

Utility and Field Macros for Components and Objects

Summary

Utility and Field Macros for Components and Objects

UTILITY MACROS

```
`uvm_field_utils_begin
`uvm_field_utils_end

`uvm_object_utils
`uvm_object_param_utils
`uvm_object_utils_begin
`uvm_object_param_utils_begin
`uvm_object_utils_end

`uvm_component_utils
`uvm_component_param_utils
`uvm_component_utils_begin
`uvm_component_param_utils_begin
`uvm_component_end

`uvm_object_registry

`uvm_component_registry
```

FIELD MACROS

`UVM_FIELD_* MACROS

```
`uvm_field_int
`uvm_field_object
`uvm_field_string
`uvm_field_enum
`uvm_field_real
`uvm_field_event
```

The *utils* macros define the infrastructure needed to enable the object/component for correct factory operation.

These macros form a block in which ``uvm_field_*` macros can be placed.

uvm_object-based class declarations may contain one of the above forms of utility macros.

uvm_component-based class declarations may contain one of the above forms of utility macros. Register a **uvm_object**-based class with the factory
Registers a **uvm_component**-based class with the factory

The ``uvm_field_*` macros are invoked inside of the ``uvm_*_utils_begin` and ``uvm_*_utils_end` macro blocks to form "automatic" implementations of the core data methods: copy, compare, pack, unpack, record, print, and sprint.

Macros that implement data operations for scalar properties.

Implements the data operations for any packed integral property.
Implements the data operations for a **uvm_object**-based property.
Implements the data operations for a string property.
Implements the data operations for an enumerated property.
Implements the data operations for any real property.
Implements the data operations for

`UVM_FIELD_SARRAY_* MACROS``uvm_field_sarray_int```uvm_field_sarray_object```uvm_field_sarray_string```uvm_field_sarray_enum`**`UVM_FIELD_ARRAY_* MACROS**``uvm_field_array_int```uvm_field_array_object```uvm_field_array_string```uvm_field_array_enum`**`UVM_FIELD_QUEUE_* MACROS**``uvm_field_queue_int```uvm_field_queue_object```uvm_field_queue_string```uvm_field_queue_enum`**`UVM_FIELD_AA_*_STRING MACROS**``uvm_field_aa_int_string```uvm_field_aa_object_string```uvm_field_aa_string_string`**`UVM_FIELD_AA_*_INT MACROS**

an event property.

Macros that implement data operations for one-dimensional static array properties.

Implements the data operations for a one-dimensional static array of integrals.

Implements the data operations for a one-dimensional static array of `uvm_object`-based objects.

Implements the data operations for a one-dimensional static array of strings.

Implements the data operations for a one-dimensional static array of enums.

Macros that implement data operations for one-dimensional dynamic array properties.

Implements the data operations for a one-dimensional dynamic array of integrals.

Implements the data operations for a one-dimensional dynamic array of `uvm_object`-based objects.

Implements the data operations for a one-dimensional dynamic array of strings.

Implements the data operations for a one-dimensional dynamic array of enums.

Macros that implement data operations for dynamic queues.

Implements the data operations for a queue of integrals.

Implements the data operations for a queue of `uvm_object`-based objects.

Implements the data operations for a queue of strings.

Implements the data operations for a one-dimensional queue of enums.

Macros that implement data operations for associative arrays indexed by *string*.

Implements the data operations for an associative array of integrals indexed by *string*.

Implements the data operations for an associative array of `uvm_object`-based objects indexed by *string*.

Implements the data operations for an associative array of strings indexed by *string*.

Macros that implement data operations for associative arrays

``uvm_field_aa_object_int`

indexed by an integral type.

Implements the data operations for an associative array of **uvm_object**-based objects indexed by the *int* data type.

``uvm_field_aa_int_int`

Implements the data operations for an associative array of integral types indexed by the *int* data type.

``uvm_field_aa_int_int_unsigned`

Implements the data operations for an associative array of integral types indexed by the *int unsigned* data type.

``uvm_field_aa_int_integer`

Implements the data operations for an associative array of integral types indexed by the *integer* data type.

``uvm_field_aa_int_integer_unsigned`

Implements the data operations for an associative array of integral types indexed by the *integer unsigned* data type.

``uvm_field_aa_int_byte`

Implements the data operations for an associative array of integral types indexed by the *byte* data type.

``uvm_field_aa_int_byte_unsigned`

Implements the data operations for an associative array of integral types indexed by the *byte unsigned* data type.

``uvm_field_aa_int_shortint`

Implements the data operations for an associative array of integral types indexed by the *shortint* data type.

``uvm_field_aa_int_shortint_unsigned`

Implements the data operations for an associative array of integral types indexed by the *shortint unsigned* data type.

``uvm_field_aa_int_longint`

Implements the data operations for an associative array of integral types indexed by the *longint* data type.

``uvm_field_aa_int_longint_unsigned`

Implements the data operations for an associative array of integral types indexed by the *longint unsigned* data type.

``uvm_field_aa_int_key`

Implements the data operations for an associative array of integral types indexed by any integral key data type.

``uvm_field_aa_int_enumkey`

Implements the data operations for an associative array of integral types indexed by any enumeration key data type.

RECORDING MACROS

The recording macros assist users who implement the **uvm_object::do_record** method.

``uvm_record_attribute`

Vendor-independent macro to hide tool-specific interface for recording attributes (fields) to a transaction database.

``uvm_record_int`

```
`uvm_record_string
`uvm_record_time
`uvm_record_real
`uvm_record_field
```

Macro for recording arbitrary name-value pairs into a transaction recording database.

PACKING MACROS

The packing macros assist users who implement the `uvm_object::do_pack` method.

PACKING - WITH SIZE INFO

```
`uvm_pack_intN
`uvm_pack_enumN
`uvm_pack_sarrayN
`uvm_pack_arrayN
`uvm_pack_queueN
```

Pack an integral variable.
Pack an integral variable.
Pack a static array of integrals.
Pack a dynamic array of integrals.
Pack a queue of integrals.

PACKING - NO SIZE INFO

```
`uvm_pack_int
`uvm_pack_enum
`uvm_pack_string
`uvm_pack_real
`uvm_pack_sarray
`uvm_pack_array
`uvm_pack_queue
```

Pack an integral variable without having to also specify the bit size.
Pack an enumeration value.
Pack a string variable.
Pack a variable of type real.
Pack a static array without having to also specify the bit size of its elements.
Pack a dynamic array without having to also specify the bit size of its elements.
Pack a queue without having to also specify the bit size of its elements.

UNPACKING MACROS

The unpacking macros assist users who implement the `uvm_object::do_unpack` method.

UNPACKING - WITH SIZE INFO

```
`uvm_unpack_intN
`uvm_unpack_enumN
`uvm_unpack_sarrayN
`uvm_unpack_arrayN
`uvm_unpack_queueN
```

Unpack into an integral variable.
Unpack enum of type *TYPE* into *VAR*.
Unpack a static (fixed) array of integrals.
Unpack into a dynamic array of integrals.
Unpack into a queue of integrals.

UNPACKING - NO SIZE INFO

```
`uvm_unpack_int
`uvm_unpack_enum
`uvm_unpack_string
`uvm_unpack_real
`uvm_unpack_sarray
`uvm_unpack_array
`uvm_unpack_queue
```

Unpack an integral variable without having to also specify the bit size.
Unpack an enumeration value, which requires its type be specified.
Unpack a string variable.
Unpack a variable of type real.
Unpack a static array without having to also specify the bit size of its elements.
Unpack a dynamic array without having to also specify the bit size of its elements.
Unpack a queue without having to

also specify the bit size of its elements.

UTILITY MACROS

The *utils* macros define the infrastructure needed to enable the object/component for correct factory operation. See ``uvm_object_utils`` and ``uvm_component_utils`` for details.

A *utils* macro should be used inside every user-defined class that extends `uvm_object` directly or indirectly, including `uvm_sequence_item` and `uvm_component`.

Below is an example usage of the *utils* macro for a user-defined object.

```
class mydata extends uvm_object;

    `uvm_object_utils(mydata)

    // declare data properties

    function new(string name="mydata_inst");
        super.new(name);
    endfunction

endclass
```

Below is an example usage of a *utils* macro for a user-defined component.

```
class my_comp extends uvm_component;

    `uvm_component_utils(my_comp)

    // declare data properties

    function new(string name, uvm_component parent=null);
        super.new(name,parent);
    endfunction

endclass
```

``uvm_field_utils_begin`

``uvm_field_utils_end`

These macros form a block in which ``uvm_field_*` macros can be placed. Used as

```
`uvm_field_utils_begin(TYPE)
    `uvm_field_* macros here
`uvm_field_utils_end
```

These macros do *not* perform factory registration nor implement the *get_type_name* and *create* methods. Use this form when you need custom implementations of these two

methods, or when you are setting up field macros for an abstract class (i.e. virtual class).

``uvm_object_utils`

``uvm_object_param_utils`

``uvm_object_utils_begin`

``uvm_object_param_utils_begin`

``uvm_object_utils_end`

`uvm_object`-based class declarations may contain one of the above forms of utility macros.

For simple objects with no field macros, use

```
`uvm_object_utils(TYPE)
```

For simple objects with field macros, use

```
`uvm_object_utils_begin(TYPE)
`uvm_field_* macro invocations here
`uvm_object_utils_end
```

For parameterized objects with no field macros, use

```
`uvm_object_param_utils(TYPE)
```

For parameterized objects, with field macros, use

```
`uvm_object_param_utils_begin(TYPE)
`uvm_field_* macro invocations here
`uvm_object_utils_end
```

Simple (non-parameterized) objects use the `uvm_object_utils*` versions, which do the following:

- Implements `get_type_name`, which returns `TYPE` as a string
- Implements `create`, which allocates an object of type `TYPE` by calling its constructor with no arguments. `TYPE`'s constructor, if defined, must have default values on all its arguments.

- Registers the TYPE with the factory, using the string TYPE as the factory lookup string for the type.
- Implements the static get_type() method which returns a factory proxy object for the type.
- Implements the virtual get_object_type() method which works just like the static get_type() method, but operates on an already allocated object.

Parameterized classes must use the uvm_object_param_utils* versions. They differ from ``uvm_object_utils` only in that they do not supply a type name when registering the object with the factory. As such, name-based lookup with the factory for parameterized classes is not possible.

The macros with _begin suffixes are the same as the non-suffixed versions except that they also start a block in which ``uvm_field_*` macros can be placed. The block must be terminated by ``uvm_object_utils_end`.

``uvm_component_utils`

``uvm_component_param_utils`

``uvm_component_utils_begin`

``uvm_component_param_utils_begin`

``uvm_component_end`

uvm_component-based class declarations may contain one of the above forms of utility macros.

For simple components with no field macros, use

```
`uvm_component_utils(TYPE)
```

For simple components with field macros, use

```
`uvm_component_utils_begin(TYPE)
`uvm_field_* macro invocations here
`uvm_component_utils_end
```

For parameterized components with no field macros, use

```
`uvm_component_param_utils(TYPE)
```

For parameterized components with field macros, use

```
`uvm_component_param_utils_begin(TYPE)
`uvm_field_* macro invocations here
`uvm_component_utils_end
```

Simple (non-parameterized) components must use the `uvm_components_utils*` versions, which do the following:

- Implements `get_type_name`, which returns `TYPE` as a string.
- Implements `create`, which allocates a component of type `TYPE` using a two argument constructor. `TYPE`'s constructor must have a name and a parent argument.
- Registers the `TYPE` with the factory, using the string `TYPE` as the factory lookup string for the type.
- Implements the static `get_type()` method which returns a factory proxy object for the type.
- Implements the virtual `get_object_type()` method which works just like the static `get_type()` method, but operates on an already allocated object.

Parameterized classes must use the `uvm_object_param_utils*` versions. They differ from ``uvm_object_utils` only in that they do not supply a type name when registering the object with the factory. As such, name-based lookup with the factory for parameterized classes is not possible.

The macros with `_begin` suffixes are the same as the non-suffixed versions except that they also start a block in which ``uvm_field_*` macros can be placed. The block must be terminated by ``uvm_component_utils_end`.

``uvm_object_registry`

Register a `uvm_object`-based class with the factory

```
`uvm_object_registry(T,S)
```

Registers a `uvm_object`-based class `T` and lookup string `S` with the factory. `S` typically is the name of the class in quotes. The ``uvm_object_utils` family of macros uses this macro.

``uvm_component_registry`

Registers a `uvm_component`-based class with the factory

```
`uvm_component_registry(T,S)
```

Registers a `uvm_component`-based class `T` and lookup string `S` with the factory. `S` typically is the name of the class in quotes. The ``uvm_object_utils` family of macros uses this macro.

FIELD MACROS

The ``uvm_field_*` macros are invoked inside of the ``uvm_*_utils_begin` and ``uvm_*_utils_end` macro blocks to form “automatic” implementations of the core data methods: copy, compare, pack, unpack, record, print, and sprint.

By using the macros, you do not have to implement any of the `do_*` methods inherited from [uvm_object](#). However, be aware that the field macros expand into general inline code that is not as run-time efficient nor as flexible as direct implementations of the `do_*` methods.

Below is an example usage of the field macros for a sequence item.

```
class my_trans extends uvm_sequence_item;

  cmd_t  cmd;
  int    addr;
  int    data[$];
  my_ext ext;
  string str;

  `uvm_object_utils_begin(my_trans)
    `uvm_field_enum      (cmd_t, cmd, UVM_ALL_ON)
    `uvm_field_int       (addr, UVM_ALL_ON)
    `uvm_field_queue_int (data, UVM_ALL_ON)
    `uvm_field_object    (ext,  UVM_ALL_ON)
    `uvm_field_string    (str,  UVM_ALL_ON)
  `uvm_object_utils_end

  function new(string name="mydata_inst");
    super.new(name);
  endfunction

endclass
```

Below is an example usage of the field macros for a component.

```
class my_comp extends uvm_component;

  my_comp_cfg  cfg;

  `uvm_component_utils_begin(my_comp)
    `uvm_field_object  (cfg,  UVM_ALL_ON)
  `uvm_object_utils_end

  function new(string name="my_comp_inst", uvm_component parent=null);
    super.new(name);
  endfunction

endclass
```

Each ``uvm_field_*` macro is named according to the particular data type it handles: integrals, strings, objects, queues, etc., and each has at least two arguments: *ARG* and *FLAG*.

ARG is the instance name of the variable, whose type must be compatible with the macro being invoked. In the example, class variable *addr* is an integral type, so we use the ``uvm_field_int` macro.

FLAG if set to *UVM_ALL_ON*, as in the example, the ARG variable will be included in all data methods. If FLAG is set to something other than *UVM_ALL_ON* or *UVM_DEFAULT*, it specifies which data method

implementations will *not* include the given variable. Thus, if *FLAG* is specified as *NO_COMPARE*, the *ARG* variable will not affect comparison operations, but it will be included in everything else.

All possible values for *FLAG* are listed and described below. Multiple flag values can be bitwise OR'ed together (in most cases they may be added together as well, but care must be taken when using the + operator to ensure that the same bit is not added more than once).

| | |
|----------------------|--|
| <i>UVM_ALL_ON</i> | Set all operations on. |
| <i>UVM_DEFAULT</i> | This is the recommended set of flags to pass to the field macros. Currently, it enables all of the operations, making it functionally identical to <i>UVM_ALL_ON</i> . In the future however, additional flags could be added with a recommended default value of <i>off</i> . |
| <i>UVM_NOCOPY</i> | Do not copy this field. |
| <i>UVM_NOCOMPARE</i> | Do not compare this field. |
| <i>UVM_NOPRINT</i> | Do not print this field. |
| <i>UVM_NOPACK</i> | Do not pack or unpack this field. |
| <i>UVM_REFERENCE</i> | For object types, operate only on the handle (e.g. no deep copy) |
| <i>UVM_PHYSICAL</i> | Treat as a physical field. Use physical setting in policy class for this field. |
| <i>UVM_ABSTRACT</i> | Treat as an abstract field. Use the abstract setting in the policy class for this field. |
| <i>UVM_READONLY</i> | Do not allow setting of this field from the <code>set_*_local</code> methods or during <code>uvm_component::apply_config_settings</code> operation. |

A radix for printing and recording can be specified by OR'ing one of the following constants in the *FLAG* argument

| | |
|---------------------|---|
| <i>UVM_BIN</i> | Print / record the field in binary (base-2). |
| <i>UVM_DEC</i> | Print / record the field in decimal (base-10). |
| <i>UVM_UNSIGNED</i> | Print / record the field in unsigned decimal (base-10). |
| <i>UVM_OCT</i> | Print / record the field in octal (base-8). |
| <i>UVM_HEX</i> | Print / record the field in hexadecimal (base-16). |
| <i>UVM_STRING</i> | Print / record the field in string format. |
| <i>UVM_TIME</i> | Print / record the field in time format. |

Radix settings for integral types. Hex is the default radix if none is specified.

A UVM component should *not* be specified using the ``uvm_field_object` macro unless its flag includes *UVM_REFERENCE*. Otherwise, the field macro will implement deep copy, which is an illegal operation for *uvm_components*. You will get a FATAL error if you tried to copy or clone an object containing a component handle that was registered with a field macro without the *UVM_REFERENCE* flag. You will also get duplicate entries when printing component topology, as this functionality is already provided by UVM.

`UVM_FIELD_* MACROS

Macros that implement data operations for scalar properties.

`uvm_field_int

Implements the data operations for any packed integral property.

```
`uvm_field_int(ARG, FLAG)
```

ARG is an integral property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_object

Implements the data operations for a [uvm_object](#)-based property.

```
`uvm_field_object(ARG, FLAG)
```

ARG is an object property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_string

Implements the data operations for a string property.

```
`uvm_field_string(ARG, FLAG)
```

ARG is a string property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_enum

Implements the data operations for an enumerated property.

```
`uvm_field_enum(T, ARG, FLAG)
```

T is an enumerated type, *ARG* is an instance of that type, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_real

Implements the data operations for any real property.

```
`uvm_field_real (ARG, FLAG)
```

ARG is an real property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_event

Implements the data operations for an event property.

```
`uvm_field_event (ARG, FLAG)
```

ARG is an event property of the class, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`UVM_FIELD_SARRAY_* MACROS

Macros that implement data operations for one-dimensional static array properties.

`uvm_field_sarray_int

Implements the data operations for a one-dimensional static array of integrals.

```
`uvm_field_sarray_int (ARG, FLAG)
```

ARG is a one-dimensional static array of integrals, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_sarray_object

Implements the data operations for a one-dimensional static array of [uvm_object](#)-based objects.

```
`uvm_field_sarray_object (ARG, FLAG)
```

ARG is a one-dimensional static array of [uvm_object](#)-based objects, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_sarray_string

Implements the data operations for a one-dimensional static array of strings.

```
`uvm_field_sarray_string(ARG, FLAG)
```

ARG is a one-dimensional static array of strings, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_sarray_enum

Implements the data operations for a one-dimensional static array of enums.

```
`uvm_field_sarray_enum(T, ARG, FLAG)
```

T is a one-dimensional dynamic array of enums *type*, *ARG* is an instance of that type, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`UVM_FIELD_ARRAY_* MACROS

Macros that implement data operations for one-dimensional dynamic array properties.

Implementation note

lines flagged with empty multi-line comments, */**/*, are not needed or need to be different for fixed arrays, which cannot be resized. Fixed arrays do not need to pack/unpack their size either, because their size is known; wouldn't hurt though if it allowed code consolidation. Unpacking would necessarily be different. **/*

`uvm_field_array_int

Implements the data operations for a one-dimensional dynamic array of integrals.

```
`uvm_field_array_int(ARG, FLAG)
```

ARG is a one-dimensional dynamic array of integrals, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_array_object

Implements the data operations for a one-dimensional dynamic array of [uvm_object](#)-based objects.

```
`uvm_field_array_object(ARG, FLAG)
```

ARG is a one-dimensional dynamic array of [uvm_object](#)-based objects, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

``uvm_field_array_string`

Implements the data operations for a one-dimensional dynamic array of strings.

```
`uvm_field_array_string(ARG, FLAG)
```

ARG is a one-dimensional dynamic array of strings, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

``uvm_field_array_enum`

Implements the data operations for a one-dimensional dynamic array of enums.

```
`uvm_field_array_enum(T, ARG, FLAG)
```

T is a one-dimensional dynamic array of enums *type*, *ARG* is an instance of that type, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

``UVM_FIELD_QUEUE_* MACROS`

Macros that implement data operations for dynamic queues.

``uvm_field_queue_int`

Implements the data operations for a queue of integrals.

```
`uvm_field_queue_int(ARG, FLAG)
```

ARG is a one-dimensional queue of integrals, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

``uvm_field_queue_object`

Implements the data operations for a queue of [uvm_object](#)-based objects.

```
`uvm_field_queue_object(ARG, FLAG)
```

ARG is a one-dimensional queue of [uvm_object](#)-based objects, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

``uvm_field_queue_string`

Implements the data operations for a queue of strings.

```
`uvm_field_queue_string(ARG, FLAG)
```

ARG is a one-dimensional queue of strings, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_queue_enum

Implements the data operations for a one-dimensional queue of enums.

```
`uvm_field_queue_enum(T, ARG, FLAG)
```

T is a queue of enums type, *ARG* is an instance of that type, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`UVM_FIELD_AA_*_STRING MACROS

Macros that implement data operations for associative arrays indexed by *string*.

`uvm_field_aa_int_string

Implements the data operations for an associative array of integrals indexed by *string*.

```
`uvm_field_aa_int_string(ARG, FLAG)
```

ARG is the name of a property that is an associative array of integrals with string key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_aa_object_string

Implements the data operations for an associative array of [uvm_object](#)-based objects indexed by *string*.

```
`uvm_field_aa_object_string(ARG, FLAG)
```

ARG is the name of a property that is an associative array of objects with string key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_aa_string_string

Implements the data operations for an associative array of strings indexed by *string*.

```
`uvm_field_aa_string_string(ARG,FLAG)
```

ARG is the name of a property that is an associative array of strings with string key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`UVM_FIELD_AA_*_INT MACROS

Macros that implement data operations for associative arrays indexed by an integral type.

`uvm_field_aa_object_int

Implements the data operations for an associative array of [uvm_object](#)-based objects indexed by the *int* data type.

```
`uvm_field_aa_object_int(ARG,FLAG)
```

ARG is the name of a property that is an associative array of objects with *int* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_aa_int_int

Implements the data operations for an associative array of integral types indexed by the *int* data type.

```
`uvm_field_aa_int_int(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *int* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_aa_int_int_unsigned

Implements the data operations for an associative array of integral types indexed by the *int unsigned* data type.

```
`uvm_field_aa_int_int_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *int unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

`uvm_field_aa_int_integer

Implements the data operations for an associative array of integral types indexed by the *integer* data type.

```
`uvm_field_aa_int_integer(ARG, FLAG)
```

ARG is the name of a property that is an associative array of integrals with *integer* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

``uvm_field_aa_int_integer_unsigned`

Implements the data operations for an associative array of integral types indexed by the *integer unsigned* data type.

```
`uvm_field_aa_int_integer_unsigned(ARG, FLAG)
```

ARG is the name of a property that is an associative array of integrals with *integer unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

``uvm_field_aa_int_byte`

Implements the data operations for an associative array of integral types indexed by the *byte* data type.

```
`uvm_field_aa_int_byte(ARG, FLAG)
```

ARG is the name of a property that is an associative array of integrals with *byte* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

``uvm_field_aa_int_byte_unsigned`

Implements the data operations for an associative array of integral types indexed by the *byte unsigned* data type.

```
`uvm_field_aa_int_byte_unsigned(ARG, FLAG)
```

ARG is the name of a property that is an associative array of integrals with *byte unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

``uvm_field_aa_int_shortint`

Implements the data operations for an associative array of integral types indexed by the

shortint data type.

```
`uvm_field_aa_int_shortint(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *shortint* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

``uvm_field_aa_int_shortint_unsigned`

Implements the data operations for an associative array of integral types indexed by the *shortint unsigned* data type.

```
`uvm_field_aa_int_shortint_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *shortint unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

``uvm_field_aa_int_longint`

Implements the data operations for an associative array of integral types indexed by the *longint* data type.

```
`uvm_field_aa_int_longint(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *longint* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

``uvm_field_aa_int_longint_unsigned`

Implements the data operations for an associative array of integral types indexed by the *longint unsigned* data type.

```
`uvm_field_aa_int_longint_unsigned(ARG,FLAG)
```

ARG is the name of a property that is an associative array of integrals with *longint unsigned* key, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

``uvm_field_aa_int_key`

Implements the data operations for an associative array of integral types indexed by any integral key data type.

```
`uvm_field_aa_int_key(KEY,ARG,FLAG)
```

KEY is the data type of the integral key, *ARG* is the name of a property that is an associative array of integrals, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

[`uvm_field_aa_int_enumkey](#)

Implements the data operations for an associative array of integral types indexed by any enumeration key data type.

```
`uvm_field_aa_int_enumkey(KEY, ARG, FLAG)
```

KEY is the enumeration type of the key, *ARG* is the name of a property that is an associative array of integrals, and *FLAG* is a bitwise OR of one or more flag settings as described in [Field Macros](#) above.

RECORDING MACROS

The recording macros assist users who implement the [uvm_object::do_record](#) method. They help ensure that the fields are recorded using a vendor- independent API. Unlike the [uvm_recorder](#) policy, fields recorded using the macros do not lose type information-- they are passed directly to the vendor-specific API. This results in more efficient recording and no artificial limit on bit-widths. See your simulator vendor's documentation for more information on its transaction recording capabilities.

[`uvm_record_attribute](#)

Vendor-independent macro to hide tool-specific interface for recording attributes (fields) to a transaction database.

```
`uvm_record_attribute(TR_HANDLE, NAME, VALUE)
```

The default implementation of the macro passes *NAME* and *VALUE* through to the [uvm_recorder::record_generic](#) method.

This macro should not be called directly by the user, the other recording macros will call it automatically if [uvm_recorder::use_record_attribute](#) returns true.

[`uvm_record_int](#)

```
`uvm_record_int(NAME,VALUE,SIZE[,RADIX])
```

The ``uvm_record_int` macro takes the same arguments as the

[uvm_recorder::record_field](#) method (including the optional *RADIX*).

The default implementation will pass the name/value pair to [`uvm_record_attribute](#) if enabled, otherwise the information will be passed to [uvm_recorder::record_field](#).

[`uvm_record_string](#)

```
`uvm_record_string(NAME,VALUE)
```

The [`uvm_record_string](#) macro takes the same arguments as the [uvm_recorder::record_string](#) method.

The default implementation will pass the name/value pair to [`uvm_record_attribute](#) if enabled, otherwise the information will be passed to [uvm_recorder::record_string](#).

[`uvm_record_time](#)

```
`uvm_record_time(NAME,VALUE)
```

The [`uvm_record_time](#) macro takes the same arguments as the [uvm_recorder::record_time](#) method.

The default implementation will pass the name/value pair to [`uvm_record_attribute](#) if enabled, otherwise the information will be passed to [uvm_recorder::record_time](#).

[`uvm_record_real](#)

```
`uvm_record_real(NAME,VALUE)
```

The [`uvm_record_real](#) macro takes the same arguments as the [uvm_recorder::record_field_real](#) method.

The default implementation will pass the name/value pair to [`uvm_record_attribute](#) if enabled, otherwise the information will be passed to [uvm_recorder::record_field_real](#).

[`uvm_record_field](#)

Macro for recording arbitrary name-value pairs into a transaction recording database. Requires a valid transaction handle, as provided by the [uvm_transaction::begin_tr](#) and [uvm_component::begin_tr](#) methods.

```
`uvm_record_field(NAME, VALUE)
```

The default implementation will pass the name/value pair to [`uvm_record_attribute](#) if

enabled, otherwise the information will be passed to `uvm_recorder::record_generic`, with the *VALUE* being converted to a string using “%p” notation.

```
recorder.record_generic(NAME,$sformatf("%p",VALUE));
```

PACKING MACROS

The packing macros assist users who implement the `uvm_object::do_pack` method. They help ensure that the pack operation is the exact inverse of the unpack operation. See also [Unpacking Macros](#).

```
virtual function void do_pack(uvm_packer packer);
    `uvm_pack_int(cmd)
    `uvm_pack_int(addr)
    `uvm_pack_array(data)
endfunction
```

The ‘N’ versions of these macros take a explicit size argument, which must be compile-time constant value greater than 0.

PACKING - WITH SIZE INFO

``uvm_pack_intN`

Pack an integral variable.

```
`uvm_pack_intN(VAR,SIZE)
```

``uvm_pack_enumN`

Pack an integral variable.

```
`uvm_pack_enumN(VAR,SIZE)
```

``uvm_pack_sarrayN`

Pack a static array of integrals.

```
`uvm_pack_sarray(VAR,SIZE)
```

`uvm_pack_arrayN

Pack a dynamic array of integrals.

```
`uvm_pack_arrayN (VAR, SIZE)
```

`uvm_pack_queueN

Pack a queue of integrals.

```
`uvm_pack_queueN (VAR, SIZE)
```

PACKING - No SIZE INFO

`uvm_pack_int

Pack an integral variable without having to also specify the bit size.

```
`uvm_pack_int (VAR)
```

`uvm_pack_enum

Pack an enumeration value. Packing does not require its type be specified.

```
`uvm_pack_enum (VAR)
```

`uvm_pack_string

Pack a string variable.

```
`uvm_pack_string (VAR)
```

`uvm_pack_real

Pack a variable of type real.

```
`uvm_pack_real (VAR)
```

``uvm_pack_sarray`

Pack a static array without having to also specify the bit size of its elements.

```
`uvm_pack_sarray (VAR)
```

``uvm_pack_array`

Pack a dynamic array without having to also specify the bit size of its elements. Array size must be non-zero.

```
`uvm_pack_array (VAR)
```

``uvm_pack_queue`

Pack a queue without having to also specify the bit size of its elements. Queue must not be empty.

```
`uvm_pack_queue (VAR)
```

UNPACKING MACROS

The unpacking macros assist users who implement the [uvm_object::do_unpack](#) method. They help ensure that the unpack operation is the exact inverse of the pack operation. See also [Packing Macros](#).

```
virtual function void do_unpack(uvm_packer packer);
  `uvm_unpack_enum(cmd, cmd_t)
  `uvm_unpack_int(addr)
  `uvm_unpack_array(data)
endfunction
```

The 'N' versions of these macros take an explicit size argument, which must be a compile-time constant value greater than 0.

UNPACKING - WITH SIZE INFO

``uvm_unpack_intN`

Unpack into an integral variable.

```
`uvm_unpack_intN (VAR, SIZE)
```

`uvm_unpack_enumN

Unpack enum of type *TYPE* into *VAR*.

```
`uvm_unpack_enumN (VAR, SIZE, TYPE)
```

`uvm_unpack_sarrayN

Unpack a static (fixed) array of integrals.

```
`uvm_unpack_sarrayN (VAR, SIZE)
```

`uvm_unpack_arrayN

Unpack into a dynamic array of integrals.

```
`uvm_unpack_arrayN (VAR, SIZE)
```

`uvm_unpack_queueN

Unpack into a queue of integrals.

```
`uvm_unpack_queue (VAR, SIZE)
```

UNPACKING - No SIZE INFO

`uvm_unpack_int

Unpack an integral variable without having to also specify the bit size.

```
`uvm_unpack_int (VAR)
```

``uvm_unpack_enum`

Unpack an enumeration value, which requires its type be specified.

```
`uvm_unpack_enum (VAR, TYPE)
```

``uvm_unpack_string`

Unpack a string variable.

```
`uvm_unpack_string (VAR)
```

``uvm_unpack_real`

Unpack a variable of type real.

```
`uvm_unpack_real (VAR)
```

``uvm_unpack_sarray`

Unpack a static array without having to also specify the bit size of its elements.

```
`uvm_unpack_sarray (VAR)
```

``uvm_unpack_array`

Unpack a dynamic array without having to also specify the bit size of its elements. Array size must be non-zero.

```
`uvm_unpack_array (VAR)
```

``uvm_unpack_queue`

Unpack a queue without having to also specify the bit size of its elements. Queue must not be empty.

```
`uvm_unpack_queue (VAR)
```

