The QEMU Object Model (QOM)

The QEMU Object Model provides a framework for registering user creatable types and instantiating objects from those types. QOM provides the following features:

- System for dynamically registering types
- Support for single-inheritance of types
- Multiple inheritance of stateless interfaces

Creating a minimal type

```
#include "qdev.h"

#define TYPE_MY_DEVICE "my-device"

// No new virtual functions: we can reuse the typedef for the
// superclass.
typedef DeviceClass MyDeviceClass;
typedef struct MyDevice
{
    DeviceState parent;

    int reg0, reg1, reg2;
} MyDevice;

static const TypeInfo my_device_info = {
    .name = TYPE_MY_DEVICE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(MyDevice),
};

static void my_device_register_types(void)
{
    type_register_static(&my_device_info);
}

type_init(my_device_register_types)
```

In the above example, we create a simple type that is described by #TypeInfo. #TypeInfo describes information about the type including what it inherits from, the instance and class size, and constructor/destructor hooks.

Alternatively several static types could be registered using helper macro DEFINE_TYPES()

```
static const TypeInfo device_types_info[] = {
    {
        .name = TYPE_MY_DEVICE_A,
        .parent = TYPE_DEVICE,
        .instance_size = sizeof(MyDeviceA),
    },
    {
        .name = TYPE_MY_DEVICE_B,
        .parent = TYPE_DEVICE,
        .instance_size = sizeof(MyDeviceB),
    },
};

DEFINE_TYPES(device_types_info)
```

Every type has an #ObjectClass associated with it. #ObjectClass derivatives are instantiated dynamically but there is only ever one instance for any given type. The #ObjectClass typically holds a table of function pointers for the virtual methods implemented by this type.

Using object_new(), a new #Object derivative will be instantiated. You can cast an #Object to a subclass (or base-class) type using object_dynamic_cast(). You typically want to define macro wrappers around OBJECT_CHECK() and OBJECT_CLASS_CHECK() to make it easier to convert to a specific type:

Typecasting macros

```
#define MY_DEVICE_GET_CLASS(obj) \
   OBJECT_GET_CLASS(MyDeviceClass, obj, TYPE_MY_DEVICE)
#define MY_DEVICE_CLASS(klass) \
   OBJECT_CLASS_CHECK(MyDeviceClass, klass, TYPE_MY_DEVICE)
#define MY_DEVICE(obj) \
   OBJECT_CHECK(MyDevice, obj, TYPE_MY_DEVICE)
```

In case the ObjectClass implementation can be built as module a module_obj() line must be added to make sure qemu loads the module when the object is needed.

```
module_obj(TYPE_MY_DEVICE);
```

Class Initialization

Before an object is initialized, the class for the object must be initialized. There is only one class object for all instance objects that is created lazily.

Classes are initialized by first initializing any parent classes (if necessary). After the parent class object has initialized, it will be copied into the current class object and any additional storage in the class object is zero filled.

The effect of this is that classes automatically inherit any virtual function pointers that the parent class has already initialized. All other fields will be zero filled.

Once all of the parent classes have been initialized, #TypeInfo::class_init is called to let the class being instantiated provide default initialize for its virtual functions. Here is how the above example might be modified to introduce an overridden virtual function:

Overriding a virtual function

```
#include "qdev.h"

void my_device_class_init(ObjectClass *klass, void *class_data)
{
    DeviceClass *dc = DEVICE_CLASS(klass);
    dc->reset = my_device_reset;
}

static const TypeInfo my_device_info = {
    .name = TYPE_MY_DEVICE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(MyDevice),
    .class_init = my_device_class_init,
};
```

Introducing new virtual methods requires a class to define its own struct and to add a .class_size member to the #TypeInfo. Each method will also have a wrapper function to call it easily:

Defining an abstract class

```
#include "qdev.h"

typedef struct MyDeviceClass
{
    DeviceClass parent;

    void (*frobnicate) (MyDevice *obj);
} MyDeviceClass;

static const TypeInfo my_device_info = {
    .name = TYPE_MY_DEVICE,
    .parent = TYPE_DEVICE,
    .instance_size = sizeof(MyDevice),
    .abstract = true, // or set a default in my_device_class_init
    .class_size = sizeof(MyDeviceClass),
};

void my_device_frobnicate(MyDevice *obj)
{
    MyDeviceClass *klass = MY_DEVICE_GET_CLASS(obj);

    klass->frobnicate(obj);
}
```

Interfaces

Interfaces allow a limited form of multiple inheritance. Instances are similar to normal types except for the fact that are only defined by their classes and never carry any state. As a consequence, a pointer to an interface instance should always be of incomplete type in order to be sure it cannot be dereferenced. That is, you should define the ‘typedef struct SomethingIf SomethingIf’ so that you can pass around `SomethingIf *si` arguments, but not define a `struct SomethingIf { ... }`. The only things you can validly do with a `SomethingIf *` are to pass it as an argument to a method on its corresponding SomethingIfClass, or to dynamically cast it to an object that implements the interface.

Methods

A *method* is a function within the namespace scope of a class. It usually operates on the object instance by passing it as a strongly-typed first argument. If it does not operate on an object instance, it is dubbed *class method*.

Methods cannot be overloaded. That is, the #ObjectClass and method name uniquely identity the function to be called; the signature does not vary except for trailing varargs.

Methods are always *virtual*. Overriding a method in #TypeInfo.class_init of a subclass leads to any user of the class obtained via OBJECT_GET_CLASS() accessing the overridden function. The original function is not automatically invoked. It is the responsibility of the overriding class to determine whether and when to invoke the method being overridden.

To invoke the method being overridden, the preferred solution is to store the original value in the overriding class before overriding the method. This corresponds to `{super,base}.method(...)` in Java and C# respectively; this frees the overriding class from hardcoding its parent class, which someone might choose to change at some point.

Overriding a virtual method

```c
typedef struct MyState MyState;

typedef void (*MyDoSomething)(MyState *obj);

typedef struct MyClass {
    ObjectClass parent_class;

    MyDoSomething do_something;
} MyClass;

static void my_do_something(MyState *obj)
{
    // do something
}

static void my_class_init(ObjectClass *oc, void *data)
{
    MyClass *mc = MY_CLASS(oc);

    mc->do_something = my_do_something;
}

static const TypeInfo my_type_info = {
    .name = TYPE_MY,
    .parent = TYPE_OBJECT,
    .instance_size = sizeof(MyState),
    .class_size = sizeof(MyClass),
    .class_init = my_class_init,
};

typedef struct DerivedClass {
    MyClass parent_class;

    MyDoSomething parent_do_something;
} DerivedClass;

static void derived_do_something(MyState *obj)
{
    DerivedClass *dc = DERIVED_GET_CLASS(obj);

    // do something here
    dc->parent_do_something(obj);
    // do something else here
}

static void derived_class_init(ObjectClass *oc, void *data)
{
    MyClass *mc = MY_CLASS(oc);
    DerivedClass *dc = DERIVED_CLASS(oc);

    dc->parent_do_something = mc->do_something;
    mc->do_something = derived_do_something;
}

static const TypeInfo derived_type_info = {
    .name = TYPE_DERIVED,
    .parent = TYPE_MY,
    .class_size = sizeof(DerivedClass),
    .class_init = derived_class_init,
};
```

Alternatively, object_class_by_name() can be used to obtain the class and its non-overridden methods for a specific type. This would correspond to `MyClass::method(...)` in C++.

The first example of such a QOM method was #CPUClass.reset, another example is #DeviceClass.realize.

Standard type declaration and definition macros

A lot of the code outlined above follows a standard pattern and naming convention. To reduce the amount of boilerplate code that needs to be written for a new type there are two sets of macros to generate the common parts in a standard format.

A type is declared using the OBJECT_DECLARE macro family. In types which do not require any virtual functions in the class, the OBJECT_DECLARE_SIMPLE_TYPE macro is suitable, and is commonly placed in the header file:

Declaring a simple type

```
OBJECT_DECLARE_SIMPLE_TYPE(MyDevice, MY_DEVICE)
```

This is equivalent to the following:

Expansion from declaring a simple type

```
typedef struct MyDevice MyDevice;
typedef struct MyDeviceClass MyDeviceClass;

G_DEFINE_AUTOPTR_CLEANUP_FUNC(MyDeviceClass, object_unref)

#define MY_DEVICE_GET_CLASS(void *obj) \
        OBJECT_GET_CLASS(MyDeviceClass, obj, TYPE_MY_DEVICE)
#define MY_DEVICE_CLASS(void *klass) \
        OBJECT_CLASS_CHECK(MyDeviceClass, klass, TYPE_MY_DEVICE)
#define MY_DEVICE(void *obj)
        OBJECT_CHECK(MyDevice, obj, TYPE_MY_DEVICE)

struct MyDeviceClass {
    DeviceClass parent_class;
};
```

The ‘struct MyDevice’ needs to be declared separately. If the type requires virtual functions to be declared in the class struct, then the alternative OBJECT_DECLARE_TYPE() macro can be used. This does the same as OBJECT_DECLARE_SIMPLE_TYPE(), but without the ‘struct MyDeviceClass’ definition.

To implement the type, the OBJECT_DEFINE macro family is available. In the simple case the OBJECT_DEFINE_TYPE macro is suitable:

Defining a simple type

```
OBJECT_DEFINE_TYPE(MyDevice, my_device, MY_DEVICE, DEVICE)
```

This is equivalent to the following:

Expansion from defining a simple type

```
static void my_device_finalize(Object *obj);
static void my_device_class_init(ObjectClass *oc, void *data);
static void my_device_init(Object *obj);

static const TypeInfo my_device_info = {
    .parent = TYPE_DEVICE,
    .name = TYPE_MY_DEVICE,
    .instance_size = sizeof(MyDevice),
    .instance_init = my_device_init,
    .instance_finalize = my_device_finalize,
    .class_size = sizeof(MyDeviceClass),
    .class_init = my_device_class_init,
};

static void
my_device_register_types(void)
{
    type_register_static(&my_device_info);
}
type_init(my_device_register_types);
```

This is sufficient to get the type registered with the type system, and the three standard methods now need to be implemented along with any other logic required for the type.

If the type needs to implement one or more interfaces, then the OBJECT_DEFINE_TYPE_WITH_INTERFACES() macro can be used instead. This accepts an array of interface type names.

Defining a simple type implementing interfaces

```
OBJECT_DEFINE_TYPE_WITH_INTERFACES(MyDevice, my_device,
                                   MY_DEVICE, DEVICE,
                                   { TYPE_USER_CREATABLE },
                                   { NULL })
```

If the type is not intended to be instantiated, then the OBJECT_DEFINE_ABSTRACT_TYPE() macro can be used instead:

Defining a simple abstract type

```
OBJECT_DEFINE_ABSTRACT_TYPE(MyDevice, my_device,
                            MY_DEVICE, DEVICE)
```

API Reference

`ObjectPropertyAccessor`

Typedef:

Syntax

`void ObjectPropertyAccessor (Object *obj, Visitor *v, const char *name, void *opaque, Error **errp)`

Parameters

`Object *obj`

the object that owns the property

`Visitor *v`

the visitor that contains the property data

`const char *name`

the name of the property

`void *opaque`

the object property opaque

`Error **errp`

a pointer to an Error that is filled if getting/setting fails.

Description

Called when trying to get/set a property.

`ObjectPropertyResolve`

Typedef:

Syntax

`Object * ObjectPropertyResolve (Object *obj, void *opaque, const char *part)`

Parameters

`Object *obj`

the object that owns the property

`void *opaque`

the opaque registered with the property

`const char *part`

the name of the property

Description

Resolves the `Object` corresponding to property **part**.

The returned object can also be used as a starting point to resolve a relative path starting with "**part**".

Return

If **path** is the path that led to **obj**, the function returns the `Object` corresponding to "**path/part**". If "**path/part**" is not a valid object path, it returns `NULL`.

`ObjectPropertyRelease`

Typedef:

Syntax

`void ObjectPropertyRelease (Object *obj, const char *name, void *opaque)`

Parameters

`Object *obj`

the object that owns the property

`const char *name`

the name of the property

`void *opaque`

the opaque registered with the property

Description

Called when a property is removed from a object.

`ObjectPropertyInit`

Typedef:

Syntax

`void ObjectPropertyInit (Object *obj, ObjectProperty *prop)`

Parameters

`Object *obj`

the object that owns the property

`ObjectProperty *prop`

the property to set

Description

Called when a property is initialized.

`ObjectUnparent`

Typedef:

Syntax

`void ObjectUnparent (Object *obj)`

Parameters

`Object *obj`

the object that is being removed from the composition tree

Description

Called when an object is being removed from the QOM composition tree. The function should remove any backlinks from children objects to **obj**.

`ObjectFree`

Typedef:

Syntax

`void ObjectFree (void *obj)`

Parameters

`void *obj`

the object being freed

Description

Called when an object's last reference is removed.

struct `ObjectClass`

Definition

```
struct ObjectClass {
};
```

Members

Description

The base for all classes. The only thing that `ObjectClass` contains is an integer type handle.

struct `Object`

Definition

```
struct Object {
};
```

Members

Description

The base for all objects. The first member of this object is a pointer to a `ObjectClass`. Since C guarantees that the first member of a structure always begins at byte 0 of that structure, as long as any sub-object places its parent as the first member, we can cast directly to a `Object`.

As a result, `Object` contains a reference to the objects type as its first member. This allows identification of the real type of the object at run time.

DECLARE_INSTANCE_CHECKER(InstanceType, OBJ_NAME, TYPENAME)

Parameters

`InstanceType`
: instance struct name

`OBJ_NAME`
: the object name in uppercase with underscore separators

`TYPENAME`
: type name

Description

Direct usage of this macro should be avoided, and the complete OBJECT_DECLARE_TYPE macro is recommended instead.

This macro will provide the instance type cast functions for a QOM type.

> **DECLARE_CLASS_CHECKERS(ClassType, OBJ_NAME, TYPENAME)**

Parameters

`ClassType`
: class struct name

`OBJ_NAME`
: the object name in uppercase with underscore separators

`TYPENAME`
: type name

Description

Direct usage of this macro should be avoided, and the complete OBJECT_DECLARE_TYPE macro is recommended instead.

This macro will provide the class type cast functions for a QOM type.

> **DECLARE_OBJ_CHECKERS(InstanceType, ClassType, OBJ_NAME, TYPENAME)**

Parameters

`InstanceType`
: instance struct name

`ClassType`
: class struct name

`OBJ_NAME`
: the object name in uppercase with underscore separators

`TYPENAME`
: type name

Description

Direct usage of this macro should be avoided, and the complete OBJECT_DECLARE_TYPE macro is recommended instead.

This macro will provide the three standard type cast functions for a QOM type.

OBJECT_DECLARE_TYPE(InstanceType, ClassType, MODULE_OBJ_NAME)

Parameters

`InstanceType`
: instance struct name

`ClassType`
: class struct name

`MODULE_OBJ_NAME`
: the object name in uppercase with underscore separators

Description

This macro is typically used in a header file, and will:

- create the typedefs for the object and class structs
- register the type for use with g_autoptr
- provide three standard type cast functions

The object struct and class struct need to be declared manually.

OBJECT_DECLARE_SIMPLE_TYPE(InstanceType, MODULE_OBJ_NAME)

Parameters

`InstanceType`
: instance struct name

`MODULE_OBJ_NAME`
: the object name in uppercase with underscore separators

Description

This does the same as OBJECT_DECLARE_TYPE(), but with no class struct declared.

This macro should be used unless the class struct needs to have virtual methods declared.

OBJECT_DEFINE_TYPE_EXTENDED(ModuleObjName, module_obj_name, MODULE_OBJ_NAME, PARENT_MODULE_OBJ_NAME, ABSTRACT, ...)

Parameters

`ModuleObjName`

the object name with initial caps

`module_obj_name`

the object name in lowercase with underscore separators

`MODULE_OBJ_NAME`

the object name in uppercase with underscore separators

`PARENT_MODULE_OBJ_NAME`

the parent object name in uppercase with underscore separators

`ABSTRACT`

boolean flag to indicate whether the object can be instantiated

`...`

list of initializers for “InterfaceInfo” to declare implemented interfaces

Description

This macro is typically used in a source file, and will:

- declare prototypes for _finalize, _class_init and _init methods
- declare the TypeInfo struct instance
- provide the constructor to register the type

After using this macro, implementations of the _finalize, _class_init, and _init methods need to be written. Any of these can be zero-line no-op impls if no special logic is required for a given type.

This macro should rarely be used, instead one of the more specialized macros is usually a better choice.

> **OBJECT_DEFINE_TYPE(ModuleObjName, module_obj_name, MODULE_OBJ_NAME, PARENT_MODULE_OBJ_NAME)**

Parameters

`ModuleObjName`

the object name with initial caps

`module_obj_name`

the object name in lowercase with underscore separators

`MODULE_OBJ_NAME`

the object name in uppercase with underscore separators

`PARENT_MODULE_OBJ_NAME`

the parent object name in uppercase with underscore separators

Description

This is a specialization of OBJECT_DEFINE_TYPE_EXTENDED, which is suitable for the common case of a non-abstract type, without any interfaces.

> **OBJECT_DEFINE_TYPE_WITH_INTERFACES(ModuleObjName, module_obj_name, MODULE_OBJ_NAME, PARENT_MODULE_OBJ_NAME, ...)**

Parameters

`ModuleObjName`
: the object name with initial caps

`module_obj_name`
: the object name in lowercase with underscore separators

`MODULE_OBJ_NAME`
: the object name in uppercase with underscore separators

`PARENT_MODULE_OBJ_NAME`
: the parent object name in uppercase with underscore separators

`...`
: list of initializers for "InterfaceInfo" to declare implemented interfaces

Description

This is a specialization of OBJECT_DEFINE_TYPE_EXTENDED, which is suitable for the common case of a non-abstract type, with one or more implemented interfaces.

Note when passing the list of interfaces, be sure to include the final NULL entry, e.g. { TYPE_USER_CREATABLE }, { NULL }

> **OBJECT_DEFINE_ABSTRACT_TYPE(ModuleObjName, module_obj_name, MODULE_OBJ_NAME, PARENT_MODULE_OBJ_NAME)**

Parameters

`ModuleObjName`
: the object name with initial caps

`module_obj_name`
: the object name in lowercase with underscore separators

`MODULE_OBJ_NAME`
: the object name in uppercase with underscore separators

`PARENT_MODULE_OBJ_NAME`

the parent object name in uppercase with underscore separators

Description

This is a specialization of OBJECT_DEFINE_TYPE_EXTENDED, which is suitable for defining an abstract type, without any interfaces.

struct `TypeInfo`

Definition

```
struct TypeInfo {
  const char *name;
  const char *parent;
  size_t instance_size;
  size_t instance_align;
  void (*instance_init)(Object *obj);
  void (*instance_post_init)(Object *obj);
  void (*instance_finalize)(Object *obj);
  bool abstract;
  size_t class_size;
  void (*class_init)(ObjectClass *klass, void *data);
  void (*class_base_init)(ObjectClass *klass, void *data);
  void *class_data;
  InterfaceInfo *interfaces;
};
```

Members

`name`

The name of the type.

`parent`

The name of the parent type.

`instance_size`

The size of the object (derivative of `Object`). If **instance_size** is 0, then the size of the object will be the size of the parent object.

`instance_align`

The required alignment of the object. If **instance_align** is 0, then normal malloc alignment is sufficient; if non-zero, then we must use qemu_memalign for allocation.

`instance_init`

This function is called to initialize an object. The parent class will have already been initialized so the type is only responsible for initializing its own members.

`instance_post_init`

This function is called to finish initialization of an object, after all **instance_init** functions were called.

`instance_finalize`

This function is called during object destruction. This is called before the parent **instance_finalize** function has been called. An object should only free the members that are unique to its type in this function.

`abstract`

If this field is true, then the class is considered abstract and cannot be directly instantiated.

`class_size`

The size of the class object (derivative of `ObjectClass`) for this object. If **class_size** is 0, then the size of the class will be assumed to be the size of the parent class. This allows a type to avoid implementing an explicit class type if they are not adding additional virtual functions.

`class_init`

This function is called after all parent class initialization has occurred to allow a class to set its default virtual method pointers. This is also the function to use to override virtual methods from a parent class.

`class_base_init`

This function is called for all base classes after all parent class initialization has occurred, but before the class itself is initialized. This is the function to use to undo the effects of memcpy from the parent class to the descendants.

`class_data`

Data to pass to the **class_init**, **class_base_init**. This can be useful when building dynamic classes.

`interfaces`

The list of interfaces associated with this type. This should point to a static array that's terminated with a zero filled element.

OBJECT(obj)

Parameters

`obj`

A derivative of `Object`

Description

Converts an object to a `Object`. Since all objects are `Objects`, this function will always succeed.

OBJECT_CLASS(class)

Parameters

`class`

A derivative of `ObjectClass`.

Description

Converts a class to an `ObjectClass`. Since all objects are `Objects`, this function will always succeed.

OBJECT_CHECK(type, obj, name)

Parameters

`type`

The C type to use for the return value.

`obj`

A derivative of **type** to cast.

`name`

The QOM typename of **type**

Description

A type safe version of **object_dynamic_cast_assert**. Typically each class will define a macro based on this type to perform type safe dynamic_casts to this object type.

If an invalid object is passed to this function, a run time assert will be generated.

OBJECT_CLASS_CHECK(class_type, class, name)

Parameters

`class_type`

The C type to use for the return value.

`class`

A derivative class of **class_type** to cast.

`name`

the QOM typename of **class_type**.

Description

A type safe version of **object_class_dynamic_cast_assert**. This macro is typically wrapped by each type to perform type safe casts of a class to a specific class type.

OBJECT_GET_CLASS(class, obj, name)

Parameters

`class`

The C type to use for the return value.

`obj`

The object to obtain the class for.

`name`

The QOM typename of **obj**.

Description

This function will return a specific class for a given object. Its generally used by each type to provide a type safe macro to get a specific class type from an object.

struct InterfaceInfo

Definition

```
struct InterfaceInfo {
  const char *type;
};
```

Members

`type`

The name of the interface.

Description

The information associated with an interface.

struct InterfaceClass

Definition

```
struct InterfaceClass {
  ObjectClass parent_class;
};
```

Members

`parent_class`

    the base class

Description

The class for all interfaces. Subclasses of this class should only add virtual methods.

> **INTERFACE_CLASS(klass)**

Parameters

`klass`

    class to cast from

Return

An `InterfaceClass` or raise an error if cast is invalid

> **INTERFACE_CHECK(interface, obj, name)**

Parameters

`interface`

    the type to return

`obj`

    the object to convert to an interface

`name`

    the interface type name

Return

obj casted to **interface** if cast is valid, otherwise raise error.

> Object * **object_new_with_class**(ObjectClass ***klass**)

Parameters

`ObjectClass *klass`

    The class to instantiate.

Description

This function will initialize a new object using heap allocated memory. The returned object has a reference count of 1, and will be freed when the last reference is dropped.

Return

The newly allocated and instantiated object.

> Object * **`object_new`(const char **typename*)**

Parameters

`const char *typename`

The name of the type of the object to instantiate.

Description

This function will initialize a new object using heap allocated memory. The returned object has a reference count of 1, and will be freed when the last reference is dropped.

Return

The newly allocated and instantiated object.

> Object * **`object_new_with_props`(const char **typename*, Object **parent*, const char **id*, Error ***errp*, ...)**

Parameters

`const char *typename`

The name of the type of the object to instantiate.

`Object *parent`

the parent object

`const char *id`

The unique ID of the object

`Error **errp`

pointer to error object

`...`

list of property names and values

Description

This function will initialize a new object using heap allocated memory. The returned object has a reference count of 1, and will be freed when the last reference is dropped.

The **id** parameter will be used when registering the object as a child of **parent** in the composition tree.

The variadic parameters are a list of pairs of (propname, propvalue) strings. The propname of `NULL` indicates the end of the property list. If the object implements the user creatable interface, the object will be marked complete once all the properties have been processed.

Creating an object with properties

```
Error *err = NULL;
Object *obj;

obj = object_new_with_props(TYPE_MEMORY_BACKEND_FILE,
                            object_get_objects_root(),
                            "hostmem0",
                            &err,
                            "share", "yes",
                            "mem-path", "/dev/shm/somefile",
                            "prealloc", "yes",
                            "size", "1048576",
                            NULL);

if (!obj) {
  error_reportf_err(err, "Cannot create memory backend: ");
}
```

The returned object will have one stable reference maintained for as long as it is present in the object hierarchy.

Return

The newly allocated, instantiated & initialized object.

Object * `object_new_with_propv`(const char *_typename_, Object *_parent_, const char *_id_, Error **_errp_, va_list _vargs_)

Parameters

`const char *typename`

The name of the type of the object to instantiate.

`Object *parent`

the parent object

`const char *id`

The unique ID of the object

`Error **errp`

pointer to error object

`va_list vargs`

list of property names and values

Description

See object_new_with_props() for documentation.

bool object_set_props(Object **obj*, Error ***errp*, ...)

Parameters

`Object *obj`

  the object instance to set properties on

`Error **errp`

  pointer to error object

`...`

  list of property names and values

Description

This function will set a list of properties on an existing object instance.

The variadic parameters are a list of pairs of (propname, propvalue) strings. The propname of `NULL` indicates the end of the property list.

Update an object's properties

```
Error *err = NULL;
Object *obj = ...get / create object...;

if (!object_set_props(obj,
                      &err,
                      "share", "yes",
                      "mem-path", "/dev/shm/somefile",
                      "prealloc", "yes",
                      "size", "1048576",
                      NULL)) {
  error_reportf_err(err, "Cannot set properties: ");
}
```

The returned object will have one stable reference maintained for as long as it is present in the object hierarchy.

Return

`true` on success, `false` on error.

bool object_set_propv(Object **obj*, Error ***errp*, va_list *vargs*)

Parameters

`Object *obj`

the object instance to set properties on

`Error **errp`

pointer to error object

`va_list vargs`

list of property names and values

Description

See object_set_props() for documentation.

Return

`true` on success, `false` on error.

> **void object_initialize(void *obj, size_t size, const char *typename)**

Parameters

`void *obj`

A pointer to the memory to be used for the object.

`size_t size`

The maximum size available at **obj** for the object.

`const char *typename`

The name of the type of the object to instantiate.

Description

This function will initialize an object. The memory for the object should have already been allocated. The returned object has a reference count of 1, and will be finalized when the last reference is dropped.

> **bool object_initialize_child_with_props(Object *parentobj, const char *propname, void *childobj, size_t size, const char *type, Error **errp, ...)**

Parameters

`Object *parentobj`

The parent object to add a property to

`const char *propname`

The name of the property

`void *childobj`

A pointer to the memory to be used for the object.

`size_t size`

The maximum size available at **childobj** for the object.

`const char *type`

The name of the type of the object to instantiate.

`Error **errp`

If an error occurs, a pointer to an area to store the error

`...`

list of property names and values

Description

This function will initialize an object. The memory for the object should have already been allocated. The object will then be added as child property to a parent with object_property_add_child() function. The returned object has a reference count of 1 (for the "child<...>" property from the parent), so the object will be finalized automatically when the parent gets removed.

The variadic parameters are a list of pairs of (propname, propvalue) strings. The propname of `NULL` indicates the end of the property list. If the object implements the user creatable interface, the object will be marked complete once all the properties have been processed.

Return

`true` on success, `false` on failure.

> **bool object_initialize_child_with_propsv(Object *_parentobj_, const char *_propname_, void *_childobj_, size_t _size_, const char *_type_, Error **_errp_, va_list _vargs_)**

Parameters

`Object *parentobj`

The parent object to add a property to

`const char *propname`

The name of the property

`void *childobj`

A pointer to the memory to be used for the object.

`size_t size`

The maximum size available at **childobj** for the object.

`const char *type`

The name of the type of the object to instantiate.

`Error **errp`

If an error occurs, a pointer to an area to store the error

`va_list vargs`

list of property names and values

Description

See object_initialize_child() for documentation.

Return

`true` on success, `false` on failure.

`object_initialize_child`(parent, propname, child, type)

Parameters

`parent`

The parent object to add a property to

`propname`

The name of the property

`child`

A precisely typed pointer to the memory to be used for the object.

`type`

The name of the type of the object to instantiate.

Description

This is like:

```
object_initialize_child_with_props(parent, propname,
                                   child, sizeof(*child), type,
                                   &error_abort, NULL)
```

Object * `object_dynamic_cast`(Object **obj*, const char **typename*)

Parameters

`Object *obj`

The object to cast.

`const char *typename`

The **typename** to cast to.

Description

This function will determine if **obj** is-a **typename**. **obj** can refer to an object or an interface associated with an object.

Return

This function returns **obj** on success or `NULL` on failure.

> Object * **object_dynamic_cast_assert(Object** ***obj*, const char** ***typename*, const char** ***file*, int** ***line*, const char** ***func*)**

Parameters

`Object *obj`

  The object to cast.

`const char *typename`

  The **typename** to cast to.

`const char *file`

  Source code file where function was called

`int line`

  Source code line where function was called

`const char *func`

  Name of function where this function was called

Description

See object_dynamic_cast() for a description of the parameters of this function. The only difference in behavior is that this function asserts instead of returning `NULL` on failure if QOM cast debugging is enabled. This function is not meant to be called directly, but only through the wrapper macro OBJECT_CHECK.

> ObjectClass * **object_get_class(Object** ***obj*)**

Parameters

`Object *obj`

  A derivative of `Object`

Return

The `ObjectClass` of the type associated with **obj**.

const char * `object_get_typename`(const Object *obj)

Parameters

`const Object *obj`

A derivative of `Object`.

Return

The QOM typename of **obj**.

Type `type_register_static`(const TypeInfo *info)

Parameters

`const TypeInfo *info`

The `TypeInfo` of the new type.

Description

info and all of the strings it points to should exist for the life time that the type is registered.

Return

the new `Type`.

Type `type_register`(const TypeInfo *info)

Parameters

`const TypeInfo *info`

The `TypeInfo` of the new type

Description

Unlike type_register_static(), this call does not require **info** or its string members to continue to exist after the call returns.

Return

the new `Type`.

void `type_register_static_array`(const TypeInfo *infos, int nr_infos)

Parameters

`const TypeInfo *infos`

: The array of the new type `TypeInfo` structures.

`int nr_infos`

: number of entries in **infos**

Description

infos and all of the strings it points to should exist for the life time that the type is registered.

DEFINE_TYPES(type_array)

Parameters

`type_array`

: The array containing `TypeInfo` structures to register

Description

type_array should be static constant that exists for the life time that the type is registered.

bool type_print_class_properties(const char *_type_)

Parameters

`const char *type`

: a QOM class name

Description

Print the object's class properties to stdout or the monitor. Return whether an object was found.

void object_set_properties_from_keyval(Object *_obj_, const QDict *_qdict_, bool _from_json_, Error **_errp_)

Parameters

`Object *obj`

: a QOM object

`const QDict *qdict`

: a dictionary with the properties to be set

`bool from_json`

true if leaf values of **qdict** are typed, false if they are strings

`Error **errp`

pointer to error object

Description

For each key in the dictionary, parse the value string if needed, then set the corresponding property in **obj**.

> **ObjectClass * `object_class_dynamic_cast_assert`(ObjectClass *klass, const char *typename, const char *file, int line, const char *func)**

Parameters

`ObjectClass *klass`

The `ObjectClass` to attempt to cast.

`const char *typename`

The QOM typename of the class to cast to.

`const char *file`

Source code file where function was called

`int line`

Source code line where function was called

`const char *func`

Name of function where this function was called

Description

See object_class_dynamic_cast() for a description of the parameters of this function. The only difference in behavior is that this function asserts instead of returning `NULL` on failure if QOM cast debugging is enabled. This function is not meant to be called directly, but only through the wrapper macro OBJECT_CLASS_CHECK.

> **ObjectClass * `object_class_dynamic_cast`(ObjectClass *klass, const char *typename)**

Parameters

`ObjectClass *klass`

The `ObjectClass` to attempt to cast.

`const char *typename`

The QOM typename of the class to cast to.

Return

If **typename** is a class, this function returns **klass** if **typename** is a subtype of **klass**, else returns `NULL`.

Description

If **typename** is an interface, this function returns the interface definition for **klass** if **klass** implements it unambiguously; `NULL` is returned if **klass** does not implement the interface or if multiple classes or interfaces on the hierarchy leading to **klass** implement it. (FIXME: perhaps this can be detected at type definition time?)

> ObjectClass * **`object_class_get_parent`**(ObjectClass ***klass*)**

Parameters

`ObjectClass *klass`

The class to obtain the parent for.

Return

The parent for **klass** or `NULL` if none.

> **const char * `object_class_get_name`**(ObjectClass ***klass*)**

Parameters

`ObjectClass *klass`

The class to obtain the QOM typename for.

Return

The QOM typename for **klass**.

> **bool `object_class_is_abstract`**(ObjectClass ***klass*)**

Parameters

`ObjectClass *klass`

The class to obtain the abstractness for.

Return

`true` if **klass** is abstract, `false` otherwise.

> ObjectClass * **`object_class_by_name`(const char ****typename*)**

Parameters

`const char *typename`

The QOM typename to obtain the class for.

Return

The class for **typename** or `NULL` if not found.

> ObjectClass * **`module_object_class_by_name`(const char *_typename_)**

Parameters

`const char *typename`

The QOM typename to obtain the class for.

Description

For objects which might be provided by a module. Behaves like object_class_by_name, but additionally tries to load the module needed in case the class is not available.

Return

The class for **typename** or `NULL` if not found.

> **GSList * `object_class_get_list`(const char *_implements_type_, bool _include_abstract_)**

Parameters

`const char *implements_type`

The type to filter for, including its derivatives.

`bool include_abstract`

Whether to include abstract classes.

Return

A singly-linked list of the classes in reverse hashtable order.

> **GSList * `object_class_get_list_sorted`(const char *_implements_type_,
> bool _include_abstract_)**

Parameters

`const char *implements_type`

The type to filter for, including its derivatives.

`bool include_abstract`

Whether to include abstract classes.

Return

A singly-linked list of the classes in alphabetical case-insensitive order.

> **Object * `object_ref`(void **obj*)**

Parameters

`void *obj`

the object

Description

Increase the reference count of a object. A object cannot be freed as long as its reference count is greater than zero.

Return

obj

> **void `object_unref`(void **obj*)**

Parameters

`void *obj`

the object

Description

Decrease the reference count of a object. A object cannot be freed as long as its reference count is greater than zero.

> **ObjectProperty * `object_property_try_add`(Object **obj*, const char **name*, const char **type*, ObjectPropertyAccessor **get*, ObjectPropertyAccessor **set*, ObjectPropertyRelease **release*, void **opaque*, Error ***errp*)**

Parameters

`Object *obj`

the object to add a property to

`const char *name`

the name of the property. This can contain any character except for a forward slash. In general, you should use hyphens '-' instead of underscores '_' when naming properties.

`const char *type`

the type name of the property. This namespace is pretty loosely defined. Sub namespaces are constructed by using a prefix and then to angle brackets. For instance, the type 'virtio-net-pci' in the 'link' namespace would be 'link<virtio-net-pci>'.

`ObjectPropertyAccessor *get`

The getter to be called to read a property. If this is NULL, then the property cannot be read.

`ObjectPropertyAccessor *set`

the setter to be called to write a property. If this is NULL, then the property cannot be written.

`ObjectPropertyRelease *release`

called when the property is removed from the object. This is meant to allow a property to free its opaque upon object destruction. This may be NULL.

`void *opaque`

an opaque pointer to pass to the callbacks for the property

`Error **errp`

pointer to error object

Return

The `ObjectProperty`; this can be used to set the **resolve** callback for child and link properties.

ObjectProperty * `object_property_add`(Object **obj*, const char **name*, const char **type*, ObjectPropertyAccessor **get*, ObjectPropertyAccessor **set*, ObjectPropertyRelease **release*, void **opaque*)

Same as object_property_try_add() with **errp** hardcoded to &error_abort.

Parameters

`Object *obj`

the object to add a property to

`const char *name`

the name of the property. This can contain any character except for a forward slash. In general, you should use hyphens '-' instead of underscores '_' when naming properties.

`const char *type`

the type name of the property. This namespace is pretty loosely defined. Sub namespaces are constructed by using a prefix and then to angle brackets. For instance, the type 'virtio-net-pci' in the 'link' namespace would be 'link<virtio-net-pci>'.

`ObjectPropertyAccessor *get`

The getter to be called to read a property. If this is NULL, then the property cannot be read.

`ObjectPropertyAccessor *set`

the setter to be called to write a property. If this is NULL, then the property cannot be written.

`ObjectPropertyRelease *release`

called when the property is removed from the object. This is meant to allow a property to free its opaque upon object destruction. This may be NULL.

`void *opaque`

an opaque pointer to pass to the callbacks for the property

> **void `object_property_set_default_bool`(ObjectProperty **prop*, bool *value*)**

Parameters

`ObjectProperty *prop`

the property to set

`bool value`

the value to be written to the property

Description

Set the property default value.

> **void `object_property_set_default_str`(ObjectProperty **prop*, const char **value*)**

Parameters

`ObjectProperty *prop`

the property to set

`const char *value`

the value to be written to the property

Description

Set the property default value.

> **void `object_property_set_default_int`(ObjectProperty **prop*, int64_t *value*)**

Parameters

`ObjectProperty *prop`

the property to set

`int64_t value`

the value to be written to the property

Description

Set the property default value.

void `object_property_set_default_uint`(ObjectProperty **prop*, uint64_t *value*)

Parameters

`ObjectProperty *prop`

the property to set

`uint64_t value`

the value to be written to the property

Description

Set the property default value.

ObjectProperty * `object_property_find`(Object **obj*, const char **name*)

Parameters

`Object *obj`

the object

`const char *name`

the name of the property

Description

Look up a property for an object.

Return its `ObjectProperty` if found, or NULL.

ObjectProperty * `object_property_find_err`(Object **obj*, const char **name*, Error ***errp*)

Parameters

`Object *obj`

the object

`const char *name`

the name of the property

`Error **errp`

returns an error if this function fails

Description

Look up a property for an object.

Return its `ObjectProperty` if found, or NULL.

> **ObjectProperty * `object_class_property_find`(ObjectClass *klass, const char *name)**

Parameters

`ObjectClass *klass`

the object class

`const char *name`

the name of the property

Description

Look up a property for an object class.

Return its `ObjectProperty` if found, or NULL.

> **ObjectProperty * `object_class_property_find_err`(ObjectClass *klass, const char *name, Error **errp)**

Parameters

`ObjectClass *klass`

the object class

`const char *name`

the name of the property

`Error **errp`

returns an error if this function fails

Description

Look up a property for an object class.

Return its `ObjectProperty` if found, or NULL.

> **void `object_property_iter_init`(ObjectPropertyIterator **iter*, Object **obj*)**

Parameters

`ObjectPropertyIterator *iter`

the iterator instance

`Object *obj`

the object

Description

Initializes an iterator for traversing all properties registered against an object instance, its class and all parent classes.

It is forbidden to modify the property list while iterating, whether removing or adding properties.

Typical usage pattern would be

Using object property iterators

```
ObjectProperty *prop;
ObjectPropertyIterator iter;

object_property_iter_init(&iter, obj);
while ((prop = object_property_iter_next(&iter))) {
  ... do something with prop ...
}
```

> **void `object_class_property_iter_init`(ObjectPropertyIterator **iter*, ObjectClass **klass*)**

Parameters

`ObjectPropertyIterator *iter`

the iterator instance

`ObjectClass *klass`

the class

Description

Initializes an iterator for traversing all properties registered against an object class and all parent classes.

It is forbidden to modify the property list while iterating, whether removing or adding properties.

This can be used on abstract classes as it does not create a temporary instance.

ObjectProperty * `object_property_iter_next`(ObjectPropertyIterator **iter*)

Parameters

`ObjectPropertyIterator *iter`

the iterator instance

Description

Return the next available property. If no further properties are available, a `NULL` value will be returned and the **iter** pointer should not be used again after this point without re-initializing it.

Return

the next property, or `NULL` when all properties have been traversed.

bool `object_property_get`(Object **obj*, const char **name*, Visitor **v*, Error ***errp*)

Parameters

`Object *obj`

the object

`const char *name`

the name of the property

`Visitor *v`

the visitor that will receive the property value. This should be an Output visitor and the data will be written with **name** as the name.

`Error **errp`

returns an error if this function fails

Description

Reads a property from a object.

Return

`true` on success, `false` on failure.

bool `object_property_set_str`(Object **obj*, const char **name*, const char **value*, Error ***errp*)

Parameters

`Object *obj`

: the object

`const char *name`

: the name of the property

`const char *value`

: the value to be written to the property

`Error **errp`

: returns an error if this function fails

Description

Writes a string value to a property.

Return

`true` on success, `false` on failure.

char * `object_property_get_str`(Object **obj*, const char **name*, Error ***errp*)

Parameters

`Object *obj`

: the object

`const char *name`

: the name of the property

`Error **errp`

: returns an error if this function fails

Return

the value of the property, converted to a C string, or NULL if an error occurs (including when the property value is not a string). The caller should free the string.

bool `object_property_set_link`(Object **obj*, const char **name*, Object **value*, Error ***errp*)

Parameters

`Object *obj`

the object

`const char *name`

the name of the property

`Object *value`

the value to be written to the property

`Error **errp`

returns an error if this function fails

Description

Writes an object’s canonical path to a property.

If the link property was created with `OBJ_PROP_LINK_STRONG` bit, the old target object is unreferenced, and a reference is added to the new target object.

Return

`true` on success, `false` on failure.

> Object * **`object_property_get_link`**(Object **obj*, const char **name*, Error ***errp*)

Parameters

`Object *obj`

the object

`const char *name`

the name of the property

`Error **errp`

returns an error if this function fails

Return

the value of the property, resolved from a path to an Object, or NULL if an error occurs (including when the property value is not a string or not a valid object path).

> bool **`object_property_set_bool`**(Object **obj*, const char **name*, bool *value*, Error ***errp*)

Parameters

`Object *obj`

the object

`const char *name`

the name of the property

`bool value`

the value to be written to the property

`Error **errp`

returns an error if this function fails

Description

Writes a bool value to a property.

Return

`true` on success, `false` on failure.

> **bool `object_property_get_bool`(Object **obj*, const char **name*, Error ***errp*)**

Parameters

`Object *obj`

the object

`const char *name`

the name of the property

`Error **errp`

returns an error if this function fails

Return

the value of the property, converted to a boolean, or false if an error occurs (including when the property value is not a bool).

> **bool `object_property_set_int`(Object **obj*, const char **name*, int64_t *value*, Error ***errp*)**

Parameters

`Object *obj`

the object

`const char *name`

the name of the property

`int64_t value`

the value to be written to the property

`Error **errp`

returns an error if this function fails

Description

Writes an integer value to a property.

Return

`true` on success, `false` on failure.

> **int64_t `object_property_get_int`(Object **obj*, const char **name*, Error ***errp*)**

Parameters

`Object *obj`

the object

`const char *name`

the name of the property

`Error **errp`

returns an error if this function fails

Return

the value of the property, converted to an integer, or -1 if an error occurs (including when the property value is not an integer).

> **bool `object_property_set_uint`(Object **obj*, const char **name*, uint64_t *value*, Error ***errp*)**

Parameters

`Object *obj`

the object

`const char *name`

the name of the property

`uint64_t value`

the value to be written to the property

`Error **errp`

returns an error if this function fails

Description

Writes an unsigned integer value to a property.

Return

`true` on success, `false` on failure.

> **uint64_t `object_property_get_uint`(Object **obj*, const char **name*, Error ***errp*)**

Parameters

`Object *obj`
: the object

`const char *name`
: the name of the property

`Error **errp`
: returns an error if this function fails

Return

the value of the property, converted to an unsigned integer, or 0 an error occurs (including when the property value is not an integer).

> **int `object_property_get_enum`(Object **obj*, const char **name*, const char **typename*, Error ***errp*)**

Parameters

`Object *obj`
: the object

`const char *name`
: the name of the property

`const char *typename`
: the name of the enum data type

`Error **errp`
: returns an error if this function fails

Return

the value of the property, converted to an integer (which can't be negative), or -1 on error (including when the property value is not an enum).

bool `object_property_set`(Object **obj*, const char **name*, Visitor **v*, Error ***errp*)

Parameters

`Object *obj`

  the object

`const char *name`

  the name of the property

`Visitor *v`

  the visitor that will be used to write the property value. This should be an Input visitor and the data will be first read with **name** as the name and then written as the property value.

`Error **errp`

  returns an error if this function fails

Description

Writes a property to a object.

Return

`true` on success, `false` on failure.

bool `object_property_parse`(Object **obj*, const char **name*, const char **string*, Error ***errp*)

Parameters

`Object *obj`

  the object

`const char *name`

  the name of the property

`const char *string`

  the string that will be used to parse the property value.

`Error **errp`

  returns an error if this function fails

Description

Parses a string and writes the result into a property of an object.

Return

`true` on success, `false` on failure.

> **char * `object_property_print`(Object **obj*, const char **name*, bool *human*, Error ***errp*)**

Parameters

`Object *obj`
: the object

`const char *name`
: the name of the property

`bool human`
: if true, print for human consumption

`Error **errp`
: returns an error if this function fails

Description

Returns a string representation of the value of the property. The caller shall free the string.

> **const char * `object_property_get_type`(Object **obj*, const char **name*, Error ***errp*)**

Parameters

`Object *obj`
: the object

`const char *name`
: the name of the property

`Error **errp`
: returns an error if this function fails

Return

The type name of the property.

> **Object * `object_get_root`(void)**

Parameters

`void`

  no arguments

Return

the root object of the composition tree

> Object * **`object_get_objects_root`**(void)

Parameters

`void`

  no arguments

Description

Get the container object that holds user created object instances. This is the object at path "/objects"

Return

the user object container

> Object * **`object_get_internal_root`**(void)

Parameters

`void`

  no arguments

Description

Get the container object that holds internally used object instances. Any object which is put into this container must not be user visible, and it will not be exposed in the QOM tree.

Return

the internal object container

> const char * **`object_get_canonical_path_component`**(const Object **obj*)

Parameters

`const Object *obj`

  the object

Return

The final component in the object's canonical path. The canonical path is the path within the composition tree starting from the root. `NULL` if the object doesn't have a parent (and thus a canonical path).

> **char * `object_get_canonical_path`(const Object *obj*)**

Parameters

`const Object *obj`

  the object

Return

The canonical path for a object, newly allocated. This is the path within the composition tree starting from the root. Use g_free() to free it.

> **Object * `object_resolve_path`(const char *path*, bool *ambiguous*)**

Parameters

`const char *path`

  the path to resolve

`bool *ambiguous`

  returns true if the path resolution failed because of an ambiguous match

Description

There are two types of supported paths–absolute paths and partial paths.

Absolute paths are derived from the root object and can follow child<> or link<> properties. Since they can follow link<> properties, they can be arbitrarily long. Absolute paths look like absolute filenames and are prefixed with a leading slash.

Partial paths look like relative filenames. They do not begin with a prefix. The matching rules for partial paths are subtle but designed to make specifying objects easy. At each level of the composition tree, the partial path is matched as an absolute path. The first match is not returned. At least two matches are searched for. A successful result is only returned if only one match is found. If more than one match is found, a flag is returned to indicate that the match was ambiguous.

Return

The matched object or NULL on path lookup failure.

Object * `object_resolve_path_type`(const char **path*, const char **typename*, bool **ambiguous*)

Parameters

`const char *path`

the path to resolve

`const char *typename`

the type to look for.

`bool *ambiguous`

returns true if the path resolution failed because of an ambiguous match

Description

This is similar to object_resolve_path. However, when looking for a partial path only matches that implement the given type are considered. This restricts the search and avoids spuriously flagging matches as ambiguous.

For both partial and absolute paths, the return value goes through a dynamic cast to **typename**. This is important if either the link, or the typename itself are of interface types.

Return

The matched object or NULL on path lookup failure.

Object * `object_resolve_path_at`(Object **parent*, const char **path*)

Parameters

`Object *parent`

the object in which to resolve the path

`const char *path`

the path to resolve

Description

This is like object_resolve_path(), except paths not starting with a slash are relative to **parent**.

Return

The resolved object or NULL on path lookup failure.

Object * `object_resolve_path_component`(Object **parent*, const char **part*)

Parameters

`Object *parent`

  the object in which to resolve the path

`const char *part`

  the component to resolve.

Description

This is similar to object_resolve_path with an absolute path, but it only resolves one element (**part**) and takes the others from **parent**.

Return

The resolved object or NULL on path lookup failure.

> **ObjectProperty * `object_property_try_add_child`(Object **obj*, const char **name*, Object **child*, Error ***errp*)**

Parameters

`Object *obj`

  the object to add a property to

`const char *name`

  the name of the property

`Object *child`

  the child object

`Error **errp`

  pointer to error object

Description

Child properties form the composition tree. All objects need to be a child of another object. Objects can only be a child of one object.

There is no way for a child to determine what its parent is. It is not a bidirectional relationship. This is by design.

The value of a child property as a C string will be the child object's canonical path. It can be retrieved using object_property_get_str(). The child object itself can be retrieved using object_property_get_link().

Return

The newly added property on success, or `NULL` on failure.

> **ObjectProperty * `object_property_add_child`(Object **obj*, const char **name*, Object **child*)**

Parameters

`Object *obj`

  the object to add a property to

`const char *name`

  the name of the property

`Object *child`

  the child object

Description

Same as object_property_try_add_child() with **errp** hardcoded to &error_abort

> **void `object_property_allow_set_link`(const Object **obj*, const char **name*, Object **child*, Error ***errp*)**

Parameters

`const Object *obj`

  the object to add a property to

`const char *name`

  the name of the property

`Object *child`

  the child object

`Error **errp`

  pointer to error object

Description

The default implementation of the object_property_add_link() check() callback function. It allows the link property to be set and never returns an error.

> **ObjectProperty * `object_property_add_link`(Object **obj*, const char **name*, const char **type*, Object ***targetp*, void (**check*)(const Object *obj, const char *name, Object *val, Error **errp), ObjectPropertyLinkFlags *flags*)**

Parameters

`Object *obj`

  the object to add a property to

`const char *name`

  the name of the property

`const char *type`

  the qobj type of the link

`Object **targetp`

  a pointer to where the link object reference is stored

`void (*check)(const Object *obj, const char *name, Object *val, Error **errp)`

  callback to veto setting or NULL if the property is read-only

`ObjectPropertyLinkFlags flags`

  additional options for the link

Description

Links establish relationships between objects. Links are unidirectional although two links can be combined to form a bidirectional relationship between objects.

Links form the graph in the object model.

The **check()** callback is invoked when object_property_set_link() is called and can raise an error to prevent the link being set. If **check** is NULL, the property is read-only and cannot be set.

Ownership of the pointer that **child** points to is transferred to the link property. The reference count for ***child** is managed by the property from after the function returns till the property is deleted with object_property_del(). If the **flags** `OBJ_PROP_LINK_STRONG` bit is set, the reference count is decremented when the property is deleted or modified.

Return

The newly added property on success, or `NULL` on failure.

> **ObjectProperty * `object_property_add_str`(Object *obj, const char *name, char *(*get)(Object *, Error **), void (*set)(Object *, const char *, Error **))**

Parameters

`Object *obj`

  the object to add a property to

`const char *name`

  the name of the property

`char *(*get)(Object *, Error **)`

> the getter or NULL if the property is write-only. This function must return a string to be freed by g_free().

`void (*set)(Object *, const char *, Error **)`

> the setter or NULL if the property is read-only

Description

Add a string property using getters/setters. This function will add a property of type 'string'.

Return

The newly added property on success, or `NULL` on failure.

> **ObjectProperty * `object_property_add_bool`(Object **obj*, const char **name*, bool (**get*)(Object *, Error **), void (*set)(Object *, bool, Error **))**

Parameters

`Object *obj`

> the object to add a property to

`const char *name`

> the name of the property

`bool (*get)(Object *, Error **)`

> the getter or NULL if the property is write-only.

`void (*set)(Object *, bool, Error **)`

> the setter or NULL if the property is read-only

Description

Add a bool property using getters/setters. This function will add a property of type 'bool'.

Return

The newly added property on success, or `NULL` on failure.

> **ObjectProperty * `object_property_add_enum`(Object **obj*, const char **name*, const char **typename*, const QEnumLookup **lookup*, int (**get*)(Object *, Error **), void (*set)(Object *, int, Error **))**

Parameters

`Object *obj`

the object to add a property to

`const char *name`

the name of the property

`const char *typename`

the name of the enum data type

`const QEnumLookup *lookup`

enum value namelookup table

`int (*get)(Object *, Error **)`

the getter or `NULL` if the property is write-only.

`void (*set)(Object *, int, Error **)`

the setter or `NULL` if the property is read-only

Description

Add an enum property using getters/setters. This function will add a property of type '**typename**'.

Return

The newly added property on success, or `NULL` on failure.

> **ObjectProperty * `object_property_add_tm`(Object **obj*, const char **name*, void (**get*)(Object *, struct tm *, Error **))**

Parameters

`Object *obj`

the object to add a property to

`const char *name`

the name of the property

`void (*get)(Object *, struct tm *, Error **)`

the getter or NULL if the property is write-only.

Description

Add a read-only struct tm valued property using a getter function. This function will add a property of type 'struct tm'.

Return

The newly added property on success, or `NULL` on failure.

> **ObjectProperty * `object_property_add_uint8_ptr`(Object **obj*, const char **name*, const uint8_t **v*, ObjectPropertyFlags *flags*)**

Parameters

`Object *obj`

    the object to add a property to

`const char *name`

    the name of the property

`const uint8_t *v`

    pointer to value

`ObjectPropertyFlags flags`

    bitwise-or’d ObjectPropertyFlags

Description

Add an integer property in memory. This function will add a property of type ‘uint8’.

Return

The newly added property on success, or `NULL` on failure.

> **ObjectProperty * `object_property_add_uint16_ptr`(Object **obj*, const char **name*, const uint16_t **v*, ObjectPropertyFlags *flags*)**

Parameters

`Object *obj`

    the object to add a property to

`const char *name`

    the name of the property

`const uint16_t *v`

    pointer to value

`ObjectPropertyFlags flags`

    bitwise-or’d ObjectPropertyFlags

Description

Add an integer property in memory. This function will add a property of type ‘uint16’.

Return

The newly added property on success, or `NULL` on failure.

> **ObjectProperty * `object_property_add_uint32_ptr`(Object *obj, const char *name, const uint32_t *v, ObjectPropertyFlags flags)**

Parameters

`Object *obj`
: the object to add a property to

`const char *name`
: the name of the property

`const uint32_t *v`
: pointer to value

`ObjectPropertyFlags flags`
: bitwise-or'd ObjectPropertyFlags

Description

Add an integer property in memory. This function will add a property of type 'uint32'.

Return

The newly added property on success, or `NULL` on failure.

> **ObjectProperty * `object_property_add_uint64_ptr`(Object *obj, const char *name, const uint64_t *v, ObjectPropertyFlags flags)**

Parameters

`Object *obj`
: the object to add a property to

`const char *name`
: the name of the property

`const uint64_t *v`
: pointer to value

`ObjectPropertyFlags flags`
: bitwise-or'd ObjectPropertyFlags

Description

Add an integer property in memory. This function will add a property of type 'uint64'.

Return

The newly added property on success, or `NULL` on failure.

> **ObjectProperty * `object_property_add_alias`(Object **obj*, const char **name*, Object **target_obj*, const char **target_name*)**

Parameters

`Object *obj`

    the object to add a property to

`const char *name`

    the name of the property

`Object *target_obj`

    the object to forward property access to

`const char *target_name`

    the name of the property on the forwarded object

Description

Add an alias for a property on an object. This function will add a property of the same type as the forwarded property.

The caller must ensure that **target_obj** stays alive as long as this property exists. In the case of a child object or an alias on the same object this will be the case. For aliases to other objects the caller is responsible for taking a reference.

Return

The newly added property on success, or `NULL` on failure.

> **ObjectProperty * `object_property_add_const_link`(Object **obj*, const char **name*, Object **target*)**

Parameters

`Object *obj`

    the object to add a property to

`const char *name`

    the name of the property

`Object *target`

    the object to be referred by the link

Description

Add an unmodifiable link for a property on an object. This function will add a property of type link<TYPE> where TYPE is the type of **target**.

The caller must ensure that **target** stays alive as long as this property exists. In the case **target** is a child of **obj**, this will be the case. Otherwise, the caller is responsible for taking a reference.

Return

The newly added property on success, or `NULL` on failure.

> **void object_property_set_description(Object *obj, const char *name, const char *description)**

Parameters

`Object *obj`
: the object owning the property

`const char *name`
: the name of the property

`const char *description`
: the description of the property on the object

Description

Set an object property's description.

Return

`true` on success, `false` on failure.

> **int object_child_foreach(Object *obj, int (*fn)(Object *child, void *opaque), void *opaque)**

Parameters

`Object *obj`
: the object whose children will be navigated

`int (*fn)(Object *child, void *opaque)`
: the iterator function to be called

`void *opaque`
: an opaque value that will be passed to the iterator

Description

Call **fn** passing each child of **obj** and **opaque** to it, until **fn** returns non-zero.

It is forbidden to add or remove children from **obj** from the **fn** callback.

Return

The last value returned by **fn**, or 0 if there is no child.

> **int object_child_foreach_recursive(Object *obj, int (*fn)(Object *child, void *opaque), void *opaque)**

Parameters

`Object *obj`

the object whose children will be navigated

`int (*fn)(Object *child, void *opaque)`

the iterator function to be called

`void *opaque`

an opaque value that will be passed to the iterator

Description

Call **fn** passing each child of **obj** and **opaque** to it, until **fn** returns non-zero. Calls recursively, all child nodes of **obj** will also be passed all the way down to the leaf nodes of the tree. Depth first ordering.

It is forbidden to add or remove children from **obj** (or its child nodes) from the **fn** callback.

Return

The last value returned by **fn**, or 0 if there is no child.

> **Object * container_get(Object *root, const char *path)**

Parameters

`Object *root`

root of the #path, e.g., object_get_root()

`const char *path`

path to the container

Description

Return a container object whose path is **path**. Create more containers along the path if necessary.

Return

the container object.

> **size_t `object_type_get_instance_size`(const char **typename*)**

Parameters

`const char *typename`

  Name of the Type whose instance_size is required

Description

Returns the instance_size of the given **typename**.

> **char * `object_property_help`(const char **name*, const char **type*, QObject **defval*, const char **description*)**

Parameters

`const char *name`

  the name of the property

`const char *type`

  the type of the property

`QObject *defval`

  the default value

`const char *description`

  description of the property

Return

a user-friendly formatted string describing the property for help purposes.