# oneAPI Technical Advisory Board Meeting: Extension Mechanism

08-26-2020

Greg Lueck

# Why Have an Extension?

- Vendor defined extension
  - Expose hardware feature specific to that vendor
  - Gather feedback on new APIs that are not yet ready for inclusion in SYCL spec
- Khronos defined extension
  - Several vendors agree to expose hardware features in a common way
  - Gather feedback before proposing API in next rev of SYCL spec
- Extensions are not part of SYCL specification
  - Vendor need not support any extension to be SYCL conformant
  - Extension has its own specification (from vendor or from Khronos)

# What is an Extension?

- Very broad, some examples:
  - Add new types or functions
  - Add new member functions to existing classes
  - Add new C++ attributes or language keywords
- Can add feature that is always available (on host, on all devices)
- Can add feature that is available only on some devices
- An extension cannot:
  - Remove any feature from the core SYCL spec
  - Change the behavior of any existing SYCL feature

# Goals of the Extension Mechanism

- For application developers
  - Way to write portable code that works on implementations that do / don’t have extension
- For vendors
  - Way to ensure extension doesn’t collide with another vendor’s extension
  - Way to ensure extension doesn’t collide with future version of SYCL spec

# Requirement for Vendor Providing Extension

- Must provide a feature-test macro
  - Allows application to conditionally use extension
- Does not avoid collisions between vendor extensions
  - Some vendors don’t care about this
  - Want maximum flexibility in extension naming

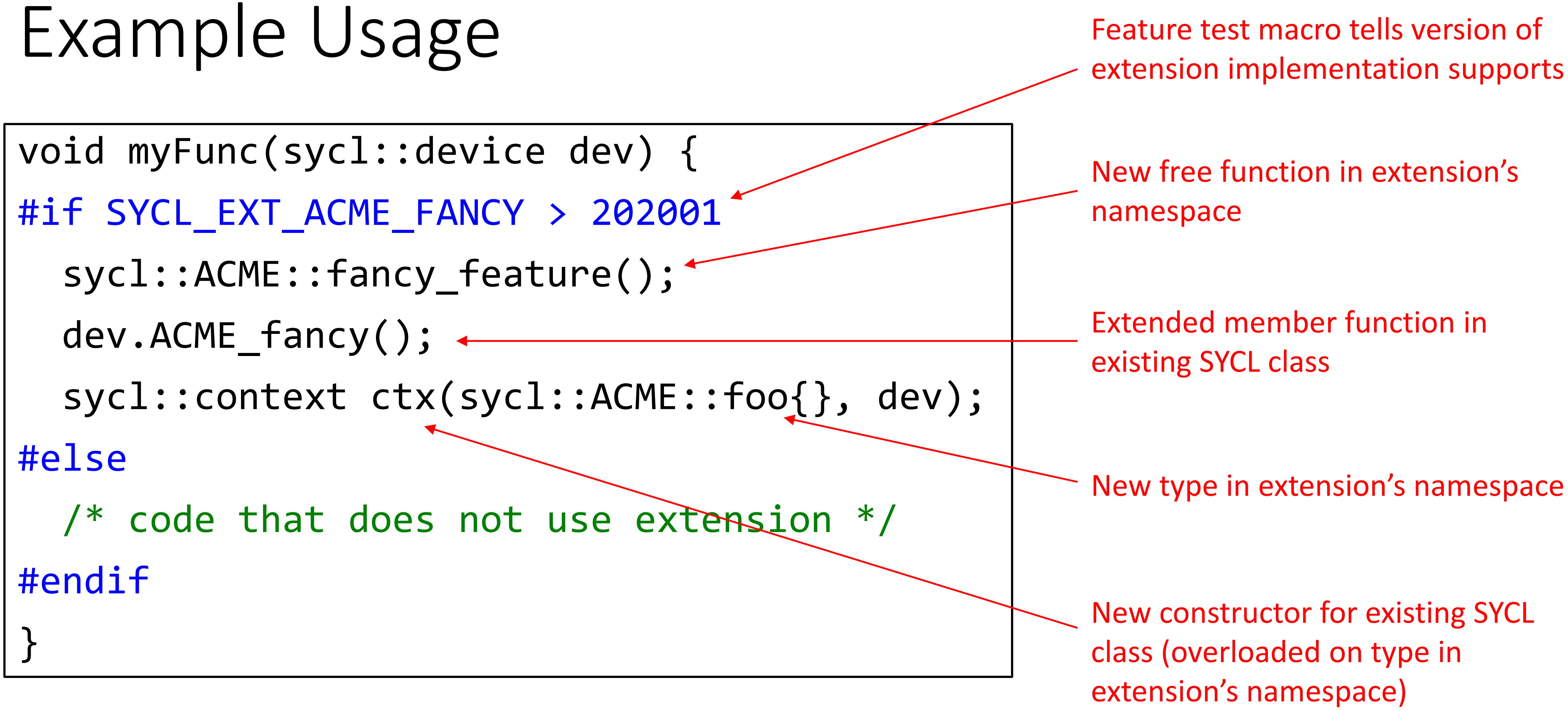
```
void myFunc() {
#ifdef SYCL_IMPLEMENTATION_ACME
  fancy_feature();
#endif
}
```

```
void myFunc() {
#ifdef SYCL_EXT_ACME_FANCY
  fancy_feature();
#endif
}
```

# Guidelines to Prevent Collisions

- Vendor chooses a “vendor string”
  - E.g. Acme corporation could choose “acme”
  - Vendor responsible for ensuring uniqueness (no Khronos registry)
- Khronos promises it will never use capital letters for identifiers
  - Extensions can use capital letters to avoid collision with future SYCL specs
- Illustrating Acme’s extension:
  - Namespace for new types / function: “sycl::ACME”
  - Adding member function to existing SYCL class: “device::ACME_fancy()”
  - Adding constructor to existing class: “context::context(sycl::ACME::foo &)”

# Example Usage



```
void myFunc(sycl::device dev) {
#if SYCL_EXT_ACME_FANCY > 202001
  sycl::ACME::fancy_feature();
  dev.ACME_fancy();
  sycl::context ctx(sycl::ACME::foo{}, dev);
#else
  /* code that does not use extension */
#endif
}
```

Feature test macro tells version of extension implementation supports

New free function in extension’s namespace

Extended member function in existing SYCL class

New type in extension’s namespace

New constructor for existing SYCL class (overloaded on type in extension’s namespace)

Style also makes it clear that APIs are part of “Acme” extension

# Device Specific Features

- Not strictly related to extensions, core SYCL features too:
  - sycl::half
  - 64-bit atomics
  - Images

All optional features that may not be supported on all devices

- Need a general strategy for optional device features
- Extensions that add features only on some devices use same strategy

# Device Aspects

```
namespace sycl {
enum class aspect {
  fp16,
  int64_base_atomics,
  int64_extended_atomics,
  image,
  /* ... */
};
}
```

Tells if device supports sycl::half

Tells if device supports 64-bit atomics

Tells if device supports images

```
void myFunc() {
  sycl::device dev = /* ... */;
  if (dev.has(sycl::aspect::fp16)) {
    // We know that sycl::half
    // is supported on device
  }
}
```

# Choose Kernel Based on Device Aspects

```
q.submit([&](handler& cgh) {
  if (q.get_device().has(aspect::fp16)) {
    cgh.parallel_for(range{N}, [=](id i) {
      half fp = /* ... */;
    });
  } else {
    cgh.parallel_for(range{N}, [=](id i) {
      /* don't use "half" type */
    });
  }
});
```

# Templatized Kernel Avoids Code Duplication

```
template<bool hasFP16> class Kernel {
 public:
  void operator()(id i) {
    /* ... */
    if constexpr (hasFP16) {
      /* use "half" type */
    } else {
      /* fallback */
    }
    /* ... */
  };
};
```

```
q.submit([&](handler& cgh) {
  device d = q.get_device();
  if (d.has(aspect::fp16)) {
    Kernel<true> k;
    cgh.parallel_for(range{N}, k);
  } else {
    Kernel<false> k;
    cgh.parallel_for(range{N}, k);
  }
});
```

# Extension Adding Device Specific Feature

```
q.submit([&](handler& cgh) {
  if (q.get_device().has(aspect::ACME_fancy)) {
    cgh.parallel_for(range{N}, [=](id i) {
      ACME::fancy();
    });
  }
});
```

Extended aspect to test for feature

Call feature only from device that supports it

# Error Behavior

```
q.submit([&](handler& cgh) {
  cgh.parallel_for(range{N},
      [=](id i) {
    ACME::fancy();
  });
});
```

- What if device doesn’t support “ACME_fancy”?
- Compiler can’t raise diagnostic
  - Can’t tell at compile time what aspects device supports
- Implementation raises synchronous exception when kernel submitted

# Attribute for Compile Time Checking

```
cgh.parallel_for(range{N}, [=](id i) [[sycl::requires(has(ACME_fancy))]] {
  long_device_function();
});
```

- Programmer’s assertion that kernel should only use “ACME_fancy” aspect.
- Compiler (may) raise diagnostic if any device code in this kernel requires a different aspect.
- Still no guarantee that device supports “ACME_fancy”.  Still get an exception if it does not.

# Recap: Check Device Aspect in Host Code

```
q.submit([&](handler& cgh) {
  if (q.get_device().has(aspect::fp16)) {
    cgh.parallel_for(range{N}, [=](id i) {
      half fp = /* ... */;
    });
  } else {
    cgh.parallel_for(range{N}, [=](id i) {
      /* don't use "half" type */
    });
  }
});
```

# Future: Check Device Aspect in Device Code

```
cgh.parallel_for(range{N}, [=](id i) {
  if devconstexpr (this_device::has(aspect::fp16)) {
    half fp = /* ... */;
  } else { /* fallback */ }
});
```

```
cgh.parallel_for(range{N}, [=](id i) {
 device_dispatch(this_device::has(aspect::fp16), [&](auto B) {
    if constexpr (B) { half fp = /* ... */; }
    else { /* fallback */ }});
});
```

# Analysis

- Advantages
  - Avoids need to templatize kernel to avoid code duplication
  - Calling code does not need to know aspect requirements of kernel
    (important for middleware developers who ship device code libraries)
- Why not use macro?
  - Hides some code paths from device compiler
    (can have different lambda captures depending on device)
  - Assumes multi-pass implementation, spec says single pass is possible

# Discussion Topics

- Does [[sycl::requires()]] seem useful?
- Is it better to raise an async exception only if kernel executes unsupported feature?
  - Allows submission of kernel using unsupported feature if not executed
  - Easier to miss diagnosing an invalid kernel if code path rarely executed
- What if not all vendors can support synchronous exception?
  - E.g. library only implementation probably could only support async exception
  - Better to specify async exception (lowest common denominator) or give implementations choice of async vs. sync?
- Thoughts on checking aspects from device code

# Rules of the Road

- DO NOT share any confidential information or trade secrets with the group
- DO keep the discussion at a High Level
  - Focus on the specific Agenda topics
  - We are asking for feedback on features for the oneAPI specification (e.g. requirements for functionality and performance)
  - We are NOT asking for feedback on any implementation details
- Please submit any implementation feedback in writing on Github in accordance with the Contribution Guidelines at spec.oneapi.com. This will allow Intel to further upstream your feedback to other standards bodies, including The Khronos Group SYCL* specification.

# Notices and Disclaimers

The content of this oneAPI Specification is licensed under the Creative Commons Attribution 4.0 International License . Unless stated otherwise, the sample code examples in this document are released to you under the MIT license.

This specification is a continuation of Intel’s decades-long history of working with standards groups and industry/academia initiatives such as The Khronos Group*, to create and define specifications in an open and fair process to achieve interoperability and interchangeability. oneAPI is intended to be an open specification and we encourage you to help us make it better. Your feedback is optional, but to enable Intel to incorporate any feedback you may provide to this specification, and to further upstream your feedback to other standards bodies, including The Khronos Group SYCL* specification, please submit your feedback under the terms and conditions below. Any contribution of your feedback to the oneAPI Specification does not prohibit you from also contributing your feedback directly to The Khronos Group or other standard bodies under their respective submission policies.

By opening an issue, providing feedback, or otherwise contributing to the specification, *you agree that Intel will be free to use, disclose, reproduce, modify, license, or otherwise distribute your feedback in its sole discretion without any obligations or restrictions of any kind, including without limitation, intellectual property rights or licensing obligations.* For complete contribution policies and guidelines, see Contribution Guidelines on www.spec.oneapi.com.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice.

Intel and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.*Other names and brands may be claimed as the property of others.

© Intel Corporation