied if it is not UEFI-based; if it is UEFI-based, this table may be supplied. When this table is not present, UEFI run time service will be utilized to save and retrieve hardware error information to and from a persistent store.
ETDT	Signature Reserved (signature == "ETDT") **Event Timer Description Table** Obsolete table, will not be supported.
FACS	Section 5.2.10 (signature == "FACS") **Firmware ACPI Control Structure** It is unlikely that this table will be terribly useful. If it is provided, the Global Lock will NOT be used since it is not part of the hardware reduced profile, and only 64-bit address fields will be considered valid.

continues on next page

Table 1 – continued from previous page

Table	Usage for ARMv8 Linux
FADT	Section 5.2.9 (signature == "FACP") **Fixed ACPI Description Table** Required for arm64. The HW_REDUCED_ACPI flag must be set. All of the fields that are to be ignored when HW_REDUCED_ACPI is set are expected to be set to zero. If an FACS table is provided, the X_FIRMWARE_CTRL field is to be used, not FIRMWARE_CTRL. If PSCI is used (as is recommended), make sure that ARM_BOOT_ARCH is filled in properly - that the PSCI_COMPLIANT flag is set and that PSCI_USE_HVC is set or unset as needed (see table 5-37). For the DSDT that is also required, the X_DSDT field is to be used, not the DSDT field.
FPDT	Section 5.2.23 (signature == "FPDT") **Firmware Performance Data Table** Optional, useful for boot performance profiling.
GTDT	Section 5.2.24 (signature == "GTDT") **Generic Timer Description Table** Required for arm64.
HEST	Section 18.3.2 (signature == "HEST") **Hardware Error Source Table** ARM-specific error sources have been defined; please use those or the PCI types such as type 6 (AER Root Port), 7 (AER Endpoint), or 8 (AER Bridge), or use type 9 (Generic Hardware Error Source). Firmware first error handling is possible if and only if Trusted Firmware is being used on arm64. Must be supplied if RAS support is provided by the platform. It is recommended this table be supplied.
HMAT	Section 5.2.28 (signature == "HMAT") **Heterogeneous Memory Attribute Table** This table describes the memory attributes, such as memory side cache attributes and bandwidth and latency details, related to Memory Proximity Domains. The OS uses this information to optimize the system memory configuration.
HPET	Signature Reserved (signature == "HPET") **High Precision Event timer Table** x86 only table, will not be supported.

continues on next page

Table 1 – continued from previous page

Table	Usage for ARMv8 Linux
IBFT	Signature Reserved (signature == "IBFT") **iSCSI Boot Firmware Table** Microsoft defined table, support TBD.
IORT	Signature Reserved (signature == "IORT") **Input Output Remapping Table** arm64 only table, required in order to describe IO topology, SMMUs, and GIC ITSs, and how those various components are connected together, such as identifying which components are behind which SMMUs/ITSs. This table will only be required on certain SBSA platforms (e.g., when using GICv3-ITS and an SMMU); on SBSA Level 0 platforms, it remains optional.
IVRS	Signature Reserved (signature == "IVRS") **I/O Virtualization Reporting Structure** x86_64 (AMD) only table, will not be supported.
LPIT	Signature Reserved (signature == "LPIT") **Low Power Idle Table** x86 only table as of ACPI 5.1; starting with ACPI 6.0, processor descriptions and power states on ARM platforms should use the DSDT and define processor container devices (_HID ACPI0010, Section 8.4, and more specifically 8.4.3 and 8.4.4).
MADT	Section 5.2.12 (signature == "APIC") **Multiple APIC Description Table** Required for arm64. Only the GIC interrupt controller structures should be used (types 0xA - 0xF).
MCFG	Signature Reserved (signature == "MCFG") **Memory-mapped ConFiGuration space** If the platform supports PCI/PCIe, an MCFG table is required.
MCHI	Signature Reserved (signature == "MCHI") **Management Controller Host Interface table** Optional, not currently supported.
MPAM	Signature Reserved (signature == "MPAM") **Memory Partitioning And Monitoring table** This table allows the OS to discover the MPAM controls implemented by the subsystems.
MPST	Section 5.2.21 (signature == "MPST") **Memory Power State Table** Optional, not currently supported.

continues on next page

Table 1 – continued from previous page

Table	Usage for ARMv8 Linux
MSCT	Section 5.2.19 (signature == "MSCT") **Maximum System Characteristic Table** Optional, not currently supported.
MSDM	Signature Reserved (signature == "MSDM") **Microsoft Data Management table** Microsoft only table, will not be supported.
NFIT	Section 5.2.25 (signature == "NFIT") **NVDIMM Firmware Interface Table** Optional, not currently supported.
OEMx	Signature of "OEMx" only **OEM Specific Tables** All tables starting with a signature of "OEM" are reserved for OEM use. Since these are not meant to be of general use but are limited to very specific end users, they are not recommended for use and are not supported by the kernel for arm64.
PCCT	Section 14.1 (signature == "PCCT) **Platform Communications Channel Table** Recommend for use on arm64; use of PCC is recommended when using CPPC to control performance and power for platform processors.
PDTT	Section 5.2.29 (signature == "PDTT") **Platform Debug Trigger Table** This table describes PCC channels used to gather debug logs of non-architectural features.
PMTT	Section 5.2.21.12 (signature == "PMTT") **Platform Memory Topology Table** Optional, not currently supported.
PPTT	Section 5.2.30 (signature == "PPTT") **Processor Properties Topology Table** This table provides the processor and cache topology.
PSDT	Section 5.2.11.3 (signature == "PSDT") **Persistent System Description Table** Obsolete table, will not be supported.
RAS2	Section 5.2.21 (signature == "RAS2") **RAS Features 2 table** This table provides interfaces for the RAS capabilities implemented in the platform.
RASF	Section 5.2.20 (signature == "RASF") **RAS Feature table** Optional, not currently supported.
RSDP	Section 5.2.5 (signature == "RSD PTR") **Root System Description PoinTeR** Required for arm64.

continues on next page

Table 1 – continued from previous page

Table	Usage for ARMv8 Linux
RSDT	Section 5.2.7 (signature == "RSDT") **Root System Description Table** Since this table can only provide 32-bit addresses, it is deprecated on arm64, and will not be used. If provided, it will be ignored.
SBST	Section 5.2.14 (signature == "SBST") **Smart Battery Subsystem Table** Optional, not currently supported.
SDEI	Signature Reserved (signature == "SDEI") **Software Delegated Exception Interface table** This table advertises the presence of the SDEI interface.
SLIC	Signature Reserved (signature == "SLIC") **Software LIcensing table** Microsoft only table, will not be supported.
SLIT	Section 5.2.17 (signature == "SLIT") **System Locality distance Information Table** Optional in general, but required for NUMA systems.
SPCR	Signature Reserved (signature == "SPCR") **Serial Port Console Redirection table** Required for arm64.
SPMI	Signature Reserved (signature == "SPMI") **Server Platform Management Interface table** Optional, not currently supported.
SRAT	Section 5.2.16 (signature == "SRAT") **System Resource Affinity Table** Optional, but if used, only the GICC Affinity structures are read. To support arm64 NUMA, this table is required.
SSDT	Section 5.2.11.2 (signature == "SSDT") **Secondary System Description Table** These tables are a continuation of the DSDT; these are recommended for use with devices that can be added to a running system, but can also serve the purpose of dividing up device descriptions into more manageable pieces. An SSDT can only ADD to the ACPI namespace. It cannot modify or replace existing device descriptions already in the namespace. These tables are optional, however. ACPI tables should contain only one DSDT but can contain many SSDTs.

continues on next page

Table 1 – continued from previous page

Table	Usage for ARMv8 Linux
STAO	Signature Reserved (signature == "STAO") **_STA Override table** Optional, but only necessary in virtualized environments in order to hide devices from guest OSs.
TCPA	Signature Reserved (signature == "TCPA") **Trusted Computing Platform Alliance table** Optional, not currently supported, and may need changes to fully interoperate with arm64.
TPM2	Signature Reserved (signature == "TPM2") **Trusted Platform Module 2 table** Optional, not currently supported, and may need changes to fully interoperate with arm64.
UEFI	Signature Reserved (signature == "UEFI") **UEFI ACPI data table** Optional, not currently supported. No known use case for arm64, at present.
WAET	Signature Reserved (signature == "WAET") **Windows ACPI Emulated devices Table** Microsoft only table, will not be supported.
WDAT	Signature Reserved (signature == "WDAT") **Watch Dog Action Table** Microsoft only table, will not be supported.
WDRT	Signature Reserved (signature == "WDRT") **Watch Dog Resource Table** Microsoft only table, will not be supported.
WPBT	Signature Reserved (signature == "WPBT") **Windows Platform Binary Table** Microsoft only table, will not be supported.
XENV	Signature Reserved (signature == "XENV") **Xen project table** Optional, used only by Xen at present.
XSDT	Section 5.2.8 (signature == "XSDT") **eXtended System Description Table** Required for arm64.

3.1.1 ACPI Objects

The expectations on individual ACPI objects that are likely to be used are shown in the list that follows; any object not explicitly mentioned below should be used as needed for a particular platform or particular subsystem, such as power management or PCI.

Name	Section	Usage for ARMv8 Linux
_CCA	6.2.17	This method must be defined for all bus masters on arm64 - there are no assumptions made about whether such devices are cache coherent or not. The _CCA value is inherited by all descendants of these devices so it does not need to be repeated. Without _CCA on arm64, the kernel does not know what to do about setting up DMA for the device. NB: this method provides default cache coherency attributes; the presence of an SMMU can be used to modify that, however. For example, a master could default to non-coherent, but be made coherent with the appropriate SMMU configuration (see Table 17 of the IORT specification, ARM Document DEN 0049B).
_CID	6.1.2	Use as needed, see also _HID.
_CLS	6.1.3	Use as needed, see also _HID.
_CPC	8.4.7.1	Use as needed, power management specific. CPPC is recommended on arm64.
_CRS	6.2.2	Required on arm64.
_CSD	8.4.2.2	Use as needed, used only in conjunction with _CST.
_CST	8.4.2.1	Low power idle states (8.4.4) are recommended instead of C-states.
_DDN	6.1.4	This field can be used for a device name. However, it is meant for DOS device names (e.g., COM1), so be careful of its use across OSes.

continues on next page

Table 2 – continued from previous page

Name	Section	Usage for ARMv8 Linux
_DSD	6.2.5	To be used with caution. If this object is used, try to use it within the constraints already defined by the Device Properties UUID. Only in rare circumstances should it be necessary to create a new _DSD UUID. In either case, submit the _DSD definition along with any driver patches for discussion, especially when device properties are used. A driver will not be considered complete without a corresponding _DSD description. Once approved by kernel maintainers, the UUID or device properties must then be registered with the UEFI Forum; this may cause some iteration as more than one OS will be registering entries.
_DSM	9.1.1	Do not use this method. It is not standardized, the return values are not well documented, and it is currently a frequent source of error.
_GL	5.7.1	This object is not to be used in hardware reduced mode, and therefore should not be used on arm64.
_GLK	6.5.7	This object requires a global lock be defined; there is no global lock on arm64 since it runs in hardware reduced mode. Hence, do not use this object on arm64.
_GPE	5.3.1	This namespace is for x86 use only. Do not use it on arm64.
_HID	6.1.5	This is the primary object to use in device probing, though _CID and _CLS may also be used.

continues on next page

Table 2 – continued from previous page

Name	Section	Usage for ARMv8 Linux
_INI	6.5.1	Not required, but can be useful in setting up devices when UEFI leaves them in a state that may not be what the driver expects before it starts probing.
_LPI	8.4.4.3	Recommended for use with processor definitions (_HID ACPI0010) on arm64. See also _RDI.
_MLS	6.1.7	Highly recommended for use in internationalization.
_OFF	7.2.2	It is recommended to define this method for any device that can be turned on or off.
_ON	7.2.3	It is recommended to define this method for any device that can be turned on or off.
_OS	5.7.3	This method will return "Linux" by default (this is the value of the macro ACPI_OS_NAME on Linux). The command line parameter acpi_os=<string> can be used to set it to some other value.
_OSC	6.2.11	This method can be a global method in ACPI (i.e., _SB._OSC), or it may be associated with a specific device (e.g., _SB.DEV0._OSC), or both. When used as a global method, only capabilities published in the ACPI specification are allowed. When used as a device-specific method, the process described for using _DSD MUST be used to create an _OSC definition; out-of-process use of _OSC is not allowed. That is, submit the device-specific _OSC usage description as part of the kernel driver submission, get it approved by the kernel community, then register it with the UEFI Forum.

continues on next page

Table 2 – continued from previous page

Name	Section	Usage for ARMv8 Linux
_OSI	5.7.2	Deprecated on ARM64. As far as ACPI firmware is concerned, _OSI is not to be used to determine what sort of system is being used or what functionality is provided. The _OSC method is to be used instead.
_PDC	8.4.1	Deprecated, do not use on arm64.
_PIC	5.8.1	The method should not be used. On arm64, the only interrupt model available is GIC.
_PR	5.3.1	This namespace is for x86 use only on legacy systems. Do not use it on arm64.
_PRT	6.2.13	Required as part of the definition of all PCI root devices.
_PRx	7.3.8-11	Use as needed; power management specific. If _PR0 is defined, _PR3 must also be defined.
_PSx	7.3.2-5	Use as needed; power management specific. If _PS0 is defined, _PS3 must also be defined. If clocks or regulators need adjusting to be consistent with power usage, change them in these methods.
_RDI	8.4.4.4	Recommended for use with processor definitions (_HID ACPI0010) on arm64. This should only be used in conjunction with _LPI.
_REV	5.7.4	Always returns the latest version of ACPI supported.
_SB	5.3.1	Required on arm64; all devices must be defined in this namespace.
_SLI	6.2.15	Use is recommended when SLIT table is in use.

continues on next page

Table 2 – continued from previous page

Name	Section	Usage for ARMv8 Linux
_STA	6.3.7, 7.2.4	It is recommended to define this method for any device that can be turned on or off. See also the STAO table that provides overrides to hide devices in virtualized environments.
_SRS	6.2.16	Use as needed; see also _PRS.
_STR	6.1.10	Recommended for conveying device names to end users; this is preferred over using _DDN.
_SUB	6.1.9	Use as needed; _HID or _CID are preferred.
_SUN	6.1.11	Use as needed, but recommended.
_SWS	7.4.3	Use as needed; power management specific; this may require specification changes for use on arm64.
_UID	6.1.12	Recommended for distinguishing devices of the same class; define it if at all possible.

3.1.2 ACPI Event Model

Do not use GPE block devices; these are not supported in the hardware reduced profile used by arm64. Since there are no GPE blocks defined for use on ARM platforms, ACPI events must be signaled differently.

There are two options: GPIO-signaled interrupts (Section 5.6.5), and interrupt-signaled events (Section 5.6.9). Interrupt-signaled events are a new feature in the ACPI 6.1 specification. Either - or both - can be used on a given platform, and which to use may be dependent of limitations in any given SoC. If possible, interrupt-signaled events are recommended.

3.1.3 ACPI Processor Control

Section 8 of the ACPI specification changed significantly in version 6.0. Processors should now be defined as Device objects with _HID ACPI0007; do not use the deprecated Processor statement in ASL. All multiprocessor systems should also define a hierarchy of processors, done with Processor Container Devices (see Section 8.4.3.1, _HID ACPI0010); do not use processor aggregator devices (Section 8.5) to describe processor topology. Section 8.4 of the specification describes the semantics of these object definitions and how they interrelate.

Most importantly, the processor hierarchy defined also defines the low power idle states that are available to the platform, along with the rules for determining which processors can be turned on or off and the circumstances that control that. Without this information, the processors will run in whatever power state they were left in by UEFI.

Note too, that the processor Device objects defined and the entries in the MADT for GICs are expected to be in synchronization. The _UID of the Device object must correspond to processor IDs used in the MADT.

It is recommended that CPPC (8.4.5) be used as the primary model for processor performance control on arm64. C-states and P-states may become available at some point in the future, but most current design work appears to favor CPPC.

Further, it is essential that the ARMv8 SoC provide a fully functional implementation of PSCI; this will be the only mechanism supported by ACPI to control CPU power state. Booting of secondary CPUs using the ACPI parking protocol is possible, but discouraged, since only PSCI is supported for ARM servers.

3.1.4 ACPI System Address Map Interfaces

In Section 15 of the ACPI specification, several methods are mentioned as possible mechanisms for conveying memory resource information to the kernel. For arm64, we will only support UEFI for booting with ACPI, hence the UEFI GetMemoryMap() boot service is the only mechanism that will be used.

3.1.5 ACPI Platform Error Interfaces (APEI)

The APEI tables supported are described above.

APEI requires the equivalent of an SCI and an NMI on ARMv8. The SCI is used to notify the OSPM of errors that have occurred but can be corrected and the system can continue correct operation, even if possibly degraded. The NMI is used to indicate fatal errors that cannot be corrected, and require immediate attention.

Since there is no direct equivalent of the x86 SCI or NMI, arm64 handles these slightly differently. The SCI is handled as a high priority interrupt; given that these are corrected (or correctable) errors being reported, this is sufficient. The NMI is emulated as the highest priority interrupt possible. This implies some caution must be used since there could be interrupts at higher privilege levels or even interrupts at the same priority as the emulated NMI. In Linux, this should not be the case but one should be aware it could happen.

3.1.6 ACPI Objects Not Supported on ARM64

While this may change in the future, there are several classes of objects that can be defined, but are not currently of general interest to ARM servers. Some of these objects have x86 equivalents, and may actually make sense in ARM servers. However, there is either no hardware available at present, or there may not even be a non-ARM implementation yet. Hence, they are not currently supported.

The following classes of objects are not supported:

- Section 9.2: ambient light sensor devices
- Section 9.3: battery devices
- Section 9.4: lids (e.g., laptop lids)

- Section 9.8.2: IDE controllers
- Section 9.9: floppy controllers
- Section 9.10: GPE block devices
- Section 9.15: PC/AT RTC/CMOS devices
- Section 9.16: user presence detection devices
- Section 9.17: I/O APIC devices; all GICs must be enumerable via MADT
- Section 9.18: time and alarm devices (see 9.15)
- Section 10: power source and power meter devices
- Section 11: thermal management
- Section 12: embedded controllers interface
- Section 13: SMBus interfaces

This also means that there is no support for the following objects:

Name	Section	Name	Section
_ALC	9.3.4	_FDM	9.10.3
_ALI	9.3.2	_FIX	6.2.7
_ALP	9.3.6	_GAI	10.4.5
_ALR	9.3.5	_GHL	10.4.7
_ALT	9.3.3	_GTM	9.9.2.1.1
_BCT	10.2.2.10	_LID	9.5.1
_BDN	6.5.3	_PAI	10.4.4
_BIF	10.2.2.1	_PCL	10.3.2
_BIX	10.2.2.1	_PIF	10.3.3
_BLT	9.2.3	_PMC	10.4.1
_BMA	10.2.2.4	_PMD	10.4.8
_BMC	10.2.2.12	_PMM	10.4.3
_BMD	10.2.2.11	_PRL	10.3.4
_BMS	10.2.2.5	_PSR	10.3.1
_BST	10.2.2.6	_PTP	10.4.2
_BTH	10.2.2.7	_SBS	10.1.3
_BTM	10.2.2.9	_SHL	10.4.6
_BTP	10.2.2.8	_STM	9.9.2.1.1
_DCK	6.5.2	_UPD	9.16.1
_EC	12.12	_UPP	9.16.2
_FDE	9.10.1	_WPC	10.5.2
_FDI	9.10.2	_WPP	10.5.3

3.2 Activity Monitors Unit (AMU) extension in AArch64 Linux

Author: Ionela Voinescu <ionela.voinescu@arm.com>

Date: 2019-09-10

This document briefly describes the provision of Activity Monitors Unit support in AArch64 Linux.

3.2.1 Architecture overview

The activity monitors extension is an optional extension introduced by the ARMv8.4 CPU architecture.

The activity monitors unit, implemented in each CPU, provides performance counters intended for system management use. The AMU extension provides a system register interface to the counter registers and also supports an optional external memory-mapped interface.

Version 1 of the Activity Monitors architecture implements a counter group of four fixed and architecturally defined 64-bit event counters.

- CPU cycle counter: increments at the frequency of the CPU.
- Constant counter: increments at the fixed frequency of the system clock.
- Instructions retired: increments with every architecturally executed instruction.
- Memory stall cycles: counts instruction dispatch stall cycles caused by misses in the last level cache within the clock domain.

When in WFI or WFE these counters do not increment.

The Activity Monitors architecture provides space for up to 16 architected event counters. Future versions of the architecture may use this space to implement additional architected event counters.

Additionally, version 1 implements a counter group of up to 16 auxiliary 64-bit event counters.

On cold reset all counters reset to 0.

3.2.2 Basic support

The kernel can safely run a mix of CPUs with and without support for the activity monitors extension. Therefore, when CONFIG_ARM64_AMU_EXTN is selected we unconditionally enable the capability to allow any late CPU (secondary or hotplugged) to detect and use the feature.

When the feature is detected on a CPU, we flag the availability of the feature but this does not guarantee the correct functionality of the counters, only the presence of the extension.

Firmware (code running at higher exception levels, e.g. arm-tf) support is needed to:

- Enable access for lower exception levels (EL2 and EL1) to the AMU registers.
- Enable the counters. If not enabled these will read as 0.
- Save/restore the counters before/after the CPU is being put/brought up from the 'off' power state.

When using kernels that have this feature enabled but boot with broken firmware the user may experience panics or lockups when accessing the counter registers. Even if these symptoms are not observed, the values returned by the register reads might not correctly reflect reality. Most commonly, the counters will read as 0, indicating that they are not enabled.

If proper support is not provided in firmware it's best to disable CONFIG_ARM64_AMU_EXTN. To be noted that for security reasons, this does not bypass the setting of AMUSERENR_EL0 to trap accesses from EL0 (userspace) to EL1 (kernel). Therefore, firmware should still ensure accesses to AMU registers are not trapped in EL2/EL3.

The fixed counters of AMUv1 are accessible though the following system register definitions:

- SYS_AMEVCNTR0_CORE_EL0
- SYS_AMEVCNTR0_CONST_EL0
- SYS_AMEVCNTR0_INST_RET_EL0
- SYS_AMEVCNTR0_MEM_STALL_EL0

Auxiliary platform specific counters can be accessed using SYS_AMEVCNTR1_EL0(n), where n is a value between 0 and 15.

Details can be found in: arch/arm64/include/asm/sysreg.h.

3.2.3 Userspace access

Currently, access from userspace to the AMU registers is disabled due to:

- Security reasons: they might expose information about code executed in secure mode.
- Purpose: AMU counters are intended for system management use.

Also, the presence of the feature is not visible to userspace.

3.2.4 Virtualization

Currently, access from userspace (EL0) and kernelspace (EL1) on the KVM guest side is disabled due to:

- Security reasons: they might expose information about code executed by other guests or the host.

Any attempt to access the AMU registers will result in an UNDEFINED exception being injected into the guest.

3.3 ACPI on Arm systems

ACPI can be used for Armv8 and Armv9 systems designed to follow the BSA (Arm Base System Architecture) [0] and BBR (Arm Base Boot Requirements) [1] specifications. Both BSA and BBR are publicly accessible documents. Arm Servers, in addition to being BSA compliant, comply with a set of rules defined in SBSA (Server Base System Architecture) [2].

The Arm kernel implements the reduced hardware model of ACPI version 5.1 or later. Links to the specification and all external documents it refers to are managed by the UEFI Forum.

The specification is available at http://www.uefi.org/specifications and documents referenced by the specification can be found via http://www.uefi.org/acpi.

If an Arm system does not meet the requirements of the BSA and BBR, or cannot be described using the mechanisms defined in the required ACPI specifications, then ACPI may not be a good fit for the hardware.

While the documents mentioned above set out the requirements for building industry-standard Arm systems, they also apply to more than one operating system. The purpose of this document is to describe the interaction between ACPI and Linux only, on an Arm system -- that is, what Linux expects of ACPI and what ACPI can expect of Linux.

3.3.1 Why ACPI on Arm?

Before examining the details of the interface between ACPI and Linux, it is useful to understand why ACPI is being used. Several technologies already exist in Linux for describing non-enumerable hardware, after all. In this section we summarize a blog post [3] from Grant Likely that outlines the reasoning behind ACPI on Arm systems. Actually, we snitch a good portion of the summary text almost directly, to be honest.

The short form of the rationale for ACPI on Arm is:

- ACPI's byte code (AML) allows the platform to encode hardware behavior, while DT explicitly does not support this. For hardware vendors, being able to encode behavior is a key tool used in supporting operating system releases on new hardware.
- ACPI's OSPM defines a power management model that constrains what the platform is allowed to do into a specific model, while still providing flexibility in hardware design.
- In the enterprise server environment, ACPI has established bindings (such as for RAS) which are currently used in production systems. DT does not. Such bindings could be defined in DT at some point, but doing so means Arm and x86 would end up using completely different code paths in both firmware and the kernel.
- Choosing a single interface to describe the abstraction between a platform and an OS is important. Hardware vendors would not be required to implement both DT and ACPI if they want to support multiple operating systems. And, agreeing on a single interface instead of being fragmented into per OS interfaces makes for better interoperability overall.
- The new ACPI governance process works well and Linux is now at the same table as hardware vendors and other OS vendors. In fact, there is no longer any reason to feel that ACPI only belongs to Windows or that Linux is in any way secondary to Microsoft in this arena. The move of ACPI governance into the UEFI forum has significantly opened up the specification development process, and currently, a large portion of the changes being made to ACPI are being driven by Linux.

Key to the use of ACPI is the support model. For servers in general, the responsibility for hardware behaviour cannot solely be the domain of the kernel, but rather must be split between the platform and the kernel, in order to allow for orderly change over time. ACPI frees the OS from needing to understand all the minute details of the hardware so that the OS doesn't need to be ported to each and every device individually. It allows the hardware vendors to take responsibility for power management behaviour without depending on an OS release cycle which is not under their control.

ACPI is also important because hardware and OS vendors have already worked out the mechanisms for supporting a general purpose computing ecosystem. The infrastructure is in place, the bindings are in place, and the processes are in place. DT does exactly what Linux needs it to when working with vertically integrated devices, but there are no good processes for supporting what the server vendors need. Linux could potentially get there with DT, but doing so really just duplicates something that already works. ACPI already does what the hardware vendors need, Microsoft won't collaborate on DT, and hardware vendors would still end up providing two completely separate firmware interfaces -- one for Linux and one for Windows.

3.3.2 Kernel Compatibility

One of the primary motivations for ACPI is standardization, and using that to provide backward compatibility for Linux kernels. In the server market, software and hardware are often used for long periods. ACPI allows the kernel and firmware to agree on a consistent abstraction that can be maintained over time, even as hardware or software change. As long as the abstraction is supported, systems can be updated without necessarily having to replace the kernel.

When a Linux driver or subsystem is first implemented using ACPI, it by definition ends up requiring a specific version of the ACPI specification -- its baseline. ACPI firmware must continue to work, even though it may not be optimal, with the earliest kernel version that first provides support for that baseline version of ACPI. There may be a need for additional drivers, but adding new functionality (e.g., CPU power management) should not break older kernel versions. Further, ACPI firmware must also work with the most recent version of the kernel.

3.3.3 Relationship with Device Tree

ACPI support in drivers and subsystems for Arm should never be mutually exclusive with DT support at compile time.

At boot time the kernel will only use one description method depending on parameters passed from the boot loader (including kernel bootargs).

Regardless of whether DT or ACPI is used, the kernel must always be capable of booting with either scheme (in kernels with both schemes enabled at compile time).

3.3.4 Booting using ACPI tables

The only defined method for passing ACPI tables to the kernel on Arm is via the UEFI system configuration table. Just so it is explicit, this means that ACPI is only supported on platforms that boot via UEFI.

When an Arm system boots, it can either have DT information, ACPI tables, or in some very unusual cases, both. If no command line parameters are used, the kernel will try to use DT for device enumeration; if there is no DT present, the kernel will try to use ACPI tables, but only if they are present. In neither is available, the kernel will not boot. If acpi=force is used on the command line, the kernel will attempt to use ACPI tables first, but fall back to DT if there are no ACPI tables present. The basic idea is that the kernel will not fail to boot unless it absolutely has no other choice.

Processing of ACPI tables may be disabled by passing acpi=off on the kernel command line; this is the default behavior.

In order for the kernel to load and use ACPI tables, the UEFI implementation MUST set the ACPI_20_TABLE_GUID to point to the RSDP table (the table with the ACPI signature "RSD PTR

"). If this pointer is incorrect and acpi=force is used, the kernel will disable ACPI and try to use DT to boot instead; the kernel has, in effect, determined that ACPI tables are not present at that point.

If the pointer to the RSDP table is correct, the table will be mapped into the kernel by the ACPI core, using the address provided by UEFI.

The ACPI core will then locate and map in all other ACPI tables provided by using the addresses in the RSDP table to find the XSDT (eXtended System Description Table). The XSDT in turn provides the addresses to all other ACPI tables provided by the system firmware; the ACPI core will then traverse this table and map in the tables listed.

The ACPI core will ignore any provided RSDT (Root System Description Table). RSDTs have been deprecated and are ignored on arm64 since they only allow for 32-bit addresses.

Further, the ACPI core will only use the 64-bit address fields in the FADT (Fixed ACPI Description Table). Any 32-bit address fields in the FADT will be ignored on arm64.

Hardware reduced mode (see Section 4.1 of the ACPI 6.1 specification) will be enforced by the ACPI core on arm64. Doing so allows the ACPI core to run less complex code since it no longer has to provide support for legacy hardware from other architectures. Any fields that are not to be used for hardware reduced mode must be set to zero.

For the ACPI core to operate properly, and in turn provide the information the kernel needs to configure devices, it expects to find the following tables (all section numbers refer to the ACPI 6.5 specification):

- RSDP (Root System Description Pointer), section 5.2.5
- XSDT (eXtended System Description Table), section 5.2.8
- FADT (Fixed ACPI Description Table), section 5.2.9
- DSDT (Differentiated System Description Table), section 5.2.11.1
- MADT (Multiple APIC Description Table), section 5.2.12
- GTDT (Generic Timer Description Table), section 5.2.24
- PPTT (Processor Properties Topology Table), section 5.2.30
- DBG2 (DeBuG port table 2), section 5.2.6, specifically Table 5-6.
- APMT (Arm Performance Monitoring unit Table), section 5.2.6, specifically Table 5-6.
- AGDI (Arm Generic diagnostic Dump and Reset Device Interface Table), section 5.2.6, specifically Table 5-6.
- If PCI is supported, the MCFG (Memory mapped ConFiGuration Table), section 5.2.6, specifically Table 5-6.
- If booting without a console=<device> kernel parameter is supported, the SPCR (Serial Port Console Redirection table), section 5.2.6, specifically Table 5-6.
- If necessary to describe the I/O topology, SMMUs and GIC ITSs, the IORT (Input Output Remapping Table, section 5.2.6, specifically Table 5-6).
- If NUMA is supported, the following tables are required:
  - SRAT (System Resource Affinity Table), section 5.2.16
  - SLIT (System Locality distance Information Table), section 5.2.17

- If NUMA is supported, and the system contains heterogeneous memory, the HMAT (Heterogeneous Memory Attribute Table), section 5.2.28.
- If the ACPI Platform Error Interfaces are required, the following tables are conditionally required:
  - BERT (Boot Error Record Table, section 18.3.1)
  - EINJ (Error INJection table, section 18.6.1)
  - ERST (Error Record Serialization Table, section 18.5)
  - HEST (Hardware Error Source Table, section 18.3.2)
  - SDEI (Software Delegated Exception Interface table, section 5.2.6, specifically Table 5-6)
  - AEST (Arm Error Source Table, section 5.2.6, specifically Table 5-6)
  - RAS2 (ACPI RAS2 feature table, section 5.2.21)
- If the system contains controllers using PCC channel, the PCCT (Platform Communications Channel Table), section 14.1
- If the system contains a controller to capture board-level system state, and communicates with the host via PCC, the PDTT (Platform Debug Trigger Table), section 5.2.29.
- If NVDIMM is supported, the NFIT (NVDIMM Firmware Interface Table), section 5.2.26
- If video framebuffer is present, the BGRT (Boot Graphics Resource Table), section 5.2.23
- If IPMI is implemented, the SPMI (Server Platform Management Interface), section 5.2.6, specifically Table 5-6.
- If the system contains a CXL Host Bridge, the CEDT (CXL Early Discovery Table), section 5.2.6, specifically Table 5-6.
- If the system supports MPAM, the MPAM (Memory Partitioning And Monitoring table), section 5.2.6, specifically Table 5-6.
- If the system lacks persistent storage, the IBFT (ISCSI Boot Firmware Table), section 5.2.6, specifically Table 5-6.

If the above tables are not all present, the kernel may or may not be able to boot properly since it may not be able to configure all of the devices available. This list of tables is not meant to be all inclusive; in some environments other tables may be needed (e.g., any of the APEI tables from section 18) to support specific functionality.

3.3.5 ACPI Detection

Drivers should determine their probe() type by checking for a null value for ACPI_HANDLE, or checking .of_node, or other information in the device structure. This is detailed further in the "Driver Recommendations" section.

In non-driver code, if the presence of ACPI needs to be detected at run time, then check the value of acpi_disabled. If CONFIG_ACPI is not set, acpi_disabled will always be 1.

3.3.6 Device Enumeration

Device descriptions in ACPI should use standard recognized ACPI interfaces. These may contain less information than is typically provided via a Device Tree description for the same device. This is also one of the reasons that ACPI can be useful -- the driver takes into account that it may have less detailed information about the device and uses sensible defaults instead. If done properly in the driver, the hardware can change and improve over time without the driver having to change at all.

Clocks provide an excellent example. In DT, clocks need to be specified and the drivers need to take them into account. In ACPI, the assumption is that UEFI will leave the device in a reasonable default state, including any clock settings. If for some reason the driver needs to change a clock value, this can be done in an ACPI method; all the driver needs to do is invoke the method and not concern itself with what the method needs to do to change the clock. Changing the hardware can then take place over time by changing what the ACPI method does, and not the driver.

In DT, the parameters needed by the driver to set up clocks as in the example above are known as “bindings”; in ACPI, these are known as “Device Properties” and provided to a driver via the _DSD object.

ACPI tables are described with a formal language called ASL, the ACPI Source Language (section 19 of the specification). This means that there are always multiple ways to describe the same thing -- including device properties. For example, device properties could use an ASL construct that looks like this: Name(KEY0, “value0”). An ACPI device driver would then retrieve the value of the property by evaluating the KEY0 object. However, using Name() this way has multiple problems: (1) ACPI limits names (“KEY0”) to four characters unlike DT; (2) there is no industry wide registry that maintains a list of names, minimizing re-use; (3) there is also no registry for the definition of property values (“value0”), again making re-use difficult; and (4) how does one maintain backward compatibility as new hardware comes out? The _DSD method was created to solve precisely these sorts of problems; Linux drivers should ALWAYS use the _DSD method for device properties and nothing else.

The _DSM object (ACPI Section 9.14.1) could also be used for conveying device properties to a driver. Linux drivers should only expect it to be used if _DSD cannot represent the data required, and there is no way to create a new UUID for the _DSD object. Note that there is even less regulation of the use of _DSM than there is of _DSD. Drivers that depend on the contents of _DSM objects will be more difficult to maintain over time because of this; as of this writing, the use of _DSM is the cause of quite a few firmware problems and is not recommended.

Drivers should look for device properties in the _DSD object ONLY; the _DSD object is described in the ACPI specification section 6.2.5, but this only describes how to define the structure of an object returned via _DSD, and how specific data structures are defined by specific UUIDs. Linux should only use the _DSD Device Properties UUID [4]:

- UUID: daffd814-6eba-4d8c-8a91-bc9bbf4aa301

Common device properties can be registered by creating a pull request to [4] so that they may be used across all operating systems supporting ACPI. Device properties that have not been registered with the UEFI Forum can be used but not as “uefi-” common properties.

Before creating new device properties, check to be sure that they have not been defined before and either registered in the Linux kernel documentation as DT bindings, or the UEFI Forum as device properties. While we do not want to simply move all DT bindings into ACPI device properties, we can learn from what has been previously defined.

If it is necessary to define a new device property, or if it makes sense to synthesize the definition of a binding so it can be used in any firmware, both DT bindings and ACPI device properties for device drivers have review processes. Use them both. When the driver itself is submitted for review to the Linux mailing lists, the device property definitions needed must be submitted at the same time. A driver that supports ACPI and uses device properties will not be considered complete without their definitions. Once the device property has been accepted by the Linux community, it must be registered with the UEFI Forum [4], which will review it again for consistency within the registry. This may require iteration. The UEFI Forum, though, will always be the canonical site for device property definitions.

It may make sense to provide notice to the UEFI Forum that there is the intent to register a previously unused device property name as a means of reserving the name for later use. Other operating system vendors will also be submitting registration requests and this may help smooth the process.

Once registration and review have been completed, the kernel provides an interface for looking up device properties in a manner independent of whether DT or ACPI is being used. This API should be used [5]; it can eliminate some duplication of code paths in driver probing functions and discourage divergence between DT bindings and ACPI device properties.

3.3.7 Programmable Power Control Resources

Programmable power control resources include such resources as voltage/current providers (regulators) and clock sources.

With ACPI, the kernel clock and regulator framework is not expected to be used at all.

The kernel assumes that power control of these resources is represented with Power Resource Objects (ACPI section 7.1). The ACPI core will then handle correctly enabling and disabling resources as they are needed. In order to get that to work, ACPI assumes each device has defined D-states and that these can be controlled through the optional ACPI methods _PS0, _PS1, _PS2, and _PS3; in ACPI, _PS0 is the method to invoke to turn a device full on, and _PS3 is for turning a device full off.

There are two options for using those Power Resources. They can:

- be managed in a _PSx method which gets called on entry to power state Dx.
- be declared separately as power resources with their own _ON and _OFF methods. They are then tied back to D-states for a particular device via _PRx which specifies which power resources a device needs to be on while in Dx. Kernel then tracks number of devices using a power resource and calls _ON/_OFF as needed.

The kernel ACPI code will also assume that the _PSx methods follow the normal ACPI rules for such methods:

- If either _PS0 or _PS3 is implemented, then the other method must also be implemented.
- If a device requires usage or setup of a power resource when on, the ASL should organize that it is allocated/enabled using the _PS0 method.
- Resources allocated or enabled in the _PS0 method should be disabled or de-allocated in the _PS3 method.
- Firmware will leave the resources in a reasonable state before handing over control to the kernel.

Such code in _PSx methods will of course be very platform specific. But, this allows the driver to abstract out the interface for operating the device and avoid having to read special non-standard values from ACPI tables. Further, abstracting the use of these resources allows the hardware to change over time without requiring updates to the driver.

3.3.8 Clocks

ACPI makes the assumption that clocks are initialized by the firmware -- UEFI, in this case -- to some working value before control is handed over to the kernel. This has implications for devices such as UARTs, or SoC-driven LCD displays, for example.

When the kernel boots, the clocks are assumed to be set to reasonable working values. If for some reason the frequency needs to change -- e.g., throttling for power management -- the device driver should expect that process to be abstracted out into some ACPI method that can be invoked (please see the ACPI specification for further recommendations on standard methods to be expected). The only exceptions to this are CPU clocks where CPPC provides a much richer interface than ACPI methods. If the clocks are not set, there is no direct way for Linux to control them.

If an SoC vendor wants to provide fine-grained control of the system clocks, they could do so by providing ACPI methods that could be invoked by Linux drivers. However, this is NOT recommended and Linux drivers should NOT use such methods, even if they are provided. Such methods are not currently standardized in the ACPI specification, and using them could tie a kernel to a very specific SoC, or tie an SoC to a very specific version of the kernel, both of which we are trying to avoid.

3.3.9 Driver Recommendations

DO NOT remove any DT handling when adding ACPI support for a driver. The same device may be used on many different systems.

DO try to structure the driver so that it is data-driven. That is, set up a struct containing internal per-device state based on defaults and whatever else must be discovered by the driver probe function. Then, have the rest of the driver operate off of the contents of that struct. Doing so should allow most divergence between ACPI and DT functionality to be kept local to the probe function instead of being scattered throughout the driver. For example:

```
static int device_probe_dt(struct platform_device *pdev)
{
       /* DT specific functionality */
       ...
}

static int device_probe_acpi(struct platform_device *pdev)
{
       /* ACPI specific functionality */
       ...
}

static int device_probe(struct platform_device *pdev)
{
       ...
       struct device_node node = pdev->dev.of_node;
       ...

       if (node)
               ret = device_probe_dt(pdev);
       else if (ACPI_HANDLE(&pdev->dev))
               ret = device_probe_acpi(pdev);
       else
               /* other initialization */
               ...
       /* Continue with any generic probe operations */
       ...
}
```

DO keep the MODULE_DEVICE_TABLE entries together in the driver to make it clear the different names the driver is probed for, both from DT and from ACPI:

```
static struct of_device_id virtio_mmio_match[] = {
        { .compatible = "virtio,mmio", },
        { }
};
MODULE_DEVICE_TABLE(of, virtio_mmio_match);

static const struct acpi_device_id virtio_mmio_acpi_match[] = {
        { "LNRO0005", },
        { }
};
MODULE_DEVICE_TABLE(acpi, virtio_mmio_acpi_match);
```

3.3.10 ASWG

The ACPI specification changes regularly. During the year 2014, for instance, version 5.1 was released and version 6.0 substantially completed, with most of the changes being driven by Arm-specific requirements. Proposed changes are presented and discussed in the ASWG (ACPI Specification Working Group) which is a part of the UEFI Forum. The current version of the ACPI specification is 6.5 release in August 2022.

Participation in this group is open to all UEFI members. Please see http://www.uefi.org/workinggroup for details on group membership.

It is the intent of the Arm ACPI kernel code to follow the ACPI specification as closely as possible, and to only implement functionality that complies with the released standards from UEFI ASWG. As a practical matter, there will be vendors that provide bad ACPI tables or violate the standards in some way. If this is because of errors, quirks and fix-ups may be necessary, but will be avoided if possible. If there are features missing from ACPI that preclude it from being used on a platform, ECRs (Engineering Change Requests) should be submitted to ASWG and go through the normal approval process; for those that are not UEFI members, many other members of the Linux community are and would likely be willing to assist in submitting ECRs.

3.3.11 Linux Code

Individual items specific to Linux on Arm, contained in the Linux source code, are in the list that follows:

ACPI_OS_NAME
: This macro defines the string to be returned when an ACPI method invokes the _OS method. On Arm systems, this macro will be "Linux" by default. The command line parameter acpi_os=<string> can be used to set it to some other value. The default value for other architectures is "Microsoft Windows NT", for example.

3.3.12 ACPI Objects

Detailed expectations for ACPI tables and object are listed in the file *ACPI Tables*.

3.3.13 References

[0] https://developer.arm.com/documentation/den0094/latest
document Arm-DEN-0094: "Arm Base System Architecture", version 1.0C, dated 6 Oct 2022

[1] https://developer.arm.com/documentation/den0044/latest
Document Arm-DEN-0044: "Arm Base Boot Requirements", version 2.0G, dated 15 Apr 2022

[2] https://developer.arm.com/documentation/den0029/latest
Document Arm-DEN-0029: "Arm Server Base System Architecture", version 7.1, dated 06 Oct 2022

[3] http://www.secretlab.ca/archives/151,
10 Jan 2015, Copyright (c) 2015, Linaro Ltd., written by Grant Likely.

[4] _DSD (Device Specific Data) Implementation Guide
https://github.com/UEFI/DSD-Guide/blob/main/dsd-guide.pdf

[5] Kernel code for the unified device
property interface can be found in include/linux/property.h and drivers/base/property.c.

3.3.14 Authors

- Al Stone <al.stone@linaro.org>
- Graeme Gregory <graeme.gregory@linaro.org>
- Hanjun Guo <hanjun.guo@linaro.org>
- Grant Likely <grant.likely@linaro.org>, for the "Why ACPI on ARM?" section

3.4 Asymmetric 32-bit SoCs

Author: Will Deacon <will@kernel.org>

This document describes the impact of asymmetric 32-bit SoCs on the execution of 32-bit (`AArch32`) applications.

Date: 2021-05-17

3.4.1 Introduction

Some Armv9 SoCs suffer from a big.LITTLE misfeature where only a subset of the CPUs are capable of executing 32-bit user applications. On such a system, Linux by default treats the asymmetry as a "mismatch" and disables support for both the `PER_LINUX32` personality and `execve(2)` of 32-bit ELF binaries, with the latter returning `-ENOEXEC`. If the mismatch is detected during late onlining of a 64-bit-only CPU, then the onlining operation fails and the new CPU is unavailable for scheduling.

Surprisingly, these SoCs have been produced with the intention of running legacy 32-bit binaries. Unsurprisingly, that doesn't work very well with the default behaviour of Linux.

It seems inevitable that future SoCs will drop 32-bit support altogether, so if you're stuck in the unenviable position of needing to run 32-bit code on one of these transitionary platforms then you would be wise to consider alternatives such as recompilation, emulation or retirement. If neither of those options are practical, then read on.

3.4.2 Enabling kernel support

Since the kernel support is not completely transparent to userspace, allowing 32-bit tasks to run on an asymmetric 32-bit system requires an explicit "opt-in" and can be enabled by passing the `allow_mismatched_32bit_el0` parameter on the kernel command-line.

For the remainder of this document we will refer to an *asymmetric system* to mean an asymmetric 32-bit SoC running Linux with this kernel command-line option enabled.

3.4.3 Userspace impact

32-bit tasks running on an asymmetric system behave in mostly the same way as on a homogeneous system, with a few key differences relating to CPU affinity.

sysfs

The subset of CPUs capable of running 32-bit tasks is described in `/sys/devices/system/cpu/aarch32_el0` and is documented further in `Documentation/ABI/testing/sysfs-devices-system-cpu`.

Note: CPUs are advertised by this file as they are detected and so late-onlining of 32-bit-capable CPUs can result in the file contents being modified by the kernel at runtime. Once advertised, CPUs are never removed from the file.

`execve(2)`

On a homogeneous system, the CPU affinity of a task is preserved across `execve(2)`. This is not always possible on an asymmetric system, specifically when the new program being executed is 32-bit yet the affinity mask contains 64-bit-only CPUs. In this situation, the kernel determines the new affinity mask as follows:

1. If the 32-bit-capable subset of the affinity mask is not empty, then the affinity is restricted to that subset and the old affinity mask is saved. This saved mask is inherited over `fork(2)` and preserved across `execve(2)` of 32-bit programs.

   Note: This step does not apply to `SCHED_DEADLINE` tasks. See *SCHED_DEADLINE*.

2. Otherwise, the cpuset hierarchy of the task is walked until an ancestor is found containing at least one 32-bit-capable CPU. The affinity of the task is then changed to match the 32-bit-capable subset of the cpuset determined by the walk.

3. On failure (i.e. out of memory), the affinity is changed to the set of all 32-bit-capable CPUs of which the kernel is aware.

A subsequent `execve(2)` of a 64-bit program by the 32-bit task will invalidate the affinity mask saved in (1) and attempt to restore the CPU affinity of the task using the saved mask if it was previously valid. This restoration may fail due to intervening changes to the deadline policy or cpuset hierarchy, in which case the `execve(2)` continues with the affinity unchanged.

Calls to `sched_setaffinity(2)` for a 32-bit task will consider only the 32-bit-capable CPUs of the requested affinity mask. On success, the affinity for the task is updated and any saved mask from a prior `execve(2)` is invalidated.

`SCHED_DEADLINE`

Explicit admission of a 32-bit deadline task to the default root domain (e.g. by calling `sched_setattr(2)`) is rejected on an asymmetric 32-bit system unless admission control is disabled by writing -1 to `/proc/sys/kernel/sched_rt_runtime_us`.

`execve(2)` of a 32-bit program from a 64-bit deadline task will return `-ENOEXEC` if the root domain for the task contains any 64-bit-only CPUs and admission control is enabled. Concurrent offlining of 32-bit-capable CPUs may still necessitate the procedure described in *execve(2)*, in which case step (1) is skipped and a warning is emitted on the console.

Note: It is recommended that a set of 32-bit-capable CPUs are placed into a separate root domain if `SCHED_DEADLINE` is to be used with 32-bit tasks on an asymmetric system. Failure to do so is likely to result in missed deadlines.

Cpusets

The affinity of a 32-bit task on an asymmetric system may include CPUs that are not explicitly allowed by the cpuset to which it is attached. This can occur as a result of the following two situations:

- A 64-bit task attached to a cpuset which allows only 64-bit CPUs executes a 32-bit program.
- All of the 32-bit-capable CPUs allowed by a cpuset containing a 32-bit task are offlined.

In both of these cases, the new affinity is calculated according to step (2) of the process described in *execve(2)* and the cpuset hierarchy is unchanged irrespective of the cgroup version.

CPU hotplug

On an asymmetric system, the first detected 32-bit-capable CPU is prevented from being offlined by userspace and any such attempt will return `-EPERM`. Note that suspend is still permitted even if the primary CPU (i.e. CPU 0) is 64-bit-only.

KVM

Although KVM will not advertise 32-bit EL0 support to any vCPUs on an asymmetric system, a broken guest at EL1 could still attempt to execute 32-bit code at EL0. In this case, an exit from a vCPU thread in 32-bit mode will return to host userspace with an `exit_reason` of `KVM_EXIT_FAIL_ENTRY` and will remain non-runnable until successfully re-initialised by a subsequent `KVM_ARM_VCPU_INIT` operation.

3.5 Booting AArch64 Linux

Author: Will Deacon <will.deacon@arm.com>

Date : 07 September 2012

This document is based on the ARM booting document by Russell King and is relevant to all public releases of the AArch64 Linux kernel.

The AArch64 exception model is made up of a number of exception levels (EL0 - EL3), with EL0, EL1 and EL2 having a secure and a non-secure counterpart. EL2 is the hypervisor level, EL3 is the highest priority level and exists only in secure mode. Both are architecturally optional.

For the purposes of this document, we will use the term *boot loader* simply to define all software that executes on the CPU(s) before control is passed to the Linux kernel. This may include secure monitor and hypervisor code, or it may just be a handful of instructions for preparing a minimal boot environment.

Essentially, the boot loader should provide (as a minimum) the following:

1. Setup and initialise the RAM
2. Setup the device tree
3. Decompress the kernel image
4. Call the kernel image

3.5.1 1. Setup and initialise RAM

Requirement: MANDATORY

The boot loader is expected to find and initialise all RAM that the kernel will use for volatile data storage in the system. It performs this in a machine dependent manner. (It may use internal algorithms to automatically locate and size all RAM, or it may use knowledge of the RAM in the machine, or any other method the boot loader designer sees fit.)

3.5.2 2. Setup the device tree

Requirement: MANDATORY

The device tree blob (dtb) must be placed on an 8-byte boundary and must not exceed 2 megabytes in size. Since the dtb will be mapped cacheable using blocks of up to 2 megabytes in size, it must not be placed within any 2M region which must be mapped with any specific attributes.

NOTE: versions prior to v4.2 also require that the DTB be placed within the 512 MB region starting at text_offset bytes below the kernel Image.

3.5.3 3. Decompress the kernel image

Requirement: OPTIONAL

The AArch64 kernel does not currently provide a decompressor and therefore requires decompression (gzip etc.) to be performed by the boot loader if a compressed Image target (e.g. Image.gz) is used. For bootloaders that do not implement this requirement, the uncompressed Image target is available instead.

3.5.4 4. Call the kernel image

Requirement: MANDATORY

The decompressed kernel image contains a 64-byte header as follows:

```
u32 code0;                      /* Executable code */
u32 code1;                      /* Executable code */
u64 text_offset;                /* Image load offset, little endian */
u64 image_size;                 /* Effective Image size, little endian */
u64 flags;                      /* kernel flags, little endian */
u64 res2      = 0;              /* reserved */
u64 res3      = 0;              /* reserved */
u64 res4      = 0;              /* reserved */
u32 magic     = 0x644d5241;     /* Magic number, little endian, "ARM\x64" */
u32 res5;                       /* reserved (used for PE COFF offset) */
```

Header notes:

- As of v3.17, all fields are little endian unless stated otherwise.
- code0/code1 are responsible for branching to stext.
- when booting through EFI, code0/code1 are initially skipped. res5 is an offset to the PE header and the PE header has the EFI entry point (efi_stub_entry). When the stub has done its work, it jumps to code0 to resume the normal boot process.
- Prior to v3.17, the endianness of text_offset was not specified. In these cases image_size is zero and text_offset is 0x80000 in the endianness of the kernel. Where image_size is non-zero image_size is little-endian and must be respected. Where image_size is zero, text_offset can be assumed to be 0x80000.
- The flags field (introduced in v3.17) is a little-endian 64-bit field composed as follows:

Bit 0	Kernel endianness. 1 if BE, 0 if LE.
Bit 1-2	Kernel Page size. – 0 - Unspecified. – 1 - 4K – 2 - 16K – 3 - 64K
Bit 3	Kernel physical placement **0** 2MB aligned base should be as close as possible to the base of DRAM, since memory below it is not accessible via the linear mapping **1** 2MB aligned base such that all image_size bytes counted from the start of the image are within the 48-bit addressable range of physical memory
Bits 4-63	Reserved.

- When image_size is zero, a bootloader should attempt to keep as much memory as possible free for use by the kernel immediately after the end of the kernel image. The amount of space required will vary depending on selected features, and is effectively unbound.

The Image must be placed text_offset bytes from a 2MB aligned base address anywhere in usable system RAM and called there. The region between the 2 MB aligned base address and the start of the image has no special significance to the kernel, and may be used for other purposes. At least image_size bytes from the start of the image must be free for use by the kernel. NOTE: versions prior to v4.6 cannot make use of memory below the physical offset of the Image so it is recommended that the Image be placed as close as possible to the start of system RAM.

If an initrd/initramfs is passed to the kernel at boot, it must reside entirely within a 1 GB aligned physical memory window of up to 32 GB in size that fully covers the kernel Image as well.

Any memory described to the kernel (even that below the start of the image) which is not marked as reserved from the kernel (e.g., with a memreserve region in the device tree) will be considered as available to the kernel.

Before jumping into the kernel, the following conditions must be met:

- Quiesce all DMA capable devices so that memory does not get corrupted by bogus network packets or disk data. This will save you many hours of debug.
- Primary CPU general-purpose register settings:
  - x0 = physical address of device tree blob (dtb) in system RAM.
  - x1 = 0 (reserved for future use)
  - x2 = 0 (reserved for future use)

  - x3 = 0 (reserved for future use)

- CPU mode

  All forms of interrupts must be masked in PSTATE.DAIF (Debug, SError, IRQ and FIQ). The CPU must be in non-secure state, either in EL2 (RECOMMENDED in order to have access to the virtualisation extensions), or in EL1.

- Caches, MMUs

  The MMU must be off.

  The instruction cache may be on or off, and must not hold any stale entries corresponding to the loaded kernel image.

  The address range corresponding to the loaded kernel image must be cleaned to the PoC. In the presence of a system cache or other coherent masters with caches enabled, this will typically require cache maintenance by VA rather than set/way operations. System caches which respect the architected cache maintenance by VA operations must be configured and may be enabled. System caches which do not respect architected cache maintenance by VA operations (not recommended) must be configured and disabled.

- Architected timers

  CNTFRQ must be programmed with the timer frequency and CNTVOFF must be programmed with a consistent value on all CPUs. If entering the kernel at EL1, CNTHCTL_EL2 must have EL1PCTEN (bit 0) set where available.

- Coherency

  All CPUs to be booted by the kernel must be part of the same coherency domain on entry to the kernel. This may require IMPLEMENTATION DEFINED initialisation to enable the receiving of maintenance operations on each CPU.

- System registers

  All writable architected system registers at or below the exception level where the kernel image will be entered must be initialised by software at a higher exception level to prevent execution in an UNKNOWN state.

  For all systems: - If EL3 is present:

  - SCR_EL3.FIQ must have the same value across all CPUs the kernel is executing on.
  - The value of SCR_EL3.FIQ must be the same as the one present at boot time whenever the kernel is executing.
  - If EL3 is present and the kernel is entered at EL2:
    * SCR_EL3.HCE (bit 8) must be initialised to 0b1.

  For systems with a GICv3 interrupt controller to be used in v3 mode: - If EL3 is present:

  - ICC_SRE_EL3.Enable (bit 3) must be initialised to 0b1.
  - ICC_SRE_EL3.SRE (bit 0) must be initialised to 0b1.
  - ICC_CTLR_EL3.PMHE (bit 6) must be set to the same value across all CPUs the kernel is executing on, and must stay constant for the lifetime of the kernel.
  - If the kernel is entered at EL1:
    * ICC.SRE_EL2.Enable (bit 3) must be initialised to 0b1

    * ICC_SRE_EL2.SRE (bit 0) must be initialised to 0b1.

  - The DT or ACPI tables must describe a GICv3 interrupt controller.

For systems with a GICv3 interrupt controller to be used in compatibility (v2) mode:

  - If EL3 is present:

    ICC_SRE_EL3.SRE (bit 0) must be initialised to 0b0.

  - If the kernel is entered at EL1:

    ICC_SRE_EL2.SRE (bit 0) must be initialised to 0b0.

  - The DT or ACPI tables must describe a GICv2 interrupt controller.

For CPUs with pointer authentication functionality:

  - If EL3 is present:
    * SCR_EL3.APK (bit 16) must be initialised to 0b1
    * SCR_EL3.API (bit 17) must be initialised to 0b1
  - If the kernel is entered at EL1:
    * HCR_EL2.APK (bit 40) must be initialised to 0b1
    * HCR_EL2.API (bit 41) must be initialised to 0b1

For CPUs with Activity Monitors Unit v1 (AMUv1) extension present:

  - If EL3 is present:
    * CPTR_EL3.TAM (bit 30) must be initialised to 0b0
    * CPTR_EL2.TAM (bit 30) must be initialised to 0b0
    * AMCNTENSET0_EL0 must be initialised to 0b1111
    * AMCNTENSET1_EL0 must be initialised to a platform specific value having 0b1 set for the corresponding bit for each of the auxiliary counters present.
  - If the kernel is entered at EL1:
    * AMCNTENSET0_EL0 must be initialised to 0b1111
    * AMCNTENSET1_EL0 must be initialised to a platform specific value having 0b1 set for the corresponding bit for each of the auxiliary counters present.

For CPUs with the Fine Grained Traps (FEAT_FGT) extension present:

  - If EL3 is present and the kernel is entered at EL2:
    * SCR_EL3.FGTEn (bit 27) must be initialised to 0b1.

For CPUs with support for HCRX_EL2 (FEAT_HCX) present:

  - If EL3 is present and the kernel is entered at EL2:
    * SCR_EL3.HXEn (bit 38) must be initialised to 0b1.

For CPUs with Advanced SIMD and floating point support:

  - If EL3 is present:
    * CPTR_EL3.TFP (bit 10) must be initialised to 0b0.

  - If EL2 is present and the kernel is entered at EL1:
    * CPTR_EL2.TFP (bit 10) must be initialised to 0b0.

For CPUs with the Scalable Vector Extension (FEAT_SVE) present:

  - if EL3 is present:
    * CPTR_EL3.EZ (bit 8) must be initialised to 0b1.
    * ZCR_EL3.LEN must be initialised to the same value for all CPUs the kernel is executed on.
  - If the kernel is entered at EL1 and EL2 is present:
    * CPTR_EL2.TZ (bit 8) must be initialised to 0b0.
    * CPTR_EL2.ZEN (bits 17:16) must be initialised to 0b11.
    * ZCR_EL2.LEN must be initialised to the same value for all CPUs the kernel will execute on.

For CPUs with the Scalable Matrix Extension (FEAT_SME):

  - If EL3 is present:
    * CPTR_EL3.ESM (bit 12) must be initialised to 0b1.
    * SCR_EL3.EnTP2 (bit 41) must be initialised to 0b1.
    * SMCR_EL3.LEN must be initialised to the same value for all CPUs the kernel will execute on.
  - If the kernel is entered at EL1 and EL2 is present:
    - CPTR_EL2.TSM (bit 12) must be initialised to 0b0.
    - CPTR_EL2.SMEN (bits 25:24) must be initialised to 0b11.
    - SCTLR_EL2.EnTP2 (bit 60) must be initialised to 0b1.
    - SMCR_EL2.LEN must be initialised to the same value for all CPUs the kernel will execute on.
    - HWFGRTR_EL2.nTPIDR2_EL0 (bit 55) must be initialised to 0b01.
    - HWFGWTR_EL2.nTPIDR2_EL0 (bit 55) must be initialised to 0b01.
    - HWFGRTR_EL2.nSMPRI_EL1 (bit 54) must be initialised to 0b01.
    - HWFGWTR_EL2.nSMPRI_EL1 (bit 54) must be initialised to 0b01.

    For CPUs with the Scalable Matrix Extension FA64 feature (FEAT_SME_FA64):

    - If EL3 is present:
      - SMCR_EL3.FA64 (bit 31) must be initialised to 0b1.
  - If the kernel is entered at EL1 and EL2 is present:
    - SMCR_EL2.FA64 (bit 31) must be initialised to 0b1.

    For CPUs with the Memory Tagging Extension feature (FEAT_MTE2):

    - If EL3 is present:

    - SCR_EL3.ATA (bit 26) must be initialised to 0b1.

  - If the kernel is entered at EL1 and EL2 is present:

    - HCR_EL2.ATA (bit 56) must be initialised to 0b1.

  For CPUs with the Scalable Matrix Extension version 2 (FEAT_SME2):

  - If EL3 is present:

    - SMCR_EL3.EZT0 (bit 30) must be initialised to 0b1.

- If the kernel is entered at EL1 and EL2 is present:

  - SMCR_EL2.EZT0 (bit 30) must be initialised to 0b1.

  For CPUs with Memory Copy and Memory Set instructions (FEAT_MOPS):

  - If the kernel is entered at EL1 and EL2 is present:

    - HCRX_EL2.MSCEn (bit 11) must be initialised to 0b1.

  For CPUs with the Extended Translation Control Register feature (FEAT_TCR2):

  - If EL3 is present:

    - SCR_EL3.TCR2En (bit 43) must be initialised to 0b1.

- If the kernel is entered at EL1 and EL2 is present:

  - HCRX_EL2.TCR2En (bit 14) must be initialised to 0b1.

  For CPUs with the Stage 1 Permission Indirection Extension feature (FEAT_S1PIE):

  - If EL3 is present:

    - SCR_EL3.PIEn (bit 45) must be initialised to 0b1.

  - If the kernel is entered at EL1 and EL2 is present:

    - HFGRTR_EL2.nPIR_EL1 (bit 58) must be initialised to 0b1.
    - HFGWTR_EL2.nPIR_EL1 (bit 58) must be initialised to 0b1.
    - HFGRTR_EL2.nPIRE0_EL1 (bit 57) must be initialised to 0b1.
    - HFGRWR_EL2.nPIRE0_EL1 (bit 57) must be initialised to 0b1.

The requirements described above for CPU mode, caches, MMUs, architected timers, coherency and system registers apply to all CPUs. All CPUs must enter the kernel in the same exception level. Where the values documented disable traps it is permissible for these traps to be enabled so long as those traps are handled transparently by higher exception levels as though the values documented were set.

The boot loader is expected to enter the kernel on each CPU in the following manner:

- The primary CPU must jump directly to the first instruction of the kernel image. The device tree blob passed by this CPU must contain an 'enable-method' property for each cpu node. The supported enable-methods are described below.

  It is expected that the bootloader will generate these device tree properties and insert them into the blob prior to kernel entry.

- CPUs with a "spin-table" enable-method must have a 'cpu-release-addr' property in their cpu node. This property identifies a naturally-aligned 64-bit zero-initalised memory location.

  These CPUs should spin outside of the kernel in a reserved area of memory (communicated to the kernel by a /memreserve/ region in the device tree) polling their cpu-release-addr location, which must be contained in the reserved region. A wfe instruction may be inserted to reduce the overhead of the busy-loop and a sev will be issued by the primary CPU. When a read of the location pointed to by the cpu-release-addr returns a non-zero value, the CPU must jump to this value. The value will be written as a single 64-bit little-endian value, so CPUs must convert the read value to their native endianness before jumping to it.

- CPUs with a "psci" enable method should remain outside of the kernel (i.e. outside of the regions of memory described to the kernel in the memory node, or in a reserved area of memory described to the kernel by a /memreserve/ region in the device tree). The kernel will issue CPU_ON calls as described in ARM document number ARM DEN 0022A ("Power State Coordination Interface System Software on ARM processors") to bring CPUs into the kernel.

  The device tree should contain a 'psci' node, as described in Documentation/devicetree/bindings/arm/psci.yaml.

- Secondary CPU general-purpose register settings
  - x0 = 0 (reserved for future use)
  - x1 = 0 (reserved for future use)
  - x2 = 0 (reserved for future use)
  - x3 = 0 (reserved for future use)

3.6 ARM64 CPU Feature Registers

Author: Suzuki K Poulose <suzuki.poulose@arm.com>

This file describes the ABI for exporting the AArch64 CPU ID/feature registers to userspace. The availability of this ABI is advertised via the HWCAP_CPUID in HWCAPs.

3.6.1 1. Motivation

The ARM architecture defines a set of feature registers, which describe the capabilities of the CPU/system. Access to these system registers is restricted from EL0 and there is no reliable way for an application to extract this information to make better decisions at runtime. There is limited information available to the application via HWCAPs, however there are some issues with their usage.

a) Any change to the HWCAPs requires an update to userspace (e.g libc) to detect the new changes, which can take a long time to appear in distributions. Exposing the registers allows applications to get the information without requiring updates to the toolchains.

b) Access to HWCAPs is sometimes limited (e.g prior to libc, or when ld is initialised at startup time).

c) HWCAPs cannot represent non-boolean information effectively. The architecture defines a canonical format for representing features in the ID registers; this is well defined and is capable of representing all valid architecture variations.

3.6.2 2. Requirements

a) Safety:

   Applications should be able to use the information provided by the infrastructure to run safely across the system. This has greater implications on a system with heterogeneous CPUs. The infrastructure exports a value that is safe across all the available CPU on the system.

   e.g, If at least one CPU doesn't implement CRC32 instructions, while others do, we should report that the CRC32 is not implemented. Otherwise an application could crash when scheduled on the CPU which doesn't support CRC32.

b) Security:

   Applications should only be able to receive information that is relevant to the normal operation in userspace. Hence, some of the fields are masked out(i.e, made invisible) and their values are set to indicate the feature is 'not supported'. See Section 4 for the list of visible features. Also, the kernel may manipulate the fields based on what it supports. e.g, If FP is not supported by the kernel, the values could indicate that the FP is not available (even when the CPU provides it).

c) Implementation Defined Features

   The infrastructure doesn't expose any register which is IMPLEMENTATION DEFINED as per ARMv8-A Architecture.

d) CPU Identification:

   MIDR_EL1 is exposed to help identify the processor. On a heterogeneous system, this could be racy (just like getcpu()). The process could be migrated to another CPU by the time it uses the register value, unless the CPU affinity is set. Hence, there is no guarantee that the value reflects the processor that it is currently executing on. The REVIDR is not exposed due to this constraint, as REVIDR makes sense only in conjunction with the MIDR. Alternately, MIDR_EL1 and REVIDR_EL1 are exposed via sysfs at:

```
/sys/devices/system/cpu/cpu$ID/regs/identification/
                                                \- midr
                                                \- revidr
```

3.6.3 3. Implementation

The infrastructure is built on the emulation of the 'MRS' instruction. Accessing a restricted system register from an application generates an exception and ends up in SIGILL being delivered to the process. The infrastructure hooks into the exception handler and emulates the operation if the source belongs to the supported system register space.

The infrastructure emulates only the following system register space:

```
Op0=3, Op1=0, CRn=0, CRm=0,2,3,4,5,6,7
```

(See Table C5-6 ‘System instruction encodings for non-Debug System register accesses’ in ARMv8 ARM DDI 0487A.h, for the list of registers).

The following rules are applied to the value returned by the infrastructure:

a) The value of an ‘IMPLEMENTATION DEFINED’ field is set to 0.

b) The value of a reserved field is populated with the reserved value as defined by the architecture.

c) The value of a ‘visible’ field holds the system wide safe value for the particular feature (except for MIDR_EL1, see section 4).

d) All other fields (i.e, invisible fields) are set to indicate the feature is missing (as defined by the architecture).

3.6.4 4. List of registers with visible features

1) ID_AA64ISAR0_EL1 - Instruction Set Attribute Register 0

Name	bits	visible
RNDR	[63-60]	y
TS	[55-52]	y
FHM	[51-48]	y
DP	[47-44]	y
SM4	[43-40]	y
SM3	[39-36]	y
SHA3	[35-32]	y
RDM	[31-28]	y
ATOMICS	[23-20]	y
CRC32	[19-16]	y
SHA2	[15-12]	y
SHA1	[11-8]	y
AES	[7-4]	y

2) ID_AA64PFR0_EL1 - Processor Feature Register 0

Name	bits	visible
DIT	[51-48]	y
SVE	[35-32]	y
GIC	[27-24]	n
AdvSIMD	[23-20]	y
FP	[19-16]	y
EL3	[15-12]	n
EL2	[11-8]	n
EL1	[7-4]	n
EL0	[3-0]	n

3) ID_AA64PFR1_EL1 - Processor Feature Register 1

Name	bits	visible
SME	[27-24]	y
MTE	[11-8]	y
SSBS	[7-4]	y
BT	[3-0]	y

4) MIDR_EL1 - Main ID Register

Name	bits	visible
Implementer	[31-24]	y
Variant	[23-20]	y
Architecture	[19-16]	y
PartNum	[15-4]	y
Revision	[3-0]	y

NOTE: The 'visible' fields of MIDR_EL1 will contain the value as available on the CPU where it is fetched and is not a system wide safe value.

5) ID_AA64ISAR1_EL1 - Instruction set attribute register 1

Name	bits	visible
I8MM	[55-52]	y
DGH	[51-48]	y
BF16	[47-44]	y
SB	[39-36]	y
FRINTTS	[35-32]	y
GPI	[31-28]	y
GPA	[27-24]	y
LRCPC	[23-20]	y
FCMA	[19-16]	y
JSCVT	[15-12]	y
API	[11-8]	y
APA	[7-4]	y
DPB	[3-0]	y

6) ID_AA64MMFR0_EL1 - Memory model feature register 0

Name	bits	visible
ECV	[63-60]	y

7) ID_AA64MMFR2_EL1 - Memory model feature register 2

Name	bits	visible
AT	[35-32]	y

8) ID_AA64ZFR0_EL1 - SVE feature ID register 0

Name	bits	visible
F64MM	[59-56]	y
F32MM	[55-52]	y
I8MM	[47-44]	y
SM4	[43-40]	y
SHA3	[35-32]	y
BF16	[23-20]	y
BitPerm	[19-16]	y
AES	[7-4]	y
SVEVer	[3-0]	y

8) ID_AA64MMFR1_EL1 - Memory model feature register 1

Name	bits	visible
AFP	[47-44]	y

9) ID_AA64ISAR2_EL1 - Instruction set attribute register 2

Name	bits	visible
CSSC	[55-52]	y
RPRFM	[51-48]	y
BC	[23-20]	y
MOPS	[19-16]	y
APA3	[15-12]	y
GPA3	[11-8]	y
RPRES	[7-4]	y
WFXT	[3-0]	y

10) MVFR0_EL1 - AArch32 Media and VFP Feature Register 0

Name	bits	visible
FPDP	[11-8]	y

11) MVFR1_EL1 - AArch32 Media and VFP Feature Register 1

Name	bits	visible
SIMDFMAC	[31-28]	y
SIMDSP	[19-16]	y
SIMDInt	[15-12]	y
SIMDLS	[11-8]	y

12) ID_ISAR5_EL1 - AArch32 Instruction Set Attribute Register 5

Name	bits	visible
CRC32	[19-16]	y
SHA2	[15-12]	y
SHA1	[11-8]	y
AES	[7-4]	y

3.6.5 Appendix I: Example

```
/*
 * Sample program to demonstrate the MRS emulation ABI.
 *
 * Copyright (C) 2015-2016, ARM Ltd
 *
 * Author: Suzuki K Poulose <suzuki.poulose@arm.com>
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License version 2 as
 * published by the Free Software Foundation.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 */

#include <asm/hwcap.h>
#include <stdio.h>
#include <sys/auxv.h>

#define get_cpu_ftr(id) ({                                      \
              unsigned long __val;                              \
              asm("mrs %0, "#id : "=r" (__val));                \
              printf("%-20s: 0x%016lx\n", #id, __val);          \
      })

int main(void)
{

      if (!(getauxval(AT_HWCAP) & HWCAP_CPUID)) {
              fputs("CPUID registers unavailable\n", stderr);
              return 1;
```

```
        }

        get_cpu_ftr(ID_AA64ISAR0_EL1);
        get_cpu_ftr(ID_AA64ISAR1_EL1);
        get_cpu_ftr(ID_AA64MMFR0_EL1);
        get_cpu_ftr(ID_AA64MMFR1_EL1);
        get_cpu_ftr(ID_AA64PFR0_EL1);
        get_cpu_ftr(ID_AA64PFR1_EL1);
        get_cpu_ftr(ID_AA64DFR0_EL1);
        get_cpu_ftr(ID_AA64DFR1_EL1);

        get_cpu_ftr(MIDR_EL1);
        get_cpu_ftr(MPIDR_EL1);
        get_cpu_ftr(REVIDR_EL1);

#if 0
        /* Unexposed register access causes SIGILL */
        get_cpu_ftr(ID_MMFR0_EL1);
#endif

        return 0;
}
```

3.7 ARM64 ELF hwcaps

This document describes the usage and semantics of the arm64 ELF hwcaps.

3.7.1 1. Introduction

Some hardware or software features are only available on some CPU implementations, and/or with certain kernel configurations, but have no architected discovery mechanism available to userspace code at EL0. The kernel exposes the presence of these features to userspace through a set of flags called hwcaps, exposed in the auxiliary vector.

Userspace software can test for features by acquiring the AT_HWCAP or AT_HWCAP2 entry of the auxiliary vector, and testing whether the relevant flags are set, e.g.:

```
bool floating_point_is_present(void)
{
        unsigned long hwcaps = getauxval(AT_HWCAP);
        if (hwcaps & HWCAP_FP)
                return true;

        return false;
}
```

Where software relies on a feature described by a hwcap, it should check the relevant hwcap flag to verify that the feature is present before attempting to make use of the feature.

Features cannot be probed reliably through other means. When a feature is not available, attempting to use it may result in unpredictable behaviour, and is not guaranteed to result in any reliable indication that the feature is unavailable, such as a SIGILL.

3.7.2 2. Interpretation of hwcaps

The majority of hwcaps are intended to indicate the presence of features which are described by architected ID registers inaccessible to userspace code at EL0. These hwcaps are defined in terms of ID register fields, and should be interpreted with reference to the definition of these fields in the ARM Architecture Reference Manual (ARM ARM).

Such hwcaps are described below in the form:

```
Functionality implied by idreg.field == val.
```

Such hwcaps indicate the availability of functionality that the ARM ARM defines as being present when idreg.field has value val, but do not indicate that idreg.field is precisely equal to val, nor do they indicate the absence of functionality implied by other values of idreg.field.

Other hwcaps may indicate the presence of features which cannot be described by ID registers alone. These may be described without reference to ID registers, and may refer to other documentation.

3.7.3 3. The hwcaps exposed in AT_HWCAP

HWCAP_FP
Functionality implied by ID_AA64PFR0_EL1.FP == 0b0000.

HWCAP_ASIMD
Functionality implied by ID_AA64PFR0_EL1.AdvSIMD == 0b0000.

HWCAP_EVTSTRM
The generic timer is configured to generate events at a frequency of approximately 10KHz.

HWCAP_AES
Functionality implied by ID_AA64ISAR0_EL1.AES == 0b0001.

HWCAP_PMULL
Functionality implied by ID_AA64ISAR0_EL1.AES == 0b0010.

HWCAP_SHA1
Functionality implied by ID_AA64ISAR0_EL1.SHA1 == 0b0001.

HWCAP_SHA2
Functionality implied by ID_AA64ISAR0_EL1.SHA2 == 0b0001.

HWCAP_CRC32
Functionality implied by ID_AA64ISAR0_EL1.CRC32 == 0b0001.

HWCAP_ATOMICS
Functionality implied by ID_AA64ISAR0_EL1.Atomic == 0b0010.

HWCAP_FPHP
Functionality implied by ID_AA64PFR0_EL1.FP == 0b0001.

HWCAP_ASIMDHP
Functionality implied by ID_AA64PFR0_EL1.AdvSIMD == 0b0001.

HWCAP_CPUID
EL0 access to certain ID registers is available, to the extent described by *ARM64 CPU Feature Registers*.

These ID registers may imply the availability of features.

HWCAP_ASIMDRDM
Functionality implied by ID_AA64ISAR0_EL1.RDM == 0b0001.

HWCAP_JSCVT
Functionality implied by ID_AA64ISAR1_EL1.JSCVT == 0b0001.

HWCAP_FCMA
Functionality implied by ID_AA64ISAR1_EL1.FCMA == 0b0001.

HWCAP_LRCPC
Functionality implied by ID_AA64ISAR1_EL1.LRCPC == 0b0001.

HWCAP_DCPOP
Functionality implied by ID_AA64ISAR1_EL1.DPB == 0b0001.

HWCAP_SHA3
Functionality implied by ID_AA64ISAR0_EL1.SHA3 == 0b0001.

HWCAP_SM3
Functionality implied by ID_AA64ISAR0_EL1.SM3 == 0b0001.

HWCAP_SM4
Functionality implied by ID_AA64ISAR0_EL1.SM4 == 0b0001.

HWCAP_ASIMDDP
Functionality implied by ID_AA64ISAR0_EL1.DP == 0b0001.

HWCAP_SHA512
Functionality implied by ID_AA64ISAR0_EL1.SHA2 == 0b0010.

HWCAP_SVE
Functionality implied by ID_AA64PFR0_EL1.SVE == 0b0001.

HWCAP_ASIMDFHM
Functionality implied by ID_AA64ISAR0_EL1.FHM == 0b0001.

HWCAP_DIT
Functionality implied by ID_AA64PFR0_EL1.DIT == 0b0001.

HWCAP_USCAT
Functionality implied by ID_AA64MMFR2_EL1.AT == 0b0001.

HWCAP_ILRCPC
Functionality implied by ID_AA64ISAR1_EL1.LRCPC == 0b0010.

HWCAP_FLAGM
Functionality implied by ID_AA64ISAR0_EL1.TS == 0b0001.

HWCAP_SSBS
Functionality implied by ID_AA64PFR1_EL1.SSBS == 0b0010.

HWCAP_SB
: Functionality implied by ID_AA64ISAR1_EL1.SB == 0b0001.

HWCAP_PACA
: Functionality implied by ID_AA64ISAR1_EL1.APA == 0b0001 or ID_AA64ISAR1_EL1.API == 0b0001, as described by *Pointer authentication in AArch64 Linux*.

HWCAP_PACG
: Functionality implied by ID_AA64ISAR1_EL1.GPA == 0b0001 or ID_AA64ISAR1_EL1.GPI == 0b0001, as described by *Pointer authentication in AArch64 Linux*.

HWCAP2_DCPODP
: Functionality implied by ID_AA64ISAR1_EL1.DPB == 0b0010.

HWCAP2_SVE2
: Functionality implied by ID_AA64ZFR0_EL1.SVEVer == 0b0001.

HWCAP2_SVEAES
: Functionality implied by ID_AA64ZFR0_EL1.AES == 0b0001.

HWCAP2_SVEPMULL
: Functionality implied by ID_AA64ZFR0_EL1.AES == 0b0010.

HWCAP2_SVEBITPERM
: Functionality implied by ID_AA64ZFR0_EL1.BitPerm == 0b0001.

HWCAP2_SVESHA3
: Functionality implied by ID_AA64ZFR0_EL1.SHA3 == 0b0001.

HWCAP2_SVESM4
: Functionality implied by ID_AA64ZFR0_EL1.SM4 == 0b0001.

HWCAP2_FLAGM2
: Functionality implied by ID_AA64ISAR0_EL1.TS == 0b0010.

HWCAP2_FRINT
: Functionality implied by ID_AA64ISAR1_EL1.FRINTTS == 0b0001.

HWCAP2_SVEI8MM
: Functionality implied by ID_AA64ZFR0_EL1.I8MM == 0b0001.

HWCAP2_SVEF32MM
: Functionality implied by ID_AA64ZFR0_EL1.F32MM == 0b0001.

HWCAP2_SVEF64MM
: Functionality implied by ID_AA64ZFR0_EL1.F64MM == 0b0001.

HWCAP2_SVEBF16
: Functionality implied by ID_AA64ZFR0_EL1.BF16 == 0b0001.

HWCAP2_I8MM
: Functionality implied by ID_AA64ISAR1_EL1.I8MM == 0b0001.

HWCAP2_BF16
: Functionality implied by ID_AA64ISAR1_EL1.BF16 == 0b0001.

HWCAP2_DGH
: Functionality implied by ID_AA64ISAR1_EL1.DGH == 0b0001.

HWCAP2_RNG
: Functionality implied by ID_AA64ISAR0_EL1.RNDR == 0b0001.

HWCAP2_BTI
Functionality implied by ID_AA64PFR0_EL1.BT == 0b0001.

HWCAP2_MTE
Functionality implied by ID_AA64PFR1_EL1.MTE == 0b0010, as described by *Memory Tagging Extension (MTE) in AArch64 Linux*.

HWCAP2_ECV
Functionality implied by ID_AA64MMFR0_EL1.ECV == 0b0001.

HWCAP2_AFP
Functionality implied by ID_AA64MFR1_EL1.AFP == 0b0001.

HWCAP2_RPRES
Functionality implied by ID_AA64ISAR2_EL1.RPRES == 0b0001.

HWCAP2_MTE3
Functionality implied by ID_AA64PFR1_EL1.MTE == 0b0011, as described by *Memory Tagging Extension (MTE) in AArch64 Linux*.

HWCAP2_SME
Functionality implied by ID_AA64PFR1_EL1.SME == 0b0001, as described by *Scalable Matrix Extension support for AArch64 Linux*.

HWCAP2_SME_I16I64
Functionality implied by ID_AA64SMFR0_EL1.I16I64 == 0b1111.

HWCAP2_SME_F64F64
Functionality implied by ID_AA64SMFR0_EL1.F64F64 == 0b1.

HWCAP2_SME_I8I32
Functionality implied by ID_AA64SMFR0_EL1.I8I32 == 0b1111.

HWCAP2_SME_F16F32
Functionality implied by ID_AA64SMFR0_EL1.F16F32 == 0b1.

HWCAP2_SME_B16F32
Functionality implied by ID_AA64SMFR0_EL1.B16F32 == 0b1.

HWCAP2_SME_F32F32
Functionality implied by ID_AA64SMFR0_EL1.F32F32 == 0b1.

HWCAP2_SME_FA64
Functionality implied by ID_AA64SMFR0_EL1.FA64 == 0b1.

HWCAP2_WFXT
Functionality implied by ID_AA64ISAR2_EL1.WFXT == 0b0010.

HWCAP2_EBF16
Functionality implied by ID_AA64ISAR1_EL1.BF16 == 0b0010.

HWCAP2_SVE_EBF16
Functionality implied by ID_AA64ZFR0_EL1.BF16 == 0b0010.

HWCAP2_CSSC
Functionality implied by ID_AA64ISAR2_EL1.CSSC == 0b0001.

HWCAP2_RPRFM
Functionality implied by ID_AA64ISAR2_EL1.RPRFM == 0b0001.

HWCAP2_SVE2P1
Functionality implied by ID_AA64ZFR0_EL1.SVEver == 0b0010.

HWCAP2_SME2
Functionality implied by ID_AA64SMFR0_EL1.SMEver == 0b0001.

HWCAP2_SME2P1
Functionality implied by ID_AA64SMFR0_EL1.SMEver == 0b0010.

HWCAP2_SMEI16I32
Functionality implied by ID_AA64SMFR0_EL1.I16I32 == 0b0101

HWCAP2_SMEBI32I32
Functionality implied by ID_AA64SMFR0_EL1.BI32I32 == 0b1

HWCAP2_SMEB16B16
Functionality implied by ID_AA64SMFR0_EL1.B16B16 == 0b1

HWCAP2_SMEF16F16
Functionality implied by ID_AA64SMFR0_EL1.F16F16 == 0b1

HWCAP2_MOPS
Functionality implied by ID_AA64ISAR2_EL1.MOPS == 0b0001.

HWCAP2_HBC
Functionality implied by ID_AA64ISAR2_EL1.BC == 0b0001.

3.7.4 4. Unused AT_HWCAP bits

For interoperation with userspace, the kernel guarantees that bits 62 and 63 of AT_HWCAP will always be returned as 0.

3.8 HugeTLBpage on ARM64

Hugepage relies on making efficient use of TLBs to improve performance of address translations. The benefit depends on both -

- the size of hugepages
- size of entries supported by the TLBs

The ARM64 port supports two flavours of hugepages.

3.8.1 1) Block mappings at the pud/pmd level

These are regular hugepages where a pmd or a pud page table entry points to a block of memory. Regardless of the supported size of entries in TLB, block mappings reduce the depth of page table walk needed to translate hugepage addresses.

3.8.2 2) Using the Contiguous bit

The architecture provides a contiguous bit in the translation table entries (D4.5.3, ARM DDI 0487C.a) that hints to the MMU to indicate that it is one of a contiguous set of entries that can be cached in a single TLB entry.

The contiguous bit is used in Linux to increase the mapping size at the pmd and pte (last) level. The number of supported contiguous entries varies by page size and level of the page table.

The following hugepage sizes are supported -

•	CONT PTE	PMD	CONT PMD	PUD
4K:	64K	2M	32M	1G
16K:	2M	32M	1G	
64K:	2M	512M	16G	

3.9 crashkernel memory reservation on arm64

Author: Baoquan He <bhe@redhat.com>

Kdump mechanism is used to capture a corrupted kernel vmcore so that it can be subsequently analyzed. In order to do this, a preliminarily reserved memory is needed to pre-load the kdump kernel and boot such kernel if corruption happens.

That reserved memory for kdump is adapted to be able to minimally accommodate the kdump kernel and the user space programs needed for the vmcore collection.

3.9.1 Kernel parameter

Through the kernel parameters below, memory can be reserved accordingly during the early stage of the first kernel booting so that a continuous large chunk of memomy can be found. The low memory reservation needs to be considered if the crashkernel is reserved from the high memory area.

- crashkernel=size@offset
- crashkernel=size
- crashkernel=size,high crashkernel=size,low

3.9.2 Low memory and high memory

For kdump reservations, low memory is the memory area under a specific limit, usually decided by the accessible address bits of the DMA-capable devices needed by the kdump kernel to run. Those devices not related to vmcore dumping can be ignored. On arm64, the low memory upper bound is not fixed: it is 1G on the RPi4 platform but 4G on most other systems. On special kernels built with CONFIG_ZONE_(DMA|DMA32) disabled, the whole system RAM is low memory. Outside of the low memory described above, the rest of system RAM is considered high memory.

3.9.3 Implementation

1) crashkernel=size@offset

The crashkernel memory must be reserved at the user-specified region or fail if already occupied.

2) crashkernel=size

The crashkernel memory region will be reserved in any available position according to the search order:

Firstly, the kernel searches the low memory area for an available region with the specified size.

If searching for low memory fails, the kernel falls back to searching the high memory area for an available region of the specified size. If the reservation in high memory succeeds, a default size reservation in the low memory will be done. Currently the default size is 128M, sufficient for the low memory needs of the kdump kernel.

Note: crashkernel=size is the recommended option for crashkernel kernel reservations. The user would not need to know the system memory layout for a specific platform.

3) crashkernel=size,high crashkernel=size,low

crashkernel=size,(high|low) are an important supplement to crashkernel=size. They allows the user to specify how much memory needs to be allocated from the high memory and low memory respectively. On many systems the low memory is precious and crashkernel reservations from this area should be kept to a minimum.

To reserve memory for crashkernel=size,high, searching is first attempted from the high memory region. If the reservation succeeds, the low memory reservation will be done subsequently.

If reservation from the high memory failed, the kernel falls back to searching the low memory with the specified size in crashkernel=,high. If it succeeds, no further reservation for low memory is needed.

Notes:

- If crashkernel=,low is not specified, the default low memory reservation will be done automatically.
- if crashkernel=0,low is specified, it means that the low memory reservation is omitted intentionally.

3.10 Legacy instructions

The arm64 port of the Linux kernel provides infrastructure to support emulation of instructions which have been deprecated, or obsoleted in the architecture. The infrastructure code uses undefined instruction hooks to support emulation. Where available it also allows turning on the instruction execution in hardware.

The emulation mode can be controlled by writing to sysctl nodes (/proc/sys/abi). The following explains the different execution behaviours and the corresponding values of the sysctl nodes -

- **Undef**
  Value: 0

  Generates undefined instruction abort. Default for instructions that have been obsoleted in the architecture, e.g., SWP
- **Emulate**
  Value: 1

  Uses software emulation. To aid migration of software, in this mode usage of emulated instruction is traced as well as rate limited warnings are issued. This is the default for deprecated instructions, .e.g., CP15 barriers
- **Hardware Execution**
  Value: 2

  Although marked as deprecated, some implementations may support the enabling/disabling of hardware support for the execution of these instructions. Using hardware execution generally provides better performance, but at the loss of ability to gather runtime statistics about the use of the deprecated instructions.

The default mode depends on the status of the instruction in the architecture. Deprecated instructions should default to emulation while obsolete instructions must be undefined by default.

Note: Instruction emulation may not be possible in all cases. See individual instruction notes for further information.

3.10.1 Supported legacy instructions

- SWP{B}

  Node
  /proc/sys/abi/swp

  Status
  Obsolete

  Default
  Undef (0)
- CP15 Barriers

  Node
  /proc/sys/abi/cp15_barrier

  Status
  Deprecated

Default
Emulate (1)

- SETEND

  Node
  /proc/sys/abi/setend

  Status
  Deprecated

  Default
  Emulate (1)*

  Note: All the cpus on the system must have mixed endian support at EL0 for this feature to be enabled. If a new CPU - which doesn't support mixed endian - is hotplugged in after this feature has been enabled, there could be unexpected results in the application.

3.11 Memory Layout on AArch64 Linux

Author: Catalin Marinas <catalin.marinas@arm.com>

This document describes the virtual memory layout used by the AArch64 Linux kernel. The architecture allows up to 4 levels of translation tables with a 4KB page size and up to 3 levels with a 64KB page size.

AArch64 Linux uses either 3 levels or 4 levels of translation tables with the 4KB page configuration, allowing 39-bit (512GB) or 48-bit (256TB) virtual addresses, respectively, for both user and kernel. With 64KB pages, only 2 levels of translation tables, allowing 42-bit (4TB) virtual address, are used but the memory layout is the same.

ARMv8.2 adds optional support for Large Virtual Address space. This is only available when running with a 64KB page size and expands the number of descriptors in the first level of translation.

User addresses have bits 63:48 set to 0 while the kernel addresses have the same bits set to 1. TTBRx selection is given by bit 63 of the virtual address. The swapper_pg_dir contains only kernel (global) mappings while the user pgd contains only user (non-global) mappings. The swapper_pg_dir address is written to TTBR1 and never written to TTBR0.

AArch64 Linux memory layout with 4KB pages + 4 levels (48-bit):

```
 Start                  End                     Size            Use
 -----------------------------------------------------------------------
 0000000000000000       0000ffffffffffff         256TB          user
 ffff000000000000       ffff7fffffffffff         128TB          kernel logical␣
↪memory map
[ffff600000000000       ffff7fffffffffff]         32TB          [kasan shadow␣
↪region]
 ffff800000000000       ffff80007fffffff           2GB          modules
 ffff800080000000       fffffbffefffffff         124TB          vmalloc
 fffffbfff0000000       fffffbfffdffffff         224MB          fixed mappings␣
↪(top down)
 fffffbfffe000000       fffffbfffe7fffff           8MB          [guard region]
 fffffbfffe800000      fffffbffff7fffff          16MB          PCI I/O space
 fffffbffff800000      fffffbffffffffff           8MB          [guard region]
 fffffc0000000000      fffffdffffffffff           2TB          vmemmap
 fffffe0000000000      ffffffffffffffff           2TB          [guard region]
```

AArch64 Linux memory layout with 64KB pages + 3 levels (52-bit with HW support):

```
 Start                 End                     Size            Use
 -----------------------------------------------------------------------
 0000000000000000      000fffffffffffff           4PB          user
 fff0000000000000      ffff7fffffffffff          ~4PB          kernel logical␣
 ↪memory map
[fffd800000000000      ffff7fffffffffff]        512TB          [kasan shadow␣
 ↪region]
 ffff800000000000      ffff80007fffffff           2GB          modules
 ffff800080000000      fffffbffefffffff         124TB          vmalloc
 fffffbfff0000000      fffffbfffdffffff         224MB          fixed mappings␣
 ↪(top down)
 fffffbfffe000000      fffffbfffe7fffff           8MB          [guard region]
 fffffbfffe800000      fffffbffff7fffff          16MB          PCI I/O space
 fffffbffff800000      fffffbffffffffff           8MB          [guard region]
 fffffc0000000000      fffffdffffffffff          ~4TB          vmemmap
 fffffe0000000000      ffffffffffffffff         128GB          [guard region]
```

Translation table lookup with 4KB pages:

```
+--------+--------+--------+--------+--------+--------+--------+--------+
|63    56|55    48|47    40|39    32|31    24|23    16|15     8|7      0|
+--------+--------+--------+--------+--------+--------+--------+--------+
 |                 |         |         |         |         |
 |                 |         |         |         |         v
 |                 |         |         |         |   [11:0]  in-page offset
 |                 |         |         |         +-> [20:12] L3 index
 |                 |         |         +-----------> [29:21] L2 index
 |                 |         +---------------------> [38:30] L1 index
 |                 +-------------------------------> [47:39] L0 index
 +-------------------------------------------------> [63] TTBR0/1
```

Translation table lookup with 64KB pages:

```
+--------+--------+--------+--------+--------+--------+--------+--------+
|63    56|55    48|47    40|39    32|31    24|23    16|15     8|7      0|
+--------+--------+--------+--------+--------+--------+--------+--------+
 |                 |    |               |              |
 |                 |    |               |              v
 |                 |    |               |            [15:0]  in-page offset
 |                 |    |               +----------> [28:16] L3 index
 |                 |    +--------------------------> [41:29] L2 index
 |                 +-------------------------------> [47:42] L1 index (48-bit)
 |                                                   [51:42] L1 index (52-bit)
 +-------------------------------------------------> [63] TTBR0/1
```

When using KVM without the Virtualization Host Extensions, the hypervisor maps kernel pages in EL2 at a fixed (and potentially random) offset from the linear mapping. See the kern_hyp_va macro and kvm_update_va_mask function for more details. MMIO devices such as GICv2 gets mapped next to the HYP idmap page, as do vectors when ARM64_SPECTRE_V3A is enabled for particular CPUs.

When using KVM with the Virtualization Host Extensions, no additional mappings are created, since the host kernel runs directly in EL2.

3.11.1 52-bit VA support in the kernel

If the ARMv8.2-LVA optional feature is present, and we are running with a 64KB page size; then it is possible to use 52-bits of address space for both userspace and kernel addresses. However, any kernel binary that supports 52-bit must also be able to fall back to 48-bit at early boot time if the hardware feature is not present.

This fallback mechanism necessitates the kernel .text to be in the higher addresses such that they are invariant to 48/52-bit VAs. Due to the kasan shadow being a fraction of the entire kernel VA space, the end of the kasan shadow must also be in the higher half of the kernel VA space for both 48/52-bit. (Switching from 48-bit to 52-bit, the end of the kasan shadow is invariant and dependent on ~0UL, whilst the start address will "grow" towards the lower addresses).

In order to optimise phys_to_virt and virt_to_phys, the PAGE_OFFSET is kept constant at 0xFFF0000000000000 (corresponding to 52-bit), this obviates the need for an extra variable read. The physvirt offset and vmemmap offsets are computed at early boot to enable this logic.

As a single binary will need to support both 48-bit and 52-bit VA spaces, the VMEMMAP must be sized large enough for 52-bit VAs and also must be sized large enough to accommodate a fixed PAGE_OFFSET.

Most code in the kernel should not need to consider the VA_BITS, for code that does need to know the VA size the variables are defined as follows:

VA_BITS constant the *maximum* VA space size

VA_BITS_MIN constant the *minimum* VA space size

vabits_actual variable the *actual* VA space size

Maximum and minimum sizes can be useful to ensure that buffers are sized large enough or that addresses are positioned close enough for the "worst" case.

3.11.2 52-bit userspace VAs

To maintain compatibility with software that relies on the ARMv8.0 VA space maximum size of 48-bits, the kernel will, by default, return virtual addresses to userspace from a 48-bit range.

Software can "opt-in" to receiving VAs from a 52-bit space by specifying an mmap hint parameter that is larger than 48-bit.

For example:

```
maybe_high_address = mmap(~0UL, size, prot, flags,...);
```

It is also possible to build a debug kernel that returns addresses from a 52-bit space by enabling the following kernel config options:

```
CONFIG_EXPERT=y && CONFIG_ARM64_FORCE_52BIT=y
```

Note that this option is only intended for debugging applications and should not be used in production.

3.12 Memory Tagging Extension (MTE) in AArch64 Linux

Authors: Vincenzo Frascino <vincenzo.frascino@arm.com>
Catalin Marinas <catalin.marinas@arm.com>

Date: 2020-02-25

This document describes the provision of the Memory Tagging Extension functionality in AArch64 Linux.

3.12.1 Introduction

ARMv8.5 based processors introduce the Memory Tagging Extension (MTE) feature. MTE is built on top of the ARMv8.0 virtual address tagging TBI (Top Byte Ignore) feature and allows software to access a 4-bit allocation tag for each 16-byte granule in the physical address space. Such memory range must be mapped with the Normal-Tagged memory attribute. A logical tag is derived from bits 59-56 of the virtual address used for the memory access. A CPU with MTE enabled will compare the logical tag against the allocation tag and potentially raise an exception on mismatch, subject to system registers configuration.

3.12.2 Userspace Support

When `CONFIG_ARM64_MTE` is selected and Memory Tagging Extension is supported by the hardware, the kernel advertises the feature to userspace via `HWCAP2_MTE`.

PROT_MTE

To access the allocation tags, a user process must enable the Tagged memory attribute on an address range using a new `prot` flag for `mmap()` and `mprotect()`:

`PROT_MTE` - Pages allow access to the MTE allocation tags.

The allocation tag is set to 0 when such pages are first mapped in the user address space and preserved on copy-on-write. `MAP_SHARED` is supported and the allocation tags can be shared between processes.

Note: `PROT_MTE` is only supported on `MAP_ANONYMOUS` and RAM-based file mappings (`tmpfs`, `memfd`). Passing it to other types of mapping will result in `-EINVAL` returned by these system calls.

Note: The `PROT_MTE` flag (and corresponding memory type) cannot be cleared by `mprotect()`.

Note: `madvise()` memory ranges with `MADV_DONTNEED` and `MADV_FREE` may have the allocation tags cleared (set to 0) at any point after the system call.

Tag Check Faults

When `PROT_MTE` is enabled on an address range and a mismatch between the logical and allocation tags occurs on access, there are three configurable behaviours:

- *Ignore* - This is the default mode. The CPU (and kernel) ignores the tag check fault.
- *Synchronous* - The kernel raises a `SIGSEGV` synchronously, with `.si_code = SEGV_MTESERR` and `.si_addr = <fault-address>`. The memory access is not performed. If `SIGSEGV` is ignored or blocked by the offending thread, the containing process is terminated with a `coredump`.
- *Asynchronous* - The kernel raises a `SIGSEGV`, in the offending thread, asynchronously following one or multiple tag check faults, with `.si_code = SEGV_MTEAERR` and `.si_addr = 0` (the faulting address is unknown).
- *Asymmetric* - Reads are handled as for synchronous mode while writes are handled as for asynchronous mode.

The user can select the above modes, per thread, using the `prctl(PR_SET_TAGGED_ADDR_CTRL, flags, 0, 0, 0)` system call where `flags` contains any number of the following values in the `PR_MTE_TCF_MASK` bit-field:

- **`PR_MTE_TCF_NONE` - *Ignore* tag check faults**
  (ignored if combined with other options)
- `PR_MTE_TCF_SYNC` - *Synchronous* tag check fault mode
- `PR_MTE_TCF_ASYNC` - *Asynchronous* tag check fault mode

If no modes are specified, tag check faults are ignored. If a single mode is specified, the program will run in that mode. If multiple modes are specified, the mode is selected as described in the "Per-CPU preferred tag checking modes" section below.

The current tag check fault configuration can be read using the `prctl(PR_GET_TAGGED_ADDR_CTRL, 0, 0, 0, 0)` system call. If multiple modes were requested then all will be reported.

Tag checking can also be disabled for a user thread by setting the `PSTATE.TCO` bit with `MSR TCO, #1`.

Note: Signal handlers are always invoked with `PSTATE.TCO = 0`, irrespective of the interrupted context. `PSTATE.TCO` is restored on `sigreturn()`.

Note: There are no *match-all* logical tags available for user applications.

Note: Kernel accesses to the user address space (e.g. `read()` system call) are not checked if the user thread tag checking mode is `PR_MTE_TCF_NONE` or `PR_MTE_TCF_ASYNC`. If the tag checking mode is `PR_MTE_TCF_SYNC`, the kernel makes a best effort to check its user address accesses, however it cannot always guarantee it. Kernel accesses to user addresses are always performed with an effective `PSTATE.TCO` value of zero, regardless of the user configuration.

Excluding Tags in the IRG, ADDG and SUBG instructions

The architecture allows excluding certain tags to be randomly generated via the `GCR_EL1.Exclude` register bit-field. By default, Linux excludes all tags other than 0. A user thread can enable specific tags in the randomly generated set using the `prctl(PR_SET_TAGGED_ADDR_CTRL, flags, 0, 0, 0)` system call where `flags` contains the tags bitmap in the `PR_MTE_TAG_MASK` bit-field.

Note: The hardware uses an exclude mask but the `prctl()` interface provides an include mask. An include mask of `0` (exclusion mask `0xffff`) results in the CPU always generating tag `0`.

Per-CPU preferred tag checking mode

On some CPUs the performance of MTE in stricter tag checking modes is similar to that of less strict tag checking modes. This makes it worthwhile to enable stricter checks on those CPUs when a less strict checking mode is requested, in order to gain the error detection benefits of the stricter checks without the performance downsides. To support this scenario, a privileged user may configure a stricter tag checking mode as the CPU's preferred tag checking mode.

The preferred tag checking mode for each CPU is controlled by `/sys/devices/system/cpu/cpu<N>/mte_tcf_preferred`, to which a privileged user may write the value `async`, `sync` or `asymm`. The default preferred mode for each CPU is `async`.

To allow a program to potentially run in the CPU's preferred tag checking mode, the user program may set multiple tag check fault mode bits in the `flags` argument to the `prctl(PR_SET_TAGGED_ADDR_CTRL, flags, 0, 0, 0)` system call. If both synchronous and asynchronous modes are requested then asymmetric mode may also be selected by the kernel. If the CPU's preferred tag checking mode is in the task's set of provided tag checking modes, that mode will be selected. Otherwise, one of the modes in the task's mode will be selected by the kernel from the task's mode set using the preference order:

1. Asynchronous
2. Asymmetric
3. Synchronous

Note that there is no way for userspace to request multiple modes and also disable asymmetric mode.

Initial process state

On `execve()`, the new process has the following configuration:

- `PR_TAGGED_ADDR_ENABLE` set to 0 (disabled)
- No tag checking modes are selected (tag check faults ignored)
- `PR_MTE_TAG_MASK` set to 0 (all tags excluded)
- `PSTATE.TCO` set to 0
- `PROT_MTE` not set on any of the initial memory maps

On `fork()`, the new process inherits the parent's configuration and memory map attributes with the exception of the `madvise()` ranges with `MADV_WIPEONFORK` which will have the data and tags cleared (set to 0).

The `ptrace()` interface

`PTRACE_PEEKMTETAGS` and `PTRACE_POKEMTETAGS` allow a tracer to read the tags from or set the tags to a tracee's address space. The `ptrace()` system call is invoked as `ptrace(request, pid, addr, data)` where:

- `request` - one of `PTRACE_PEEKMTETAGS` or `PTRACE_POKEMTETAGS`.
- `pid` - the tracee's PID.
- `addr` - address in the tracee's address space.
- `data` - pointer to a `struct iovec` where `iov_base` points to a buffer of `iov_len` length in the tracer's address space.

The tags in the tracer's `iov_base` buffer are represented as one 4-bit tag per byte and correspond to a 16-byte MTE tag granule in the tracee's address space.

Note: If `addr` is not aligned to a 16-byte granule, the kernel will use the corresponding aligned address.

`ptrace()` return value:

- 0 - tags were copied, the tracer's `iov_len` was updated to the number of tags transferred. This may be smaller than the requested `iov_len` if the requested address range in the tracee's or the tracer's space cannot be accessed or does not have valid tags.
- `-EPERM` - the specified process cannot be traced.
- `-EIO` - the tracee's address range cannot be accessed (e.g. invalid address) and no tags copied. `iov_len` not updated.
- `-EFAULT` - fault on accessing the tracer's memory (`struct iovec` or `iov_base` buffer) and no tags copied. `iov_len` not updated.
- `-EOPNOTSUPP` - the tracee's address does not have valid tags (never mapped with the `PROT_MTE` flag). `iov_len` not updated.

Note: There are no transient errors for the requests above, so user programs should not retry in case of a non-zero system call return.

`PTRACE_GETREGSET` and `PTRACE_SETREGSET` with `addr == ``NT_ARM_TAGGED_ADDR_CTRL` allow `ptrace()` access to the tagged address ABI control and MTE configuration of a process as per the `prctl()` options described in *AArch64 TAGGED ADDRESS ABI* and above. The corresponding `regset` is 1 element of 8 bytes (`sizeof(long)`).

Core dump support

The allocation tags for user memory mapped with `PROT_MTE` are dumped in the core file as additional `PT_AARCH64_MEMTAG_MTE` segments. The program header for such segment is defined as:

p_type
`PT_AARCH64_MEMTAG_MTE`

p_flags
0

p_offset
: segment file offset

p_vaddr
: segment virtual address, same as the corresponding PT_LOAD segment

p_paddr
: 0

p_filesz
: segment size in file, calculated as p_mem_sz / 32 (two 4-bit tags cover 32 bytes of memory)

p_memsz
: segment size in memory, same as the corresponding PT_LOAD segment

p_align
: 0

The tags are stored in the core file at p_offset as two 4-bit tags in a byte. With the tag granule of 16 bytes, a 4K page requires 128 bytes in the core file.

3.12.3 Example of correct usage

MTE Example code

```
/*
 * To be compiled with -march=armv8.5-a+memtag
 */
#include <errno.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/auxv.h>
#include <sys/mman.h>
#include <sys/prctl.h>

/*
 * From arch/arm64/include/uapi/asm/hwcap.h
 */
#define HWCAP2_MTE              (1 << 18)

/*
 * From arch/arm64/include/uapi/asm/mman.h
 */
#define PROT_MTE                 0x20

/*
 * From include/uapi/linux/prctl.h
 */
#define PR_SET_TAGGED_ADDR_CTRL 55
#define PR_GET_TAGGED_ADDR_CTRL 56
# define PR_TAGGED_ADDR_ENABLE  (1UL << 0)
# define PR_MTE_TCF_SHIFT       1
# define PR_MTE_TCF_NONE        (0UL << PR_MTE_TCF_SHIFT)
# define PR_MTE_TCF_SYNC        (1UL << PR_MTE_TCF_SHIFT)
# define PR_MTE_TCF_ASYNC       (2UL << PR_MTE_TCF_SHIFT)
# define PR_MTE_TCF_MASK        (3UL << PR_MTE_TCF_SHIFT)
# define PR_MTE_TAG_SHIFT       3
# define PR_MTE_TAG_MASK        (0xffffUL << PR_MTE_TAG_SHIFT)

/*
 * Insert a random logical tag into the given pointer.
 */
#define insert_random_tag(ptr) ({                       \
        uint64_t __val;                                 \
        asm("irg %0, %1" : "=r" (__val) : "r" (ptr));   \
        __val;                                          \
})

/*
 * Set the allocation tag on the destination address.
 */
#define set_tag(tagged_addr) do {                                      \
        asm volatile("stg %0, [%0]" : : "r" (tagged_addr) : "memory"); \
} while (0)

int main()
{
        unsigned char *a;
        unsigned long page_sz = sysconf(_SC_PAGESIZE);
        unsigned long hwcap2 = getauxval(AT_HWCAP2);

        /* check if MTE is present */
        if (!(hwcap2 & HWCAP2_MTE))
                return EXIT_FAILURE;

        /*
         * Enable the tagged address ABI, synchronous or asynchronous MTE
         * tag check faults (based on per-CPU preference) and allow all
         * non-zero tags in the randomly generated set.
         */
        if (prctl(PR_SET_TAGGED_ADDR_CTRL,
                  PR_TAGGED_ADDR_ENABLE | PR_MTE_TCF_SYNC | PR_MTE_TCF_ASYNC |
                  (0xfffe << PR_MTE_TAG_SHIFT),
                  0, 0, 0)) {
                perror("prctl() failed");
                return EXIT_FAILURE;
        }

        a = mmap(0, page_sz, PROT_READ | PROT_WRITE,
                 MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
        if (a == MAP_FAILED) {
                perror("mmap() failed");
                return EXIT_FAILURE;
        }

        /*
         * Enable MTE on the above anonymous mmap. The flag could be passed
         * directly to mmap() and skip this step.
         */
        if (mprotect(a, page_sz, PROT_READ | PROT_WRITE | PROT_MTE)) {
                perror("mprotect() failed");
                return EXIT_FAILURE;
        }

        /* access with the default tag (0) */
        a[0] = 1;
        a[1] = 2;

        printf("a[0] = %hhu a[1] = %hhu\n", a[0], a[1]);

        /* set the logical and allocation tags */
        a = (unsigned char *)insert_random_tag(a);
        set_tag(a);

        printf("%p\n", a);

        /* non-zero tag access */
        a[0] = 3;
        printf("a[0] = %hhu a[1] = %hhu\n", a[0], a[1]);

        /*
         * If MTE is enabled correctly the next instruction will generate an
         * exception.
         */
        printf("Expecting SIGSEGV...\n");
        a[16] = 0xdd;

        /* this should not be printed in the PR_MTE_TCF_SYNC mode */
        printf("...haven't got one\n");

        return EXIT_FAILURE;
}
```

3.13 Perf

3.13.1 Perf Event Attributes

Author
Andrew Murray <andrew.murray@arm.com>

Date
2019-03-06

exclude_user

This attribute excludes userspace.

Userspace always runs at EL0 and thus this attribute will exclude EL0.

exclude_kernel

This attribute excludes the kernel.

The kernel runs at EL2 with VHE and EL1 without. Guest kernels always run at EL1.

For the host this attribute will exclude EL1 and additionally EL2 on a VHE system.

For the guest this attribute will exclude EL1. Please note that EL2 is never counted within a guest.

exclude_hv

This attribute excludes the hypervisor.

For a VHE host this attribute is ignored as we consider the host kernel to be the hypervisor.

For a non-VHE host this attribute will exclude EL2 as we consider the hypervisor to be any code that runs at EL2 which is predominantly used for guest/host transitions.

For the guest this attribute has no effect. Please note that EL2 is never counted within a guest.

exclude_host / exclude_guest

These attributes exclude the KVM host and guest, respectively.

The KVM host may run at EL0 (userspace), EL1 (non-VHE kernel) and EL2 (VHE kernel or non-VHE hypervisor).

The KVM guest may run at EL0 (userspace) and EL1 (kernel).

Due to the overlapping exception levels between host and guests we cannot exclusively rely on the PMU's hardware exception filtering - therefore we must enable/disable counting on the entry and exit to the guest. This is performed differently on VHE and non-VHE systems.

For non-VHE systems we exclude EL2 for exclude_host - upon entering and exiting the guest we disable/enable the event as appropriate based on the exclude_host and exclude_guest attributes.

For VHE systems we exclude EL1 for exclude_guest and exclude both EL0,EL2 for exclude_host. Upon entering and exiting the guest we modify the event to include/exclude EL0 as appropriate based on the exclude_host and exclude_guest attributes.

The statements above also apply when these attributes are used within a non-VHE guest however please note that EL2 is never counted within a guest.

Accuracy

On non-VHE hosts we enable/disable counters on the entry/exit of host/guest transition at EL2 - however there is a period of time between enabling/disabling the counters and entering/exiting the guest. We are able to eliminate counters counting host events on the boundaries of guest entry/exit when counting guest events by filtering out EL2 for exclude_host. However when using !exclude_hv there is a small blackout window at the guest entry/exit where host events are not captured.

On VHE systems there are no blackout windows.

3.13.2 Perf Userspace PMU Hardware Counter Access

Overview

The perf userspace tool relies on the PMU to monitor events. It offers an abstraction layer over the hardware counters since the underlying implementation is cpu-dependent. Arm64 allows userspace tools to have access to the registers storing the hardware counters' values directly.

This targets specifically self-monitoring tasks in order to reduce the overhead by directly accessing the registers without having to go through the kernel.

How-to

The focus is set on the armv8 PMUv3 which makes sure that the access to the pmu registers is enabled and that the userspace has access to the relevant information in order to use them.

In order to have access to the hardware counters, the global sysctl kernel/perf_user_access must first be enabled:

```
echo 1 > /proc/sys/kernel/perf_user_access
```

It is necessary to open the event using the perf tool interface with config1:1 attr bit set: the sys_perf_event_open syscall returns a fd which can subsequently be used with the mmap syscall in order to retrieve a page of memory containing information about the event. The PMU driver uses this page to expose to the user the hardware counter's index and other necessary data. Using this index enables the user to access the PMU registers using the *mrs* instruction. Access to the PMU registers is only valid while the sequence lock is unchanged. In particular, the PMSELR_EL0 register is zeroed each time the sequence lock is changed.

The userspace access is supported in libperf using the perf_evsel__mmap() and perf_evsel__read() functions. See tools/lib/perf/tests/test-evsel.c for an example.

About heterogeneous systems

On heterogeneous systems such as big.LITTLE, userspace PMU counter access can only be enabled when the tasks are pinned to a homogeneous subset of cores and the corresponding PMU instance is opened by specifying the 'type' attribute. The use of generic event types is not supported in this case.

Have a look at tools/perf/arch/arm64/tests/user-events.c for an example. It can be run using the perf tool to check that the access to the registers works correctly from userspace:

```
perf test -v user
```

About chained events and counter sizes

The user can request either a 32-bit (config1:0 == 0) or 64-bit (config1:0 == 1) counter along with userspace access. The sys_perf_event_open syscall will fail if a 64-bit counter is requested and the hardware doesn't support 64-bit counters. Chained events are not supported in conjunction with userspace counter access. If a 32-bit counter is requested on hardware with 64-bit counters, then userspace must treat the upper 32-bits read from the counter as UNKNOWN. The 'pmc_width' field in the user page will indicate the valid width of the counter and should be used to mask the upper bits as needed.

3.14 Pointer authentication in AArch64 Linux

Author: Mark Rutland <mark.rutland@arm.com>

Date: 2017-07-19

This document briefly describes the provision of pointer authentication functionality in AArch64 Linux.

3.14.1 Architecture overview

The ARMv8.3 Pointer Authentication extension adds primitives that can be used to mitigate certain classes of attack where an attacker can corrupt the contents of some memory (e.g. the stack).

The extension uses a Pointer Authentication Code (PAC) to determine whether pointers have been modified unexpectedly. A PAC is derived from a pointer, another value (such as the stack pointer), and a secret key held in system registers.

The extension adds instructions to insert a valid PAC into a pointer, and to verify/remove the PAC from a pointer. The PAC occupies a number of high-order bits of the pointer, which varies dependent on the configured virtual address size and whether pointer tagging is in use.

A subset of these instructions have been allocated from the HINT encoding space. In the absence of the extension (or when disabled), these instructions behave as NOPs. Applications and libraries using these instructions operate correctly regardless of the presence of the extension.

The extension provides five separate keys to generate PACs - two for instruction addresses (APIAKey, APIBKey), two for data addresses (APDAKey, APDBKey), and one for generic authentication (APGAKey).

3.14.2 Basic support

When CONFIG_ARM64_PTR_AUTH is selected, and relevant HW support is present, the kernel will assign random key values to each process at exec*() time. The keys are shared by all threads within the process, and are preserved across fork().

Presence of address authentication functionality is advertised via HWCAP_PACA, and generic authentication functionality via HWCAP_PACG.

The number of bits that the PAC occupies in a pointer is 55 minus the virtual address size configured by the kernel. For example, with a virtual address size of 48, the PAC is 7 bits wide.

When ARM64_PTR_AUTH_KERNEL is selected, the kernel will be compiled with HINT space pointer authentication instructions protecting function returns. Kernels built with this option will work on hardware with or without pointer authentication support.

In addition to exec(), keys can also be reinitialized to random values using the PR_PAC_RESET_KEYS prctl. A bitmask of PR_PAC_APIAKEY, PR_PAC_APIBKEY, PR_PAC_APDAKEY, PR_PAC_APDBKEY and PR_PAC_APGAKEY specifies which keys are to be reinitialized; specifying 0 means "all keys".

3.14.3 Debugging

When CONFIG_ARM64_PTR_AUTH is selected, and HW support for address authentication is present, the kernel will expose the position of TTBR0 PAC bits in the NT_ARM_PAC_MASK regset (struct user_pac_mask), which userspace can acquire via PTRACE_GETREGSET.

The regset is exposed only when HWCAP_PACA is set. Separate masks are exposed for data pointers and instruction pointers, as the set of PAC bits can vary between the two. Note that the masks apply to TTBR0 addresses, and are not valid to apply to TTBR1 addresses (e.g. kernel pointers).

Additionally, when CONFIG_CHECKPOINT_RESTORE is also set, the kernel will expose the NT_ARM_PACA_KEYS and NT_ARM_PACG_KEYS regsets (struct user_pac_address_keys and struct user_pac_generic_keys). These can be used to get and set the keys for a thread.

3.14.4 Virtualization

Pointer authentication is enabled in KVM guest when each virtual cpu is initialised by passing flags KVM_ARM_VCPU_PTRAUTH_[ADDRESS/GENERIC] and requesting these two separate cpu features to be enabled. The current KVM guest implementation works by enabling both features together, so both these userspace flags are checked before enabling pointer authentication. The separate userspace flag will allow to have no userspace ABI changes if support is added in the future to allow these two features to be enabled independently of one another.

As Arm Architecture specifies that Pointer Authentication feature is implemented along with the VHE feature so KVM arm64 ptrauth code relies on VHE mode to be present.

Additionally, when these vcpu feature flags are not set then KVM will filter out the Pointer Authentication system key registers from KVM_GET/SET_REG_* ioctls and mask those features from cpufeature ID register. Any attempt to use the Pointer Authentication instructions will result in an UNDEFINED exception being injected into the guest.

3.14.5 Enabling and disabling keys

The prctl PR_PAC_SET_ENABLED_KEYS allows the user program to control which PAC keys are enabled in a particular task. It takes two arguments, the first being a bitmask of PR_PAC_APIAKEY, PR_PAC_APIBKEY, PR_PAC_APDAKEY and PR_PAC_APDBKEY specifying which keys shall be affected by this prctl, and the second being a bitmask of the same bits specifying whether the key should be enabled or disabled. For example:

```
prctl(PR_PAC_SET_ENABLED_KEYS,
      PR_PAC_APIAKEY | PR_PAC_APIBKEY | PR_PAC_APDAKEY | PR_PAC_APDBKEY,
      PR_PAC_APIBKEY, 0, 0);
```

disables all keys except the IB key.

The main reason why this is useful is to enable a userspace ABI that uses PAC instructions to sign and authenticate function pointers and other pointers exposed outside of the function, while still allowing binaries conforming to the ABI to interoperate with legacy binaries that do not sign or authenticate pointers.

The idea is that a dynamic loader or early startup code would issue this prctl very early after establishing that a process may load legacy binaries, but before executing any PAC instructions.

For compatibility with previous kernel versions, processes start up with IA, IB, DA and DB enabled, and are reset to this state on exec(). Processes created via fork() and clone() inherit the key enabled state from the calling process.

It is recommended to avoid disabling the IA key, as this has higher performance overhead than disabling any of the other keys.

3.15 Kernel page table dump

ptdump is a debugfs interface that provides a detailed dump of the kernel page tables. It offers a comprehensive overview of the kernel virtual memory layout as well as the attributes associated with the various regions in a human-readable format. It is useful to dump the kernel page tables to verify permissions and memory types. Examining the page table entries and permissions helps identify potential security vulnerabilities such as mappings with overly permissive access rights or improper memory protections.

Memory hotplug allows dynamic expansion or contraction of available memory without requiring a system reboot. To maintain the consistency and integrity of the memory management data structures, arm64 makes use of the `mem_hotplug_lock` semaphore in write mode. Additionally, in read mode, `mem_hotplug_lock` supports an efficient implementation of `get_online_mems()` and `put_online_mems()`. These protect the offlining of memory being accessed by the ptdump code.

In order to dump the kernel page tables, enable the following configurations and mount debugfs:

```
CONFIG_GENERIC_PTDUMP=y
CONFIG_PTDUMP_CORE=y
CONFIG_PTDUMP_DEBUGFS=y

mount -t debugfs nodev /sys/kernel/debug
cat /sys/kernel/debug/kernel_page_tables
```

On analysing the output of `cat /sys/kernel/debug/kernel_page_tables` one can derive information about the virtual address range of the entry, followed by size of the memory region covered by this entry, the hierarchical structure of the page tables and finally the attributes associated with each page. The page attributes provide information about access permissions, execution capability, type of mapping such as leaf level PTE or block level PGD, PMD and PUD, and access status of a page within the kernel memory. Assessing these attributes can assist in understanding the memory layout, access patterns and security characteristics of the kernel pages.

Kernel virtual memory layout example:

```
start address        end address         size             attributes
+------------------------------------------------------------------------------
↪---------+
| ---[ Linear Mapping start ]--------------------------------------------------
↪-------- |
| ..................                                                          ␣
↪         |
| 0xfff0000000000000-0xfff0000000210000  2112K PTE RW NX SHD AF   UXN  MEM/
↪NORMAL-TAGGED |
| 0xfff0000000210000-0xfff0000001c00000 26560K PTE ro NX SHD AF   UXN  MEM/
↪NORMAL        |
| ..................                                                          ␣
↪         |
| ---[ Linear Mapping end ]----------------------------------------------------
↪-------- |
+------------------------------------------------------------------------------
↪---------+
| ---[ Modules start ]---------------------------------------------------------
↪-------- |
| ..................                                                          ␣
↪         |
| 0xffff800000000000-0xffff800008000000   128M PTE                            ␣
↪         |
| ..................                                                          ␣
↪         |
| ---[ Modules end ]-----------------------------------------------------------
↪-------- |
+------------------------------------------------------------------------------
↪---------+
| ---[ vmalloc() area ]--------------------------------------------------------
↪-------- |
| ..................                                                          ␣
↪         |
| 0xffff800008010000-0xffff800008200000  1984K PTE ro x  SHD AF       UXN  MEM/
↪NORMAL   |
| 0xffff800008200000-0xffff800008e00000    12M PTE ro x  SHD AF   CON  UXN  MEM/
↪NORMAL   |
| ..................                                                          ␣
↪         |
| ---[ vmalloc() end ]---------------------------------------------------------
↪-------- |
+-------------------------------------------------------------------------------
↪---------+
| ---[ Fixmap start ]-----------------------------------------------------------
↪-------- |
| ..................                                                            ␣
↪         |
| 0xfffffbfffdb80000-0xfffffbfffdb90000    64K PTE ro x  SHD AF  UXN  MEM/
↪NORMAL        |
| 0xfffffbfffdb90000-0xfffffbfffdba0000    64K PTE ro NX SHD AF  UXN  MEM/
↪NORMAL        |
| ..................                                                            ␣
↪         |
| ---[ Fixmap end ]-------------------------------------------------------------
↪-------- |
+-------------------------------------------------------------------------------
↪---------+
| ---[ PCI I/O start ]----------------------------------------------------------
↪-------- |
| ..................                                                            ␣
↪         |
| 0xfffffbfffe800000-0xfffffbffff800000    16M PTE                              ␣
↪         |
| ..................                                                            ␣
↪         |
| ---[ PCI I/O end ]------------------------------------------------------------
↪-------- |
+-------------------------------------------------------------------------------
↪---------+
| ---[ vmemmap start ]----------------------------------------------------------
↪-------- |
| ..................                                                            ␣
↪         |
| 0xfffffc0002000000-0xfffffc0002200000     2M PTE RW NX SHD AF  UXN  MEM/
↪NORMAL        |
| 0xfffffc0002200000-0xfffffc0020000000   478M PTE                              ␣
↪         |
| ..................                                                            ␣
↪         |
| ---[ vmemmap end ]------------------------------------------------------------
↪-------- |
+-------------------------------------------------------------------------------
↪---------+
```

`cat /sys/kernel/debug/kernel_page_tables` output:

```
0xfff0000001c00000-0xfff0000080000000     2020M PTE  RW NX SHD AF   UXN    MEM/
↪NORMAL-TAGGED
0xfff0000080000000-0xfff0000800000000       30G PMD
0xfff0000800000000-0xfff0000800700000        7M PTE  RW NX SHD AF   UXN    MEM/
↪NORMAL-TAGGED
0xfff0000800700000-0xfff0000800710000       64K PTE  ro NX SHD AF   UXN    MEM/
```

```
↪NORMAL-TAGGED
0xfff0000800710000-0xfff0000880000000  2089920K PTE  RW NX SHD AF   UXN    MEM/
↪NORMAL-TAGGED
0xfff0000880000000-0xfff0040000000000     4062G PMD
0xfff0040000000000-0xffff800000000000     3964T PGD
```

3.16 Silicon Errata and Software Workarounds

Author: Will Deacon <will.deacon@arm.com>

Date : 27 November 2015

It is an unfortunate fact of life that hardware is often produced with so-called "errata", which can cause it to deviate from the architecture under specific circumstances. For hardware produced by ARM, these errata are broadly classified into the following categories:

Category A	A critical error without a viable workaround.
Category B	A significant or critical error with an acceptable workaround.
Category C	A minor error that is not expected to occur under normal operation.

For more information, consult one of the "Software Developers Errata Notice" documents available on infocenter.arm.com (registration required).

As far as Linux is concerned, Category B errata may require some special treatment in the operating system. For example, avoiding a particular sequence of code, or configuring the processor in a particular way. A less common situation may require similar actions in order to declassify a Category A erratum into a Category C erratum. These are collectively known as "software workarounds" and are only required in the minority of cases (e.g. those cases that both require a non-secure workaround *and* can be triggered by Linux).

For software workarounds that may adversely impact systems unaffected by the erratum in question, a Kconfig entry is added under "Kernel Features" -> "ARM errata workarounds via the alternatives framework". These are enabled by default and patched in at runtime when an affected CPU is detected. For less-intrusive workarounds, a Kconfig option is not available and the code is structured (preferably with a comment) in such a way that the erratum will not be hit.

This approach can make it slightly onerous to determine exactly which errata are worked around in an arbitrary kernel source tree, so this file acts as a registry of software workarounds in the Linux Kernel and will be updated when new workarounds are committed and backported to stable kernels.

Implementor	Component	Erratum ID	Kconfig
Allwinner	A64/R18	UNKNOWN1	SUN50I_ERRA
Ampere	AmpereOne	AC03_CPU_38	AMPERE_ERR
ARM	Cortex-A510	#2457168	ARM64_ERRA
ARM	Cortex-A510	#2064142	ARM64_ERRA

co

Table 3 – continued from previous page

Implementor	Component	Erratum ID	Kconfig
ARM	Cortex-A510	#2038923	ARM64_ERRAT
ARM	Cortex-A510	#1902691	ARM64_ERRAT
ARM	Cortex-A510	#2051678	ARM64_ERRAT
ARM	Cortex-A510	#2077057	ARM64_ERRAT
ARM	Cortex-A510	#2441009	ARM64_ERRAT
ARM	Cortex-A510	#2658417	ARM64_ERRAT
ARM	Cortex-A510	#3117295	ARM64_ERRAT
ARM	Cortex-A520	#2966298	ARM64_ERRAT
ARM	Cortex-A53	#826319	ARM64_ERRAT
ARM	Cortex-A53	#827319	ARM64_ERRAT
ARM	Cortex-A53	#824069	ARM64_ERRAT
ARM	Cortex-A53	#819472	ARM64_ERRAT
ARM	Cortex-A53	#845719	ARM64_ERRAT
ARM	Cortex-A53	#843419	ARM64_ERRAT
ARM	Cortex-A55	#1024718	ARM64_ERRAT
ARM	Cortex-A55	#1530923	ARM64_ERRAT
ARM	Cortex-A55	#2441007	ARM64_ERRAT
ARM	Cortex-A57	#832075	ARM64_ERRAT
ARM	Cortex-A57	#852523	N/A
ARM	Cortex-A57	#834220	ARM64_ERRAT
ARM	Cortex-A57	#1319537	ARM64_ERRAT
ARM	Cortex-A57	#1742098	ARM64_ERRAT
ARM	Cortex-A72	#853709	N/A
ARM	Cortex-A72	#1319367	ARM64_ERRAT
ARM	Cortex-A72	#1655431	ARM64_ERRAT
ARM	Cortex-A73	#858921	ARM64_ERRAT
ARM	Cortex-A76	#1188873,1418040	ARM64_ERRAT
ARM	Cortex-A76	#1165522	ARM64_ERRAT
ARM	Cortex-A76	#1286807	ARM64_ERRAT
ARM	Cortex-A76	#1463225	ARM64_ERRAT
ARM	Cortex-A77	#1508412	ARM64_ERRAT
ARM	Cortex-A710	#2119858	ARM64_ERRAT
ARM	Cortex-A710	#2054223	ARM64_ERRAT
ARM	Cortex-A710	#2224489	ARM64_ERRAT
ARM	Cortex-A715	#2645198	ARM64_ERRAT
ARM	Cortex-X2	#2119858	ARM64_ERRAT
ARM	Cortex-X2	#2224489	ARM64_ERRAT
ARM	Neoverse-N1	#1188873,1418040	ARM64_ERRAT
ARM	Neoverse-N1	#1349291	N/A
ARM	Neoverse-N1	#1542419	ARM64_ERRAT
ARM	Neoverse-N2	#2139208	ARM64_ERRAT
ARM	Neoverse-N2	#2067961	ARM64_ERRAT
ARM	Neoverse-N2	#2253138	ARM64_ERRAT
ARM	MMU-500	#841119,826419	N/A
ARM	MMU-600	#1076982,1209401	N/A
ARM	MMU-700	#2268618,2812531	N/A
ARM	GIC-700	#2941627	ARM64_ERRAT

co

Table 3 – continued from previous page

Implementor	Component	Erratum ID	Kconfig
Broadcom	Brahma-B53	N/A	ARM64_ERRAT
Broadcom	Brahma-B53	N/A	ARM64_ERRAT
Cavium	ThunderX ITS	#22375,24313	CAVIUM_ERRA
Cavium	ThunderX ITS	#23144	CAVIUM_ERRA
Cavium	ThunderX GICv3	#23154,38545	CAVIUM_ERRA
Cavium	ThunderX GICv3	#38539	N/A
Cavium	ThunderX Core	#27456	CAVIUM_ERRA
Cavium	ThunderX Core	#30115	CAVIUM_ERRA
Cavium	ThunderX SMMUv2	#27704	N/A
Cavium	ThunderX2 SMMUv3	#74	N/A
Cavium	ThunderX2 SMMUv3	#126	N/A
Cavium	ThunderX2 Core	#219	CAVIUM_TX2_
Marvell	ARM-MMU-500	#582743	N/A
NVIDIA	Carmel Core	N/A	NVIDIA_CARM
NVIDIA	T241 GICv3/4.x	T241-FABRIC-4	N/A
Freescale/NXP	LS2080A/LS1043A	A-008585	FSL_ERRATU
Hisilicon	Hip0{5,6,7}	#161010101	HISILICON_E
Hisilicon	Hip0{6,7}	#161010701	N/A
Hisilicon	Hip0{6,7}	#161010803	N/A
Hisilicon	Hip07	#161600802	HISILICON_E
Hisilicon	Hip08 SMMU PMCG	#162001800	N/A
Hisilicon	Hip08 SMMU PMCG Hip09 SMMU PMCG	#162001900	N/A
Qualcomm Tech.	Kryo/Falkor v1	E1003	QCOM_FALKO
Qualcomm Tech.	Kryo/Falkor v1	E1009	QCOM_FALKO
Qualcomm Tech.	QDF2400 ITS	E0065	QCOM_QDF24
Qualcomm Tech.	Falkor v{1,2}	E1041	QCOM_FALKO
Qualcomm Tech.	Kryo4xx Gold	N/A	ARM64_ERRAT
Qualcomm Tech.	Kryo4xx Gold	N/A	ARM64_ERRAT
Qualcomm Tech.	Kryo4xx Silver	N/A	ARM64_ERRAT
Qualcomm Tech.	Kryo4xx Silver	N/A	ARM64_ERRAT
Qualcomm Tech.	Kryo4xx Gold	N/A	ARM64_ERRAT
Rockchip	RK3588	#3588001	ROCKCHIP_E

Fujitsu	A64FX	E#010001	FUJITSU_ERRATUM_010001

ASR	ASR8601	#8601001	N/A
Microsoft	Azure Cobalt 100	#2139208	ARM64_ERRATUM_2139208
Microsoft	Azure Cobalt 100	#2067961	ARM64_ERRATUM_2067961
Microsoft	Azure Cobalt 100	#2253138	ARM64_ERRATUM_2253138

3.17 Scalable Matrix Extension support for AArch64 Linux

This document outlines briefly the interface provided to userspace by Linux in order to support use of the ARM Scalable Matrix Extension (SME).

This is an outline of the most important features and issues only and not intended to be exhaustive. It should be read in conjunction with the SVE documentation in sve.rst which provides details on the Streaming SVE mode included in SME.

This document does not aim to describe the SME architecture or programmer's model. To aid understanding, a minimal description of relevant programmer's model features for SME is included in Appendix A.

3.17.1 1. General

- PSTATE.SM, PSTATE.ZA, the streaming mode vector length, the ZA and (when present) ZTn register state and TPIDR2_EL0 are tracked per thread.
- The presence of SME is reported to userspace via HWCAP2_SME in the aux vector AT_HWCAP2 entry. Presence of this flag implies the presence of the SME instructions and registers, and the Linux-specific system interfaces described in this document. SME is reported in /proc/cpuinfo as "sme".
- The presence of SME2 is reported to userspace via HWCAP2_SME2 in the aux vector AT_HWCAP2 entry. Presence of this flag implies the presence of the SME2 instructions and ZT0, and the Linux-specific system interfaces described in this document. SME2 is reported in /proc/cpuinfo as "sme2".
- Support for the execution of SME instructions in userspace can also be detected by reading the CPU ID register ID_AA64PFR1_EL1 using an MRS instruction, and checking that the value of the SME field is nonzero. [3]

  It does not guarantee the presence of the system interfaces described in the following sections: software that needs to verify that those interfaces are present must check for HWCAP2_SME instead.
- There are a number of optional SME features, presence of these is reported through AT_HWCAP2 through:

  HWCAP2_SME_I16I64 HWCAP2_SME_F64F64 HWCAP2_SME_I8I32 HW-CAP2_SME_F16F32 HWCAP2_SME_B16F32 HWCAP2_SME_F32F32 HW-CAP2_SME_FA64 HWCAP2_SME2

  This list may be extended over time as the SME architecture evolves.

These extensions are also reported via the CPU ID register ID_AA64SMFR0_EL1, which userspace can read using an MRS instruction. See elf_hwcaps.txt and cpu-feature-registers.txt for details.

- Debuggers should restrict themselves to interacting with the target via the NT_ARM_SVE, NT_ARM_SSVE, NT_ARM_ZA and NT_ARM_ZT regsets. The recommended way of detecting support for these regsets is to connect to a target process first and then attempt a

  ptrace(PTRACE_GETREGSET, pid, NT_ARM_<regset>, &iov).
- Whenever ZA register values are exchanged in memory between userspace and the kernel, the register value is encoded in memory as a series of horizontal vectors from 0 to VL/8-1 stored in the same endianness invariant format as is used for SVE vectors.
- On thread creation TPIDR2_EL0 is preserved unless CLONE_SETTLS is specified, in which case it is set to 0.

3.17.2 2. Vector lengths

SME defines a second vector length similar to the SVE vector length which is controls the size of the streaming mode SVE vectors and the ZA matrix array. The ZA matrix is square with each side having as many bytes as a streaming mode SVE vector.

3.17.3 3. Sharing of streaming and non-streaming mode SVE state

It is implementation defined which if any parts of the SVE state are shared between streaming and non-streaming modes. When switching between modes via software interfaces such as ptrace if no register content is provided as part of switching no state will be assumed to be shared and everything will be zeroed.

3.17.4 4. System call behaviour

- On syscall PSTATE.ZA is preserved, if PSTATE.ZA==1 then the contents of the ZA matrix and ZTn (if present) are preserved.
- On syscall PSTATE.SM will be cleared and the SVE registers will be handled as per the standard SVE ABI.
- None of the SVE registers, ZA or ZTn are used to pass arguments to or receive results from any syscall.
- On process creation (eg, clone()) the newly created process will have PSTATE.SM cleared.
- All other SME state of a thread, including the currently configured vector length, the state of the PR_SME_VL_INHERIT flag, and the deferred vector length (if any), is preserved across all syscalls, subject to the specific exceptions for execve() described in section 6.

3.17.5 5. Signal handling

- Signal handlers are invoked with streaming mode and ZA disabled.
- A new signal frame record TPIDR2_MAGIC is added formatted as a struct tpidr2_context to allow access to TPIDR2_EL0 from signal handlers.
- A new signal frame record za_context encodes the ZA register contents on signal delivery. [1]
- The signal frame record for ZA always contains basic metadata, in particular the thread's vector length (in za_context.vl).
- The ZA matrix may or may not be included in the record, depending on the value of PSTATE.ZA. The registers are present if and only if: za_context.head.size >= ZA_SIG_CONTEXT_SIZE(sve_vq_from_vl(za_context.vl)) in which case PSTATE.ZA == 1.
- If matrix data is present, the remainder of the record has a vl-dependent size and layout. Macros ZA_SIG_* are defined [1] to facilitate access to them.
- The matrix is stored as a series of horizontal vectors in the same format as is used for SVE vectors.
- If the ZA context is too big to fit in sigcontext.__reserved[], then extra space is allocated on the stack, an extra_context record is written in __reserved[] referencing this space. za_context is then written in the extra space. Refer to [1] for further details about this mechanism.
- If ZTn is supported and PSTATE.ZA==1 then a signal frame record for ZTn will be generated.
- The signal record for ZTn has magic ZT_MAGIC (0x5a544e01) and consists of a standard signal frame header followed by a struct zt_context specifying the number of ZTn registers supported by the system, then zt_context.nregs blocks of 64 bytes of data per register.

3.17.6 5. Signal return

When returning from a signal handler:

- If there is no za_context record in the signal frame, or if the record is present but contains no register data as described in the previous section, then ZA is disabled.
- If za_context is present in the signal frame and contains matrix data then PSTATE.ZA is set to 1 and ZA is populated with the specified data.
- The vector length cannot be changed via signal return. If za_context.vl in the signal frame does not match the current vector length, the signal return attempt is treated as illegal, resulting in a forced SIGSEGV.
- If ZTn is not supported or PSTATE.ZA==0 then it is illegal to have a signal frame record for ZTn, resulting in a forced SIGSEGV.

3.17.7 6. prctl extensions

Some new prctl() calls are added to allow programs to manage the SME vector length:

prctl(PR_SME_SET_VL, unsigned long arg)

Sets the vector length of the calling thread and related flags, where arg == vl | flags. Other threads of the calling process are unaffected.

vl is the desired vector length, where sve_vl_valid(vl) must be true.

flags:

PR_SME_VL_INHERIT

Inherit the current vector length across execve(). Otherwise, the vector length is reset to the system default at execve(). (See Section 9.)

PR_SME_SET_VL_ONEXEC

Defer the requested vector length change until the next execve() performed by this thread.

The effect is equivalent to implicit execution of the following call immediately after the next execve() (if any) by the thread:

prctl(PR_SME_SET_VL, arg & ~PR_SME_SET_VL_ONEXEC)

This allows launching of a new program with a different vector length, while avoiding runtime side effects in the caller.

Without PR_SME_SET_VL_ONEXEC, the requested change takes effect immediately.

Return value: a nonnegative on success, or a negative value on error:

EINVAL: SME not supported, invalid vector length requested, or invalid flags.

On success:

- Either the calling thread's vector length or the deferred vector length to be applied at the next execve() by the thread (dependent on whether PR_SME_SET_VL_ONEXEC is present in arg), is set to the largest value supported by the system that is less than or equal to vl. If vl == SVE_VL_MAX, the value set will be the largest value supported by the system.
- Any previously outstanding deferred vector length change in the calling thread is cancelled.
- The returned value describes the resulting configuration, encoded as for PR_SME_GET_VL. The vector length reported in this value is the new current vector length for this thread if PR_SME_SET_VL_ONEXEC was not present in arg; otherwise, the reported vector length is the deferred vector length that will be applied at the next execve() by the calling thread.
- Changing the vector length causes all of ZA, ZTn, P0..P15, FFR and all bits of Z0..Z31 except for Z0 bits [127:0] .. Z31 bits [127:0] to become unspecified, including both streaming and non-streaming SVE state. Calling PR_SME_SET_VL with vl equal to the thread's current vector length, or calling PR_SME_SET_VL

with the PR_SVE_SET_VL_ONEXEC flag, does not constitute a change to the vector length for this purpose.

- Changing the vector length causes PSTATE.ZA and PSTATE.SM to be cleared. Calling PR_SME_SET_VL with vl equal to the thread's current vector length, or calling PR_SME_SET_VL with the PR_SVE_SET_VL_ONEXEC flag, does not constitute a change to the vector length for this purpose.

prctl(PR_SME_GET_VL)

Gets the vector length of the calling thread.

The following flag may be OR-ed into the result:

PR_SME_VL_INHERIT

Vector length will be inherited across execve().

There is no way to determine whether there is an outstanding deferred vector length change (which would only normally be the case between a fork() or vfork() and the corresponding execve() in typical use).

To extract the vector length from the result, bitwise and it with PR_SME_VL_LEN_MASK.

Return value: a nonnegative value on success, or a negative value on error:
EINVAL: SME not supported.

3.17.8 7. ptrace extensions

- A new regset NT_ARM_SSVE is defined for access to streaming mode SVE state via PTRACE_GETREGSET and PTRACE_SETREGSET, this is documented in sve.rst.
- A new regset NT_ARM_ZA is defined for ZA state for access to ZA state via PTRACE_GETREGSET and PTRACE_SETREGSET.

  Refer to [2] for definitions.

The regset data starts with struct user_za_header, containing:

size

Size of the complete regset, in bytes. This depends on vl and possibly on other things in the future.

If a call to PTRACE_GETREGSET requests less data than the value of size, the caller can allocate a larger buffer and retry in order to read the complete regset.

max_size

Maximum size in bytes that the regset can grow to for the target thread. The regset won't grow bigger than this even if the target thread changes its vector length etc.

vl

Target thread's current streaming vector length, in bytes.

max_vl

Maximum possible streaming vector length for the target thread.

flags

Zero or more of the following flags, which have the same meaning and behaviour as the corresponding PR_SET_VL_* flags:

SME_PT_VL_INHERIT

SME_PT_VL_ONEXEC (SETREGSET only).

- The effects of changing the vector length and/or flags are equivalent to those documented for PR_SME_SET_VL.

  The caller must make a further GETREGSET call if it needs to know what VL is actually set by SETREGSET, unless is it known in advance that the requested VL is supported.

- The size and layout of the payload depends on the header fields. The ZA_PT_ZA*() macros are provided to facilitate access to the data.
- In either case, for SETREGSET it is permissible to omit the payload, in which case the vector length and flags are changed and PSTATE.ZA is set to 0 (along with any consequences of those changes). If a payload is provided then PSTATE.ZA will be set to 1.
- For SETREGSET, if the requested VL is not supported, the effect will be the same as if the payload were omitted, except that an EIO error is reported. No attempt is made to translate the payload data to the correct layout for the vector length actually set. It is up to the caller to translate the payload layout for the actual VL and retry.
- The effect of writing a partial, incomplete payload is unspecified.
- A new regset NT_ARM_ZT is defined for access to ZTn state via PTRACE_GETREGSET and PTRACE_SETREGSET.
- The NT_ARM_ZT regset consists of a single 512 bit register.
- When PSTATE.ZA==0 reads of NT_ARM_ZT will report all bits of ZTn as 0.
- Writes to NT_ARM_ZT will set PSTATE.ZA to 1.

3.17.9 8. ELF coredump extensions

- NT_ARM_SSVE notes will be added to each coredump for each thread of the dumped process. The contents will be equivalent to the data that would have been read if a PTRACE_GETREGSET of the corresponding type were executed for each thread when the coredump was generated.
- A NT_ARM_ZA note will be added to each coredump for each thread of the dumped process. The contents will be equivalent to the data that would have been read if a PTRACE_GETREGSET of NT_ARM_ZA were executed for each thread when the coredump was generated.
- A NT_ARM_ZT note will be added to each coredump for each thread of the dumped process. The contents will be equivalent to the data that would have been read if a PTRACE_GETREGSET of NT_ARM_ZT were executed for each thread when the coredump was generated.

- The NT_ARM_TLS note will be extended to two registers, the second register will contain TPIDR2_EL0 on systems that support SME and will be read as zero with writes ignored otherwise.

3.17.10 9. System runtime configuration

- To mitigate the ABI impact of expansion of the signal frame, a policy mechanism is provided for administrators, distro maintainers and developers to set the default vector length for userspace processes:

/proc/sys/abi/sme_default_vector_length

> Writing the text representation of an integer to this file sets the system default vector length to the specified value, unless the value is greater than the maximum vector length supported by the system in which case the default vector length is set to that maximum.
>
> The result can be determined by reopening the file and reading its contents.
>
> At boot, the default vector length is initially set to 32 or the maximum supported vector length, whichever is smaller and supported. This determines the initial vector length of the init process (PID 1).
>
> Reading this file returns the current system default vector length.

- At every execve() call, the new vector length of the new process is set to the system default vector length, unless
  - PR_SME_VL_INHERIT (or equivalently SME_PT_VL_INHERIT) is set for the calling thread, or
  - a deferred vector length change is pending, established via the PR_SME_SET_VL_ONEXEC flag (or SME_PT_VL_ONEXEC).
- Modifying the system default vector length does not affect the vector length of any existing process or thread that does not make an execve() call.

Appendix A. SME programmer’s model (informative)

This section provides a minimal description of the additions made by SME to the ARMv8-A programmer’s model that are relevant to this document.

Note: This section is for information only and not intended to be complete or to replace any architectural specification.

3.17.11 A.1. Registers

In A64 state, SME adds the following:

- A new mode, streaming mode, in which a subset of the normal FPSIMD and SVE features are available. When supported EL0 software may enter and leave streaming mode at any time.

  For best system performance it is strongly encouraged for software to enable streaming mode only when it is actively being used.

- A new vector length controlling the size of ZA and the Z registers when in streaming mode, separately to the vector length used for SVE when not in streaming mode. There is no requirement that either the currently selected vector length or the set of vector lengths supported for the two modes in a given system have any relationship. The streaming mode vector length is referred to as SVL.

- A new ZA matrix register. This is a square matrix of SVLxSVL bits. Most operations on ZA require that streaming mode be enabled but ZA can be enabled without streaming mode in order to load, save and retain data.

  For best system performance it is strongly encouraged for software to enable ZA only when it is actively being used.

- A new ZT0 register is introduced when SME2 is present. This is a 512 bit register which is accessible when PSTATE.ZA is set, as ZA itself is.

- Two new 1 bit fields in PSTATE which may be controlled via the SMSTART and SMSTOP instructions or by access to the SVCR system register:

  - PSTATE.ZA, if this is 1 then the ZA matrix is accessible and has valid data while if it is 0 then ZA can not be accessed. When PSTATE.ZA is changed from 0 to 1 all bits in ZA are cleared.

  - PSTATE.SM, if this is 1 then the PE is in streaming mode. When the value of PSTATE.SM is changed then it is implementation defined if the subset of the floating point register bits valid in both modes may be retained. Any other bits will be cleared.

References

[1] arch/arm64/include/uapi/asm/sigcontext.h
AArch64 Linux signal ABI definitions

[2] arch/arm64/include/uapi/asm/ptrace.h
AArch64 Linux ptrace ABI definitions

[3] *ARM64 CPU Feature Registers*

3.18 Scalable Vector Extension support for AArch64 Linux

Author: Dave Martin <Dave.Martin@arm.com>

Date: 4 August 2017

This document outlines briefly the interface provided to userspace by Linux in order to support use of the ARM Scalable Vector Extension (SVE), including interactions with Streaming SVE mode added by the Scalable Matrix Extension (SME).

This is an outline of the most important features and issues only and not intended to be exhaustive.

This document does not aim to describe the SVE architecture or programmer's model. To aid understanding, a minimal description of relevant programmer's model features for SVE is included in Appendix A.

3.18.1 1. General

- SVE registers Z0..Z31, P0..P15 and FFR and the current vector length VL, are tracked per-thread.
- In streaming mode FFR is not accessible unless HWCAP2_SME_FA64 is present in the system, when it is not supported and these interfaces are used to access streaming mode FFR is read and written as zero.
- The presence of SVE is reported to userspace via HWCAP_SVE in the aux vector AT_HWCAP entry. Presence of this flag implies the presence of the SVE instructions and registers, and the Linux-specific system interfaces described in this document. SVE is reported in /proc/cpuinfo as "sve".
- Support for the execution of SVE instructions in userspace can also be detected by reading the CPU ID register ID_AA64PFR0_EL1 using an MRS instruction, and checking that the value of the SVE field is nonzero. [3]

  It does not guarantee the presence of the system interfaces described in the following sections: software that needs to verify that those interfaces are present must check for HWCAP_SVE instead.
- On hardware that supports the SVE2 extensions, HWCAP2_SVE2 will also be reported in the AT_HWCAP2 aux vector entry. In addition to this, optional extensions to SVE2 may be reported by the presence of:

  > HWCAP2_SVE2 HWCAP2_SVEAES HWCAP2_SVEPMULL HW-CAP2_SVEBITPERM HWCAP2_SVESHA3 HWCAP2_SVESM4 HWCAP2_SVE2P1

  This list may be extended over time as the SVE architecture evolves.

  These extensions are also reported via the CPU ID register ID_AA64ZFR0_EL1, which userspace can read using an MRS instruction. See elf_hwcaps.txt and cpu-feature-registers.txt for details.
- On hardware that supports the SME extensions, HWCAP2_SME will also be reported in the AT_HWCAP2 aux vector entry. Among other things SME adds streaming mode which provides a subset of the SVE feature set using a separate SME vector length and the same Z/V registers. See sme.rst for more details.

- Debuggers should restrict themselves to interacting with the target via the NT_ARM_SVE regset. The recommended way of detecting support for this regset is to connect to a target process first and then attempt a ptrace(PTRACE_GETREGSET, pid, NT_ARM_SVE, &iov). Note that when SME is present and streaming SVE mode is in use the FPSIMD subset of registers will be read via NT_ARM_SVE and NT_ARM_SVE writes will exit streaming mode in the target.
- Whenever SVE scalable register values (Zn, Pn, FFR) are exchanged in memory between userspace and the kernel, the register value is encoded in memory in an endianness-invariant layout, with bits [(8 * i + 7) : (8 * i)] encoded at byte offset i from the start of the memory representation. This affects for example the signal frame (struct sve_context) and ptrace interface (struct user_sve_header) and associated data.

  Beware that on big-endian systems this results in a different byte order than for the FPSIMD V-registers, which are stored as single host-endian 128-bit values, with bits [(127 - 8 * i) : (120 - 8 * i)] of the register encoded at byte offset i. (struct fpsimd_context, struct user_fpsimd_state).

3.18.2 2. Vector length terminology

The size of an SVE vector (Z) register is referred to as the "vector length".

To avoid confusion about the units used to express vector length, the kernel adopts the following conventions:

- Vector length (VL) = size of a Z-register in bytes
- Vector quadwords (VQ) = size of a Z-register in units of 128 bits

(So, VL = 16 * VQ.)

The VQ convention is used where the underlying granularity is important, such as in data structure definitions. In most other situations, the VL convention is used. This is consistent with the meaning of the "VL" pseudo-register in the SVE instruction set architecture.

3.18.3 3. System call behaviour

- On syscall, V0..V31 are preserved (as without SVE). Thus, bits [127:0] of Z0..Z31 are preserved. All other bits of Z0..Z31, and all of P0..P15 and FFR become zero on return from a syscall.
- The SVE registers are not used to pass arguments to or receive results from any syscall.
- In practice the affected registers/bits will be preserved or will be replaced with zeros on return from a syscall, but userspace should not make assumptions about this. The kernel behaviour may vary on a case-by-case basis.
- All other SVE state of a thread, including the currently configured vector length, the state of the PR_SVE_VL_INHERIT flag, and the deferred vector length (if any), is preserved across all syscalls, subject to the specific exceptions for execve() described in section 6.

  In particular, on return from a fork() or clone(), the parent and new child process or thread share identical SVE configuration, matching that of the parent before the call.

3.18.4 4. Signal handling

- A new signal frame record sve_context encodes the SVE registers on signal delivery. [1]
- This record is supplementary to fpsimd_context. The FPSR and FPCR registers are only present in fpsimd_context. For convenience, the content of V0..V31 is duplicated between sve_context and fpsimd_context.
- The record contains a flag field which includes a flag SVE_SIG_FLAG_SM which if set indicates that the thread is in streaming mode and the vector length and register data (if present) describe the streaming SVE data and vector length.
- The signal frame record for SVE always contains basic metadata, in particular the thread's vector length (in sve_context.vl).
- The SVE registers may or may not be included in the record, depending on whether the registers are live for the thread. The registers are present if and only if: sve_context.head.size >= SVE_SIG_CONTEXT_SIZE(sve_vq_from_vl(sve_context.vl)).
- If the registers are present, the remainder of the record has a vl-dependent size and layout. Macros SVE_SIG_* are defined [1] to facilitate access to the members.
- Each scalable register (Zn, Pn, FFR) is stored in an endianness-invariant layout, with bits [(8 * i + 7) : (8 * i)] stored at byte offset i from the start of the register's representation in memory.
- If the SVE context is too big to fit in sigcontext.__reserved[], then extra space is allocated on the stack, an extra_context record is written in __reserved[] referencing this space. sve_context is then written in the extra space. Refer to [1] for further details about this mechanism.

3.18.5 5. Signal return

When returning from a signal handler:

- If there is no sve_context record in the signal frame, or if the record is present but contains no register data as described in the previous section, then the SVE registers/bits become non-live and take unspecified values.
- If sve_context is present in the signal frame and contains full register data, the SVE registers become live and are populated with the specified data. However, for backward compatibility reasons, bits [127:0] of Z0..Z31 are always restored from the corresponding members of fpsimd_context.vregs[] and not from sve_context. The remaining bits are restored from sve_context.
- Inclusion of fpsimd_context in the signal frame remains mandatory, irrespective of whether sve_context is present or not.
- The vector length cannot be changed via signal return. If sve_context.vl in the signal frame does not match the current vector length, the signal return attempt is treated as illegal, resulting in a forced SIGSEGV.
- It is permitted to enter or leave streaming mode by setting or clearing the SVE_SIG_FLAG_SM flag but applications should take care to ensure that when doing so sve_context.vl and any register data are appropriate for the vector length in the new mode.

3.18.6 6. prctl extensions

Some new prctl() calls are added to allow programs to manage the SVE vector length:

prctl(PR_SVE_SET_VL, unsigned long arg)

> Sets the vector length of the calling thread and related flags, where arg == vl | flags. Other threads of the calling process are unaffected.
>
> vl is the desired vector length, where sve_vl_valid(vl) must be true.
>
> flags:
>
> PR_SVE_VL_INHERIT
>
> > Inherit the current vector length across execve(). Otherwise, the vector length is reset to the system default at execve(). (See Section 9.)
>
> PR_SVE_SET_VL_ONEXEC
>
> > Defer the requested vector length change until the next execve() performed by this thread.
> >
> > The effect is equivalent to implicit execution of the following call immediately after the next execve() (if any) by the thread:
> >
> > > prctl(PR_SVE_SET_VL, arg & ~PR_SVE_SET_VL_ONEXEC)
> >
> > This allows launching of a new program with a different vector length, while avoiding runtime side effects in the caller.
> >
> > Without PR_SVE_SET_VL_ONEXEC, the requested change takes effect immediately.
>
> **Return value: a nonnegative on success, or a negative value on error:**
>
> **EINVAL: SVE not supported, invalid vector length requested, or** invalid flags.
>
> On success:
>
> - Either the calling thread's vector length or the deferred vector length to be applied at the next execve() by the thread (dependent on whether PR_SVE_SET_VL_ONEXEC is present in arg), is set to the largest value supported by the system that is less than or equal to vl. If vl == SVE_VL_MAX, the value set will be the largest value supported by the system.
> - Any previously outstanding deferred vector length change in the calling thread is cancelled.
> - The returned value describes the resulting configuration, encoded as for PR_SVE_GET_VL. The vector length reported in this value is the new current vector length for this thread if PR_SVE_SET_VL_ONEXEC was not present in arg; otherwise, the reported vector length is the deferred vector length that will be applied at the next execve() by the calling thread.
> - Changing the vector length causes all of P0..P15, FFR and all bits of Z0..Z31 except for Z0 bits [127:0] .. Z31 bits [127:0] to become unspecified. Calling PR_SVE_SET_VL with vl equal to the thread's current vector length, or calling PR_SVE_SET_VL with the PR_SVE_SET_VL_ONEXEC flag, does not constitute a change to the vector length for this purpose.

prctl(PR_SVE_GET_VL)

Gets the vector length of the calling thread.

The following flag may be OR-ed into the result:

PR_SVE_VL_INHERIT

Vector length will be inherited across execve().

There is no way to determine whether there is an outstanding deferred vector length change (which would only normally be the case between a fork() or vfork() and the corresponding execve() in typical use).

To extract the vector length from the result, bitwise and it with PR_SVE_VL_LEN_MASK.

Return value: a nonnegative value on success, or a negative value on error:
EINVAL: SVE not supported.

3.18.7 7. ptrace extensions

- New regsets NT_ARM_SVE and NT_ARM_SSVE are defined for use with PTRACE_GETREGSET and PTRACE_SETREGSET. NT_ARM_SSVE describes the streaming mode SVE registers and NT_ARM_SVE describes the non-streaming mode SVE registers.

  In this description a register set is referred to as being "live" when the target is in the appropriate streaming or non-streaming mode and is using data beyond the subset shared with the FPSIMD Vn registers.

  Refer to [2] for definitions.

The regset data starts with struct user_sve_header, containing:

size

Size of the complete regset, in bytes. This depends on vl and possibly on other things in the future.

If a call to PTRACE_GETREGSET requests less data than the value of size, the caller can allocate a larger buffer and retry in order to read the complete regset.

max_size

Maximum size in bytes that the regset can grow to for the target thread. The regset won't grow bigger than this even if the target thread changes its vector length etc.

vl

Target thread's current vector length, in bytes.

max_vl

Maximum possible vector length for the target thread.

flags

at most one of

SVE_PT_REGS_FPSIMD

SVE registers are not live (GETREGSET) or are to be made non-live (SETREGSET).

The payload is of type struct user_fpsimd_state, with the same meaning as for NT_PRFPREG, starting at offset SVE_PT_FPSIMD_OFFSET from the start of user_sve_header.

Extra data might be appended in the future: the size of the payload should be obtained using SVE_PT_FPSIMD_SIZE(vq, flags).

vq should be obtained using sve_vq_from_vl(vl).

or

SVE_PT_REGS_SVE

SVE registers are live (GETREGSET) or are to be made live (SETREGSET).

The payload contains the SVE register data, starting at offset SVE_PT_SVE_OFFSET from the start of user_sve_header, and with size SVE_PT_SVE_SIZE(vq, flags);

… OR-ed with zero or more of the following flags, which have the same meaning and behaviour as the corresponding PR_SET_VL_* flags:

SVE_PT_VL_INHERIT

SVE_PT_VL_ONEXEC (SETREGSET only).

If neither FPSIMD nor SVE flags are provided then no register payload is available, this is only possible when SME is implemented.

- The effects of changing the vector length and/or flags are equivalent to those documented for PR_SVE_SET_VL.

  The caller must make a further GETREGSET call if it needs to know what VL is actually set by SETREGSET, unless is it known in advance that the requested VL is supported.

- In the SVE_PT_REGS_SVE case, the size and layout of the payload depends on the header fields. The SVE_PT_SVE_*() macros are provided to facilitate access to the members.
- In either case, for SETREGSET it is permissible to omit the payload, in which case only the vector length and flags are changed (along with any consequences of those changes).
- In systems supporting SME when in streaming mode a GETREGSET for NT_REG_SVE will return only the user_sve_header with no register data, similarly a GETREGSET for NT_REG_SSVE will not return any register data when not in streaming mode.
- A GETREGSET for NT_ARM_SSVE will never return SVE_PT_REGS_FPSIMD.
- For SETREGSET, if an SVE_PT_REGS_SVE payload is present and the requested VL is not supported, the effect will be the same as if the payload were omitted, except that an EIO error is reported. No attempt is made to translate the payload data to the correct layout for the vector length actually set. The thread’s FPSIMD state is preserved, but the remaining bits of the SVE registers become unspecified. It is up to the caller to translate the payload layout for the actual VL and retry.

- Where SME is implemented it is not possible to GETREGSET the register state for normal SVE when in streaming mode, nor the streaming mode register state when in normal mode, regardless of the implementation defined behaviour of the hardware for sharing data between the two modes.
- Any SETREGSET of NT_ARM_SVE will exit streaming mode if the target was in streaming mode and any SETREGSET of NT_ARM_SSVE will enter streaming mode if the target was not in streaming mode.
- The effect of writing a partial, incomplete payload is unspecified.

3.18.8 8. ELF coredump extensions

- NT_ARM_SVE and NT_ARM_SSVE notes will be added to each coredump for each thread of the dumped process. The contents will be equivalent to the data that would have been read if a PTRACE_GETREGSET of the corresponding type were executed for each thread when the coredump was generated.

3.18.9 9. System runtime configuration

- To mitigate the ABI impact of expansion of the signal frame, a policy mechanism is provided for administrators, distro maintainers and developers to set the default vector length for userspace processes:

/proc/sys/abi/sve_default_vector_length

  Writing the text representation of an integer to this file sets the system default vector length to the specified value, unless the value is greater than the maximum vector length supported by the system in which case the default vector length is set to that maximum.

  The result can be determined by reopening the file and reading its contents.

  At boot, the default vector length is initially set to 64 or the maximum supported vector length, whichever is smaller. This determines the initial vector length of the init process (PID 1).

  Reading this file returns the current system default vector length.

- At every execve() call, the new vector length of the new process is set to the system default vector length, unless
  - PR_SVE_VL_INHERIT (or equivalently SVE_PT_VL_INHERIT) is set for the calling thread, or
  - a deferred vector length change is pending, established via the PR_SVE_SET_VL_ONEXEC flag (or SVE_PT_VL_ONEXEC).
- Modifying the system default vector length does not affect the vector length of any existing process or thread that does not make an execve() call.

3.18.10 10. Perf extensions

- The arm64 specific DWARF standard [5] added the VG (Vector Granule) register at index 46. This register is used for DWARF unwinding when variable length SVE registers are pushed onto the stack.
- Its value is equivalent to the current SVE vector length (VL) in bits divided by 64.
- The value is included in Perf samples in the regs[46] field if PERF_SAMPLE_REGS_USER is set and the sample_regs_user mask has bit 46 set.
- The value is the current value at the time the sample was taken, and it can change over time.
- If the system doesn't support SVE when perf_event_open is called with these settings, the event will fail to open.

Appendix A. SVE programmer's model (informative)

This section provides a minimal description of the additions made by SVE to the ARMv8-A programmer's model that are relevant to this document.

Note: This section is for information only and not intended to be complete or to replace any architectural specification.

3.18.11 A.1. Registers

In A64 state, SVE adds the following:

- 32 8VL-bit vector registers Z0..Z31 For each Zn, Zn bits [127:0] alias the ARMv8-A vector register Vn.

  A register write using a Vn register name zeros all bits of the corresponding Zn except for bits [127:0].
- 16 VL-bit predicate registers P0..P15
- 1 VL-bit special-purpose predicate register FFR (the "first-fault register")
- a VL "pseudo-register" that determines the size of each vector register

  The SVE instruction set architecture provides no way to write VL directly. Instead, it can be modified only by EL1 and above, by writing appropriate system registers.
- The value of VL can be configured at runtime by EL1 and above: 16 <= VL <= VLmax, where VL must be a multiple of 16.
- The maximum vector length is determined by the hardware: 16 <= VLmax <= 256.

  (The SVE architecture specifies 256, but permits future architecture revisions to raise this limit.)
- FPSR and FPCR are retained from ARMv8-A, and interact with SVE floating-point operations in a similar way to the way in which they interact with ARMv8 floating-point operations:

```
     8VL-1                       128               0  bit index
    +----          ////            -----------------+
 Z0 |                               :       V0      |
  :                                          :
 Z7 |                               :       V7      |
 Z8 |                               :     * V8      |
  :                                       :  :
Z15 |                               :     *V15      |
Z16 |                               :      V16      |
  :                                          :
Z31 |                               :      V31      |
    +----          ////            -----------------+
                                             31    0
     VL-1                  0                +-------+
    +----       ////      --+          FPSR |       |
 P0 |                       |               +-------+
  : |                       |         *FPCR |       |
P15 |                       |               +-------+
    +----       ////      --+
FFR |                       |               +-----+
    +----       ////      --+            VL |     |
                                            +-----+
```

(*) callee-save:
This only applies to bits [63:0] of Z-/V-registers. FPCR contains callee-save and caller-save bits. See [4] for details.

3.18.12 A.2. Procedure call standard

The ARMv8-A base procedure call standard is extended as follows with respect to the additional SVE register state:

- All SVE register bits that are not shared with FP/SIMD are caller-save.
- Z8 bits [63:0] .. Z15 bits [63:0] are callee-save.

  This follows from the way these bits are mapped to V8..V15, which are caller- save in the base procedure call standard.

Appendix B. ARMv8-A FP/SIMD programmer's model

Note: This section is for information only and not intended to be complete or to replace any architectural specification.

Refer to [4] for more information.

ARMv8-A defines the following floating-point / SIMD register state:

- 32 128-bit vector registers V0..V31
- 2 32-bit status/control registers FPSR, FPCR

```
      127           0  bit index
     +---------------+
  V0 |               |
   : :               :
  V7 |               |
*  V8 |               |
:  : :               :
*V15 |               |
 V16 |               |
   : :               :
 V31 |               |
     +---------------+

             31    0
             +-------+
        FPSR |       |
             +-------+
       *FPCR |       |
             +-------+
```

(*) callee-save:
This only applies to bits [63:0] of V-registers. FPCR contains a mixture of callee-save and caller-save bits.

References

[1] arch/arm64/include/uapi/asm/sigcontext.h
AArch64 Linux signal ABI definitions

[2] arch/arm64/include/uapi/asm/ptrace.h
AArch64 Linux ptrace ABI definitions

[3] *ARM64 CPU Feature Registers*

[4] ARM IHI0055C
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0055c/IHI0055C_beta_aapcs64.pdf
http://infocenter.arm.com/help/topic/com.arm.doc.subset.swdev.abi/index.html Procedure Call Standard for the ARM 64-bit Architecture (AArch64)

[5] https://github.com/ARM-software/abi-aa/blob/main/aadwarf64/aadwarf64.rst

3.19 AArch64 TAGGED ADDRESS ABI

Authors: Vincenzo Frascino <vincenzo.frascino@arm.com>
Catalin Marinas <catalin.marinas@arm.com>

Date: 21 August 2019

This document describes the usage and semantics of the Tagged Address ABI on AArch64 Linux.

3.19.1 1. Introduction

On AArch64 the `TCR_EL1.TBI0` bit is set by default, allowing userspace (EL0) to perform memory accesses through 64-bit pointers with a non-zero top byte. This document describes the relaxation of the syscall ABI that allows userspace to pass certain tagged pointers to kernel syscalls.

3.19.2 2. AArch64 Tagged Address ABI

From the kernel syscall interface perspective and for the purposes of this document, a "valid tagged pointer" is a pointer with a potentially non-zero top-byte that references an address in the user process address space obtained in one of the following ways:

- `mmap()` syscall where either:
  - flags have the `MAP_ANONYMOUS` bit set or
  - the file descriptor refers to a regular file (including those returned by `memfd_create()`) or `/dev/zero`
- `brk()` syscall (i.e. the heap area between the initial location of the program break at process creation and its current location).
- any memory mapped by the kernel in the address space of the process during creation and with the same restrictions as for `mmap()` above (e.g. data, bss, stack).

The AArch64 Tagged Address ABI has two stages of relaxation depending on how the user addresses are used by the kernel:

1. User addresses not accessed by the kernel but used for address space management (e.g. `mprotect()`, `madvise()`). The use of valid tagged pointers in this context is allowed with these exceptions:
   - `brk()`, `mmap()` and the `new_address` argument to `mremap()` as these have the potential to alias with existing user addresses.

     NOTE: This behaviour changed in v5.6 and so some earlier kernels may incorrectly accept valid tagged pointers for the `brk()`, `mmap()` and `mremap()` system calls.
   - The `range.start, start and dst arguments to the UFFDIO_* ioctl()``s used on a file descriptor obtained from ``userfaultfd()`, as fault addresses subsequently obtained by reading the file descriptor will be untagged, which may otherwise confuse tag-unaware programs.

     NOTE: This behaviour changed in v5.14 and so some earlier kernels may incorrectly accept valid tagged pointers for this system call.
2. User addresses accessed by the kernel (e.g. `write()`). This ABI relaxation is disabled by default and the application thread needs to explicitly enable it via `prctl()` as follows:
   - `PR_SET_TAGGED_ADDR_CTRL`: enable or disable the AArch64 Tagged Address ABI for the calling thread.

     The `(unsigned int) arg2` argument is a bit mask describing the control mode used:
     - `PR_TAGGED_ADDR_ENABLE`: enable AArch64 Tagged Address ABI. Default status is disabled.

     Arguments `arg3`, `arg4`, and `arg5` must be 0.

  - `PR_GET_TAGGED_ADDR_CTRL`: get the status of the AArch64 Tagged Address ABI for the calling thread.

    Arguments `arg2`, `arg3`, `arg4`, and `arg5` must be 0.

  The ABI properties described above are thread-scoped, inherited on clone() and fork() and cleared on exec().

  Calling `prctl(PR_SET_TAGGED_ADDR_CTRL, PR_TAGGED_ADDR_ENABLE, 0, 0, 0)` returns `-EINVAL` if the AArch64 Tagged Address ABI is globally disabled by `sysctl abi.tagged_addr_disabled=1`. The default `sysctl abi.tagged_addr_disabled` configuration is 0.

When the AArch64 Tagged Address ABI is enabled for a thread, the following behaviours are guaranteed:

- All syscalls except the cases mentioned in section 3 can accept any valid tagged pointer.
- The syscall behaviour is undefined for invalid tagged pointers: it may result in an error code being returned, a (fatal) signal being raised, or other modes of failure.
- The syscall behaviour for a valid tagged pointer is the same as for the corresponding untagged pointer.

A definition of the meaning of tagged pointers on AArch64 can be found in *Tagged virtual addresses in AArch64 Linux*.

3.19.3 3. AArch64 Tagged Address ABI Exceptions

The following system call parameters must be untagged regardless of the ABI relaxation:

- `prctl()` other than pointers to user data either passed directly or indirectly as arguments to be accessed by the kernel.
- `ioctl()` other than pointers to user data either passed directly or indirectly as arguments to be accessed by the kernel.
- `shmat()` and `shmdt()`.
- `brk()` (since kernel v5.6).
- `mmap()` (since kernel v5.6).
- `mremap()`, the `new_address` argument (since kernel v5.6).

Any attempt to use non-zero tagged pointers may result in an error code being returned, a (fatal) signal being raised, or other modes of failure.

3.19.4 4. Example of correct usage

```
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/prctl.h>

#define PR_SET_TAGGED_ADDR_CTRL      55
```

```
#define PR_TAGGED_ADDR_ENABLE        (1UL << 0)

#define TAG_SHIFT            56

int main(void)
{
     int tbi_enabled = 0;
     unsigned long tag = 0;
     char *ptr;

     /* check/enable the tagged address ABI */
     if (!prctl(PR_SET_TAGGED_ADDR_CTRL, PR_TAGGED_ADDR_ENABLE, 0, 0, 0))
             tbi_enabled = 1;

     /* memory allocation */
     ptr = mmap(NULL, sysconf(_SC_PAGE_SIZE), PROT_READ | PROT_WRITE,
                MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
     if (ptr == MAP_FAILED)
             return 1;

     /* set a non-zero tag if the ABI is available */
     if (tbi_enabled)
             tag = rand() & 0xff;
     ptr = (char *)((unsigned long)ptr | (tag << TAG_SHIFT));

     /* memory access to a tagged address */
     strcpy(ptr, "tagged pointer\n");

     /* syscall with a tagged pointer */
     write(1, ptr, strlen(ptr));

     return 0;
}
```

3.20 Tagged virtual addresses in AArch64 Linux

Author: Will Deacon <will.deacon@arm.com>

Date : 12 June 2013

This document briefly describes the provision of tagged virtual addresses in the AArch64 translation system and their potential uses in AArch64 Linux.

The kernel configures the translation tables so that translations made via TTBR0 (i.e. userspace mappings) have the top byte (bits 63:56) of the virtual address ignored by the translation hardware. This frees up this byte for application use.

3.20.1 Passing tagged addresses to the kernel

All interpretation of userspace memory addresses by the kernel assumes an address tag of 0x00, unless the application enables the AArch64 Tagged Address ABI explicitly (*AArch64 TAGGED ADDRESS ABI*).

This includes, but is not limited to, addresses found in:

- pointer arguments to system calls, including pointers in structures passed to system calls,
- the stack pointer (sp), e.g. when interpreting it to deliver a signal,
- the frame pointer (x29) and frame records, e.g. when interpreting them to generate a backtrace or call graph.

Using non-zero address tags in any of these locations when the userspace application did not enable the AArch64 Tagged Address ABI may result in an error code being returned, a (fatal) signal being raised, or other modes of failure.

For these reasons, when the AArch64 Tagged Address ABI is disabled, passing non-zero address tags to the kernel via system calls is forbidden, and using a non-zero address tag for sp is strongly discouraged.

Programs maintaining a frame pointer and frame records that use non-zero address tags may suffer impaired or inaccurate debug and profiling visibility.

3.20.2 Preserving tags

When delivering signals, non-zero tags are not preserved in siginfo.si_addr unless the flag SA_EXPOSE_TAGBITS was set in sigaction.sa_flags when the signal handler was installed. This means that signal handlers in applications making use of tags cannot rely on the tag information for user virtual addresses being maintained in these fields unless the flag was set.

Due to architecture limitations, bits 63:60 of the fault address are not preserved in response to synchronous tag check faults (SEGV_MTESERR) even if SA_EXPOSE_TAGBITS was set. Applications should treat the values of these bits as undefined in order to accommodate future architecture revisions which may preserve the bits.

For signals raised in response to watchpoint debug exceptions, the tag information will be preserved regardless of the SA_EXPOSE_TAGBITS flag setting.

Non-zero tags are never preserved in sigcontext.fault_address regardless of the SA_EXPOSE_TAGBITS flag setting.

The architecture prevents the use of a tagged PC, so the upper byte will be set to a sign-extension of bit 55 on exception return.

This behaviour is maintained when the AArch64 Tagged Address ABI is enabled.

3.20.3 Other considerations

Special care should be taken when using tagged pointers, since it is likely that C compilers will not hazard two virtual addresses differing only in the upper byte.

3.21 Feature status on arm64 architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	ok
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok
core	thread-info-in-task	THREAD_INFO_IN_TASK	ok
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	ok
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	ok
debug	KASAN	HAVE_ARCH_KASAN	ok
debug	kcov	ARCH_HAS_KCOV	ok
debug	kgdb	HAVE_ARCH_KGDB	ok
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok
debug	kprobes	HAVE_KPROBES	ok
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	TODO
debug	kretprobes	HAVE_KRETPROBES	ok
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	ok
debug	uprobes	ARCH_SUPPORTS_UPROBES	ok
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	ok
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	ok
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	ok
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	ok
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok
perf	perf-regs	HAVE_PERF_REGS	ok
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	ok
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	ok
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	ok
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	ok
time	clockevents	!LEGACY_TIMER_TICK	ok
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	ok
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	ok
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	ok
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	ok
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	ok
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	ok
vm	ioremap_prot	HAVE_IOREMAP_PROT	ok
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO

Table 4 – continued fron

Subsystem	Feature	Kconfig	Status
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	ok

CHAPTER

FOUR

IA-64 ARCHITECTURE

4.1 Linux kernel release for the IA-64 Platform

These are the release notes for Linux since version 2.4 for IA-64 platform. This document provides information specific to IA-64 ONLY, to get additional information about the Linux kernel also read the original Linux README provided with the kernel.

4.1.1 Installing the Kernel

- IA-64 kernel installation is the same as the other platforms, see original README for details.

4.1.2 Software Requirements

Compiling and running this kernel requires an IA-64 compliant GCC compiler. And various software packages also compiled with an IA-64 compliant GCC compiler.

4.1.3 Configuring the Kernel

Configuration is the same, see original README for details.

Compiling the Kernel:

- Compiling this kernel doesn't differ from other platform so read the original README for details BUT make sure you have an IA-64 compliant GCC compiler.

4.1.4 IA-64 Specifics

- General issues:
  - Hardly any performance tuning has been done. Obvious targets include the library routines (IP checksum, etc.). Less obvious targets include making sure we don't flush the TLB needlessly, etc.
  - SMP locks cleanup/optimization
  - IA32 support. Currently experimental. It mostly works.

4.2 Memory Attribute Aliasing on IA-64

Bjorn Helgaas <bjorn.helgaas@hp.com>

May 4, 2006

4.2.1 Memory Attributes

Itanium supports several attributes for virtual memory references. The attribute is part of the virtual translation, i.e., it is contained in the TLB entry. The ones of most interest to the Linux kernel are:

WB	Write-back (cacheable)
UC	Uncacheable
WC	Write-coalescing

System memory typically uses the WB attribute. The UC attribute is used for memory-mapped I/O devices. The WC attribute is uncacheable like UC is, but writes may be delayed and combined to increase performance for things like frame buffers.

The Itanium architecture requires that we avoid accessing the same page with both a cacheable mapping and an uncacheable mapping[1].

The design of the chipset determines which attributes are supported on which regions of the address space. For example, some chipsets support either WB or UC access to main memory, while others support only WB access.

4.2.2 Memory Map

Platform firmware describes the physical memory map and the supported attributes for each region. At boot-time, the kernel uses the EFI GetMemoryMap() interface. ACPI can also describe memory devices and the attributes they support, but Linux/ia64 currently doesn't use this information.

The kernel uses the efi_memmap table returned from GetMemoryMap() to learn the attributes supported by each region of physical address space. Unfortunately, this table does not completely describe the address space because some machines omit some or all of the MMIO regions from the map.

The kernel maintains another table, kern_memmap, which describes the memory Linux is actually using and the attribute for each region. This contains only system memory; it does not contain MMIO space.

The kern_memmap table typically contains only a subset of the system memory described by the efi_memmap. Linux/ia64 can't use all memory in the system because of constraints imposed by the identity mapping scheme.

The efi_memmap table is preserved unmodified because the original boot-time information is required for kexec.

4.2.3 Kernel Identity Mappings

Linux/ia64 identity mappings are done with large pages, currently either 16MB or 64MB, referred to as "granules." Cacheable mappings are speculative[2], so the processor can read any location in the page at any time, independent of the programmer's intentions. This means that to avoid attribute aliasing, Linux can create a cacheable identity mapping only when the entire granule supports cacheable access.

Therefore, kern_memmap contains only full granule-sized regions that can referenced safely by an identity mapping.

Uncacheable mappings are not speculative, so the processor will generate UC accesses only to locations explicitly referenced by software. This allows UC identity mappings to cover granules that are only partially populated, or populated with a combination of UC and WB regions.

4.2.4 User Mappings

User mappings are typically done with 16K or 64K pages. The smaller page size allows more flexibility because only 16K or 64K has to be homogeneous with respect to memory attributes.

4.2.5 Potential Attribute Aliasing Cases

There are several ways the kernel creates new mappings:

mmap of /dev/mem

This uses remap_pfn_range(), which creates user mappings. These mappings may be either WB or UC. If the region being mapped happens to be in kern_memmap, meaning that it may also be mapped by a kernel identity mapping, the user mapping must use the same attribute as the kernel mapping.

If the region is not in kern_memmap, the user mapping should use an attribute reported as being supported in the EFI memory map.

Since the EFI memory map does not describe MMIO on some machines, this should use an uncacheable mapping as a fallback.

mmap of /sys/class/pci_bus/.../legacy_mem

This is very similar to mmap of /dev/mem, except that legacy_mem only allows mmap of the one megabyte "legacy MMIO" area for a specific PCI bus. Typically this is the first megabyte of physical address space, but it may be different on machines with several VGA devices.

"X" uses this to access VGA frame buffers. Using legacy_mem rather than /dev/mem allows multiple instances of X to talk to different VGA cards.

The /dev/mem mmap constraints apply.

mmap of /proc/bus/pci/.../??.?

This is an MMIO mmap of PCI functions, which additionally may or may not be requested as using the WC attribute.

If WC is requested, and the region in kern_memmap is either WC or UC, and the EFI memory map designates the region as WC, then the WC mapping is allowed.

Otherwise, the user mapping must use the same attribute as the kernel mapping.

read/write of /dev/mem

This uses copy_from_user(), which implicitly uses a kernel identity mapping. This is obviously safe for things in kern_memmap.

There may be corner cases of things that are not in kern_memmap, but could be accessed this way. For example, registers in MMIO space are not in kern_memmap, but could be accessed with a UC mapping. This would not cause attribute aliasing. But registers typically can be accessed only with four-byte or eight-byte accesses, and the copy_from_user() path doesn't allow any control over the access size, so this would be dangerous.

ioremap()

This returns a mapping for use inside the kernel.

If the region is in kern_memmap, we should use the attribute specified there.

If the EFI memory map reports that the entire granule supports WB, we should use that (granules that are partially reserved or occupied by firmware do not appear in kern_memmap).

If the granule contains non-WB memory, but we can cover the region safely with kernel page table mappings, we can use ioremap_page_range() as most other architectures do.

Failing all of the above, we have to fall back to a UC mapping.

4.2.6 Past Problem Cases

mmap of various MMIO regions from /dev/mem by "X" on Intel platforms

The EFI memory map may not report these MMIO regions.

These must be allowed so that X will work. This means that when the EFI memory map is incomplete, every /dev/mem mmap must succeed. It may create either WB or UC user mappings, depending on whether the region is in kern_memmap or the EFI memory map.

mmap of 0x0-0x9FFFF /dev/mem by "hwinfo" on HP sx1000 with VGA enabled

The EFI memory map reports the following attributes:

0x00000-0x9FFFF	WB only	
0xA0000-0xBFFFF	UC only	(VGA frame buffer)
0xC0000-0xFFFFF	WB only	

This mmap is done with user pages, not kernel identity mappings, so it is safe to use WB mappings.

The kernel VGA driver may ioremap the VGA frame buffer at 0xA0000, which uses a granule-sized UC mapping. This granule will cover some WB-only memory, but since UC is non-speculative, the processor will never generate an uncacheable reference to the WB-only areas unless the driver explicitly touches them.

mmap of 0x0-0xFFFFF legacy_mem by "X"

If the EFI memory map reports that the entire range supports the same attributes, we can allow the mmap (and we will prefer WB if supported, as is the case with HP sx[12]000 machines with VGA disabled).

If EFI reports the range as partly WB and partly UC (as on sx[12]000 machines with VGA enabled), we must fail the mmap because there's no safe attribute to use.

If EFI reports some of the range but not all (as on Intel firmware that doesn't report the VGA frame buffer at all), we should fail the mmap and force the user to map just the specific region of interest.

mmap of 0xA0000-0xBFFFF legacy_mem by "X" on HP sx1000 with VGA disabled

The EFI memory map reports the following attributes:

```
0x00000-0xFFFFF WB only (no VGA MMIO hole)
```

This is a special case of the previous case, and the mmap should fail for the same reason as above.

read of /sys/devices/.../rom

For VGA devices, this may cause an ioremap() of 0xC0000. This used to be done with a UC mapping, because the VGA frame buffer at 0xA0000 prevents use of a WB granule. The UC mapping causes an MCA on HP sx[12]000 chipsets.

We should use WB page table mappings to avoid covering the VGA frame buffer.

4.2.7 Notes

[1] SDM rev 2.2, vol 2, sec 4.4.1. [2] SDM rev 2.2, vol 2, sec 4.4.6.

4.3 EFI Real Time Clock driver

S. Eranian <eranian@hpl.hp.com>

March 2000

4.3.1 1. Introduction

This document describes the efirtc.c driver has provided for the IA-64 platform.

The purpose of this driver is to supply an API for kernel and user applications to get access to the Time Service offered by EFI version 0.92.

EFI provides 4 calls one can make once the OS is booted: GetTime(), SetTime(), GetWakeupTime(), SetWakeupTime() which are all supported by this driver. We describe those calls as well the design of the driver in the following sections.

4.3.2 2. Design Decisions

The original ideas was to provide a very simple driver to get access to, at first, the time of day service. This is required in order to access, in a portable way, the CMOS clock. A program like /sbin/hwclock uses such a clock to initialize the system view of the time during boot.

Because we wanted to minimize the impact on existing user-level apps using the CMOS clock, we decided to expose an API that was very similar to the one used today with the legacy RTC driver (driver/char/rtc.c). However, because EFI provides a simpler services, not all ioctl() are available. Also new ioctl()s have been introduced for things that EFI provides but not the legacy.

EFI uses a slightly different way of representing the time, noticeably the reference date is different. Year is the using the full 4-digit format. The Epoch is January 1st 1998. For backward compatibility reasons we don't expose this new way of representing time. Instead we use something very similar to the struct tm, i.e. struct rtc_time, as used by hwclock. One of the reasons for doing it this way is to allow for EFI to still evolve without necessarily impacting any of the user applications. The decoupling enables flexibility and permits writing wrapper code is ncase things change.

The driver exposes two interfaces, one via the device file and a set of ioctl()s. The other is read-only via the /proc filesystem.

As of today we don't offer a /proc/sys interface.

To allow for a uniform interface between the legacy RTC and EFI time service, we have created the include/linux/rtc.h header file to contain only the "public" API of the two drivers. The specifics of the legacy RTC are still in include/linux/mc146818rtc.h.

4.3.3 3. Time of day service

The part of the driver gives access to the time of day service of EFI. Two ioctl()s, compatible with the legacy RTC calls:

Read the CMOS clock:

```
ioctl(d, RTC_RD_TIME, &rtc);
```

Write the CMOS clock:

```
ioctl(d, RTC_SET_TIME, &rtc);
```

The rtc is a pointer to a data structure defined in rtc.h which is close to a struct tm:

```
struct rtc_time {
        int tm_sec;
        int tm_min;
        int tm_hour;
        int tm_mday;
        int tm_mon;
        int tm_year;
        int tm_wday;
        int tm_yday;
        int tm_isdst;
};
```

The driver takes care of converting back an forth between the EFI time and this format.

Those two ioctl()s can be exercised with the hwclock command:

For reading:

```
# /sbin/hwclock --show
Mon Mar  6 15:32:32 2000  -0.910248 seconds
```

For setting:

```
# /sbin/hwclock --systohc
```

Root privileges are required to be able to set the time of day.

4.3.4 4. Wakeup Alarm service

EFI provides an API by which one can program when a machine should wakeup, i.e. reboot. This is very different from the alarm provided by the legacy RTC which is some kind of interval timer alarm. For this reason we don't use the same ioctl()s to get access to the service. Instead we have introduced 2 news ioctl()s to the interface of an RTC.

We have added 2 new ioctl()s that are specific to the EFI driver:

Read the current state of the alarm:

```
ioctl(d, RTC_WKALM_RD, &wkt)
```

Set the alarm or change its status:

```
ioctl(d, RTC_WKALM_SET, &wkt)
```

The wkt structure encapsulates a struct rtc_time + 2 extra fields to get status information:

```
struct rtc_wkalrm {

        unsigned char enabled; /* =1 if alarm is enabled */
        unsigned char pending; /* =1 if alarm is pending  */

        struct rtc_time time;
}
```

As of today, none of the existing user-level apps supports this feature. However writing such a program should be hard by simply using those two ioctl().

Root privileges are required to be able to set the alarm.

4.3.5 5. References

Checkout the following Web site for more information on EFI:

http://developer.intel.com/technology/efi/

4.4 IPF Machine Check (MC) error inject tool

IPF Machine Check (MC) error inject tool is used to inject MC errors from Linux. The tool is a test bed for IPF MC work flow including hardware correctable error handling, OS recoverable error handling, MC event logging, etc.

The tool includes two parts: a kernel driver and a user application sample. The driver provides interface to PAL to inject error and query error injection capabilities. The driver code is in arch/ia64/kernel/err_inject.c. The application sample (shown below) provides a combination of various errors and calls the driver's interface (sysfs interface) to inject errors or query error injection capabilities.

The tool can be used to test Intel IPF machine MC handling capabilities. It's especially useful for people who can not access hardware MC injection tool to inject error. It's also very useful to integrate with other software test suits to do stressful testing on IPF.

Below is a sample application as part of the whole tool. The sample can be used as a working test tool. Or it can be expanded to include more features. It also can be a integrated into a library or other user application to have more thorough test.

The sample application takes err.conf as error configuration input. GCC compiles the code. After you install err_inject driver, you can run this sample application to inject errors.

Errata: Itanium 2 Processors Specification Update lists some errata against the pal_mc_error_inject PAL procedure. The following err.conf has been tested on latest Montecito PAL.

err.conf:

```
#This is configuration file for err_inject_tool.
#The format of the each line is:
#cpu, loop, interval, err_type_info, err_struct_info, err_data_buffer
#where
#     cpu: logical cpu number the error will be inject in.
#     loop: times the error will be injected.
#     interval: In second. every so often one error is injected.
#     err_type_info, err_struct_info: PAL parameters.
#
#Note: All values are hex w/o or w/ 0x prefix.


#On cpu2, inject only total 0x10 errors, interval 5 seconds
#corrected, data cache, hier-2, physical addr(assigned by tool code).
#working on Montecito latest PAL.
2, 10, 5, 4101, 95

#On cpu4, inject and consume total 0x10 errors, interval 5 seconds
#corrected, data cache, hier-2, physical addr(assigned by tool code).
#working on Montecito latest PAL.
4, 10, 5, 4109, 95

#On cpu15, inject and consume total 0x10 errors, interval 5 seconds
#recoverable, DTR0, hier-2.
#working on Montecito latest PAL.
0xf, 0x10, 5, 4249, 15
```

The sample application source code:

err_injection_tool.c:

```
/*
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful, but
 * WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE, GOOD TITLE or
 * NON INFRINGEMENT.  See the GNU General Public License for more
 * details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 *
 * Copyright (C) 2006 Intel Co
 *    Fenghua Yu <fenghua.yu@intel.com>
 *
 */
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <sched.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdarg.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/wait.h>
#include <sys/mman.h>
#include <sys/shm.h>

#define MAX_FN_SIZE           256
#define MAX_BUF_SIZE          256
#define DATA_BUF_SIZE                 256
#define NR_CPUS               512
#define MAX_TASK_NUM          2048
#define MIN_INTERVAL          5       // seconds
#define       ERR_DATA_BUFFER_SIZE    3       // Three 8-byte.
#define PARA_FIELD_NUM                5
#define MASK_SIZE             (NR_CPUS/64)
#define PATH_FORMAT "/sys/devices/system/cpu/cpu%d/err_inject/"

int sched_setaffinity(pid_t pid, unsigned int len, unsigned long *mask);

int verbose;
#define vbprintf if (verbose) printf

int log_info(int cpu, const char *fmt, ...)
{
      FILE *log;
      char fn[MAX_FN_SIZE];
      char buf[MAX_BUF_SIZE];
      va_list args;

      sprintf(fn, "%d.log", cpu);
      log=fopen(fn, "a+");
      if (log==NULL) {
              perror("Error open:");
              return -1;
      }

      va_start(args, fmt);
      vprintf(fmt, args);
      memset(buf, 0, MAX_BUF_SIZE);
      vsprintf(buf, fmt, args);
      va_end(args);

      fwrite(buf, sizeof(buf), 1, log);
      fclose(log);

      return 0;
}

typedef unsigned long u64;
typedef unsigned int  u32;

typedef union err_type_info_u {
      struct {
              u64     mode            : 3,    /* 0-2 */
                      err_inj         : 3,    /* 3-5 */
                      err_sev         : 2,    /* 6-7 */
                      err_struct      : 5,    /* 8-12 */
                      struct_hier     : 3,    /* 13-15 */
                      reserved        : 48;   /* 16-63 */
      } err_type_info_u;
      u64     err_type_info;
} err_type_info_t;

typedef union err_struct_info_u {
      struct {
              u64     siv             : 1,    /* 0     */
                      c_t             : 2,    /* 1-2   */
                      cl_p            : 3,    /* 3-5   */
                      cl_id           : 3,    /* 6-8   */
                      cl_dp           : 1,    /* 9     */
                      reserved1       : 22,   /* 10-31 */
                      tiv             : 1,    /* 32    */
                      trigger         : 4,    /* 33-36 */
                      trigger_pl      : 3,    /* 37-39 */
                      reserved2       : 24;   /* 40-63 */
      } err_struct_info_cache;
      struct {
              u64     siv             : 1,    /* 0     */
                      tt              : 2,    /* 1-2   */
                      tc_tr           : 2,    /* 3-4   */
                      tr_slot         : 8,    /* 5-12  */
                      reserved1       : 19,   /* 13-31 */
                      tiv             : 1,    /* 32    */
                      trigger         : 4,    /* 33-36 */
                      trigger_pl      : 3,    /* 37-39 */
                      reserved2       : 24;   /* 40-63 */
      } err_struct_info_tlb;
      struct {
```

```
                u64     siv             : 1,    /* 0     */
                        regfile_id      : 4,    /* 1-4   */
                        reg_num         : 7,    /* 5-11  */
                        reserved1       : 20,   /* 12-31 */
                        tiv             : 1,    /* 32    */
                        trigger         : 4,    /* 33-36 */
                        trigger_pl      : 3,    /* 37-39 */
                        reserved2       : 24;   /* 40-63 */
        } err_struct_info_register;
        struct {
                u64     reserved;
        } err_struct_info_bus_processor_interconnect;
        u64     err_struct_info;
} err_struct_info_t;

typedef union err_data_buffer_u {
        struct {
                u64     trigger_addr;           /* 0-63         */
                u64     inj_addr;               /* 64-127       */
                u64     way             : 5,    /* 128-132      */
                        index           : 20,   /* 133-152      */
                                        : 39;   /* 153-191      */
        } err_data_buffer_cache;
        struct {
                u64     trigger_addr;           /* 0-63         */
                u64     inj_addr;               /* 64-127       */
                u64     way             : 5,    /* 128-132      */
                        index           : 20,   /* 133-152      */
                        reserved        : 39;   /* 153-191      */
        } err_data_buffer_tlb;
        struct {
                u64     trigger_addr;           /* 0-63         */
        } err_data_buffer_register;
        struct {
                u64     reserved;               /* 0-63         */
        } err_data_buffer_bus_processor_interconnect;
        u64 err_data_buffer[ERR_DATA_BUFFER_SIZE];
} err_data_buffer_t;

typedef union capabilities_u {
        struct {
                u64     i               : 1,
                        d               : 1,
                        rv              : 1,
                        tag             : 1,
                        data            : 1,
                        mesi            : 1,
                        dp              : 1,
                        reserved1       : 3,
                        pa              : 1,
                        va              : 1,
                        wi              : 1,
                        reserved2       : 20,
                        trigger         : 1,
                        trigger_pl      : 1,
                        reserved3       : 30;
        } capabilities_cache;
        struct {
                u64     d               : 1,
                        i               : 1,
                        rv              : 1,
                        tc              : 1,
                        tr              : 1,
                        reserved1       : 27,
                        trigger         : 1,
                        trigger_pl      : 1,
                        reserved2       : 30;
        } capabilities_tlb;
        struct {
                u64     gr_b0           : 1,
                        gr_b1           : 1,
                        fr              : 1,
                        br              : 1,
                        pr              : 1,
                        ar              : 1,
                        cr              : 1,
                        rr              : 1,
                        pkr             : 1,
                        dbr             : 1,
                        ibr             : 1,
                        pmc             : 1,
                        pmd             : 1,
                        reserved1       : 3,
                        regnum          : 1,
                        reserved2       : 15,
                        trigger         : 1,
                        trigger_pl      : 1,
                        reserved3       : 30;
        } capabilities_register;
        struct {
                u64     reserved;
        } capabilities_bus_processor_interconnect;
} capabilities_t;

typedef struct resources_s {
        u64     ibr0            : 1,
                ibr2            : 1,
                ibr4            : 1,
                ibr6            : 1,
                dbr0            : 1,
		dbr2		: 1,
		dbr4		: 1,
		dbr6		: 1,
		reserved	: 48;
} resources_t;


long get_page_size(void)
{
	long page_size=sysconf(_SC_PAGESIZE);
	return page_size;
}

#define PAGE_SIZE (get_page_size()==-1?0x4000:get_page_size())
#define SHM_SIZE (2*PAGE_SIZE*NR_CPUS)
#define SHM_VA 0x2000000100000000

int shmid;
void *shmaddr;

int create_shm(void)
{
	key_t key;
	char fn[MAX_FN_SIZE];

	/* cpu0 is always existing */
	sprintf(fn, PATH_FORMAT, 0);
	if ((key = ftok(fn, 's')) == -1) {
		perror("ftok");
		return -1;
	}

	shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT);
	if (shmid == -1) {
		if (errno==EEXIST) {
			shmid = shmget(key, SHM_SIZE, 0);
			if (shmid == -1) {
				perror("shmget");
				return -1;
			}
		}
		else {
			perror("shmget");
			return -1;
		}
	}
	vbprintf("shmid=%d", shmid);

	/* connect to the segment: */
	shmaddr = shmat(shmid, (void *)SHM_VA, 0);
```

```
      if (shmaddr == (void*)-1) {
              perror("shmat");
              return -1;
      }

      memset(shmaddr, 0, SHM_SIZE);
      mlock(shmaddr, SHM_SIZE);

      return 0;
}

int free_shm()
{
      munlock(shmaddr, SHM_SIZE);
        shmdt(shmaddr);
      semctl(shmid, 0, IPC_RMID);

      return 0;
}

#ifdef _SEM_SEMUN_UNDEFINED
union semun
{
      int val;
      struct semid_ds *buf;
      unsigned short int *array;
      struct seminfo *__buf;
};
#endif

u32 mode=1; /* 1: physical mode; 2: virtual mode. */
int one_lock=1;
key_t key[NR_CPUS];
int semid[NR_CPUS];

int create_sem(int cpu)
{
      union semun arg;
      char fn[MAX_FN_SIZE];
      int sid;

      sprintf(fn, PATH_FORMAT, cpu);
      sprintf(fn, "%s/%s", fn, "err_type_info");
      if ((key[cpu] = ftok(fn, 'e')) == -1) {
              perror("ftok");
              return -1;
      }

      if (semid[cpu]!=0)
              return 0;
      /* clear old semaphore */
      if ((sid = semget(key[cpu], 1, 0)) != -1)
              semctl(sid, 0, IPC_RMID);

      /* get one semaphore */
      if ((semid[cpu] = semget(key[cpu], 1, IPC_CREAT | IPC_EXCL)) == -1) {
              perror("semget");
              printf("Please remove semaphore with key=0x%lx, then run the␣
 →tool.\n",
                      (u64)key[cpu]);
              return -1;
      }

      vbprintf("semid[%d]=0x%lx, key[%d]=%lx\n",cpu,(u64)semid[cpu],cpu,
              (u64)key[cpu]);
      /* initialize the semaphore to 1: */
      arg.val = 1;
      if (semctl(semid[cpu], 0, SETVAL, arg) == -1) {
              perror("semctl");
              return -1;
      }

      return 0;
}

static int lock(int cpu)
{
      struct sembuf lock;

      lock.sem_num = cpu;
      lock.sem_op = 1;
      semop(semid[cpu], &lock, 1);

        return 0;
}

static int unlock(int cpu)
{
      struct sembuf unlock;

      unlock.sem_num = cpu;
      unlock.sem_op = -1;
      semop(semid[cpu], &unlock, 1);

        return 0;
}

void free_sem(int cpu)
{
      semctl(semid[cpu], 0, IPC_RMID);
}

int wr_multi(char *fn, unsigned long *data, int size)
{
      int fd;
      char buf[MAX_BUF_SIZE];
      int ret;

      if (size==1)
              sprintf(buf, "%lx", *data);
      else if (size==3)
              sprintf(buf, "%lx,%lx,%lx", data[0], data[1], data[2]);
      else {
              fprintf(stderr,"write to file with wrong size!\n");
              return -1;
      }

      fd=open(fn, O_RDWR);
      if (!fd) {
              perror("Error:");
              return -1;
      }
      ret=write(fd, buf, sizeof(buf));
      close(fd);
      return ret;
}

int wr(char *fn, unsigned long data)
{
      return wr_multi(fn, &data, 1);
}

int rd(char *fn, unsigned long *data)
{
      int fd;
      char buf[MAX_BUF_SIZE];

      fd=open(fn, O_RDONLY);
      if (fd<0) {
              perror("Error:");
              return -1;
      }
      read(fd, buf, MAX_BUF_SIZE);
      *data=strtoul(buf, NULL, 16);
      close(fd);
      return 0;
}

int rd_status(char *path, int *status)
```

```
{
	char fn[MAX_FN_SIZE];
	sprintf(fn, "%s/status", path);
	if (rd(fn, (u64*)status)<0) {
		perror("status reading error.\n");
		return -1;
	}

	return 0;
}

int rd_capabilities(char *path, u64 *capabilities)
{
	char fn[MAX_FN_SIZE];
	sprintf(fn, "%s/capabilities", path);
	if (rd(fn, capabilities)<0) {
		perror("capabilities reading error.\n");
		return -1;
	}

	return 0;
}

int rd_all(char *path)
{
	unsigned long err_type_info, err_struct_info, err_data_buffer;
	int status;
	unsigned long capabilities, resources;
	char fn[MAX_FN_SIZE];

	sprintf(fn, "%s/err_type_info", path);
	if (rd(fn, &err_type_info)<0) {
		perror("err_type_info reading error.\n");
		return -1;
	}
	printf("err_type_info=%lx\n", err_type_info);

	sprintf(fn, "%s/err_struct_info", path);
	if (rd(fn, &err_struct_info)<0) {
		perror("err_struct_info reading error.\n");
		return -1;
	}
	printf("err_struct_info=%lx\n", err_struct_info);

	sprintf(fn, "%s/err_data_buffer", path);
	if (rd(fn, &err_data_buffer)<0) {
		perror("err_data_buffer reading error.\n");
		return -1;
	}
	printf("err_data_buffer=%lx\n", err_data_buffer);
```

```
        sprintf(fn, "%s/status", path);
        if (rd("status", (u64*)&status)<0) {
                perror("status reading error.\n");
                return -1;
        }
        printf("status=%d\n", status);

        sprintf(fn, "%s/capabilities", path);
        if (rd(fn,&capabilities)<0) {
                perror("capabilities reading error.\n");
                return -1;
        }
        printf("capabilities=%lx\n", capabilities);

        sprintf(fn, "%s/resources", path);
        if (rd(fn, &resources)<0) {
                perror("resources reading error.\n");
                return -1;
        }
        printf("resources=%lx\n", resources);

        return 0;
}

int query_capabilities(char *path, err_type_info_t err_type_info,
                      u64 *capabilities)
{
        char fn[MAX_FN_SIZE];
        err_struct_info_t err_struct_info;
        err_data_buffer_t err_data_buffer;

        err_struct_info.err_struct_info=0;
        memset(err_data_buffer.err_data_buffer, -1, ERR_DATA_BUFFER_SIZE*8);

        sprintf(fn, "%s/err_type_info", path);
        wr(fn, err_type_info.err_type_info);
        sprintf(fn, "%s/err_struct_info", path);
        wr(fn, 0x0);
        sprintf(fn, "%s/err_data_buffer", path);
        wr_multi(fn, err_data_buffer.err_data_buffer, ERR_DATA_BUFFER_SIZE);

        // Fire pal_mc_error_inject procedure.
        sprintf(fn, "%s/call_start", path);
        wr(fn, mode);

        if (rd_capabilities(path, capabilities)<0)
                return -1;

        return 0;
```

```
}

int query_all_capabilities()
{
	int status;
	err_type_info_t err_type_info;
	int err_sev, err_struct, struct_hier;
	int cap=0;
	u64 capabilities;
	char path[MAX_FN_SIZE];

	err_type_info.err_type_info=0;			// Initial
	err_type_info.err_type_info_u.mode=0;		// Query mode;
	err_type_info.err_type_info_u.err_inj=0;

	printf("All capabilities implemented in pal_mc_error_inject:\n");
	sprintf(path, PATH_FORMAT ,0);
	for (err_sev=0;err_sev<3;err_sev++)
		for (err_struct=0;err_struct<5;err_struct++)
			for (struct_hier=0;struct_hier<5;struct_hier++)
	{
		status=-1;
		capabilities=0;
		err_type_info.err_type_info_u.err_sev=err_sev;
		err_type_info.err_type_info_u.err_struct=err_struct;
		err_type_info.err_type_info_u.struct_hier=struct_hier;

		if (query_capabilities(path, err_type_info, &capabilities)<0)
			continue;

		if (rd_status(path, &status)<0)
			continue;

		if (status==0) {
			cap=1;
			printf("For err_sev=%d, err_struct=%d, struct_hier=%d: ",
				err_sev, err_struct, struct_hier);
			printf("capabilities 0x%lx\n", capabilities);
		}
	}
	if (!cap) {
		printf("No capabilities supported.\n");
		return 0;
	}

	return 0;
}

int err_inject(int cpu, char *path, err_type_info_t err_type_info,
		err_struct_info_t err_struct_info,
              err_data_buffer_t err_data_buffer)
{
      int status;
      char fn[MAX_FN_SIZE];

      log_info(cpu, "err_type_info=%lx, err_struct_info=%lx, ",
              err_type_info.err_type_info,
              err_struct_info.err_struct_info);
      log_info(cpu,"err_data_buffer=[%lx,%lx,%lx]\n",
              err_data_buffer.err_data_buffer[0],
              err_data_buffer.err_data_buffer[1],
              err_data_buffer.err_data_buffer[2]);
      sprintf(fn, "%s/err_type_info", path);
      wr(fn, err_type_info.err_type_info);
      sprintf(fn, "%s/err_struct_info", path);
      wr(fn, err_struct_info.err_struct_info);
      sprintf(fn, "%s/err_data_buffer", path);
      wr_multi(fn, err_data_buffer.err_data_buffer, ERR_DATA_BUFFER_SIZE);

      // Fire pal_mc_error_inject procedure.
      sprintf(fn, "%s/call_start", path);
      wr(fn,mode);

      if (rd_status(path, &status)<0) {
              vbprintf("fail: read status\n");
              return -100;
      }

      if (status!=0) {
              log_info(cpu, "fail: status=%d\n", status);
              return status;
      }

      return status;
}

static int construct_data_buf(char *path, err_type_info_t err_type_info,
              err_struct_info_t err_struct_info,
              err_data_buffer_t *err_data_buffer,
              void *va1)
{
      char fn[MAX_FN_SIZE];
      u64 virt_addr=0, phys_addr=0;

      vbprintf("va1=%lx\n", (u64)va1);
      memset(&err_data_buffer->err_data_buffer_cache, 0, ERR_DATA_BUFFER_
↪SIZE*8);

      switch (err_type_info.err_type_info_u.err_struct) {
              case 1: // Cache
```

```
                        switch (err_struct_info.err_struct_info_cache.cl_id) {
                                case 1: //Virtual addr
                                        err_data_buffer->err_data_buffer_cache.
↪inj_addr=(u64)va1;
                                        break;
                                case 2: //Phys addr
                                        sprintf(fn, "%s/virtual_to_phys", path);
                                        virt_addr=(u64)va1;
                                        if (wr(fn,virt_addr)<0)
                                                return -1;
                                        rd(fn, &phys_addr);
                                        err_data_buffer->err_data_buffer_cache.
↪inj_addr=phys_addr;
                                        break;
                                default:
                                        printf("Not supported cl_id\n");
                                        break;
                        }
                        break;
                case 2: //  TLB
                        break;
                case 3: //  Register file
                        break;
                case 4: //  Bus/system interconnect
                default:
                        printf("Not supported err_struct\n");
                        break;
        }

        return 0;
}

typedef struct {
        u64 cpu;
        u64 loop;
        u64 interval;
        u64 err_type_info;
        u64 err_struct_info;
        u64 err_data_buffer[ERR_DATA_BUFFER_SIZE];
} parameters_t;

parameters_t line_para;
int para;

static int empty_data_buffer(u64 *err_data_buffer)
{
        int empty=1;
        int i;

        for (i=0;i<ERR_DATA_BUFFER_SIZE; i++)
            if (err_data_buffer[i]!=-1)
                empty=0;

        return empty;
}

int err_inj()
{
        err_type_info_t err_type_info;
        err_struct_info_t err_struct_info;
        err_data_buffer_t err_data_buffer;
        int count;
        FILE *fp;
        unsigned long cpu, loop, interval, err_type_info_conf, err_struct_info_
↪conf;
        u64 err_data_buffer_conf[ERR_DATA_BUFFER_SIZE];
        int num;
        int i;
        char path[MAX_FN_SIZE];
        parameters_t parameters[MAX_TASK_NUM]={};
        pid_t child_pid[MAX_TASK_NUM];
        time_t current_time;
        int status;

        if (!para) {
            fp=fopen("err.conf", "r");
            if (fp==NULL) {
                perror("Error open err.conf");
                return -1;
            }

            num=0;
            while (!feof(fp)) {
                char buf[256];
                memset(buf,0,256);
                fgets(buf, 256, fp);
                count=sscanf(buf, "%lx, %lx, %lx, %lx, %lx, %lx, %lx, %lx\n",
                                &cpu, &loop, &interval,&err_type_info_conf,
                                &err_struct_info_conf,
                                &err_data_buffer_conf[0],
                                &err_data_buffer_conf[1],
                                &err_data_buffer_conf[2]);
                if (count!=PARA_FIELD_NUM+3) {
                        err_data_buffer_conf[0]=-1;
                        err_data_buffer_conf[1]=-1;
                        err_data_buffer_conf[2]=-1;
                        count=sscanf(buf, "%lx, %lx, %lx, %lx, %lx\n",
                                &cpu, &loop, &interval,&err_type_info_conf,
                                &err_struct_info_conf);
                        if (count!=PARA_FIELD_NUM)
```

```
                                continue;
                }

                parameters[num].cpu=cpu;
                parameters[num].loop=loop;
                parameters[num].interval= interval>MIN_INTERVAL
                                          ?interval:MIN_INTERVAL;
                parameters[num].err_type_info=err_type_info_conf;
                parameters[num].err_struct_info=err_struct_info_conf;
                memcpy(parameters[num++].err_data_buffer,
                        err_data_buffer_conf,ERR_DATA_BUFFER_SIZE*8) ;

                if (num>=MAX_TASK_NUM)
                        break;
            }
        }
        else {
                parameters[0].cpu=line_para.cpu;
                parameters[0].loop=line_para.loop;
                parameters[0].interval= line_para.interval>MIN_INTERVAL
                                          ?line_para.interval:MIN_INTERVAL;
                parameters[0].err_type_info=line_para.err_type_info;
                parameters[0].err_struct_info=line_para.err_struct_info;
                memcpy(parameters[0].err_data_buffer,
                        line_para.err_data_buffer,ERR_DATA_BUFFER_SIZE*8) ;

                num=1;
        }

        /* Create semaphore: If one_lock, one semaphore for all processors.
           Otherwise, one semaphore for each processor. */
        if (one_lock) {
                if (create_sem(0)) {
                        printf("Can not create semaphore...exit\n");
                        free_sem(0);
                        return -1;
                }
        }
        else {
                for (i=0;i<num;i++) {
                   if (create_sem(parameters[i].cpu)) {
                        printf("Can not create semaphore for cpu%d...exit\n",i);
                        free_sem(parameters[num].cpu);
                        return -1;
                   }
                }
        }

        /* Create a shm segment which will be used to inject/consume errors on.*/
        if (create_shm()==-1) {
                printf("Error to create shm...exit\n");
                return -1;
        }

        for (i=0;i<num;i++) {
                pid_t pid;

                current_time=time(NULL);
                log_info(parameters[i].cpu, "\nBegine at %s", ctime(&current_
↪time));
                log_info(parameters[i].cpu, "Configurations:\n");
                log_info(parameters[i].cpu,"On cpu%ld: loop=%lx, interval=%lx(s)
↪",
                        parameters[i].cpu,
                        parameters[i].loop,
                        parameters[i].interval);
                log_info(parameters[i].cpu," err_type_info=%lx,err_struct_info=
↪%lx\n",
                        parameters[i].err_type_info,
                        parameters[i].err_struct_info);

                sprintf(path, PATH_FORMAT, (int)parameters[i].cpu);
                err_type_info.err_type_info=parameters[i].err_type_info;
                err_struct_info.err_struct_info=parameters[i].err_struct_info;
                memcpy(err_data_buffer.err_data_buffer,
                        parameters[i].err_data_buffer,
                        ERR_DATA_BUFFER_SIZE*8);

                pid=fork();
                if (pid==0) {
                        unsigned long mask[MASK_SIZE];
                        int j, k;

                        void *va1, *va2;

                        /* Allocate two memory areas va1 and va2 in shm */
                        va1=shmaddr+parameters[i].cpu*PAGE_SIZE;
                        va2=shmaddr+parameters[i].cpu*PAGE_SIZE+PAGE_SIZE;

                        vbprintf("va1=%lx, va2=%lx\n", (u64)va1, (u64)va2);
                        memset(va1, 0x1, PAGE_SIZE);
                        memset(va2, 0x2, PAGE_SIZE);

                        if (empty_data_buffer(err_data_buffer.err_data_buffer))
                                /* If not specified yet, construct data buffer
                                 * with va1
                                 */
                                construct_data_buf(path, err_type_info,
                                        err_struct_info, &err_data_buffer,va1);
                        for (j=0;j<MASK_SIZE;j++)
                                mask[j]=0;

                        cpu=parameters[i].cpu;
                        k = cpu%64;
                        j = cpu/64;
                        mask[j] = 1UL << k;

                        if (sched_setaffinity(0, MASK_SIZE*8, mask)==-1) {
                                perror("Error sched_setaffinity:");
                                return -1;
                        }

                        for (j=0; j<parameters[i].loop; j++) {
                                log_info(parameters[i].cpu,"Injection ");
                                log_info(parameters[i].cpu,"on cpu%ld: #%d/%ld ",

                                        parameters[i].cpu,j+1, parameters[i].
↪loop);

                                /* Hold the lock */
                                if (one_lock)
                                        lock(0);
                                else
                                /* Hold lock on this cpu */
                                        lock(parameters[i].cpu);

                                if ((status=err_inject(parameters[i].cpu,
                                           path, err_type_info,
                                           err_struct_info, err_data_buffer))
                                           ==0) {
                                        /* consume the error for "inject only"*/
                                        memcpy(va2, va1, PAGE_SIZE);
                                        memcpy(va1, va2, PAGE_SIZE);
                                        log_info(parameters[i].cpu,
                                                "successful\n");
                                }
                                else {
                                        log_info(parameters[i].cpu,"fail:");
                                        log_info(parameters[i].cpu,
                                                "status=%d\n", status);
                                        unlock(parameters[i].cpu);
                                        break;
                                }
                                if (one_lock)
                                /* Release the lock */
                                        unlock(0);
                                /* Release lock on this cpu */
                                else
                                        unlock(parameters[i].cpu);
                                if (j < parameters[i].loop-1)
                                        sleep(parameters[i].interval);
                        }
                        current_time=time(NULL);
                        log_info(parameters[i].cpu, "Done at %s", ctime(&current_
↪time));
                        return 0;
                }
                else if (pid<0) {
                        perror("Error fork:");
                        continue;
                }
                child_pid[i]=pid;
        }
        for (i=0;i<num;i++)
                waitpid(child_pid[i], NULL, 0);

        if (one_lock)
                free_sem(0);
        else
                for (i=0;i<num;i++)
                        free_sem(parameters[i].cpu);

        printf("All done.\n");

        return 0;
}

void help()
{
        printf("err_inject_tool:\n");
        printf("\t-q: query all capabilities. default: off\n");
        printf("\t-m: procedure mode. 1: physical 2: virtual. default: 1\n");
        printf("\t-i: inject errors. default: off\n");
        printf("\t-l: one lock per cpu. default: one lock for all\n");
        printf("\t-e: error parameters:\n");
        printf("\t\tcpu,loop,interval,err_type_info,err_struct_info[,err_data_
↪buffer[0],err_data_buffer[1],err_data_buffer[2]]\n");
        printf("\t\t   cpu: logical cpu number the error will be inject in.\n");
        printf("\t\t   loop: times the error will be injected.\n");
        printf("\t\t   interval: In second. every so often one error is injected.
↪\n");
        printf("\t\t   err_type_info, err_struct_info: PAL parameters.\n");
        printf("\t\t   err_data_buffer: PAL parameter. Optional. If not present,\
↪n");
        printf("\t\t                       it's constructed by tool automatically.␣
↪Be\n");
        printf("\t\t                       careful to provide err_data_buffer and␣
↪make\n");
```

```
      printf("\t\t                    sure it's working with the environment.\n
↪");
      printf("\t    Note:no space between error parameters.\n");
      printf("\t    default: Take error parameters from err.conf instead of␣
↪command line.\n");
      printf("\t-v: verbose. default: off\n");
      printf("\t-h: help\n\n");
      printf("The tool will take err.conf file as ");
      printf("input to inject single or multiple errors ");
      printf("on one or multiple cpus in parallel.\n");
}

int main(int argc, char **argv)
{
      char c;
      int do_err_inj=0;
      int do_query_all=0;
      int count;
      u32 m;

      /* Default one lock for all cpu's */
      one_lock=1;
      while ((c = getopt(argc, argv, "m:iqvhle:")) != EOF)
              switch (c) {
                      case 'm':       /* Procedure mode. 1: phys 2: virt */
                              count=sscanf(optarg, "%x", &m);
                              if (count!=1 || (m!=1 && m!=2)) {
                                      printf("Wrong mode number.\n");
                                      help();
                                      return -1;
                              }
                              mode=m;
                              break;
                      case 'i':       /* Inject errors */
                              do_err_inj=1;
                              break;
                      case 'q':       /* Query */
                              do_query_all=1;
                              break;
                      case 'v':       /* Verbose */
                              verbose=1;
                              break;
                      case 'l':       /* One lock per cpu */
                              one_lock=0;
                              break;
                      case 'e':       /* error arguments */
                              /* Take parameters:
                               * #cpu, loop, interval, err_type_info, err_
↪struct_info[, err_data_buffer]
                               * err_data_buffer is optional. Recommend not to␣
 ↪specify
                               * err_data_buffer. Better to use tool to␣
 ↪generate it.
                               */
                              count=sscanf(optarg,
                                      "%lx, %lx, %lx, %lx, %lx, %lx, %lx, %lx\n
 ↪",
                                      &line_para.cpu,
                                      &line_para.loop,
                                      &line_para.interval,
                                      &line_para.err_type_info,
                                      &line_para.err_struct_info,
                                      &line_para.err_data_buffer[0],
                                      &line_para.err_data_buffer[1],
                                      &line_para.err_data_buffer[2]);
                              if (count!=PARA_FIELD_NUM+3) {
                                  line_para.err_data_buffer[0]=-1,
                                  line_para.err_data_buffer[1]=-1,
                                  line_para.err_data_buffer[2]=-1;
                                  count=sscanf(optarg, "%lx, %lx, %lx, %lx,
 ↪%lx\n",
                                              &line_para.cpu,
                                              &line_para.loop,
                                              &line_para.interval,
                                              &line_para.err_type_info,
                                              &line_para.err_struct_info);
                                  if (count!=PARA_FIELD_NUM) {
                                      printf("Wrong error arguments.\n");
                                      help();
                                      return -1;
                                  }
                              }
                              para=1;
                              break;
                      continue;
                              break;
                      case 'h':
                              help();
                              return 0;
                      default:
                              break;
              }

      if (do_query_all)
              query_all_capabilities();
      if (do_err_inj)
              err_inj();

      if (!do_query_all &&  !do_err_inj)
              help();
```

```
        return 0;
}
```

4.5 Light-weight System Calls for IA-64

Started: 13-Jan-2003

Last update: 27-Sep-2003

David Mosberger-Tang <davidm@hpl.hp.com>

Using the "epc" instruction effectively introduces a new mode of execution to the ia64 linux kernel. We call this mode the "fsys-mode". To recap, the normal states of execution are:

- **kernel mode:**
  Both the register stack and the memory stack have been switched over to kernel memory. The user-level state is saved in a pt-regs structure at the top of the kernel memory stack.
- **user mode:**
  Both the register stack and the kernel stack are in user memory. The user-level state is contained in the CPU registers.
- **bank 0 interruption-handling mode:**
  This is the non-interruptible state which all interruption-handlers start execution in. The user-level state remains in the CPU registers and some kernel state may be stored in bank 0 of registers r16-r31.

In contrast, fsys-mode has the following special properties:

- execution is at privilege level 0 (most-privileged)
- CPU registers may contain a mixture of user-level and kernel-level state (it is the responsibility of the kernel to ensure that no security-sensitive kernel-level state is leaked back to user-level)
- execution is interruptible and preemptible (an fsys-mode handler can disable interrupts and avoid all other interruption-sources to avoid preemption)
- neither the memory-stack nor the register-stack can be trusted while in fsys-mode (they point to the user-level stacks, which may be invalid, or completely bogus addresses)

In summary, fsys-mode is much more similar to running in user-mode than it is to running in kernel-mode. Of course, given that the privilege level is at level 0, this means that fsys-mode requires some care (see below).

4.5.1 How to tell fsys-mode

Linux operates in fsys-mode when (a) the privilege level is 0 (most privileged) and (b) the stacks have NOT been switched to kernel memory yet. For convenience, the header file <asm-ia64/ptrace.h> provides three macros:

```
user_mode(regs)
user_stack(task,regs)
fsys_mode(task,regs)
```

The "regs" argument is a pointer to a pt_regs structure. The "task" argument is a pointer to the task structure to which the "regs" pointer belongs to. user_mode() returns TRUE if the CPU state pointed to by "regs" was executing in user mode (privilege level 3). user_stack() returns TRUE if the state pointed to by "regs" was executing on the user-level stack(s). Finally, fsys_mode() returns TRUE if the CPU state pointed to by "regs" was executing in fsys-mode. The fsys_mode() macro is equivalent to the expression:

```
!user_mode(regs) && user_stack(task,regs)
```

4.5.2 How to write an fsyscall handler

The file arch/ia64/kernel/fsys.S contains a table of fsyscall-handlers (fsyscall_table). This table contains one entry for each system call. By default, a system call is handled by fsys_fallback_syscall(). This routine takes care of entering (full) kernel mode and calling the normal Linux system call handler. For performance-critical system calls, it is possible to write a hand-tuned fsyscall_handler. For example, fsys.S contains fsys_getpid(), which is a hand-tuned version of the getpid() system call.

The entry and exit-state of an fsyscall handler is as follows:

Machine state on entry to fsyscall handler

r10	0
r11	saved ar.pfs (a user-level value)
r15	system call number
r16	"current" task pointer (in normal kernel-mode, this is in r13)
r32-r39	system call arguments
b6	return address (a user-level value)
ar.pfs	previous frame-state (a user-level value)
PSR.be	cleared to zero (i.e., little-endian byte order is in effect)
•	all other registers may contain values passed in from user-mode

Required machine state on exit to fsyscall handler

r11	saved ar.pfs (as passed into the fsyscall handler)
r15	system call number (as passed into the fsyscall handler)
r32-r39	system call arguments (as passed into the fsyscall handler)
b6	return address (as passed into the fsyscall handler)
ar.pfs	previous frame-state (as passed into the fsyscall handler)

Fsyscall handlers can execute with very little overhead, but with that speed comes a set of restrictions:

- Fsyscall-handlers MUST check for any pending work in the flags member of the thread-info structure and if any of the TIF_ALLWORK_MASK flags are set, the handler needs to fall back on doing a full system call (by calling fsys_fallback_syscall).
- Fsyscall-handlers MUST preserve incoming arguments (r32-r39, r11, r15, b6, and ar.pfs) because they will be needed in case of a system call restart. Of course, all "preserved" registers also must be preserved, in accordance to the normal calling conventions.
- Fsyscall-handlers MUST check argument registers for containing a NaT value before using them in any way that could trigger a NaT-consumption fault. If a system call argument is found to contain a NaT value, an fsyscall-handler may return immediately with r8=EINVAL, r10=-1.
- Fsyscall-handlers MUST NOT use the "alloc" instruction or perform any other operation that would trigger mandatory RSE (register-stack engine) traffic.
- Fsyscall-handlers MUST NOT write to any stacked registers because it is not safe to assume that user-level called a handler with the proper number of arguments.
- Fsyscall-handlers need to be careful when accessing per-CPU variables: unless proper safe-guards are taken (e.g., interruptions are avoided), execution may be pre-empted and resumed on another CPU at any given time.
- Fsyscall-handlers must be careful not to leak sensitive kernel' information back to user-level. In particular, before returning to user-level, care needs to be taken to clear any scratch registers that could contain sensitive information (note that the current task pointer is not considered sensitive: it's already exposed through ar.k6).
- Fsyscall-handlers MUST NOT access user-memory without first validating access-permission (this can be done typically via probe.r.fault and/or probe.w.fault) and without guarding against memory access exceptions (this can be done with the EX() macros defined by asmmacro.h).

The above restrictions may seem draconian, but remember that it's possible to trade off some of the restrictions by paying a slightly higher overhead. For example, if an fsyscall-handler could benefit from the shadow register bank, it could temporarily disable PSR.i and PSR.ic, switch to bank 0 (bsw.0) and then use the shadow registers as needed. In other words, following the above rules yields extremely fast system call execution (while fully preserving system call semantics), but there is also a lot of flexibility in handling more complicated cases.

4.5.3 Signal handling

The delivery of (asynchronous) signals must be delayed until fsys-mode is exited. This is accomplished with the help of the lower-privilege transfer trap: arch/ia64/kernel/process.c:do_notify_resume_user() checks whether the interrupted task was in fsys-mode and, if so, sets PSR.lp and returns immediately. When fsys-mode is exited via the “br.ret” instruction that lowers the privilege level, a trap will occur. The trap handler clears PSR.lp again and returns immediately. The kernel exit path then checks for and delivers any pending signals.

4.5.4 PSR Handling

The “epc” instruction doesn’t change the contents of PSR at all. This is in contrast to a regular interruption, which clears almost all bits. Because of that, some care needs to be taken to ensure things work as expected. The following discussion describes how each PSR bit is handled.

PSR.be	Cleared when entering fsys-mode. A srlz.d instruction is used to ensure the CPU is in litt
PSR.up	Unchanged.
PSR.ac	Unchanged.
PSR.mfl	Unchanged. Note: fsys-mode handlers must not write-registers!
PSR.mfh	Unchanged. Note: fsys-mode handlers must not write-registers!
PSR.ic	Unchanged. Note: fsys-mode handlers can clear the bit, if needed.
PSR.i	Unchanged. Note: fsys-mode handlers can clear the bit, if needed.
PSR.pk	Unchanged.
PSR.dt	Unchanged.
PSR.dfl	Unchanged. Note: fsys-mode handlers must not write-registers!
PSR.dfh	Unchanged. Note: fsys-mode handlers must not write-registers!
PSR.sp	Unchanged.
PSR.pp	Unchanged.
PSR.di	Unchanged.
PSR.si	Unchanged.
PSR.db	Unchanged. The kernel prevents user-level from setting a hardware breakpoint that trig
PSR.lp	Unchanged.
PSR.tb	Lazy redirect. If a taken-branch trap occurs while in fsys-mode, the trap-handler modifie
PSR.rt	Unchanged.
PSR.cpl	Cleared to 0.
PSR.is	Unchanged (guaranteed to be 0 on entry to the gate page).
PSR.mc	Unchanged.
PSR.it	Unchanged (guaranteed to be 1).
PSR.id	Unchanged. Note: the ia64 linux kernel never sets this bit.
PSR.da	Unchanged. Note: the ia64 linux kernel never sets this bit.
PSR.dd	Unchanged. Note: the ia64 linux kernel never sets this bit.
PSR.ss	Lazy redirect. If set, “epc” will cause a Single Step Trap to be taken. The trap handler th
PSR.ri	Unchanged.
PSR.ed	Unchanged. Note: This bit could only have an effect if an fsys-mode handler performed a
PSR.bn	Unchanged. Note: fsys-mode handlers may clear the bit, if needed. Doing so requires cle
PSR.ia	Unchanged. Note: the ia64 linux kernel never sets this bit.

4.5.5 Using fast system calls

To use fast system calls, userspace applications need simply call __kernel_syscall_via_epc(). For example

-- example fgettimeofday() call --

-- fgettimeofday.S --

```
#include <asm/asmmacro.h>

GLOBAL_ENTRY(fgettimeofday)
.prologue
.save ar.pfs, r11
mov r11 = ar.pfs
.body

mov r2 = 0xa000000000020660;;  // gate address
                             // found by inspection of System.map for the
                             // __kernel_syscall_via_epc() function.  See
                             // below for how to do this for real.

mov b7 = r2
mov r15 = 1087                       // gettimeofday syscall
;;
br.call.sptk.many b6 = b7
;;

.restore sp

mov ar.pfs = r11
br.ret.sptk.many rp;;       // return to caller
END(fgettimeofday)
```

-- end fgettimeofday.S --

In reality, getting the gate address is accomplished by two extra values passed via the ELF auxiliary vector (include/asm-ia64/elf.h)

- AT_SYSINFO : is the address of __kernel_syscall_via_epc()
- AT_SYSINFO_EHDR : is the address of the kernel gate ELF DSO

The ELF DSO is a pre-linked library that is mapped in by the kernel at the gate page. It is a proper ELF shared object so, with a dynamic loader that recognises the library, you should be able to make calls to the exported functions within it as with any other shared library. AT_SYSINFO points into the kernel DSO at the __kernel_syscall_via_epc() function for historical reasons (it was used before the kernel DSO) and as a convenience.

4.6 IRQ affinity on IA64 platforms

07.01.2002, Erich Focht <efocht@ess.nec.de>

By writing to /proc/irq/IRQ#/smp_affinity the interrupt routing can be controlled. The behavior on IA64 platforms is slightly different from that described in Documentation/core-api/irq/irq-affinity.rst for i386 systems.

Because of the usage of SAPIC mode and physical destination mode the IRQ target is one particular CPU and cannot be a mask of several CPUs. Only the first non-zero bit is taken into account.

4.6.1 Usage examples

The target CPU has to be specified as a hexadecimal CPU mask. The first non-zero bit is the selected CPU. This format has been kept for compatibility reasons with i386.

Set the delivery mode of interrupt 41 to fixed and route the interrupts to CPU #3 (logical CPU number) (2^3=0x08):

```
echo "8" >/proc/irq/41/smp_affinity
```

Set the default route for IRQ number 41 to CPU 6 in lowest priority delivery mode (redirectable):

```
echo "r 40" >/proc/irq/41/smp_affinity
```

The output of the command:

```
cat /proc/irq/IRQ#/smp_affinity
```

gives the target CPU mask for the specified interrupt vector. If the CPU mask is preceded by the character "r", the interrupt is redirectable (i.e. lowest priority mode routing is used), otherwise its route is fixed.

4.6.2 Initialization and default behavior

If the platform features IRQ redirection (info provided by SAL) all IO-SAPIC interrupts are initialized with CPU#0 as their default target and the routing is the so called "lowest priority mode" (actually fixed SAPIC mode with hint). The XTP chipset registers are used as hints for the IRQ routing. Currently in Linux XTP registers can have three values:

- minimal for an idle task,
- normal if any other task runs,
- maximal if the CPU is going to be switched off.

The IRQ is routed to the CPU with lowest XTP register value, the search begins at the default CPU. Therefore most of the interrupts will be handled by CPU #0.

If the platform doesn't feature interrupt redirection IOSAPIC fixed routing is used. The target CPUs are distributed in a round robin manner. IRQs will be routed only to the selected target CPUs. Check with:

```
cat /proc/interrupts
```

4.6.3 Comments

On large (multi-node) systems it is recommended to route the IRQs to the node to which the corresponding device is connected. For systems like the NEC AzusA we get IRQ node-affinity for free. This is because usually the chipsets on each node redirect the interrupts only to their own CPUs (as they cannot see the XTP registers on the other nodes).

4.7 An ad-hoc collection of notes on IA64 MCA and INIT processing

Feel free to update it with notes about any area that is not clear.

MCA/INIT are completely asynchronous. They can occur at any time, when the OS is in any state. Including when one of the cpus is already holding a spinlock. Trying to get any lock from MCA/INIT state is asking for deadlock. Also the state of structures that are protected by locks is indeterminate, including linked lists.

The complicated ia64 MCA process. All of this is mandated by Intel's specification for ia64 SAL, error recovery and unwind, it is not as if we have a choice here.

- MCA occurs on one cpu, usually due to a double bit memory error. This is the monarch cpu.
- SAL sends an MCA rendezvous interrupt (which is a normal interrupt) to all the other cpus, the slaves.
- Slave cpus that receive the MCA interrupt call down into SAL, they end up spinning disabled while the MCA is being serviced.
- If any slave cpu was already spinning disabled when the MCA occurred then it cannot service the MCA interrupt. SAL waits ~20 seconds then sends an unmaskable INIT event to the slave cpus that have not already rendezvoused.
- Because MCA/INIT can be delivered at any time, including when the cpu is down in PAL in physical mode, the registers at the time of the event are _completely_ undefined. In particular the MCA/INIT handlers cannot rely on the thread pointer, PAL physical mode can (and does) modify TP. It is allowed to do that as long as it resets TP on return. However MCA/INIT events expose us to these PAL internal TP changes. Hence curr_task().
- If an MCA/INIT event occurs while the kernel was running (not user space) and the kernel has called PAL then the MCA/INIT handler cannot assume that the kernel stack is in a fit state to be used. Mainly because PAL may or may not maintain the stack pointer internally. Because the MCA/INIT handlers cannot trust the kernel stack, they have to use their own, per-cpu stacks. The MCA/INIT stacks are preformatted with just enough task state to let the relevant handlers do their job.

- Unlike most other architectures, the ia64 struct task is embedded in the kernel stack[1]. So switching to a new kernel stack means that we switch to a new task as well. Because various bits of the kernel assume that current points into the struct task, switching to a new stack also means a new value for current.
- Once all slaves have rendezvoused and are spinning disabled, the monarch is entered. The monarch now tries to diagnose the problem and decide if it can recover or not.
- Part of the monarch's job is to look at the state of all the other tasks. The only way to do that on ia64 is to call the unwinder, as mandated by Intel.
- The starting point for the unwind depends on whether a task is running or not. That is, whether it is on a cpu or is blocked. The monarch has to determine whether or not a task is on a cpu before it knows how to start unwinding it. The tasks that received an MCA or INIT event are no longer running, they have been converted to blocked tasks. But (and its a big but), the cpus that received the MCA rendezvous interrupt are still running on their normal kernel stacks!
- To distinguish between these two cases, the monarch must know which tasks are on a cpu and which are not. Hence each slave cpu that switches to an MCA/INIT stack, registers its new stack using set_curr_task(), so the monarch can tell that the _original_ task is no longer running on that cpu. That gives us a decent chance of getting a valid backtrace of the _original_ task.
- MCA/INIT can be nested, to a depth of 2 on any cpu. In the case of a nested error, we want diagnostics on the MCA/INIT handler that failed, not on the task that was originally running. Again this requires set_curr_task() so the MCA/INIT handlers can register their own stack as running on that cpu. Then a recursive error gets a trace of the failing handler's "task".

[1]
My (Keith Owens) original design called for ia64 to separate its struct task and the kernel stacks. Then the MCA/INIT data would be chained stacks like i386 interrupt stacks. But that required radical surgery on the rest of ia64, plus extra hard wired TLB entries with its associated performance degradation. David Mosberger vetoed that approach. Which meant that separate kernel stacks meant separate "tasks" for the MCA/INIT handlers.

INIT is less complicated than MCA. Pressing the nmi button or using the equivalent command on the management console sends INIT to all cpus. SAL picks one of the cpus as the monarch and the rest are slaves. All the OS INIT handlers are entered at approximately the same time. The OS monarch prints the state of all tasks and returns, after which the slaves return and the system resumes.

At least that is what is supposed to happen. Alas there are broken versions of SAL out there. Some drive all the cpus as monarchs. Some drive them all as slaves. Some drive one cpu as monarch, wait for that cpu to return from the OS then drive the rest as slaves. Some versions of SAL cannot even cope with returning from the OS, they spin inside SAL on resume. The OS INIT code has workarounds for some of these broken SAL symptoms, but some simply cannot be fixed from the OS side.

The scheduler hooks used by ia64 (curr_task, set_curr_task) are layer violations. Unfortunately MCA/INIT start off as massive layer violations (can occur at _any_ time) and they build from there.

At least ia64 makes an attempt at recovering from hardware errors, but it is a difficult problem because of the asynchronous nature of these errors. When processing an unmaskable interrupt we sometimes need special code to cope with our inability to take any locks.

How is ia64 MCA/INIT different from x86 NMI?

- x86 NMI typically gets delivered to one cpu. MCA/INIT gets sent to all cpus.
- x86 NMI cannot be nested. MCA/INIT can be nested, to a depth of 2 per cpu.
- x86 has a separate struct task which points to one of multiple kernel stacks. ia64 has the struct task embedded in the single kernel stack, so switching stack means switching task.
- x86 does not call the BIOS so the NMI handler does not have to worry about any registers having changed. MCA/INIT can occur while the cpu is in PAL in physical mode, with undefined registers and an undefined kernel stack.
- i386 backtrace is not very sensitive to whether a process is running or not. ia64 unwind is very, very sensitive to whether a process is running or not.

What happens when MCA/INIT is delivered what a cpu is running user space code?

The user mode registers are stored in the RSE area of the MCA/INIT on entry to the OS and are restored from there on return to SAL, so user mode registers are preserved across a recoverable MCA/INIT. Since the OS has no idea what unwind data is available for the user space stack, MCA/INIT never tries to backtrace user space. Which means that the OS does not bother making the user space process look like a blocked task, i.e. the OS does not copy pt_regs and switch_stack to the user space stack. Also the OS has no idea how big the user space RSE and memory stacks are, which makes it too risky to copy the saved state to a user mode stack.

How do we get a backtrace on the tasks that were running when MCA/INIT was delivered?

mca.c:::ia64_mca_modify_original_stack(). That identifies and verifies the original kernel stack, copies the dirty registers from the MCA/INIT stack's RSE to the original stack's RSE, copies the skeleton struct pt_regs and switch_stack to the original stack, fills in the skeleton structures from the PAL minstate area and updates the original stack's thread.ksp. That makes the original stack look exactly like any other blocked task, i.e. it now appears to be sleeping. To get a backtrace, just start with thread.ksp for the original task and unwind like any other sleeping task.

How do we identify the tasks that were running when MCA/INIT was delivered?

If the previous task has been verified and converted to a blocked state, then sos->prev_task on the MCA/INIT stack is updated to point to the previous task. You can look at that field in dumps or debuggers. To help distinguish between the handler and the original tasks, handlers have _TIF_MCA_INIT set in thread_info.flags.

The sos data is always in the MCA/INIT handler stack, at offset MCA_SOS_OFFSET. You can get that value from mca_asm.h or calculate it as KERNEL_STACK_SIZE - sizeof(struct pt_regs) - sizeof(struct ia64_sal_os_state), with 16 byte alignment for all structures.

Also the comm field of the MCA/INIT task is modified to include the pid of the original task, for humans to use. For example, a comm field of 'MCA 12159' means that pid 12159 was running when the MCA was delivered.

4.8 Serial Devices

4.8.1 Serial Device Naming

As of 2.6.10, serial devices on ia64 are named based on the order of ACPI and PCI enumeration. The first device in the ACPI namespace (if any) becomes /dev/ttyS0, the second becomes /dev/ttyS1, etc., and PCI devices are named sequentially starting after the ACPI devices.

Prior to 2.6.10, there were confusing exceptions to this:

- Firmware on some machines (mostly from HP) provides an HCDP table[1] that tells the kernel about devices that can be used as a serial console. If the user specified "console=ttyS0" or the EFI ConOut path contained only UART devices, the kernel registered the device described by the HCDP as /dev/ttyS0.
- If there was no HCDP, we assumed there were UARTs at the legacy COM port addresses (I/O ports 0x3f8 and 0x2f8), so the kernel registered those as /dev/ttyS0 and /dev/ttyS1.

Any additional ACPI or PCI devices were registered sequentially after /dev/ttyS0 as they were discovered.

With an HCDP, device names changed depending on EFI configuration and "console=" arguments. Without an HCDP, device names didn't change, but we registered devices that might not really exist.

For example, an HP rx1600 with a single built-in serial port (described in the ACPI namespace) plus an MP[2] (a PCI device) has these ports:

Type	MMIO address	pre-2.6.10 (EFI console on builtin)	pre-2.6.10 (EFI console on MP port)	2.6.10+
builtin	0xff5e0000	ttyS0	ttyS1	ttyS0
MP UPS	0xf8031000	ttyS1	ttyS2	ttyS1
MP Console	0xf8030000	ttyS2	ttyS0	ttyS2
MP 2	0xf8030010	ttyS3	ttyS3	ttyS3
MP 3	0xf8030038	ttyS4	ttyS4	ttyS4

4.8.2 Console Selection

EFI knows what your console devices are, but it doesn't tell the kernel quite enough to actually locate them. The DIG64 HCDP table[1] does tell the kernel where potential serial console devices are, but not all firmware supplies it. Also, EFI supports multiple simultaneous consoles and doesn't tell the kernel which should be the "primary" one.

So how do you tell Linux which console device to use?

- If your firmware supplies the HCDP, it is simplest to configure EFI with a single device (either a UART or a VGA card) as the console. Then you don't need to tell Linux anything; the kernel will automatically use the EFI console.

  (This works only in 2.6.6 or later; prior to that you had to specify "console=ttyS0" to get a serial console.)

- Without an HCDP, Linux defaults to a VGA console unless you specify a "console=" argument.

NOTE: Don't assume that a serial console device will be /dev/ttyS0. It might be ttyS1, ttyS2, etc. Make sure you have the appropriate entries in /etc/inittab (for getty) and /etc/securetty (to allow root login).

4.8.3 Early Serial Console

The kernel can't start using a serial console until it knows where the device lives. Normally this happens when the driver enumerates all the serial devices, which can happen a minute or more after the kernel starts booting.

2.6.10 and later kernels have an "early uart" driver that works very early in the boot process. The kernel will automatically use this if the user supplies an argument like "console=uart,io,0x3f8", or if the EFI console path contains only a UART device and the firmware supplies an HCDP.

4.8.4 Troubleshooting Serial Console Problems

No kernel output after elilo prints "Uncompressing Linux… done":

- You specified "console=ttyS0" but Linux changed the device to which ttyS0 refers. Configure exactly one EFI console device[3] and remove the "console=" option.
- The EFI console path contains both a VGA device and a UART. EFI and elilo use both, but Linux defaults to VGA. Remove the VGA device from the EFI console path[3].
- Multiple UARTs selected as EFI console devices. EFI and elilo use all selected devices, but Linux uses only one. Make sure only one UART is selected in the EFI console path[3].
- You're connected to an HP MP port[2] but have a non-MP UART selected as EFI console device. EFI uses the MP as a console device even when it isn't explicitly selected. Either move the console cable to the non-MP UART, or change the EFI console path[3] to the MP UART.

Long pause (60+ seconds) between "Uncompressing Linux... done" and start of kernel output:

- No early console because you used "console=ttyS<n>". Remove the "console=" option if your firmware supplies an HCDP.
- If you don't have an HCDP, the kernel doesn't know where your console lives until the driver discovers serial devices. Use "console=uart,io,0x3f8" (or appropriate address for your machine).

Kernel and init script output works fine, but no "login:" prompt:

- Add getty entry to /etc/inittab for console tty. Look for the "Adding console on ttyS<n>" message that tells you which device is the console.

"login:" prompt, but can't login as root:

- Add entry to /etc/securetty for console tty.

No ACPI serial devices found in 2.6.17 or later:

- Turn on CONFIG_PNP and CONFIG_PNPACPI. Prior to 2.6.17, ACPI serial devices were discovered by 8250_acpi. In 2.6.17, 8250_acpi was replaced by the combination of 8250_pnp and CONFIG_PNPACPI.

[1]

http://www.dig64.org/specifications/agreement The table was originally defined as the "HCDP" for "Headless Console/Debug Port." The current version is the "PCDP" for "Primary Console and Debug Port Devices."

[2]

The HP MP (management processor) is a PCI device that provides several UARTs. One of the UARTs is often used as a console; the EFI Boot Manager identifies it as "Acpi(HWP0002,700)/Pci(...)/Uart". The external connection is usually a 25-pin connector, and a special dongle converts that to three 9-pin connectors, one of which is labelled "Console."

[3]

EFI console devices are configured using the EFI Boot Manager "Boot option maintenance" menu. You may have to interrupt the boot sequence to use this menu, and you will have to reset the box after changing console configuration.

4.9 Feature status on ia64 architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	TODO
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	TODO
core	thread-info-in-task	THREAD_INFO_IN_TASK	TODO
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	TODO
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	TODO

Table 2 – continued fron

Subsystem	Feature	Kconfig	Status
debug	KASAN	HAVE_ARCH_KASAN	TODO
debug	kcov	ARCH_HAS_KCOV	TODO
debug	kgdb	HAVE_ARCH_KGDB	TODO
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	TODO
debug	kprobes	HAVE_KPROBES	ok
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	TODO
debug	kretprobes	HAVE_KRETPROBES	ok
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	TODO
debug	uprobes	ARCH_SUPPORTS_UPROBES	TODO
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	TODO
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO
locking	lockdep	LOCKDEP_SUPPORT	TODO
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	TODO
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	TODO
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	TODO
perf	perf-regs	HAVE_PERF_REGS	TODO
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	TODO
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	TODO
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	TODO
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	TODO
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	TODO
time	clockevents	!LEGACY_TIMER_TICK	TODO
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	---
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	TODO
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	ok
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	TODO
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	TODO
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO
vm	ioremap_prot	HAVE_IOREMAP_PROT	TODO
vm	PG_uncached	ARCH_USES_PG_UNCACHED	ok
vm	pte_special	ARCH_HAS_PTE_SPECIAL	TODO
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	TODO

CHAPTER

FIVE

LOONGARCH ARCHITECTURE

5.1 Introduction to LoongArch

LoongArch is a new RISC ISA, which is a bit like MIPS or RISC-V. There are currently 3 variants: a reduced 32-bit version (LA32R), a standard 32-bit version (LA32S) and a 64-bit version (LA64). There are 4 privilege levels (PLVs) defined in LoongArch: PLV0~PLV3, from high to low. Kernel runs at PLV0 while applications run at PLV3. This document introduces the registers, basic instruction set, virtual memory and some other topics of LoongArch.

5.1.1 Registers

LoongArch registers include general purpose registers (GPRs), floating point registers (FPRs), vector registers (VRs) and control status registers (CSRs) used in privileged mode (PLV0).

GPRs

LoongArch has 32 GPRs (`$r0` ~ `$r31`); each one is 32-bit wide in LA32 and 64-bit wide in LA64. `$r0` is hard-wired to zero, and the other registers are not architecturally special. (Except `$r1`, which is hard-wired as the link register of the BL instruction.)

The kernel uses a variant of the LoongArch register convention, as described in the LoongArch ELF psABI spec, in *References*:

Name	Alias	Usage	Preserved across calls
`$r0`	`$zero`	Constant zero	Unused
`$r1`	`$ra`	Return address	No
`$r2`	`$tp`	TLS/Thread pointer	Unused
`$r3`	`$sp`	Stack pointer	Yes
`$r4-$r11`	`$a0-$a7`	Argument registers	No
`$r4-$r5`	`$v0-$v1`	Return value	No
`$r12-$r20`	`$t0-$t8`	Temp registers	No
`$r21`	`$u0`	Percpu base address	Unused
`$r22`	`$fp`	Frame pointer	Yes
`$r23-$r31`	`$s0-$s8`	Static registers	Yes

Note: The register `$r21` is reserved in the ELF psABI, but used by the Linux kernel for storing the percpu base address. It normally has no ABI name, but is called `$u0` in the kernel. You may also see `$v0` or `$v1` in some old code,however they are deprecated aliases of `$a0` and `$a1` respectively.

FPRs

LoongArch has 32 FPRs (`$f0` ~ `$f31`) when FPU is present. Each one is 64-bit wide on the LA64 cores.

The floating-point register convention is the same as described in the LoongArch ELF psABI spec:

Name	Alias	Usage	Preserved across calls
`$f0-$f7`	`$fa0-$fa7`	Argument registers	No
`$f0-$f1`	`$fv0-$fv1`	Return value	No
`$f8-$f23`	`$ft0-$ft15`	Temp registers	No
`$f24-$f31`	`$fs0-$fs7`	Static registers	Yes

Note: You may see `$fv0` or `$fv1` in some old code, however they are deprecated aliases of `$fa0` and `$fa1` respectively.

VRs

There are currently 2 vector extensions to LoongArch:

- LSX (Loongson SIMD eXtension) with 128-bit vectors,
- LASX (Loongson Advanced SIMD eXtension) with 256-bit vectors.

LSX brings `$v0` ~ `$v31` while LASX brings `$x0` ~ `$x31` as the vector registers.

The VRs overlap with FPRs: for example, on a core implementing LSX and LASX, the lower 128 bits of `$x0` is shared with `$v0`, and the lower 64 bits of `$v0` is shared with `$f0`; same with all other VRs.

CSRs

CSRs can only be accessed from privileged mode (PLV0):

Address	Full Name	Abbrev Name
0x0	Current Mode Information	CRMD
0x1	Pre-exception Mode Information	PRMD
0x2	Extension Unit Enable	EUEN
0x3	Miscellaneous Control	MISC
0x4	Exception Configuration	ECFG
0x5	Exception Status	ESTAT
0x6	Exception Return Address	ERA

continues on next page

Table 1 – continued from previous page

Address	Full Name	Abbrev Name
0x7	Bad (Faulting) Virtual Address	BADV
0x8	Bad (Faulting) Instruction Word	BADI
0xC	Exception Entrypoint Address	EENTRY
0x10	TLB Index	TLBIDX
0x11	TLB Entry High-order Bits	TLBEHI
0x12	TLB Entry Low-order Bits 0	TLBELO0
0x13	TLB Entry Low-order Bits 1	TLBELO1
0x18	Address Space Identifier	ASID
0x19	Page Global Directory Address for Lower-half Address Space	PGDL
0x1A	Page Global Directory Address for Higher-half Address Space	PGDH
0x1B	Page Global Directory Address	PGD
0x1C	Page Walk Control for Lower- half Address Space	PWCL
0x1D	Page Walk Control for Higher- half Address Space	PWCH
0x1E	STLB Page Size	STLBPS
0x1F	Reduced Virtual Address Configuration	RVACFG
0x20	CPU Identifier	CPUID
0x21	Privileged Resource Configuration 1	PRCFG1
0x22	Privileged Resource Configuration 2	PRCFG2
0x23	Privileged Resource Configuration 3	PRCFG3
0x30+n (0≤n≤15)	Saved Data register	SAVEn
0x40	Timer Identifier	TID
0x41	Timer Configuration	TCFG
0x42	Timer Value	TVAL
0x43	Compensation of Timer Count	CNTC
0x44	Timer Interrupt Clearing	TICLR
0x60	LLBit Control	LLBCTL
0x80	Implementation-specific Control 1	IMPCTL1
0x81	Implementation-specific Control 2	IMPCTL2
0x88	TLB Refill Exception Entrypoint Address	TLBRENTRY
0x89	TLB Refill Exception BAD (Faulting) Virtual Address	TLBRBADV
0x8A	TLB Refill Exception Return Address	TLBRERA
0x8B	TLB Refill Exception Saved Data Register	TLBRSAVE
0x8C	TLB Refill Exception Entry Low-order Bits 0	TLBRELO0
0x8D	TLB Refill Exception Entry Low-order Bits 1	TLBRELO1
0x8E	TLB Refill Exception Entry High-order Bits	TLBEHI
0x8F	TLB Refill Exception Pre-exception Mode Information	TLBRPRMD
0x90	Machine Error Control	MERRCTL
0x91	Machine Error Information 1	MERRINFO1
0x92	Machine Error Information 2	MERRINFO2
0x93	Machine Error Exception Entrypoint Address	MERRENTRY
0x94	Machine Error Exception Return Address	MERRERA
0x95	Machine Error Exception Saved Data Register	MERRSAVE
0x98	Cache TAGs	CTAG
0x180+n (0≤n≤3)	Direct Mapping Configuration Window n	DMWn
0x200+2n (0≤n≤31)	Performance Monitor Configuration n	PMCFGn
0x201+2n (0≤n≤31)	Performance Monitor Overall Counter n	PMCNTn
0x300	Memory Load/Store WatchPoint Overall Control	MWPC
0x301	Memory Load/Store WatchPoint Overall Status	MWPS

continues on next page

Table 1 – continued from previous page

Address	Full Name	Abbrev Name
0x310+8n (0≤n≤7)	Memory Load/Store WatchPoint n Configuration 1	MWPnCFG1
0x311+8n (0≤n≤7)	Memory Load/Store WatchPoint n Configuration 2	MWPnCFG2
0x312+8n (0≤n≤7)	Memory Load/Store WatchPoint n Configuration 3	MWPnCFG3
0x313+8n (0≤n≤7)	Memory Load/Store WatchPoint n Configuration 4	MWPnCFG4
0x380	Instruction Fetch WatchPoint Overall Control	FWPC
0x381	Instruction Fetch WatchPoint Overall Status	FWPS
0x390+8n (0≤n≤7)	Instruction Fetch WatchPoint n Configuration 1	FWPnCFG1
0x391+8n (0≤n≤7)	Instruction Fetch WatchPoint n Configuration 2	FWPnCFG2
0x392+8n (0≤n≤7)	Instruction Fetch WatchPoint n Configuration 3	FWPnCFG3
0x393+8n (0≤n≤7)	Instruction Fetch WatchPoint n Configuration 4	FWPnCFG4
0x500	Debug Register	DBG
0x501	Debug Exception Return Address	DERA
0x502	Debug Exception Saved Data Register	DSAVE

ERA, TLBRERA, MERRERA and DERA are sometimes also known as EPC, TLBREPC, MERREPC and DEPC respectively.

5.1.2 Basic Instruction Set

Instruction formats

LoongArch instructions are 32 bits wide, belonging to 9 basic instruction formats (and variants of them):

Format name	Composition
2R	Opcode + Rj + Rd
3R	Opcode + Rk + Rj + Rd
4R	Opcode + Ra + Rk + Rj + Rd
2RI8	Opcode + I8 + Rj + Rd
2RI12	Opcode + I12 + Rj + Rd
2RI14	Opcode + I14 + Rj + Rd
2RI16	Opcode + I16 + Rj + Rd
1RI21	Opcode + I21L + Rj + I21H
I26	Opcode + I26L + I26H

Rd is the destination register operand, while Rj, Rk and Ra ("a" stands for "additional") are the source register operands. I8/I12/I14/I16/I21/I26 are immediate operands of respective width. The longer I21 and I26 are stored in separate higher and lower parts in the instruction word, denoted by the "L" and "H" suffixes.

List of Instructions

For brevity, only instruction names (mnemonics) are listed here; please see the *References* for details.

1. Arithmetic Instructions:

```
ADD.W SUB.W ADDI.W ADD.D SUB.D ADDI.D
SLT SLTU SLTI SLTUI
AND OR NOR XOR ANDN ORN ANDI ORI XORI
MUL.W MULH.W MULH.WU DIV.W DIV.WU MOD.W MOD.WU
MUL.D MULH.D MULH.DU DIV.D DIV.DU MOD.D MOD.DU
PCADDI PCADDU12I PCADDU18I
LU12I.W LU32I.D LU52I.D ADDU16I.D
```

2. Bit-shift Instructions:

```
SLL.W SRL.W SRA.W ROTR.W SLLI.W SRLI.W SRAI.W ROTRI.W
SLL.D SRL.D SRA.D ROTR.D SLLI.D SRLI.D SRAI.D ROTRI.D
```

3. Bit-manipulation Instructions:

```
EXT.W.B EXT.W.H CLO.W CLO.D SLZ.W CLZ.D CTO.W CTO.D CTZ.W CTZ.D
BYTEPICK.W BYTEPICK.D BSTRINS.W BSTRINS.D BSTRPICK.W BSTRPICK.D
REVB.2H REVB.4H REVB.2W REVB.D REVH.2W REVH.D BITREV.4B BITREV.8B BITREV.W␣
↪BITREV.D
MASKEQZ MASKNEZ
```

4. Branch Instructions:

```
BEQ BNE BLT BGE BLTU BGEU BEQZ BNEZ B BL JIRL
```

5. Load/Store Instructions:

```
LD.B LD.BU LD.H LD.HU LD.W LD.WU LD.D ST.B ST.H ST.W ST.D
LDX.B LDX.BU LDX.H LDX.HU LDX.W LDX.WU LDX.D STX.B STX.H STX.W STX.D
LDPTR.W LDPTR.D STPTR.W STPTR.D
PRELD PRELDX
```

6. Atomic Operation Instructions:

```
LL.W SC.W LL.D SC.D
AMSWAP.W AMSWAP.D AMADD.W AMADD.D AMAND.W AMAND.D AMOR.W AMOR.D AMXOR.W␣
↪AMXOR.D
AMMAX.W AMMAX.D AMMIN.W AMMIN.D
```

7. Barrier Instructions:

```
IBAR DBAR
```

8. Special Instructions:

```
SYSCALL BREAK CPUCFG NOP IDLE ERTN(ERET) DBCL(DBGCALL) RDTIMEL.W RDTIMEH.W␣
↪RDTIME.D
ASRTLE.D ASRTGT.D
```

9. Privileged Instructions:

```
CSRRD CSRWR CSRXCHG
IOCSRRD.B IOCSRRD.H IOCSRRD.W IOCSRRD.D IOCSRWR.B IOCSRWR.H IOCSRWR.W␣
↪IOCSRWR.D
CACOP TLBP(TLBSRCH) TLBRD TLBWR TLBFILL TLBCLR TLBFLUSH INVTLB LDDIR LDPTE
```

5.1.3 Virtual Memory

LoongArch supports direct-mapped virtual memory and page-mapped virtual memory.

Direct-mapped virtual memory is configured by CSR.DMWn (n=0~3), it has a simple relationship between virtual address (VA) and physical address (PA):

```
VA = PA + FixedOffset
```

Page-mapped virtual memory has arbitrary relationship between VA and PA, which is recorded in TLB and page tables. LoongArch's TLB includes a fully-associative MTLB (Multiple Page Size TLB) and set-associative STLB (Single Page Size TLB).

By default, the whole virtual address space of LA32 is configured like this:

Name	Address Range	Attributes
UVRANGE	0x00000000 - 0x7FFFFFFF	Page-mapped, Cached, PLV0~3
KPRANGE0	0x80000000 - 0x9FFFFFFF	Direct-mapped, Uncached, PLV0
KPRANGE1	0xA0000000 - 0xBFFFFFFF	Direct-mapped, Cached, PLV0
KVRANGE	0xC0000000 - 0xFFFFFFFF	Page-mapped, Cached, PLV0

User mode (PLV3) can only access UVRANGE. For direct-mapped KPRANGE0 and KPRANGE1, PA is equal to VA with bit30~31 cleared. For example, the uncached direct-mapped VA of 0x00001000 is 0x80001000, and the cached direct-mapped VA of 0x00001000 is 0xA0001000.

By default, the whole virtual address space of LA64 is configured like this:

Name	Address Range	Attributes
XUVRANGE	0x0000000000000000 - 0x3FFFFFFFFFFFFFFF	Page-mapped, Cached, PLV0~3
XSPRANGE	0x4000000000000000 - 0x7FFFFFFFFFFFFFFF	Direct-mapped, Cached / Uncached, PLV0
XKPRANGE	0x8000000000000000 - 0xBFFFFFFFFFFFFFFF	Direct-mapped, Cached / Uncached, PLV0
XKVRANGE	0xC000000000000000 - 0xFFFFFFFFFFFFFFFF	Page-mapped, Cached, PLV0

User mode (PLV3) can only access XUVRANGE. For direct-mapped XSPRANGE and XKPRANGE, PA is equal to VA with bits 60~63 cleared, and the cache attribute is configured by bits 60~61 in VA: 0 is for strongly-ordered uncached, 1 is for coherent cached, and 2 is for weakly-ordered uncached.

Currently we only use XKPRANGE for direct mapping and XSPRANGE is reserved.

To put this in action: the strongly-ordered uncached direct-mapped VA (in XKPRANGE) of 0x00000000_00001000 is 0x80000000_00001000, the coherent cached direct-mapped VA (in XKPRANGE) of 0x00000000_00001000 is 0x90000000_00001000, and the weakly-ordered uncached direct-mapped VA (in XKPRANGE) of 0x00000000 _00001000 is 0xA0000000_00001000.

5.1.4 Relationship of Loongson and LoongArch

LoongArch is a RISC ISA which is different from any other existing ones, while Loongson is a family of processors. Loongson includes 3 series: Loongson-1 is the 32-bit processor series, Loongson-2 is the low-end 64-bit processor series, and Loongson-3 is the high-end 64-bit processor series. Old Loongson is based on MIPS, while New Loongson is based on LoongArch. Take Loongson-3 as an example: Loongson-3A1000/3B1500/3A2000/3A3000/3A4000 are MIPS-compatible, while Loongson- 3A5000 (and future revisions) are all based on LoongArch.

5.1.5 References

Official web site of Loongson Technology Corp. Ltd.:

 http://www.loongson.cn/

Developer web site of Loongson and LoongArch (Software and Documentation):

 http://www.loongnix.cn/

 https://github.com/loongson/

 https://loongson.github.io/LoongArch-Documentation/

Documentation of LoongArch ISA:

 https://github.com/loongson/LoongArch-Documentation/releases/latest/download/LoongArch-Vol1-v1.02-CN.pdf (in Chinese)

 https://github.com/loongson/LoongArch-Documentation/releases/latest/download/LoongArch-Vol1-v1.02-EN.pdf (in English)

Documentation of LoongArch ELF psABI:

 https://github.com/loongson/LoongArch-Documentation/releases/latest/download/LoongArch-ELF-ABI-v2.01-CN.pdf (in Chinese)

 https://github.com/loongson/LoongArch-Documentation/releases/latest/download/LoongArch-ELF-ABI-v2.01-EN.pdf (in English)

Linux kernel repository of Loongson and LoongArch:

 https://git.kernel.org/pub/scm/linux/kernel/git/chenhuacai/linux-loongson.git

5.2 Booting Linux/LoongArch

Author
Yanteng Si <siyanteng@loongson.cn>

Date
18 Nov 2022

5.2.1 Information passed from BootLoader to kernel

LoongArch supports ACPI and FDT. The information that needs to be passed to the kernel includes the memmap, the initrd, the command line, optionally the ACPI/FDT tables, and so on.

The kernel is passed the following arguments on *kernel_entry* :

- a0 = efi_boot: *efi_boot* is a flag indicating whether this boot environment is fully UEFI-compliant.
- a1 = cmdline: *cmdline* is a pointer to the kernel command line.
- a2 = systemtable: *systemtable* points to the EFI system table. All pointers involved at this stage are in physical addresses.

5.2.2 Header of Linux/LoongArch kernel images

Linux/LoongArch kernel images are EFI images. Being PE files, they have a 64-byte header structured like:

```
u32     MZ_MAGIC                /* "MZ", MS-DOS header */
u32     res0 = 0                /* Reserved */
u64     kernel_entry            /* Kernel entry point */
u64     _end - _text            /* Kernel image effective size */
u64     load_offset             /* Kernel image load offset from start of RAM␣
↪*/
u64     res1 = 0                /* Reserved */
u64     res2 = 0                /* Reserved */
u64     res3 = 0                /* Reserved */
u32     LINUX_PE_MAGIC          /* Magic number */
u32     pe_header - _head       /* Offset to the PE header */
```

5.3 IRQ chip model (hierarchy) of LoongArch

Currently, LoongArch based processors (e.g. Loongson-3A5000) can only work together with LS7A chipsets. The irq chips in LoongArch computers include CPUINTC (CPU Core Interrupt Controller), LIOINTC (Legacy I/O Interrupt Controller), EIOINTC (Extended I/O Interrupt Controller), HTVECINTC (Hyper-Transport Vector Interrupt Controller), PCH-PIC (Main Interrupt Controller in LS7A chipset), PCH-LPC (LPC Interrupt Controller in LS7A chipset) and PCH-MSI (MSI Interrupt Controller).

CPUINTC is a per-core controller (in CPU), LIOINTC/EIOINTC/HTVECINTC are per-package controllers (in CPU), while PCH-PIC/PCH-LPC/PCH-MSI are controllers out of CPU (i.e., in chipsets). These controllers (in other words, irqchips) are linked in a hierarchy, and there are two models of hierarchy (legacy model and extended model).

5.3.1 Legacy IRQ model

In this model, IPI (Inter-Processor Interrupt) and CPU Local Timer interrupt go to CPUINTC directly, CPU UARTS interrupts go to LIOINTC, while all other devices interrupts go to PCH-PIC/PCH-LPC/PCH-MSI and gathered by HTVECINTC, and then go to LIOINTC, and then CPUINTC:

```
+-----+     +---------+     +-------+
| IPI | --> | CPUINTC | <-- | Timer |
+-----+     +---------+     +-------+
                 ^
                 |
            +---------+     +-------+
            | LIOINTC | <-- | UARTs |
            +---------+     +-------+
                 ^
                 |
           +-----------+
           | HTVECINTC |
           +-----------+
            ^         ^
            |         |
      +---------+ +---------+
      | PCH-PIC | | PCH-MSI |
      +---------+ +---------+
        ^     ^           ^
        |     |           |
+---------+ +---------+ +---------+
| PCH-LPC | | Devices | | Devices |
+---------+ +---------+ +---------+
     ^
     |
+---------+
| Devices |
+---------+
```

5.3.2 Extended IRQ model

In this model, IPI (Inter-Processor Interrupt) and CPU Local Timer interrupt go to CPUINTC directly, CPU UARTS interrupts go to LIOINTC, while all other devices interrupts go to PCH-PIC/PCH-LPC/PCH-MSI and gathered by EIOINTC, and then go to to CPUINTC directly:

```
      +-----+     +---------+     +-------+
      | IPI | --> | CPUINTC | <-- | Timer |
      +-----+     +---------+     +-------+
                   ^       ^
                   |       |
            +---------+ +---------+     +-------+
            | EIOINTC | | LIOINTC | <-- | UARTs |
            +---------+ +---------+     +-------+
             ^       ^
             |       |
      +---------+ +---------+
      | PCH-PIC | | PCH-MSI |
      +---------+ +---------+
        ^     ^           ^
        |     |           |
+---------+ +---------+ +---------+
| PCH-LPC | | Devices | | Devices |
+---------+ +---------+ +---------+
     ^
     |
+---------+
| Devices |
+---------+
```

5.3.3 ACPI-related definitions

CPUINTC:

```
ACPI_MADT_TYPE_CORE_PIC;
struct acpi_madt_core_pic;
enum acpi_madt_core_pic_version;
```

LIOINTC:

```
ACPI_MADT_TYPE_LIO_PIC;
struct acpi_madt_lio_pic;
enum acpi_madt_lio_pic_version;
```

EIOINTC:

```
ACPI_MADT_TYPE_EIO_PIC;
struct acpi_madt_eio_pic;
enum acpi_madt_eio_pic_version;
```

HTVECINTC:

```
ACPI_MADT_TYPE_HT_PIC;
struct acpi_madt_ht_pic;
enum acpi_madt_ht_pic_version;
```

PCH-PIC:

```
ACPI_MADT_TYPE_BIO_PIC;
struct acpi_madt_bio_pic;
enum acpi_madt_bio_pic_version;
```

PCH-MSI:

```
ACPI_MADT_TYPE_MSI_PIC;
struct acpi_madt_msi_pic;
enum acpi_madt_msi_pic_version;
```

PCH-LPC:

```
ACPI_MADT_TYPE_LPC_PIC;
struct acpi_madt_lpc_pic;
enum acpi_madt_lpc_pic_version;
```

5.3.4 References

Documentation of Loongson-3A5000:

> https://github.com/loongson/LoongArch-Documentation/releases/latest/download/Loongson-3A5000-usermanual-1.02-CN.pdf (in Chinese)
>
> https://github.com/loongson/LoongArch-Documentation/releases/latest/download/Loongson-3A5000-usermanual-1.02-EN.pdf (in English)

Documentation of Loongson's LS7A chipset:

> https://github.com/loongson/LoongArch-Documentation/releases/latest/download/Loongson-7A1000-usermanual-2.00-CN.pdf (in Chinese)
>
> https://github.com/loongson/LoongArch-Documentation/releases/latest/download/Loongson-7A1000-usermanual-2.00-EN.pdf (in English)

Note:

- CPUINTC is CSR.ECFG/CSR.ESTAT and its interrupt controller described in Section 7.4 of "LoongArch Reference Manual, Vol 1";
- LIOINTC is "Legacy I/OInterrupts" described in Section 11.1 of "Loongson 3A5000 Processor Reference Manual";
- EIOINTC is "Extended I/O Interrupts" described in Section 11.2 of "Loongson 3A5000 Processor Reference Manual";
- HTVECINTC is "HyperTransport Interrupts" described in Section 14.3 of "Loongson 3A5000 Processor Reference Manual";
- PCH-PIC/PCH-MSI is "Interrupt Controller" described in Section 5 of "Loongson 7A1000 Bridge User Manual";
- PCH-LPC is "LPC Interrupts" described in Section 24.3 of "Loongson 7A1000 Bridge User Manual".

5.4 Feature status on loongarch architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	ok
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok
core	thread-info-in-task	THREAD_INFO_IN_TASK	TODO
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	TODO
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	TODO
debug	KASAN	HAVE_ARCH_KASAN	ok
debug	kcov	ARCH_HAS_KCOV	ok
debug	kgdb	HAVE_ARCH_KGDB	ok
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok
debug	kprobes	HAVE_KPROBES	ok
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	ok
debug	kretprobes	HAVE_KRETPROBES	ok
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	ok
debug	uprobes	ARCH_SUPPORTS_UPROBES	ok
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	ok
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	ok
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	ok
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok
perf	perf-regs	HAVE_PERF_REGS	ok
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	ok
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	TODO
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	ok
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	ok
time	clockevents	!LEGACY_TIMER_TICK	ok
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	ok
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	ok
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	ok
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	TODO
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	ok
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO
vm	ioremap_prot	HAVE_IOREMAP_PROT	ok
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	ok

CHAPTER

SIX

M68K ARCHITECTURE

6.1 Command Line Options for Linux/m68k

Last Update: 2 May 1999

Linux/m68k version: 2.2.6

Author: Roman.Hodek@informatik.uni-erlangen.de (Roman Hodek)

Update: jds@kom.auc.dk (Jes Sorensen) and faq@linux-m68k.org (Chris Lawrence)

6.1.1 0) Introduction

Often I've been asked which command line options the Linux/m68k kernel understands, or how the exact syntax for the … option is, or … about the option … . I hope, this document supplies all the answers…

Note that some options might be outdated, their descriptions being incomplete or missing. Please update the information and send in the patches.

6.1.2 1) Overview of the Kernel's Option Processing

The kernel knows three kinds of options on its command line:

1) kernel options
2) environment settings
3) arguments for init

To which of these classes an argument belongs is determined as follows: If the option is known to the kernel itself, i.e. if the name (the part before the '=') or, in some cases, the whole argument string is known to the kernel, it belongs to class 1. Otherwise, if the argument contains an '=', it is of class 2, and the definition is put into init's environment. All other arguments are passed to init as command line options.

This document describes the valid kernel options for Linux/m68k in the version mentioned at the start of this file. Later revisions may add new such options, and some may be missing in older versions.

In general, the value (the part after the '=') of an option is a list of values separated by commas. The interpretation of these values is up to the driver that "owns" the option. This association of options with drivers is also the reason that some are further subdivided.

6.1.3 2) General Kernel Options

2.1) root=

Syntax
root=/dev/<device>

or
root=<hex_number>

This tells the kernel which device it should mount as the root filesystem. The device must be a block device with a valid filesystem on it.

The first syntax gives the device by name. These names are converted into a major/minor number internally in the kernel in an unusual way. Normally, this "conversion" is done by the device files in /dev, but this isn't possible here, because the root filesystem (with /dev) isn't mounted yet... So the kernel parses the name itself, with some hardcoded name to number mappings. The name must always be a combination of two or three letters, followed by a decimal number. Valid names are:

```
/dev/ram: -> 0x0100 (initial ramdisk)
/dev/hda: -> 0x0300 (first IDE disk)
/dev/hdb: -> 0x0340 (second IDE disk)
/dev/sda: -> 0x0800 (first SCSI disk)
/dev/sdb: -> 0x0810 (second SCSI disk)
/dev/sdc: -> 0x0820 (third SCSI disk)
/dev/sdd: -> 0x0830 (forth SCSI disk)
/dev/sde: -> 0x0840 (fifth SCSI disk)
/dev/fd : -> 0x0200 (floppy disk)
```

The name must be followed by a decimal number, that stands for the partition number. Internally, the value of the number is just added to the device number mentioned in the table above. The exceptions are /dev/ram and /dev/fd, where /dev/ram refers to an initial ramdisk loaded by your bootstrap program (please consult the instructions for your bootstrap program to find out how to load an initial ramdisk). As of kernel version 2.0.18 you must specify /dev/ram as the root device if you want to boot from an initial ramdisk. For the floppy devices, /dev/fd, the number stands for the floppy drive number (there are no partitions on floppy disks). I.e., /dev/fd0 stands for the first drive, /dev/fd1 for the second, and so on. Since the number is just added, you can also force the disk format by adding a number greater than 3. If you look into your /dev directory, use can see the /dev/fd0D720 has major 2 and minor 16. You can specify this device for the root FS by writing "root=/dev/fd16" on the kernel command line.

[Strange and maybe uninteresting stuff ON]

This unusual translation of device names has some strange consequences: If, for example, you have a symbolic link from /dev/fd to /dev/fd0D720 as an abbreviation for floppy driver #0 in DD format, you cannot use this name for specifying the root device, because the kernel cannot see this symlink before mounting the root FS and it isn't in the table above. If you use it, the root device will not be set at all, without an error message. Another example: You cannot use a partition on e.g. the sixth SCSI disk as the root filesystem, if you want to specify it by name. This is, because only the devices up to /dev/sde are in the table above, but not /dev/sdf. Although, you can use the sixth SCSI disk for the root FS, but you have to specify the device by number... (see below). Or, even more strange, you can use the fact that there is no range checking of the partition number, and your knowledge that each disk uses 16 minors, and write "root=/dev/sde17" (for /dev/sdf1).

[Strange and maybe uninteresting stuff OFF]

If the device containing your root partition isn't in the table above, you can also specify it by major and minor numbers. These are written in hex, with no prefix and no separator between. E.g., if you have a CD with contents appropriate as a root filesystem in the first SCSI CD-ROM drive, you boot from it by "root=0b00". Here, hex "0b" = decimal 11 is the major of SCSI CD-ROMs, and the minor 0 stands for the first of these. You can find out all valid major numbers by looking into include/linux/major.h.

In addition to major and minor numbers, if the device containing your root partition uses a partition table format with unique partition identifiers, then you may use them. For instance, "root=PARTUUID=00112233-4455-6677-8899-AABBCCDDEEFF". It is also possible to reference another partition on the same device using a known partition UUID as the starting point. For example, if partition 5 of the device has the UUID of 00112233-4455-6677-8899-AABBCCDDEEFF then partition 3 may be found as follows:

> PARTUUID=00112233-4455-6677-8899-AABBCCDDEEFF/PARTNROFF=-2

Authoritative information can be found in "Documentation/admin-guide/kernel-parameters.rst".

2.2) ro, rw

Syntax
ro

or
rw

These two options tell the kernel whether it should mount the root filesystem read-only or read-write. The default is read-only, except for ramdisks, which default to read-write.

2.3) debug

Syntax
debug

This raises the kernel log level to 10 (the default is 7). This is the same level as set by the "dmesg" command, just that the maximum level selectable by dmesg is 8.

2.4) debug=

Syntax
debug=<device>

This option causes certain kernel messages be printed to the selected debugging device. This can aid debugging the kernel, since the messages can be captured and analyzed on some other machine. Which devices are possible depends on the machine type. There are no checks for the validity of the device name. If the device isn't implemented, nothing happens.

Messages logged this way are in general stack dumps after kernel memory faults or bad kernel traps, and kernel panics. To be exact: all messages of level 0 (panic messages) and all messages printed while the log level is 8 or more (their level doesn't matter). Before stack dumps, the kernel sets the log level to 10 automatically. A level of at least 8 can also be set by the "debug" command line option (see 2.3) and at run time with "dmesg -n 8".

Devices possible for Amiga:

- **"ser":**
  built-in serial port; parameters: 9600bps, 8N1
- **"mem":**
  Save the messages to a reserved area in chip mem. After rebooting, they can be read under AmigaOS with the tool 'dmesg'.

Devices possible for Atari:

- **"ser1":**
  ST-MFP serial port ("Modem1"); parameters: 9600bps, 8N1
- **"ser2":**
  SCC channel B serial port ("Modem2"); parameters: 9600bps, 8N1
- **"ser" :**
  default serial port This is "ser2" for a Falcon, and "ser1" for any other machine
- **"midi":**
  The MIDI port; parameters: 31250bps, 8N1
- **"par" :**
  parallel port

  The printing routine for this implements a timeout for the case there's no printer connected (else the kernel would lock up). The timeout is not exact, but usually a few seconds.

2.6) ramdisk_size=

Syntax
ramdisk_size=<size>

This option instructs the kernel to set up a ramdisk of the given size in KBytes. Do not use this option if the ramdisk contents are passed by bootstrap! In this case, the size is selected automatically and should not be overwritten.

The only application is for root filesystems on floppy disks, that should be loaded into memory. To do that, select the corresponding size of the disk as ramdisk size, and set the root device to the disk drive (with "root=").

2.7) swap=

I can't find any sign of this option in 2.2.6.

2.8) buff=

I can't find any sign of this option in 2.2.6.

6.1.4 3) General Device Options (Amiga and Atari)

3.1) ether=

Syntax
ether=[<irq>[,<base_addr>[,<mem_start>[,<mem_end>]]]],<dev-name>

<dev-name> is the name of a net driver, as specified in drivers/net/Space.c in the Linux source. Most prominent are eth0, … eth3, sl0, … sl3, ppp0, …, ppp3, dummy, and lo.

The non-ethernet drivers (sl, ppp, dummy, lo) obviously ignore the settings by this options. Also, the existing ethernet drivers for Linux/m68k (ariadne, a2065, hydra) don't use them because Zorro boards are really Plug-'n-Play, so the "ether=" option is useless altogether for Linux/m68k.

3.2) hd=

Syntax
hd=<cylinders>,<heads>,<sectors>

This option sets the disk geometry of an IDE disk. The first hd= option is for the first IDE disk, the second for the second one. (I.e., you can give this option twice.) In most cases, you won't have to use this option, since the kernel can obtain the geometry data itself. It exists just for the case that this fails for one of your disks.

3.3) max_scsi_luns=

Syntax
max_scsi_luns=<n>

Sets the maximum number of LUNs (logical units) of SCSI devices to be scanned. Valid values for <n> are between 1 and 8. Default is 8 if "Probe all LUNs on each SCSI device" was selected during the kernel configuration, else 1.

3.4) st=

Syntax
st=<buffer_size>,[<write_thres>,[<max_buffers>]]

Sets several parameters of the SCSI tape driver. <buffer_size> is the number of 512-byte buffers reserved for tape operations for each device. <write_thres> sets the number of blocks which must be filled to start an actual write operation to the tape. Maximum value is the total number of buffers. <max_buffer> limits the total number of buffers allocated for all tape devices.

3.5) dmasound=

Syntax
dmasound=[<buffers>,<buffer-size>[,<catch-radius>]]

This option controls some configurations of the Linux/m68k DMA sound driver (Amiga and Atari): <buffers> is the number of buffers you want to use (minimum 4, default 4), <buffer-size> is the size of each buffer in kilobytes (minimum 4, default 32) and <catch-radius> says how much percent of error will be tolerated when setting a frequency (maximum 10, default 0). For example with 3% you can play 8000Hz AU-Files on the Falcon with its hardware frequency of 8195Hz and thus don't need to expand the sound.

6.1.5 4) Options for Atari Only

4.1) video=

Syntax
video=<fbname>:<sub-options...>

The <fbname> parameter specifies the name of the frame buffer, eg. most atari users will want to specify *atafb* here. The <sub-options> is a comma-separated list of the sub-options listed below.

NB:
Please notice that this option was renamed from *atavideo* to *video* during the development of the 1.3.x kernels, thus you might need to update your boot-scripts if upgrading to 2.x from an 1.2.x kernel.

NBB:
The behavior of video= was changed in 2.1.57 so the recommended option is to specify the name of the frame buffer.

4.1.1) Video Mode

This sub-option may be any of the predefined video modes, as listed in atari/atafb.c in the Linux/m68k source tree. The kernel will activate the given video mode at boot time and make it the default mode, if the hardware allows. Currently defined names are:

- stlow : 320x200x4
- stmid, default5 : 640x200x2
- sthigh, default4: 640x400x1
- ttlow : 320x480x8, TT only
- ttmid, default1 : 640x480x4, TT only
- tthigh, default2: 1280x960x1, TT only
- vga2 : 640x480x1, Falcon only
- vga4 : 640x480x2, Falcon only
- vga16, default3 : 640x480x4, Falcon only
- vga256 : 640x480x8, Falcon only

- falh2 : 896x608x1, Falcon only
- falh16 : 896x608x4, Falcon only

If no video mode is given on the command line, the kernel tries the modes names "default<n>" in turn, until one is possible with the hardware in use.

A video mode setting doesn't make sense, if the external driver is activated by a "external:" sub-option.

4.1.2) inverse

Invert the display. This affects only text consoles. Usually, the background is chosen to be black. With this option, you can make the background white.

4.1.3) font

Syntax
font:<fontname>

Specify the font to use in text modes. Currently you can choose only between *VGA8x8*, *VGA8x16* and *PEARL8x8*. *VGA8x8* is default, if the vertical size of the display is less than 400 pixel rows. Otherwise, the *VGA8x16* font is the default.

4.1.4) *hwscroll_*

Syntax
hwscroll_<n>

The number of additional lines of video memory to reserve for speeding up the scrolling ("hardware scrolling"). Hardware scrolling is possible only if the kernel can set the video base address in steps fine enough. This is true for STE, MegaSTE, TT, and Falcon. It is not possible with plain STs and graphics cards (The former because the base address must be on a 256 byte boundary there, the latter because the kernel doesn't know how to set the base address at all.)

By default, <n> is set to the number of visible text lines on the display. Thus, the amount of video memory is doubled, compared to no hardware scrolling. You can turn off the hardware scrolling altogether by setting <n> to 0.

4.1.5) internal:

Syntax
internal:<xres>;<yres>[;<xres_max>;<yres_max>;<offset>]

This option specifies the capabilities of some extended internal video hardware, like e.g. OverScan. <xres> and <yres> give the (extended) dimensions of the screen.

If your OverScan needs a black border, you have to write the last three arguments of the "internal:". <xres_max> is the maximum line length the hardware allows, <yres_max> the maximum number of lines. <offset> is the offset of the visible part of the screen memory to its physical start, in bytes.

Often, extended interval video hardware has to be activated somehow. For this, see the "sw_*" options below.

4.1.6) external:

Syntax
external:<xres>;<yres>;<depth>;<org>;<scrmem>[;<scrlen>[;<vgabase>[;<colw>[;<coltype>[;<xres_virtual>]]]]]

This is probably the most complicated parameter... It specifies that you have some external video hardware (a graphics board), and how to use it under Linux/m68k. The kernel cannot know more about the hardware than you tell it here! The kernel also is unable to set or change any video modes, since it doesn't know about any board internal. So, you have to switch to that video mode before you start Linux, and cannot switch to another mode once Linux has started.

The first 3 parameters of this sub-option should be obvious: <xres>, <yres> and <depth> give the dimensions of the screen and the number of planes (depth). The depth is the logarithm to base 2 of the number of colors possible. (Or, the other way round: The number of colors is 2^depth).

You have to tell the kernel furthermore how the video memory is organized. This is done by a letter as <org> parameter:

'n':
"normal planes", i.e. one whole plane after another

'i':
"interleaved planes", i.e. 16 bit of the first plane, than 16 bit of the next, and so on... This mode is used only with the built-in Atari video modes, I think there is no card that supports this mode.

'p':
"packed pixels", i.e. <depth> consecutive bits stand for all planes of one pixel; this is the most common mode for 8 planes (256 colors) on graphic cards

't':
"true color" (more or less packed pixels, but without a color lookup table); usually depth is 24

For monochrome modes (i.e., <depth> is 1), the <org> letter has a different meaning:

'n':
normal colors, i.e. 0=white, 1=black

'i':
inverted colors, i.e. 0=black, 1=white

The next important information about the video hardware is the base address of the video memory. That is given in the <scrmem> parameter, as a hexadecimal number with a "0x" prefix. You have to find out this address in the documentation of your hardware.

The next parameter, <scrlen>, tells the kernel about the size of the video memory. If it's missing, the size is calculated from <xres>, <yres>, and <depth>. For now, it is not useful to write a value here. It would be used only for hardware scrolling (which isn't possible with the external driver, because the kernel cannot set the video base address), or for virtual resolutions under X (which the X server doesn't support yet). So, it's currently best to leave this field empty, either by ending the "external:" after the video address or by writing two consecutive semicolons, if you want to give a <vgabase> (it is allowed to leave this parameter empty).

The <vgabase> parameter is optional. If it is not given, the kernel cannot read or write any color registers of the video hardware, and thus you have to set appropriate colors before you start Linux. But if your card is somehow VGA compatible, you can tell the kernel the base address of the VGA register set, so it can change the color lookup table. You have to look up this address in your board's documentation. To avoid misunderstandings: <vgabase> is the _base_ address, i.e. a 4k aligned address. For read/writing the color registers, the kernel uses the addresses vgabase+0x3c7...vgabase+0x3c9. The <vgabase> parameter is written in hexadecimal with a "0x" prefix, just as <scrmem>.

<colw> is meaningful only if <vgabase> is specified. It tells the kernel how wide each of the color register is, i.e. the number of bits per single color (red/green/blue). Default is 6, another quite usual value is 8.

Also <coltype> is used together with <vgabase>. It tells the kernel about the color register model of your gfx board. Currently, the types "vga" (which is also the default) and "mv300" (SANG MV300) are implemented.

Parameter <xres_virtual> is required for ProMST or ET4000 cards where the physical linelength differs from the visible length. With ProMST, xres_virtual must be set to 2048. For ET4000, xres_virtual depends on the initialisation of the video-card. If you're missing a corresponding yres_virtual: the external part is legacy, therefore we don't support hardware-dependent functions like hardware-scroll, panning or blanking.

4.1.7) eclock:

The external pixel clock attached to the Falcon VIDEL shifter. This currently works only with the ScreenWonder!

4.1.8) monitorcap:

Syntax
monitorcap:<vmin>;<vmax>;<hmin>;<hmax>

This describes the capabilities of a multisync monitor. Don't use it with a fixed-frequency monitor! For now, only the Falcon frame buffer uses the settings of "monitorcap:".

<vmin> and <vmax> are the minimum and maximum, resp., vertical frequencies your monitor can work with, in Hz. <hmin> and <hmax> are the same for the horizontal frequency, in kHz.

The defaults are 58;62;31;32 (VGA compatible).

The defaults for TV/SC1224/SC1435 cover both PAL and NTSC standards.

4.1.9) keep

If this option is given, the framebuffer device doesn't do any video mode calculations and settings on its own. The only Atari fb device that does this currently is the Falcon.

What you reach with this: Settings for unknown video extensions aren't overridden by the driver, so you can still use the mode found when booting, when the driver doesn't know to set this mode itself. But this also means, that you can't switch video modes anymore...

An example where you may want to use "keep" is the ScreenBlaster for the Falcon.

4.2) atamouse=

Syntax
atamouse=<x-threshold>,[<y-threshold>]

With this option, you can set the mouse movement reporting threshold. This is the number of pixels of mouse movement that have to accumulate before the IKBD sends a new mouse packet to the kernel. Higher values reduce the mouse interrupt load and thus reduce the chance of keyboard overruns. Lower values give a slightly faster mouse responses and slightly better mouse tracking.

You can set the threshold in x and y separately, but usually this is of little practical use. If there's just one number in the option, it is used for both dimensions. The default value is 2 for both thresholds.

4.3) ataflop=

Syntax
ataflop=<drive type>[,<trackbuffering>[,<steprateA>[,<steprateB>]]]

The drive type may be 0, 1, or 2, for DD, HD, and ED, resp. This setting affects how many buffers are reserved and which formats are probed (see also below). The default is 1 (HD). Only one drive type can be selected. If you have two disk drives, select the "better" type.

The second parameter <trackbuffer> tells the kernel whether to use track buffering (1) or not (0). The default is machine-dependent: no for the Medusa and yes for all others.

With the two following parameters, you can change the default steprate used for drive A and B, resp.

4.4) atascsi=

Syntax
atascsi=<can_queue>[,<cmd_per_lun>[,<scat-gat>[,<host-id>[,<tagged>]]]]

This option sets some parameters for the Atari native SCSI driver. Generally, any number of arguments can be omitted from the end. And for each of the numbers, a negative value means "use default". The defaults depend on whether TT-style or Falcon-style SCSI is used. Below, defaults are noted as n/m, where the first value refers to TT-SCSI and the latter to Falcon-SCSI.

If an illegal value is given for one parameter, an error message is printed and that one setting is ignored (others aren't affected).

<can_queue>:
This is the maximum number of SCSI commands queued internally to the Atari SCSI driver. A value of 1 effectively turns off the driver internal multitasking (if it causes problems). Legal values are >= 1. <can_queue> can be as high as you like, but values greater than <cmd_per_lun> times the number of SCSI targets (LUNs) you have don't make sense. Default: 16/8.

<cmd_per_lun>:
Maximum number of SCSI commands issued to the driver for one logical unit (LUN, usually one SCSI target). Legal values start from 1. If tagged queuing (see below) is not used, values greater than 2 don't make sense, but waste memory. Otherwise, the maximum is the number of command tags available to the driver (currently 32). Default: 8/1. (Note: Values > 1 seem to cause problems on a Falcon, cause not yet known.)

The <cmd_per_lun> value at a great part determines the amount of memory SCSI reserves for itself. The formula is rather complicated, but I can give you some hints:

no scatter-gather:
cmd_per_lun * 232 bytes

full scatter-gather:
cmd_per_lun * approx. 17 Kbytes

<scat-gat>:
Size of the scatter-gather table, i.e. the number of requests consecutive on the disk that can be merged into one SCSI command. Legal values are between 0 and 255. Default: 255/0. Note: This value is forced to 0 on a Falcon, since scatter-gather isn't possible with the ST-DMA. Not using scatter-gather hurts performance significantly.

<host-id>:
The SCSI ID to be used by the initiator (your Atari). This is usually 7, the highest possible ID. Every ID on the SCSI bus must be unique. Default: determined at run time: If the NV-RAM checksum is valid, and bit 7 in byte 30 of the NV-RAM is set, the lower 3 bits of this byte are used as the host ID. (This method is defined by Atari and also used by some TOS HD drivers.) If the above isn't given, the default ID is 7. (both, TT and Falcon).

<tagged>:
0 means turn off tagged queuing support, all other values > 0 mean use tagged queuing for targets that support it. Default: currently off, but this may change when tagged queuing handling has been proved to be reliable.

Tagged queuing means that more than one command can be issued to one LUN, and the SCSI device itself orders the requests so they can be performed in optimal order. Not all SCSI devices support tagged queuing (:-().

4.5 switches=

Syntax
switches=<list of switches>

With this option you can switch some hardware lines that are often used to enable/disable certain hardware extensions. Examples are OverScan, overclocking, …

The <list of switches> is a comma-separated list of the following items:

ikbd:
set RTS of the keyboard ACIA high

midi:
set RTS of the MIDI ACIA high

snd6:
set bit 6 of the PSG port A

snd7:
set bit 6 of the PSG port A

It doesn't make sense to mention a switch more than once (no difference to only once), but you can give as many switches as you want to enable different features. The switch lines are set as early as possible during kernel initialization (even before determining the present hardware.)

All of the items can also be prefixed with *ov_*, i.e. *ov_ikbd*, *ov_midi*, … These options are meant for switching on an OverScan video extension. The difference to the bare option is that the switch-on is done after video initialization, and somehow synchronized to the HBLANK. A speciality is that ov_ikbd and ov_midi are switched off before rebooting, so that OverScan is disabled and TOS boots correctly.

If you give an option both, with and without the *ov_* prefix, the earlier initialization (*ov_*-less) takes precedence. But the switching-off on reset still happens in this case.

6.1.6 5) Options for Amiga Only:

5.1) video=

Syntax
video=<fbname>:<sub-options…>

The <fbname> parameter specifies the name of the frame buffer, valid options are *amifb*, *cyber*, 'virge', *retz3* and *clgen*, provided that the respective frame buffer devices have been compiled into the kernel (or compiled as loadable modules). The behavior of the <fbname> option was changed in 2.1.57 so it is now recommended to specify this option.

The <sub-options> is a comma-separated list of the sub-options listed below. This option is organized similar to the Atari version of the "video"-option (4.1), but knows fewer sub-options.

5.1.1) video mode

Again, similar to the video mode for the Atari (see 4.1.1). Predefined modes depend on the used frame buffer device.

OCS, ECS and AGA machines all use the color frame buffer. The following predefined video modes are available:

NTSC modes:

- ntsc : 640x200, 15 kHz, 60 Hz
- ntsc-lace : 640x400, 15 kHz, 60 Hz interlaced

PAL modes:

- pal : 640x256, 15 kHz, 50 Hz
- pal-lace : 640x512, 15 kHz, 50 Hz interlaced

ECS modes:

- multiscan : 640x480, 29 kHz, 57 Hz
- multiscan-lace : 640x960, 29 kHz, 57 Hz interlaced
- euro36 : 640x200, 15 kHz, 72 Hz
- euro36-lace : 640x400, 15 kHz, 72 Hz interlaced
- euro72 : 640x400, 29 kHz, 68 Hz
- euro72-lace : 640x800, 29 kHz, 68 Hz interlaced
- super72 : 800x300, 23 kHz, 70 Hz
- super72-lace : 800x600, 23 kHz, 70 Hz interlaced
- dblntsc-ff : 640x400, 27 kHz, 57 Hz
- dblntsc-lace : 640x800, 27 kHz, 57 Hz interlaced
- dblpal-ff : 640x512, 27 kHz, 47 Hz
- dblpal-lace : 640x1024, 27 kHz, 47 Hz interlaced
- dblntsc : 640x200, 27 kHz, 57 Hz doublescan
- dblpal : 640x256, 27 kHz, 47 Hz doublescan

VGA modes:

- vga : 640x480, 31 kHz, 60 Hz
- vga70 : 640x400, 31 kHz, 70 Hz

Please notice that the ECS and VGA modes require either an ECS or AGA chipset, and that these modes are limited to 2-bit color for the ECS chipset and 8-bit color for the AGA chipset.

5.1.2) depth

Syntax
depth:<nr. of bit-planes>

Specify the number of bit-planes for the selected video-mode.

5.1.3) inverse

Use inverted display (black on white). Functionally the same as the "inverse" sub-option for the Atari.

5.1.4) font

Syntax
font:<fontname>

Specify the font to use in text modes. Functionally the same as the "font" sub-option for the Atari, except that *PEARL8x8* is used instead of *VGA8x8* if the vertical size of the display is less than 400 pixel rows.

5.1.5) monitorcap:

Syntax
monitorcap:<vmin>;<vmax>;<hmin>;<hmax>

This describes the capabilities of a multisync monitor. For now, only the color frame buffer uses the settings of "monitorcap:".

<vmin> and <vmax> are the minimum and maximum, resp., vertical frequencies your monitor can work with, in Hz. <hmin> and <hmax> are the same for the horizontal frequency, in kHz.

The defaults are 50;90;15;38 (Generic Amiga multisync monitor).

5.2) fd_def_df0=

Syntax
fd_def_df0=<value>

Sets the df0 value for "silent" floppy drives. The value should be in hexadecimal with "0x" prefix.

5.3) wd33c93=

Syntax
wd33c93=<sub-options...>

These options affect the A590/A2091, A3000 and GVP Series II SCSI controllers.

The <sub-options> is a comma-separated list of the sub-options listed below.

5.3.1) nosync

Syntax
nosync:bitmask

bitmask is a byte where the 1st 7 bits correspond with the 7 possible SCSI devices. Set a bit to prevent sync negotiation on that device. To maintain backwards compatibility, a command-line such as "wd33c93=255" will be automatically translated to "wd33c93=nosync:0xff". The default is to disable sync negotiation for all devices, eg. nosync:0xff.

5.3.2) period

Syntax
period:ns

ns is the minimum # of nanoseconds in a SCSI data transfer period. Default is 500; acceptable values are 250 - 1000.

5.3.3) disconnect

Syntax
disconnect:x

Specify x = 0 to never allow disconnects, 2 to always allow them. x = 1 does 'adaptive' disconnects, which is the default and generally the best choice.

5.3.4) debug

Syntax
debug:x

If *DEBUGGING_ON* is defined, x is a bit mask that causes various types of debug output to printed - see the DB_xxx defines in wd33c93.h.

5.3.5) clock

Syntax
clock:x

x = clock input in MHz for WD33c93 chip. Normal values would be from 8 through 20. The default value depends on your hostadapter(s), default for the A3000 internal controller is 14, for the A2091 it's 8 and for the GVP hostadapters it's either 8 or 14, depending on the hostadapter and the SCSI-clock jumper present on some GVP hostadapters.

5.3.6) next

No argument. Used to separate blocks of keywords when there's more than one wd33c93-based host adapter in the system.

5.3.7) nodma

Syntax
nodma:x

If x is 1 (or if the option is just written as "nodma"), the WD33c93 controller will not use DMA (= direct memory access) to access the Amiga's memory. This is useful for some systems (like A3000's and A4000's with the A3640 accelerator, revision 3.0) that have problems using DMA to chip memory. The default is 0, i.e. to use DMA if possible.

5.4) gvp11=

Syntax
gvp11=<addr-mask>

The earlier versions of the GVP driver did not handle DMA address-mask settings correctly which made it necessary for some people to use this option, in order to get their GVP controller running under Linux. These problems have hopefully been solved and the use of this option is now highly unrecommended!

Incorrect use can lead to unpredictable behavior, so please only use this option if you *know* what you are doing and have a reason to do so. In any case if you experience problems and need to use this option, please inform us about it by mailing to the Linux/68k kernel mailing list.

The address mask set by this option specifies which addresses are valid for DMA with the GVP Series II SCSI controller. An address is valid, if no bits are set except the bits that are set in the mask, too.

Some versions of the GVP can only DMA into a 24 bit address range, some can address a 25 bit address range while others can use the whole 32 bit address range for DMA. The correct setting depends on your controller and should be autodetected by the driver. An example is the 24 bit region which is specified by a mask of 0x00fffffe.

6.2 Amiga Buddha and Catweasel IDE Driver

The Amiga Buddha and Catweasel IDE Driver (part of ide.c) was written by Geert Uytterhoeven based on the following specifications:

Register map of the Buddha IDE controller and the Buddha-part of the Catweasel Zorro-II version

The Autoconfiguration has been implemented just as Commodore described in their manuals, no tricks have been used (for example leaving some address lines out of the equations...). If you want to configure the board yourself (for example let a Linux kernel configure the card), look at the Commodore Docs. Reading the nibbles should give this information:

```
Vendor number: 4626 ($1212)
product number: 0 (42 for Catweasel Z-II)
Serial number: 0
Rom-vector: $1000
```

The card should be a Z-II board, size 64K, not for freemem list, Rom-Vektor is valid, no second Autoconfig-board on the same card, no space preference, supports "Shutup_forever".

Setting the base address should be done in two steps, just as the Amiga Kickstart does: The lower nibble of the 8-Bit address is written to $4a, then the whole Byte is written to $48, while it doesn't matter how often you're writing to $4a as long as $48 is not touched. After $48 has been written, the whole card disappears from $e8 and is mapped to the new address just written. Make sure $4a is written before $48, otherwise your chance is only 1:16 to find the board :-).

The local memory-map is even active when mapped to $e8:

$0-$7e	Autokonfig-space, see Z-II docs.
$80-$7fd	reserved
$7fe	Speed-select Register: Read & Write (description see further down)
$800-$8ff	IDE-Select 0 (Port 0, Register set 0)
$900-$9ff	IDE-Select 1 (Port 0, Register set 1)
$a00-$aff	IDE-Select 2 (Port 1, Register set 0)
$b00-$bff	IDE-Select 3 (Port 1, Register set 1)
$c00-$cff	IDE-Select 4 (Port 2, Register set 0, Catweasel only!)
$d00-$dff	IDE-Select 5 (Port 3, Register set 1, Catweasel only!)
$e00-$eff	local expansion port, on Catweasel Z-II the Catweasel registers are also mapped here. Never touch, use multidisk.device!
$f00	read only, Byte-access: Bit 7 shows the level of the IRQ-line of IDE port 0.
$f01-$f3f	mirror of $f00
$f40	read only, Byte-access: Bit 7 shows the level of the IRQ-line of IDE port 1.
$f41-$f7f	mirror of $f40
$f80	read only, Byte-access: Bit 7 shows the level of the IRQ-line of IDE port 2. (Catweasel only!)
$f81-$fbf	mirror of $f80
$fc0	write-only: Writing any value to this register enables IRQs to be passed from the IDE ports to the Zorro bus. This mechanism has been implemented to be compatible with harddisks that are either defective or have a buggy firmware and pull the IRQ line up while starting up. If interrupts would always be passed to the bus, the computer might not start up. Once enabled, this flag can not be disabled again. The level of the flag can not be determined by software (what for? Write to me if it's necessary!).
$fc1-$fff	mirror of $fc0
$1000-$ffff	Buddha-Rom with offset $1000 in the rom chip. The addresses $0 to $fff of the rom chip cannot be read. Rom is Byte-wide and mapped to even addresses.

The IDE ports issue an INT2. You can read the level of the IRQ-lines of the IDE-ports by reading from the three (two for Buddha-only) registers $f00, $f40 and $f80. This way more than one I/O request can be handled and you can easily determine what driver has to serve the INT2. Buddha and Catweasel expansion boards can issue an INT6. A separate memory map is available for the I/O module and the sysop's I/O module.

The IDE ports are fed by the address lines A2 to A4, just as the Amiga 1200 and Amiga 4000 IDE ports are. This way existing drivers can be easily ported to Buddha. A move.l polls two words out of the same address of IDE port since every word is mirrored once. movem is not possible, but it's not necessary either, because you can only speedup 68000 systems with this technique. A 68020 system with fastmem is faster with move.l.

If you're using the mirrored registers of the IDE-ports with A6=1, the Buddha doesn't care about the speed that you have selected in the speed register (see further down). With A6=1 (for example $840 for port 0, register set 0), a 780ns access is being made. These registers should be used for a command access to the harddisk/CD-Rom, since command accesses are Byte-wide and have to be made slower according to the ATA-X3T9 manual.

Now for the speed-register: The register is byte-wide, and only the upper three bits are used (Bits 7 to 5). Bit 4 must always be set to 1 to be compatible with later Buddha versions (if I'll ever update this one). I presume that I'll never use the lower four bits, but they have to be set to 1 by definition.

The values in this table have to be shifted 5 bits to the left and or'd with $1f (this sets the lower 5 bits).

All the timings have in common: Select and IOR/IOW rise at the same time. IOR and IOW have a propagation delay of about 30ns to the clocks on the Zorro bus, that's why the values are no multiple of 71. One clock-cycle is 71ns long (exactly 70,5 at 14,18 Mhz on PAL systems).

value 0 (Default after reset)
497ns Select (7 clock cycles) , IOR/IOW after 172ns (2 clock cycles) (same timing as the Amiga 1200 does on it's IDE port without accelerator card)

value 1
639ns Select (9 clock cycles), IOR/IOW after 243ns (3 clock cycles)

value 2
781ns Select (11 clock cycles), IOR/IOW after 314ns (4 clock cycles)

value 3
355ns Select (5 clock cycles), IOR/IOW after 101ns (1 clock cycle)

value 4
355ns Select (5 clock cycles), IOR/IOW after 172ns (2 clock cycles)

value 5
355ns Select (5 clock cycles), IOR/IOW after 243ns (3 clock cycles)

value 6
1065ns Select (15 clock cycles), IOR/IOW after 314ns (4 clock cycles)

value 7
355ns Select, (5 clock cycles), IOR/IOW after 101ns (1 clock cycle)

When accessing IDE registers with A6=1 (for example $84x), the timing will always be mode 0 8-bit compatible, no matter what you have selected in the speed register:

781ns select, IOR/IOW after 4 clock cycles (=314ns) aktive.

All the timings with a very short select-signal (the 355ns fast accesses) depend on the accelerator card used in the system: Sometimes two more clock cycles are inserted by the bus interface, making the whole access 497ns long. This doesn't affect the reliability of the controller nor the performance of the card, since this doesn't happen very often.

All the timings are calculated and only confirmed by measurements that allowed me to count the clock cycles. If the system is clocked by an oscillator other than 28,37516 Mhz (for example the NTSC-frequency 28,63636 Mhz), each clock cycle is shortened to a bit less than 70ns (not worth mentioning). You could think of a small performance boost by overclocking the system, but you would either need a multisync monitor, or a graphics card, and your internal diskdrive would go crazy, that's why you shouldn't tune your Amiga this way.

Giving you the possibility to write software that is compatible with both the Buddha and the Catweasel Z-II, The Buddha acts just like a Catweasel Z-II with no device connected to the third IDE-port. The IRQ-register $f80 always shows a "no IRQ here" on the Buddha, and accesses to the third IDE port are going into data's Nirwana on the Buddha.

Jens Schönfeld february 19th, 1997

updated may 27th, 1997

eMail: sysop@nostlgic.tng.oche.de

6.3 Feature status on m68k architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	TODO
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	TODO
core	jump-labels	HAVE_ARCH_JUMP_LABEL	TODO
core	thread-info-in-task	THREAD_INFO_IN_TASK	TODO
core	tracehook	HAVE_ARCH_TRACEHOOK	TODO
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	TODO
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	TODO
debug	KASAN	HAVE_ARCH_KASAN	TODO
debug	kcov	ARCH_HAS_KCOV	TODO
debug	kgdb	HAVE_ARCH_KGDB	TODO
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	TODO
debug	kprobes	HAVE_KPROBES	TODO
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	TODO
debug	kretprobes	HAVE_KRETPROBES	TODO
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	TODO
debug	uprobes	ARCH_SUPPORTS_UPROBES	TODO
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	TODO
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO
locking	lockdep	LOCKDEP_SUPPORT	TODO
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	TODO
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	TODO
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	TODO
perf	perf-regs	HAVE_PERF_REGS	TODO
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	TODO
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	TODO
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	---
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	TODO
time	clockevents	!LEGACY_TIMER_TICK	TODO
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	TODO
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	TODO
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	TODO
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	---
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	TODO
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO
vm	ioremap_prot	HAVE_IOREMAP_PROT	TODO
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	TODO
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	---

CHAPTER

SEVEN

MIPS-SPECIFIC DOCUMENTATION

7.1 BMIPS DeviceTree Booting

Some bootloaders only support a single entry point, at the start of the kernel image. Other bootloaders will jump to the ELF start address. Both schemes are supported; CONFIG_BOOT_RAW=y and CONFIG_NO_EXCEPT_FILL=y, so the first instruction immediately jumps to kernel_entry().

Similar to the arch/arm case (b), a DT-aware bootloader is expected to set up the following registers:

a0 : 0

a1 : 0xffffffff

a2 : Physical pointer to the device tree block (defined in chapter II) in RAM. The device tree can be located anywhere in the first 512MB of the physical address space (0x00000000 - 0x1fffffff), aligned on a 64 bit boundary.

Legacy bootloaders do not use this convention, and they do not pass in a DT block. In this case, Linux will look for a builtin DTB, selected via CONFIG_DT_*.

This convention is defined for 32-bit systems only, as there are not currently any 64-bit BMIPS implementations.

7.2 Ingenic JZ47xx SoCs Timer/Counter Unit hardware

The Timer/Counter Unit (TCU) in Ingenic JZ47xx SoCs is a multi-function hardware block. It features up to eight channels, that can be used as counters, timers, or PWM.

- JZ4725B, JZ4750, JZ4755 only have six TCU channels. The other SoCs all have eight channels.
- JZ4725B introduced a separate channel, called Operating System Timer (OST). It is a 32-bit programmable timer. On JZ4760B and above, it is 64-bit.
- Each one of the TCU channels has its own clock, which can be reparented to three different clocks (pclk, ext, rtc), gated, and reclocked, through their TCSR register.
  - The watchdog and OST hardware blocks also feature a TCSR register with the same format in their register space.
  - The TCU registers used to gate/ungate can also gate/ungate the watchdog and OST clocks.

- Each TCU channel works in one of two modes:
  - mode TCU1: channels cannot work in sleep mode, but are easier to operate.
  - mode TCU2: channels can work in sleep mode, but the operation is a bit more complicated than with TCU1 channels.
- The mode of each TCU channel depends on the SoC used:
  - On the oldest SoCs (up to JZ4740), all of the eight channels operate in TCU1 mode.
  - On JZ4725B, channel 5 operates as TCU2, the others operate as TCU1.
  - On newest SoCs (JZ4750 and above), channels 1-2 operate as TCU2, the others operate as TCU1.
- Each channel can generate an interrupt. Some channels share an interrupt line, some don't, and this changes between SoC versions:
  - on older SoCs (JZ4740 and below), channel 0 and channel 1 have their own interrupt line; channels 2-7 share the last interrupt line.
  - On JZ4725B, channel 0 has its own interrupt; channels 1-5 share one interrupt line; the OST uses the last interrupt line.
  - on newer SoCs (JZ4750 and above), channel 5 has its own interrupt; channels 0-4 and (if eight channels) 6-7 all share one interrupt line; the OST uses the last interrupt line.

7.2.1 Implementation

The functionalities of the TCU hardware are spread across multiple drivers:

clocks	drivers/clk/ingenic/tcu.c
interrupts	drivers/irqchip/irq-ingenic-tcu.c
timers	drivers/clocksource/ingenic-timer.c
OST	drivers/clocksource/ingenic-ost.c
PWM	drivers/pwm/pwm-jz4740.c
watchdog	drivers/watchdog/jz4740_wdt.c

Because various functionalities of the TCU that belong to different drivers and frameworks can be controlled from the same registers, all of these drivers access their registers through the same regmap.

For more information regarding the devicetree bindings of the TCU drivers, have a look at Documentation/devicetree/bindings/timer/ingenic,tcu.yaml.

7.3 Feature status on mips architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	ok
core	eBPF-JIT	HAVE_EBPF_JIT	ok
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok
core	thread-info-in-task	THREAD_INFO_IN_TASK	TODO
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	TODO
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	ok
debug	KASAN	HAVE_ARCH_KASAN	TODO
debug	kcov	ARCH_HAS_KCOV	ok
debug	kgdb	HAVE_ARCH_KGDB	ok
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok
debug	kprobes	HAVE_KPROBES	ok
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	TODO
debug	kretprobes	HAVE_KRETPROBES	ok
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	ok
debug	uprobes	ARCH_SUPPORTS_UPROBES	ok
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	ok
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	ok
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	ok
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok
perf	perf-regs	HAVE_PERF_REGS	ok
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	ok
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	TODO
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	TODO
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	ok
time	clockevents	!LEGACY_TIMER_TICK	ok
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	ok
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	ok
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	ok
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	TODO
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	ok
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO
vm	ioremap_prot	HAVE_IOREMAP_PROT	ok
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	ok

CHAPTER

EIGHT

NIOS II SPECIFIC DOCUMENTATION

8.1 Linux on the Nios II architecture

This is a port of Linux to Nios II (nios2) processor.

In order to compile for Nios II, you need a version of GCC with support for the generic system call ABI. Please see this link for more information on how compiling and booting software for the Nios II platform: http://www.rocketboards.org/foswiki/Documentation/NiosIILinuxUserManual

For reference, please see the following link: http://www.altera.com/literature/lit-nio2.jsp

8.1.1 What is Nios II?

Nios II is a 32-bit embedded-processor architecture designed specifically for the Altera family of FPGAs. In order to support Linux, Nios II needs to be configured with MMU and hardware multiplier enabled.

8.1.2 Nios II ABI

Please refer to chapter "Application Binary Interface" in Nios II Processor Reference Handbook.

8.2 Feature status on nios2 architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	TODO
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	TODO
core	jump-labels	HAVE_ARCH_JUMP_LABEL	TODO
core	thread-info-in-task	THREAD_INFO_IN_TASK	TODO
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	TODO
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	TODO
debug	KASAN	HAVE_ARCH_KASAN	TODO
debug	kcov	ARCH_HAS_KCOV	TODO
debug	kgdb	HAVE_ARCH_KGDB	ok
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	TODO

Table 1 – continued fron

Subsystem	Feature	Kconfig	Status
debug	kprobes	HAVE_KPROBES	TODO
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	TODO
debug	kretprobes	HAVE_KRETPROBES	TODO
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	TODO
debug	uprobes	ARCH_SUPPORTS_UPROBES	TODO
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	TODO
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO
locking	lockdep	LOCKDEP_SUPPORT	TODO
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	TODO
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	TODO
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	TODO
perf	perf-regs	HAVE_PERF_REGS	TODO
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	TODO
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	TODO
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	---
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	TODO
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	TODO
time	clockevents	!LEGACY_TIMER_TICK	ok
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	TODO
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	TODO
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	TODO
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	---
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	TODO
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO
vm	ioremap_prot	HAVE_IOREMAP_PROT	TODO
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	TODO
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	---

CHAPTER

NINE

OPENRISC ARCHITECTURE

9.1 OpenRISC Linux

This is a port of Linux to the OpenRISC class of microprocessors; the initial target architecture, specifically, is the 32-bit OpenRISC 1000 family (or1k).

For information about OpenRISC processors and ongoing development:

website	https://openrisc.io
email	openrisc@lists.librecores.org

9.1.1 Build instructions for OpenRISC toolchain and Linux

In order to build and run Linux for OpenRISC, you'll need at least a basic toolchain and, perhaps, the architectural simulator. Steps to get these bits in place are outlined here.

1) Toolchain

Toolchain binaries can be obtained from openrisc.io or our github releases page. Instructions for building the different toolchains can be found on openrisc.io or Stafford's toolchain build and release scripts.

binaries	https://github.com/openrisc/or1k-gcc/releases
toolchains	https://openrisc.io/software
building	https://github.com/stffrdhrn/or1k-toolchain-build

2) Building

Build the Linux kernel as usual:

```
make ARCH=openrisc CROSS_COMPILE="or1k-linux-" defconfig
make ARCH=openrisc CROSS_COMPILE="or1k-linux-"
```

3) Running on FPGA (optional)

The OpenRISC community typically uses FuseSoC to manage building and programming an SoC into an FPGA. The below is an example of programming a De0 Nano development board with the OpenRISC SoC. During the build FPGA RTL is code downloaded from the FuseSoC IP cores repository and built using the FPGA vendor tools. Binaries are loaded onto the board with openocd.

```
git clone https://github.com/olofk/fusesoc
cd fusesoc
sudo pip install -e .

fusesoc init
fusesoc build de0_nano
fusesoc pgm de0_nano

openocd -f interface/altera-usb-blaster.cfg \
        -f board/or1k_generic.cfg

telnet localhost 4444
> init
> halt; load_image vmlinux ; reset
```

4) Running on a Simulator (optional)

QEMU is a processor emulator which we recommend for simulating the OpenRISC platform. Please follow the OpenRISC instructions on the QEMU website to get Linux running on QEMU. You can build QEMU yourself, but your Linux distribution likely provides binary packages to support OpenRISC.

qemu openrisc	https://wiki.qemu.org/Documentation/Platforms/OpenRISC

9.1.2 Terminology

In the code, the following particles are used on symbols to limit the scope to more or less specific processor implementations:

openrisc:	the OpenRISC class of processors
or1k:	the OpenRISC 1000 family of processors
or1200:	the OpenRISC 1200 processor

9.1.3 History

18-11-2003 Matjaz Breskvar (phoenix@bsemi.com)
initial port of linux to OpenRISC/or32 architecture. all the core stuff is implemented and seams usable.

08-12-2003 Matjaz Breskvar (phoenix@bsemi.com)
complete change of TLB miss handling. rewrite of exceptions handling. fully functional sash-3.6 in default initrd. a much improved version with changes all around.

10-04-2004 Matjaz Breskvar (phoenix@bsemi.com)
a lot of bugfixes all over. ethernet support, functional http and telnet servers. running many standard linux apps.

26-06-2004 Matjaz Breskvar (phoenix@bsemi.com)
port to 2.6.x

30-11-2004 Matjaz Breskvar (phoenix@bsemi.com)
lots of bugfixes and enhancements. added opencores framebuffer driver.

09-10-2010 Jonas Bonn (jonas@southpole.se)
major rewrite to bring up to par with upstream Linux 2.6.36

9.2 TODO

The OpenRISC Linux port is fully functional and has been tracking upstream since 2.6.35. There are, however, remaining items to be completed within the coming months. Here's a list of known-to-be-less-than-stellar items that are due for investigation shortly, i.e. our TODO list:

- Implement the rest of the DMA API… dma_map_sg, etc.
- Finish the renaming cleanup… there are references to or32 in the code which was an older name for the architecture. The name we've settled on is or1k and this change is slowly trickling through the stack. For the time being, or32 is equivalent to or1k.

9.3 Feature status on openrisc architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	TODO
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	TODO
core	thread-info-in-task	THREAD_INFO_IN_TASK	TODO
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	TODO
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	TODO
debug	KASAN	HAVE_ARCH_KASAN	TODO
debug	kcov	ARCH_HAS_KCOV	TODO
debug	kgdb	HAVE_ARCH_KGDB	TODO
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	TODO
debug	kprobes	HAVE_KPROBES	TODO
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	TODO
debug	kretprobes	HAVE_KRETPROBES	TODO
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	TODO
debug	uprobes	ARCH_SUPPORTS_UPROBES	TODO
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	TODO
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO

Table 1 – continued fron

Subsystem	Feature	Kconfig	Status
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	ok
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	ok
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	TODO
perf	perf-regs	HAVE_PERF_REGS	TODO
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	TODO
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	TODO
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	---
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	TODO
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	TODO
time	clockevents	!LEGACY_TIMER_TICK	ok
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	TODO
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	TODO
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	TODO
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	---
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	TODO
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO
vm	ioremap_prot	HAVE_IOREMAP_PROT	TODO
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	TODO
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	---

CHAPTER

TEN

PA-RISC ARCHITECTURE

10.1 PA-RISC Debugging

okay, here are some hints for debugging the lower-level parts of linux/parisc.

10.1.1 1. Absolute addresses

A lot of the assembly code currently runs in real mode, which means absolute addresses are used instead of virtual addresses as in the rest of the kernel. To translate an absolute address to a virtual address you can lookup in System.map, add __PAGE_OFFSET (0x10000000 currently).

10.1.2 2. HPMCs

When real-mode code tries to access non-existent memory, you'll get an HPMC instead of a kernel oops. To debug an HPMC, try to find the System Responder/Requestor addresses. The System Requestor address should match (one of the) processor HPAs (high addresses in the I/O range); the System Responder address is the address real-mode code tried to access.

Typical values for the System Responder address are addresses larger than __PAGE_OFFSET (0x10000000) which mean a virtual address didn't get translated to a physical address before real-mode code tried to access it.

10.1.3 3. Q bit fun

Certain, very critical code has to clear the Q bit in the PSW. What happens when the Q bit is cleared is the CPU does not update the registers interruption handlers read to find out where the machine was interrupted - so if you get an interruption between the instruction that clears the Q bit and the RFI that sets it again you don't know where exactly it happened. If you're lucky the IAOQ will point to the instruction that cleared the Q bit, if you're not it points anywhere at all. Usually Q bit problems will show themselves in unexplainable system hangs or running off the end of physical memory.

10.2 Register Usage for Linux/PA-RISC

[an asterisk is used for planned usage which is currently unimplemented]

10.2.1 General Registers as specified by ABI

Control Registers

CR 0 (Recovery Counter)	used for ptrace
CR 1-CR 7(undefined)	unused
CR 8 (Protection ID)	per-process value*
CR 9, 12, 13 (PIDS)	unused
CR10 (CCR)	lazy FPU saving*
CR11	as specified by ABI (SAR)
CR14 (interruption vector)	initialized to fault_vector
CR15 (EIEM)	initialized to all ones*
CR16 (Interval Timer)	read for cycle count/write starts Interval Tmr
CR17-CR22	interruption parameters
CR19	Interrupt Instruction Register
CR20	Interrupt Space Register
CR21	Interrupt Offset Register
CR22	Interrupt PSW
CR23 (EIRR)	read for pending interrupts/write clears bits
CR24 (TR 0)	Kernel Space Page Directory Pointer
CR25 (TR 1)	User Space Page Directory Pointer
CR26 (TR 2)	not used
CR27 (TR 3)	Thread descriptor pointer
CR28 (TR 4)	not used
CR29 (TR 5)	not used
CR30 (TR 6)	current / 0
CR31 (TR 7)	Temporary register, used in various places

Space Registers (kernel mode)

SR0	temporary space register
SR4-SR7	set to 0
SR1	temporary space register
SR2	kernel should not clobber this
SR3	used for userspace accesses (current process)

Space Registers (user mode)

SR0	temporary space register
SR1	temporary space register
SR2	holds space of linux gateway page
SR3	holds user address space value while in kernel
SR4-SR7	Defines short address space for user/kernel

Processor Status Word

W (64-bit addresses)	0
E (Little-endian)	0
S (Secure Interval Timer)	0
T (Taken Branch Trap)	0
H (Higher-privilege trap)	0
L (Lower-privilege trap)	0
N (Nullify next instruction)	used by C code
X (Data memory break disable)	0
B (Taken Branch)	used by C code
C (code address translation)	1, 0 while executing real-mode code
V (divide step correction)	used by C code
M (HPMC mask)	0, 1 while executing HPMC handler*
C/B (carry/borrow bits)	used by C code
O (ordered references)	1*
F (performance monitor)	0
R (Recovery Counter trap)	0
Q (collect interruption state)	1 (0 in code directly preceding an rfi)
P (Protection Identifiers)	1*
D (Data address translation)	1, 0 while executing real-mode code
I (external interrupt mask)	used by cli()/sti() macros

“Invisible” Registers

PSW default W value	0
PSW default E value	0
Shadow Registers	used by interruption handler code
TOC enable bit	1

The PA-RISC architecture defines 7 registers as “shadow registers”. Those are used in RETURN FROM INTERRUPTION AND RESTORE instruction to reduce the state save and restore time by eliminating the need for general register (GR) saves and restores in interruption handlers. Shadow registers are the GRs 1, 8, 9, 16, 17, 24, and 25.

Register usage notes, originally from John Marvin, with some additional notes from Randolph Chung.

For the general registers:

r1,r2,r19-r26,r28,r29 & r31 can be used without saving them first. And of course, you need to save them if you care about them, before calling another procedure. Some of the above registers do have special meanings that you should be aware of:

r1:
: The addil instruction is hardwired to place its result in r1, so if you use that instruction be aware of that.

r2:
: This is the return pointer. In general you don't want to use this, since you need the pointer to get back to your caller. However, it is grouped with this set of registers since the caller can't rely on the value being the same when you return, i.e. you can copy r2 to another register and return through that register after trashing r2, and that should not cause a problem for the calling routine.

r19-r22:
: these are generally regarded as temporary registers. Note that in 64 bit they are arg7-arg4.

r23-r26:
: these are arg3-arg0, i.e. you can use them if you don't care about the values that were passed in anymore.

r28,r29:
: are ret0 and ret1. They are what you pass return values in. r28 is the primary return. When returning small structures r29 may also be used to pass data back to the caller.

r30:
: stack pointer

r31:
: the ble instruction puts the return pointer in here.

r3-r18,r27,r30 need to be saved and restored. r3-r18 are just general purpose registers. r27 is the data pointer, and is used to make references to global variables easier. r30 is the stack pointer.

10.3 Feature status on parisc architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	TODO
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok
core	thread-info-in-task	THREAD_INFO_IN_TASK	ok
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	ok

Table 1 – continued fron

Subsystem	Feature	Kconfig	Status
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	TODO
debug	KASAN	HAVE_ARCH_KASAN	TODO
debug	kcov	ARCH_HAS_KCOV	TODO
debug	kgdb	HAVE_ARCH_KGDB	ok
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	TODO
debug	kprobes	HAVE_KPROBES	ok
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	ok
debug	kretprobes	HAVE_KRETPROBES	ok
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	TODO
debug	uprobes	ARCH_SUPPORTS_UPROBES	TODO
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	TODO
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	TODO
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	TODO
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok
perf	perf-regs	HAVE_PERF_REGS	TODO
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	TODO
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	TODO
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	---
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	TODO
time	clockevents	!LEGACY_TIMER_TICK	TODO
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	---
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	TODO
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	ok
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	TODO
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	ok
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO
vm	ioremap_prot	HAVE_IOREMAP_PROT	TODO
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	TODO

CHAPTER

ELEVEN

POWERPC

11.1 NUMA resource associativity

Associativity represents the groupings of the various platform resources into domains of substantially similar mean performance relative to resources outside of that domain. Resources subsets of a given domain that exhibit better performance relative to each other than relative to other resources subsets are represented as being members of a sub-grouping domain. This performance characteristic is presented in terms of NUMA node distance within the Linux kernel. From the platform view, these groups are also referred to as domains.

PAPR interface currently supports different ways of communicating these resource grouping details to the OS. These are referred to as Form 0, Form 1 and Form2 associativity grouping. Form 0 is the oldest format and is now considered deprecated.

Hypervisor indicates the type/form of associativity used via "ibm,architecture-vec-5 property". Bit 0 of byte 5 in the "ibm,architecture-vec-5" property indicates usage of Form 0 or Form 1. A value of 1 indicates the usage of Form 1 associativity. For Form 2 associativity bit 2 of byte 5 in the "ibm,architecture-vec-5" property is used.

11.1.1 Form 0

Form 0 associativity supports only two NUMA distances (LOCAL and REMOTE).

11.1.2 Form 1

With Form 1 a combination of ibm,associativity-reference-points, and ibm,associativity device tree properties are used to determine the NUMA distance between resource groups/domains.

The "ibm,associativity" property contains a list of one or more numbers (domainID) representing the resource's platform grouping domains.

The "ibm,associativity-reference-points" property contains a list of one or more numbers (domainID index) that represents the 1 based ordinal in the associativity lists. The list of domainID indexes represents an increasing hierarchy of resource grouping.

ex: { primary domainID index, secondary domainID index, tertiary domainID index.. }

Linux kernel uses the domainID at the primary domainID index as the NUMA node id. Linux kernel computes NUMA distance between two domains by recursively comparing if they belong to the same higher-level domains. For mismatch at every higher level of the resource group, the kernel doubles the NUMA distance between the comparing domains.

11.1.3 Form 2

Form 2 associativity format adds separate device tree properties representing NUMA node distance thereby making the node distance computation flexible. Form 2 also allows flexible primary domain numbering. With numa distance computation now detached from the index value in "ibm,associativity-reference-points" property, Form 2 allows a large number of primary domain ids at the same domainID index representing resource groups of different performance/latency characteristics.

Hypervisor indicates the usage of FORM2 associativity using bit 2 of byte 5 in the "ibm,architecture-vec-5" property.

"ibm,numa-lookup-index-table" property contains a list of one or more numbers representing the domainIDs present in the system. The offset of the domainID in this property is used as an index while computing numa distance information via "ibm,numa-distance-table".

prop-encoded-array: The number N of the domainIDs encoded as with encode-int, followed by N domainID encoded as with encode-int

For ex: "ibm,numa-lookup-index-table" = {4, 0, 8, 250, 252}. The offset of domainID 8 (2) is used when computing the distance of domain 8 from other domains present in the system. For the rest of this document, this offset will be referred to as domain distance offset.

"ibm,numa-distance-table" property contains a list of one or more numbers representing the NUMA distance between resource groups/domains present in the system.

prop-encoded-array: The number N of the distance values encoded as with encode-int, followed by N distance values encoded as with encode-bytes. The max distance value we could encode is 255. The number N must be equal to the square of m where m is the number of domainIDs in the numa-lookup-index-table.

For ex: ibm,numa-lookup-index-table = <3 0 8 40>; ibm,numa-distace-table = <9>, /bits/ 8 < 10 20 80 20 10 160 80 160 10>;

```
  | 0    8   40
--|------------
  |
0 | 10   20  80
  |
8 | 20   10  160
  |
40| 80   160  10
```

A possible "ibm,associativity" property for resources in node 0, 8 and 40

{ 3, 6, 7, 0 } { 3, 6, 9, 8 } { 3, 6, 7, 40}

With "ibm,associativity-reference-points" { 0x3 }

"ibm,lookup-index-table" helps in having a compact representation of distance matrix. Since domainID can be sparse, the matrix of distances can also be effectively sparse. With "ibm,lookup-index-table" we can achieve a compact representation of distance information.

11.2 DeviceTree Booting

During the development of the Linux/ppc64 kernel, and more specifically, the addition of new platform types outside of the old IBM pSeries/iSeries pair, it was decided to enforce some strict rules regarding the kernel entry and bootloader <-> kernel interfaces, in order to avoid the degeneration that had become the ppc32 kernel entry point and the way a new platform should be added to the kernel. The legacy iSeries platform breaks those rules as it predates this scheme, but no new board support will be accepted in the main tree that doesn't follow them properly. In addition, since the advent of the arch/powerpc merged architecture for ppc32 and ppc64, new 32-bit platforms and 32-bit platforms which move into arch/powerpc will be required to use these rules as well.

The main requirement that will be defined in more detail below is the presence of a device-tree whose format is defined after Open Firmware specification. However, in order to make life easier to embedded board vendors, the kernel doesn't require the device-tree to represent every device in the system and only requires some nodes and properties to be present. For example, the kernel does not require you to create a node for every PCI device in the system. It is a requirement to have a node for PCI host bridges in order to provide interrupt routing information and memory/IO ranges, among others. It is also recommended to define nodes for on chip devices and other buses that don't specifically fit in an existing OF specification. This creates a great flexibility in the way the kernel can then probe those and match drivers to device, without having to hard code all sorts of tables. It also makes it more flexible for board vendors to do minor hardware upgrades without significantly impacting the kernel code or cluttering it with special cases.

11.2.1 Entry point

There is one single entry point to the kernel, at the start of the kernel image. That entry point supports two calling conventions:

> a) Boot from Open Firmware. If your firmware is compatible with Open Firmware (IEEE 1275) or provides an OF compatible client interface API (support for "interpret" callback of forth words isn't required), you can enter the kernel with:
>
> > r5 : OF callback pointer as defined by IEEE 1275 bindings to powerpc. Only the 32-bit client interface is currently supported
> >
> > r3, r4 : address & length of an initrd if any or 0
> >
> > The MMU is either on or off; the kernel will run the trampoline located in arch/powerpc/kernel/prom_init.c to extract the device-tree and other information from open firmware and build a flattened device-tree as described in b). prom_init() will then re-enter the kernel using the second method. This trampoline code runs in the context of the firmware, which is supposed to handle all exceptions during that time.
>
> b) Direct entry with a flattened device-tree block. This entry point is called by a) after the OF trampoline and can also be called directly by a bootloader that does not support the Open Firmware client interface. It is also used by "kexec" to implement "hot" booting of a new kernel from a previous running one. This method is what I will describe in more details in this document, as method a) is simply standard Open Firmware, and thus should be implemented according to the various standard documents defining it and its binding to the PowerPC platform. The entry point definition

then becomes:

r3 : physical pointer to the device-tree block (defined in chapter II) in RAM

r4 : physical pointer to the kernel itself. This is used by the assembly code to properly disable the MMU in case you are entering the kernel with MMU enabled and a non-1:1 mapping.

r5 : NULL (as to differentiate with method a)

Note about SMP entry: Either your firmware puts your other CPUs in some sleep loop or spin loop in ROM where you can get them out via a soft reset or some other means, in which case you don't need to care, or you'll have to enter the kernel with all CPUs. The way to do that with method b) will be described in a later revision of this document.

Board supports (platforms) are not exclusive config options. An arbitrary set of board supports can be built in a single kernel image. The kernel will "know" what set of functions to use for a given platform based on the content of the device-tree. Thus, you should:

a) add your platform support as a _boolean_ option in arch/powerpc/Kconfig, following the example of PPC_PSERIES, PPC_PMAC and PPC_MAPLE. The latter is probably a good example of a board support to start from.

b) create your main platform file as "arch/powerpc/platforms/myplatform/myboard_setup.c" and add it to the Makefile under the condition of your `CONFIG_` option. This file will define a structure of type "ppc_md" containing the various callbacks that the generic code will use to get to your platform specific code

A kernel image may support multiple platforms, but only if the platforms feature the same core architecture. A single kernel build cannot support both configurations with Book E and configurations with classic Powerpc architectures.

11.3 The PowerPC boot wrapper

Copyright (C) Secret Lab Technologies Ltd.

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.

11.3.1 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.

11.4 CPU Families

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

11.4.1 Book3S (aka sPAPR)

- Hash MMU (except 603 and e300)
- Radix MMU (POWER9 and later)
- Software loaded TLB (603 and e300)
- Selectable Software loaded TLB in addition to hash MMU (755, 7450, e600)
- Mix of 32 & 64 bit:

```
+--------------+                      +----------------+
|  Old POWER   | --------------> | RS64 (threads) |
+--------------+                      +----------------+
       |
       |
       v
+--------------+                  +----------------+      +------+
|     601      | --------------> |      603       | ---> | e300 |
+--------------+                  +----------------+      +------+
       |                                 |
       |                                 |
       v                                 v
+--------------+    +-----+       +----------------+      +-------+
|     604      |    | 755 | <---  |    750 (G3)    | ---> | 750CX |
+--------------+    +-----+       +----------------+      +-------+
       |                                 |                   |
       |                                 |                   |
       v                                 v                   v
+--------------+                  +----------------+      +-------+
| 620 (64 bit) |                  |      7400      |      | 750CL |
+--------------+                  +----------------+      +-------+
       |                                 |                   |
       |                                 |                   |
       v                                 v                   v
+--------------+                  +----------------+      +-------+
|  POWER3/630  |                  |      7410      |      | 750FX |
+--------------+                  +----------------+      +-------+
       |                                 |
       |                                 |
       v                                 v
+--------------+                  +----------------+
|   POWER3+    |                  |      7450      |
+--------------+                  +----------------+
       |                                 |
       |                                 |
       v                                 v
+--------------+                  +----------------+
|    POWER4    |                  |      7455      |
+--------------+                  +----------------+
       |                                 |
       |                                 |
       v                                 v
+--------------+     +-------+    +----------------+
|   POWER4+    | --> |  970  |    |      7447      |
+--------------+     +-------+    +----------------+
       |                 |               |
       |                 |               |
       v                 v               v
+--------------+     +-------+    +----------------+
|    POWER5    |     | 970FX |    |      7448      |
+--------------+     +-------+    +----------------+
       |                 |               |
       |                 |               |
       v                  v                v
+--------------+     +-------+   +----------------+
|   POWER5+    |     | 970MP |   |      e600      |
+--------------+     +-------+   +----------------+
       |
       |
       v
+--------------+
|   POWER5++   |
+--------------+
       |
       |
       v
+--------------+       +-------+
|    POWER6    | <-?-> | Cell  |
+--------------+       +-------+
       |
       |
       v
+--------------+
|    POWER7    |
+--------------+
       |
       |
       v
+--------------+
|   POWER7+    |
+--------------+
       |
       |
       v
+--------------+
|    POWER8    |
+--------------+
       |
       |
       v
+--------------+
|    POWER9    |
+--------------+
       |
       |
       v
+--------------+
|   POWER10    |
+--------------+


+---------------+
| PA6T (64 bit) |
+---------------+
```

11.4.2 IBM BookE

- Software loaded TLB.
- All 32 bit:

```
+--------------+
|     401      |
+--------------+
       |
       |
       v
+--------------+
|     403      |
+--------------+
       |
       |
       v
+--------------+
|     405      |
+--------------+
       |
       |
       v
+--------------+
|     440      |
+--------------+
       |
       |
       v
+--------------+     +----------------+
|     450      | --> |      BG/P      |
+--------------+     +----------------+
       |
       |
       v
+--------------+
|     460      |
+--------------+
       |
       |
       v
+--------------+
|     476      |
+--------------+
```

11.4.3 Motorola/Freescale 8xx

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

```
+-------------+
| MPC8xx Core |
+-------------+
```

11.4.4 Freescale BookE

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

```
+--------------+
|     e200     |
+--------------+


+--------------------------------+
|              e500              |
+--------------------------------+
                |
                |
                v
+--------------------------------+
|             e500v2             |
+--------------------------------+
                |
                |
                v
+--------------------------------+
|        e500mc (Book3e)         |
+--------------------------------+
                |
                |
                v
+--------------------------------+
|          e5500 (64 bit)        |
+--------------------------------+
                |
                |
                v
+--------------------------------+
| e6500 (HW TLB) (Multithreaded) |
+--------------------------------+
```

11.4.5 IBM A2 core

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

```
+--------------+     +----------------+
|   A2 core    | --> |      WSP       |
+--------------+     +----------------+
        |
        |
        v
+--------------+
|     BG/Q     |
+--------------+
```

11.5 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.

11.6 Coherent Accelerator Interface (CXL)

11.6.1 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.

11.6.2 Hardware overview

```
  POWER8/9             FPGA
+----------+        +---------+
|          |        |         |
|   CPU    |        |   AFU   |
|          |        |         |
|          |        |         |
|          |        |         |
+----------+        +---------+
|   PHB    |        |         |
|   +------+        |   PSL   |
|   | CAPP |<------>|         |
+---+------+  PCIE  +---------+
```

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.

11.6.3 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.

11.6.4 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.

11.6.5 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.

11.6.6 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.

11.6.7 User API

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

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.

11.6.8 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

11.6.9 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"
```

11.7 Coherent Accelerator (CXL) Flash

11.7.1 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 *Coherent Accelerator Interface (CXL)*.

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.

11.7.2 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).

11.7.3 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

11.7.4 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

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.

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.

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.

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.

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.

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.

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.

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.

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).

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).

11.7.5 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

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.

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.

11.8 DAWR issues on POWER9

On older POWER9 processors, the Data Address Watchpoint Register (DAWR) can cause a checkstop if it points to cache inhibited (CI) memory. Currently Linux has no way to distinguish CI memory when configuring the DAWR, so on affected systems, the DAWR is disabled.

11.8.1 Affected processor revisions

This issue is only present on processors prior to v2.3. The revision can be found in /proc/cpuinfo:

```
processor       : 0
cpu             : POWER9, altivec supported
clock           : 3800.000000MHz
revision        : 2.3 (pvr 004e 1203)
```

On a system with the issue, the DAWR is disabled as detailed below.

11.8.2 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.

11.8.3 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.

11.8.4 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.

11.9 DEXCR (Dynamic Execution Control Register)

11.9.1 Overview

The DEXCR is a privileged special purpose register (SPR) introduced in PowerPC ISA 3.1B (Power10) that allows per-cpu control over several dynamic execution behaviours. These behaviours include speculation (e.g., indirect branch target prediction) and enabling return-oriented programming (ROP) protection instructions.

The execution control is exposed in hardware as up to 32 bits ('aspects') in the DEXCR. Each aspect controls a certain behaviour, and can be set or cleared to enable/disable the aspect. There are several variants of the DEXCR for different purposes:

DEXCR
A privileged SPR that can control aspects for userspace and kernel space

HDEXCR
A hypervisor-privileged SPR that can control aspects for the hypervisor and enforce aspects for the kernel and userspace.

UDEXCR
An optional ultravisor-privileged SPR that can control aspects for the ultravisor.

Userspace can examine the current DEXCR state using a dedicated SPR that provides a non-privileged read-only view of the userspace DEXCR aspects. There is also an SPR that provides a read-only view of the hypervisor enforced aspects, which ORed with the userspace DEXCR view gives the effective DEXCR state for a process.

11.9.2 Configuration

The DEXCR is currently unconfigurable. All threads are run with the NPHIE aspect enabled.

11.9.3 coredump and ptrace

The userspace values of the DEXCR and HDEXCR (in this order) are exposed under `NT_PPC_DEXCR`. These are each 64 bits and readonly, and are intended to assist with core dumps. The DEXCR may be made writable in future. The top 32 bits of both registers (corresponding to the non-userspace bits) are masked off.

If the kernel config `CONFIG_CHECKPOINT_RESTORE` is enabled, then `NT_PPC_HASHKEYR` is available and exposes the HASHKEYR value of the process for reading and writing. This is a tradeoff between increased security and checkpoint/restore support: a process should normally have no need to know its secret key, but restoring a process requires setting its original key. The key therefore appears in core dumps, and an attacker may be able to retrieve it from a coredump and effectively bypass ROP protection on any threads that share this key (potentially all threads from the same parent that have not run `exec()`).

11.10 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, its 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.

11.11 PCI Bus EEH Error Recovery

Linas Vepstas <linas@austin.ibm.com>

12 January 2005

11.11.1 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.

11.11.2 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.

11.11.3 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 continue 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.

11.11.4 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).

11.11.5 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
                       {
                         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.
```

11.11.6 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.

11.11.7 Conclusions

There's forward progress …

11.12 POWERPC ELF HWCAPs

This document describes the usage and semantics of the powerpc ELF HWCAPs.

11.12.1 1. Introduction

Some hardware or software features are only available on some CPU implementations, and/or with certain kernel configurations, but have no other discovery mechanism available to userspace code. The kernel exposes the presence of these features to userspace through a set of flags called HWCAPs, exposed in the auxiliary vector.

Userspace software can test for features by acquiring the AT_HWCAP or AT_HWCAP2 entry of the auxiliary vector, and testing whether the relevant flags are set, e.g.:

```
bool floating_point_is_present(void)
{
        unsigned long HWCAPs = getauxval(AT_HWCAP);
        if (HWCAPs & PPC_FEATURE_HAS_FPU)
                return true;

        return false;
}
```

Where software relies on a feature described by a HWCAP, it should check the relevant HWCAP flag to verify that the feature is present before attempting to make use of the feature.

HWCAP is the preferred method to test for the presence of a feature rather than probing through other means, which may not be reliable or may cause unpredictable behaviour.

Software that targets a particular platform does not necessarily have to test for required or implied features. For example if the program requires FPU, VMX, VSX, it is not necessary to test those HWCAPs, and it may be impossible to do so if the compiler generates code requiring those features.

11.12.2 2. Facilities

The Power ISA uses the term "facility" to describe a class of instructions, registers, interrupts, etc. The presence or absence of a facility indicates whether this class is available to be used, but the specifics depend on the ISA version. For example, if the VSX facility is available, the VSX instructions that can be used differ between the v3.0B and v3.1B ISA versions.

11.12.3 3. Categories

The Power ISA before v3.0 uses the term "category" to describe certain classes of instructions and operating modes which may be optional or mutually exclusive, the exact meaning of the HWCAP flag may depend on context, e.g., the presence of the BOOKE feature implies that the server category is not implemented.

11.12.4 4. HWCAP allocation

HWCAPs are allocated as described in Power Architecture 64-Bit ELF V2 ABI Specification (which will be reflected in the kernel's uapi headers).

11.12.5 5. The HWCAPs exposed in AT_HWCAP

PPC_FEATURE_32
32-bit CPU

PPC_FEATURE_64
64-bit CPU (userspace may be running in 32-bit mode).

PPC_FEATURE_601_INSTR
The processor is PowerPC 601. Unused in the kernel since f0ed73f3fa2c ("powerpc: Remove PowerPC 601")

PPC_FEATURE_HAS_ALTIVEC
Vector (aka Altivec, VMX) facility is available.

PPC_FEATURE_HAS_FPU
Floating point facility is available.

PPC_FEATURE_HAS_MMU
Memory management unit is present and enabled.

PPC_FEATURE_HAS_4xxMAC
The processor is 40x or 44x family.

PPC_FEATURE_UNIFIED_CACHE
The processor has a unified L1 cache for instructions and data, as found in NXP e200. Unused in the kernel since 39c8bf2b3cc1 ("powerpc: Retire e200 core (mpc555x processor)")

PPC_FEATURE_HAS_SPE
Signal Processing Engine facility is available.

PPC_FEATURE_HAS_EFP_SINGLE
Embedded Floating Point single precision operations are available.

PPC_FEATURE_HAS_EFP_DOUBLE
Embedded Floating Point double precision operations are available.

PPC_FEATURE_NO_TB
The timebase facility (mftb instruction) is not available. This is a 601 specific HWCAP, so if it is known that the processor running is not a 601, via other HWCAPs or other means, it is not required to test this bit before using the timebase. Unused in the kernel since f0ed73f3fa2c ("powerpc: Remove PowerPC 601")

PPC_FEATURE_POWER4
The processor is POWER4 or PPC970/FX/MP. POWER4 support dropped from the kernel since 471d7ff8b51b ("powerpc/64s: Remove POWER4 support")

PPC_FEATURE_POWER5
The processor is POWER5.

PPC_FEATURE_POWER5_PLUS
The processor is POWER5+.

PPC_FEATURE_CELL
The processor is Cell.

PPC_FEATURE_BOOKE
The processor implements the embedded category ("BookE") architecture.

PPC_FEATURE_SMT
The processor implements SMT.

PPC_FEATURE_ICACHE_SNOOP
The processor icache is coherent with the dcache, and instruction storage can be made consistent with data storage for the purpose of executing instructions with the sequence (as described in, e.g., POWER9 Processor User's Manual, 4.6.2.2 Instruction Cache Block Invalidate (icbi)):

```
sync
icbi (to any address)
isync
```

PPC_FEATURE_ARCH_2_05
The processor supports the v2.05 userlevel architecture. Processors supporting later architectures DO NOT set this feature.

PPC_FEATURE_PA6T
The processor is PA6T.

PPC_FEATURE_HAS_DFP
DFP facility is available.

PPC_FEATURE_POWER6_EXT
The processor is POWER6.

PPC_FEATURE_ARCH_2_06
The processor supports the v2.06 userlevel architecture. Processors supporting later architectures also set this feature.

PPC_FEATURE_HAS_VSX
VSX facility is available.

PPC_FEATURE_PSERIES_PERFMON_COMPAT
The processor supports architected PMU events in the range 0xE0-0xFF.

PPC_FEATURE_TRUE_LE
The processor supports true little-endian mode.

PPC_FEATURE_PPC_LE
The processor supports "PowerPC Little-Endian", that uses address munging to make storage access appear to be little-endian, but the data is stored in a different format that is unsuitable to be accessed by other agents not running in this mode.

11.12.6 6. The HWCAPs exposed in AT_HWCAP2

PPC_FEATURE2_ARCH_2_07
The processor supports the v2.07 userlevel architecture. Processors supporting later architectures also set this feature.

PPC_FEATURE2_HTM
Transactional Memory feature is available.

PPC_FEATURE2_DSCR
DSCR facility is available.

PPC_FEATURE2_EBB
EBB facility is available.

PPC_FEATURE2_ISEL
isel instruction is available. This is superseded by ARCH_2_07 and later.

PPC_FEATURE2_TAR
TAR facility is available.

PPC_FEATURE2_VEC_CRYPTO
v2.07 crypto instructions are available.

PPC_FEATURE2_HTM_NOSC
System calls fail if called in a transactional state, see *Power Architecture 64-bit Linux system call ABI*

PPC_FEATURE2_ARCH_3_00
The processor supports the v3.0B / v3.0C userlevel architecture. Processors supporting later architectures also set this feature.

PPC_FEATURE2_HAS_IEEE128
IEEE 128-bit binary floating point is supported with VSX quad-precision instructions and data types.

PPC_FEATURE2_DARN
darn instruction is available.

PPC_FEATURE2_SCV
The scv 0 instruction may be used for system calls, see *Power Architecture 64-bit Linux system call ABI*.

PPC_FEATURE2_HTM_NO_SUSPEND
A limited Transactional Memory facility that does not support suspend is available, see *Transactional Memory support*.

PPC_FEATURE2_ARCH_3_1
The processor supports the v3.1 userlevel architecture. Processors supporting later architectures also set this feature.

PPC_FEATURE2_MMA
MMA facility is available.

11.13 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.

11.13.1 Types and Descriptors

The types to be used with the "PowerPC" namespace 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.

11.13.2 References

11.14 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.

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

- 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

11.14.1 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)    |
                                                        |
                                                        |
                    Metadata: This area holds a metadata structure whose
                    address is registered with f/w and retrieved in the
                    second kernel after crash, on platforms that support
                    tags (OPAL). Having such structure with info needed
                    to process the crashdump eases dump capture process.

                 Fig. 1


o Memory Reservation during second kernel after crash

Low memory                                             Top of memory
0      boot memory size                                     |
|           |<------------ Crash preserved area ------------>|
V           V           |<--- Reserved dump area --->|       |
+-----------+-----/ /---+---+----+-------+-----+-----+----+--+
|           |           |///|////|  DUMP | HDR | ELF |////|  |
+-----------+-----/ /---+---+----+-------+-----+-----+----+--+
      |                                           |
      V                                           V
 Used by second                            /proc/vmcore
 kernel to boot

      +---+
      |///| -> Regions (CPU, HPTE & Metadata) marked like this in the above
      +---+    figures are not always present. For example, OPAL platform
               does not have CPU & HPTE regions while Metadata region is
               not supported on pSeries currently.

                 Fig. 2
```

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 seamlessly (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.

11.14.2 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.

11.14.3 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
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.

11.14.4 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.

11.15 HVCS IBM "Hypervisor Virtual Console Server" Installation Guide

for Linux Kernel 2.6.4+

Copyright (C) 2004 IBM Corporation

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

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

11.15.1 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.

11.15.2 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.

11.15.3 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.

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.

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
```

11.15.4 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.

11.15.5 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.

11.15.6 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.

11.15.7 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".

11.15.8 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.

11.15.9 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.

11.16 IMC (In-Memory Collection Counters)

Anju T Sudhakar, 10 May 2019

Contents

11.16.1 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.

11.16.2 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
```

11.16.3 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.

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

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.

11.16.4 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.

11.16.5 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.

11.17 CPU to ISA Version Mapping

Mapping of some CPU versions to relevant ISA versions.

Note Power4 and Power4+ are not supported.

CPU	Architecture version
Power10	Power ISA v3.1
Power9	Power ISA v3.0B
Power8	Power ISA v2.07
e6500	Power ISA v2.06 with some exceptions
e5500	Power ISA v2.06 with some exceptions, no Altivec
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
Power4+	• 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
Power4	• PowerPC User Instruction Set Architecture Book I v2.00 • PowerPC Virtual Environment Architecture Book II v2.00 • PowerPC Operating Environment Architecture Book III v2.00

11.17.1 Key Features

CPU	VMX (aka. Altivec)
Power10	Yes
Power9	Yes
Power8	Yes
e6500	Yes
e5500	No
Power7	Yes
Power6	Yes
PA6T	Yes
Cell PPU	Yes
Power5++	No
Power5+	No
Power5	No
PPC970	Yes
Power4+	No
Power4	No

CPU	VSX
Power10	Yes
Power9	Yes
Power8	Yes
e6500	No
e5500	No
Power7	Yes
Power6	No
PA6T	No
Cell PPU	No
Power5++	No
Power5+	No
Power5	No
PPC970	No
Power4+	No
Power4	No

CPU	Transactional Memory
Power10	No (* see Power ISA v3.1, “Appendix A. Notes on the Removal of Transactional Memory from the Architecture”)
Power9	Yes (* see transactional_memory.txt)
Power8	Yes
e6500	No
e5500	No
Power7	No
Power6	No
PA6T	No
Cell PPU	No
Power5++	No
Power5+	No
Power5	No
PPC970	No
Power4+	No
Power4	No

11.18 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:

```
KERNELBASE

    |-->   64M   <--|
    |               |
    +---------------+    +----------------+---------------+
    |               |....|    |kernel|    |               |
    +---------------+    +----------------+---------------+
    |                         |
    |----->   offset    <-----|

                          kernstart_virt_addr
```

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

11.19 Linux 2.6.x on MPC52xx family

For the latest info, go to https://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 interested 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.

11.20 Hypercall Op-codes (hcalls)

11.20.1 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 privileged[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.

11.20.2 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.

Once control 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.

[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

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

11.20.3 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	

11.20.4 DRC & DRC Indexes

```
DR1                                  Guest
+--+        +------------+         +---------+
|  | <----> |            |         |  User   |
+--+  DRC1  |            |   DRC   |  Space  |
            |    PAPR    |  Index  +---------+
DR2         | Hypervisor |         |         |
+--+        |            | <-----> |  Kernel |
|  | <----> |            |  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.

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

11.20.5 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.

11.20.6 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[Page 377, 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 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 *(startingScm-BlockIndex, 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, num-ScmBlocksToUnbind | 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.

H_SCM_FLUSH

Input: *drcIndex, continue-token*
Out: *continue-token*
Return Value: *H_SUCCESS, H_Parameter, H_P2, H_BUSY*

Given a DRC Index Flush the data to backend NVDIMM device.

The hcall returns H_BUSY when the flush takes longer time and the hcall needs to be issued multiple times in order to be completely serviced. The *continue-token* from the output to be passed in the argument list of subsequent hcalls to the hypervisor until the hcall is completely serviced at which point H_SUCCESS or other error is returned by the hypervisor.

11.20.7 References

11.21 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.

11.21.1 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.

11.21.2 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.

11.21.3 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:

```
0      1                     total_VFs - 1
+------+------+-     -+------+------+
|      |      |  ...  |      |      |
+------+------+-     -+------+------+

                      VF(n) BAR space

0      1                     total_VFs - 1                255
+------+------+-     -+------+------+-      -+------+------+
|      |      |  ...  |      |      |   ...  |      |      |
+------+------+-     -+------+------+-      -+------+------+

                      M64 window

           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:

```
0      1                     total_VFs - 1                255
+------+------+-     -+------+------+-      -+------+------+
|      |      |  ...  |      |      |   ...  |      |      |
+------+------+-     -+------+------+-      -+------+------+

                      VF(n) BAR space + extra

0      1                     total_VFs - 1                255
+------+------+-     -+------+------+-      -+------+------+
|      |      |  ...  |      |      |   ...  |      |      |
+------+------+-     -+------+------+-      -+------+------+

                      M64 window

           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].

11.21.4 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.

11.22 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.

11.22.1 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".

11.22.2 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.

11.22.3 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.

11.22.4 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.

11.22.5 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.

11.22.6 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.

11.22.7 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.

11.22.8 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().

11.23 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.

11.23.1 1. PPC_PTRACE_GETHWDBGINFO

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
#define PPC_DEBUG_FEATURE_DATA_BP_ARCH_31             0x20
```

2. PPC_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 PPC_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. PPC_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..

11.24 Freescale QUICC Engine Firmware Uploading

11.24.1 Revision Information

November 30, 2007: Rev 1.0 - Initial version

11.24.2 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.

11.24.3 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.

11.24.4 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.

11.24.5 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 |                    | Destination Offset | Size of |
|   array   |     Protocol       |   within PRAM      | Operand |
--------------------------------------------------------------|
|     0     | Ethernet           |      0xF8          | 4 bytes |
|           | interworking       |                    |         |
---------------------------------------------------------------
|     4     | ATM                |      0xF8          | 4 bytes |
|           | interworking       |                    |         |
---------------------------------------------------------------
|     8     | PPP                |      0xF8          | 4 bytes |
|           | interworking       |                    |         |
---------------------------------------------------------------
|     12    | Ethernet RX        |      0x22          | 1 byte  |
|           | Distributor Page   |                    |         |
---------------------------------------------------------------
|     16    | ATM Globtal        |      0x28          | 1 byte  |
|           | Params Table       |                    |         |
---------------------------------------------------------------
|     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      | Indicates that prior to each host command   |
|        | push command | given by the application, the software must |
|        |              | assert a special host command (push command)|
|        |              | CECDR = 0x00800000.                         |
|        |              | CECR = 0x01c1000f.                          |
-----------------------------------------------------------------------
```

```
|   1    | UCC ATM       | Indicates that after issuing ATM RX INIT   |
|        | RX INIT       | command, the host must issue another special|
|        | push command  | 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    | Indicates that following the specific host  |
|        | command       | command: "Add/Remove entry in Hash Lookup   |
|        | validation    | Table" used in Interworking setup, the user |
|        |               | must issue another command.                |
|        |               | CECDR = 0xce000003.                        |
|        |               | CECR = 0x01c10f58.                         |
----------------------------------------------------------------------
|   3    | General push  | Indicates that the s/w has to initialize   |
|        | command       | 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  | Indicates that after issuing Ethernet TX   |
|        | command       | 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.                     |
----------------------------------------------------------------------
```

11.24.6 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 calls 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 calls 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;
}
```

11.24.7 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.

11.25 Power Architecture 64-bit Linux system call ABI

11.25.1 syscall

Invocation

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

If PPC_FEATURE2_SCV appears in the AT_HWCAP2 ELF auxiliary vector, the scv 0 instruction is an alternative that may provide better performance, with some differences to calling sequence.

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.

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

Parameters

The system call number is specified in r0.

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

Return value

- For the sc instruction, both a value and an error condition are returned. cr0.SO is the error condition, and r3 is the return value. 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 value (that normally corresponds to errno).
- For the scv 0 instruction, the return value indicates failure if it is -4095..-1 (i.e., it is >= -MAX_ERRNO (-4095) as an unsigned comparison), in which case the error value is the negated return value.

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.

Register preservation rules

Register preservation rules match the ELF ABI calling sequence with some differences.

For the sc instruction, the differences from the ELF ABI are as follows:

Register	Preservation Rules	Purpose
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	

For the scv 0 instruction, the differences from the ELF ABI are as follows:

Register	Preservation Rules	Purpose
r0	Volatile	(System call number.)
r3	Volatile	(Parameter 1, and return value.)
r4-r8	Volatile	(Parameters 2-6.)

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

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.

scv 0 syscalls will always behave as PPC_FEATURE2_HTM_NOSC.

ptrace

When ptracing system calls (PTRACE_SYSCALL), the pt_regs.trap value contains the system call type that can be used to distinguish between sc and scv 0 system calls, and the different register conventions can be accounted for.

If the value of (pt_regs.trap & 0xfff0) is 0xc00 then the system call was performed with the sc instruction, if it is 0x3000 then the system call was performed with the scv 0 instruction.

11.25.2 vsyscall

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

Parameters and return value

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

Stack

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

Register preservation rules

r0	Volatile
cr1, cr5-7	Volatile
lr	Volatile

Invocation

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

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.

11.26 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.

11.26.1 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 processor 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.

11.26.2 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.

11.26.3 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.

11.26.4 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.

11.26.5 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 considered persistent. For example a TM_CAUSE_ALIGNMENT will be persistent while a TM_CAUSE_RESCHED will not.

11.26.6 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.

11.26.7 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.

11.26.8 Kernel implementation

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 its early exception handling.

11.27 Protected Execution Facility

Contents

11.27.1 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.

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).

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.

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.

11.27.2 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.

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.

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.

11.27.3 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.

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.
- H_STATE if the VM is not in a position to switch to secure.

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).
- **H_STATE if the hypervisor could not successfully**
  transition the VM to Secure VM.

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.

11.27.4 References

- Supporting Protected Computing on IBM Power Architecture

11.28 Virtual Accelerator Switchboard (VAS) userspace API

11.28.1 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.

11.28.2 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 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 descriptor (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.

11.28.3 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.

11.28.4 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.

11.28.5 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.

11.28.6 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.

11.28.7 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.

11.28.8 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

11.28.9 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.

11.28.10 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_FAULT_ADDRESS;
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 address;
```

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/doc/power_nx_gzip_um.pdf

11.28.11 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 failed\n");
                return -1;
        }
        memset(&txattr, 0, sizeof(txattr));
        txattr.version = 1;
        txattr.vas_id = -1
        rc = ioctl(fd, VAS_TX_WIN_OPEN,
                        (unsigned long)&txattr);
        if (rc < 0) {
                fprintf(stderr, "ioctl() n %d, error %d\n",
                                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 %d\n",
                                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/libnxz/power-gzip for tests or more use cases.

11.29 VCPU Dispatch Statistics

For Shared Processor LPARs, the POWER Hypervisor maintains a relatively static mapping of the LPAR processors (vcpus) to physical processor chips (representing the "home" node) and tries to always dispatch vcpus on their associated physical processor chip. However, under certain scenarios, vcpus may be dispatched on a different processor chip (away from its home node).

/proc/powerpc/vcpudispatch_stats can be used to obtain statistics related to the vcpu dispatch behavior. Writing '1' to this file enables collecting the statistics, while writing '0' disables the statistics. By default, the DTLB log for each vcpu is processed 50 times a second so as not to miss any entries. This processing frequency can be changed through /proc/powerpc/vcpudispatch_stats_freq.

The statistics themselves are available by reading the procfs file /proc/powerpc/vcpudispatch_stats. Each line in the output corresponds to a vcpu as represented by the first field, followed by 8 numbers.

The first number corresponds to:

1. total vcpu dispatches since the beginning of statistics collection

The next 4 numbers represent vcpu dispatch dispersions:

2. number of times this vcpu was dispatched on the same processor as last time
3. number of times this vcpu was dispatched on a different processor core as last time, but within the same chip
4. number of times this vcpu was dispatched on a different chip
5. number of times this vcpu was dispatches on a different socket/drawer (next numa boundary)

The final 3 numbers represent statistics in relation to the home node of the vcpu:

6. number of times this vcpu was dispatched in its home node (chip)
7. number of times this vcpu was dispatched in a different node
8. number of times this vcpu was dispatched in a node further away (numa distance)

An example output:

```
$ sudo cat /proc/powerpc/vcpudispatch_stats
cpu0 6839 4126 2683 30 0 6821 18 0
```

```
cpu1 2515 1274 1229 12 0 2509 6 0
cpu2 2317 1198 1109 10 0 2312 5 0
cpu3 2259 1165 1088 6 0 2256 3 0
cpu4 2205 1143 1056 6 0 2202 3 0
cpu5 2165 1121 1038 6 0 2162 3 0
cpu6 2183 1127 1050 6 0 2180 3 0
cpu7 2193 1133 1052 8 0 2187 6 0
cpu8 2165 1115 1032 18 0 2156 9 0
cpu9 2301 1252 1033 16 0 2293 8 0
cpu10 2197 1138 1041 18 0 2187 10 0
cpu11 2273 1185 1062 26 0 2260 13 0
cpu12 2186 1125 1043 18 0 2177 9 0
cpu13 2161 1115 1030 16 0 2153 8 0
cpu14 2206 1153 1033 20 0 2196 10 0
cpu15 2163 1115 1032 16 0 2155 8 0
```

In the output above, for vcpu0, there have been 6839 dispatches since statistics were enabled. 4126 of those dispatches were on the same physical cpu as the last time. 2683 were on a different core, but within the same chip, while 30 dispatches were on a different chip compared to its last dispatch.

Also, out of the total of 6839 dispatches, we see that there have been 6821 dispatches on the vcpu's home node, while 18 dispatches were outside its home node, on a neighbouring chip.

11.30 Device DAX

The device-dax interface uses the tail deduplication technique explained in Documentation/mm/vmemmap_dedup.rst

On powerpc, vmemmap deduplication is only used with radix MMU translation. Also with a 64K page size, only the devdax namespace with 1G alignment uses vmemmap deduplication.

With 2M PMD level mapping, we require 32 struct pages and a single 64K vmemmap page can contain 1024 struct pages (64K/sizeof(struct page)). Hence there is no vmemmap deduplication possible.

With 1G PUD level mapping, we require 16384 struct pages and a single 64K vmemmap page can contain 1024 struct pages (64K/sizeof(struct page)). Hence we require 16 64K pages in vmemmap to map the struct page for 1G PUD level mapping.

Here's how things look like on device-dax after the sections are populated::

+-----------+ ---virt_to_page---> +-----------+ mapping to +-----------+ | | | 0 | -------------> | 0 | | |
+-----------+ +-----------+ | | | 1 | -------------> | 1 | | | +-----------+ +-----------+ | | | 2 | ----------------^
^ ^ ^ ^ ^ | | +-----------+ | | | | | | | | 3 | ------------------+ | | | | | | +-----------+ | | | | | | | 4 | ----------
----------+ | | | | PUD | +-----------+ | | | | level | | . | ----------------------+ | | | mapping | +-----------+
| | | | | . | -----------------------+ | | | +-----------+ | | | | 15 | -------------------------+ | | +-----------+ | | | |
| | +-----------+

With 4K page size, 2M PMD level mapping requires 512 struct pages and a single 4K vmemmap page contains 64 struct pages(4K/sizeof(struct page)). Hence we require 8 4K pages in vmemmap to map the struct page for 2M pmd level mapping.

Here's how things look like on device-dax after the sections are populated:

```
+-----------+ ---virt_to_page---> +-----------+   mapping to   +-----------+
|           |                     |     0     | -------------> |     0     |
|           |                     +-----------+                +-----------+
|           |                     |     1     | -------------> |     1     |
|           |                     +-----------+                +-----------+
|           |                     |     2     | ----------------^ ^ ^ ^ ^ ^
|           |                     +-----------+                   | | | | |
|           |                     |     3     | ------------------+ | | | |
|           |                     +-----------+                     | | | |
|           |                     |     4     | --------------------+ | | |
|    PMD    |                     +-----------+                       | | |
|   level   |                     |     5     | ----------------------+ | |
|  mapping  |                     +-----------+                         | |
|           |                     |     6     | ------------------------+ |
|           |                     +-----------+                           |
|           |                     |     7     | --------------------------+
|           |                     +-----------+
|           |
|           |
|           |
+-----------+
```

With 1G PUD level mapping, we require 262144 struct pages and a single 4K vmemmap page can contain 64 struct pages (4K/sizeof(struct page)). Hence we require 4096 4K pages in vmemmap to map the struct pages for 1G PUD level mapping.

Here's how things look like on device-dax after the sections are populated:

```
+-----------+ ---virt_to_page---> +-----------+   mapping to   +-----------+
|           |                     |     0     | -------------> |     0     |
|           |                     +-----------+                +-----------+
|           |                     |     1     | -------------> |     1     |
|           |                     +-----------+                +-----------+
|           |                     |     2     | ----------------^ ^ ^ ^ ^ ^
|           |                     +-----------+                   | | | | |
|           |                     |     3     | ------------------+ | | | |
|           |                     +-----------+                     | | | |
|           |                     |     4     | --------------------+ | | |
|    PUD    |                     +-----------+                       | | |
|   level   |                     |     .     | ----------------------+ | |
|  mapping  |                     +-----------+                         | |
|           |                     |     .     | ------------------------+ |
|           |                     +-----------+                           |
|           |                     |   4095    | --------------------------+
|           |                     +-----------+
|           |
|           |
|           |
+-----------+
```

11.31 Feature status on powerpc architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	ok
core	eBPF-JIT	HAVE_EBPF_JIT	ok
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok
core	thread-info-in-task	THREAD_INFO_IN_TASK	ok
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	ok
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	ok
debug	KASAN	HAVE_ARCH_KASAN	ok
debug	kcov	ARCH_HAS_KCOV	ok
debug	kgdb	HAVE_ARCH_KGDB	ok
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok
debug	kprobes	HAVE_KPROBES	ok
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	ok
debug	kretprobes	HAVE_KRETPROBES	ok
debug	optprobes	HAVE_OPTPROBES	ok
debug	stackprotector	HAVE_STACKPROTECTOR	ok
debug	uprobes	ARCH_SUPPORTS_UPROBES	ok
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	TODO
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	ok
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	ok
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok
perf	perf-regs	HAVE_PERF_REGS	ok
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	ok
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	ok
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	ok
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	ok
time	clockevents	!LEGACY_TIMER_TICK	ok
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	ok
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	ok
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	ok
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	TODO
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	ok
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	ok
vm	ioremap_prot	HAVE_IOREMAP_PROT	ok
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	ok

CHAPTER

TWELVE

RISC-V ARCHITECTURE

12.1 ACPI on RISC-V

The ISA string parsing rules for ACPI are defined by Version ASCIIDOC Conversion, 12/2022 of the RISC-V specifications, as defined by tag "riscv-isa-release-1239329-2023-05-23" (commit 1239329)

12.2 RISC-V Kernel Boot Requirements and Constraints

Author
Alexandre Ghiti <alexghiti@rivosinc.com>

Date
23 May 2023

This document describes what the RISC-V kernel expects from bootloaders and firmware, and also the constraints that any developer must have in mind when touching the early boot process. For the purposes of this document, the `early boot process` refers to any code that runs before the final virtual mapping is set up.

12.2.1 Pre-kernel Requirements and Constraints

The RISC-V kernel expects the following of bootloaders and platform firmware:

Register state

The RISC-V kernel expects:

- `$a0` to contain the hartid of the current core.
- `$a1` to contain the address of the devicetree in memory.

CSR state

The RISC-V kernel expects:

- `$satp = 0`: the MMU, if present, must be disabled.

Reserved memory for resident firmware

The RISC-V kernel must not map any resident memory, or memory protected with PMPs, in the direct mapping, so the firmware must correctly mark those regions as per the devicetree specification and/or the UEFI specification.

Kernel location

The RISC-V kernel expects to be placed at a PMD boundary (2MB aligned for rv64 and 4MB aligned for rv32). Note that the EFI stub will physically relocate the kernel if that's not the case.

Hardware description

The firmware can pass either a devicetree or ACPI tables to the RISC-V kernel.

The devicetree is either passed directly to the kernel from the previous stage using the `$a1` register, or when booting with UEFI, it can be passed using the EFI configuration table.

The ACPI tables are passed to the kernel using the EFI configuration table. In this case, a tiny devicetree is still created by the EFI stub. Please refer to "EFI stub and devicetree" section below for details about this devicetree.

Kernel entry

On SMP systems, there are 2 methods to enter the kernel:

- `RISCV_BOOT_SPINWAIT`: the firmware releases all harts in the kernel, one hart wins a lottery and executes the early boot code while the other harts are parked waiting for the initialization to finish. This method is mostly used to support older firmwares without SBI HSM extension and M-mode RISC-V kernel.
- `Ordered booting`: the firmware releases only one hart that will execute the initialization phase and then will start all other harts using the SBI HSM extension. The ordered booting method is the preferred booting method for booting the RISC-V kernel because it can support CPU hotplug and kexec.

UEFI

UEFI memory map

When booting with UEFI, the RISC-V kernel will use only the EFI memory map to populate the system memory.

The UEFI firmware must parse the subnodes of the `/reserved-memory` devicetree node and abide by the devicetree specification to convert the attributes of those subnodes (`no-map` and `reusable`) into their correct EFI equivalent (refer to section "3.5.4 /reserved-memory and UEFI" of the devicetree specification v0.4-rc1).

RISCV_EFI_BOOT_PROTOCOL

When booting with UEFI, the EFI stub requires the boot hartid in order to pass it to the RISC-V kernel in `$a1`. The EFI stub retrieves the boot hartid using one of the following methods:

- `RISCV_EFI_BOOT_PROTOCOL` (**preferred**).
- `boot-hartid` devicetree subnode (**deprecated**).

Any new firmware must implement `RISCV_EFI_BOOT_PROTOCOL` as the devicetree based approach is deprecated now.

12.2.2 Early Boot Requirements and Constraints

The RISC-V kernel's early boot process operates under the following constraints:

EFI stub and devicetree

When booting with UEFI, the devicetree is supplemented (or created) by the EFI stub with the same parameters as arm64 which are described at the paragraph "UEFI kernel support on ARM" in *The Unified Extensible Firmware Interface (UEFI)*.

Virtual mapping installation

The installation of the virtual mapping is done in 2 steps in the RISC-V kernel:

1. `setup_vm()` installs a temporary kernel mapping in `early_pg_dir` which allows discovery of the system memory. Only the kernel text/data are mapped at this point. When establishing this mapping, no allocation can be done (since the system memory is not known yet), so `early_pg_dir` page table is statically allocated (using only one table for each level).
2. `setup_vm_final()` creates the final kernel mapping in `swapper_pg_dir` and takes advantage of the discovered system memory to create the linear mapping. When establishing this mapping, the kernel can allocate memory but cannot access it directly (since the direct mapping is not present yet), so it uses temporary mappings in the fixmap region to be able to access the newly allocated page table levels.

For `virt_to_phys()` and `phys_to_virt()` to be able to correctly convert direct mapping addresses to physical addresses, they need to know the start of the DRAM. This happens after step 1, right before step 2 installs the direct mapping (see `setup_bootmem()` function in arch/riscv/mm/init.c). Any usage of those macros before the final virtual mapping is installed must be carefully examined.

Devicetree mapping via fixmap

As the `reserved_mem` array is initialized with virtual addresses established by `setup_vm()`, and used with the mapping established by `setup_vm_final()`, the RISC-V kernel uses the fixmap region to map the devicetree. This ensures that the devicetree remains accessible by both virtual mappings.

Pre-MMU execution

A few pieces of code need to run before even the first virtual mapping is established. These are the installation of the first virtual mapping itself, patching of early alternatives and the early parsing of the kernel command line. That code must be very carefully compiled as:

- `-fno-pie`: This is needed for relocatable kernels which use `-fPIE`, since otherwise, any access to a global symbol would go through the GOT which is only relocated virtually.
- `-mcmodel=medany`: Any access to a global symbol must be PC-relative to avoid any relocations to happen before the MMU is setup.
- *all* instrumentation must also be disabled (that includes KASAN, ftrace and others).

As using a symbol from a different compilation unit requires this unit to be compiled with those flags, we advise, as much as possible, not to use external symbols.

12.3 Boot image header in RISC-V Linux

Author
Atish Patra <atish.patra@wdc.com>

Date
20 May 2019

This document only describes the boot image header details for RISC-V Linux.

The following 64-byte header is present in decompressed Linux kernel image:

```
u32 code0;                /* Executable code */
u32 code1;                /* Executable code */
u64 text_offset;          /* Image load offset, little endian */
u64 image_size;           /* Effective Image size, little endian */
u64 flags;                /* kernel flags, little endian */
u32 version;              /* Version of this header */
u32 res1 = 0;             /* Reserved */
u64 res2 = 0;             /* Reserved */
u64 magic = 0x5643534952; /* Magic number, little endian, "RISCV" */
u32 magic2 = 0x05435352;  /* Magic number 2, little endian, "RSC\x05" */
u32 res3;                 /* Reserved for PE COFF offset */
```

This header format is compliant with PE/COFF header and largely inspired from ARM64 header. Thus, both ARM64 & RISC-V header can be combined into one common header in future.

12.3.1 Notes

- This header is also reused to support EFI stub for RISC-V. EFI specification needs PE/COFF image header in the beginning of the kernel image in order to load it as an EFI application. In order to support EFI stub, code0 is replaced with "MZ" magic string and res3(at offset 0x3c) points to the rest of the PE/COFF header.
- version field indicate header version number

Bits 0:15	Minor version
Bits 16:31	Major version

  This preserves compatibility across newer and older version of the header. The current version is defined as 0.2.
- The "magic" field is deprecated as of version 0.2. In a future release, it may be removed. This originally should have matched up with the ARM64 header "magic" field, but unfortunately does not. The "magic2" field replaces it, matching up with the ARM64 header.
- In current header, the flags field has only one field.

Bit 0	Kernel endianness. 1 if BE, 0 if LE.

- Image size is mandatory for boot loader to load kernel image. Booting will fail otherwise.

12.4 Virtual Memory Layout on RISC-V Linux

Author
Alexandre Ghiti <alex@ghiti.fr>

Date
12 February 2021

This document describes the virtual memory layout used by the RISC-V Linux Kernel.

12.4.1 RISC-V Linux Kernel 32bit

RISC-V Linux Kernel SV32

TODO

12.4.2 RISC-V Linux Kernel 64bit

The RISC-V privileged architecture document states that the 64bit addresses "must have bits 63–48 all equal to bit 47, or else a page-fault exception will occur.": that splits the virtual address space into 2 halves separated by a very big hole, the lower half is where the userspace resides, the upper half is where the RISC-V Linux Kernel resides.

RISC-V Linux Kernel SV39

```
===========================================================================================
    Start addr    |   Offset   |     End addr     |  Size   | VM area␣
↪description
===========================================================================================
                  |            |                  |         |
 0000000000000000 |    0       | 0000003fffffffff |  256 GB | user-space␣
↪virtual memory, different per mm
__________________|____________|__________________|_________|__________________
↪_________________________________________
                  |            |                  |         |
 0000004000000000 | +256    GB | ffffffbfffffffff | ~16M TB | ... huge, almost␣
↪64 bits wide hole of non-canonical
                  |            |                  |         |     virtual␣
↪memory addresses up to the -256 GB
                  |            |                  |         |     starting␣
↪offset of kernel mappings.
__________________|____________|__________________|_________|__________________
↪_________________________________________
                                                            |
                                                            | Kernel-space␣
↪virtual memory, shared between all processes:
____________________________________________________________|__________________
↪_________________________________________
                  |            |                  |         |
 ffffffc6fea00000 | -228    GB | ffffffc6feffffff |    6 MB | fixmap
 ffffffc6ff000000 | -228    GB | ffffffc6ffffffff |   16 MB | PCI io
 ffffffc700000000 | -228    GB | ffffffc7ffffffff |    4 GB | vmemmap
 ffffffc800000000 | -224    GB | ffffffd7ffffffff |   64 GB | vmalloc/ioremap␣
↪space
 ffffffd800000000 | -160    GB | fffffff6ffffffff |  124 GB | direct mapping␣
↪of all physical memory
 fffffff700000000 |  -36    GB | fffffffeffffffff |   32 GB | kasan
__________________|____________|__________________|_________|__________________
↪__________________________________________
                                                            |
                                                            |
____________________________________________________________|__________________
↪__________________________________________
                  |            |                  |         |
 ffffffff00000000 |   -4    GB | ffffffff7fffffff |    2 GB | modules, BPF
 ffffffff80000000 |   -2    GB | ffffffffffffffff |    2 GB | kernel
```

```
___________________|____________|___________________|_________|___________________
↪___________________________________________
```

RISC-V Linux Kernel SV48

```
========================================================================================
    Start addr    |   Offset   |     End addr     |  Size   | VM area␣
↪description
========================================================================================
                  |            |                  |         |
 0000000000000000 |    0       | 00007fffffffffff |  128 TB | user-space␣
↪virtual memory, different per mm
__________________|____________|__________________|_________|___________________
↪___________________________________________
                  |            |                  |         |
 0000800000000000 | +128    TB | ffff7fffffffffff | ~16M TB | ... huge,␣
↪almost 64 bits wide hole of non-canonical
                  |            |                  |         | virtual memory␣
↪addresses up to the -128 TB
                  |            |                  |         | starting offset␣
↪of kernel mappings.
__________________|____________|__________________|_________|___________________
↪___________________________________________
                                                            |
                                                            | Kernel-space␣
↪virtual memory, shared between all processes:
____________________________________________________________|___________________
↪___________________________________________
                  |            |                  |         |
 ffff8d7ffea00000 |  -114.5 TB | ffff8d7ffeffffff |    6 MB | fixmap
 ffff8d7fff000000 |  -114.5 TB | ffff8d7fffffffff |   16 MB | PCI io
 ffff8d8000000000 |  -114.5 TB | ffff8f7fffffffff |    2 TB | vmemmap
 ffff8f8000000000 |  -112.5 TB | ffffaf7fffffffff |   32 TB | vmalloc/ioremap␣
↪space
 ffffaf8000000000 |  -80.5   TB | ffffef7fffffffff |   64 TB | direct mapping␣
↪of all physical memory
 ffffef8000000000 |  -16.5   TB | fffffffeffffffff | 16.5 TB | kasan
__________________|____________|__________________|_________|___________________
↪____________________________________________
                                                            |
                                                            | Identical␣
↪layout to the 39-bit one from here on:
____________________________________________________________|___________________
↪____________________________________________
                  |            |                  |         |
 ffffffff00000000 |   -4    GB | ffffffff7fffffff |    2 GB | modules, BPF
 ffffffff80000000 |   -2    GB | ffffffffffffffff |    2 GB | kernel
__________________|____________|__________________|_________|___________________
↪____________________________________________
```

RISC-V Linux Kernel SV57

```
=========================================================================================
    Start addr    |   Offset   |     End addr     |  Size   | VM area␣
↪description
=========================================================================================
                  |            |                  |         |
 0000000000000000 |   0        | 00ffffffffffffff |   64 PB | user-space␣
↪virtual memory, different per mm
__________________|____________|__________________|_________|_________________
↪__________________________________________
                  |            |                  |         |
 0100000000000000 | +64     PB | feffffffffffffff | ~16K PB | ... huge,␣
↪almost 64 bits wide hole of non-canonical
                  |            |                  |         | virtual memory␣
↪addresses up to the -64 PB
                  |            |                  |         | starting offset␣
↪of kernel mappings.
__________________|____________|__________________|_________|_________________
↪__________________________________________
                                                            |
                                                            | Kernel-space␣
↪virtual memory, shared between all processes:
____________________________________________________________|_________________
↪__________________________________________
                  |            |                  |         |
 ff1bfffffea00000 | -57     PB | ff1bfffffeffffff |    6 MB | fixmap
 ff1bffffff000000 | -57     PB | ff1bffffffffffff |   16 MB | PCI io
 ff1c000000000000 | -57     PB | ff1fffffffffffff |    1 PB | vmemmap
 ff20000000000000 | -56     PB | ff5fffffffffffff |   16 PB | vmalloc/ioremap␣
↪space
 ff60000000000000 | -40     PB | ffdeffffffffffff |   32 PB | direct mapping␣
↪of all physical memory
 ffdf000000000000 |  -8     PB | fffffffeffffffff |    8 PB | kasan
__________________|____________|__________________|_________|_________________
↪___________________________________________
                                                            |
                                                            | Identical␣
↪layout to the 39-bit one from here on:
____________________________________________________________|_________________
↪___________________________________________
                  |            |                  |         |
 ffffffff00000000 |  -4     GB | ffffffff7fffffff |    2 GB | modules, BPF
 ffffffff80000000 |  -2     GB | ffffffffffffffff |    2 GB | kernel
__________________|____________|__________________|_________|_________________
↪___________________________________________
```

Userspace VAs

To maintain compatibility with software that relies on the VA space with a maximum of 48 bits the kernel will, by default, return virtual addresses to userspace from a 48-bit range (sv48). This default behavior is achieved by passing 0 into the hint address parameter of mmap. On CPUs with an address space smaller than sv48, the CPU maximum supported address space will be the default.

Software can "opt-in" to receiving VAs from another VA space by providing a hint address to mmap. A hint address passed to mmap will cause the largest address space that fits entirely into the hint to be used, unless there is no space left in the address space. If there is no space available in the requested address space, an address in the next smallest available address space will be returned.

For example, in order to obtain 48-bit VA space, a hint address greater than `1 << 47` must be provided. Note that this is 47 due to sv48 userspace ending at `1 << 47` and the addresses beyond this are reserved for the kernel. Similarly, to obtain 57-bit VA space addresses, a hint address greater than or equal to `1 << 56` must be provided.

12.5 RISC-V Hardware Probing Interface

The RISC-V hardware probing interface is based around a single syscall, which is defined in <asm/hwprobe.h>:

```
struct riscv_hwprobe {
    __s64 key;
    __u64 value;
};

long sys_riscv_hwprobe(struct riscv_hwprobe *pairs, size_t pair_count,
                       size_t cpu_count, cpu_set_t *cpus,
                       unsigned int flags);
```

The arguments are split into three groups: an array of key-value pairs, a CPU set, and some flags. The key-value pairs are supplied with a count. Userspace must prepopulate the key field for each element, and the kernel will fill in the value if the key is recognized. If a key is unknown to the kernel, its key field will be cleared to -1, and its value set to 0. The CPU set is defined by CPU_SET(3). For value-like keys (eg. vendor/arch/impl), the returned value will be only be valid if all CPUs in the given set have the same value. Otherwise -1 will be returned. For boolean-like keys, the value returned will be a logical AND of the values for the specified CPUs. Usermode can supply NULL for cpus and 0 for cpu_count as a shortcut for all online CPUs. There are currently no flags, this value must be zero for future compatibility.

On success 0 is returned, on failure a negative error code is returned.

The following keys are defined:

- `RISCV_HWPROBE_KEY_MVENDORID`: Contains the value of `mvendorid`, as defined by the RISC-V privileged architecture specification.
- `RISCV_HWPROBE_KEY_MARCHID`: Contains the value of `marchid`, as defined by the RISC-V privileged architecture specification.

- `RISCV_HWPROBE_KEY_MIMPLID`: Contains the value of `mimplid`, as defined by the RISC-V privileged architecture specification.
- `RISCV_HWPROBE_KEY_BASE_BEHAVIOR`: A bitmask containing the base user-visible behavior that this kernel supports. The following base user ABIs are defined:
  - `RISCV_HWPROBE_BASE_BEHAVIOR_IMA`: Support for rv32ima or rv64ima, as defined by version 2.2 of the user ISA and version 1.10 of the privileged ISA, with the following known exceptions (more exceptions may be added, but only if it can be demonstrated that the user ABI is not broken):
    - The `fence.i` instruction cannot be directly executed by userspace programs (it may still be executed in userspace via a kernel-controlled mechanism such as the vDSO).
- `RISCV_HWPROBE_KEY_IMA_EXT_0`: A bitmask containing the extensions that are compatible with the `RISCV_HWPROBE_BASE_BEHAVIOR_IMA`: base system behavior.
  - `RISCV_HWPROBE_IMA_FD`: The F and D extensions are supported, as defined by commit cd20cee ("FMIN/FMAX now implement minimumNumber/maximumNumber, not minNum/maxNum") of the RISC-V ISA manual.
  - `RISCV_HWPROBE_IMA_C`: The C extension is supported, as defined by version 2.2 of the RISC-V ISA manual.
  - `RISCV_HWPROBE_IMA_V`: The V extension is supported, as defined by version 1.0 of the RISC-V Vector extension manual.
  - **`RISCV_HWPROBE_EXT_ZBA`: The Zba address generation extension is**
    supported, as defined in version 1.0 of the Bit-Manipulation ISA extensions.
  - **`RISCV_HWPROBE_EXT_ZBB`: The Zbb extension is supported, as defined**
    in version 1.0 of the Bit-Manipulation ISA extensions.
  - **`RISCV_HWPROBE_EXT_ZBS`: The Zbs extension is supported, as defined**
    in version 1.0 of the Bit-Manipulation ISA extensions.
- `RISCV_HWPROBE_KEY_CPUPERF_0`: A bitmask that contains performance information about the selected set of processors.
  - `RISCV_HWPROBE_MISALIGNED_UNKNOWN`: The performance of misaligned accesses is unknown.
  - `RISCV_HWPROBE_MISALIGNED_EMULATED`: Misaligned accesses are emulated via software, either in or below the kernel. These accesses are always extremely slow.
  - `RISCV_HWPROBE_MISALIGNED_SLOW`: Misaligned accesses are slower than equivalent byte accesses. Misaligned accesses may be supported directly in hardware, or trapped and emulated by software.
  - `RISCV_HWPROBE_MISALIGNED_FAST`: Misaligned accesses are faster than equivalent byte accesses.
  - `RISCV_HWPROBE_MISALIGNED_UNSUPPORTED`: Misaligned accesses are not supported at all and will generate a misaligned address fault.

12.6 arch/riscv maintenance guidelines for developers

12.6.1 Overview

The RISC-V instruction set architecture is developed in the open: in-progress drafts are available for all to review and to experiment with implementations. New module or extension drafts can change during the development process - sometimes in ways that are incompatible with previous drafts. This flexibility can present a challenge for RISC-V Linux maintenance. Linux maintainers disapprove of churn, and the Linux development process prefers well-reviewed and tested code over experimental code. We wish to extend these same principles to the RISC-V-related code that will be accepted for inclusion in the kernel.

12.6.2 Patchwork

RISC-V has a patchwork instance, where the status of patches can be checked:

> https://patchwork.kernel.org/project/linux-riscv/list/

If your patch does not appear in the default view, the RISC-V maintainers have likely either requested changes, or expect it to be applied to another tree.

Automation runs against this patchwork instance, building/testing patches as they arrive. The automation applies patches against the current HEAD of the RISC-V *for-next* and *fixes* branches, depending on whether the patch has been detected as a fix. Failing those, it will use the RISC-V *master* branch. The exact commit to which a series has been applied will be noted on patchwork. Patches for which any of the checks fail are unlikely to be applied and in most cases will need to be resubmitted.

12.6.3 Submit Checklist Addendum

We'll only accept patches for new modules or extensions if the specifications for those modules or extensions are listed as being unlikely to be incompatibly changed in the future. For specifications from the RISC-V foundation this means "Frozen" or "Ratified", for the UEFI forum specifications this means a published ECR. (Developers may, of course, maintain their own Linux kernel trees that contain code for any draft extensions that they wish.)

Additionally, the RISC-V specification allows implementers to create their own custom extensions. These custom extensions aren't required to go through any review or ratification process by the RISC-V Foundation. To avoid the maintenance complexity and potential performance impact of adding kernel code for implementor-specific RISC-V extensions, we'll only consider patches for extensions that either:

- Have been officially frozen or ratified by the RISC-V Foundation, or
- Have been implemented in hardware that is widely available, per standard Linux practice.

(Implementers, may, of course, maintain their own Linux kernel trees containing code for any custom extensions that they wish.)

12.7 RISC-V Linux User ABI

12.7.1 ISA string ordering in /proc/cpuinfo

The canonical order of ISA extension names in the ISA string is defined in chapter 27 of the unprivileged specification. The specification uses vague wording, such as should, when it comes to ordering, so for our purposes the following rules apply:

1. Single-letter extensions come first, in canonical order. The canonical order is "IMAFDQLCBKJTPVH".
2. All multi-letter extensions will be separated from other extensions by an underscore.
3. Additional standard extensions (starting with 'Z') will be sorted after single-letter extensions and before any higher-privileged extensions.
4. For additional standard extensions, the first letter following the 'Z' conventionally indicates the most closely related alphabetical extension category. If multiple 'Z' extensions are named, they will be ordered first by category, in canonical order, as listed above, then alphabetically within a category.
5. Standard supervisor-level extensions (starting with 'S') will be listed after standard unprivileged extensions. If multiple supervisor-level extensions are listed, they will be ordered alphabetically.
6. Standard machine-level extensions (starting with 'Zxm') will be listed after any lower-privileged, standard extensions. If multiple machine-level extensions are listed, they will be ordered alphabetically.
7. Non-standard extensions (starting with 'X') will be listed after all standard extensions. If multiple non-standard extensions are listed, they will be ordered alphabetically.

An example string following the order is:

```
rv64imadc_zifoo_zigoo_zafoo_sbar_scar_zxmbaz_xqux_xrux
```

12.7.2 Misaligned accesses

Misaligned accesses are supported in userspace, but they may perform poorly.

12.8 Vector Extension Support for RISC-V Linux

This document briefly outlines the interface provided to userspace by Linux in order to support the use of the RISC-V Vector Extension.

12.8.1 1. prctl() Interface

Two new prctl() calls are added to allow programs to manage the enablement status for the use of Vector in userspace. The intended usage guideline for these interfaces is to give init systems a way to modify the availability of V for processes running under its domain. Calling these interfaces is not recommended in libraries routines because libraries should not override policies configured from the parant process. Also, users must noted that these interfaces are not portable to non-Linux, nor non-RISC-V environments, so it is discourage to use in a portable code. To get the availability of V in an ELF program, please read `COMPAT_HWCAP_ISA_V` bit of `ELF_HWCAP` in the auxiliary vector.

- prctl(PR_RISCV_V_SET_CONTROL, unsigned long arg)

  Sets the Vector enablement status of the calling thread, where the control argument consists of two 2-bit enablement statuses and a bit for inheritance mode. Other threads of the calling process are unaffected.

  Enablement status is a tri-state value each occupying 2-bit of space in the control argument:

  - `PR_RISCV_V_VSTATE_CTRL_DEFAULT`: Use the system-wide default enablement status on execve(). The system-wide default setting can be controlled via sysctl interface (see sysctl section below).
  - `PR_RISCV_V_VSTATE_CTRL_ON`: Allow Vector to be run for the thread.
  - `PR_RISCV_V_VSTATE_CTRL_OFF`: Disallow Vector. Executing Vector instructions under such condition will trap and casuse the termination of the thread.

  arg: The control argument is a 5-bit value consisting of 3 parts, and accessed by 3 masks respectively.

  The 3 masks, PR_RISCV_V_VSTATE_CTRL_CUR_MASK, PR_RISCV_V_VSTATE_CTRL_NEXT_MASK, and PR_RISCV_V_VSTATE_CTRL_INHERIT represents bit[1:0], bit[3:2], and bit[4]. bit[1:0] accounts for the enablement status of current thread, and the setting at bit[3:2] takes place at next execve(). bit[4] defines the inheritance mode of the setting in bit[3:2].

  - `PR_RISCV_V_VSTATE_CTRL_CUR_MASK`: bit[1:0]: Account for the Vector enablement status for the calling thread. The calling thread is not able to turn off Vector once it has been enabled. The prctl() call fails with EPERM if the value in this mask is PR_RISCV_V_VSTATE_CTRL_OFF but the current enablement status is not off. Setting PR_RISCV_V_VSTATE_CTRL_DEFAULT here takes no effect but to set back the original enablement status.
  - `PR_RISCV_V_VSTATE_CTRL_NEXT_MASK`: bit[3:2]: Account for the Vector enablement setting for the calling thread at the next execve() system call. If PR_RISCV_V_VSTATE_CTRL_DEFAULT is used in this mask, then the enablement status will be decided by the system-wide enablement status when execve() happen.
  - `PR_RISCV_V_VSTATE_CTRL_INHERIT`: bit[4]: the inheritance mode for the setting at PR_RISCV_V_VSTATE_CTRL_NEXT_MASK. If the bit is set then the following execve() will not clear the setting in both PR_RISCV_V_VSTATE_CTRL_NEXT_MASK and PR_RISCV_V_VSTATE_CTRL_INHERIT. This setting persists across changes in the system-wide default value.

Return value:

- 0 on success;
- EINVAL: Vector not supported, invalid enablement status for current or next mask;
- EPERM: Turning off Vector in PR_RISCV_V_VSTATE_CTRL_CUR_MASK if Vector was enabled for the calling thread.

On success:

- A valid setting for PR_RISCV_V_VSTATE_CTRL_CUR_MASK takes place immediately. The enablement status specified in PR_RISCV_V_VSTATE_CTRL_NEXT_MASK happens at the next execve() call, or all following execve() calls if PR_RISCV_V_VSTATE_CTRL_INHERIT bit is set.
- Every successful call overwrites a previous setting for the calling thread.

- prctl(PR_RISCV_V_GET_CONTROL)

  Gets the same Vector enablement status for the calling thread. Setting for next execve() call and the inheritance bit are all OR-ed together.

  Note that ELF programs are able to get the availability of V for itself by reading `COMPAT_HWCAP_ISA_V` bit of `ELF_HWCAP` in the auxiliary vector.

  Return value:

  - a nonnegative value on success;
  - EINVAL: Vector not supported.

12.8.2 2. System runtime configuration (sysctl)

To mitigate the ABI impact of expansion of the signal stack, a policy mechanism is provided to the administrators, distro maintainers, and developers to control the default Vector enablement status for userspace processes in form of sysctl knob:

- /proc/sys/abi/riscv_v_default_allow

  Writing the text representation of 0 or 1 to this file sets the default system enablement status for new starting userspace programs. Valid values are:

  - 0: Do not allow Vector code to be executed as the default for new processes.
  - 1: Allow Vector code to be executed as the default for new processes.

  Reading this file returns the current system default enablement status.

  At every execve() call, a new enablement status of the new process is set to the system default, unless:

  - PR_RISCV_V_VSTATE_CTRL_INHERIT is set for the calling process, and the setting in PR_RISCV_V_VSTATE_CTRL_NEXT_MASK is not PR_RISCV_V_VSTATE_CTRL_DEFAULT. Or,
  - The setting in PR_RISCV_V_VSTATE_CTRL_NEXT_MASK is not PR_RISCV_V_VSTATE_CTRL_DEFAULT.

Modifying the system default enablement status does not affect the enablement status of any existing process of thread that do not make an execve() call.

12.8.3 3. Vector Register State Across System Calls

As indicated by version 1.0 of the V extension [1], vector registers are clobbered by system calls.

1: https://github.com/riscv/riscv-v-spec/blob/master/calling-convention.adoc

12.9 Feature status on riscv architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	ok
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok
core	thread-info-in-task	THREAD_INFO_IN_TASK	ok
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	ok
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	ok
debug	KASAN	HAVE_ARCH_KASAN	ok
debug	kcov	ARCH_HAS_KCOV	ok
debug	kgdb	HAVE_ARCH_KGDB	ok
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok
debug	kprobes	HAVE_KPROBES	ok
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	ok
debug	kretprobes	HAVE_KRETPROBES	ok
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	ok
debug	uprobes	ARCH_SUPPORTS_UPROBES	ok
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	ok
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	ok
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	TODO
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok
perf	perf-regs	HAVE_PERF_REGS	ok
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	ok
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	TODO
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	ok
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	ok
time	clockevents	!LEGACY_TIMER_TICK	ok
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	ok
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	ok

Table 1 – continued fron

Subsystem	Feature	Kconfig	Status
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	TODO
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	TODO
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	ok
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	ok
vm	ioremap_prot	HAVE_IOREMAP_PROT	TODO
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	ok

CHAPTER

THIRTEEN

S390 ARCHITECTURE

13.1 Linux for S/390 and zSeries

Common Device Support (CDS) Device Driver I/O Support Routines

Authors:

- Ingo Adlung
- Cornelia Huck

Copyright, IBM Corp. 1999-2002

13.1.1 Introduction

This document describes the common device support routines for Linux/390. Different than other hardware architectures, ESA/390 has defined a unified I/O access method. This gives relief to the device drivers as they don't have to deal with different bus types, polling versus interrupt processing, shared versus non-shared interrupt processing, DMA versus port I/O (PIO), and other hardware features more. However, this implies that either every single device driver needs to implement the hardware I/O attachment functionality itself, or the operating system provides for a unified method to access the hardware, providing all the functionality that every single device driver would have to provide itself.

The document does not intend to explain the ESA/390 hardware architecture in every detail.This information can be obtained from the ESA/390 Principles of Operation manual (IBM Form. No. SA22-7201).

In order to build common device support for ESA/390 I/O interfaces, a functional layer was introduced that provides generic I/O access methods to the hardware.

The common device support layer comprises the I/O support routines defined below. Some of them implement common Linux device driver interfaces, while some of them are ESA/390 platform specific.

Note:
In order to write a driver for S/390, you also need to look into the interface described in *S/390 driver model interfaces*.

Note for porting drivers from 2.4:

The major changes are:

- The functions use a ccw_device instead of an irq (subchannel).

- All drivers must define a ccw_driver (see driver-model.txt) and the associated functions.
- request_irq() and free_irq() are no longer done by the driver.
- The oper_handler is (kindof) replaced by the probe() and set_online() functions of the ccw_driver.
- The not_oper_handler is (kindof) replaced by the remove() and set_offline() functions of the ccw_driver.
- The channel device layer is gone.
- The interrupt handlers must be adapted to use a ccw_device as argument. Moreover, they don't return a devstat, but an irb.
- Before initiating an io, the options must be set via ccw_device_set_options().
- Instead of calling read_dev_chars()/read_conf_data(), the driver issues the channel program and handles the interrupt itself.

ccw_device_get_ciw()
get commands from extended sense data.

ccw_device_start(), ccw_device_start_timeout(), ccw_device_start_key(), ccw_device_start_key_timeout()
initiate an I/O request.

ccw_device_resume()
resume channel program execution.

ccw_device_halt()
terminate the current I/O request processed on the device.

do_IRQ()
generic interrupt routine. This function is called by the interrupt entry routine whenever an I/O interrupt is presented to the system. The do_IRQ() routine determines the interrupt status and calls the device specific interrupt handler according to the rules (flags) defined during I/O request initiation with do_IO().

The next chapters describe the functions other than do_IRQ() in more details. The do_IRQ() interface is not described, as it is called from the Linux/390 first level interrupt handler only and does not comprise a device driver callable interface. Instead, the functional description of do_IO() also describes the input to the device specific interrupt handler.

Note:
All explanations apply also to the 64 bit architecture s390x.

13.1.2 Common Device Support (CDS) for Linux/390 Device Drivers

General Information

The following chapters describe the I/O related interface routines the Linux/390 common device support (CDS) provides to allow for device specific driver implementations on the IBM ESA/390 hardware platform. Those interfaces intend to provide the functionality required by every device driver implementation to allow to drive a specific hardware device on the ESA/390 platform. Some of the interface routines are specific to Linux/390 and some of them can be found on other Linux platforms implementations too. Miscellaneous function prototypes, data declarations, and macro definitions can be found in the architecture specific C header file linux/arch/s390/include/asm/irq.h.

Overview of CDS interface concepts

Different to other hardware platforms, the ESA/390 architecture doesn't define interrupt lines managed by a specific interrupt controller and bus systems that may or may not allow for shared interrupts, DMA processing, etc.. Instead, the ESA/390 architecture has implemented a so called channel subsystem, that provides a unified view of the devices physically attached to the systems. Though the ESA/390 hardware platform knows about a huge variety of different peripheral attachments like disk devices (aka. DASDs), tapes, communication controllers, etc. they can all be accessed by a well defined access method and they are presenting I/O completion a unified way : I/O interruptions. Every single device is uniquely identified to the system by a so called subchannel, where the ESA/390 architecture allows for 64k devices be attached.

Linux, however, was first built on the Intel PC architecture, with its two cascaded 8259 programmable interrupt controllers (PICs), that allow for a maximum of 15 different interrupt lines. All devices attached to such a system share those 15 interrupt levels. Devices attached to the ISA bus system must not share interrupt levels (aka. IRQs), as the ISA bus bases on edge triggered interrupts. MCA, EISA, PCI and other bus systems base on level triggered interrupts, and therewith allow for shared IRQs. However, if multiple devices present their hardware status by the same (shared) IRQ, the operating system has to call every single device driver registered on this IRQ in order to determine the device driver owning the device that raised the interrupt.

Up to kernel 2.4, Linux/390 used to provide interfaces via the IRQ (subchannel). For internal use of the common I/O layer, these are still there. However, device drivers should use the new calling interface via the ccw_device only.

During its startup the Linux/390 system checks for peripheral devices. Each of those devices is uniquely defined by a so called subchannel by the ESA/390 channel subsystem. While the subchannel numbers are system generated, each subchannel also takes a user defined attribute, the so called device number. Both subchannel number and device number cannot exceed 65535. During sysfs initialisation, the information about control unit type and device types that imply specific I/O commands (channel command words - CCWs) in order to operate the device are gathered. Device drivers can retrieve this set of hardware information during their initialization step to recognize the devices they support using the information saved in the struct ccw_device given to them. This methods implies that Linux/390 doesn't require to probe for free (not armed) interrupt request lines (IRQs) to drive its devices with. Where applicable, the device drivers can use issue the READ DEVICE CHARACTERISTICS ccw to retrieve device characteristics in its online routine.

In order to allow for easy I/O initiation the CDS layer provides a ccw_device_start() interface that takes a device specific channel program (one or more CCWs) as input sets up the required architecture specific control blocks and initiates an I/O request on behalf of the device driver. The ccw_device_start() routine allows to specify whether it expects the CDS layer to notify the device driver for every interrupt it observes, or with final status only. See ccw_device_start() for more details. A device driver must never issue ESA/390 I/O commands itself, but must use the Linux/390 CDS interfaces instead.

For long running I/O request to be canceled, the CDS layer provides the ccw_device_halt() function. Some devices require to initially issue a HALT SUBCHANNEL (HSCH) command without having pending I/O requests. This function is also covered by ccw_device_halt().

get_ciw() - get command information word

This call enables a device driver to get information about supported commands from the extended SenseID data.

```
struct ciw *
ccw_device_get_ciw(struct ccw_device *cdev, __u32 cmd);
```

cdev	The ccw_device for which the command is to be retrieved.
cmd	The command type to be retrieved.

ccw_device_get_ciw() returns:

NULL	No extended data available, invalid device or command not found.
!NULL	The command requested.

```
ccw_device_start() - Initiate I/O Request
```

The ccw_device_start() routines is the I/O request front-end processor. All device driver I/O requests must be issued using this routine. A device driver must not issue ESA/390 I/O commands itself. Instead the ccw_device_start() routine provides all interfaces required to drive arbitrary devices.

This description also covers the status information passed to the device driver's interrupt handler as this is related to the rules (flags) defined with the associated I/O request when calling ccw_device_start().

```
int ccw_device_start(struct ccw_device *cdev,
                     struct ccw1 *cpa,
                     unsigned long intparm,
                     __u8 lpm,
                     unsigned long flags);
int ccw_device_start_timeout(struct ccw_device *cdev,
                             struct ccw1 *cpa,
                             unsigned long intparm,
                             __u8 lpm,
                             unsigned long flags,
                             int expires);
int ccw_device_start_key(struct ccw_device *cdev,
                         struct ccw1 *cpa,
                         unsigned long intparm,
                         __u8 lpm,
                         __u8 key,
                         unsigned long flags);
int ccw_device_start_key_timeout(struct ccw_device *cdev,
                                 struct ccw1 *cpa,
                                 unsigned long intparm,
                                 __u8 lpm,
                                 __u8 key,
                                 unsigned long flags,
                                 int expires);
```

cdev	ccw_device the I/O is destined for
cpa	logical start address of channel program
user_intparm	user specific interrupt information; will be presented back to the device driver's interrupt handler. Allows a device driver to associate the interrupt with a particular I/O request.
lpm	defines the channel path to be used for a specific I/O request. A value of 0 will make cio use the opm.
key	the storage key to use for the I/O (useful for operating on a storage with a storage key != default key)
flag	defines the action to be performed for I/O processing
expires	timeout value in jiffies. The common I/O layer will terminate the running program after this and call the interrupt handler with ERR_PTR(-ETIMEDOUT) as irb.

Possible flag values are:

DOIO_ALLOW_SUSPEND	channel program may become suspended
DOIO_DENY_PREFETCH	don't allow for CCW prefetch; usually this implies the channel program might become modified
DOIO_SUPPRESS_INTER	don't call the handler on intermediate status

The cpa parameter points to the first format 1 CCW of a channel program:

```
struct ccw1 {
      __u8  cmd_code;/* command code */
      __u8  flags;   /* flags, like IDA addressing, etc. */
      __u16 count;   /* byte count */
      __u32 cda;     /* data address */
} __attribute__ ((packed,aligned(8)));
```

with the following CCW flags values defined:

CCW_FLAG_DC	data chaining
CCW_FLAG_CC	command chaining
CCW_FLAG_SLI	suppress incorrect length
CCW_FLAG_SKIP	skip
CCW_FLAG_PCI	PCI
CCW_FLAG_IDA	indirect addressing
CCW_FLAG_SUSPEND	suspend

Via ccw_device_set_options(), the device driver may specify the following options for the device:

DOIO_EARLY_NOTIFICATION	allow for early interrupt notification
DOIO_REPORT_ALL	report all interrupt conditions

The ccw_device_start() function returns:

0	successful completion or request successfully initiated
-EBUSY	The device is currently processing a previous I/O request, or there is a status pending at the device.
-ENODEV	cdev is invalid, the device is not operational or the ccw_device is not online.

When the I/O request completes, the CDS first level interrupt handler will accumulate the status in a struct irb and then call the device interrupt handler. The intparm field will contain the value the device driver has associated with a particular I/O request. If a pending device status was recognized, intparm will be set to 0 (zero). This may happen during I/O initiation or delayed by an alert status notification. In any case this status is not related to the current (last) I/O request. In case of a delayed status notification no special interrupt will be presented to indicate I/O completion as the I/O request was never started, even though ccw_device_start() returned with successful completion.

The irb may contain an error value, and the device driver should check for this first:

-ETIMEDOUT	the common I/O layer terminated the request after the specified timeout value
-EIO	the common I/O layer terminated the request due to an error state

If the concurrent sense flag in the extended status word (esw) in the irb is set, the field erw.scnt in the esw describes the number of device specific sense bytes available in the extended control word irb->scsw.ecw[]. No device sensing by the device driver itself is required.

The device interrupt handler can use the following definitions to investigate the primary unit check source coded in sense byte 0 :

SNS0_CMD_REJECT	0x80
SNS0_INTERVENTION_REQ	0x40
SNS0_BUS_OUT_CHECK	0x20
SNS0_EQUIPMENT_CHECK	0x10
SNS0_DATA_CHECK	0x08
SNS0_OVERRUN	0x04
SNS0_INCOMPL_DOMAIN	0x01

Depending on the device status, multiple of those values may be set together. Please refer to the device specific documentation for details.

The irb->scsw.cstat field provides the (accumulated) subchannel status :

SCHN_STAT_PCI	program controlled interrupt
SCHN_STAT_INCORR_LEN	incorrect length
SCHN_STAT_PROG_CHECK	program check
SCHN_STAT_PROT_CHECK	protection check
SCHN_STAT_CHN_DATA_CHK	channel data check
SCHN_STAT_CHN_CTRL_CHK	channel control check
SCHN_STAT_INTF_CTRL_CHK	interface control check
SCHN_STAT_CHAIN_CHECK	chaining check

The irb->scsw.dstat field provides the (accumulated) device status :

DEV_STAT_ATTENTION	attention
DEV_STAT_STAT_MOD	status modifier
DEV_STAT_CU_END	control unit end
DEV_STAT_BUSY	busy
DEV_STAT_CHN_END	channel end
DEV_STAT_DEV_END	device end
DEV_STAT_UNIT_CHECK	unit check
DEV_STAT_UNIT_EXCEP	unit exception

Please see the ESA/390 Principles of Operation manual for details on the individual flag meanings.

Usage Notes:

ccw_device_start() must be called disabled and with the ccw device lock held.

The device driver is allowed to issue the next ccw_device_start() call from within its interrupt handler already. It is not required to schedule a bottom-half, unless a non deterministically long running error recovery procedure or similar needs to be scheduled. During I/O processing the Linux/390 generic I/O device driver support has already obtained the IRQ lock, i.e. the handler must not try to obtain it again when calling ccw_device_start() or we end in a deadlock situation!

If a device driver relies on an I/O request to be completed prior to start the next it can reduce I/O processing overhead by chaining a NoOp I/O command CCW_CMD_NOOP to the end of the submitted CCW chain. This will force Channel-End and Device-End status to be presented together, with a single interrupt. However, this should be used with care as it implies the channel will remain busy, not being able to process I/O requests for other devices on the same channel. Therefore e.g. read commands should never use this technique, as the result will be presented by a single interrupt anyway.

In order to minimize I/O overhead, a device driver should use the DOIO_REPORT_ALL only if the device can report intermediate interrupt information prior to device-end the device driver urgently relies on. In this case all I/O interruptions are presented to the device driver until final status is recognized.

If a device is able to recover from asynchronously presented I/O errors, it can perform overlapping I/O using the DOIO_EARLY_NOTIFICATION flag. While some devices always report channel-end and device-end together, with a single interrupt, others present primary status (channel-end) when the channel is ready for the next I/O request and secondary status (device-end) when the data transmission has been completed at the device.

Above flag allows to exploit this feature, e.g. for communication devices that can handle lost data on the network to allow for enhanced I/O processing.

Unless the channel subsystem at any time presents a secondary status interrupt, exploiting this feature will cause only primary status interrupts to be presented to the device driver while overlapping I/O is performed. When a secondary status without error (alert status) is presented, this indicates successful completion for all overlapping ccw_device_start() requests that have been issued since the last secondary (final) status.

Channel programs that intend to set the suspend flag on a channel command word (CCW) must start the I/O operation with the DOIO_ALLOW_SUSPEND option or the suspend flag will cause a channel program check. At the time the channel program becomes suspended an intermediate interrupt will be generated by the channel subsystem.

ccw_device_resume() - Resume Channel Program Execution

If a device driver chooses to suspend the current channel program execution by setting the CCW suspend flag on a particular CCW, the channel program execution is suspended. In order to resume channel program execution the CIO layer provides the ccw_device_resume() routine.

```
int ccw_device_resume(struct ccw_device *cdev);
```

cdev	ccw_device the resume operation is requested for

The ccw_device_resume() function returns:

0	suspended channel program is resumed
-EBUSY	status pending
-ENODEV	cdev invalid or not-operational subchannel
-EINVAL	resume function not applicable
-ENOTCONN	there is no I/O request pending for completion

Usage Notes:

Please have a look at the ccw_device_start() usage notes for more details on suspended channel programs.

ccw_device_halt() - Halt I/O Request Processing

Sometimes a device driver might need a possibility to stop the processing of a long-running channel program or the device might require to initially issue a halt subchannel (HSCH) I/O command. For those purposes the ccw_device_halt() command is provided.

ccw_device_halt() must be called disabled and with the ccw device lock held.

```
int ccw_device_halt(struct ccw_device *cdev,
                    unsigned long intparm);
```

cdev	ccw_device the halt operation is requested for
intparm	interruption parameter; value is only used if no I/O is outstanding, otherwise the intparm associated with the I/O request is returned

The ccw_device_halt() function returns:

0	request successfully initiated
-EBUSY	the device is currently busy, or status pending.
-ENODEV	cdev invalid.
-EINVAL	The device is not operational or the ccw device is not online.

Usage Notes:

A device driver may write a never-ending channel program by writing a channel program that at its end loops back to its beginning by means of a transfer in channel (TIC) command (CCW_CMD_TIC). Usually this is performed by network device drivers by setting the PCI CCW flag (CCW_FLAG_PCI). Once this CCW is executed a program controlled interrupt (PCI) is generated. The device driver can then perform an appropriate action. Prior to interrupt of an outstanding read to a network device (with or without PCI flag) a ccw_device_halt() is required to end the pending operation.

```
ccw_device_clear() - Terminage I/O Request Processing
```

In order to terminate all I/O processing at the subchannel, the clear subchannel (CSCH) command is used. It can be issued via ccw_device_clear().

ccw_device_clear() must be called disabled and with the ccw device lock held.

```
int ccw_device_clear(struct ccw_device *cdev, unsigned long intparm);
```

cdev	ccw_device the clear operation is requested for
intparm	interruption parameter (see ccw_device_halt())

The ccw_device_clear() function returns:

0	request successfully initiated
-ENODEV	cdev invalid
-EINVAL	The device is not operational or the ccw device is not online.

Miscellaneous Support Routines

This chapter describes various routines to be used in a Linux/390 device driver programming environment.

get_ccwdev_lock()

Get the address of the device specific lock. This is then used in spin_lock() / spin_unlock() calls.

```
__u8 ccw_device_get_path_mask(struct ccw_device *cdev);
```

Get the mask of the path currently available for cdev.

13.2 IBM 3270 Display System support

This file describes the driver that supports local channel attachment of IBM 3270 devices. It consists of three sections:

- Introduction
- Installation
- Operation

13.2.1 Introduction

This paper describes installing and operating 3270 devices under Linux/390. A 3270 device is a block-mode rows-and-columns terminal of which I'm sure hundreds of millions were sold by IBM and clonemakers twenty and thirty years ago.

You may have 3270s in-house and not know it. If you're using the VM-ESA operating system, define a 3270 to your virtual machine by using the command "DEF GRAF <hex-address>" This paper presumes you will be defining four 3270s with the CP/CMS commands:

- DEF GRAF 620
- DEF GRAF 621
- DEF GRAF 622
- DEF GRAF 623

Your network connection from VM-ESA allows you to use x3270, tn3270, or another 3270 emulator, started from an xterm window on your PC or workstation. With the DEF GRAF command, an application such as xterm, and this Linux-390 3270 driver, you have another way of talking to your Linux box.

This paper covers installation of the driver and operation of a dialed-in x3270.

13.2.2 Installation

You install the driver by installing a patch, doing a kernel build, and running the configuration script (config3270.sh, in this directory).

WARNING: If you are using 3270 console support, you must rerun the configuration script every time you change the console's address (perhaps by using the condev= parameter in silo's /boot/parmfile). More precisely, you should rerun the configuration script every time your set of 3270s, including the console 3270, changes subchannel identifier relative to one another. ReIPL as soon as possible after running the configuration script and the resulting /tmp/mkdev3270.

If you have chosen to make tub3270 a module, you add a line to a configuration file under /etc/modprobe.d/. If you are working on a VM virtual machine, you can use DEF GRAF to define virtual 3270 devices.

You may generate both 3270 and 3215 console support, or one or the other, or neither. If you generate both, the console type under VM is not changed. Use #CP Q TERM to see what the current console type is. Use #CP TERM CONMODE 3270 to change it to 3270. If you generate only 3270 console support, then the driver automatically converts your console at boot time to a 3270 if it is a 3215.

In brief, these are the steps:

1. Install the tub3270 patch
2. (If a module) add a line to a file in */etc/modprobe.d/*.conf*
3. (If VM) define devices with DEF GRAF
4. Reboot
5. Configure

To test that everything works, assuming VM and x3270,

1. Bring up an x3270 window.
2. Use the DIAL command in that window.
3. You should immediately see a Linux login screen.

Here are the installation steps in detail:

1. The 3270 driver is a part of the official Linux kernel source. Build a tree with the kernel source and any necessary patches. Then do:

```
make oldconfig
(If you wish to disable 3215 console support, edit
.config; change CONFIG_TN3215's value to "n";
and rerun "make oldconfig".)
make image
make modules
make modules_install
```

2. (Perform this step only if you have configured tub3270 as a module.) Add a line to a file */etc/modprobe.d/*.conf* to automatically load the driver when it's needed. With this line added, you will see login prompts appear on your 3270s as soon as boot is complete (or with emulated 3270s, as soon as you dial into your vm guest using the command "DIAL <vmguestname>"). Since the line-mode major number is 227, the line to add should be:

```
alias char-major-227 tub3270
```

3. Define graphic devices to your vm guest machine, if you haven't already. Define them before you reboot (reipl):

- DEFINE GRAF 620
- DEFINE GRAF 621
- DEFINE GRAF 622
- DEFINE GRAF 623

4. Reboot. The reboot process scans hardware devices, including 3270s, and this enables the tub3270 driver once loaded to respond correctly to the configuration requests of the next step. If you have chosen 3270 console support, your console now behaves as a 3270, not a 3215.

5. Run the 3270 configuration script config3270. It is distributed in this same directory, Documentation/arch/s390, as config3270.sh. Inspect the output script it produces, /tmp/mkdev3270, and then run that script. This will create the necessary character special device files and make the necessary changes to /etc/inittab.

Then notify /sbin/init that /etc/inittab has changed, by issuing the telinit command with the q operand:

```
cd Documentation/arch/s390
sh config3270.sh
sh /tmp/mkdev3270
telinit q
```

This should be sufficient for your first time. If your 3270 configuration has changed and you're reusing config3270, you should follow these steps:

```
Change 3270 configuration
Reboot
Run config3270 and /tmp/mkdev3270
Reboot
```

Here are the testing steps in detail:

1. Bring up an x3270 window, or use an actual hardware 3278 or 3279, or use the 3270 emulator of your choice. You would be running the emulator on your PC or workstation. You would use the command, for example:

```
x3270 vm-esa-domain-name &
```

if you wanted a 3278 Model 4 with 43 rows of 80 columns, the default model number. The driver does not take advantage of extended attributes.

The screen you should now see contains a VM logo with input lines near the bottom. Use TAB to move to the bottom line, probably labeled "COMMAND ===>".

2. Use the DIAL command instead of the LOGIN command to connect to one of the virtual 3270s you defined with the DEF GRAF commands:

```
dial my-vm-guest-name
```

3. You should immediately see a login prompt from your Linux-390 operating system. If that does not happen, you would see instead the line "DIALED TO my-vm-guest-name 0620".

To troubleshoot: do these things.

A. Is the driver loaded? Use the lsmod command (no operands) to find out. Probably it isn't. Try loading it manually, with the command "insmod tub3270". Does that command give error messages? Ha! There's your problem.

B. Is the /etc/inittab file modified as in installation step 3 above? Use the grep command to find out; for instance, issue "grep 3270 /etc/inittab". Nothing found? There's your problem!

C. Are the device special files created, as in installation step 2 above? Use the ls -l command to find out; for instance, issue "ls -l /dev/3270/tty620". The output should start with the letter "c" meaning character device and should contain "227, 1" just to the left of the device name. No such file? no "c"? Wrong major number? Wrong minor number? There's your problem!

D. Do you get the message:

```
"HCPDIA047E my-vm-guest-name 0620 does not exist"?
```

If so, you must issue the command "DEF GRAF 620" from your VM 3215 console and then reboot the system.

13.2.3 OPERATION.

The driver defines three areas on the 3270 screen: the log area, the input area, and the status area.

The log area takes up all but the bottom two lines of the screen. The driver writes terminal output to it, starting at the top line and going down. When it fills, the status area changes from "Linux Running" to "Linux More...". After a scrolling timeout of (default) 5 sec, the screen clears and more output is written, from the top down.

The input area extends from the beginning of the second-to-last screen line to the start of the status area. You type commands in this area and hit ENTER to execute them.

The status area initializes to "Linux Running" to give you a warm fuzzy feeling. When the log area fills up and output awaits, it changes to "Linux More...". At this time you can do several things or nothing. If you do nothing, the screen will clear in (default) 5 sec and more output will appear. You may hit ENTER with nothing typed in the input area to toggle between "Linux More..." and "Linux Holding", which indicates no scrolling will occur. (If you hit ENTER with "Linux Running" and nothing typed, the application receives a newline.)

You may change the scrolling timeout value. For example, the following command line:

```
echo scrolltime=60 > /proc/tty/driver/tty3270
```

changes the scrolling timeout value to 60 sec. Set scrolltime to 0 if you wish to prevent scrolling entirely.

Other things you may do when the log area fills up are: hit PA2 to clear the log area and write more output to it, or hit CLEAR to clear the log area and the input area and write more output to the log area.

Some of the Program Function (PF) and Program Attention (PA) keys are preassigned special functions. The ones that are not yield an alarm when pressed.

PA1 causes a SIGINT to the currently running application. You may do the same thing from the input area, by typing "^C" and hitting ENTER.

PA2 causes the log area to be cleared. If output awaits, it is then written to the log area.

PF3 causes an EOF to be received as input by the application. You may cause an EOF also by typing "^D" and hitting ENTER.

No PF key is preassigned to cause a job suspension, but you may cause a job suspension by typing "^Z" and hitting ENTER. You may wish to assign this function to a PF key. To make PF7 cause job suspension, execute the command:

```
echo pf7=^z > /proc/tty/driver/tty3270
```

If the input you type does not end with the two characters "^n", the driver appends a newline character and sends it to the tty driver; otherwise the driver strips the "^n" and does not append a newline. The IBM 3215 driver behaves similarly.

Pf10 causes the most recent command to be retrieved from the tube's command stack (default depth 20) and displayed in the input area. You may hit PF10 again for the next-most-recent command, and so on. A command is entered into the stack only when the input area is not made invisible (such as for password entry) and it is not identical to the current top entry.

PF10 rotates backward through the command stack; PF11 rotates forward. You may assign the backward function to any PF key (or PA key, for that matter), say, PA3, with the command:

```
echo -e pa3=\\033k > /proc/tty/driver/tty3270
```

This assigns the string ESC-k to PA3. Similarly, the string ESC-j performs the forward function. (Rationale: In bash with vi-mode line editing, ESC-k and ESC-j retrieve backward and forward history. Suggestions welcome.)

Is a stack size of twenty commands not to your liking? Change it on the fly. To change to saving the last 100 commands, execute the command:

```
echo recallsize=100 > /proc/tty/driver/tty3270
```

Have a command you issue frequently? Assign it to a PF or PA key! Use the command:

```
echo pf24="mkdir foobar; cd foobar" > /proc/tty/driver/tty3270
```

to execute the commands mkdir foobar and cd foobar immediately when you hit PF24. Want to see the command line first, before you execute it? Use the -n option of the echo command:

```
echo -n pf24="mkdir foo; cd foo" > /proc/tty/driver/tty3270
```

Happy testing! I welcome any and all comments about this document, the driver, etc etc.

Dick Hitt <rbh00@utsglobal.com>

13.3 S/390 driver model interfaces

13.3.1 1. CCW devices

All devices which can be addressed by means of ccws are called 'CCW devices' - even if they aren't actually driven by ccws.

All ccw devices are accessed via a subchannel, this is reflected in the structures under devices/:

```
devices/
   - system/
   - css0/
         - 0.0.0000/0.0.0815/
         - 0.0.0001/0.0.4711/
         - 0.0.0002/
         - 0.1.0000/0.1.1234/
         ...
         - defunct/
```

In this example, device 0815 is accessed via subchannel 0 in subchannel set 0, device 4711 via subchannel 1 in subchannel set 0, and subchannel 2 is a non-I/O subchannel. Device 1234 is accessed via subchannel 0 in subchannel set 1.

The subchannel named 'defunct' does not represent any real subchannel on the system; it is a pseudo subchannel where disconnected ccw devices are moved to if they are displaced by another ccw device becoming operational on their former subchannel. The ccw devices will be moved again to a proper subchannel if they become operational again on that subchannel.

You should address a ccw device via its bus id (e.g. 0.0.4711); the device can be found under bus/ccw/devices/.

All ccw devices export some data via sysfs.

cutype:
The control unit type / model.

devtype:
The device type / model, if applicable.

availability:
Can be 'good' or 'boxed'; 'no path' or 'no device' for disconnected devices.

online:
An interface to set the device online and offline. In the special case of the device being disconnected (see the notify function under 1.2), piping 0 to online will forcibly delete the device.

The device drivers can add entries to export per-device data and interfaces.

There is also some data exported on a per-subchannel basis (see under bus/css/devices/):

chpids:
Via which chpids the device is connected.

pimpampom:
The path installed, path available and path operational masks.

There also might be additional data, for example for block devices.

13.3.2 1.1 Bringing up a ccw device

This is done in several steps.

a. Each driver can provide one or more parameter interfaces where parameters can be specified. These interfaces are also in the driver's responsibility.

b. After a. has been performed, if necessary, the device is finally brought up via the 'online' interface.

13.3.3 1.2 Writing a driver for ccw devices

The basic struct ccw_device and struct ccw_driver data structures can be found under include/asm/ccwdev.h:

```
struct ccw_device {
      spinlock_t *ccwlock;
      struct ccw_device_private *private;
      struct ccw_device_id id;

      struct ccw_driver *drv;
      struct device dev;
      int online;

      void (*handler) (struct ccw_device *dev, unsigned long intparm,
                       struct irb *irb);
};

struct ccw_driver {
      struct module *owner;
      struct ccw_device_id *ids;
      int (*probe) (struct ccw_device *);
      int (*remove) (struct ccw_device *);
      int (*set_online) (struct ccw_device *);
      int (*set_offline) (struct ccw_device *);
      int (*notify) (struct ccw_device *, int);
      struct device_driver driver;
      char *name;
};
```

The 'private' field contains data needed for internal i/o operation only, and is not available to the device driver.

Each driver should declare in a MODULE_DEVICE_TABLE into which CU types/models and/or device types/models it is interested. This information can later be found in the struct ccw_device_id fields:

```
struct ccw_device_id {
      __u16   match_flags;

      __u16   cu_type;
      __u16   dev_type;
      __u8    cu_model;
      __u8    dev_model;

      unsigned long driver_info;
};
```

The functions in ccw_driver should be used in the following way:

probe:
This function is called by the device layer for each device the driver is interested in. The driver should only allocate private structures to put in dev->driver_data and create attributes (if needed). Also, the interrupt handler (see below) should be set here.

```
int (*probe) (struct ccw_device *cdev);
```

Parameters:

cdev

- the device to be probed.

remove:
This function is called by the device layer upon removal of the driver, the device or the module. The driver should perform cleanups here.

```
int (*remove) (struct ccw_device *cdev);
```

Parameters:

cdev

- the device to be removed.

set_online:
This function is called by the common I/O layer when the device is activated via the 'online' attribute. The driver should finally setup and activate the device here.

```
int (*set_online) (struct ccw_device *);
```

Parameters:

cdev

- the device to be activated. The common layer has verified that the device is not already online.

set_offline: This function is called by the common I/O layer when the device is
de-activated via the 'online' attribute. The driver should shut down the device, but not de-allocate its private data.

```
int (*set_offline) (struct ccw_device *);
```

Parameters:

cdev

- **the device to be deactivated. The common layer has**
  verified that the device is online.

notify:
This function is called by the common I/O layer for some state changes of the device.

Signalled to the driver are:

- In online state, device detached (CIO_GONE) or last path gone (CIO_NO_PATH). The driver must return !0 to keep the device; for return code 0, the device will be deleted as usual (also when no notify function is registered). If the driver wants to keep the device, it is moved into disconnected state.
- In disconnected state, device operational again (CIO_OPER). The common I/O layer performs some sanity checks on device number and Device / CU to be reasonably sure if it is still the same device. If not, the old device is removed and a new one registered. By the return code of the notify function the device driver signals if it wants the device back: !0 for keeping, 0 to make the device being removed and re-registered.

```
int (*notify) (struct ccw_device *, int);
```

Parameters:

cdev

- the device whose state changed.

event

- the event that happened. This can be one of CIO_GONE, CIO_NO_PATH or CIO_OPER.

The handler field of the struct ccw_device is meant to be set to the interrupt handler for the device. In order to accommodate drivers which use several distinct handlers (e.g. multi subchannel devices), this is a member of ccw_device instead of ccw_driver. The handler is registered with the common layer during set_online() processing before the driver is called, and is deregistered during set_offline() after the driver has been called. Also, after registering / before deregistering, path grouping resp. disbanding of the path group (if applicable) are performed.

```
void (*handler) (struct ccw_device *dev, unsigned long intparm, struct irb␣
↪*irb);
```

Parameters: dev - the device the handler is called for

intparm - the intparm which allows the device driver to identify
the i/o the interrupt is associated with, or to recognize the interrupt as unsolicited.

irb - interruption response block which contains the accumulated
status.

The device driver is called from the common ccw_device layer and can retrieve information about the interrupt from the irb parameter.

13.3.4 1.3 ccwgroup devices

The ccwgroup mechanism is designed to handle devices consisting of multiple ccw devices, like lcs or ctc.

The ccw driver provides a 'group' attribute. Piping bus ids of ccw devices to this attributes creates a ccwgroup device consisting of these ccw devices (if possible). This ccwgroup device can be set online or offline just like a normal ccw device.

Each ccwgroup device also provides an 'ungroup' attribute to destroy the device again (only when offline). This is a generic ccwgroup mechanism (the driver does not need to implement anything beyond normal removal routines).

A ccw device which is a member of a ccwgroup device carries a pointer to the ccwgroup device in the driver_data of its device struct. This field must not be touched by the driver - it should use the ccwgroup device's driver_data for its private data.

To implement a ccwgroup driver, please refer to include/asm/ccwgroup.h. Keep in mind that most drivers will need to implement both a ccwgroup and a ccw driver.

13.3.5 2. Channel paths

Channel paths show up, like subchannels, under the channel subsystem root (css0) and are called 'chp0.<chpid>'. They have no driver and do not belong to any bus. Please note, that unlike /proc/chpids in 2.4, the channel path objects reflect only the logical state and not the physical state, since we cannot track the latter consistently due to lacking machine support (we don't need to be aware of it anyway).

status

- Can be 'online' or 'offline'. Piping 'on' or 'off' sets the chpid logically online/offline. Piping 'on' to an online chpid triggers path reprobing for all devices the chpid connects to. This can be used to force the kernel to re-use a channel path the user knows to be online, but the machine hasn't created a machine check for.

type

- The physical type of the channel path.

shared

- Whether the channel path is shared.

cmg

- The channel measurement group.

13.3.6 3. System devices

13.3.7 3.1 xpram

xpram shows up under devices/system/ as 'xpram'.

13.3.8 3.2 cpus

For each cpu, a directory is created under devices/system/cpu/. Each cpu has an attribute 'online' which can be 0 or 1.

13.3.9 4. Other devices

13.3.10 4.1 Netiucv

The netiucv driver creates an attribute 'connection' under bus/iucv/drivers/netiucv. Piping to this attribute creates a new netiucv connection to the specified host.

Netiucv connections show up under devices/iucv/ as "netiucv<ifnum>". The interface number is assigned sequentially to the connections defined via the 'connection' attribute.

user

- shows the connection partner.

buffer

- maximum buffer size. Pipe to it to change buffer size.

13.4 Linux API for read access to z/VM Monitor Records

Date : 2004-Nov-26

Author: Gerald Schaefer (geraldsc@de.ibm.com)

13.4.1 Description

This item delivers a new Linux API in the form of a misc char device that is usable from user space and allows read access to the z/VM Monitor Records collected by the **MONITOR* System Service of z/VM.

13.4.2 User Requirements

The z/VM guest on which you want to access this API needs to be configured in order to allow IUCV connections to the **MONITOR* service, i.e. it needs the IUCV **MONITOR* statement in its user entry. If the monitor DCSS to be used is restricted (likely), you also need the NAMESAVE <DCSS NAME> statement. This item will use the IUCV device driver to access the z/VM services, so you need a kernel with IUCV support. You also need z/VM version 4.4 or 5.1.

There are two options for being able to load the monitor DCSS (examples assume that the monitor DCSS begins at 144 MB and ends at 152 MB). You can query the location of the monitor DCSS with the Class E privileged CP command Q NSS MAP (the values BEGPAG and ENDPAG are given in units of 4K pages).

See also “CP Command and Utility Reference” (SC24-6081-00) for more information on the DEF STOR and Q NSS MAP commands, as well as “Saved Segments Planning and Administration” (SC24-6116-00) for more information on DCSSes.

1st option:

You can use the CP command DEF STOR CONFIG to define a “memory hole” in your guest virtual storage around the address range of the DCSS.

Example: DEF STOR CONFIG 0.140M 200M.200M

This defines two blocks of storage, the first is 140MB in size an begins at address 0MB, the second is 200MB in size and begins at address 200MB, resulting in a total storage of 340MB. Note that the first block should always start at 0 and be at least 64MB in size.

2nd option:

Your guest virtual storage has to end below the starting address of the DCSS and you have to specify the “mem=” kernel parameter in your parmfile with a value greater than the ending address of the DCSS.

Example:

```
DEF STOR 140M
```

This defines 140MB storage size for your guest, the parameter “mem=160M” is added to the parmfile.

13.4.3 User Interface

The char device is implemented as a kernel module named "monreader", which can be loaded via the modprobe command, or it can be compiled into the kernel instead. There is one optional module (or kernel) parameter, "mondcss", to specify the name of the monitor DCSS. If the module is compiled into the kernel, the kernel parameter "monreader.mondcss=<DCSS NAME>" can be specified in the parmfile.

The default name for the DCSS is "MONDCSS" if none is specified. In case that there are other users already connected to the **MONITOR* service (e.g. Performance Toolkit), the monitor DCSS is already defined and you have to use the same DCSS. The CP command Q MONITOR (Class E privileged) shows the name of the monitor DCSS, if already defined, and the users connected to the **MONITOR* service. Refer to the "z/VM Performance" book (SC24-6109-00) on how to create a monitor DCSS if your z/VM doesn't have one already, you need Class E privileges to define and save a DCSS.

Example:

```
modprobe monreader mondcss=MYDCSS
```

This loads the module and sets the DCSS name to "MYDCSS".

NOTE:

This API provides no interface to control the **MONITOR* service, e.g. specify which data should be collected. This can be done by the CP command MONITOR (Class E privileged), see "CP Command and Utility Reference".

Device nodes with udev:

After loading the module, a char device will be created along with the device node /<udev directory>/monreader.

Device nodes without udev:

If your distribution does not support udev, a device node will not be created automatically and you have to create it manually after loading the module. Therefore you need to know the major and minor numbers of the device. These numbers can be found in /sys/class/misc/monreader/dev.

Typing cat /sys/class/misc/monreader/dev will give an output of the form <major>:<minor>. The device node can be created via the mknod command, enter mknod <name> c <major> <minor>, where <name> is the name of the device node to be created.

Example:

```
# modprobe monreader
# cat /sys/class/misc/monreader/dev
10:63
# mknod /dev/monreader c 10 63
```

This loads the module with the default monitor DCSS (MONDCSS) and creates a device node.

File operations:

The following file operations are supported: open, release, read, poll. There are two alternative methods for reading: either non-blocking read in conjunction with polling, or blocking read without polling. IOCTLs are not supported.

Read:

Reading from the device provides a 12 Byte monitor control element (MCE), followed by a set of one or more contiguous monitor records (similar to the output of the CMS utility MONWRITE without the 4K control blocks). The MCE contains information on the type of the following record set (sample/event data), the monitor domains contained within it and the start and end address of the record set in the monitor DCSS. The start and end address can be used to determine the size of the record set, the end address is the address of the last byte of data. The start address is needed to handle "end-of-frame" records correctly (domain 1, record 13), i.e. it can be used to determine the record start offset relative to a 4K page (frame) boundary.

See "Appendix A: **MONITOR*" in the "z/VM Performance" document for a description of the monitor control element layout. The layout of the monitor records can be found here (z/VM 5.1): https://www.vm.ibm.com/pubs/mon510/index.html

The layout of the data stream provided by the monreader device is as follows:

```
...
<0 byte read>
<first MCE>              \
<first set of records>    |
...                       |- data set
<last MCE>                |
<last set of records>    /
<0 byte read>
...
```

There may be more than one combination of MCE and corresponding record set within one data set and the end of each data set is indicated by a successful read with a return value of 0 (0 byte read). Any received data must be considered invalid until a complete set was read successfully, including the closing 0 byte read. Therefore you should always read the complete set into a buffer before processing the data.

The maximum size of a data set can be as large as the size of the monitor DCSS, so design the buffer adequately or use dynamic memory allocation. The size of the monitor DCSS will be printed into syslog after loading the module. You can also use the (Class E privileged) CP command Q NSS MAP to list all available segments and information about them.

As with most char devices, error conditions are indicated by returning a negative value for the number of bytes read. In this case, the errno variable indicates the error condition:

EIO:
: reply failed, read data is invalid and the application should discard the data read since the last successful read with 0 size.

EFAULT:
: copy_to_user failed, read data is invalid and the application should discard the data read since the last successful read with 0 size.

EAGAIN:
: occurs on a non-blocking read if there is no data available at the moment. There is no data missing or corrupted, just try again or rather use polling for non-blocking reads.

EOVERFLOW:
: message limit reached, the data read since the last successful read with 0 size is valid but subsequent records may be missing.

In the last case (EOVERFLOW) there may be missing data, in the first two cases (EIO, EFAULT) there will be missing data. It's up to the application if it will continue reading subsequent data or rather exit.

Open:

Only one user is allowed to open the char device. If it is already in use, the open function will fail (return a negative value) and set errno to EBUSY. The open function may also fail if an IUCV connection to the **MONITOR* service cannot be established. In this case errno will be set to EIO and an error message with an IPUSER SEVER code will be printed into syslog. The IPUSER SEVER codes are described in the "z/VM Performance" book, Appendix A.

NOTE:

As soon as the device is opened, incoming messages will be accepted and they will account for the message limit, i.e. opening the device without reading from it will provoke the "message limit reached" error (EOVERFLOW error code) eventually.

13.5 IBM s390 QDIO Ethernet Driver

13.5.1 OSA and HiperSockets Bridge Port Support

Uevents

To generate the events the device must be assigned a role of either a primary or a secondary Bridge Port. For more information, see "z/VM Connectivity, SC24-6174".

When run on an OSA or HiperSockets Bridge Capable Port hardware, and the state of some configured Bridge Port device on the channel changes, a udev event with ACTION=CHANGE is emitted on behalf of the corresponding ccwgroup device. The event has the following attributes:

BRIDGEPORT=statechange
: indicates that the Bridge Port device changed its state.

ROLE={primary|secondary|none}
the role assigned to the port.

STATE={active|standby|inactive}
the newly assumed state of the port.

When run on HiperSockets Bridge Capable Port hardware with host address notifications enabled, a udev event with ACTION=CHANGE is emitted. It is emitted on behalf of the corresponding ccwgroup device when a host or a VLAN is registered or unregistered on the network served by the device. The event has the following attributes:

BRIDGEDHOST={reset|register|deregister|abort}
host address notifications are started afresh, a new host or VLAN is registered or deregistered on the Bridge Port HiperSockets channel, or address notifications are aborted.

VLAN=numeric-vlan-id
VLAN ID on which the event occurred. Not included if no VLAN is involved in the event.

MAC=xx:xx:xx:xx:xx:xx
MAC address of the host that is being registered or deregistered from the HiperSockets channel. Not reported if the event reports the creation or destruction of a VLAN.

NTOK_BUSID=x.y.zzzz
device bus ID (CSSID, SSID and device number).

NTOK_IID=xx
device IID.

NTOK_CHPID=xx
device CHPID.

NTOK_CHID=xxxx
device channel ID.

Note that the *NTOK_** attributes refer to devices other than the one connected to the system on which the OS is running.

13.6 S390 Debug Feature

files:

- arch/s390/kernel/debug.c
- arch/s390/include/asm/debug.h

13.6.1 Description:

The goal of this feature is to provide a kernel debug logging API where log records can be stored efficiently in memory, where each component (e.g. device drivers) can have one separate debug log. One purpose of this is to inspect the debug logs after a production system crash in order to analyze the reason for the crash.

If the system still runs but only a subcomponent which uses dbf fails, it is possible to look at the debug logs on a live system via the Linux debugfs filesystem.

The debug feature may also very useful for kernel and driver development.

13.6.2 Design:

Kernel components (e.g. device drivers) can register themselves at the debug feature with the function call *debug_register()*. This function initializes a debug log for the caller. For each debug log exists a number of debug areas where exactly one is active at one time. Each debug area consists of contiguous pages in memory. In the debug areas there are stored debug entries (log records) which are written by event- and exception-calls.

An event-call writes the specified debug entry to the active debug area and updates the log pointer for the active area. If the end of the active debug area is reached, a wrap around is done (ring buffer) and the next debug entry will be written at the beginning of the active debug area.

An exception-call writes the specified debug entry to the log and switches to the next debug area. This is done in order to be sure that the records which describe the origin of the exception are not overwritten when a wrap around for the current area occurs.

The debug areas themselves are also ordered in form of a ring buffer. When an exception is thrown in the last debug area, the following debug entries are then written again in the very first area.

There are four versions for the event- and exception-calls: One for logging raw data, one for text, one for numbers (unsigned int and long), and one for sprintf-like formatted strings.

Each debug entry contains the following data:

- Timestamp
- Cpu-Number of calling task
- Level of debug entry (0…6)
- Return Address to caller
- Flag, if entry is an exception or not

The debug logs can be inspected in a live system through entries in the debugfs-filesystem. Under the toplevel directory "`s390dbf`" there is a directory for each registered component, which is named like the corresponding component. The debugfs normally should be mounted to `/sys/kernel/debug` therefore the debug feature can be accessed under `/sys/kernel/debug/s390dbf`.

The content of the directories are files which represent different views to the debug log. Each component can decide which views should be used through registering them with the function *debug_register_view()*. Predefined views for hex/ascii and sprintf data are provided. It is also possible to define other views. The content of a view can be inspected simply by reading the corresponding debugfs file.

All debug logs have an actual debug level (range from 0 to 6). The default level is 3. Event and Exception functions have a `level` parameter. Only debug entries with a level that is lower or equal than the actual level are written to the log. This means, when writing events, high priority log entries should have a low level value whereas low priority entries should have a high one. The actual debug level can be changed with the help of the debugfs-filesystem through writing a number string "x" to the `level` debugfs file which is provided for every debug log. Debugging can be switched off completely by using "-" on the `level` debugfs file.

Example:

```
> echo "-" > /sys/kernel/debug/s390dbf/dasd/level
```

It is also possible to deactivate the debug feature globally for every debug log. You can change the behavior using 2 sysctl parameters in `/proc/sys/s390dbf`:

There are currently 2 possible triggers, which stop the debug feature globally. The first possibility is to use the `debug_active` sysctl. If set to 1 the debug feature is running. If `debug_active` is set to 0 the debug feature is turned off.

The second trigger which stops the debug feature is a kernel oops. That prevents the debug feature from overwriting debug information that happened before the oops. After an oops you can reactivate the debug feature by piping 1 to `/proc/sys/s390dbf/debug_active`. Nevertheless, it's not suggested to use an oopsed kernel in a production environment.

If you want to disallow the deactivation of the debug feature, you can use the `debug_stoppable` sysctl. If you set `debug_stoppable` to 0 the debug feature cannot be stopped. If the debug feature is already stopped, it will stay deactivated.

13.6.3 Kernel Interfaces:

debug_info_t ***debug_register_mode**(const char *name, int pages_per_area, int nr_areas, int buf_size, umode_t mode, uid_t uid, gid_t gid)

creates and initializes debug area.

Parameters

`const char *name`
Name of debug log (e.g. used for debugfs entry)

`int pages_per_area`
Number of pages, which will be allocated per area

`int nr_areas`
Number of debug areas

`int buf_size`
Size of data area in each debug entry

`umode_t mode`
File mode for debugfs files. E.g. S_IRWXUGO

`uid_t uid`
User ID for debugfs files. Currently only 0 is supported.

`gid_t gid`
Group ID for debugfs files. Currently only 0 is supported.

Return

- Handle for generated debug area
- `NULL` if register failed

Description

Allocates memory for a debug log. Must not be called within an interrupt handler.

debug_info_t ***debug_register**(const char *name, int pages_per_area, int nr_areas, int buf_size)

creates and initializes debug area with default file mode.

Parameters

const char *name
Name of debug log (e.g. used for debugfs entry)

int pages_per_area
Number of pages, which will be allocated per area

int nr_areas
Number of debug areas

int buf_size
Size of data area in each debug entry

Return

- Handle for generated debug area
- `NULL` if register failed

Description

Allocates memory for a debug log. The debugfs file mode access permissions are read and write for user. Must not be called within an interrupt handler.

void **debug_register_static**(debug_info_t *id, int pages_per_area, int nr_areas)

registers a static debug area

Parameters

debug_info_t *id
Handle for static debug area

int pages_per_area
Number of pages per area

int nr_areas
Number of debug areas

Description

Register debug_info_t defined using DEFINE_STATIC_DEBUG_INFO.

Note

This function is called automatically via an initcall generated by
DEFINE_STATIC_DEBUG_INFO.

void **debug_unregister**(debug_info_t *id)

give back debug area.

Parameters

debug_info_t *id
handle for debug log

Return

none

void **debug_set_level**(debug_info_t *id, int new_level)
    Sets new actual debug level if new_level is valid.

Parameters

debug_info_t *id
    handle for debug log

int new_level
    new debug level

Return

    none

void **debug_stop_all**(void)
    stops the debug feature if stopping is allowed.

Parameters

void
    no arguments

Return

- none

Description

Currently used in case of a kernel oops.

void **debug_set_critical**(void)
    event/exception functions try lock instead of spin.

Parameters

void
    no arguments

Return

- none

Description

Currently used in case of stopping all CPUs but the current one. Once in this state, functions to write a debug entry for an event or exception no longer spin on the debug area lock, but only try to get it and fail if they do not get the lock.

int **debug_register_view**(debug_info_t *id, struct debug_view *view)
    registers new debug view and creates debugfs dir entry

Parameters

debug_info_t *id
    handle for debug log

struct debug_view *view
    pointer to debug view struct

Return

- 0 : ok

- < 0: Error

int **debug_unregister_view**(debug_info_t *id, struct debug_view *view)
unregisters debug view and removes debugfs dir entry

Parameters

debug_info_t *id
handle for debug log

struct debug_view *view
pointer to debug view struct

Return

- 0 : ok
- < 0: Error

bool **debug_level_enabled**(debug_info_t *id, int level)
Returns true if debug events for the specified level would be logged. Otherwise returns false.

Parameters

debug_info_t *id
handle for debug log

int level
debug level

Return

- `true` if level is less or equal to the current debug level.

debug_entry_t ***debug_event**(debug_info_t *id, int level, void *data, int length)
writes binary debug entry to active debug area (if level <= actual debug level)

Parameters

debug_info_t *id
handle for debug log

int level
debug level

void *data
pointer to data for debug entry

int length
length of data in bytes

Return

- Address of written debug entry
- `NULL` if error

debug_entry_t ***debug_int_event**(debug_info_t *id, int level, unsigned int tag)
writes unsigned integer debug entry to active debug area (if level <= actual debug level)

Parameters

debug_info_t *id
handle for debug log

int level
debug level

unsigned int tag
integer value for debug entry

Return

- Address of written debug entry
- NULL if error

debug_entry_t ***debug_long_event**(debug_info_t *id, int level, unsigned long tag)
writes unsigned long debug entry to active debug area (if level <= actual debug level)

Parameters

debug_info_t *id
handle for debug log

int level
debug level

unsigned long tag
long integer value for debug entry

Return

- Address of written debug entry
- NULL if error

debug_entry_t ***debug_text_event**(debug_info_t *id, int level, const char *txt)
writes string debug entry in ascii format to active debug area (if level <= actual debug level)

Parameters

debug_info_t *id
handle for debug log

int level
debug level

const char *txt
string for debug entry

Return

- Address of written debug entry
- NULL if error

debug_sprintf_event

`debug_sprintf_event (_id, _level, _fmt, ...)`

writes debug entry with format string and varargs (longs) to active debug area (if level $<=$ actual debug level).

Parameters

`_id`
handle for debug log

`_level`
debug level

`_fmt`
format string for debug entry

`...`
varargs used as in sprintf()

Return

- Address of written debug entry
- `NULL` if error

Description

floats and long long datatypes cannot be used as varargs.

debug_entry_t ***`debug_exception`**(debug_info_t *id, int level, void *data, int length)
writes binary debug entry to active debug area (if level <= actual debug level) and switches to next debug area

Parameters

`debug_info_t *id`
handle for debug log

`int level`
debug level

`void *data`
pointer to data for debug entry

`int length`
length of data in bytes

Return

- Address of written debug entry
- `NULL` if error

debug_entry_t ***`debug_int_exception`**(debug_info_t *id, int level, unsigned int tag)
writes unsigned int debug entry to active debug area (if level <= actual debug level) and switches to next debug area

Parameters

`debug_info_t *id`
handle for debug log

`int level`
debug level

`unsigned int tag`
integer value for debug entry

Return

- Address of written debug entry
- `NULL` if error

debug_entry_t ***`debug_long_exception`**(debug_info_t *id, int level, unsigned long tag)

writes long debug entry to active debug area (if level <= actual debug level) and switches to next debug area

Parameters

`debug_info_t *id`
handle for debug log

`int level`
debug level

`unsigned long tag`
long integer value for debug entry

Return

- Address of written debug entry
- `NULL` if error

debug_entry_t ***`debug_text_exception`**(debug_info_t *id, int level, const char *txt)

writes string debug entry in ascii format to active debug area (if level <= actual debug level) and switches to next debug area area

Parameters

`debug_info_t *id`
handle for debug log

`int level`
debug level

`const char *txt`
string for debug entry

Return

- Address of written debug entry
- `NULL` if error

`debug_sprintf_exception`

`debug_sprintf_exception (_id, _level, _fmt, ...)`

writes debug entry with format string and varargs (longs) to active debug area (if level <= actual debug level) and switches to next debug area.

Parameters

`_id`
handle for debug log

`_level`
debug level

`_fmt`
: format string for debug entry

`...`
: varargs used as in sprintf()

Return

- Address of written debug entry
- `NULL` if error

Description

floats and long long datatypes cannot be used as varargs.

`DEFINE_STATIC_DEBUG_INFO`

`DEFINE_STATIC_DEBUG_INFO (var, name, pages, nr_areas, buf_size, view)`

Define static debug_info_t

Parameters

`var`
: Name of debug_info_t variable

`name`
: Name of debug log (e.g. used for debugfs entry)

`pages`
: Number of pages per area

`nr_areas`
: Number of debug areas

`buf_size`
: Size of data area in each debug entry

`view`
: Pointer to debug view struct

Description

Define a static debug_info_t for early tracing. The associated debugfs log is automatically registered with the specified debug view.

Important: Users of this macro must not call any of the debug_register/_unregister() functions for this debug_info_t!

Note

Tracing will start with a fixed number of initial pages and areas. The debug area will be changed to use the specified numbers during arch_initcall.

13.6.4 Predefined views:

```
extern struct debug_view debug_hex_ascii_view;

extern struct debug_view debug_sprintf_view;
```

13.6.5 Examples

```
/*
 * hex_ascii-view Example
 */

#include <linux/init.h>
#include <asm/debug.h>

static debug_info_t *debug_info;

static int init(void)
{
    /* register 4 debug areas with one page each and 4 byte data field */

    debug_info = debug_register("test", 1, 4, 4 );
    debug_register_view(debug_info, &debug_hex_ascii_view);

    debug_text_event(debug_info, 4 , "one ");
    debug_int_exception(debug_info, 4, 4711);
    debug_event(debug_info, 3, &debug_info, 4);

    return 0;
}

static void cleanup(void)
{
    debug_unregister(debug_info);
}

module_init(init);
module_exit(cleanup);
```

```
/*
 * sprintf-view Example
 */

#include <linux/init.h>
#include <asm/debug.h>

static debug_info_t *debug_info;

static int init(void)
```

```
{
    /* register 4 debug areas with one page each and data field for */
    /* format string pointer + 2 varargs (= 3 * sizeof(long))       */

    debug_info = debug_register("test", 1, 4, sizeof(long) * 3);
    debug_register_view(debug_info, &debug_sprintf_view);

    debug_sprintf_event(debug_info, 2 , "first event in %s:%i\n",__FILE__,__
↪LINE__);
    debug_sprintf_exception(debug_info, 1, "pointer to debug info: %p\n",&
↪debug_info);

    return 0;
}

static void cleanup(void)
{
    debug_unregister(debug_info);
}

module_init(init);
module_exit(cleanup);
```

13.6.6 Debugfs Interface

Views to the debug logs can be investigated through reading the corresponding debugfs-files:

Example:

```
> ls /sys/kernel/debug/s390dbf/dasd
flush  hex_ascii  level pages
> cat /sys/kernel/debug/s390dbf/dasd/hex_ascii | sort -k2,2 -s
00 00974733272:680099 2 - 02 0006ad7e  07 ea 4a 90 | ....
00 00974733272:682210 2 - 02 0006ade6  46 52 45 45 | FREE
00 00974733272:682213 2 - 02 0006adf6  07 ea 4a 90 | ....
00 00974733272:682281 1 * 02 0006ab08  41 4c 4c 43 | EXCP
01 00974733272:682284 2 - 02 0006ab16  45 43 4b 44 | ECKD
01 00974733272:682287 2 - 02 0006ab28  00 00 00 04 | ....
01 00974733272:682289 2 - 02 0006ab3e  00 00 00 20 | ...
01 00974733272:682297 2 - 02 0006ad7e  07 ea 4a 90 | ....
01 00974733272:684384 2 - 00 0006ade6  46 52 45 45 | FREE
01 00974733272:684388 2 - 00 0006adf6  07 ea 4a 90 | ....
```

See section about predefined views for explanation of the above output!

13.6.7 Changing the debug level

Example:

```
> cat /sys/kernel/debug/s390dbf/dasd/level
3
> echo "5" > /sys/kernel/debug/s390dbf/dasd/level
> cat /sys/kernel/debug/s390dbf/dasd/level
5
```

13.6.8 Flushing debug areas

Debug areas can be flushed with piping the number of the desired area (0...n) to the debugfs file "flush". When using "-" all debug areas are flushed.

Examples:

1. Flush debug area 0:

   ```
   > echo "0" > /sys/kernel/debug/s390dbf/dasd/flush
   ```

2. Flush all debug areas:

   ```
   > echo "-" > /sys/kernel/debug/s390dbf/dasd/flush
   ```

13.6.9 Changing the size of debug areas

It is possible the change the size of debug areas through piping the number of pages to the debugfs file "pages". The resize request will also flush the debug areas.

Example:

Define 4 pages for the debug areas of debug feature "dasd":

```
> echo "4" > /sys/kernel/debug/s390dbf/dasd/pages
```

13.6.10 Stopping the debug feature

Example:

1. Check if stopping is allowed:

   ```
   > cat /proc/sys/s390dbf/debug_stoppable
   ```

2. Stop debug feature:

   ```
   > echo 0 > /proc/sys/s390dbf/debug_active
   ```

13.6.11 crash Interface

The `crash` tool since v5.1.0 has a built-in command `s390dbf` to display all the debug logs or export them to the file system. With this tool it is possible to investigate the debug logs on a live system and with a memory dump after a system crash.

13.6.12 Investigating raw memory

One last possibility to investigate the debug logs at a live system and after a system crash is to look at the raw memory under VM or at the Service Element. It is possible to find the anchor of the debug-logs through the `debug_area_first` symbol in the System map. Then one has to follow the correct pointers of the data-structures defined in debug.h and find the debug-areas in memory. Normally modules which use the debug feature will also have a global variable with the pointer to the debug-logs. Following this pointer it will also be possible to find the debug logs in memory.

For this method it is recommended to use '16 * x + 4' byte (x = 0..n) for the length of the data field in *`debug_register()`* in order to see the debug entries well formatted.

13.6.13 Predefined Views

There are two predefined views: hex_ascii and sprintf. The hex_ascii view shows the data field in hex and ascii representation (e.g. `45 43 4b 44 | ECKD`).

The sprintf view formats the debug entries in the same way as the sprintf function would do. The sprintf event/exception functions write to the debug entry a pointer to the format string (size = sizeof(long)) and for each vararg a long value. So e.g. for a debug entry with a format string plus two varargs one would need to allocate a (3 * sizeof(long)) byte data area in the *`debug_register()`* function.

IMPORTANT:

Using "%s" in sprintf event functions is dangerous. You can only use "%s" in the sprintf event functions, if the memory for the passed string is available as long as the debug feature exists. The reason behind this is that due to performance considerations only a pointer to the string is stored in the debug feature. If you log a string that is freed afterwards, you will get an OOPS when inspecting the debug feature, because then the debug feature will access the already freed memory.

NOTE:

If using the sprintf view do NOT use other event/exception functions than the sprintf-event and -exception functions.

The format of the hex_ascii and sprintf view is as follows:

- Number of area
- Timestamp (formatted as seconds and microseconds since 00:00:00 Coordinated Universal Time (UTC), January 1, 1970)
- level of debug entry
- Exception flag (* = Exception)
- Cpu-Number of calling task

- Return Address to caller
- data field

A typical line of the hex_ascii view will look like the following (first line is only for explanation and will not be displayed when 'cating' the view):

```
area  time           level exception cpu caller    data (hex + ascii)
--------------------------------------------------------------------------
00    00964419409:440690 1 -         00  88023fe
```

13.6.14 Defining views

Views are specified with the 'debug_view' structure. There are defined callback functions which are used for reading and writing the debugfs files:

```
struct debug_view {
      char name[DEBUG_MAX_PROCF_LEN];
      debug_prolog_proc_t* prolog_proc;
      debug_header_proc_t* header_proc;
      debug_format_proc_t* format_proc;
      debug_input_proc_t*  input_proc;
      void*                private_data;
};
```

where:

```
typedef int (debug_header_proc_t) (debug_info_t* id,
                                   struct debug_view* view,
                                   int area,
                                   debug_entry_t* entry,
                                   char* out_buf);

typedef int (debug_format_proc_t) (debug_info_t* id,
                                   struct debug_view* view, char* out_buf,
                                   const char* in_buf);
typedef int (debug_prolog_proc_t) (debug_info_t* id,
                                   struct debug_view* view,
                                   char* out_buf);
typedef int (debug_input_proc_t) (debug_info_t* id,
                                  struct debug_view* view,
                                  struct file* file, const char* user_buf,
                                  size_t in_buf_size, loff_t* offset);
```

The "private_data" member can be used as pointer to view specific data. It is not used by the debug feature itself.

The output when reading a debugfs file is structured like this:

```
"prolog_proc output"

"header_proc output 1"  "format_proc output 1"
```

```
"header_proc output 2"  "format_proc output 2"
"header_proc output 3"  "format_proc output 3"
...
```

When a view is read from the debugfs, the Debug Feature calls the 'prolog_proc' once for writing the prolog. Then 'header_proc' and 'format_proc' are called for each existing debug entry.

The input_proc can be used to implement functionality when it is written to the view (e.g. like with `echo "0" > /sys/kernel/debug/s390dbf/dasd/level`).

For header_proc there can be used the default function `debug_dflt_header_fn()` which is defined in debug.h. and which produces the same header output as the predefined views. E.g:

```
00 00964419409:440761 2 - 00 88023ec
```

In order to see how to use the callback functions check the implementation of the default views!

Example:

```
#include <asm/debug.h>

#define UNKNOWNSTR "data: %08x"

const char* messages[] =
{"This error...........\n",
 "That error...........\n",
 "Problem..............\n",
 "Something went wrong.\n",
 "Everything ok........\n",
 NULL
};

static int debug_test_format_fn(
   debug_info_t *id, struct debug_view *view,
   char *out_buf, const char *in_buf
)
{
  int i, rc = 0;

  if (id->buf_size >= 4) {
     int msg_nr = *((int*)in_buf);
     if (msg_nr < sizeof(messages) / sizeof(char*) - 1)
        rc += sprintf(out_buf, "%s", messages[msg_nr]);
     else
        rc += sprintf(out_buf, UNKNOWNSTR, msg_nr);
  }
  return rc;
}

struct debug_view debug_test_view = {
  "myview",                 /* name of view */
  NULL,                     /* no prolog */
  &debug_dflt_header_fn,    /* default header for each entry */
  &debug_test_format_fn,    /* our own format function */
  NULL,                     /* no input function */
  NULL                      /* no private data */
};
```

test:

```
debug_info_t *debug_info;
int i;
...
debug_info = debug_register("test", 0, 4, 4);
debug_register_view(debug_info, &debug_test_view);
for (i = 0; i < 10; i ++)
  debug_int_event(debug_info, 1, i);
```

```
> cat /sys/kernel/debug/s390dbf/test/myview
00 00964419734:611402 1 - 00 88042ca   This error...........
00 00964419734:611405 1 - 00 88042ca   That error...........
00 00964419734:611408 1 - 00 88042ca   Problem..............
00 00964419734:611411 1 - 00 88042ca   Something went wrong.
00 00964419734:611414 1 - 00 88042ca   Everything ok........
00 00964419734:611417 1 - 00 88042ca   data: 00000005
00 00964419734:611419 1 - 00 88042ca   data: 00000006
00 00964419734:611422 1 - 00 88042ca   data: 00000007
00 00964419734:611425 1 - 00 88042ca   data: 00000008
00 00964419734:611428 1 - 00 88042ca   data: 00000009
```

13.7 Adjunct Processor (AP) facility

13.7.1 Introduction

The Adjunct Processor (AP) facility is an IBM Z cryptographic facility comprised of three AP instructions and from 1 up to 256 PCIe cryptographic adapter cards. The AP devices provide cryptographic functions to all CPUs assigned to a linux system running in an IBM Z system LPAR.

The AP adapter cards are exposed via the AP bus. The motivation for vfio-ap is to make AP cards available to KVM guests using the VFIO mediated device framework. This implementation relies considerably on the s390 virtualization facilities which do most of the hard work of providing direct access to AP devices.

13.7.2 AP Architectural Overview

To facilitate the comprehension of the design, let's start with some definitions:

- AP adapter

  An AP adapter is an IBM Z adapter card that can perform cryptographic functions. There can be from 0 to 256 adapters assigned to an LPAR. Adapters assigned to the LPAR in which a linux host is running will be available to the linux host. Each adapter is identified by a number from 0 to 255; however, the maximum adapter number is determined by machine model and/or adapter type. When installed, an AP adapter is accessed by AP instructions executed by any CPU.

  The AP adapter cards are assigned to a given LPAR via the system's Activation Profile which can be edited via the HMC. When the linux host system is IPL'd in the LPAR, the AP bus detects the AP adapter cards assigned to the LPAR and creates a sysfs device for each assigned adapter. For example, if AP adapters 4 and 10 (0x0a) are assigned to the LPAR, the AP bus will create the following sysfs device entries:

  ```
  /sys/devices/ap/card04
  /sys/devices/ap/card0a
  ```

  Symbolic links to these devices will also be created in the AP bus devices sub-directory:

  ```
  /sys/bus/ap/devices/[card04]
  /sys/bus/ap/devices/[card04]
  ```

- AP domain

  An adapter is partitioned into domains. An adapter can hold up to 256 domains depending upon the adapter type and hardware configuration. A domain is identified by a number from 0 to 255; however, the maximum domain number is determined by machine model and/or adapter type.. A domain can be thought of as a set of hardware registers and memory used for processing AP commands. A domain can be configured with a secure private key used for clear key encryption. A domain is classified in one of two ways depending upon how it may be accessed:

  - Usage domains are domains that are targeted by an AP instruction to process an AP command.
  - Control domains are domains that are changed by an AP command sent to a usage domain; for example, to set the secure private key for the control domain.

  The AP usage and control domains are assigned to a given LPAR via the system's Activation Profile which can be edited via the HMC. When a linux host system is IPL'd in the LPAR, the AP bus module detects the AP usage and control domains assigned to the LPAR. The domain number of each usage domain and adapter number of each AP adapter are combined to create AP queue devices (see AP Queue section below). The domain number of each control domain will be represented in a bitmask and stored in a sysfs file /sys/bus/ap/ap_control_domain_mask. The bits in the mask, from most to least significant bit, correspond to domains 0-255.

- AP Queue

  An AP queue is the means by which an AP command is sent to a usage domain inside a specific adapter. An AP queue is identified by a tuple comprised of an AP adapter ID (APID) and an AP queue index (APQI). The APQI corresponds to a given usage domain number within the adapter. This tuple forms an AP Queue Number (APQN) uniquely identifying an AP queue. AP instructions include a field containing the APQN to identify the AP queue to which the AP command is to be sent for processing.

The AP bus will create a sysfs device for each APQN that can be derived from the cross product of the AP adapter and usage domain numbers detected when the AP bus module is loaded. For example, if adapters 4 and 10 (0x0a) and usage domains 6 and 71 (0x47) are assigned to the LPAR, the AP bus will create the following sysfs entries:

```
/sys/devices/ap/card04/04.0006
/sys/devices/ap/card04/04.0047
/sys/devices/ap/card0a/0a.0006
/sys/devices/ap/card0a/0a.0047
```

The following symbolic links to these devices will be created in the AP bus devices subdirectory:

```
/sys/bus/ap/devices/[04.0006]
/sys/bus/ap/devices/[04.0047]
/sys/bus/ap/devices/[0a.0006]
/sys/bus/ap/devices/[0a.0047]
```

- AP Instructions:

  There are three AP instructions:

  - NQAP: to enqueue an AP command-request message to a queue
  - DQAP: to dequeue an AP command-reply message from a queue
  - PQAP: to administer the queues

  AP instructions identify the domain that is targeted to process the AP command; this must be one of the usage domains. An AP command may modify a domain that is not one of the usage domains, but the modified domain must be one of the control domains.

13.7.3 AP and SIE

Let's now take a look at how AP instructions executed on a guest are interpreted by the hardware.

A satellite control block called the Crypto Control Block (CRYCB) is attached to our main hardware virtualization control block. The CRYCB contains an AP Control Block (APCB) that has three fields to identify the adapters, usage domains and control domains assigned to the KVM guest:

- The AP Mask (APM) field is a bit mask that identifies the AP adapters assigned to the KVM guest. Each bit in the mask, from left to right, corresponds to an APID from 0-255. If a bit is set, the corresponding adapter is valid for use by the KVM guest.
- The AP Queue Mask (AQM) field is a bit mask identifying the AP usage domains assigned to the KVM guest. Each bit in the mask, from left to right, corresponds to an AP queue index (APQI) from 0-255. If a bit is set, the corresponding queue is valid for use by the KVM guest.

- The AP Domain Mask field is a bit mask that identifies the AP control domains assigned to the KVM guest. The ADM bit mask controls which domains can be changed by an AP command-request message sent to a usage domain from the guest. Each bit in the mask, from left to right, corresponds to a domain from 0-255. If a bit is set, the corresponding domain can be modified by an AP command-request message sent to a usage domain.

If you recall from the description of an AP Queue, AP instructions include an APQN to identify the AP queue to which an AP command-request message is to be sent (NQAP and PQAP instructions), or from which a command-reply message is to be received (DQAP instruction). The validity of an APQN is defined by the matrix calculated from the APM and AQM; it is the Cartesian product of all assigned adapter numbers (APM) with all assigned queue indexes (AQM). For example, if adapters 1 and 2 and usage domains 5 and 6 are assigned to a guest, the APQNs (1,5), (1,6), (2,5) and (2,6) will be valid for the guest.

The APQNs can provide secure key functionality - i.e., a private key is stored on the adapter card for each of its domains - so each APQN must be assigned to at most one guest or to the linux host:

```
Example 1: Valid configuration:
------------------------------
Guest1: adapters 1,2  domains 5,6
Guest2: adapter  1,2  domain 7

This is valid because both guests have a unique set of APQNs:
   Guest1 has APQNs (1,5), (1,6), (2,5), (2,6);
   Guest2 has APQNs (1,7), (2,7)

Example 2: Valid configuration:
------------------------------
Guest1: adapters 1,2 domains 5,6
Guest2: adapters 3,4 domains 5,6

This is also valid because both guests have a unique set of APQNs:
   Guest1 has APQNs (1,5), (1,6), (2,5), (2,6);
   Guest2 has APQNs (3,5), (3,6), (4,5), (4,6)

Example 3: Invalid configuration:
--------------------------------
Guest1: adapters 1,2  domains 5,6
Guest2: adapter  1    domains 6,7

This is an invalid configuration because both guests have access to
APQN (1,6).
```

13.7.4 The Design

The design introduces three new objects:

1. AP matrix device
2. VFIO AP device driver (vfio_ap.ko)
3. VFIO AP mediated pass-through device

The VFIO AP device driver

The VFIO AP (vfio_ap) device driver serves the following purposes:

1. Provides the interfaces to secure APQNs for exclusive use of KVM guests.
2. Sets up the VFIO mediated device interfaces to manage a vfio_ap mediated device and creates the sysfs interfaces for assigning adapters, usage domains, and control domains comprising the matrix for a KVM guest.
3. Configures the APM, AQM and ADM in the APCB contained in the CRYCB referenced by a KVM guest's SIE state description to grant the guest access to a matrix of AP devices

Reserve APQNs for exclusive use of KVM guests

The following block diagram illustrates the mechanism by which APQNs are reserved:

```
                              +------------------+
               7 remove       |                  |
         +--------------------> cex4queue driver |
         |                    |                  |
         |                    +------------------+
         |
         |
         |                    +------------------+         +----------------+
         |  5 register driver |                  | 3 create |                |
         |   +---------------->   Device core    +--------->  matrix device |
         |   |                |                  |         |                |
         |   |                +--------^---------+         +----------------+
         |   |                         |
         |   |                         +-------------------+
         |   | +-----------------------------------+       |
         |   | |      4 register AP driver         |       | 2 register device
         |   | |                                   |       |
+--------+---+-v---+                      +--------+-------+-+
|                  |                      |                  |
|      ap_bus      +--------------------- >  vfio_ap driver  |
|                  |       8 probe        |                  |
+--------^---------+                      +--^--^------------+
6 edit   |                                   |  |
  apmask |     +-----------------------------+  | 11 mdev create
  aqmask |     |           1 modprobe           |
+--------+-----+---+           +----------------+-+        +----------------+
|                  |           |                  |10 create|     mediated   |
|      admin       |           | VFIO device core |--------->     matrix     |
|                  +           |                  |         |     device     |
+------+-+---------+           +--------^---------+         +--------^-------+
       | |                              |                            |
       | | 9 create vfio_ap-passthrough |                            |
       | +------------------------------+                            |
       +-------------------------------------------------------------+
                   12  assign adapter/domain/control domain
```

The process for reserving an AP queue for use by a KVM guest is:

1. The administrator loads the vfio_ap device driver
2. The vfio-ap driver during its initialization will register a single 'matrix' device with the device core. This will serve as the parent device for all vfio_ap mediated devices used to configure an AP matrix for a guest.
3. The /sys/devices/vfio_ap/matrix device is created by the device core
4. The vfio_ap device driver will register with the AP bus for AP queue devices of type 10 and higher (CEX4 and newer). The driver will provide the vfio_ap driver's probe and remove callback interfaces. Devices older than CEX4 queues are not supported to simplify the implementation by not needlessly complicating the design by supporting older devices that will go out of service in the relatively near future, and for which there are few older systems around on which to test.
5. The AP bus registers the vfio_ap device driver with the device core
6. The administrator edits the AP adapter and queue masks to reserve AP queues for use by the vfio_ap device driver.
7. The AP bus removes the AP queues reserved for the vfio_ap driver from the default zcrypt cex4queue driver.
8. The AP bus probes the vfio_ap device driver to bind the queues reserved for it.
9. The administrator creates a passthrough type vfio_ap mediated device to be used by a guest
10. The administrator assigns the adapters, usage domains and control domains to be exclusively used by a guest.

Set up the VFIO mediated device interfaces

The VFIO AP device driver utilizes the common interfaces of the VFIO mediated device core driver to:

- Register an AP mediated bus driver to add a vfio_ap mediated device to and remove it from a VFIO group.
- Create and destroy a vfio_ap mediated device
- Add a vfio_ap mediated device to and remove it from the AP mediated bus driver
- Add a vfio_ap mediated device to and remove it from an IOMMU group

The following high-level block diagram shows the main components and interfaces of the VFIO AP mediated device driver:

```
+-------------+
|             |
| +---------+ | mdev_register_driver() +--------------+
| |  Mdev   | +<-----------------------+              |
| |  bus    | |                        | vfio_mdev.ko |
| | driver  | +----------------------->+              |<-> VFIO user
| +---------+ |    probe()/remove()    +--------------+    APIs
|             |
|  MDEV CORE  |
|   MODULE    |
|   mdev.ko   |
| +---------+ | mdev_register_parent() +--------------+
| |Physical | +<-----------------------+              |
| | device  | |                        |  vfio_ap.ko  |<-> matrix
| |interface| +----------------------->+              |    device
| +---------+ |       callback         +--------------+
+-------------+
```

During initialization of the vfio_ap module, the matrix device is registered with an 'mdev_parent_ops' structure that provides the sysfs attribute structures, mdev functions and callback interfaces for managing the mediated matrix device.

- sysfs attribute structures:

  supported_type_groups

  The VFIO mediated device framework supports creation of user-defined mediated device types. These mediated device types are specified via the 'supported_type_groups' structure when a device is registered with the mediated device framework. The registration process creates the sysfs structures for each mediated device type specified in the 'mdev_supported_types' sub-directory of the device being registered. Along with the device type, the sysfs attributes of the mediated device type are provided.

  The VFIO AP device driver will register one mediated device type for passthrough devices:

  /sys/devices/vfio_ap/matrix/mdev_supported_types/vfio_ap-passthrough

  Only the read-only attributes required by the VFIO mdev framework will be provided:

  ```
  ... name
  ... device_api
  ... available_instances
  ... device_api
  ```

  Where:

  - **name:**
    specifies the name of the mediated device type
  - **device_api:**
    the mediated device type's API

  - **available_instances:**
    the number of vfio_ap mediated passthrough devices that can be created
  - **device_api:**
    specifies the VFIO API

mdev_attr_groups
This attribute group identifies the user-defined sysfs attributes of the mediated device. When a device is registered with the VFIO mediated device framework, the sysfs attribute files identified in the 'mdev_attr_groups' structure will be created in the vfio_ap mediated device's directory. The sysfs attributes for a vfio_ap mediated device are:

assign_adapter / unassign_adapter:
Write-only attributes for assigning/unassigning an AP adapter to/from the vfio_ap mediated device. To assign/unassign an adapter, the APID of the adapter is echoed into the respective attribute file.

assign_domain / unassign_domain:
Write-only attributes for assigning/unassigning an AP usage domain to/from the vfio_ap mediated device. To assign/unassign a domain, the domain number of the usage domain is echoed into the respective attribute file.

matrix:
A read-only file for displaying the APQNs derived from the Cartesian product of the adapter and domain numbers assigned to the vfio_ap mediated device.

guest_matrix:
A read-only file for displaying the APQNs derived from the Cartesian product of the adapter and domain numbers assigned to the APM and AQM fields respectively of the KVM guest's CRYCB. This may differ from the the APQNs assigned to the vfio_ap mediated device if any APQN does not reference a queue device bound to the vfio_ap device driver (i.e., the queue is not in the host's AP configuration).

assign_control_domain / unassign_control_domain:
Write-only attributes for assigning/unassigning an AP control domain to/from the vfio_ap mediated device. To assign/unassign a control domain, the ID of the domain to be assigned/unassigned is echoed into the respective attribute file.

control_domains:
A read-only file for displaying the control domain numbers assigned to the vfio_ap mediated device.

- functions:

create:
allocates the ap_matrix_mdev structure used by the vfio_ap driver to:

  - Store the reference to the KVM structure for the guest using the mdev
  - Store the AP matrix configuration for the adapters, domains, and control domains assigned via the corresponding sysfs attributes files
  - Store the AP matrix configuration for the adapters, domains and control domains available to a guest. A guest may not be provided access to APQNs referencing queue devices that do not exist, or are not bound to the vfio_ap device driver.

remove:

deallocates the vfio_ap mediated device's ap_matrix_mdev structure. This will be allowed only if a running guest is not using the mdev.

- callback interfaces

  open_device:
  The vfio_ap driver uses this callback to register a VFIO_GROUP_NOTIFY_SET_KVM notifier callback function for the matrix mdev devices. The open_device callback is invoked by userspace to connect the VFIO iommu group for the matrix mdev device to the MDEV bus. Access to the KVM structure used to configure the KVM guest is provided via this callback. The KVM structure, is used to configure the guest's access to the AP matrix defined via the vfio_ap mediated device's sysfs attribute files.

  close_device:
  unregisters the VFIO_GROUP_NOTIFY_SET_KVM notifier callback function for the matrix mdev device and deconfigures the guest's AP matrix.

  ioctl:
  this callback handles the VFIO_DEVICE_GET_INFO and VFIO_DEVICE_RESET ioctls defined by the vfio framework.

Configure the guest's AP resources

Configuring the AP resources for a KVM guest will be performed when the VFIO_GROUP_NOTIFY_SET_KVM notifier callback is invoked. The notifier function is called when userspace connects to KVM. The guest's AP resources are configured via its APCB by:

- Setting the bits in the APM corresponding to the APIDs assigned to the vfio_ap mediated device via its 'assign_adapter' interface.
- Setting the bits in the AQM corresponding to the domains assigned to the vfio_ap mediated device via its 'assign_domain' interface.
- Setting the bits in the ADM corresponding to the domain dIDs assigned to the vfio_ap mediated device via its 'assign_control_domains' interface.

The linux device model precludes passing a device through to a KVM guest that is not bound to the device driver facilitating its pass-through. Consequently, an APQN that does not reference a queue device bound to the vfio_ap device driver will not be assigned to a KVM guest's matrix. The AP architecture, however, does not provide a means to filter individual APQNs from the guest's matrix, so the adapters, domains and control domains assigned to vfio_ap mediated device via its sysfs 'assign_adapter', 'assign_domain' and 'assign_control_domain' interfaces will be filtered before providing the AP configuration to a guest:

- The APIDs of the adapters, the APQIs of the domains and the domain numbers of the control domains assigned to the matrix mdev that are not also assigned to the host's AP configuration will be filtered.
- Each APQN derived from the Cartesian product of the APIDs and APQIs assigned to the vfio_ap mdev is examined and if any one of them does not reference a queue device bound to the vfio_ap device driver, the adapter will not be plugged into the guest (i.e., the bit corresponding to its APID will not be set in the APM of the guest's APCB).

The CPU model features for AP

The AP stack relies on the presence of the AP instructions as well as three facilities: The AP Facilities Test (APFT) facility; the AP Query Configuration Information (QCI) facility; and the AP Queue Interruption Control facility. These features/facilities are made available to a KVM guest via the following CPU model features:

1. ap: Indicates whether the AP instructions are installed on the guest. This feature will be enabled by KVM only if the AP instructions are installed on the host.
2. apft: Indicates the APFT facility is available on the guest. This facility can be made available to the guest only if it is available on the host (i.e., facility bit 15 is set).
3. apqci: Indicates the AP QCI facility is available on the guest. This facility can be made available to the guest only if it is available on the host (i.e., facility bit 12 is set).
4. apqi: Indicates AP Queue Interruption Control faclity is available on the guest. This facility can be made available to the guest only if it is available on the host (i.e., facility bit 65 is set).

Note: If the user chooses to specify a CPU model different than the 'host' model to QEMU, the CPU model features and facilities need to be turned on explicitly; for example:

```
/usr/bin/qemu-system-s390x ... -cpu z13,ap=on,apqci=on,apft=on,apqi=on
```

A guest can be precluded from using AP features/facilities by turning them off explicitly; for example:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=off,apqci=off,apft=off,apqi=off
```

Note: If the APFT facility is turned off (apft=off) for the guest, the guest will not see any AP devices. The zcrypt device drivers on the guest that register for type 10 and newer AP devices - i.e., the cex4card and cex4queue device drivers - need the APFT facility to ascertain the facilities installed on a given AP device. If the APFT facility is not installed on the guest, then no adapter or domain devices will get created by the AP bus running on the guest because only type 10 and newer devices can be configured for guest use.

13.7.5 Example

Let's now provide an example to illustrate how KVM guests may be given access to AP facilities. For this example, we will show how to configure three guests such that executing the lszcrypt command on the guests would look like this:

Guest1

CARD.DOMAIN	TYPE	MODE
05	CEX5C	CCA-Coproc
05.0004	CEX5C	CCA-Coproc
05.00ab	CEX5C	CCA-Coproc
06	CEX5A	Accelerator
06.0004	CEX5A	Accelerator
06.00ab	CEX5A	Accelerator

Guest2

CARD.DOMAIN	TYPE	MODE
05	CEX5C	CCA-Coproc
05.0047	CEX5C	CCA-Coproc
05.00ff	CEX5C	CCA-Coproc

Guest3

CARD.DOMAIN	TYPE	MODE
06	CEX5A	Accelerator
06.0047	CEX5A	Accelerator
06.00ff	CEX5A	Accelerator

These are the steps:

1. Install the vfio_ap module on the linux host. The dependency chain for the vfio_ap module is: * iommu * s390 * zcrypt * vfio * vfio_mdev * vfio_mdev_device * KVM

   To build the vfio_ap module, the kernel build must be configured with the following Kconfig elements selected: * IOMMU_SUPPORT * S390 * ZCRYPT * VFIO * KVM

   If using make menuconfig select the following to build the vfio_ap module:

   ```
   -> Device Drivers
      -> IOMMU Hardware Support
         select S390 AP IOMMU Support
      -> VFIO Non-Privileged userspace driver framework
         -> Mediated device driver frramework
            -> VFIO driver for Mediated devices
   -> I/O subsystem
      -> VFIO support for AP devices
   ```

2. Secure the AP queues to be used by the three guests so that the host can not access them. To secure them, there are two sysfs files that specify bitmasks marking a subset of the APQN range as usable only by the default AP queue device drivers. All remaining APQNs are available for use by any other device driver. The vfio_ap device driver is currently the only non-default device driver. The location of the sysfs files containing the masks are:

   ```
   /sys/bus/ap/apmask
   /sys/bus/ap/aqmask
   ```

   The 'apmask' is a 256-bit mask that identifies a set of AP adapter IDs (APID). Each bit in the mask, from left to right, corresponds to an APID from 0-255. If a bit is set, the APID belongs to the subset of APQNs marked as available only to the default AP queue device drivers.

   The 'aqmask' is a 256-bit mask that identifies a set of AP queue indexes (APQI). Each bit in the mask, from left to right, corresponds to an APQI from 0-255. If a bit is set, the APQI belongs to the subset of APQNs marked as available only to the default AP queue device drivers.

The Cartesian product of the APIDs corresponding to the bits set in the apmask and the APQIs corresponding to the bits set in the aqmask comprise the subset of APQNs that can be used only by the host default device drivers. All other APQNs are available to the non-default device drivers such as the vfio_ap driver.

Take, for example, the following masks:

```
apmask:
0x7d00000000000000000000000000000000000000000000000000000000000000

aqmask:
0x8000000000000000000000000000000000000000000000000000000000000000
```

The masks indicate:

- Adapters 1, 2, 3, 4, 5, and 7 are available for use by the host default device drivers.
- Domain 0 is available for use by the host default device drivers
- The subset of APQNs available for use only by the default host device drivers are:

  (1,0), (2,0), (3,0), (4.0), (5,0) and (7,0)
- All other APQNs are available for use by the non-default device drivers.

The APQN of each AP queue device assigned to the linux host is checked by the AP bus against the set of APQNs derived from the Cartesian product of APIDs and APQIs marked as available to the default AP queue device drivers. If a match is detected, only the default AP queue device drivers will be probed; otherwise, the vfio_ap device driver will be probed.

By default, the two masks are set to reserve all APQNs for use by the default AP queue device drivers. There are two ways the default masks can be changed:

1. The sysfs mask files can be edited by echoing a string into the respective sysfs mask file in one of two formats:
   - An absolute hex string starting with 0x - like "0x12345678" - sets the mask. If the given string is shorter than the mask, it is padded with 0s on the right; for example, specifying a mask value of 0x41 is the same as specifying:

     ```
     0x4100000000000000000000000000000000000000000000000000000000000000
     ```

     Keep in mind that the mask reads from left to right, so the mask above identifies device numbers 1 and 7 (01000001).

     If the string is longer than the mask, the operation is terminated with an error (EINVAL).
   - Individual bits in the mask can be switched on and off by specifying each bit number to be switched in a comma separated list. Each bit number string must be prepended with a ('+') or minus ('-') to indicate the corresponding bit is to be switched on ('+') or off ('-'). Some valid values are:
     - "+0" switches bit 0 on
     - "-13" switches bit 13 off

- "+0x41" switches bit 65 on
- "-0xff" switches bit 255 off

The following example:

+0,-6,+0x47,-0xf0

Switches bits 0 and 71 (0x47) on

Switches bits 6 and 240 (0xf0) off

Note that the bits not specified in the list remain as they were before the operation.

2. The masks can also be changed at boot time via parameters on the kernel command line like this:

   ap.apmask=0xffff ap.aqmask=0x40

   This would create the following masks:

```
apmask:
0xffff000000000000000000000000000000000000000000000000000000000000

aqmask:
0x4000000000000000000000000000000000000000000000000000000000000000
```

   Resulting in these two pools:

```
default drivers pool:    adapter 0-15, domain 1
alternate drivers pool:  adapter 16-255, domains 0, 2-255
```

Note: Changing a mask such that one or more APQNs will be taken from a vfio_ap mediated device (see below) will fail with an error (EBUSY). A message is logged to the kernel ring buffer which can be viewed with the 'dmesg' command. The output identifies each APQN flagged as 'in use' and identifies the vfio_ap mediated device to which it is assigned; for example:

Userspace may not re-assign queue 05.0054 already assigned to 62177883-f1bb-47f0-914d-32a22e3a8804 Userspace may not re-assign queue 04.0054 already assigned to cef03c3c-903d-4ecc-9a83-40694cb8aee4

Securing the APQNs for our example

To secure the AP queues 05.0004, 05.0047, 05.00ab, 05.00ff, 06.0004, 06.0047, 06.00ab, and 06.00ff for use by the vfio_ap device driver, the corresponding APQNs can be removed from the default masks using either of the following commands:

```
echo -5,-6 > /sys/bus/ap/apmask

echo -4,-0x47,-0xab,-0xff > /sys/bus/ap/aqmask
```

Or the masks can be set as follows:

```
echo␣
↪0xf9ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff \
> apmask

echo␣
↪0xf7fffffffffffffffeffffffffffffffffffffffffeffffffffffffffffffffe \
> aqmask
```

This will result in AP queues 05.0004, 05.0047, 05.00ab, 05.00ff, 06.0004, 06.0047, 06.00ab, and 06.00ff getting bound to the vfio_ap device driver. The sysfs directory for the vfio_ap device driver will now contain symbolic links to the AP queue devices bound to it:

```
/sys/bus/ap
... [drivers]
...... [vfio_ap]
......... [05.0004]
......... [05.0047]
......... [05.00ab]
......... [05.00ff]
......... [06.0004]
......... [06.0047]
......... [06.00ab]
......... [06.00ff]
```

Keep in mind that only type 10 and newer adapters (i.e., CEX4 and later) can be bound to the vfio_ap device driver. The reason for this is to simplify the implementation by not needlessly complicating the design by supporting older devices that will go out of service in the relatively near future and for which there are few older systems on which to test.

The administrator, therefore, must take care to secure only AP queues that can be bound to the vfio_ap device driver. The device type for a given AP queue device can be read from the parent card's sysfs directory. For example, to see the hardware type of the queue 05.0004:

> cat /sys/bus/ap/devices/card05/hwtype

The hwtype must be 10 or higher (CEX4 or newer) in order to be bound to the vfio_ap device driver.

3. Create the mediated devices needed to configure the AP matrixes for the three guests and to provide an interface to the vfio_ap driver for use by the guests:

```
/sys/devices/vfio_ap/matrix/
--- [mdev_supported_types]
------ [vfio_ap-passthrough] (passthrough vfio_ap mediated device type)
--------- create
--------- [devices]
```

To create the mediated devices for the three guests:

```
uuidgen > create
uuidgen > create
uuidgen > create
```

```
or

echo $uuid1 > create
echo $uuid2 > create
echo $uuid3 > create
```

This will create three mediated devices in the [devices] subdirectory named after the UUID written to the create attribute file. We call them $uuid1, $uuid2 and $uuid3 and this is the sysfs directory structure after creation:

```
/sys/devices/vfio_ap/matrix/
--- [mdev_supported_types]
------ [vfio_ap-passthrough]
--------- [devices]
------------ [$uuid1]
--------------- assign_adapter
--------------- assign_control_domain
--------------- assign_domain
--------------- matrix
--------------- unassign_adapter
--------------- unassign_control_domain
--------------- unassign_domain

------------ [$uuid2]
--------------- assign_adapter
--------------- assign_control_domain
--------------- assign_domain
--------------- matrix
--------------- unassign_adapter
----------------unassign_control_domain
----------------unassign_domain

------------ [$uuid3]
--------------- assign_adapter
--------------- assign_control_domain
--------------- assign_domain
--------------- matrix
--------------- unassign_adapter
----------------unassign_control_domain
----------------unassign_domain
```

Note *: The vfio_ap mdevs do not persist across reboots unless the
mdevctl tool is used to create and persist them.

4. The administrator now needs to configure the matrixes for the mediated devices $uuid1 (for Guest1), $uuid2 (for Guest2) and $uuid3 (for Guest3).

   This is how the matrix is configured for Guest1:

```
echo 5 > assign_adapter
echo 6 > assign_adapter
echo 4 > assign_domain
echo 0xab > assign_domain
```

Control domains can similarly be assigned using the assign_control_domain sysfs file.

If a mistake is made configuring an adapter, domain or control domain, you can use the unassign_xxx files to unassign the adapter, domain or control domain.

To display the matrix configuration for Guest1:

```
cat matrix
```

To display the matrix that is or will be assigned to Guest1:

```
cat guest_matrix
```

This is how the matrix is configured for Guest2:

```
echo 5 > assign_adapter
echo 0x47 > assign_domain
echo 0xff > assign_domain
```

This is how the matrix is configured for Guest3:

```
echo 6 > assign_adapter
echo 0x47 > assign_domain
echo 0xff > assign_domain
```

In order to successfully assign an adapter:

- The adapter number specified must represent a value from 0 up to the maximum adapter number configured for the system. If an adapter number higher than the maximum is specified, the operation will terminate with an error (ENODEV).

  Note: The maximum adapter number can be obtained via the sysfs
  /sys/bus/ap/ap_max_adapter_id attribute file.

- Each APQN derived from the Cartesian product of the APID of the adapter being assigned and the APQIs of the domains previously assigned:
  - Must only be available to the vfio_ap device driver as specified in the sysfs /sys/bus/ap/apmask and /sys/bus/ap/aqmask attribute files. If even one APQN is reserved for use by the host device driver, the operation will terminate with an error (EADDRNOTAVAIL).
  - Must NOT be assigned to another vfio_ap mediated device. If even one APQN is assigned to another vfio_ap mediated device, the operation will terminate with an error (EBUSY).
  - Must NOT be assigned while the sysfs /sys/bus/ap/apmask and sys/bus/ap/aqmask attribute files are being edited or the operation may terminate with an error (EBUSY).

In order to successfully assign a domain:

- The domain number specified must represent a value from 0 up to the maximum domain number configured for the system. If a domain number higher than the maximum is specified, the operation will terminate with an error (ENODEV).

  Note: The maximum domain number can be obtained via the sysfs
  /sys/bus/ap/ap_max_domain_id attribute file.

  - Each APQN derived from the Cartesian product of the APQI of the domain being assigned and the APIDs of the adapters previously assigned:
  - Must only be available to the vfio_ap device driver as specified in the sysfs /sys/bus/ap/apmask and /sys/bus/ap/aqmask attribute files. If even one APQN is reserved for use by the host device driver, the operation will terminate with an error (EADDRNOTAVAIL).
  - Must NOT be assigned to another vfio_ap mediated device. If even one APQN is assigned to another vfio_ap mediated device, the operation will terminate with an error (EBUSY).
  - Must NOT be assigned while the sysfs /sys/bus/ap/apmask and sys/bus/ap/aqmask attribute files are being edited or the operation may terminate with an error (EBUSY).

In order to successfully assign a control domain:

- The domain number specified must represent a value from 0 up to the maximum domain number configured for the system. If a control domain number higher than the maximum is specified, the operation will terminate with an error (ENODEV).

5. Start Guest1:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=on,apqci=on,apft=on,apqi=on \
   -device vfio-ap,sysfsdev=/sys/devices/vfio_ap/matrix/$uuid1 ...
```

7. Start Guest2:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=on,apqci=on,apft=on,apqi=on \
   -device vfio-ap,sysfsdev=/sys/devices/vfio_ap/matrix/$uuid2 ...
```

7. Start Guest3:

```
/usr/bin/qemu-system-s390x ... -cpu host,ap=on,apqci=on,apft=on,apqi=on \
   -device vfio-ap,sysfsdev=/sys/devices/vfio_ap/matrix/$uuid3 ...
```

When the guest is shut down, the vfio_ap mediated devices may be removed.

Using our example again, to remove the vfio_ap mediated device $uuid1:

```
/sys/devices/vfio_ap/matrix/
   --- [mdev_supported_types]
   ------ [vfio_ap-passthrough]
   --------- [devices]
   ------------ [$uuid1]
   --------------- remove
```

```
echo 1 > remove
```

This will remove all of the matrix mdev device's sysfs structures including the mdev device itself. To recreate and reconfigure the matrix mdev device, all of the steps starting with step 3 will have to be performed again. Note that the remove will fail if a guest using the vfio_ap mdev is still running.

It is not necessary to remove a vfio_ap mdev, but one may want to remove it if no guest will use it during the remaining lifetime of the linux host. If the vfio_ap mdev is removed, one may want to also reconfigure the pool of adapters and queues reserved for use by the default drivers.

13.7.6 Hot plug/unplug support:

An adapter, domain or control domain may be hot plugged into a running KVM guest by assigning it to the vfio_ap mediated device being used by the guest if the following conditions are met:

- The adapter, domain or control domain must also be assigned to the host's AP configuration.
- Each APQN derived from the Cartesian product comprised of the APID of the adapter being assigned and the APQIs of the domains assigned must reference a queue device bound to the vfio_ap device driver.
- To hot plug a domain, each APQN derived from the Cartesian product comprised of the APQI of the domain being assigned and the APIDs of the adapters assigned must reference a queue device bound to the vfio_ap device driver.

An adapter, domain or control domain may be hot unplugged from a running KVM guest by unassigning it from the vfio_ap mediated device being used by the guest.

13.7.7 Over-provisioning of AP queues for a KVM guest:

Over-provisioning is defined herein as the assignment of adapters or domains to a vfio_ap mediated device that do not reference AP devices in the host's AP configuration. The idea here is that when the adapter or domain becomes available, it will be automatically hot-plugged into the KVM guest using the vfio_ap mediated device to which it is assigned as long as each new APQN resulting from plugging it in references a queue device bound to the vfio_ap device driver.

13.7.8 Limitations

Live guest migration is not supported for guests using AP devices without intervention by a system administrator. Before a KVM guest can be migrated, the vfio_ap mediated device must be removed. Unfortunately, it can not be removed manually (i.e., echo 1 > /sys/devices/vfio_ap/matrix/$UUID/remove) while the mdev is in use by a KVM guest. If the guest is being emulated by QEMU, its mdev can be hot unplugged from the guest in one of two ways:

1. If the KVM guest was started with libvirt, you can hot unplug the mdev via the following commands:

virsh detach-device <guestname> <path-to-device-xml>

For example, to hot unplug mdev 62177883-f1bb-47f0-914d-32a22e3a8804 from the guest named 'my-guest':

virsh detach-device my-guest ~/config/my-guest-hostdev.xml

The contents of my-guest-hostdev.xml:

```
      <hostdev mode='subsystem' type='mdev' managed='no' model='vfio-ap'>
        <source>
          <address uuid='62177883-f1bb-47f0-914d-32a22e3a8804'/>
        </source>
      </hostdev>


virsh qemu-monitor-command <guest-name> --hmp "device-del <device-id>"

For example, to hot unplug the vfio_ap mediated device identified on the
qemu command line with 'id=hostdev0' from the guest named 'my-guest':
```

```
virsh qemu-monitor-command my-guest --hmp "device_del hostdev0"
```

2. A vfio_ap mediated device can be hot unplugged by attaching the qemu monitor to the guest and using the following qemu monitor command:

   (QEMU) device-del id=<device-id>

   For example, to hot unplug the vfio_ap mediated device that was specified on the qemu command line with 'id=hostdev0' when the guest was started:

   (QEMU) device-del id=hostdev0

After live migration of the KVM guest completes, an AP configuration can be restored to the KVM guest by hot plugging a vfio_ap mediated device on the target system into the guest in one of two ways:

1. If the KVM guest was started with libvirt, you can hot plug a matrix mediated device into the guest via the following virsh commands:

   virsh attach-device <guestname> <path-to-device-xml>

   For example, to hot plug mdev 62177883-f1bb-47f0-914d-32a22e3a8804 into the guest named 'my-guest':

   virsh attach-device my-guest ~/config/my-guest-hostdev.xml

   The contents of my-guest-hostdev.xml:

```
         <hostdev mode='subsystem' type='mdev' managed='no' model='vfio-ap'>
           <source>
             <address uuid='62177883-f1bb-47f0-914d-32a22e3a8804'/>
           </source>
         </hostdev>


virsh qemu-monitor-command <guest-name> --hmp \
   "device_add vfio-ap,sysfsdev=<path-to-mdev>,id=<device-id>"

      For example, to hot plug the vfio_ap mediated device
      62177883-f1bb-47f0-914d-32a22e3a8804 into the guest named 'my-guest' with
      device-id hostdev0:

      virsh qemu-monitor-command my-guest --hmp \
      "device_add vfio-ap,\
      sysfsdev=/sys/devices/vfio_ap/matrix/62177883-f1bb-47f0-914d-32a22e3a8804,\
      id=hostdev0"
```

2. A vfio_ap mediated device can be hot plugged by attaching the qemu monitor to the guest and using the following qemu monitor command:

   (qemu) device_add “vfio-ap,sysfsdev=<path-to-mdev>,id=<device-id>”

   For example, to plug the vfio_ap mediated device 62177883-f1bb-47f0-914d-32a22e3a8804 into the guest with the device-id hostdev0:

   (QEMU) device-add "vfio-ap,sysfsdev=/sys/devices/vfio_ap/matrix/62177883-f1bb-47f0-914d-32a22e3a8804,id=hostdev0"

13.8 VFIO AP Locks Overview

This document describes the locks that are pertinent to the secure operation of the vfio_ap device driver. Throughout this document, the following variables will be used to denote instances of the structures herein described:

```
struct ap_matrix_dev *matrix_dev;
struct ap_matrix_mdev *matrix_mdev;
struct kvm *kvm;
```

13.8.1 The Matrix Devices Lock (drivers/s390/crypto/vfio_ap_private.h)

```
struct ap_matrix_dev {
      ...
      struct list_head mdev_list;
      struct mutex mdevs_lock;
      ...
}
```

The Matrix Devices Lock (matrix_dev->mdevs_lock) is implemented as a global mutex contained within the single object of struct ap_matrix_dev. This lock controls access to all fields contained within each matrix_mdev (matrix_dev->mdev_list). This lock must be held while reading from, writing to or using the data from a field contained within a matrix_mdev instance representing one of the vfio_ap device driver's mediated devices.

13.8.2 The KVM Lock (include/linux/kvm_host.h)

```
struct kvm {
      ...
      struct mutex lock;
      ...
}
```

The KVM Lock (kvm->lock) controls access to the state data for a KVM guest. This lock must be held by the vfio_ap device driver while one or more AP adapters, domains or control domains are being plugged into or unplugged from the guest.

The KVM pointer is stored in the in the matrix_mdev instance (matrix_mdev->kvm = kvm) containing the state of the mediated device that has been attached to the KVM guest.

13.8.3 The Guests Lock (drivers/s390/crypto/vfio_ap_private.h)

```
struct ap_matrix_dev {
      ...
      struct list_head mdev_list;
      struct mutex guests_lock;
      ...
}
```

The Guests Lock (matrix_dev->guests_lock) controls access to the matrix_mdev instances (matrix_dev->mdev_list) that represent mediated devices that hold the state for the mediated devices that have been attached to a KVM guest. This lock must be held:

1. To control access to the KVM pointer (matrix_mdev->kvm) while the vfio_ap device driver is using it to plug/unplug AP devices passed through to the KVM guest.
2. To add matrix_mdev instances to or remove them from matrix_dev->mdev_list. This is necessary to ensure the proper locking order when the list is perused to find an ap_matrix_mdev instance for the purpose of plugging/unplugging AP devices passed through to a KVM guest.

   For example, when a queue device is removed from the vfio_ap device driver, if the adapter is passed through to a KVM guest, it will have to be unplugged. In order to figure out whether the adapter is passed through, the matrix_mdev object to which the queue is assigned will have to be found. The KVM pointer (matrix_mdev->kvm) can then be used to determine if the mediated device is passed through (matrix_mdev->kvm != NULL) and if so, to unplug the adapter.

It is not necessary to take the Guests Lock to access the KVM pointer if the pointer is not used to plug/unplug devices passed through to the KVM guest; however, in this case, the Matrix Devices Lock (matrix_dev->mdevs_lock) must be held in order to access the KVM pointer since it is set and cleared under the protection of the Matrix Devices Lock. A case in point is the function that handles interception of the PQAP(AQIC) instruction sub-function. This handler needs to access the KVM pointer only for the purposes of setting or clearing IRQ resources, so only the matrix_dev->mdevs_lock needs to be held.

13.8.4 The PQAP Hook Lock (arch/s390/include/asm/kvm_host.h)

```
typedef int (*crypto_hook)(struct kvm_vcpu *vcpu);

struct kvm_s390_crypto {
      ...
      struct rw_semaphore pqap_hook_rwsem;
      crypto_hook *pqap_hook;
      ...
};
```

The PQAP Hook Lock is a r/w semaphore that controls access to the function pointer of the handler (`*kvm->arch.crypto.pqap_hook`) to invoke when the PQAP(AQIC) instruction sub-function is intercepted by the host. The lock must be held in write mode when pqap_hook value is set, and in read mode when the pqap_hook function is called.

13.9 vfio-ccw: the basic infrastructure

13.9.1 Introduction

Here we describe the vfio support for I/O subchannel devices for Linux/s390. Motivation for vfio-ccw is to passthrough subchannels to a virtual machine, while vfio is the means.

Different than other hardware architectures, s390 has defined a unified I/O access method, which is so called Channel I/O. It has its own access patterns:

- Channel programs run asynchronously on a separate (co)processor.
- The channel subsystem will access any memory designated by the caller in the channel program directly, i.e. there is no iommu involved.

Thus when we introduce vfio support for these devices, we realize it with a mediated device (mdev) implementation. The vfio mdev will be added to an iommu group, so as to make itself able to be managed by the vfio framework. And we add read/write callbacks for special vfio I/O regions to pass the channel programs from the mdev to its parent device (the real I/O subchannel device) to do further address translation and to perform I/O instructions.

This document does not intend to explain the s390 I/O architecture in every detail. More information/reference could be found here:

- A good start to know Channel I/O in general: https://en.wikipedia.org/wiki/Channel_I/O
- s390 architecture: s390 Principles of Operation manual (IBM Form. No. SA22-7832)
- The existing QEMU code which implements a simple emulated channel subsystem could also be a good reference. It makes it easier to follow the flow. qemu/hw/s390x/css.c

For vfio mediated device framework: - Documentation/driver-api/vfio-mediated-device.rst

13.9.2 Motivation of vfio-ccw

Typically, a guest virtualized via QEMU/KVM on s390 only sees paravirtualized virtio devices via the "Virtio Over Channel I/O (virtio-ccw)" transport. This makes virtio devices discoverable via standard operating system algorithms for handling channel devices.

However this is not enough. On s390 for the majority of devices, which use the standard Channel I/O based mechanism, we also need to provide the functionality of passing through them to a QEMU virtual machine. This includes devices that don't have a virtio counterpart (e.g. tape drives) or that have specific characteristics which guests want to exploit.

For passing a device to a guest, we want to use the same interface as everybody else, namely vfio. We implement this vfio support for channel devices via the vfio mediated device framework and the subchannel device driver "vfio_ccw".

13.9.3 Access patterns of CCW devices

s390 architecture has implemented a so called channel subsystem, that provides a unified view of the devices physically attached to the systems. Though the s390 hardware platform knows about a huge variety of different peripheral attachments like disk devices (aka. DASDs), tapes, communication controllers, etc. They can all be accessed by a well defined access method and they are presenting I/O completion a unified way: I/O interruptions.

All I/O requires the use of channel command words (CCWs). A CCW is an instruction to a specialized I/O channel processor. A channel program is a sequence of CCWs which are executed by the I/O channel subsystem. To issue a channel program to the channel subsystem, it is required to build an operation request block (ORB), which can be used to point out the format of the CCW and other control information to the system. The operating system signals the I/O channel subsystem to begin executing the channel program with a SSCH (start sub-channel) instruction. The central processor is then free to proceed with non-I/O instructions until interrupted. The I/O completion result is received by the interrupt handler in the form of interrupt response block (IRB).

Back to vfio-ccw, in short:

- ORBs and channel programs are built in guest kernel (with guest physical addresses).
- ORBs and channel programs are passed to the host kernel.
- Host kernel translates the guest physical addresses to real addresses and starts the I/O with issuing a privileged Channel I/O instruction (e.g SSCH).
- channel programs run asynchronously on a separate processor.
- I/O completion will be signaled to the host with I/O interruptions. And it will be copied as IRB to user space to pass it back to the guest.

13.9.4 Physical vfio ccw device and its child mdev

As mentioned above, we realize vfio-ccw with a mdev implementation.

Channel I/O does not have IOMMU hardware support, so the physical vfio-ccw device does not have an IOMMU level translation or isolation.

Subchannel I/O instructions are all privileged instructions. When handling the I/O instruction interception, vfio-ccw has the software policing and translation how the channel program is programmed before it gets sent to hardware.

Within this implementation, we have two drivers for two types of devices:

- The vfio_ccw driver for the physical subchannel device. This is an I/O subchannel driver for the real subchannel device. It realizes a group of callbacks and registers to the mdev framework as a parent (physical) device. As a consequence, mdev provides vfio_ccw a generic interface (sysfs) to create mdev devices. A vfio mdev could be created by vfio_ccw then and added to the mediated bus. It is the vfio device that added to an IOMMU group and a vfio group. vfio_ccw also provides an I/O region to accept channel program request from user space and store I/O interrupt result for user space to retrieve. To notify user space an I/O completion, it offers an interface to setup an eventfd fd for asynchronous signaling.
- The vfio_mdev driver for the mediated vfio ccw device. This is provided by the mdev framework. It is a vfio device driver for the mdev that created by vfio_ccw. It realizes a group of vfio device driver callbacks, adds itself to a vfio group, and registers itself to the mdev framework as a mdev driver. It uses a vfio iommu backend that uses the existing map and unmap ioctls, but rather than programming them into an IOMMU for a device, it simply stores the translations for use by later requests. This means that a device programmed in a VM with guest physical addresses can have the vfio kernel convert that address to process virtual address, pin the page and program the hardware with the host physical address in one step. For a mdev, the vfio iommu backend will not pin the pages during the VFIO_IOMMU_MAP_DMA ioctl. Mdev framework will only maintain a database of the iova<->vaddr mappings in this operation. And they export a vfio_pin_pages and a vfio_unpin_pages interfaces from the vfio iommu backend for the physical devices to pin and unpin pages by demand.

Below is a high Level block diagram:

```
+-------------+
|             |
| +---------+ | mdev_register_driver() +--------------+
| |  Mdev   | +<-----------------------+              |
| |  bus    | |                        | vfio_mdev.ko |
| | driver  | +----------------------->+              |<-> VFIO user
| +---------+ |    probe()/remove()    +--------------+    APIs
|             |
|  MDEV CORE  |
|   MODULE    |
|   mdev.ko   |
| +---------+ | mdev_register_parent() +--------------+
| |Physical | +<-----------------------+              |
| | device  | |                        |  vfio_ccw.ko |<-> subchannel
| |interface| +----------------------->+              |     device
| +---------+ |       callback        +--------------+
+-------------+
```

The process of how these work together.

1. vfio_ccw.ko drives the physical I/O subchannel, and registers the physical device (with callbacks) to mdev framework. When vfio_ccw probing the subchannel device, it registers device pointer and callbacks to the mdev framework. Mdev related file nodes under the device node in sysfs would be created for the subchannel device, namely 'mdev_create', 'mdev_destroy' and 'mdev_supported_types'.
2. Create a mediated vfio ccw device. Use the 'mdev_create' sysfs file, we need to manually create one (and only one for our case) mediated device.
3. vfio_mdev.ko drives the mediated ccw device. vfio_mdev is also the vfio device driver. It will probe the mdev and add it to an iommu_group and a vfio_group. Then we could pass through the mdev to a guest.

13.9.5 VFIO-CCW Regions

The vfio-ccw driver exposes MMIO regions to accept requests from and return results to userspace.

13.9.6 vfio-ccw I/O region

An I/O region is used to accept channel program request from user space and store I/O interrupt result for user space to retrieve. The definition of the region is:

```
struct ccw_io_region {
#define ORB_AREA_SIZE 12
        __u8    orb_area[ORB_AREA_SIZE];
#define SCSW_AREA_SIZE 12
        __u8    scsw_area[SCSW_AREA_SIZE];
#define IRB_AREA_SIZE 96
        __u8    irb_area[IRB_AREA_SIZE];
        __u32   ret_code;
} __packed;
```

This region is always available.

While starting an I/O request, orb_area should be filled with the guest ORB, and scsw_area should be filled with the SCSW of the Virtual Subchannel.

irb_area stores the I/O result.

ret_code stores a return code for each access of the region. The following values may occur:

0
: The operation was successful.

-EOPNOTSUPP
: The ORB specified transport mode or the SCSW specified a function other than the start function.

-EIO
: A request was issued while the device was not in a state ready to accept requests, or an internal error occurred.

-EBUSY
: The subchannel was status pending or busy, or a request is already active.

-EAGAIN
: A request was being processed, and the caller should retry.

-EACCES
: The channel path(s) used for the I/O were found to be not operational.

-ENODEV
: The device was found to be not operational.

-EINVAL
: The orb specified a chain longer than 255 ccws, or an internal error occurred.

13.9.7 vfio-ccw cmd region

The vfio-ccw cmd region is used to accept asynchronous instructions from userspace:

```
#define VFIO_CCW_ASYNC_CMD_HSCH (1 << 0)
#define VFIO_CCW_ASYNC_CMD_CSCH (1 << 1)
struct ccw_cmd_region {
       __u32 command;
       __u32 ret_code;
} __packed;
```

This region is exposed via region type VFIO_REGION_SUBTYPE_CCW_ASYNC_CMD.

Currently, CLEAR SUBCHANNEL and HALT SUBCHANNEL use this region.

command specifies the command to be issued; ret_code stores a return code for each access of the region. The following values may occur:

0
: The operation was successful.

-ENODEV
: The device was found to be not operational.

-EINVAL
: A command other than halt or clear was specified.

-EIO
: A request was issued while the device was not in a state ready to accept requests.

-EAGAIN
: A request was being processed, and the caller should retry.

-EBUSY
: The subchannel was status pending or busy while processing a halt request.

13.9.8 vfio-ccw schib region

The vfio-ccw schib region is used to return Subchannel-Information Block (SCHIB) data to userspace:

```
struct ccw_schib_region {
#define SCHIB_AREA_SIZE 52
       __u8 schib_area[SCHIB_AREA_SIZE];
} __packed;
```

This region is exposed via region type VFIO_REGION_SUBTYPE_CCW_SCHIB.

Reading this region triggers a STORE SUBCHANNEL to be issued to the associated hardware.

13.9.9 vfio-ccw crw region

The vfio-ccw crw region is used to return Channel Report Word (CRW) data to userspace:

```
struct ccw_crw_region {
       __u32 crw;
       __u32 pad;
} __packed;
```

This region is exposed via region type VFIO_REGION_SUBTYPE_CCW_CRW.

Reading this region returns a CRW if one that is relevant for this subchannel (e.g. one reporting changes in channel path state) is pending, or all zeroes if not. If multiple CRWs are pending (including possibly chained CRWs), reading this region again will return the next one, until no more CRWs are pending and zeroes are returned. This is similar to how STORE CHANNEL REPORT WORD works.

13.9.10 vfio-ccw operation details

vfio-ccw follows what vfio-pci did on the s390 platform and uses vfio-iommu-type1 as the vfio iommu backend.

- CCW translation APIs A group of APIs (start with *cp_*) to do CCW translation. The CCWs passed in by a user space program are organized with their guest physical memory addresses. These APIs will copy the CCWs into kernel space, and assemble a runnable kernel channel program by updating the guest physical addresses with their corresponding host physical addresses. Note that we have to use IDALs even for direct-access CCWs, as the referenced memory can be located anywhere, including above 2G.
- vfio_ccw device driver This driver utilizes the CCW translation APIs and introduces vfio_ccw, which is the driver for the I/O subchannel devices you want to pass through. vfio_ccw implements the following vfio ioctls:

  ```
  VFIO_DEVICE_GET_INFO
  VFIO_DEVICE_GET_IRQ_INFO
  VFIO_DEVICE_GET_REGION_INFO
  VFIO_DEVICE_RESET
  VFIO_DEVICE_SET_IRQS
  ```

This provides an I/O region, so that the user space program can pass a channel program to the kernel, to do further CCW translation before issuing them to a real device. This also provides the SET_IRQ ioctl to setup an event notifier to notify the user space program the I/O completion in an asynchronous way.

The use of vfio-ccw is not limited to QEMU, while QEMU is definitely a good example to get understand how these patches work. Here is a little bit more detail how an I/O request triggered by the QEMU guest will be handled (without error handling).

Explanation:

- Q1-Q7: QEMU side process.
- K1-K5: Kernel side process.

Q1.
Get I/O region info during initialization.

Q2.
Setup event notifier and handler to handle I/O completion.

… …

Q3.
Intercept a ssch instruction.

Q4.
Write the guest channel program and ORB to the I/O region.

K1.
Copy from guest to kernel.

K2.
Translate the guest channel program to a host kernel space channel program, which becomes runnable for a real device.

K3.
With the necessary information contained in the orb passed in by QEMU, issue the ccwchain to the device.

K4.
Return the ssch CC code.

Q5.
Return the CC code to the guest.

… …

K5.
Interrupt handler gets the I/O result and write the result to the I/O region.

K6.
Signal QEMU to retrieve the result.

Q6.
Get the signal and event handler reads out the result from the I/O region.

Q7.
Update the irb for the guest.

13.9.11 Limitations

The current vfio-ccw implementation focuses on supporting basic commands needed to implement block device functionality (read/write) of DASD/ECKD device only. Some commands may need special handling in the future, for example, anything related to path grouping.

DASD is a kind of storage device. While ECKD is a data recording format. More information for DASD and ECKD could be found here: https://en.wikipedia.org/wiki/Direct-access_storage_device https://en.wikipedia.org/wiki/Count_key_data

Together with the corresponding work in QEMU, we can bring the passed through DASD/ECKD device online in a guest now and use it as a block device.

The current code allows the guest to start channel programs via START SUBCHANNEL, and to issue HALT SUBCHANNEL, CLEAR SUBCHANNEL, and STORE SUBCHANNEL.

Currently all channel programs are prefetched, regardless of the p-bit setting in the ORB. As a result, self modifying channel programs are not supported. For this reason, IPL has to be handled as a special case by a userspace/guest program; this has been implemented in QEMU's s390-ccw bios as of QEMU 4.1.

vfio-ccw supports classic (command mode) channel I/O only. Transport mode (HPF) is not supported.

QDIO subchannels are currently not supported. Classic devices other than DASD/ECKD might work, but have not been tested.

13.9.12 Reference

1. ESA/s390 Principles of Operation manual (IBM Form. No. SA22-7832)
2. ESA/390 Common I/O Device Commands manual (IBM Form. No. SA22-7204)
3. https://en.wikipedia.org/wiki/Channel_I/O
4. *Linux for S/390 and zSeries*
5. Documentation/driver-api/vfio.rst
6. Documentation/driver-api/vfio-mediated-device.rst

13.10 The s390 SCSI dump tool (zfcpdump)

System z machines (z900 or higher) provide hardware support for creating system dumps on SCSI disks. The dump process is initiated by booting a dump tool, which has to create a dump of the current (probably crashed) Linux image. In order to not overwrite memory of the crashed Linux with data of the dump tool, the hardware saves some memory plus the register sets of the boot CPU before the dump tool is loaded. There exists an SCLP hardware interface to obtain the saved memory afterwards. Currently 32 MB are saved.

This zfcpdump implementation consists of a Linux dump kernel together with a user space dump tool, which are loaded together into the saved memory region below 32 MB. zfcpdump is installed on a SCSI disk using zipl (as contained in the s390-tools package) to make the device bootable. The operator of a Linux system can then trigger a SCSI dump by booting the SCSI disk, where zfcpdump resides on.

The user space dump tool accesses the memory of the crashed system by means of the /proc/vmcore interface. This interface exports the crashed system's memory and registers in ELF core dump format. To access the memory which has been saved by the hardware SCLP requests will be created at the time the data is needed by /proc/vmcore. The tail part of the crashed systems memory which has not been stashed by hardware can just be copied from real memory.

To build a dump enabled kernel the kernel config option CONFIG_CRASH_DUMP has to be set.

To get a valid zfcpdump kernel configuration use "make zfcpdump_defconfig".

The s390 zipl tool looks for the zfcpdump kernel and optional initrd/initramfs under the following locations:

- kernel: <zfcpdump directory>/zfcpdump.image
- ramdisk: <zfcpdump directory>/zfcpdump.rd

The zfcpdump directory is defined in the s390-tools package.

The user space application of zfcpdump can reside in an intitramfs or an initrd. It can also be included in a built-in kernel initramfs. The application reads from /proc/vmcore or zcore/mem and writes the system dump to a SCSI disk.

The s390-tools package version 1.24.0 and above builds an external zfcpdump initramfs with a user space application that writes the dump to a SCSI partition.

For more information on how to use zfcpdump refer to the s390 'Using the Dump Tools' book, which is available from IBM Knowledge Center: https://www.ibm.com/support/knowledgecenter/linuxonibm/liaaf/lnz_r_dt.html

13.11 S/390 common I/O-Layer

13.11.1 command line parameters, procfs and debugfs entries

Command line parameters

- ccw_timeout_log

  Enable logging of debug information in case of ccw device timeouts.

- cio_ignore = device[,device[,..]]

      device := {all | [!]ipldev | [!]condev | [!]<devno> | [!]<devno>-<devno>}

  The given devices will be ignored by the common I/O-layer; no detection and device sensing will be done on any of those devices. The subchannel to which the device in question is attached will be treated as if no device was attached.

  An ignored device can be un-ignored later; see the "/proc entries"-section for details.

  The devices must be given either as bus ids (0.x.abcd) or as hexadecimal device numbers (0xabcd or abcd, for 2.4 backward compatibility). If you give a device number 0xabcd, it will be interpreted as 0.0.abcd.

  You can use the 'all' keyword to ignore all devices. The 'ipldev' and 'condev' keywords can be used to refer to the CCW based boot device and CCW console device respectively (these are probably useful only when combined with the '!' operator). The '!' operator will cause the I/O-layer to _not_ ignore a device. The command line is parsed from left to right.

For example:

```
cio_ignore=0.0.0023-0.0.0042,0.0.4711
```

will ignore all devices ranging from 0.0.0023 to 0.0.0042 and the device 0.0.4711, if detected.

As another example:

```
cio_ignore=all,!0.0.4711,!0.0.fd00-0.0.fd02
```

will ignore all devices but 0.0.4711, 0.0.fd00, 0.0.fd01, 0.0.fd02.

By default, no devices are ignored.

/proc entries

- /proc/cio_ignore

  Lists the ranges of devices (by bus id) which are ignored by common I/O.

  You can un-ignore certain or all devices by piping to /proc/cio_ignore. "free all" will un-ignore all ignored devices, "free <device range>, <device range>, …" will un-ignore the specified devices.

  For example, if devices 0.0.0023 to 0.0.0042 and 0.0.4711 are ignored,

  - echo free 0.0.0030-0.0.0032 > /proc/cio_ignore will un-ignore devices 0.0.0030 to 0.0.0032 and will leave devices 0.0.0023 to 0.0.002f, 0.0.0033 to 0.0.0042 and 0.0.4711 ignored;
  - echo free 0.0.0041 > /proc/cio_ignore will furthermore un-ignore device 0.0.0041;
  - echo free all > /proc/cio_ignore will un-ignore all remaining ignored devices.

  When a device is un-ignored, device recognition and sensing is performed and the device driver will be notified if possible, so the device will become available to the system. Note that un-ignoring is performed asynchronously.

  You can also add ranges of devices to be ignored by piping to /proc/cio_ignore; "add <device range>, <device range>, …" will ignore the specified devices.

  Note: While already known devices can be added to the list of devices to be
  ignored, there will be no effect on then. However, if such a device disappears and then reappears, it will then be ignored. To make known devices go away, you need the "purge" command (see below).

  For example:

  ```
  "echo add 0.0.a000-0.0.accc, 0.0.af00-0.0.afff > /proc/cio_ignore"
  ```

  will add 0.0.a000-0.0.accc and 0.0.af00-0.0.afff to the list of ignored devices.

  You can remove already known but now ignored devices via:

```
"echo purge > /proc/cio_ignore"
```

All devices ignored but still registered and not online (= not in use) will be deregistered and thus removed from the system.

The devices can be specified either by bus id (0.x.abcd) or, for 2.4 backward compatibility, by the device number in hexadecimal (0xabcd or abcd). Device numbers given as 0xabcd will be interpreted as 0.0.abcd.

- /proc/cio_settle

  A write request to this file is blocked until all queued cio actions are handled. This will allow userspace to wait for pending work affecting device availability after changing cio_ignore or the hardware configuration.

- For some of the information present in the /proc filesystem in 2.4 (namely, /proc/subchannels and /proc/chpids), see driver-model.txt. Information formerly in /proc/irq_count is now in /proc/interrupts.

debugfs entries

- /sys/kernel/debug/s390dbf/cio_*/ (S/390 debug feature)

  Some views generated by the debug feature to hold various debug outputs.

  - /sys/kernel/debug/s390dbf/cio_crw/sprintf Messages from the processing of pending channel report words (machine check handling).
  - /sys/kernel/debug/s390dbf/cio_msg/sprintf Various debug messages from the common I/O-layer.
  - /sys/kernel/debug/s390dbf/cio_trace/hex_ascii Logs the calling of functions in the common I/O-layer and, if applicable, which subchannel they were called for, as well as dumps of some data structures (like irb in an error case).

  The level of logging can be changed to be more or less verbose by piping to /sys/kernel/debug/s390dbf/cio_*/level a number between 0 and 6; see the documentation on the S/390 debug feature (*S390 Debug Feature*) for details.

13.12 S/390 PCI

Authors:

- Pierre Morel

Copyright, IBM Corp. 2020

13.12.1 Command line parameters and debugfs entries

Command line parameters

- nomio

  Do not use PCI Mapped I/O (MIO) instructions.

- norid

  Ignore the RID field and force use of one PCI domain per PCI function.

debugfs entries

The S/390 debug feature (s390dbf) generates views to hold various debug results in sysfs directories of the form:

- /sys/kernel/debug/s390dbf/pci_*/

For example:

  - /sys/kernel/debug/s390dbf/pci_msg/sprintf Holds messages from the processing of PCI events, like machine check handling and setting of global functionality, like UID checking.

  Change the level of logging to be more or less verbose by piping a number between 0 and 6 to /sys/kernel/debug/s390dbf/pci_*/level. For details, see the documentation on the S/390 debug feature at *S390 Debug Feature*.

13.12.2 Sysfs entries

Entries specific to zPCI functions and entries that hold zPCI information.

- /sys/bus/pci/slots/XXXXXXXX

  The slot entries are set up using the function identifier (FID) of the PCI function. The format depicted as XXXXXXXX above is 8 hexadecimal digits with 0 padding and lower case hexadecimal digits.

  – /sys/bus/pci/slots/XXXXXXXX/power

  A physical function that currently supports a virtual function cannot be powered off until all virtual functions are removed with: echo 0 > /sys/bus/pci/devices/XXXX:XX:XX.X/sriov_numvf

- /sys/bus/pci/devices/XXXX:XX:XX.X/

  – function_id A zPCI function identifier that uniquely identifies the function in the Z server.

  – function_handle Low-level identifier used for a configured PCI function. It might be useful for debugging.

  – pchid Model-dependent location of the I/O adapter.

  – pfgid PCI function group ID, functions that share identical functionality use a common identifier. A PCI group defines interrupts, IOMMU, IOTLB, and DMA specifics.

  - vfn The virtual function number, from 1 to N for virtual functions, 0 for physical functions.
  - pft The PCI function type
  - port The port corresponds to the physical port the function is attached to. It also gives an indication of the physical function a virtual function is attached to.
  - uid The user identifier (UID) may be defined as part of the machine configuration or the z/VM or KVM guest configuration. If the accompanying uid_is_unique attribute is 1 the platform guarantees that the UID is unique within that instance and no devices with the same UID can be attached during the lifetime of the system.
  - uid_is_unique Indicates whether the user identifier (UID) is guaranteed to be and remain unique within this Linux instance.
  - pfip/segmentX The segments determine the isolation of a function. They correspond to the physical path to the function. The more the segments are different, the more the functions are isolated.

13.12.3 Enumeration and hotplug

The PCI address consists of four parts: domain, bus, device and function, and is of this form: DDDD:BB:dd.f

- When not using multi-functions (norid is set, or the firmware does not support multi-functions):
  - There is only one function per domain.
  - The domain is set from the zPCI function's UID as defined during the LPAR creation.
- When using multi-functions (norid parameter is not set), zPCI functions are addressed differently:
  - There is still only one bus per domain.
  - There can be up to 256 functions per bus.
  - The domain part of the address of all functions for a multi-Function device is set from the zPCI function's UID as defined in the LPAR creation for the function zero.
  - New functions will only be ready for use after the function zero (the function with devfn 0) has been enumerated.

13.13 ibm 3270 changelog

```
ChangeLog for the UTS Global 3270-support patch

Sep 2002:       Get bootup colors right on 3270 console
        * In tubttybld.c, substantially revise ESC processing so that
          ESC sequences (especially coloring ones) and the strings
          they affect work as right as 3270 can get them.  Also, set
          screen height to omit the two rows used for input area, in
          tty3270_open() in tubtty.c.
```

```
Sep 2002:       Dynamically get 3270 input buffer
        * Oversize 3270 screen widths may exceed GEOM_MAXINPLEN columns,
          so get input-area buffer dynamically when sizing the device in
          tubmakemin() in tuball.c (if it's the console) or tty3270_open()
          in tubtty.c (if needed).  Change tubp->tty_input to be a
          pointer rather than an array, in tubio.h.

Sep 2002:       Fix tubfs kmalloc()s
        * Do read and write lengths correctly in fs3270_read()
          and fs3270_write(), while never asking kmalloc()
          for more than 0x800 bytes.  Affects tubfs.c and tubio.h.

Sep 2002:       Recognize 3270 control unit type 3174
        * Recognize control-unit type 0x3174 as well as 0x327?.
          The IBM 2047 device emulates a 3174 control unit.
          Modularize control-unit recognition in tuball.c by
          adding and invoking new tub3270_is_ours().

Apr 2002:       Fix 3270 console reboot loop
        * (Belated log entry) Fixed reboot loop if 3270 console,
          in tubtty.c:ttu3270_bh().

Feb 6, 2001:
        * This changelog is new
        * tub3270 now supports 3270 console:
                Specify y for CONFIG_3270 and y for CONFIG_3270_CONSOLE.
                Support for 3215 will not appear if 3270 console support
                is chosen.
                NOTE:  The default is 3270 console support, NOT 3215.
        * the components are remodularized: added source modules are
          tubttybld.c and tubttyscl.c, for screen-building code and
          scroll-timeout code.
        * tub3270 source for this (2.4.0) version is #ifdeffed to
          build with both 2.4.0 and 2.2.16.2.
        * color support and minimal other ESC-sequence support is added.
```

13.14 ibm 3270 config3270.sh

```
#!/bin/sh
#
# config3270 -- Autoconfigure /dev/3270/* and /etc/inittab
#
#       Usage:
#               config3270
#
#       Output:
#               /tmp/mkdev3270
#
#       Operation:
```

```
#                1. Run this script
#                2. Run the script it produces: /tmp/mkdev3270
#                3. Issue "telinit q" or reboot, as appropriate.
#
P=/proc/tty/driver/tty3270
ROOT=
D=$ROOT/dev
SUBD=3270
TTY=$SUBD/tty
TUB=$SUBD/tub
SCR=$ROOT/tmp/mkdev3270
SCRTMP=$SCR.a
GETTYLINE=:2345:respawn:/sbin/mingetty
INITTAB=$ROOT/etc/inittab
NINITTAB=$ROOT/etc/NEWinittab
OINITTAB=$ROOT/etc/OLDinittab
ADDNOTE=\\"# Additional mingettys for the 3270/tty* driver, tub3270 ---\\"

if ! ls $P > /dev/null 2>&1; then
        modprobe tub3270 > /dev/null 2>&1
fi
ls $P > /dev/null 2>&1 || exit 1

# Initialize two files, one for /dev/3270 commands and one
# to replace the /etc/inittab file (old one saved in OLDinittab)
echo "#!/bin/sh" > $SCR || exit 1
echo " " >> $SCR
echo "# Script built by /sbin/config3270" >> $SCR
if [ ! -d /dev/dasd ]; then
        echo rm -rf "$D/$SUBD/*" >> $SCR
fi
echo "grep -v $TTY $INITTAB > $NINITTAB" > $SCRTMP || exit 1
echo "echo $ADDNOTE >> $NINITTAB" >> $SCRTMP
if [ ! -d /dev/dasd ]; then
        echo mkdir -p $D/$SUBD >> $SCR
fi

# Now query the tub3270 driver for 3270 device information
# and add appropriate mknod and mingetty lines to our files
echo what=config > $P
while read devno maj min;do
        if [ $min = 0 ]; then
                fsmaj=$maj
                if [ ! -d /dev/dasd ]; then
                        echo mknod $D/$TUB c $fsmaj 0 >> $SCR
                        echo chmod 666 $D/$TUB >> $SCR
                fi
        elif [ $maj = CONSOLE ]; then
                if [ ! -d /dev/dasd ]; then
                        echo mknod $D/$TUB$devno c $fsmaj $min >> $SCR
                fi
        else
                if [ ! -d /dev/dasd ]; then
                        echo mknod $D/$TTY$devno c $maj $min >>$SCR
                        echo mknod $D/$TUB$devno c $fsmaj $min >> $SCR
                fi
                echo "echo t$min$GETTYLINE $TTY$devno >> $NINITTAB" >> $SCRTMP
        fi
done < $P

echo mv $INITTAB $OINITTAB >> $SCRTMP || exit 1
echo mv $NINITTAB $INITTAB >> $SCRTMP
cat $SCRTMP >> $SCR
rm $SCRTMP
exit 0
```

13.15 Feature status on s390 architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	ok
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok
core	thread-info-in-task	THREAD_INFO_IN_TASK	ok
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	ok
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	ok
debug	KASAN	HAVE_ARCH_KASAN	ok
debug	kcov	ARCH_HAS_KCOV	ok
debug	kgdb	HAVE_ARCH_KGDB	TODO
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok
debug	kprobes	HAVE_KPROBES	ok
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	ok
debug	kretprobes	HAVE_KRETPROBES	ok
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	TODO
debug	uprobes	ARCH_SUPPORTS_UPROBES	ok
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	ok
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	ok
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	TODO
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	TODO
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok
perf	perf-regs	HAVE_PERF_REGS	ok
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	ok
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	ok

Table 1 – continued fron

Subsystem	Feature	Kconfig	Status
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	ok
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	TODO
time	clockevents	!LEGACY_TIMER_TICK	ok
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	---
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	TODO
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	ok
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	TODO
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	ok
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO
vm	ioremap_prot	HAVE_IOREMAP_PROT	ok
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	ok

CHAPTER

FOURTEEN

SUPERH INTERFACES GUIDE

Author
Paul Mundt

14.1 DeviceTree Booting

Device-tree compatible SH bootloaders are expected to provide the physical address of the device tree blob in r4. Since legacy bootloaders did not guarantee any particular initial register state, kernels built to inter-operate with old bootloaders must either use a builtin DTB or select a legacy board option (something other than CONFIG_SH_DEVICE_TREE) that does not use device tree. Support for the latter is being phased out in favor of device tree.

14.2 Adding a new board to LinuxSH

Paul Mundt <lethal@linux-sh.org>

This document attempts to outline what steps are necessary to add support for new boards to the LinuxSH port under the new 2.5 and 2.6 kernels. This also attempts to outline some of the noticeable changes between the 2.4 and the 2.5/2.6 SH backend.

14.2.1 1. New Directory Structure

The first thing to note is the new directory structure. Under 2.4, most of the board-specific code (with the exception of stboards) ended up in arch/sh/kernel/ directly, with board-specific headers ending up in include/asm-sh/. For the new kernel, things are broken out by board type, companion chip type, and CPU type. Looking at a tree view of this directory hierarchy looks like the following:

Board-specific code:

```
.
|-- arch
|   `-- sh
|       `-- boards
|           |-- adx
|           |   `-- board-specific files
|           |-- bigsur
|           |   `-- board-specific files
|           |
|           ... more boards here ...
|
`-- include
    `-- asm-sh
        |-- adx
        |   `-- board-specific headers
        |-- bigsur
        |   `-- board-specific headers
        |
        .. more boards here ...
```

Next, for companion chips:

```
.
`-- arch
    `-- sh
        `-- cchips
            `-- hd6446x
                `-- hd64461
                    `-- cchip-specific files
```

... and so on. Headers for the companion chips are treated the same way as board-specific headers. Thus, include/asm-sh/hd64461 is home to all of the hd64461-specific headers.

Finally, CPU family support is also abstracted:

```
.
|-- arch
|   `-- sh
|       |-- kernel
|       |   `-- cpu
|       |       |-- sh2
|       |       |   `-- SH-2 generic files
|       |       |-- sh3
|       |       |   `-- SH-3 generic files
|       |       `-- sh4
|       |           `-- SH-4 generic files
|       `-- mm
|           `-- This is also broken out per CPU family, so each family can
|               have their own set of cache/tlb functions.
|
`-- include
    `-- asm-sh
        |-- cpu-sh2
        |   `-- SH-2 specific headers
        |-- cpu-sh3
        |   `-- SH-3 specific headers
        `-- cpu-sh4
            `-- SH-4 specific headers
```

It should be noted that CPU subtypes are _not_ abstracted. Thus, these still need to be dealt with by the CPU family specific code.

14.2.2 2. Adding a New Board

The first thing to determine is whether the board you are adding will be isolated, or whether it will be part of a family of boards that can mostly share the same board-specific code with minor differences.

In the first case, this is just a matter of making a directory for your board in arch/sh/boards/ and adding rules to hook your board in with the build system (more on this in the next section). However, for board families it makes more sense to have a common top-level arch/sh/boards/ directory and then populate that with sub-directories for each member of the family. Both the Solution Engine and the hp6xx boards are an example of this.

After you have setup your new arch/sh/boards/ directory, remember that you should also add a directory in include/asm-sh for headers localized to this board (if there are going to be more than one). In order to interoperate seamlessly with the build system, it's best to have this directory the same as the arch/sh/boards/ directory name, though if your board is again part of a family, the build system has ways of dealing with this (via incdir-y overloading), and you can feel free to name the directory after the family member itself.

There are a few things that each board is required to have, both in the arch/sh/boards and the include/asm-sh/ hierarchy. In order to better explain this, we use some examples for adding an imaginary board. For setup code, we're required at the very least to provide definitions for get_system_type() and platform_setup(). For our imaginary board, this might look something like:

```
/*
* arch/sh/boards/vapor/setup.c - Setup code for imaginary board
*/
#include <linux/init.h>

const char *get_system_type(void)
{
        return "FooTech Vaporboard";
}

int __init platform_setup(void)
{
        /*
        * If our hardware actually existed, we would do real
        * setup here. Though it's also sane to leave this empty
        * if there's no real init work that has to be done for
        * this board.
        */

        /* Start-up imaginary PCI ... */

        /* And whatever else ... */

        return 0;
}
```

Our new imaginary board will also have to tie into the machvec in order for it to be of any use.

machvec functions fall into a number of categories:

- I/O functions to IO memory (inb etc) and PCI/main memory (readb etc).
- I/O mapping functions (ioport_map, ioport_unmap, etc).
- a 'heartbeat' function.
- PCI and IRQ initialization routines.
- Consistent allocators (for boards that need special allocators, particularly for allocating out of some board-specific SRAM for DMA handles).

There are machvec functions added and removed over time, so always be sure to consult include/asm-sh/machvec.h for the current state of the machvec.

The kernel will automatically wrap in generic routines for undefined function pointers in the machvec at boot time, as machvec functions are referenced unconditionally throughout most of the tree. Some boards have incredibly sparse machvecs (such as the dreamcast and sh03), whereas others must define virtually everything (rts7751r2d).

Adding a new machine is relatively trivial (using vapor as an example):

If the board-specific definitions are quite minimalistic, as is the case for the vast majority of boards, simply having a single board-specific header is sufficient.

- add a new file include/asm-sh/vapor.h which contains prototypes for any machine specific IO functions prefixed with the machine name, for example vapor_inb. These will be needed when filling out the machine vector.

  Note that these prototypes are generated automatically by setting __IO_PREFIX to something sensible. A typical example would be:

  ```
  #define __IO_PREFIX vapor
  #include <asm/io_generic.h>
  ```

  somewhere in the board-specific header. Any boards being ported that still have a legacy io.h should remove it entirely and switch to the new model.

- Add machine vector definitions to the board's setup.c. At a bare minimum, this must be defined as something like:

  ```
  struct sh_machine_vector mv_vapor __initmv = {
          .mv_name = "vapor",
  };
  ALIAS_MV(vapor)
  ```

- finally add a file arch/sh/boards/vapor/io.c, which contains definitions of the machine specific io functions (if there are enough to warrant it).

14.2.3 3. Hooking into the Build System

Now that we have the corresponding directories setup, and all of the board-specific code is in place, it's time to look at how to get the whole mess to fit into the build system.

Large portions of the build system are now entirely dynamic, and merely require the proper entry here and there in order to get things done.

The first thing to do is to add an entry to arch/sh/Kconfig, under the "System type" menu:

```
config SH_VAPOR
        bool "Vapor"
        help
        select Vapor if configuring for a FooTech Vaporboard.
```

next, this has to be added into arch/sh/Makefile. All boards require a machdir-y entry in order to be built. This entry needs to be the name of the board directory as it appears in arch/sh/boards, even if it is in a sub-directory (in which case, all parent directories below arch/sh/boards/ need to be listed). For our new board, this entry can look like:

```
machdir-$(CONFIG_SH_VAPOR)  += vapor
```

provided that we've placed everything in the arch/sh/boards/vapor/ directory.

Next, the build system assumes that your include/asm-sh directory will also be named the same. If this is not the case (as is the case with multiple boards belonging to a common family), then the directory name needs to be implicitly appended to incdir-y. The existing code manages this for the Solution Engine and hp6xx boards, so see these for an example.

Once that is taken care of, it's time to add an entry for the mach type. This is done by adding an entry to the end of the arch/sh/tools/mach-types list. The method for doing this is self explanatory, and so we won't waste space restating it here. After this is done, you will be able to use implicit checks for your board if you need this somewhere throughout the common code, such as:

```
/* Make sure we're on the FooTech Vaporboard */
if (!mach_is_vapor())
        return -ENODEV;
```

also note that the mach_is_boardname() check will be implicitly forced to lowercase, regardless of the fact that the mach-types entries are all uppercase. You can read the script if you really care, but it's pretty ugly, so you probably don't want to do that.

Now all that's left to do is providing a defconfig for your new board. This way, other people who end up with this board can simply use this config for reference instead of trying to guess what settings are supposed to be used on it.

Also, as soon as you have copied over a sample .config for your new board (assume arch/sh/configs/vapor_defconfig), you can also use this directly as a build target, and it will be implicitly listed as such in the help text.

Looking at the 'make help' output, you should now see something like:

Architecture specific targets (sh):

zImage	Compressed kernel image (arch/sh/boot/zImage)
adx_defconfig	Build for adx
cqreek_defconfig	Build for cqreek
dreamcast_defconfig	Build for dreamcast
…	
vapor_defconfig	Build for vapor

which then allows you to do:

```
$ make ARCH=sh CROSS_COMPILE=sh4-linux- vapor_defconfig vmlinux
```

which will in turn copy the defconfig for this board, run it through oldconfig (prompting you for any new options since the time of creation), and start you on your way to having a functional kernel for your new board.

14.3 Notes on register bank usage in the kernel

14.3.1 Introduction

The SH-3 and SH-4 CPU families traditionally include a single partial register bank (selected by SR.RB, only r0 … r7 are banked), whereas other families may have more full-featured banking or simply no such capabilities at all.

14.3.2 SR.RB banking

In the case of this type of banking, banked registers are mapped directly to r0 … r7 if SR.RB is set to the bank we are interested in, otherwise ldc/stc can still be used to reference the banked registers (as r0_bank … r7_bank) when in the context of another bank. The developer must keep the SR.RB value in mind when writing code that utilizes these banked registers, for obvious reasons. Userspace is also not able to poke at the bank1 values, so these can be used rather effectively as scratch registers by the kernel.

Presently the kernel uses several of these registers.

- r0_bank, r1_bank (referenced as k0 and k1, used for scratch registers when doing exception handling).
- r2_bank (used to track the EXPEVT/INTEVT code)
  - Used by do_IRQ() and friends for doing irq mapping based off of the interrupt exception vector jump table offset
- r6_bank (global interrupt mask)
  - The SR.IMASK interrupt handler makes use of this to set the interrupt priority level (used by local_irq_enable())
- r7_bank (current)

14.4 Feature status on sh architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	TODO
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	TODO
core	thread-info-in-task	THREAD_INFO_IN_TASK	TODO
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	TODO
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	ok
debug	KASAN	HAVE_ARCH_KASAN	TODO
debug	kcov	ARCH_HAS_KCOV	TODO
debug	kgdb	HAVE_ARCH_KGDB	ok
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok
debug	kprobes	HAVE_KPROBES	ok
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	TODO
debug	kretprobes	HAVE_KRETPROBES	ok
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	ok
debug	uprobes	ARCH_SUPPORTS_UPROBES	TODO
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	TODO
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	TODO
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	TODO
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok
perf	perf-regs	HAVE_PERF_REGS	TODO
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	TODO
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	TODO
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	---
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	ok
time	clockevents	!LEGACY_TIMER_TICK	ok
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	TODO
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	TODO
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	TODO
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	TODO
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	TODO
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO
vm	ioremap_prot	HAVE_IOREMAP_PROT	ok
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	---

14.5 Memory Management

14.5.1 SH-4

Store Queue API

void **sq_flush_range**(unsigned long start, unsigned int len)

Flush (prefetch) a specific SQ range

Parameters

unsigned long start
the store queue address to start flushing from

unsigned int len
the length to flush

Description

Flushes the store queue cache from **start** to **start** + **len** in a linear fashion.

unsigned long **sq_remap**(unsigned long phys, unsigned int size, const char *name, pgprot_t prot)

Map a physical address through the Store Queues

Parameters

unsigned long phys
Physical address of mapping.

unsigned int size
Length of mapping.

const char *name
User invoking mapping.

pgprot_t prot
Protection bits.

Description

Remaps the physical address **phys** through the next available store queue address of **size** length. **name** is logged at boot time as well as through the sysfs interface.

void **sq_unmap**(unsigned long vaddr)

Unmap a Store Queue allocation

Parameters

unsigned long vaddr
Pre-allocated Store Queue mapping.

Description

Unmaps the store queue allocation **map** that was previously created by *sq_remap()*. Also frees up the pte that was previously inserted into the kernel page table and discards the UTLB translation.

14.6 Machine Specific Interfaces

14.6.1 mach-dreamcast

int **aica_rtc_gettimeofday**(struct device *dev, struct rtc_time *tm)

Get the time from the AICA RTC

Parameters

struct device *dev
the RTC device (ignored)

struct rtc_time *tm
pointer to resulting RTC time structure

Description

Grabs the current RTC seconds counter and adjusts it to the Unix Epoch.

int **aica_rtc_settimeofday**(struct device *dev, struct rtc_time *tm)

Set the AICA RTC to the current time

Parameters

struct device *dev
the RTC device (ignored)

struct rtc_time *tm
pointer to new RTC time structure

Description

Adjusts the given **tv** to the AICA Epoch and sets the RTC seconds counter.

14.6.2 mach-x3proto

int **ilsel_enable**(ilsel_source_t set)

Enable an ILSEL set.

Parameters

ilsel_source_t set
ILSEL source (see ilsel_source_t enum in include/asm-sh/ilsel.h).

Description

Enables a given non-aliased ILSEL source (<= ILSEL_KEY) at the highest available interrupt level. Callers should take care to order callsites noting descending interrupt levels. Aliasing FPGA and external board IRQs need to use *ilsel_enable_fixed()*.

The return value is an IRQ number that can later be taken down with *ilsel_disable()*.

int **ilsel_enable_fixed**(ilsel_source_t set, unsigned int level)

Enable an ILSEL set at a fixed interrupt level

Parameters

ilsel_source_t set
ILSEL source (see ilsel_source_t enum in include/asm-sh/ilsel.h).

`unsigned int level`
Interrupt level (1 - 15)

Description

Enables a given ILSEL source at a fixed interrupt level. Necessary both for level reservation as well as for aliased sources that only exist on special ILSEL#s.

Returns an IRQ number (as *`ilsel_enable()`*).

void **`ilsel_disable`**(unsigned int irq)
Disable an ILSEL set

Parameters

`unsigned int irq`
Bit position for ILSEL set value (retval from enable routines)

Description

Disable a previously enabled ILSEL set.

14.7 Busses

14.7.1 SuperHyway

int **`superhyway_add_device`**(unsigned long base, struct superhyway_device *sdev, struct superhyway_bus *bus)
Add a SuperHyway module

Parameters

`unsigned long base`
Physical address where module is mapped.

`struct superhyway_device *sdev`
SuperHyway device to add, or NULL to allocate a new one.

`struct superhyway_bus *bus`
Bus where SuperHyway module resides.

Description

This is responsible for adding a new SuperHyway module. This sets up a new struct superhyway_device for the module being added if **sdev** == NULL.

Devices are initially added in the order that they are scanned (from the top-down of the memory map), and are assigned an ID based on the order that they are added. Any manual addition of a module will thus get the ID after the devices already discovered regardless of where it resides in memory.

Further work can and should be done in superhyway_scan_bus(), to be sure that any new modules are properly discovered and subsequently registered.

int **`superhyway_register_driver`**(struct superhyway_driver *drv)
Register a new SuperHyway driver

Parameters

struct superhyway_driver *drv
SuperHyway driver to register.

Description

This registers the passed in **drv**. Any devices matching the id table will automatically be populated and handed off to the driver's specified probe routine.

void **superhyway_unregister_driver**(struct superhyway_driver *drv)
Unregister a SuperHyway driver

Parameters

struct superhyway_driver *drv
SuperHyway driver to unregister.

Description

This cleans up after *superhyway_register_driver()*, and should be invoked in the exit path of any module drivers.

14.7.2 Maple

int **maple_driver_register**(struct maple_driver *drv)
register a maple driver

Parameters

struct maple_driver *drv
maple driver to be registered.

Description

Registers the passed in **drv**, while updating the bus type. Devices with matching function IDs will be automatically probed.

void **maple_driver_unregister**(struct maple_driver *drv)
unregister a maple driver.

Parameters

struct maple_driver *drv
maple driver to unregister.

Description

Cleans up after *maple_driver_register()*. To be invoked in the exit path of any module drivers.

void **maple_getcond_callback**(struct maple_device *dev, void (*callback)(struct mapleq *mq), unsigned long interval, unsigned long function)
setup handling MAPLE_COMMAND_GETCOND

Parameters

struct maple_device *dev
device responding

void (*callback) (struct mapleq *mq)
handler callback

`unsigned long interval`
: interval in jiffies between callbacks

`unsigned long function`
: the function code for the device

int **`maple_add_packet`**(struct maple_device *mdev, u32 function, u32 command, size_t length, void *data)

add a single instruction to the maple bus queue

Parameters

`struct maple_device *mdev`
: maple device

`u32 function`
: function on device being queried

`u32 command`
: maple command to add

`size_t length`
: length of command string (in 32 bit words)

`void *data`
: remainder of command string

CHAPTER

FIFTEEN

SPARC ARCHITECTURE

15.1 Steps for sending 'break' on sunhv console

On Baremetal:

1. press Esc + 'B'

On LDOM:

1. press Ctrl + ']'
2. telnet> send break

15.2 Application Data Integrity (ADI)

SPARC M7 processor adds the Application Data Integrity (ADI) feature. ADI allows a task to set version tags on any subset of its address space. Once ADI is enabled and version tags are set for ranges of address space of a task, the processor will compare the tag in pointers to memory in these ranges to the version set by the application previously. Access to memory is granted only if the tag in given pointer matches the tag set by the application. In case of mismatch, processor raises an exception.

Following steps must be taken by a task to enable ADI fully:

1. Set the user mode PSTATE.mcde bit. This acts as master switch for the task's entire address space to enable/disable ADI for the task.
2. Set TTE.mcd bit on any TLB entries that correspond to the range of addresses ADI is being enabled on. MMU checks the version tag only on the pages that have TTE.mcd bit set.
3. Set the version tag for virtual addresses using stxa instruction and one of the MCD specific ASIs. Each stxa instruction sets the given tag for one ADI block size number of bytes. This step must be repeated for entire page to set tags for entire page.

ADI block size for the platform is provided by the hypervisor to kernel in machine description tables. Hypervisor also provides the number of top bits in the virtual address that specify the version tag. Once version tag has been set for a memory location, the tag is stored in the physical memory and the same tag must be present in the ADI version tag bits of the virtual address being presented to the MMU. For example on SPARC M7 processor, MMU uses bits 63-60 for version tags and ADI block size is same as cacheline size which is 64 bytes. A task that sets ADI version to, say 10, on a range of memory, must access that memory using virtual addresses that contain 0xa in bits 63-60.

ADI is enabled on a set of pages using mprotect() with PROT_ADI flag. When ADI is enabled on a set of pages by a task for the first time, kernel sets the PSTATE.mcde bit for the task. Version tags for memory addresses are set with an stxa instruction on the addresses using ASI_MCD_PRIMARY or ASI_MCD_ST_BLKINIT_PRIMARY. ADI block size is provided by the hypervisor to the kernel. Kernel returns the value of ADI block size to userspace using auxiliary vector along with other ADI info. Following auxiliary vectors are provided by the kernel:

AT_ADI_BLKSZ	ADI block size. This is the granularity and alignment, in bytes, of ADI versioning.
AT_ADI_NBITS	Number of ADI version bits in the VA

15.2.1 IMPORTANT NOTES

- Version tag values of 0x0 and 0xf are reserved. These values match any tag in virtual address and never generate a mismatch exception.
- Version tags are set on virtual addresses from userspace even though tags are stored in physical memory. Tags are set on a physical page after it has been allocated to a task and a pte has been created for it.
- When a task frees a memory page it had set version tags on, the page goes back to free page pool. When this page is re-allocated to a task, kernel clears the page using block initialization ASI which clears the version tags as well for the page. If a page allocated to a task is freed and allocated back to the same task, old version tags set by the task on that page will no longer be present.
- ADI tag mismatches are not detected for non-faulting loads.
- Kernel does not set any tags for user pages and it is entirely a task's responsibility to set any version tags. Kernel does ensure the version tags are preserved if a page is swapped out to the disk and swapped back in. It also preserves that version tags if a page is migrated.
- ADI works for any size pages. A userspace task need not be aware of page size when using ADI. It can simply select a virtual address range, enable ADI on the range using mprotect() and set version tags for the entire range. mprotect() ensures range is aligned to page size and is a multiple of page size.
- ADI tags can only be set on writable memory. For example, ADI tags can not be set on read-only mappings.

15.2.2 ADI related traps

With ADI enabled, following new traps may occur:

Disrupting memory corruption

When a store accesses a memory location that has TTE.mcd=1, the task is running with ADI enabled (PSTATE.mcde=1), and the ADI tag in the address used (bits 63:60) does not match the tag set on the corresponding cacheline, a memory corruption trap occurs. By default, it is a disrupting trap and is sent to the hypervisor first. Hypervisor creates a sun4v error report and sends a resumable error (TT=0x7e) trap to the kernel. The kernel sends a SIGSEGV to the task that resulted in this trap with the following info:

```
siginfo.si_signo = SIGSEGV;
siginfo.errno = 0;
siginfo.si_code = SEGV_ADIDERR;
siginfo.si_addr = addr; /* PC where first mismatch occurred */
siginfo.si_trapno = 0;
```

Precise memory corruption

When a store accesses a memory location that has TTE.mcd=1, the task is running with ADI enabled (PSTATE.mcde=1), and the ADI tag in the address used (bits 63:60) does not match the tag set on the corresponding cacheline, a memory corruption trap occurs. If MCD precise exception is enabled (MCDPERR=1), a precise exception is sent to the kernel with TT=0x1a. The kernel sends a SIGSEGV to the task that resulted in this trap with the following info:

```
siginfo.si_signo = SIGSEGV;
siginfo.errno = 0;
siginfo.si_code = SEGV_ADIPERR;
siginfo.si_addr = addr; /* address that caused trap */
siginfo.si_trapno = 0;
```

NOTE:
ADI tag mismatch on a load always results in precise trap.

MCD disabled

When a task has not enabled ADI and attempts to set ADI version on a memory address, processor sends an MCD disabled trap. This trap is handled by hypervisor first and the hypervisor vectors this trap through to the kernel as Data Access Exception trap with fault type set to 0xa (invalid ASI). When this occurs, the kernel sends the task SIGSEGV signal with following info:

```
siginfo.si_signo = SIGSEGV;
siginfo.errno = 0;
siginfo.si_code = SEGV_ACCADI;
siginfo.si_addr = addr; /* address that caused trap */
siginfo.si_trapno = 0;
```

Sample program to use ADI

Following sample program is meant to illustrate how to use the ADI functionality:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <elf.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/mman.h>
#include <asm/asi.h>

#ifndef AT_ADI_BLKSZ
#define AT_ADI_BLKSZ  48
#endif
#ifndef AT_ADI_NBITS
#define AT_ADI_NBITS  49
#endif

#ifndef PROT_ADI
#define PROT_ADI      0x10
#endif

#define BUFFER_SIZE     32*1024*1024UL

main(int argc, char* argv[], char* envp[])
{
        unsigned long i, mcde, adi_blksz, adi_nbits;
        char *shmaddr, *tmp_addr, *end, *veraddr, *clraddr;
        int shmid, version;
      Elf64_auxv_t *auxv;

      adi_blksz = 0;

      while(*envp++ != NULL);
      for (auxv = (Elf64_auxv_t *)envp; auxv->a_type != AT_NULL; auxv++) {
              switch (auxv->a_type) {
              case AT_ADI_BLKSZ:
                      adi_blksz = auxv->a_un.a_val;
                      break;
              case AT_ADI_NBITS:
                      adi_nbits = auxv->a_un.a_val;
                      break;
              }
      }
      if (adi_blksz == 0) {
              fprintf(stderr, "Oops! ADI is not supported\n");
              exit(1);
      }
```

```
        printf("ADI capabilities:\n");
        printf("\tBlock size = %ld\n", adi_blksz);
        printf("\tNumber of bits = %ld\n", adi_nbits);

          if ((shmid = shmget(2, BUFFER_SIZE,
                                  IPC_CREAT | SHM_R | SHM_W)) < 0) {
                  perror("shmget failed");
                  exit(1);
          }

          shmaddr = shmat(shmid, NULL, 0);
          if (shmaddr == (char *)-1) {
                  perror("shm attach failed");
                  shmctl(shmid, IPC_RMID, NULL);
                  exit(1);
          }

        if (mprotect(shmaddr, BUFFER_SIZE, PROT_READ|PROT_WRITE|PROT_ADI)) {
                perror("mprotect failed");
                goto err_out;
        }

          /* Set the ADI version tag on the shm segment
           */
          version = 10;
          tmp_addr = shmaddr;
          end = shmaddr + BUFFER_SIZE;
          while (tmp_addr < end) {
                  asm volatile(
                          "stxa %1, [%0]0x90\n\t"
                          :
                          : "r" (tmp_addr), "r" (version));
                  tmp_addr += adi_blksz;
          }
        asm volatile("membar #Sync\n\t");

          /* Create a versioned address from the normal address by placing
         * version tag in the upper adi_nbits bits
           */
          tmp_addr = (void *) ((unsigned long)shmaddr << adi_nbits);
          tmp_addr = (void *) ((unsigned long)tmp_addr >> adi_nbits);
          veraddr = (void *) (((unsigned long)version << (64-adi_nbits))
                          | (unsigned long)tmp_addr);

          printf("Starting the writes:\n");
          for (i = 0; i < BUFFER_SIZE; i++) {
                  veraddr[i] = (char)(i);
                  if (!(i % (1024 * 1024)))
                          printf(".");
          }
          printf("\n");

          printf("Verifying data...");
        fflush(stdout);
          for (i = 0; i < BUFFER_SIZE; i++)
                  if (veraddr[i] != (char)i)
                          printf("\nIndex %lu mismatched\n", i);
          printf("Done.\n");

          /* Disable ADI and clean up
           */
        if (mprotect(shmaddr, BUFFER_SIZE, PROT_READ|PROT_WRITE)) {
                perror("mprotect failed");
                goto err_out;
        }

          if (shmdt((const void *)shmaddr) != 0)
                  perror("Detach failure");
          shmctl(shmid, IPC_RMID, NULL);

          exit(0);

  err_out:
          if (shmdt((const void *)shmaddr) != 0)
                  perror("Detach failure");
          shmctl(shmid, IPC_RMID, NULL);
          exit(1);
  }
```

15.3 Oracle Data Analytics Accelerator (DAX)

DAX is a coprocessor which resides on the SPARC M7 (DAX1) and M8 (DAX2) processor chips, and has direct access to the CPU's L3 caches as well as physical memory. It can perform several operations on data streams with various input and output formats. A driver provides a transport mechanism and has limited knowledge of the various opcodes and data formats. A user space library provides high level services and translates these into low level commands which are then passed into the driver and subsequently the Hypervisor and the coprocessor. The library is the recommended way for applications to use the coprocessor, and the driver interface is not intended for general use. This document describes the general flow of the driver, its structures, and its programmatic interface. It also provides example code sufficient to write user or kernel applications that use DAX functionality.

The user library is open source and available at:

> https://oss.oracle.com/git/gitweb.cgi?p=libdax.git

The Hypervisor interface to the coprocessor is described in detail in the accompanying document, dax-hv-api.txt, which is a plain text excerpt of the (Oracle internal) "UltraSPARC Virtual Machine Specification" version 3.0.20+15, dated 2017-09-25.

15.3.1 High Level Overview

A coprocessor request is described by a Command Control Block (CCB). The CCB contains an opcode and various parameters. The opcode specifies what operation is to be done, and the parameters specify options, flags, sizes, and addresses. The CCB (or an array of CCBs) is passed to the Hypervisor, which handles queueing and scheduling of requests to the available coprocessor execution units. A status code returned indicates if the request was submitted successfully or if there was an error. One of the addresses given in each CCB is a pointer to a "completion area", which is a 128 byte memory block that is written by the coprocessor to provide execution status. No interrupt is generated upon completion; the completion area must be polled by software to find out when a transaction has finished, but the M7 and later processors provide a mechanism to pause the virtual processor until the completion status has been updated by the coprocessor. This is done using the monitored load and mwait instructions, which are described in more detail later. The DAX coprocessor was designed so that after a request is submitted, the kernel is no longer involved in the processing of it. The polling is done at the user level, which results in almost zero latency between completion of a request and resumption of execution of the requesting thread.

15.3.2 Addressing Memory

The kernel does not have access to physical memory in the Sun4v architecture, as there is an additional level of memory virtualization present. This intermediate level is called "real" memory, and the kernel treats this as if it were physical. The Hypervisor handles the translations between real memory and physical so that each logical domain (LDOM) can have a partition of physical memory that is isolated from that of other LDOMs. When the kernel sets up a virtual mapping, it specifies a virtual address and the real address to which it should be mapped.

The DAX coprocessor can only operate on physical memory, so before a request can be fed to the coprocessor, all the addresses in a CCB must be converted into physical addresses. The kernel cannot do this since it has no visibility into physical addresses. So a CCB may contain either the virtual or real addresses of the buffers or a combination of them. An "address type" field is available for each address that may be given in the CCB. In all cases, the Hypervisor will translate all the addresses to physical before dispatching to hardware. Address translations are performed using the context of the process initiating the request.

15.3.3 The Driver API

An application makes requests to the driver via the write() system call, and gets results (if any) via read(). The completion areas are made accessible via mmap(), and are read-only for the application.

The request may either be an immediate command or an array of CCBs to be submitted to the hardware.

Each open instance of the device is exclusive to the thread that opened it, and must be used by that thread for all subsequent operations. The driver open function creates a new context for the thread and initializes it for use. This context contains pointers and values used internally by the driver to keep track of submitted requests. The completion area buffer is also allocated, and this is large enough to contain the completion areas for many concurrent requests. When the device is closed, any outstanding transactions are flushed and the context is cleaned up.

On a DAX1 system (M7), the device will be called "oradax1", while on a DAX2 system (M8) it will be "oradax2". If an application requires one or the other, it should simply attempt to open the appropriate device. Only one of the devices will exist on any given system, so the name can be used to determine what the platform supports.

The immediate commands are CCB_DEQUEUE, CCB_KILL, and CCB_INFO. For all of these, success is indicated by a return value from write() equal to the number of bytes given in the call. Otherwise -1 is returned and errno is set.

CCB_DEQUEUE

Tells the driver to clean up resources associated with past requests. Since no interrupt is generated upon the completion of a request, the driver must be told when it may reclaim resources. No further status information is returned, so the user should not subsequently call read().

CCB_KILL

Kills a CCB during execution. The CCB is guaranteed to not continue executing once this call returns successfully. On success, read() must be called to retrieve the result of the action.

CCB_INFO

Retrieves information about a currently executing CCB. Note that some Hypervisors might return 'notfound' when the CCB is in 'inprogress' state. To ensure a CCB in the 'notfound' state will never be executed, CCB_KILL must be invoked on that CCB. Upon success, read() must be called to retrieve the details of the action.

Submission of an array of CCBs for execution

A write() whose length is a multiple of the CCB size is treated as a submit operation. The file offset is treated as the index of the completion area to use, and may be set via lseek() or using the pwrite() system call. If -1 is returned then errno is set to indicate the error. Otherwise, the return value is the length of the array that was actually accepted by the coprocessor. If the accepted length is equal to the requested length, then the submission was completely successful and there is no further status needed; hence, the user should not subsequently call read(). Partial acceptance of the CCB array is indicated by a return value less than the requested length, and read() must be called to retrieve further status information. The status will reflect the error caused by the first CCB that was not accepted, and status_data will provide additional data in some cases.

MMAP

The mmap() function provides access to the completion area allocated in the driver. Note that the completion area is not writeable by the user process, and the mmap call must not specify PROT_WRITE.

15.3.4 Completion of a Request

The first byte in each completion area is the command status which is updated by the coprocessor hardware. Software may take advantage of new M7/M8 processor capabilities to efficiently poll this status byte. First, a “monitored load” is achieved via a Load from Alternate Space (ldxa, lduba, etc.) with ASI 0x84 (ASI_MONITOR_PRIMARY). Second, a “monitored wait” is achieved via the mwait instruction (a write to %asr28). This instruction is like pause in that it suspends execution of the virtual processor for the given number of nanoseconds, but in addition will terminate early when one of several events occur. If the block of data containing the monitored location is modified, then the mwait terminates. This causes software to resume execution immediately (without a context switch or kernel to user transition) after a transaction completes. Thus the latency between transaction completion and resumption of execution may be just a few nanoseconds.

15.3.5 Application Life Cycle of a DAX Submission

- open dax device
- call mmap() to get the completion area address
- allocate a CCB and fill in the opcode, flags, parameters, addresses, etc.
- submit CCB via write() or pwrite()
- go into a loop executing monitored load + monitored wait and terminate when the command status indicates the request is complete (CCB_KILL or CCB_INFO may be used any time as necessary)
- perform a CCB_DEQUEUE
- call munmap() for completion area
- close the dax device

15.3.6 Memory Constraints

The DAX hardware operates only on physical addresses. Therefore, it is not aware of virtual memory mappings and the discontiguities that may exist in the physical memory that a virtual buffer maps to. There is no I/O TLB or any scatter/gather mechanism. All buffers, whether input or output, must reside in a physically contiguous region of memory.

The Hypervisor translates all addresses within a CCB to physical before handing off the CCB to DAX. The Hypervisor determines the virtual page size for each virtual address given, and uses this to program a size limit for each address. This prevents the coprocessor from reading or writing beyond the bound of the virtual page, even though it is accessing physical memory directly. A simpler way of saying this is that a DAX operation will never “cross” a virtual page boundary. If an 8k virtual page is used, then the data is strictly limited to 8k. If a user’s buffer is larger than 8k, then a larger page size must be used, or the transaction size will be truncated to 8k.

Huge pages. A user may allocate huge pages using standard interfaces. Memory buffers residing on huge pages may be used to achieve much larger DAX transaction sizes, but the rules must still be followed, and no transaction will cross a page boundary, even a huge page. A major caveat is that Linux on Sparc presents 8Mb as one of the huge page sizes. Sparc does not actually provide a 8Mb hardware page size, and this size is synthesized by pasting together two 4Mb pages. The reasons for this are historical, and it creates an issue because only half of this 8Mb page can actually be used for any given buffer in a DAX request, and it must be either the first half or the second half; it cannot be a 4Mb chunk in the middle, since that crosses a (hardware) page boundary. Note that this entire issue may be hidden by higher level libraries.

CCB Structure

A CCB is an array of 8 64-bit words. Several of these words provide command opcodes, parameters, flags, etc., and the rest are addresses for the completion area, output buffer, and various inputs:

```
struct ccb {
    u64   control;
    u64   completion;
    u64   input0;
    u64   access;
    u64   input1;
    u64   op_data;
    u64   output;
    u64   table;
};
```

See libdax/common/sys/dax1/dax1_ccb.h for a detailed description of each of these fields, and see dax-hv-api.txt for a complete description of the Hypervisor API available to the guest OS (ie, Linux kernel).

The first word (control) is examined by the driver for the following:

- CCB version, which must be consistent with hardware version
- Opcode, which must be one of the documented allowable commands
- Address types, which must be set to “virtual” for all the addresses given by the user, thereby ensuring that the application can only access memory that it owns

15.3.7 Example Code

The DAX is accessible to both user and kernel code. The kernel code can make hypercalls directly while the user code must use wrappers provided by the driver. The setup of the CCB is nearly identical for both; the only difference is in preparation of the completion area. An example of user code is given now, with kernel code afterwards.

In order to program using the driver API, the file arch/sparc/include/uapi/asm/oradax.h must be included.

First, the proper device must be opened. For M7 it will be /dev/oradax1 and for M8 it will be /dev/oradax2. The simplest procedure is to attempt to open both, as only one will succeed:

```
fd = open("/dev/oradax1", O_RDWR);
if (fd < 0)
        fd = open("/dev/oradax2", O_RDWR);
if (fd < 0)
       /* No DAX found */
```

Next, the completion area must be mapped:

```
completion_area = mmap(NULL, DAX_MMAP_LEN, PROT_READ, MAP_SHARED, fd, 0);
```

All input and output buffers must be fully contained in one hardware page, since as explained above, the DAX is strictly constrained by virtual page boundaries. In addition, the output buffer must be 64-byte aligned and its size must be a multiple of 64 bytes because the coprocessor writes in units of cache lines.

This example demonstrates the DAX Scan command, which takes as input a vector and a match value, and produces a bitmap as the output. For each input element that matches the value, the corresponding bit is set in the output.

In this example, the input vector consists of a series of single bits, and the match value is 0. So each 0 bit in the input will produce a 1 in the output, and vice versa, which produces an output bitmap which is the input bitmap inverted.

For details of all the parameters and bits used in this CCB, please refer to section 36.2.1.3 of the DAX Hypervisor API document, which describes the Scan command in detail:

```
ccb->control =       /* Table 36.1, CCB Header Format */
          (2L << 48)     /* command = Scan Value */
        | (3L << 40)     /* output address type = primary virtual */
        | (3L << 34)     /* primary input address type = primary virtual */
                     /* Section 36.2.1, Query CCB Command Formats */
        | (1 << 28)     /* 36.2.1.1.1 primary input format = fixed width bit␣
↪packed */
        | (0 << 23)     /* 36.2.1.1.2 primary input element size = 0 (1 bit) */
        | (8 << 10)     /* 36.2.1.1.6 output format = bit vector */
        | (0 <<  5)     /* 36.2.1.3 First scan criteria size = 0 (1 byte) */
        | (31 << 0);    /* 36.2.1.3 Disable second scan criteria */

ccb->completion = 0;    /* Completion area address, to be filled in by driver␣
↪*/

ccb->input0 = (unsigned long) input; /* primary input address */

ccb->access =       /* Section 36.2.1.2, Data Access Control */
          (2 << 24)    /* Primary input length format = bits */
        | (nbits - 1); /* number of bits in primary input stream, minus 1 */

ccb->input1 = 0;       /* secondary input address, unused */

ccb->op_data = 0;      /* scan criteria (value to be matched) */
```

```
ccb->output = (unsigned long) output;   /* output address */

ccb->table = 0;         /* table address, unused */
```

The CCB submission is a write() or pwrite() system call to the driver. If the call fails, then a read() must be used to retrieve the status:

```
if (pwrite(fd, ccb, 64, 0) != 64) {
        struct ccb_exec_result status;
        read(fd, &status, sizeof(status));
        /* bail out */
}
```

After a successful submission of the CCB, the completion area may be polled to determine when the DAX is finished. Detailed information on the contents of the completion area can be found in section 36.2.2 of the DAX HV API document:

```
while (1) {
        /* Monitored Load */
        __asm__ __volatile__("lduba [%1] 0x84, %0\n"
                             : "=r" (status)
                             : "r"  (completion_area));

        if (status)          /* 0 indicates command in progress */
                break;

        /* MWAIT */
        __asm__ __volatile__("wr %%g0, 1000, %%asr28\n" ::);    /* 1000 ns */
}
```

A completion area status of 1 indicates successful completion of the CCB and validity of the output bitmap, which may be used immediately. All other non-zero values indicate error conditions which are described in section 36.2.2:

```
if (completion_area[0] != 1) {  /* section 36.2.2, 1 = command ran and␣
↪succeeded */
        /* completion_area[0] contains the completion status */
        /* completion_area[1] contains an error code, see 36.2.2 */
}
```

After the completion area has been processed, the driver must be notified that it can release any resources associated with the request. This is done via the dequeue operation:

```
struct dax_command cmd;
cmd.command = CCB_DEQUEUE;
if (write(fd, &cmd, sizeof(cmd)) != sizeof(cmd)) {
        /* bail out */
}
```

Finally, normal program cleanup should be done, i.e., unmapping completion area, closing the dax device, freeing memory etc.

Kernel example

The only difference in using the DAX in kernel code is the treatment of the completion area. Unlike user applications which mmap the completion area allocated by the driver, kernel code must allocate its own memory to use for the completion area, and this address and its type must be given in the CCB:

```
ccb->control |=      /* Table 36.1, CCB Header Format */
        (3L << 32);     /* completion area address type = primary virtual */

ccb->completion = (unsigned long) completion_area;   /* Completion area␣
↪address */
```

The dax submit hypercall is made directly. The flags used in the ccb_submit call are documented in the DAX HV API in section 36.3.1/

```
#include <asm/hypervisor.h>

      hv_rv = sun4v_ccb_submit((unsigned long)ccb, 64,
                               HV_CCB_QUERY_CMD |
                               HV_CCB_ARG0_PRIVILEGED | HV_CCB_ARG0_TYPE_
↪PRIMARY |
                               HV_CCB_VA_PRIVILEGED,
                               0, &bytes_accepted, &status_data);

      if (hv_rv != HV_EOK) {
              /* hv_rv is an error code, status_data contains */
              /* potential additional status, see 36.3.1.1 */
      }
```

After the submission, the completion area polling code is identical to that in user land:

```
while (1) {
        /* Monitored Load */
        __asm__ __volatile__("lduba [%1] 0x84, %0\n"
                             : "=r" (status)
                             : "r"  (completion_area));

        if (status)          /* 0 indicates command in progress */
                break;

        /* MWAIT */
        __asm__ __volatile__("wr %%g0, 1000, %%asr28\n" ::);    /* 1000 ns */
}

if (completion_area[0] != 1) {  /* section 36.2.2, 1 = command ran and␣
↪succeeded */
        /* completion_area[0] contains the completion status */
        /* completion_area[1] contains an error code, see 36.2.2 */
}
```

The output bitmap is ready for consumption immediately after the completion status indicates success.

15.3.8 Excer[t from UltraSPARC Virtual Machine Specification

```
Excerpt from UltraSPARC Virtual Machine Specification
Compiled from version 3.0.20+15
Publication date 2017-09-25 08:21
Copyright © 2008, 2015 Oracle and/or its affiliates. All rights reserved.
Extracted via "pdftotext -f 547 -l 572 -layout sun4v_20170925.pdf"
Authors:
         Charles Kunzman
         Sam Glidden
         Mark Cianchetti


Chapter 36. Coprocessor services
        The following APIs provide access via the Hypervisor to hardware␣
↪assisted data processing functionality.
        These APIs may only be provided by certain platforms, and may not␣
↪be available to all virtual machines
        even on supported platforms. Restrictions on the use of these APIs␣
↪may be imposed in order to support
        live-migration and other system management activities.

36.1. Data Analytics Accelerator
        The Data Analytics Accelerator (DAX) functionality is a collection␣
↪of hardware coprocessors that provide
        high speed processoring of database-centric operations. The␣
↪coprocessors may support one or more of
        the following data query operations: search, extraction,␣
↪compression, decompression, and translation. The
        functionality offered may vary by virtual machine implementation.

        The DAX is a virtual device to sun4v guests, with supported data␣
↪operations indicated by the virtual device
        compatibility property. Functionality is accessed through the␣
↪submission of Command Control Blocks
        (CCBs) via the ccb_submit API function. The operations are␣
↪processed asynchronously, with the status
        of the submitted operations reported through a Completion Area␣
↪linked to each CCB. Each CCB has a
        separate Completion Area and, unless execution order is␣
↪specifically restricted through the use of serial-
        conditional flags, the execution order of submitted CCBs is␣
↪arbitrary. Likewise, the time to completion
        for a given CCB is never guaranteed.

        Guest software may implement a software timeout on CCB operations,␣
↪and if the timeout is exceeded, the
        operation may be cancelled or killed via the ccb_kill API function.
↪ It is recommended for guest software
```

```
        to implement a software timeout to account for certain RAS errors␣
↪which may result in lost CCBs. It is
        recommended such implementation use the ccb_info API function to␣
↪check the status of a CCB prior to
        killing it in order to determine if the CCB is still in queue, or␣
↪may have been lost due to a RAS error.

        There is no fixed limit on the number of outstanding CCBs guest␣
↪software may have queued in the virtual
        machine, however, internal resource limitations within the virtual␣
↪machine can cause CCB submissions
        to be temporarily rejected with EWOULDBLOCK. In such cases, guests␣
↪should continue to attempt
        submissions until they succeed; waiting for an outstanding CCB to␣
↪complete is not necessary, and would
        not be a guarantee that a future submission would succeed.

        The availablility of DAX coprocessor command service is indicated␣
↪by the presence of the DAX virtual
        device node in the guest MD (Section 8.24.17, “Database Analytics␣
↪Accelerators (DAX) virtual-device
        node”).

36.1.1. DAX Compatibility Property
        The query functionality may vary based on the compatibility␣
↪property of the virtual device:

36.1.1.1. "ORCL,sun4v-dax" Device Compatibility
        Available CCB commands:

        • No-op/Sync

        • Extract

        • Scan Value

        • Inverted Scan Value

        • Scan Range

                                                            509

                                                    Coprocessor services


        • Inverted Scan Range

        • Translate

        • Inverted Translate
        • Select

        See Section 36.2.1, “Query CCB Command Formats” for the␣
↪corresponding CCB input and output formats.

        Only version 0 CCBs are available.

36.1.1.2. "ORCL,sun4v-dax-fc" Device Compatibility
        "ORCL,sun4v-dax-fc" is compatible with the "ORCL,sun4v-dax"␣
↪interface, and includes additional CCB
        bit fields and controls.

36.1.1.3. "ORCL,sun4v-dax2" Device Compatibility
        Available CCB commands:

        • No-op/Sync

        • Extract

        • Scan Value

        • Inverted Scan Value

        • Scan Range

        • Inverted Scan Range

        • Translate

        • Inverted Translate

        • Select

        See Section 36.2.1, “Query CCB Command Formats” for the␣
↪corresponding CCB input and output formats.

        Version 0 and 1 CCBs are available. Only version 0 CCBs may use␣
↪Huffman encoded data, whereas only
        version 1 CCBs may use OZIP.

36.1.2. DAX Virtual Device Interrupts
        The DAX virtual device has multiple interrupts associated with it␣
↪which may be used by the guest if
        desired. The number of device interrupts available to the guest is␣
↪indicated in the virtual device node of the
        guest MD (Section 8.24.17, “Database Analytics Accelerators (DAX)␣
↪virtual-device node”). If the device
        node indicates N interrupts available, the guest may use any value␣
↪from 0 to N - 1 (inclusive) in a CCB
        interrupt number field. Using values outside this range will␣
```

```
↪result in the CCB being rejected for an invalid
        field value.

        The interrupts may be bound and managed using the standard sun4v␣
↪device interrupts API (Chapter 16,
        Device interrupt services). Sysino interrupts are not available␣
↪for DAX devices.

36.2. Coprocessor Control Block (CCB)
        CCBs are either 64 or 128 bytes long, depending on the operation␣
↪type. The exact contents of the CCB
        are command specific, but all CCBs contain at least one memory␣
↪buffer address. All memory locations


                                                            510

                                             Coprocessor services


referenced by a CCB must be pinned in memory until the CCB either␣
↪completes execution or is killed
via the ccb_kill API call. Changes in virtual address mappings occurring␣
↪after CCB submission are not
guaranteed to be visible, and as such all virtual address updates need to␣
↪be synchronized with CCB
execution.

All CCBs begin with a common 32-bit header.

Table 36.1. CCB Header Format
Bits          Field Description
[31:28]       CCB version. For API version 2.0: set to 1 if CCB uses OZIP␣
↪encoding; set to 0 if the CCB
              uses Huffman encoding; otherwise either 0 or 1. For API␣
↪version 1.0: always set to 0.
[27]          When API version 2.0 is negotiated, this is the Pipeline␣
↪Flag [512]. It is reserved in
              API version 1.0
[26]          Long CCB flag [512]
[25]          Conditional synchronization flag [512]
[24]          Serial synchronization flag
[23:16]       CCB operation code:
               0x00        No Operation (No-op) or Sync
               0x01        Extract
               0x02        Scan Value
               0x12        Inverted Scan Value
               0x03        Scan Range
               0x13        Inverted Scan Range
               0x04        Translate
               0x14        Inverted Translate
```

```
                  0x05        Select
    [15:13]      Reserved
    [12:11]      Table address type
                  0b'00       No address
                  0b'01       Alternate context virtual address
                  0b'10       Real address
                  0b'11       Primary context virtual address
    [10:8]       Output/Destination address type
                  0b'000      No address
                  0b'001      Alternate context virtual address
                  0b'010      Real address
                  0b'011      Primary context virtual address
                  0b'100      Reserved
                  0b'101      Reserved
                  0b'110      Reserved
                  0b'111      Reserved
    [7:5]        Secondary source address type


                                                    511

                                            Coprocessor services



    Bits           Field Description
                    0b'000      No address
                    0b'001      Alternate context virtual address
                    0b'010      Real address
                    0b'011      Primary context virtual address
                    0b'100      Reserved
                    0b'101      Reserved
                    0b'110      Reserved
                    0b'111      Reserved
    [4:2]          Primary source address type
                    0b'000      No address
                    0b'001      Alternate context virtual address
                    0b'010      Real address
                    0b'011      Primary context virtual address
                    0b'100      Reserved
                    0b'101      Reserved
                    0b'110      Reserved
                    0b'111      Reserved
    [1:0]          Completion area address type
                    0b'00       No address
                    0b'01       Alternate context virtual address
                    0b'10       Real address
                    0b'11       Primary context virtual address

    The Long CCB flag indicates whether the submitted CCB is 64 or 128 bytes␣
     ↪long; value is 0 for 64 bytes
    and 1 for 128 bytes.
```

```
The Serial and Conditional flags allow simple relative ordering between␣
↪CCBs. Any CCB with the Serial
flag set will execute sequentially relative to any previous CCB that is␣
↪also marked as Serial in the same
CCB submission. CCBs without the Serial flag set execute independently,␣
↪even if they are between CCBs
with the Serial flag set. CCBs marked solely with the Serial flag will␣
↪execute upon the completion of the
previous Serial CCB, regardless of the completion status of that CCB. The␣
↪Conditional flag allows CCBs
to conditionally execute based on the successful execution of the closest␣
↪CCB marked with the Serial flag.
A CCB may only be conditional on exactly one CCB, however, a CCB may be␣
↪marked both Conditional
and Serial to allow execution chaining. The flags do NOT allow fan-out␣
↪chaining, where multiple CCBs
execute in parallel based on the completion of another CCB.

The Pipeline flag is an optimization that directs the output of one CCB␣
↪(the "source" CCB) directly to
the input of the next CCB (the "target" CCB). The target CCB thus does not␣
↪need to read the input from
memory. The Pipeline flag is advisory and may be dropped.

Both the Pipeline and Serial bits must be set in the source CCB. The␣
↪Conditional bit must be set in the
target CCB. Exactly one CCB must be made conditional on the source CCB;␣
↪either 0 or 2 target CCBs
is invalid. However, Pipelines can be extended beyond two CCBs: the␣
↪sequence would start with a CCB
with both the Pipeline and Serial bits set, proceed through CCBs with the␣
↪Pipeline, Serial, and Conditional
bits set, and terminate at a CCB that has the Conditional bit set, but not␣
↪the Pipeline bit.


                                                     512

                                                       Coprocessor services


          The input of the target CCB must start within 64 bytes of the␣
↪output of the source CCB or the pipeline flag
          will be ignored. All CCBs in a pipeline must be submitted in the␣
↪same call to ccb_submit.

          The various address type fields indicate how the various address␣
↪values used in the CCB should be
          interpreted by the virtual machine. Not all of the types␣
↪specified are used by every CCB format. Types
```

```
        which are not applicable to the given CCB command should be␣
↪indicated as type 0 (No address). Virtual
        addresses used in the CCB must have translation entries present␣
↪in either the TLB or a configured TSB
        for the submitting virtual processor. Virtual addresses which␣
↪cannot be translated by the virtual machine
        will result in the CCB submission being rejected, with the␣
↪causal virtual address indicated. The CCB
        may be resubmitted after inserting the translation, or the␣
↪address may be translated by guest software and
        resubmitted using the real address translation.

36.2.1. Query CCB Command Formats
36.2.1.1. Supported Data Formats, Elements Sizes and Offsets
        Data for query commands may be encoded in multiple possible␣
↪formats. The data query commands use a
        common set of values to indicate the encoding formats of the␣
↪data being processed. Some encoding formats
        require multiple data streams for processing, requiring the␣
↪specification of both primary data formats (the
        encoded data) and secondary data streams (meta-data for the␣
↪encoded data).

36.2.1.1.1. Primary Input Format

        The primary input format code is a 4-bit field when it is used.␣
↪There are 10 primary input formats available.
        The packed formats are not endian neutral. Code values not␣
↪listed below are reserved.

        Code        Format                              Description
        0x0         Fixed width byte packed             Up to 16 bytes
        0x1         Fixed width bit packed              Up to 15 bits␣
↪(CCB version 0) or 23 bits (CCB version
                                                        1); bits are␣
↪read most significant bit to least significant bit
                                                        within a byte
        0x2         Variable width byte packed          Data stream of␣
↪lengths must be provided as a secondary
                                                        input
        0x4         Fixed width byte packed with run Up to 16 bytes;␣
↪data stream of run lengths must be
                    length encoding                  provided as a␣
↪secondary input
        0x5         Fixed width bit packed with run Up to 15 bits (CCB␣
↪version 0) or 23 bits (CCB version
                    length encoding                 1); bits are read␣
↪most significant bit to least significant bit
                                                    within a byte; data␣
↪stream of run lengths must be provided
                                                    as a secondary input
        0x8         Fixed width byte packed with Up to 16 bytes before␣
↪the encoding; compressed stream
                    Huffman (CCB version 0) or bits are read most␣
↪significant bit to least significant bit
                    OZIP (CCB version 1) encoding within a byte; pointer␣
↪to the encoding table must be
                                                  provided
        0x9         Fixed width bit packed with Up to 15 bits (CCB␣
↪version 0) or 23 bits (CCB version
                    Huffman (CCB version 0) or 1); compressed stream␣
↪bits are read most significant bit to
                    OZIP (CCB version 1) encoding least significant bit␣
↪within a byte; pointer to the encoding
                                                  table must be provided
        0xA         Variable width byte packed with Up to 16 bytes␣
↪before the encoding; compressed stream
                    Huffman (CCB version 0) or bits are read most␣
↪significant bit to least significant bit
                    OZIP (CCB version 1) encoding within a byte; data␣
↪stream of lengths must be provided as
                                                    a secondary input;␣
↪pointer to the encoding table must be
                                                    provided


                                                      513

                                             Coprocessor services


        Code        Format                              Description
        0xC         Fixed width byte packed with        Up to 16 bytes␣
↪before the encoding; compressed stream
                    run length encoding, followed by    bits are read␣
↪most significant bit to least significant bit
                    Huffman (CCB version 0) or          within a byte;␣
↪data stream of run lengths must be provided
                    OZIP (CCB version 1) encoding       as a secondary␣
↪input; pointer to the encoding table must
                                                        be provided
        0xD         Fixed width bit packed with         Up to 15 bits␣
↪(CCB version 0) or 23 bits(CCB version 1)
                    run length encoding, followed by    before the␣
↪encoding; compressed stream bits are read most
                    Huffman (CCB version 0) or          significant bit␣
↪to least significant bit within a byte; data
                    OZIP (CCB version 1) encoding       stream of run␣
↪lengths must be provided as a secondary
                                                        input; pointer␣
↪to the encoding table must be provided
```

```
          If OZIP encoding is used, there must be no reserved bytes in the␣
↪table.

36.2.1.1.2. Primary Input Element Size

          For primary input data streams with fixed size elements, the␣
↪element size must be indicated in the CCB
          command. The size is encoded as the number of bits or bytes,␣
↪minus one. The valid value range for this
          field depends on the input format selected, as listed in the␣
↪table above.

36.2.1.1.3. Secondary Input Format

          For primary input data streams which require a secondary input␣
↪stream, the secondary input stream is
          always encoded in a fixed width, bit-packed format. The bits are␣
↪read from most significant bit to least
          significant bit within a byte. There are two encoding options␣
↪for the secondary input stream data elements,
          depending on whether the value of 0 is needed:

          Secondary          Input Description
          Format Code
          0                          Element is stored as value minus 1 (0␣
↪evaluates to 1, 1 evaluates
                                     to 2, etc)
          1                          Element is stored as value

36.2.1.1.4. Secondary Input Element Size

          Secondary input element size is encoded as a two bit field:

          Secondary Input Size Description
          Code
          0x0                        1 bit
          0x1                        2 bits
          0x2                        4 bits
          0x3                        8 bits

36.2.1.1.5. Input Element Offsets

          Bit-wise input data streams may have any alignment within the␣
↪base addressed byte. The offset, specified
          from most significant bit to least significant bit, is provided␣
↪as a fixed 3 bit field for each input type. A
          value of 0 indicates that the first input element begins at the␣
↪most significant bit in the first byte, and a
          value of 7 indicates it begins with the least significant bit.

          This field should be zero for any byte-wise primary input data␣
```

```
↪streams.


                                                                   514

                                                         Coprocessor services


36.2.1.1.6. Output Format

          Query commands support multiple sizes and encodings for output␣
↪data streams. There are four possible
          output encodings, and up to four supported element sizes per␣
↪encoding. Not all output encodings are
          supported for every command. The format is indicated by a 4-bit␣
↪field in the CCB:

           Output Format Code          Description
           0x0                         Byte aligned, 1 byte elements
           0x1                         Byte aligned, 2 byte elements
           0x2                         Byte aligned, 4 byte elements
           0x3                         Byte aligned, 8 byte elements
           0x4                         16 byte aligned, 16 byte elements
           0x5                         Reserved
           0x6                         Reserved
           0x7                         Reserved
           0x8                         Packed vector of single bit elements
           0x9                         Reserved
           0xA                         Reserved
           0xB                         Reserved
           0xC                         Reserved
           0xD                         2 byte elements where each element is␣
↪the index value of a bit,
                                       from an bit vector, which was 1.
           0xE                         4 byte elements where each element is␣
↪the index value of a bit,
                                       from an bit vector, which was 1.
           0xF                         Reserved

36.2.1.1.7. Application Data Integrity (ADI)

          On platforms which support ADI, the ADI version number may be␣
↪specified for each separate memory
          access type used in the CCB command. ADI checking only occurs␣
↪when reading data. When writing data,
          the specified ADI version number overwrites any existing ADI␣
↪value in memory.

          An ADI version value of 0 or 0xF indicates the ADI checking is␣
↪disabled for that data access, even if it is
          enabled in memory. By setting the appropriate flag in CCB_SUBMIT␣
```

```
↪(Section 36.3.1, “ccb_submit”) it is
         also an option to disable ADI checking for all inputs accessed␣
↪via virtual address for all CCBs submitted
         during that hypercall invocation.

         The ADI value is only guaranteed to be checked on the first 64␣
↪bytes of each data access. Mismatches on
         subsequent data chunks may not be detected, so guest software␣
↪should be careful to use page size checking
         to protect against buffer overruns.

36.2.1.1.8. Page size checking

         All data accesses used in CCB commands must be bounded within a␣
↪single memory page. When addresses
         are provided using a virtual address, the page size for checking␣
↪is extracted from the TTE for that virtual
         address. When using real addresses, the guest must supply the␣
↪page size in the same field as the address
         value. The page size must be one of the sizes supported by the␣
↪underlying virtual machine. Using a value
         that is not supported may result in the CCB submission being␣
↪rejected or the generation of a CCB parsing
         error in the completion area.


                                                     515

                                             Coprocessor services


36.2.1.2. Extract command

       Converts an input vector in one format to an output vector in␣
↪another format. All input format types are
       supported.

       The only supported output format is a padded, byte-aligned output␣
↪stream, using output codes 0x0 - 0x4.
       When the specified output element size is larger than the␣
↪extracted input element size, zeros are padded to
       the extracted input element. First, if the decompressed input size␣
↪is not a whole number of bytes, 0 bits are
       padded to the most significant bit side till the next byte␣
↪boundary. Next, if the output element size is larger
       than the byte padded input element, bytes of value 0 are added␣
↪based on the Padding Direction bit in the
       CCB. If the output element size is smaller than the byte-padded␣
↪input element size, the input element is
       truncated by dropped from the least significant byte side until␣
↪the selected output size is reached.
```

```
      The return value of the CCB completion area is invalid. The␣
↪"number of elements processed" field in the
      CCB completion area will be valid.

      The extract CCB is a 64-byte "short format" CCB.

      The extract CCB command format can be specified by the following␣
↪packed C structure for a big-endian
      machine:


               struct extract_ccb {
                      uint32_t header;
                      uint32_t control;
                      uint64_t completion;
                      uint64_t primary_input;
                      uint64_t data_access_control;
                      uint64_t secondary_input;
                      uint64_t reserved;
                      uint64_t output;
                      uint64_t table;
               };


      The exact field offsets, sizes, and composition are as follows:

       Offset         Size           Field Description
       0              4              CCB header (Table 36.1, "CCB␣
↪Header Format")
       4              4              Command control
                                     Bits          Field Description
                                     [31:28]       Primary Input Format␣
↪(see Section 36.2.1.1.1, "Primary Input
                                                   Format")
                                     [27:23]       Primary Input Element␣
↪Size (see Section 36.2.1.1.2, "Primary
                                                   Input Element Size")
                                     [22:20]       Primary Input Starting␣
↪Offset (see Section 36.2.1.1.5, "Input
                                                   Element Offsets")
                                     [19]          Secondary Input Format␣
↪(see Section 36.2.1.1.3, "Secondary
                                                   Input Format")
                                     [18:16]       Secondary Input␣
↪Starting Offset (see Section 36.2.1.1.5, "Input
                                                   Element Offsets")


                                                      516
```

```
                            Coprocessor services


    Offset   Size   Field Description
                    Bits         Field Description
                    [15:14]      Secondary Input Element Size (see Section 36.
    ↪2.1.1.4,
                                 “Secondary Input Element Size”
                    [13:10]      Output Format (see Section 36.2.1.1.6,␣
    ↪“Output Format”)
                    [9]          Padding Direction selector: A value of 1␣
    ↪causes padding bytes
                                 to be added to the left side of output␣
    ↪elements. A value of 0
                                 causes padding bytes to be added to the right␣
    ↪side of output
                                 elements.
                    [8:0]        Reserved
    8        8      Completion
                    Bits         Field Description
                    [63:60]      ADI version (see Section 36.2.1.1.7,␣
    ↪“Application Data
                                 Integrity (ADI)”)
                    [59]         If set to 1, a virtual device interrupt will␣
    ↪be generated using
                                 the device interrupt number specified in the␣
    ↪lower bits of this
                                 completion word. If 0, the lower bits of this␣
    ↪completion word
                                 are ignored.
                    [58:6]       Completion area address bits [58:6]. Address␣
    ↪type is
                                 determined by CCB header.
                    [5:0]        Virtual device interrupt number for␣
    ↪completion interrupt, if
                                 enabled.
    16       8      Primary Input
                    Bits         Field Description
                    [63:60]      ADI version (see Section 36.2.1.1.7,␣
    ↪“Application Data
                                 Integrity (ADI)”)
                    [59:56]      If using real address, these bits should be␣
    ↪filled in with the
                                 page size code for the page boundary checking␣
    ↪the guest wants
                                 the virtual machine to use when accessing␣
    ↪this data stream
                                 (checking is only guaranteed to be performed␣
    ↪when using API
                                 version 1.1 and later). If using a virtual␣
    ↪address, this field will
```

```
                              be used as as primary input address bits␣
 ↪[59:56].
                [55:0]        Primary input address bits [55:0]. Address␣
 ↪type is determined
                              by CCB header.
24       8      Data Access Control
                Bits          Field Description
                [63:62]       Flow Control
                              Value      Description
                              0b'00      Disable flow control
                              0b'01      Enable flow control (only valid␣
 ↪with "ORCL,sun4v-
                                         dax-fc" compatible virtual device␣
 ↪variants)
                              0b'10      Reserved
                              0b'11      Reserved
                [61:60]       Reserved (API 1.0)


                                 517

                         Coprocessor services


Offset   Size   Field Description
                Bits         Field Description
                             Pipeline target (API 2.0)
                             Value      Description
                             0b'00      Connect to primary input
                             0b'01      Connect to secondary input
                             0b'10      Reserved
                             0b'11      Reserved
                [59:40]      Output buffer size given in units of 64 bytes,␣
 ↪minus 1. Value of
                             0 means 64 bytes, value of 1 means 128 bytes,␣
 ↪etc. Buffer size is
                             only enforced if flow control is enabled in␣
 ↪Flow Control field.
                [39:32]      Reserved
                [31:30]      Output Data Cache Allocation
                             Value      Description
                             0b'00      Do not allocate cache lines for␣
 ↪output data stream.
                             0b'01      Force cache lines for output data␣
 ↪stream to be
                                        allocated in the cache that is␣
 ↪local to the submitting
                                        virtual cpu.
                             0b'10      Allocate cache lines for output␣
 ↪data stream, but allow
                                        existing cache lines associated␣
```

```
↪with the data to remain
                                        in their current cache instance.␣
↪Any memory not
                                        already in cache will be allocated␣
↪in the cache local
                                        to the submitting virtual cpu.
                               0b'11      Reserved
                   [29:26]     Reserved
                   [25:24]     Primary Input Length Format
                               Value      Description
                               0b'00      Number of primary symbols
                               0b'01      Number of primary bytes
                               0b'10      Number of primary bits
                               0b'11      Reserved
                   [23:0]      Primary Input Length
                               Format                 Field Value
                               # of primary symbols   Number of input␣
↪elements to process,
                                                      minus 1. Command␣
↪execution stops
                                                      once count is␣
↪reached.
                               # of primary bytes     Number of input␣
↪bytes to process,
                                                      minus 1. Command␣
↪execution stops
                                                      once count is␣
↪reached. The count is
                                                      done before any␣
↪decompression or
                                                      decoding.
                               # of primary bits      Number of input␣
↪bits to process,
                                                      minus 1. Command␣
↪execution stops


                                  518

                                                Coprocessor services


         Offset          Size           Field Description
                                         Bits         Field Description
                                                      Format           ␣
↪     Field Value
                                                                       ␣
↪     once count is reached. The count is
                                                                       ␣
↪     done before any decompression or
```

```
                                                                              ␣
↪     decoding, and does not include any
                                                                              ␣
↪     bits skipped by the Primary Input
                                                                              ␣
↪     Offset field value of the command
                                                                              ␣
↪     control word.
       32              8              Secondary Input, if used by Primary␣
↪Input Format. Same fields as Primary
                                      Input.
       40              8              Reserved
       48              8              Output (same fields as Primary␣
↪Input)
       56              8              Symbol Table (if used by Primary␣
↪Input)
                                       Bits         Field Description
                                       [63:60]      ADI version (see␣
↪Section 36.2.1.1.7, “Application Data
                                                    Integrity (ADI)”)
                                       [59:56]      If using real address,
↪ these bits should be filled in with the
                                                    page size code for␣
↪the page boundary checking the guest wants
                                                    the virtual machine␣
↪to use when accessing this data stream
                                                    (checking is only␣
↪guaranteed to be performed when using API
                                                    version 1.1 and␣
↪later). If using a virtual address, this field will
                                                    be used as as symbol␣
↪table address bits [59:56].
                                       [55:4]       Symbol table address␣
↪bits [55:4]. Address type is determined
                                                    by CCB header.
                                       [3:0]        Symbol table version
                                                    Value     Description
                                                    0         Huffman␣
↪encoding. Must use 64 byte aligned table
                                                              address.␣
↪(Only available when using version 0 CCBs)
                                                    1         OZIP␣
↪encoding. Must use 16 byte aligned table
                                                              address.␣
↪(Only available when using version 1 CCBs)


36.2.1.3. Scan commands

        The scan commands search a stream of input data elements for␣
↪values which match the selection criteria.
```

```
        All the input format types are supported. There are multiple␣
↪formats for the scan commands, allowing the
        scan to search for exact matches to one value, exact matches to␣
↪either of two values, or any value within
        a specified range. The specific type of scan is indicated by the␣
↪command code in the CCB header. For the
        scan range commands, the boundary conditions can be specified as␣
↪greater-than-or-equal-to a value, less-
        than-or-equal-to a value, or both by using two boundary values.

        There are two supported formats for the output stream: the bit␣
↪vector and index array formats (codes 0x8,
        0xD, and 0xE). For the standard scan command using the bit vector␣
↪output, for each input element there
        exists one bit in the vector that is set if the input element␣
↪matched the scan criteria, or clear if not. The
        inverted scan command inverts the polarity of the bits in the␣
↪output. The most significant bit of the first
        byte of the output stream corresponds to the first element in the␣
↪input stream. The standard index array
        output format contains one array entry for each input element that␣
↪matched the scan criteria. Each array



                                                                      519

                                                     Coprocessor services


entry is the index of an input element that matched the scan criteria. An␣
↪inverted scan command produces
a similar array, but of all the input elements which did NOT match the␣
↪scan criteria.

The return value of the CCB completion area contains the number of input␣
↪elements found which match
the scan criteria (or number that did not match for the inverted scans).␣
↪The “number of elements processed”
field in the CCB completion area will be valid, indicating the number of␣
↪input elements processed.

These commands are 128-byte “long format” CCBs.

The scan CCB command format can be specified by the following packed C␣
↪structure for a big-endian
machine:


         struct scan_ccb         {
                uint32_t         header;
                uint32_t        control;
                uint64_t        completion;
                uint64_t        primary_input;
                uint64_t        data_access_control;
                uint64_t        secondary_input;
                uint64_t        match_criteria0;
                uint64_t        output;
                uint64_t        table;
                uint64_t        match_criteria1;
                uint64_t        match_criteria2;
                uint64_t        match_criteria3;
                uint64_t        reserved[5];
        };


    The exact field offsets, sizes, and composition are as follows:

    Offset          Size            Field Description
    0               4               CCB header (Table 36.1, “CCB Header Format”)
    4               4               Command control
                                    Bits            Field Description
                                    [31:28]         Primary Input Format (see␣
↪Section 36.2.1.1.1, “Primary Input
                                                    Format”)
                                    [27:23]         Primary Input Element Size␣
↪(see Section 36.2.1.1.2, “Primary
                                                    Input Element Size”)
                                    [22:20]         Primary Input Starting Offset␣
↪(see Section 36.2.1.1.5, “Input
                                                    Element Offsets”)
                                    [19]            Secondary Input Format (see␣
↪Section 36.2.1.1.3, “Secondary
                                                    Input Format”)
                                    [18:16]         Secondary Input Starting␣
↪Offset (see Section 36.2.1.1.5, “Input
                                                    Element Offsets”)
                                    [15:14]         Secondary Input Element Size␣
↪(see Section 36.2.1.1.4,
                                                    “Secondary Input Element Size”
                                    [13:10]         Output Format (see Section 36.
↪2.1.1.6, “Output Format”)
                                    [9:5]           Operand size for first scan␣
↪criteria value. In a scan value
                                                    operation, this is one of two␣
↪potential exact match values.
                                                    In a scan range operation,␣
↪this is the size of the upper range

                                                    520
```

```
                              Coprocessor services


    Offset   Size   Field Description
                    Bits          Field Description
                                  boundary. The value of this field is the␣
     ↪number of bytes in the
                                  operand, minus 1. Values 0xF-0x1E are␣
     ↪reserved. A value of
                                  0x1F indicates this operand is not in use for␣
     ↪this scan operation.
                    [4:0]         Operand size for second scan criteria value.␣
     ↪In a scan value
                                  operation, this is one of two potential exact␣
     ↪match values.
                                  In a scan range operation, this is the size␣
     ↪of the lower range
                                  boundary. The value of this field is the␣
     ↪number of bytes in the
                                  operand, minus 1. Values 0xF-0x1E are␣
     ↪reserved. A value of
                                  0x1F indicates this operand is not in use for␣
     ↪this scan operation.
    8        8      Completion (same fields as Section 36.2.1.2, “Extract␣
     ↪command”)
    16       8      Primary Input (same fields as Section 36.2.1.2, “Extract␣
     ↪command”)
    24       8      Data Access Control (same fields as Section 36.2.1.2,␣
     ↪“Extract command”)
    32       8      Secondary Input, if used by Primary Input Format. Same␣
     ↪fields as Primary
                    Input.
    40       4      Most significant 4 bytes of first scan criteria operand.␣
     ↪If first operand is less
                    than 4 bytes, the value is left-aligned to the lowest␣
     ↪address bytes.
    44       4      Most significant 4 bytes of second scan criteria operand.␣
     ↪If second operand
                    is less than 4 bytes, the value is left-aligned to the␣
     ↪lowest address bytes.
    48       8      Output (same fields as Primary Input)
    56       8      Symbol Table (if used by Primary Input). Same fields as␣
     ↪Section 36.2.1.2,
                    “Extract command”
    64       4      Next 4 most significant bytes of first scan criteria␣
     ↪operand occurring after the
                    bytes specified at offset 40, if needed by the operand␣
     ↪size. If first operand
                    is less than 8 bytes, the valid bytes are left-aligned to␣
     ↪the lowest address.
    68       4      Next 4 most significant bytes of second scan criteria␣
```

```
↪operand occurring after
                the bytes specified at offset 44, if needed by the operand␣
↪size. If second
                operand is less than 8 bytes, the valid bytes are␣
↪left-aligned to the lowest
                address.
72      4       Next 4 most significant bytes of first scan criteria␣
↪operand occurring after the
                bytes specified at offset 64, if needed by the operand␣
↪size. If first operand
                is less than 12 bytes, the valid bytes are left-aligned to␣
↪the lowest address.
76      4       Next 4 most significant bytes of second scan criteria␣
↪operand occurring after
                the bytes specified at offset 68, if needed by the operand␣
↪size. If second
                operand is less than 12 bytes, the valid bytes are␣
↪left-aligned to the lowest
                address.
80      4       Next 4 most significant bytes of first scan criteria␣
↪operand occurring after the
                bytes specified at offset 72, if needed by the operand␣
↪size. If first operand
                is less than 16 bytes, the valid bytes are left-aligned to␣
↪the lowest address.
84      4       Next 4 most significant bytes of second scan criteria␣
↪operand occurring after
                the bytes specified at offset 76, if needed by the operand␣
↪size. If second
                operand is less than 16 bytes, the valid bytes are␣
↪left-aligned to the lowest
                address.




                                521

                                            Coprocessor services


36.2.1.4. Translate commands

        The translate commands takes an input array of indices, and a␣
↪table of single bit values indexed by those
        indices, and outputs a bit vector or index array created by␣
↪reading the tables bit value at each index in
        the input array. The output should therefore contain exactly one␣
↪bit per index in the input data stream,
        when outputting as a bit vector. When outputting as an index array,
↪ the number of elements depends on the
```

```
        values read in the bit table, but will always be less than, or␣
↪equal to, the number of input elements. Only
        a restricted subset of the possible input format types are␣
↪supported. No variable width or Huffman/OZIP
        encoded input streams are allowed. The primary input data element␣
↪size must be 3 bytes or less.

        The maximum table index size allowed is 15 bits, however, larger␣
↪input elements may be used to provide
        additional processing of the output values. If 2 or 3 byte values␣
↪are used, the least significant 15 bits are
        used as an index into the bit table. The most significant 9 bits␣
↪(when using 3-byte input elements) or single
        bit (when using 2-byte input elements) are compared against a␣
↪fixed 9-bit test value provided in the CCB.
        If the values match, the value from the bit table is used as the␣
↪output element value. If the values do not
        match, the output data element value is forced to 0.

        In the inverted translate operation, the bit value read from bit␣
↪table is inverted prior to its use. The additional
        additional processing based on any additional non-index bits␣
↪remains unchanged, and still forces the output
        element value to 0 on a mismatch. The specific type of translate␣
↪command is indicated by the command
        code in the CCB header.

        There are two supported formats for the output stream: the bit␣
↪vector and index array formats (codes 0x8,
        0xD, and 0xE). The index array format is an array of indices of␣
↪bits which would have been set if the
        output format was a bit array.

        The return value of the CCB completion area contains the number of␣
↪bits set in the output bit vector,
        or number of elements in the output index array. The “number of␣
↪elements processed” field in the CCB
        completion area will be valid, indicating the number of input␣
↪elements processed.

        These commands are 64-byte “short format” CCBs.

        The translate CCB command format can be specified by the following␣
↪packed C structure for a big-endian
        machine:


                struct translate_ccb {
                        uint32_t header;
                        uint32_t control;
                        uint64_t completion;
```

```
                              uint64_t primary_input;
                              uint64_t data_access_control;
                              uint64_t secondary_input;
                              uint64_t reserved;
                              uint64_t output;
                              uint64_t table;
                       };


          The exact field offsets, sizes, and composition are as follows:


          Offset          Size             Field Description
          0               4                CCB header (Table 36.1, “CCB␣
 ↪Header Format”)


                                                          522

                              Coprocessor services


  Offset    Size    Field Description
  4         4       Command control
                    Bits         Field Description
                    [31:28]      Primary Input Format (see Section 36.2.1.1.1,␣
 ↪“Primary Input
                                 Format”)
                    [27:23]      Primary Input Element Size (see Section 36.2.
 ↪1.1.2, “Primary
                                 Input Element Size”)
                    [22:20]      Primary Input Starting Offset (see Section 36.
 ↪2.1.1.5, “Input
                                 Element Offsets”)
                    [19]         Secondary Input Format (see Section 36.2.1.1.
 ↪3, “Secondary
                                 Input Format”)
                    [18:16]      Secondary Input Starting Offset (see Section␣
 ↪36.2.1.1.5, “Input
                                 Element Offsets”)
                    [15:14]      Secondary Input Element Size (see Section 36.
 ↪2.1.1.4,
                                 “Secondary Input Element Size”
                    [13:10]      Output Format (see Section 36.2.1.1.6,␣
 ↪“Output Format”)
                    [9]          Reserved
                    [8:0]        Test value used for comparison against the␣
 ↪most significant bits
                                 in the input values, when using 2 or 3 byte␣
 ↪input elements.
  8         8       Completion (same fields as Section 36.2.1.2, “Extract␣
↪command”
16       8      Primary Input (same fields as Section 36.2.1.2, “Extract␣
↪command”
24       8      Data Access Control (same fields as Section 36.2.1.2,␣
↪“Extract command”,
                except Primary Input Length Format may not use the 0x0␣
↪value)
32       8      Secondary Input, if used by Primary Input Format. Same␣
↪fields as Primary
                Input.
40       8      Reserved
48       8      Output (same fields as Primary Input)
56       8      Bit Table
                Bits         Field Description
                [63:60]      ADI version (see Section 36.2.1.1.7,␣
↪“Application Data
                             Integrity (ADI)”)
                [59:56]      If using real address, these bits should be␣
↪filled in with the
                             page size code for the page boundary checking␣
↪the guest wants
                             the virtual machine to use when accessing␣
↪this data stream
                             (checking is only guaranteed to be performed␣
↪when using API
                             version 1.1 and later). If using a virtual␣
↪address, this field will
                             be used as as bit table address bits [59:56]
                [55:4]       Bit table address bits [55:4]. Address type␣
↪is determined by
                             CCB header. Address must be 64-byte aligned␣
↪(CCB version
                             0) or 16-byte aligned (CCB version 1).
                [3:0]        Bit table version
                             Value      Description
                             0          4KB table size
                             1          8KB table size


                                523

                                            Coprocessor services


36.2.1.5. Select command
        The select command filters the primary input data stream by using␣
↪a secondary input bit vector to determine
        which input elements to include in the output. For each bit set at␣
↪a given index N within the bit vector,
        the Nth input element is included in the output. If the bit is not␣
```

```
↪set, the element is not included. Only a
        restricted subset of the possible input format types are supported.
↪ No variable width or run length encoded
        input streams are allowed, since the secondary input stream is␣
↪used for the filtering bit vector.

        The only supported output format is a padded, byte-aligned output␣
↪stream. The stream follows the same
        rules and restrictions as padded output stream described in␣
↪Section 36.2.1.2, “Extract command”.

        The return value of the CCB completion area contains the number of␣
↪bits set in the input bit vector. The
        "number of elements processed" field in the CCB completion area␣
↪will be valid, indicating the number
        of input elements processed.

        The select CCB is a 64-byte “short format” CCB.

        The select CCB command format can be specified by the following␣
↪packed C structure for a big-endian
        machine:


                  struct select_ccb {
                        uint32_t header;
                        uint32_t control;
                        uint64_t completion;
                        uint64_t primary_input;
                        uint64_t data_access_control;
                        uint64_t secondary_input;
                        uint64_t reserved;
                        uint64_t output;
                        uint64_t table;
                  };


        The exact field offsets, sizes, and composition are as follows:

         Offset        Size             Field Description
         0             4                CCB header (Table 36.1, “CCB Header␣
↪Format”)
         4             4                Command control
                                        Bits         Field Description
                                        [31:28]      Primary Input Format␣
↪(see Section 36.2.1.1.1, “Primary Input
                                                     Format”)
                                        [27:23]      Primary Input Element␣
↪Size (see Section 36.2.1.1.2, “Primary
                                                     Input Element Size”)
                                        [22:20]      Primary Input Starting␣
```

```
↪Offset (see Section 36.2.1.1.5, “Input
                                                      Element Offsets”)
                                        [19]          Secondary Input Format␣
↪(see Section 36.2.1.1.3, “Secondary
                                                      Input Format”)
                                        [18:16]       Secondary Input␣
↪Starting Offset (see Section 36.2.1.1.5, “Input
                                                      Element Offsets”)
                                        [15:14]       Secondary Input Element␣
↪Size (see Section 36.2.1.1.4,
                                                      “Secondary Input␣
↪Element Size”


                                                         524

                                                  Coprocessor services


      Offset         Size               Field Description
                                        Bits          Field Description
                                        [13:10]       Output Format (see␣
↪Section 36.2.1.1.6, “Output Format”)
                                        [9]           Padding Direction␣
↪selector: A value of 1 causes padding bytes
                                                      to be added to the␣
↪left side of output elements. A value of 0
                                                      causes padding bytes␣
↪to be added to the right side of output
                                                      elements.
                                        [8:0]         Reserved
      8              8                  Completion (same fields as Section␣
↪36.2.1.2, “Extract command”
      16             8                  Primary Input (same fields as␣
↪Section 36.2.1.2, “Extract command”
      24             8                  Data Access Control (same fields as␣
↪Section 36.2.1.2, “Extract command”)
      32             8                  Secondary Bit Vector Input. Same␣
↪fields as Primary Input.
      40             8                  Reserved
      48             8                  Output (same fields as Primary␣
↪Input)
      56             8                  Symbol Table (if used by Primary␣
↪Input). Same fields as Section 36.2.1.2,
                                        “Extract command”

36.2.1.6. No-op and Sync commands
      The no-op (no operation) command is a CCB which has no processing␣
↪effect. The CCB, when processed
      by the virtual machine, simply updates the completion area with␣
↪its execution status. The CCB may have
```

```
      the serial-conditional flags set in order to restrict when it␣
↪executes.

      The sync command is a variant of the no-op command which with␣
↪restricted execution timing. A sync
      command CCB will only execute when all previous commands submitted␣
↪in the same request have
      completed. This is stronger than the conditional flag sequencing,␣
↪which is only dependent on a single
      previous serial CCB. While the relative ordering is guaranteed,␣
↪virtual machine implementations with
      shared hardware resources may cause the sync command to wait for␣
↪longer than the minimum required
      time.

      The return value of the CCB completion area is invalid for these␣
↪CCBs. The “number of elements
      processed” field is also invalid for these CCBs.

      These commands are 64-byte “short format” CCBs.

      The no-op CCB command format can be specified by the following␣
↪packed C structure for a big-endian
      machine:


               struct nop_ccb {
                      uint32_t header;
                      uint32_t control;
                      uint64_t completion;
                      uint64_t reserved[6];
               };


      The exact field offsets, sizes, and composition are as follows:

      Offset          Size            Field Description
      0               4               CCB header (Table 36.1, “CCB Header␣
↪Format”)


                                                            525

                                                Coprocessor services


     Offset          Size            Field Description
     4               4               Command control
                                     Bits         Field Description
                                     [31]         If set, this CCB functions␣
↪as a Sync command. If clear, this
```

```
                                               CCB functions as a No-op␣
↪command.
                                   [30:0]      Reserved
       8             8             Completion (same fields as Section 36.2.
↪1.2, “Extract command”
       16            46            Reserved

36.2.2. CCB Completion Area
       All CCB commands use a common 128-byte Completion Area format,␣
↪which can be specified by the
       following packed C structure for a big-endian machine:


                struct completion_area {
                       uint8_t status_flag;
                       uint8_t error_note;
                       uint8_t rsvd0[2];
                       uint32_t error_values;
                       uint32_t output_size;
                       uint32_t rsvd1;
                       uint64_t run_time;
                       uint64_t run_stats;
                       uint32_t elements;
                       uint8_t rsvd2[20];
                       uint64_t return_value;
                       uint64_t extra_return_value[8];
                };


       The Completion Area must be a 128-byte aligned memory location. The␣
↪exact layout can be described
       using byte offsets and sizes relative to the memory base:

       Offset        Size          Field Description
       0             1             CCB execution status
                                   0x0                    Command not yet␣
↪completed
                                   0x1                    Command ran and␣
↪succeeded
                                   0x2                    Command ran and␣
↪failed (partial results may be been
                                                          produced)
                                   0x3                    Command ran and␣
↪was killed (partial execution may
                                                          have occurred)
                                   0x4                    Command was not run
                                   0x5-0xF                Reserved
       1             1             Error reason code
                                   0x0                    Reserved
                                   0x1                    Buffer overflow
```

```
                                                    526

                                              Coprocessor services


    Offset          Size          Field Description
                                   0x2                 CCB decoding error
                                   0x3                 Page overflow
                                   0x4-0x6             Reserved
                                   0x7                 Command was killed
                                   0x8                 Command execution␣
     ↪timeout
                                   0x9                 ADI miscompare error
                                   0xA                 Data format error
                                   0xB-0xD             Reserved
                                   0xE                 Unexpected hardware␣
     ↪error (Do not retry)
                                   0xF                 Unexpected hardware␣
     ↪error (Retry is ok)
                                   0x10-0x7F           Reserved
                                   0x80                Partial Symbol Warning
                                   0x81-0xFF           Reserved
    2               2             Reserved
    4               4             If a partial symbol warning was generated,␣
     ↪this field contains the number
                                  of remaining bits which were not decoded.
    8               4             Number of bytes of output produced
    12              4             Reserved
    16              8             Runtime of command (unspecified time units)
    24              8             Reserved
    32              4             Number of elements processed
    36              20            Reserved
    56              8             Return value
    64              64            Extended return value

    The CCB completion area should be treated as read-only by guest software.␣
     ↪The CCB execution status
    byte will be cleared by the Hypervisor to reflect the pending execution␣
     ↪status when the CCB is submitted
    successfully. All other fields are considered invalid upon CCB submission␣
     ↪until the CCB execution status
    byte becomes non-zero.

    CCBs which complete with status 0x2 or 0x3 may produce partial results and/
     ↪or side effects due to partial
    execution of the CCB command. Some valid data may be accessible depending␣
     ↪on the fault type, however,
    it is recommended that guest software treat the destination buffer as␣
     ↪being in an unknown state. If a CCB
    completes with a status byte of 0x2, the error reason code byte can be␣
```

```
↪read to determine what corrective
action should be taken.

A buffer overflow indicates that the results of the operation exceeded the␣
↪size of the output buffer indicated
in the CCB. The operation can be retried by resubmitting the CCB with a␣
↪larger output buffer.

A CCB decoding error indicates that the CCB contained some invalid field␣
↪values. It may be also be
triggered if the CCB output is directed at a non-existent secondary input␣
↪and the pipelining hint is followed.

A page overflow error indicates that the operation required accessing a␣
↪memory location beyond the page
size associated with a given address. No data will have been read or␣
↪written past the page boundary, but
partial results may have been written to the destination buffer. The CCB␣
↪can be resubmitted with a larger
page size memory allocation to complete the operation.


                                                    527

                                                  Coprocessor services


      In the case of pipelined CCBs, a page overflow error will be␣
↪triggered if the output from the pipeline source
      CCB ends before the input of the pipeline target CCB. Page␣
↪boundaries are ignored when the pipeline
      hint is followed.

      Command kill indicates that the CCB execution was halted or␣
↪prevented by use of the ccb_kill API call.

      Command timeout indicates that the CCB execution began, but did not␣
↪complete within a pre-determined
      limit set by the virtual machine. The command may have produced␣
↪some or no output. The CCB may be
      resubmitted with no alterations.

      ADI miscompare indicates that the memory buffer version specified␣
↪in the CCB did not match the value
      in memory when accessed by the virtual machine. Guest software␣
↪should not attempt to resubmit the CCB
      without determining the cause of the version mismatch.

      A data format error indicates that the input data stream did not␣
↪follow the specified data input formatting
      selected in the CCB.
```

```
        Some CCBs which encounter hardware errors may be resubmitted␣
↪without change. Persistent hardware
        errors may result in multiple failures until RAS software can␣
↪identify and isolate the faulty component.

        The output size field indicates the number of bytes of valid output␣
↪in the destination buffer. This field is
        not valid for all possible CCB commands.

        The runtime field indicates the execution time of the CCB command␣
↪once it leaves the internal virtual
        machine queue. The time units are fixed, but unspecified, allowing␣
↪only relative timing comparisons
        by guest software. The time units may also vary by hardware␣
↪platform, and should not be construed to
        represent any absolute time value.

        Some data query commands process data in units of elements. If␣
↪applicable to the command, the number of
        elements processed is indicated in the listed field. This field is␣
↪not valid for all possible CCB commands.

        The return value and extended return value fields are output␣
↪locations for commands which do not use
        a destination output buffer, or have secondary return results. The␣
↪field is not valid for all possible CCB
        commands.

36.3. Hypervisor API Functions
36.3.1. ccb_submit
        trap#                FAST_TRAP
        function#            CCB_SUBMIT
        arg0                 address
        arg1                 length
        arg2                 flags
        arg3                 reserved
        ret0                 status
        ret1                 length
        ret2                 status data
        ret3                 reserved

        Submit one or more coprocessor control blocks (CCBs) for evaluation␣
↪and processing by the virtual
        machine. The CCBs are passed in a linear array indicated by address.
↪ length indicates the size of
        the array in bytes.


                                                            528
```

```
                                          Coprocessor services


    The address should be aligned to the size indicated by length, rounded up␣
     ↪to the nearest power of
    two. Virtual machines implementations may reject submissions which do not␣
     ↪adhere to that alignment.
    length must be a multiple of 64 bytes. If length is zero, the maximum␣
     ↪supported array length will be
    returned as length in ret1. In all other cases, the length value in ret1␣
     ↪will reflect the number of bytes
    successfully consumed from the input CCB array.

          Implementation note
          Virtual machines should never reject submissions based on the␣
     ↪alignment of address if the
          entire array is contained within a single memory page of the␣
     ↪smallest page size supported by the
          virtual machine.

    A guest may choose to submit addresses used in this API function,␣
     ↪including the CCB array address,
    as either a real or virtual addresses, with the type of each address␣
     ↪indicated in flags. Virtual addresses
    must be present in either the TLB or an active TSB to be processed. The␣
     ↪translation context for virtual
    addresses is determined by a combination of CCB contents and the flags␣
     ↪argument.

    The flags argument is divided into multiple fields defined as follows:


    Bits              Field Description
    [63:16]           Reserved
    [15]              Disable ADI for VA reads (in API 2.0)
                      Reserved (in API 1.0)
    [14]              Virtual addresses within CCBs are translated in privileged␣
     ↪context
    [13:12]           Alternate translation context for virtual addresses within␣
     ↪CCBs:
                       0b'00        CCBs requesting alternate context are␣
     ↪rejected
                       0b'01        Reserved
                       0b'10        CCBs requesting alternate context use␣
     ↪secondary context
                       0b'11        CCBs requesting alternate context use␣
     ↪nucleus context
    [11:9]            Reserved
    [8]               Queue info flag
    [7]               All-or-nothing flag
    [6]               If address is a virtual address, treat its translation␣
```

```
↪context as privileged
[5:4]           Address type of address:
                 0b'00       Real address
                 0b'01       Virtual address in primary context
                 0b'10       Virtual address in secondary context
                 0b'11       Virtual address in nucleus context
[3:2]           Reserved
[1:0]           CCB command type:
                 0b'00       Reserved
                 0b'01       Reserved
                 0b'10       Query command
                 0b'11       Reserved



                                                        529

                                                       Coprocessor services


        The CCB submission type and address type for the CCB array must␣
↪be provided in the flags argument.
        All other fields are optional values which change the default␣
↪behavior of the CCB processing.

        When set to one, the "Disable ADI for VA reads" bit will turn off␣
↪ADI checking when using a virtual
        address to load data. ADI checking will still be done when␣
↪loading real-addressed memory. This bit is only
        available when using major version 2 of the coprocessor API group;
↪ at major version 1 it is reserved. For
        more information about using ADI and DAX, see Section 36.2.1.1.7,␣
↪“Application Data Integrity (ADI)”.

        By default, all virtual addresses are treated as user addresses.␣
↪If the virtual address translations are
        privileged, they must be marked as such in the appropriate flags␣
↪field. The virtual addresses used within
        the submitted CCBs must all be translated with the same privilege␣
↪level.

        By default, all virtual addresses used within the submitted CCBs␣
↪are translated using the primary context
        active at the time of the submission. The address type field␣
↪within a CCB allows each address to request
        translation in an alternate address context. The address context␣
↪used when the alternate address context is
        requested is selected in the flags argument.

        The all-or-nothing flag specifies whether the virtual machine␣
↪should allow partial submissions of the
```

```
        input CCB array. When using CCBs with serial-conditional flags,␣
↪it is strongly recommended to use
        the all-or-nothing flag to avoid broken conditional chains. Using␣
↪long CCB chains on a machine under
        high coprocessor load may make this impractical, however, and␣
↪require submitting without the flag.
        When submitting serial-conditional CCBs without the␣
↪all-or-nothing flag, guest software must manually
        implement the serial-conditional behavior at any point where the␣
↪chain was not submitted in a single API
        call, and resubmission of the remaining CCBs should clear any␣
↪conditional flag that might be set in the
        first remaining CCB. Failure to do so will produce indeterminate␣
↪CCB execution status and ordering.

        When the all-or-nothing flag is not specified, callers should␣
↪check the value of length in ret1 to determine
        how many CCBs from the array were successfully submitted. Any␣
↪remaining CCBs can be resubmitted
        without modifications.

        The value of length in ret1 is also valid when the API call␣
↪returns an error, and callers should always
        check its value to determine which CCBs in the array were already␣
↪processed. This will additionally
        identify which CCB encountered the processing error, and was not␣
↪submitted successfully.

        If the queue info flag is used during submission, and at least␣
↪one CCB was successfully submitted, the
        length value in ret1 will be a multi-field value defined as␣
↪follows:
         Bits          Field Description
         [63:48]       DAX unit instance identifier
         [47:32]       DAX queue instance identifier
         [31:16]       Reserved
         [15:0]        Number of CCB bytes successfully submitted

        The value of status data depends on the status value. See error␣
↪status code descriptions for details.
        The value is undefined for status values that do not specifically␣
↪list a value for the status data.

        The API has a reserved input and output register which will be␣
↪used in subsequent minor versions of this
        API function. Guest software implementations should treat that␣
↪register as voltile across the function call
        in order to maintain forward compatibility.

36.3.1.1. Errors
         EOK                       One or more CCBs have been accepted␣
```

```
↪and enqueued in the virtual machine
                                    and no errors were been encountered␣
↪during submission. Some submitted
                                    CCBs may not have been enqueued due to␣
↪internal virtual machine limitations,
                                    and may be resubmitted without changes.


                                                      530

                             Coprocessor services


EWOULDBLOCK    An internal resource conflict within the virtual machine␣
↪has prevented it from
               being able to complete the CCB submissions sufficiently␣
↪quickly, requiring
               it to abandon processing before it was complete. Some CCBs␣
↪may have been
               successfully enqueued prior to the block, and all remaining␣
↪CCBs may be
               resubmitted without changes.
EBADALIGN      CCB array is not on a 64-byte boundary, or the array length␣
↪is not a multiple
               of 64 bytes.
ENORADDR       A real address used either for the CCB array, or within one␣
↪of the submitted
               CCBs, is not valid for the guest. Some CCBs may have been␣
↪enqueued prior
               to the error being detected.
ENOMAP         A virtual address used either for the CCB array, or within␣
↪one of the submitted
               CCBs, could not be translated by the virtual machine using␣
↪either the TLB
               or TSB contents. The submission may be retried after adding␣
↪the required
               mapping, or by converting the virtual address into a real␣
↪address. Due to the
               shared nature of address translation resources, there is no␣
↪theoretical limit on
               the number of times the translation may fail, and it is␣
↪recommended all guests
               implement some real address based backup. The virtual␣
↪address which failed
               translation is returned as status data in ret2. Some CCBs␣
↪may have been
               enqueued prior to the error being detected.
EINVAL         The virtual machine detected an invalid CCB during␣
↪submission, or invalid
               input arguments, such as bad flag values. Note that not all␣
↪invalid CCB values
```

```
                  will be detected during submission, and some may be␣
↪reported as errors in the
                  completion area instead. Some CCBs may have been enqueued␣
↪prior to the
                  error being detected. This error may be returned if the CCB␣
↪version is invalid.
ETOOMANY          The request was submitted with the all-or-nothing flag set,␣
↪and the array size is
                  greater than the virtual machine can support in a single␣
↪request. The maximum
                  supported size for the current virtual machine can be␣
↪queried by submitting a
                  request with a zero length array, as described above.
ENOACCESS         The guest does not have permission to submit CCBs, or an␣
↪address used in a
                  CCBs lacks sufficient permissions to perform the required␣
↪operation (no write
                  permission on the destination buffer address, for example).␣
↪A virtual address
                  which fails permission checking is returned as status data␣
↪in ret2. Some
                  CCBs may have been enqueued prior to the error being␣
↪detected.
EUNAVAILABLE      The requested CCB operation could not be performed at this␣
↪time. The
                  restricted operation availability may apply only to the␣
↪first unsuccessfully
                  submitted CCB, or may apply to a larger scope. The status␣
↪should not be
                  interpreted as permanent, and the guest should attempt to␣
↪submit CCBs in
                  the future which had previously been unable to be performed.
↪ The status
                  data provides additional information about scope of the␣
↪restricted availability
                  as follows:
                  Value         Description
                  0             Processing for the exact CCB instance submitted␣
↪was unavailable,
                                and it is recommended the guest emulate the␣
↪operation. The
                                guest should continue to submit all other CCBs,␣
↪and assume no
                                restrictions beyond this exact CCB instance.
                  1             Processing is unavailable for all CCBs using␣
↪the requested opcode,
                                and it is recommended the guest emulate the␣
↪operation. The
                                guest should continue to submit all other CCBs␣
↪that use different
                                opcodes, but can expect continued rejections of␣
```

```
↪CCBs using the
                                            same opcode in the near future.

                                                   531

                                                                 Coprocessor services


                                                        Value     Description
                                                        2         Processing is unavailable␣
↪for all CCBs using the requested CCB
                                                                  version, and it is␣
↪recommended the guest emulate the operation.
                                                                  The guest should continue␣
↪to submit all other CCBs that use
                                                                  different CCB versions,␣
↪but can expect continued rejections of
                                                                  CCBs using the same CCB␣
↪version in the near future.
                                                        3         Processing is unavailable␣
↪for all CCBs on the submitting vcpu,
                                                                  and it is recommended the␣
↪guest emulate the operation or resubmit
                                                                  the CCB on a different␣
↪vcpu. The guest should continue to submit
                                                                  CCBs on all other vcpus␣
↪but can expect continued rejections of all
                                                                  CCBs on this vcpu in the␣
↪near future.
                                                        4         Processing is unavailable␣
↪for all CCBs, and it is recommended
                                                                  the guest emulate the␣
↪operation. The guest should expect all CCB
                                                                  submissions to be␣
↪similarly rejected in the near future.


36.3.2. ccb_info

         trap#                   FAST_TRAP
         function#               CCB_INFO
         arg0                    address
         ret0                    status
         ret1                    CCB state
         ret2                    position
         ret3                    dax
         ret4                    queue

        Requests status information on a previously submitted CCB. The␣
↪previously submitted CCB is identified
```

```
        by the 64-byte aligned real address of the CCBs completion area.

        A CCB can be in one of 4 states:


         State                      Value       Description
         COMPLETED                  0           The CCB has been fetched and␣
↪executed, and is no longer active in
                                                the virtual machine.
         ENQUEUED                   1           The requested CCB is current␣
↪in a queue awaiting execution.
         INPROGRESS                 2           The CCB has been fetched and␣
↪is currently being executed. It may still
                                                be possible to stop the␣
↪execution using the ccb_kill hypercall.
         NOTFOUND                   3           The CCB could not be located␣
↪in the virtual machine, and does not
                                                appear to have been executed.
↪ This may occur if the CCB was lost
                                                due to a hardware error, or␣
↪the CCB may not have been successfully
                                                submitted to the virtual␣
↪machine in the first place.

                Implementation note
                Some platforms may not be able to report CCBs that are␣
↪currently being processed, and therefore
                guest software should invoke the ccb_kill hypercall prior␣
↪to assuming the request CCB will never
                be executed because it was in the NOTFOUND state.


                                                         532

                                                Coprocessor services


        The position return value is only valid when the state is␣
↪ENQUEUED. The value returned is the number
        of other CCBs ahead of the requested CCB, to provide a relative␣
↪estimate of when the CCB may execute.

        The dax return value is only valid when the state is ENQUEUED.␣
↪The value returned is the DAX unit
        instance identifier for the DAX unit processing the queue where␣
↪the requested CCB is located. The value
        matches the value that would have been, or was, returned by ccb_
↪submit using the queue info flag.

        The queue return value is only valid when the state is ENQUEUED.␣
↪The value returned is the DAX
```

```
        queue instance identifier for the DAX unit processing the queue␣
↪where the requested CCB is located. The
        value matches the value that would have been, or was, returned by␣
↪ccb_submit using the queue info flag.

36.3.2.1. Errors

         EOK                         The request was processed and the CCB␣
↪state is valid.
         EBADALIGN                   address is not on a 64-byte aligned.
         ENORADDR                    The real address provided for address␣
↪is not valid.
         EINVAL                      The CCB completion area contents are␣
↪not valid.
         EWOULDBLOCK                 Internal resource constraints␣
↪prevented the CCB state from being queried at this
                                     time. The guest should retry the␣
↪request.
         ENOACCESS                   The guest does not have permission to␣
↪access the coprocessor virtual device
                                     functionality.

36.3.3. ccb_kill

         trap#           FAST_TRAP
         function#       CCB_KILL
         arg0            address
         ret0            status
         ret1            result

        Request to stop execution of a previously submitted CCB. The␣
↪previously submitted CCB is identified by
        the 64-byte aligned real address of the CCBs completion area.

        The kill attempt can produce one of several values in the result␣
↪return value, reflecting the CCB state
        and actions taken by the Hypervisor:

         Result                Value       Description
         COMPLETED             0           The CCB has been fetched and␣
↪executed, and is no longer active in
                                           the virtual machine. It could␣
↪not be killed and no action was taken.
         DEQUEUED              1           The requested CCB was still␣
↪enqueued when the kill request was
                                           submitted, and has been␣
↪removed from the queue. Since the CCB
                                           never began execution, no␣
↪memory modifications were produced by
                                           it, and the completion area␣
↪will never be updated. The same CCB may
```

```
                                                      be submitted again, if desired,
↪ with no modifications required.
         KILLED                2              The CCB had been fetched and␣
↪was being executed when the kill
                                                      request was submitted. The CCB␣
↪execution was stopped, and the CCB
                                                      is no longer active in the␣
↪virtual machine. The CCB completion area
                                                      will reflect the killed status,
↪ with the subsequent implications that
                                                      partial results may have been␣
↪produced. Partial results may include full

                                                               533

                                                        Coprocessor services


         Result                 Value        Description
                                               command execution if the␣
↪command was stopped just prior to writing
                                               to the completion area.
         NOTFOUND               3              The CCB could not be located␣
↪in the virtual machine, and does not
                                               appear to have been executed.␣
↪This may occur if the CCB was lost
                                               due to a hardware error, or␣
↪the CCB may not have been successfully
                                               submitted to the virtual␣
↪machine in the first place. CCBs in the state
                                               are guaranteed to never␣
↪execute in the future unless resubmitted.

36.3.3.1. Interactions with Pipelined CCBs

        If the pipeline target CCB is killed but the pipeline source CCB␣
↪was skipped, the completion area of the
        target CCB may contain status (4,0) "Command was skipped" instead␣
↪of (3,7) "Command was killed".

        If the pipeline source CCB is killed, the pipeline target CCB's␣
↪completion status may read (1,0) "Success".
        This does not mean the target CCB was processed; since the source␣
↪CCB was killed, there was no
        meaningful output on which the target CCB could operate.

36.3.3.2. Errors

         EOK                         The request was processed and the␣
↪result is valid.
```

```
        EBADALIGN                   address is not on a 64-byte aligned.
        ENORADDR                    The real address provided for address␣
↪is not valid.
        EINVAL                      The CCB completion area contents are␣
↪not valid.
        EWOULDBLOCK                 Internal resource constraints␣
↪prevented the CCB from being killed at this time.
                                    The guest should retry the request.
        ENOACCESS                   The guest does not have permission to␣
↪access the coprocessor virtual device
                                    functionality.

36.3.4. dax_info
        trap#            FAST_TRAP
        function#        DAX_INFO
        ret0             status
        ret1             Number of enabled DAX units
        ret2             Number of disabled DAX units

       Returns the number of DAX units that are enabled for the calling␣
↪guest to submit CCBs. The number of
       DAX units that are disabled for the calling guest are also␣
↪returned. A disabled DAX unit would have been
       available for CCB submission to the calling guest had it not been␣
↪offlined.

36.3.4.1. Errors

        EOK                         The request was processed and the␣
↪number of enabled/disabled DAX units
                                    are valid.




                                                              534
```

15.4 Feature status on sparc architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	ok
core	eBPF-JIT	HAVE_EBPF_JIT	ok
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok
core	thread-info-in-task	THREAD_INFO_IN_TASK	TODO

Table 1 – continued fron

Subsystem	Feature	Kconfig	Status
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	TODO
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	TODO
debug	KASAN	HAVE_ARCH_KASAN	TODO
debug	kcov	ARCH_HAS_KCOV	TODO
debug	kgdb	HAVE_ARCH_KGDB	ok
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok
debug	kprobes	HAVE_KPROBES	ok
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	TODO
debug	kretprobes	HAVE_KRETPROBES	ok
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	TODO
debug	uprobes	ARCH_SUPPORTS_UPROBES	ok
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	TODO
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	ok
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	ok
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok
perf	perf-regs	HAVE_PERF_REGS	TODO
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	TODO
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	TODO
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	TODO
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	TODO
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	TODO
time	clockevents	!LEGACY_TIMER_TICK	ok
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	---
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	ok
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	ok
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	TODO
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	TODO
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO
vm	ioremap_prot	HAVE_IOREMAP_PROT	TODO
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	ok

CHAPTER

SIXTEEN

X86-SPECIFIC DOCUMENTATION

16.1 The Linux/x86 Boot Protocol

On the x86 platform, the Linux kernel uses a rather complicated boot convention. This has evolved partially due to historical aspects, as well as the desire in the early days to have the kernel itself be a bootable image, the complicated PC memory model and due to changed expectations in the PC industry caused by the effective demise of real-mode DOS as a mainstream operating system.

Currently, the following versions of the Linux/x86 boot protocol exist.

Old kernels	zImage/Image support only. Some very early kernels may not even support a command line.
Protocol 2.00	(Kernel 1.3.73) Added bzImage and initrd support, as well as a formalized way to communicate between the boot loader and the kernel. setup.S made relocatable, although the traditional setup area still assumed writable.
Protocol 2.01	(Kernel 1.3.76) Added a heap overrun warning.
Protocol 2.02	(Kernel 2.4.0-test3-pre3) New command line protocol. Lower the conventional memory ceiling. No overwrite of the traditional setup area, thus making booting safe for systems which use the EBDA from SMM or 32-bit BIOS entry points. zImage deprecated but still supported.
Protocol 2.03	(Kernel 2.4.18-pre1) Explicitly makes the highest possible initrd address available to the bootloader.
Protocol 2.04	(Kernel 2.6.14) Extend the syssize field to four bytes.
Protocol 2.05	(Kernel 2.6.20) Make protected mode kernel relocatable. Introduce relocatable_kernel and kernel_alignment fields.
Protocol 2.06	(Kernel 2.6.22) Added a field that contains the size of the boot command line.
Protocol 2.07	(Kernel 2.6.24) Added paravirtualised boot protocol. Introduced hardware_subarch and hardware_subarch_data and KEEP_SEGMENTS flag in load_flags.
Protocol 2.08	(Kernel 2.6.26) Added crc32 checksum and ELF format payload. Introduced payload_offset and payload_length fields to aid in locating the payload.
Protocol 2.09	(Kernel 2.6.26) Added a field of 64-bit physical pointer to single linked list of struct setup_data.
Protocol 2.10	(Kernel 2.6.31) Added a protocol for relaxed alignment beyond the kernel_alignment added, new init_size and pref_address fields. Added extended boot loader IDs.
Protocol 2.11	(Kernel 3.6) Added a field for offset of EFI handover protocol entry point.
Protocol 2.12	(Kernel 3.8) Added the xloadflags field and extension fields to struct boot_params for loading bzImage and ramdisk above 4G in 64bit.
Protocol 2.13	(Kernel 3.14) Support 32- and 64-bit flags being set in xloadflags to support booting a 64-bit kernel from 32-bit EFI
Protocol 2.14	BURNT BY INCORRECT COMMIT ae7e1238e68f2a472a125673ab506d49158c1889 (x86/boot: Add ACPI RSDP address to setup_header) DO NOT USE!!! ASSUME SAME AS 2.13.
Protocol 2.15	(Kernel 5.5) Added the kernel_info and kernel_info.setup_type_max.

Note: The protocol version number should be changed only if the setup header is changed. There is no need to update the version number if boot_params or kernel_info are changed. Additionally, it is recommended to use xloadflags (in this case the protocol version number should not be updated either) or kernel_info to communicate supported Linux kernel features to the boot loader. Due to very limited space available in the original setup header every update to it should be considered with great care. Starting from the protocol 2.15 the primary way to communicate things to the boot loader is the kernel_info.

16.1.1 Memory Layout

The traditional memory map for the kernel loader, used for Image or zImage kernels, typically looks like:

```
        |                        |
0A0000  +------------------------+
        |  Reserved for BIOS     |      Do not use.  Reserved for BIOS EBDA.
09A000  +------------------------+
        |  Command line          |
        |  Stack/heap            |      For use by the kernel real-mode code.
098000  +------------------------+
        |  Kernel setup          |      The kernel real-mode code.
090200  +------------------------+
        |  Kernel boot sector    |      The kernel legacy boot sector.
090000  +------------------------+
        |  Protected-mode kernel |      The bulk of the kernel image.
010000  +------------------------+
        |  Boot loader           |      <- Boot sector entry point 0000:7C00
001000  +------------------------+
        |  Reserved for MBR/BIOS |
000800  +------------------------+
        |  Typically used by MBR |
000600  +------------------------+
        |  BIOS use only         |
000000  +------------------------+
```

When using bzImage, the protected-mode kernel was relocated to 0x100000 ("high memory"), and the kernel real-mode block (boot sector, setup, and stack/heap) was made relocatable to any address between 0x10000 and end of low memory. Unfortunately, in protocols 2.00 and 2.01 the 0x90000+ memory range is still used internally by the kernel; the 2.02 protocol resolves that problem.

It is desirable to keep the "memory ceiling" -- the highest point in low memory touched by the boot loader -- as low as possible, since some newer BIOSes have begun to allocate some rather large amounts of memory, called the Extended BIOS Data Area, near the top of low memory. The boot loader should use the "INT 12h" BIOS call to verify how much low memory is available.

Unfortunately, if INT 12h reports that the amount of memory is too low, there is usually nothing the boot loader can do but to report an error to the user. The boot loader should therefore be designed to take up as little space in low memory as it reasonably can. For zImage or old bzImage kernels, which need data written into the 0x90000 segment, the boot loader should make sure not to use memory above the 0x9A000 point; too many BIOSes will break above that point.

For a modern bzImage kernel with boot protocol version >= 2.02, a memory layout like the following is suggested:

```
              ~                        ~
              |  Protected-mode kernel |
      100000  +------------------------+
              |  I/O memory hole       |
      0A0000  +------------------------+
              |  Reserved for BIOS     |      Leave as much as possible unused
              ~                        ~
              |  Command line          |      (Can also be below the X+10000␣
↪mark)
     X+10000 +------------------------+
              |  Stack/heap            |      For use by the kernel real-mode␣
↪code.
     X+08000 +------------------------+
              |  Kernel setup          |      The kernel real-mode code.
              |  Kernel boot sector    |      The kernel legacy boot sector.
     X        +------------------------+
              |  Boot loader           |      <- Boot sector entry point␣
↪0000:7C00
     001000   +------------------------+
              |  Reserved for MBR/BIOS |
     000800   +------------------------+
              |  Typically used by MBR |
     000600   +------------------------+
              |  BIOS use only         |
     000000   +------------------------+

... where the address X is as low as the design of the boot loader permits.
```

16.1.2 The Real-Mode Kernel Header

In the following text, and anywhere in the kernel boot sequence, "a sector" refers to 512 bytes. It is independent of the actual sector size of the underlying medium.

The first step in loading a Linux kernel should be to load the real-mode code (boot sector and setup code) and then examine the following header at offset 0x01f1. The real-mode code can total up to 32K, although the boot loader may choose to load only the first two sectors (1K) and then examine the bootup sector size.

The header looks like:

Offset/Size	Proto	Name	Meaning
01F1/1	ALL(1)	setup_sects	The size of the setup in sectors
01F2/2	ALL	root_flags	If set, the root is mounted readonly
01F4/4	2.04+(2)	syssize	The size of the 32-bit code in 16-byte paras
01F8/2	ALL	ram_size	DO NOT USE - for bootsect.S use only
01FA/2	ALL	vid_mode	Video mode control
01FC/2	ALL	root_dev	Default root device number
01FE/2	ALL	boot_flag	0xAA55 magic number
0200/2	2.00+	jump	Jump instruction
0202/4	2.00+	header	Magic signature "HdrS"
0206/2	2.00+	version	Boot protocol version supported
0208/4	2.00+	realmode_swtch	Boot loader hook (see below)
020C/2	2.00+	start_sys_seg	The load-low segment (0x1000) (obsolete)
020E/2	2.00+	kernel_version	Pointer to kernel version string

continues on nex

Table 1 – continued from previous page

Offset/Size	Proto	Name	Meaning
0210/1	2.00+	type_of_loader	Boot loader identifier
0211/1	2.00+	loadflags	Boot protocol option flags
0212/2	2.00+	setup_move_size	Move to high memory size (used with hooks)
0214/4	2.00+	code32_start	Boot loader hook (see below)
0218/4	2.00+	ramdisk_image	initrd load address (set by boot loader)
021C/4	2.00+	ramdisk_size	initrd size (set by boot loader)
0220/4	2.00+	bootsect_kludge	DO NOT USE - for bootsect.S use only
0224/2	2.01+	heap_end_ptr	Free memory after setup end
0226/1	2.02+(3)	ext_loader_ver	Extended boot loader version
0227/1	2.02+(3)	ext_loader_type	Extended boot loader ID
0228/4	2.02+	cmd_line_ptr	32-bit pointer to the kernel command line
022C/4	2.03+	initrd_addr_max	Highest legal initrd address
0230/4	2.05+	kernel_alignment	Physical addr alignment required for kernel
0234/1	2.05+	relocatable_kernel	Whether kernel is relocatable or not
0235/1	2.10+	min_alignment	Minimum alignment, as a power of two
0236/2	2.12+	xloadflags	Boot protocol option flags
0238/4	2.06+	cmdline_size	Maximum size of the kernel command line
023C/4	2.07+	hardware_subarch	Hardware subarchitecture
0240/8	2.07+	hardware_subarch_data	Subarchitecture-specific data
0248/4	2.08+	payload_offset	Offset of kernel payload
024C/4	2.08+	payload_length	Length of kernel payload
0250/8	2.09+	setup_data	64-bit physical pointer to linked list of struct setu
0258/8	2.10+	pref_address	Preferred loading address
0260/4	2.10+	init_size	Linear memory required during initialization
0264/4	2.11+	handover_offset	Offset of handover entry point
0268/4	2.15+	kernel_info_offset	Offset of the kernel_info

Note:

(1) For backwards compatibility, if the setup_sects field contains 0, the real value is 4.

(2) For boot protocol prior to 2.04, the upper two bytes of the syssize field are unusable, which means the size of a bzImage kernel cannot be determined.

(3) Ignored, but safe to set, for boot protocols 2.02-2.09.

If the “HdrS” (0x53726448) magic number is not found at offset 0x202, the boot protocol version is “old”. Loading an old kernel, the following parameters should be assumed:

```
Image type = zImage
initrd not supported
Real-mode kernel must be located at 0x90000.
```

Otherwise, the “version” field contains the protocol version, e.g. protocol version 2.01 will contain 0x0201 in this field. When setting fields in the header, you must make sure only to set fields supported by the protocol version in use.

16.1.3 Details of Header Fields

For each field, some are information from the kernel to the bootloader ("read"), some are expected to be filled out by the bootloader ("write"), and some are expected to be read and modified by the bootloader ("modify").

All general purpose boot loaders should write the fields marked (obligatory). Boot loaders who want to load the kernel at a nonstandard address should fill in the fields marked (reloc); other boot loaders can ignore those fields.

The byte order of all fields is littleendian (this is x86, after all.)

Field name:	setup_sects
Type:	read
Offset/size:	0x1f1/1
Protocol:	ALL

The size of the setup code in 512-byte sectors. If this field is 0, the real value is 4. The real-mode code consists of the boot sector (always one 512-byte sector) plus the setup code.

Field name:	root_flags
Type:	modify (optional)
Offset/size:	0x1f2/2
Protocol:	ALL

If this field is nonzero, the root defaults to readonly. The use of this field is deprecated; use the "ro" or "rw" options on the command line instead.

Field name:	syssize
Type:	read
Offset/size:	0x1f4/4 (protocol 2.04+) 0x1f4/2 (protocol ALL)
Protocol:	2.04+

The size of the protected-mode code in units of 16-byte paragraphs. For protocol versions older than 2.04 this field is only two bytes wide, and therefore cannot be trusted for the size of a kernel if the LOAD_HIGH flag is set.

Field name:	ram_size
Type:	kernel internal
Offset/size:	0x1f8/2
Protocol:	ALL

This field is obsolete.

Field name:	vid_mode
Type:	modify (obligatory)
Offset/size:	0x1fa/2

Please see the section on SPECIAL COMMAND LINE OPTIONS.

Field name:	root_dev
Type:	modify (optional)
Offset/size:	0x1fc/2
Protocol:	ALL

The default root device device number. The use of this field is deprecated, use the "root=" option on the command line instead.

Field name:	boot_flag
Type:	read
Offset/size:	0x1fe/2
Protocol:	ALL

Contains 0xAA55. This is the closest thing old Linux kernels have to a magic number.

Field name:	jump
Type:	read
Offset/size:	0x200/2
Protocol:	2.00+

Contains an x86 jump instruction, 0xEB followed by a signed offset relative to byte 0x202. This can be used to determine the size of the header.

Field name:	header
Type:	read
Offset/size:	0x202/4
Protocol:	2.00+

Contains the magic number "HdrS" (0x53726448).

Field name:	version
Type:	read
Offset/size:	0x206/2
Protocol:	2.00+

Contains the boot protocol version, in (major << 8)+minor format, e.g. 0x0204 for version 2.04, and 0x0a11 for a hypothetical version 10.17.

Field name:	realmode_swtch
Type:	modify (optional)
Offset/size:	0x208/4
Protocol:	2.00+

Boot loader hook (see ADVANCED BOOT LOADER HOOKS below.)

Field name:	start_sys_seg
Type:	read
Offset/size:	0x20c/2
Protocol:	2.00+

The load low segment (0x1000). Obsolete.

Field name:	kernel_version
Type:	read
Offset/size:	0x20e/2
Protocol:	2.00+

If set to a nonzero value, contains a pointer to a NUL-terminated human-readable kernel version number string, less 0x200. This can be used to display the kernel version to the user. This value should be less than (0x200*setup_sects).

For example, if this value is set to 0x1c00, the kernel version number string can be found at offset 0x1e00 in the kernel file. This is a valid value if and only if the "setup_sects" field contains the value 15 or higher, as:

```
0x1c00  < 15*0x200 (= 0x1e00) but
0x1c00 >= 14*0x200 (= 0x1c00)

0x1c00 >> 9 = 14, So the minimum value for setup_secs is 15.
```

Field name:	type_of_loader
Type:	write (obligatory)
Offset/size:	0x210/1
Protocol:	2.00+

If your boot loader has an assigned id (see table below), enter 0xTV here, where T is an identifier for the boot loader and V is a version number. Otherwise, enter 0xFF here.

For boot loader IDs above T = 0xD, write T = 0xE to this field and write the extended ID minus 0x10 to the ext_loader_type field. Similarly, the ext_loader_ver field can be used to provide more than four bits for the bootloader version.

For example, for T = 0x15, V = 0x234, write:

```
type_of_loader  <- 0xE4
ext_loader_type <- 0x05
ext_loader_ver  <- 0x23
```

Assigned boot loader ids (hexadecimal):

0	LILO (0x00 reserved for pre-2.00 bootloader)
1	Loadlin
2	bootsect-loader (0x20, all other values reserved)
3	Syslinux
4	Etherboot/gPXE/iPXE
5	ELILO
7	GRUB
8	U-Boot
9	Xen
A	Gujin
B	Qemu
C	Arcturus Networks uCbootloader
D	kexec-tools
E	Extended (see ext_loader_type)
F	Special (0xFF = undefined)
10	Reserved
11	Minimal Linux Bootloader <http://sebastian-plotz.blogspot.de>
12	OVMF UEFI virtualization stack
13	barebox

Please contact <hpa@zytor.com> if you need a bootloader ID value assigned.

Field name:	loadflags
Type:	modify (obligatory)
Offset/size:	0x211/1
Protocol:	2.00+

This field is a bitmask.

Bit 0 (read): LOADED_HIGH

- If 0, the protected-mode code is loaded at 0x10000.
- If 1, the protected-mode code is loaded at 0x100000.

Bit 1 (kernel internal): KASLR_FLAG

- Used internally by the compressed kernel to communicate KASLR status to kernel proper.
  - If 1, KASLR enabled.
  - If 0, KASLR disabled.

Bit 5 (write): QUIET_FLAG

- If 0, print early messages.
- If 1, suppress early messages.

  This requests to the kernel (decompressor and early kernel) to not write early messages that require accessing the display hardware directly.

Bit 6 (obsolete): KEEP_SEGMENTS

Protocol: 2.07+

- This flag is obsolete.

Bit 7 (write): CAN_USE_HEAP

Set this bit to 1 to indicate that the value entered in the heap_end_ptr is valid. If this field is clear, some setup code functionality will be disabled.

Field name:	setup_move_size
Type:	modify (obligatory)
Offset/size:	0x212/2
Protocol:	2.00-2.01

When using protocol 2.00 or 2.01, if the real mode kernel is not loaded at 0x90000, it gets moved there later in the loading sequence. Fill in this field if you want additional data (such as the kernel command line) moved in addition to the real-mode kernel itself.

The unit is bytes starting with the beginning of the boot sector.

This field is can be ignored when the protocol is 2.02 or higher, or if the real-mode code is loaded at 0x90000.

Field name:	code32_start
Type:	modify (optional, reloc)
Offset/size:	0x214/4
Protocol:	2.00+

The address to jump to in protected mode. This defaults to the load address of the kernel, and can be used by the boot loader to determine the proper load address.

This field can be modified for two purposes:

1. as a boot loader hook (see Advanced Boot Loader Hooks below.)
2. if a bootloader which does not install a hook loads a relocatable kernel at a non-standard address it will have to modify this field to point to the load address.

Field name:	ramdisk_image
Type:	write (obligatory)
Offset/size:	0x218/4
Protocol:	2.00+

The 32-bit linear address of the initial ramdisk or ramfs. Leave at zero if there is no initial ramdisk/ramfs.

Field name:	ramdisk_size
Type:	write (obligatory)
Offset/size:	0x21c/4
Protocol:	2.00+

Size of the initial ramdisk or ramfs. Leave at zero if there is no initial ramdisk/ramfs.

Field name:	bootsect_kludge
Type:	kernel internal
Offset/size:	0x220/4
Protocol:	2.00+

This field is obsolete.

Field name:	heap_end_ptr
Type:	write (obligatory)
Offset/size:	0x224/2
Protocol:	2.01+

Set this field to the offset (from the beginning of the real-mode code) of the end of the setup stack/heap, minus 0x0200.

Field name:	ext_loader_ver
Type:	write (optional)
Offset/size:	0x226/1
Protocol:	2.02+

This field is used as an extension of the version number in the type_of_loader field. The total version number is considered to be (type_of_loader & 0x0f) + (ext_loader_ver << 4).

The use of this field is boot loader specific. If not written, it is zero.

Kernels prior to 2.6.31 did not recognize this field, but it is safe to write for protocol version 2.02 or higher.

Field name:	ext_loader_type
Type:	write (obligatory if (type_of_loader & 0xf0) == 0xe0)
Offset/size:	0x227/1
Protocol:	2.02+

This field is used as an extension of the type number in type_of_loader field. If the type in type_of_loader is 0xE, then the actual type is (ext_loader_type + 0x10).

This field is ignored if the type in type_of_loader is not 0xE.

Kernels prior to 2.6.31 did not recognize this field, but it is safe to write for protocol version 2.02 or higher.

Field name:	cmd_line_ptr
Type:	write (obligatory)
Offset/size:	0x228/4
Protocol:	2.02+

Set this field to the linear address of the kernel command line. The kernel command line can be located anywhere between the end of the setup heap and 0xA0000; it does not have to be located in the same 64K segment as the real-mode code itself.

Fill in this field even if your boot loader does not support a command line, in which case you can point this to an empty string (or better yet, to the string "auto".) If this field is left at zero, the kernel will assume that your boot loader does not support the 2.02+ protocol.

Field name:	initrd_addr_max
Type:	read
Offset/size:	0x22c/4
Protocol:	2.03+

The maximum address that may be occupied by the initial ramdisk/ramfs contents. For boot protocols 2.02 or earlier, this field is not present, and the maximum address is 0x37FFFFFF. (This address is defined as the address of the highest safe byte, so if your ramdisk is exactly 131072 bytes long and this field is 0x37FFFFFF, you can start your ramdisk at 0x37FE0000.)

Field name:	kernel_alignment
Type:	read/modify (reloc)
Offset/size:	0x230/4
Protocol:	2.05+ (read), 2.10+ (modify)

Alignment unit required by the kernel (if relocatable_kernel is true.) A relocatable kernel that is loaded at an alignment incompatible with the value in this field will be realigned during kernel initialization.

Starting with protocol version 2.10, this reflects the kernel alignment preferred for optimal performance; it is possible for the loader to modify this field to permit a lesser alignment. See the min_alignment and pref_address field below.

Field name:	relocatable_kernel
Type:	read (reloc)
Offset/size:	0x234/1
Protocol:	2.05+

If this field is nonzero, the protected-mode part of the kernel can be loaded at any address that satisfies the kernel_alignment field. After loading, the boot loader must set the code32_start field to point to the loaded code, or to a boot loader hook.

Field name:	min_alignment
Type:	read (reloc)
Offset/size:	0x235/1
Protocol:	2.10+

This field, if nonzero, indicates as a power of two the minimum alignment required, as opposed to preferred, by the kernel to boot. If a boot loader makes use of this field, it should update the kernel_alignment field with the alignment unit desired; typically:

```
kernel_alignment = 1 << min_alignment
```

There may be a considerable performance cost with an excessively misaligned kernel. Therefore, a loader should typically try each power-of-two alignment from kernel_alignment down to this alignment.

Field name:	xloadflags
Type:	read
Offset/size:	0x236/2
Protocol:	2.12+

This field is a bitmask.

Bit 0 (read): XLF_KERNEL_64

- If 1, this kernel has the legacy 64-bit entry point at 0x200.

Bit 1 (read): XLF_CAN_BE_LOADED_ABOVE_4G

- If 1, kernel/boot_params/cmdline/ramdisk can be above 4G.

Bit 2 (read): XLF_EFI_HANDOVER_32

- If 1, the kernel supports the 32-bit EFI handoff entry point given at handover_offset.

Bit 3 (read): XLF_EFI_HANDOVER_64

- If 1, the kernel supports the 64-bit EFI handoff entry point given at handover_offset + 0x200.

Bit 4 (read): XLF_EFI_KEXEC

- If 1, the kernel supports kexec EFI boot with EFI runtime support.

Field name:	cmdline_size
Type:	read
Offset/size:	0x238/4
Protocol:	2.06+

The maximum size of the command line without the terminating zero. This means that the command line can contain at most cmdline_size characters. With protocol version 2.05 and earlier, the maximum size was 255.

Field name:	hardware_subarch
Type:	write (optional, defaults to x86/PC)
Offset/size:	0x23c/4
Protocol:	2.07+

In a paravirtualized environment the hardware low level architectural pieces such as interrupt handling, page table handling, and accessing process control registers needs to be done differently.

This field allows the bootloader to inform the kernel we are in one one of those environments.

0x00000000	The default x86/PC environment
0x00000001	lguest
0x00000002	Xen
0x00000003	Moorestown MID
0x00000004	CE4100 TV Platform

Field name:	hardware_subarch_data
Type:	write (subarch-dependent)
Offset/size:	0x240/8
Protocol:	2.07+

A pointer to data that is specific to hardware subarch This field is currently unused for the default x86/PC environment, do not modify.

Field name:	payload_offset
Type:	read
Offset/size:	0x248/4
Protocol:	2.08+

If non-zero then this field contains the offset from the beginning of the protected-mode code to the payload.

The payload may be compressed. The format of both the compressed and uncompressed data should be determined using the standard magic numbers. The currently supported compression formats are gzip (magic numbers 1F 8B or 1F 9E), bzip2 (magic number 42 5A), LZMA (magic number 5D 00), XZ (magic number FD 37), LZ4 (magic number 02 21) and ZSTD (magic number 28 B5). The uncompressed payload is currently always ELF (magic number 7F 45 4C 46).

Field name:	payload_length
Type:	read
Offset/size:	0x24c/4
Protocol:	2.08+

The length of the payload.

Field name:	setup_data
Type:	write (special)
Offset/size:	0x250/8
Protocol:	2.09+

The 64-bit physical pointer to NULL terminated single linked list of struct setup_data. This is used to define a more extensible boot parameters passing mechanism. The definition of struct setup_data is as follow:

```
struct setup_data {
        u64 next;
        u32 type;
        u32 len;
        u8  data[0];
};
```

Where, the next is a 64-bit physical pointer to the next node of linked list, the next field of the last node is 0; the type is used to identify the contents of data; the len is the length of data field; the data holds the real payload.

This list may be modified at a number of points during the bootup process. Therefore, when modifying this list one should always make sure to consider the case where the linked list already contains entries.

The setup_data is a bit awkward to use for extremely large data objects, both because the setup_data header has to be adjacent to the data object and because it has a 32-bit length field. However, it is important that intermediate stages of the boot process have a way to identify which chunks of memory are occupied by kernel data.

Thus setup_indirect struct and SETUP_INDIRECT type were introduced in protocol 2.15:

```
struct setup_indirect {
  __u32 type;
  __u32 reserved;  /* Reserved, must be set to zero. */
  __u64 len;
  __u64 addr;
};
```

The type member is a SETUP_INDIRECT | SETUP_* type. However, it cannot be SETUP_INDIRECT itself since making the setup_indirect a tree structure could require a lot of stack space in something that needs to parse it and stack space can be limited in boot contexts.

Let's give an example how to point to SETUP_E820_EXT data using setup_indirect. In this case setup_data and setup_indirect will look like this:

```
struct setup_data {
  __u64 next = 0 or <addr_of_next_setup_data_struct>;
  __u32 type = SETUP_INDIRECT;
  __u32 len = sizeof(setup_indirect);
  __u8 data[sizeof(setup_indirect)] = struct setup_indirect {
    __u32 type = SETUP_INDIRECT | SETUP_E820_EXT;
    __u32 reserved = 0;
    __u64 len = <len_of_SETUP_E820_EXT_data>;
    __u64 addr = <addr_of_SETUP_E820_EXT_data>;
  }
}
```

Note: SETUP_INDIRECT | SETUP_NONE objects cannot be properly distinguished from SETUP_INDIRECT itself. So, this kind of objects cannot be provided by the bootloaders.

Field name:	pref_address
Type:	read (reloc)
Offset/size:	0x258/8
Protocol:	2.10+

This field, if nonzero, represents a preferred load address for the kernel. A relocating bootloader should attempt to load at this address if possible.

A non-relocatable kernel will unconditionally move itself and to run at this address.

Field name:	init_size
Type:	read
Offset/size:	0x260/4

This field indicates the amount of linear contiguous memory starting at the kernel runtime start address that the kernel needs before it is capable of examining its memory map. This is not the same thing as the total amount of memory the kernel needs to boot, but it can be used by a relocating boot loader to help select a safe load address for the kernel.

The kernel runtime start address is determined by the following algorithm:

```
if (relocatable_kernel)
runtime_start = align_up(load_address, kernel_alignment)
else
runtime_start = pref_address
```

Field name:	handover_offset
Type:	read
Offset/size:	0x264/4

This field is the offset from the beginning of the kernel image to the EFI handover protocol entry point. Boot loaders using the EFI handover protocol to boot the kernel should jump to this offset.

See EFI HANDOVER PROTOCOL below for more details.

Field name:	kernel_info_offset
Type:	read
Offset/size:	0x268/4
Protocol:	2.15+

This field is the offset from the beginning of the kernel image to the kernel_info. The kernel_info structure is embedded in the Linux image in the uncompressed protected mode region.

16.1.4 The kernel_info

The relationships between the headers are analogous to the various data sections:

> setup_header = .data boot_params/setup_data = .bss

What is missing from the above list? That's right:

> kernel_info = .rodata

We have been (ab)using .data for things that could go into .rodata or .bss for a long time, for lack of alternatives and -- especially early on -- inertia. Also, the BIOS stub is responsible for creating boot_params, so it isn't available to a BIOS-based loader (setup_data is, though).

setup_header is permanently limited to 144 bytes due to the reach of the 2-byte jump field, which doubles as a length field for the structure, combined with the size of the "hole" in struct boot_params that a protected-mode loader or the BIOS stub has to copy it into. It is currently 119 bytes long, which leaves us with 25 very precious bytes. This isn't something that can be fixed without revising the boot protocol entirely, breaking backwards compatibility.

boot_params proper is limited to 4096 bytes, but can be arbitrarily extended by adding setup_data entries. It cannot be used to communicate properties of the kernel image, because it is .bss and has no image-provided content.

kernel_info solves this by providing an extensible place for information about the kernel image. It is readonly, because the kernel cannot rely on a bootloader copying its contents anywhere, but that is OK; if it becomes necessary it can still contain data items that an enabled bootloader would be expected to copy into a setup_data chunk.

All kernel_info data should be part of this structure. Fixed size data have to be put before kernel_info_var_len_data label. Variable size data have to be put after kernel_info_var_len_data label. Each chunk of variable size data has to be prefixed with header/magic and its size, e.g.:

```
kernel_info:
        .ascii  "LToP"          /* Header, Linux top (structure). */
        .long   kernel_info_var_len_data - kernel_info
        .long   kernel_info_end - kernel_info
        .long   0x01234567      /* Some fixed size data for the bootloaders. */
kernel_info_var_len_data:
example_struct:                 /* Some variable size data for the bootloaders.
↪ */
        .ascii  "0123"          /* Header/Magic. */
        .long   example_struct_end - example_struct
        .ascii  "Struct"
        .long   0x89012345
example_struct_end:
example_strings:                /* Some variable size data for the bootloaders.
↪ */
        .ascii  "ABCD"          /* Header/Magic. */
        .long   example_strings_end - example_strings
        .asciz  "String_0"
        .asciz  "String_1"
example_strings_end:
kernel_info_end:
```

This way the kernel_info is self-contained blob.

Note: Each variable size data header/magic can be any 4-character string, without 0 at the end of the string, which does not collide with existing variable length data headers/magics.

16.1.5 Details of the kernel_info Fields

Field name:	header
Offset/size:	0x0000/4

Contains the magic number "LToP" (0x506f544c).

Field name:	size
Offset/size:	0x0004/4

This field contains the size of the kernel_info including kernel_info.header. It does not count kernel_info.kernel_info_var_len_data size. This field should be used by the bootloaders to detect supported fixed size fields in the kernel_info and beginning of kernel_info.kernel_info_var_len_data.

Field name:	size_total
Offset/size:	0x0008/4

This field contains the size of the kernel_info including kernel_info.header and kernel_info.kernel_info_var_len_data.

Field name:	setup_type_max
Offset/size:	0x000c/4

This field contains maximal allowed type for setup_data and setup_indirect structs.

16.1.6 The Image Checksum

From boot protocol version 2.08 onwards the CRC-32 is calculated over the entire file using the characteristic polynomial 0x04C11DB7 and an initial remainder of 0xffffffff. The checksum is appended to the file; therefore the CRC of the file up to the limit specified in the syssize field of the header is always 0.

16.1.7 The Kernel Command Line

The kernel command line has become an important way for the boot loader to communicate with the kernel. Some of its options are also relevant to the boot loader itself, see "special command line options" below.

The kernel command line is a null-terminated string. The maximum length can be retrieved from the field cmdline_size. Before protocol version 2.06, the maximum was 255 characters. A string that is too long will be automatically truncated by the kernel.

If the boot protocol version is 2.02 or later, the address of the kernel command line is given by the header field cmd_line_ptr (see above.) This address can be anywhere between the end of the setup heap and 0xA0000.

If the protocol version is *not* 2.02 or higher, the kernel command line is entered using the following protocol:

- At offset 0x0020 (word), "cmd_line_magic", enter the magic number 0xA33F.
- At offset 0x0022 (word), "cmd_line_offset", enter the offset of the kernel command line (relative to the start of the real-mode kernel).
- The kernel command line *must* be within the memory region covered by setup_move_size, so you may need to adjust this field.

16.1.8 Memory Layout of The Real-Mode Code

The real-mode code requires a stack/heap to be set up, as well as memory allocated for the kernel command line. This needs to be done in the real-mode accessible memory in bottom megabyte.

It should be noted that modern machines often have a sizable Extended BIOS Data Area (EBDA). As a result, it is advisable to use as little of the low megabyte as possible.

Unfortunately, under the following circumstances the 0x90000 memory segment has to be used:

- When loading a zImage kernel ((loadflags & 0x01) == 0).
- When loading a 2.01 or earlier boot protocol kernel.

Note: For the 2.00 and 2.01 boot protocols, the real-mode code can be loaded at another address, but it is internally relocated to 0x90000. For the "old" protocol, the real-mode code must be loaded at 0x90000.

When loading at 0x90000, avoid using memory above 0x9a000.

For boot protocol 2.02 or higher, the command line does not have to be located in the same 64K segment as the real-mode setup code; it is thus permitted to give the stack/heap the full 64K segment and locate the command line above it.

The kernel command line should not be located below the real-mode code, nor should it be located in high memory.

16.1.9 Sample Boot Configuration

As a sample configuration, assume the following layout of the real mode segment.

When loading below 0x90000, use the entire segment:

0x0000-0x7fff	Real mode kernel
0x8000-0xdfff	Stack and heap
0xe000-0xffff	Kernel command line

When loading at 0x90000 OR the protocol version is 2.01 or earlier:

0x0000-0x7fff	Real mode kernel
0x8000-0x97ff	Stack and heap
0x9800-0x9fff	Kernel command line

Such a boot loader should enter the following fields in the header:

```
unsigned long base_ptr; /* base address for real-mode segment */

if ( setup_sects == 0 ) {
        setup_sects = 4;
}

if ( protocol >= 0x0200 ) {
        type_of_loader = <type code>;
        if ( loading_initrd ) {
                ramdisk_image = <initrd_address>;
                ramdisk_size = <initrd_size>;
        }

        if ( protocol >= 0x0202 && loadflags & 0x01 )
                heap_end = 0xe000;
        else
                heap_end = 0x9800;

        if ( protocol >= 0x0201 ) {
                heap_end_ptr = heap_end - 0x200;
                loadflags |= 0x80; /* CAN_USE_HEAP */
        }

        if ( protocol >= 0x0202 ) {
                cmd_line_ptr = base_ptr + heap_end;
                strcpy(cmd_line_ptr, cmdline);
        } else {
                cmd_line_magic  = 0xA33F;
                cmd_line_offset = heap_end;
                setup_move_size = heap_end + strlen(cmdline)+1;
                strcpy(base_ptr+cmd_line_offset, cmdline);
        }
} else {
        /* Very old kernel */

        heap_end = 0x9800;

        cmd_line_magic  = 0xA33F;
        cmd_line_offset = heap_end;

        /* A very old kernel MUST have its real-mode code
           loaded at 0x90000 */

        if ( base_ptr != 0x90000 ) {
                /* Copy the real-mode kernel */
                memcpy(0x90000, base_ptr, (setup_sects+1)*512);
                base_ptr = 0x90000;              /* Relocated */
        }

        strcpy(0x90000+cmd_line_offset, cmdline);

        /* It is recommended to clear memory up to the 32K mark */
        memset(0x90000 + (setup_sects+1)*512, 0,
               (64-(setup_sects+1))*512);
}
```

16.1.10 Loading The Rest of The Kernel

The 32-bit (non-real-mode) kernel starts at offset (setup_sects+1)*512 in the kernel file (again, if setup_sects == 0 the real value is 4.) It should be loaded at address 0x10000 for Image/zImage kernels and 0x100000 for bzImage kernels.

The kernel is a bzImage kernel if the protocol >= 2.00 and the 0x01 bit (LOAD_HIGH) in the loadflags field is set:

```
is_bzImage = (protocol >= 0x0200) && (loadflags & 0x01);
load_address = is_bzImage ? 0x100000 : 0x10000;
```

Note that Image/zImage kernels can be up to 512K in size, and thus use the entire 0x10000-0x90000 range of memory. This means it is pretty much a requirement for these kernels to load the real-mode part at 0x90000. bzImage kernels allow much more flexibility.

16.1.11 Special Command Line Options

If the command line provided by the boot loader is entered by the user, the user may expect the following command line options to work. They should normally not be deleted from the kernel command line even though not all of them are actually meaningful to the kernel. Boot loader authors who need additional command line options for the boot loader itself should get them registered in Documentation/admin-guide/kernel-parameters.rst to make sure they will not conflict with actual kernel options now or in the future.

vga=<mode>
: <mode> here is either an integer (in C notation, either decimal, octal, or hexadecimal) or one of the strings "normal" (meaning 0xFFFF), "ext" (meaning 0xFFFE) or "ask" (meaning 0xFFFD). This value should be entered into the vid_mode field, as it is used by the kernel before the command line is parsed.

mem=<size>
: <size> is an integer in C notation optionally followed by (case insensitive) K, M, G, T, P or E (meaning << 10, << 20, << 30, << 40, << 50 or << 60). This specifies the end of memory to the kernel. This affects the possible placement of an initrd, since an initrd should be placed near end of memory. Note that this is an option to *both* the kernel and the bootloader!

initrd=<file>
: An initrd should be loaded. The meaning of <file> is obviously bootloader-dependent, and some boot loaders (e.g. LILO) do not have such a command.

In addition, some boot loaders add the following options to the user-specified command line:

BOOT_IMAGE=<file>
: The boot image which was loaded. Again, the meaning of <file> is obviously bootloader-dependent.

auto
: The kernel was booted without explicit user intervention.

If these options are added by the boot loader, it is highly recommended that they are located *first*, before the user-specified or configuration-specified command line. Otherwise, "init=/bin/sh" gets confused by the "auto" option.

16.1.12 Running the Kernel

The kernel is started by jumping to the kernel entry point, which is located at *segment* offset 0x20 from the start of the real mode kernel. This means that if you loaded your real-mode kernel code at 0x90000, the kernel entry point is 9020:0000.

At entry, ds = es = ss should point to the start of the real-mode kernel code (0x9000 if the code is loaded at 0x90000), sp should be set up properly, normally pointing to the top of the heap, and interrupts should be disabled. Furthermore, to guard against bugs in the kernel, it is recommended that the boot loader sets fs = gs = ds = es = ss.

In our example from above, we would do:

```
/* Note: in the case of the "old" kernel protocol, base_ptr must
   be == 0x90000 at this point; see the previous sample code */

seg = base_ptr >> 4;

cli();  /* Enter with interrupts disabled! */

/* Set up the real-mode kernel stack */
_SS = seg;
_SP = heap_end;

```

```
_DS = _ES = _FS = _GS = seg;
jmp_far(seg+0x20, 0);   /* Run the kernel */
```

If your boot sector accesses a floppy drive, it is recommended to switch off the floppy motor before running the kernel, since the kernel boot leaves interrupts off and thus the motor will not be switched off, especially if the loaded kernel has the floppy driver as a demand-loaded module!

16.1.13 Advanced Boot Loader Hooks

If the boot loader runs in a particularly hostile environment (such as LOADLIN, which runs under DOS) it may be impossible to follow the standard memory location requirements. Such a boot loader may use the following hooks that, if set, are invoked by the kernel at the appropriate time. The use of these hooks should probably be considered an absolutely last resort!

IMPORTANT: All the hooks are required to preserve %esp, %ebp, %esi and %edi across invocation.

realmode_swtch:
A 16-bit real mode far subroutine invoked immediately before entering protected mode. The default routine disables NMI, so your routine should probably do so, too.

code32_start:
A 32-bit flat-mode routine *jumped* to immediately after the transition to protected mode, but before the kernel is uncompressed. No segments, except CS, are guaranteed to be set up (current kernels do, but older ones do not); you should set them up to BOOT_DS (0x18) yourself.

After completing your hook, you should jump to the address that was in this field before your boot loader overwrote it (relocated, if appropriate.)

16.1.14 32-bit Boot Protocol

For machine with some new BIOS other than legacy BIOS, such as EFI, LinuxBIOS, etc, and kexec, the 16-bit real mode setup code in kernel based on legacy BIOS can not be used, so a 32-bit boot protocol needs to be defined.

In 32-bit boot protocol, the first step in loading a Linux kernel should be to setup the boot parameters (struct boot_params, traditionally known as “zero page”). The memory for struct boot_params should be allocated and initialized to all zero. Then the setup header from offset 0x01f1 of kernel image on should be loaded into struct boot_params and examined. The end of setup header can be calculated as follow:

```
0x0202 + byte value at offset 0x0201
```

In addition to read/modify/write the setup header of the struct boot_params as that of 16-bit boot protocol, the boot loader should also fill the additional fields of the struct boot_params as described in chapter *Zero Page*.

After setting up the struct boot_params, the boot loader can load the 32/64-bit kernel in the same way as that of 16-bit boot protocol.

In 32-bit boot protocol, the kernel is started by jumping to the 32-bit kernel entry point, which is the start address of loaded 32/64-bit kernel.

At entry, the CPU must be in 32-bit protected mode with paging disabled; a GDT must be loaded with the descriptors for selectors __BOOT_CS(0x10) and __BOOT_DS(0x18); both descriptors must be 4G flat segment; __BOOT_CS must have execute/read permission, and __BOOT_DS must have read/write permission; CS must be __BOOT_CS and DS, ES, SS must be __BOOT_DS; interrupt must be disabled; %esi must hold the base address of the struct boot_params; %ebp, %edi and %ebx must be zero.

16.1.15 64-bit Boot Protocol

For machine with 64bit cpus and 64bit kernel, we could use 64bit bootloader and we need a 64-bit boot protocol.

In 64-bit boot protocol, the first step in loading a Linux kernel should be to setup the boot parameters (struct boot_params, traditionally known as "zero page"). The memory for struct boot_params could be allocated anywhere (even above 4G) and initialized to all zero. Then, the setup header at offset 0x01f1 of kernel image on should be loaded into struct boot_params and examined. The end of setup header can be calculated as follows:

```
0x0202 + byte value at offset 0x0201
```

In addition to read/modify/write the setup header of the struct boot_params as that of 16-bit boot protocol, the boot loader should also fill the additional fields of the struct boot_params as described in chapter *Zero Page*.

After setting up the struct boot_params, the boot loader can load 64-bit kernel in the same way as that of 16-bit boot protocol, but kernel could be loaded above 4G.

In 64-bit boot protocol, the kernel is started by jumping to the 64-bit kernel entry point, which is the start address of loaded 64-bit kernel plus 0x200.

At entry, the CPU must be in 64-bit mode with paging enabled. The range with setup_header.init_size from start address of loaded kernel and zero page and command line buffer get ident mapping; a GDT must be loaded with the descriptors for selectors __BOOT_CS(0x10) and __BOOT_DS(0x18); both descriptors must be 4G flat segment; __BOOT_CS must have execute/read permission, and __BOOT_DS must have read/write permission; CS must be __BOOT_CS and DS, ES, SS must be __BOOT_DS; interrupt must be disabled; %rsi must hold the base address of the struct boot_params.

16.1.16 EFI Handover Protocol (deprecated)

This protocol allows boot loaders to defer initialisation to the EFI boot stub. The boot loader is required to load the kernel/initrd(s) from the boot media and jump to the EFI handover protocol entry point which is hdr->handover_offset bytes from the beginning of startup_{32,64}.

The boot loader MUST respect the kernel's PE/COFF metadata when it comes to section alignment, the memory footprint of the executable image beyond the size of the file itself, and any other aspect of the PE/COFF header that may affect correct operation of the image as a PE/COFF binary in the execution context provided by the EFI firmware.

The function prototype for the handover entry point looks like this:

```
efi_stub_entry(void *handle, efi_system_table_t *table, struct boot_params *bp)
```

'handle' is the EFI image handle passed to the boot loader by the EFI firmware, 'table' is the EFI system table - these are the first two arguments of the "handoff state" as described in section 2.3 of the UEFI specification. 'bp' is the boot loader-allocated boot params.

The boot loader *must* fill out the following fields in bp:

```
- hdr.cmd_line_ptr
- hdr.ramdisk_image (if applicable)
- hdr.ramdisk_size  (if applicable)
```

All other fields should be zero.

NOTE: The EFI Handover Protocol is deprecated in favour of the ordinary PE/COFF
entry point, combined with the LINUX_EFI_INITRD_MEDIA_GUID based initrd loading protocol (refer to [0] for an example of the bootloader side of this), which removes the need for any knowledge on the part of the EFI bootloader regarding the internal representation of boot_params or any requirements/limitations regarding the placement of the command line and ramdisk in memory, or the placement of the kernel image itself.

[0] https://github.com/u-boot/u-boot/commit/ec80b4735a593961fe701cc3a5d717d4739b0fd0

16.2 DeviceTree Booting

There is one single 32bit entry point to the kernel at code32_start, the decompressor (the real mode entry point goes to the same 32bit entry point once it switched into protected mode). That entry point supports one calling convention which is documented in *The Linux/x86 Boot Protocol* The physical pointer to the device-tree block is passed via setup_data which requires at least boot protocol 2.09. The type filed is defined as

#define SETUP_DTB 2

This device-tree is used as an extension to the "boot page". As such it does not parse / consider data which is already covered by the boot page. This includes memory size, reserved ranges, command line arguments or initrd address. It simply holds information which can not be retrieved otherwise like interrupt routing or a list of devices behind an I2C bus.

16.3 x86 Feature Flags

16.3.1 Introduction

On x86, flags appearing in /proc/cpuinfo have an X86_FEATURE definition in arch/x86/include/asm/cpufeatures.h. If the kernel cares about a feature or KVM want to expose the feature to a KVM guest, it can and should have an X86_FEATURE_* defined. These flags represent hardware features as well as software features.

If users want to know if a feature is available on a given system, they try to find the flag in /proc/cpuinfo. If a given flag is present, it means that the kernel supports it and is currently making it available. If such flag represents a hardware feature, it also means that the hardware supports it.

If the expected flag does not appear in /proc/cpuinfo, things are murkier. Users need to find out the reason why the flag is missing and find the way how to enable it, which is not always easy. There are several factors that can explain missing flags: the expected feature failed to enable, the feature is missing in hardware, platform firmware did not enable it, the feature is disabled at build or run time, an old kernel is in use, or the kernel does not support the feature and thus has not enabled it. In general, /proc/cpuinfo shows features which the kernel supports. For a full list of CPUID flags which the CPU supports, use tools/arch/x86/kcpuid.

16.3.2 How are feature flags created?

a: Feature flags can be derived from the contents of CPUID leaves.

These feature definitions are organized mirroring the layout of CPUID leaves and grouped in words with offsets as mapped in enum cpuid_leafs in cpufeatures.h (see arch/x86/include/asm/cpufeatures.h for details). If a feature is defined with a X86_FEATURE_<name> definition in cpufeatures.h, and if it is detected at run time, the flags will be displayed accordingly in /proc/cpuinfo. For example, the flag "avx2" comes from X86_FEATURE_AVX2 in cpufeatures.h.

b: Flags can be from scattered CPUID-based features.

Hardware features enumerated in sparsely populated CPUID leaves get software-defined values. Still, CPUID needs to be queried to determine if a given feature is present. This is done in init_scattered_cpuid_features(). For instance, X86_FEATURE_CQM_LLC is defined as 11*32 + 0 and its presence is checked at runtime in the respective CPUID leaf [EAX=f, ECX=0] bit EDX[1].

The intent of scattering CPUID leaves is to not bloat struct cpuinfo_x86.x86_capability[] unnecessarily. For instance, the CPUID leaf [EAX=7, ECX=0] has 30 features and is dense, but the CPUID leaf [EAX=7, EAX=1] has only one feature and would waste 31 bits of space in the x86_capability[] array. Since there is a struct cpuinfo_x86 for each possible CPU, the wasted memory is not trivial.

c: Flags can be created synthetically under certain conditions for hardware features.

Examples of conditions include whether certain features are present in MSR_IA32_CORE_CAPS or specific CPU models are identified. If the needed conditions are met, the features are enabled by the set_cpu_cap or setup_force_cpu_cap macros. For example, if bit 5 is set in MSR_IA32_CORE_CAPS, the feature X86_FEATURE_SPLIT_LOCK_DETECT will be enabled and "split_lock_detect" will be displayed. The flag "ring3mwait" will be displayed only when running on INTEL_FAM6_XEON_PHI_[KNL|KNM] processors.

d: Flags can represent purely software features.

These flags do not represent hardware features. Instead, they represent a software feature implemented in the kernel. For example, Kernel Page Table Isolation is purely software feature and its feature flag X86_FEATURE_PTI is also defined in cpufeatures.h.

16.3.3 Naming of Flags

The script arch/x86/kernel/cpu/mkcapflags.sh processes the #define X86_FEATURE_<name> from cpufeatures.h and generates the x86_cap/bug_flags[] arrays in kernel/cpu/capflags.c. The names in the resulting x86_cap/bug_flags[] are used to populate /proc/cpuinfo. The naming of flags in the x86_cap/bug_flags[] are as follows:

a: The name of the flag is from the string in X86_FEATURE_<name> by default.

By default, the flag <name> in /proc/cpuinfo is extracted from the respective X86_FEATURE_<name> in cpufeatures.h. For example, the flag "avx2" is from X86_FEATURE_AVX2.

b: The naming can be overridden.

If the comment on the line for the #define X86_FEATURE_* starts with a double-quote character (""), the string inside the double-quote characters will be the name of the flags. For example, the flag "sse4_1" comes from the comment "sse4_1" following the X86_FEATURE_XMM4_1 definition.

There are situations in which overriding the displayed name of the flag is needed. For instance, /proc/cpuinfo is a userspace interface and must remain constant. If, for some reason, the naming of X86_FEATURE_<name> changes, one shall override the new naming with the name already used in /proc/cpuinfo.

c: The naming override can be "", which means it will not appear in /proc/cpuinfo.

The feature shall be omitted from /proc/cpuinfo if it does not make sense for the feature to be exposed to userspace. For example, X86_FEATURE_ALWAYS is defined in cpufeatures.h but that flag is an internal kernel feature used in the alternative runtime patching functionality. So, its name is overridden with "". Its flag will not appear in /proc/cpuinfo.

16.3.4 Flags are missing when one or more of these happen

a: The hardware does not enumerate support for it.

For example, when a new kernel is running on old hardware or the feature is not enabled by boot firmware. Even if the hardware is new, there might be a problem enabling the feature at run time, the flag will not be displayed.

b: The kernel does not know about the flag.

For example, when an old kernel is running on new hardware.

c: The kernel disabled support for it at compile-time.

For example, if 5-level-paging is not enabled when building (i.e., CONFIG_X86_5LEVEL is not selected) the flag "la57" will not show up[1]. Even though the feature will still be detected via CPUID, the kernel disables it by clearing via setup_clear_cpu_cap(X86_FEATURE_LA57).

d: The feature is disabled at boot-time.

A feature can be disabled either using a command-line parameter or because it failed to be enabled. The command-line parameter clearcpuid= can be used to disable features using the feature number as defined in /arch/x86/include/asm/cpufeatures.h. For instance, User Mode Instruction Protection can be disabled using clearcpuid=514. The number 514 is calculated from #define X86_FEATURE_UMIP (16*32 + 2).

In addition, there exists a variety of custom command-line parameters that disable specific features. The list of parameters includes, but is not limited to, nofsgsbase, nosgx, noxsave, etc. 5-level paging can also be disabled using "no5lvl".

e: The feature was known to be non-functional.

The feature was known to be non-functional because a dependency was missing at runtime. For example, AVX flags will not show up if XSAVE feature is disabled since they depend on XSAVE feature. Another example would be broken CPUs and them missing microcode patches. Due to that, the kernel decides not to enable a feature.

16.4 x86 Topology

This documents and clarifies the main aspects of x86 topology modelling and representation in the kernel. Update/change when doing changes to the respective code.

The architecture-agnostic topology definitions are in Documentation/admin-guide/cputopology.rst. This file holds x86-specific differences/specialities which must not necessarily apply to the generic definitions. Thus, the way to read up on Linux topology on x86 is to start with the generic one and look at this one in parallel for the x86 specifics.

Needless to say, code should use the generic functions - this file is *only* here to *document* the inner workings of x86 topology.

Started by Thomas Gleixner <tglx@linutronix.de> and Borislav Petkov <bp@alien8.de>.

The main aim of the topology facilities is to present adequate interfaces to code which needs to know/query/use the structure of the running system wrt threads, cores, packages, etc.

The kernel does not care about the concept of physical sockets because a socket has no relevance to software. It's an electromechanical component. In the past a socket always contained

[1] 5-level paging uses linear address of 57 bits.

a single package (see below), but with the advent of Multi Chip Modules (MCM) a socket can hold more than one package. So there might be still references to sockets in the code, but they are of historical nature and should be cleaned up.

The topology of a system is described in the units of:

- packages
- cores
- threads

16.4.1 Package

Packages contain a number of cores plus shared resources, e.g. DRAM controller, shared caches etc.

Modern systems may also use the term 'Die' for package.

AMD nomenclature for package is 'Node'.

Package-related topology information in the kernel:

- cpuinfo_x86.x86_max_cores:

  The number of cores in a package. This information is retrieved via CPUID.

- cpuinfo_x86.x86_max_dies:

  The number of dies in a package. This information is retrieved via CPUID.

- cpuinfo_x86.cpu_die_id:

  The physical ID of the die. This information is retrieved via CPUID.

- cpuinfo_x86.phys_proc_id:

  The physical ID of the package. This information is retrieved via CPUID and deduced from the APIC IDs of the cores in the package.

  Modern systems use this value for the socket. There may be multiple packages within a socket. This value may differ from cpu_die_id.

- cpuinfo_x86.logical_proc_id:

  The logical ID of the package. As we do not trust BIOSes to enumerate the packages in a consistent way, we introduced the concept of logical package ID so we can sanely calculate the number of maximum possible packages in the system and have the packages enumerated linearly.

- topology_max_packages():

  The maximum possible number of packages in the system. Helpful for per package facilities to preallocate per package information.

- cpu_llc_id:

  A per-CPU variable containing:

  - On Intel, the first APIC ID of the list of CPUs sharing the Last Level Cache
  - On AMD, the Node ID or Core Complex ID containing the Last Level Cache. In general, it is a number identifying an LLC uniquely on the system.

16.4.2 Cores

A core consists of 1 or more threads. It does not matter whether the threads are SMT- or CMT-type threads.

AMDs nomenclature for a CMT core is “Compute Unit”. The kernel always uses “core”.

Core-related topology information in the kernel:

- smp_num_siblings:

  The number of threads in a core. The number of threads in a package can be calculated by:

  ```
  threads_per_package = cpuinfo_x86.x86_max_cores * smp_num_siblings
  ```

16.4.3 Threads

A thread is a single scheduling unit. It’s the equivalent to a logical Linux CPU.

AMDs nomenclature for CMT threads is “Compute Unit Core”. The kernel always uses “thread”.

Thread-related topology information in the kernel:

- topology_core_cpumask():

  The cpumask contains all online threads in the package to which a thread belongs.

  The number of online threads is also printed in /proc/cpuinfo “siblings.”

- topology_sibling_cpumask():

  The cpumask contains all online threads in the core to which a thread belongs.

- topology_logical_package_id():

  The logical package ID to which a thread belongs.

- topology_physical_package_id():

  The physical package ID to which a thread belongs.

- topology_core_id();

  The ID of the core to which a thread belongs. It is also printed in /proc/cpuinfo “core_id.”

16.4.4 System topology examples

Note: The alternative Linux CPU enumeration depends on how the BIOS enumerates the threads. Many BIOSes enumerate all threads 0 first and then all threads 1. That has the “advantage” that the logical Linux CPU numbers of threads 0 stay the same whether threads are enabled or not. That’s merely an implementation detail and has no practical impact.

1) Single Package, Single Core:

   ```
   [package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
   ```

2) Single Package, Dual Core

   a) One thread per core:

   ```
   [package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
               -> [core 1] -> [thread 0] -> Linux CPU 1
   ```

   b) Two threads per core:

   ```
   [package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
                          -> [thread 1] -> Linux CPU 1
               -> [core 1] -> [thread 0] -> Linux CPU 2
                          -> [thread 1] -> Linux CPU 3
   ```

   Alternative enumeration:

   ```
   [package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
                          -> [thread 1] -> Linux CPU 2
               -> [core 1] -> [thread 0] -> Linux CPU 1
                          -> [thread 1] -> Linux CPU 3
   ```

   AMD nomenclature for CMT systems:

   ```
   [node 0] -> [Compute Unit 0] -> [Compute Unit Core 0] -> Linux CPU 0
                                -> [Compute Unit Core 1] -> Linux CPU 1
            -> [Compute Unit 1] -> [Compute Unit Core 0] -> Linux CPU 2
                                -> [Compute Unit Core 1] -> Linux CPU 3
   ```

4) Dual Package, Dual Core

   a) One thread per core:

   ```
   [package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
               -> [core 1] -> [thread 0] -> Linux CPU 1

   [package 1] -> [core 0] -> [thread 0] -> Linux CPU 2
               -> [core 1] -> [thread 0] -> Linux CPU 3
   ```

   b) Two threads per core:

   ```
   [package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
                          -> [thread 1] -> Linux CPU 1
               -> [core 1] -> [thread 0] -> Linux CPU 2
                          -> [thread 1] -> Linux CPU 3

   [package 1] -> [core 0] -> [thread 0] -> Linux CPU 4
                          -> [thread 1] -> Linux CPU 5
               -> [core 1] -> [thread 0] -> Linux CPU 6
                          -> [thread 1] -> Linux CPU 7
   ```

   Alternative enumeration:

   ```
   [package 0] -> [core 0] -> [thread 0] -> Linux CPU 0
                          -> [thread 1] -> Linux CPU 4
   ```

```
            -> [core 1] -> [thread 0] -> Linux CPU 1
                        -> [thread 1] -> Linux CPU 5

[package 1] -> [core 0] -> [thread 0] -> Linux CPU 2
                        -> [thread 1] -> Linux CPU 6
            -> [core 1] -> [thread 0] -> Linux CPU 3
                        -> [thread 1] -> Linux CPU 7
```

AMD nomenclature for CMT systems:

```
[node 0] -> [Compute Unit 0] -> [Compute Unit Core 0] -> Linux CPU 0
                             -> [Compute Unit Core 1] -> Linux CPU 1
         -> [Compute Unit 1] -> [Compute Unit Core 0] -> Linux CPU 2
                             -> [Compute Unit Core 1] -> Linux CPU 3

[node 1] -> [Compute Unit 0] -> [Compute Unit Core 0] -> Linux CPU 4
                             -> [Compute Unit Core 1] -> Linux CPU 5
         -> [Compute Unit 1] -> [Compute Unit Core 0] -> Linux CPU 6
                             -> [Compute Unit Core 1] -> Linux CPU 7
```

16.5 Kernel level exception handling

Commentary by Joerg Pommnitz <joerg@raleigh.ibm.com>

When a process runs in kernel mode, it often has to access user mode memory whose address has been passed by an untrusted program. To protect itself the kernel has to verify this address.

In older versions of Linux this was done with the int verify_area(int type, const void * addr, unsigned long size) function (which has since been replaced by access_ok()).

This function verified that the memory area starting at address 'addr' and of size 'size' was accessible for the operation specified in type (read or write). To do this, verify_read had to look up the virtual memory area (vma) that contained the address addr. In the normal case (correctly working program), this test was successful. It only failed for a few buggy programs. In some kernel profiling tests, this normally unneeded verification used up a considerable amount of time.

To overcome this situation, Linus decided to let the virtual memory hardware present in every Linux-capable CPU handle this test.

How does this work?

Whenever the kernel tries to access an address that is currently not accessible, the CPU generates a page fault exception and calls the page fault handler:

```
void exc_page_fault(struct pt_regs *regs, unsigned long error_code)
```

in arch/x86/mm/fault.c. The parameters on the stack are set up by the low level assembly glue in arch/x86/entry/entry_32.S. The parameter regs is a pointer to the saved registers on the stack, error_code contains a reason code for the exception.

exc_page_fault() first obtains the inaccessible address from the CPU control register CR2. If the address is within the virtual address space of the process, the fault probably occurred, because the page was not swapped in, write protected or something similar. However, we are interested in the other case: the address is not valid, there is no vma that contains this address. In this case, the kernel jumps to the bad_area label.

There it uses the address of the instruction that caused the exception (i.e. regs->eip) to find an address where the execution can continue (fixup). If this search is successful, the fault handler modifies the return address (again regs->eip) and returns. The execution will continue at the address in fixup.

Where does fixup point to?

Since we jump to the contents of fixup, fixup obviously points to executable code. This code is hidden inside the user access macros. I have picked the get_user() macro defined in arch/x86/include/asm/uaccess.h as an example. The definition is somewhat hard to follow, so let's peek at the code generated by the preprocessor and the compiler. I selected the get_user() call in drivers/char/sysrq.c for a detailed examination.

The original code in sysrq.c line 587:

```
get_user(c, buf);
```

The preprocessor output (edited to become somewhat readable):

```
(
  {
    long __gu_err = - 14 , __gu_val = 0;
    const __typeof__(*( (  buf ) )) *__gu_addr = ((buf));
    if (((((0 + current_set[0])->tss.segment) == 0x18 )  ||
      (((sizeof(*(buf))) <= 0xC0000000UL) &&
      ((unsigned long)(__gu_addr ) <= 0xC0000000UL - (sizeof(*(buf)))))))
      do {
        __gu_err  = 0;
        switch ((sizeof(*(buf)))) {
          case 1:
            __asm__ __volatile__(
              "1:      mov" "b" " %2,%" "b" "1\n"
              "2:\n"
              ".section .fixup,\"ax\"\n"
              "3:      movl %3,%0\n"
              "        xor" "b" " %" "b" "1,%" "b" "1\n"
              "        jmp 2b\n"
              ".section __ex_table,\"a\"\n"
              "        .align 4\n"
              "        .long 1b,3b\n"
              ".text"        : "=r"(__gu_err), "=q" (__gu_val): "m"((*(struct _
↪_large_struct *)
                              (    __gu_addr   )) ), "i"(- 14 ), "0"(   __gu_err ␣
↪)) ;
              break;
          case 2:
            __asm__ __volatile__(
              "1:      mov" "w" " %2,%" "w" "1\n"
              "2:\n"
              ".section .fixup,\"ax\"\n"
              "3:      movl %3,%0\n"
              "        xor" "w" " %" "w" "1,%" "w" "1\n"
              "        jmp 2b\n"
              ".section __ex_table,\"a\"\n"
              "        .align 4\n"
              "        .long 1b,3b\n"
              ".text"        : "=r"(__gu_err), "=r" (__gu_val) : "m"((*(struct␣
↪__large_struct *)
                            (   __gu_addr   )) ), "i"(- 14 ), "0"(  __gu_err ␣
↪));
              break;
          case 4:
            __asm__ __volatile__(
              "1:      mov" "l" " %2,%" "" "1\n"
              "2:\n"
              ".section .fixup,\"ax\"\n"
              "3:      movl %3,%0\n"
              "        xor" "l" " %" "" "1,%" "" "1\n"
              "        jmp 2b\n"
              ".section __ex_table,\"a\"\n"
              "        .align 4\n"        "        .long 1b,3b\n"
              ".text"        : "=r"(__gu_err), "=r" (__gu_val) : "m"((*(struct␣
↪__large_struct *)
                            (   __gu_addr   )) ), "i"(- 14 ), "0"(__gu_err));
              break;
          default:
            (__gu_val) = __get_user_bad();
        }
      } while (0) ;
    ((c)) = (__typeof__(*((buf))))__gu_val;
    __gu_err;
  }
);
```

WOW! Black GCC/assembly magic. This is impossible to follow, so let's see what code gcc generates:

```
>         xorl %edx,%edx
>         movl current_set,%eax
>         cmpl $24,788(%eax)
>         je .L1424
>         cmpl $-1073741825,64(%esp)
>         ja .L1423
> .L1424:
>         movl %edx,%eax
>         movl 64(%esp),%ebx
> #APP
> 1:      movb (%ebx),%dl                /* this is the actual user access */
> 2:
> .section .fixup,"ax"
> 3:      movl $-14,%eax
>         xorb %dl,%dl
>         jmp 2b
> .section __ex_table,"a"
>         .align 4
>         .long 1b,3b
> .text
> #NO_APP
> .L1423:
>         movzbl %dl,%esi
```

The optimizer does a good job and gives us something we can actually understand. Can we? The actual user access is quite obvious. Thanks to the unified address space we can just access the address in user memory. But what does the .section stuff do?????

To understand this we have to look at the final kernel:

```
> objdump --section-headers vmlinux
>
> vmlinux:     file format elf32-i386
>
> Sections:
> Idx Name          Size      VMA       LMA       File off  Algn
>   0 .text         00098f40  c0100000  c0100000  00001000  2**4
>                   CONTENTS, ALLOC, LOAD, READONLY, CODE
>   1 .fixup        000016bc  c0198f40  c0198f40  00099f40  2**0
>                   CONTENTS, ALLOC, LOAD, READONLY, CODE
>   2 .rodata       0000f127  c019a5fc  c019a5fc  0009b5fc  2**2
>                   CONTENTS, ALLOC, LOAD, READONLY, DATA
>   3 __ex_table    000015c0  c01a9724  c01a9724  000aa724  2**2
>                   CONTENTS, ALLOC, LOAD, READONLY, DATA
>   4 .data         0000ea58  c01abcf0  c01abcf0  000abcf0  2**4
>                   CONTENTS, ALLOC, LOAD, DATA
>   5 .bss          00018e21  c01ba748  c01ba748  000ba748  2**2
>                   ALLOC
>   6 .comment      00000ec4  00000000  00000000  000ba748  2**0
>                   CONTENTS, READONLY
>   7 .note         00001068  00000ec4  00000ec4  000bb60c  2**0
>                   CONTENTS, READONLY
```

There are obviously 2 non standard ELF sections in the generated object file. But first we want to find out what happened to our code in the final kernel executable:

```
> objdump --disassemble --section=.text vmlinux
>
> c017e785 <do_con_write+c1> xorl   %edx,%edx
> c017e787 <do_con_write+c3> movl   0xc01c7bec,%eax
> c017e78c <do_con_write+c8> cmpl   $0x18,0x314(%eax)
> c017e793 <do_con_write+cf> je     c017e79f <do_con_write+db>
> c017e795 <do_con_write+d1> cmpl   $0xbfffffff,0x40(%esp,1)
> c017e79d <do_con_write+d9> ja     c017e7a7 <do_con_write+e3>
> c017e79f <do_con_write+db> movl   %edx,%eax
> c017e7a1 <do_con_write+dd> movl   0x40(%esp,1),%ebx
> c017e7a5 <do_con_write+e1> movb   (%ebx),%dl
> c017e7a7 <do_con_write+e3> movzbl %dl,%esi
```

The whole user memory access is reduced to 10 x86 machine instructions. The instructions bracketed in the .section directives are no longer in the normal execution path. They are located in a different section of the executable file:

```
> objdump --disassemble --section=.fixup vmlinux
>
> c0199ff5 <.fixup+10b5> movl   $0xfffffff2,%eax
> c0199ffa <.fixup+10ba> xorb   %dl,%dl
> c0199ffc <.fixup+10bc> jmp    c017e7a7 <do_con_write+e3>
```

And finally:

```
> objdump --full-contents --section=__ex_table vmlinux
>
>  c01aa7c4 93c017c0 e09f19c0 97c017c0 99c017c0  ................
>  c01aa7d4 f6c217c0 e99f19c0 a5e717c0 f59f19c0  ................
>  c01aa7e4 080a18c0 01a019c0 0a0a18c0 04a019c0  ................
```

or in human readable byte order:

```
>  c01aa7c4 c017c093 c0199fe0 c017c097 c017c099  ................
>  c01aa7d4 c017c2f6 c0199fe9 c017e7a5 c0199ff5  ................
                              ^^^^^^^^^^^^^^^^^
                              this is the interesting part!
>  c01aa7e4 c0180a08 c019a001 c0180a0a c019a004  ................
```

What happened? The assembly directives:

```
.section .fixup,"ax"
.section __ex_table,"a"
```

told the assembler to move the following code to the specified sections in the ELF object file. So the instructions:

```
3:      movl $-14,%eax
        xorb %dl,%dl
        jmp 2b
```

ended up in the .fixup section of the object file and the addresses:

```
.long 1b,3b
```

ended up in the __ex_table section of the object file. 1b and 3b are local labels. The local label 1b (1b stands for next label 1 backward) is the address of the instruction that might fault, i.e. in our case the address of the label 1 is c017e7a5: the original assembly code: > 1: movb (%ebx),%dl and linked in vmlinux : > c017e7a5 <do_con_write+e1> movb (%ebx),%dl

The local label 3 (backwards again) is the address of the code to handle the fault, in our case the actual value is c0199ff5: the original assembly code: > 3: movl $-14,%eax and linked in vmlinux : > c0199ff5 <.fixup+10b5> movl $0xfffffff2,%eax

If the fixup was able to handle the exception, control flow may be returned to the instruction after the one that triggered the fault, ie. local label 2b.

The assembly code:

```
> .section __ex_table,"a"
>         .align 4
>         .long 1b,3b
```

becomes the value pair:

```
>  c01aa7d4 c017c2f6 c0199fe9 c017e7a5 c0199ff5  ................
                              ^this is ^this is
                              1b       3b
```

c017e7a5,c0199ff5 in the exception table of the kernel.

So, what actually happens if a fault from kernel mode with no suitable vma occurs?

1. access to invalid address:

   ```
   > c017e7a5 <do_con_write+e1> movb   (%ebx),%dl
   ```

2. MMU generates exception
3. CPU calls exc_page_fault()
4. exc_page_fault() calls do_user_addr_fault()
5. do_user_addr_fault() calls kernelmode_fixup_or_oops()
6. kernelmode_fixup_or_oops() calls fixup_exception() (regs->eip == c017e7a5);
7. fixup_exception() calls search_exception_tables()
8. search_exception_tables() looks up the address c017e7a5 in the exception table (i.e. the contents of the ELF section __ex_table) and returns the address of the associated fault handle code c0199ff5.
9. fixup_exception() modifies its own return address to point to the fault handle code and returns.
10. execution continues in the fault handling code.
11. a) EAX becomes -EFAULT (== -14)
    b) DL becomes zero (the value we “read” from user space)
    c) execution continues at local label 2 (address of the instruction immediately after the faulting user access).

The steps 8a to 8c in a certain way emulate the faulting instruction.

That’s it, mostly. If you look at our example, you might ask why we set EAX to -EFAULT in the exception handler code. Well, the get_user() macro actually returns a value: 0, if the user access was successful, -EFAULT on failure. Our original code did not test this return value, however the inline assembly code in get_user() tries to return -EFAULT. GCC selected EAX to return this value.

NOTE: Due to the way that the exception table is built and needs to be ordered, only use exceptions for code in the .text section. Any other section will cause the exception table to not be sorted correctly, and the exceptions will fail.

Things changed when 64-bit support was added to x86 Linux. Rather than double the size of the exception table by expanding the two entries from 32-bits to 64 bits, a clever trick was used to store addresses as relative offsets from the table itself. The assembly code changed from:

```
  .long 1b,3b
to:
        .long (from) - .
        .long (to) - .
```

and the C-code that uses these values converts back to absolute addresses like this:

```
ex_insn_addr(const struct exception_table_entry *x)
{
        return (unsigned long)&x->insn + x->insn;
}
```

In v4.6 the exception table entry was expanded with a new field "handler". This is also 32-bits wide and contains a third relative function pointer which points to one of:

1) **`int ex_handler_default(const struct exception_table_entry *fixup)`**
   This is legacy case that just jumps to the fixup code

2) **`int ex_handler_fault(const struct exception_table_entry *fixup)`**
   This case provides the fault number of the trap that occurred at entry->insn. It is used to distinguish page faults from machine check.

More functions can easily be added.

CONFIG_BUILDTIME_TABLE_SORT allows the __ex_table section to be sorted post link of the kernel image, via a host utility scripts/sorttable. It will set the symbol main_extable_sort_needed to 0, avoiding sorting the __ex_table section at boot time. With the exception table sorted, at runtime when an exception occurs we can quickly lookup the __ex_table entry via binary search.

This is not just a boot time optimization, some architectures require this table to be sorted in order to handle exceptions relatively early in the boot process. For example, i386 makes use of this form of exception handling before paging support is even enabled!

16.6 Kernel Stacks

16.6.1 Kernel stacks on x86-64 bit

Most of the text from Keith Owens, hacked by AK

x86_64 page size (PAGE_SIZE) is 4K.

Like all other architectures, x86_64 has a kernel stack for every active thread. These thread stacks are THREAD_SIZE (4*PAGE_SIZE) big. These stacks contain useful data as long as a thread is alive or a zombie. While the thread is in user space the kernel stack is empty except for the thread_info structure at the bottom.

In addition to the per thread stacks, there are specialized stacks associated with each CPU. These stacks are only used while the kernel is in control on that CPU; when a CPU returns to user space the specialized stacks contain no useful data. The main CPU stacks are:

- Interrupt stack. IRQ_STACK_SIZE

  Used for external hardware interrupts. If this is the first external hardware interrupt (i.e. not a nested hardware interrupt) then the kernel switches from the current task to the interrupt stack. Like the split thread and interrupt stacks on i386, this gives more room for kernel interrupt processing without having to increase the size of every per thread stack.

  The interrupt stack is also used when processing a softirq.

Switching to the kernel interrupt stack is done by software based on a per CPU interrupt nest counter. This is needed because x86-64 "IST" hardware stacks cannot nest without races.

x86_64 also has a feature which is not available on i386, the ability to automatically switch to a new stack for designated events such as double fault or NMI, which makes it easier to handle these unusual events on x86_64. This feature is called the Interrupt Stack Table (IST). There can be up to 7 IST entries per CPU. The IST code is an index into the Task State Segment (TSS). The IST entries in the TSS point to dedicated stacks; each stack can be a different size.

An IST is selected by a non-zero value in the IST field of an interrupt-gate descriptor. When an interrupt occurs and the hardware loads such a descriptor, the hardware automatically sets the new stack pointer based on the IST value, then invokes the interrupt handler. If the interrupt came from user mode, then the interrupt handler prologue will switch back to the per-thread stack. If software wants to allow nested IST interrupts then the handler must adjust the IST values on entry to and exit from the interrupt handler. (This is occasionally done, e.g. for debug exceptions.)

Events with different IST codes (i.e. with different stacks) can be nested. For example, a debug interrupt can safely be interrupted by an NMI. arch/x86_64/kernel/entry.S::paranoidentry adjusts the stack pointers on entry to and exit from all IST events, in theory allowing IST events with the same code to be nested. However in most cases, the stack size allocated to an IST assumes no nesting for the same code. If that assumption is ever broken then the stacks will become corrupt.

The currently assigned IST stacks are:

- ESTACK_DF. EXCEPTION_STKSZ (PAGE_SIZE).

  Used for interrupt 8 - Double Fault Exception (#DF).

  Invoked when handling one exception causes another exception. Happens when the kernel is very confused (e.g. kernel stack pointer corrupt). Using a separate stack allows the kernel to recover from it well enough in many cases to still output an oops.

- ESTACK_NMI. EXCEPTION_STKSZ (PAGE_SIZE).

  Used for non-maskable interrupts (NMI).

  NMI can be delivered at any time, including when the kernel is in the middle of switching stacks. Using IST for NMI events avoids making assumptions about the previous state of the kernel stack.

- ESTACK_DB. EXCEPTION_STKSZ (PAGE_SIZE).

  Used for hardware debug interrupts (interrupt 1) and for software debug interrupts (INT3).

When debugging a kernel, debug interrupts (both hardware and software) can occur at any time. Using IST for these interrupts avoids making assumptions about the previous state of the kernel stack.

To handle nested #DB correctly there exist two instances of DB stacks. On #DB entry the IST stackpointer for #DB is switched to the second instance so a nested #DB starts from a clean stack. The nested #DB switches the IST stackpointer to a guard hole to catch triple nesting.

- ESTACK_MCE. EXCEPTION_STKSZ (PAGE_SIZE).

  Used for interrupt 18 - Machine Check Exception (#MC).

  MCE can be delivered at any time, including when the kernel is in the middle of switching stacks. Using IST for MCE events avoids making assumptions about the previous state of the kernel stack.

For more details see the Intel IA32 or AMD AMD64 architecture manuals.

16.6.2 Printing backtraces on x86

The question about the '?' preceding function names in an x86 stacktrace keeps popping up, here's an indepth explanation. It helps if the reader stares at print_context_stack() and the whole machinery in and around arch/x86/kernel/dumpstack.c.

Adapted from Ingo's mail, Message-ID: <20150521101614.GA10889@gmail.com>:

We always scan the full kernel stack for return addresses stored on the kernel stack(s)[1], from stack top to stack bottom, and print out anything that 'looks like' a kernel text address.

If it fits into the frame pointer chain, we print it without a question mark, knowing that it's part of the real backtrace.

If the address does not fit into our expected frame pointer chain we still print it, but we print a '?'. It can mean two things:

- either the address is not part of the call chain: it's just stale values on the kernel stack, from earlier function calls. This is the common case.
- or it is part of the call chain, but the frame pointer was not set up properly within the function, so we don't recognize it.

This way we will always print out the real call chain (plus a few more entries), regardless of whether the frame pointer was set up correctly or not - but in most cases we'll get the call chain right as well. The entries printed are strictly in stack order, so you can deduce more information from that as well.

The most important property of this method is that we _never_ lose information: we always strive to print _all_ addresses on the stack(s) that look like kernel text addresses, so if debug information is wrong, we still print out the real call chain as well - just with more question marks than ideal.

[1] For things like IRQ and IST stacks, we also scan those stacks, in the right order, and try to cross from one stack into another reconstructing the call chain. This works most of the time.

16.7 Kernel Entries

This file documents some of the kernel entries in arch/x86/entry/entry_64.S. A lot of this explanation is adapted from an email from Ingo Molnar:

https://lore.kernel.org/r/20110529191055.GC9835%40elte.hu

The x86 architecture has quite a few different ways to jump into kernel code. Most of these entry points are registered in arch/x86/kernel/traps.c and implemented in arch/x86/entry/entry_64.S for 64-bit, arch/x86/entry/entry_32.S for 32-bit and finally arch/x86/entry/entry_64_compat.S which implements the 32-bit compatibility syscall entry points and thus provides for 32-bit processes the ability to execute syscalls when running on 64-bit kernels.

The IDT vector assignments are listed in arch/x86/include/asm/irq_vectors.h.

Some of these entries are:

- system_call: syscall instruction from 64-bit code.
- entry_INT80_compat: int 0x80 from 32-bit or 64-bit code; compat syscall either way.
- entry_INT80_compat, ia32_sysenter: syscall and sysenter from 32-bit code
- interrupt: An array of entries. Every IDT vector that doesn't explicitly point somewhere else gets set to the corresponding value in interrupts. These point to a whole array of magically-generated functions that make their way to common_interrupt() with the interrupt number as a parameter.
- APIC interrupts: Various special-purpose interrupts for things like TLB shootdown.
- Architecturally-defined exceptions like divide_error.

There are a few complexities here. The different x86-64 entries have different calling conventions. The syscall and sysenter instructions have their own peculiar calling conventions. Some of the IDT entries push an error code onto the stack; others don't. IDT entries using the IST alternative stack mechanism need their own magic to get the stack frames right. (You can find some documentation in the AMD APM, Volume 2, Chapter 8 and the Intel SDM, Volume 3, Chapter 6.)

Dealing with the swapgs instruction is especially tricky. Swapgs toggles whether gs is the kernel gs or the user gs. The swapgs instruction is rather fragile: it must nest perfectly and only in single depth, it should only be used if entering from user mode to kernel mode and then when returning to user-space, and precisely so. If we mess that up even slightly, we crash.

So when we have a secondary entry, already in kernel mode, we *must not* use SWAPGS blindly - nor must we forget doing a SWAPGS when it's not switched/swapped yet.

Now, there's a secondary complication: there's a cheap way to test which mode the CPU is in and an expensive way.

The cheap way is to pick this info off the entry frame on the kernel stack, from the CS of the ptregs area of the kernel stack:

```
xorl %ebx,%ebx
testl $3,CS+8(%rsp)
je error_kernelspace
SWAPGS
```

The expensive (paranoid) way is to read back the MSR_GS_BASE value (which is what SWAPGS modifies):

```
      movl $1,%ebx
      movl $MSR_GS_BASE,%ecx
      rdmsr
      testl %edx,%edx
      js 1f   /* negative -> in kernel */
      SWAPGS
      xorl %ebx,%ebx
1:    ret
```

If we are at an interrupt or user-trap/gate-alike boundary then we can use the faster check: the stack will be a reliable indicator of whether SWAPGS was already done: if we see that we are a secondary entry interrupting kernel mode execution, then we know that the GS base has already been switched. If it says that we interrupted user-space execution then we must do the SWAPGS.

But if we are in an NMI/MCE/DEBUG/whatever super-atomic entry context, which might have triggered right after a normal entry wrote CS to the stack but before we executed SWAPGS, then the only safe way to check for GS is the slower method: the RDMSR.

Therefore, super-atomic entries (except NMI, which is handled separately) must use idtentry with paranoid=1 to handle gsbase correctly. This triggers three main behavior changes:

- Interrupt entry will use the slower gsbase check.
- Interrupt entry from user mode will switch off the IST stack.
- Interrupt exit to kernel mode will not attempt to reschedule.

We try to only use IST entries and the paranoid entry code for vectors that absolutely need the more expensive check for the GS base - and we generate all 'normal' entry points with the regular (faster) paranoid=0 variant.

16.8 Early Printk

Mini-HOWTO for using the earlyprintk=dbgp boot option with a USB2 Debug port key and a debug cable, on x86 systems.

You need two computers, the 'USB debug key' special gadget and two USB cables, connected like this:

```
[host/target] <-------> [USB debug key] <-------> [client/console]
```

16.8.1 Hardware requirements

a) Host/target system needs to have USB debug port capability.

   You can check this capability by looking at a 'Debug port' bit in the lspci -vvv output:

```
# lspci -vvv
...
00:1d.7 USB Controller: Intel Corporation 82801H (ICH8 Family) USB2 EHCI␣
↪Controller #1 (rev 03) (prog-if 20 [EHCI])
        Subsystem: Lenovo ThinkPad T61
        Control: I/O- Mem+ BusMaster+ SpecCycle- MemWINV- VGASnoop- ParErr-
↪ Stepping- SERR+ FastB2B- DisINTx-
        Status: Cap+ 66MHz- UDF- FastB2B+ ParErr- DEVSEL=medium >TAbort-
↪<TAbort- <MAbort- >SERR- <PERR- INTx-
        Latency: 0
        Interrupt: pin D routed to IRQ 19
        Region 0: Memory at fe227000 (32-bit, non-prefetchable) [size=1K]
        Capabilities: [50] Power Management version 2
                Flags: PMEClk- DSI- D1- D2- AuxCurrent=375mA PME(D0+,D1-,
↪D2-,D3hot+,D3cold+)
                Status: D0 PME-Enable- DSel=0 DScale=0 PME+
        Capabilities: [58] Debug port: BAR=1 offset=00a0
                     ^^^^^^^^^^^ <==================== [ HERE ]
        Kernel driver in use: ehci_hcd
        Kernel modules: ehci-hcd
...
```

   Note: If your system does not list a debug port capability then you probably won't be able to use the USB debug key.

b) You also need a NetChip USB debug cable/key:

   http://www.plxtech.com/products/NET2000/NET20DC/default.asp

   This is a small blue plastic connector with two USB connections; it draws power from its USB connections.

c) You need a second client/console system with a high speed USB 2.0 port.

d) The NetChip device must be plugged directly into the physical debug port on the "host/target" system. You cannot use a USB hub in between the physical debug port and the "host/target" system.

   The EHCI debug controller is bound to a specific physical USB port and the NetChip device will only work as an early printk device in this port. The EHCI host controllers are electrically wired such that the EHCI debug controller is hooked up to the first physical port and there is no way to change this via software. You can find the physical port through experimentation by trying each physical port on the system and rebooting. Or you can try and use lsusb or look at the kernel info messages emitted by the usb stack when you plug a usb device into various ports on the "host/target" system.

   Some hardware vendors do not expose the usb debug port with a physical connector and if you find such a device send a complaint to the hardware vendor, because there is no reason not to wire this port into one of the physically accessible ports.

e) It is also important to note, that many versions of the NetChip device require the "client/console" system to be plugged into the right hand side of the device (with the product logo facing up and readable left to right). The reason being is that the 5 volt power supply is taken from only one side of the device and it must be the side that does not get rebooted.

16.8.2 Software requirements

a) On the host/target system:

   You need to enable the following kernel config option:

   ```
   CONFIG_EARLY_PRINTK_DBGP=y
   ```

   And you need to add the boot command line: "earlyprintk=dbgp".

   Note: If you are using Grub, append it to the 'kernel' line in /etc/grub.conf. If you are using Grub2 on a BIOS firmware system, append it to the 'linux' line in /boot/grub2/grub.cfg. If you are using Grub2 on an EFI firmware system, append it to the 'linux' or 'linuxefi' line in /boot/grub2/grub.cfg or /boot/efi/EFI/<distro>/grub.cfg.

   On systems with more than one EHCI debug controller you must specify the correct EHCI debug controller number. The ordering comes from the PCI bus enumeration of the EHCI controllers. The default with no number argument is "0" or the first EHCI debug controller. To use the second EHCI debug controller, you would use the command line: "earlyprintk=dbgp1"

   Note: normally earlyprintk console gets turned off once the regular console is alive - use "earlyprintk=dbgp,keep" to keep this channel open beyond early bootup. This can be useful for debugging crashes under Xorg, etc.

b) On the client/console system:

   You should enable the following kernel config option:

   ```
   CONFIG_USB_SERIAL_DEBUG=y
   ```

   On the next bootup with the modified kernel you should get a /dev/ttyUSBx device(s).

   Now this channel of kernel messages is ready to be used: start your favorite terminal emulator (minicom, etc.) and set it up to use /dev/ttyUSB0 - or use a raw 'cat /dev/ttyUSBx' to see the raw output.

c) On Nvidia Southbridge based systems: the kernel will try to probe and find out which port has a debug device connected.

16.8.3 Testing

You can test the output by using earlyprintk=dbgp,keep and provoking kernel messages on the host/target system. You can provoke a harmless kernel message by for example doing:

```
echo h > /proc/sysrq-trigger
```

On the host/target system you should see this help line in "dmesg" output:

```
SysRq : HELP : loglevel(0-9) reBoot Crashdump terminate-all-tasks(E) memory-
↪full-oom-kill(F) kill-all-tasks(I) saK show-backtrace-all-active-cpus(L)␣
↪show-memory-usage(M) nice-all-RT-tasks(N) powerOff show-registers(P) show-
↪all-timers(Q) unRaw Sync show-task-states(T) Unmount show-blocked-tasks(W)␣
↪dump-ftrace-buffer(Z)
```

On the client/console system do:

```
cat /dev/ttyUSB0
```

And you should see the help line above displayed shortly after you've provoked it on the host system.

If it does not work then please ask about it on the linux-kernel@vger.kernel.org mailing list or contact the x86 maintainers.

16.9 ORC unwinder

16.9.1 Overview

The kernel CONFIG_UNWINDER_ORC option enables the ORC unwinder, which is similar in concept to a DWARF unwinder. The difference is that the format of the ORC data is much simpler than DWARF, which in turn allows the ORC unwinder to be much simpler and faster.

The ORC data consists of unwind tables which are generated by objtool. They contain out-of-band data which is used by the in-kernel ORC unwinder. Objtool generates the ORC data by first doing compile-time stack metadata validation (CONFIG_STACK_VALIDATION). After analyzing all the code paths of a .o file, it determines information about the stack state at each instruction address in the file and outputs that information to the .orc_unwind and .orc_unwind_ip sections.

The per-object ORC sections are combined at link time and are sorted and post-processed at boot time. The unwinder uses the resulting data to correlate instruction addresses with their stack states at run time.

16.9.2 ORC vs frame pointers

With frame pointers enabled, GCC adds instrumentation code to every function in the kernel. The kernel's .text size increases by about 3.2%, resulting in a broad kernel-wide slowdown. Measurements by Mel Gorman[1] have shown a slowdown of 5-10% for some workloads.

In contrast, the ORC unwinder has no effect on text size or runtime performance, because the debuginfo is out of band. So if you disable frame pointers and enable the ORC unwinder, you get a nice performance improvement across the board, and still have reliable stack traces.

Ingo Molnar says:

> "Note that it's not just a performance improvement, but also an instruction cache locality improvement: 3.2% .text savings almost directly transform into a similarly sized reduction in cache footprint. That can transform to even higher speedups for workloads whose cache locality is borderline."

Another benefit of ORC compared to frame pointers is that it can reliably unwind across interrupts and exceptions. Frame pointer based unwinds can sometimes skip the caller of the interrupted function, if it was a leaf function or if the interrupt hit before the frame pointer was saved.

The main disadvantage of the ORC unwinder compared to frame pointers is that it needs more memory to store the ORC unwind tables: roughly 2-4MB depending on the kernel config.

16.9.3 ORC vs DWARF

ORC debuginfo's advantage over DWARF itself is that it's much simpler. It gets rid of the complex DWARF CFI state machine and also gets rid of the tracking of unnecessary registers. This allows the unwinder to be much simpler, meaning fewer bugs, which is especially important for mission critical oops code.

The simpler debuginfo format also enables the unwinder to be much faster than DWARF, which is important for perf and lockdep. In a basic performance test by Jiri Slaby[2], the ORC unwinder was about 20x faster than an out-of-tree DWARF unwinder. (Note: That measurement was taken before some performance tweaks were added, which doubled performance, so the speedup over DWARF may be closer to 40x.)

The ORC data format does have a few downsides compared to DWARF. ORC unwind tables take up ~50% more RAM (+1.3MB on an x86 defconfig kernel) than DWARF-based eh_frame tables.

Another potential downside is that, as GCC evolves, it's conceivable that the ORC data may end up being *too* simple to describe the state of the stack for certain optimizations. But IMO this is unlikely because GCC saves the frame pointer for any unusual stack adjustments it does, so I suspect we'll really only ever need to keep track of the stack pointer and the frame pointer between call frames. But even if we do end up having to track all the registers DWARF tracks, at least we will still be able to control the format, e.g. no complex state machines.

[1] https://lore.kernel.org/r/20170602104048.jkkzssljsompjdwy@suse.de

[2] https://lore.kernel.org/r/d2ca5435-6386-29b8-db87-7f227c2b713a@suse.cz

16.9.4 ORC unwind table generation

The ORC data is generated by objtool. With the existing compile-time stack metadata validation feature, objtool already follows all code paths, and so it already has all the information it needs to be able to generate ORC data from scratch. So it's an easy step to go from stack validation to ORC data generation.

It should be possible to instead generate the ORC data with a simple tool which converts DWARF to ORC data. However, such a solution would be incomplete due to the kernel's extensive use of asm, inline asm, and special sections like exception tables.

That could be rectified by manually annotating those special code paths using GNU assembler .cfi annotations in .S files, and homegrown annotations for inline asm in .c files. But asm annotations were tried in the past and were found to be unmaintainable. They were often incorrect/incomplete and made the code harder to read and keep updated. And based on looking at glibc code, annotating inline asm in .c files might be even worse.

Objtool still needs a few annotations, but only in code which does unusual things to the stack like entry code. And even then, far fewer annotations are needed than what DWARF would need, so they're much more maintainable than DWARF CFI annotations.

So the advantages of using objtool to generate ORC data are that it gives more accurate debuginfo, with very few annotations. It also insulates the kernel from toolchain bugs which can be very painful to deal with in the kernel since we often have to workaround issues in older versions of the toolchain for years.

The downside is that the unwinder now becomes dependent on objtool's ability to reverse engineer GCC code flow. If GCC optimizations become too complicated for objtool to follow, the ORC data generation might stop working or become incomplete. (It's worth noting that livepatch already has such a dependency on objtool's ability to follow GCC code flow.)

If newer versions of GCC come up with some optimizations which break objtool, we may need to revisit the current implementation. Some possible solutions would be asking GCC to make the optimizations more palatable, or having objtool use DWARF as an additional input, or creating a GCC plugin to assist objtool with its analysis. But for now, objtool follows GCC code quite well.

16.9.5 Unwinder implementation details

Objtool generates the ORC data by integrating with the compile-time stack metadata validation feature, which is described in detail in tools/objtool/Documentation/objtool.txt. After analyzing all the code paths of a .o file, it creates an array of orc_entry structs, and a parallel array of instruction addresses associated with those structs, and writes them to the .orc_unwind and .orc_unwind_ip sections respectively.

The ORC data is split into the two arrays for performance reasons, to make the searchable part of the data (.orc_unwind_ip) more compact. The arrays are sorted in parallel at boot time.

Performance is further improved by the use of a fast lookup table which is created at runtime. The fast lookup table associates a given address with a range of indices for the .orc_unwind table, so that only a small subset of the table needs to be searched.

16.9.6 Etymology

Orcs, fearsome creatures of medieval folklore, are the Dwarves' natural enemies. Similarly, the ORC unwinder was created in opposition to the complexity and slowness of DWARF.

"Although Orcs rarely consider multiple solutions to a problem, they do excel at getting things done because they are creatures of action, not thought."[3] Similarly, unlike the esoteric DWARF unwinder, the veracious ORC unwinder wastes no time or siloconic effort decoding variable-length zero-extended unsigned-integer byte-coded state-machine-based debug information entries.

Similar to how Orcs frequently unravel the well-intentioned plans of their adversaries, the ORC unwinder frequently unravels stacks with brutal, unyielding efficiency.

ORC stands for Oops Rewind Capability.

16.10 Zero Page

The additional fields in struct boot_params as a part of 32-bit boot protocol of kernel. These should be filled by bootloader or 16-bit real-mode setup code of the kernel. References/settings to it mainly are in:

```
arch/x86/include/uapi/asm/bootparam.h
```

[3] http://dustin.wikidot.com/half-orcs-and-orcs

Offset/Size	Proto	Name	Meaning
000/040	ALL	screen_info	Text mode or frame buffer information (struct screen_info)
040/014	ALL	apm_bios_info	APM BIOS information (struct apm_bios_info)
058/008	ALL	tboot_addr	Physical address of tboot shared page
060/010	ALL	ist_info	Intel SpeedStep (IST) BIOS support information (struct ist_info)
070/008	ALL	acpi_rsdp_addr	Physical address of ACPI RSDP table
080/010	ALL	hd0_info	hd0 disk parameter, OBSOLETE!!
090/010	ALL	hd1_info	hd1 disk parameter, OBSOLETE!!
0A0/010	ALL	sys_desc_table	System description table (struct sys_desc_table), OBSOLETE!!
0B0/010	ALL	olpc_ofw_header	OLPC's OpenFirmware CIF and friends
0C0/004	ALL	ext_ramdisk_image	ramdisk_image high 32bits
0C4/004	ALL	ext_ramdisk_size	ramdisk_size high 32bits
0C8/004	ALL	ext_cmd_line_ptr	cmd_line_ptr high 32bits
13C/004	ALL	cc_blob_address	Physical address of Confidential Computing blob
140/080	ALL	edid_info	Video mode setup (struct edid_info)
1C0/020	ALL	efi_info	EFI 32 information (struct efi_info)
1E0/004	ALL	alt_mem_k	Alternative mem check, in KB
1E4/004	ALL	scratch	Scratch field for the kernel setup code
1E8/001	ALL	e820_entries	Number of entries in e820_table (below)
1E9/001	ALL	eddbuf_entries	Number of entries in eddbuf (below)
1EA/001	ALL	edd_mbr_sig_buf_entri	Number of entries in edd_mbr_sig_buffer (below)
1EB/001	ALL	kbd_status	Numlock is enabled
1EC/001	ALL	secure_boot	Secure boot is enabled in the firmware
1EF/001	ALL	sentinel	Used to detect broken bootloaders
290/040	ALL	edd_mbr_sig_buffer	EDD MBR signatures
2D0/A00	ALL	e820_table	E820 memory map table (array of struct e820_entry)
D00/1EC	ALL	eddbuf	EDD data (array of struct edd_info)

16.11 The TLB

When the kernel unmaps or modified the attributes of a range of memory, it has two choices:

1. Flush the entire TLB with a two-instruction sequence. This is a quick operation, but it causes collateral damage: TLB entries from areas other than the one we are trying to flush will be destroyed and must be refilled later, at some cost.
2. Use the invlpg instruction to invalidate a single page at a time. This could potentially cost many more instructions, but it is a much more precise operation, causing no collateral damage to other TLB entries.

Which method to do depends on a few things:

1. The size of the flush being performed. A flush of the entire address space is obviously better performed by flushing the entire TLB than doing 2^48/PAGE_SIZE individual flushes.
2. The contents of the TLB. If the TLB is empty, then there will be no collateral damage caused

by doing the global flush, and all of the individual flush will have ended up being wasted work.

3. The size of the TLB. The larger the TLB, the more collateral damage we do with a full flush. So, the larger the TLB, the more attractive an individual flush looks. Data and instructions have separate TLBs, as do different page sizes.
4. The microarchitecture. The TLB has become a multi-level cache on modern CPUs, and the global flushes have become more expensive relative to single-page flushes.

There is obviously no way the kernel can know all these things, especially the contents of the TLB during a given flush. The sizes of the flush will vary greatly depending on the workload as well. There is essentially no "right" point to choose.

You may be doing too many individual invalidations if you see the invlpg instruction (or instructions _near_ it) show up high in profiles. If you believe that individual invalidations being called too often, you can lower the tunable:

```
/sys/kernel/debug/x86/tlb_single_page_flush_ceiling
```

This will cause us to do the global flush for more cases. Lowering it to 0 will disable the use of the individual flushes. Setting it to 1 is a very conservative setting and it should never need to be 0 under normal circumstances.

Despite the fact that a single individual flush on x86 is guaranteed to flush a full 2MB[1], hugetlbfs always uses the full flushes. THP is treated exactly the same as normal memory.

You might see invlpg inside of flush_tlb_mm_range() show up in profiles, or you can use the trace_tlb_flush() tracepoints. to determine how long the flush operations are taking.

Essentially, you are balancing the cycles you spend doing invlpg with the cycles that you spend refilling the TLB later.

You can measure how expensive TLB refills are by using performance counters and 'perf stat', like this:

```
perf stat -e
  cpu/event=0x8,umask=0x84,name=dtlb_load_misses_walk_duration/,
  cpu/event=0x8,umask=0x82,name=dtlb_load_misses_walk_completed/,
  cpu/event=0x49,umask=0x4,name=dtlb_store_misses_walk_duration/,
  cpu/event=0x49,umask=0x2,name=dtlb_store_misses_walk_completed/,
  cpu/event=0x85,umask=0x4,name=itlb_misses_walk_duration/,
  cpu/event=0x85,umask=0x2,name=itlb_misses_walk_completed/
```

That works on an IvyBridge-era CPU (i5-3320M). Different CPUs may have differently-named counters, but they should at least be there in some form. You can use pmu-tools 'ocperf list' (https://github.com/andikleen/pmu-tools) to find the right counters for a given CPU.

[1] A footnote in Intel's SDM "4.10.4.2 Recommended Invalidation" says: "One execution of INVLPG is sufficient even for a page with size greater than 4 KBytes."

16.12 MTRR (Memory Type Range Register) control

Authors

- Richard Gooch <rgooch@atnf.csiro.au> - 3 Jun 1999
- Luis R. Rodriguez <mcgrof@do-not-panic.com> - April 9, 2015

16.12.1 Phasing out MTRR use

MTRR use is replaced on modern x86 hardware with PAT. Direct MTRR use by drivers on Linux is now completely phased out, device drivers should use arch_phys_wc_add() in combination with ioremap_wc() to make MTRR effective on non-PAT systems while a no-op but equally effective on PAT enabled systems.

Even if Linux does not use MTRRs directly, some x86 platform firmware may still set up MTRRs early before booting the OS. They do this as some platform firmware may still have implemented access to MTRRs which would be controlled and handled by the platform firmware directly. An example of platform use of MTRRs is through the use of SMI handlers, one case could be for fan control, the platform code would need uncachable access to some of its fan control registers. Such platform access does not need any Operating System MTRR code in place other than mtrr_type_lookup() to ensure any OS specific mapping requests are aligned with platform MTRR setup. If MTRRs are only set up by the platform firmware code though and the OS does not make any specific MTRR mapping requests mtrr_type_lookup() should always return MTRR_TYPE_INVALID.

For details refer to *PAT (Page Attribute Table)*.

Tip: On Intel P6 family processors (Pentium Pro, Pentium II and later) the Memory Type Range Registers (MTRRs) may be used to control processor access to memory ranges. This is most useful when you have a video (VGA) card on a PCI or AGP bus. Enabling write-combining allows bus write transfers to be combined into a larger transfer before bursting over the PCI/AGP bus. This can increase performance of image write operations 2.5 times or more.

The Cyrix 6x86, 6x86MX and M II processors have Address Range Registers (ARRs) which provide a similar functionality to MTRRs. For these, the ARRs are used to emulate the MTRRs.

The AMD K6-2 (stepping 8 and above) and K6-3 processors have two MTRRs. These are supported. The AMD Athlon family provide 8 Intel style MTRRs.

The Centaur C6 (WinChip) has 8 MCRs, allowing write-combining. These are supported.

The VIA Cyrix III and VIA C3 CPUs offer 8 Intel style MTRRs.

The CONFIG_MTRR option creates a /proc/mtrr file which may be used to manipulate your MTRRs. Typically the X server should use this. This should have a reasonably generic interface so that similar control registers on other processors can be easily supported.

There are two interfaces to /proc/mtrr: one is an ASCII interface which allows you to read and write. The other is an ioctl() interface. The ASCII interface is meant for administration. The ioctl() interface is meant for C programs (i.e. the X server). The interfaces are described below, with sample commands and C code.

16.12.2 Reading MTRRs from the shell

```
% cat /proc/mtrr
reg00: base=0x00000000 (   0MB), size= 128MB: write-back, count=1
reg01: base=0x08000000 ( 128MB), size=  64MB: write-back, count=1
```

Creating MTRRs from the C-shell:

```
# echo "base=0xf8000000 size=0x400000 type=write-combining" >! /proc/mtrr
```

or if you use bash:

```
# echo "base=0xf8000000 size=0x400000 type=write-combining" >| /proc/mtrr
```

And the result thereof:

```
% cat /proc/mtrr
reg00: base=0x00000000 (   0MB), size= 128MB: write-back, count=1
reg01: base=0x08000000 ( 128MB), size=  64MB: write-back, count=1
reg02: base=0xf8000000 (3968MB), size=   4MB: write-combining, count=1
```

This is for video RAM at base address 0xf8000000 and size 4 megabytes. To find out your base address, you need to look at the output of your X server, which tells you where the linear framebuffer address is. A typical line that you may get is:

```
(--) S3: PCI: 968 rev 0, Linear FB @ 0xf8000000
```

Note that you should only use the value from the X server, as it may move the framebuffer base address, so the only value you can trust is that reported by the X server.

To find out the size of your framebuffer (what, you don't actually know?), the following line will tell you:

```
(--) S3: videoram:  4096k
```

That's 4 megabytes, which is 0x400000 bytes (in hexadecimal). A patch is being written for XFree86 which will make this automatic: in other words the X server will manipulate /proc/mtrr using the ioctl() interface, so users won't have to do anything. If you use a commercial X server, lobby your vendor to add support for MTRRs.

16.12.3 Creating overlapping MTRRs

```
%echo "base=0xfb000000 size=0x1000000 type=write-combining" >/proc/mtrr
%echo "base=0xfb000000 size=0x1000 type=uncachable" >/proc/mtrr
```

And the results:

```
% cat /proc/mtrr
reg00: base=0x00000000 (   0MB), size=  64MB: write-back, count=1
reg01: base=0xfb000000 (4016MB), size=  16MB: write-combining, count=1
reg02: base=0xfb000000 (4016MB), size=   4kB: uncachable, count=1
```

Some cards (especially Voodoo Graphics boards) need this 4 kB area excluded from the beginning of the region because it is used for registers.

NOTE: You can only create type=uncachable region, if the first region that you created is type=write-combining.

16.12.4 Removing MTRRs from the C-shel

```
% echo "disable=2" >! /proc/mtrr
```

or using bash:

```
% echo "disable=2" >| /proc/mtrr
```

16.12.5 Reading MTRRs from a C program using ioctl()'s

```
/*  mtrr-show.c

    Source file for mtrr-show (example program to show MTRRs using ioctl()'s)

    Copyright (C) 1997-1998  Richard Gooch

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

    Richard Gooch may be reached by email at  rgooch@atnf.csiro.au
    The postal address is:
      Richard Gooch, c/o ATNF, P. O. Box 76, Epping, N.S.W., 2121, Australia.
*/

/*
    This program will use an ioctl() on /proc/mtrr to show the current MTRR
    settings. This is an alternative to reading /proc/mtrr.


    Written by      Richard Gooch   17-DEC-1997

    Last updated by Richard Gooch   2-MAY-1998
```

```
*/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <asm/mtrr.h>

#define TRUE 1
#define FALSE 0
#define ERRSTRING strerror (errno)

static char *mtrr_strings[MTRR_NUM_TYPES] =
{
    "uncachable",               /* 0 */
    "write-combining",          /* 1 */
    "?",                        /* 2 */
    "?",                        /* 3 */
    "write-through",            /* 4 */
    "write-protect",            /* 5 */
    "write-back",               /* 6 */
};

int main ()
{
    int fd;
    struct mtrr_gentry gentry;

    if ( ( fd = open ("/proc/mtrr", O_RDONLY, 0) ) == -1 )
    {
  if (errno == ENOENT)
  {
      fputs ("/proc/mtrr not found: not supported or you don't have a PPro?\n",
      stderr);
      exit (1);
  }
  fprintf (stderr, "Error opening /proc/mtrr\t%s\n", ERRSTRING);
  exit (2);
    }
    for (gentry.regnum = 0; ioctl (fd, MTRRIOC_GET_ENTRY, &gentry) == 0;
  ++gentry.regnum)
    {
  if (gentry.size < 1)
  {
      fprintf (stderr, "Register: %u disabled\n", gentry.regnum);
      continue;
  }
  fprintf (stderr, "Register: %u base: 0x%lx size: 0x%lx type: %s\n",
    gentry.regnum, gentry.base, gentry.size,
    mtrr_strings[gentry.type]);
    }
    if (errno == EINVAL) exit (0);
    fprintf (stderr, "Error doing ioctl(2) on /dev/mtrr\t%s\n", ERRSTRING);
    exit (3);
}   /*  End Function main  */
```

16.12.6 Creating MTRRs from a C programme using ioctl()'s

```
/*  mtrr-add.c

    Source file for mtrr-add (example programme to add an MTRRs using ioctl())

    Copyright (C) 1997-1998  Richard Gooch

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

    Richard Gooch may be reached by email at  rgooch@atnf.csiro.au
    The postal address is:
      Richard Gooch, c/o ATNF, P. O. Box 76, Epping, N.S.W., 2121, Australia.
*/

/*
    This programme will use an ioctl() on /proc/mtrr to add an entry. The first
    available mtrr is used. This is an alternative to writing /proc/mtrr.


    Written by      Richard Gooch   17-DEC-1997

    Last updated by Richard Gooch   2-MAY-1998


*/
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <errno.h>
#include <asm/mtrr.h>

#define TRUE 1
#define FALSE 0
#define ERRSTRING strerror (errno)

static char *mtrr_strings[MTRR_NUM_TYPES] =
{
    "uncachable",               /* 0 */
    "write-combining",          /* 1 */
    "?",                        /* 2 */
    "?",                        /* 3 */
    "write-through",            /* 4 */
    "write-protect",            /* 5 */
    "write-back",               /* 6 */
};

int main (int argc, char **argv)
{
    int fd;
    struct mtrr_sentry sentry;

    if (argc != 4)
    {
  fprintf (stderr, "Usage:\tmtrr-add base size type\n");
  exit (1);
    }
    sentry.base = strtoul (argv[1], NULL, 0);
    sentry.size = strtoul (argv[2], NULL, 0);
    for (sentry.type = 0; sentry.type < MTRR_NUM_TYPES; ++sentry.type)
    {
  if (strcmp (argv[3], mtrr_strings[sentry.type]) == 0) break;
    }
    if (sentry.type >= MTRR_NUM_TYPES)
    {
  fprintf (stderr, "Illegal type: \"%s\"\n", argv[3]);
  exit (2);
    }
    if ( ( fd = open ("/proc/mtrr", O_WRONLY, 0) ) == -1 )
    {
  if (errno == ENOENT)
  {
      fputs ("/proc/mtrr not found: not supported or you don't have a PPro?\n",
      stderr);
      exit (3);
  }
  fprintf (stderr, "Error opening /proc/mtrr\t%s\n", ERRSTRING);
  exit (4);
    }
    if (ioctl (fd, MTRRIOC_ADD_ENTRY, &sentry) == -1)
    {
  fprintf (stderr, "Error doing ioctl(2) on /dev/mtrr\t%s\n", ERRSTRING);
  exit (5);
    }
    fprintf (stderr, "Sleeping for 5 seconds so you can see the new entry\n");
    sleep (5);
    close (fd);
    fputs ("I've just closed /proc/mtrr so now the new entry should be gone\n",
    stderr);
}   /*  End Function main  */
```

16.13 PAT (Page Attribute Table)

x86 Page Attribute Table (PAT) allows for setting the memory attribute at the page level granularity. PAT is complementary to the MTRR settings which allows for setting of memory types over physical address ranges. However, PAT is more flexible than MTRR due to its capability to set attributes at page level and also due to the fact that there are no hardware limitations on number of such attribute settings allowed. Added flexibility comes with guidelines for not having memory type aliasing for the same physical memory with multiple virtual addresses.

PAT allows for different types of memory attributes. The most commonly used ones that will be supported at this time are:

WB	Write-back
UC	Uncached
WC	Write-combined
WT	Write-through
UC-	Uncached Minus

16.13.1 PAT APIs

There are many different APIs in the kernel that allows setting of memory attributes at the page level. In order to avoid aliasing, these interfaces should be used thoughtfully. Below is a table of interfaces available, their intended usage and their memory attribute relationships. Internally, these APIs use a reserve_memtype()/free_memtype() interface on the physical address range to avoid any aliasing.

API	RAM	ACPI,...	Reserved/Holes
ioremap	--	UC-	UC-
ioremap_cache	--	WB	WB
ioremap_uc	--	UC	UC
ioremap_wc	--	--	WC
ioremap_wt	--	--	WT
set_memory_uc, set_memory_wb	UC-	--	--
set_memory_wc, set_memory_wb	WC	--	--
set_memory_wt, set_memory_wb	WT	--	--
pci sysfs resource	--	--	UC-
pci sysfs resource_wc is IORESOURCE_PREFETCH	--	--	WC
pci proc !PCIIOC_WRITE_COMBINI	--	--	UC-
pci proc PCIIOC_WRITE_COMBINI	--	--	WC
/dev/mem read-write	--	WB/WC/UC-	WB/WC/UC-
/dev/mem mmap SYNC flag	--	UC-	UC-
/dev/mem mmap !SYNC flag and any alias to this area	--	WB/WC/UC- (from existing alias)	WB/WC/UC- (from existing alias)
/dev/mem mmap !SYNC flag no alias to this area and MTRR says WB	--	WB	WB
/dev/mem mmap !SYNC flag no alias to this area and MTRR says !WB	--	--	UC-

16.13.2 Advanced APIs for drivers

A. Exporting pages to users with remap_pfn_range, io_remap_pfn_range, vmf_insert_pfn.

Drivers wanting to export some pages to userspace do it by using mmap interface and a combination of:

1) pgprot_noncached()
2) io_remap_pfn_range() or remap_pfn_range() or vmf_insert_pfn()

With PAT support, a new API pgprot_writecombine is being added. So, drivers can continue to use the above sequence, with either pgprot_noncached() or pgprot_writecombine() in step 1, followed by step 2.

In addition, step 2 internally tracks the region as UC or WC in memtype list in order to ensure no conflicting mapping.

Note that this set of APIs only works with IO (non RAM) regions. If driver wants to export a RAM region, it has to do set_memory_uc() or set_memory_wc() as step 0 above and also track the usage of those pages and use set_memory_wb() before the page is freed to free pool.

16.13.3 MTRR effects on PAT / non-PAT systems

The following table provides the effects of using write-combining MTRRs when using ioremap*() calls on x86 for both non-PAT and PAT systems. Ideally mtrr_add() usage will be phased out in favor of arch_phys_wc_add() which will be a no-op on PAT enabled systems. The region over which a arch_phys_wc_add() is made, should already have been ioremapped with WC attributes or PAT entries, this can be done by using ioremap_wc() / set_memory_wc(). Devices which combine areas of IO memory desired to remain uncacheable with areas where write-combining is desirable should consider use of ioremap_uc() followed by set_memory_wc() to white-list effective write-combined areas. Such use is nevertheless discouraged as the effective memory type is considered implementation defined, yet this strategy can be used as last resort on devices with size-constrained regions where otherwise MTRR write-combining would otherwise not be effective.

```
====  =======  ===  =========================  =====================
MTRR  Non-PAT  PAT  Linux ioremap value        Effective memory type
====  =======  ===  =========================  =====================
      PAT                                        Non-PAT |  PAT
      |PCD                                               |
      ||PWT                                              |
      |||                                                |
WC    000      WB   _PAGE_CACHE_MODE_WB             WC   |   WC
WC    001      WC   _PAGE_CACHE_MODE_WC             WC*  |   WC
WC    010      UC-  _PAGE_CACHE_MODE_UC_MINUS       WC*  |   UC
WC    011      UC   _PAGE_CACHE_MODE_UC             UC   |   UC
====  =======  ===  =========================  =====================

(*) denotes implementation defined and is discouraged
```

Note: -- in the above table mean “Not suggested usage for the API”. Some of the --‘s are strictly enforced by the kernel. Some others are not really enforced today, but may be enforced in future.

For ioremap and pci access through /sys or /proc - The actual type returned can be more restrictive, in case of any existing aliasing for that address. For example: If there is an existing uncached mapping, a new ioremap_wc can return uncached mapping in place of write-combine requested.

set_memory_[uc|wc|wt] and set_memory_wb should be used in pairs, where driver will first make a region uc, wc or wt and switch it back to wb after use.

Over time writes to /proc/mtrr will be deprecated in favor of using PAT based interfaces. Users writing to /proc/mtrr are suggested to use above interfaces.

Drivers should use ioremap_[uc|wc] to access PCI BARs with [uc|wc] access types.

Drivers should use set_memory_[uc|wc|wt] to set access type for RAM ranges.

16.13.4 PAT debugging

With CONFIG_DEBUG_FS enabled, PAT memtype list can be examined by:

```
# mount -t debugfs debugfs /sys/kernel/debug
# cat /sys/kernel/debug/x86/pat_memtype_list
PAT memtype list:
uncached-minus @ 0x7fadf000-0x7fae0000
uncached-minus @ 0x7fb19000-0x7fb1a000
uncached-minus @ 0x7fb1a000-0x7fb1b000
uncached-minus @ 0x7fb1b000-0x7fb1c000
uncached-minus @ 0x7fb1c000-0x7fb1d000
uncached-minus @ 0x7fb1d000-0x7fb1e000
uncached-minus @ 0x7fb1e000-0x7fb25000
uncached-minus @ 0x7fb25000-0x7fb26000
uncached-minus @ 0x7fb26000-0x7fb27000
uncached-minus @ 0x7fb27000-0x7fb28000
uncached-minus @ 0x7fb28000-0x7fb2e000
uncached-minus @ 0x7fb2e000-0x7fb2f000
uncached-minus @ 0x7fb2f000-0x7fb30000
uncached-minus @ 0x7fb31000-0x7fb32000
uncached-minus @ 0x80000000-0x90000000
```

This list shows physical address ranges and various PAT settings used to access those physical address ranges.

Another, more verbose way of getting PAT related debug messages is with “debugpat” boot parameter. With this parameter, various debug messages are printed to dmesg log.

16.13.5 PAT Initialization

The following table describes how PAT is initialized under various configurations. The PAT MSR must be updated by Linux in order to support WC and WT attributes. Otherwise, the PAT MSR has the value programmed in it by the firmware. Note, Xen enables WC attribute in the PAT MSR for guests.

MTRR	PAT	Call Sequence	PAT State	PAT MSR
E	E	MTRR -> PAT init	Enabled	OS
E	D	MTRR -> PAT init	Disabled	•
D	E	MTRR -> PAT disable	Disabled	BIOS
D	D	MTRR -> PAT disable	Disabled	•
•	np/E	PAT -> PAT disable	Disabled	BIOS
•	np/D	PAT -> PAT disable	Disabled	•
E	!P/E	MTRR -> PAT init	Disabled	BIOS
D	!P/E	MTRR -> PAT disable	Disabled	BIOS
!M	!P/E	MTRR stub -> PAT disable	Disabled	BIOS

Legend

E	Feature enabled in CPU
D	Feature disabled/unsupported in CPU
np	"nopat" boot option specified
!P	CONFIG_X86_PAT option unset
!M	CONFIG_MTRR option unset
Enabled	PAT state set to enabled
Disabled	PAT state set to disabled
OS	PAT initializes PAT MSR with OS setting
BIOS	PAT keeps PAT MSR with BIOS setting

16.14 Hardware-Feedback Interface for scheduling on Intel Hardware

16.14.1 Overview

Intel has described the Hardware Feedback Interface (HFI) in the Intel 64 and IA-32 Architectures Software Developer's Manual (Intel SDM) Volume 3 Section 14.6[1].

The HFI gives the operating system a performance and energy efficiency capability data for each CPU in the system. Linux can use the information from the HFI to influence task placement decisions.

[1] https://www.intel.com/sdm

16.14.2 The Hardware Feedback Interface

The Hardware Feedback Interface provides to the operating system information about the performance and energy efficiency of each CPU in the system. Each capability is given as a unit-less quantity in the range [0-255]. Higher values indicate higher capability. Energy efficiency and performance are reported in separate capabilities. Even though on some systems these two metrics may be related, they are specified as independent capabilities in the Intel SDM.

These capabilities may change at runtime as a result of changes in the operating conditions of the system or the action of external factors. The rate at which these capabilities are updated is specific to each processor model. On some models, capabilities are set at boot time and never change. On others, capabilities may change every tens of milliseconds. For instance, a remote mechanism may be used to lower Thermal Design Power. Such change can be reflected in the HFI. Likewise, if the system needs to be throttled due to excessive heat, the HFI may reflect reduced performance on specific CPUs.

The kernel or a userspace policy daemon can use these capabilities to modify task placement decisions. For instance, if either the performance or energy capabilities of a given logical processor becomes zero, it is an indication that the hardware recommends to the operating system to not schedule any tasks on that processor for performance or energy efficiency reasons, respectively.

16.14.3 Implementation details for Linux

The infrastructure to handle thermal event interrupts has two parts. In the Local Vector Table of a CPU's local APIC, there exists a register for the Thermal Monitor Register. This register controls how interrupts are delivered to a CPU when the thermal monitor generates and interrupt. Further details can be found in the Intel SDM Vol. 3 Section 10.5[Page 655, 1].

The thermal monitor may generate interrupts per CPU or per package. The HFI generates package-level interrupts. This monitor is configured and initialized via a set of machine-specific registers. Specifically, the HFI interrupt and status are controlled via designated bits in the IA32_PACKAGE_THERM_INTERRUPT and IA32_PACKAGE_THERM_STATUS registers, respectively. There exists one HFI table per package. Further details can be found in the Intel SDM Vol. 3 Section 14.9[Page 655, 1].

The hardware issues an HFI interrupt after updating the HFI table and is ready for the operating system to consume it. CPUs receive such interrupt via the thermal entry in the Local APIC's Local Vector Table.

When servicing such interrupt, the HFI driver parses the updated table and relays the update to userspace using the thermal notification framework. Given that there may be many HFI updates every second, the updates relayed to userspace are throttled at a rate of CONFIG_HZ jiffies.

16.14.4 References

16.15 Control-flow Enforcement Technology (CET) Shadow Stack

16.15.1 CET Background

Control-flow Enforcement Technology (CET) covers several related x86 processor features that provide protection against control flow hijacking attacks. CET can protect both applications and the kernel.

CET introduces shadow stack and indirect branch tracking (IBT). A shadow stack is a secondary stack allocated from memory which cannot be directly modified by applications. When executing a CALL instruction, the processor pushes the return address to both the normal stack and the shadow stack. Upon function return, the processor pops the shadow stack copy and compares it to the normal stack copy. If the two differ, the processor raises a control-protection fault. IBT verifies indirect CALL/JMP targets are intended as marked by the compiler with 'ENDBR' opcodes. Not all CPU's have both Shadow Stack and Indirect Branch Tracking. Today in the 64-bit kernel, only userspace shadow stack and kernel IBT are supported.

16.15.2 Requirements to use Shadow Stack

To use userspace shadow stack you need HW that supports it, a kernel configured with it and userspace libraries compiled with it.

The kernel Kconfig option is X86_USER_SHADOW_STACK. When compiled in, shadow stacks can be disabled at runtime with the kernel parameter: nousershstk.

To build a user shadow stack enabled kernel, Binutils v2.29 or LLVM v6 or later are required.

At run time, /proc/cpuinfo shows CET features if the processor supports CET. "user_shstk" means that userspace shadow stack is supported on the current kernel and HW.

16.15.3 Application Enabling

An application's CET capability is marked in its ELF note and can be verified from readelf/llvm-readelf output:

```
readelf -n <application> | grep -a SHSTK
    properties: x86 feature: SHSTK
```

The kernel does not process these applications markers directly. Applications or loaders must enable CET features using the interface described in section 4. Typically this would be done in dynamic loader or static runtime objects, as is the case in GLIBC.

16.15.4 Enabling arch_prctl()'s

Elf features should be enabled by the loader using the below arch_prctl's. They are only supported in 64 bit user applications. These operate on the features on a per-thread basis. The enablement status is inherited on clone, so if the feature is enabled on the first thread, it will propagate to all the thread's in an app.

arch_prctl(ARCH_SHSTK_ENABLE, unsigned long feature)
: Enable a single feature specified in 'feature'. Can only operate on one feature at a time.

arch_prctl(ARCH_SHSTK_DISABLE, unsigned long feature)
: Disable a single feature specified in 'feature'. Can only operate on one feature at a time.

arch_prctl(ARCH_SHSTK_LOCK, unsigned long features)
: Lock in features at their current enabled or disabled status. 'features' is a mask of all features to lock. All bits set are processed, unset bits are ignored. The mask is ORed with the existing value. So any feature bits set here cannot be enabled or disabled afterwards.

arch_prctl(ARCH_SHSTK_UNLOCK, unsigned long features)
: Unlock features. 'features' is a mask of all features to unlock. All bits set are processed, unset bits are ignored. Only works via ptrace.

arch_prctl(ARCH_SHSTK_STATUS, unsigned long addr)
: Copy the currently enabled features to the address passed in addr. The features are described using the bits passed into the others in 'features'.

The return values are as follows. On success, return 0. On error, errno can be:

```
-EPERM if any of the passed feature are locked.
-ENOTSUPP if the feature is not supported by the hardware or
 kernel.
-EINVAL arguments (non existing feature, etc)
-EFAULT if could not copy information back to userspace
```

The feature's bits supported are:

```
ARCH_SHSTK_SHSTK - Shadow stack
ARCH_SHSTK_WRSS  - WRSS
```

Currently shadow stack and WRSS are supported via this interface. WRSS can only be enabled with shadow stack, and is automatically disabled if shadow stack is disabled.

16.15.5 Proc Status

To check if an application is actually running with shadow stack, the user can read the /proc/$PID/status. It will report "wrss" or "shstk" depending on what is enabled. The lines look like this:

```
x86_Thread_features: shstk wrss
x86_Thread_features_locked: shstk wrss
```

16.15.6 Implementation of the Shadow Stack

Shadow Stack Size

A task's shadow stack is allocated from memory to a fixed size of MIN(RLIMIT_STACK, 4 GB). In other words, the shadow stack is allocated to the maximum size of the normal stack, but capped to 4 GB. In the case of the clone3 syscall, there is a stack size passed in and shadow stack uses this instead of the rlimit.

Signal

The main program and its signal handlers use the same shadow stack. Because the shadow stack stores only return addresses, a large shadow stack covers the condition that both the program stack and the signal alternate stack run out.

When a signal happens, the old pre-signal state is pushed on the stack. When shadow stack is enabled, the shadow stack specific state is pushed onto the shadow stack. Today this is only the old SSP (shadow stack pointer), pushed in a special format with bit 63 set. On sigreturn this old SSP token is verified and restored by the kernel. The kernel will also push the normal restorer address to the shadow stack to help userspace avoid a shadow stack violation on the sigreturn path that goes through the restorer.

So the shadow stack signal frame format is as follows:

```
|1...old SSP| - Pointer to old pre-signal ssp in sigframe token format
                (bit 63 set to 1)
|        ...| - Other state may be added in the future
```

32 bit ABI signals are not supported in shadow stack processes. Linux prevents 32 bit execution while shadow stack is enabled by the allocating shadow stacks outside of the 32 bit address space. When execution enters 32 bit mode, either via far call or returning to userspace, a #GP is generated by the hardware which, will be delivered to the process as a segfault. When transitioning to userspace the register's state will be as if the userspace ip being returned to caused the segfault.

Fork

The shadow stack's vma has VM_SHADOW_STACK flag set; its PTEs are required to be read-only and dirty. When a shadow stack PTE is not RO and dirty, a shadow access triggers a page fault with the shadow stack access bit set in the page fault error code.

When a task forks a child, its shadow stack PTEs are copied and both the parent's and the child's shadow stack PTEs are cleared of the dirty bit. Upon the next shadow stack access, the resulting shadow stack page fault is handled by page copy/re-use.

When a pthread child is created, the kernel allocates a new shadow stack for the new thread. New shadow stack creation behaves like mmap() with respect to ASLR behavior. Similarly, on thread exit the thread's shadow stack is disabled.

Exec

On exec, shadow stack features are disabled by the kernel. At which point, userspace can choose to re-enable, or lock them.

16.16 x86 IOMMU Support

The architecture specs can be obtained from the below locations.

- Intel: http://www.intel.com/content/dam/www/public/us/en/documents/product-specifications/vt-directed-io-spec.pdf
- AMD: https://www.amd.com/system/files/TechDocs/48882_IOMMU.pdf

This guide gives a quick cheat sheet for some basic understanding.

16.16.1 Basic stuff

ACPI enumerates and lists the different IOMMUs on the platform, and device scope relationships between devices and which IOMMU controls them.

Some ACPI Keywords:

- DMAR - Intel DMA Remapping table
- DRHD - Intel DMA Remapping Hardware Unit Definition
- RMRR - Intel Reserved Memory Region Reporting Structure
- IVRS - AMD I/O Virtualization Reporting Structure
- IVDB - AMD I/O Virtualization Definition Block
- IVHD - AMD I/O Virtualization Hardware Definition

What is Intel RMRR?

There are some devices the BIOS controls, for e.g USB devices to perform PS2 emulation. The regions of memory used for these devices are marked reserved in the e820 map. When we turn on DMA translation, DMA to those regions will fail. Hence BIOS uses RMRR to specify these regions along with devices that need to access these regions. OS is expected to setup unity mappings for these regions for these devices to access these regions.

What is AMD IVRS?

The architecture defines an ACPI-compatible data structure called an I/O Virtualization Reporting Structure (IVRS) that is used to convey information related to I/O virtualization to system software. The IVRS describes the configuration and capabilities of the IOMMUs contained in the platform as well as information about the devices that each IOMMU virtualizes.

The IVRS provides information about the following:

- IOMMUs present in the platform including their capabilities and proper configuration

- System I/O topology relevant to each IOMMU
- Peripheral devices that cannot be otherwise enumerated
- Memory regions used by SMI/SMM, platform firmware, and platform hardware. These are generally exclusion ranges to be configured by system software.

16.16.2 How is an I/O Virtual Address (IOVA) generated?

Well behaved drivers call dma_map_*() calls before sending command to device that needs to perform DMA. Once DMA is completed and mapping is no longer required, driver performs dma_unmap_*() calls to unmap the region.

16.16.3 Intel Specific Notes

Graphics Problems?

If you encounter issues with graphics devices, you can try adding option intel_iommu=igfx_off to turn off the integrated graphics engine. If this fixes anything, please ensure you file a bug reporting the problem.

Some exceptions to IOVA

Interrupt ranges are not address translated, (0xfee00000 - 0xfeefffff). The same is true for peer to peer transactions. Hence we reserve the address from PCI MMIO ranges so they are not allocated for IOVA addresses.

16.16.4 AMD Specific Notes

Graphics Problems?

If you encounter issues with integrated graphics devices, you can try adding option iommu=pt to the kernel command line use a 1:1 mapping for the IOMMU. If this fixes anything, please ensure you file a bug reporting the problem.

16.16.5 Fault reporting

When errors are reported, the IOMMU signals via an interrupt. The fault reason and device that caused it is printed on the console.

16.16.6 Kernel Log Samples

Intel Boot Messages

Something like this gets printed indicating presence of DMAR tables in ACPI:

```
ACPI: DMAR (v001 A M I  OEMDMAR  0x00000001 MSFT 0x00000097) @␣
↪0x000000007f5b5ef0
```

When DMAR is being processed and initialized by ACPI, prints DMAR locations and any RMRR's processed:

```
ACPI DMAR:Host address width 36
ACPI DMAR:DRHD (flags: 0x00000000)base: 0x00000000fed90000
ACPI DMAR:DRHD (flags: 0x00000000)base: 0x00000000fed91000
ACPI DMAR:DRHD (flags: 0x00000001)base: 0x00000000fed93000
ACPI DMAR:RMRR base: 0x00000000000ed000 end: 0x00000000000effff
ACPI DMAR:RMRR base: 0x000000007f600000 end: 0x000000007fffffff
```

When DMAR is enabled for use, you will notice:

```
PCI-DMA: Using DMAR IOMMU
```

Intel Fault reporting

```
DMAR:[DMA Write] Request device [00:02.0] fault addr 6df084000
DMAR:[fault reason 05] PTE Write access is not set
DMAR:[DMA Write] Request device [00:02.0] fault addr 6df084000
DMAR:[fault reason 05] PTE Write access is not set
```

AMD Boot Messages

Something like this gets printed indicating presence of the IOMMU:

```
iommu: Default domain type: Translated
iommu: DMA domain TLB invalidation policy: lazy mode
```

AMD Fault reporting

```
AMD-Vi: Event logged [IO_PAGE_FAULT domain=0x0007 address=0xffffc02000␣
↪flags=0x0000]
AMD-Vi: Event logged [IO_PAGE_FAULT device=07:00.0 domain=0x0007␣
↪address=0xffffc02000 flags=0x0000]
```

16.17 Intel(R) TXT Overview

Intel's technology for safer computing, Intel(R) Trusted Execution Technology (Intel(R) TXT), defines platform-level enhancements that provide the building blocks for creating trusted platforms.

Intel TXT was formerly known by the code name LaGrande Technology (LT).

Intel TXT in Brief:

- Provides dynamic root of trust for measurement (DRTM)
- Data protection in case of improper shutdown
- Measurement and verification of launched environment

Intel TXT is part of the vPro(TM) brand and is also available some non-vPro systems. It is currently available on desktop systems based on the Q35, X38, Q45, and Q43 Express chipsets (e.g. Dell Optiplex 755, HP dc7800, etc.) and mobile systems based on the GM45, PM45, and GS45 Express chipsets.

For more information, see http://www.intel.com/technology/security/. This site also has a link to the Intel TXT MLE Developers Manual, which has been updated for the new released platforms.

Intel TXT has been presented at various events over the past few years, some of which are:

- **LinuxTAG 2008:**
  http://www.linuxtag.org/2008/en/conf/events/vp-donnerstag.html
- **TRUST2008:**
  http://www.trust-conference.eu/downloads/Keynote-Speakers/ 3_David-Grawrock_The-Front-Door-of-Trusted-Computing.pdf
- **IDF, Shanghai:**
  http://www.prcidf.com.cn/index_en.html
- **IDFs 2006, 2007**
  (I'm not sure if/where they are online)

16.17.1 Trusted Boot Project Overview

Trusted Boot (tboot) is an open source, pre-kernel/VMM module that uses Intel TXT to perform a measured and verified launch of an OS kernel/VMM.

It is hosted on SourceForge at http://sourceforge.net/projects/tboot. The mercurial source repo is available at http://www.bughost.org/ repos.hg/tboot.hg.

Tboot currently supports launching Xen (open source VMM/hypervisor w/ TXT support since v3.2), and now Linux kernels.

16.17.2 Value Proposition for Linux or "Why should you care?"

While there are many products and technologies that attempt to measure or protect the integrity of a running kernel, they all assume the kernel is "good" to begin with. The Integrity Measurement Architecture (IMA) and Linux Integrity Module interface are examples of such solutions.

To get trust in the initial kernel without using Intel TXT, a static root of trust must be used. This bases trust in BIOS starting at system reset and requires measurement of all code executed between system reset through the completion of the kernel boot as well as data objects used by that code. In the case of a Linux kernel, this means all of BIOS, any option ROMs, the bootloader and the boot config. In practice, this is a lot of code/data, much of which is subject to change from boot to boot (e.g. changing NICs may change option ROMs). Without reference hashes, these measurement changes are difficult to assess or confirm as benign. This process also does not provide DMA protection, memory configuration/alias checks and locks, crash protection, or policy support.

By using the hardware-based root of trust that Intel TXT provides, many of these issues can be mitigated. Specifically: many pre-launch components can be removed from the trust chain, DMA protection is provided to all launched components, a large number of platform configuration checks are performed and values locked, protection is provided for any data in the event of an improper shutdown, and there is support for policy-based execution/verification. This provides a more stable measurement and a higher assurance of system configuration and initial state than would be otherwise possible. Since the tboot project is open source, source code for almost all parts of the trust chain is available (excepting SMM and Intel-provided firmware).

16.17.3 How Does it Work?

- Tboot is an executable that is launched by the bootloader as the "kernel" (the binary the bootloader executes).
- It performs all of the work necessary to determine if the platform supports Intel TXT and, if so, executes the GETSEC[SENTER] processor instruction that initiates the dynamic root of trust.
  - If tboot determines that the system does not support Intel TXT or is not configured correctly (e.g. the SINIT AC Module was incorrect), it will directly launch the kernel with no changes to any state.
  - Tboot will output various information about its progress to the terminal, serial port, and/or an in-memory log; the output locations can be configured with a command line switch.
- The GETSEC[SENTER] instruction will return control to tboot and tboot then verifies certain aspects of the environment (e.g. TPM NV lock, e820 table does not have invalid entries, etc.).
- It will wake the APs from the special sleep state the GETSEC[SENTER] instruction had put them in and place them into a wait-for-SIPI state.
  - Because the processors will not respond to an INIT or SIPI when in the TXT environment, it is necessary to create a small VT-x guest for the APs. When they run in this guest, they will simply wait for the INIT-SIPI-SIPI sequence, which will cause VMEXITs, and then disable VT and jump to the SIPI vector. This approach seemed

  like a better choice than having to insert special code into the kernel's MP wakeup sequence.

- Tboot then applies an (optional) user-defined launch policy to verify the kernel and initrd.
  - This policy is rooted in TPM NV and is described in the tboot project. The tboot project also contains code for tools to create and provision the policy.
  - Policies are completely under user control and if not present then any kernel will be launched.
  - Policy action is flexible and can include halting on failures or simply logging them and continuing.
- Tboot adjusts the e820 table provided by the bootloader to reserve its own location in memory as well as to reserve certain other TXT-related regions.
- As part of its launch, tboot DMA protects all of RAM (using the VT-d PMRs). Thus, the kernel must be booted with 'intel_iommu=on' in order to remove this blanket protection and use VT-d's page-level protection.
- Tboot will populate a shared page with some data about itself and pass this to the Linux kernel as it transfers control.
  - The location of the shared page is passed via the boot_params struct as a physical address.
- The kernel will look for the tboot shared page address and, if it exists, map it.
- As one of the checks/protections provided by TXT, it makes a copy of the VT-d DMARs in a DMA-protected region of memory and verifies them for correctness. The VT-d code will detect if the kernel was launched with tboot and use this copy instead of the one in the ACPI table.
- At this point, tboot and TXT are out of the picture until a shutdown (S<n>)
- In order to put a system into any of the sleep states after a TXT launch, TXT must first be exited. This is to prevent attacks that attempt to crash the system to gain control on reboot and steal data left in memory.
  - The kernel will perform all of its sleep preparation and populate the shared page with the ACPI data needed to put the platform in the desired sleep state.
  - Then the kernel jumps into tboot via the vector specified in the shared page.
  - Tboot will clean up the environment and disable TXT, then use the kernel-provided ACPI information to actually place the platform into the desired sleep state.
  - In the case of S3, tboot will also register itself as the resume vector. This is necessary because it must re-establish the measured environment upon resume. Once the TXT environment has been restored, it will restore the TPM PCRs and then transfer control back to the kernel's S3 resume vector. In order to preserve system integrity across S3, the kernel provides tboot with a set of memory ranges (RAM and RESERVED_KERN in the e820 table, but not any memory that BIOS might alter over the S3 transition) that tboot will calculate a MAC (message authentication code) over and then seal with the TPM. On resume and once the measured environment has been re-established, tboot will re-calculate the MAC and verify it against the sealed value. Tboot's policy determines what happens if the verification fails. Note that the c/s 194 of tboot which has the new MAC code supports this.

That's pretty much it for TXT support.

16.17.4 Configuring the System

This code works with 32bit, 32bit PAE, and 64bit (x86_64) kernels.

In BIOS, the user must enable: TPM, TXT, VT-x, VT-d. Not all BIOSes allow these to be individually enabled/disabled and the screens in which to find them are BIOS-specific.

grub.conf needs to be modified as follows:

```
title Linux 2.6.29-tip w/ tboot
  root (hd0,0)
        kernel /tboot.gz logging=serial,vga,memory
        module /vmlinuz-2.6.29-tip intel_iommu=on ro
               root=LABEL=/ rhgb console=ttyS0,115200 3
        module /initrd-2.6.29-tip.img
        module /Q35_SINIT_17.BIN
```

The kernel option for enabling Intel TXT support is found under the Security top-level menu and is called "Enable Intel(R) Trusted Execution Technology (TXT)". It is considered EXPERIMENTAL and depends on the generic x86 support (to allow maximum flexibility in kernel build options), since the tboot code will detect whether the platform actually supports Intel TXT and thus whether any of the kernel code is executed.

The Q35_SINIT_17.BIN file is what Intel TXT refers to as an Authenticated Code Module. It is specific to the chipset in the system and can also be found on the Trusted Boot site. It is an (unencrypted) module signed by Intel that is used as part of the DRTM process to verify and configure the system. It is signed because it operates at a higher privilege level in the system than any other macrocode and its correct operation is critical to the establishment of the DRTM. The process for determining the correct SINIT ACM for a system is documented in the SINIT-guide.txt file that is on the tboot SourceForge site under the SINIT ACM downloads.

16.18 AMD Memory Encryption

Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV) are features found on AMD processors.

SME provides the ability to mark individual pages of memory as encrypted using the standard x86 page tables. A page that is marked encrypted will be automatically decrypted when read from DRAM and encrypted when written to DRAM. SME can therefore be used to protect the contents of DRAM from physical attacks on the system.

SEV enables running encrypted virtual machines (VMs) in which the code and data of the guest VM are secured so that a decrypted version is available only within the VM itself. SEV guest VMs have the concept of private and shared memory. Private memory is encrypted with the guest-specific key, while shared memory may be encrypted with hypervisor key. When SME is enabled, the hypervisor key is the same key which is used in SME.

A page is encrypted when a page table entry has the encryption bit set (see below on how to determine its position). The encryption bit can also be specified in the cr3 register, allowing the PGD table to be encrypted. Each successive level of page tables can also be encrypted by setting the encryption bit in the page table entry that points to the next table. This allows the full page table hierarchy to be encrypted. Note, this means that just because the encryption bit is set in cr3, doesn't imply the full hierarchy is encrypted. Each page table entry in the hierarchy needs to have the encryption bit set to achieve that. So, theoretically, you could have the encryption bit set in cr3 so that the PGD is encrypted, but not set the encryption bit in the PGD entry for a PUD which results in the PUD pointed to by that entry to not be encrypted.

When SEV is enabled, instruction pages and guest page tables are always treated as private. All the DMA operations inside the guest must be performed on shared memory. Since the memory encryption bit is controlled by the guest OS when it is operating in 64-bit or 32-bit PAE mode, in all other modes the SEV hardware forces the memory encryption bit to 1.

Support for SME and SEV can be determined through the CPUID instruction. The CPUID function 0x8000001f reports information related to SME:

```
0x8000001f[eax]:
        Bit[0] indicates support for SME
        Bit[1] indicates support for SEV
0x8000001f[ebx]:
        Bits[5:0]  pagetable bit number used to activate memory
                   encryption
        Bits[11:6] reduction in physical address space, in bits, when
                   memory encryption is enabled (this only affects
                   system physical addresses, not guest physical
                   addresses)
```

If support for SME is present, MSR 0xc00100010 (MSR_AMD64_SYSCFG) can be used to determine if SME is enabled and/or to enable memory encryption:

```
0xc0010010:
        Bit[23]   0 = memory encryption features are disabled
                  1 = memory encryption features are enabled
```

If SEV is supported, MSR 0xc0010131 (MSR_AMD64_SEV) can be used to determine if SEV is active:

```
0xc0010131:
        Bit[0]    0 = memory encryption is not active
                  1 = memory encryption is active
```

Linux relies on BIOS to set this bit if BIOS has determined that the reduction in the physical address space as a result of enabling memory encryption (see CPUID information above) will not conflict with the address space resource requirements for the system. If this bit is not set upon Linux startup then Linux itself will not set it and memory encryption will not be possible.

The state of SME in the Linux kernel can be documented as follows:

- Supported: The CPU supports SME (determined through CPUID instruction).
- Enabled: Supported and bit 23 of MSR_AMD64_SYSCFG is set.
- Active: Supported, Enabled and the Linux kernel is actively applying the encryption bit to page table entries (the SME mask in the kernel is non-zero).

SME can also be enabled and activated in the BIOS. If SME is enabled and activated in the

BIOS, then all memory accesses will be encrypted and it will not be necessary to activate the Linux memory encryption support.

If the BIOS merely enables SME (sets bit 23 of the MSR_AMD64_SYSCFG), then memory encryption can be enabled by supplying mem_encrypt=on on the kernel command line. However, if BIOS does not enable SME, then Linux will not be able to activate memory encryption, even if configured to do so by default or the mem_encrypt=on command line parameter is specified.

16.18.1 Secure Nested Paging (SNP)

SEV-SNP introduces new features (SEV_FEATURES[1:63]) which can be enabled by the hypervisor for security enhancements. Some of these features need guest side implementation to function correctly. The below table lists the expected guest behavior with various possible scenarios of guest/hypervisor SNP feature support.

Feature Enabled by the HV	Guest needs implementation	Guest has implementation	Guest boot behaviour
No	No	No	Boot
No	Yes	No	Boot
No	Yes	Yes	Boot
Yes	No	No	Boot with feature enabled
Yes	Yes	No	Graceful boot failure
Yes	Yes	Yes	Boot with feature enabled

More details in AMD64 APM[1] Vol 2: 15.34.10 SEV_STATUS MSR

[1] https://www.amd.com/system/files/TechDocs/40332.pdf

16.19 AMD HSMP interface

Newer Fam19h EPYC server line of processors from AMD support system management functionality via HSMP (Host System Management Port).

The Host System Management Port (HSMP) is an interface to provide OS-level software with access to system management functions via a set of mailbox registers.

More details on the interface can be found in chapter "7 Host System Management Port (HSMP)" of the family/model PPR Eg: https://www.amd.com/system/files/TechDocs/55898_B1_pub_0.50.zip

HSMP interface is supported on EPYC server CPU models only.

16.19.1 HSMP device

amd_hsmp driver under the drivers/platforms/x86/ creates miscdevice /dev/hsmp to let user space programs run hsmp mailbox commands.

$ ls -al /dev/hsmp crw-r--r-- 1 root root 10, 123 Jan 21 21:41 /dev/hsmp

Characteristics of the dev node:

- Write mode is used for running set/configure commands
- Read mode is used for running get/status monitor commands

Access restrictions:

- Only root user is allowed to open the file in write mode.
- The file can be opened in read mode by all the users.

In-kernel integration:

- Other subsystems in the kernel can use the exported transport function hsmp_send_message().
- Locking across callers is taken care by the driver.

16.19.2 An example

To access hsmp device from a C program. First, you need to include the headers:

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

Which defines the supported messages/message IDs.

Next thing, open the device file, as follows:

```
int file;

file = open("/dev/hsmp", O_RDWR);
if (file < 0) {
  /* ERROR HANDLING; you can check errno to see what went wrong */
  exit(1);
}
```

The following IOCTL is defined:

ioctl(file, HSMP_IOCTL_CMD, struct hsmp_message *msg)
The argument is a pointer to a:

```
struct hsmp_message {
    __u32   msg_id;                         /* Message ID */
    __u16   num_args;                       /* Number of input argument␣
↪words in message */
    __u16   response_sz;                    /* Number of expected output/
↪response words */
    __u32   args[HSMP_MAX_MSG_LEN];         /* argument/response buffer */
    __u16   sock_ind;                       /* socket number */
};
```

The ioctl would return a non-zero on failure; you can read errno to see what happened. The transaction returns 0 on success.

More details on the interface and message definitions can be found in chapter "7 Host System Management Port (HSMP)" of the respective family/model PPR eg: https://www.amd.com/system/files/TechDocs/55898_B1_pub_0.50.zip

User space C-APIs are made available by linking against the esmi library, which is provided by the E-SMS project https://developer.amd.com/e-sms/. See: https://github.com/amd/esmi_ib_library

16.20 Intel Trust Domain Extensions (TDX)

Intel's Trust Domain Extensions (TDX) protect confidential guest VMs from the host and physical attacks by isolating the guest register state and by encrypting the guest memory. In TDX, a special module running in a special mode sits between the host and the guest and manages the guest/host separation.

Since the host cannot directly access guest registers or memory, much normal functionality of a hypervisor must be moved into the guest. This is implemented using a Virtualization Exception (#VE) that is handled by the guest kernel. A #VE is handled entirely inside the guest kernel, but some require the hypervisor to be consulted.

TDX includes new hypercall-like mechanisms for communicating from the guest to the hypervisor or the TDX module.

16.20.1 New TDX Exceptions

TDX guests behave differently from bare-metal and traditional VMX guests. In TDX guests, otherwise normal instructions or memory accesses can cause #VE or #GP exceptions.

Instructions marked with an '*' conditionally cause exceptions. The details for these instructions are discussed below.

Instruction-based #VE

- Port I/O (INS, OUTS, IN, OUT)
- HLT
- MONITOR, MWAIT
- WBINVD, INVD
- VMCALL
- RDMSR*,WRMSR*
- CPUID*

Instruction-based #GP

- All VMX instructions: INVEPT, INVVPID, VMCLEAR, VMFUNC, VMLAUNCH, VMPTRLD, VMPTRST, VMREAD, VMRESUME, VMWRITE, VMXOFF, VMXON
- ENCLS, ENCLU
- GETSEC
- RSM
- ENQCMD
- RDMSR*,WRMSR*

RDMSR/WRMSR Behavior

MSR access behavior falls into three categories:

- #GP generated
- #VE generated
- "Just works"

In general, the #GP MSRs should not be used in guests. Their use likely indicates a bug in the guest. The guest may try to handle the #GP with a hypercall but it is unlikely to succeed.

The #VE MSRs are typically able to be handled by the hypervisor. Guests can make a hypercall to the hypervisor to handle the #VE.

The "just works" MSRs do not need any special guest handling. They might be implemented by directly passing through the MSR to the hardware or by trapping and handling in the TDX module. Other than possibly being slow, these MSRs appear to function just as they would on bare metal.

CPUID Behavior

For some CPUID leaves and sub-leaves, the virtualized bit fields of CPUID return values (in guest EAX/EBX/ECX/EDX) are configurable by the hypervisor. For such cases, the Intel TDX module architecture defines two virtualization types:

- Bit fields for which the hypervisor controls the value seen by the guest TD.
- Bit fields for which the hypervisor configures the value such that the guest TD either sees their native value or a value of 0. For these bit fields, the hypervisor can mask off the native values, but it can not turn *on* values.

A #VE is generated for CPUID leaves and sub-leaves that the TDX module does not know how to handle. The guest kernel may ask the hypervisor for the value with a hypercall.

16.20.2 #VE on Memory Accesses

There are essentially two classes of TDX memory: private and shared. Private memory receives full TDX protections. Its content is protected against access from the hypervisor. Shared memory is expected to be shared between guest and hypervisor and does not receive full TDX protections.

A TD guest is in control of whether its memory accesses are treated as private or shared. It selects the behavior with a bit in its page table entries. This helps ensure that a guest does not place sensitive information in shared memory, exposing it to the untrusted hypervisor.

#VE on Shared Memory

Access to shared mappings can cause a #VE. The hypervisor ultimately controls whether a shared memory access causes a #VE, so the guest must be careful to only reference shared pages it can safely handle a #VE. For instance, the guest should be careful not to access shared memory in the #VE handler before it reads the #VE info structure (TDG.VP.VEINFO.GET).

Shared mapping content is entirely controlled by the hypervisor. The guest should only use shared mappings for communicating with the hypervisor. Shared mappings must never be used for sensitive memory content like kernel stacks. A good rule of thumb is that hypervisor-shared memory should be treated the same as memory mapped to userspace. Both the hypervisor and userspace are completely untrusted.

MMIO for virtual devices is implemented as shared memory. The guest must be careful not to access device MMIO regions unless it is also prepared to handle a #VE.

#VE on Private Pages

An access to private mappings can also cause a #VE. Since all kernel memory is also private memory, the kernel might theoretically need to handle a #VE on arbitrary kernel memory accesses. This is not feasible, so TDX guests ensure that all guest memory has been “accepted” before memory is used by the kernel.

A modest amount of memory (typically 512M) is pre-accepted by the firmware before the kernel runs to ensure that the kernel can start up without being subjected to a #VE.

The hypervisor is permitted to unilaterally move accepted pages to a “blocked” state. However, if it does this, page access will not generate a #VE. It will, instead, cause a “TD Exit” where the hypervisor is required to handle the exception.

16.20.3 Linux #VE handler

Just like page faults or #GP’s, #VE exceptions can be either handled or be fatal. Typically, an unhandled userspace #VE results in a SIGSEGV. An unhandled kernel #VE results in an oops.

Handling nested exceptions on x86 is typically nasty business. A #VE could be interrupted by an NMI which triggers another #VE and hilarity ensues. The TDX #VE architecture anticipated this scenario and includes a feature to make it slightly less nasty.

During #VE handling, the TDX module ensures that all interrupts (including NMIs) are blocked. The block remains in place until the guest makes a TDG.VP.VEINFO.GET TDCALL. This allows the guest to control when interrupts or a new #VE can be delivered.

However, the guest kernel must still be careful to avoid potential #VE-triggering actions (discussed above) while this block is in place. While the block is in place, any #VE is elevated to a double fault (#DF) which is not recoverable.

16.20.4 MMIO handling

In non-TDX VMs, MMIO is usually implemented by giving a guest access to a mapping which will cause a VMEXIT on access, and then the hypervisor emulates the access. That is not possible in TDX guests because VMEXIT will expose the register state to the host. TDX guests don't trust the host and can't have their state exposed to the host.

In TDX, MMIO regions typically trigger a #VE exception in the guest. The guest #VE handler then emulates the MMIO instruction inside the guest and converts it into a controlled TDCALL to the host, rather than exposing guest state to the host.

MMIO addresses on x86 are just special physical addresses. They can theoretically be accessed with any instruction that accesses memory. However, the kernel instruction decoding method is limited. It is only designed to decode instructions like those generated by io.h macros.

MMIO access via other means (like structure overlays) may result in an oops.

16.20.5 Shared Memory Conversions

All TDX guest memory starts out as private at boot. This memory can not be accessed by the hypervisor. However, some kernel users like device drivers might have a need to share data with the hypervisor. To do this, memory must be converted between shared and private. This can be accomplished using some existing memory encryption helpers:

- set_memory_decrypted() converts a range of pages to shared.
- set_memory_encrypted() converts memory back to private.

Device drivers are the primary user of shared memory, but there's no need to touch every driver. DMA buffers and ioremap() do the conversions automatically.

TDX uses SWIOTLB for most DMA allocations. The SWIOTLB buffer is converted to shared on boot.

For coherent DMA allocation, the DMA buffer gets converted on the allocation. Check force_dma_unencrypted() for details.

16.20.6 Attestation

Attestation is used to verify the TDX guest trustworthiness to other entities before provisioning secrets to the guest. For example, a key server may want to use attestation to verify that the guest is the desired one before releasing the encryption keys to mount the encrypted rootfs or a secondary drive.

The TDX module records the state of the TDX guest in various stages of the guest boot process using the build time measurement register (MRTD) and runtime measurement registers (RTMR). Measurements related to the guest initial configuration and firmware image are recorded in the MRTD register. Measurements related to initial state, kernel image, firmware image, command line options, initrd, ACPI tables, etc are recorded in RTMR registers. For more details, as an example, please refer to TDX Virtual Firmware design specification, section titled "TD Measurement". At TDX guest runtime, the attestation process is used to attest to these measurements.

The attestation process consists of two steps: TDREPORT generation and Quote generation.

TDX guest uses TDCALL[TDG.MR.REPORT] to get the TDREPORT (TDREPORT_STRUCT) from the TDX module. TDREPORT is a fixed-size data structure generated by the TDX module which contains guest-specific information (such as build and boot measurements), platform security version, and the MAC to protect the integrity of the TDREPORT. A user-provided 64-Byte REPORTDATA is used as input and included in the TDREPORT. Typically it can be some nonce provided by attestation service so the TDREPORT can be verified uniquely. More details about the TDREPORT can be found in Intel TDX Module specification, section titled "TDG.MR.REPORT Leaf".

After getting the TDREPORT, the second step of the attestation process is to send it to the Quoting Enclave (QE) to generate the Quote. TDREPORT by design can only be verified on the local platform as the MAC key is bound to the platform. To support remote verification of the TDREPORT, TDX leverages Intel SGX Quoting Enclave to verify the TDREPORT locally and convert it to a remotely verifiable Quote. Method of sending TDREPORT to QE is implementation specific. Attestation software can choose whatever communication channel available (i.e. vsock or TCP/IP) to send the TDREPORT to QE and receive the Quote.

16.20.7 References

TDX reference material is collected here:

https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html

16.21 Page Table Isolation (PTI)

16.21.1 Overview

Page Table Isolation (pti, previously known as KAISER[1]) is a countermeasure against attacks on the shared user/kernel address space such as the "Meltdown" approach[2].

To mitigate this class of attacks, we create an independent set of page tables for use only when running userspace applications. When the kernel is entered via syscalls, interrupts or exceptions, the page tables are switched to the full "kernel" copy. When the system switches back to user mode, the user copy is used again.

The userspace page tables contain only a minimal amount of kernel data: only what is needed to enter/exit the kernel such as the entry/exit functions themselves and the interrupt descriptor table (IDT). There are a few strictly unnecessary things that get mapped such as the first C function when entering an interrupt (see comments in pti.c).

This approach helps to ensure that side-channel attacks leveraging the paging structures do not function when PTI is enabled. It can be enabled by setting CONFIG_PAGE_TABLE_ISOLATION=y at compile time. Once enabled at compile-time, it can be disabled at boot with the 'nopti' or 'pti=' kernel parameters (see kernel-parameters.txt).

[1] https://gruss.cc/files/kaiser.pdf

[2] https://meltdownattack.com/meltdown.pdf

16.21.2 Page Table Management

When PTI is enabled, the kernel manages two sets of page tables. The first set is very similar to the single set which is present in kernels without PTI. This includes a complete mapping of userspace that the kernel can use for things like copy_to_user().

Although _complete_, the user portion of the kernel page tables is crippled by setting the NX bit in the top level. This ensures that any missed kernel->user CR3 switch will immediately crash userspace upon executing its first instruction.

The userspace page tables map only the kernel data needed to enter and exit the kernel. This data is entirely contained in the 'struct cpu_entry_area' structure which is placed in the fixmap which gives each CPU's copy of the area a compile-time-fixed virtual address.

For new userspace mappings, the kernel makes the entries in its page tables like normal. The only difference is when the kernel makes entries in the top (PGD) level. In addition to setting the entry in the main kernel PGD, a copy of the entry is made in the userspace page tables' PGD.

This sharing at the PGD level also inherently shares all the lower layers of the page tables. This leaves a single, shared set of userspace page tables to manage. One PTE to lock, one set of accessed bits, dirty bits, etc...

16.21.3 Overhead

Protection against side-channel attacks is important. But, this protection comes at a cost:

1. Increased Memory Use

a. Each process now needs an order-1 PGD instead of order-0. (Consumes an additional 4k per process).

b. The 'cpu_entry_area' structure must be 2MB in size and 2MB aligned so that it can be mapped by setting a single PMD entry. This consumes nearly 2MB of RAM once the kernel is decompressed, but no space in the kernel image itself.

2. Runtime Cost

a. CR3 manipulation to switch between the page table copies must be done at interrupt, syscall, and exception entry and exit (it can be skipped when the kernel is interrupted, though.) Moves to CR3 are on the order of a hundred cycles, and are required at every entry and exit.

b. A "trampoline" must be used for SYSCALL entry. This trampoline depends on a smaller set of resources than the non-PTI SYSCALL entry code, so requires mapping fewer things into the userspace page tables. The downside is that stacks must be switched at entry time.

c. Global pages are disabled for all kernel structures not mapped into both kernel and userspace page tables. This feature of the MMU allows different processes to share TLB entries mapping the kernel. Losing the feature means more TLB misses after a context switch. The actual loss of performance is very small, however, never exceeding 1%.

d. Process Context IDentifiers (PCID) is a CPU feature that allows us to skip flushing the entire TLB when switching page tables by setting a special bit in CR3 when the page tables are changed. This makes switching the page tables (at context switch, or kernel entry/exit) cheaper. But, on systems with PCID support, the context switch code must flush both the user and kernel entries out of the TLB. The user PCID TLB flush is deferred until the exit to userspace, minimizing the cost. See intel.com/sdm for the gory PCID/INVPCID details.

e. The userspace page tables must be populated for each new process. Even without PTI, the shared kernel mappings are created by copying top-level (PGD) entries into each new process. But, with PTI, there are now *two* kernel mappings: one in the kernel page tables that maps everything and one for the entry/exit structures. At fork(), we need to copy both.

f. In addition to the fork()-time copying, there must also be an update to the userspace PGD any time a set_pgd() is done on a PGD used to map userspace. This ensures that the kernel and userspace copies always map the same userspace memory.

g. On systems without PCID support, each CR3 write flushes the entire TLB. That means that each syscall, interrupt or exception flushes the TLB.

h. INVPCID is a TLB-flushing instruction which allows flushing of TLB entries for non-current PCIDs. Some systems support PCIDs, but do not support INVPCID. On these systems, addresses can only be flushed from the TLB for the current PCID. When flushing a kernel address, we need to flush all PCIDs, so a single kernel address flush will require a TLB-flushing CR3 write upon the next use of every PCID.

16.21.4 Possible Future Work

1. We can be more careful about not actually writing to CR3 unless its value is actually changed.
2. Allow PTI to be enabled/disabled at runtime in addition to the boot-time switching.

16.21.5 Testing

To test stability of PTI, the following test procedure is recommended, ideally doing all of these in parallel:

1. Set CONFIG_DEBUG_ENTRY=y
2. Run several copies of all of the tools/testing/selftests/x86/ tests (excluding MPX and protection_keys) in a loop on multiple CPUs for several minutes. These tests frequently uncover corner cases in the kernel entry code. In general, old kernels might cause these tests themselves to crash, but they should never crash the kernel.
3. Run the 'perf' tool in a mode (top or record) that generates many frequent performance monitoring non-maskable interrupts (see "NMI" in /proc/interrupts). This exercises the NMI entry/exit code which is known to trigger bugs in code paths that did not expect to be interrupted, including nested NMIs. Using "-c" boosts the rate of NMIs, and using two -c with separate counters encourages nested NMIs and less deterministic behavior.

   ```
   while true; do perf record -c 10000 -e instructions,cycles -a sleep 10;␣
   ↪done
   ```

4. Launch a KVM virtual machine.
5. Run 32-bit binaries on systems supporting the SYSCALL instruction. This has been a lightly-tested code path and needs extra scrutiny.

16.21.6 Debugging

Bugs in PTI cause a few different signatures of crashes that are worth noting here.

- Failures of the selftests/x86 code. Usually a bug in one of the more obscure corners of entry_64.S
- Crashes in early boot, especially around CPU bringup. Bugs in the trampoline code or mappings cause these.
- Crashes at the first interrupt. Caused by bugs in entry_64.S, like screwing up a page table switch. Also caused by incorrectly mapping the IRQ handler entry code.
- Crashes at the first NMI. The NMI code is separate from main interrupt handlers and can have bugs that do not affect normal interrupts. Also caused by incorrectly mapping NMI code. NMIs that interrupt the entry code must be very careful and can be the cause of crashes that show up when running perf.
- Kernel crashes at the first exit to userspace. entry_64.S bugs, or failing to map some of the exit code.
- Crashes at first interrupt that interrupts userspace. The paths in entry_64.S that return to userspace are sometimes separate from the ones that return to the kernel.
- Double faults: overflowing the kernel stack because of page faults upon page faults. Caused by touching non-pti-mapped data in the entry code, or forgetting to switch to kernel CR3 before calling into C functions which are not pti-mapped.
- Userspace segfaults early in boot, sometimes manifesting as mount(8) failing to mount the rootfs. These have tended to be TLB invalidation issues. Usually invalidating the wrong PCID, or otherwise missing an invalidation.

16.22 Microarchitectural Data Sampling (MDS) mitigation

16.22.1 Overview

Microarchitectural Data Sampling (MDS) is a family of side channel attacks on internal buffers in Intel CPUs. The variants are:

- Microarchitectural Store Buffer Data Sampling (MSBDS) (CVE-2018-12126)
- Microarchitectural Fill Buffer Data Sampling (MFBDS) (CVE-2018-12130)
- Microarchitectural Load Port Data Sampling (MLPDS) (CVE-2018-12127)
- Microarchitectural Data Sampling Uncacheable Memory (MDSUM) (CVE-2019-11091)

MSBDS leaks Store Buffer Entries which can be speculatively forwarded to a dependent load (store-to-load forwarding) as an optimization. The forward can also happen to a faulting or assisting load operation for a different memory address, which can be exploited under certain conditions. Store buffers are partitioned between Hyper-Threads so cross thread forwarding is not possible. But if a thread enters or exits a sleep state the store buffer is repartitioned which can expose data from one thread to the other.

MFBDS leaks Fill Buffer Entries. Fill buffers are used internally to manage L1 miss situations and to hold data which is returned or sent in response to a memory or I/O operation. Fill buffers can forward data to a load operation and also write data to the cache. When the fill buffer is deallocated it can retain the stale data of the preceding operations which can then be forwarded to a faulting or assisting load operation, which can be exploited under certain conditions. Fill buffers are shared between Hyper-Threads so cross thread leakage is possible.

MLPDS leaks Load Port Data. Load ports are used to perform load operations from memory or I/O. The received data is then forwarded to the register file or a subsequent operation. In some implementations the Load Port can contain stale data from a previous operation which can be forwarded to faulting or assisting loads under certain conditions, which again can be exploited eventually. Load ports are shared between Hyper-Threads so cross thread leakage is possible.

MDSUM is a special case of MSBDS, MFBDS and MLPDS. An uncacheable load from memory that takes a fault or assist can leave data in a microarchitectural structure that may later be observed using one of the same methods used by MSBDS, MFBDS or MLPDS.

16.22.2 Exposure assumptions

It is assumed that attack code resides in user space or in a guest with one exception. The rationale behind this assumption is that the code construct needed for exploiting MDS requires:

- to control the load to trigger a fault or assist
- to have a disclosure gadget which exposes the speculatively accessed data for consumption through a side channel.
- to control the pointer through which the disclosure gadget exposes the data

The existence of such a construct in the kernel cannot be excluded with 100% certainty, but the complexity involved makes it extremely unlikely.

There is one exception, which is untrusted BPF. The functionality of untrusted BPF is limited, but it needs to be thoroughly investigated whether it can be used to create such a construct.

16.22.3 Mitigation strategy

All variants have the same mitigation strategy at least for the single CPU thread case (SMT off): Force the CPU to clear the affected buffers.

This is achieved by using the otherwise unused and obsolete VERW instruction in combination with a microcode update. The microcode clears the affected CPU buffers when the VERW instruction is executed.

For virtualization there are two ways to achieve CPU buffer clearing. Either the modified VERW instruction or via the L1D Flush command. The latter is issued when L1TF mitigation is enabled so the extra VERW can be avoided. If the CPU is not affected by L1TF then VERW needs to be issued.

If the VERW instruction with the supplied segment selector argument is executed on a CPU without the microcode update there is no side effect other than a small number of pointlessly wasted CPU cycles.

This does not protect against cross Hyper-Thread attacks except for MSBDS which is only exploitable cross Hyper-thread when one of the Hyper-Threads enters a C-state.

The kernel provides a function to invoke the buffer clearing:

    mds_clear_cpu_buffers()

Also macro CLEAR_CPU_BUFFERS can be used in ASM late in exit-to-user path. Other than CFLAGS.ZF, this macro doesn't clobber any registers.

The mitigation is invoked on kernel/userspace, hypervisor/guest and C-state (idle) transitions.

As a special quirk to address virtualization scenarios where the host has the microcode updated, but the hypervisor does not (yet) expose the MD_CLEAR CPUID bit to guests, the kernel issues the VERW instruction in the hope that it might actually clear the buffers. The state is reflected accordingly.

According to current knowledge additional mitigations inside the kernel itself are not required because the necessary gadgets to expose the leaked data cannot be controlled in a way which allows exploitation from malicious user space or VM guests.

16.22.4 Kernel internal mitigation modes

off	Mitigation is disabled. Either the CPU is not affected or mds=off is supplied on the kernel command line
full	Mitigation is enabled. CPU is affected and MD_CLEAR is advertised in CPUID.
vmwerv	Mitigation is enabled. CPU is affected and MD_CLEAR is not advertised in CPUID. That is mainly for virtualization scenarios where the host has the updated microcode but the hypervisor does not expose MD_CLEAR in CPUID. It's a best effort approach without guarantee.

If the CPU is affected and mds=off is not supplied on the kernel command line then the kernel selects the appropriate mitigation mode depending on the availability of the MD_CLEAR CPUID bit.

16.22.5 Mitigation points

1. Return to user space

When transitioning from kernel to user space the CPU buffers are flushed on affected CPUs when the mitigation is not disabled on the kernel command line. The mitigation is enabled through the feature flag X86_FEATURE_CLEAR_CPU_BUF.

The mitigation is invoked just before transitioning to userspace after user registers are restored. This is done to minimize the window in which kernel data could be accessed after VERW e.g. via an NMI after VERW.

Corner case not handled Interrupts returning to kernel don't clear CPUs buffers since the exit-to-user path is expected to do that anyways. But, there could be a case when an NMI is generated in kernel after the exit-to-user path has cleared the buffers. This case is not handled and NMI returning to kernel don't clear CPU buffers because:

1. It is rare to get an NMI after VERW, but before returning to userspace.
2. For an unprivileged user, there is no known way to make that NMI less rare or target it.
3. It would take a large number of these precisely-timed NMIs to mount an actual attack. There's presumably not enough bandwidth.

4. The NMI in question occurs after a VERW, i.e. when user state is restored and most interesting data is already scrubbed. Whats left is only the data that NMI touches, and that may or may not be of any interest.

2. C-State transition

When a CPU goes idle and enters a C-State the CPU buffers need to be cleared on affected CPUs when SMT is active. This addresses the repartitioning of the store buffer when one of the Hyper-Threads enters a C-State.

When SMT is inactive, i.e. either the CPU does not support it or all sibling threads are offline CPU buffer clearing is not required.

The idle clearing is enabled on CPUs which are only affected by MSBDS and not by any other MDS variant. The other MDS variants cannot be protected against cross Hyper-Thread attacks because the Fill Buffer and the Load Ports are shared. So on CPUs affected by other variants, the idle clearing would be a window dressing exercise and is therefore not activated.

The invocation is controlled by the static key mds_idle_clear which is switched depending on the chosen mitigation mode and the SMT state of the system.

The buffer clear is only invoked before entering the C-State to prevent that stale data from the idling CPU from spilling to the Hyper-Thread sibling after the store buffer got repartitioned and all entries are available to the non idle sibling.

When coming out of idle the store buffer is partitioned again so each sibling has half of it available. The back from idle CPU could be then speculatively exposed to contents of the sibling. The buffers are flushed either on exit to user space or on VMENTER so malicious code in user space or the guest cannot speculatively access them.

The mitigation is hooked into all variants of halt()/mwait(), but does not cover the legacy ACPI IO-Port mechanism because the ACPI idle driver has been superseded by the intel_idle driver around 2010 and is preferred on all affected CPUs which are expected to gain the MD_CLEAR functionality in microcode. Aside of that the IO-Port mechanism is a legacy interface which is only used on older systems which are either not affected or do not receive microcode updates anymore.

16.23 The Linux Microcode Loader

Authors

- Fenghua Yu <fenghua.yu@intel.com>
- Borislav Petkov <bp@suse.de>
- Ashok Raj <ashok.raj@intel.com>

The kernel has a x86 microcode loading facility which is supposed to provide microcode loading methods in the OS. Potential use cases are updating the microcode on platforms beyond the OEM End-Of-Life support, and updating the microcode on long-running systems without rebooting.

The loader supports three loading methods:

16.23.1 Early load microcode

The kernel can update microcode very early during boot. Loading microcode early can fix CPU issues before they are observed during kernel boot time.

The microcode is stored in an initrd file. During boot, it is read from it and loaded into the CPU cores.

The format of the combined initrd image is microcode in (uncompressed) cpio format followed by the (possibly compressed) initrd image. The loader parses the combined initrd image during boot.

The microcode files in cpio name space are:

on Intel:
kernel/x86/microcode/GenuineIntel.bin

on AMD :
kernel/x86/microcode/AuthenticAMD.bin

During BSP (BootStrapping Processor) boot (pre-SMP), the kernel scans the microcode file in the initrd. If microcode matching the CPU is found, it will be applied in the BSP and later on in all APs (Application Processors).

The loader also saves the matching microcode for the CPU in memory. Thus, the cached microcode patch is applied when CPUs resume from a sleep state.

Here's a crude example how to prepare an initrd with microcode (this is normally done automatically by the distribution, when recreating the initrd, so you don't really have to do it yourself. It is documented here for future reference only).

```
#!/bin/bash

if [ -z "$1" ]; then
    echo "You need to supply an initrd file"
    exit 1
fi

INITRD="$1"

DSTDIR=kernel/x86/microcode
TMPDIR=/tmp/initrd

rm -rf $TMPDIR

mkdir $TMPDIR
cd $TMPDIR
mkdir -p $DSTDIR

if [ -d /lib/firmware/amd-ucode ]; then
        cat /lib/firmware/amd-ucode/microcode_amd*.bin > $DSTDIR/AuthenticAMD.
↪bin
fi

if [ -d /lib/firmware/intel-ucode ]; then
```

```
        cat /lib/firmware/intel-ucode/* > $DSTDIR/GenuineIntel.bin
fi

find . | cpio -o -H newc >../ucode.cpio
cd ..
mv $INITRD $INITRD.orig
cat ucode.cpio $INITRD.orig > $INITRD

rm -rf $TMPDIR
```

The system needs to have the microcode packages installed into /lib/firmware or you need to fixup the paths above if yours are somewhere else and/or you've downloaded them directly from the processor vendor's site.

16.23.2 Late loading

You simply install the microcode packages your distro supplies and run:

```
# echo 1 > /sys/devices/system/cpu/microcode/reload
```

as root.

The loading mechanism looks for microcode blobs in /lib/firmware/{intel-ucode,amd-ucode}. The default distro installation packages already put them there.

Since kernel 5.19, late loading is not enabled by default.

The /dev/cpu/microcode method has been removed in 5.19.

16.23.3 Why is late loading dangerous?

Synchronizing all CPUs

The microcode engine which receives the microcode update is shared between the two logical threads in a SMT system. Therefore, when the update is executed on one SMT thread of the core, the sibling "automatically" gets the update.

Since the microcode can "simulate" MSRs too, while the microcode update is in progress, those simulated MSRs transiently cease to exist. This can result in unpredictable results if the SMT sibling thread happens to be in the middle of an access to such an MSR. The usual observation is that such MSR accesses cause #GPs to be raised to signal that former are not present.

The disappearing MSRs are just one common issue which is being observed. Any other instruction that's being patched and gets concurrently executed by the other SMT sibling, can also result in similar, unpredictable behavior.

To eliminate this case, a stop_machine()-based CPU synchronization was introduced as a way to guarantee that all logical CPUs will not execute any code but just wait in a spin loop, polling an atomic variable.

While this took care of device or external interrupts, IPIs including LVT ones, such as CMCI etc, it cannot address other special interrupts that can't be shut off. Those are Machine Check (#MC), System Management (#SMI) and Non-Maskable interrupts (#NMI).

Machine Checks

Machine Checks (#MC) are non-maskable. There are two kinds of MCEs. Fatal un-recoverable MCEs and recoverable MCEs. While un-recoverable errors are fatal, recoverable errors can also happen in kernel context are also treated as fatal by the kernel.

On certain Intel machines, MCEs are also broadcast to all threads in a system. If one thread is in the middle of executing WRMSR, a MCE will be taken at the end of the flow. Either way, they will wait for the thread performing the wrmsr(0x79) to rendezvous in the MCE handler and shutdown eventually if any of the threads in the system fail to check in to the MCE rendezvous.

To be paranoid and get predictable behavior, the OS can choose to set MCG_STATUS.MCIP. Since MCEs can be at most one in a system, if an MCE was signaled, the above condition will promote to a system reset automatically. OS can turn off MCIP at the end of the update for that core.

System Management Interrupt

SMIs are also broadcast to all CPUs in the platform. Microcode update requests exclusive access to the core before writing to MSR 0x79. So if it does happen such that, one thread is in WRMSR flow, and the 2nd got an SMI, that thread will be stopped in the first instruction in the SMI handler.

Since the secondary thread is stopped in the first instruction in SMI, there is very little chance that it would be in the middle of executing an instruction being patched. Plus OS has no way to stop SMIs from happening.

Non-Maskable Interrupts

When thread0 of a core is doing the microcode update, if thread1 is pulled into NMI, that can cause unpredictable behavior due to the reasons above.

OS can choose a variety of methods to avoid running into this situation.

Is the microcode suitable for late loading?

Late loading is done when the system is fully operational and running real workloads. Late loading behavior depends on what the base patch on the CPU is before upgrading to the new patch.

This is true for Intel CPUs.

Consider, for example, a CPU has patch level 1 and the update is to patch level 3.

Between patch1 and patch3, patch2 might have deprecated a software-visible feature.

This is unacceptable if software is even potentially using that feature. For instance, say MSR_X is no longer available after an update, accessing that MSR will cause a #GP fault.

Basically there is no way to declare a new microcode update suitable for late-loading. This is another one of the problems that caused late loading to be not enabled by default.

16.23.4 Builtin microcode

The loader supports also loading of a builtin microcode supplied through the regular builtin firmware method CONFIG_EXTRA_FIRMWARE. Only 64-bit is currently supported.

Here's an example:

```
CONFIG_EXTRA_FIRMWARE="intel-ucode/06-3a-09 amd-ucode/microcode_amd_fam15h.bin"
CONFIG_EXTRA_FIRMWARE_DIR="/lib/firmware"
```

This basically means, you have the following tree structure locally:

```
/lib/firmware/
|-- amd-ucode
...
|   |-- microcode_amd_fam15h.bin
...
|-- intel-ucode
...
|   |-- 06-3a-09
...
```

so that the build system can find those files and integrate them into the final kernel image. The early loader finds them and applies them.

Needless to say, this method is not the most flexible one because it requires rebuilding the kernel each time updated microcode from the CPU vendor is available.

16.24 User Interface for Resource Control feature

Copyright
© 2016 Intel Corporation

Authors

- Fenghua Yu <fenghua.yu@intel.com>
- Tony Luck <tony.luck@intel.com>
- Vikas Shivappa <vikas.shivappa@intel.com>

Intel refers to this feature as Intel Resource Director Technology(Intel(R) RDT). AMD refers to this feature as AMD Platform Quality of Service(AMD QoS).

This feature is enabled by the CONFIG_X86_CPU_RESCTRL and the x86 /proc/cpuinfo flag bits:

RDT (Resource Director Technology) Allocation	"rdt_a"
CAT (Cache Allocation Technology)	"cat_l3", "cat_l2"
CDP (Code and Data Prioritization)	"cdp_l3", "cdp_l2"
CQM (Cache QoS Monitoring)	"cqm_llc", "cqm_occup_llc"
MBM (Memory Bandwidth Monitoring)	"cqm_mbm_total", "cqm_mbm_local"
MBA (Memory Bandwidth Allocation)	"mba"
SMBA (Slow Memory Bandwidth Allocation)	""
BMEC (Bandwidth Monitoring Event Configuration)	""

Historically, new features were made visible by default in /proc/cpuinfo. This resulted in the feature flags becoming hard to parse by humans. Adding a new flag to /proc/cpuinfo should be avoided if user space can obtain information about the feature from resctrl's info directory.

To use the feature mount the file system:

```
# mount -t resctrl resctrl [-o cdp[,cdpl2][,mba_MBps]] /sys/fs/resctrl
```

mount options are:

"cdp":
Enable code/data prioritization in L3 cache allocations.

"cdpl2":
Enable code/data prioritization in L2 cache allocations.

"mba_MBps":
Enable the MBA Software Controller(mba_sc) to specify MBA bandwidth in MBps

L2 and L3 CDP are controlled separately.

RDT features are orthogonal. A particular system may support only monitoring, only control, or both monitoring and control. Cache pseudo-locking is a unique way of using cache control to "pin" or "lock" data in the cache. Details can be found in "Cache Pseudo-Locking".

The mount succeeds if either of allocation or monitoring is present, but only those files and directories supported by the system will be created. For more details on the behavior of the interface during monitoring and allocation, see the "Resource alloc and monitor groups" section.

16.24.1 Info directory

The 'info' directory contains information about the enabled resources. Each resource has its own subdirectory. The subdirectory names reflect the resource names.

Each subdirectory contains the following files with respect to allocation:

Cache resource(L3/L2) subdirectory contains the following files related to allocation:

"num_closids":
The number of CLOSIDs which are valid for this resource. The kernel uses the smallest number of CLOSIDs of all enabled resources as limit.

"cbm_mask":
The bitmask which is valid for this resource. This mask is equivalent to 100%.

"min_cbm_bits":
The minimum number of consecutive bits which must be set when writing a mask.

"shareable_bits":
Bitmask of shareable resource with other executing entities (e.g. I/O). User can use this when setting up exclusive cache partitions. Note that some platforms support devices that have their own settings for cache use which can over-ride these bits.

"bit_usage":
Annotated capacity bitmasks showing how all instances of the resource are used. The legend is:

"0":
Corresponding region is unused. When the system's resources have been allocated and a "0" is found in "bit_usage" it is a sign that resources are wasted.

"H":
Corresponding region is used by hardware only but available for software use. If a resource has bits set in "shareable_bits" but not all of these bits appear in the resource groups' schematas then the bits appearing in "shareable_bits" but no resource group will be marked as "H".

"X":
Corresponding region is available for sharing and used by hardware and software. These are the bits that appear in "shareable_bits" as well as a resource group's allocation.

"S":
Corresponding region is used by software and available for sharing.

"E":
Corresponding region is used exclusively by one resource group. No sharing allowed.

"P":
Corresponding region is pseudo-locked. No sharing allowed.

Memory bandwidth(MB) subdirectory contains the following files with respect to allocation:

"min_bandwidth":
The minimum memory bandwidth percentage which user can request.

"bandwidth_gran":
The granularity in which the memory bandwidth percentage is allocated. The allocated b/w percentage is rounded off to the next control step available on the hardware. The available bandwidth control steps are: min_bandwidth + N * bandwidth_gran.

"delay_linear":
Indicates if the delay scale is linear or non-linear. This field is purely informational only.

"thread_throttle_mode":
Indicator on Intel systems of how tasks running on threads of a physical core are throttled in cases where they request different memory bandwidth percentages:

"max":
the smallest percentage is applied to all threads

"per-thread":
bandwidth percentages are directly applied to the threads running on the core

If RDT monitoring is available there will be an "L3_MON" directory with the following files:

"num_rmids":
The number of RMIDs available. This is the upper bound for how many "CTRL_MON" + "MON" groups can be created.

"mon_features":
Lists the monitoring events if monitoring is enabled for the resource. Example:

```
# cat /sys/fs/resctrl/info/L3_MON/mon_features
llc_occupancy
```

```
mbm_total_bytes
mbm_local_bytes
```

If the system supports Bandwidth Monitoring Event Configuration (BMEC), then the bandwidth events will be configurable. The output will be:

```
# cat /sys/fs/resctrl/info/L3_MON/mon_features
llc_occupancy
mbm_total_bytes
mbm_total_bytes_config
mbm_local_bytes
mbm_local_bytes_config
```

"mbm_total_bytes_config", "mbm_local_bytes_config":

Read/write files containing the configuration for the mbm_total_bytes and mbm_local_bytes events, respectively, when the Bandwidth Monitoring Event Configuration (BMEC) feature is supported. The event configuration settings are domain specific and affect all the CPUs in the domain. When either event configuration is changed, the bandwidth counters for all RMIDs of both events (mbm_total_bytes as well as mbm_local_bytes) are cleared for that domain. The next read for every RMID will report "Unavailable" and subsequent reads will report the valid value.

Following are the types of events supported:

Bits	Description
6	Dirty Victims from the QOS domain to all types of memory
5	Reads to slow memory in the non-local NUMA domain
4	Reads to slow memory in the local NUMA domain
3	Non-temporal writes to non-local NUMA domain
2	Non-temporal writes to local NUMA domain
1	Reads to memory in the non-local NUMA domain
0	Reads to memory in the local NUMA domain

By default, the mbm_total_bytes configuration is set to 0x7f to count all the event types and the mbm_local_bytes configuration is set to 0x15 to count all the local memory events.

Examples:

- To view the current configuration::

  ```
  # cat /sys/fs/resctrl/info/L3_MON/mbm_total_bytes_config
  0=0x7f;1=0x7f;2=0x7f;3=0x7f

  # cat /sys/fs/resctrl/info/L3_MON/mbm_local_bytes_config
  0=0x15;1=0x15;3=0x15;4=0x15
  ```

- To change the mbm_total_bytes to count only reads on domain 0, the bits 0, 1, 4 and 5 needs to be set, which is 110011b in binary (in hexadecimal 0x33):

  ```
  # echo  "0=0x33" > /sys/fs/resctrl/info/L3_MON/mbm_total_bytes_config
  ```

```
# cat /sys/fs/resctrl/info/L3_MON/mbm_total_bytes_config
0=0x33;1=0x7f;2=0x7f;3=0x7f
```

- To change the mbm_local_bytes to count all the slow memory reads on domain 0 and 1, the bits 4 and 5 needs to be set, which is 110000b in binary (in hexadecimal 0x30):

```
# echo  "0=0x30;1=0x30" > /sys/fs/resctrl/info/L3_MON/mbm_local_bytes_
↪config

# cat /sys/fs/resctrl/info/L3_MON/mbm_local_bytes_config
0=0x30;1=0x30;3=0x15;4=0x15
```

"max_threshold_occupancy":
Read/write file provides the largest value (in bytes) at which a previously used LLC_occupancy counter can be considered for re-use.

Finally, in the top level of the "info" directory there is a file named "last_cmd_status". This is reset with every "command" issued via the file system (making new directories or writing to any of the control files). If the command was successful, it will read as "ok". If the command failed, it will provide more information that can be conveyed in the error returns from file operations. E.g.

```
# echo L3:0=f7 > schemata
bash: echo: write error: Invalid argument
# cat info/last_cmd_status
mask f7 has non-consecutive 1-bits
```

16.24.2 Resource alloc and monitor groups

Resource groups are represented as directories in the resctrl file system. The default group is the root directory which, immediately after mounting, owns all the tasks and cpus in the system and can make full use of all resources.

On a system with RDT control features additional directories can be created in the root directory that specify different amounts of each resource (see "schemata" below). The root and these additional top level directories are referred to as "CTRL_MON" groups below.

On a system with RDT monitoring the root directory and other top level directories contain a directory named "mon_groups" in which additional directories can be created to monitor subsets of tasks in the CTRL_MON group that is their ancestor. These are called "MON" groups in the rest of this document.

Removing a directory will move all tasks and cpus owned by the group it represents to the parent. Removing one of the created CTRL_MON groups will automatically remove all MON groups below it.

Moving MON group directories to a new parent CTRL_MON group is supported for the purpose of changing the resource allocations of a MON group without impacting its monitoring data or assigned tasks. This operation is not allowed for MON groups which monitor CPUs. No other move operation is currently allowed other than simply renaming a CTRL_MON or MON group.

All groups contain the following files:

"tasks":
Reading this file shows the list of all tasks that belong to this group. Writing a task id to the file will add a task to the group. If the group is a CTRL_MON group the task is removed from whichever previous CTRL_MON group owned the task and also from any MON group that owned the task. If the group is a MON group, then the task must already belong to the CTRL_MON parent of this group. The task is removed from any previous MON group.

"cpus":
Reading this file shows a bitmask of the logical CPUs owned by this group. Writing a mask to this file will add and remove CPUs to/from this group. As with the tasks file a hierarchy is maintained where MON groups may only include CPUs owned by the parent CTRL_MON group. When the resource group is in pseudo-locked mode this file will only be readable, reflecting the CPUs associated with the pseudo-locked region.

"cpus_list":
Just like "cpus", only using ranges of CPUs instead of bitmasks.

When control is enabled all CTRL_MON groups will also contain:

"schemata":
A list of all the resources available to this group. Each resource has its own line and format - see below for details.

"size":
Mirrors the display of the "schemata" file to display the size in bytes of each allocation instead of the bits representing the allocation.

"mode":
The "mode" of the resource group dictates the sharing of its allocations. A "shareable" resource group allows sharing of its allocations while an "exclusive" resource group does not. A cache pseudo-locked region is created by first writing "pseudo-locksetup" to the "mode" file before writing the cache pseudo-locked region's schemata to the resource group's "schemata" file. On successful pseudo-locked region creation the mode will automatically change to "pseudo-locked".

When monitoring is enabled all MON groups will also contain:

"mon_data":
This contains a set of files organized by L3 domain and by RDT event. E.g. on a system with two L3 domains there will be subdirectories "mon_L3_00" and "mon_L3_01". Each of these directories have one file per event (e.g. "llc_occupancy", "mbm_total_bytes", and "mbm_local_bytes"). In a MON group these files provide a read out of the current value of the event for all tasks in the group. In CTRL_MON groups these files provide the sum for all tasks in the CTRL_MON group and all tasks in MON groups. Please see example section for more details on usage.

Resource allocation rules

When a task is running the following rules define which resources are available to it:

1) If the task is a member of a non-default group, then the schemata for that group is used.
2) Else if the task belongs to the default group, but is running on a CPU that is assigned to some specific group, then the schemata for the CPU's group is used.
3) Otherwise the schemata for the default group is used.

Resource monitoring rules

1) If a task is a member of a MON group, or non-default CTRL_MON group then RDT events for the task will be reported in that group.
2) If a task is a member of the default CTRL_MON group, but is running on a CPU that is assigned to some specific group, then the RDT events for the task will be reported in that group.
3) Otherwise RDT events for the task will be reported in the root level "mon_data" group.

16.24.3 Notes on cache occupancy monitoring and control

When moving a task from one group to another you should remember that this only affects *new* cache allocations by the task. E.g. you may have a task in a monitor group showing 3 MB of cache occupancy. If you move to a new group and immediately check the occupancy of the old and new groups you will likely see that the old group is still showing 3 MB and the new group zero. When the task accesses locations still in cache from before the move, the h/w does not update any counters. On a busy system you will likely see the occupancy in the old group go down as cache lines are evicted and re-used while the occupancy in the new group rises as the task accesses memory and loads into the cache are counted based on membership in the new group.

The same applies to cache allocation control. Moving a task to a group with a smaller cache partition will not evict any cache lines. The process may continue to use them from the old partition.

Hardware uses CLOSid(Class of service ID) and an RMID(Resource monitoring ID) to identify a control group and a monitoring group respectively. Each of the resource groups are mapped to these IDs based on the kind of group. The number of CLOSid and RMID are limited by the hardware and hence the creation of a "CTRL_MON" directory may fail if we run out of either CLOSID or RMID and creation of "MON" group may fail if we run out of RMIDs.

max_threshold_occupancy - generic concepts

Note that an RMID once freed may not be immediately available for use as the RMID is still tagged the cache lines of the previous user of RMID. Hence such RMIDs are placed on limbo list and checked back if the cache occupancy has gone down. If there is a time when system has a lot of limbo RMIDs but which are not ready to be used, user may see an -EBUSY during mkdir.

max_threshold_occupancy is a user configurable value to determine the occupancy at which an RMID can be freed.

Schemata files - general concepts

Each line in the file describes one resource. The line starts with the name of the resource, followed by specific values to be applied in each of the instances of that resource on the system.

Cache IDs

On current generation systems there is one L3 cache per socket and L2 caches are generally just shared by the hyperthreads on a core, but this isn't an architectural requirement. We could have multiple separate L3 caches on a socket, multiple cores could share an L2 cache. So instead of using "socket" or "core" to define the set of logical cpus sharing a resource we use a "Cache ID". At a given cache level this will be a unique number across the whole system (but it isn't guaranteed to be a contiguous sequence, there may be gaps). To find the ID for each logical CPU look in /sys/devices/system/cpu/cpu*/cache/index*/id

Cache Bit Masks (CBM)

For cache resources we describe the portion of the cache that is available for allocation using a bitmask. The maximum value of the mask is defined by each cpu model (and may be different for different cache levels). It is found using CPUID, but is also provided in the "info" directory of the resctrl file system in "info/{resource}/cbm_mask". Intel hardware requires that these masks have all the '1' bits in a contiguous block. So 0x3, 0x6 and 0xC are legal 4-bit masks with two bits set, but 0x5, 0x9 and 0xA are not. On a system with a 20-bit mask each bit represents 5% of the capacity of the cache. You could partition the cache into four equal parts with masks: 0x1f, 0x3e0, 0x7c00, 0xf8000.

16.24.4 Memory bandwidth Allocation and monitoring

For Memory bandwidth resource, by default the user controls the resource by indicating the percentage of total memory bandwidth.

The minimum bandwidth percentage value for each cpu model is predefined and can be looked up through "info/MB/min_bandwidth". The bandwidth granularity that is allocated is also dependent on the cpu model and can be looked up at "info/MB/bandwidth_gran". The available bandwidth control steps are: min_bw + N * bw_gran. Intermediate values are rounded to the next control step available on the hardware.

The bandwidth throttling is a core specific mechanism on some of Intel SKUs. Using a high bandwidth and a low bandwidth setting on two threads sharing a core may result in both threads being throttled to use the low bandwidth (see "thread_throttle_mode").

The fact that Memory bandwidth allocation(MBA) may be a core specific mechanism where as memory bandwidth monitoring(MBM) is done at the package level may lead to confusion when users try to apply control via the MBA and then monitor the bandwidth to see if the controls are effective. Below are such scenarios:

1. User may *not* see increase in actual bandwidth when percentage values are increased:

This can occur when aggregate L2 external bandwidth is more than L3 external bandwidth. Consider an SKL SKU with 24 cores on a package and where L2 external is 10GBps (hence aggregate L2 external bandwidth is 240GBps) and L3 external bandwidth is 100GBps. Now a workload with '20 threads, having 50% bandwidth, each consuming 5GBps' consumes the max L3 bandwidth of 100GBps although the percentage value specified is only 50% << 100%. Hence increasing the bandwidth percentage will not yield any more bandwidth. This is because although the L2 external bandwidth still has capacity, the L3 external bandwidth is fully used. Also note that this would be dependent on number of cores the benchmark is run on.

2. Same bandwidth percentage may mean different actual bandwidth depending on # of threads:

For the same SKU in #1, a 'single thread, with 10% bandwidth' and '4 thread, with 10% bandwidth' can consume upto 10GBps and 40GBps although they have same percentage bandwidth of 10%. This is simply because as threads start using more cores in an rdtgroup, the actual bandwidth may increase or vary although user specified bandwidth percentage is same.

In order to mitigate this and make the interface more user friendly, resctrl added support for specifying the bandwidth in MBps as well. The kernel underneath would use a software feedback mechanism or a "Software Controller(mba_sc)" which reads the actual bandwidth using MBM counters and adjust the memory bandwidth percentages to ensure:

```
"actual bandwidth < user specified bandwidth".
```

By default, the schemata would take the bandwidth percentage values where as user can switch to the "MBA software controller" mode using a mount option 'mba_MBps'. The schemata format is specified in the below sections.

L3 schemata file details (code and data prioritization disabled)

With CDP disabled the L3 schemata format is:

```
L3:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
```

L3 schemata file details (CDP enabled via mount option to resctrl)

When CDP is enabled L3 control is split into two separate resources so you can specify independent masks for code and data like this:

```
L3DATA:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
L3CODE:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
```

L2 schemata file details

CDP is supported at L2 using the ‘cdpl2’ mount option. The schemata format is either:

```
L2:<cache_id0>=<cbm>;<cache_id1>=<cbm>;...
```

or

> L2DATA:<cache_id0>=<cbm>;<cache_id1>=<cbm>;... L2CODE:<cache_id0>=<cbm>;<cach

Memory bandwidth Allocation (default mode)

Memory b/w domain is L3 cache.

```
MB:<cache_id0>=bandwidth0;<cache_id1>=bandwidth1;...
```

Memory bandwidth Allocation specified in MBps

Memory bandwidth domain is L3 cache.

```
MB:<cache_id0>=bw_MBps0;<cache_id1>=bw_MBps1;...
```

Slow Memory Bandwidth Allocation (SMBA)

AMD hardware supports Slow Memory Bandwidth Allocation (SMBA). CXL.memory is the only supported “slow” memory device. With the support of SMBA, the hardware enables bandwidth allocation on the slow memory devices. If there are multiple such devices in the system, the throttling logic groups all the slow sources together and applies the limit on them as a whole.

The presence of SMBA (with CXL.memory) is independent of slow memory devices presence. If there are no such devices on the system, then configuring SMBA will have no impact on the performance of the system.

The bandwidth domain for slow memory is L3 cache. Its schemata file is formatted as:

```
SMBA:<cache_id0>=bandwidth0;<cache_id1>=bandwidth1;...
```

Reading/writing the schemata file

Reading the schemata file will show the state of all resources on all domains. When writing you only need to specify those values which you wish to change. E.g.

```
# cat schemata
L3DATA:0=fffff;1=fffff;2=fffff;3=fffff
L3CODE:0=fffff;1=fffff;2=fffff;3=fffff
# echo "L3DATA:2=3c0;" > schemata
# cat schemata
L3DATA:0=fffff;1=fffff;2=3c0;3=fffff
L3CODE:0=fffff;1=fffff;2=fffff;3=fffff
```

Reading/writing the schemata file (on AMD systems)

Reading the schemata file will show the current bandwidth limit on all domains. The allocated resources are in multiples of one eighth GB/s. When writing to the file, you need to specify what cache id you wish to configure the bandwidth limit.

For example, to allocate 2GB/s limit on the first cache id:

```
# cat schemata
  MB:0=2048;1=2048;2=2048;3=2048
  L3:0=ffff;1=ffff;2=ffff;3=ffff

# echo "MB:1=16" > schemata
# cat schemata
  MB:0=2048;1=  16;2=2048;3=2048
  L3:0=ffff;1=ffff;2=ffff;3=ffff
```

Reading/writing the schemata file (on AMD systems) with SMBA feature

Reading and writing the schemata file is the same as without SMBA in above section.

For example, to allocate 8GB/s limit on the first cache id:

```
# cat schemata
  SMBA:0=2048;1=2048;2=2048;3=2048
    MB:0=2048;1=2048;2=2048;3=2048
    L3:0=ffff;1=ffff;2=ffff;3=ffff

# echo "SMBA:1=64" > schemata
# cat schemata
  SMBA:0=2048;1=  64;2=2048;3=2048
    MB:0=2048;1=2048;2=2048;3=2048
    L3:0=ffff;1=ffff;2=ffff;3=ffff
```

16.24.5 Cache Pseudo-Locking

CAT enables a user to specify the amount of cache space that an application can fill. Cache pseudo-locking builds on the fact that a CPU can still read and write data pre-allocated outside its current allocated area on a cache hit. With cache pseudo-locking, data can be preloaded into a reserved portion of cache that no application can fill, and from that point on will only serve cache hits. The cache pseudo-locked memory is made accessible to user space where an application can map it into its virtual address space and thus have a region of memory with reduced average read latency.

The creation of a cache pseudo-locked region is triggered by a request from the user to do so that is accompanied by a schemata of the region to be pseudo-locked. The cache pseudo-locked region is created as follows:

- Create a CAT allocation CLOSNEW with a CBM matching the schemata from the user of the cache region that will contain the pseudo-locked memory. This region must not overlap with any current CAT allocation/CLOS on the system and no future overlap with this cache region is allowed while the pseudo-locked region exists.
- Create a contiguous region of memory of the same size as the cache region.
- Flush the cache, disable hardware prefetchers, disable preemption.
- Make CLOSNEW the active CLOS and touch the allocated memory to load it into the cache.
- Set the previous CLOS as active.
- At this point the closid CLOSNEW can be released - the cache pseudo-locked region is protected as long as its CBM does not appear in any CAT allocation. Even though the cache pseudo-locked region will from this point on not appear in any CBM of any CLOS an application running with any CLOS will be able to access the memory in the pseudo-locked region since the region continues to serve cache hits.
- The contiguous region of memory loaded into the cache is exposed to user-space as a character device.

Cache pseudo-locking increases the probability that data will remain in the cache via carefully configuring the CAT feature and controlling application behavior. There is no guarantee that data is placed in cache. Instructions like INVD, WBINVD, CLFLUSH, etc. can still evict "locked" data from cache. Power management C-states may shrink or power off cache. Deeper C-states will automatically be restricted on pseudo-locked region creation.

It is required that an application using a pseudo-locked region runs with affinity to the cores (or a subset of the cores) associated with the cache on which the pseudo-locked region resides. A sanity check within the code will not allow an application to map pseudo-locked memory unless it runs with affinity to cores associated with the cache on which the pseudo-locked region resides. The sanity check is only done during the initial mmap() handling, there is no enforcement afterwards and the application self needs to ensure it remains affine to the correct cores.

Pseudo-locking is accomplished in two stages:

1) During the first stage the system administrator allocates a portion of cache that should be dedicated to pseudo-locking. At this time an equivalent portion of memory is allocated, loaded into allocated cache portion, and exposed as a character device.
2) During the second stage a user-space application maps (mmap()) the pseudo-locked memory into its address space.

Cache Pseudo-Locking Interface

A pseudo-locked region is created using the resctrl interface as follows:

1) Create a new resource group by creating a new directory in /sys/fs/resctrl.
2) Change the new resource group's mode to "pseudo-locksetup" by writing "pseudo-locksetup" to the "mode" file.
3) Write the schemata of the pseudo-locked region to the "schemata" file. All bits within the schemata should be "unused" according to the "bit_usage" file.

On successful pseudo-locked region creation the "mode" file will contain "pseudo-locked" and a new character device with the same name as the resource group will exist in /dev/pseudo_lock. This character device can be mmap()'ed by user space in order to obtain access to the pseudo-locked memory region.

An example of cache pseudo-locked region creation and usage can be found below.

Cache Pseudo-Locking Debugging Interface

The pseudo-locking debugging interface is enabled by default (if CONFIG_DEBUG_FS is enabled) and can be found in /sys/kernel/debug/resctrl.

There is no explicit way for the kernel to test if a provided memory location is present in the cache. The pseudo-locking debugging interface uses the tracing infrastructure to provide two ways to measure cache residency of the pseudo-locked region:

1) Memory access latency using the pseudo_lock_mem_latency tracepoint. Data from these measurements are best visualized using a hist trigger (see example below). In this test the pseudo-locked region is traversed at a stride of 32 bytes while hardware prefetchers and preemption are disabled. This also provides a substitute visualization of cache hits and misses.
2) Cache hit and miss measurements using model specific precision counters if available. Depending on the levels of cache on the system the pseudo_lock_l2 and pseudo_lock_l3 tracepoints are available.

When a pseudo-locked region is created a new debugfs directory is created for it in debugfs as /sys/kernel/debug/resctrl/<newdir>. A single write-only file, pseudo_lock_measure, is present in this directory. The measurement of the pseudo-locked region depends on the number written to this debugfs file:

1:
: writing "1" to the pseudo_lock_measure file will trigger the latency measurement captured in the pseudo_lock_mem_latency tracepoint. See example below.

2:
: writing "2" to the pseudo_lock_measure file will trigger the L2 cache residency (cache hits and misses) measurement captured in the pseudo_lock_l2 tracepoint. See example below.

3:
: writing "3" to the pseudo_lock_measure file will trigger the L3 cache residency (cache hits and misses) measurement captured in the pseudo_lock_l3 tracepoint.

All measurements are recorded with the tracing infrastructure. This requires the relevant tracepoints to be enabled before the measurement is triggered.

Example of latency debugging interface

In this example a pseudo-locked region named "newlock" was created. Here is how we can measure the latency in cycles of reading from this region and visualize this data with a histogram that is available if CONFIG_HIST_TRIGGERS is set:

```
# :> /sys/kernel/tracing/trace
# echo 'hist:keys=latency' > /sys/kernel/tracing/events/resctrl/pseudo_lock_
↪mem_latency/trigger
# echo 1 > /sys/kernel/tracing/events/resctrl/pseudo_lock_mem_latency/enable
# echo 1 > /sys/kernel/debug/resctrl/newlock/pseudo_lock_measure
# echo 0 > /sys/kernel/tracing/events/resctrl/pseudo_lock_mem_latency/enable
# cat /sys/kernel/tracing/events/resctrl/pseudo_lock_mem_latency/hist

# event histogram
#
# trigger info: hist:keys=latency:vals=hitcount:sort=hitcount:size=2048␣
↪[active]
#

{ latency:        456 } hitcount:          1
{ latency:         50 } hitcount:         83
{ latency:         36 } hitcount:         96
{ latency:         44 } hitcount:        174
{ latency:         48 } hitcount:        195
{ latency:         46 } hitcount:        262
{ latency:         42 } hitcount:        693
{ latency:         40 } hitcount:       3204
{ latency:         38 } hitcount:       3484

Totals:
    Hits: 8192
    Entries: 9
  Dropped: 0
```

Example of cache hits/misses debugging

In this example a pseudo-locked region named "newlock" was created on the L2 cache of a platform. Here is how we can obtain details of the cache hits and misses using the platform's precision counters.

```
# :> /sys/kernel/tracing/trace
# echo 1 > /sys/kernel/tracing/events/resctrl/pseudo_lock_l2/enable
# echo 2 > /sys/kernel/debug/resctrl/newlock/pseudo_lock_measure
# echo 0 > /sys/kernel/tracing/events/resctrl/pseudo_lock_l2/enable
# cat /sys/kernel/tracing/trace

# tracer: nop
#
#                              _-----=> irqs-off
#                            / _----=> need-resched
#                           | / _---=> hardirq/softirq
#                           || / _--=> preempt-depth
#                           ||| /     delay
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
pseudo_lock_mea-1672  [002] ....  3132.860500: pseudo_lock_l2: hits=4097 miss=0
```

Examples for RDT allocation usage

1) Example 1

On a two socket machine (one L3 cache per socket) with just four bits for cache bit masks, minimum b/w of 10% with a memory bandwidth granularity of 10%.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir p0 p1
# echo "L3:0=3;1=c\nMB:0=50;1=50" > /sys/fs/resctrl/p0/schemata
# echo "L3:0=3;1=3\nMB:0=50;1=50" > /sys/fs/resctrl/p1/schemata
```

The default resource group is unmodified, so we have access to all parts of all caches (its schemata file reads "L3:0=f;1=f").

Tasks that are under the control of group "p0" may only allocate from the "lower" 50% on cache ID 0, and the "upper" 50% of cache ID 1. Tasks in group "p1" use the "lower" 50% of cache on both sockets.

Similarly, tasks that are under the control of group "p0" may use a maximum memory b/w of 50% on socket0 and 50% on socket 1. Tasks in group "p1" may also use 50% memory b/w on both sockets. Note that unlike cache masks, memory b/w cannot specify whether these allocations can overlap or not. The allocations specifies the maximum b/w that the group may be able to use and the system admin can configure the b/w accordingly.

If resctrl is using the software controller (mba_sc) then user can enter the max b/w in MB rather than the percentage values.

```
# echo "L3:0=3;1=c\nMB:0=1024;1=500" > /sys/fs/resctrl/p0/schemata
# echo "L3:0=3;1=3\nMB:0=1024;1=500" > /sys/fs/resctrl/p1/schemata
```

In the above example the tasks in "p1" and "p0" on socket 0 would use a max b/w of 1024MB where as on socket 1 they would use 500MB.

2) Example 2

Again two sockets, but this time with a more realistic 20-bit mask.

Two real time tasks pid=1234 running on processor 0 and pid=5678 running on processor 1 on socket 0 on a 2-socket and dual core machine. To avoid noisy neighbors, each of the two real-time tasks exclusively occupies one quarter of L3 cache on socket 0.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
```

First we reset the schemata for the default group so that the “upper” 50% of the L3 cache on socket 0 and 50% of memory b/w cannot be used by ordinary tasks:

```
# echo "L3:0=3ff;1=fffff\nMB:0=50;1=100" > schemata
```

Next we make a resource group for our first real time task and give it access to the “top” 25% of the cache on socket 0.

```
# mkdir p0
# echo "L3:0=f8000;1=fffff" > p0/schemata
```

Finally we move our first real time task into this resource group. We also use taskset(1) to ensure the task always runs on a dedicated CPU on socket 0. Most uses of resource groups will also constrain which processors tasks run on.

```
# echo 1234 > p0/tasks
# taskset -cp 1 1234
```

Ditto for the second real time task (with the remaining 25% of cache):

```
# mkdir p1
# echo "L3:0=7c00;1=fffff" > p1/schemata
# echo 5678 > p1/tasks
# taskset -cp 2 5678
```

For the same 2 socket system with memory b/w resource and CAT L3 the schemata would look like(Assume min_bandwidth 10 and bandwidth_gran is 10):

For our first real time task this would request 20% memory b/w on socket 0.

```
# echo -e "L3:0=f8000;1=fffff\nMB:0=20;1=100" > p0/schemata
```

For our second real time task this would request an other 20% memory b/w on socket 0.

```
# echo -e "L3:0=f8000;1=fffff\nMB:0=20;1=100" > p0/schemata
```

3) Example 3

A single socket system which has real-time tasks running on core 4-7 and non real-time workload assigned to core 0-3. The real-time tasks share text and data, so a per task association is not required and due to interaction with the kernel it’s desired that the kernel on these cores shares L3 with the tasks.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
```

First we reset the schemata for the default group so that the “upper” 50% of the L3 cache on socket 0, and 50% of memory bandwidth on socket 0 cannot be used by ordinary tasks:

```
# echo "L3:0=3ff\nMB:0=50" > schemata
```

Next we make a resource group for our real time cores and give it access to the “top” 50% of the cache on socket 0 and 50% of memory bandwidth on socket 0.

```
# mkdir p0
# echo "L3:0=ffc00\nMB:0=50" > p0/schemata
```

Finally we move core 4-7 over to the new group and make sure that the kernel and the tasks running there get 50% of the cache. They should also get 50% of memory bandwidth assuming that the cores 4-7 are SMT siblings and only the real time threads are scheduled on the cores 4-7.

```
# echo F0 > p0/cpus
```

4) Example 4

The resource groups in previous examples were all in the default “shareable” mode allowing sharing of their cache allocations. If one resource group configures a cache allocation then nothing prevents another resource group to overlap with that allocation.

In this example a new exclusive resource group will be created on a L2 CAT system with two L2 cache instances that can be configured with an 8-bit capacity bitmask. The new exclusive resource group will be configured to use 25% of each cache instance.

```
# mount -t resctrl resctrl /sys/fs/resctrl/
# cd /sys/fs/resctrl
```

First, we observe that the default group is configured to allocate to all L2 cache:

```
# cat schemata
L2:0=ff;1=ff
```

We could attempt to create the new resource group at this point, but it will fail because of the overlap with the schemata of the default group:

```
# mkdir p0
# echo 'L2:0=0x3;1=0x3' > p0/schemata
# cat p0/mode
shareable
# echo exclusive > p0/mode
-sh: echo: write error: Invalid argument
# cat info/last_cmd_status
schemata overlaps
```

To ensure that there is no overlap with another resource group the default resource group’s schemata has to change, making it possible for the new resource group to become exclusive.

```
# echo 'L2:0=0xfc;1=0xfc' > schemata
# echo exclusive > p0/mode
# grep . p0/*
p0/cpus:0
p0/mode:exclusive
p0/schemata:L2:0=03;1=03
p0/size:L2:0=262144;1=262144
```

A new resource group will on creation not overlap with an exclusive resource group:

```
# mkdir p1
# grep . p1/*
p1/cpus:0
p1/mode:shareable
p1/schemata:L2:0=fc;1=fc
p1/size:L2:0=786432;1=786432
```

The bit_usage will reflect how the cache is used:

```
# cat info/L2/bit_usage
0=SSSSSSEE;1=SSSSSSEE
```

A resource group cannot be forced to overlap with an exclusive resource group:

```
# echo 'L2:0=0x1;1=0x1' > p1/schemata
-sh: echo: write error: Invalid argument
# cat info/last_cmd_status
overlaps with exclusive group
```

Example of Cache Pseudo-Locking

Lock portion of L2 cache from cache id 1 using CBM 0x3. Pseudo-locked region is exposed at /dev/pseudo_lock/newlock that can be provided to application for argument to mmap().

```
# mount -t resctrl resctrl /sys/fs/resctrl/
# cd /sys/fs/resctrl
```

Ensure that there are bits available that can be pseudo-locked, since only unused bits can be pseudo-locked the bits to be pseudo-locked needs to be removed from the default resource group's schemata:

```
# cat info/L2/bit_usage
0=SSSSSSSS;1=SSSSSSSS
# echo 'L2:1=0xfc' > schemata
# cat info/L2/bit_usage
0=SSSSSSSS;1=SSSSSS00
```

Create a new resource group that will be associated with the pseudo-locked region, indicate that it will be used for a pseudo-locked region, and configure the requested pseudo-locked region capacity bitmask:

```
# mkdir newlock
# echo pseudo-locksetup > newlock/mode
# echo 'L2:1=0x3' > newlock/schemata
```

On success the resource group's mode will change to pseudo-locked, the bit_usage will reflect the pseudo-locked region, and the character device exposing the pseudo-locked region will exist:

```
# cat newlock/mode
pseudo-locked
```

```
# cat info/L2/bit_usage
0=SSSSSSSS;1=SSSSSSPP
# ls -l /dev/pseudo_lock/newlock
crw------- 1 root root 243, 0 Apr  3 05:01 /dev/pseudo_lock/newlock
```

```
/*
* Example code to access one page of pseudo-locked cache region
* from user space.
*/
#define _GNU_SOURCE
#include <fcntl.h>
#include <sched.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

/*
* It is required that the application runs with affinity to only
* cores associated with the pseudo-locked region. Here the cpu
* is hardcoded for convenience of example.
*/
static int cpuid = 2;

int main(int argc, char *argv[])
{
  cpu_set_t cpuset;
  long page_size;
  void *mapping;
  int dev_fd;
  int ret;

  page_size = sysconf(_SC_PAGESIZE);

  CPU_ZERO(&cpuset);
  CPU_SET(cpuid, &cpuset);
  ret = sched_setaffinity(0, sizeof(cpuset), &cpuset);
  if (ret < 0) {
    perror("sched_setaffinity");
    exit(EXIT_FAILURE);
  }

  dev_fd = open("/dev/pseudo_lock/newlock", O_RDWR);
  if (dev_fd < 0) {
    perror("open");
    exit(EXIT_FAILURE);
  }

  mapping = mmap(0, page_size, PROT_READ | PROT_WRITE, MAP_SHARED,
          dev_fd, 0);
  if (mapping == MAP_FAILED) {
    perror("mmap");
    close(dev_fd);
    exit(EXIT_FAILURE);
  }

  /* Application interacts with pseudo-locked memory @mapping */

  ret = munmap(mapping, page_size);
  if (ret < 0) {
    perror("munmap");
    close(dev_fd);
    exit(EXIT_FAILURE);
  }

  close(dev_fd);
  exit(EXIT_SUCCESS);
}
```

Locking between applications

Certain operations on the resctrl filesystem, composed of read/writes to/from multiple files, must be atomic.

As an example, the allocation of an exclusive reservation of L3 cache involves:

1. Read the cbmmasks from each directory or the per-resource "bit_usage"
2. Find a contiguous set of bits in the global CBM bitmask that is clear in any of the directory cbmmasks
3. Create a new directory
4. Set the bits found in step 2 to the new directory "schemata" file

If two applications attempt to allocate space concurrently then they can end up allocating the same bits so the reservations are shared instead of exclusive.

To coordinate atomic operations on the resctrlfs and to avoid the problem above, the following locking procedure is recommended:

Locking is based on flock, which is available in libc and also as a shell script command

Write lock:

A) Take flock(LOCK_EX) on /sys/fs/resctrl
B) Read/write the directory structure.
C) funlock

Read lock:

A) Take flock(LOCK_SH) on /sys/fs/resctrl
B) If success read the directory structure.
C) funlock

Example with bash:

```
# Atomically read directory structure
$ flock -s /sys/fs/resctrl/ find /sys/fs/resctrl

# Read directory contents and create new subdirectory

$ cat create-dir.sh
find /sys/fs/resctrl/ > output.txt
mask = function-of(output.txt)
mkdir /sys/fs/resctrl/newres/
echo mask > /sys/fs/resctrl/newres/schemata

$ flock /sys/fs/resctrl/ ./create-dir.sh
```

Example with C:

```
/*
* Example code do take advisory locks
* before accessing resctrl filesystem
*/
#include <sys/file.h>
#include <stdlib.h>

void resctrl_take_shared_lock(int fd)
{
  int ret;

  /* take shared lock on resctrl filesystem */
  ret = flock(fd, LOCK_SH);
  if (ret) {
    perror("flock");
    exit(-1);
  }
}

void resctrl_take_exclusive_lock(int fd)
{
  int ret;

  /* release lock on resctrl filesystem */
  ret = flock(fd, LOCK_EX);
  if (ret) {
    perror("flock");
    exit(-1);
  }
}

void resctrl_release_lock(int fd)
{
  int ret;
  /* take shared lock on resctrl filesystem */
  ret = flock(fd, LOCK_UN);
  if (ret) {
    perror("flock");
    exit(-1);
  }
}

void main(void)
{
  int fd, ret;

  fd = open("/sys/fs/resctrl", O_DIRECTORY);
  if (fd == -1) {
    perror("open");
    exit(-1);
  }
  resctrl_take_shared_lock(fd);
  /* code to read directory contents */
  resctrl_release_lock(fd);

  resctrl_take_exclusive_lock(fd);
  /* code to read and write directory contents */
  resctrl_release_lock(fd);
}
```

16.24.6 Examples for RDT Monitoring along with allocation usage

Reading monitored data

Reading an event file (for ex: mon_data/mon_L3_00/llc_occupancy) would show the current snapshot of LLC occupancy of the corresponding MON group or CTRL_MON group.

Example 1 (Monitor CTRL_MON group and subset of tasks in CTRL_MON group)

On a two socket machine (one L3 cache per socket) with just four bits for cache bit masks:

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir p0 p1
# echo "L3:0=3;1=c" > /sys/fs/resctrl/p0/schemata
# echo "L3:0=3;1=3" > /sys/fs/resctrl/p1/schemata
# echo 5678 > p1/tasks
# echo 5679 > p1/tasks
```

The default resource group is unmodified, so we have access to all parts of all caches (its schemata file reads “L3:0=f;1=f”).

Tasks that are under the control of group “p0” may only allocate from the “lower” 50% on cache ID 0, and the “upper” 50% of cache ID 1. Tasks in group “p1” use the “lower” 50% of cache on both sockets.

Create monitor groups and assign a subset of tasks to each monitor group.

```
# cd /sys/fs/resctrl/p1/mon_groups
# mkdir m11 m12
# echo 5678 > m11/tasks
# echo 5679 > m12/tasks
```

fetch data (data shown in bytes)

```
# cat m11/mon_data/mon_L3_00/llc_occupancy
16234000
# cat m11/mon_data/mon_L3_01/llc_occupancy
14789000
# cat m12/mon_data/mon_L3_00/llc_occupancy
16789000
```

The parent ctrl_mon group shows the aggregated data.

```
# cat /sys/fs/resctrl/p1/mon_data/mon_l3_00/llc_occupancy
31234000
```

Example 2 (Monitor a task from its creation)

On a two socket machine (one L3 cache per socket):

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir p0 p1
```

An RMID is allocated to the group once its created and hence the <cmd> below is monitored from its creation.

```
# echo $$ > /sys/fs/resctrl/p1/tasks
# <cmd>
```

Fetch the data:

```
# cat /sys/fs/resctrl/p1/mon_data/mon_l3_00/llc_occupancy
31789000
```

Example 3 (Monitor without CAT support or before creating CAT groups)

Assume a system like HSW has only CQM and no CAT support. In this case the resctrl will still mount but cannot create CTRL_MON directories. But user can create different MON groups within the root group thereby able to monitor all tasks including kernel threads.

This can also be used to profile jobs cache size footprint before being able to allocate them to different allocation groups.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir mon_groups/m01
# mkdir mon_groups/m02

# echo 3478 > /sys/fs/resctrl/mon_groups/m01/tasks
# echo 2467 > /sys/fs/resctrl/mon_groups/m02/tasks
```

Monitor the groups separately and also get per domain data. From the below its apparent that the tasks are mostly doing work on domain(socket) 0.

```
# cat /sys/fs/resctrl/mon_groups/m01/mon_L3_00/llc_occupancy
31234000
# cat /sys/fs/resctrl/mon_groups/m01/mon_L3_01/llc_occupancy
34555
# cat /sys/fs/resctrl/mon_groups/m02/mon_L3_00/llc_occupancy
31234000
# cat /sys/fs/resctrl/mon_groups/m02/mon_L3_01/llc_occupancy
32789
```

Example 4 (Monitor real time tasks)

A single socket system which has real time tasks running on cores 4-7 and non real time tasks on other cpus. We want to monitor the cache occupancy of the real time threads on these cores.

```
# mount -t resctrl resctrl /sys/fs/resctrl
# cd /sys/fs/resctrl
# mkdir p1
```

Move the cpus 4-7 over to p1:

```
# echo f0 > p1/cpus
```

View the llc occupancy snapshot:

```
# cat /sys/fs/resctrl/p1/mon_data/mon_L3_00/llc_occupancy
11234000
```

16.24.7 Intel RDT Errata

Intel MBM Counters May Report System Memory Bandwidth Incorrectly

Errata SKX99 for Skylake server and BDF102 for Broadwell server.

Problem: Intel Memory Bandwidth Monitoring (MBM) counters track metrics according to the assigned Resource Monitor ID (RMID) for that logical core. The IA32_QM_CTR register (MSR 0xC8E), used to report these metrics, may report incorrect system bandwidth for certain RMID values.

Implication: Due to the errata, system memory bandwidth may not match what is reported.

Workaround: MBM total and local readings are corrected according to the following correction factor table:

core count	rmid count	rmid threshold	correction factor
1	8	0	1.000000
2	16	0	1.000000
3	24	15	0.969650
4	32	0	1.000000
6	48	31	0.969650
7	56	47	1.142857
8	64	0	1.000000
9	72	63	1.185115
10	80	63	1.066553
11	88	79	1.454545
12	96	0	1.000000
13	104	95	1.230769
14	112	95	1.142857
15	120	95	1.066667
16	128	0	1.000000
17	136	127	1.254863
18	144	127	1.185255
19	152	0	1.000000
20	160	127	1.066667
21	168	0	1.000000
22	176	159	1.454334
23	184	0	1.000000
24	192	127	0.969744
25	200	191	1.280246
26	208	191	1.230921
27	216	0	1.000000
28	224	191	1.143118

If rmid > rmid threshold, MBM total and local values should be multiplied by the correction factor.

See:

1. Erratum SKX99 in Intel Xeon Processor Scalable Family Specification Update: http://web.archive.org/web/20200716124958/https://www.intel.com/content/www/us/en/processors/xeon/scalable/xeon-scalable-spec-update.html

2. Erratum BDF102 in Intel Xeon E5-2600 v4 Processor Product Family Specification Update: http://web.archive.org/web/20191125200531/https://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v4-spec-update.pdf

3. The errata in Intel Resource Director Technology (Intel RDT) on 2nd Generation Intel Xeon Scalable Processors Reference Manual: https://software.intel.com/content/www/us/en/develop/articles/intel-resource-director-technology-rdt-reference-manual.html

for further information.

16.25 TSX Async Abort (TAA) mitigation

16.25.1 Overview

TSX Async Abort (TAA) is a side channel attack on internal buffers in some Intel processors similar to Microachitectural Data Sampling (MDS). In this case certain loads may speculatively pass invalid data to dependent operations when an asynchronous abort condition is pending in a Transactional Synchronization Extensions (TSX) transaction. This includes loads with no fault or assist condition. Such loads may speculatively expose stale data from the same uarch data structures as in MDS, with same scope of exposure i.e. same-thread and cross-thread. This issue affects all current processors that support TSX.

16.25.2 Mitigation strategy

a) TSX disable - one of the mitigations is to disable TSX. A new MSR IA32_TSX_CTRL will be available in future and current processors after microcode update which can be used to disable TSX. In addition, it controls the enumeration of the TSX feature bits (RTM and HLE) in CPUID.

b) Clear CPU buffers - similar to MDS, clearing the CPU buffers mitigates this vulnerability. More details on this approach can be found in *Documentation/admin-guide/hw-vuln/mds.rst*.

16.25.3 Kernel internal mitigation modes

off	Mitigation is disabled. Either the CPU is not affected or tsx_async_abort=off is supplied on the kernel command line.
tsx disabled	Mitigation is enabled. TSX feature is disabled by default at bootup on processors that support TSX control.
verw	Mitigation is enabled. CPU is affected and MD_CLEAR is advertised in CPUID.
ucode needed	Mitigation is enabled. CPU is affected and MD_CLEAR is not advertised in CPUID. That is mainly for virtualization scenarios where the host has the updated microcode but the hypervisor does not expose MD_CLEAR in CPUID. It's a best effort approach without guarantee.

If the CPU is affected and the "tsx_async_abort" kernel command line parameter is not provided then the kernel selects an appropriate mitigation depending on the status of RTM and MD_CLEAR CPUID bits.

Below tables indicate the impact of tsx=on|off|auto cmdline options on state of TAA mitigation, VERW behavior and TSX feature for various combinations of MSR_IA32_ARCH_CAPABILITIES bits.

1. "tsx=off"

MSR_IA32_ARCH_CAPABILITIES bits			Result with cmdline tsx=off			
TAA_NO	MDS_NO	TSX_CTRL_M	TSX state after bootup	VERW can clear CPU buffers	TAA mitigation tsx_async_ab	TAA mitigation tsx_async_abort=fu
0	0	0	HW default	Yes	Same as MDS	Same as MDS
0	0	1	Invalid case	Invalid case	Invalid case	Invalid case
0	1	0	HW default	No	Need ucode update	Need ucode update
0	1	1	Disabled	Yes	TSX disabled	TSX disabled
1	X	1	Disabled	X	None needed	None needed

2. “tsx=on”

MSR_IA32_ARCH_CAPABILITIES bits			Result with cmdline tsx=on			
TAA_NO	MDS_NO	TSX_CTRL_M	TSX state after bootup	VERW can clear CPU buffers	TAA mitigation tsx_async_ab	TAA mitigation tsx_async_abort=fu
0	0	0	HW default	Yes	Same as MDS	Same as MDS
0	0	1	Invalid case	Invalid case	Invalid case	Invalid case
0	1	0	HW default	No	Need ucode update	Need ucode update
0	1	1	Enabled	Yes	None	Same as MDS
1	X	1	Enabled	X	None needed	None needed

3. “tsx=auto”

MSR_IA32_ARCH_CAPABILITIES bits			Result with cmdline tsx=auto			
TAA_NO	MDS_NO	TSX_CTRL_M	TSX state after bootup	VERW can clear CPU buffers	TAA mitigation tsx_async_ab	TAA mitigation tsx_async_abort=fu
0	0	0	HW default	Yes	Same as MDS	Same as MDS
0	0	1	Invalid case	Invalid case	Invalid case	Invalid case
0	1	0	HW default	No	Need ucode update	Need ucode update
0	1	1	Disabled	Yes	TSX disabled	TSX disabled
1	X	1	Enabled	X	None needed	None needed

In the tables, TSX_CTRL_MSR is a new bit in MSR_IA32_ARCH_CAPABILITIES that indicates whether MSR_IA32_TSX_CTRL is supported.

There are two control bits in IA32_TSX_CTRL MSR:

Bit 0: When set it disables the Restricted Transactional Memory (RTM)
sub-feature of TSX (will force all transactions to abort on the XBEGIN instruction).

Bit 1: When set it disables the enumeration of the RTM and HLE feature
(i.e. it will make CPUID(EAX=7).EBX{bit4} and CPUID(EAX=7).EBX{bit11} read as 0).

16.26 Bus lock detection and handling

Authors

- Fenghua Yu <fenghua.yu@intel.com>
- Tony Luck <tony.luck@intel.com>

16.26.1 Problem

A split lock is any atomic operation whose operand crosses two cache lines. Since the operand spans two cache lines and the operation must be atomic, the system locks the bus while the CPU accesses the two cache lines.

A bus lock is acquired through either split locked access to writeback (WB) memory or any locked access to non-WB memory. This is typically thousands of cycles slower than an atomic operation within a cache line. It also disrupts performance on other cores and brings the whole system to its knees.

16.26.2 Detection

Intel processors may support either or both of the following hardware mechanisms to detect split locks and bus locks.

#AC exception for split lock detection

Beginning with the Tremont Atom CPU split lock operations may raise an Alignment Check (#AC) exception when a split lock operation is attempted.

#DB exception for bus lock detection

Some CPUs have the ability to notify the kernel by an #DB trap after a user instruction acquires a bus lock and is executed. This allows the kernel to terminate the application or to enforce throttling.

16.26.3 Software handling

The kernel #AC and #DB handlers handle bus lock based on the kernel parameter "split_lock_detect". Here is a summary of different options:

split_lock_detect=	#AC for split lock	#DB for bus lock
off	Do nothing	Do nothing
warn (default)	Kernel OOPs Warn once per task, add a delay, add synchronization to prevent more than one core from executing a split lock in parallel. sysctl split_lock_mitigate can be used to avoid the delay and synchronization When both features are supported, warn in #AC	Warn once per task and and continues to run.
fatal	Kernel OOPs Send SIGBUS to user When both features are supported, fatal in #AC	Send SIGBUS to user.
ratelimit:N (0 < N <= 1000)	Do nothing	Limit bus lock rate to N bus locks per second system wide and warn on bus locks.

16.26.4 Usages

Detecting and handling bus lock may find usages in various areas:

It is critical for real time system designers who build consolidated real time systems. These systems run hard real time code on some cores and run "untrusted" user processes on other cores. The hard real time cannot afford to have any bus lock from the untrusted processes to hurt real time performance. To date the designers have been unable to deploy these solutions as they have no way to prevent the "untrusted" user code from generating split lock and bus lock to block the hard real time code to access memory during bus locking.

It's also useful for general computing to prevent guests or user applications from slowing down the overall system by executing instructions with bus lock.

16.26.5 Guidance

off

Disable checking for split lock and bus lock. This option can be useful if there are legacy applications that trigger these events at a low rate so that mitigation is not needed.

warn

A warning is emitted when a bus lock is detected which allows to identify the offending application. This is the default behavior.

fatal

In this case, the bus lock is not tolerated and the process is killed.

ratelimit

A system wide bus lock rate limit N is specified where 0 < N <= 1000. This allows a bus lock rate up to N bus locks per second. When the bus lock rate is exceeded then any task which is caught via the buslock #DB exception is throttled by enforced sleeps until the rate goes under the limit again.

This is an effective mitigation in cases where a minimal impact can be tolerated, but an eventual Denial of Service attack has to be prevented. It allows to identify the offending processes and analyze whether they are malicious or just badly written.

Selecting a rate limit of 1000 allows the bus to be locked for up to about seven million cycles each second (assuming 7000 cycles for each bus lock). On a 2 GHz processor that would be about 0.35% system slowdown.

16.27 USB Legacy support

Author
Vojtech Pavlik <vojtech@suse.cz>, January 2004

Also known as "USB Keyboard" or "USB Mouse support" in the BIOS Setup is a feature that allows one to use the USB mouse and keyboard as if they were their classic PS/2 counterparts. This means one can use an USB keyboard to type in LILO for example.

It has several drawbacks, though:

1) On some machines, the emulated PS/2 mouse takes over even when no USB mouse is present and a real PS/2 mouse is present. In that case the extra features (wheel, extra buttons, touchpad mode) of the real PS/2 mouse may not be available.
2) If CONFIG_HIGHMEM64G is enabled, the PS/2 mouse emulation can cause system crashes, because the SMM BIOS is not expecting to be in PAE mode. The Intel E7505 is a typical machine where this happens.
3) If AMD64 64-bit mode is enabled, again system crashes often happen, because the SMM BIOS isn't expecting the CPU to be in 64-bit mode. The BIOS manufacturers only test with Windows, and Windows doesn't do 64-bit yet.

Solutions:

Problem 1)
can be solved by loading the USB drivers prior to loading the PS/2 mouse driver. Since the PS/2 mouse driver is in 2.6 compiled into the kernel unconditionally, this means the USB drivers need to be compiled-in, too.

Problem 2)
can currently only be solved by either disabling HIGHMEM64G in the kernel config or USB Legacy support in the BIOS. A BIOS update could help, but so far no such update exists.

Problem 3)
is usually fixed by a BIOS update. Check the board manufacturers web site. If an update is not available, disable USB Legacy support in the BIOS. If this alone doesn't help, try also adding idle=poll on the kernel command line. The BIOS may be entering the SMM on the HLT instruction as well.

16.28 i386 Support

16.28.1 IO-APIC

Author
Ingo Molnar <mingo@kernel.org>

Most (all) Intel-MP compliant SMP boards have the so-called 'IO-APIC', which is an enhanced interrupt controller. It enables us to route hardware interrupts to multiple CPUs, or to CPU groups. Without an IO-APIC, interrupts from hardware will be delivered only to the CPU which boots the operating system (usually CPU#0).

Linux supports all variants of compliant SMP boards, including ones with multiple IO-APICs. Multiple IO-APICs are used in high-end servers to distribute IRQ load further.

There are (a few) known breakages in certain older boards, such bugs are usually worked around by the kernel. If your MP-compliant SMP board does not boot Linux, then consult the linux-smp mailing list archives first.

If your box boots fine with enabled IO-APIC IRQs, then your /proc/interrupts will look like this one:

```
hell:~> cat /proc/interrupts
           CPU0
  0:    1360293    IO-APIC-edge  timer
  1:          4    IO-APIC-edge  keyboard
  2:          0          XT-PIC  cascade
 13:          1          XT-PIC  fpu
 14:       1448    IO-APIC-edge  ide0
 16:      28232   IO-APIC-level  Intel EtherExpress Pro 10/100 Ethernet
 17:      51304   IO-APIC-level  eth0
NMI:          0
ERR:          0
hell:~>
```

Some interrupts are still listed as 'XT PIC', but this is not a problem; none of those IRQ sources is performance-critical.

In the unlikely case that your board does not create a working mp-table, you can use the pirq= boot parameter to 'hand-construct' IRQ entries. This is non-trivial though and cannot be automated. One sample /etc/lilo.conf entry:

```
append="pirq=15,11,10"
```

The actual numbers depend on your system, on your PCI cards and on their PCI slot position. Usually PCI slots are 'daisy chained' before they are connected to the PCI chipset IRQ routing facility (the incoming PIRQ1-4 lines):

```
          ,-.        ,-.        ,-.        ,-.        ,-.
PIRQ4 ----| |-.    ,-| |-.    ,-| |-.    ,-| |--------| |
          |S|  \  /  |S|  \  /  |S|  \  /  |S|        |S|
PIRQ3 ----|l|-. `/---|l|-. `/---|l|-. `/---|l|--------|l|
          |o|  \/    |o|  \/    |o|  \/    |o|        |o|
PIRQ2 ----|t|-./`----|t|-./`----|t|-./`----|t|--------|t|
          |1| /\     |2| /\     |3| /\     |4|        |5|
PIRQ1 ----| |-  `----| |-  `----| |-  `----| |--------| |
          `-'        `-'        `-'        `-'        `-'
```

Every PCI card emits a PCI IRQ, which can be INTA, INTB, INTC or INTD:

```
      ,-.
INTD--| |
      |S|
INTC--|l|
      |o|
INTB--|t|
      |x|
INTA--| |
      `-'
```

These INTA-D PCI IRQs are always 'local to the card', their real meaning depends on which slot they are in. If you look at the daisy chaining diagram, a card in slot4, issuing INTA IRQ, it will end up as a signal on PIRQ4 of the PCI chipset. Most cards issue INTA, this creates optimal distribution between the PIRQ lines. (distributing IRQ sources properly is not a necessity, PCI IRQs can be shared at will, but it's a good for performance to have non shared interrupts). Slot5 should be used for videocards, they do not use interrupts normally, thus they are not daisy chained either.

so if you have your SCSI card (IRQ11) in Slot1, Tulip card (IRQ9) in Slot2, then you'll have to specify this pirq= line:

```
append="pirq=11,9"
```

the following script tries to figure out such a default pirq= line from your PCI configuration:

```
echo -n pirq=; echo `scanpci | grep T_L | cut -c56-` | sed 's/ /,/g'
```

note that this script won't work if you have skipped a few slots or if your board does not do default daisy-chaining. (or the IO-APIC has the PIRQ pins connected in some strange way). E.g. if in the above case you have your SCSI card (IRQ11) in Slot3, and have Slot1 empty:

```
append="pirq=0,9,11"
```

[value '0' is a generic 'placeholder', reserved for empty (or non-IRQ emitting) slots.]

Generally, it's always possible to find out the correct pirq= settings, just permute all IRQ numbers properly … it will take some time though. An 'incorrect' pirq line will cause the booting process to hang, or a device won't function properly (e.g. if it's inserted as a module).

If you have 2 PCI buses, then you can use up to 8 pirq values, although such boards tend to have a good configuration.

Be prepared that it might happen that you need some strange pirq line:

```
append="pirq=0,0,0,0,0,0,9,11"
```

Use smart trial-and-error techniques to find out the correct pirq line …

Good luck and mail to linux-smp@vger.kernel.org or linux-kernel@vger.kernel.org if you have any problems that are not covered by this document.

16.29 x86_64 Support

16.29.1 AMD64 Specific Boot Options

There are many others (usually documented in driver documentation), but only the AMD64 specific ones are listed here.

Machine check

Please see *Configurable sysfs parameters for the x86-64 machine check code* for sysfs runtime tunables.

mce=off
Disable machine check

mce=no_cmci
Disable CMCI(Corrected Machine Check Interrupt) that Intel processor supports. Usually this disablement is not recommended, but it might be handy if your hardware is misbehaving. Note that you'll get more problems without CMCI than with due to the shared banks, i.e. you might get duplicated error logs.

mce=dont_log_ce
Don't make logs for corrected errors. All events reported as corrected are silently cleared by OS. This option will be useful if you have no interest in any of corrected errors.

mce=ignore_ce
Disable features for corrected errors, e.g. polling timer and CMCI. All events reported as corrected are not cleared by OS and remained in its error banks. Usually this disablement is not recommended, however if there is an agent checking/clearing corrected errors (e.g. BIOS or hardware monitoring applications), conflicting with OS's error handling, and you cannot deactivate the agent, then this option will be a help.

mce=no_lmce
Do not opt-in to Local MCE delivery. Use legacy method to broadcast MCEs.

mce=bootlog
Enable logging of machine checks left over from booting. Disabled by default on AMD Fam10h and older because some BIOS leave bogus ones. If your BIOS doesn't do that it's a good idea to enable though to make sure you log even machine check events that result in a reboot. On Intel systems it is enabled by default.

mce=nobootlog
Disable boot machine check logging.

mce=monarchtimeout (number)
monarchtimeout: Sets the time in us to wait for other CPUs on machine checks. 0 to disable.

mce=bios_cmci_threshold
Don't overwrite the bios-set CMCI threshold. This boot option prevents Linux from overwriting the CMCI threshold set by the bios. Without this option, Linux always sets the CMCI threshold to 1. Enabling this may make memory predictive failure analysis less effective if the bios sets thresholds for memory errors since we will not see details for all errors.

mce=recovery
Force-enable recoverable machine check code paths

nomce (for compatibility with i386)
same as mce=off

Everything else is in sysfs now.

APICs

apic
: Use IO-APIC. Default

noapic
: Don't use the IO-APIC.

disableapic
: Don't use the local APIC

nolapic
: Don't use the local APIC (alias for i386 compatibility)

pirq=...
: See *IO-APIC*

noapictimer
: Don't set up the APIC timer

no_timer_check
: Don't check the IO-APIC timer. This can work around problems with incorrect timer initialization on some boards.

apicpmtimer
: Do APIC timer calibration using the pmtimer. Implies apicmaintimer. Useful when your PIT timer is totally broken.

Timing

notsc
: Deprecated, use tsc=unstable instead.

nohpet
: Don't use the HPET timer.

Idle loop

idle=poll
: Don't do power saving in the idle loop using HLT, but poll for rescheduling event. This will make the CPUs eat a lot more power, but may be useful to get slightly better performance in multiprocessor benchmarks. It also makes some profiling using performance counters more accurate. Please note that on systems with MONITOR/MWAIT support (like Intel EM64T CPUs) this option has no performance advantage over the normal idle loop. It may also interact badly with hyperthreading.

Rebooting

reboot=b[ios] | t[riple] | k[bd] | a[cpi] | e[fi] | p[ci] [, [w]arm | [c]old]

bios
Use the CPU reboot vector for warm reset

warm
Don't set the cold reboot flag

cold
Set the cold reboot flag

triple
Force a triple fault (init)

kbd
Use the keyboard controller. cold reset (default)

acpi
Use the ACPI RESET_REG in the FADT. If ACPI is not configured or the ACPI reset does not work, the reboot path attempts the reset using the keyboard controller.

efi
Use efi reset_system runtime service. If EFI is not configured or the EFI reset does not work, the reboot path attempts the reset using the keyboard controller.

pci
Use a write to the PCI config space register 0xcf9 to trigger reboot.

Using warm reset will be much faster especially on big memory systems because the BIOS will not go through the memory check. Disadvantage is that not all hardware will be completely reinitialized on reboot so there may be boot problems on some systems.

reboot=force
Don't stop other CPUs on reboot. This can make reboot more reliable in some cases.

reboot=default
There are some built-in platform specific "quirks" - you may see: "reboot: <name> series board detected. Selecting <type> for reboots." In the case where you think the quirk is in error (e.g. you have newer BIOS, or newer board) using this option will ignore the built-in quirk table, and use the generic default reboot actions.

NUMA

numa=off
: Only set up a single NUMA node spanning all memory.

numa=noacpi
: Don't parse the SRAT table for NUMA setup

numa=nohmat
: Don't parse the HMAT table for NUMA setup, or soft-reserved memory partitioning.

numa=fake=<size>[MG]
: If given as a memory unit, fills all system RAM with nodes of size interleaved over physical nodes.

numa=fake=<N>
: If given as an integer, fills all system RAM with N fake nodes interleaved over physical nodes.

numa=fake=<N>U
: If given as an integer followed by 'U', it will divide each physical node into N emulated nodes.

ACPI

acpi=off
: Don't enable ACPI

acpi=ht
: Use ACPI boot table parsing, but don't enable ACPI interpreter

acpi=force
: Force ACPI on (currently not needed)

acpi=strict
: Disable out of spec ACPI workarounds.

acpi_sci={edge,level,high,low}
: Set up ACPI SCI interrupt.

acpi=noirq
: Don't route interrupts

acpi=nocmcff
: Disable firmware first mode for corrected errors. This disables parsing the HEST CMC error source to check if firmware has set the FF flag. This may result in duplicate corrected error reports.

PCI

pci=off
Don’t use PCI

pci=conf1
Use conf1 access.

pci=conf2
Use conf2 access.

pci=rom
Assign ROMs.

pci=assign-busses
Assign busses

pci=irqmask=MASK
Set PCI interrupt mask to MASK

pci=lastbus=NUMBER
Scan up to NUMBER busses, no matter what the mptable says.

pci=noacpi
Don’t use ACPI to set up PCI interrupt routing.

IOMMU (input/output memory management unit)

Multiple x86-64 PCI-DMA mapping implementations exist, for example:

1. <kernel/dma/direct.c>: use no hardware/software IOMMU at all (e.g. because you have < 3 GB memory). Kernel boot message: “PCI-DMA: Disabling IOMMU”
2. <arch/x86/kernel/amd_gart_64.c>: AMD GART based hardware IOMMU. Kernel boot message: “PCI-DMA: using GART IOMMU”
3. <arch/x86_64/kernel/pci-swiotlb.c> : Software IOMMU implementation. Used e.g. if there is no hardware IOMMU in the system and it is need because you have >3GB memory or told the kernel to us it (iommu=soft)) Kernel boot message: “PCI-DMA: Using software bounce buffering for IO (SWIOTLB)”

```
iommu=[<size>][,noagp][,off][,force][,noforce]
[,memaper[=<order>]][,merge][,fullflush][,nomerge]
[,noaperture]
```

General iommu options:

off
Don’t initialize and use any kind of IOMMU.

noforce
Don’t force hardware IOMMU usage when it is not needed. (default).

force
Force the use of the hardware IOMMU even when it is not actually needed (e.g. because < 3 GB memory).

soft
Use software bounce buffering (SWIOTLB) (default for Intel machines). This can be used to prevent the usage of an available hardware IOMMU.

iommu options only relevant to the AMD GART hardware IOMMU:

<size>
Set the size of the remapping area in bytes.

allowed
Overwrite iommu off workarounds for specific chipsets.

fullflush
Flush IOMMU on each allocation (default).

nofullflush
Don’t use IOMMU fullflush.

memaper[=<order>]
Allocate an own aperture over RAM with size 32MB<<order. (default: order=1, i.e. 64MB)

merge
Do scatter-gather (SG) merging. Implies “force” (experimental).

nomerge
Don’t do scatter-gather (SG) merging.

noaperture
Ask the IOMMU not to touch the aperture for AGP.

noagp
Don’t initialize the AGP driver and use full aperture.

panic
Always panic when IOMMU overflows.

iommu options only relevant to the software bounce buffering (SWIOTLB) IOMMU implementation:

swiotlb=<slots>[,force,noforce]

<slots>
Prereserve that many 2K slots for the software IO bounce buffering.

force
Force all IO through the software TLB.

noforce
Do not initialize the software TLB.

Miscellaneous

nogbpages
  Do not use GB pages for kernel direct mappings.

gbpages
  Use GB pages for kernel direct mappings.

AMD SEV (Secure Encrypted Virtualization)

Options relating to AMD SEV, specified via the following format:

```
sev=option1[,option2]
```

The available options are:

debug
  Enable debug messages.

16.29.2 General note on [U]EFI x86_64 support

The nomenclature EFI and UEFI are used interchangeably in this document.

Although the tools below are _not_ needed for building the kernel, the needed bootloader support and associated tools for x86_64 platforms with EFI firmware and specifications are listed below.

1. UEFI specification: http://www.uefi.org
2. Booting Linux kernel on UEFI x86_64 platform requires bootloader support. Elilo with x86_64 support can be used.
3. x86_64 platform with EFI/UEFI firmware.

Mechanics

- Build the kernel with the following configuration:

  ```
  CONFIG_FB_EFI=y
  CONFIG_FRAMEBUFFER_CONSOLE=y
  ```

  If EFI runtime services are expected, the following configuration should be selected:

  ```
  CONFIG_EFI=y
  CONFIG_EFIVAR_FS=y or m         # optional
  ```

- Create a VFAT partition on the disk
- Copy the following to the VFAT partition:

  elilo bootloader with x86_64 support, elilo configuration file, kernel image built in first step and corresponding initrd. Instructions on building elilo and its dependencies can be found in the elilo sourceforge project.

- Boot to EFI shell and invoke elilo choosing the kernel image built in first step.
- If some or all EFI runtime services don't work, you can try following kernel command line parameters to turn off some or all EFI runtime services.

  noefi
  turn off all EFI runtime services

  reboot_type=k
  turn off EFI reboot runtime service

- If the EFI memory map has additional entries not in the E820 map, you can include those entries in the kernels memory map of available physical RAM by using the following kernel command line parameter.

  add_efi_memmap
  include EFI memory map of available physical RAM

16.29.3 Memory Management

Complete virtual memory map with 4-level page tables

Note:

- Negative addresses such as "-23 TB" are absolute addresses in bytes, counted down from the top of the 64-bit address space. It's easier to understand the layout when seen both in absolute addresses and in distance-from-top notation.

  For example 0xffffe90000000000 == -23 TB, it's 23 TB lower than the top of the 64-bit address space (ffffffffffffffff).

  Note that as we get closer to the top of the address space, the notation changes from TB to GB and then MB/KB.

- "16M TB" might look weird at first sight, but it's an easier way to visualize size notation than "16 EB", which few will recognize at first sight as 16 exabytes. It also shows it nicely how incredibly large 64-bit address space is.

```
==========================================================================
   Start addr    |   Offset   |     End addr     |  Size   | VM area␣
↪description
==========================================================================
                 |            |                  |         |
 0000000000000000 |    0       | 00007fffffffffff |  128 TB | user-space␣
↪virtual memory, different per mm
_________________|____________|__________________|_________|__________________
↪_________________________________________
                 |            |                  |         |
 0000800000000000 | +128    TB | ffff7fffffffffff | ~16M TB | ... huge, almost␣
↪64 bits wide hole of non-canonical
                 |            |                  |         |     virtual␣
↪memory addresses up to the -128 TB
                 |            |                  |         |     starting␣
```

```
↪offset of kernel mappings.
__________________|____________|__________________|_________|___________________
↪_________________________________________
                                                            |
                                                            | Kernel-space␣
↪virtual memory, shared between all processes:
____________________________________________________________|___________________
↪_________________________________________
                  |            |                  |         |
 ffff800000000000 | -128    TB | ffff87ffffffffff |    8 TB | ... guard hole,␣
↪also reserved for hypervisor
 ffff880000000000 | -120    TB | ffff887fffffffff |  0.5 TB | LDT remap for PTI
 ffff888000000000 | -119.5  TB | ffffc87fffffffff |   64 TB | direct mapping␣
↪of all physical memory (page_offset_base)
 ffffc88000000000 |  -55.5  TB | ffffc8ffffffffff |  0.5 TB | ... unused hole
 ffffc90000000000 |  -55    TB | ffffe8ffffffffff |   32 TB | vmalloc/ioremap␣
↪space (vmalloc_base)
 ffffe90000000000 |  -23    TB | ffffe9ffffffffff |    1 TB | ... unused hole
 ffffea0000000000 |  -22    TB | ffffeaffffffffff |    1 TB | virtual memory␣
↪map (vmemmap_base)
 ffffeb0000000000 |  -21    TB | ffffebffffffffff |    1 TB | ... unused hole
 ffffec0000000000 |  -20    TB | fffffbffffffffff |   16 TB | KASAN shadow␣
↪memory
__________________|____________|__________________|_________|___________________
↪__________________________________________
                                                            |
                                                            | Identical layout␣
↪to the 56-bit one from here on:
____________________________________________________________|___________________
↪__________________________________________
                  |            |                  |         |
 fffffc0000000000 |   -4    TB | fffffdffffffffff |    2 TB | ... unused hole
                  |            |                  |         | vaddr_end for␣
↪KASLR
 fffffe0000000000 |   -2    TB | fffffe7fffffffff |  0.5 TB | cpu_entry_area␣
↪mapping
 fffffe8000000000 |   -1.5  TB | fffffeffffffffff |  0.5 TB | ... unused hole
 ffffff0000000000 |   -1    TB | ffffff7fffffffff |  0.5 TB | %esp fixup stacks
 ffffff8000000000 | -512    GB | ffffffeeffffffff |  444 GB | ... unused hole
 ffffffef00000000 |  -68    GB | fffffffeffffffff |   64 GB | EFI region␣
↪mapping space
 ffffffff00000000 |   -4    GB | ffffffff7fffffff |    2 GB | ... unused hole
 ffffffff80000000 |   -2    GB | ffffffff9fffffff |  512 MB | kernel text␣
↪mapping, mapped to physical address 0
 ffffffff80000000 |-2048    MB |                  |         |
 ffffffffa0000000 |-1536    MB | fffffffffeffffff | 1520 MB | module mapping␣
↪space
 ffffffffff000000 |  -16    MB |                  |         |
    FIXADDR_START | ~-11    MB | ffffffffff5fffff | ~0.5 MB | kernel-internal␣
↪fixmap range, variable size and offset
```

```
 ffffffffff600000 |  -10    MB | ffffffffff600fff |    4 kB | legacy vsyscall␣
↪ABI
 ffffffffffe00000 |   -2    MB | ffffffffffffffff |    2 MB | ... unused hole
__________________|____________|__________________|_________|___________________
↪__________________________________________
```

Complete virtual memory map with 5-level page tables

Note:

- With 56-bit addresses, user-space memory gets expanded by a factor of 512x, from 0.125 PB to 64 PB. All kernel mappings shift down to the -64 PB starting offset and many of the regions expand to support the much larger physical memory supported.

```
========================================================================================
    Start addr    |   Offset   |     End addr     |  Size   | VM area␣
↪description
========================================================================================
                  |            |                  |         |
 0000000000000000 |    0       | 00ffffffffffffff |   64 PB | user-space␣
↪virtual memory, different per mm
__________________|____________|__________________|_________|___________________
↪__________________________________________
                  |            |                  |         |
 0100000000000000 |  +64    PB | feffffffffffffff | ~16K PB | ... huge, still␣
↪almost 64 bits wide hole of non-canonical
                  |            |                  |         |     virtual␣
↪memory addresses up to the -64 PB
                  |            |                  |         |     starting␣
↪offset of kernel mappings.
__________________|____________|__________________|_________|___________________
↪__________________________________________
                                                            |
                                                            | Kernel-space␣
↪virtual memory, shared between all processes:
____________________________________________________________|___________________
↪__________________________________________
                  |            |                  |         |
 ff00000000000000 |  -64    PB | ff0fffffffffffff |    4 PB | ... guard hole,␣
↪also reserved for hypervisor
 ff10000000000000 |  -60    PB | ff10ffffffffffff | 0.25 PB | LDT remap for PTI
 ff11000000000000 |  -59.75 PB | ff90ffffffffffff |   32 PB | direct mapping␣
↪of all physical memory (page_offset_base)
 ff91000000000000 |  -27.75 PB | ff9fffffffffffff | 3.75 PB | ... unused hole
 ffa0000000000000 |  -24    PB | ffd1ffffffffffff | 12.5 PB | vmalloc/ioremap␣
↪space (vmalloc_base)
 ffd2000000000000 |  -11.5  PB | ffd3ffffffffffff |  0.5 PB | ... unused hole
 ffd4000000000000 |  -11    PB | ffd5ffffffffffff |  0.5 PB | virtual memory␣
```

```
↪map (vmemmap_base)
 ffd6000000000000 |  -10.5  PB | ffdeffffffffffff | 2.25 PB | ... unused hole
 ffdf000000000000 |   -8.25 PB | fffffbffffffffff |   ~8 PB | KASAN shadow␣
↪memory
__________________|____________|__________________|_________|___________________
↪___________________________________________
                                                            |
                                                            | Identical layout␣
↪to the 47-bit one from here on:
____________________________________________________________|___________________
↪___________________________________________
                  |            |                  |         |
 fffffc0000000000 |   -4    TB | fffffdffffffffff |    2 TB | ... unused hole
                  |            |                  |         | vaddr_end for␣
↪KASLR
 fffffe0000000000 |   -2    TB | fffffe7fffffffff |  0.5 TB | cpu_entry_area␣
↪mapping
 fffffe8000000000 |   -1.5  TB | fffffeffffffffff |  0.5 TB | ... unused hole
 ffffff0000000000 |   -1    TB | ffffff7fffffffff |  0.5 TB | %esp fixup stacks
 ffffff8000000000 | -512    GB | ffffffeeffffffff |  444 GB | ... unused hole
 ffffffef00000000 |  -68    GB | fffffffeffffffff |   64 GB | EFI region␣
↪mapping space
 ffffffff00000000 |   -4    GB | ffffffff7fffffff |    2 GB | ... unused hole
 ffffffff80000000 |   -2    GB | ffffffff9fffffff |  512 MB | kernel text␣
↪mapping, mapped to physical address 0
 ffffffff80000000 |-2048    MB |                  |         |
 ffffffffa0000000 |-1536    MB | fffffffffeffffff | 1520 MB | module mapping␣
↪space
 ffffffffff000000 |  -16    MB |                  |         |
    FIXADDR_START | ~-11    MB | ffffffffff5fffff | ~0.5 MB | kernel-internal␣
↪fixmap range, variable size and offset
 ffffffffff600000 |  -10    MB | ffffffffff600fff |    4 kB | legacy vsyscall␣
↪ABI
 ffffffffffe00000 |   -2    MB | ffffffffffffffff |    2 MB | ... unused hole
__________________|____________|__________________|_________|___________________
↪__________________________________________
```

Architecture defines a 64-bit virtual address. Implementations can support less. Currently supported are 48- and 57-bit virtual addresses. Bits 63 through to the most-significant implemented bit are sign extended. This causes hole between user space and kernel addresses if you interpret them as unsigned.

The direct mapping covers all memory in the system up to the highest memory address (this means in some cases it can also include PCI memory holes).

We map EFI runtime services in the 'efi_pgd' PGD in a 64GB large virtual memory window (this size is arbitrary, it can be raised later if needed). The mappings are not part of any other kernel PGD and are only available during EFI runtime calls.

Note that if CONFIG_RANDOMIZE_MEMORY is enabled, the direct mapping of all physical memory, vmalloc/ioremap space and virtual memory map are randomized. Their order is preserved but their base will be offset early at boot time.

Be very careful vs. KASLR when changing anything here. The KASLR address range must not overlap with anything except the KASAN shadow area, which is correct as KASAN disables KASLR.

For both 4- and 5-level layouts, the STACKLEAK_POISON value in the last 2MB hole: ffffffffffff4111

16.29.4 5-level paging

Overview

Original x86-64 was limited by 4-level paging to 256 TiB of virtual address space and 64 TiB of physical address space. We are already bumping into this limit: some vendors offer servers with 64 TiB of memory today.

To overcome the limitation upcoming hardware will introduce support for 5-level paging. It is a straight-forward extension of the current page table structure adding one more layer of translation.

It bumps the limits to 128 PiB of virtual address space and 4 PiB of physical address space. This "ought to be enough for anybody" ©.

QEMU 2.9 and later support 5-level paging.

Virtual memory layout for 5-level paging is described in *Memory Management*

Enabling 5-level paging

CONFIG_X86_5LEVEL=y enables the feature.

Kernel with CONFIG_X86_5LEVEL=y still able to boot on 4-level hardware. In this case additional page table level -- p4d -- will be folded at runtime.

User-space and large virtual address space

On x86, 5-level paging enables 56-bit userspace virtual address space. Not all user space is ready to handle wide addresses. It's known that at least some JIT compilers use higher bits in pointers to encode their information. It collides with valid pointers with 5-level paging and leads to crashes.

To mitigate this, we are not going to allocate virtual address space above 47-bit by default.

But userspace can ask for allocation from full address space by specifying hint address (with or without MAP_FIXED) above 47-bits.

If hint address set above 47-bit, but MAP_FIXED is not specified, we try to look for unmapped area by specified address. If it's already occupied, we look for unmapped area in *full* address space, rather than from 47-bit window.

A high hint address would only affect the allocation in question, but not any future mmap()s.

Specifying high hint address on older kernel or on machine without 5-level paging support is safe. The hint will be ignored and kernel will fall back to allocation from 47-bit address space.

This approach helps to easily make application's memory allocator aware about large address space without manually tracking allocated virtual address space.

One important case we need to handle here is interaction with MPX. MPX (without MAWA extension) cannot handle addresses above 47-bit, so we need to make sure that MPX cannot be enabled we already have VMA above the boundary and forbid creating such VMAs once MPX is enabled.

16.29.5 Fake NUMA For CPUSets

Author
David Rientjes <rientjes@cs.washington.edu>

Using numa=fake and CPUSets for Resource Management

This document describes how the numa=fake x86_64 command-line option can be used in conjunction with cpusets for coarse memory management. Using this feature, you can create fake NUMA nodes that represent contiguous chunks of memory and assign them to cpusets and their attached tasks. This is a way of limiting the amount of system memory that are available to a certain class of tasks.

For more information on the features of cpusets, see Documentation/admin-guide/cgroup-v1/cpusets.rst. There are a number of different configurations you can use for your needs. For more information on the numa=fake command line option and its various ways of configuring fake nodes, see *AMD64 Specific Boot Options*.

For the purposes of this introduction, we'll assume a very primitive NUMA emulation setup of "numa=fake=4*512,". This will split our system memory into four equal chunks of 512M each that we can now use to assign to cpusets. As you become more familiar with using this combination for resource control, you'll determine a better setup to minimize the number of nodes you have to deal with.

A machine may be split as follows with "numa=fake=4*512," as reported by dmesg:

```
Faking node 0 at 0000000000000000-0000000020000000 (512MB)
Faking node 1 at 0000000020000000-0000000040000000 (512MB)
Faking node 2 at 0000000040000000-0000000060000000 (512MB)
Faking node 3 at 0000000060000000-0000000080000000 (512MB)
...
On node 0 totalpages: 130975
On node 1 totalpages: 131072
On node 2 totalpages: 131072
On node 3 totalpages: 131072
```

Now following the instructions for mounting the cpusets filesystem from Documentation/admin-guide/cgroup-v1/cpusets.rst, you can assign fake nodes (i.e. contiguous memory address spaces) to individual cpusets:

```
[root@xroads /]# mkdir exampleset
[root@xroads /]# mount -t cpuset none exampleset
[root@xroads /]# mkdir exampleset/ddset
[root@xroads /]# cd exampleset/ddset
[root@xroads /exampleset/ddset]# echo 0-1 > cpus
[root@xroads /exampleset/ddset]# echo 0-1 > mems
```

Now this cpuset, 'ddset', will only allowed access to fake nodes 0 and 1 for memory allocations (1G).

You can now assign tasks to these cpusets to limit the memory resources available to them according to the fake nodes assigned as mems:

```
[root@xroads /exampleset/ddset]# echo $$ > tasks
[root@xroads /exampleset/ddset]# dd if=/dev/zero of=tmp bs=1024 count=1G
[1] 13425
```

Notice the difference between the system memory usage as reported by /proc/meminfo between the restricted cpuset case above and the unrestricted case (i.e. running the same 'dd' command without assigning it to a fake NUMA cpuset):

Name	Unrestricted	Restricted
MemTotal	3091900 kB	3091900 kB
MemFree	42113 kB	1513236 kB

This allows for coarse memory management for the tasks you assign to particular cpusets. Since cpusets can form a hierarchy, you can create some pretty interesting combinations of use-cases for various classes of tasks for your memory management needs.

16.29.6 Firmware support for CPU hotplug under Linux/x86-64

Linux/x86-64 supports CPU hotplug now. For various reasons Linux wants to know in advance of boot time the maximum number of CPUs that could be plugged into the system. ACPI 3.0 currently has no official way to supply this information from the firmware to the operating system.

In ACPI each CPU needs an LAPIC object in the MADT table (5.2.11.5 in the ACPI 3.0 specification). ACPI already has the concept of disabled LAPIC objects by setting the Enabled bit in the LAPIC object to zero.

For CPU hotplug Linux/x86-64 expects now that any possible future hotpluggable CPU is already available in the MADT. If the CPU is not available yet it should have its LAPIC Enabled bit set to 0. Linux will use the number of disabled LAPICs to compute the maximum number of future CPUs.

In the worst case the user can overwrite this choice using a command line option (additional_cpus=...), but it is recommended to supply the correct number (or a reasonable approximation of it, with erring towards more not less) in the MADT to avoid manual configuration.

16.29.7 Configurable sysfs parameters for the x86-64 machine check code

Machine checks report internal hardware error conditions detected by the CPU. Uncorrected errors typically cause a machine check (often with panic), corrected ones cause a machine check log entry.

Machine checks are organized in banks (normally associated with a hardware subsystem) and subevents in a bank. The exact meaning of the banks and subevent is CPU specific.

mcelog knows how to decode them.

When you see the "Machine check errors logged" message in the system log then mcelog should run to collect and decode machine check entries from /dev/mcelog. Normally mcelog should be run regularly from a cronjob.

Each CPU has a directory in /sys/devices/system/machinecheck/machinecheckN (N = CPU number).

The directory contains some configurable entries. See Documentation/ABI/testing/sysfs-mce for more details.

TBD document entries for AMD threshold interrupt configuration

For more details about the x86 machine check architecture see the Intel and AMD architecture manuals from their developer websites.

For more details about the architecture see http://one.firstfloor.org/~andi/mce.pdf

16.29.8 Using FS and GS segments in user space applications

The x86 architecture supports segmentation. Instructions which access memory can use segment register based addressing mode. The following notation is used to address a byte within a segment:

> Segment-register:Byte-address

The segment base address is added to the Byte-address to compute the resulting virtual address which is accessed. This allows to access multiple instances of data with the identical Byte-address, i.e. the same code. The selection of a particular instance is purely based on the base-address in the segment register.

In 32-bit mode the CPU provides 6 segments, which also support segment limits. The limits can be used to enforce address space protections.

In 64-bit mode the CS/SS/DS/ES segments are ignored and the base address is always 0 to provide a full 64bit address space. The FS and GS segments are still functional in 64-bit mode.

Common FS and GS usage

The FS segment is commonly used to address Thread Local Storage (TLS). FS is usually managed by runtime code or a threading library. Variables declared with the '__thread' storage class specifier are instantiated per thread and the compiler emits the FS: address prefix for accesses to these variables. Each thread has its own FS base address so common code can be used without complex address offset calculations to access the per thread instances. Applications should not use FS for other purposes when they use runtimes or threading libraries which manage the per thread FS.

The GS segment has no common use and can be used freely by applications. GCC and Clang support GS based addressing via address space identifiers.

Reading and writing the FS/GS base address

There exist two mechanisms to read and write the FS/GS base address:

- the arch_prctl() system call
- the FSGSBASE instruction family

Accessing FS/GS base with arch_prctl()

The arch_prctl(2) based mechanism is available on all 64-bit CPUs and all kernel versions.

Reading the base:

arch_prctl(ARCH_GET_FS, &fsbase); arch_prctl(ARCH_GET_GS, &gsbase);

Writing the base:

arch_prctl(ARCH_SET_FS, fsbase); arch_prctl(ARCH_SET_GS, gsbase);

The ARCH_SET_GS prctl may be disabled depending on kernel configuration and security settings.

Accessing FS/GS base with the FSGSBASE instructions

With the Ivy Bridge CPU generation Intel introduced a new set of instructions to access the FS and GS base registers directly from user space. These instructions are also supported on AMD Family 17H CPUs. The following instructions are available:

RDFSBASE %reg	Read the FS base register
RDGSBASE %reg	Read the GS base register
WRFSBASE %reg	Write the FS base register
WRGSBASE %reg	Write the GS base register

The instructions avoid the overhead of the arch_prctl() syscall and allow more flexible usage of the FS/GS addressing modes in user space applications. This does not prevent conflicts between threading libraries and runtimes which utilize FS and applications which want to use it for their own purpose.

FSGSBASE instructions enablement

The instructions are enumerated in CPUID leaf 7, bit 0 of EBX. If available /proc/cpuinfo shows 'fsgsbase' in the flag entry of the CPUs.

The availability of the instructions does not enable them automatically. The kernel has to enable them explicitly in CR4. The reason for this is that older kernels make assumptions about the values in the GS register and enforce them when GS base is set via arch_prctl(). Allowing user space to write arbitrary values to GS base would violate these assumptions and cause malfunction.

On kernels which do not enable FSGSBASE the execution of the FSGSBASE instructions will fault with a #UD exception.

The kernel provides reliable information about the enabled state in the ELF AUX vector. If the HWCAP2_FSGSBASE bit is set in the AUX vector, the kernel has FSGSBASE instructions enabled and applications can use them. The following code example shows how this detection works:

```
#include <sys/auxv.h>
#include <elf.h>

/* Will be eventually in asm/hwcap.h */
#ifndef HWCAP2_FSGSBASE
#define HWCAP2_FSGSBASE        (1 << 1)
#endif

....

unsigned val = getauxval(AT_HWCAP2);

if (val & HWCAP2_FSGSBASE)
     printf("FSGSBASE enabled\n");
```

FSGSBASE instructions compiler support

GCC version 4.6.4 and newer provide instrinsics for the FSGSBASE instructions. Clang 5 supports them as well.

_readfsbase_u64()	Read the FS base register
_readfsbase_u64()	Read the GS base register
_writefsbase_u64()	Write the FS base register
_writegsbase_u64()	Write the GS base register

To utilize these instrinsics <immintrin.h> must be included in the source code and the compiler option -mfsgsbase has to be added.

Compiler support for FS/GS based addressing

GCC version 6 and newer provide support for FS/GS based addressing via Named Address Spaces. GCC implements the following address space identifiers for x86:

__seg_fs	Variable is addressed relative to FS
__seg_gs	Variable is addressed relative to GS

The preprocessor symbols __SEG_FS and __SEG_GS are defined when these address spaces are supported. Code which implements fallback modes should check whether these symbols are defined. Usage example:

```
#ifdef __SEG_GS

long data0 = 0;
long data1 = 1;

long __seg_gs *ptr;

/* Check whether FSGSBASE is enabled by the kernel (HWCAP2_FSGSBASE) */
....

/* Set GS base to point to data0 */
_writegsbase_u64(&data0);

/* Access offset 0 of GS */
ptr = 0;
printf("data0 = %ld\n", *ptr);

/* Set GS base to point to data1 */
_writegsbase_u64(&data1);
/* ptr still addresses offset 0! */
printf("data1 = %ld\n", *ptr);
```

Clang does not provide the GCC address space identifiers, but it provides address spaces via an attribute based mechanism in Clang 2.6 and newer versions:

__attribute__((address_space(256))	Variable is addressed relative to GS
__attribute__((address_space(257))	Variable is addressed relative to FS

FS/GS based addressing with inline assembly

In case the compiler does not support address spaces, inline assembly can be used for FS/GS based addressing mode:

```
mov %fs:offset, %reg
mov %gs:offset, %reg

mov %reg, %fs:offset
mov %reg, %gs:offset
```

In-Field Scan

16.30 In-Field Scan

16.30.1 Introduction

In Field Scan (IFS) is a hardware feature to run circuit level tests on a CPU core to detect problems that are not caught by parity or ECC checks. Future CPUs will support more than one type of test which will show up with a new platform-device instance-id.

16.30.2 IFS Image

Intel provides a firmware file containing the scan tests via github[1]. Similar to microcode there is a separate file for each family-model-stepping. IFS Images are not applicable for some test types. Wherever applicable the sysfs directory would provide a “current_batch” file (see below) for loading the image.

16.30.3 IFS Image Loading

The driver loads the tests into memory reserved BIOS local to each CPU socket in a two step process using writes to MSRs to first load the SHA hashes for the test. Then the tests themselves. Status MSRs provide feedback on the success/failure of these steps.

The test files are kept in a fixed location: /lib/firmware/intel/ifs_<n>/ For e.g if there are 3 test files, they would be named in the following fashion: ff-mm-ss-01.scan ff-mm-ss-02.scan ff-mm-ss-03.scan (where ff refers to family, mm indicates model and ss indicates stepping)

A different test file can be loaded by writing the numerical portion (e.g 1, 2 or 3 in the above scenario) into the curent_batch file. To load ff-mm-ss-02.scan, the following command can be used:

```
# echo 2 > /sys/devices/virtual/misc/intel_ifs_<n>/current_batch
```

The above file can also be read to know the currently loaded image.

[1] https://github.com/intel/TBD

16.30.4 Running tests

Tests are run by the driver synchronizing execution of all threads on a core and then writing to the ACTIVATE_SCAN MSR on all threads. Instruction execution continues when:

1) All tests have completed.
2) Execution was interrupted.
3) A test detected a problem.

Note that ALL THREADS ON THE CORE ARE EFFECTIVELY OFFLINE FOR THE DURATION OF THE TEST. This can be up to 200 milliseconds. If the system is running latency sensitive applications that cannot tolerate an interruption of this magnitude, the system administrator must arrange to migrate those applications to other cores before running a core test. It may also be necessary to redirect interrupts to other CPUs.

In all cases reading the corresponding test's STATUS MSR provides details on what happened. The driver makes the value of this MSR visible to applications via the "details" file (see below). Interrupted tests may be restarted.

The IFS driver provides sysfs interfaces via /sys/devices/virtual/misc/intel_ifs_<n>/ to control execution:

Test a specific core:

```
# echo <cpu#> > /sys/devices/virtual/misc/intel_ifs_<n>/run_test
```

when HT is enabled any of the sibling cpu# can be specified to test its corresponding physical core. Since the tests are per physical core, the result of testing any thread is same. All siblings must be online to run a core test. It is only necessary to test one thread.

For e.g. to test core corresponding to cpu5

> # echo 5 > /sys/devices/virtual/misc/intel_ifs_<n>/run_test

Results of the last test is provided in /sys:

```
$ cat /sys/devices/virtual/misc/intel_ifs_<n>/status
pass
```

Status can be one of pass, fail, untested

Additional details of the last test is provided by the details file:

```
$ cat /sys/devices/virtual/misc/intel_ifs_<n>/details
0x8081
```

The details file reports the hex value of the test specific status MSR. Hardware defined error codes are documented in volume 4 of the Intel Software Developer's Manual but the error_code field may contain one of the following driver defined software codes:

0xFD	Software timeout
0xFE	Partial completion

16.30.5 Driver design choices

1) The ACTIVATE_SCAN MSR allows for running any consecutive subrange of available tests. But the driver always tries to run all tests and only uses the subrange feature to restart an interrupted test.

2) Hardware allows for some number of cores to be tested in parallel. The driver does not make use of this, it only tests one core at a time.

struct **ifs_data**

attributes related to intel IFS driver

Definition:

```
struct ifs_data {
    int loaded_version;
    bool loaded;
    bool loading_error;
    int valid_chunks;
    int status;
    u64 scan_details;
    u32 cur_batch;
};
```

Members

loaded_version
stores the currently loaded ifs image version.

loaded
If a valid test binary has been loaded into the memory

loading_error
Error occurred on another CPU while loading image

valid_chunks
number of chunks which could be validated.

status
it holds simple status pass/fail/untested

scan_details
opaque scan status code from h/w

cur_batch
number indicating the currently loaded test file

16.31 Shared Virtual Addressing (SVA) with ENQCMD

16.31.1 Background

Shared Virtual Addressing (SVA) allows the processor and device to use the same virtual addresses avoiding the need for software to translate virtual addresses to physical addresses. SVA is what PCIe calls Shared Virtual Memory (SVM).

In addition to the convenience of using application virtual addresses by the device, it also doesn't require pinning pages for DMA. PCIe Address Translation Services (ATS) along with Page Request Interface (PRI) allow devices to function much the same way as the CPU handling application page-faults. For more information please refer to the PCIe specification Chapter 10: ATS Specification.

Use of SVA requires IOMMU support in the platform. IOMMU is also required to support the PCIe features ATS and PRI. ATS allows devices to cache translations for virtual addresses. The IOMMU driver uses the mmu_notifier() support to keep the device TLB cache and the CPU cache in sync. When an ATS lookup fails for a virtual address, the device should use the PRI in order to request the virtual address to be paged into the CPU page tables. The device must use ATS again in order the fetch the translation before use.

16.31.2 Shared Hardware Workqueues

Unlike Single Root I/O Virtualization (SR-IOV), Scalable IOV (SIOV) permits the use of Shared Work Queues (SWQ) by both applications and Virtual Machines (VM's). This allows better hardware utilization vs. hard partitioning resources that could result in under utilization. In order to allow the hardware to distinguish the context for which work is being executed in the hardware by SWQ interface, SIOV uses Process Address Space ID (PASID), which is a 20-bit number defined by the PCIe SIG.

PASID value is encoded in all transactions from the device. This allows the IOMMU to track I/O on a per-PASID granularity in addition to using the PCIe Resource Identifier (RID) which is the Bus/Device/Function.

16.31.3 ENQCMD

ENQCMD is a new instruction on Intel platforms that atomically submits a work descriptor to a device. The descriptor includes the operation to be performed, virtual addresses of all parameters, virtual address of a completion record, and the PASID (process address space ID) of the current process.

ENQCMD works with non-posted semantics and carries a status back if the command was accepted by hardware. This allows the submitter to know if the submission needs to be retried or other device specific mechanisms to implement fairness or ensure forward progress should be provided.

ENQCMD is the glue that ensures applications can directly submit commands to the hardware and also permits hardware to be aware of application context to perform I/O operations via use of PASID.

16.31.4 Process Address Space Tagging

A new thread-scoped MSR (IA32_PASID) provides the connection between user processes and the rest of the hardware. When an application first accesses an SVA-capable device, this MSR is initialized with a newly allocated PASID. The driver for the device calls an IOMMU-specific API that sets up the routing for DMA and page-requests.

For example, the Intel Data Streaming Accelerator (DSA) uses iommu_sva_bind_device(), which will do the following:

- Allocate the PASID, and program the process page-table (%cr3 register) in the PASID context entries.
- Register for mmu_notifier() to track any page-table invalidations to keep the device TLB in sync. For example, when a page-table entry is invalidated, the IOMMU propagates the invalidation to the device TLB. This will force any future access by the device to this virtual address to participate in ATS. If the IOMMU responds with proper response that a page is not present, the device would request the page to be paged in via the PCIe PRI protocol before performing I/O.

This MSR is managed with the XSAVE feature set as “supervisor state” to ensure the MSR is updated during context switch.

16.31.5 PASID Management

The kernel must allocate a PASID on behalf of each process which will use ENQCMD and program it into the new MSR to communicate the process identity to platform hardware. ENQCMD uses the PASID stored in this MSR to tag requests from this process. When a user submits a work descriptor to a device using the ENQCMD instruction, the PASID field in the descriptor is auto-filled with the value from MSR_IA32_PASID. Requests for DMA from the device are also tagged with the same PASID. The platform IOMMU uses the PASID in the transaction to perform address translation. The IOMMU APIs setup the corresponding PASID entry in IOMMU with the process address used by the CPU (e.g. %cr3 register in x86).

The MSR must be configured on each logical CPU before any application thread can interact with a device. Threads that belong to the same process share the same page tables, thus the same MSR value.

16.31.6 PASID Life Cycle Management

PASID is initialized as IOMMU_PASID_INVALID (-1) when a process is created.

Only processes that access SVA-capable devices need to have a PASID allocated. This allocation happens when a process opens/binds an SVA-capable device but finds no PASID for this process. Subsequent binds of the same, or other devices will share the same PASID.

Although the PASID is allocated to the process by opening a device, it is not active in any of the threads of that process. It’s loaded to the IA32_PASID MSR lazily when a thread tries to submit a work descriptor to a device using the ENQCMD.

That first access will trigger a #GP fault because the IA32_PASID MSR has not been initialized with the PASID value assigned to the process when the device was opened. The Linux #GP handler notes that a PASID has been allocated for the process, and so initializes the IA32_PASID MSR and returns so that the ENQCMD instruction is re-executed.

On fork(2) or exec(2) the PASID is removed from the process as it no longer has the same address space that it had when the device was opened.

On clone(2) the new task shares the same address space, so will be able to use the PASID allocated to the process. The IA32_PASID is not preemptively initialized as the PASID value might not be allocated yet or the kernel does not know whether this thread is going to access the device and the cleared IA32_PASID MSR reduces context switch overhead by xstate init optimization. Since #GP faults have to be handled on any threads that were created before the PASID was assigned to the mm of the process, newly created threads might as well be treated in a consistent way.

Due to complexity of freeing the PASID and clearing all IA32_PASID MSRs in all threads in unbind, free the PASID lazily only on mm exit.

If a process does a close(2) of the device file descriptor and munmap(2) of the device MMIO portal, then the driver will unbind the device. The PASID is still marked VALID in the PASID_MSR for any threads in the process that accessed the device. But this is harmless as without the MMIO portal they cannot submit new work to the device.

16.31.7 Relationships

- Each process has many threads, but only one PASID.
- Devices have a limited number (~10's to 1000's) of hardware workqueues. The device driver manages allocating hardware workqueues.
- A single mmap() maps a single hardware workqueue as a "portal" and each portal maps down to a single workqueue.
- For each device with which a process interacts, there must be one or more mmap()'d portals.
- Many threads within a process can share a single portal to access a single device.
- Multiple processes can separately mmap() the same portal, in which case they still share one device hardware workqueue.
- The single process-wide PASID is used by all threads to interact with all devices. There is not, for instance, a PASID for each thread or each thread<->device pair.

16.31.8 FAQ

- What is SVA/SVM?

Shared Virtual Addressing (SVA) permits I/O hardware and the processor to work in the same address space, i.e., to share it. Some call it Shared Virtual Memory (SVM), but Linux community wanted to avoid confusing it with POSIX Shared Memory and Secure Virtual Machines which were terms already in circulation.

- What is a PASID?

A Process Address Space ID (PASID) is a PCIe-defined Transaction Layer Packet (TLP) prefix. A PASID is a 20-bit number allocated and managed by the OS. PASID is included in all transactions between the platform and the device.

- How are shared workqueues different?

Traditionally, in order for userspace applications to interact with hardware, there is a separate hardware instance required per process. For example, consider doorbells as a mechanism of informing hardware about work to process. Each doorbell is required to be spaced 4k (or page-size) apart for process isolation. This requires hardware to provision that space and reserve it in MMIO. This doesn't scale as the number of threads becomes quite large. The hardware also manages the queue depth for Shared Work Queues (SWQ), and consumers don't need to track queue depth. If there is no space to accept a command, the device will return an error indicating retry.

A user should check Deferrable Memory Write (DMWr) capability on the device and only submits ENQCMD when the device supports it. In the new DMWr PCIe terminology, devices need to support DMWr completer capability. In addition, it requires all switch ports to support DMWr routing and must be enabled by the PCIe subsystem, much like how PCIe atomic operations are managed for instance.

SWQ allows hardware to provision just a single address in the device. When used with ENQCMD to submit work, the device can distinguish the process submitting the work since it will include the PASID assigned to that process. This helps the device scale to a large number of processes.

- Is this the same as a user space device driver?

Communicating with the device via the shared workqueue is much simpler than a full blown user space driver. The kernel driver does all the initialization of the hardware. User space only needs to worry about submitting work and processing completions.

- Is this the same as SR-IOV?

Single Root I/O Virtualization (SR-IOV) focuses on providing independent hardware interfaces for virtualizing hardware. Hence, it's required to be almost fully functional interface to software supporting the traditional BARs, space for interrupts via MSI-X, its own register layout. Virtual Functions (VFs) are assisted by the Physical Function (PF) driver.

Scalable I/O Virtualization builds on the PASID concept to create device instances for virtualization. SIOV requires host software to assist in creating virtual devices; each virtual device is represented by a PASID along with the bus/device/function of the device. This allows device hardware to optimize device resource creation and can grow dynamically on demand. SR-IOV creation and management is very static in nature. Consult references below for more details.

- Why not just create a virtual function for each app?

Creating PCIe SR-IOV type Virtual Functions (VF) is expensive. VFs require duplicated hardware for PCI config space and interrupts such as MSI-X. Resources such as interrupts have to be hard partitioned between VFs at creation time, and cannot scale dynamically on demand. The VFs are not completely independent from the Physical Function (PF). Most VFs require some communication and assistance from the PF driver. SIOV, in contrast, creates a software-defined device where all the configuration and control aspects are mediated via the slow path. The work submission and completion happen without any mediation.

- Does this support virtualization?

ENQCMD can be used from within a guest VM. In these cases, the VMM helps with setting up a translation table to translate from Guest PASID to Host PASID. Please consult the ENQCMD instruction set reference for more details.

- Does memory need to be pinned?

When devices support SVA along with platform hardware such as IOMMU supporting such devices, there is no need to pin memory for DMA purposes. Devices that support SVA also support other PCIe features that remove the pinning requirement for memory.

Device TLB support - Device requests the IOMMU to lookup an address before use via Address Translation Service (ATS) requests. If the mapping exists but there is no page allocated by the OS, IOMMU hardware returns that no mapping exists.

Device requests the virtual address to be mapped via Page Request Interface (PRI). Once the OS has successfully completed the mapping, it returns the response back to the device. The device requests again for a translation and continues.

IOMMU works with the OS in managing consistency of page-tables with the device. When removing pages, it interacts with the device to remove any device TLB entry that might have been cached before removing the mappings from the OS.

16.31.9 References

VT-D: https://01.org/blogs/ashokraj/2018/recent-enhancements-intel-virtualization-technology-directe
o-intel-vt-d

SIOV: https://01.org/blogs/2019/assignable-interfaces-intel-scalable-i/o-virtualization-linux

ENQCMD in ISE: https://software.intel.com/sites/default/files/managed/c5/15/architecture-instruction-set-extensions-programming-reference.pdf

DSA spec: https://software.intel.com/sites/default/files/341204-intel-data-streaming-accelerator-spec
pdf

16.32 Software Guard eXtensions (SGX)

16.32.1 Overview

Software Guard eXtensions (SGX) hardware enables for user space applications to set aside private memory regions of code and data:

- Privileged (ring-0) ENCLS functions orchestrate the construction of the regions.
- Unprivileged (ring-3) ENCLU functions allow an application to enter and execute inside the regions.

These memory regions are called enclaves. An enclave can be only entered at a fixed set of entry points. Each entry point can hold a single hardware thread at a time. While the enclave is loaded from a regular binary file by using ENCLS functions, only the threads inside the enclave can access its memory. The region is denied from outside access by the CPU, and encrypted before it leaves from LLC.

The support can be determined by

```
grep sgx /proc/cpuinfo
```

SGX must both be supported in the processor and enabled by the BIOS. If SGX appears to be unsupported on a system which has hardware support, ensure support is enabled in the BIOS. If a BIOS presents a choice between “Enabled” and “Software Enabled” modes for SGX, choose “Enabled”.

16.32.2 Enclave Page Cache

SGX utilizes an *Enclave Page Cache (EPC)* to store pages that are associated with an enclave. It is contained in a BIOS-reserved region of physical memory. Unlike pages used for regular memory, pages can only be accessed from outside of the enclave during enclave construction with special, limited SGX instructions.

Only a CPU executing inside an enclave can directly access enclave memory. However, a CPU executing inside an enclave may access normal memory outside the enclave.

The kernel manages enclave memory similar to how it treats device memory.

Enclave Page Types

SGX Enclave Control Structure (SECS)
    Enclave's address range, attributes and other global data are defined by this structure.

Regular (REG)
    Regular EPC pages contain the code and data of an enclave.

Thread Control Structure (TCS)
    Thread Control Structure pages define the entry points to an enclave and track the execution state of an enclave thread.

Version Array (VA)
    Version Array pages contain 512 slots, each of which can contain a version number for a page evicted from the EPC.

Enclave Page Cache Map

The processor tracks EPC pages in a hardware metadata structure called the *Enclave Page Cache Map (EPCM)*. The EPCM contains an entry for each EPC page which describes the owning enclave, access rights and page type among the other things.

EPCM permissions are separate from the normal page tables. This prevents the kernel from, for instance, allowing writes to data which an enclave wishes to remain read-only. EPCM permissions may only impose additional restrictions on top of normal x86 page permissions.

For all intents and purposes, the SGX architecture allows the processor to invalidate all EPCM entries at will. This requires that software be prepared to handle an EPCM fault at any time. In practice, this can happen on events like power transitions when the ephemeral key that encrypts enclave memory is lost.

16.32.3 Application interface

Enclave build functions

In addition to the traditional compiler and linker build process, SGX has a separate enclave "build" process. Enclaves must be built before they can be executed (entered). The first step in building an enclave is opening the **/dev/sgx_enclave** device. Since enclave memory is protected from direct access, special privileged instructions are then used to copy data into enclave pages and establish enclave page permissions.

long **`sgx_ioc_enclave_create`**(struct sgx_encl *encl, void __user *arg)
handler for `SGX_IOC_ENCLAVE_CREATE`

Parameters

`struct sgx_encl *encl`
An enclave pointer.

`void __user *arg`
The ioctl argument.

Description

Allocate kernel data structures for the enclave and invoke ECREATE.

Return

- 0: Success.
- -EIO: ECREATE failed.
- -errno: POSIX error.

long **`sgx_ioc_enclave_add_pages`**(struct sgx_encl *encl, void __user *arg)
The handler for `SGX_IOC_ENCLAVE_ADD_PAGES`

Parameters

`struct sgx_encl *encl`
an enclave pointer

`void __user *arg`
a user pointer to a struct sgx_enclave_add_pages instance

Description

Add one or more pages to an uninitialized enclave, and optionally extend the measurement with the contents of the page. The SECINFO and measurement mask are applied to all pages.

A SECINFO for a TCS is required to always contain zero permissions because CPU silently zeros them. Allowing anything else would cause a mismatch in the measurement.

mmap()'s protection bits are capped by the page permissions. For each page address, the maximum protection bits are computed with the following heuristics:

1. A regular page: PROT_R, PROT_W and PROT_X match the SECINFO permissions.
2. A TCS page: PROT_R | PROT_W.

mmap() is not allowed to surpass the minimum of the maximum protection bits within the given address range.

The function deinitializes kernel data structures for enclave and returns -EIO in any of the following conditions:

- Enclave Page Cache (EPC), the physical memory holding enclaves, has been invalidated. This will cause EADD and EEXTEND to fail.
- If the source address is corrupted somehow when executing EADD.

Return

- 0: Success.

- -EACCES: The source page is located in a noexec partition.
- -ENOMEM: Out of EPC pages.
- -EINTR: The call was interrupted before data was processed.
- **-EIO: Either EADD or EEXTEND failed because invalid source address**
  or power cycle.
- -errno: POSIX error.

long **`sgx_ioc_enclave_init`**(struct sgx_encl *encl, void __user *arg)
handler for `SGX_IOC_ENCLAVE_INIT`

Parameters

`struct sgx_encl *encl`
an enclave pointer

`void __user *arg`
userspace pointer to a struct sgx_enclave_init instance

Description

Flush any outstanding enqueued EADD operations and perform EINIT. The Launch Enclave Public Key Hash MSRs are rewritten as necessary to match the enclave's MRSIGNER, which is caculated from the provided sigstruct.

Return

- 0: Success.
- -EPERM: Invalid SIGSTRUCT.
- -EIO: EINIT failed because of a power cycle.
- -errno: POSIX error.

long **`sgx_ioc_enclave_provision`**(struct sgx_encl *encl, void __user *arg)
handler for `SGX_IOC_ENCLAVE_PROVISION`

Parameters

`struct sgx_encl *encl`
an enclave pointer

`void __user *arg`
userspace pointer to a struct sgx_enclave_provision instance

Description

Allow ATTRIBUTE.PROVISION_KEY for an enclave by providing a file handle to /dev/sgx_provision.

Return

- 0: Success.
- -errno: Otherwise.

Enclave runtime management

Systems supporting SGX2 additionally support changes to initialized enclaves: modifying enclave page permissions and type, and dynamically adding and removing of enclave pages. When an enclave accesses an address within its address range that does not have a backing page then a new regular page will be dynamically added to the enclave. The enclave is still required to run EACCEPT on the new page before it can be used.

long **`sgx_ioc_enclave_restrict_permissions`**(struct sgx_encl *encl, void __user *arg)
    handler for `SGX_IOC_ENCLAVE_RESTRICT_PERMISSIONS`

Parameters

`struct sgx_encl *encl`
    an enclave pointer

`void __user *arg`
    userspace pointer to a `struct sgx_enclave_restrict_permissions` instance

Description

SGX2 distinguishes between relaxing and restricting the enclave page permissions maintained by the hardware (EPCM permissions) of pages belonging to an initialized enclave (after SGX_IOC_ENCLAVE_INIT).

EPCM permissions cannot be restricted from within the enclave, the enclave requires the kernel to run the privileged level 0 instructions ENCLS[EMODPR] and ENCLS[ETRACK]. An attempt to relax EPCM permissions with this call will be ignored by the hardware.

Return

- 0: Success
- -errno: Otherwise

long **`sgx_ioc_enclave_modify_types`**(struct sgx_encl *encl, void __user *arg)
    handler for `SGX_IOC_ENCLAVE_MODIFY_TYPES`

Parameters

`struct sgx_encl *encl`
    an enclave pointer

`void __user *arg`
    userspace pointer to a `struct sgx_enclave_modify_types` instance

Description

Ability to change the enclave page type supports the following use cases:

- It is possible to add TCS pages to an enclave by changing the type of regular pages (`SGX_PAGE_TYPE_REG`) to TCS (`SGX_PAGE_TYPE_TCS`) pages. With this support the number of threads supported by an initialized enclave can be increased dynamically.
- Regular or TCS pages can dynamically be removed from an initialized enclave by changing the page type to `SGX_PAGE_TYPE_TRIM`. Changing the page type to `SGX_PAGE_TYPE_TRIM` marks the page for removal with actual removal done by handler of `SGX_IOC_ENCLAVE_REMOVE_PAGES` ioctl() called after ENCLU[EACCEPT] is run on `SGX_PAGE_TYPE_TRIM` page from within the enclave.

Return

- 0: Success
- -errno: Otherwise

long **`sgx_ioc_enclave_remove_pages`**(struct sgx_encl *encl, void __user *arg)

handler for `SGX_IOC_ENCLAVE_REMOVE_PAGES`

Parameters

`struct sgx_encl *encl`
an enclave pointer

`void __user *arg`
userspace pointer to `struct sgx_enclave_remove_pages` instance

Description

Final step of the flow removing pages from an initialized enclave. The complete flow is:

1) User changes the type of the pages to be removed to `SGX_PAGE_TYPE_TRIM` using the `SGX_IOC_ENCLAVE_MODIFY_TYPES` ioctl().
2) User approves the page removal by running ENCLU[EACCEPT] from within the enclave.
3) User initiates actual page removal using the `SGX_IOC_ENCLAVE_REMOVE_PAGES` ioctl() that is handled here.

First remove any page table entries pointing to the page and then proceed with the actual removal of the enclave page and data in support of it.

VA pages are not affected by this removal. It is thus possible that the enclave may end up with more VA pages than needed to support all its pages.

Return

- 0: Success
- -errno: Otherwise

Enclave vDSO

Entering an enclave can only be done through SGX-specific EENTER and ERESUME functions, and is a non-trivial process. Because of the complexity of transitioning to and from an enclave, enclaves typically utilize a library to handle the actual transitions. This is roughly analogous to how glibc implementations are used by most applications to wrap system calls.

Another crucial characteristic of enclaves is that they can generate exceptions as part of their normal operation that need to be handled in the enclave or are unique to SGX.

Instead of the traditional signal mechanism to handle these exceptions, SGX can leverage special exception fixup provided by the vDSO. The kernel-provided vDSO function wraps low-level transitions to/from the enclave like EENTER and ERESUME. The vDSO function intercepts exceptions that would otherwise generate a signal and return the fault information directly to its caller. This avoids the need to juggle signal handlers.

`vdso_sgx_enter_enclave_t`

Typedef: Prototype for __vdso_sgx_enter_enclave(), a vDSO function to enter an SGX enclave.

Syntax

```
int vdso_sgx_enter_enclave_t (unsigned long rdi, unsigned long rsi,
unsigned long rdx, unsigned int function, unsigned long r8, unsigned
long r9, struct sgx_enclave_run *run)
```

Parameters

`unsigned long rdi`
Pass-through value for RDI

`unsigned long rsi`
Pass-through value for RSI

`unsigned long rdx`
Pass-through value for RDX

`unsigned int function`
ENCLU function, must be EENTER or ERESUME

`unsigned long r8`
Pass-through value for R8

`unsigned long r9`
Pass-through value for R9

`struct sgx_enclave_run *run`
struct sgx_enclave_run, must be non-NULL

NOTE

__vdso_sgx_enter_enclave() does not ensure full compliance with the x86-64 ABI, e.g. doesn't handle XSAVE state. Except for non-volatile general purpose registers, EFLAGS.DF, and RSP alignment, preserving/setting state in accordance with the x86-64 ABI is the responsibility of the enclave and its runtime, i.e. __vdso_sgx_enter_enclave() cannot be called from C code without careful consideration by both the enclave and its runtime.

Description

All general purpose registers except RAX, RBX and RCX are passed as-is to the enclave. RAX, RBX and RCX are consumed by EENTER and ERESUME and are loaded with **function**, asynchronous exit pointer, and **run.tcs** respectively.

RBP and the stack are used to anchor __vdso_sgx_enter_enclave() to the pre-enclave state, e.g. to retrieve **run.exception** and **run.user_handler** after an enclave exit. All other registers are available for use by the enclave and its runtime, e.g. an enclave can push additional data onto the stack (and modify RSP) to pass information to the optional user handler (see below).

Most exceptions reported on ENCLU, including those that occur within the enclave, are fixed up and reported synchronously instead of being delivered via a standard signal. Debug Exceptions (#DB) and Breakpoints (#BP) are never fixed up and are always delivered via standard signals. On synchronously reported exceptions, -EFAULT is returned and details about the exception are recorded in **run.exception**, the optional sgx_enclave_exception struct.

Return

- 0: ENCLU function was successfully executed.
- -EINVAL: Invalid ENCL number (neither EENTER nor ERESUME).

16.32.4 ksgxd

SGX support includes a kernel thread called *ksgxd*.

EPC sanitization

ksgxd is started when SGX initializes. Enclave memory is typically ready for use when the processor powers on or resets. However, if SGX has been in use since the reset, enclave pages may be in an inconsistent state. This might occur after a crash and kexec() cycle, for instance. At boot, ksgxd reinitializes all enclave pages so that they can be allocated and re-used.

The sanitization is done by going through EPC address space and applying the EREMOVE function to each physical page. Some enclave pages like SECS pages have hardware dependencies on other pages which prevents EREMOVE from functioning. Executing two EREMOVE passes removes the dependencies.

Page reclaimer

Similar to the core kswapd, ksgxd, is responsible for managing the overcommitment of enclave memory. If the system runs out of enclave memory, *ksgxd* "swaps" enclave memory to normal memory.

16.32.5 Launch Control

SGX provides a launch control mechanism. After all enclave pages have been copied, kernel executes EINIT function, which initializes the enclave. Only after this the CPU can execute inside the enclave.

EINIT function takes an RSA-3072 signature of the enclave measurement. The function checks that the measurement is correct and signature is signed with the key hashed to the four **IA32_SGXLEPUBKEYHASH{0, 1, 2, 3}** MSRs representing the SHA256 of a public key.

Those MSRs can be configured by the BIOS to be either readable or writable. Linux supports only writable configuration in order to give full control to the kernel on launch control policy. Before calling EINIT function, the driver sets the MSRs to match the enclave's signing key.

16.32.6 Encryption engines

In order to conceal the enclave data while it is out of the CPU package, the memory controller has an encryption engine to transparently encrypt and decrypt enclave memory.

In CPUs prior to Ice Lake, the Memory Encryption Engine (MEE) is used to encrypt pages leaving the CPU caches. MEE uses a n-ary Merkle tree with root in SRAM to maintain integrity of the encrypted data. This provides integrity and anti-replay protection but does not scale to large memory sizes because the time required to update the Merkle tree grows logarithmically in relation to the memory size.

CPUs starting from Icelake use Total Memory Encryption (TME) in the place of MEE. TME-based SGX implementations do not have an integrity Merkle tree, which means integrity and replay-attacks are not mitigated. B, it includes additional changes to prevent cipher text from being returned and SW memory aliases from being created.

DMA to enclave memory is blocked by range registers on both MEE and TME systems (SDM section 41.10).

16.32.7 Usage Models

Shared Library

Sensitive data and the code that acts on it is partitioned from the application into a separate library. The library is then linked as a DSO which can be loaded into an enclave. The application can then make individual function calls into the enclave through special SGX instructions. A run-time within the enclave is configured to marshal function parameters into and out of the enclave and to call the correct library function.

Application Container

An application may be loaded into a container enclave which is specially configured with a library OS and run-time which permits the application to run. The enclave run-time and library OS work together to execute the application when a thread enters the enclave.

16.32.8 Impact of Potential Kernel SGX Bugs

EPC leaks

When EPC page leaks happen, a WARNING like this is shown in dmesg:

"EREMOVE returned … and an EPC page was leaked. SGX may become unusable…"

This is effectively a kernel use-after-free of an EPC page, and due to the way SGX works, the bug is detected at freeing. Rather than adding the page back to the pool of available EPC pages, the kernel intentionally leaks the page to avoid additional errors in the future.

When this happens, the kernel will likely soon leak more EPC pages, and SGX will likely become unusable because the memory available to SGX is limited. However, while this may be fatal to SGX, the rest of the kernel is unlikely to be impacted and should continue to work.

As a result, when this happens, user should stop running any new SGX workloads, (or just any new workloads), and migrate all valuable workloads. Although a machine reboot can recover all EPC memory, the bug should be reported to Linux developers.

16.32.9 Virtual EPC

The implementation has also a virtual EPC driver to support SGX enclaves in guests. Unlike the SGX driver, an EPC page allocated by the virtual EPC driver doesn't have a specific enclave associated with it. This is because KVM doesn't track how a guest uses EPC pages.

As a result, the SGX core page reclaimer doesn't support reclaiming EPC pages allocated to KVM guests through the virtual EPC driver. If the user wants to deploy SGX applications both on the host and in guests on the same machine, the user should reserve enough EPC (by taking out total virtual EPC size of all SGX VMs from the physical EPC size) for host SGX applications so they can run with acceptable performance.

Architectural behavior is to restore all EPC pages to an uninitialized state also after a guest reboot. Because this state can be reached only through the privileged `ENCLS[EREMOVE]` instruction, `/dev/sgx_vepc` provides the `SGX_IOC_VEPC_REMOVE_ALL` ioctl to execute the instruction on all pages in the virtual EPC.

`EREMOVE` can fail for three reasons. Userspace must pay attention to expected failures and handle them as follows:

1. Page removal will always fail when any thread is running in the enclave to which the page belongs. In this case the ioctl will return `EBUSY` independent of whether it has successfully removed some pages; userspace can avoid these failures by preventing execution of any vcpu which maps the virtual EPC.
2. Page removal will cause a general protection fault if two calls to `EREMOVE` happen concurrently for pages that refer to the same "SECS" metadata pages. This can happen if there are concurrent invocations to `SGX_IOC_VEPC_REMOVE_ALL`, or if a `/dev/sgx_vepc` file descriptor in the guest is closed at the same time as `SGX_IOC_VEPC_REMOVE_ALL`; it will also be reported as `EBUSY`. This can be avoided in userspace by serializing calls to the ioctl() and to close(), but in general it should not be a problem.
3. Finally, page removal will fail for SECS metadata pages which still have child pages. Child pages can be removed by executing `SGX_IOC_VEPC_REMOVE_ALL` on all `/dev/sgx_vepc` file descriptors mapped into the guest. This means that the ioctl() must be called twice: an initial set of calls to remove child pages and a subsequent set of calls to remove SECS pages. The second set of calls is only required for those mappings that returned a nonzero value from the first call. It indicates a bug in the kernel or the userspace client if any of the second round of `SGX_IOC_VEPC_REMOVE_ALL` calls has a return code other than 0.

16.33 Feature status on x86 architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	ok
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok
core	thread-info-in-task	THREAD_INFO_IN_TASK	ok
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	ok
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	ok
debug	KASAN	HAVE_ARCH_KASAN	ok
debug	kcov	ARCH_HAS_KCOV	ok
debug	kgdb	HAVE_ARCH_KGDB	ok
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok
debug	kprobes	HAVE_KPROBES	ok
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	ok
debug	kretprobes	HAVE_KRETPROBES	ok
debug	optprobes	HAVE_OPTPROBES	ok
debug	stackprotector	HAVE_STACKPROTECTOR	ok
debug	uprobes	ARCH_SUPPORTS_UPROBES	ok
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	ok

Table 2 – continued fron

Subsystem	Feature	Kconfig	Status
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	ok
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	ok
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	ok
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	ok
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	ok
perf	perf-regs	HAVE_PERF_REGS	ok
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	ok
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	ok
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	ok
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	TODO
time	clockevents	!LEGACY_TIMER_TICK	ok
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	ok
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	ok
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	ok
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	ok
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	ok
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	ok
vm	ioremap_prot	HAVE_IOREMAP_PROT	ok
vm	PG_uncached	ARCH_USES_PG_UNCACHED	ok
vm	pte_special	ARCH_HAS_PTE_SPECIAL	ok
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	ok

16.34 x86-specific ELF Auxiliary Vectors

This document describes the semantics of the x86 auxiliary vectors.

16.34.1 Introduction

ELF Auxiliary vectors enable the kernel to efficiently provide configuration-specific parameters to userspace. In this example, a program allocates an alternate stack based on the kernel-provided size:

```
#include <sys/auxv.h>
#include <elf.h>
#include <signal.h>
#include <stdlib.h>
#include <assert.h>
#include <err.h>

#ifndef AT_MINSIGSTKSZ
#define AT_MINSIGSTKSZ      51
#endif

....
```

```
stack_t ss;

ss.ss_sp = malloc(ss.ss_size);
assert(ss.ss_sp);

ss.ss_size = getauxval(AT_MINSIGSTKSZ) + SIGSTKSZ;
ss.ss_flags = 0;

if (sigaltstack(&ss, NULL))
     err(1, "sigaltstack");
```

16.34.2 The exposed auxiliary vectors

AT_SYSINFO is used for locating the vsyscall entry point. It is not exported on 64-bit mode.

AT_SYSINFO_EHDR is the start address of the page containing the vDSO.

AT_MINSIGSTKSZ denotes the minimum stack size required by the kernel to deliver a signal to user-space. AT_MINSIGSTKSZ comprehends the space consumed by the kernel to accommodate the user context for the current hardware configuration. It does not comprehend subsequent user-space stack consumption, which must be added by the user. (e.g. Above, user-space adds SIGSTKSZ to AT_MINSIGSTKSZ.)

16.35 Using XSTATE features in user space applications

The x86 architecture supports floating-point extensions which are enumerated via CPUID. Applications consult CPUID and use XGETBV to evaluate which features have been enabled by the kernel XCR0.

Up to AVX-512 and PKRU states, these features are automatically enabled by the kernel if available. Features like AMX TILE_DATA (XSTATE component 18) are enabled by XCR0 as well, but the first use of related instruction is trapped by the kernel because by default the required large XSTATE buffers are not allocated automatically.

16.35.1 The purpose for dynamic features

Legacy userspace libraries often have hard-coded, static sizes for alternate signal stacks, often using MINSIGSTKSZ which is typically 2KB. That stack must be able to store at *least* the signal frame that the kernel sets up before jumping into the signal handler. That signal frame must include an XSAVE buffer defined by the CPU.

However, that means that the size of signal stacks is dynamic, not static, because different CPUs have differently-sized XSAVE buffers. A compiled-in size of 2KB with existing applications is too small for new CPU features like AMX. Instead of universally requiring larger stack, with the dynamic enabling, the kernel can enforce userspace applications to have properly-sized altstacks.

16.35.2 Using dynamically enabled XSTATE features in user space applications

The kernel provides an arch_prctl(2) based mechanism for applications to request the usage of such features. The arch_prctl(2) options related to this are:

-ARCH_GET_XCOMP_SUPP arch_prctl(ARCH_GET_XCOMP_SUPP, &features);

ARCH_GET_XCOMP_SUPP stores the supported features in userspace storage of type uint64_t. The second argument is a pointer to that storage.

-ARCH_GET_XCOMP_PERM arch_prctl(ARCH_GET_XCOMP_PERM, &features);

ARCH_GET_XCOMP_PERM stores the features for which the userspace process has permission in userspace storage of type uint64_t. The second argument is a pointer to that storage.

-ARCH_REQ_XCOMP_PERM arch_prctl(ARCH_REQ_XCOMP_PERM, feature_nr);

ARCH_REQ_XCOMP_PERM allows to request permission for a dynamically enabled feature or a feature set. A feature set can be mapped to a facility, e.g. AMX, and can require one or more XSTATE components to be enabled.

The feature argument is the number of the highest XSTATE component which is required for a facility to work.

When requesting permission for a feature, the kernel checks the availability. The kernel ensures that sigaltstacks in the process's tasks are large enough to accommodate the resulting large signal frame. It enforces this both during ARCH_REQ_XCOMP_SUPP and during any subsequent sigaltstack(2) calls. If an installed sigaltstack is smaller than the resulting sigframe size, ARCH_REQ_XCOMP_SUPP results in -ENOSUPP. Also, sigaltstack(2) results in -ENOMEM if the requested altstack is too small for the permitted features.

Permission, when granted, is valid per process. Permissions are inherited on fork(2) and cleared on exec(3).

The first use of an instruction related to a dynamically enabled feature is trapped by the kernel. The trap handler checks whether the process has permission to use the feature. If the process has no permission then the kernel sends SIGILL to the application. If the process has permission then the handler allocates a larger xstate buffer for the task so the large state can be context switched. In the unlikely cases that the allocation fails, the kernel sends SIGSEGV.

AMX TILE_DATA enabling example

Below is the example of how userspace applications enable TILE_DATA dynamically:

1. The application first needs to query the kernel for AMX support:

```
#include <asm/prctl.h>
#include <sys/syscall.h>
#include <stdio.h>
#include <unistd.h>

#ifndef ARCH_GET_XCOMP_SUPP
#define ARCH_GET_XCOMP_SUPP  0x1021
```

```
#endif

#ifndef ARCH_XCOMP_TILECFG
#define ARCH_XCOMP_TILECFG   17
#endif

#ifndef ARCH_XCOMP_TILEDATA
#define ARCH_XCOMP_TILEDATA  18
#endif

#define MASK_XCOMP_TILE      ((1 << ARCH_XCOMP_TILECFG) | \
                              (1 << ARCH_XCOMP_TILEDATA))

unsigned long features;
long rc;

...

rc = syscall(SYS_arch_prctl, ARCH_GET_XCOMP_SUPP, &features);

if (!rc && (features & MASK_XCOMP_TILE) == MASK_XCOMP_TILE)
    printf("AMX is available.\n");
```

2. After that, determining support for AMX, an application must explicitly ask permission to use it:

```
#ifndef ARCH_REQ_XCOMP_PERM
#define ARCH_REQ_XCOMP_PERM  0x1023
#endif

...

rc = syscall(SYS_arch_prctl, ARCH_REQ_XCOMP_PERM, ARCH_XCOMP_TILEDATA);

if (!rc)
    printf("AMX is ready for use.\n");
```

Note this example does not include the sigaltstack preparation.

16.35.3 Dynamic features in signal frames

Dynamcally enabled features are not written to the signal frame upon signal entry if the feature is in its initial configuration. This differs from non-dynamic features which are always written regardless of their configuration. Signal handlers can examine the XSAVE buffer's XSTATE_BV field to determine if a features was written.

16.35.4 Dynamic features for virtual machines

The permission for the guest state component needs to be managed separately from the host, as they are exclusive to each other. A coupled of options are extended to control the guest permission:

-ARCH_GET_XCOMP_GUEST_PERM arch_prctl(ARCH_GET_XCOMP_GUEST_PERM, &features);

ARCH_GET_XCOMP_GUEST_PERM is a variant of ARCH_GET_XCOMP_PERM. So it provides the same semantics and functionality but for the guest components.

-ARCH_REQ_XCOMP_GUEST_PERM arch_prctl(ARCH_REQ_XCOMP_GUEST_PERM, feature_nr);

ARCH_REQ_XCOMP_GUEST_PERM is a variant of ARCH_REQ_XCOMP_PERM. It has the same semantics for the guest permission. While providing a similar functionality, this comes with a constraint. Permission is frozen when the first VCPU is created. Any attempt to change permission after that point is going to be rejected. So, the permission has to be requested before the first VCPU creation.

Note that some VMMs may have already established a set of supported state components. These options are not presumed to support any particular VMM.

CHAPTER SEVENTEEN

XTENSA ARCHITECTURE

17.1 Atomic Operation Control (ATOMCTL) Register

We Have Atomic Operation Control (ATOMCTL) Register. This register determines the effect of using a S32C1I instruction with various combinations of:

1. With and without an Coherent Cache Controller which can do Atomic Transactions to the memory internally.
2. With and without An Intelligent Memory Controller which can do Atomic Transactions itself.

The Core comes up with a default value of for the three types of cache ops:

```
0x28: (WB: Internal, WT: Internal, BY:Exception)
```

On the FPGA Cards we typically simulate an Intelligent Memory controller which can implement RCW transactions. For FPGA cards with an External Memory controller we let it to the atomic operations internally while doing a Cached (WB) transaction and use the Memory RCW for un-cached operations.

For systems without an coherent cache controller, non-MX, we always use the memory controllers RCW, though non-MX controllers likely support the Internal Operation.

CUSTOMER-WARNING:
: Virtually all customers buy their memory controllers from vendors that don't support atomic RCW memory transactions and will likely want to configure this register to not use RCW.

Developers might find using RCW in Bypass mode convenient when testing with the cache being bypassed; for example studying cache alias problems.

See Section 4.3.12.4 of ISA; Bits:

```
  WB     WT      BY
5   4 | 3   2 | 1   0
```

2 Bit Field Values	WB - Write Back	WT - Write Thru	BY - Bypass
0	Exception	Exception	Exception
1	RCW Transaction	RCW Transaction	RCW Transaction
2	Internal Operation	Internal Operation	Reserved
3	Reserved	Reserved	Reserved

17.2 Passing boot parameters to the kernel

Boot parameters are represented as a TLV list in the memory. Please see arch/xtensa/include/asm/bootparam.h for definition of the bp_tag structure and tag value constants. First entry in the list must have type BP_TAG_FIRST, last entry must have type BP_TAG_LAST. The address of the first list entry is passed to the kernel in the register a2. The address type depends on MMU type:

- For configurations without MMU, with region protection or with MPU the address must be the physical address.
- For configurations with region translarion MMU or with MMUv3 and CONFIG_MMU=n the address must be a valid address in the current mapping. The kernel will not change the mapping on its own.
- For configurations with MMUv2 the address must be a virtual address in the default virtual mapping (0xd0000000..0xffffffff).
- For configurations with MMUv3 and CONFIG_MMU=y the address may be either a virtual or physical address. In either case it must be within the default virtual mapping. It is considered physical if it is within the range of physical addresses covered by the default KSEG mapping (XCHAL_KSEG_PADDR.. XCHAL_KSEG_PADDR + XCHAL_KSEG_SIZE), otherwise it is considered virtual.

17.3 MMUv3 initialization sequence

The code in the initialize_mmu macro sets up MMUv3 memory mapping identically to MMUv2 fixed memory mapping. Depending on CONFIG_INITIALIZE_XTENSA_MMU_INSIDE_VMLINUX symbol this code is located in addresses it was linked for (symbol undefined), or not (symbol defined), so it needs to be position-independent.

The code has the following assumptions:

- This code fragment is run only on an MMU v3.
- TLBs are in their reset state.
- ITLBCFG and DTLBCFG are zero (reset state).
- RASID is 0x04030201 (reset state).
- PS.RING is zero (reset state).

- LITBASE is zero (reset state, PC-relative literals); required to be PIC.

TLB setup proceeds along the following steps.

Legend:

- VA = virtual address (two upper nibbles of it);
- PA = physical address (two upper nibbles of it);
- pc = physical range that contains this code;

After step 2, we jump to virtual address in the range 0x40000000..0x5fffffff or 0x00000000..0x1fffffff, depending on whether the kernel was loaded below 0x40000000 or above. That address corresponds to next instruction to execute in this code. After step 4, we jump to intended (linked) address of this code. The scheme below assumes that the kernel is loaded below 0x40000000.

•	Step0	Step1	Step2	Step3		Step4	Step5
VA	PA	PA	PA	PA	VA	PA	PA
E0..FF	-> E0	-> E0	-> E0		F0..FF	-> F0	-> F0
C0..DF	-> C0	-> C0	-> C0		E0..EF	-> F0	-> F0
A0..BF	-> A0	-> A0	-> A0		D8..DF	-> 00	-> 00
80..9F	-> 80	-> 80	-> 80		D0..D7	-> 00	-> 00
60..7F	-> 60	-> 60	-> 60				
40..5F	-> 40		-> pc	-> pc	40..5F	-> pc	
20..3F	-> 20	-> 20	-> 20				
00..1F	-> 00	-> 00	-> 00				

The default location of IO peripherals is above 0xf0000000. This may be changed using a "ranges" property in a device tree simple-bus node. See the Devicetree Specification, section 4.5 for details on the syntax and semantics of simple-bus nodes. The following limitations apply:

1. Only top level simple-bus nodes are considered
2. Only one (first) simple-bus node is considered
3. Empty "ranges" properties are not supported
4. Only the first triplet in the "ranges" property is considered
5. The parent-bus-address value is rounded down to the nearest 256MB boundary
6. The IO area covers the entire 256MB segment of parent-bus-address; the "ranges" triplet length field is ignored

17.3.1 MMUv3 address space layouts.

Default MMUv2-compatible layout:

```
                      Symbol                   VADDR       Size
+------------------+
| Userspace        |                           0x00000000  TASK_SIZE
+------------------+                           0x40000000
+------------------+
| Page table       |  XCHAL_PAGE_TABLE_VADDR   0x80000000  XCHAL_PAGE_TABLE_
↪SIZE
+------------------+
| KASAN shadow map |  KASAN_SHADOW_START       0x80400000  KASAN_SHADOW_SIZE
+------------------+                           0x8e400000
+------------------+
| VMALLOC area     |  VMALLOC_START            0xc0000000  128MB - 64KB
+------------------+  VMALLOC_END
+------------------+
| Cache aliasing   |  TLBTEMP_BASE_1           0xc8000000  DCACHE_WAY_SIZE
| remap area 1     |
+------------------+
| Cache aliasing   |  TLBTEMP_BASE_2                       DCACHE_WAY_SIZE
| remap area 2     |
+------------------+
+------------------+
| KMAP area        |  PKMAP_BASE                           PTRS_PER_PTE *
|                  |                                       DCACHE_N_COLORS *
|                  |                                       PAGE_SIZE
|                  |                                       (4MB * DCACHE_N_
↪COLORS)
+------------------+
| Atomic KMAP area |  FIXADDR_START                        KM_TYPE_NR *
|                  |                                       NR_CPUS *
|                  |                                       DCACHE_N_COLORS *
|                  |                                       PAGE_SIZE
+------------------+  FIXADDR_TOP              0xcffff000
+------------------+
| Cached KSEG      |  XCHAL_KSEG_CACHED_VADDR  0xd0000000  128MB
+------------------+
| Uncached KSEG    |  XCHAL_KSEG_BYPASS_VADDR  0xd8000000  128MB
+------------------+
| Cached KIO       |  XCHAL_KIO_CACHED_VADDR   0xe0000000  256MB
+------------------+
| Uncached KIO     |  XCHAL_KIO_BYPASS_VADDR   0xf0000000  256MB
+------------------+
```

256MB cached + 256MB uncached layout:

```
                      Symbol                   VADDR       Size
+------------------+
| Userspace        |                           0x00000000  TASK_SIZE
+------------------+                           0x40000000
```

```
+------------------+
| Page table       |  XCHAL_PAGE_TABLE_VADDR   0x80000000  XCHAL_PAGE_TABLE_
↪SIZE
+------------------+
| KASAN shadow map |  KASAN_SHADOW_START       0x80400000  KASAN_SHADOW_SIZE
+------------------+                           0x8e400000
+------------------+
| VMALLOC area     |  VMALLOC_START            0xa0000000  128MB - 64KB
+------------------+  VMALLOC_END
+------------------+
| Cache aliasing   |  TLBTEMP_BASE_1           0xa8000000  DCACHE_WAY_SIZE
| remap area 1     |
+------------------+
| Cache aliasing   |  TLBTEMP_BASE_2                       DCACHE_WAY_SIZE
| remap area 2     |
+------------------+
+------------------+
| KMAP area        |  PKMAP_BASE                           PTRS_PER_PTE *
|                  |                                       DCACHE_N_COLORS *
|                  |                                       PAGE_SIZE
|                  |                                       (4MB * DCACHE_N_
↪COLORS)
+------------------+
| Atomic KMAP area |  FIXADDR_START                        KM_TYPE_NR *
|                  |                                       NR_CPUS *
|                  |                                       DCACHE_N_COLORS *
|                  |                                       PAGE_SIZE
+------------------+  FIXADDR_TOP              0xaffff000
+------------------+
| Cached KSEG      |  XCHAL_KSEG_CACHED_VADDR  0xb0000000  256MB
+------------------+
| Uncached KSEG    |  XCHAL_KSEG_BYPASS_VADDR  0xc0000000  256MB
+------------------+
+------------------+
| Cached KIO       |  XCHAL_KIO_CACHED_VADDR   0xe0000000  256MB
+------------------+
| Uncached KIO     |  XCHAL_KIO_BYPASS_VADDR   0xf0000000  256MB
+------------------+
```

512MB cached + 512MB uncached layout:

```
                      Symbol                   VADDR       Size
+------------------+
| Userspace        |                           0x00000000  TASK_SIZE
+------------------+                           0x40000000
+------------------+
| Page table       |  XCHAL_PAGE_TABLE_VADDR   0x80000000  XCHAL_PAGE_TABLE_
↪SIZE
+------------------+
| KASAN shadow map |  KASAN_SHADOW_START       0x80400000  KASAN_SHADOW_SIZE
+------------------+                           0x8e400000
```

```
+------------------+
| VMALLOC area     |  VMALLOC_START            0x90000000  128MB - 64KB
+------------------+  VMALLOC_END
+------------------+
| Cache aliasing   |  TLBTEMP_BASE_1           0x98000000  DCACHE_WAY_SIZE
| remap area 1     |
+------------------+
| Cache aliasing   |  TLBTEMP_BASE_2                       DCACHE_WAY_SIZE
| remap area 2     |
+------------------+
+------------------+
| KMAP area        |  PKMAP_BASE                           PTRS_PER_PTE *
|                  |                                       DCACHE_N_COLORS *
|                  |                                       PAGE_SIZE
|                  |                                       (4MB * DCACHE_N_
↪COLORS)
+------------------+
| Atomic KMAP area |  FIXADDR_START                        KM_TYPE_NR *
|                  |                                       NR_CPUS *
|                  |                                       DCACHE_N_COLORS *
|                  |                                       PAGE_SIZE
+------------------+  FIXADDR_TOP              0x9ffff000
+------------------+
| Cached KSEG      |  XCHAL_KSEG_CACHED_VADDR  0xa0000000  512MB
+------------------+
| Uncached KSEG    |  XCHAL_KSEG_BYPASS_VADDR  0xc0000000  512MB
+------------------+
| Cached KIO       |  XCHAL_KIO_CACHED_VADDR   0xe0000000  256MB
+------------------+
| Uncached KIO     |  XCHAL_KIO_BYPASS_VADDR   0xf0000000  256MB
+------------------+
```

17.4 Feature status on xtensa architecture

Subsystem	Feature	Kconfig	Status
core	cBPF-JIT	HAVE_CBPF_JIT	TODO
core	eBPF-JIT	HAVE_EBPF_JIT	TODO
core	generic-idle-thread	GENERIC_SMP_IDLE_THREAD	ok
core	jump-labels	HAVE_ARCH_JUMP_LABEL	ok
core	thread-info-in-task	THREAD_INFO_IN_TASK	TODO
core	tracehook	HAVE_ARCH_TRACEHOOK	ok
debug	debug-vm-pgtable	ARCH_HAS_DEBUG_VM_PGTABLE	ok
debug	gcov-profile-all	ARCH_HAS_GCOV_PROFILE_ALL	ok
debug	KASAN	HAVE_ARCH_KASAN	ok
debug	kcov	ARCH_HAS_KCOV	ok
debug	kgdb	HAVE_ARCH_KGDB	TODO
debug	kmemleak	HAVE_DEBUG_KMEMLEAK	ok

Table 1 – continued fron

Subsystem	Feature	Kconfig	Status
debug	kprobes	HAVE_KPROBES	TODO
debug	kprobes-on-ftrace	HAVE_KPROBES_ON_FTRACE	TODO
debug	kretprobes	HAVE_KRETPROBES	TODO
debug	optprobes	HAVE_OPTPROBES	TODO
debug	stackprotector	HAVE_STACKPROTECTOR	ok
debug	uprobes	ARCH_SUPPORTS_UPROBES	TODO
debug	user-ret-profiler	HAVE_USER_RETURN_NOTIFIER	TODO
io	dma-contiguous	HAVE_DMA_CONTIGUOUS	ok
locking	cmpxchg-local	HAVE_CMPXCHG_LOCAL	TODO
locking	lockdep	LOCKDEP_SUPPORT	ok
locking	queued-rwlocks	ARCH_USE_QUEUED_RWLOCKS	ok
locking	queued-spinlocks	ARCH_USE_QUEUED_SPINLOCKS	ok
perf	kprobes-event	HAVE_REGS_AND_STACK_ACCESS_API	TODO
perf	perf-regs	HAVE_PERF_REGS	TODO
perf	perf-stackdump	HAVE_PERF_USER_STACK_DUMP	TODO
sched	membarrier-sync-core	ARCH_HAS_MEMBARRIER_SYNC_CORE	TODO
sched	numa-balancing	ARCH_SUPPORTS_NUMA_BALANCING	---
seccomp	seccomp-filter	HAVE_ARCH_SECCOMP_FILTER	ok
time	arch-tick-broadcast	ARCH_HAS_TICK_BROADCAST	TODO
time	clockevents	!LEGACY_TIMER_TICK	ok
time	irq-time-acct	HAVE_IRQ_TIME_ACCOUNTING	ok
time	user-context-tracking	HAVE_CONTEXT_TRACKING_USER	ok
time	virt-cpuacct	HAVE_VIRT_CPU_ACCOUNTING	ok
vm	batch-unmap-tlb-flush	ARCH_WANT_BATCHED_UNMAP_TLB_FLUSH	TODO
vm	ELF-ASLR	ARCH_WANT_DEFAULT_TOPDOWN_MMAP_LAYOUT	TODO
vm	huge-vmap	HAVE_ARCH_HUGE_VMAP	TODO
vm	ioremap_prot	HAVE_IOREMAP_PROT	TODO
vm	PG_uncached	ARCH_USES_PG_UNCACHED	TODO
vm	pte_special	ARCH_HAS_PTE_SPECIAL	TODO
vm	THP	HAVE_ARCH_TRANSPARENT_HUGEPAGE	---

INDEX