1 Introduction

Device Modeling Language 1.4 Reference Manual

1 Introduction

1 Introduction

Device Modeling Language (DML) is a domain-specific programming language for developing device models to be simulated with Simics. DML has been designed to make it easy to represent the kind of things that are needed by a device model, and uses special syntactic constructs to describe common elements such as memory mapped hardware registers, connections to other Simics configuration objects, and checkpointable device state.

DML is an object-oriented language, where each device is represented through an object, which — as members — may feature pieces of mutable state, configurable parameters and attributes, subroutines (called methods), and subobjects that may have their own members. In contrast to most general-purpose object-oriented languages, objects in DML are statically declared rather than dynamically created, and typically represent logical components of the device.

A complete DML model specifies exactly one device model, together with:

- Associated register banks, and how these may be memory mapped
- Specifications of connections to other devices that the device expects to have access to, and thus may make use of.
- Specifications of connections that other devices may establish to the device, and how messages sent through those connections are handled by the device.
- Specification of attributes that Simics may access for the purposes of configuring the device, inspect the device operation, or to checkpoint the device state.
- The name and description of the device, together with other static meta-information

These are the crucial properties of device models that must be made visible to Simics, and each of these have specialized language features in order to declare them. Beyond these, DML also has a number of features to improve the expressive power of the language and simplify development; for instance, DML offers *templates*, a powerful metaprogramming tool that allows for code reduction and reuse, as well as a means of building abstractions.

The DML compiler is called *Device Modeling Language Compiler*, `dmlc`. It translates a device model description written in DML into C source code that can be compiled and loaded as a Simics module.

This document describes the DML language, the standard libraries, and the `dmlc` compiler, as of version 1.4 of DML. See also *Simics Model Builder User's Guide* for an introduction to DML.

1.1 Source File Example

The following is an example of a small DML model defining a very simple device. This lacks many details that would appear in a real device.

```
dml 1.4;

device contraption;

connect wakeup {
    interface signal;
}

bank config_registers {
    register cfg1 size 4 @ 0x0000 {
        field status @ [7:6] is (read, write) {
            method read() -> (uint64) {
                local uint2 ret;
                ret[0] = enable.val;
                ret[1] = coefficient[1] & 1;
                return ret;
            }
        }
        field enable @ [8] is (read_unimpl, write) {
            method write(uint64 val) {
                if (this.val == 0 and val == 1) {
                    wakeup.signal.signal_raise();
                } else if (this.val == 1 and val == 0) {
                    wakeup.signal.signal_lower();
                }
            }
        }
    }

    register coefficient[i < 4] size 8 @ 0x0008 + i * 8 is (read, write) {}
}
```

- The `device contraption;` statement declares the name of the device.
- The `connect wakeup` part declares that the device can be configured to communicate with other devices using the `signal` interface.
- The `bank config_registers` part declares a bank of memory-mapped registers. If the bank is mapped into a memory space, then software can use this to control the device through reads and writes.
- The bank contains *registers*, which declare sizes and offsets statically. When the bank is accessed with a memory transaction, it will check which register the transaction hits, and redirect as a read or write operation in that register.
- The `cfg1` register is further subdivided into *fields*, one covering bits 7 and 6 and one covering bit 8.
- The bank, registers and fields form a *hierarchy* of objects, which provides a simple mechanism for encapsulation. Each object is a separate namespace, and the object hierarchy naturally forms a nested scope.

- The `is` statement specifies a number of *templates* to be instantiated for the associated object. The `read` and `write` templates prepare code for the targeted field which makes it software readable and writable, as well as methods `read` and `write` that may be overridden in order to customize the behavior upon a software read or write. In contrast, the `read_unimpl` template prepares code that causes the field to log a warning if read by software.

- Methods in DML are much like functions in C. The statements of the method body are similar to C with some differences; e.g., integers support bitslicing using the syntax `value[a:b]`. Methods also have a place in the object hierarchy, and can freely access the object's state and connections.

- `coefficient` is an *array* of register objects, marked by the use of `[i < size]`. The object specification provided to an object array is used for each element of the array, and the `i` parameter can be used for element-specific logic. In this case, `i` is used to assign each register of the array to different address offsets.

1.2 Object types

The above example demonstrates how a DML device is built from a hierarchy of objects, such as banks and register. The hierarchy is composed of the following object types:

- Each DML model defines a single `device` object, which can be instantiated as a configuration object in Simics. All objects declared at the top level are members of the device object.

- An `attribute` object creates a Simics configuration attribute of the device. An attribute usually has one of three uses:

  - Configuring some aspect of the device on instantiation
  - Saving and restoring simulation state for checkpointing. For simple types this is easier to achieve with saved variables, but attributes can be necessary to serialize more complex types.
  - Exposing a back-door into the model for inspection or debugging, called a *pseudo* attribute

- A `bank` object makes sets of registers accessible by placing them in an *address space*. Register banks can be individually mapped into Simics memory spaces.

- A `register` object holds an integer value, and is generally used to model a hardware register, used for communication via memory-mapped I/O. A register is between 1 and 8 bytes wide. Registers divide the address space of a bank into discrete elements with non-overlapping addresses.

- `field` objects constitute a further subdivision of `register` objects, on the bit level. Each field can be accessed separately, both for reading and for writing. The fields of a register may not overlap.

- `group` is the general-purpose object type, without any special properties or restrictions. Groups are mainly used as container objects — in particular, to define logical groups of registers within a bank. The generic nature of groups also makes them useful as a tool for creating abstractions.

- A `connect` object holds a reference to a Simics configuration object. (Typically, the connected object is expected to implement some particular Simics-interface.) An attribute with the same name is added to the `device`; thus, a `connect` is similar to a simple `attribute` object. Usually, initialization is done when the device is configured, e.g. when loading a checkpoint or instantiating a component.

- An `interface object` may be declared within a `connect` object, and specifies a Simics interface assumed to be implemented by the connected object. In many cases, the name of the interface is sufficient, and the body of the object can be left empty.
- A `port object` represents a point where an outside device can connect to this device. This is done by creating a separate Simics object; if a device has a declaration `port irq` and the device is instantiated as `dev` in Simics, then the port object is named `dev.port.irq`.
- An `implement object` specifies an implementation of a *Simics interface* that the device implements. An `implement` object is normally declared inside a `port`, and defines the interfaces registered on the corresponding Simics configuration object; however, `implement` can also be declared on the top-level `device` object or in a `bank` object.

  The methods defined within the `implement` object must correspond to the functions of the Simics interface.

  A device can implement the same interface several times, by creating multiple `port` objects with `implement` objects of the same name.
- An `event object` is an encapsulation of a Simics event, that can be posted on a time queue (CPU or clock).
- A `subdevice object` represents a subsystem of a device, which can contain its own ports, banks, and attributes.

1.3 Methods and Mutable Variables

Methods are the DML representation of subroutines. They may be declared as members of any object or template. Any method may have multiple input parameters, specified similarly as C functions. Unlike C, DML methods may have multiple return values, and the lack of a return value is indicated through an empty list of return values rather than `void`. The following demonstrates a method declaration with no input parameters or return values:

```
method noop() -> () {
    return;
}
```

Alternatively:

```
method noop() {
    return;
}
```

The following demonstrates a method declaration with multiple input and parameters and return values:

```
method div_mod(uint64 dividend, uint64 divisor)
    -> (uint64, uint64) {
    local uint64 quot = dividend / divisor;
    local uint64 rem = dividend % divisor;
    return (quot, rem);
}
```

This also demonstrates how local, stack-allocated variables within methods may be declared; through the `local` keyword. This is analogous to C's `auto` variable kind, but unlike C, the keyword must be explicitly given. DML features two other variable kinds: `session` and `saved`. Unlike `local` variables, `session` and `saved` variables may also be declared as members of any object within the DML model, and can only be initialized with constant expressions.

`session` variables represent statically allocated variables, and act as the DML equivalent of static variables in C. The value of a `session` variable is preserved for the duration of the current simulation session, but are not automatically serialized and restored during checkpointing. This means that it is the model developer's responsibility to manually serialize and restore any `session` variables upon saving or restoring a checkpoint. `saved` variables behave exactly like `session` variables, except the value of `saved` variables are serialized and restored during checkpointing. Because of this, a `saved` variable must be of a type that DML knows how to serialize. Most built-in non-pointer C types are serializable, and any `struct` that consists solely of serializable types are also considered serializable. Pointer types are never considered serializable.

Methods have access to a basic exception-handling mechanism through the `throw` statement, which raises an exception without associated data. Such exceptions may be caught via the `try { ... } except { ... }` statement. If a method may throw an uncaught exception, that method must be declared `throws`; for example:

```
method demand(bool condition) throws {
    if (!condition) {
        throw;
    }
}
```

1.4 Templates and Parameters

A *template* specifies a block of code that may be inserted into objects. Templates may only be declared at the top-level, which is done as follows:

```
template name { body }
```

where *name* is the name of the template, and *body* is a set of object statements. A template may be instantiated through the `is` object statement, which can be used within either objects, or within templates. For example:

```
bank regs {
    // Instantiate a single template: templateA
    is templateA;
    // Instantiate multiple templates: templateB and templateC
    is (templateB, templateC);

    register reg size 1 @0x0;
}
```

The `is` object statement causes the body of the specified templates to be injected into the object or template in which the statement was used.

`is` can also be used in a more idiomatic fashion together with the declaration of an object or template as follows:

```
// Instantiate templates templateA, templateB, and templateC
bank regs is (templateA, templateB, templateC) {
    register reg size 1 @0x0;
}
```

A language feature closely related to templates are *parameters*. A parameter can be thought of as an *expression macro* that is a member of a particular object or template. Parameters may optionally be declared without an accompanying definition — which will result in a compile-time error if not overridden — or with a *default*, overridable definition. Parameters declared this way can be overridden by any later declaration of the same parameter. This can be leveraged by templates in order to declare a parameter that the template may make use of, while requiring any instance of the template to provide a definition for the parameter (or allow instances to override the default definition of that parameter).

Parameters are declared as follows:

- Abstract, without definition:

  ```
  param name;
  ```

- With overridable definition:

  ```
  param name default expression;
  ```

- With unoverridable definition:

  ```
  param name = expression;
  ```

Much of the DML infrastructure, as well as DML's built-in features, rely heavily on templates. Due to the importance of templates, DML has a number of features to generically manipulate and reference template instances, both at compile time and at run time. These are *templates as types*, `each-in` expressions, and `in each` declarations.

2 The DML compiler

A DML source file can be compiled into a runnable device model using the DML compiler, `dmlc`. The main output of the compiler is a C file, that can be compiled into a Simics module.

The DML compiler and its libraries are available as part of the *Simics Base* package.

2.1 Building dmlc

The `dmlc` compiler can be build locally. This requires an installation of the Simics 6 base package.

In order to build the compiler, checkout the DML repository into the into the `modules/dmlc` subdirectory of your Simics project. The compiler can be built using the `make dmlc` command. The build result ends up in *`host`*`/bin/dml` (where *`host`* is `linux64` or `win64`), and consists of three parts:

- *`host`*`/bin/dml/python` contains the Python module that implements the compiler
- *`host`*`/bin/dml/1.4` contains the standard libraries required to compile a device
- *`host`*`/bin/dml/api` contains `.dml` files that expose the Simics API

In order to use a locally built version of `dmlc` to compile your devices, you can add the following line to your `config-user.mk` file:

```
DMLC_DIR = $(SIMICS_PROJECT)/$(HOST_TYPE)/bin
```

2.2 Running dmlc

The syntax for running `dmlc` from the command line is:

```
dmlc [options] input [output-base]
```

where *input* should be the name of a DML source file. If *output-base* is provided, it will be used to name the created files. The name of a DML source file should normally have the suffix "`.dml`". The *input* file must contain a device declaration.

The main output of `dmlc` is a C file named *`<output-base>`*`.c`, which that can be compiled and linked into a Simics module using the `gcc` compiler. The compiler also produces some other helper files with `.h` and `.c` suffix, which the main output file includes.

2.3 Command Line Options

The following are the available command line options to `dmlc`:

-h, --help

Print usage help.

-I *path*

Add *path* to the search path for imported modules.

-D *name=definition*

Define a compile-time parameter. The definition must be a literal expression, and can be a quoted string, a boolean, an integer, or a floating point constant. The parameter will appear in the top-level scope.

--dep

Output makefile rules describing dependencies.

-T

Show tags on warning messages. The tags can be used with the `--nowarn` and `--warn` options.

-g

Generate artifacts that allow for easier source-level debugging. This generates a DML debug file leveraged by debug-simics, and causes generated C code to follow the DML code more closely.

--coverity

Adds Synopsys® Coverity® analysis annotations to suppress common false positives in generated C code created from DML 1.4 device models. It also allows for false positives to be suppressed manually through the use of the `COVERITY` pragma

Analysis annotation generation impacts the generation of line directives in a way that may cause debugging or coverage tools besides Coverity to display unexpected behavior. Because of this, it's recommended that `--coverity` is only used when needed.

--warn=*tag*

Enable selected warnings. The tags can be found using the `-T` option.

--nowarn=*tag*

Suppress selected warnings. The tags can be found using the `-T` option.

--werror

Turn all warnings into errors.

--strict

Report errors for some constructs that will be forbidden in future versions of the DML language

--noline

Suppress line directives for the C preprocessor so that the C code can be debugged.

--info

Enable the output of an XML file describing register layout.

--version

Print version information.

--simics-api=*version*

Use Simics API version *version*.

--max-errors=*N*

Limit the number of error messages to *N*.

1 Introduction | 3 Device Modeling Language, version 1.4

3 Device Modeling Language, version 1.4

This chapter describes the Device Modeling Language (DML), version 1.4. It will help to have read and understood the object model in the previous chapter before reading this chapter.

3.1 Overview

DML is not a general-purpose programming language, but a modeling language, targeted at writing Simics device models. The algorithmic part of the language is similar to ISO C; however, the main power of DML is derived from its simple object-oriented constructs for defining and accessing the static data structures that a device model requires, and the automatic creation of bindings to Simics.

Furthermore, DML provides syntax for *bit-slicing*, which much simplifies the manipulation of bit fields in integers; `new` and `delete` operators for allocating and deallocating memory; a basic `try`/`throw` mechanism for error handling; built-in `log` and `assert` statements; and a powerful metaprogramming facility using *templates* and *in each statements*.

Most of the built-in Simics-specific logic is implemented directly in DML, in standard library modules that are automatically imported; the `dmlc` compiler itself contains as little knowledge as possible about the specifics of Simics.

3.2 Lexical Structure

A major difference from C is that names do not generally need to be defined before their first use. This is quite useful, but might sometimes appear confusing to C programmers.

Character encoding

DML source files are written using UTF-8 encoding. Non-ASCII characters are only allowed in comments and in string literals. Unicode BiDi control characters (U+2066 to U+2069 and U+202a to U+202e) are not allowed. String values are still handled as byte arrays, which means that a string value written with a literal of three characters may actually create an array of more than three bytes.

Reserved words

All ISO/ANSI C reserved words are reserved words in DML (even if currently unused). In addition, the C99 and C++ reserved words `restrict`, `inline`, `this`, `new`, `delete`, `throw`, `try`, `catch`, and `template` are also reserved in DML. The C++ reserved words `class`, `namespace`, `private`, `protected`, `public`, `using`, and `virtual`, are reserved in DML for future use; as are identifiers starting with an underscore (_).

The following words are reserved specially by DML: `after`, `assert`, `call`, `cast`, `defined`, `each`, `error`, `foreach`, `in`, `is`, `local`, `log`, `param`, `saved`, `select`, `session`, `shared`, `sizeoftype`, `typeof`, `undefined`, `vect`, `where`, `async`, `await`, `with`, and `stringify`.

Identifiers

Identifiers in DML are defined as in C; an identifier may begin with a letter or underscore, followed by any number of letters, numbers, or underscores. Identifiers that begin with an underscore (_) are reserved by the DML language and standard library and should not be used.

Constant Literals

DML has literals for strings, characters, integers, booleans, and floating-point numbers. The integer literals can be written in decimal (`01234`), hexadecimal (`0x12af`), or binary (`0b110110`) form.

Underscores (_) can be used between digits, or immediately following the `0b`, `0x` prefixes, in integer literals to separate groups of digits for improved readability. For example, `123_456`, `0b10_1110`, `0x_eace_f9b6` are valid integer constants, whereas `_78`, `0xab_` are not.

String literals are surrounded by double quotes (`"`). To include a double quote or a backslash (`\`) in a string literal, precede them with a backslash (`\"` and `\\`, respectively). Newline, carriage return, tab and backspace characters are represented by `\n`, `\r`, `\t` and `\b`. Arbitrary byte values can be encoded as `\x` followed by exactly two hexadecimal digits, such as `\x1f`. Such escaped byte values are restricted to 00-7f for strings containing Unicode characters above U+007F.

Character literals consist of a pair of single quotes (`'`) surrounding either a single printable ASCII character or one of the escape sequences `\'`, `\\`, `\n`, `\r`, `\t` or `\b`. The value of a character literal is the character's ASCII value.

Comments

C-style comments are used in DML. This includes both in-line comments (`/*...*/`) and comments that continue to the end of the line (`//...`).

3.3 Module System

DML employs a very simple module system, where a *module* is any source file that can be imported using the `import` directive. Such files may not contain a [`device` declaration], but otherwise look like normal DML source files. The imported files are merged into the main model as if all the code was contained in a single file (with some exceptions). This is similar to C preprocessor `#include` directives, but in DML each imported file must be possible to parse in isolation, and may contain declarations (such as `bitorder`) that are only effective for that file. Also, DML imports are automatically idempotent, in the sense that importing the same file twice does not yield any duplicate definitions.

The import hierarchy has semantic significance in DML: If a module defines some method or parameter declarations that can be overridden, then *only* files that explicitly import the module are allowed to override these declarations. It is however sufficient to import the module indirectly via some other module. For instance, if A.dml contains a default declaration of a method, and B.dml wants to override it, then B.dml must either import A.dml, or some file C.dml that in turn imports A.dml. Without that import, it is an error to import both A.dml and B.dml in the same device.

3.4 Source File Structure

A DML source file describes both the structure of the modeled device and the actions to be taken when the device is accessed.

A DML source file defining a device starts with a *language version declaration* and a *device declaration*. After that, any number of *parameter declarations*, *methods*, *data fields*, *object declarations*, or *global declarations* can be written. A DML file intended to be *imported* (by an `import` statement in another DML file) has the same layout except for the device declaration.

3.4.1 Language Version Declaration

Every DML source file should contain a version declaration, on the form `dml 1.4;`. The version declaration allows the `dmlc` compiler to select the proper versions of the DML parser and standard libraries to be used for the file. A file can not import a file with a different language version than its own.

The version declaration must be the first declaration in the file, possibly preceded by comments. For example:

```
// My Device
dml 1.4;
...
```

3.4.2 Device Declaration

Every DML source file that contains a device declaration is a *DML model*, and defines a Simics device class with the specified name. Such a file may *import* other files, but only the initial file may contain a device declaration.

The device declaration must be the first proper declaration in the file, only preceded by comments and the language version declaration. For example:

```
/*
 *  My New Device
 */
dml 1.4;
device my_device;
...
```

3.5 Pragmas

DML has a syntax for pragmas: directives to the DML compiler that are orthogonal to DML as a language, both in the sense of that they are not considered part of DML proper, and that their use do not affect the semantics of DML (unless by accident.) The syntax for pragmas are as follows:

```
/*% tag ... %*/
```

Where *tag* specifies the pragma used, and which determines the syntax of everything following it before the pragma is closed. Tags are case insensitive, but are fully capitilized by convention. DMLC will print a warning if a pragma is given with a tag that the compiler does not recognize.

A pragma may be given anywhere an inline comment may; however, the meaning of a pragma is dependent on its placement, and a specified pragma can be completely meaningless if not properly placed.

DMLC supports the following pragmas:

3.5.1 COVERITY pragma

The `COVERITY` pragma provides a means to manually suppress defects reported by Synopsys® Coverity® stemming from a particular DML line. A `COVERITY` pragma has no effect unless `--coverity` is passed to DMLC, in which case it will cause an analysis annotation to be specified for every generated C line corresponding to the DML line that the pragma applies to.

The syntax for the `COVERITY` pragma is as follows:

```
/*% COVERITY event classification %*/
```

where *`classification`* is optional. This corresponds to the following analysis annotation in generated C:

```
/* coverity[event : classification] */
```

or, if *`classification`* is omitted:

```
/* coverity[event] */
```

A DML line will be affected by every `COVERITY` pragma specified in preceding lines, up until the first line not containing any `COVERITY` pragma. For example:

```
/*% COVERITY unreachable %*/

/*% COVERITY var_deref_model %*/
/*% COVERITY check_return %*/ /*% COVERITY copy_paste_error FALSE %*/
some_function(...);
```

Any C line corresponding to the call to `some_function(...)` will receive analysis annotations for `var_deref_model`, `check_return`, and `copy_paste_error` (with `copy_paste_error` specifically being classified as a false positive), but not any analysis annotation for `unreachable`, as the empty line breaks the consecutive specifications of COVERITY pragmas.

3.6 The Object Model

DML is structured around an *object model*, where each DML model describes a single *device object*, which can contain a number of *member objects*. Each member object can in its turn have a number of members of its own, and so on in a nested fashion.

Every object (including the device itself) can also have *methods*, which implement the functionality of the object, and *parameters*, which are members that describe static properties of the object.

Each object is of a certain *object type*, e.g., `bank` or `register`. There is no way of adding user-defined object types in DML; however, each object is in general locally modified by adding (or overriding) members, methods and parameters - this can be viewed as creating a local one-shot subtype for each object.

A DML model can only be instantiated as a whole: Individual objects can not be instantiated standalone; instead, the whole hierarchy of objects is instantiated atomically together with the model. This way, it is safe for sibling objects in the hierarchy to assume each other's existence, and any method can freely access state from any part of the object hierarchy.

Another unit of instantiation in DML is the *template*. A template contains a reusable block of code that can be instantiated in an object, which can be understood as expanding the template's code into the object.

Many parts of the DML object model are automatically mapped onto the Simics *configuration object* model; most importantly, the device object maps to a Simics configuration class, such that configuration objects of that class correspond to instances of the DML model, and the attribute and interface objects of the DML model map to Simics attributes and interfaces associated with the created Simics configuration class (See *Simics Model Builder User's Guide* for details.)

3.6.1 Device Structure

A device is made up of a number of member objects and methods, where any object may contain further objects and methods of its own. Many object types only make sense in a particular context and are not allowed elsewhere:

- There is exactly one `device` object. It resides on the top level.
- Objects of type `bank`, `port` or `subdevice` may only appear as part of a `device` or `subdevice` object.
- Objects of type `implement` may only appear as part of a `device`, `port`, `bank`, or `subdevice` object.
- Objects of type `register` may only appear as part of a `bank`.
- Objects of type `field` may only appear as part of a `register`.
- Objects of type `connect` may only appear as part of a `device`, `subdevice`, `bank`, or `port` object.
- Objects of type `interface` may only appear directly below a `connect` object.
- Objects of type `attribute` may only appear as part of a `device`, `bank`, `port`, `subdevice`, or `implement` object.
- Objects of type `event` may appear anywhere **except** as part of a `field`, `interface`, `implement`, or another `event`.
- Objects of type `group` are neutral: Any object may contain a `group` object, and a `group` object may contain any object that its parent object may contain, with the exception that a `group` cannot contain an object of type `interface` or `implement`.

3.6.2 Parameters

Parameters (shortened as "`param`") are a kind of object members that *describe expressions*. During compilation, any parameter reference will be expanded to the definition of that parameter. In this sense, parameters are similar to macros, and indeed have some usage patterns in common - in particular, parameters are typically used to represent constant expressions.

Like macros, no type declarations are necessary for parameters, and every usage of a parameter will re-evaluate the expression. Unlike macros, any parameter definition must be a syntactically valid expression, and every unfolded parameter expression is always evaluated using the scope in which the parameter was defined, rather than the scope in which the parameter is referenced.

Parameters cannot be dynamically updated at run-time; however, a parameter can be declared to allow it being overridden by later definitions - see Parameter Declarations.

Within DML's built-in modules and standard library, parameters are used to describe static properties of objects, such as names, sizes, and offsets. Many of these are overridable, allowing some properties to be configured by users. For example, every bank object has a `byte_order` parameter that controls the byte order of registers within that bank. By default, this parameter is defined to be `"little-endian"` - but by overriding it, users may specify the byte order on a bank-by-bank basis.

3.6.3 Methods

Methods are object members providing implementation of the functionality of the object. Although similar to C functions, DML methods can have any number of input parameters and return values. DML methods also support a basic exception handling mechanism using `throw` and `try`.

In-detail description of method declarations are covered in a separate section.

3.6.4 The Device

The *device* defined by a DML model corresponds directly to a Simics *configuration object*, i.e., something that can be included in a Simics configuration.

In DML's object hierarchy, the device object is represented by the top-level scope.

The DML file passed to the DML compiler must *start* with a `device` declaration following the language version specification:

```
dml 1.4;
device name;
```

A `device` declaration may not appear anywhere else, neither in the main file or in imported files. Thus, the device declaration is limited to two purposes:

- to give a *name* to the configuration class registered with Simics
- to declare which DML file is the top-level file in a DML model

3.6.5 Register Banks

A *register bank* (or simply *bank*) is an abstraction that is used to group *registers* in DML, and to expose these to the outside world. Registers are exposed to the rest of the simulated system through the Simics interface `io_memory`, and exposed to scripting and user interfaces through the `register_view`, `register_view_read_only`, `register_view_catalog` and `bank_instrumentation_subscribe` Simics interfaces.

It is possible to define *bank arrays* to model a row of similar banks. Each element in the bank array is a separate configuration object in Simics, and can thus be individually mapped in a memory space.

Simics configuration objects for bank instances are named like the bank but with a `.bank` prefix. For instance, if a device model has a declaration `bank regs[i < 2]` on top level, and a device instance is named `dev` in Simics, then the two banks are represented in Simics by configuration objects named `dev.bank.regs[0]` and `dev.bank.regs[1]`.

3.6.6 Registers

A *register* is an object that contains an integer value. Normally, a register corresponds to a segment of consecutive locations in the address space of the bank; however, it is also possible (and often useful) to have registers that are not mapped to any address within the bank. All registers must be part of a register bank.

Every register has a fixed *size*, which is an integral, nonzero number of 8-bit bytes. A single register cannot be wider than 8 bytes. The size of the register is given by the `size` parameter, which can be specified either by a normal parameter assignment, as in

```
register r1 {
    param size = 4;
    ...
}
```

or, more commonly, using the following short-hand syntax:

```
register r1 size 4 {
    ...
}
```

which has the same meaning. The default size is provided by the `register_size` parameter of the containing register bank, if that is defined.

There are multiple ways to manipulate the value of a register: the simplest approach is to make use of the `val` member of registers, as in:

```
log info: "the value of register r1 is %d", r1.val;
```

or

```
++r1.val;
```

For more information, see Section Register Objects.

3.6.6.1 Mapping Addresses To Registers

For a register to be mapped into the internal address space of the containing bank, its starting address within the bank must be given by setting the `offset` parameter. The address range occupied by the register is then from `offset` to `offset` + `size` - 1. The offset can be specified by a normal parameter assignment, as in

```
register r1 {
    param offset = 0x0100;
    ...
}
```

or using the following short-hand syntax:

```
register r1 @ 0x0100 {
    ...
}
```

similar to the `size` parameter above. Usually, a normal read/write register does not need any additional specifications apart from the size and offset, and can simply be written like this:

```
register r1 size 4 @ 0x0100;
```

or, if the bank contains several registers of the same size:

```
bank b1 {
    param register_size = 4;
    register r1 @ 0x0100;
    register r2 @ 0x0104;
    ...
}
```

The translation from the bank address space to the actual value of the register is controlled by the `byte_order` parameter. When it is set to `"little-endian"` (the default), the lowest address, i.e., that defined by `offset`, corresponds to the least significant byte in the register, and when set to `"big-endian"`, the lowest address corresponds to the most significant byte in the register.

3.6.6.2 Not Mapping Addresses To Registers

An important thing to note is that registers do not have to be mapped at all. This may be useful for internal registers that are not directly accessible from software. By using an unmapped register, you can get the advantages of using register, such as automatic checkpointing and register fields. This internal register can then be used from the implementations of other registers, or other parts of the model.

Historically, unmapped registers were commonly used to store simple device state, but this usage is no longer recommended — Saved Variables should be preferred if possible. Unmapped registers should only be used if saved variables do not fit a particular use case.

To make a register unmapped, set the offset to `unmapped_offset` or use the standard template `unmapped`:

```
register r is (unmapped);
```

3.6.6.3 Register Attributes

For every register, an attribute of integer type is automatically added to the Simics configuration class generated from the device model. The name of the attribute corresponds to the hierarchy in the DML model; e.g., a register named `r1` in a bank named `bank0` will get a corresponding attribute named `bank0_r1`.

The register value is automatically saved when Simics creates a checkpoint, unless the `configuration` parameter indicates otherwise.

The value of a register is stored in a member named `val`. E.g., the `r1` register will store its value in `r1.val`. This is normally the value that is saved in checkpoints; however, checkpointing is defined by the `get` and `set` methods, so if they are overridden, then some other value can be saved instead.

3.6.6.4 Fields

Real hardware registers often have a number of *fields* with separate meaning. For example, the lowest three bits of the register could be a status code, the next six bits could be a set of flags, and the rest of the bits could be reserved.

To make this easy to express, a `register` object can contain a number of `field` objects. Each field is defined to correspond to a bit range of the containing register.

The value of a field is stored in the corresponding bits of the containing register's storage. The easiest way to access the value of a register or field is to use the `get` and `set` methods.

The read and write behaviour of registers and fields is in most cases controlled by instantiating *templates*. There are three categories of templates:

- Registers and fields where a read or write just updates the value with no side-effects, should use the `read` and `write` templates, respectively.
- Custom behaviour can be supplied by instantiating the `read` or `write` template. The template leaves a simple method `read` (or `write`) abstract; custom behaviour is provided by overriding the method. There is also a pair of templates `read_field` and `write_field`, which similarly provide abstract methods `read_field` and `write_field`. These functions have some extra parameters, making them less convenient to use, but they also offer some extra information about the access.
- There are many pre-defined templates with for common specialized behaviour. The most common ones are `unimpl`, for registers or fields whose behaviour has not yet been implemented, and `read_only` for registers or fields that cannot be written.

A register or field can often instantiate two templates, one for reads and one for writes; e.g., `read` to supply a read method manually, and `read_only` to supply a standard write method. If a register with fields instantiates a read or write template, then the register will use that behaviour *instead* of descending into fields. For instance, if a register instantiates the `read_only` template, then all writes will be captured, and only reads will descend into its fields.

The register described above could be modeled as follows, using the default little-endian bit numbering.

```
bank b2 {
    register r0 size 2 @ 0x0000 {
        field status @ [2:0];
        field flags @ [8:3];
        field reserved @ [15:9];
    }
    ...
}
```

Note that the most significant bit number is always the first number (to the left of the colon) in the range, regardless of whether little-endian or big-endian bit numbering is used. (The bit numbering convention used in a source file can be selected by a `bitorder` declaration.)

The value of the field can be accessed by using the `get` and `set` methods, e.g.:

```
log info: "the value of the status field is %d", r0.status.get();
```

3.6.7 Attributes

An `attribute` object in DML represents a Simics configuration object attribute of the device. As mentioned above, Simics attributes are created automatically for `register` and `connect` objects to allow external inspection and modification; explicit `attribute` objects can be used to expose additional data. There are mainly three use cases for explicit attributes:

- Exposing a parameter for the end-user to configure or tweak. Such attributes can often be *required* in order to instantiate a device, and they usually come with documentation.

- Exposing internal device state, required for checkpointing to work correctly. Most device state is usually saved in registers or saved variables, but attributes may sometimes be needed to save non-trivial state such as FIFOs.
- Attributes can also be created as synthetic back-doors for additional control or inspection of the device. Such attributes are called *pseudo attributes*, and are not saved in checkpoints.

An attribute is basically a name with an associated pair of `get` and `set` functions. The type of the value read and written through the get/set functions is controlled by the `type` parameter. More information about configuration object attributes can be found in *Simics Model Builder User's Guide*.

The `init` template and associated method is often useful together with `attribute` objects to initialize any associated state.

Four standard templates are provided for attributes: `bool_attr`, `int64_attr`, `uint64_attr` and `double_attr`. They provide overridable `get` and `set` methods, and store the attribute's value in a session variable named `val`, using the corresponding type. For example, if `int64_attr` is used in the attribute `a`, then one can access it as follows:

```
log info: "the value of attribute a is %d", dev.a.val;
```

These templates also provide an overridable implementation of `init()` that initializes the `val` session variable. The value that `val` is initialized to is controlled by the `init_val` parameter, whose default definition simply causes `val` to be zero-initialized.

Defining `init_val` is typically the most convenient way of initializing any attribute instantiating any one of the these templates — however, overriding the default `init()` implementation with a custom one may still be desirable in certain cases. In particular, the definition of `init_val` must be constant, so a custom `init()` implementation is necessary if `val` should be initialized to a non-constant value.

Note that using an attribute object purely to store and checkpoint simple internal device state is not recommended; prefer Saved Variables for such use cases.

3.6.8 Connects

A `connect` object is a container for a reference to an arbitrary Simics configuration object. An attribute with the same name as the connect is added to the Simics configuration class generated from the device model. This attribute can be assigned a value of type "Simics object".

A `connect` declaration is very similar to a simple `attribute` declaration, but specialized to handle connections to other objects.

Typically, the connected object is expected to implement one or more particular Simics interfaces, such as `signal` or `ethernet_common` (see *Simics Model Builder User's Guide* for details). This is described using `interface` declarations inside the `connect`.

Initialization of the connect (i.e., setting the object reference) is done from outside the device, usually in a Simics configuration file. Just like other attributes, the parameter `configuration` controls whether the value must be initialized when the object is created, and whether it is automatically saved when a checkpoint is created.

The actual object pointer, which is of type `conf_object_t*` is stored in a `session` member called `obj`. This means that to access the current object pointer in a connect called *otherdev*, you need to write `otherdev.obj`.

If the `configuration` parameter is not `required`, the object pointer may have a null value, so any code that tries to use it must check if it is set first.

This is an example of how a connect can be declared and used:

```
connect plugin {
    param configuration = "optional";
}

method mymethod() {
    if (plugin.obj)
        log info: "The plugin is connected";
    else
        log info: "The plugin is not connected";
}
```

3.6.8.1 Interfaces

In order to use the Simics interfaces that a connected object implements, they must be declared within the `connect`. This is done through `interface` objects. These name the expected interfaces and may also specify additional properties.

An important property of an interface object is whether or not a connected object is *required* to implement the interface. This can be controlled through the interface parameter `required`, which is `true` by default. Attempting to connect an object that does not implement the required interfaces will cause a runtime error. The presence of optional interfaces can be verified by testing if the `val` member of the interface object has a null value.

By default, the C type of the Simics interface corresponding to a particular interface object is assumed to be the name of the object itself with the string `"_interface_t"` appended. (The C type is typically a `typedef`:ed name for a struct containing function pointers).

The following is an example of a connect with two interfaces, one of which is not required:

```
connect plugin {
    interface serial_device;
    interface rs232_device { param required = false; }
}
```

Calling interface functions is done in the same way as any C function is called, but the first argument which is the target object pointer is omitted.

The `serial_device` used above has a function with the following definition:

```
int (*write)(conf_object_t *obj, int value);
```

This interface function is called like this in DML:

```
method call_write(int value) {
    local int n = plugin.serial_device.write(value);
    // code to check the return value omitted
}
```

3.6.9 Implements

When a device needs to export a Simics interface, this is specified by an `implement` object, containing the methods that implement the interface. The name of the object is also used as the name of the Simics interface registered for the generated device, and the names and signatures of the methods must correspond to the C functions of the Simics interface. (A device object pointer is automatically added as the first parameter of the generated C functions.)

In most cases, a device exposes interfaces by adding `implement` object as subobjects of named `port` objects. A port object often represents a hardware connection

The C type of the Simics interface is assumed to be the value of the object's `name` parameter (which defaults to the name of the object itself), with the string `"_interface_t"` appended. The C type is typically a `typedef`:ed name for a struct containing function pointers.

For example, to implement the `ethernet_common` Simics interface, we can write:

```
implement ethernet_common {
    method frame(const frags_t *frame, eth_frame_crc_status_t crc_status) {
        ...
    }
}
```

This requires that `ethernet_common_interface_t` is defined as a struct type with a field `frame` with the function pointer type `void (*)(conf_object_t *, const frags_t *, eth_frame_crc_status_t)`.

Definitions of all standard Simics interface types are available as DML files named like the corresponding C header files; for instance, the `ethernet_common` interface can be imported as follows:

```
import "simics/devs/ethernet.dml"
```

3.6.10 Events

An *event* object is an encapsulation of a Simics event that can be posted on a processor time or step queue. The location of event objects in the object hierarchy of the device is not important, so an event object can generally be placed wherever it is most convenient.

An event has a built-in `post` method, which inserts the event in the default queue associated with the device. An event also defines an abstract method `event`, which the user must implement. That method is called when the event is triggered.

An event must instantiate one of six predefined templates: `simple_time_event`, `simple_cycle_event`, `uint64_time_event`, `uint64_cycle_event`, `custom_time_event` or `custom_cycle_event`. The choice of template affects the signature of the `post` and `event` methods: In a time event, the delay is specified as a floating-point value, denoting number of seconds, while in a cycle event, the delay is specified in CPU cycles.

A posted event may have data associated with it. This data is given to the `post` method and is provided to the `event` callback method. They type of data depends on the template used: No data is provided in simple events, and in uint64 events it is provided as a uint64 parameter. In custom events, data is provided as a `void *` parameter, and extra methods `get_event_info` `set_event_info` and `destroy` must be provided in order to provide proper checkpointing of the event.

3.6.11 Groups

Objects of type `attribute`, `connect`, `event`, `field`, `register`, `bank`, `port` and `subdevice` can be organized into *groups*. A group is a neutral object, which can be used just for namespacing, or to help structuring an array of a collection of objects. Groups may appear anywhere, but are most commonly used to group registers: If a bank has a sequence of blocks, each containing the same registers, it can be written as a group array. In the following example eight homogeneous groups of registers are created, resulting in 8×6 instances of register `r3`.

```
bank regs {
    param register_size = 4;
    group blocks[i < 8] {
        register r1 @ i * 32 + 0;
        register r2 @ i * 32 + 4;
        register r3[j < 6] @ i * 32 + 8 + j * 4;
    }
}
```

Another typical use of `group` is in combination with a template for the group that contains common registers and more that are shared between several groups, as in the following example.

```
template weird {
    param group_offset;
    register a size 4 @ group_offset is (read, write);
    register b size 4 @ group_offset + 4 is (read, write) {
        method read() -> (uint64) {
            // When register b is read, return a
            return a.val;
        }
    }
}

bank regs {
    group block_a is (weird) { param group_offset = 128; }
    group block_b is (weird) { param group_offset = 1024; }
}
```

In addition, groups can be nested.

```
bank regs {
    param register_size = 4;
    group blocks[i < 8] {
        register r1 @ i * 52 + 0;
        group sub_blocks[j < 4] {
            register r2 @ i * 52 + j * 12 + 4;
            register r3[k < 3] @ i * 52 + j * 12 + k * 4 + 8;
        }
    }
}
```

Banks, ports and subdevices can be placed inside groups; in this case, the Simics configuration object that represents the bank, port or subdevice will be placed under a namespace object; for instance, if a device with `group g { bank regs; }` is instantiated as `dev`, then the bank is represented by an object `dev.g.bank.regs`, where `g` and `bank` are both `namespace` objects.

As groups have no special properties or restrictions, they can be used as a tool for building abstractions — in particular in combination with templates.

For example, a template can be used to create an abstraction for finite state machine objects, by letting users create FSMs by declaring group objects instantiating that template. FSM states can also be represented through a template instantiated by groups.

The following demonstrates a simple example of how such an abstraction may be implemented:

```
// Template for finite state machines
template fsm is init {
    saved fsm_state curr_state;

    // The initial FSM state.
    // Must be defined by any object instantiating this template.
    param init_fsm_state : fsm_state;

    shared method init() default {
        curr_state = init_fsm_state;
    }

    // Execute the action associated to the current state
    shared method action() {
        curr_state.action();
    }
}

// Template for states of an FSM. Such states must be sub-objects
// of an FSM.
template fsm_state {
    param parent_fsm : fsm;
    param parent_fsm = cast(parent, fsm);

    // The action associated to this state
    shared method action();

    // Transitions the parent FSM to this state
    shared method set() {
        parent_fsm.curr_state = cast(this, fsm_state);
    }
}
```

These templates can then be used as follows:

```
group main_fsm is fsm {
    param init_fsm_state = cast(init_state, fsm_state);

    group init_state is fsm_state {
        method action() {
          log info: "init_state -> second_state";
          // Transition to second_state
          second_state.set();
        }
    }

    group second_state is fsm_state {
        method action() {
          log info: "second_state -> final_state";
          // Transition to final_state
          final_state.set();
        }
    }

    group final_state is fsm_state {
        method action() {
            log info: "in final_state";
        }
    }
}

method trigger_fsm() {
    // Execute the action of main_fsm's current state.
    main_fsm.action();
}
```

3.6.12 Ports

An interface port is a structural element that groups implementations of one or more interfaces. When one configuration object connects to another, it can connect to one of the target object's ports, using the interfaces in the port. This way, the device model can expose different interfaces to different objects.

Sometimes a port is as simple as a single pin, implementing the `signal` interface, and sometimes it can be more complex, implementing high-level communication interfaces.

It is also possible to define port arrays that are indexed with an integer parameter, to model a row of similar connectors.

In Simics, a port is represented by a separate configuration object, named like the port but with a `.port` prefix. For instance, if a device model has a declaration `port p[i<2]` on top level, and a device instance is named `dev` in Simics, then the two ports are represented in Simics by objects named `dev.port.p[0]` and `dev.port.p[1]`.

3.6.13 Subdevices

A subdevice is a structural element that represents a distinct subsystem of the device. Like a `group`, a subdevice can be used to group a set of related banks, ports and attributes, but a

subdevice is presented to the end-user as a separate configuration object. If a subdevice contains `attribute` or `connect` objects, or `saved` declarations, then the corresponding configuration attributes appears as members of the subdevice object rather than the device.

3.7 Templates

```
template name { ... }
```

Defines a *template*, a piece of code that can be reused in multiple locations. The body of the template contains a number of declarations that will be added to any object that uses the template.

Templates are imported into an object declaration body using `is` statements, written as

```
is name;
```

for example:

```
field F {
    is A;
}
```

It is also possible to use templates when declaring an object, as in

```
field F is (name1, name2);
```

These can be used in any context where an object declaration may be written, and has the effect of expanding the body of the template at the point of the `is`. Using `is` together with object declarations is typically more idiomatic than the standalone `is` object statement; however, the latter is useful in order to instantiate templates in the top-level device object, and also for use in conjunction with `in each` declarations; for example:

```
register r {
    in each field {
        is A;
    }

    field F1 @ [7:6];
    ...
}
```

If two templates define methods or parameters with the same name, then the template instantiation hierarchy is used to deduce which method overrides the other: If one template *B* instantiates another template *A*, directly or indirectly, then methods from *B* override methods from *A*. Note, however, that overrides can only happen on methods and parameters that are declared `default`. Example:

```
template A {
    method hello() default {
        log info: "hello";
    }
}
template B is A {
    // this method overrides the
    // method from A
    method hello() default {
        default();
        log info "world";
    }
}
```

See Resolution of Overrides for a formal specification of override rules.

3.7.1 Templates as types

Each template defines a *type*, which is similar to a class in an object oriented language like Java. The type allows you to store references to a DML object in a variable. Some, but not all, top-level declarations inside a template appear as members of the template type. A template type has the following members:

- All session and saved variables declared within the template. E.g., the declaration `session int val;` gives a type member `val`.
- All declarations of typed parameters, further discussed below. E.g., the declaration `param foo : uint64;` gives a type member `foo`.
- All method declarations declared with the `shared` keyword, further discussed below. E.g., the declaration `shared method fun() { ... }` gives a type member `fun`, which can be called.
- Every `shared` hook declared within the template. E.g. the declaration `shared hook(int, bool) h;` gives a type member `h`.
- All type members of inherited templates. E.g., the declaration `is simple_time_event;` adds two type members `post` and `next`, since `post` and `next` are members of the `simple_time_event` template type.
- The `templates` member, which permits template-qualified method implementation calls to the `shared` method implementations of the template type's ancestor templates.

Template members are dereferenced using the `.` operator, much like struct members.

A template's type is named like the template, and an object reference can be converted to a value using the `cast` operator. For instance, a reference to the register `regs.r0` can be created and used as follows (all register objects automatically implement the template `register`):

```
local register x = cast(regs.r0, register);
x.val = 14;  // sets regs.r0.val
```

Two values of the same template type can be compared for equality, and are considered equal when they both reference the same object.

A value of a template type can be upcast to an ancestor template type; for example:

```
local uint64_attr x = cast(attr, uint64_attr);
local attribute y = cast(x, attribute);
```

In addition, a value of any template type can be cast to the template type `object`, even if `object` is not an ancestor of the template.

3.7.2 Shared methods

If a method is declared in a template, then one copy of the method will appear in each object where the template is instantiated; therefore, the method can access all methods and parameters of that object. This is often convenient, but comes with a cost; in particular, if a template is instantiated in many objects, then this gives unnecessarily large code size and slow compilation. To address this problem, a method can be declared *shared* to operate on the template type rather than the implementing object. The implementation of a shared method is compiled once and shared between all instances of the template, rather than duplicated between instances.

Declaring a method as shared imposes restrictions on its implementation, in particular which symbols it is permitted to access: Apart from symbols in the global scope, a shared method may only access members of the template's type; it is an error to access any other symbols defined by the template. Members can be referenced directly by name, or as fields of the automatic `this` variable. When accessed in the scope of the shared method's body, the `this` variable evaluates to a value whose type is the template's type.

Example:

```
template base {
    // abstract method: must be instantiated in sub-template or object
    shared method m(int i) -> (int);
    shared method n() -> (int) default { return 5; }
}
template sub is base {
    // override
    shared method m(int i) -> (int) default {
        return i + this.n();
    }
}
```

If code duplication is not a concern, it is possible to define a shared method whose implementation is not subject to above restrictions while still retaining the benefit of having the method be a member of the template type. This is done by defining the implementation separately from the declaration of the shared method, for example:

```
template get_qname {
    shared method get_qname() -> (const char *);
    method get_qname() -> (const char *) {
        // qname is an untyped parameter, and would thus not be accessible
        // within a shared implementation of get_qname()
        return this.qname;
    }
}
```

3.7.3 Typed Parameters

A *typed parameter declaration* is a parameter declaration form that may only appear within templates:

```
param name : type;
```

A typed parameter declaration adds a member to the template type with the same name as the specified parameter, and with the specified type. That member is associated with the specified parameter, in the sense that the definition of the parameter is used as the value of the template type member.

A typed parameter declaration places a number of requirements on the named parameter:

- The named parameter must be defined (through a regular parameter declaration). This can be done either within the template itself, within sub-templates, or within individual objects instantiating the template.
- The parameter definition must be a valid expression of the specified type.
- The parameter definition must be free of side-effects, and must not rely on the specific device instance of the DML model — in particular, the definition must be independent of device state.

  This essentially means that the definition must be a constant expression, except that it may also make use of device-independent expressions whose values are known to be constant. For example, index parameters, each-in expressions, and object references cast to template types are allowed. It is also allowed to reference other parameters that obey this rule.

  Examples of expressions that may *not* be used include method calls and references to `session`/`saved` variables.
- The parameter definition must not contain calls to independent methods.

Typed parameters are most often used to allow a shared method defined within the template to access parameters of the template. For example:

```
template max_val_reg is write {
    param max_val : uint64;

    shared method write(uint64 v) {
        if (v > max_val) {
            log info: "Ignoring write to register exceeding max value %u",
                      max_val;
        } else {
            default(v);
        }
    }
}

bank regs {
    register reg[i < 2] size 8 @0x0 + i*8 is max_val_reg {
        param max_val = 128 * (i + 1) - 1;
    }
}
```

3.8 Parameter Declarations

A parameter declaration has the general form "`param` *`name specification`*`;`", where *`specification`* is either "`=` *`expr`*" or "`default` *`expr`*". For example:

```
param offset = 8;
param byte_order default "little-endian";
```

A default value is overridden by an assignment (`=`). There can be at most one assignment for each parameter. Typically, a default value for a parameter is specified in a template, and the programmer may then choose to override it where the template is used.

The *`specification`* part is in fact optional; if omitted, it means that the parameter is declared to exist (and *must* be given a value, or the model will not compile). This is sometimes useful in templates, as in:

```
template constant is register {
    param value;
    method get() -> (uint64) {
        return value;
    }
}
```

so that wherever the template `constant` is used, the programmer is also forced to define the parameter `value`. E.g.:

```
register r0 size 2 @ 0x0000 is (constant) {
    param value = 0xffff;
}
```

Note that simply leaving out the parameter declaration from the template definition can have unwanted effects if the programmer forgets to specify its value where the template is used. At best, it will only cause a more obscure error message, such as "unknown identifier"; at worst, the scoping rules will select an unrelated definition of the same parameter name.

Also note that a parameter declaration without definition is redundant if a typed parameter declaration for that parameter already exists, as that already enforces that the parameter must be defined.

> **Note**
>
> You may see the following special form in some standard library files:
>
> ```
> param name auto;
> ```
>
> for example,
>
> ```
> param parent auto;
> ```

> This is used to explicitly declare the built-in automatic parameters, and should never be used outside the libraries.

3.9 Data types

The type system in DML builds on the type system in C, with a few modifications. There are eight kinds of data types. New names for types can also be assigned using a `typedef` declaration.

Integers

Integer types guarantee a certain *minimum* bit width and may be signed or unsigned. The basic integer types are named `uint1`, `uint2`, ..., `uint64` for the unsigned types, and `int1`, `int2`, ..., `int64` for the signed types. Note that the size of the integer type is only a hint and the type is guaranteed to be able to hold at least that many bits. Assigning a value that would not fit into the type is undefined, thus it is an error to assume that values will be truncated. For bit-exact types, refer to `bitfields` and `layout`.

The familiar integer types `char` and `int` are available as aliases for `int8` and `int32`, respectively. The C keywords `short`, `signed` and `unsigned` are reserved words in DML and not allowed in type declarations.

The types `size_t` and `uintptr_t`, `long`, `uint64_t`, `int64_t`, are defined as in C. The types `long`, `uint64_t` and `int64_t` are provided mainly for compatibility with third party libraries; they are needed because they are incompatible with the corresponding Simics types (`uint64`, etc) on some platforms.

Endian integers

Endian integer types hold similar values as integer types, but in addition have the following attributes:

- *They are guaranteed to be stored in the exact number of bytes required for their bitsize, without padding.*
- *They have a defined byte order.*
- *They have a natural alignment of 1 byte.*

Endian integer types are named after the integer type with which they share a bitsize and sign but in addition have a `_be_t` or `_le_t` suffix, for big-endian and little-endian integers, respectively. The full list of endian types is:

```
int8_be_t    int8_le_t    uint8_be_t    uint8_le_t
int16_be_t   int16_le_t   uint16_be_t   uint16_le_t
int24_be_t   int24_le_t   uint24_be_t   uint24_le_t
int32_be_t   int32_le_t   uint32_be_t   uint32_le_t
int40_be_t   int40_le_t   uint40_be_t   uint40_le_t
int48_be_t   int48_le_t   uint48_be_t   uint48_le_t
int56_be_t   int56_le_t   uint56_be_t   uint56_le_t
int64_be_t   int64_le_t   uint64_be_t   uint64_le_t
```

These types can be transparently used interchangeably with regular integer types, values of one type will be coerced to the other as needed. Note that operations on integers will always produce regular integer types, even if all operands are of endian integer type.

Floating-point numbers

There is only one floating-point type, called `double`. It corresponds to the C type `double`.

Booleans

The boolean type `bool` has two values, `true` and `false`.

Arrays

An array is a sequence of elements of another type, and works as in C.

Pointers

Pointers to types, work as in C. String literals have the type `const char *`. A pointer has undefined meaning if the pointer target type is an integer whose bit-width is neither 8, 16, 32, nor 64.

Structures

A `struct` type defines a composite type that contains named members of different types. DML makes no assumptions about the data layout in struct types, but see the layout types below for that. Note that there is no struct label as in C, and struct member declarations are permitted to refer to types that are defined further down in the file. Thus, new struct types can always be declared using the following syntax:

```
typedef struct { member declarations } name;
```

Layouts

A layout is similar to a struct in many ways. The important difference is that there is a well-defined mapping between a layout object and the underlying memory representation, and layouts may specify that in great detail.

A basic layout type looks like this:

```
layout "big-endian" {
    uint24 x;
    uint16 y;
    uint32 z;
}
```

By casting a pointer to a piece of host memory to a pointer of this layout type, you can access the fourth and fifth byte as a 16-bit unsigned integer with big-endian byte order by simply writing `p->y`.

The allowed types of layout members in a layout type declaration are integers, endian integers, other layout types, bitfields (see below), and arrays of these.

The byte order declaration is mandatory, and is either `"big-endian"` or `"little-endian"`.

Access of layout members do not always provide a value of the type used for the member in the declaration. Bitfields and integer members (and arrays of similar) are translated to endian integers (or arrays of such) of similar size, with endianness matching the layout. Layout and endian integer members are accessed normally.

Bitfields

A bitfield type works similar to an integer type where you use bit slicing to access individual bits, but where the bit ranges are assigned names. A `bitfields` declaration looks like this:

```
bitfields 32 {
    uint3  a @ [31:29];
    uint16 b @ [23:8];
    uint7  c @ [7:1];
    uint1  d @ [0];
}
```

The bit numbering is determined by the `bitorder` declaration in the current file.

Accessing bit fields is done as with a struct or layout, but the whole bitfield can also be used as an unsigned integer. See the following example:

```
local bitfields 32 { uint8 x @ [7:0] } bf;
bf = 0x000000ff;
bf.x = bf.x - 1;
local uint32 v = bf;
```

3.9.1 Serializable types

Serializable types are types that the DML compiler knows how to serialize and deserialize for the purposes of checkpointing. This is important for the use of saved variables and the `after` statement.

All primitive non-pointer data types (integers, floating-point types, booleans, etc.) are considered serializable, as is any struct, layout, or array type consisting entirely of serializable types. Template types and hook reference types are also considered serializable.

Any type not fitting the above criteria is not considered serializable: in particular, any pointer type is not considered serializable, nor is any `extern` struct type; the latter is because it's impossible for the compiler to ensure it's aware of all members of the struct type.

3.10 Methods

Methods are similar to C functions, but also have an implicit (invisible) parameter which allows them to refer to the current device instance, i.e., the Simics configuration object representing the device. Methods also support exception handling in DML, using `try` and `throw`. The body of the method is a compound statement in an extended subset of C. It is an error to have more than one method declaration using the same name within the same scope.

3.10.1 Input Parameters and Return Values

A DML method can have any number of return values, in contrast to C functions which have at most one return value. DML methods do not use the keyword `void` — an empty pair of parentheses always means "zero parameters". Furthermore, lack of return value can even be omitted. Apart from this, the parameter declarations of a method are ordinary C-style declarations.

For example,

```
method m1() -> () {...}
```

and

```
method m1() {...}
```

are equivalent, and define a method that takes no input parameters and returns nothing.

```
method m2(int a) -> () {...}
```

defines a method that takes a single input parameter, and also returns nothing.

```
method m3(int a, int b) -> (int) {
    return a + b;
}
```

defines a method with two input parameters and a single return value. A method that has a return value must end with a return statement.

```
method m4() -> (int, int) {
    ...;
    return (x, y);
}
```

has no input parameters, but two return values.

A method that can throw an exception must declare so, using the `throws` keyword:

```
method m5(int x) -> (int) throws {
    if (x < 0)
        throw;
    return x * x;
}
```

3.10.2 Default Methods

A parameter or method can now be overridden more than once.

When there are multiple declarations of a parameter, then the template and import hierarchy are used to deduce which declaration to use: A declaration that appears in a block that instantiates a template will override any declaration in that template, and a declaration that appears in a file that imports another file will override any declaration from that file. The declarations of one parameter must appear so that one declaration overrides all other declarations of he parameter; otherwise the declaration is considered ambiguous and an error is signalled.

Examples: A file `common.dml` might contain:

```
param num_banks default 2;
bank banks[num_banks] {
    ...
}
```

Your device `my-dev.dml` can then contain:

```
device my_dev;
import "common.dml";
// overrides the declaration in common.dml
param num_banks = 4;
```

The assignment in `my-dev.dml` takes precedence, because `my-dev.dml` imports `common.dml`.

Another example: The following example gives an compile error:

```
template my_read_constant {
    param value default 0;
    ...
}
template my_write_constant {
    param value default 0;
    ...
}
bank b {
    // ERROR: Two declarations exist, and neither takes precedence
    register r is (my_read_constant, my_write_constant);
}
```

The conflict can be resolved by declaring the parameter a third time, in a location that overrides both the conflicting declarations:

```
bank b {
    register r is (my_read_constant, my_write_constant) {
        param value default 0;
    }
}
```

Furthermore, an assignment (`=`) of a parameter may not be overridden by another declaration.

If more than one declaration of a method appears in the same object, then the template and import hierarchies are used to deduce the override order. This is done in a similar way to how parameters are handled:

- A method declaration that appears in a block that instantiates a template will override any declaration from that template
- A method declaration that appears in a file that imports another file will override any declaration from that file.
- The declarations of one method must appear so that one declaration overrides all other declarations of the method; otherwise the declaration is considered ambiguous and an error is signalled.
- A method can only be overridden by another method if it is declared `default`.

> **Note**

> An overridable built-in method is defined by a template named as the object type. So, if you want to write a template that overrides the `read` method of a register, and want to make your implementation overridable, then your template must explicitly instantiate the `register` template using a statement `is register;`.

3.10.3 Calling Methods

In DML, a method call looks much like in C, with some exceptions. For instance,

```
(a, b) = access(...);
```

calls the method 'access' in the same object, assigning the return values to variables `a` and `b`.

If one method overrides another, it is possible to refer to the overridden method from within the body of the overriding method using the identifier `default`:

```
x = default(...);
```

In addition to `default`, there exists the `templates` member of objects which allows for calling the particular implementation of a method as provided by a specified template. This is particularly useful when `default` *can't* be used due to the method overriding implementations provided by multiple hierarchically unrelated templates, such that `default` can't be unambiguously resolved (see Resolution of overrides.) Unlike `default`, `templates` can also be used even outside the body of the overriding method.

DML supports *compound initializer syntax* for the arguments of called methods, meaning arguments of struct-like types can be constructed using `{...}`. For example:

```
typedef struct {
    int x;
    int y;
} struct_t;

method copy_struct(struct_t *tgt, struct_t src) {
    *tgt = src
}

method m() {
    local struct_t s;
    copy_struct(&s, {1, 4});
    copy_struct(&s, {.y = 1, .x = 4});
    copy_struct(&s, {.y = 1, ...}); // Partial designated initializer
}
```

This syntax can't be used for variadic arguments or inline arguments.

3.10.4 Inline Methods

Methods can also be defined as inline, meaning that at least one of the input arguments is declared `inline` instead of declaring a type. The method body is re-evaluated every time it is invoked, and when a constant is passed for an inline argument, it will be propagated into the method as a constant.

Inline methods were popular in previous versions of the language, when constant folding across methods was a useful way to reduce the size of the compiled model. DML 1.4 provides better ways to reduce code size, and inline methods remain mainly for compatibility reasons.

3.10.5 Exported Methods

In DML 1.4, methods can be `exported` using the `export` declaration.

3.10.6 Retrieving Function Pointers to Methods

In DML 1.4, method references can be converted to function pointers using `&`.

3.10.7 Independent Methods

Methods that do not rely on the particular instance of the device model may be declared `independent`:

```
independent method m(...) -> (...) {...}
```

Exported independent methods do not have the input parameter corresponding to the device instance, allowing them to be called in greater number of contexts. The body of independent methods may not contain statements or expressions that rely on the device instance in any way; for example, `session` or `saved` variables may not be referenced, `after` and `log` statements may not be used, and non-`independent` methods may not be called.

Within a template, `shared` independent methods may be declared.

When independent methods are used as callbacks, it can sometimes be desirable to mutate device state. In order to do this safely, device state should be mutated within a method not declared `independent`, which can called from independent methods through the use of `&`. Device state should not be mutated directly within an independent method as this could cause certain Simics breakpoints to not function correctly; for example, an independent method should not mutate a session variable through a pointer to that variable.

3.10.7.1 Independent Startup Methods

Independent methods may also be declared `startup`, which causes them to be called when the model is loaded into the simulation, *before* any device is created. In order for this to be possible, `independent startup methods` may not have any return values nor be declared `throw`s. In addition, independent startup methods may not be declared with an overridable definition due to technical limitations — this restriction can be worked around by having an independent startup method call an overridable independent method. Note that abstract `shared` independent startup methods are allowed.

The order in which independent startup methods are implicitly called at model load is not defined, with the exception that independent startup methods not declared memoized are called before any independent startup methods that are.

3.10.7.2 Independent Startup Memoized Methods

Independent startup methods may also be declared `memoized`. Unlike regular `independent startup` methods, `independent startup memoized` methods may — indeed, are required to — have return values and/or be declared `throws`.

After the first call of a memoized method, all subsequent calls for the simulation session return the results of the first call without executing the body of the method. If a memoized method call throws, then subsequent calls will throw without executing the body.

The first call to an independent startup memoized method will typically be the one implicitly performed at model load, but it may also occur beforehand (for example, if the method is called as part of another independent startup method).

Result caching is shared across all instances of the device model. This mechanism can be used to compute device-independent data which is then shared across all instances of the device model.

The results of `shared` memoized methods are cached per template instance, and are not shared across all objects instantiating the template.

(Indirectly) recursive memoized method calls are not allowed; the result of such a call is a run-time critical error.

3.11 Session variables

A *session* declaration creates a number of named storage locations for arbitrary run-time values. The names belongs to the same namespace as objects and methods. The general form is:

```
session declarations = initializer;
```

where `= initializers` is optional and *`declarations`* is a variable declaration similar to C, or a sequence of such declarations; for example,

```
session int id = 1;
session bool active;
session double table[4] = {0.1, 0.2, 0.4, 0.8};
session (int x, int y) = (4, 3);
session conf_object_t *obj;
```

In the absence of explicit initializer expressions, a default "all zero" initializer will be applied to each declared object.

Note that the number of initializers — together given as a tuple — must match the number of declared variables. In addition, the number of elements in each initializer must match with the number of elements or fields of the type of the declared *session* variable. This also implies that each sub-element, if itself being a compound data structure, must also be enclosed in braces.

C99-style designated initializers are supported for `struct`, `layout`, and `bitfields` types:

```
typedef struct { int x; struct { int i; int j; } y; } struct_t;
session struct_t s = { .x = 1, .y = { .i = 2, .j = 3 } }
```

Unlike C, partial initialization is not allowed implicitly; a designated initializer for each member must be specified. However, partial initialization can be done explicitly through the use of trailing `...` syntax:

```
session struct_t s = { .y = { .i = 2, ... }, ... }
```

Also unlike C, designator lists are not supported, and designated initializers for arrays are not supported.

> **Note**

> Previously `session` variables were known as `data` variables.

3.12 Saved variables

A *saved* declaration creates a named storage location for an arbitrary run-time value, and automatically creates an attribute that checkpoints this variable. Saved variables can be declared in object or statement scope, and the name will belong to the namespace of other declarations in that scope. The general form is:

```
saved declaration = initializer;
```

where `= initializer` is optional and *`declaration`* is similar to a C variable declaration; for example,

```
saved int id = 1;
saved bool active;
saved double table[4] = {0.1, 0.2, 0.4, 0.8};
```

In the absence of explicit initializer expression, a default "all zero" initializer will be applied to the declared object.

Note that the number of elements in the initializer must match with the number of elements or fields of the type of the *saved* variable. This also implies that each sub-element, if itself being a compound data structure, must also be enclosed in braces.

C99-style designated initializers are supported for `struct`, `layout`, and `bitfields` types:

```
typedef struct { int x; struct { int i; int j; } y; } struct_t;
saved struct_t s = { .x = 1, .y = { .i = 2, .j = 3 } }
```

Unlike C, partial initialization is not allowed implicitly; a designated initializer for each member must be specified. However, partial initialization can be done explicitly through the use of trailing `...` syntax:

```
session struct_t s = { .y = { .i = 2, ... }, ... }
```

Also unlike C, designator lists are not supported, and designated initializers for arrays are not supported.

In addition, the types of saved declaration variables are currently restricted to primitive data types, or structs or arrays containing only data types that could be saved. Such types are called *serializable*.

> **Note**
>
> Saved variables are primarily intended for making checkpointable state easier. For configuration, `attribute` objects should be used instead. Additional data types for saved declarations are planned to be supported.

3.13 Hook Declarations

```
hook(msgtype1, ... msgtypeN) name;
```

A *hook* declaration defines a named object member to which *suspended computations* may be attached for execution at a later point. By sending a *message* through the hook, every computation suspended on the hook will become detached from the hook, and then executed — receiving the message as data. Computations suspended on a hook are executed in order of least recently attached; in other words, FIFO semantics.

Currently, the only computations that can be suspended and attached to hooks are single method calls, which is done through the use of the `after` statement. This will later be expanded upon: hooks will play a central role in the future introduction of *coroutines*, as hooks will serve as the primitive mechanism through which coroutines suspend themselves and become resumed.

Every hook has an associated list of *message component types*, specified during declaration through the `(msgtype1, ... msgtypeN)` syntax. This specifies what form of data is sent and received via the hook. Any number of message component types can be given, including zero, in which case a message sent via the hook has no associated data.

Example declarations:

```
// Hook with no associated message component types
hook() h1;
// Hook with a single message component type
hook(int) h2;
// Hook with two message component types
hook(int *, bool) h3;
```

Beyond suspending computations on it, a hook *h* has two associated operations:

- ```
  h.send(msg1, ... msgN)
  ```

  Sends a message through the hook, with message components *msg1* through *msgN*. The number of message components must match the number of message component types of the hook, and each message component must be compatible with the corresponding message component type of the hook.

  `send` is *asynchronous*: the message will only be sent — and suspended computations executed — once all current device entries on the call stack have been completed. It is exactly equivalent to `after: h.send_now(msg1, ... msgN)`, except it's not possible to prevent the message from being sent via `cancel_after()`. For more information, see Immediate After Statements.

  Like immediate after statements, pointers to stack-allocated data **must not** be passed as message components to a `send`. If you must use pointers to stack-allocated data, then `send_now` should be used instead of `send`. If you want the message to be delayed to avoid ordering bugs, create a method which wraps the `send_now` call together with the declarations of the local variable(s) which are pointed to, and then use an immediate after statement (`after: m(...)`) to delay the call to that method.

- ```
  h.send_now(msg1, ... msgN)
  ```

Sends a message through the hook, with message components *msg1* through *msgN*. The number of message components must match the number of message component types of the hook, and each message component must be compatible with the corresponding message component type of the hook.

send_now is *synchronous*: every computation suspended on the hook will execute before send_now completes.

send_now returns the number of suspended computations that were successfully resumed from the message being sent. Currently, every suspended computation is guaranteed to successfully be resumed unless cancelled by a preceding computation resumed by the send_now. This will not remain true in the future: coroutines are planned to be able to reject a message and reattach themselves to the hook.

- ```
  h.suspended
  ```

  Evaluates to the number of computations currently suspended on the hook.

References to hooks are valid run-time values: a reference to a hook with message component types *msgtype1* through *msgtypeN* will have the hook reference type hook(*msgtype1, ... msgtypeN*). This means hook references can be stored in variables, and can be passed around as method arguments or return values. In fact, hook references are even serializable.

Two hook references of the same hook reference type can be compared for equality, and are considered equal when they both reference the same hook.

> **Note**
>
> Hooks have a notable shortcoming in their lack of configurability; for example, there is no way to configure a hook to log an error when a message is sent through the hook and there is no computation suspended on the hook to act upon the message. Proper hook configurability is planned to be added by the time or together with coroutines being introduced to DML. Until then, the suggested approach is to create wrappers around usages of send_now(). Hook reference types can be leveraged to cut down on the needed number of such wrappers, for example:
>
> ```
> method send_now_checked_no_data(hook() h) {
>     local uint64 resumed = h.send_now();
>     if (resumed == 0) {
>         log error: "Unhandled message to hook";
>     }
> }
>
> method send_now_checked_int(hook(int) h, int x) {
>     local uint64 resumed = h.send_now(x);
>     if (resumed == 0) {
>         log error: "Unhandled message to hook";
>     }
> }
> ```

3.14 Object Declarations
  ```

The general form of an object declaration is "*type name extras* `is (`*template, ...*`)` *desc* `{ ... }`" or "*type name extras* `is (`*template, ...*`)` *desc*`;`", where *type* is an object type such as `bank`, *name* is an identifier naming the object, and *extras* is optional special notation which depends on the object type. The `is (`*template, ...*`)` part is optional and will make the object inherit the named templates. The surrounding parenthesis can be omitted if only one template is inherited. The *desc* is an optional string constant giving a very short summary of the object. It can consist of several string literals concatenated by the '+' operator. Ending the declaration with a semicolon is equivalent to ending with an empty pair of braces. The *body* (the section within the braces) may contain *parameter declarations*, *methods*, *session variable declarations*, *saved variable declarations*, *in each declarations* and *object declarations*.

For example, a `register` object may be declared as

```
register r0 @ 0x0100 "general-purpose register 0";
```

where the "`@` *offset*" notation is particular for the `register` object type; see below for details.

Using `is (`*template1, template2*`)` is equivalent to using `is` statements in the body, so the following two declarations are equivalent:

```
register r0 @ 0x0100 is (read_only,autoreg);

register r0 @ 0x0100 {
    is read_only;
    is autoreg;
}
```

An object declaration with a *desc* section, on the form

```
type name ... desc { ... }
```

is equivalent to defining the parameter `desc`, as in

```
type name ... {
    param desc = desc;
    ...
}
```

In the following sections, we will leave out *desc* from the object declarations, since it is always optional. Another parameter, `documentation` (for which there is no short-hand), may also be defined for each object, and for some object types it is used to give a more detailed description. See Section Universal Templates for details.)

If two object declarations with the same name occur within the same containing object, and they specify the same object type, then the declarations are concatenated; e.g.,

```
bank b {
    register r size 4 { ...body1... }
    ...
    register r @ 0x0100 { ...body2... }
    ...
}
```

is equivalent to

```
bank b {
    register r size 4 @ 0x0100  {
        ...body1...
        ...body2...
    }
    ...
}
```

However, it is an error if the object types should differ.

Most object types (`bank`, `register`, `field`, `group`, `attribute`, `connect`, `event`, and `port`) may be used in *arrays*. The general form of an object array declaration is

```
type name[var < size]... extras { ... }
```

Here each `[var < size]` declaration defines a dimension of resulting array. *var* defines the name of the index in that dimension, and *size* defines the size of the dimension. Each *variable* defines a parameter in the object scope, and thus must be unique. The size must be a compile time constant. For instance,

```
register regs[i < 16] size 2 {
    param offset = 0x0100 + 2 * i;
    ...
}
```

or written more compactly

```
register regs[i < 16] size 2 @ 0x0100 + 2 * i;
```

defines an array named `regs` of 16 registers (numbered from 0 to 15) of 2 bytes each, whose offsets start at 0x0100. See Section Universal Templates for details about arrays and index parameters.

The size specification of an array dimension may be replaced with `...` if the size has already been defined by a different declaration of the same object array. For example, the following is valid:

```
register regs[i < 16][j < ...] size 2 @ 0x0100 + 16 * i + 2 * j;
register regs[i < ...][j < 8] is (read_only);
```

The following sections give further details on declarations for object types that have special conventions.

3.14.1 Register Declarations

The general form of a `register` declaration is

```
register name size n @ d is (templates) { ... }
```

Each of the "`size n`", "`@ d`", and "`is (templates)`" sections is optional, but if present, they must be specified in the above order.

- A declaration

  ```
  register name size n ... { ... }
  ```

  is equivalent to

  ```
  register name ... { param size = n; ... }
  ```

- A declaration

  ```
  register name ... @ d ... { ... }
  ```

  is equivalent to

  ```
  register name  ... { param offset = d; ... }
  ```

3.14.2 Field Declarations

The general form of a `field` object declaration is

```
field name @ [highbit:lowbit] is (templates) { ... }
```

or simply

```
field name @ [bit] ... { ... }
```

specifying a range of bits of the containing register, where the syntax `[bit]` is short for `[bit:bit]`. Both the "`@ [...]`" and the `is (templates)` sections are optional; in fact, the "`[...]`" syntax is merely a much more convenient way of defining the (required) field parameters `lsb` and `msb`.

For a range of two or more bits, the first (leftmost) number always indicates the *most significant bit*, regardless of the bit numbering scheme used in the file. This corresponds to how bit fields are usually visualized, with the most significant bit to the left.

The bits of a register are always numbered from zero to *n* - 1, where *n* is the width of the register. If the default little-endian bit numbering is used, the least significant bit has index zero, and the most significant bit has index *n* - 1. In this case, a 32-bit register with two fields corresponding to the high and low half-words may be declared as

```
register HL size 4 ... {
    field H @ [31:16];
    field L @ [15:0];
}
```

If instead big-endian bit numbering is selected in the file, the most significant bit has index zero, and the least significant bit has the highest index. In that case, the register above may be written as

```
register HL size 4 ... {
    field H @ [0:15];
    field L @ [16:31];
}
```

This is useful when modeling a system where the documentation uses big-endian bit numbering, so it can be compared directly to the model.

3.15 Conditional Objects

It is also possible to conditionally include or exclude one or more object declarations, depending on the value of a boolean expression. This is especially useful when reusing source files between several similar models that differ in some of the details.

The syntax is very similar to the `#if` statements used in methods.

```
#if (enable_target) {
    connect target (
        interface signal;
    }
}
```

One difference is that the braces are required. It is also possible to add else branches, or else-if branches.

```
#if (modeltype == "Mark I") {
    ...
} #else #if (modeltype == "Mark II" {
    ...
} #else {
    ...
}
```

The general syntax is

```
#if ( conditional ) { object declarations ... }
#else #if ( conditional ) { object declarations ... }
...
#else { object declarations ... }
```

The *conditional* is an expression with a constant boolean value. It may reference parameters declared at the same level in the object hierarchy, or in parent levels.

The *object declarations* are any number of declarations of objects, session variables, saved variables, methods, or other `#if` statements, but not parameters, `is` statements, or `in each` statements . When the conditional is `true` (or if it's the else branch of a false conditional), the object declarations are treated as if they had appeared without any surrounding *#if*. So the two following declarations are equivalent:

```
#if (true) {
    register R size 4;
} #else {
    register R size 2;
}
```

is equivalent to

```
register R size 4;
```

3.16 In Each Declarations

In Each declarations are a convenient mechanism to apply a pattern to a group of objects. The syntax is:

`in each` (`template-name`, ...) `{ body }`

where `template-name` is the name of a template and `body` is a list of object statements, much like the body of a template. The statements in `body` are expanded in any subobjects that instantiate the template `template-name`, either directly or indirectly. If more than one `template-name` is given, then the body will be expanded only in objects that instantiate *all* the listed templates.

The `in each` construct can be used as a convenient way to express when many objects share a common property. For example, a bank can contain the following to conveniently set the size of all its registers:

```
in each register { param size = 2; }
```

Declarations in an `in each` block will override any declarations in the extended template. Furthermore, declarations in the scope that contains an `in each` statement, will override declarations from that `in each` statement. This can be used to define exceptions for the `in each` rule:

```
bank regs {
    in each register { param size default 2; }
    register r1 @ 0;
    register r2 @ 2;
    register r3 @ 4 { param size = 1; }
    register r4 @ 5 { param size = 1; }
}
```

An `in each` block is only expanded in subobjects; the object where the `in each` statement is present is unaffected even if it instantiates the extended template.

An `in each` statement with multiple template names can be used to cause a template to act differently depending on context:

```
template greeting { is read; }
template field_greeting is write {
    method write(uint64 val) {
        log info: "hello";
    }
}
in each (greeting, field) { is field_greeting; }
template register_greeting is write {
    method write(uint64 val) {
        log info: "world";
    }
}
in each (greeting, register) { is register_greeting; }

bank regs {
    register r0 @ 0 {
        // logs "hello" on write
        field f @ [0] is (greeting);
    }
    // logs "world" on write
    register r1 @ 4 is (greeting);
}
```

3.17 Global Declarations

The following sections describe the global declarations in DML. These can only occur on the top level of a DML model, i.e., not within an object or method. Unless otherwise noted, their scope is the entire model.

3.17.1 Import Declarations

```
import filename;
```

Imports the contents of the named file. *filename* must be a string literal, such as `"utility.dml"`. The `-I` option to the `dmlc` compiler can be used to specify directories to be searched for import files.

If *filename* starts with `./` or `../`, the compiler disregards the `-I` flag, and the path is instead interpreted relative to the directory of the importing file.

Note that imported files are parsed as separate units, and use their own language version and bit order declarations. A DML 1.4 file is not allowed to import a DML 1.2 file, but a DML 1.2 file may import a DML 1.4 file.

3.17.2 Template Declarations

Templates may only be declared on the top level, and the syntax and semantics for such declarations have been described previously.

Templates share the same namespace as types, as each template declaration defines a corresponding template type of the same name. It is illegal to define a template whose name conflicts with that of another type.

3.17.3 Bitorder Declarations

```
bitorder order;
```

Selects the default bit numbering scheme to be used for interpreting bit-slicing expressions and bit field declarations in the file. The *order* is one of the identifiers `le` or `be`, implying little-endian or big-endian, respectively. The little-endian numbering scheme means that bit zero is the least significant bit in a word, while in the big-endian scheme, bit zero is the most significant bit.

A `bitorder` declaration should be placed before any other global declaration in each DML-file, but must follow immediately after the `device` declaration if such one is present. The scope of the declaration is the whole of the file it occurs in. If no `bitorder` declaration is present in a file, the default bit order is `le` (little-endian). The bitorder does not extend to imported files; for example, if a file containing a declaration "`bitorder be;`" imports a file with no bit order declaration, the latter file will still use the default `le` order.

3.17.4 Constant Declarations

```
constant name = expr;
```

Defines a named constant. *expr* must be a constant-valued expression.

Parameters have a similar behaviour as constants but are more powerful, so constants are rarely useful. The only advantage of constants over parameters is that they can be used in `typedef` declarations.

3.17.5 Loggroup Declarations

```
loggroup name;
```

Defines a log group, for use in `log` statements. More generally, the identifier *name* is bound to an unsigned integer value that is a power of 2, and can be used anywhere in C context; this is similar to a `constant` declaration, but the value is allocated automatically so that all log groups are represented by distinct powers of 2 and can be combined with bitwise *or*.

A maximum of 63 log groups may be declared per device (61 excluding the built-in `Register_Read` and `Register_Write` log groups.)

3.17.6 Typedef Declarations

```
typedef declaration;
extern typedef declaration;
```

Defines a name for a data type.

When the `extern` form is used, the type is assumed to exist in the C environment. No definition of the type is added to the generated C code, and the generated C code blindly assume that the type

exists and has the given definition.

An `extern typedef` declaration may not contain a `layout` or `endian int` type.

If a `struct` type appears within an `extern typedef` declaration, then DMLC will assume that there is a corresponding C type, which has members of given types that can be accessed with the `.` operator. No assumptions are made on completeness or size; so the C struct may have additional fields, or it might be a `union` type. An empty member list is even allowed; this can make sense for opaque structs. DML variables of `extern` struct type are initialized such that any members of the C struct which are unknown to DML are initialized to 0.

Nested struct definitions are permitted in an `extern typedef` declaration, but an inner struct type only supports member access; it cannot be used as a standalone type. For instance, if you have:

```
extern typedef struct {
    struct { int x; } inner;
} outer_t;
```

then you can declare `local outer_t var;` and access the member `var.inner.x`, but the inner type is unknown to DML so you cannot declare a variable `local typeof var.inner *inner_p;`.

3.17.7 Extern Declarations

```
extern declaration;
```

Declares an external identifier, similar to a C `extern` declaration; for example,

```
extern char *motd;
extern double table[16];
extern conf_object_t *obj;
extern int foo(int x);
extern int printf(const char *format, ...);
```

Multiple `extern` declarations for the same identifier are permitted as long as they all declare the same type for the identifier.

3.17.8 Header Declarations

```
header %{
...
%}
```

Specifies a section of C code which will be included verbatim in the generated C header file for the device. There must be no whitespace between the `%` and the corresponding brace in the `%{` and `%}` markers. The contents of the header section are not examined in any way by the `dmlc` compiler; declarations made in C code must also be specified separately in the DML code proper.

This feature should only be used to solve problems that cannot easily be handled directly in DML. It is most often used to make the generated code include particular C header files, as in:

```
header %{
#include "extra_defs.h"
%}
```

The expanded header block will appear in the generated C file, which usually is in a different directory than the source DML file. Therefore, when including a file with a relative path, the C compiler will not automatically look for the `.h` file in the directory of the `.dml` file, unless a corresponding `-I` flag is passed. To avoid this problem, DMLC inserts a C macro definition to permit including a *companion header file*. For instance, if the file `/path/to/hello-world.dml` includes a header block, then the macro `DMLDIR_HELLO_WORLD_H` is defined as the string `"/path/to/hello-world.h"` within this header block. This allows the header block to contain `#include DMLDIR_HELLO_WORLD_H`, as a way to include `hello-world.h` without passing `-I/path/to` to the C compiler.

DMLC only defines one such macro in each header block, by taking the DML file name and substituting the `.dml` suffix for `.h`. Furthermore, the macro is undefined after the header. Hence, the macro can only be used to access one specific companion header file; if other header files are desired, then `#include` directives can be added to the companion header file, where relative paths are expanded as expected.

See also `footer` declarations, below.

3.17.9 Footer Declarations

```
footer %{
...
%}
```

Specifies a piece of C code which will be included verbatim at the end of the generated code for the device. There must be no whitespace between the `%` and the corresponding brace in the `%{` and `%}` markers. The contents of the footer section are not examined in any way by the `dmlc` compiler.

This feature should only be used to solve problems that cannot easily be handled directly in DML. See also `header` declarations, above.

3.17.10 Export Declarations

```
export method as name;
```

Exposes a method specified by *`method`* to other C modules within the same Simics module under the name *`name`* with external linkage. Note that inline methods, shared methods, methods that throw, methods with more than one return argument, and methods declared inside object arrays cannot be exported. It is sometimes possible to write wrapper methods that call into non-exportable methods to handle such cases, and export the wrapper instead.

Exported methods are rarely used; it is better to use Simics interfaces for communication between devices. However, exported methods can sometimes be practical in tight cross-language integrations, when the implementation of one device is split between one DML part and one C/C++ part.

Example: the following code in DML:

```
method my_method(int x) { ... }
export my_method as "my_c_function";
```

will export `my_method` as a C function with external linkage, using the following signature:

```
void my_c_function(conf_object_t *obj, int x);
```

The `conf_object_t *obj` parameter corresponds to the device instance, and is omitted when the referenced method is independent.

3.18 Resolution of overrides

This section describes in detail the rules for how DML handles when there are multiple definitions of the same parameter or method. A less technical but incomplete description can be found in the section on templates.

- Each declaration in every DML file is assigned a *rank*. The set of ranks form a partial order, and are defined as follows:
  - The top level of each file has a rank.
  - Each template definition has a rank.
  - The block in an `in each` declaration has a rank.
  - If one object declaration has rank R, then any subobject declaration inside it, also those inside an `#if` block, has rank R.
  - `param` and `method` declarations has the rank of the object they are declared within. This includes shared methods.
  - If an object declaration contains `is` *`T`*, then that object declaration has higher rank than the body of the template *`T`*.
  - If one file F_1 imports another file F_2, then the top level of F_1 has higher rank than the top level of F_2.
  - A declaration has higher rank than the block of any `in each` declaration it contains.
  - An `in each` block has higher rank than the templates it applies to
  - If there are three declarations D_1, D_2 and D_3, where D_1 has higher rank than D_2 and D_2 has higher rank than D_3, then D_1 has higher rank than D_3.
  - A declaration may not have higher rank than itself.
- In a set of `method` or `param` declarations that declare the same object in the hierarchy, then we say that one declaration *dominates* the set if it has higher rank than all other declarations in the set. Abstract `param` declarations (`param` *`name`*`;` or `param` *`name`* `:` *`type`*`;`) and abstract method definitions (`method` *`name`*`(`*`args...`*`);`) are excluded here; they cannot dominate a set, and a dominating declaration in a set does not need to have higher declaration than any abstract `param` or `method` declaration in the set.
- There may be any number of *untyped* abstract definitions of a parameter (`param` *`name`*`;`).
- There may be at most one *typed* abstract definition of a parameter (`param` *`name`* `:` *`type`*`;`)
- There may be at most one abstract shared definition of a method. Any other *shared* definition of this method must have higher rank than the abstract definition, but any rank is permitted for non-shared definitions. For instance:

```
template a {
    method m() default {}
}
template b {
    shared method m() default {}
}
template aa is a {
    // OK: overrides non-shared method
    shared method m();
}
template bb is b {
    // Error: abstract shared definition overrides non-abstract
    shared method m();
}
```

- When there is a set of declarations of the same a `method` or `param` object in the hierarchy, then there must be (exactly) one of these declarations that dominates the set; it is an error if there is not.
- If there is a `method` or `param` that is *not* declared `default`, then it must dominate the set of declarations of that method or parameter; it is an error if it does not.
- In the above two rules, "the set of declarations" of an object does not include declarations that are disabled through an `#if` statement, or definitions that appear in a template that never is instantiated in an object. However, the rules *do* also apply to *shared* method declarations in templates, regardless whether the templates are used. For instance:

```
template t1 {
    method a() {}
    shared method b() {}
}
template t2 is t1 {
    // OK, as long as t2 never is instantiated
    method a default {}
    // Error, even if t2 is unused
    shared method b() default {}
}
```

- If the set of declarations D_1, D_2, ..., D_n of a method M is dominated by the declaration D_n, then:
  - If there is a k, $1 \leq k \leq$ n-1, such that D_k dominates the set D_1, ..., D_{n-1}, then the symbol `default` refers to the method implementation of D_k within the scope of the method implementation of D_n.
  - If not, then `default` is an illegal value within the method implementation of D_n.

It follows that:

- The following code is illegal, because it would otherwise give T a higher rank than itself:

```
template T {
    #if (p) {
        group g is T {
            param p = false;
        }
    }
}
```

- Cyclic imports are not permitted, for the same reason.
- If an object is declared twice on the top level in the same file, then both declarations have the same rank. Thus, the following declarations of the parameter `p` count as conflicting, because neither has a rank that dominates the other:

```
bank b {
    register r {
        param p default 3;
    }
}
bank b {
    register r {
        param p = 4;
    }
}
```

3.19 Comparison to C/C++

The algorithmic language used to express method bodies in DML is an extended subset of ISO C, with some C++ extensions such as `new` and `delete`. The DML-specific statements and expressions are described in Sections Method Statements and Expressions.

DML defines the following additional built-in data types:

`int1`, ..., `int64`, `uint1`, ..., `uint64`

Signed and unsigned specific-width integer types. Widths from 1 to 64 are allowed.

`bool`

The generic boolean datatype, consisting of the values `true` and `false`. It is not an integer type, and the only implicit conversion is to `uint1`

DML also supports the non-standard C extension `typeof(`*`expr`*`)` operator, as provided by some modern C compilers such as GCC.

DML deviates from the C language in a number of ways:

- All integer arithmetic is performed on 64-bit numbers in DML, and truncated to target types on assignment. This is similar to how arithmetic would work in C on a platform where the `int` type is 64 bits wide (though in DML, `int` is an alias of `int32`). Similarly, all floating-point arithmetic is performed on the `double` type.

  For instance, consider the following:

```
local int24 x = -3;
local uint32 y = 2;
local uint64 sum = x + y;
```

In C, the expression `x + y` would cast both operands up to unsigned 32-bit integers before performing a 32-bit addition; overflow gives the result is $2^{32} - 1$, which is promoted without sign extension into a 64-bit integer before stored in the `sum` variable. In DML, both operands are instead promoted to 64-bit signed integers, so the addition evaluates to -1, which is stored as $2^{64} - 1$ in the `sum` variable.

Formally, if any of the two operands of an arithmetic binary operator (including bitwise operators) has the type `uint64`, then both operands are promoted into `uint64` before the operation; otherwise, both operands are promoted into `int64` before the operation. If any operand has floating-point type, then both operands are promoted into the `double` type.

- Comparison operators (`==`, `!=`, `<`, `<=`, `>` and `>=`) do *not* promote signed integers to unsigned before comparison. Thus, unlike in C, the following comparison yields `true`:

```
int32 x = -1;
uint64 val = 0;
if (val > x) { ... }
```

- The shift operators (`<<` and `>>`) have well-defined semantics when the right operand is large: Shifting by more than 63 bits gives zero (-1 if the left operand is negative). Shifting a negative number of bits is an error.
- Division by zero is an error.
- Signed overflow in arithmetic operations (`+`, `-`, `*`, `/`, `<<`) is well-defined. The overflow value is calculated assuming two's complement representation; i.e., the result is the unique value v such that $v \equiv r \pmod{2^{64}}$, where r is the result of operation using arbitrary precision arithmetic.
- Local variable declarations must use the keyword local, session, or saved; as in

```
method m() {
    session int call_count = 0;
    saved bool called = false;
    local int n = 0;
    local float f;
    ...
}
```

Session and saved variables have a similar meaning to static variables as in C: they retain value over function calls. However, such variables in DML are allocated per device object, and are not globally shared between device instances.

Unlike C, multiple simultaneous variable declaration and initialization is done through tuple syntax:

```
method m() {
    local (int n, bool b) = (0, true);
    local (float f, void *p);
    ...
}
```

- Plain C functions (i.e., not DML methods) can be called using normal function call syntax, as in `f(x)`.

  In order to call a C function from DML, three steps are needed:

  - In order for DML to recognize an identifier as a C function, it must be declared in DML, using an `extern` declaration.
  - In order for the C *compiler* to recognize the identifier when compiling generated C code, a function declaration must also be declared in a `header` section, or in a header file included from this section.
  - In order for the C *linker* to resolve the symbol, a function definition must be present, either in a separate C file or in a header or `footer` section.

  foo.c

```
int foo(int i)
{
    return ~i + 1;
}
```

  foo.h

```
int foo(int i);
```

  bar.dml

```
// tell DML that these functions are available
extern int foo(int);
extern int bar(int);

header %{
    // tell generated C that these functions are available
    #include "foo.h"
    int bar(int);  // defined in the DML footer section
%}

footer %{
    int bar(int i)
    {
        return -i;
    }
%}
```

Makefile

```
SRC_FILES=foo.c bar.dml
```

- Assignments (=) are required to be separate statements. You are still allowed to assign multiple variables in one statement, as in:

```
i = j = 0;
```

- Multiple simultaneous assignment can be performed in one statement through tuple syntax, allowing e.g. the following:

```
(i, j) = (j, i);
```

  However, such assignments are not allowed to be chained.

- If a method can throw exceptions, or if it has more than one return argument, then the call must be a separate statement. If it has one or more return values, these must be assigned. If a method has multiple return arguments, these are enclosed in a parenthesis, as in:

```
method divmod(int x, int y) -> (int, int) {
    return (x / y, x % y);
}
...
(quotient, remainder) = divmod(17, 5);
```

- Type casts must be written as `cast(expr, type)`.
- Comparison operators and logical operators produce results of type `bool`, not integers.
- Conditions in `if`, `for`, `while`, etc. must be proper booleans; e.g., `if (i == 0)` is allowed, and `if (b)` is allowed if `b` is a boolean variable, but `if (i)` is not, if `i` is an integer.
- The `sizeof` operator can only be used on lvalue expressions. To take the size of a datatype, the `sizeoftype` operator must be used.
- Comma-expressions are only allowed in the head of `for`-statements, as in

```
for (i = 10, k = 0; i > 0; --i, ++k) ...
```

- `delete` and `throw` can only be used as statements in DML, not as expressions.
- `throw` does not take any argument, and `catch` cannot switch on the type or value of an exception.
- Type declarations do not allow the use of `union`. However, the `extern typedef` construct can be used to achieve the same result. For example, consider the union data type declared in C as:

```
typedef union { int i; bool b; } u_t;
```

  The data type can be exposed in DML as follows:

```
header %{
    typedef union { int i; bool b; } u_t;
%}
extern typedef struct { int i; bool b; } u_t;
```

This will make `u_t` look like a struct to DML, but since union and struct syntax is identical in C, the C code generated from uses of `u_t` will work correctly together with the definition from the `header` declaration.

3.20 Method Statements

All ISO C statements are available in DML, and have the same semantics as in C. Like ordinary C expressions, all DML expressions can also be used in expression-statements.

DML adds the following statements:

3.20.1 Assignment Statements

```
target1 [= target2 = ...] = initializer;
(target1, target2, ...) = initializer;
```

Assign values to targets according to an initializer. Unlike C, assignments are not expressions, and the right-hand side can be any initializer — such as compound initializers (`{...}`) for struct-like types.

The first form is chaining assignments. The initializer is executed once and the value it evaluates to is assigned to each target.

The second form is multiple simultaneous assignment. The initializer describes multiple values — one for each target. This can be done either through:

- Providing an initializer for each target through tuple syntax, e.g.:

```
(a, i) = (false, 4);
```

- Performing a method call where each target is a return value recipient, e.g.:

```
method m() -> (bool, int) {
    ...
}
```

```
(a, i) = m();
```

Targets are updated simultaneously, meaning it's possible to e.g. swap the contents of variables through the following:

```
(a, b) = (b, a)
```

3.20.2 Local Statements

```
local type identifier [= initializer];
local (type1 identifier1, type2 identifier2, ...) [= initializer];
```

Declares one or multiple local variables in the current scope. The right-hand side is an initializer, meaning, for example, that compound initializers (`{...}`) can be used.

The initializer must provide the exact number of values needed to initialize the variables, and they must be of compatible type. Multiple values can be provided either through:

- Providing an initializer for each variable through tuple syntax, e.g.:

```
local (bool a, int i) = (false, 4);
```

- Performing a method call where each return value initializes a variable, e.g.:

```
method m() -> (bool, int) {
    ...
}
```

```
local (bool a, int i) = m();
```

In the absence of explicit initializer expressions, a default "all zero" initializer will be applied to each declared object.

3.20.3 Session Statements

```
session type identifier [= initializer];
session (type1 identifier1, type2 identifier2, ...) [= (initializer1, initializer2, ...)];
```

Declares one or multiple session variables in the current scope. Note that initializers of such variables are evaluated *once* when initializing the device, and thus must be a compile-time constant.

3.20.4 Saved Statements

```
saved type identifier [= initializer];
sabed (type1 identifier1, type2 identifier2, ...) [= (initializer1, initializer2, ...)];
```

Declares one or multiple saved variables in the current scope. Note that initializers of such variables are evaluated *once* when initializing the device, and thus must be a compile-time constant.

3.20.5 Return Statements

```
return [initializer];
```

Returns from method with the value(s) specified by the argument. Unlike C, the argument is an *initializer*, meaning, for example, return values of struct-like type can be constructed using `{...}`.

The initializer must provide the exact number of values corresponding as the return values of the method, and they must be of compatible type. Multiple values can be provided either through:

- Providing an initializer for each return value through tuple syntax, e.g.:

```
method m() -> (bool, int) {
    return (false, 4);
}
```

- Performing a method call and propagating the return values:

```
method n() -> (bool, int) {
    return m();
}
```

3.20.6 Delete Statements

```
delete expr;
```

Deallocates the memory pointed to by the result of evaluating *expr*. The memory must have been allocated with the new operator, and must not have been deallocated previously. Equivalent to delete in C++; however, in DML, delete can only be used as a statement, not as an expression.

3.20.7 Try Statements

```
try protected-stmt catch handle-stmt
```

Executes *protected-stmt*; if that completes normally, the whole try-statement completes normally. Otherwise, *handle-stmt* is executed. This is similar to exception handling in C++, but in DML there is only one kind of exception. Note that Simics C-exceptions are not handled. See also throw.

3.20.8 Throw Statements

```
throw;
```

Throws (raises) an exception, which may be caught by a try-statement. This is similar to throw in C++, but in DML it is not possible to specify a value to be thrown. Furthermore, in DML, throw is a statement, not an expression.

If an exception is not caught inside a method body, then the method must be declared as throws, and the exception is propagated over the method call boundary.

3.20.9 Method Calls

```
(d1, ... dM) = method(e1, ... eN);
```

A DML method is called similarly as a C function, with the exception that you must have assignment destinations according to the number of return values of the method. Here a DML method is called with input arguments *e1*, ... *eN*, assigning return values to destinations *d1*, ... *dM*.

The destinations are usually variables, but they can be arbitrary L-values (even bit slices) as long as their types match the method signature.

If the method has no return value, the call is simply expressed as:

```
p(...);
```

A method with exactly one return value can also be called in any expression, unless it is an inline method, or a method that can throw exceptions. For example:

```
method m() -> (int) { ... }
...
if (m() + i == 13) { ... }
```

A method call (even if it is throwing or has multiple return values) can be used as an initializer in any context that accepts non-constant initializers; i.e., in assignment statements (as shown above), local variable declarations, and return statements. For example:

```
// declare multiple variables, and initialize them from one method call
local (int i, uint8 j) = m(e1);

// Propagate all return values from a method call as the return values of the
// caller.
return m(e1)
```

3.20.10 Template-Qualified Method Implementation Calls

Every object, as well as every template type, has a `templates` member to allow for calling *particular* implementations of that object's methods, as opposed to only the final overriding implementations that are reachable directly. Specifically, `templates` allows for invoking any particular implementation as provided by a specified template instantiated by the object. Such invocations are called *template-qualified method implementation calls*, and are made as follows:

```
template t {
    method m() default {
        log info: "implementation from 't'"
    }
}

template u is t {
    method m() default {
        log info: "implementation from 'u'"
    }
}

group g is u {
    method m() {
        log info: "final implementation"
    }
}

method call_ms() {
    // Logs "final implementation"
    g.m();
    // Logs "implementation from 'u'"
    g.templates.u.m();
    // Logs "implementation from 't'"
    g.templates.t.m();
}
```

Template-qualified method implementation calls are primarily meant as a way for an overriding method to reference overridden implementations, *even when* the implementations are provided by hierarchically unrelated templates such that `default` can't be used (see Resolution of overrides.) In particular, this typically allows for ergonomically resolving conflicts introduced when multiple orthogonal templates are instantiated, as long as all conflicting implementations are overridable, and one of the following is true:

- The implementations can be combined together by calling each one of them, as long as that can be done without risking e.g. side-effects being duplicated.
- The implementations can be combined by choosing one particular template's implementation to invoke (typically the one most complex), and then adding code around that implementation call in order to replicate the behaviour of the implementations of the other templates. Ideally, the other templates would provide methods that may be leveraged so that their behaviour may be replicated without the need for excessive boilerplate.

The following is an example of the first case:

```
template alter_write is write {
    method write(uint64 written) {
        default(alter_write(written));
    }

    method alter_write(uint64 curr, uint64 written) -> (uint64);
}

template gated_write is alter_write {
    method write_allowed() -> (bool) default {
        return true;
    }

    method alter_write(uint64 curr, uint64 written) -> (uint64) default {
        return write_allowed() ? written : curr;
    }
}

template write_1_clears is alter_write {
    method alter_write(uint64 curr, uint64 written) -> (uint64) default {
        return curr & ~written;
    }
}

template gated_write_1_clears is (gated_write, write_1_clears) {
    method alter_write(uint64 curr, uint64 written) default {
        local uint64 new = this.templates.write_1_clears.alter_write(
            curr, written);
        return this.templates.gated_write.alter_write(curr, new);
    }
}

// Resolve the conflict introduced whenever the two orthogonal templates are
// instantiated by also instantiating gated_write_1_clears when that happens
in each (gated_write, write_1_clears) { is gated_write_1_clears; }
```

The following is an example of the second case:

```
template very_complex_register is register {
    method write_register(uint64 written, uint64 enabled_bytes,
                          void *aux) default {
        ... // An extremely complicated implementation
    }
}

template gated_register is register {
    method write_allowed() -> (bool) default {
        return true;
    }

    method on_write_attempted_when_not_allowed() default {
        log spec_viol: "%s was written to when not allowed", qname;
    }

    method write_register(uint64 written, uint64 enabled_bytes,
                          void *aux) default {
        if (write_allowed()) {
            default(written, enabled_bytes, aux);
        } else {
            on_write_attempted_when_not_allowed();
        }
    }
}

template very_complex_gated_register is (very_complex_register,
                                         gated_register) {
    // No sensible way to combine the two implementations by calling both.
    // Even if there were, calling both implementations would cause each field
    // of the register to be written to multiple times, potentially duplicating
    // side-effects, which is undesirable.
    // Instead, very_complex_register is chosen as the base implementation
    // called, and the behaviour of gated_register is replicated around that
    // call.
    method write_register(uint64 written, uint64 enabled_bytes,
                          void *aux) default {
        if (write_allowed()) {
            this.templates.very_complex_register.write_register(
                written, enabled_bytes, aux);
        } else {
            on_write_attempted_when_not_allowed();
        }
    }
}

in each (gated_register, very_complex_register) {
    is very_complex_gated_register;
}
```

A template-qualified method implementation call is resolved by using the method implementation provided to the object by the named template. If no such implementation is provided (whether it be because the template does not specify one, or specifies one which is not provided to the object due to its definition being eliminated by an `#if`), then the ancestor templates of the named template are recursively searched for the highest-rank (most specific) implementation provided by them. If the ancestor templates provide multiple hierarchically unrelated implementations, then the choice is ambiguous and the call will be rejected by the compiler. In this case, the modeller must refine the template-qualified method implementation call to name the ancestor template whose implementation they would like to use.

A template-qualified method implementation call done via a value of template type functions differently compared to compile-time object references. In particular, `this.templates` within the bodies of `shared` methods functions differently. The specified template must be an ancestor template of the value's template type, the `object` template, or the template type itself; furthermore, the specified template **must provide or inherit a `shared` implementation of the named method**. It is not sufficient that the method is simply *declared* `shared` such that it is part of the template type: the implementation itself must also be `shared`. For more information, see the documentation of the `ENSHAREDTQMIC` error message.

3.20.11 After Statements

```
after ...: method(e1, ... eN);
```

The `after` statement sets up the given method call (the *callback*) such that it will be performed with the provided arguments at a specified point in the future. There are three different forms of the `after` statement, syntactically determined through what appears before the `:` — each form corresponds to different specifications of at what future point the method should be called.

A method call suspended using an `after` statement will be performed at most once per execution of the `after` statement; it will not recur. If it's desirable to have a suspended method call recur, then the called method must itself make use of `after` to set up a method call to itself.

The referenced method must be a regular or independent method with no return values. It may not be a C function, or a `shared` method. The only exception to this is that the `send_now` operation of `hooks` is also supported for use as a callback.

All method calls suspended via an `after` statement are *associated* with the object that contains the method containing the statement. It is possible to cancel all suspended method calls associated with an object through that object's `cancel_after()` method, as provided by the `object` template.

> **Note**
>
> We plan to extend the `after` statement to allow for users to explicitly state what objects the suspended method call is to be associated with.

3.20.11.1 After Delay Statements

```
after scalar unit: method(e1, ... eN);
```

In this form, the specified point in the future is given through a time delay (in simulated time, measured in the specified time unit) relative to the time when the after delay statement is executed. The currently supported time units are `s` for seconds (with type `double`), `ps` for picoseconds (with type `uint64`), and `cycles` for cycles (with type `uint64`).

Every argument to the called method is evaluated at the time the `after` statement is executed, and stored so that they may be used when the method call is to be performed. In order to allow the suspended method call to be represented in checkpoints, every input parameter of the method must be of *serializable type*. This means that after delay statements cannot be used with methods that e.g. have pointer input parameters. unless the arguments for those input parameters are message component parameters of the `after`.

Example:

```
after 0.1 s: my_callback(1, false);
```

The after delay statement is equivalent to creating a named `event` object with an event-method that performs the specified call, and posting that event at the given time, with associated data corresponding to the provided arguments.

3.20.11.2 Hook-Bound After Statements

```
after hookref[-> (msg1, ... msgN)]: method(e1, ... eM);
```

In this form, the suspended method call is bound to the hook specified by *`hookref`*. The point in the future when the method call is executed is thus the next time a message is sent through the specified hook.

The *binding syntax* `-> (msg1, ... msgN)` is used to bind each component of the message received to a corresponding identifier, called a *message component parameter*. These message component parameters can be used as arguments of the called method, thus propagating the contents of the message to the method call.

Every argument to the called method which isn't a message component parameter is evaluated at the time the `after` statement is executed, and stored so that they may be used when the method call is to be performed. In order to allow the suspended method call to be represented in checkpoints, every input parameter of the method must be of *serializable type*, unless that input parameter receives a message component. This means that hook-bound after statements cannot be used with methods that e.g. have pointer input parameters, unless the arguments for those input parameters are message component parameters of the `after`.

Example use:

```
hook(int, float) h;

method my_callback(int i, float f, bool b) {
    ...
}

method m() {
    after h -> (x, y): my_callback(x, y, false);
}

method send_message() {
    // Assuming m() has been called once before, this 'send_now' will result in
    // `my_callback(1, 3.7, false)` being called.
    h.send_now(1, 3.7);
}
```

If the hook has only one message component, the syntax `-> msg` can be used instead, and if the hook has no message components, then the binding syntax can be entirely omitted. Any message component parameter can be used for any number of arguments, but cannot be used as anything *but* a direct argument. For example, using the definitions of `h` and `my_callback` as above, the following use of `after` is valid:

```
after h -> (x, y): my_callback(x, x, false)
```

Note that the first message component is used multiple times, and the second is not used at all.

In contrast, the following use of `after` is invalid:

```
after h -> (x, y): my_callback(i, y + 1.5, false)
```

as the message component parameter `y` is used, but not as a direct argument.

3.20.11.3 Immediate After Statements

```
after: method(e1, ... eN);
```

In this form, the specified point in the future is when control is given back to the simulation engine such that the ongoing simulation of the current processor may progress, and would otherwise be ready to move onto the next cycle. This happens after all entries to devices on the call stack have been completed.

Immediate after statements are most useful to avoid ordering bugs. It can be used to delay a method call until all current lines of execution into the device have been completed, and the device is guaranteed to be in a consistent state where it is ready to handle the method call.

Semantically, the immediate after statement is very close to `after 0 cycles: ...`, but has a number of advantages. In general, the immediate after statement is designed to execute the callback as promptly as possible while satisfying the semantics stated above, while `after 0 cycles: ...` is not. In particular, in Simics, callbacks delayed via `after 0 cycles` are always bound to the clock associated with the device instance, which is not always that of the processor currently under simulation — in such cases the simulated processor may progress indefinitely without the posted callback being executed. The immediate after statement does not have this issue. In addition, if an immediate after statement is executed while the simulation is stopped (due to a device entry such as an attribute get/set performed from a script/CLI) then the callback is registered as *work*, thus guaranteeing that it is called before the simulation starts again.

Within a particular device instance, method calls suspended by immediate after statements are executed in order of least recently suspended; in other words, FIFO semantics. The order in which method calls suspended by immediate after statements are executed across multiple device instances is not defined.

Within an immediate after statement, every argument provided to the called method is evaluated at the time the `after` statement is executed, and stored so that they may be used when the method call is to be performed. Unlike the other forms of `after` statements, the input parameters of the method are never required to be of serializable type, meaning pointers can be passed as arguments to the callback. But **beware**: pointers to stack-allocated data (pointers to or within `local` variables) must **never** be passed as arguments. The stack with which the `after` statement is executed is *not* preserved, so any pointers to stack-allocated data will point to invalid data by the time the callback is called. The DML compiler has some checks in place to warn about the most

obvious cases where pointers to stack-allocated data are provided as arguments, but it is unable to detect all cases. It is ultimately the modeller's responsibility to ensure it doesn't happen.

To detail a scenario exemplifying the kind of issues that immediate after may be leveraged to solve, consider the following device, which needs to communicate with a *manager* device and receive permission in order to perform a particular action. Its function is simple: once prompted, the device will raise a signal to the manager in order to request permission, and waits for it to respond with an acknowledgement. Once received, the device lowers the signal to the manager and performs the action it just received permission for. In order to implement the asynchronous logic needed for this, a simple FSM is used.

```
param STATE_IDLE = 0;
param STATE_EXPECTING_ACK = 1;
saved int curr_state = STATE_IDLE;

port manager_link {
    connect manager {
        interface signal;
    }

    implement signal {
        method signal_raise() {
            on_acknowledgement();
        }
    }
}

method request_permission_for_action() {
    if (curr_state != STATE_IDLE) {
        log error: "Request already in progress";
        return;
    }
    manager_link.manager.signal.signal_raise();
    curr_state = STATE_EXPECTING_ACK;
}

method on_acknowledgement() {
    if (curr_state != STATE_EXPECTING_ACK) {
        log spec_viol: "Received ack when not expecting it";
        return;
    }
    manager_link.manager.signal.signal_lower();
    perform_permission_gated_action();
    curr_state = STATE_IDLE;
}
```

This device has a subtle bug: it can't handle if the manager responds to the `signal_raise()` call synchronously. The FSM transitions to the state capable of handling the acknowledgement only after the `signal_raise()` call returns, so if the manager responds synchronously — as part of the `signal_raise()` call — then `on_acknowledgement` will be called while the device still considers itself to be in its idle state.

This bug can be solved in numerous ways — the most obvious is to transition the state before making the `signal_raise()` call — but immediate after provides a solution which doesn't require carefully managing the FSM's logic, by delaying the call to `on_acknowledgement` until the device is done with all other logic.

```
implement signal {
    method signal_raise() {
        after: on_acknowledgement();
    }
}
```

This guarantees that the FSM is able to finish its current line of execution and properly transition itself to its new state before it's asked to manage any response of manager, even if the manager responds synchronously.

3.20.12 Log Statements

```
log log-type[, level [ then subsequent-level ] [, groups] ]: format-string, e1, ..., eN;
```

Outputs a formatted string to the Simics logging facility. The string following the colon is a normal C `printf` format string, optionally followed by one or more arguments separated by commas. (The format string should not contain the name of the device, or the type of the message, e.g., "error:..."; these things are automatically prefixed.) Either both of *`level`* and *`groups`* may be omitted, or only the latter; i.e., if *`groups`* is specified, then *`level`* must also be given explicitly.

A Simics user can configure the logging facility to show only specific messages, by matching on the three main properties of each message:

- The *`log-type`* specifies the general category of the message. The value must be one of the identifiers `info`, `warning`, `error`, `critical`, `spec_viol`, or `unimpl`.
- The *`level`* specifies at what verbosity level the log messages are displayed. The value must be an integer from 1 to 4; if omitted, the default level is 1. The different levels have the following meaning:

    1. Important messages (displayed at the normal verbosity level)
    2. High level informative messages (like mode changes and important events)
    3. Medium level information (the lowest log level for SW development)
    4. Debugging level with low level model detail (Mainly used for model development)

    If the *`log-type`* is one of `warning`, `error` or `critical`, then *`level`* may only be 1.

- If *`subsequent-level`* is specified, then all logs after the first issued will be on the level *`subsequent-level`*. You are allowed to specify a *`subsequent-level`* of 5, meaning no logging after the initial log.

    If the *`log-type`* is one of `warning`, `error` or `critical`, then *`subsequent-level`* may only be either 1 or 5.

- The *`groups`* argument is an integer whose bit representation is used to select which log groups the message belongs to. If omitted, the default value is 0. The log groups are specific for the device, and must be declared using the `loggroup` device-level declaration. For example, a DML source file containing the declarations

```
loggroup good;
loggroup bad;
loggroup ugly;
```

could also contain a log statement such as

```
log info, 2, (bad | ugly): "...";
```

(note the `|` bitwise-or operator), which would be displayed if the user chooses to view messages from group `bad` or `ugly`, but not if only group `good` is shown.

Groups allow the user to create arbitrary classifications of log messages, e.g., to indicate things that occur in different states, or in different parts of the device, etc. The two log groups `Register_Read` and `Register_Write` are predefined by DML, and are used by several of the built-in methods.

The *`format-string`* should be one or several string literals concatenated by the '+' operator, all optionally surrounded by round brackets.

See also *Simics Model Builder User's Guide*, section "Logging", for further details.

3.20.13 Assert Statements

```
assert expr;
```

Evaluates *`expr`*. If the result is `true`, the statement has no effect; otherwise, a runtime-error is generated. *`expr`* must have type `bool`.

3.20.14 Error Statements

```
error [string];
```

Attempting to compile an `error` statement causes the compiler to generate an error, using the specified string as error message. The string may be omitted; in that case, a default error message is used.

The *`string`*, if present, should be one or several string literals concatenated by the '+' operator, all optionally surrounded by round brackets.

3.20.15 Foreach Statements

```
foreach identifier in (expr) statement
```

The `foreach` statement repeats its body (the *`statement`* part) once for each element given by *`expr`*. The *`identifier`* is used to refer to the current element within the body.

DML currently only supports `foreach` iteration on values of `sequence` types — which are created through Each-In expressions.

The `continue` statement can be used within a `foreach` loop to continue to the next element, and the `break` statement can be used to exit the loop.

```
#foreach identifier in (expr) statement
```

In this alternative form the *expr* is required to be a DML compile-time constant, and the loop is completely unrolled by the DML compiler. This can be combined with tests on the value of *identifier* within the body, which will be evaluated at compile time.

DML currently only supports `#foreach` iteration on compile-time list constants.

For example:

```
#foreach x in ([3,2,1]) {
    #if (x == 1) foo();
    #else #if (x == 2) bar();
    #else #if (x == 3) baz();
    #else error "out of range";
}
```

would be equivalent to

```
baz();
bar();
foo();
```

Only `#if` can be used to make such selections; `switch` or `if` statements are *not* evaluated at compile time. (Also note the use of `error` above to catch any compile-time mistakes.)

The `break` statement can be used within a `#foreach` loop to exit it.

3.20.16 Select Statements

```
select identifier in (expr) where (cond-expr) statement else default-statement
```

The `select` statement resembles a C `switch` statement and is very similar to the `foreach` statement, but executes the *statement* exactly once for the first matching element of those given by *expr*, i.e., for the first element such that *cond-expr* is `true`; or if no element matches, it executes the *default-statement*.

```
#select identifier in (expr) where (cond-expr) statement #else default-statement
```

In this alternative form the *expr* is required to be a DML compile-time constant, and *cond-expr* can only depend on compile-time constants, apart from *identifier*. The selection will then be performed by the DML compiler at compile-time, and code will only be generated for the selected case.

DML currently only supports `#select` iteration on compile-time list constants.

> **Note**
>
> The `select` statement has been temporarily removed from DML 1.4 due to semantic issues, and only the `#select` form may currently be used. The `select` statement will be reintroduced in the near future.

3.20.17 #if and #else Statements

```
#if (condition) { true_body } #else { false_body }
```

The `#if` statement resembles a C `if` statement. The difference being that the `#if` statement must have a constant-valued *condition* and the statement is evaluated at compile-time. The *true_body* of the `#if` is only processed if the condition evaluates to `true`, and will be dead-code eliminated otherwise.

Similarly, the `#else` statement can immediately follow the body of an `#if` statement and the *false_body* will only be processed if the *condition* in the preceding `#if` evaluates to `false`.

3.21 Expressions

All ISO C operators are available in DML, except for certain limitations on the comma-operator, the `sizeof` operator, and type casts; see Section Comparison to C/C++. Operators have the same precedences and semantics as in C

DML adds the following expressions:

3.21.1 The Undefined Constant

```
undefined
```

The constant `undefined` is an abstract *compile-time only* value, mostly used as a default for parameters that are intended to optionally be overridden. The `undefined` expression may only appear as a parameter value and as argument to the `defined` *expr* test (see below).

3.21.2 References

```
identifier
```

To reference something in the DML object structure, members may be selected using `.` and `->` as in C. (However, most objects in the DML object structure are proper substructures selected with the `.` operator.) For example,

```
this.size # a parameter
dev.bank1 # a bank object
bank1.r0.hard_reset # a method
```

The DML object structure is a compile-time construction; references to certain objects are not considered to be proper values, and result in compile errors if they occur as standalone expressions.

Some DML objects are proper values, while others are not:

- `session`/`saved` variables are proper values
- Composite object references (to `bank`, `group`, `register`, etc.) are not proper values unless cast to a template type.
- Inside an object array, the index variable (named `i` by default) may evaluate to an *unknown index* if accessed from a location where the index is not statically known. For instance, in `group`

`g[i < 4] { #if (i == 0) { ... } }`, the `#if` statement is invoked once, statically, across all indices, meaning that the `i` reference is an unknown index, and will yield a compile error.

- A reference to a `param` member is a proper value only if the parameter value is a proper value: A parameter value can be a reference to an object, an object array, a list, the `undefined` expression, or a static index (discussed above), in which case the parameter is not allowed as a standalone expression.
- When the object structure contains an array of objects, e.g. `register r[4] { ... }`, then a reference to the array itself (i.e. `r` as opposed to `r[0]`), is not considered a proper value.

If a DML object is not a proper value, then a reference to the object will give a compile error unless it appears in one of the following contexts:

- As the left operand of the `.` operator
- As the definition of a `param`
- As a list element in a compile-time list
- As the operand of the `defined` operator
- A `method` object may be called
- An object array may appear in an index expression *`array`*`[`*`index`*`]`
- An unknown index may be used as an index to an object array; in the resulting object reference, the corresponding index variable of the object array will have an unknown value.

3.21.3 Method References as Function Pointers

It is possible to retrieve a function pointer for a method by using the prefix operator `&` with a reference to that method. The methods this is possible with are subject to the same restrictions as with the `export` object statement: it's not possible to retrieve a function pointer to any inline method, shared method, method that throws, method with more than one return argument, or method declared inside an object array.

For example, with the following method in DML:

```
method my_method(int x) { ... }
```

then the expression `&my_method` will be a function pointer of type:

```
void (*)(conf_object_t *, int);
```

The `conf_object_t *` parameter corresponds to the device instance, and is omitted when the referenced method is independent.

Note that due to the precedence rules of `&`, if you want to immediately call a method reference converted to a function pointer, then you will need to wrap parentheses around the converted method reference. An example of where this may be useful is in order to call a non-independent method from within an independent method:

```
independent method callback(int i, void *aux) {
  local conf_object_t *obj = aux;
  (&my_method)(obj, i);
}
```

3.21.4 New Expressions

```
new type

new type[count]
```

Allocates a chunk of memory large enough for a value of the specified type. If the second form is used, memory for *count* values will be allocated. The result is a pointer to the allocated memory. (The pointer is never null; if allocation should fail, the Simics application will be terminated.)

When the memory is no longer needed, it should be deallocated using a `delete` statement.

3.21.5 Cast Expressions

```
cast(expr, type)
```

Type casts in DML must be written with the above explicit `cast` operator, for syntactical reasons.

Semantically, `cast(expr, type)` is equivalent to the C expression `(type) expr`.

3.21.6 Sizeoftype Expressions

```
sizeoftype type
```

The `sizeof` operator in DML can only be used on expressions, not on types, for syntactical reasons. To take the size of a datatype, the `sizeoftype` operator must be used, as in

```
int size = sizeoftype io_memory_interface_t;
```

Semantically, `sizeoftype type` is equivalent to the C expression `sizeof (type)`.

DML does not know the sizes of all types statically; DML usually regards a `sizeoftype` expression as non-constant and delegates size calculations to the C compiler. DML does evaluate the sizes of integer types, layout types, and constant-sized arrays thereof, as constants.

3.21.7 Defined Expressions

```
defined expr
```

This compile-time test evaluates to `false` if *expr* has the value `undefined`, and to `true` otherwise.

3.21.8 Each-In Expressions

An expression `each-in` is available to traverse all objects that implement a specific template. This can be used as a generic hook mechanism for a specific template, e.g. to implement custom reset patterns. For example, the following can be used to reset all registers in the bank `regs`:

```
foreach obj in (each hard_reset_t in (regs)) {
    obj.hard_reset();
}
```

An each-in expression can currently only be used for iteration in a foreach statement. The expression's type is sequence(*template-name*).

An each-in expression searches recursively in the object hierarchy for objects implementing the template, but once it finds such an object, it does not continue searching inside that subobject. Recursive traversal can be achieved by letting the template itself contain a method that descends into subobjects; the implementation of hard_reset in utility.dml demonstrates how this can be done.

The order in which objects are given by a specific each-in expression is not defined, except for that it is deterministic. That is, for a particular choice of template X and object Y in an each X in (Y) expression, for a particular iteration of the device model, and for the particular DMLC build used, the order in which objects are given by that expression is guaranteed to be consistent.

3.21.9 List Expressions

```
[e1, ..., eN]
```

A list is a *compile-time only* value, and is an ordered sequence of zero or more expressions. Lists are in particular used in combination with foreach and select statements.

A list expression may only appear in the following contexts:

- As the list to iterate over in a #foreach or #select statement
- As the value in a param or constant declaration
- As a list element in another compile-time list
- In an index expression, *list*[*index*]
- As the operand of the defined operator

3.21.10 Length Expressions

```
list.len

sequence.len

object-array.len

value-array.len
```

Used to obtain the length of a *list*, *sequence*, *object-array*, or *value-array* expression. This expression is constant for each form but *sequence* expressions.

The *value-array* form can only be used with arrays of known constant size: it can't be used with pointers, arrays of unknown size, or variable-length arrays.

3.21.11 Bit Slicing Expressions

```
expr[e1:e2]

expr[e1:e2, bitorder]

expr[e1]

expr[e1, bitorder]
```

If *expr* is of integer type, then the above *bit-slicing* syntax can be used in DML to simplify extracting or updating particular bit fields of the integer. Bit slice syntax can be used both as an expression producing a value, or as the target of an assignment (an L-value), e.g., on the left-hand side of an = operator.

Both *e1* and *e2* must be integers. The syntax *expr*`[`*e1*`]` is a short-hand for *expr*`[`*e1*`:`*e1*`]` (but only evaluating *e1* once).

The *bitorder* part is optional, and selects the bit numbering scheme (the "endianness") used to interpret the values of *e1* and *e2*. If present, it must be one of the identifiers `be` or `le`, just as in the `bitorder` device-level declaration. If no *bitorder* is given in the expression, the global bit numbering (as defined by the `bitorder` declaration) is used.

The first bit index *e1* always indicates the *most significant bit* of the field, regardless of the bit numbering scheme. If the default little-endian bit numbering is used, the least significant bit of the integer has index zero, and the most significant bit of the integer has index *n* - 1, where *n* is the width of the integer type.

If big-endian bit numbering is used, e.g., due to a `bitorder be;` declaration in the file, or using a specific local bit numbering as in *expr*`[`*e1*`:`*e2*`, be]`, then the bit corresponding to the little-endian bit number *n* - 1 has index zero, and the least significant bit has the index *n* - 1, where *n* is the bit width of *expr*. Note that big-endian numbering is illegal if *expr* isn't a simple expression with a well-defined bit width. This means that only local variables, method parameters, device variables (registers, data etc), and explicit cast expressions are allowed. For little-endian numbering, any expressions are allowed, since there is never any doubt that bit 0 is the least significant bit.

If the bit-slicing expression results in a zero or negative sized range of bits, the behavior is undefined.

3.21.12 Stringify Expressions

```
stringify(expr)
```

Translates the value of *expr* (which must be a compile-time constant) into a string constant. This is similar to the use of # in the C preprocessor, but is performed on the level of compile time values, not tokens. The result is often used with the + string operator.

3.21.13 String Concatenation Expressions

```
expr1 + expr2
```

If both *expr1* and *expr2* are compile-time string constants, the expression *expr1 + expr2* concatenates the two strings at compile time. This is often used in combination with the # operator, or to break long lines for source code formatting purposes.

3.21.14 Compile-Time Conditional Expressions

```
condition #? expr1 #: expr2
```

Similar to the C `conditional` expression, with the difference that the *condition* must have a constant value and the expression is evaluated at compile-time. *expr1* is only processed if the *condition* is `true` and *expr2* is only processed if *condition* is `false`, so an expression like `false #? 1/0 #: 0` is equivalent to `0`.

2 The DML compiler | 4 Libraries and Built-ins

4 Libraries and Built-ins

Most standard functionality in Device Modeling Language (DML) is implemented in templates. Built-in templates can be categorized as follows:

- Each object type has a corresponding template which is instantiated for all objects of that type. For instance, the template `register` is automatically instantiated in all registers. All such templates inherit the `object` template, and define the `objtype` parameter to the name of the object type, e.g. `"register"` for registers.
- Some templates primarily provide a standard implementation of some behaviour. For instance, the `uint64_attr` template can be applied on `attribute` objects to make it a simple integer attribute.
- Some templates primarily specify a programming interface, typically by providing an abstract or overrideable method or parameter. Such templates are often named like the provided member. For instance, objects that implement the `init` template provide the abstract method `init`. Interface templates have a number of uses:
  - In some cases, an interface template extends an existing template, altering its default behaviour to make sure its interface method is called. Often, this means that when you provide an implementation of an interface method, you *must* instantiate the corresponding template; otherwise, the method will not be called. For instance, if you implement the `write` method in a register, it is not called by default upon a write access; however, if you instantiate the `write` template in the register, then the register's behaviour is altered to call the `write` method. Thus, in order to provide custom side-effects on a write, you must write something like:

    ```
    register r @ 0 is write { method write(uint64 value) { default();
    log info: "wrote r"; } }
    ```

  - When writing a method or parameter override *inside a template*, you must explicitly instantiate the template you are overriding in order to specify that your declaration takes precedence over the default implementation. If your template is intended for a specific object type, then it is sufficient to override that template, but it is often better to override a more specific template if possible. For instance, the `init_val` parameter belongs to the `init_val` template, which is inherited by all registers. So a template that overrides this parameter may be implemented as follows:

    ```
    template init_to_ten is register {
        param init_val = 10;
    }
    ```

    However, it is even better to only inherit the `init_val` template:

```
template init_to_ten is init_val {
    param init_val = 10;
}
```

The latter improves efficiency and permits `init_to_ten` to also be used on fields.

- Similarly, if you write a template that needs to *access* some member of the object, then it must inherit a template that provides that member. For instance:

```
template log_on_change is (write, get, name) {
    method write(uint64 value) {
        if (this.get() != value) {
            log info: "%s changed!", this.name;
        }
        default();
    }
}
```

Again, it would also work to inherit `register` instead of `get` and `name`, but at the cost of reduced flexibility and efficiency.

4.1 Universal templates

The following templates are applicable to all object kinds:

4.1.1 name

Provides a string parameter `name`, containing the name of the object, as exposed to the end-user. This parameter is typically used in log messages and names of configuration attributes. The name can be overridden in order to hide confidential information from the end-user.

4.1.2 desc

Provides a string parameter `desc`, with a short description in plain text. By convention, this should preferably be a few descriptive words, but may also be a long name. The description can appear during simulation, when inspecting the device, and also serves as a docstring that enriches the DML source code. The parameter's default value is `undefined`. The `desc` parameter has a short-hand syntax described in section Object Declarations.

Also provides a string parameter `shown_desc`, which is the string actually exposed to the end-user during simulation. This parameter defaults to `desc`, and can be overridden in order to hide confidential information from the end-user.

See also template `documentation`.

4.1.3 shown_desc

A subtemplate of `desc` that makes the `shown_desc` parameter a typed parameter. This is inherited by objects that need to access `shown_desc` from the context of a shared method.

4.1.4 documentation

Provides a string parameter `documentation`, with a longer description. The documentation may appear when extracting documentation from the device.

If you have the *Documentation and Packaging* package and intend to generate Simics reference documentation for the device then the `documentation` string must follow the Simics documentation XML format, otherwise you will get a syntax error during the documentation build. See the *Writing Documentation* application note.

Also provides a string parameter `shown_documentation`, defaulting to `documentation`. This parameter is similar to `shown_desc` in the `desc` template, and is mainly a convenient way to suppress documentation.

4.1.5 limitations

Provides a string parameter `limitations`, describing limitations in the implementation of this object. The documentation may appear when extracting documentation from the device.

If you have the *Documentation and Packaging* package and intend to generate Simics reference documentation for the device then the `limitations` string must follow the Simics documentation XML format, otherwise you will get a syntax error during the documentation build. See the *Writing Documentation* application note.

Also provides a string parameter `shown_limitations`, defaulting to `limitations`. This parameter is similar to `shown_desc`, and is mainly a convenient way to suppress documentation.

4.1.6 init

Provides an abstract method `init`, which is called when the device is created, *before* any attributes have been initialized. Typically used to initialize a default value, or to set up data structures.

The method `init` is automatically called on all objects that implement the `init` template. The order in which `init` of objects are called is not defined, except that `init` of a particular object is guaranteed to be called before `init` of any of its parent objects. In particular, `init` of the device object will be called only after all other implementations of `init`.

4.1.7 post_init

Provides an abstract method `post_init`, which is called when the device is created, *after* any attributes have been initialized. Typically used to establish connections to other devices, or to set up data structures that depend on configured attribute values.

The method `post_init` is automatically called on all objects that implement the `post_init` template.

4.1.8 destroy

Provides an abstract method `destroy`, which is called when the device is being deleted. This provides a means to clean up resources associated with the device instance that are not managed by DMLC (and `destroy` should not be used for any other purpose). This typically amounts to invoking `delete` on any device state that has been dynamically allocated using `new`.

The method `destroy` is automatically called on all objects that implement the `destroy` template. The order in which `destroy` of objects are called is not defined, except that `destroy` of a particular object is guaranteed to be called before `destroy` of any of its parent objects. In particular, `destroy` of the device object will be called only after all other implementations of `destroy`.

Usage notes:

- While the device is being deleted, it is not allowed to communicate with any other Simics object (whether that is by accessing `connect`ed devices or by leveraging the Simics API.) This

includes even the device's own clock; don't attempt to post or cancel time/cycle-based events in `destroy()` (including posting events using `after` with a time/cycle delay). The cancellation of such events is handled automatically, before the `destroy()` calls are invoked. The use of `.cancel_after()` inside `destroy()` is tolerated (even if almost always redundant.)

- Exiting Simics, even gracefully, will not perform device deletion and cause `destroy()` to be called on its own. Simics instead simply relies on all resources being released by virtue of the process terminating. This means that side-effects within `destroy()` such as logging will not be visible upon program exit unless deletion is explicitly performed beforehand — hence why `destroy` should *only* be used to ensure resources get cleaned up.

> **Note**
>
> the `destroy` template can't be instantiated for `event` objects. This is because `event` objects, through the `custom_time_event` or `custom_cycle_event` template, may already require defining a method named `destroy` whose purpose is different from the one required by the `destroy` template. To work around this limitation, you may declare a `group` within the `event` object that instantiates the `destroy` template instead.

4.1.9 object

Base template, implemented by all objects. Inherits the templates `name`, `desc`, `documentation` and `limitations`. Provides the following additional parameters, which cannot be overridden:

- `this` (reference): Always refers to the current object, i.e., the nearest enclosing object definition.
- `objtype`: string constant describing the object type, e.g. `"register"`
- `parent` (reference or `undefined`): Always refers to the parent (containing) object. Has the value `undefined` in the `device` object.
- qname: The fully qualified name, including indices, such as `some_bank.r0`. Constructed from the `name` parameter. In the device object, this is equal to the `name` parameter.
- dev: The top-level `device` object
- templates: see Template-Qualified Method Implementation Calls.
- indices: List of local indices for this object. For a non-array object, this is the empty list. In a register array of size N, it is a list with one element, a non-negative integer smaller than N. The parameter is *not* cumulative across the object hierarchy, so for a single field inside a register array, the value is the empty list.
- Each array has an *individual index parameter*, to make it possible to refer to both inner and outer indexes when arrays are nested (cf. the `indices` parameter, above). The parameter name is specified in the array declaration; for instance, the declaration `register regs[i < 4][j < 11];` defines two index parameters, `i` and `j`. In this case, the `indices` parameter is `[i, j]`.

The `object` template provides the non-overridable method `cancel_after()`, which cancels all pending events posted using `after` which are associated with the object (any events associated with subobjects are unaffected).

There are no other methods common for all objects, but the methods `init`, `post_init`, and `destroy` are automatically called on all objects that implement the `init`, `post_init`, and `destroy` template, respectively.

4.2 Device objects

The top-level scope of a DML file defines the *device object*, defined by the template `device`. This template inherits the `init`, `post_init`, and `destroy` templates.

The `device` template contains the following methods:

- `init()`: Called when the device object is loaded, but before its configuration-object attributes have been initialized.
- `post_init()`: Called when the device object is loaded, *after* its configuration-object attributes have been initialized.
- `destroy()`: Called when the device object is being deleted.

The `device` template contains the following parameters:

- `classname` *[string]*: The name of the Simics configuration object class defined by the device model. Defaults to the name of the device object.
- `register_size` *[integer | undefined]*: The default size (width) in bytes used for `register` objects; inherited by `bank` objects. The default value is `undefined`.
- `byte_order` *[string]*: The default byte order used when accessing registers wider than a single byte; inherited by `bank` objects. Allowed values are `"little-endian"` and `"big-endian"`. The default value is `"little-endian"`.
- `be_bitorder` *[bool]*: The default value of the `be_bitorder` in banks. The default value is `true` if the DML file containing the `device` statement declares `bitorder be;`, and `false` otherwise.
- `use_io_memory` *[bool]*: The default value of the `use_io_memory` parameter in banks. The current default value is `true`, but in future Simics versions it will be `false`.
- `obj` *[conf_object_t]*: A pointer to the `conf_object_t` C struct that Simics associates with this device instance
- `simics_api_version` *[string]*: The Simics API version used when building this device, as specified by the `--simics-api` command-line argument; e.g. `"6"` for the Simics 6 API.

4.3 Group objects

Group objects are generic container objects, used to group other objects. They can appear anywhere in the object hierarchy, but some object types (currently `implement` and `interface`) may not have a group as parent.

The `group` template contains no particular methods or parameters other than what is inherited from the `object` template.

You may not declare any `bank`, `port` or `subdevice` underneath any group named "bank" or "port". This is to avoid namespace clashes in Simics.

4.4 Attribute objects

The `attribute` template contains the following methods:

get() -> (attr_value_t)

Abstract method. Returns the value of the attribute.

set(attr_value_t value) throws

Abstract method. Sets the value of the attribute. If the provided value is not allowed, use a `throw` statement to signal the error.

get_attribute -> (attr_value_t), set_attribute(attr_value_t value) -> (set_error_t)

Not intended to be used directly. Called by Simics for reading and writing the attribute value. Calls the `get` and `set` methods.

The `attribute` template contains the following parameters:

type [string | undefined]

A Simics configuration-object attribute type description string, such as `"i"` or `"[s*]"`, specifying the type of the attribute. (See the documentation of `SIM_register_typed_attribute` in the *Model Builder Reference Manual* for details.) For simple types this can easily be set by standard attribute templates.

configuration ["required" | "optional" | "pseudo" | "none"]

Specifies how Simics treats the attribute. The default value is `"optional"`. A *required* attribute must be initialized to a value when the object is created, while an *optional* attribute can be left undefined. In both cases, the value is saved when a checkpoint is created. For a *pseudo*-attribute, the value is *not* saved when a checkpoint is created (and it is not required to be initialized upon object creation). Setting the value to `"none"` suppresses creation of the attribute. This is seldom useful in attribute objects, but can be used in related object types like `register` to suppress checkpointing.

persistent [bool]

If this parameter is `true`, the attribute will be treated as persistent, which means that its value will be saved when using the `save-persistent-state` command. The default value is `false`.

readable [bool]

If false, the attribute cannot be read. This can only be used if the `configuration` parameter is set to `"pseudo"`. Normally set by the `write_only_attr` template.

writable [bool]

If false, the attribute cannot be written. This should normally be set by instantiating the `read_only_attr` template, and requires that the `configuration` parameter is `"pseudo"`. Normally set by the `read_only_attr` template.

internal [bool]

If this parameter is `true`, the attribute will be treated as internal, meaning that it will be excluded from documentation. The default value is `true` if the `documentation` and `desc` parameters are undefined, and `false` otherwise.

4.5 Attribute templates

Four templates are used to create a simple checkpointable attribute with standard types. Each store the attribute value in a member `val`, provide default implementations of methods `get` and `set` according to the type, and provide a default implementation of `init` that initializes `val` using the `init_val` parameter also provided by the template (whose default definition simply zero-initializes `val`.) These four templates are:

bool_attr

boolean-valued attribute, `val` has type `bool`

int64_attr

integer-valued attribute, `val` has type `int64`

uint64_attr

integer-valued attribute, `val` has type `uint64`

double_attr

floating-point-valued attribute, `val` has type `double`

In addition, three templates can be used to define a pseudo attribute, and cannot be used with the above templates:

pseudo_attr

Pseudo attribute. Will not be checkpointed. Methods `get` and `set` are abstract.

read_only_attr

Pseudo attribute that cannot be written. Method `get` is abstract.

write_only_attr

Pseudo attribute that cannot be read. Method `set` is abstract

These templates are not compatible with the `bool_attr`, `uint64_attr`, `int64_attr`, or `float_attr` templates.

4.6 Connect objects

The `connect` template contains the following methods:

validate(conf_object_t *obj) -> (bool)

Called when Simics attempts to assign a new target object. If the return value is `false`, the attempted connection will fail, and any existing connection will be kept. The default is to return `true`.

If connecting to a port interface (rather than a port object), then a session variable `port`, of type `char *`, is set to the new port name during the `validate` call. This will be removed in future versions of Simics.

set(conf_object_t *obj)

Called after validation, to assign a new target object. Can be overridden to add side-effects before or after the assignment

get_attribute -> (attr_value_t), set_attribute(attr_value_t value) -> (set_error_t)

Internal, not intended to be used directly. Called by Simics for accessing the attribute value.

The `connect` template contains the following parameters:

configuration `["required" | "optional" | "pseudo" | "none"]`

Specifies how Simics treats the automatically created attribute corresponding to the connect object. The default value is `"optional"`.

The attribute can be set to a nil value only if this parameter is `"optional"` or `"pseudo"`. In an array of connects, this applies element-wise.

internal [bool]

Specifies whether the attribute should be internal.

4.7 Connect templates

A `connect` object can instantiate the template `init_as_subobj`. This causes the connect to automatically set itself to an automatically created object. This can be used to create a private helper object.

The `init_as_subobj` template accepts one parameter, `classname`. This defines the class of the automatically created object.

> **Note**
>
> The subobject class defined by the `classname` parameter is looked up using `SIM_get_class` while the module of the device class is being loaded. This may cause problems if the subobject class is defined within the same module as the device class: if the device class is defined before the subobject class, then the subobject class will not yet be defined, and the `SIM_get_class` call will fail. This problem can be resolved by moving the subobject class to a separate module.

The template also overrides the `configuration` parameter to `"none"` by default, which makes the connect invisible to the end-user.

The `init_as_subobj` inherits the `init` and `connect` templates.

4.8 Interface objects

The `interface` template contains one parameter, `required`. Defaults to `true`. If overridden to `false`, the interface is optional and the parent `connect` object can connect to an object that does not implement the interface.

The template provides a session variable `val` of type `const void *`. It points to the Simics interface struct of the currently connected object. If the interface is optional, then the variable can be compared to NULL to check whether the currently connected object implements the interface.

4.9 Port objects

The `port` template exposes one parameter, `obj`. When compiling with Simics API 5 or earlier, evaluates to dev.obj. When compiling with Simics API 6 or newer, evaluates to the `conf_object_t *` of the port's port object.

4.10 Subdevice objects

The `subdevice` template exposes one parameter, `obj`, which evaluates to the `conf_object_t *` of the Simics object that represents the subdevice.

4.11 Implement objects

The `implement` template provides no particular parameters or methods.

4.12 Implement templates

There is a single template for `implement` objects, namely `bank_io_memory`. The template can be instantiated when implementing the `io_memory` interface, and redirects the access to a bank, specified by the `bank` parameter.

Bank objects contain an implementation of `io_memory` that inherits this template.

4.13 Bank objects

In addition to the `object` template, the `bank` template also inherits the `shown_desc` template.

The `bank` template contains the following methods:

io_memory_access(generic_transaction_t *memop, uint64 offset, void *aux) -> (bool)

> Entry point for an access based on `generic_transaction_t`. Extracts all needed info from `memop`, calls appropriate memop-free methods, updates the `memop` parameter accordingly, and returns `true` if the access succeeded. The `offset` parameter is the offset of the access relative to the bank. The `aux` parameter is NULL by default, and is passed on to bank methods. In order to pass additional information on the access down to register and field methods, one can override `io_memory_access`, decode needed information from the incoming memop, and call `default` with the extracted information in the `aux` argument.

transaction_access(transaction_t *t, uint64 offset, void *aux) -> (exception_type_t)

> Entry point for an access based on the `transaction` interface. Extracts all needed info from `t`, calls appropriate access methods (`read`, `write`, `get`, `set`), and updates the `t` parameter accordingly. Returns `Sim_PE_No_Exception` if the access succeeded, and `Sim_PE_IO_Not_Taken` otherwise. The `offset` parameter is the offset of the access relative to the bank. The `aux` parameter is NULL by default, and is passed on to bank methods. In order to pass additional information on the access down to register and field methods, one can override `transaction_access`, decode needed information from the incoming transaction, and call `default` with the extracted information in the `aux` argument. Accesses that bigger than 8 bytes are split into smaller sized chunks before being completed, the exact details of which are *undefined*.

write(uint64 offset, uint64 value, uint64 enabled_bytes, void *aux) throws

> A write operation at the given offset. Throwing an exception makes the access fail, and is typically signaled for writes outside registers. The default behavior is to forward the access to registers, as follows:
>
> 1. Deduce which registers are hit by the access. The `offset` and `size` parameters of each register object is used to deduce whether the register is covered by the access. A register which is only partially covered will be considered hit if the bank parameter `partial` is true, and a register which does not fully cover the access is considered hit if the bank parameter `overlapping` is set.
>
> 2. If any portion of the access is not covered by a hit register, then the `unmapped_write` method is called with a bit pattern showing what parts of the access are unmapped. Any exception thrown by unmapped_write is propagated, causing the access to fail.

3. The `write_register` method is called in all hit registers, starting with the register at the lowest offset.

unmapped_write(uint64 offset, uint64 value, uint64 bits, void *aux) throws

If an access is not fully covered by registers, then this method is called before the access is performed. Throwing an exception aborts the entire access. *bits* is a bit pattern showing which bits are affected; in the lowest 'size' bits, each 0xff byte represents the position of one unmapped byte in the access. The *value* parameter contains the originally written value, including parts that are mapped to registers. Both *bits* and *value* are expressed in the host's endianness. The default behavior is to log a `spec-viol` message on level 1, and throw an exception.

read(uint64 offset, uint64 enabled_bytes, void *aux) -> (uint64 value) throws

A read operation at the given offset. The access is decomposed in the same way as in `write`. If there are unmapped portions of the access, `unmapped_read` is invoked, possibly aborting the operation. The return value is composed by the results from calling `read_register` in hit registers, combined with the result of `unmapped_read` if the access is not fully mapped.

unmapped_read(uint64 offset, uint64 bits, void *aux) throws

Like `unmapped_write` but for reads. The default implementation unconditionally throws an exception.

The `bank` template contains the following parameters:

mappable [boolean]

Controls whether a bank is visible as an interface port for the `io_memory` interface, which makes it mappable in a memory space. This defaults to true.

overlapping [bool]

Specifies whether this bank allows accesses that cover more than one register. (This translates to one or more, possibly partial, accesses to adjacent registers.) Defaults to `true`. This parameter must have the same value among all elements in a bank array object, i.e., it must not depend on the index of the bank.

partial [bool]

Specifies whether this bank allows accesses that cover only parts of a register. A partial read will read the touched register fields (or the whole register if there are no fields) and extract the bits covered by the read. A partial write will call the `get` method on the touched register fields (or the whole register when there are no fields) and replace the written bits with the written value and then call the `write` method on the fields (or the register) with the merged value. Defaults to `true`. This parameter must have the same value among all elements in a bank array object, i.e., it must not depend on the index of the bank.

register_size [integer | undefined]

Inherited from the `device` object; provides the default value for the `size` parameter of `register` objects.

byte_order [string]

Specifies the byte order used when accessing registers wider than a single byte; inherited from `device` objects. Allowed values are `"little-endian"` and `"big-endian"`. This

parameter must have the same value among all elements in a bank array object, i.e., it must not depend on the index of the bank.

be_bitorder [bool]

Controls the preferred bit ordering of registers within this bank. Whenever the register is presented to the user as a bitfield, bit 0 refers to the least significant bit if the parameter is `false` (the default), and to the most significant bit if the parameter is `true`. The parameter is only a presentation hint and does not affect the model's behaviour. The parameter is technically unrelated to the top-level `bitorder` declaration, though in most cases the two should match.

use_io_memory [bool]

If `true`, this bank is exposed using the legacy `io_memory` interface. In this case, the `io_memory_access` method can be called and overridden, but the `transaction_access` method can not.

If `false`, this bank is exposed using the `transaction` interface. In this case, the `transaction_access` method can be called and overridden, but the `io_memory_access` method can not.

The default is inherited from `dev.use_io_memory`.

obj [conf_object_t *]

When compiling with Simics API 5 or earlier, evaluates to dev.obj. When compiling with Simics API 6 or newer, evaluates to the bank's port object.

4.14 Register objects

In addition to `object`, the `register` template inherits the templates `get`, `set`, `shown_desc`, `read_register`, `write_register`, and `init_val`.

The `register` template contains the following parameters:

val [integer]

The contents of the register. Manipulating `val` is a simpler, but less safe alternative to using `get_val()` and `set_val()` — unlike `set_val()`, it is *undefined behavior* to write a value to `val` larger than what the register can hold.

size [integer]

The size (width) of the register, in bytes. This parameter can also be specified using the "`size` *n*" short-hand syntax for register objects. The default value is provided by the `register_size` parameter of the enclosing `bank` object.

bitsize [integer]

The size (width) of the register, in bits. This is equivalent to the value of the `size` parameter multiplied by 8, and cannot be overridden.

offset [integer]

The address offset of the register, in bytes relative to the start address of the bank that contains it. This parameter can also be specified using the "`@` *n*" short-hand syntax for register objects. There is no default value. If the register inherits the `unmapped` template, the

register is not mapped to an address. This parameter must have the same value among all elements in a bank array object, i.e., it must not depend on the index of the bank.

fields [list of references]

A list of references to all the `field` objects of a register object.

init_val [integer]

The value used by the default implementation of the `init` method, when the device is instantiated. The value is also used by the default implementations of hard reset, soft reset and power-on reset. Defaults to 0.

configuration `["required" | "optional" | "pseudo" | "none"]`

Specifies how Simics treats the automatically created [attribute] (#attribute-objects) corresponding to the register. The default value is `"optional"`.

persistent [bool]

Specifies whether the register attribute should be persistent.

internal [bool]

Specifies whether the register attribute should be internal, default is `true`.

The `register` template provides the following overridable methods:

`read_unmapped_bits(uint64 unmapped_enabled_bits, void *aux) -> (uint64)`

`read_unmapped_bits` is used by the default implementation of `read_register` to read the bits not covered by fields from a register, possibly with side-effects.

The default implementation of `read_unmapped_bits` acts similarly to if the unmapped regions of the register were covered by fields, the `value` of the register is masked by `unmapped_enabled_bits` and returned.

`write_unmapped_bits(uint64 val, uint64 enabled_bits, void *aux).`

The `write_unmapped_bits` method is called from the default implementation of the `write_register` method when `enabled_bytes` specifies bytes not completely covered by field in the register (and the register has at least one field). `unmapped_enabled_bits` is defined as in the `read_unmapped_bits` method. `val` is the `val` argument passed to `write_register`, masked with the bits not covered by fields.

Default behaviour of `write_unmapped_bits` is to compare the `unmapped_enabled_bits` in `value` to those in the register `val`; if they do not match, a message of type `spec-viol` is logged for each bitrange that does not match, but `val` is not modified.

4.15 Field objects

In addition to `object`, the `field` template inherits the templates `init_val` and `shown_desc`.

The template inherits methods `get`, `set` and `init`, and the parameter `init_val`

The `field` template contains the following parameters:

val [integer]

The bitslice of the parent register corresponding to the field. Manipulating `val` is a simpler alternative to using `get_val()` and `set_val()`, while being just as safe (unlike with register objects). Unlike `get_val()` and `set_val()`, `val` is not a member of the `field` template type, and thus can't be used in certain contexts.

reg [reference]

Always refers to the containing register object.

lsb [integer]

Required parameter. The bit number in the containing register of the field's least significant bit. Represented in little-endian bit order, regardless of `bitorder` declarations. The preferred way of defining this parameter is to use the "`[highbit:lowbit]`" short-hand syntax for field ranges, whose interpretation *is* dependent on the `bitorder` declaration of the file. Care must be taken when referring to this parameter in a big-endian bit numbering system - if possible, put such code in a separate file that uses little-endian bit order interpretation.

msb [integer]

Required parameter. The bit number in the containing register of the field's most significant bit. Represented in little-endian bit order. See `lsb` for details.

bitsize [integer]

The size (width) of the field, in bits. This is automatically set from the `lsb` and `msb` parameters and cannot be overridden.

init_val [integer]

The value used by the default implementation of the `init` method, when the device is instantiated. The value is also used by the default implementations of hard reset, soft reset and power-on reset. Defaults to 0.

4.16 Templates for registers and fields

This section lists templates that are specific for `register` and `field` objects.

All templates (except `read_register` and `write_register`) are applicable to both registers and fields. When writing a template that is applicable to both registers and fields, one should normally inherit one or more of the `read`, `write`, `get` and `set` methods.

Some methods have an argument `void *aux`. By default, that argument is NULL. The value can be overridden to carry arbitrary extra information about the access; this is done by overriding the `io_memory_access` method in the parent bank.

4.16.1 get_val

Provides a single non-overrideable method `get_val() -> (uint64)`. In a register, it returns the value of the .val member; in a field, it returns the bits of the parent register's `val` member that are covered by the field.

`get_val` is very similar to `get`; the difference is that `get_val` is unaffected if `get` is overridden. Thus, `get_val` is slightly more efficient, at the cost of flexibility. It is generally advisable to use `get`.

4.16.2 set_val

Provides a single non-overrideable method `set_val(uint64)`. In a register, it sets the value of the .val member; in a field, it sets the bits in the parent register's `val` member that are covered by the

field.

`set_val` is very similar to (`set`)[#set); the difference is that `set_val` is unaffected if `set` is overridden. Thus, `set_val` is slightly more efficient, at the cost of flexibility. It is generally advisable to use `set`.

4.16.3 get

Extends the `get_val` template. Provides a single overrideable method `get() -> (uint64)`, which retrieves the register's value, without side-effects, used for checkpointing and inspection. The default is to retrieve the value using the `get_val` method.

In a field, this template must be explicitly instantiated in order for an override to take effect. Note however that field objects do provide a callable default implementation of the method.

4.16.4 set

Extends the `set_val` template. Provides a single overrideable method `set(uint64)`, which modifies the register's value, without triggering other side-effects. Used for checkpointing and inspection. The default is to set the value using the `set_val` method.

In a field, this template must be explicitly instantiated in order for an override to take effect. Note however that field objects do provide a callable default implementation of the method.

4.16.5 read_register

Implemented only by registers, not applicable to fields.

Provides a single abstract method `read_register(uint64 enabled_bytes, void *aux) -> (uint64)`.

The method reads from a register, possibly with side-effects. The returned value is represented in the host's native endianness. The `enabled_bytes` argument defines which bytes of the register is accessed, as a bitmask; each byte of the returned value has significance only if the corresponding byte in `enabled_bytes` is 0xff. If the access covers more than one register, then the parts of `enabled_bytes` that correspond to other registers are still zero.

Register objects provide a default implementation of `read_register`. The implementation invokes the `read_field` method of all sub-fields at least partially covered by `enabled_bytes`, in order from least to most significant bit. Bits not covered by fields are retrieved by calling the `read_unmapped_bits`, with `unmapped_enabled_bits` set as the `enabled_bytes` not covered by fields. If a register implements no fields, then the `read_unmapped_bits` is not called by default.

If a register inherits the `read_field` or `read` templates, then that template takes precedence over `read_register`, and the register's read behaviour is specified by the `read_field` or `read` method.

4.16.6 write_register

Implemented only by registers, not applicable to fields.

Provides a single abstract method: `write_register(uint64 value, uint64 enabled_bytes, void *aux)`.

The method writes to the register, possibly with side-effects. The `enabled_bytes` parameter is defined as in the `read_register` method.

Register objects provide a default implementation of write_register. The default behaviour depends on whether the register has fields:

- If the register has no fields, then the default behaviour is to set the register's `val` member to the new value, using the `set` method.
- If the register has fields, then the default behavior is to invoke the `write_field` method of all sub-fields covered at least partially by `enabled_bytes`, in order from least to most significant bit. Then `write_unmapped_bits` is called with the enabled bits that were not covered by fields.

  If a register inherits the `write` or `write_field` template, then that template takes precedence over `write_register`, and the register's write behaviour is specified by the `write` (or `write_field`) method.

4.16.7 read_field

Provides a single abstract method `read_field(uint64 enabled_bits, void *aux) -> (uint64)`.

The method reads from a field or register, possibly with side-effects. The returned value is represented in the host's native endianness. The `enabled_bits` argument defines which bits of the register is accessed, as a bitmask; each bit of the returned value has significance only if the corresponding bit in `enabled_bits` is 1. If the access covers more than one field, then the parts of `enabled_bits` that correspond to other fields are still zero.

The `read_field` template is *not* implemented by fields or registers by default, and must be explicitly instantiated in order for a method override to have effect. `read_field` is the interface used for access by registers; in most cases, it is easier to express read operations using the `read` template.

Note that instantiating `read_field` on a register means that register reads behave as if the register consists of one single field; a read access will ignore any actual field subobjects in the register.

4.16.8 write_field

Provides a single abstract method `write_field(uint64 value, uint64 enabled_bits, void *aux)`.

The method writes to a field or register, possibly with side-effects. The value is represented in the host's native endianness. The `enabled_bits` argument is defined as in the `read_field` method.

The `write_field` template is *not* implemented by fields or registers by default, and must be explicitly instantiated in order for a method override to have effect. `write_field` is the interface used for access by registers; in most cases, it is easier to express write operations using the `write` template.

Note that instantiating `write_field` on a register means that register writes behave as if the register consists of one single field; a write access will ignore any actual field subobjects in the register. This is often useful in read-only registers, as it allows reads to propagate to fields, while a violating write can be handled centrally for the whole register.

4.16.9 read

Extends templates `read_field` and `get_val`.

Provides a single overrideable method `read() -> (uint64)`.

The method reads from a field or register, possibly with side-effects. The returned value is represented in the host's native endianness. The default behaviour is to retrieve the value using the `get` method.

The read template is *not* implemented by fields or registers by default, and must be explicitly instantiated in order for a method override to have effect.

Note that instantiating read on a register means that register reads behave as if the register consists of one single field; a read access will ignore any actual field subobjects in the register.

4.16.10 write

Extends templates write_field, get_val and set_val.

Provides a single overrideable method write(uint64).

The method writes to a field or register, possibly with side-effects. The value is represented in the host's native endianness. The default behaviour is to set the value using the set method.

The write template is *not* implemented by fields or registers by default, and must be explicitly instantiated in order for a method override to have effect.

Note that instantiating write on a register means that register writes behave as if the register consists of one single field; a write access will ignore any actual field subobjects in the register.

4.16.11 init_val

Extends the init template.

Provides a parameter init_val : uint64, defining the initial value of the register's val member when the object is created. In a field, defines the initial value of the bits of val that are covered by this field. The parameter is also used by default reset methods.

The template is inherited by both registers and fields. The value is 0 by default. In a register with fields, parameter overrides are permitted both in the register and in the field objects:

- if the parameter is overridden *only* in the register, then this defines the full value of the register.
- if the parameter is overridden both in the register and in some fields, then the field overrides take precedence and define the value of the bits covered by the field

On a technical level, the default value of init_val in a field is the field's corresponding bits in the parent register's init_val; furthermore, the init_val template provides a default implementation of the init method which in register objects sets bits in val not covered by fields, and in field objects sets corresponding bits of the parent register's val member.

4.17 Event objects

The event template contains little functionality in itself; it requires one of six predefined templates simple_time_event, simple_cycle_event, uint64_time_event, uint64_cycle_event, custom_time_event, and custom_cycle_event to be instantiated. These templates expose the methods event and post, and possibly others.

In addition to the object template, the event template inherits the shown_desc template.

4.18 Event templates

Each event object is required to instantiate one of six predefined templates: simple_time_event, simple_cycle_event, uint64_time_event, uint64_cycle_event, custom_time_event, and custom_cycle_event. These are defined as follows:

- The `simple_*_event` templates are used for events that carry no data
- The `uint64_*_event` templates are used for events parameterized with a single 64-bit integer value
- The `custom_*_event` templates are used for events that carry more complex data; the user must supply explicit serialization and deserialization methods.
- In the `*_time_event` templates, time is provided in seconds, as a floating-point number
- In the `*_cycle_event` templates, time is provided in cycles, as a 64-bit integer

The following methods are defined by all six templates:

event(), event(uint64 data), event(void *data)

> Abstract method, called when the event is triggered. When one of the `custom_*_event` templates is used, the `event` method is responsible for deallocating the data.

post(time), post(time, uint64 data), post(time, void *data)

> Non-overrideable method. Posts the event on the associated queue of the device. The time argument is specified in cycles or seconds, depending on which template was instantiated. The event will be triggered after the specified amount of time has elapsed. The data parameter in a `uint64` or `custom` event will be passed on to the event() method.

The following methods are specific to the `simple_time_event`, `simple_cycle_event`, `uint64_time_event` and `uint64_cycle_event` templates:

remove(), remove(uint64 data)

> Removes all events of this type with matching data from the queue.

posted() -> (bool), posted(uint64 value) -> (bool)

> Returns `true` if the event is in the queue, and `false` otherwise.

next(), next(uint64 data) -> (double or cycles_t)

> Returns the time to the next occurrence of the event in the queue (relative to the current time), in cycles or seconds depending on which event template was instantiated. If there is no such event in the queue, a negative value is returned.

The following methods are specific to the `custom_time_event` and `custom_cycle_event` templates:

get_event_info(void *data) -> (attr_value_t)

> This method is called once for each pending event instance when saving a checkpoint. It should create an attribute value that can be used to restore the event. The `data` parameter is the user data provided in the `post` call. The default implementation always returns a nil value.

set_event_info(attr_value_t info) -> (void *)

> This method is used to restore event information when loading a checkpoint. It should use the attribute value to create a user data pointer, as if it had been provided in a `post`. The default implementation only checks that the checkpointed information is nil.

destroy(void *data)

This method is called on any posted events when the device object is deleted.

If memory was allocated for the `data` argument to `post`, then `destroy` should free this memory.

The `destroy` method is *not* called automatically when an event is triggered; therefore, the method should typically also be called explicitly from the `event` method.

3 Device Modeling Language, version 1.4 | 5 Standard Templates

5 Standard Templates

This chapter describes the standard templates included the Device Modeling Language (DML) library. The templates can be used for both registers and fields. The templates can be accessed after importing `utility.dml`.

The most common device register functionality is included in the standard templates.

Note that many standard templates has the same functionality and only differ by name or log-messages printed when writing or reading them. The name of the template help developers to get a quick overview of the device functionality. An example are the *undocumented* and *reserved* templates. Both have the same functionality. However, the *undocumented* template hints that something in the device documentation is unclear or missing, and the *reserved* template that the register or field should not be used by software.

The sub-sections use *object* as a combined name for registers and fields. The sub-sections refers to software and hardware reads and writes. Software reads and writes are defined as accesses using the `io_memory` interface (write/reads to memory/io mapped device). Software reads and writes use the DML built-in read and write methods. Hardware read writes are defined as accesses using Simics configuration attributes, using the DML built-in set and get methods. Device code can still modify a register or device even if hardware modification is prohibited.

5.1 Templates for reset

Reset behavior can vary quite a bit between different devices. DML has no built-in support for handling reset, but there are standard templates in `utility.dml` to cover some common reset mechanisms.

There are three standard reset types:

Power-on reset

> The reset that happens when power is first supplied to a device.

Hard reset

> Typically triggered by a physical hard reset line that can be controlled from various sources, such as a watchdog timer or a physical button. Often the same as a power-on reset.

Soft reset

> This usually refers to a reset induced by software, e.g. by a register write. Not all devices have a soft reset, and some systems may support more than one type of soft reset.

Usually, the effect of a reset is that all registers are restored to some pre-defined value.

In DML, the reset types can be enabled by instantiating the templates `poreset`, `hreset` and `sreset`, respectively. These will define a port with the corresponding upper-case name (`POWER`, `HRESET`, `SRESET`), which implements the `signal` interface, triggering the corresponding reset type on raising edge. This happens by invoking a corresponding method (`power_on_reset`, `hard_reset`

and soft_reset, respectively), in all objects implementing a given template (power_on_reset, hard_reset and soft_reset, respectively). The default is that all registers and fields implement these templates, with the default behavior being to restore to the value of the init_val parameter. The methods signal_raise and signal_lower can be overridden to add additional side effects upon device reset. One example would be to track if the reset signal is asserted to prevent interaction with the device during reset.

The default implementation of all reset methods recursively calls the corresponding reset method in all sub-objects. Thus, if a reset method is overridden in an object without calling default(), then reset is effectively suppressed in all sub-objects.

The two most common overrides of reset behavior are:

- to reset to a different value. In the general case, this can be done with an explicit method override. In the case of soft reset, you can also use the standard template soft_reset_val which allows you to configure the reset value with a parameter soft_reset_val.
- to suppress reset. This can be done with a standard template: sticky suppresses soft reset only, while no_reset suppresses all resets.

  It is quite common that hard reset and power-on reset behave identically. In this case, we recommend that only a HRESET port is created; this way, the presence of a POWER port is an indication that the device actually provides a distinct behavior on power cycling.

  There are some less common reset use cases:

- In some devices, the standard reset types may not map well to the hardware, typically because there may be more than one reset type that could be viewed as a soft reset. In this case, we recommend that SRESET is replaced with device-specific port names that map better to the hardware, but that the POWER and HRESET port names are preserved if they are unambiguous.
- In some cases, it is desirable to accurately simulate how a device acts in a powered-off state. This would typically mean that the device does not react to other stimuli until power is turned on again.

  The recommended way to simulate a powered-off state, is to let the high signal level of the POWER port represent that the device has power, and react as if power is down while the signal is low. Furthermore, the device is reset when the POWER signal is lowered.

  If this scheme is used by a device, then a device will be considered turned off after instantiation, so the POWER signal must be raised explicitly before the device can function normally.

  Thus, there are two rather different ways to handle devices that have a POWER port:

  - POWER can be treated as a pure reset port, which stays low for most of the time and is rapidly raised and lowered to mark a power-on reset. This is the most convenient way to provide a power signal, but it only works if the device only uses POWER for power-on reset. If the device models power supply accurately, then it will not function as expected, because it will consider power to be off.
  - POWER can be accurately treated as a power supply, meaning that the signal is raised before simulation starts, and lowered when power goes down. A reset is triggered by a lower followed by a raise. This approach is less convenient to implement, but more correct: The device will function correctly both if the device does an accurate power supply simulation, and if it only uses POWER for power-on reset.

5.1.1 power_on_reset, hard_reset, soft_reset

5.1.1.1 Description

Implemented on any object to get a callback on the corresponding reset event. Automatically implemented by registers and fields.

5.1.1.2 Related Templates

`poreset`, `hreset`, `sreset`

5.1.2 poreset, hreset, sreset

5.1.2.1 Description

Implemented on the top level to get standard reset behaviour, for power-on reset, hard reset and soft reset, respectively.

5.1.2.2 Related Templates

`power_on_reset`, `hard_reset`, `soft_reset`

5.2 Templates for registers and fields

The following templates can be applied to both registers and fields. Most of them affect either the write or read operation; if applied on a register it will disregard fields. For instance, when applying the `read_unimpl` template on a register with fields, then the read will ignore any implementations of `read` or `read_field` in fields, and return the current register value (through `get`), ignoring any `read` overrides in fields. However, writes will still propagate to the fields.

5.2.1 soft_reset_val

5.2.1.1 Description

Implemented on a register or field. Upon soft reset, the reset value is defined by the required `soft_reset_val` parameter, instead of the default `init_val`.

5.2.1.2 Related Templates

`soft_reset`

5.2.2 ignore_write

5.2.2.1 Description

Writes are ignored. This template might also be useful for read-only fields inside an otherwise writable register. See the documentation for the `read_only` template for more information.

5.2.3 read_zero

5.2.3.1 Description

Reads return 0, regardless of register/field value. Writes are unaffected by this template.

5.2.3.2 Related Templates

`read_constant`

5.2.4 read_only

5.2.4.1 Description

The object value is read-only for software, the object value can be modified by hardware.

5.2.4.2 Log Output

First software write results in a spec_violation log-message on log-level 1, remaining writes on log-level 2. Fields will only log if the written value is different from the old value.

If the register containing the read-only field also contains writable fields, it may be better to use the `ignore_write` template instead, since software often do not care about what gets written to a

read-only field, causing unnecessary logging.

5.2.5 write_only

5.2.5.1 Description

The register value can be modified by software but can't be read back, reads return 0. Only for use on registers; use `read_zero` for write-only fields.

5.2.5.2 Log Output

The first time the object is read there is a spec_violation log-message on log-level 1, remaining reads on log-level 2.

5.2.6 write_1_clears

5.2.6.1 Description

Software can only clear bits. This feature is often used when hardware sets bits and software clears them to acknowledge. Software write 1's to clear bits. The new object value is a bitwise AND of the old object value and the bitwise complement of the value written by software.

5.2.7 clear_on_read

5.2.7.1 Description

Software reads return the object value. The object value is then reset to 0 as a side-effect of the read.

5.2.8 write_1_only

5.2.8.1 Description

Software can only set bits to 1. The new object value is the bitwise OR of the old object value and the value written by software.

5.2.8.2 Related Templates

`write_0_only`

5.2.9 write_0_only

5.2.9.1 Description

Software can only set bits to 0. The new object value is the bitwise AND of the old object value and the value written by software.

5.2.9.2 Related Templates

`write_1_only`

5.2.10 read_constant

5.2.10.1 Description

Reads return a constant value.

Writes are unaffected by this template. The read value is unaffected by the value of the register or field.

The template is intended for registers or fields that have a stored value that is affected by writes, but where reads disregard the stored value and return a constant value. The attribute for the register will reflect the stored value, not the value that is returned by read operations. For constant registers or fields that do not store a value, use the `constant` template instead.

5.2.10.2 Parameters

`read_val`: the constant value

5.2.10.3 Related Templates
`constant`, `silent_constant`, `read_zero`

5.2.11 constant

5.2.11.1 Description
Writes are forbidden and have no effect.

The object still has backing storage, which affects the value being read. Thus, an end-user can modify the constant value by writing to the register's attribute. Such tweaks will survive a reset.

Using the `constant` template marks that the object is intended to stay constant, so the model should not update the register value, and not override the `read` method. Use the template `read_only` if that is desired.

5.2.11.2 Log Output
First write to register or field (if field value is not equal to write value) results in a spec_violation log-message on log-level 1, remaining writes on log-level 2.

5.2.11.3 Parameters
init_val: the constant value

5.2.11.4 Related Templates
`read_constant`, `silent_constant`, `read_only`

5.2.12 silent_constant

5.2.12.1 Description
The object value will remain constant. Writes are ignored and do not update the object value.

The end-user can tweak the constant value; any tweaks will survive a reset.

By convention, the object value should not be modified by the model; if that behaviour is wanted, use the `ignore_write` template instead.

5.2.12.2 Parameters
init_val: the constant value

5.2.12.3 Related Templates
`constant`, `read_constant`

5.2.13 zeros

5.2.13.1 Description
The object value is constant 0. Software writes are forbidden and do not update the object value.

5.2.13.2 Log Output
First software write to register or field (if field value is not equal to write value) results in a spec_violation log-message on log-level 1, remaining writes on log-level 2.

5.2.14 ones

5.2.14.1 Description
The object is constant all 1's. Software writes do not update the object value. The object value is all 1's.

5.2.14.2 Log Output
First software write to register or field (if field value is not equal to write value) results in a spec_violation log-message on log-level 1, remaining writes on log-level 2.

5.2.15 ignore

5.2.15.1 Description

The object's functionality is unimportant. Reads return 0. Writes are ignored.

5.2.16 reserved

5.2.16.1 Description

The object is marked reserved and should not be used by software. Writes update the object value. Reads return the object value.

5.2.16.2 Log Output

First software write to register or field (if field value is not equal to write value) results in a `spec-viol` log-message on log-level 2. No logs on subsequent writes.

5.2.17 unimpl

5.2.17.1 Description

The object functionality is unimplemented. Warn when software is using the object. Writes and reads are implemented as default writes and reads.

5.2.17.2 Log Output

First read from a register results in an unimplemented log-message on log-level 1, remaining reads on log-level 3. Reads from a field does not result in a log-message. First write to a register results in an unimplemented log-message on log-level 1, remaining writes on log-level 3. First write to a field (if field value is not equal to write value) results in an unimplemented log-message on log-level 1, remaining writes on log-level 3.

5.2.17.3 Related Templates

`read_unimpl`, `write_unimpl`, `silent_unimpl`, `design_limitation`

5.2.18 read_unimpl

5.2.18.1 Description

The object functionality associated to a read access is unimplemented. Write access is using default implementation and can be overridden (for instance by the `read_only` template).

5.2.18.2 Log Output

First software read to a register results in an unimplemented log-message on log-level 1, remaining reads on log-level 3. Software reads to fields does not result in a log-message.

5.2.18.3 Related Templates

`unimpl`, `write_unimpl`, `silent_unimpl`, `design_limitation`

5.2.19 write_unimpl

5.2.19.1 Description

The object functionality associated to a write access is unimplemented. Read access is using default implementation and can be overridden (for instance by the `write_only` template).

5.2.19.2 Log Output

First software write to registers results in an unimplemented log-message on log-level 1, remaining writes on log-level 3. First write to a field (if field value is not equal to write value) results in an unimplemented log-message on log-level 1, remaining writes on log-level 3.

5.2.19.3 Related Templates

`unimpl`, `read_unimpl`, `silent_unimpl`, `design_limitation`

5.2.20 silent_unimpl

5.2.20.1 Description

The object functionality is unimplemented, but do not print a lot of log-messages when reading or writing. Writes and reads are implemented as default writes and reads.

5.2.20.2 Log Output

First software read to a register results in an unimplemented log-message on log-level 2, remaining reads on log-level 3. Software reads to fields does not result in a log-message. First software write to a register results in an unimplemented log-message on log-level 2, remaining writes on log-level 3. First write to a field (if field value is not equal to write value) results in an unimplemented log-message on log-level 2, remaining writes on log-level 3.

5.2.20.3 Related Templates

`unimpl`, `design_limitation`

5.2.21 undocumented

5.2.21.1 Description

The object functionality is undocumented or poorly documented. Writes and reads are implemented as default writes and reads.

5.2.21.2 Log Output

First software write and read result in a spec_violation log-message on log-level 1, remaining on log-level 2.

5.2.22 unmapped

5.2.22.1 Description

The register is excluded from the address space of the containing bank.

5.2.23 sticky

5.2.23.1 Description

Do not reset object value on soft-reset, keep current value.

5.2.24 design_limitation

5.2.24.1 Description

The object's functionality is not in the model's scope and has been left unimplemented as a design decision. Software and hardware writes and reads are implemented as default writes and reads. Debug registers are a prime example of when to use this template. This is different from *unimplemented* which is intended to be implement (if required) but is a limitation in the current model.

5.2.24.2 Related Templates

`unimpl`, `silent_unimpl`

5.2.25 no_reset

5.2.25.1 Description

The register's or field's value will not be changed on a hard or soft reset.

5.3 Bank related templates

5.3.1 function_mapped_bank

5.3.1.1 Description

Only valid in bank objects. The bank is recognized as a function mapped bank by the function_io_memory template, and is mapped to a specified function by whoever instantiates that template.

5.3.1.2 Parameters

function: the function number, an integer

5.3.1.3 Related Templates

function_io_memory

5.3.2 function_io_memory

5.3.2.1 Description

Only valid in implement objects named io_memory. Implements the io_memory interface by function mapping: An incoming memory transaction is handled by finding a bank that instantiates the function_mapped_bank template inside the same (sub)device as implement, and has a function number that matches the memory transaction's. If such a bank exists, the transaction is handled by that bank. If no such bank exists, an error message is logged and a miss is reported for the access.

Mapping banks by function number is a deprecated practice, still used by PCI devices for legacy reasons. It is usually easier to map a bank directly into a memory space, than using a function number as an indirection.

Note also that function numbers as defined by the PCI standard are unrelated to the function numbers of banks. They can sometimes coincide, though.

5.3.2.2 Parameters

function: the function number, an integer

5.3.2.3 Related Templates

function_mapped_bank

5.3.3 miss_pattern_bank

5.3.3.1 Description

Only valid in bank objects. Handles unmapped accesses by ignoring write accesses, and returning a given value for each unmapped byte. If you want to customize this behaviour, overriding unmapped_get is sufficient to also customize unmapped_read.

5.3.3.2 Parameters

miss_pattern: each missed byte in a miss read is set to this value

5.4 Connect related templates

5.4.1 map_target

A connect object can instantiate the template map_target. The template provides an easy way to send memory transactions to objects that can be mapped into Simics memory maps. It defines a default implementation of set which assigns the session variable map_target of type map_target_t *, which can be used to issue transactions to the connected object. It also defines a default implementation of validate which verifies that the object can be used to create a map target, i.e. the Simics API SIM_new_map_target returns a valid pointer.

The template defines the following methods:

- read(uint64 addr, uint64 size) -> (uint64) throws

Reads `size` bytes starting at `addr` in the connected object. Size must be 8 or less. Byte order is little-endian. Throws an exception if the read fails.

- `read_bytes(uint64 addr, uint64 size, uint8 *bytes) throws`

  Reads `size` bytes into `bytes`, starting at `addr` in the connected object. Throws an exception if the read fails.

- `write(uint64 addr, uint64 size, uint64 value) throws`

  Writes `value` of `size` bytes, starting at `addr` in the connected object. Size must be 8 or less. Byte order is little-endian. Throws an exception if the write fails.

- `write_bytes(uint64 addr, uint64 size, const uint8 *bytes) throws`

  Writes `size` bytes from `bytes`, starting at `addr` in the connected object. Throws an exception if the write fails.

- `issue(transaction_t *t, uint64 addr) -> (exception_type_t)`

  Provides a shorthand to the API function `SIM_issue_transaction`. This method is called by the read/write methods in this template. It can be overridden, e.g. to add additional atoms to the transactions, while still allowing the ease-of-use from the simpler methods.

5.5 Signal related templates

5.5.1 signal_port

Implements a signal interface with saved state. The current state of the signal is stored in the saved boolean `high`, and a spec-violation message is logged on level 2 if the signal is raised or lowered when already high or low. The methods `signal_raise` and `signal_lower` can be overridden to add additional side effects.

5.5.2 signal_connect

Implements a connect with a signal interface, with saved state. The current state of the signal is stored in the saved boolean `signal.high`. If the connect is changed while `signal.high` is `true` and the device is configured, the `signal_lower` method will be called on the old object, and the `signal_raise` method will be called on the new object. Similarly, if the device is created with `signal.high` set to `true`, the `signal_raise` method will be called on the connected object in the finalize phase. This behaviour can be changed by overriding the `set` method and/or the `post_init` method. The template defines the following method:

- `set_level(uint1 high)`: Sets the level of the signal, by calling `signal_raise` or `signal_lower`, as required. Also sets `signal.high` to `high`.

5.5.2.1 Related Templates

`signal_port`

A Messages

The following sections list the warnings and error messages from `dmlc`, with some clarifications.

A.1 Warning Messages

The messages are listed in alphabetical order; the corresponding tags are shown within brackets, e.g., `[WNDOC]`.

... [WSYSTEMC]

SystemC specific warnings

... [WWRNSTMT]

The source code contained a statement "`warning;`", which causes a warning to be printed.

'X then Y' log level has no effect when the levels are the same [WREDUNDANTLEVEL]

`X then Y` log level syntax has no effect when the first and subsequent levels are the same.

***INCREDIBLY UNSAFE* use of immediate 'after' statement: the callback argument '...' is a pointer to stack-allocated data! [WIMMAFTER]**

An immediate `after` statement was specified where some argument to the callback is a pointer to some stack-allocated data — i.e. a pointer to data stored within a local variable. That data is guaranteed to be invalid by the point the callback is called, which presents an enormous security risk!

***INCREDIBLY UNSAFE* use of the 'send' operation of a hook: the message component '...' is a pointer to stack-allocated data! [WHOOKSEND]**

The `send` operation of a hook was called, and some provided message component is a pointer to some stack-allocated data — i.e. a pointer to data stored within a local variable. That data is guaranteed to be invalid by the point the message is sent, which presents an enormous security risk!

If you must use pointers to stack-allocated data, then `send_now` should be used instead of `send`. If you want the message to be delayed to avoid ordering bugs, create a method which wraps the `send_now` call together with the declarations of the local variable(s) which you need pointers to, and then use immediate after (`after: m(...)`) to delay the call to that method.

Comparing negative constant to unsigned integer has a constant result [WNEGCONSTCOMP]

DML uses a special method when comparing an unsigned and signed integer, meaning that comparing a negative constant to an unsigned integer always has the same result, which is usually not the intended behaviour.

Outdated AST file: ... [WOLDAST]

A precompiled DML file has an old time-stamp. This may happen if a user accidentally edits a DML file from the standard library. A safe way to suppress the warning is to remove the outdated `.dmlast` file.

The assignment source is a constant value which does not fit the assign target of type '...', and will thus be truncated [WASTRUNC]

The source of an assignment is a constant value that can't fit in the type of the target, and is thus truncated. This warning can be silenced by explicitly casting the expression to the target type.

Use of unsupported feature: ... [WEXPERIMENTAL]

This part of the language is experimental, and not yet officially supported. Code relying on the feature may break without notice in future releases.

Use of unsupported feature: ... [WEXPERIMENTAL_UNMAPPED]

This part of the language is experimental, and not yet officially supported. Code relying on the feature may break without notice in future releases.

deprecation: ... [WDEPRECATED]

This part of the language is deprecated, usually because the underlying support in Simics is deprecated.

duplicate event checkpoint names: ... [WDUPEVENT]

Two or more events will be checkpointed using the same name, which means that the checkpoint cannot be safely read back.

file has no version tag, assuming version 1.2 [WNOVER]

A DML file must start with a version statement, such as `dml 1.4;`

implementation of ...() without 'is ...' is ignored by the standard library [WNOIS]

Many standard method overrides will only be recognized if a template named like the method is also instantiated. For instance, the method `set` in a field has no effect unless the `set` template is instantiated.

log statement with likely misspecified log level(s) and log groups: ... [WLOGMIXUP]

A specified log level of a `log` looks as though you meant to specify the log groups instead, and/or vice versa. For example:

```
// Log group used as log level, when the intention is instead to
// specify log groups and implicitly use log level 1
log spec_viol, some_log_group: ...;

// Log groups and log level mistakenly specified in reverse order
log info, (some_log_group | another_log_group), 2: ...;

// Log level used as log groups, when the intention is instead to
// specify the subsequent log level
log info, 2, 3: ...;
```

If you want to specify log groups, make sure to (explicitly) specify the log level beforehand. If you want to specify the subsequent log level, use `then` syntax.

```
log spec_viol, 1, some_log_group: ...;
log info, 2, (some_log_group | another_log_group): ...;
log info, 2 then 3: ...;
```

This warning is only enabled by default with Simics API version 7 or above (due to the compatibility feature `suppress_WLOGMIXUP`.)

negative register offset: *N* [WNEGOFFS]

A negative integer expression is given as a register offset. Register offsets are unsigned 64-bit numbers, which means that a negative offset expression translates to a very large offset.

no 'desc' parameter specified for device [WNSHORTDESC]

No short description string was specified using the 'desc' parameter. (This warning is disabled by default.)

no documentation for '...' [WNDOC]

No documentation string was specified for the attribute. (This warning is disabled by default.)

no documentation for required attribute '...' [WNDOCRA]

No documentation string was specified for a *required* attribute.

overriding non-throwing DML 1.4 method with throwing DML 1.2 method [WTHROWS_DML12]

In DML 1.2, a method is by default permitted to throw an exception, while in DML 1.4, an annotation `throws` is required for that. So, if a method without annotations is ported to DML 1.4, it will no longer permit exceptions. If such method is overridden by a DML 1.2 file, then a non-throwing method is overridden by a potentially throwing method, which is normally a type error. However, this particular case is reduced to this warning. If an exception is uncaught in the override, then this will automatically be caught in runtime and an error message will be printed.

potential leak of confidential information [WCONFIDENTIAL]

The object's name/qname is used as part of an expression in a context other than the log statement, which could potentially lead to the leak of confidential information.

prefer 'is' statement outside template braces, 'template ... is (x, y) {' [WTEMPLATEIS]

In a template with methods marked `shared`, it is recommended that other templates are instantiated on the same line

shifting away all data [WSHALL]

The result of the shift operation will always be zero. (This warning is disabled by default.)

sizeof on a type is not legal, use sizeoftype instead [WSIZEOFTYPE]

The 'sizeof' operator is used on a type name, but expects an expression. Use the 'sizeoftype' operator for types.

the time value of type '...' is implicitly converted to the type '...' expected by the specified time unit '...'. [WTTYPEC]

The delay value provided to an `after` call is subject to implicit type conversion which may be unexpected for certain types. To silence this warning, explicitly cast the delay value to the expected type.

unused implementation of DML 1.2 method ...; enclose in #if (dml_1_2) ? [WUNUSED_DML12]

A DML 1.4 file contains a method implementation that would override a library method in DML 1.2, but which is not part of the DML 1.4 library, because some methods have been renamed. For instance, implementing `read_access` in a register makes no sense in DML 1.4, because the method has been renamed to `read_register`.

If a DML 1.4 file contains common code that also is imported from DML 1.2 devices, then it may need to implement methods like `read_access` to get the right callbacks when compiled for DML 1.2. Such implementations can be placed inside `#if (dml_1_2) { }` blocks to avoid this warning.

unused parameter ... contains ... [WREF]

An unused parameter refers to an object that has not been declared.

This warning message will be replaced with a hard error in future major versions of Simics.

unused: ... [WUNUSED]

The object is not referenced anywhere. (This warning is disabled by default.; it typically causes many false warnings.)

unused: ... methods are not called automatically for ... objects in ... [WUNUSEDDEFAULT]

The object is not referenced anywhere but it matches a name of an object automatically referenced in another scope. This is the same as WUNUSED but only for known common errors and it will never be emitted if WUNUSED is enabled.

very suspect pointer-to-pointer cast: the new base type has incompatible representation. This could lead to your code getting mangled by the C compiler, with unpredictable results. [WPCAST]

A pointer is cast to a base type which has incompatible representation compared to the original. Accessing the pointed-to object via the new pointer type will almost certainly constitute undefined behavior.

This warning is extremely limited in scope: don't rely on it to catch every bad pointer cast.

To silence this warning, first cast the pointer to `void *`, then cast it to the desired type.

A.2 Error Messages

The messages are listed in alphabetical order; the corresponding tags are shown within brackets, e.g., `[ENBOOL]`.

... [EERRSTMT]

The source code contained a statement "`error;`", which forces a compilation error with the given message, or the standard message "forced compilation error in source code".

... in template ... does not belong to the template type [ENSHARED]

If a template provides an object that is not accessible from shared methods, such as an untyped parameter or a non-shared method, then that object's name is reserved within the scope of the shared method. I.e., if a shared method tries to access a symbol that isn't accessible, then ENSHARED is reported, even before looking for the symbol in the global scope. Section 3.7.2 describes which template symbols are accessible from a shared method.

'...' has no member named '...' [EMEMBER]

Attempt to access a nonexisting member of a compound data structure.

'...' is a message component parameter, and can only be used as a direct argument to the callback method of the after statement [EAFTERMSGCOMPPARAM]

Message component parameters bound by a hook-bound after statement can only be used as direct arguments to the specified callback method, and cannot be used in arbitrary expressions.

'...' is a not a valid message component type for a hook, as it is or contains some ... [EHOOKTYPE]

There are some minor restrictions to a hook's message component types. Anonymous structs and arrays of variable/unknown size are not supported.

'.len' cannot be used with variable-length arrays [EVLALEN]

`.len` cannot be used with variable-length arrays

Ambiguous invocation of default implementation [EAMBDEFAULT]

A method may not invoke its default implementation if multiple methods are overridden, and the template inheritance graph is insufficient to infer that one default implementation overrides the others. See section 3.10.3 for details.

Ambiguous invocation of template-qualified method implementation call. '...' does not provide an implementation of '...', and inherits multiple unrelated implementations from its ancestor templates.... [EAMBTQMIC]

A template-qualified method implementation call was made, when the template inheritance graph for specified template is insufficient to infer that one implementation overrides the others. To resolve this, the template-qualified method implementation call should instead be qualified with the specific ancestor template that has the desired implementation.

Cannot declare '...' variable in an inline method [ESTOREDINLINE]

You cannot declare session or saved variables in methods marked with 'inline'

DML version ... does not support API version ... [ESIMAPI]

The DML file is written in a too old version of DML. Use the `--simics-api` option to use a sufficiently old Simics API.

Declaration would result in conflicting attribute name [EATTRCOLL]

This error is signalled if two DML declarations would result in two Simics attributes being registered with the same name.

This most commonly happens when an attribute name is a result of the object hierarchy, and there is another object named similarly. For example, if a bank contains one register named

`g_r` and a group `g` containing a register named `r`.

Instantiating template ... requires abstract ... ... to be implemented [EABSTEMPLATE]

If a template has any abstract methods or parameters, they must all be implemented when instantiating the template.

No such provisional feature .... Valid values are: ... [ENOPROV]

An invalid identifier was passed in the `provisional` statement.

The interface struct member ... is not a function pointer [EIMPLMEMBER]

A method in an `implement` object corresponds to a struct member that isn't a function pointer

Too many loggroup declarations. A maximum of 63 log groups (61 excluding builtins) may be declared per device. [ELOGGROUPS]

Too many log groups were declared. A device may have a maximum of 63 `loggroup` declarations (61 excluding the built-in `Register_Read` and `Register_Write` loggroups).

Unknown pragma: ... [EPRAGMA]

An unknown pragma was specified

abstract method ... overrides existing method [EAMETH]

An abstract method cannot override another method.

an anonymous ... cannot implement interfaces [EANONPORT]

An `implement` definition can only exist in a port or bank that has a name.

array has too many elements (*N* >= 2147483648) [EASZLARGE]

Object arrays with huge dimensions are not allowed; the product of dimension sizes must be smaller than 2^{31}.

array index out of bounds [EOOB]

The used index is outside the defined range.

array range must start at 0 [EZRANGE]

An array index range must start at zero.

array size is less than 1 [EASZR]

An array must have at least one element.

array upper bound is not a constant integer: ... [EASZVAR]

The size of an array must be a constant integer.

assignment to constant [ECONST]

The lvalue that is assigned to is declared as a `const` and thus can't be assigned to.

attempt to override non-default method '...' [EDMETH]

A method can only be overridden if it is declared as `default`

attempt to override non-shared method ... with shared method [ETMETH]

A shared method cannot override a non-shared method

attribute has no get or set method [EANULL]

An attribute must have a set or a get method to be useful.

attribute type undefined: ... [EATYPE]

Either the `attr_type` or the `type` parameter of the attribute must be specified.

bad declaration of automatic parameter '...' [EAUTOPARAM]

Some parameters are predefined by DML, using the `auto` keyword. Such parameters may only be declared by the standard library, and they may not be overridden.

bit range of field '...' outside register boundaries [EBITRR]

The bit range of a field can only use bits present in the register.

bit range of field '...' overlaps with field '...' [EBITRO]

The fields of a register must not overlap.

bitslice size of ... bits is not between 1 and 64 [EBSSIZE]

Bit slices cannot be larger than 64 bits.

bitslice with big-endian bit order and uncertain bit width [EBSBE]

A big-endian bit slice can only be done on an expression whose type is explicitly defined, such as a local variable or a register field.

call to method '...' in unsupported context [EAPPLYMETH]

Calls to inline methods, methods that may throw, or methods that have multiple output parameters cannot be used as arbitrary expressions. In DML 1.2, any such method must be called via the `call` or `inline` statements, and in DML 1.4 any such method must be called either as a standalone statement, or as an initializer (e.g., RHS of an assignment or argument of a `return` statement).

cannot access device instance in device independent context [EINDEPENDENTVIOL]

Expressions that depend on values stored in a device instance cannot be evaluated in contexts where the device instance is not available. This is within static contexts — for example when initializing typed template parameters — or within independent methods.

cannot assign to inlined parameter: '...' [EASSINL]

The target of the assignment is a method parameter that has been given a constant or undefined value when inlining the method.

cannot assign to this expression: '...' [EASSIGN]

The target of the assignment is not an l-value, and thus cannot be assigned to.

cannot convert this method reference to a function pointer [ESTATICEXPORT]

A method reference can only be converted to a function pointer if the method is non-inline, non-shared, non-throwing, and declared outside an object array.

cannot define both 'allocate_type' parameter and local data objects [EATTRDATA]

Specifying `allocate_type` and using 'data' declarations in the same attribute object is not allowed.

cannot export this method [EEXPORT]

Can only export non-inline, non-shared, non-throwing methods declared outside object arrays.

cannot find file to import: ... [EIMPORT]

The file to imported could not be found. Use the `-I` option to specify additional directories to search for imported files.

cannot import file containing device declaration [EDEVIMP]

Source files that are used with `import` directives may not contain `device` declarations.

cannot use a register with fields as a value: ... [EREGVAL]

When a register has been specified with explicit fields, you have to use the `get` and `set` methods to access the register as a single value.

cannot use an array as a value: '...' [EARRAY]

A whole array cannot be used as a single value.

cannot use endian integer as argument type in declaration [EEARG]

Function and method arguments in declarations cannot be of endian integer type.

cannot use variable index in a constant list [EAVAR]

Indexing into constant lists can only be done with constant indexes.

checkpointable attribute missing set or get method [EACHK]

An attribute must have set and get methods to be checkpointable. This attribute has neither, and the 'configuration' parameter is either "required" or "optional".

circular dependency in parameter value [ERECPARAM]

The value of a parameter may not reference the parameter itself, neither directly nor indirectly.

conditional 'in each' is not allowed [ECONDINEACH]

It is not permitted to have an `in each` statement directly inside an `if` conditional.

conditional parameters are not allowed [ECONDP]

It is not permitted to declare a parameter directly inside an `if` conditional.

conditional templates are not allowed [ECONDT]

It is not permitted to use a template directly inside an `if` conditional.

conflicting default definitions for method '...' [EDDEFMETH]

If a method has two default implementations, then at least one of them must be defined in a template.

conflicting definitions of ... when instantiating ... and ... [EAMBINH]

If a method or parameter has multiple definitions, then there must be a unique definition that overrides all other definitions.

const qualified function type [ECONSTFUN]

A function type cannot be `const` qualified;

const qualifier discarded [EDISCONST]

A pointer to a constant value has been assigned to a pointer to a non-constant.

continue is not possible here [ECONTU]

A `continue` statement cannot be used in a `#foreach` or `#select` statement.

cyclic import [ECYCLICIMP]

A DML file imports itself, either directly or indirectly.

cyclic template inheritance [ECYCLICTEMPLATE]

A template inherits from itself, either directly or indirectly.

declaration of vect type without simics_util_vect provisional [EOLDVECT]

`vect` types are only permitted if the `simics_util_vect` provisional feature is enabled.

duplicate bank function number: *N* [EDBFUNC]

The device contains two differently-named banks that use the same function number.

duplicate definition of variable '...' [EDVAR]

A local variable has more than one definition in the same code block.

duplicate method parameter name '...' [EARGD]

All parameter names of a method must be distinct.

expression may not depend on the index variable ... [EIDXVAR]

Expressions that are evaluated statically to constants cannot have different values for different elements in a register array. This includes, for instance, the `allocate` parameter in registers and fields, and object-level `if` statements.

file not found [ENOFILE]

The main input file could not be found.

heterogeneous bitsize in field array [EFARRSZ]

The bit width must be identical across the elements of a field array.

illegal 'after' statement bound to hook '...': hook has *N* message components, but *N* message component parameters are given [EAFTERHOOK]

An illegal hook-bound `after` statement was specified. The number of message component parameters must be equal to the number of message components of the hook.

illegal 'after' statement... with callback '....send_now': every message component of '...' ...must be of serializable type... [EAFTERSENDNOW]

An illegal `after` statement was specified where the callback is `send_now` of a hook. Every message component type of the hook must be serializable (unless that component is provided through a message component parameter of the `after` statement, if the `after` statement is attaching the callback to another hook.)

illegal 'after' statement... with callback method '...': ... [EAFTER]

An illegal `after` statement was specified. The method callback specified may not have any output parameters/return values. If the after is with a time delay or bound to a hook, every input parameter must be of serializable type (unless that input parameter receives a message component of a hook).

illegal attribute name: ... [EANAME]

This name is not available as the name of an attribute, since it is used for an automatically added attribute.

illegal bitfields definition: ... [EBFLD]

A `bitfield` declaration must have an integer type that matches the width of the field.

illegal bitorder: '...' [EBITO]

The specified bit-order is not allowed.

illegal bitslice operation [EBSLICE]

A bitslice operation was attempted on an expression that is not an integer.

illegal cast to '...' [ECAST]

The cast operation was not allowed. It is illegal to cast to void.

illegal comparison; mismatching types [EILLCOMP]

The values being compared do not have matching types.

illegal function application of '...' [EAPPLY]

The applied value is not a function.

illegal increment/decrement operation [EINC]

An increment or decrement operation can only be performed on simple lvalues such as variables.

illegal interface method reference: ... [EIFREF]

Interface function calls must be simple references to the method.

illegal layout definition: ... [ELAYOUT]

The type of a member of a `layout` declaration must be an integer or bitfield with a bit width that is a multiple of 8, or another layout.

illegal operands to binary '...' [EBINOP]

One or both of the operands have the wrong type for the given binary operator.

illegal pointer type: ... [EINTPTRTYPE]

Pointer types that point to integers with a bit width that is not a power of two are not allowed.

illegal register size for '...' [EREGISZ]

The specified register size is not allowed. Possible values are 1-8.

illegal type: array of functions [EFUNARRAY]

It is illegal to express an array type where the base type is a function type.

illegal use of void type [EVOID]

The type `void` is not a value, and thus cannot be used as the type of e.g. a variable or struct member

illegal value for parameter '...' [EPARAM]

The parameter is not bound to a legal value.

incompatible array declarations: ... [EAINCOMP]

The array has been declared more than once, in an incompatible way.

incompatible extern declarations for '...': type mismatch [EEXTERNINCOMP]

Multiple `extern` declarations with mismatching types are given for the same identifier.

incompatible method definitions: ... [EMETH]

The default implementation is overridden by an implementation with different input/output parameters.

incompatible version (...) while compiling a ... device [EVERS]

A device declared to be written in one DML language version tried to import a file written in an incompatible language version.

invalid data initializer: ... [EDATAINIT]

An invalid initializer was detected. The error message provides the detailed information.

invalid expression: '...' [EINVALID]

The expression does not produce a proper value.

invalid log type: '...' [ELTYPE]

Log-statement type must be one of `info`, `warning`, `error`, `spec_viol`, and `unimpl`.

invalid name parameter value: '...' [ENAMEID]

The name parameter does not follow identifier syntax.

invalid override of non-default declaration ... [EINVOVER]

Only default declarations of parameters can be overridden.

invalid template-qualified method implementation call made via a value of template type: '...' does not provide nor inherit a shared implementation of '...' [ENSHAREDTQMIC]

A template-qualified method implementation call via a value of template type, including when `this.templates` is used within the body of a `shared` method, can only be done if the specified template provides or inherits a `shared` implementation of the specified method. If an implementation is never provided or inherited by the template, or the template provides or inherits a non-`shared` implementation, then the call can't be made.

For example, the following is permitted:

```
template t {
    shared method m();
}

template u is t {
    shared method m() default {
        log info: "implementation from 'u'";
    }
}

template v is t {
    shared method m() default {
        log info: "implementation from 'v'";
    }
}

template uv is (u, v) {
    shared method m() {
        // 'this' is a value of the template type 'uv'
        this.templates.u.m();
        // Equivalent to 'this.templates.v.m()'
        templates.v.m();
    }
}
```

But the following is not:

```
template t {
    shared method m();
}

template u is t {
    shared method m() default {
        log info: "implementation from 'u'";
    }
}

template v is t {
    method m() default {
        log info: "implementation from 'v'";
    }
}

template uv is (u, v) {
    // Indirection as a shared implementation is not allowed to override a
    // non-shared implementation, but even if it were...
    method m() {
        m_impl();
    }

    shared method m_impl() {
        this.templates.u.m();
        // This is rejected because the implementation of 'm' provided by
        // 'v' is not shared.
        this.templates.v.m();
    }
}
```

As a result, resolving a conflict between a non-`shared` method implementation and a `shared` method implementation can typically only be done by having most parts of the overriding implementation be non-`shared`:

```
template uv is (u, v) {
    method m() {
        // OK; 'this' is a compile-time reference to the object
        // instantiating the template rather than a value of template type.
        this.templates.u.m();
        this.templates.v.m();
    }
}
```

Alternatively, a new `shared` method with non-`shared` implementation can be declared to allow access to the specific non-`shared` implementation needed (at the cost of increasing the memory overhead needed for the template type):

```
template uv is (u, v) {
    method m() {
        m_impl();
    }

    shared method m_impl_by_v();
    method m_impl_by_v() {
        this.templates.v.m();
    }

    shared method m_impl() {
        this.templates.u.m();
        // OK
        m_impl_by_v();
    }
}
```

invalid template-qualified method implementation call, '...' does not instantiate '...' [ETQMIC]

A template-qualified method implementation call can only be done if the specified template is actually instantiated by the object.

invalid template-qualified method implementation call, '...' does not provide nor inherit an implementation of a method '...'... [EMEMBERTQMIC]

A template-qualified method implementation call can only be done if the specified template actually does provide or inherit an implementation of the named method for the object instantiating the template. That the template provides or inherits an abstract declaration of the method is not sufficient.

Apart from more mundane causes (e.g. misspellings), this error could happen if all implementations that the specified template may provide/inherit end up not being provided to the object instantiating the template, due to every implementation being eliminated by an `#if` statement.

invalid template-qualified method implementation call, '...' not a subtemplate of '...' [ETTQMIC]

A template-qualified method implementation call via a value of template type, including when `this.templates` is used within the body of a `shared` method, can only be done if the specified template is an ancestor template of the template type, the `object` template type, or the template type itself.

invalid upcast, ... not a subtemplate of ... [ETEMPLATEUPCAST]

When casting to a template type, the source expression must be either an object implementing the template, or an expression whose type is a subtemplate of the target type.

log level must be ... [ELLEV]

The log level given in a log statement must be an integer between 1 and 4, or 1 and 5 for a subsequent log level (`then ...`), unless the log kind is one of "warning", "error", or "critical", in which case it must be 1 (or 5 for subsequent log level).

malformed format string: unknown format at position *N* [EFORMAT]

The log-statement format string is malformed.

malformed switch statement: ... [ESWITCH]

A switch statement must start with a `case` label, and there may be at most one `default` label which must appear after all `case` labels

method return type declarations may not be named: ... [ERETARGNAME]

In DML 1.4, the output arguments of a method are anonymous

missing device declaration [EDEVICE]

The main source file given to the DML compiler must contain a `device` declaration.

missing return statement in method with output argument [ENORET]

If a method has output arguments, then control flow may not reach the end of the method. Either an explicit value must be returned in a return statement, or the execution must be aborted by an exception or assertion failure. Note that DMLC's control flow analysis is rather rudimentary, and can issue this error on code that provably will return. In this case, the error message can be silenced by adding `assert false;` to the end of the method body.

more than one output parameter not allowed in interface methods [EIMPRET]

Methods within an `interface` declaration may have only have zero or one output parameter.

name collision on '...' [ENAMECOLL]

The name is already in use in the same scope.

negative size (*N* < *N*) of bit range for '...' [EBITRN]

The size of the bit range must be positive. Note that the [msb:lsb] syntax requires that the most significant bit (msb) is written to the left of the colon, regardless of the actual bit numbering used.

no assignment to parameter '...' [ENPARAM]

The parameter has been declared, but is not assigned a value or a default value.

no default implementation [ENDEFAULT]

The default implementation of a method was invoked, but there was no default implementation.

no return type [ERETTYPE]

The type of the return value (if any) must be specified for methods that implement interfaces.

no type for ... parameter '...' [ENARGT]

Methods that are called must have data type declarations for all their parameters. (Methods that are only inlined do not need this.)

non-boolean condition: '...' of type '...' [ENBOOL]

Conditions must be properly boolean expressions; e.g., "`if (i == 0)`" is allowed, but "`if (i)`" is not, if `i` is an integer.

non-constant expression: ... [ENCONST]

A constant expression was expected.

non-constant parameter, or circular parameter dependencies: '...' [EVARPARAM]

The value assigned to the parameter is not a well-defined constant.

non-constant strings cannot be concatenated using '+' [ECSADD]

Non-constant strings cannot be concatenated using `+`.

not a list: ... [ENLST]

A list was expected.

not a method: '...' [ENMETH]

A method name was expected. This might be caused by using `call` or `inline` on something that counts as a C function rather than a method.

not a pointer: ... (...) [ENOPTR]

A pointer value was expected.

not a value: ... [ENVAL]

Only some objects can be used as values directly. An attribute can only be accessed directly as a value if it has been declared using the `allocate_type` parameter.

nothing to break from [EBREAK]

A `break` statement can only be used inside a loop or switch construct.

nothing to continue [ECONT]

A `continue` statement can only be used inside a loop construct.

object expected: ... [ENOBJ]

A reference to an object was expected.

object is not allocated at run-time: ... [ENALLOC]

An object which is not allocated at run-time cannot be referenced as a run-time value.

operand of '...' is not an lvalue [ERVAL]

The operand of `sizeof`, `typeof` and `&` must be a lvalue.

overlapping registers: '...' and '...' [EREGOL]

The registers are mapped to overlapping address ranges.

parameter '...' not declared previously. To declare and define a new parameter, use the ':...' syntax. [ENOVERRIDE]

When the `explict_param_decls` provisional feature is enabled, parameter definitions written using `=` and `default` are only accepted if the parameter has already been declared. To declare and define a new parameter not already declared, use the `:=` or `:default` syntax.

passing const reference for nonconst parameter ... in ... [ECONSTP]

C function called with a pointer to a constant value for a parameter declared without const in the prototype.

recursive inline of ... [ERECUR]

Methods may not be inlined recursively.

recursive type definition of ... [ETREC]

The definition of a structure type can not have itself as direct or indirect member.

reference to unknown object '...' [EREF]

The referenced object has not been declared.

right-hand side operand of '...' is zero [EDIVZ]

The right-hand side of the given / or % operator is always zero.

saved variable declared with (partially) const-qualified type ... [ESAVEDCONST]

Declaring a saved variable with a type that is (partially) const-qualified is not allowed, as they can be modified due to checkpoint restoration.

shift with negative shift count: '... [ESHNEG]

The right-hand side operand to a shift operator must not be negative.

struct declaration not allowed in a ... [EANONSTRUCT]

Declarations of new structs are not permitted in certain contexts, such as method arguments, `new` expressions, `sizeoftype` expressions and `cast` expressions.

struct member is a function [EFUNSTRUCT]

A member of a struct cannot have a function type.

struct or layout with no fields [EEMPTYSTRUCT]

A struct or layout type must have at least one field. This restriction does not apply to structs declared in a `extern typedef`.

syntax error..... [ESYNTAX]

The code is malformed.

the parameter '...' has already been declared (':...' syntax may not be used for parameter overrides) [EOVERRIDE]

When the `explict_param_decls` provisional feature is enabled, any parameter declared via `:=` or `:default` may not already have been declared. This means `:=` or `:default` syntax can't be used to override existing parameter declarations (not even those lacking a definition of the parameter.)

the size of dimension *N* (with index variable '...') is never defined [EAUNKDIMSIZE]

The size of an array dimension of an object array must be defined at least once across all declarations of that object array.

this object is not allowed here [ENALLOW]

Many object types have limitations on the contexts in which they may appear.

trying to get a member of a non-struct: '...' of type '...' [ENOSTRUCT]

The left-hand side operand of the `.` operator is not of struct type.

trying to index something that isn't an array: '...' [ENARRAY]

Indexing can only be applied to arrays, integers (bit-slicing), and lists.

typed parameter definitions may not contain independent methods calls [ETYPEDPARAMVIOL]

Independent method calls are not allowed within the definitions of typed parameters.

uncaught exception [EBADFAIL]

An exception is thrown in a context where it will not be caught.

uncaught exception in call to DML 1.2 method '...' [EBADFAIL_dml12]

If a DML 1.2 method lacks a `nothrow` annotation, and a non-throwing DML 1.4 method calls it, then DMLC will analyze whether the method call can actually cause an exception. If it can, this error is reported; if not, the call is permitted.

For this error, a 1.2 method counts as throwing if it throws an exception, or calls a `throws` marked 1.4 method, or (recursively) if it invokes a method that counts as throwing. A call or throw statement inside a `try` block does not cause the method to count as throwing. The methods `attribute.set`, `bank.read_access` and `bank.write_access` count as throwing even if they don't throw.

This error is normally reported while porting common DML 1.2 code to DML 1.4: most 1.2 methods are not meant to throw exceptions, and when converted to DML 1.4 this becomes a strict requirement unless the method is annotated with the `throws` keyword. The remedy for this error message is normally to insert a `try` block around some call along the throwing call chain, with a `catch` block that handles the exception gracefully. The `try` block should usually be as close as possible to the `throw` in the call chain.

undefined register size for '...' [EREGNSZ]

All registers must have a specified constant size.

undefined value: '...' [EUNDEF]

Caused by an attempt to generate code for an expression that contains the `undefined` value.

unknown identifier: '...' [EIDENT]

The identifier has not been declared anywhere.

unknown interface type: ... [EIFTYPE]

The interface datatype is unknown.

unknown template: '...' [ENTMPL]

The template has not been defined.

unknown type of expression [ENTYPE]

This expression has an unknown type.

unknown type: '...' [ETYPE]

The data type is not defined in the DML code.

unknown value identifier in the operand of 'sizeof': '...' [EIDENTSIZEOF]

A variant of the EIDENT message exclusive to usages of `sizeof`: it is emitted when the operand of `sizeof` makes use of an identifier which is not present in value scope, but *is* present in type scope. This likely means `sizeof` was used when `sizeoftype` was intended.

unserializable type: ... [ESERIALIZE]

Some complex types, in particular most pointer types, cannot be automatically checkpointed by DML, and are therefore disallowed in contexts such as `saved` declarations.

value of parameter ... is not yet initialized [EUNINITIALIZED]

Some parameters that are automatically supplied by DML cannot be accessed in early stages of compilation, such as in object-level if statements.

variable length array declared with (partially) const-qualified type [EVLACONST]

Variable length arrays may not be declared const-qualified or with a base type that is (partially) const-qualified.

variable or field declared ... [EVARTYPE]

A variable has been declared with a given type but the type is not acceptable.

wrong number of ... arguments [EARG]

The number of input/output arguments given in the call differs from the method definition.

wrong number of arguments for format string [EFMTARGN]

The log-statement has too few or too many arguments for the given format string.

wrong number of return value recipients: Expected *N*, got *N* [ERETLVALS]

The number of return value recipients differs from the number of values the called method returns.

wrong number of return values: Expected *N*, got *N* [ERETARGS]

The number of return values in a return statement must match the number of outputs in the method.

wrong type [EBTYPE]

An expression had the wrong type.

wrong type for '...' operator [EINCTYPE]

The prefix and postfix increment/decrement operators can only be used on integer and pointer expressions.

wrong type for argument *N* of format string ('...') [EFMTARGT]

Argument type mismatch in a log-statement format string.

wrong type for initializer [EASTYPE]

The target of an initializer is incompatible with the type of the initializer.

wrong type for parameter ... in ... call [EPTYPE]

The data type of the argument value given for the mentioned method or function parameter differs from the function prototype.

wrong type in ... parameter '...' when ... '...' [EARGT]

The data type of the argument value given for the mentioned method parameter differs from the method definition.

5 Standard Templates | B Provisional language features

B Provisional language features

Sometimes, we may choose to extend the DML compiler with a feature before it is ready to be fully incorporated into the language. This can happen for different reasons, e.g. if the design is not fully evaluated or if the feature is backward incompatible. Currently, all provisional features are enabled on a per-file basis, by adding `provisional feature_name, other_feature_name;` just after the `dml 1.4;` statement.

Provisional features can come in two flavours:

- *Stable* provisional features have a proven design and are expected to remain pretty stable over time. Details in semantics may still change between versions, but if we decide to make a significant incompatible change to a supported provisional, then we will create a second version of the provisional, under a new name, and keep both versions in parallel for some time. It can make sense to use supported provisional features in production code.
- *Unstable* provisional features are expected to undergo significant incompatible changes over time, and are generally exposed to allow a dedicated team of modelers to evaluate an early design. It can be used to play around with, but should not be used in production code without first communicating with the DML team.

B.1 List of stable provisional features

`explicit_param_decls`

This feature extends the DML syntax for parameter definitions to distinguish between an intent to declare a new parameter, and an intent to override an existing parameter (including when providing a definition for an abstract parameter). This distinction allows DML to capture misspelled parameter overrides as compile errors.

The following new forms are introduced to mark the intent of declaring a new parameter:

- For typed parameters, `param NAME: TYPE = value;` is essentially a shorthand for

```
param NAME: TYPE;
param NAME = value;
```

and similarly, `param NAME: TYPE default value;` is essentially a shorthand for

```
param NAME: TYPE;
param NAME default value;
```

- For untyped parameters, `param NAME := value;` is essentially a shorthand for

```
param NAME;
param NAME = value;
```

and similarly `param :default value;` is essentially a shorthand for

```
param NAME;
param NAME default value;
```

If one of these forms is used for overriding an existing parameter, then DMLC will signal an error, because the declaration was not intended as an override. DMLC will also signal an error if a plain `param NAME = value;` or `param NAME default value;` declaration appears that does not override a pre-existing parameter.

In some rare cases, you may need to declare a parameter without knowing if it's an override or a new parameter. In this case, one can accompany a `param NAME = value;` or `param NAME default value;` declaration with a `param NAME;` declaration in the same scope/rank. This marks that the parameter assignment may be either an override or a new parameter, and no error will be printed.

Enabling the `explicit_param_decls` feature in a file only affects the parameter definitions specified in that file.

`simics_util_vect`

This feature enables the `vect` type, based on the `VECT` macro from the Simics C API (`simics/util/vect.h`).

This is a simple wrapping that behaves inconsistently in many ways, and we plan to eventually introduce a cleaner mechanism for vectors; the `simics_util_vect` is supported as an interim solution until we have that in place.

The syntax is `BASETYPE vect`, e.g. `typedef int vect int_vect_t;` to define a type for vectors of the `int` type.

Some caveats:

- `vect` types typically need to be `typedef`:ed before they are used. This is because `int vect` is blindly expanded into `VECT(int)` in C, which in turn expands into a `struct` definition, meaning that saying `VECT(int)` twice yields two incompatible types. This means, for instance, that `typeof` in DML doesn't work properly for `vect` types unless `typedef`:ed
- Importing `"internal.dml"` exposes various C macros from `vect.h` to DML: `VINIT`, `VELEMSIZE`, `VRESIZE`, `VRESIZE_FREE`, `VADD`, `VREMOVE`, `VDELETE_ORDER`, `VINSERT`, `VSETLAST`, `VLEN`, `VVEC`, `VGROW`, `VSHRINK`, `VFREE`, `VTRUNCATE`, `VCLEAR`, and `VCOPY`.
- DML natively supports indexing syntax, which is translated to `VGET` (or `VSET` for assignment). For instance:

```
typedef int vect int_vect_t;
method first_element(int_vect_t v) -> (int) {
    assert VLEN(v) > 0;
    return v[0];
}
```

Enabling the `simics_util_vect` feature in a file only affects the `vect` declarations in that file.

When the `simics_util_vect` feature is disabled, usage of `vect` is an error unless the `experimental_vect` compatibility feature is enabled.

A Messages | C Managing deprecated language features

C Managing deprecated language features

As the DML language evolves, we sometimes need to change the language in incompatible ways, which requires DML users to migrate their code. This appendix describes the mechanisms we provide to make this migration process smooth for users with large DML code bases.

In DML, deprecations can come in many forms. Deprecations in the form of removed or renamed symbols in libraries are rather easy to manage, since they give clear compile errors that often are straightforward to fix. A slightly harder type of deprecation is when some language construct or API function adjusts its semantics; this can make the model behave differently without signalling error messages. A third kind of deprecation is when DML changes how compiled models appear in Simics, typically to adjust changes in the Simics API. Such changes add another dimension because they typically affect the end-users of the DML models, rather than the authors of the models. Thus, as an author of a model you may need to synchronize your migration of such features with your end-users, to ease their transition to a new major version.

C.1 Deprecation mechanisms

The simplest deprecation mechanism is Simics API versions: Each deprecated DML feature is associated with a Simics API version, and each Simics version supports a number of such API versions. Features reach end-of-life when moving to a new Simics major version, the features belonging to a previous Simics API version are dropped. Since Simics is currently the primary distribution channel for DML, this deprecation scheme is used for DML features as well.

This scheme allows users with a large code base to smoothly migrate from one Simics major version, N, to the next, N+1:

- First, while still using version N, make sure all Simics modules are updated to use API version N. Modules can be migrated one by one.
- Second, update the Simics version to N+1. This should normally have no effect on DML, but may come with other challenges.
- Third, update modules to API N+1, one by one. Simics version N+1 will always offers full support for API N, so there is no rush to update, but changing the API version early makes sure deprecated features are not introduced in new code.

In addition to the API version, DML offers some compiler flags for selectively disabling deprecated features that are normally part of the used API. This has some uses, in particular:

- During migration to a new API version, disabling one deprecated feature at a time can allow a smoother, more gradual, migration.
- If a legacy feature is still fully supported in the latest API version, then it cannot be disabled by selecting an API version, so selectively disabling it is the only way to turn it off. There are reasons to do this, e.g.:
  - Disabling a feature before it is deprecated guarantees that it is not relied upon in new code, which eases later migration.

- Avoiding a feature that has a newer replacement makes the code base cleaner and more consistent.
- Some legacy features can also bloat models, by exposing features in a redundant manner. This can also have a negative impact on performance.

C.2 Controlling deprecation on the DML command-line

DMLC provides a command-line flag `--api-version` to specify the API version to be used for a model. When building with the standard Simics build system, this is controlled by the `SIMICS_API_VERSION` variable in `make`, or the `SIMICS_API`/`MODULE_SIMICS_API` variable in `CMake`.

DMLC also provides the `--no-compat=`*`tag`* flag, which disables the feature represented by *`tag`*. The available tags are listed in the next section. The tag also controls the name of a global boolean parameter that the DML program may use to check whether the feature is available. The parameter's name is the tag name preceded by `_compat_`.

C.3 List of deprecated features

C.3.1 Features available up to and including --simics-api=6

These features correspond to functionality removed when compiling using Simics API 7 or newer. With older Simics API versions, these features can be disabled individually by passing `--no-compat=`*`TAG`* to the `dmlc` compiler.

dml12_goto

> The `goto` statement is deprecated; this compatibility feature preserves it. Most `goto` based control structures can be reworked by changing the `goto` into a `throw`, and its label into a `catch` block; since this is sometimes nontrivial, it can be useful to disable the `goto` statement separately.

dml12_inline

> When using `inline` to inline a method in a DML 1.2 device, constant arguments passed in typed parameters are inlined as constants when this feature is enabled. This can improve compilation time in some cases, but has some unintuitive semantic implications.

dml12_int

> This compatibility feature affects many semantic details of integer arithmetic. When this feature is enabled, DMLC translates most integer operations directly into C, without compensating for DML-specifics, like the support for odd-sized `uint`*`NN`* types; this can sometimes have unexpected consequences. The most immediate effect of disabling this feature is that DMLC will report errors on statements like `assert 0;` and `while (1) { ... }`, which need to change into `assert false;` and `while (true) { ... }`, respectively. Other effects include:
>
> - Integers of non-standard sizes are represented as a native C type, e.g. `uint5` is represented as `uint8`, allowing it to store numbers too large to fit in 5 bits. With modern DML semantics, arithmetic is done on 64-bit integers and bits are truncated if casting or storing in a smaller type.
>
>   Old code sometimes relies on this feature by comparing variables of type `int1` to the value `1`. In DML 1.4, the only values of type `int1` are `0` and `-1`, so such code should be

rewritten to use the `uint1` type. It can be a good idea to grep for `[^a-z_]int1[^0-9]` and review if `uint1` is a better choice.

- Some operations have undefined behaviour in C, which is inherited by traditional DML 1.2. In modern DML this is well-defined, e.g., an unconditional critical error on negative shift or division by zero, and truncation on too large shift operands or signed shift overflow.

- Comparison operators `<`, `<=`, `==`, `>=`, `>` inherit C semantics in traditional DML, whereas in modern DML they are compared as integers. This sometimes makes a difference when comparing signed and unsigned numbers; in particular, `-1 != 0xffffffffffffffff` consistently in modern DML, whereas with compatibility semantics, they are consiered different only if both are constant.

The `dml12_int` feature only applies to DML 1.2 files; if a DML 1.4 file is imported from a DML 1.2 file, then modern DML semantics is still used for operations in that file.

dml12_misc

This compatibility feature preserves a number of minor language quirks that were originally in DML 1.2, but were cleaned up in DML 1.4. When this feature is enabled, DML 1.2 will permit the following:

- `sizeof(typename)` (see `WSIZEOFTYPE`)
- the `typeof` operator on an expression that isn't an lvalue
- `select` statements over `vect` types
- Passing a string literal in a (non-`const`) `char *` method argument
- Using the character `-` in the `c_name` parameter of `interface` objects
- Using the `c_name` parameter to override interface type in `implement` objects
- `loggroup` identifiers are accessible under the same name in generated C code
- Applying the `&` operator on something that isn't an lvalue (typically gives broken C code)
- `extern` statements that do not specify a type (`extern foo;`)
- Anonymous banks (`bank { ... }`)
- Unused templates may instantiate non-existing templates
- The same symbol may be used both for a top-level object (`$` scope) and a top-level symbol (non-`$` scope, e.g. `extern`, `constant` or `loggroup`)

io_memory

The `transaction` interface was introduced in 6, and will eventually replace the `io_memory` interface. When this feature is enabled, the top-level parameter `use_io_memory` defaults to `true`, causing `bank` objects to implement `io_memory` instead of `transaction` by default.

shared_logs_on_device

This compatibility feature changes the semantics of log statements inside shared methods so that they always log on the device object, instead of the nearest enclosing configuration object like with non-shared methods. This behaviour was a bug present since the very introduction of shared methods, which has lead to plenty of script code having become

reliant on it, especially in regards to how banks log. This feature preserves the bugged behaviour.

suppress_WLOGMIXUP

This compatibility feature makes it so the warning `WLOGMIXUP` is suppressed by default. `WLOGMIXUP` warns about usages of a common faulty pattern which results in broken log statements — for more information, see the documentation of `WLOGMIXUP` in the Messages section.

`WLOGMIXUP` is suppressed by default below Simics API version 7 in order to avoid overwhelming users with warnings, as the faulty pattern that `WLOGMIXUP` reports is very prevalent within existing code. Addressing applications of the faulty pattern should be done before or as part of migration to Simics API version 7.

Passing `--no-compat=suppress_WLOGMIXUP` to DMLC has almost the same effect as passing `--warn=WLOGMIXUP`; either will cause DMLC to report the warning even when the Simics API version in use is below 7. The only difference between these two options is that if `--no-compat=suppress_WLOGMIXUP` is used (and `--warn=WLOGMIXUP` is not), then `WLOGMIXUP` may still be explicitly suppressed via `--no-warn=WLOGMIXUP`. In contrast, `--warn=WLOGMIXUP` doesn't allow for `WLOGMIXUP` to be suppressed at all.

C.3.2 Features available up to and including --simics-api=7

These features correspond to functionality removed when compiling using Simics API 8 or newer. With older Simics API versions, these features can be disabled individually by passing `--no-compat=`*`TAG`* to the `dmlc` compiler.

broken_conditional_is

This compatibility feature prevents DML from reporting errors when instantiating a template within an `#if` block:

```
#if (true) {
    group g {
        // should be an error, but silently ignored when this
        // feature is enabled
        is nonexisting_template;
    }
}
```

Up to Simics 7, a bug prevented DMLC from reporting an error; this feature exists to preserve that behaviour.

broken_unused_types

This compatibility feature prevents DML from reporting errors on unused `extern`-declared types:

```
extern typedef struct {
    undefined_type_t member;
} never_used_t;
```

Up to Simics 7, a bug prevented DMLC from reporting an error; this feature exists to preserve that behaviour.

experimental_vect

This compat feature controls how DMLC reacts to uses of the `vect` syntax in files where the `simics_util_vect` provisional feature is not enabled.

When the `experimental_vect` compatibility feature is enabled, such uses are permitted, and give a `WEXPERIMENTAL` warning in DML 1.4 (but no warning in DML 1.2). When `experimental_vect` is disabled, DMLC forbids the `vect` syntax.

function_in_extern_struct

This compatibility feature enables a traditionally allowed syntax for function pointer members of `extern typedef struct` declarations, where the `*` is omitted in the pointer type. When disabling this feature, any declarations on this form:

```
extern typedef struct {
    void m(conf_object_t *);
} my_interface_t;
```

need to be changed to the standard C form:

```
extern typedef struct {
    void (*m)(conf_object_t *);
} my_interface_t;
```

legacy_attributes

This compatibility feature makes DMLC register all attributes using the legacy `SIM_register_typed_attribute` API function instead of the modern `SIM_register_attribute` family of API functions.

Disabling this feature will make the dictionary attribute type ("D" in type strings) to become unsupported, and any usage of it rejected by Simics. Thus, when migrating away from this feature, any attribute of the model that leverages dictionary values should be changed to leverage a different representation. In general, any dictionary can instead be represented by a list of two-element lists, e.g. `[[X,Y]*]`, where *X* describes the type of keys, and *Y* describes the type of values.

lenient_typechecking

This compatibility feature makes DMLC's type checking very inexact and lenient in multiple respects when compared to GCC's type checking of the generated C. This discrepency mostly affects method overrides or uses of `extern`:d C macros, because in those scenarios DMLC can become wholly responsible for proper type checking.

While migrating away from this feature, the most common type errors that its disablement introduces are due to discrepencies between pointer types. In particular, implicitly discarding `const`-qualification of a pointer's base type will never be tolerated, and `void` pointers are only considered equivalent with any other pointer type in the same contexts as C.

Novel type errors from uses of `extern`:d macros can often be resolved by changing the signature of the `extern` declaration to more accurately reflect the macro's effective type.

meaningless_log_levels

The log level that may be specified for logs of kind "warning", "error" or "critical" typically must be 1, and any subsequent log level must typically be 5. This compatibility feature makes it so either log level may be any integer between 1 and 4 for these log kinds. The primary log level is always treated as 1, and any other value than 1 for the subsequent log level will be treated as 5 (that is, the log will only happen once)

no_method_index_asserts

This compatibility feature makes it so that methods defined under object arrays don't implicitly assert that the indices used to reference the object array when calling the method are in bounds.

Migrating away from this compatibility feature should be a priority. If its disablement makes the simulation crash due to an assertion failing, then that **definitely signifies a bug in your model; a bug that would very likely result in memory corruption if the assertion were not to be made.**

optional_version_statement

When this compatibility feature is enabled, the version specification statement (`dml 1.4;`) statement at the start of each file is optional (but the compiler warns if it is omitted). Also, `dml 1.3;` is permitted as a deprecated alias for `dml 1.4;`, with a warning.

port_proxy_attrs

In Simics 5, configuration attributes for `connect`, `attribute` and `register` objects inside banks and ports were registered on the device object, named like *`bankname_attrname`*. Such proxy attributes are only created When this feature is enabled. Proxy attributes are not created for all banks and ports, in the same manner as documented in the `port_proxy_ifaces` feature.

port_proxy_ifaces

Version 5 and earlier of Simics relied on interface ports (as registered by the `SIM_register_port_interface` API function) for exposing the interfaces of ports and banks. In newer versions of Simics, interfaces are instead exposed on separate configuration objects. When this feature is enabled, old-style interface ports are created as proxies to the interfaces on the respective port objects. Such proxies are not created for all banks and ports; e.g., banks inside groups were not allowed in Simics 5, so such banks do not need proxies for backward compatibility.

warning_statement

This compatibility feature enables the `_warning` statement. This turned out to not be very useful, so in Simics API 8 and newer the feature is no longer allowed.

B Provisional language features | D Changes from DML 1.2 to DML 1.4

D Changes from DML 1.2 to DML 1.4

DML 1.4 is in general very similar to DML 1.2 in syntax and semantics. However, some key differences do exist which we will highlight here.

D.1 Toy DML 1.2 device

```
dml 1.2;

device example;

method foo(a) -> (int b, bool c) nothrow {
    b = a;
    c = false;
    if (a == 0) {
        c = true;
    }
}

method bar(int a) {
    if (a == 0) {
        throw;
    }
}

// Bank
bank b {
    register r[20] size 4 @ 0x0000 + $i * 4 is read {
        method after_read(memop) {
            local int b;
            inline $foo(2) -> (b, $c);
        }

        field f [7:0];

        data bool c;
    }
}

template t1 {
    parameter p default 1;
}

template t2 {
    parameter p = 2;
}

attribute a is (t1, t2) {
    parameter allocate_type = "int32";

    method after_set() {
        call $bar($this);
        $this = $p;
    }
}
```

D.2 The corresponding DML 1.4 device

```
dml 1.4;

device example;

inline method foo(inline a) -> (int, bool) /* b, c */ {
    local int b;
    local bool c;
    b = a;
    c = false;
    #if (a == 0) {
       c = true;
    }
    return (b, c);
}

method bar(int a) throws {
    if (a == 0) {
        throw;
    }
}

// Bank
bank b {
    register r[i < 20] size 4 @ 0x0000 + i * 4 is read_register {
        method read_register(uint64 enabled_bytes, void *aux) -> (uint64) {
            local uint64 value = default(enabled_bytes, aux);
            local int b;
            (b, c) = foo(2);
            return value;
        }

        field f @ [7:0];

        session bool c;
    }
}

template t1 {
    param p default 1;
}

template t2 is t1 {
    param p = 2;
}

attribute a is (int64_attr, t1, t2) {

    method set(attr_value_t value) throws {
        default(value);
        this.val = p;
```

```
    }
}
```

D.3 Key Differences

```
dml 1.4;
```

In very DML 1.4 file, this must be the first declaration.

```
inline method foo(inline a) ...
```

The syntax for inlining a method call has changed. It is now strictly an attribute of the method being *called*, declaring itself to be inline and which arguments are inline.

```
... -> (int, bool) /* b, c */
```

Return values are no longer named in method declarations, rather only their types need to be declared. This also means they are not inherently available as variables in the method body scope.

```
local int b;
local bool c;
```

Previously, these variables were declared as return values. They are now declared as locals instead.

```
#if (a == 0) {
```

The `#if` syntax is useful to do compile-time evaluation of constants, in DML 1.2 this would be done with regular `if`.

```
return (b, c);
```

In DML 1.4 you must explicitly return the return values.

```
method bar(int a) throws
```

A method that might throw must be annotated with the `throws` keyword.

```
register r[i < 20] ...
```

The syntax for object arrays has changed. The index name can no longer be implicit, and the range syntax is now `[index < size]`.

```
.., @ 0x0000 + i * 4 ...
```

The $ prefix on variables has been removed. See Backward Incompatible changes for the variable scope implications of this.

```
... is read_register {
   method read_register(uint64 enabled_bytes, void *aux) -> (uint64) {
            local uint64 value = default(enabled_bytes, aux);
            ...
   }
```

The API for the builtin library has changed. In this case the `after_read` method has been replaced, and instead the `read_register` method must be overridden in order to call its `default` implementation and then implement the after read logic.

```
field f  @ [7:0];
```

The syntax for field declarations has changed, you must now specify an `@` before the bit range of the field.

```
session bool c;
```

Data declarations have been replaced with session declarations.

```
template t1 {
    param p default 1;
}

template t2 is t1 {
    param p = 2;
}

attribute a is (int64_attr, t1, t2) {
```

The parameter override behaviour has been made stricter. In the case where multiple declarations of a parameter conflict DML will not pick a particular declaration if it is the only non-default one, rather it will check the template hierarchy to determine which declarations override which.

```
method set(attr_value_t value) throws {
        default(value);
```

Similar to above for registers, the `after_set` method of attributes is no more. Instead the set method must be overridden and `default` called.

```
this.val = p;
```

Registers, fields, and attributes are no longer proper values in DML 1.4, their session variable `.val` must be accessed, or a `get`, or `set` method be called.

- D.4 Porting DML 1.2 to DML 1.4
- D.5 Language differences handled by the port-dml script

- D.6 Backward incompatible changes, not automatically converted

C Managing deprecated language features | D.4 Porting DML 1.2 to DML 1.4

E Formal Grammar

dml →
    maybe_provisional maybe_device maybe_bitorder device_statements
maybe_provisional →
    provisional *ident_list* "**;**"
    | <empty>
maybe_device →
    device *objident* "**;**"
    | <empty>
maybe_bitorder →
    <empty>
    | **bitorder** *ident* "**;**"
device_statements →
    device_statements device_statement
    | <empty>
device_statement →
    toplevel
    | *object*
    | *toplevel_param*
    | *method*
    | *bad_shared_method*
    | *istemplate* "**;**"
    | *toplevel_if*
    | *error_stmt*
    | *in_each*
toplevel_param →
    param
toplevel_if →
    hashif "**(**" *expression* "**)**" "**{**" *device_statements* "**}**" *toplevel_else*
toplevel_else →
    <empty>
    | *hashelse* "**{**" *device_statements* "**}**"
    | *hashelse toplevel_if*
array_list →
    <empty>
    | *array_list* "**[**" *arraydef* "**]**"
object →
    register *objident array_list sizespec offsetspec maybe_istemplate object_spec*
bitrangespec →
    "**@**" *bitrange*
    | <empty>
object →
    field *objident array_list bitrangespec maybe_istemplate object_spec*
bitrange →

"**[**" *expression* "**]**"
| "**[**" *expression* "**:**" *expression* "**]**"

data →
session

object →
session_decl

session_decl →
data named_cdecl "**;**"
| *data named_cdecl* "**=**" *initializer* "**;**"
| *data* "**(**" *cdecl_list_nonempty* "**)**" "**;**"
| *data* "**(**" *cdecl_list_nonempty* "**)**" "**=**" *initializer* "**;**"

object →
saved_decl

saved_decl →
saved *named_cdecl* "**;**"
| **saved** *named_cdecl* "**=**" *initializer* "**;**"
| **saved** "**(**" *cdecl_list_nonempty* "**)**" "**;**"
| **saved** "**(**" *cdecl_list_nonempty* "**)**" "**=**" *initializer* "**;**"

object →
connect *objident array_list maybe_istemplate object_spec*
| **interface** *objident array_list maybe_istemplate object_spec*
| **attribute** *objident array_list maybe_istemplate object_spec*
| **bank** *objident array_list maybe_istemplate object_spec*
| **event** *objident array_list maybe_istemplate object_spec*
| **group** *objident array_list maybe_istemplate object_spec*
| **port** *objident array_list maybe_istemplate object_spec*
| **implement** *objident array_list maybe_istemplate object_spec*
| **subdevice** *objident array_list maybe_istemplate object_spec*

maybe_default →
default
| <empty>

method →
method_qualifiers **method** *objident method_params_typed maybe_default compound_statement*
| **inline method** *objident method_params_maybe_untyped maybe_default compound_statement*

arraydef →
ident "**<**" *expression*
| *ident* "**<**" *"..."*

template_stmts →
<empty>
| *template_stmts template_stmt*

template_stmt →
object_statement_or_typedparam
| **shared** *method_qualifiers* **method** *shared_method*
| **shared** *hook_decl*

method_qualifiers →
<empty>
| **independent**
| **independent startup**
| **independent startup memoized**

shared_method →
ident method_params_typed "**;**"
| *ident method_params_typed* **default** *compound_statement*

| *ident method_params_typed compound_statement*

toplevel →

template *objident maybe_istemplate* "**{**" *template_stmts* "**}**"
| **header** "**%{ ... %}**"
| **footer** "**%{ ... %}**"
| **_header** "**%{ ... %}**"
| **loggroup** *ident* "**;**"
| **constant** *ident* "**=**" *expression* "**;**"
| **extern** *cdecl* "**;**"
| **typedef** *named_cdecl* "**;**"
| **extern typedef** *named_cdecl* "**;**"
| **import** *utf8_sconst* "**;**"

object_desc →

composed_string_literal
| <empty>

object_spec →

object_desc "**;**"
| *object_desc* "**{**" *object_statements* "**}**"

object_statements →

object_statements object_statement
| <empty>

object_statement →

object_statement_or_typedparam
| *bad_shared_method*

bad_shared_method →

shared *method_qualifiers* **method** *shared_method*

toplevel →

export *expression* **as** *expression* "**;**"

object_statement_or_typedparam →

object
| *param*
| *method*
| *istemplate* "**;**"
| *object_if*
| *error_stmt*
| *in_each*

in_each →

in each *istemplate_list* "**{**" *object_statements* "**}**"

hashif →

"**#if**"
| **if**

hashelse →

"**#else**"
| **else**

object_if →

hashif "**(**" *expression* "**)**" "**{**" *object_statements* "**}**" *object_else*

object_else →

<empty>
| *hashelse* "**{**" *object_statements* "**}**"
| *hashelse object_if*

param →

param *objident paramspec_maybe_empty*
| **param** *objident* **auto** "**;**"

| **param** *objident* "**:**" *paramspec*
| **param** *objident* "**:**" *ctypedecl paramspec_maybe_empty*

paramspec_maybe_empty →

paramspec
| "**;**"

paramspec →

"**=**" *expression* "**;**"
| **default** *expression* "**;**"

method_outparams →

<empty>
| "**->**" "**(**" *cdecl_list* "**)**"

method_params_maybe_untyped →

"**(**" *cdecl_or_ident_list* "**)**" *method_outparams throws*

method_params_typed →

"**(**" *cdecl_list* "**)**" *method_outparams throws*

throws →

throws
| <empty>

maybe_istemplate →

<empty>
| *istemplate*

istemplate →

is *istemplate_list*

istemplate_list →

objident
| "**(**" *objident_list* "**)**"

sizespec →

size *expression*
| <empty>

offsetspec →

"**@**" *expression*
| <empty>

cdecl_or_ident →

named_cdecl
| **inline** *ident*

named_cdecl →

cdecl

cdecl →

basetype cdecl2
| **const** *basetype cdecl2*

basetype →

typeident
| *struct*
| *layout*
| *bitfields*
| *typeof*
| **sequence** "**(**" *typeident* "**)**"
| **hook** "**(**" *cdecl_list* "**)**"

cdecl2 →

cdecl3
| **const** *cdecl2*
| "*****" *cdecl2*
| **vect** *cdecl2*

cdecl3 →

ident
| <empty>
| *cdecl3* "**[**" *expression* "**]**"
| *cdecl3* "**(**" *cdecl_list_opt_ellipsis* "**)**"
| "**(**" *cdecl2* "**)**"

cdecl_list →
<empty>
| *cdecl_list_nonempty*

cdecl_list_nonempty →
cdecl
| *cdecl_list_nonempty* "**,**" *cdecl*

cdecl_list_opt_ellipsis →
cdecl_list
| *cdecl_list_ellipsis*

cdecl_list_ellipsis →
"..."
| *cdecl_list_nonempty* "**,**" *"..."*

cdecl_or_ident_list →
<empty>
| *cdecl_or_ident_list2*

cdecl_or_ident_list2 →
cdecl_or_ident
| *cdecl_or_ident_list2* "**,**" *cdecl_or_ident*

typeof →
typeof *expression*

struct →
struct "**{**" *struct_decls* "**}**"

struct_decls →
struct_decls named_cdecl "**;**"
| <empty>

layout_decl →
layout *utf8_sconst* "**{**" *layout_decls* "**}**"

layout →
layout_decl

layout_decls →
layout_decls named_cdecl "**;**"
| <empty>

bitfields →
bitfields *integer-literal* "**{**" *bitfields_decls* "**}**"

bitfields_decls →
bitfields_decls named_cdecl "**@**" "**[**" *bitfield_range* "**]**" "**;**"

bitfield_range →
expression
| *expression* "**:**" *expression*

bitfields_decls →
<empty>

ctypedecl →
const_opt basetype ctypedecl_ptr

ctypedecl_ptr →
stars ctypedecl_array

stars →
<empty>
| "*" **const** *stars*
| "*" *stars*

ctypedecl_array →
    ctypedecl_simple
ctypedecl_simple →
    "**(**" *ctypedecl_ptr* "**)**"
    | <empty>
const_opt →
    const
    | <empty>
typeident →
    ident
    | **char**
    | **double**
    | **float**
    | **int**
    | **long**
    | **short**
    | **signed**
    | **unsigned**
    | **void**
    | **register**
assignop →
    expression "**+=**" *expression*
    | *expression* "**-=**" *expression*
    | *expression* "***=**" *expression*
    | *expression* "**/=**" *expression*
    | *expression* "**%=**" *expression*
    | *expression* "**|=**" *expression*
    | *expression* "**&=**" *expression*
    | *expression* "**^=**" *expression*
    | *expression* "**<<=**" *expression*
    | *expression* "**>>=**" *expression*
expression →
    expression "**?**" *expression* "**:**" *expression*
    | *expression* **#?** *expression* **#:** *expression*
    | *expression* "**+**" *expression*
    | *expression* "**-**" *expression*
    | *expression* "*****" *expression*
    | *expression* "**/**" *expression*
    | *expression* "**%**" *expression*
    | *expression* "**<<**" *expression*
    | *expression* "**>>**" *expression*
    | *expression* "**==**" *expression*
    | *expression* "**!=**" *expression*
    | *expression* "**<**" *expression*
    | *expression* "**>**" *expression*
    | *expression* "**<=**" *expression*
    | *expression* "**>=**" *expression*
    | *expression* "**||**" *expression*
    | *expression* "**&&**" *expression*
    | *expression* "**|**" *expression*
    | *expression* "**^**" *expression*
    | *expression* "**&**" *expression*
    | **cast** "**(**" *expression* "**,**" *ctypedecl* "**)**"
    | **sizeof** *expression*

| "**-**" *expression*
| "**+**" *expression*
| "**!**" *expression*
| "**~**" *expression*
| "**&**" *expression*
| "*****" *expression*
| **defined** *expression*
| **stringify** "**(**" *expression* "**)**"
| "**++**" *expression*
| "**--**" *expression*
| *expression* "**++**"
| *expression* "**--**"
| *expression* "**(**" "**)**"
| *expression* "**(**" *single_initializer_list* "**)**"
| *expression* "**(**" *single_initializer_list* "**,**" "**)**"
| *integer-literal*
| *hex-literal*
| *binary-literal*
| *char-literal*
| *float-literal*
| *string-literal*

utf8_sconst →

string-literal

expression →

undefined
| *objident*
| **default**
| **this**
| *expression* "**.**" *objident*
| *expression* "**->**" *objident*
| **sizeoftype** *typeoparg*

typeoparg →

ctypedecl
| "**(**" *ctypedecl* "**)**"

expression →

new *ctypedecl*
| **new** *ctypedecl* "**[**" *expression* "**]**"
| "**(**" *expression* "**)**"
| "**[**" *expression_list* "**]**"
| *expression* "**[**" *expression* "**]**"
| *expression* "**[**" *expression* "**,**" *identifier* "**]**"
| *expression* "**[**" *expression* "**:**" *expression endianflag* "**]**"
| **each** *objident* **in** "**(**" *expression* "**)**"

endianflag →

"**,**" *identifier*
| <empty>

expression_opt →

expression
| <empty>

expression_list →

<empty>
| *expression*
| *expression* "**,**" *expression_list*

expression_list_ntc_nonempty →

expression
| *expression* "**,**" *expression_list_ntc_nonempty*

composed_string_literal →
utf8_sconst
| *composed_string_literal* "**+**" *utf8_sconst*

bracketed_string_literal →
composed_string_literal
| "**(**" *composed_string_literal* "**)**"

single_initializer →
expression
| "**{**" *single_initializer_list* "**}**"
| "**{**" *single_initializer_list* "**,**" "**}**"

initializer →
single_initializer
| "**(**" *single_initializer* "**,**" *single_initializer_list* "**)**"

single_initializer_list →
single_initializer
| *single_initializer_list* "**,**" *single_initializer*

single_initializer →
"**{**" *designated_struct_initializer_list* "**}**"
| "**{**" *designated_struct_initializer_list* "**,**" "**}**"
| "**{**" *designated_struct_initializer_list* "**,**" *"..."* "**}**"

designated_struct_initializer →
"**.**" *ident* "**=**" *single_initializer*

designated_struct_initializer_list →
designated_struct_initializer
| *designated_struct_initializer_list* "**,**" *designated_struct_initializer*

statement →
statement_except_hashif

statement_except_hashif →
compound_statement
| *local* "**;**"
| *assign_stmt* "**;**"
| *assignop* "**;**"

assign_stmt →
assign_chain
| *tuple_literal* "**=**" *initializer*

assign_chain →
expression "**=**" *assign_chain*
| *expression* "**=**" *initializer*

tuple_literal →
"**(**" *expression* "**,**" *expression_list_ntc_nonempty* "**)**"

statement_except_hashif →
"**;**"
| *expression* "**;**"
| **if** "**(**" *expression* "**)**" *statement*
| **if** "**(**" *expression* "**)**" *statement* **else** *statement*

statement →
"**#if**" "**(**" *expression* "**)**" *statement*
| "**#if**" "**(**" *expression* "**)**" *statement* "**#else**" *statement*

statement_except_hashif →
while "**(**" *expression* "**)**" *statement*
| **do** *statement* **while** "**(**" *expression* "**)**" "**;**"

for_post →

<empty>
| *for_post_nonempty*

for_post_nonempty →

for_post_one
| *for_post_nonempty* "**,**" *for_post_one*

for_post_one →

assign_stmt
| *assignop*
| *expression*

for_pre →

local
| *for_post*

statement_except_hashif →

for "**(**" *for_pre* "**;**" *expression_opt* "**;**" *for_post* "**)**" *statement*
| **switch** "**(**" *expression* "**)**" "**{**" *stmt_or_case_list* "**}**"

stmt_or_case →

statement_except_hashif
| *cond_case_statement*
| *case_statement*

cond_case_statement →

"**#if**" "**(**" *expression* "**)**" "**{**" *stmt_or_case_list* "**}**"
| "**#if**" "**(**" *expression* "**)**" "**{**" *stmt_or_case_list* "**}**" "**#else**" "**{**" *stmt_or_case_list* "**}**"

stmt_or_case_list →

<empty>
| *stmt_or_case_list stmt_or_case*

statement_except_hashif →

delete *expression* "**;**"
| **try** *statement* **catch** *statement*
| **after** *expression identifier* "**:**" *expression* "**;**"

ident_list →

<empty>
| *nonempty_ident_list*

nonempty_ident_list →

ident
| *nonempty_ident_list* "**,**" *ident*

statement_except_hashif →

after *expression* "**->**" "**(**" *ident_list* "**)**" "**:**" *expression* "**;**"
| **after** *expression* "**->**" *ident* "**:**" *expression* "**;**"
| **after** *expression* "**:**" *expression* "**;**"
| **after** "**:**" *expression* "**;**"
| **assert** *expression* "**;**"

log_kind →

identifier
| **error**

log_level →

expression **then** *expression*
| *expression*

statement_except_hashif →

log *log_kind* "**,**" *log_level* "**,**" *expression* "**:**" *bracketed_string_literal log_args* "**;**"
| **log** *log_kind* "**,**" *log_level* "**:**" *bracketed_string_literal log_args* "**;**"
| **log** *log_kind* "**:**" *bracketed_string_literal log_args* "**;**"

hashselect →

"**#select**"

statement_except_hashif →

hashselect ident **in** "**(**" *expression* "**)**" **where** "**(**" *expression* "**)**" *statement hashelse statement*
| **foreach** *ident* **in** "**(**" *expression* "**)**" *statement*
| "**#foreach**" *ident* **in** "**(**" *expression* "**)**" *statement*

case_statement →
case *expression* "**:**"
| **default** "**:**"

statement_except_hashif →
goto *ident* "**;**"
| **break** "**;**"
| **continue** "**;**"
| **throw** "**;**"
| **return** "**;**"
| **return** *initializer* "**;**"
| *error_stmt*

error_stmt →
error "**;**"
| **error** *bracketed_string_literal* "**;**"

statement_except_hashif →
warning_stmt

warning_stmt →
_warning *bracketed_string_literal* "**;**"

log_args →
<empty>
| *log_args* "**,**" *expression*

compound_statement →
"**{**" *statement_list* "**}**"

statement_list →
<empty>
| *statement_list statement*

local_keyword →
local

static →
session

local_decl_kind →
local_keyword
| *static*

local →
local_decl_kind cdecl
| **saved** *cdecl*
| *local_decl_kind cdecl* "**=**" *initializer*
| **saved** *cdecl* "**=**" *initializer*
| *local_decl_kind* "**(**" *cdecl_list_nonempty* "**)**"
| **saved** "**(**" *cdecl_list_nonempty* "**)**"
| *local_decl_kind* "**(**" *cdecl_list_nonempty* "**)**" "**=**" *initializer*
| **saved** "**(**" *cdecl_list_nonempty* "**)**" "**=**" *initializer*

simple_array_list →
<empty>
| *simple_array_list* "**[**" *expression* "**]**"

hook_decl →
hook "**(**" *cdecl_list* "**)**" *ident simple_array_list* "**;**"

object →
hook_decl

objident_list →

objident
| *objident_list* "**,**" *objident*

objident →
ident
| **register**

ident →
attribute
| **bank**
| **bitorder**
| **connect**
| **constant**
| **data**
| **device**
| **event**
| **field**
| **footer**
| **group**
| **header**
| **implement**
| **import**
| **interface**
| **loggroup**
| **method**
| **port**
| **size**
| **subdevice**
| **nothrow**
| **then**
| **throws**
| **_header**
| **provisional**
| **param**
| **saved**
| **independent**
| **startup**
| **memoized**
| *identifier*
| **class**
| **enum**
| **namespace**
| **private**
| **protected**
| **public**
| **restrict**
| **union**
| **using**
| **virtual**
| **volatile**
| **call**
| **auto**
| **static**
| **select**
| **async**

| await
| with

D.6 Backward incompatible changes, not automatically converted