Linux Arm64 Documentation

The kernel development community

Jun 10, 2024

CONTENTS

1 ACPI Tables 1

2 Activity Monitors Unit (AMU) extension in AArch64 Linux 19

3 ACPI on ARMv8 Servers 23

4 Booting AArch64 Linux 33

5 ARM64 CPU Feature Registers 39

6 ARM64 ELF hwcaps 45

7 HugeTLBpage on ARM64 51

8 Legacy instructions 53

9 Memory Layout on AArch64 Linux 55

10 Memory Tagging Extension (MTE) in AArch64 Linux 59

11 Perf Event Attributes 65

12 Pointer authentication in AArch64 Linux 67

13 Silicon Errata and Software Workarounds 71

14 Scalable Vector Extension support for AArch64 Linux 75

15 AArch64 TAGGED ADDRESS ABI 85

16 Tagged virtual addresses in AArch64 Linux 89

CHAPTER

ONE

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: BGRT, CPEP, CSRT, DBG2, DRTM, ECDT, FACS, FPDT, IORT, MCHI, MPST, MSCT, NFIT, PMTT, RASF, SBST, SLIT, SPMI, SRAT, STAO, TCPA, TPM2, UEFI, XENV
- Not supported: BOOT, DBGP, DMAR, ETDT, HPET, IBFT, IVRS, LPIT, MSDM, OEMx, PSDT, RSDT, SLIC, WAET, WDAT, WDRT, WPBT

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.

continues on next page

Table 1 – continued from previous page

Table	Usage for ARMv8 Linux
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, not currently supported.
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.
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.

continues on next page

Table 1 – continued from previous page

Table	Usage for ARMv8 Linux
MCHI	Signature Reserved (signature == "MCHI") **Management Controller Host Interface table** Optional, not currently supported.
MPST	Section 5.2.21 (signature == "MPST") **Memory Power State Table** Optional, not currently supported.
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.
PMTT	Section 5.2.21.12 (signature == "PMTT") **Platform Memory Topology Table** Optional, not currently supported.
PSDT	Section 5.2.11.3 (signature == "PSDT") **Persistent System Description Table** Obsolete table, will not be supported.

continues on next page

Table 1 – continued from previous page

Table	Usage for ARMv8 Linux
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.
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.
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.

continues on next page

Table 1 – continued from previous page

Table	Usage for ARMv8 Linux
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.
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.

continues on next page

Table 1 – continued from previous page

Table	Usage for ARMv8 Linux
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.

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.

continues on next page

Table 2 – continued from previous page

Name	Section	Usage for ARMv8 Linux
_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.
_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.

continues on next page

Table 2 – continued from previous page

Name	Section	Usage for ARMv8 Linux
_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.
_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.

continues on next page

Table 2 – continued from previous page

Name	Section	Usage for ARMv8 Linux
_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.
_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.

continues on next page

Table 2 – continued from previous page

Name	Section	Usage for ARMv8 Linux
_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.

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.

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.

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.

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.

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

CHAPTER

TWO

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.

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.

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.

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.

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.

CHAPTER

THREE

ACPI ON ARMV8 SERVERS

ACPI can be used for ARMv8 general purpose servers designed to follow the ARM SBSA (Server Base System Architecture) [0] and SBBR (Server Base Boot Requirements) [1] specifications. Please note that the SBBR can be retrieved simply by visiting [1], but the SBSA is currently only available to those with an ARM login due to ARM IP licensing concerns.

The ARMv8 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 ARMv8 system does not meet the requirements of the SBSA and SBBR, 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 ARMv8 servers, 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 ARMv8 system – that is, what Linux expects of ACPI and what ACPI can expect of Linux.

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 [2] from Grant Likely that outlines the reasoning behind ACPI on ARMv8 servers. 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.

3.2 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 – it's 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 Relationship with Device Tree

ACPI support in drivers and subsystems for ARMv8 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).

3.4 Booting using ACPI tables

The only defined method for passing ACPI tables to the kernel on ARMv8 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 ARMv8 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 6.1 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
- If PCI is supported, the MCFG (Memory mapped ConFiGuration Table), section 5.2.6, specifically Table 5-31.
- If booting without a console=<device> kernel parameter is supported, the SPCR (Serial Port Console Redirection table), section 5.2.6, specifically Table 5-31.
- If necessary to describe the I/O topology, SMMUs and GIC ITSs, the IORT (Input Output Remapping Table, section 5.2.6, specifically Table 5-31).
- If NUMA is supported, the SRAT (System Resource Affinity Table) and SLIT (System Locality distance Information Table), sections 5.2.16 and 5.2.17, respectively.

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.

3.5 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.

3.6 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 [5]:

- UUID: daffd814-6eba-4d8c-8a91-bc9bbf4aa301
- https://www.uefi.org/sites/default/files/resources/_DSD-device-properties-UUID.

pdf

The UEFI Forum provides a mechanism for registering device properties [4] so that they may be used across all operating systems supporting ACPI. Device properties that have not been registered with the UEFI Forum should not be used.

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 [6]; it can eliminate some duplication of code paths in driver probing functions and discourage divergence between DT bindings and ACPI device properties.

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.

3.8 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.

3.9 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[] = {
```

(continues on next page)

(continued from previous page)

```
        { "LNRO0005", },
        { }
};
MODULE_DEVICE_TABLE(acpi, virtio_mmio_acpi_match);
```

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 6.1 release in January 2016.

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 ARMv8 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.

3.11 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 ARM64 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.

3.12 ACPI Objects

Detailed expectations for ACPI tables and object are listed in the file *ACPI Tables*.

3.13 References

[0] http://silver.arm.com
document ARM-DEN-0029, or newer: "Server Base System Architecture", version 2.3, dated 27 Mar 2014

[1] http://infocenter.arm.com/help/topic/com.arm.doc.den0044a/Server_Base_Boot_Requirements.pdf
Document ARM-DEN-0044A, or newer: "Server Base Boot Requirements, System Software on ARM Platforms", dated 16 Aug 2014

[2] http://www.secretlab.ca/archives/151,
10 Jan 2015, Copyright (c) 2015, Linaro Ltd., written by Grant Likely.

[3] AMD ACPI for Seattle platform documentation
http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/Seattle_ACPI_Guide.pdf

[4] http://www.uefi.org/acpi
please see the link for the "ACPI _DSD Device Property Registry Instructions"

[5] http://www.uefi.org/acpi
please see the link for the "_DSD (Device Specific Data) Implementation Guide"

[6] Kernel code for the unified device
property interface can be found in include/linux/property.h and drivers/base/property.c.

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

CHAPTER

FOUR

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 and EL1 having a secure and a non-secure counterpart. EL2 is the hypervisor level and exists only in non-secure mode. EL3 is the highest priority level and exists only in secure mode.

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

4.1 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.)

4.2 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.

4.3 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.

4.4 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 may be anywhere in 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 either EL2 (RECOMMENDED in order to have access to the virtualisation extensions) or non-secure 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 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.
  - 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.

  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 initialiased 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.

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.

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)

CHAPTER

FIVE

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.

5.1 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.

5.2 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
```

5.3 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=0,4,5,6,7
```

(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).

5.4 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
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
RPRES	[7-4]	y

5.5 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
```

(continues on next page)

(continued from previous page)

```
 * 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;
}
```

CHAPTER

SIX

ARM64 ELF HWCAPS

This document describes the usage and semantics of the arm64 ELF hwcaps.

6.1 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 auxilliary 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.

6.2 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.

6.3 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 100KHz.

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.

6.4 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.

CHAPTER

SEVEN

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.

7.1 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.

7.2 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	

CHAPTER

EIGHT

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.

8.1 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.

CHAPTER

NINE

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
ffff800000000000      ffff9fffffffffff          32TB          kasan␣
 ↪shadow region
ffffa00000000000      ffffa00007ffffff         128MB          bpf␣
 ↪jit region
ffffa00008000000      ffffa0000fffffff         128MB          ␣
 ↪modules
ffffa00010000000      fffffdffbffeffff         ~93TB          ␣
 ↪vmalloc
fffffdffbfff0000      fffffdfffe5f8fff        ~998MB          ␣
 ↪[guard region]
fffffdfffe5f9000      fffffdfffe9fffff        4124KB          fixed␣
```

(continues on next page)

(continued from previous page)

```
 ↪mappings
fffffdfffea00000      fffffdfffebfffff           2MB          ␣
 ↪[guard region]
fffffdfffec00000      fffffdffffbfffff          16MB           PCI I/
 ↪O space
fffffdffffc00000      fffffdffffdfffff           2MB          ␣
 ↪[guard region]
fffffdfffffe00000      fffffffffdfffff           2TB          ␣
 ↪vmemmap
ffffffffffe00000      ffffffffffffffff           2MB          ␣
 ↪[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      fff7fffffffffffff          2PB          ␣
 ↪kernel logical memory map
fff8000000000000      fffd9fffffffffff        1440TB           [gap]
fffda00000000000      ffff9fffffffffff         512TB           kasan␣
 ↪shadow region
ffffa00000000000      ffffa00007ffffff         128MB           bpf␣
 ↪jit region
ffffa00008000000      ffffa0000fffffff         128MB          ␣
 ↪modules
ffffa00010000000      fffff81fffefffff         ~88TB          ␣
 ↪vmalloc
fffff81fffff0000      fffffc1ffe58ffff          ~3TB          ␣
 ↪[guard region]
fffffc1ffe590000      fffffc1ffe9fffff        4544KB           fixed␣
 ↪mappings
fffffc1ffea00000      fffffc1ffebfffff           2MB          ␣
 ↪[guard region]
fffffc1ffec00000      fffffc1fffbfffff          16MB           PCI I/
 ↪O space
fffffc1fffc00000      fffffc1fffdfffff           2MB          ␣
 ↪[guard region]
fffffc1fffe00000      ffffffffffdfffff        3968GB          ␣
 ↪vmemmap
ffffffffffe00000      ffffffffffffffff           2MB          ␣
 ↪[guard region]
```

Translation table lookup with 4KB pages:

```
+--------+--------+--------+--------+--------+--------+--------+----
 ↪----+
|63    56|55    48|47    40|39    32|31    24|23    16|15     8|7   ␣
```

(continues on next page)

(continued from previous page)

```
↪   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_HARDEN_EL2_VECTORS is selected for particular CPUs.

When using KVM with the Virtualization Host Extensions, no additional mappings are created, since the host kernel runs directly in EL2.

9.1 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.

9.2 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.

CHAPTER

TEN

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.

10.1 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.

10.2 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`.

10.2.1 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.

10.2.2 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).

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 contain one of the following values in the PR_MTE_TCF_MASK bit-field:

- PR_MTE_TCF_NONE - *Ignore* tag check faults
- PR_MTE_TCF_SYNC - *Synchronous* tag check fault mode
- PR_MTE_TCF_ASYNC - *Asynchronous* tag check fault mode

The current tag check fault mode can be read using the prctl(PR_GET_TAGGED_ADDR_CTRL, 0, 0, 0, 0) system call.

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.

10.2.3 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.

10.2.4 Initial process state

On execve(), the new process has the following configuration:

- PR_TAGGED_ADDR_ENABLE set to 0 (disabled)
- Tag checking mode set to PR_MTE_TCF_NONE
- 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).

10.2.5 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)`).

10.3 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)
```

(continues on next page)

(continued from previous page)

```
# 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 MTE tag check␣
↪faults 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 | (0xfffe
↪<< PR_MTE_TAG_SHIFT),
                  0, 0, 0)) {
                perror("prctl() failed");
                return EXIT_FAILURE;
        }
```

(continues on next page)

(continued from previous page)

```
        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;
}
```

CHAPTER

ELEVEN

PERF EVENT ATTRIBUTES

Author
Andrew Murray <andrew.murray@arm.com>

Date
2019-03-06

11.1 exclude_user

This attribute excludes userspace.

Userspace always runs at EL0 and thus this attribute will exclude EL0.

11.2 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.

11.3 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.

11.4 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.

11.5 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.

CHAPTER

TWELVE

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.

12.1 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).

12.2 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.

Recent versions of GCC can compile code with APIAKey-based return address protection when passed the -msign-return-address option. This uses instructions in the HINT space (unless -march=armv8.3-a or higher is also passed), and such code can run on systems without the pointer authentication extension.

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”.

12.3 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.

12.4 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.

CHAPTER

THIRTEEN

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
ARM	Cortex-A53	#826319	ARM64_
ARM	Cortex-A53	#827319	ARM64_
ARM	Cortex-A53	#824069	ARM64_
ARM	Cortex-A53	#819472	ARM64_
ARM	Cortex-A53	#845719	ARM64_
ARM	Cortex-A53	#843419	ARM64_
ARM	Cortex-A55	#1024718	ARM64_
ARM	Cortex-A55	#1530923	ARM64_
ARM	Cortex-A57	#832075	ARM64_
ARM	Cortex-A57	#852523	N/A
ARM	Cortex-A57	#834220	ARM64_
ARM	Cortex-A57	#1319537	ARM64_
ARM	Cortex-A57	#1742098	ARM64_
ARM	Cortex-A72	#853709	N/A
ARM	Cortex-A72	#1319367	ARM64_
ARM	Cortex-A72	#1655431	ARM64_
ARM	Cortex-A73	#858921	ARM64_
ARM	Cortex-A76	#1188873,1418040	ARM64_
ARM	Cortex-A76	#1165522	ARM64_
ARM	Cortex-A76	#1286807	ARM64_
ARM	Cortex-A76	#1463225	ARM64_
ARM	Cortex-A77	#1508412	ARM64_
ARM	Cortex-A510	#2457168	ARM64_
ARM	Neoverse-N1	#1188873,1418040	ARM64_
ARM	Neoverse-N1	#1349291	N/A
ARM	Neoverse-N1	#1542419	ARM64_
ARM	MMU-500	#841119,826419	N/A
Broadcom	Brahma-B53	N/A	ARM64_
Broadcom	Brahma-B53	N/A	ARM64_
Cavium	ThunderX ITS	#22375,24313	CAVIUM
Cavium	ThunderX ITS	#23144	CAVIUM
Cavium	ThunderX GICv3	#23154	CAVIUM
Cavium	ThunderX GICv3	#38539	N/A
Cavium	ThunderX Core	#27456	CAVIUM
Cavium	ThunderX Core	#30115	CAVIUM
Cavium	ThunderX SMMUv2	#27704	N/A
Cavium	ThunderX2 SMMUv3	#74	N/A
Cavium	ThunderX2 SMMUv3	#126	N/A
Cavium	ThunderX2 Core	#219	CAVIUM
Marvell	ARM-MMU-500	#582743	N/A
Freescale/NXP	LS2080A/LS1043A	A-008585	FSL_ER

Table 1 – continued from previous page

Implementor	Component	Erratum ID	Kconfig
Hisilicon	Hip0{5,6,7}	#161010101	HISILIC
Hisilicon	Hip0{6,7}	#161010701	N/A
Hisilicon	Hip0{6,7}	#161010803	N/A
Hisilicon	Hip07	#161600802	HISILIC
Hisilicon	Hip08 SMMU PMCG	#162001800	N/A
Hisilicon	Hip08 SMMU PMCG Hip09 SMMU PMCG	#162001900	N/A
Qualcomm Tech.	Kryo/Falkor v1	E1003	QCOM_
Qualcomm Tech.	Kryo/Falkor v1	E1009	QCOM_
Qualcomm Tech.	QDF2400 ITS	E0065	QCOM_
Qualcomm Tech.	Falkor v{1,2}	E1041	QCOM_
Qualcomm Tech.	Kryo4xx Gold	N/A	ARM64_
Qualcomm Tech.	Kryo4xx Gold	N/A	ARM64_
Qualcomm Tech.	Kryo4xx Silver	N/A	ARM64_
Qualcomm Tech.	Kryo4xx Silver	N/A	ARM64_
Qualcomm Tech.	Kryo4xx Gold	N/A	ARM64_

Fujitsu	A64FX	E#010001	FUJITSU_ERRATUM_010001

CHAPTER

FOURTEEN

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).

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.

14.1 1. General

- SVE registers Z0..Z31, P0..P15 and FFR and the current vector length VL, are tracked per-thread.
- 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 HWCAP2_SVEBITPERM HWCAP2_SVESHA3 HWCAP2_SVESM4

  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.

- 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).
- 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).

14.2 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.

14.3 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 unspecified 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.

14.4 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 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.

14.5 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 desribed 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.

14.6 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 exceution 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, 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.

14.7 7. ptrace extensions

- A new regset NT_ARM_SVE is defined for use with PTRACE_GETREGSET and PTRACE_SETREGSET.

  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

either

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).

- 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).
- 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.
- The effect of writing a partial, incomplete payload is unspecified.

14.8 8. ELF coredump extensions

- A NT_ARM_SVE 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_SVE were executed for each thread when the coredump was generated.

14.9 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.

14.9.1 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.

14.10 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].

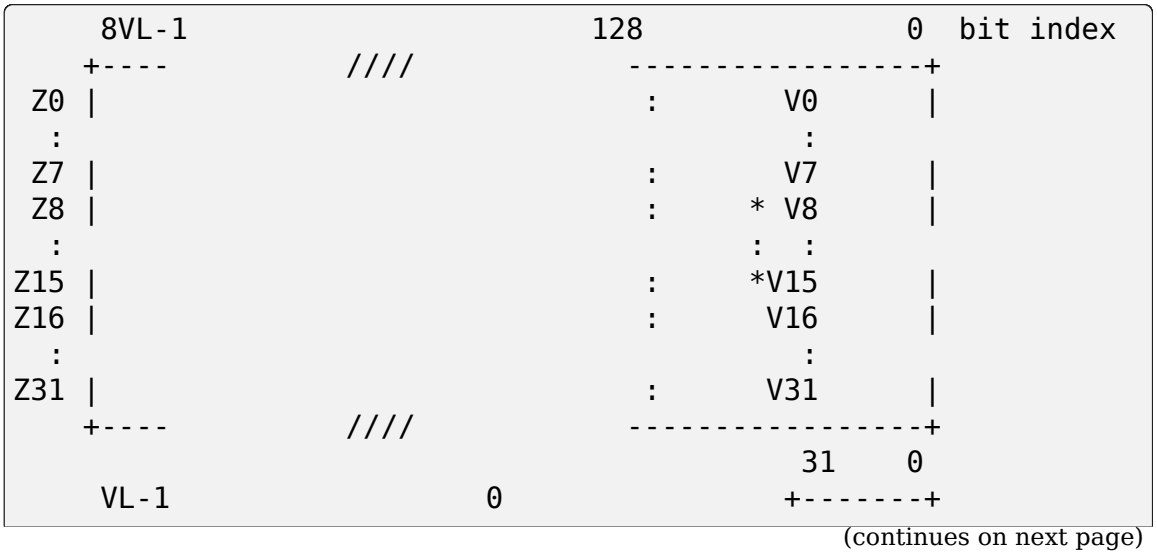
- 16 VL-bit predicate registers P0..P15
- 1 VL-bit special-purpose predicate register FFR (the "first-fault register")
- a VL "pseudo-register" that determines the size of each vector register

  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:



(continues on next page)

(continued from previous page)

```
    +----       ////      --+           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.

14.11 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.

14.11.1 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 |               |
    +---------------+
```

(continues on next page)

(continued from previous page)

```
              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.

14.11.2 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)

CHAPTER

FIFTEEN

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.

15.1 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.

15.2 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 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*.

15.3 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()`.

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.

15.4 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;
```

(continues on next page)

(continued from previous page)

```
    /* 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;
}
```

CHAPTER

SIXTEEN

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.

16.1 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.

16.2 Preserving tags

Non-zero tags are not preserved when delivering signals. This means that signal handlers in applications making use of tags cannot rely on the tag information for user virtual addresses being maintained for fields inside siginfo_t. One exception to this rule is for signals raised in response to watchpoint debug exceptions, where the tag information will be preserved.

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.

16.3 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.