Unified Discovery

Table of Contents

Introducion

Unified Discovery is intended to be a low-level discovery mechanism. A low-level discovery mechanism is distinguished from a high-level discovery mechanism. The former typically prepare and produce necessary data that is consumed by the latter. Diversified applications would require different high-level mechanism, meanwhile, the low-level mechanism provides the foundation.

This proposal describes a low-level discovery mechanism that is capable of supporting the following use cases:

1. Hosted discovery of features by firmware, operating systems and applications
    a. Rich operating systems
    b. Simple software applications
2. Discovery of features by external debug tools
3. Out-of-band discovery of features to allow development tools to specialise
4. firmware (e.g. choose the appropriate target flags for compilation and link in the required libraries) for deeply embedded applications

As an example, consider a typical Linux stack:

1. Firmware performs system/machine-dependent discovery and populates either a Linux device tree or ACPI tables.
2. The Linux kernel parses either a Linux device tree or ACPI tables, exposing system specifics to userland processes through multiple interfaces:
    a. special files the /proc filesystem
    b. device drivers available under /dev
    c. files in a mounted sysfs instance
    d. flags injected into ELF binaries through the ELF auxiliary vector and accessible through getauxval()
    e. configuration retrieved through the sysconf() system call
    f. information retrievable through the vdso (a virtual DSO mapped by the kernel into each processes' address space)

No Central Registry

The proposal provides a solution that does not require a central registry.

RISC-V allows the vendor-specific extensibility of the ISA without any coordination with RISC-V International (as long as no required features are removed and no incompatible features are introduced).

This should also be reflected in the architecture of the discovery format and require a minimum coordination between implementation and RISC-V International:

1. Based on the published "rules of the land" (i.e. modelling language, encoding rules and the top-level message description), implementers (both soft- and hardware) shall be able to:
    a. add proprietary entries in the configuration message, that can safely be identified, skipped and linked back to the vendor (without a global vendor registry being operated by RISC-V) that specified the proprietary entry
    b. parse any configuration message, including the ability:
        i. to parse a newer-version message, identifying the new "extensions" and being able to safely skip over them
        ii. To parse a message containing vendor-extensions
2. Publish a basic message format that enforces the presence of required fields as a machine-readable document/schema.

Complete Consumption Requirement

A key requirement in any discovery process that avoids a centralized registry is a client's ability to discover that it has read the entire message (including parts that it can not understand and skips over) and whether any unparsable extra elements were included in the message. This is termed the Complete Consumption Requirement.

NOTE	The complete consumption requirement is one of the unresolvable issues for CPUID-style instructions that query for values using keys.

Note on Security

Independent of the underlying mechanism (i.e. whether a memory-based configuration message is read or CPUID-style instructions are used), securing the discovery mechanism will require cryptographically signed checksums (i.e. electronic signatures) to ascertain the authenticity, integrity and the originator of the configuration data.

Signing the configuration message should be an integral (albeit optional) part of the message format. While this can not address the playback of a valid configuration message, it allows the discovery of modified messages.

We do not believe that the goal of end-to-end security can be efficiently achieved using a design-approach similar to Intel's CPUID instruction: a cryptographic signature would need to be computed across the entire configuration space (and not merely individual elements). This precludes the absence of a central registry, as the valid key space needs to be known in advance to concatenate the plaintext for signing.

Solution Outline

1. Schema + Value Notation (human readable form) + Parser
2. Reuse existing standards → ITU standards & examples (SNMP)
3. vendor-specific info

A binary-encoded representation of a device's configuration is made available to software within the device's physical address space. The data structure is described as a subset of ASN.1 (see ITU-T X.680 and ISO/IEC 8824) and encoded using standardized encoding rules (see ITU-T X.690 and ISO 8825). For in-memory representations, the unaligned packed encoding rules (unaligned PER, see ITU-T X.691) are used. The configuration data can (optionally) be cryptographically signed.

This proposal provides a schema of the data structure that is generic and extensible. See Section 8 for the schema. Vendor-specific data can be included without hindering the successful parsing of the configuration.(?)

The base-address of the binary-encoded representation is accessible through a single CSR. No other ISA considerations, beyond the provision of an additional CSR, are required.

Target software (usually firmware) that performs discovery will read the uPER-encoded message to retrieve the relevant configuration elements. The message can be decoded either using a stream parser with small memory footprint (i.e. the parser reads from the beginning until it retrieves the requested data element) or can be converted start-to-finish into a firmware-specific data structure. Given the compact representation and the low memory requirements for parsers, a uPER message can be efficiently parsed even during the startup of a deeply embedded microcontroller application (even though we envision out-of-band discovery and specialization for deeply embedded and resource-constrained use-cases).

The unified discovery mechanism for RISC-V builds on the following technology stack:

1. ASN.1 (X.680) for modelling the data structures, independent of their encoding
2. Packed Encoding Rules (X.691) for the binary encoding of data structures (in-band)
3. XML Encoding Rules (X.693) for the XML encoding of data structures (out-of-band)
4. RISC-V International specific guidelines to allow the efficient aggregation of RISC-V global and vendor-specific data elements without a central registration authority
5. RISC-V International specific guidelines for the encoding of detached signatures (PKCS#7/CMS) using Packed Encoding Rules

NOTE	The benefits of using X.680 and X.693 over vendor-specific (e.g., Google Protobuf, Apache Avro, …) marshalling frameworks are its international standardization, widespread adoption and availability of open-source and commercial codec libraries.

Retrieval and decoding of the configuration structure can happen in any of the following scenarios:

- Software (in-band)

Firmware will access the CSR and read the configuration message to extract the device's configuration as part of its discovery process. The implementation details of this process (e.g., whether firmware initiates a read from the top and searches for individual tags, or if firmware converts the entire discovery information into an in-memory representation at once) are left to device implementers.

- External debug (in-band)

External debug will retrieve the CSR and then read out (once) the referenced memory region to retrieve the configuration information for a specific target device. The retrieved configuration message is then parsed by the external debugger to determine the configuration, features and capabilities of the device.

- Software development environment (out-of-band)

For (deeply) embedded applications, firmware will be specialised to target the specific target device only by pushing the discovery and configuration to the software development environment. These cases can be efficiently supported either by reading the configuration structure from a target device using an external debugger, or by retrieving a configuration structure from the manufacturer's website.

The mconfigptr CSR

The machine config pointer (mconfigptr) CSR provides the base-address of the binary-encoded representation. The mconfigptr is a machine-mode CSR. On platforms that does not require runtime update of the address of the binary representation of the configuration, this register can be hardwired to zero.

For backward compatibility, the firmware can emulate this CSR on platforms that does not implement this CSR prior to this proposal.

Hypervisor // U/D for guest OSes

For virtualisation purposes, only the retrieval of the mconfigptr CSR has to be intercepted (i.e. either a virtualized CSR would be provided to the guest that can be written by the hypervisor — or trap-and-emulate would be used) if the guest is to be provided with a configuration structure from what is retrieved from the underlying hardware.

Referenced standards

Guidelines/Mappings from discoverable elements → ASN.1

Extensibility, versioning & “container format”

What types of discoverable elements do we support?

Existence

Structural elements (lists, arrays)

Parameters (enums, integer ranges, addresses)

How to map these to ASN.1

Encoding rules

Reference back to X.69x ?

Top-level schema → appendix (normative)

container format

standard elements (vectors, bitmanip, …)