

The lt3rawobjects package

Paolo De Donato

Released on 2023/02/16 Version 2.3-beta-2

Contents

1	Introduction	2
2	Addresses	2
3	Objects	3
4	Items	3
4.1	Constants	4
4.2	Methods	4
4.3	Members	4
5	Object members	4
5.1	Create a pointer member	4
5.2	Clone the inner structure	5
5.3	Embedded objects	6
6	Library functions	7
6.1	Common functions	7
6.2	Base object functions	7
6.3	Members	8
6.4	Constants	9
6.5	Methods	10
6.6	Creation of constants	11
6.7	Macros	11
6.8	Proxies and object creation	12
7	Examples	14
8	Implementation	16

1 Introduction

Package `lt3rawobjects` introduces a new mechanism to create and manage structured data called “objects” like the well known C structures. The functions exported by this package are quite low level, and many important mechanisms like member protection and name resolution aren’t already defined and should be introduced by intermediate packages. Higher level libraries built on top of `lt3rawobjects` could also implement an improved and simplified syntax since the main focus of `lt3rawobjects` is versatility and expandability rather than common usage.

This packages follows the [SemVer](https://semver.org/) specification (<https://semver.org/>). In particular any major version update (for example from 1.2 to 2.0) may introduce incompatible changes and so it’s not advisable to work with different packages that require different major versions of `lt3rawobjects`. Instead changes introduced in minor and patch version updates are always backward compatible, and any withdrawn function is declared deprecated instead of being removed.

2 Addresses

In this package a *pure address* is any string without spaces (so a sequence of tokens with category code 12 “other”) that uniquely identifies a resource or an entity. An example of pure address is the name of a control sequence `\<name>` that can be obtained by full expanding `\cs_to_str:N \<name>`. Instead an *expanded address* is a token list that contains only tokens with category code 11 (letters) or 12 (other) that can be directly converted to a pure address with a simple call to `\tl_to_str:n` or by assigning it to a string variable.

An *address* is instead a fully expandable token list which full expansion is an expanded address, where full expansion means the expansion process performed inside `c`, `x` and `e` parameters. Moreover, any address should be fully expandable according to the rules of `x` and `e` parameter types with same results, and the name of control sequence resulting from a `c`-type expansion of such address must be equal to its full expansion. For these reasons addresses should not contain parameter tokens like `#` (because they’re threat differently by `x` and `e`) or control sequences that prevents expansion like `\exp_not:n` (because they leave unexpanded control sequences after an `x` or `e` expansion, and expanded addresses can’t have control sequences inside them). In particular, `\tl_to_str:n{ ## }` is *not* a valid address (assuming standard category codes).

Addresses could be not full expanded inside an `f` argument, thus an address expanded in an `f` argument should be `x`, `e` or `c` expended later to get the actual pure address. If you need to fully expand an address in an `f` argument (because, for example, your macro should be fully expandable and your engine is too old to support `e` expansion efficiently) then you can put your address inside `\rwoobj_address_f:n` and pass them to your function. For example,

```
\your_function:f{ \rwoobj_address_f:n { your \address } }
```

Remember that `\rwoobj_address_f:n` only works with addresses, can’t be used to fully expand any token list.

Like functions and variables names, pure addresses should follows some basic naming conventions in order to avoid clashes between addresses in different modules. Each pure

address starts with the $\langle module \rangle$ name in which such address is allocated, then an underscore (`_`) and the $\langle identifier \rangle$ that uniquely identifies the resource inside the module. The $\langle module \rangle$ should contain only lowercase ASCII letters.

A *pointer* is just a L^AT_EX3 string variable that holds a pure address. We don't enforce to use `str` or any special suffix to denote pointers so you're free to use `str` or a custom $\langle type \rangle$ as suffix for your pointers in order to distinguish between them according to their type.

In `lt3rawobjects` all the macros ending with `_adr` or `_address` are fully expandable and can be used to compose valid addresses as explained in their documentation.

3 Objects

An *object* is just a collection of several related entities called *item*. Objects are themselves entities so they have addresses and could be contained inside other objects. Objects addresses are also used to compose the addresses of each of their inner entity, thus different objects can have items with the same name without clashing each other. Each object is uniquely identified by its pure address, which is composed by a $\langle module \rangle$ and an $\langle identifier \rangle$ as explained before. The use of underscore character in objects identifiers is reserved. You can retrieve the address of an object via the `\object_address:nn` function.

Objects are always created from already existing objects. An object that can be used to create other objects is called *proxy*, and the proxy that has created an object is its *generator*. In the `rawobjects` module is already allocated a particular proxy that can be used to create every other proxy. Its identifier is just `proxy` and its pure address is stored in `\c_proxy_address_str`. The functions `\object_create` can be used to create new objects.

4 Items

Remember that objects are just a collection of different items uniquely identified by a pure address. Here an item could be one of the following entities:

- a L^AT_EX3 variable, in which case the item is called *member*;
- a L^AT_EX3 constant, in which case the item is called just *constant*;
- a L^AT_EX3 function, in which case the item is called *method*;
- generic control sequences, in which case the item is called simply *macro*;
- an entire object, in which case the item is called *embedded object*.

Objects could be declared *local* or *global*. The only difference between a local and a global object is the scope of their members (that are L^AT_EX3 variables). You should always create global object unless you specifically need local members.

4.1 Constants

Constants in an object could be *near* and *remote*. A near constant is just a constant declared in such object and could be referred only by it, instead a remote constant is declared inside its generator and can be referred by any object created from that proxy, thus it's shared between all the generated objects. Functions in this library that work with near constants usually contain `ncmember` in their names, whereas those involving remote constants contain `rcmember` instead.

Both near and remote constants are created in the same way via the `_newconst` functions, however remote constant should be created in a proxy whereas near constant are created directly in the target object.

4.2 Methods

Methods are \LaTeX functions that can't be changed once they're created. Like constant, methods could be near or remote. Moreover, functions in this library dealing with near methods contain `ncmethod` whereas those dealing with remote methods contain `rcmethod` in their names.

4.3 Members

Members are just mutable \LaTeX variables. You can manually create new members in already existing objects or you can put the definition of a new member directly in a proxy with the `\proxy_push_member` functions. In this way all the objects created with that proxy will have a member according to such definition. If the object is local/global then all its members are automatically local/global.

A member can be *tracked* or *not tracked*. A tracked member have additional information, like its type, stored in the object or in its generator. In particular, you don't need to specify the type of a tracked member and some functions in `lt3rawobjects` are able to retrieve the required information. All the members declared in the generator are automatically tracked.

5 Object members

Sometimes it's necessary to store an instance of an object inside another object, since objects are structured entities that can't be entirely contained in a single \LaTeX variable you can't just put it inside a member or constant. However, there are some very easy workarounds to insert object instances as items of other objects.

For example, we're in module `MOD` and we have an object with id `PAR`. We want to provide `PAR` with an item that holds an instance of an object created by proxy `PRX`. We can achieve this in three ways:

5.1 Create a pointer member

We first create a new object from `PRX`

```
1 \object_create:nnn
2 { \object_address:nn { MOD }{ PRX } }{ MOD }{ INST }
```

then we create an **str** member in **PAR** that will hold the address of the newly created object.

```

1  \object_new_member:nnn
2  {
3      \object_address:nn { MOD }{ PAR }
4  }{ pointer }{ str }
5
6  \object_member_set:nnnx
7  {
8      \object_address:nn { MOD }{ PAR }
9  }
10 { pointer }{ str }
11 {
12     \object_address:nn { MOD }{ INST }
13 }

```

You can then get the pointed object by just using the **pointer** member. Notice that you're not forced to use the **str** type for the pointer member, but you can also use **tl** or any custom $\langle type \rangle$. In the latter case be sure to at least define the following functions: $\backslash\langle type \rangle_new:c$, $\backslash\langle type \rangle_set:cn$ and $\backslash\langle type \rangle_use:c$.

Advantages

- Simple and no additional function needed to create and manage included objects;
- you can share the same object between different containers;
- included objects are objects too, you can use address stored in pointer member just like any object address.

Disadvantages

- You must manually create both the objects and link them;
- if you forgot to properly initialize the pointer member it'll contain the **null address** (the empty string). Despite other programming languages the null address is not treated specially by **lt3rawobjects**, which makes finding null pointer errors more difficult.

5.2 Clone the inner structure

Another solution is to copy the members declared in **PRX** to **PAR**. For example, if in **PRX** are declared a member with name **x** and type **str**, and a member with name **y** and type **int** then

```

1  \object_new_member:nnn
2  {
3      \object_address:nn { MOD }{ PAR }
4  }{ prx-x }{ str }
5  \object_new_member:nnn
6  {

```

```

7   \object_address:nn { MOD }{ PAR }
8   }{ prx-y }{ int }

```

Advantages

- Very simple;
- no hidden item is created, this procedure has the lowest overhead among all the proposed solutions here.

Disadvantages

- If you need the original instance of the stored object then you should create a temporary object and manually copy each item to it. Don't use this method if you later need to retrieve the stored object entirely and not only its items.

5.3 Embedded objects

From `lt3rawobjects 2.2` you can put *embedded objects* inside objects. Embedded objects are created with `\embedded_create` function

```

1   \embedded_create:nnn
2   {
3     \object_address:nn { MOD }{ PAR }
4   }
5   { PRX }{ emb }

```

and addresses of emmbedded objects can be retrieved with function `\object_embedded_adr`. You can also put the definition of embedded objects in a proxy by using `\proxy_push_embedded` just like `\proxy_push_member`.

Advantages

- You can put a declaration inside a proxy so that embedded objects are automatically created during creation of parent object;
- included objects are objects too, you can use address stored in pointer member just like any object address.

Disadvantages

- Needs additional functions available for version 2.2 or later;
- embedded objects must have the same scope and visibility of parent one;
- creating objects also creates additional hidden variables, taking so (little) additional space.

6 Library functions

6.1 Common functions

<code>\rwobj_address_f:n</code>	★	<code>\rwobj_address_f:n {<address>}</code>
---------------------------------	---	---

Fully expand an address in an f-type argument.
From: 2.3

6.2 Base object functions

<code>\object_address:nn</code>	☆	<code>\object_address:nn {<module>} {<id>}</code>
---------------------------------	---	---

Composes the address of object in module *<module>* with identifier *<id>* and places it in the input stream. Notice that both *<module>* and *<id>* are converted to strings before composing them in the address, so they shouldn't contain any command inside.
From: 1.0

<code>\object_address_set:Nnn</code>	★	<code>\object_address_set:nn <str var> {<module>} {<id>}</code>
<code>\object_address_gset:Nnn</code>	★	

Stores the address of selected object inside the string variable *<str var>*.
From: 1.1

<code>\object_embedded_adr:nn</code>	☆	<code>\object_embedded_adr:nn {<address>} {<id>}</code>
<code>\object_embedded_adr:Vn</code>	☆	

Compose the address of embedded object with name *<id>* inside the parent object with address *<address>*. Since an embedded object is also an object you can use this function for any function that accepts object addresses as an argument.
From: 2.2

<code>\object_if_exist_p:n</code>	★	<code>\object_if_exist_p:n {<address>}</code>
<code>\object_if_exist_p:V</code>	★	<code>\object_if_exist:nTF {<address>} {<true code>} {<false code>}</code>
<code>\object_if_exist:nTF</code>	★	Tests if an object was instantiated at the specified address.
<code>\object_if_exist:VTF</code>	★	From: 1.0

<code>\object_get_module:n</code>	★	<code>\object_get_module:n {<address>}</code>
<code>\object_get_module:V</code>	★	<code>\object_get_proxy_adr:n {<address>}</code>
<code>\object_get_proxy_adr:n</code>	★	Get the object module and its generator.
<code>\object_get_proxy_adr:V</code>	★	From: 1.0

<code>\object_if_local_p:n</code>	★	<code>\object_if_local_p:n {<address>}</code>
<code>\object_if_local_p:V</code>	★	<code>\object_if_local:nTF {<address>} {<true code>} {<false code>}</code>
<code>\object_if_local:nTF</code>	★	Tests if the object is local or global.
<code>\object_if_local:VTF</code>	★	From: 1.0
<code>\object_if_global_p:n</code>	★	
<code>\object_if_global_p:V</code>	★	
<code>\object_if_global:nTF</code>	★	
<code>\object_if_global:VTF</code>	★	

```

\object_if_public_p:n * \object_if_public_p:n {\address}
\object_if_public_p:V * \object_if_public:nTF {\address} {\true code} {\false code}
\object_if_public:nTF * Tests if the object is public or private.
\object_if_public:VTF *
                        From: 1.0
\object_if_private_p:n *
\object_if_private_p:V *
\object_if_private:nTF *
\object_if_private:VTF *

```

6.3 Members

```

\object_member_adr:nnn ☆ \object_member_adr:nnn {\address} {\member name} {\member type}
\object_member_adr:(Vnn|nnv) ☆ \object_member_adr:nn {\address} {\member name}
\object_member_adr:nn ☆
\object_member_adr:Vn ☆

```

Fully expands to the address of specified member variable. If the member is tracked then you can omit the type field.

From: 1.0

```

\object_member_if_exist_p:nnn * \object_member_if_exist_p:nnn {\address} {\member name} {\member
\object_member_if_exist_p:Vnn * type}
\object_member_if_exist:nnnTF * \object_member_if_exist:nnnTF {\address} {\member name} {\member
\object_member_if_exist:VnnTF * type} {\true code} {\false code}
\object_member_if_exist_p:nn * \object_member_if_exist_p:nn {\address} {\member name}
\object_member_if_exist_p:Vn * \object_member_if_exist:nnTF {\address} {\member name} {\true code}
\object_member_if_exist:nnTF * {\false code}
\object_member_if_exist:VnTF *

```

Tests if the specified member exist. The :nn version returns true if and only if the specified member exists and it is a tracked member.

From: 2.0

```

\object_member_type:nn * \object_member_type:nn {\address} {\member name}
\object_member_type:Vn *

```

Fully expands to the type of member *{\member name}*. Use this function only with member variables specified in the generator proxy, not with other member variables.

From: 1.0

```

\object_new_member:nnn \object_new_member:nnn {\address} {\member name} {\member type}
\object_new_member:(Vnn|nnv)

```

Creates a new member variable with specified name and type. You can't retrieve the type of these variables with `\object_member_type` functions.

From: 1.0

```

\object_member_use:nnn * \object_member_use:nnn {\address} {\member name} {\member type}
\object_member_use:(Vnn|nnv) * \object_member_use:nn {\address} {\member name}
\object_member_use:nn *
\object_member_use:Vn *

```

Uses the specified member variable.

From: 1.0

<code>\object_member_set:nnnn</code>	<code>\object_member_set:nnnn {<address>} {<member name>} {<member type>}</code>
<code>\object_member_set:(nnvn Vnnn)</code>	<code>{<value>}</code>
<code>\object_member_set:nnn</code>	<code>\object_member_set:nnn {<address>} {<member name>} {<value>}</code>
<code>\object_member_set:Vnn</code>	

Sets the value of specified member to `{<value>}`. It calls implicitly `\<member type>_(g)set:cn` then be sure to define it before calling this method.

From: 2.1

<code>\object_member_set_eq:nnnN</code>	<code>\object_member_set_eq:nnnN {<address>} {<member name>}</code>
<code>\object_member_set_eq:(nnvN VnnN nnnc Vnnnc)</code>	<code>{<member type>} {<variable>}</code>
<code>\object_member_set_eq:nnN</code>	<code>\object_member_set_eq:nnN {<address>} {<member name>}</code>
<code>\object_member_set_eq:(VnN nnnc Vnc)</code>	<code>{<variable>}</code>

Sets the value of specified member equal to the value of `<variable>`.

From: 1.0

6.4 Constants

<code>\object_ncmember_adr:nnn</code>	☆ <code>\object_ncmember_adr:nnn {<address>} {<member name>} {<member type>}</code>
<code>\object_ncmember_adr:(Vnn vnn)</code>	☆
<code>\object_rcmember_adr:nnn</code>	☆
<code>\object_rcmember_adr:Vnn</code>	☆

Fully expands to the address of specified near/remote constant member.

From: 2.0

<code>\object_ncmember_if_exist_p:nnn</code>	★ <code>\object_ncmember_if_exist_p:nnn {<address>} {<member name>} {<member</code>
<code>\object_ncmember_if_exist_p:Vnn</code>	★ <code>type>}</code>
<code>\object_ncmember_if_exist:nnnTF</code>	★ <code>\object_ncmember_if_exist:nnnTF {<address>} {<member name>} {<member</code>
<code>\object_ncmember_if_exist:VnnTF</code>	★ <code>type>} {<true code>} {<false code>}</code>
<code>\object_rcmember_if_exist_p:nnn</code>	★
<code>\object_rcmember_if_exist_p:Vnn</code>	★
<code>\object_rcmember_if_exist:nnnTF</code>	★
<code>\object_rcmember_if_exist:VnnTF</code>	★

Tests if the specified member constant exist.

From: 2.0

<code>\object_ncmember_use:nnn</code>	★ <code>\object_ncmember_use:nnn {<address>} {<member name>} {<member type>}</code>
<code>\object_ncmember_use:Vnn</code>	★
<code>\object_rcmember_use:nnn</code>	★ Uses the specified near/remote constant member.
<code>\object_rcmember_use:Vnn</code>	★ From: 2.0

6.5 Methods

<code>\object_ncmethod_adr:nnn</code>	☆	<code>\object_ncmethod_adr:nnn {⟨address⟩} {⟨method name⟩} {⟨method</code>
<code>\object_ncmethod_adr:(Vnn vnn)</code>	☆	<code>variant⟩}</code>
<code>\object_rcmethod_adr:nnn</code>	☆	
<code>\object_rcmethod_adr:Vnn</code>	☆	

Fully expands to the address of the specified

- near constant method if `\object_ncmethod_adr` is used;
- remote constant method if `\object_rcmethod_adr` is used.

From: 2.0

<code>\object_ncmethod_if_exist_p:nnn</code>	★	<code>\object_ncmethod_if_exist_p:nnn {⟨address⟩} {⟨method name⟩} {⟨method</code>
<code>\object_ncmethod_if_exist_p:Vnn</code>	★	<code>variant⟩}</code>
<code>\object_ncmethod_if_exist:nnnTF</code>	★	<code>\object_ncmethod_if_exist:nnnTF {⟨address⟩} {⟨method name⟩} {⟨method</code>
<code>\object_ncmethod_if_exist:VnnTF</code>	★	<code>variant⟩} {⟨true code⟩} {⟨false code⟩}</code>
<code>\object_rcmethod_if_exist_p:nnn</code>	★	
<code>\object_rcmethod_if_exist_p:Vnn</code>	★	
<code>\object_rcmethod_if_exist:nnnTF</code>	★	
<code>\object_rcmethod_if_exist:VnnTF</code>	★	

Tests if the specified method constant exist.

From: 2.0

<code>\object_new_cmethod:nnnn</code>	<code>\object_new_cmethod:nnnn {⟨address⟩} {⟨method name⟩} {⟨method arguments⟩} {⟨code⟩}</code>
<code>\object_new_cmethod:Vnnn</code>	

Creates a new method with specified name and argument types. The `{⟨method arguments⟩}` should be a string composed only by n and N characters that are passed to `\cs_new:Nn`.

From: 2.0

<code>\object_ncmethod_call:nnn</code>	★	<code>\object_ncmethod_call:nnn {⟨address⟩} {⟨method name⟩} {⟨method variant⟩}</code>
<code>\object_ncmethod_call:Vnn</code>	★	
<code>\object_rcmethod_call:nnn</code>	★	
<code>\object_rcmethod_call:Vnn</code>	★	

Calls the specified method. This function is expandable if and only if the specified method was not declared `protected`.

From: 2.0

6.6 Creation of constants

<code>\object_newconst_tl:nnn</code>	<code>\object_newconst_⟨type⟩:nnn {⟨address⟩} {⟨constant name⟩} {⟨value⟩}</code>
<code>\object_newconst_tl:Vnn</code>	Creates a constant variable with type <code>⟨type⟩</code> and sets its value to <code>⟨value⟩</code> .
<code>\object_newconst_str:nnn</code>	From: 1.1
<code>\object_newconst_str:Vnn</code>	
<code>\object_newconst_int:nnn</code>	
<code>\object_newconst_int:Vnn</code>	
<code>\object_newconst_clist:nnn</code>	
<code>\object_newconst_clist:Vnn</code>	
<code>\object_newconst_dim:nnn</code>	
<code>\object_newconst_dim:Vnn</code>	
<code>\object_newconst_skip:nnn</code>	
<code>\object_newconst_skip:Vnn</code>	
<code>\object_newconst_fp:nnn</code>	
<code>\object_newconst_fp:Vnn</code>	

<code>\object_newconst_seq_from_clist:nnn</code>	<code>\object_newconst_seq_from_clist:nnn {⟨address⟩} {⟨constant name⟩}</code>
<code>\object_newconst_seq_from_clist:Vnn</code>	<code>{⟨comma-list⟩}</code>

Creates a `seq` constant which is set to contain all the items in `⟨comma-list⟩`.
From: 1.1

<code>\object_newconst_prop_from_keyval:nnn</code>	<code>\object_newconst_prop_from_keyval:nnn {⟨address⟩} {⟨constant name⟩}</code>
<code>\object_newconst_prop_from_keyval:Vnn</code>	<code>{</code> <code> ⟨key⟩ = ⟨value⟩, ...</code> <code>}</code>

Creates a `prop` constant which is set to contain all the specified key-value pairs.
From: 1.1

<code>\object_newconst:nnnn</code>	<code>\object_newconst:nnnn {⟨address⟩} {⟨constant name⟩} {⟨type⟩} {⟨value⟩}</code>
	Invokes <code>\⟨type⟩_const:cn</code> to create the specified constant.
	From: 2.1

6.7 Macros

<code>\object_macro_adr:nn</code>	☆ <code>\object_macro_adr:nn {⟨address⟩} {⟨macro name⟩}</code>
<code>\object_macro_adr:Vn</code>	☆ Address of specified macro.
	From: 2.2

<code>\object_macro_use:nn</code>	★ <code>\object_macro_use:nn {⟨address⟩} {⟨macro name⟩}</code>
<code>\object_macro_use:Vn</code>	★ Uses the specified macro. This function is expandable if and only if the specified macro is it.
	From: 2.2

There isn't any standard function to create macros, and macro declarations can't be inserted in a `proxy` object. In fact a macro is just an unspecialized control sequence at the disposal of users that usually already know how to implement them.

6.8 Proxies and object creation

<code>\object_if_proxy_p:n</code>	<code>*</code>	<code>\object_if_proxy_p:n {⟨address⟩}</code>
<code>\object_if_proxy_p:V</code>	<code>*</code>	<code>\object_if_proxy:nTF {⟨address⟩} {⟨true code⟩} {⟨false code⟩}</code>
<code>\object_if_proxy:nTF</code>	<code>*</code>	Test if the specified object is a proxy object.
<code>\object_if_proxy:VTF</code>	<code>*</code>	From: 1.0

<code>\object_test_proxy_p:nn</code>	<code>*</code>	<code>\object_test_proxy_p:nn {⟨object address⟩} {⟨proxy address⟩}</code>
<code>\object_test_proxy_p:Vn</code>	<code>*</code>	<code>\object_test_proxy:nnTF {⟨object address⟩} {⟨proxy address⟩} {⟨true code⟩} {⟨false code⟩}</code>
<code>\object_test_proxy:nnTF</code>	<code>*</code>	
<code>\object_test_proxy:VnTF</code>	<code>*</code>	Test if the specified object is generated by the selected proxy, where <i>⟨proxy variable⟩</i> is a string variable holding the proxy address.

TeXhackers note: Remember that this command uses internally an `e` expansion so in older engines (any different from Lua[®]TeX before 2019) it'll require slow processing. Don't use it in speed critical parts, instead use `\object_test_proxy:nN`.

From: 2.0

<code>\object_test_proxy_p:nN</code>	<code>*</code>	<code>\object_test_proxy_p:nN {⟨object address⟩} {⟨proxy variable⟩}</code>
<code>\object_test_proxy_p:VN</code>	<code>*</code>	<code>\object_test_proxy:nNTF {⟨object address⟩} {⟨proxy variable⟩} {⟨true code⟩} {⟨false code⟩}</code>
<code>\object_test_proxy:nNTF</code>	<code>*</code>	
<code>\object_test_proxy:VNNTF</code>	<code>*</code>	Test if the specified object is generated by the selected proxy, where <i>⟨proxy variable⟩</i> is a string variable holding the proxy address. The <code>:nN</code> variant don't use <code>e</code> expansion, instead of <code>:nn</code> command, so it can be safely used with older compilers.

From: 2.0

<code>\c_proxy_address_str</code>	The address of the proxy object in the <code>rawobjects</code> module.
-----------------------------------	--

From: 1.0

<code>\object_create:nnnNN</code>	<code>\object_create:nnnNN {⟨proxy address⟩} {⟨module⟩} {⟨id⟩} {⟨scope⟩} {⟨visibility⟩}</code>
<code>\object_create:VnnNN</code>	

Creates an object by using the proxy at *⟨proxy address⟩* and the specified parameters. Use this function only if you need to create private objects (at present private objects are functionally equivalent to public objects) or if you need to compile your project with an old version of this library (< 2.3).

From: 1.0

<code>\object_create:nnnN</code>	<code>\object_create:nnnN {⟨proxy address⟩} {⟨module⟩} {⟨id⟩} {⟨scope⟩}</code>
<code>\object_create:VnnN</code>	<code>\object_create:nnn {⟨proxy address⟩} {⟨module⟩} {⟨id⟩}</code>
<code>\object_create:nnn</code>	
<code>\object_create:Vnn</code>	Same as <code>\object_create:nnnNN</code> but both create only public objects, and the <code>:nnn</code> version only global ones. Always use these two function instead of <code>\object_create:nnnNN</code> unless you strictly need private objects.

From: 2.3

<code>\embedded_create:nnn</code>	<code>\embedded_create:nnn {⟨parent object⟩} {⟨proxy address⟩} {⟨id⟩}</code>
<code>\embedded_create:(Vnn nvn)</code>	

Creates an embedded object with name *⟨id⟩* inside *⟨parent object⟩*.

From: 2.2

<code>\c_object_local_str</code>	Possible values for $\langle scope \rangle$ parameter.
<code>\c_object_global_str</code>	From: 1.0

<code>\c_object_public_str</code>	Possible values for $\langle visibility \rangle$ parameter.
<code>\c_object_private_str</code>	From: 1.0

<code>\object_create_set:NnnnNN</code>	<code>\object_create_set:NnnnNN</code> $\langle str var \rangle$ $\{\langle proxy address \rangle\}$ $\{\langle module \rangle\}$
<code>\object_create_set:(NVnnNN NnnfNN)</code>	$\{\langle id \rangle\}$ $\langle scope \rangle$ $\langle visibility \rangle$
<code>\object_create_gset:NnnnNN</code>	
<code>\object_create_gset:(NVnnNN NnnfNN)</code>	

Creates an object and sets its fully expanded address inside $\langle str var \rangle$.
From: 1.0

<code>\object_allocate_incr:NNnnNN</code>	<code>\object_allocate_incr:NNnnNN</code> $\langle str var \rangle$ $\langle int var \rangle$ $\{\langle proxy address \rangle\}$
<code>\object_allocate_incr:NNVnNN</code>	$\{\langle module \rangle\}$ $\langle scope \rangle$ $\langle visibility \rangle$
<code>\object_gallocate_incr:NNnnNN</code>	
<code>\object_gallocate_incr:NNVnNN</code>	
<code>\object_allocate_gincr:NNnnNN</code>	
<code>\object_allocate_gincr:NNVnNN</code>	
<code>\object_gallocate_gincr:NNnnNN</code>	
<code>\object_gallocate_gincr:NNVnNN</code>	

Build a new object address with module $\langle module \rangle$ and an identifier generated from $\langle proxy address \rangle$ and the integer contained inside $\langle int var \rangle$, then increments $\langle int var \rangle$. This is very useful when you need to create a lot of objects, each of them on a different address. the `_incr` version increases $\langle int var \rangle$ locally whereas `_gincr` does it globally.
From: 1.1

<code>\proxy_create:nnN</code>	<code>\proxy_create:nnN</code> $\{\langle module \rangle\}$ $\{\langle id \rangle\}$ $\langle visibility \rangle$
<code>\proxy_create_set:NnnN</code>	<code>\proxy_create_set:NnnN</code> $\langle str var \rangle$ $\{\langle module \rangle\}$ $\{\langle id \rangle\}$ $\langle visibility \rangle$
<code>\proxy_create_gset:NnnN</code>	These commands are deprecated because proxies should be global and public. Use instead <code>\proxy_create:nn</code> , <code>\proxy_create_set:Nnn</code> and <code>\proxy_create_gset:Nnn</code> . From: 1.0 Deprecated in: 2.3

<code>\proxy_create:nn</code>	<code>\proxy_create:nn</code> $\{\langle module \rangle\}$ $\{\langle id \rangle\}$
<code>\proxy_create_set:Nnn</code>	<code>\proxy_create_set:Nnn</code> $\langle str var \rangle$ $\{\langle module \rangle\}$ $\{\langle id \rangle\}$
<code>\proxy_create_gset:Nnn</code>	Creates a global public proxy object. From: 2.3

<code>\proxy_push_member:nnn</code>	<code>\proxy_push_member:nnn</code> $\{\langle proxy address \rangle\}$ $\{\langle member name \rangle\}$ $\{\langle member type \rangle\}$
<code>\proxy_push_member:Vnn</code>	Updates a proxy object with a new member specification, so that every subsequential object created with this proxy will have a member variable with the specified name and type that can be retrieved with <code>\object_member_type</code> functions. From: 1.0

<hr/>	
<code>\proxy_push_embedded:nnn</code>	<code>\proxy_push_embedded:nnn {<proxy address>} {<embedded object name>} {<embedded object proxy>}</code>
<code>\proxy_push_embedded:Vnn</code>	
<hr/>	
	Updates a proxy object with a new embedded object specification. From: 2.2
<hr/>	
<code>\proxy_add_initializer:nN</code>	<code>\proxy_add_initializer:nN {<proxy address>} <initializer></code>
<code>\proxy_add_initializer:Vn</code>	
<hr/>	
	Pushes a new initializer that will be executed on each created objects. An initializer is a function that should accept five arguments in this order: <ul style="list-style-type: none"> • the full expanded address of used proxy as an <code>n</code> argument; • the module name as an <code>n</code> argument; • the full expanded address of created object as an <code>n</code> argument. <p>Initializer will be executed in the same order they're added.</p>
<hr/>	
<code>\object_assign:nn</code>	<code>\object_assign:nn {<to address>} {<from address>}</code>
<code>\object_assign:(Vn nV VV)</code>	
<hr/>	
	Assigns the content of each variable of object at <code><from address></code> to each corresponsive variable in <code><to address></code> . Both the objects should be created with the same proxy object and only variables listed in the proxy are assigned. From: 1.0

7 Examples

Example 1

Create a public proxy with id `myproxy` with the specification of a single member variable with name `myvar` and type `tl`, then set its address inside `\g_myproxy_str`.

```

1 \str_new:N \g_myproxy_str
2 \proxy_create_gset:Nnn \g_myproxy_str { example }{ myproxy }
3 \proxy_push_member:Vnn \g_myproxy_str { myvar }{ tl }

```

Then create a new object with name `myobj` with that proxy, assign then token list `\c_dollar_str{} ~ dollar ~ \c_dollar_str{}` to `myvar` and then print it.

```

1 \str_new:N \g_myobj_str
2 \object_create_gset:NVnn \g_myobj_str \g_myproxy_str
3 { example }{ myobj }
4 \tl_gset:cn
5 {
6   \object_member_adr:Vn \g_myobj_str { myvar }
7 }
8 { \c_dollar_str{} ~ dollar ~ \c_dollar_str{} }
9 \object_member_use:Vn \g_myobj_str { myvar }

```

Output: \$ dollar \$

You can also avoid to specify an object identify and use `\object_gallocate_gincr` instead:

```

1 \int_new:N \g_intc_int
2 \object_gallocate_gincr:NNVnNN \g_myobj_str \g_intc_int \g_myproxy_str
3   { example } \c_object_local_str \c_object_public_str
4 \tl_gset:cn
5   {
6     \object_member_adr:Vn \g_myobj_str { myvar }
7   }
8   { \c_dollar_str{} ~ dollar ~ \c_dollar_str{} }
9 \object_member_use:Vn \g_myobj_str { myvar }

```

Output: \$ dollar \$

Example 2

In this example we create a proxy object with an embedded object inside.
Internal proxy

```

1 \proxy_create:nn { mymod }{ INT }
2 \proxy_push_member:nnn
3   {
4     \object_address:nn { mymod }{ INT }
5   }{ var }{ tl }

```

Container proxy

```

1 \proxy_create:nn { mymod }{ EXT }
2 \proxy_push_embedded:nnn
3   {
4     \object_address:nn { mymod }{ EXT }
5   }
6   { emb }
7   {
8     \object_address:nn { mymod }{ INT }
9   }

```

Now we create a new object from proxy EXT. It'll contain an embedded object created with INT proxy:

```

1 \str_new:N \g_EXTObj_str
2 \int_new:N \g_intcount_int
3 \object_gallocate_gincr:NNnnNN
4   \g_EXTObj_str \g_intcount_int
5   {
6     \object_address:nn { mymod }{ EXT }
7   }
8   { mymod }
9   \c_object_local_str \c_object_public_str

```

and use the embedded object in the following way:

```

1  \object_member_set:nnn
2  {
3    \object_embedded_adr:Vn \g_EXTObj_str { emb }
4  }{ var }{ Hi }
5  \object_member_use:nn
6  {
7    \object_embedded_adr:Vn \g_EXTObj_str { emb }
8  }{ var }

```

Output: Hi

8 Implementation

```

1  <*package>
2  <@@=rawobjects>
3  Deprecation message
4  \msg_new:nnn { rawobjects }{ deprecate }
5  {
6    Command ~ #1 ~ is ~ deprecated. ~ Use ~ instead ~ #2
7  }
8
9  \cs_new_protected:Nn \__rawobjects_launch_deprecate:NN
10 {
11   \msg_warning:nnnn{ rawobjects }{ deprecate }{ #1 }{ #2 }
12 }
13

```

`\rwoobj_address_f:n` It just performs a c expansion before passing it to `\cs_to_str:N`.

```

14
15 \cs_new:Nn \rwoobj_address_f:n
16 {
17   \exp_args:Nc \cs_to_str:N { #1 }
18 }
19

```

(End definition for `\rwoobj_address_f:n`. This function is documented on page 7.)

```

\c_object_local_str
\c_object_global_str
\c_object_public_str
\c_object_private_str
20 \str_const:Nn \c_object_local_str {l}
21 \str_const:Nn \c_object_global_str {g}
22 \str_const:Nn \c_object_public_str {_}
23 \str_const:Nn \c_object_private_str {__}
24
25
26 \cs_new:Nn \__rawobjects_scope:N
27 {
28   \str_use:N #1
29 }
30
31 \cs_new:Nn \__rawobjects_scope_pfx:N

```



```

32 {
33   \str_if_eq:NNF #1 \c_object_local_str
34   { g }
35 }
36
37 \cs_generate_variant:Nn \__rawobjects_scope_pfx:N { c }
38
39 \cs_new:Nn \__rawobjects_scope_pfx_cl:n
40 {
41   \__rawobjects_scope_pfx:c{
42     \object_ncmember_adr:nnn
43     {
44       \object_embedded_adr:nn { #1 } { /_I_/ }
45     }
46   { S } { str }
47 }
48 }
49
50 \cs_new:Nn \__rawobjects_vis_var:N
51 {
52   \str_use:N #1
53 }
54
55 \cs_new:Nn \__rawobjects_vis_fun:N
56 {
57   \str_if_eq:NNT #1 \c_object_private_str
58   {
59     --
60   }
61 }
62

```

(End definition for `\c_object_local_str` and others. These variables are documented on page 13.)

\object_address:nn Get address of an object

```

63 \cs_new:Nn \object_address:nn {
64   \tl_to_str:n { #1 _ #2 }
65 }

```

(End definition for `\object_address:nn`. This function is documented on page 7.)

\object_embedded_adr:nn Address of embedded object

```

66
67 \cs_new:Nn \object_embedded_adr:nn
68 {
69   #1 \tl_to_str:n{ _SUB_ #2 }
70 }
71
72 \cs_generate_variant:Nn \object_embedded_adr:nn{ Vn }
73

```

(End definition for `\object_embedded_adr:nn`. This function is documented on page 7.)

`\object_address_set:Nnn` Saves the address of an object into a string variable

`\object_address_gset:Nnn`

```
74
75 \cs_new_protected:Nn \object_address_set:Nnn {
76   \str_set:Nn #1 { #2 _ #3 }
77 }
78
79 \cs_new_protected:Nn \object_address_gset:Nnn {
80   \str_gset:Nn #1 { #2 _ #3 }
81 }
82
```

(End definition for \object_address_set:Nnn and \object_address_gset:Nnn. These functions are documented on page 7.)

`\object_if_exist_p:n` Tests if object exists.

`\object_if_exist:nTF`

```
83
84 \prg_new_conditional:Nnn \object_if_exist:n { p, T, F, TF }
85 {
86   \cs_if_exist:cTF
87   {
88     \object_ncmember_adr:nnn
89     {
90       \object_embedded_adr:nn{ #1 }{ /_I_/ }
91     }
92     { S }{ str }
93   }
94   {
95     \prg_return_true:
96   }
97   {
98     \prg_return_false:
99   }
100 }
101
102 \prg_generate_conditional_variant:Nnn \object_if_exist:n { V }
103 { p, T, F, TF }
104
```

(End definition for \object_if_exist:nTF. This function is documented on page 7.)

`\object_get_module:n` Retrieve the name, module and generating proxy of an object

`\object_get_proxy_adr:n`

```
105 \cs_new:Nn \object_get_module:n {
106   \object_ncmember_use:nnn
107   {
108     \object_embedded_adr:nn{ #1 }{ /_I_/ }
109   }
110   { M }{ str }
111 }
112 \cs_new:Nn \object_get_proxy_adr:n {
113   \object_ncmember_use:nnn
114   {
115     \object_embedded_adr:nn{ #1 }{ /_I_/ }
116   }
117   { P }{ str }
```

```

118 }
119
120 \cs_generate_variant:Nn \object_get_module:n { V }
121 \cs_generate_variant:Nn \object_get_proxy_adr:n { V }

```

(End definition for `\object_get_module:n` and `\object_get_proxy_adr:n`. These functions are documented on page 7.)

```

\object_if_local_p:n Test the specified parameters.
\object_if_local:nTF
\object_if_global_p:n
\object_if_global:nTF
\object_if_public_p:n
\object_if_public:nTF
\object_if_private_p:n
\object_if_private:nTF
122 \prg_new_conditional:Nnn \object_if_local:n {p, T, F, TF}
123 {
124   \str_if_eq:cNTF
125   {
126     \object_ncmember_adr:nnn
127     {
128       \object_embedded_adr:nn{ #1 }{ /_I_/ }
129     }
130     { S }{ str }
131   }
132   \c_object_local_str
133   {
134     \prg_return_true:
135   }
136   {
137     \prg_return_false:
138   }
139 }
140
141 \prg_new_conditional:Nnn \object_if_global:n {p, T, F, TF}
142 {
143   \str_if_eq:cNTF
144   {
145     \object_ncmember_adr:nnn
146     {
147       \object_embedded_adr:nn{ #1 }{ /_I_/ }
148     }
149     { S }{ str }
150   }
151   \c_object_global_str
152   {
153     \prg_return_true:
154   }
155   {
156     \prg_return_false:
157   }
158 }
159
160 \prg_new_conditional:Nnn \object_if_public:n {p, T, F, TF}
161 {
162   \str_if_eq:cNTF
163   {
164     \object_ncmember_adr:nnn
165     {
166       \object_embedded_adr:nn{ #1 }{ /_I_/ }

```

```

167     }
168     { V }{ str }
169 }
170 \c_object_public_str
171 {
172     \prg_return_true:
173 }
174 {
175     \prg_return_false:
176 }
177 }
178
179 \prg_new_conditional:Nnn \object_if_private:n {p, T, F, TF}
180 {
181     \str_if_eq:cNTF
182     {
183         \object_ncmember_adr:nnn
184         {
185             \object_embedded_adr:nn{ #1 }{ /_I_/ }
186         }
187         { V }{ str }
188     }
189     \c_object_private_str
190     {
191         \prg_return_true:
192     }
193     {
194         \prg_return_false:
195     }
196 }
197
198 \prg_generate_conditional_variant:Nnn \object_if_local:n { V }
199 { p, T, F, TF }
200 \prg_generate_conditional_variant:Nnn \object_if_global:n { V }
201 { p, T, F, TF }
202 \prg_generate_conditional_variant:Nnn \object_if_public:n { V }
203 { p, T, F, TF }
204 \prg_generate_conditional_variant:Nnn \object_if_private:n { V }
205 { p, T, F, TF }

```

(End definition for `\object_if_local:nTF` and others. These functions are documented on page 7.)

`\object_macro_adr:nn` Generic macro address

`\object_macro_use:nn`

```

206
207 \cs_new:Nn \object_macro_adr:nn
208 {
209     #1 \tl_to_str:n{ _MACRO_ #2 }
210 }
211
212 \cs_generate_variant:Nn \object_macro_adr:nn{ Vn }
213
214 \cs_new:Nn \object_macro_use:nn
215 {
216     \use:c

```

```

217     {
218         \object_macro_adr:nn{ #1 }{ #2 }
219     }
220 }
221
222 \cs_generate_variant:Nn \object_macro_use:nn{ Vn }
223

```

(End definition for \object_macro_adr:nn and \object_macro_use:nn. These functions are documented on page 11.)

_rawobjects_member_adr:nnnNN Macro address without object inference

```

224
225 \cs_new:Nn \_rawobjects_member_adr:nnnNN
226 {
227     \_rawobjects_scope:N #4
228     \_rawobjects_vis_var:N #5
229     #1 \tl_to_str:n { _ MEMBER _ #2 _ #3 }
230 }
231
232 \cs_generate_variant:Nn \_rawobjects_member_adr:nnnNN { VnnNN, nnncc }
233

```

(End definition for _rawobjects_member_adr:nnnNN.)

\object_member_adr:nnn Get the address of a member variable
\object_member_adr:nn

```

234
235 \cs_new:Nn \object_member_adr:nnn
236 {
237     \_rawobjects_member_adr:nnncc { #1 }{ #2 }{ #3 }
238     {
239         \object_ncmember_adr:nnn
240         {
241             \object_embedded_adr:nn{ #1 }{ /_I_/ }
242         }
243         { S }{ str }
244     }
245     {
246         \object_ncmember_adr:nnn
247         {
248             \object_embedded_adr:nn{ #1 }{ /_I_/ }
249         }
250         { V }{ str }
251     }
252 }
253
254 \cs_generate_variant:Nn \object_member_adr:nnn { Vnn, vnn, nnv }
255
256 \cs_new:Nn \object_member_adr:nn
257 {
258     \object_member_adr:nnv { #1 }{ #2 }
259     {
260         \object_rcmember_adr:nnn { #1 }
261         { #2 _ type }{ str }
262     }

```

```

263 }
264
265 \cs_generate_variant:Nn \object_member_adr:nn { Vn }
266

```

(End definition for `\object_member_adr:nnn` and `\object_member_adr:nn`. These functions are documented on page 8.)

The first argument is the new function name without argument. The second one is the function name you'll use, here #1 is the member type and #2 is equal to `g` if the object is global. The third one are the argument of the second function without the first N.

```

267
268 \cs_new_protected:Nn \__rawobjects_generator_mem:nnn
269 {
270   \cs_new:cn
271   {
272     rwobj-aux_ #1 : nn
273   }
274   {
275     \use:c{ #2 : c #3 }
276   }
277   \cs_new:cpn {#1 : nnn #3} ##1##2##3
278   {
279     \use:c
280     {
281       rwobj-aux_ #1 : nn
282     }
283     { ##3 }
284     {
285       \__rawobjects_scope_pfx_cl:n{ ##1 }
286     }
287     {
288       \object_member_adr:nnn{ ##1 }{ ##2 }{ ##3 }
289     }
290   }
291   \cs_generate_variant:cn { #1 : nnn #3 }{ Vnn #3, nnv #3 }
292
293   \cs_new:cpn { #1 : nn #3 } ##1##2
294   {
295     \use:c{ #1 : nnv #3 }
296     { ##1 }{ ##2 }
297     {
298       \object_rcmember_adr:nnn
299       { ##1 }{ ##2 _ type }{ str }
300     }
301   }
302
303   \cs_generate_variant:cn { #1 : nn #3 }{ Vn #3 }
304 }
305
306
307 \msg_new:nnn{ rawobjects }{ nonew }{ Unknown ~ function ~ #1 }
308
309 \cs_new_protected:Nn \__rawobjects_generator_mem_protected:nnn

```

```

310 {
311   \cs_new_protected:cn
312   {
313     rwojb-aux_ #1 : nn
314   }
315   {
316     \cs_if_exist_use:cF{ #2 : c #3 }
317     {
318       \msg_error:nnx{ rawobjects }{ nonew }{ #2 :c #3 }
319     }
320   }
321   \cs_new_protected:cpn {#1 : nnn #3} ##1##2##3
322   {
323     \use:c
324     {
325       rwojb-aux_ #1 : nn
326     }
327     { ##3 }
328     {
329       \__rawobjects_scope_pfx_cl:n{ ##1 }
330     }
331     {
332       \object_member_adr:nnn{ ##1 }{ ##2 }{ ##3 }
333     }
334   }
335   \cs_generate_variant:cn { #1 : nnn #3 }{ Vnn #3, nnv #3 }
336
337   \cs_new_protected:cpn { #1 : nn #3 } ##1##2
338   {
339     \use:c{ #1 : nnv #3 }
340     { ##1 }{ ##2 }
341     {
342       \object_rcmember_adr:nnn
343       { ##1 }{ ##2 _ type }{ str }
344     }
345   }
346
347   \cs_generate_variant:cn { #1 : nn #3 }{ Vn #3 }
348 }
349

```

\object_member_type:nn Deduce the member type from the generating proxy.

```

350
351 \cs_new:Nn \object_member_type:nn
352 {
353   \object_rcmember_use:nnn { #1 }
354   { #2 _ type }{ str }
355 }
356

```

(End definition for \object_member_type:nn. This function is documented on page 8.)

```

357
358 \msg_new:nnnn { rawobjects }{ noerr }{ Unspecified ~ scope }
359 {

```

```

360     Object ~ #1 ~ hasn't ~ a ~ scope ~ variable
361 }
362
363 \msg_new:nnnn { rawobjects }{ scoperr }{ Nonstandard ~ scope }
364 {
365     Operation ~ not ~ permitted ~ on ~ object ~ #1 ~
366     ~ since ~ it ~ wasn't ~ declared ~ local ~ or ~ global
367 }
368
369 \cs_new_protected:Nn \__rawobjects_force_scope:n
370 {
371     \cs_if_exist:cTF
372     {
373         \object_ncmember_adr:nnn
374         {
375             \object_embedded_adr:nn{ #1 }{ /_I_/ }
376         }
377         { S }{ str }
378     }
379     {
380         \bool_if:nF
381         {
382             \object_if_local_p:n { #1 } || \object_if_global_p:n { #1 }
383         }
384         {
385             \msg_error:nnx { rawobjects }{ scoperr }{ #1 }
386         }
387     }
388     {
389         \msg_error:nnx { rawobjects }{ noerr }{ #1 }
390     }
391 }
392

```

\object_member_if_exist_p:nnn

Tests if the specified member exists

\object_member_if_exist:nnnTF
\object_member_if_exist_p:nn
\object_member_if_exist:nnTF

```

393
394 \prg_new_conditional:Nnn \object_member_if_exist:nnn {p, T, F, TF }
395 {
396     \cs_if_exist:cTF
397     {
398         \object_member_adr:nnn { #1 }{ #2 }{ #3 }
399     }
400     {
401         \prg_return_true:
402     }
403     {
404         \prg_return_false:
405     }
406 }
407
408 \prg_new_conditional:Nnn \object_member_if_exist:nn {p, T, F, TF }
409 {
410     \cs_if_exist:cTF
411     {

```



```

412     \object_member_adr:nn { #1 }{ #2 }
413   }
414   {
415     \prg_return_true:
416   }
417   {
418     \prg_return_false:
419   }
420 }
421
422 \prg_generate_conditional_variant:Nnn \object_member_if_exist:nnn
423 { Vnn }{ p, T, F, TF }
424 \prg_generate_conditional_variant:Nnn \object_member_if_exist:nn
425 { Vn }{ p, T, F, TF }
426

```

(End definition for `\object_member_if_exist:nnnTF` and `\object_member_if_exist:nnTF`. These functions are documented on page 8.)

`\object_new_member:nnn` Creates a new member variable

```

427
428
429 \cs_new_protected:Nn \object_new_member:nnn
430 {
431   \cs_if_exist_use:cT { #3 _ new:c }
432   {
433     { \object_member_adr:nnn { #1 }{ #2 }{ #3 } }
434   }
435 }
436
437 \cs_generate_variant:Nn \object_new_member:nnn { Vnn, nnv }
438

```

(End definition for `\object_new_member:nnn`. This function is documented on page 8.)

`\object_member_use:nnn` Uses a member variable

```

\object_member_use:nn
439
440 \__rawobjects_generator_mem:nnn {object_member_use}{ #1_use }{ }
441
442 \cs_generate_variant:Nn \object_member_use:nnn {vnn}
443

```

(End definition for `\object_member_use:nnn` and `\object_member_use:nn`. These functions are documented on page 8.)

`\object_member_set:nnnn` Set the value a member.

```

\object_member_set:nnn
444
445 \__rawobjects_generator_mem:nnn {object_member_set}{ #1_#2 set }{n}
446

```

(End definition for `\object_member_set:nnnn` and `\object_member_set:nnn`. These functions are documented on page 9.)

`\object_member_set_eq:nnnN`
`\object_member_set_eq:nnN`

Make a member equal to another variable.

```

447
448 \__rawobjects_generator_mem_protected:nnn { object_member_set_eq }{ #1 _ #2 set_eq }{ N }
449
450 \cs_generate_variant:Nn \object_member_set_eq:nnnN { nnc, Vnc }
451
452 \cs_generate_variant:Nn \object_member_set_eq:nnN { nnc, Vnc }
453

```

(End definition for \object_member_set_eq:nnnN and \object_member_set_eq:nnN. These functions are documented on page 9.)

`\object_ncmember_adr:nnn` Get address of near constant

```

454
455 \cs_new:Nn \object_ncmember_adr:nnn
456 {
457   \tl_to_str:n{ c _ } #1 \tl_to_str:n { _ CONST _ #2 _ #3 }
458 }
459
460 \cs_generate_variant:Nn \object_ncmember_adr:nnn { Vnn, vnn }
461

```

(End definition for \object_ncmember_adr:nnn. This function is documented on page 9.)

`\object_rcmember_adr:nnn` Get the address of a remote constant.

```

462
463 \cs_new:Nn \object_rcmember_adr:nnn
464 {
465   \object_ncmember_adr:vnn
466   {
467     \object_ncmember_adr:nnn
468     {
469       \object_embedded_adr:nn{ #1 }{ /_I_/ }
470     }
471     { P }{ str }
472   }
473   { #2 }{ #3 }
474 }
475
476 \cs_generate_variant:Nn \object_rcmember_adr:nnn { Vnn }

```

(End definition for \object_rcmember_adr:nnn. This function is documented on page 9.)

The first argument is the new function name without argument. The second one is the function name you'll use, here #1 is the constant type. The third one are the argument of the second function without the first N.

```

477
478 \cs_new_protected:Nn \__rawobjects_generator_ncmem:nnn
479 {
480   \cs_new:cn
481   {
482     rwobj-aux_ #1 : n
483   }
484   {
485     \use:c{ #2 : c #3 }

```

```

486     }
487     \cs_new:cpn {#1 : nnn #3} ##1##2##3
488     {
489         \use:c
490         {
491             rwojb-aux_ #1 : n
492         }
493         { ##3 }
494         {
495             \object_ncmember_adr:nnn{ ##1 }{ ##2 }{ ##3 }
496         }
497     }
498     \cs_generate_variant:cn { #1 : nnn #3 }{ Vnn #3 }
499 }
500
501 \cs_new_protected:Nn \__rawobjects_generator_ncmem_protected:nnn
502 {
503     \cs_new_protected:cn
504     {
505         rwojb-aux_ #1 : n
506     }
507     {
508         \cs_if_exist_use:cF{ #2 : c #3 }
509         {
510             \msg_error:nnx{ rawobjects }{ none }{ #2 :c #3 }
511         }
512     }
513     \cs_new_protected:cpn {#1 : nnn #3} ##1##2##3
514     {
515         \use:c
516         {
517             rwojb-aux_ #1 : n
518         }
519         { ##3 }
520         {
521             \object_ncmember_adr:nnn{ ##1 }{ ##2 }{ ##3 }
522         }
523     }
524     \cs_generate_variant:cn { #1 : nnn #3 }{ Vnn #3 }
525 }
526

```

\object_ncmember_if_exist_p:nnn
\object_ncmember_if_exist:nnnTF
\object_rcmember_if_exist_p:nnn
\object_rcmember_if_exist:nnnTF

Tests if the specified member constant exists.

```

527
528 \prg_new_conditional:Nnn \object_ncmember_if_exist:nnn {p, T, F, TF }
529 {
530     \cs_if_exist:cTF
531     {
532         \object_ncmember_adr:nnn { #1 }{ #2 }{ #3 }
533     }
534     {
535         \prg_return_true:
536     }
537     {

```

```

538         \prg_return_false:
539     }
540 }
541
542 \prg_new_conditional:Nnn \object_rcmember_if_exist:nnn {p, T, F, TF }
543 {
544     \cs_if_exist:cTF
545     {
546         \object_rcmember_adr:nnn { #1 }{ #2 }{ #3 }
547     }
548     {
549         \prg_return_true:
550     }
551     {
552         \prg_return_false:
553     }
554 }
555
556 \prg_generate_conditional_variant:Nnn \object_ncmember_if_exist:nnn
557 { Vnn }{ p, T, F, TF }
558 \prg_generate_conditional_variant:Nnn \object_rcmember_if_exist:nnn
559 { Vnn }{ p, T, F, TF }
560

```

(End definition for `\object_ncmember_if_exist:nnnTF` and `\object_rcmember_if_exist:nnnTF`. These functions are documented on page 9.)

`\object_ncmember_use:nnn` Uses a near/remote constant.

```

\object_rcmember_use:nnn
561
562 \__rawobjects_generator_ncmem:nnn{ object_ncmember_use }{ #1_use}{ }
563
564 \cs_new:Nn \object_rcmember_use:nnn
565 {
566     \cs_if_exist_use:cT { #3 _ use:c }
567     {
568         { \object_rcmember_adr:nnn { #1 }{ #2 }{ #3 } }
569     }
570 }
571
572 \cs_generate_variant:Nn \object_rcmember_use:nnn { Vnn }
573

```

(End definition for `\object_ncmember_use:nnn` and `\object_rcmember_use:nnn`. These functions are documented on page 9.)

`\object_newconst:nnnn` Creates a constant variable, use with caution

```

574
575 \__rawobjects_generator_ncmem_protected:nnn { object_newconst }{ #1 _ const }{n}
576

```

(End definition for `\object_newconst:nnnn`. This function is documented on page 11.)

`\object_newconst_tl:nnn` Create constants

```

\object_newconst_str:nnn
\object_newconst_int:nnn
\object_newconst_clist:nnn
\object_newconst_dim:nnn
\object_newconst_skip:nnn
\object_newconst_fp:nnn
577
578 \cs_new_protected:Nn \object_newconst_tl:nnn

```

```

579 {
580   \object_newconst:nnnn { #1 }{ #2 }{ t1 }{ #3 }
581 }
582 \cs_new_protected:Nn \object_newconst_str:nnn
583 {
584   \object_newconst:nnnn { #1 }{ #2 }{ str }{ #3 }
585 }
586 \cs_new_protected:Nn \object_newconst_int:nnn
587 {
588   \object_newconst:nnnn { #1 }{ #2 }{ int }{ #3 }
589 }
590 \cs_new_protected:Nn \object_newconst_clist:nnn
591 {
592   \object_newconst:nnnn { #1 }{ #2 }{ clist }{ #3 }
593 }
594 \cs_new_protected:Nn \object_newconst_dim:nnn
595 {
596   \object_newconst:nnnn { #1 }{ #2 }{ dim }{ #3 }
597 }
598 \cs_new_protected:Nn \object_newconst_skip:nnn
599 {
600   \object_newconst:nnnn { #1 }{ #2 }{ skip }{ #3 }
601 }
602 \cs_new_protected:Nn \object_newconst_fp:nnn
603 {
604   \object_newconst:nnnn { #1 }{ #2 }{ fp }{ #3 }
605 }
606
607 \cs_generate_variant:Nn \object_newconst_tl:nnn { Vnn }
608 \cs_generate_variant:Nn \object_newconst_str:nnn { Vnn }
609 \cs_generate_variant:Nn \object_newconst_int:nnn { Vnn }
610 \cs_generate_variant:Nn \object_newconst_clist:nnn { Vnn }
611 \cs_generate_variant:Nn \object_newconst_dim:nnn { Vnn }
612 \cs_generate_variant:Nn \object_newconst_skip:nnn { Vnn }
613 \cs_generate_variant:Nn \object_newconst_fp:nnn { Vnn }
614
615
616 \cs_generate_variant:Nn \object_newconst_str:nnn { nnx }
617 \cs_generate_variant:Nn \object_newconst_str:nnn { nnV }
618

```

(End definition for `\object_newconst_tl:nnn` and others. These functions are documented on page 11.)

`\object_newconst_seq_from_clist:nnn` Creates a seq constant.

```

619
620 \cs_new_protected:Nn \object_newconst_seq_from_clist:nnn
621 {
622   \seq_const_from_clist:cn
623   {
624     \object_ncmember_adr:nnn { #1 }{ #2 }{ seq }
625   }
626   { #3 }
627 }
628

```

```

629 \cs_generate_variant:Nn \object_newconst_seq_from_clist:nnn { Vnn }
630

```

(End definition for `\object_newconst_seq_from_clist:nnn`. This function is documented on page 11.)

`\object_newconst_prop_from_keyval:nnn` Creates a prop constant.

```

631
632 \cs_new_protected:Nn \object_newconst_prop_from_keyval:nnn
633 {
634   \prop_const_from_keyval:cn
635   {
636     \object_ncmember_adr:nnn { #1 }{ #2 }{ prop }
637   }
638   { #3 }
639 }
640
641 \cs_generate_variant:Nn \object_newconst_prop_from_keyval:nnn { Vnn }
642

```

(End definition for `\object_newconst_prop_from_keyval:nnn`. This function is documented on page 11.)

`\object_ncmethod_adr:nnn` Fully expands to the method address.

`\object_rcmethod_adr:nnn`

```

643
644 \cs_new:Nn \object_ncmethod_adr:nnn
645 {
646   #1 \tl_to_str:n { _ CMETHOD _ #2 : #3 }
647 }
648
649 \cs_generate_variant:Nn \object_ncmethod_adr:nnn { Vnn , vnn }
650
651 \cs_new:Nn \object_rcmethod_adr:nnn
652 {
653   \object_ncmethod_adr:vnn
654   {
655     \object_ncmember_adr:nnn
656     {
657       \object_embedded_adr:nn{ #1 }{ /_I_/ }
658     }
659     { P }{ str }
660   }
661   { #2 }{ #3 }
662 }
663
664 \cs_generate_variant:Nn \object_ncmethod_adr:nnn { Vnn , vnn }
665 \cs_generate_variant:Nn \object_rcmethod_adr:nnn { Vnn }
666

```

(End definition for `\object_ncmethod_adr:nnn` and `\object_rcmethod_adr:nnn`. These functions are documented on page 10.)

`\object_ncmethod_if_exist_p:nnn` Tests if the specified member constant exists.

`\object_ncmethod_if_exist:nnn` *TF*

`\object_rcmethod_if_exist_p:nnn`

`\object_rcmethod_if_exist:nnn` *TF*

```

667
668 \prg_new_conditional:Nnn \object_ncmethod_if_exist:nnn {p, T, F, TF }
669 {
670   \cs_if_exist:cTF

```

```

671     {
672     \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
673     }
674     {
675     \prg_return_true:
676     }
677     {
678     \prg_return_false:
679     }
680 }
681
682 \prg_new_conditional:Nnn \object_rcmethod_if_exist:nnn {p, T, F, TF }
683 {
684   \cs_if_exist:cTF
685   {
686     \object_rcmethodr_adr:nnn { #1 }{ #2 }{ #3 }
687   }
688   {
689     \prg_return_true:
690   }
691   {
692     \prg_return_false:
693   }
694 }
695
696 \prg_generate_conditional_variant:Nnn \object_ncmethod_if_exist:nnn
697 { Vnn }{ p, T, F, TF }
698 \prg_generate_conditional_variant:Nnn \object_rcmethod_if_exist:nnn
699 { Vnn }{ p, T, F, TF }
700

```

(End definition for `\object_ncmethod_if_exist:nnnTF` and `\object_rcmethod_if_exist:nnnTF`. These functions are documented on page 10.)

`\object_new_cmethod:nnnn` Creates a new method

```

701
702 \cs_new_protected:Nn \object_new_cmethod:nnnn
703 {
704   \cs_new:cn
705   {
706     \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
707   }
708   { #4 }
709 }
710
711 \cs_generate_variant:Nn \object_new_cmethod:nnnn { Vnnn }
712

```

(End definition for `\object_new_cmethod:nnnn`. This function is documented on page 10.)

`\object_ncmethod_call:nnn` Calls the specified method.

`\object_rcmethod_call:nnn`

```

713
714 \cs_new:Nn \object_ncmethod_call:nnn
715 {
716   \use:c

```

```

717 {
718   \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
719 }
720 }
721
722 \cs_new:Nn \object_rcmethod_call:nnn
723 {
724   \use:c
725   {
726     \object_rcmethod_adr:nnn { #1 }{ #2 }{ #3 }
727   }
728 }
729
730 \cs_generate_variant:Nn \object_ncmethod_call:nnn { Vnn }
731 \cs_generate_variant:Nn \object_rcmethod_call:nnn { Vnn }
732

```

(End definition for `\object_ncmethod_call:nnn` and `\object_rcmethod_call:nnn`. These functions are documented on page 10.)

```

733
734 \cs_new_protected:Nn \__rawobjects_initproxy:nnn
735 {
736   \object_newconst:nnnn
737   {
738     \object_embedded_adr:nn{ #3 }{ /_I_/ }
739   }
740   { ifprox }{ bool }{ \c_true_bool }
741 }
742 \cs_generate_variant:Nn \__rawobjects_initproxy:nnn { VnV }
743

```

`\object_if_proxy_p:n`
`\object_if_proxy:nTF`

Test if an object is a proxy.

```

744
745 \cs_new:Nn \__rawobjects_bol_com:N
746 {
747   \cs_if_exist_p:N #1 && \bool_if_p:N #1
748 }
749
750 \cs_generate_variant:Nn \__rawobjects_bol_com:N { c }
751
752 \prg_new_conditional:Nnn \object_if_proxy:n {p, T, F, TF}
753 {
754   \cs_if_exist:cTF
755   {
756     \object_ncmember_adr:nnn
757     {
758       \object_embedded_adr:nn{ #1 }{ /_I_/ }
759     }
760     { ifprox }{ bool }
761   }
762   {
763     \bool_if:cTF
764     {
765       \object_ncmember_adr:nnn

```



```

766         {
767             \object_embedded_adr:nn{ #1 }{ /_I_/ }
768         }
769         { ifprox }{ bool }
770     }
771     {
772         \prg_return_true:
773     }
774     {
775         \prg_return_false:
776     }
777 }
778 {
779     \prg_return_false:
780 }
781 }
782

```

(End definition for \object_if_proxy:nTF. This function is documented on page 12.)

```

\object_test_proxy_p:nn Test if an object is generated from selected proxy.
\object_test_proxy:nnTF
\object_test_proxy_p:nN
\object_test_proxy:nNTF
783
784 \prg_generate_conditional_variant:Nnn \str_if_eq:nn { ve }{ TF }
785
786 \prg_new_conditional:Nnn \object_test_proxy:nn {p, T, F, TF}
787 {
788     \str_if_eq:veTF
789     {
790         \object_ncmember_adr:nnn
791         {
792             \object_embedded_adr:nn{ #1 }{ /_I_/ }
793         }
794         { P }{ str }
795     }
796     { #2 }
797     {
798         \prg_return_true:
799     }
800     {
801         \prg_return_false:
802     }
803 }
804
805 \prg_new_conditional:Nnn \object_test_proxy:nN {p, T, F, TF}
806 {
807     \str_if_eq:cNTF
808     {
809         \object_ncmember_adr:nnn
810         {
811             \object_embedded_adr:nn{ #1 }{ /_I_/ }
812         }
813         { P }{ str }
814     }
815     #2

```

```

816     {
817         \prg_return_true:
818     }
819     {
820         \prg_return_false:
821     }
822 }
823
824 \prg_generate_conditional_variant:Nnn \object_test_proxy:nn
825 { Vn }{p, T, F, TF}
826 \prg_generate_conditional_variant:Nnn \object_test_proxy:nN
827 { VN }{p, T, F, TF}
828

```

(End definition for `\object_test_proxy:nnTF` and `\object_test_proxy:nNTF`. These functions are documented on page 12.)

```

\object_create:nnnNN
\object_create_set:NnnnNN
\object_create_gset:NnnnNN
\object_create:nnnN
\object_create_set:NnnnN
\object_create_gset:NnnnN
\object_create:nnn
\object_create_set:Nnnn
\object_create_gset:Nnnn
\embedded_create:nnn

Creates an object from a proxy.

829
830 \msg_new:nnnn { rawobjects }{ notproxy }{ Fake ~ proxy }
831 {
832     Object ~ #1 ~ is ~ not ~ a ~ proxy.
833 }
834
835 \cs_new_protected:Nn \__rawobjects_force_proxy:n
836 {
837     \object_if_proxy:nF { #1 }
838     {
839         \msg_error:nnn { rawobjects }{ notproxy }{ #1 }
840     }
841 }
842
843 \cs_new_protected:Nn \__rawobjects_create_anon:nnnNN
844 {
845     \tl_if_empty:nF{ #1 }
846     {
847
848         \__rawobjects_force_proxy:n { #1 }
849
850
851         \object_newconst_str:nnn
852         {
853             \object_embedded_adr:nn{ #3 }{ /_I_/ }
854         }
855         { M }{ #2 }
856         \object_newconst_str:nnn
857         {
858             \object_embedded_adr:nn{ #3 }{ /_I_/ }
859         }
860         { P }{ #1 }
861         \object_newconst_str:nnV
862         {
863             \object_embedded_adr:nn{ #3 }{ /_I_/ }
864         }

```

```

865     { S } #4
866 \object_newconst_str:nnV
867 {
868     \object_embedded_adr:nn{ #3 }{ /_I_/ }
869 }
870 { V } #5
871
872 \seq_map_inline:cn
873 {
874     \object_member_adr:nnn { #1 }{ varlist }{ seq }
875 }
876 {
877     \object_new_member:nnv { #3 }{ ##1 }
878     {
879         \object_ncmember_adr:nnn { #1 }{ ##1 _ type }{ str }
880     }
881 }
882
883 \seq_map_inline:cn
884 {
885     \object_member_adr:nnn { #1 }{ objlist }{ seq }
886 }
887 {
888     \embedded_create:nvn
889     { #3 }
890     {
891         \object_ncmember_adr:nnn { #1 }{ ##1 _ proxy }{ str }
892     }
893     { ##1 }
894 }
895
896 \tl_map_inline:cn
897 {
898     \object_member_adr:nnn { #1 }{ init }{ tl }
899 }
900 {
901     ##1 { #1 }{ #2 }{ #3 }
902 }
903
904 }
905 }
906
907 \cs_generate_variant:Nn \__rawobjects_create_anon:nnnNN { xnxNN, xvxcc }
908
909 \cs_new_protected:Nn \object_create:nnnNN
910 {
911     \__rawobjects_create_anon:xnxNN { #1 }{ #2 }
912     { \object_address:nn { #2 }{ #3 } }
913     #4 #5
914 }
915
916 \cs_generate_variant:Nn \object_create:nnnNN { VnnNN }
917
918 \cs_new_protected:Nn \object_create_set:NnnnNN

```

```

919 {
920     \object_create:nnnNN { #2 }{ #3 }{ #4 } #5 #6
921     \str_set:Nx #1 { \object_address:nn { #3 }{ #4 } }
922 }
923
924 \cs_new_protected:Nn \object_create_gset:NnnnNN
925 {
926     \object_create:nnnNN { #2 }{ #3 }{ #4 } #5 #6
927     \str_gset:Nx #1 { \object_address:nn { #3 }{ #4 } }
928 }
929
930 \cs_generate_variant:Nn \object_create_set:NnnnNN { NVnnNN, NnnfNN }
931 \cs_generate_variant:Nn \object_create_gset:NnnnNN { NVnnNN, NnnfNN }
932
933
934
935 \cs_new_protected:Nn \object_create:nnnN
936 {
937     \object_create:nnnNN { #1 }{ #2 }{ #3 } #4 \c_object_public_str
938 }
939
940 \cs_generate_variant:Nn \object_create:nnnN { VnnN }
941
942 \cs_new_protected:Nn \object_create_set:NnnnN
943 {
944     \object_create_set:NnnnNN #1 { #2 }{ #3 }{ #4 } #5 \c_object_public_str
945 }
946
947 \cs_new_protected:Nn \object_create_gset:NnnnN
948 {
949     \object_create_gset:NnnnNN #1 { #2 }{ #3 }{ #4 } #5 \c_object_public_str
950 }
951
952 \cs_generate_variant:Nn \object_create_set:NnnnN { NVnnN }
953 \cs_generate_variant:Nn \object_create_gset:NnnnN { NVnnN }
954
955 \cs_new_protected:Nn \object_create:nnn
956 {
957     \object_create:nnnNN { #1 }{ #2 }{ #3 }
958     \c_object_global_str \c_object_public_str
959 }
960
961 \cs_generate_variant:Nn \object_create:nnn { Vnn }
962
963 \cs_new_protected:Nn \object_create_set:Nnnn
964 {
965     \object_create_set:NnnnNN #1 { #2 }{ #3 }{ #4 }
966     \c_object_global_str \c_object_public_str
967 }
968
969 \cs_new_protected:Nn \object_create_gset:Nnnn
970 {
971     \object_create_gset:NnnnNN #1 { #2 }{ #3 }{ #4 }
972     \c_object_global_str \c_object_public_str

```

```

973     }
974
975     \cs_generate_variant:Nn \object_create_set:Nnnn { NVnn }
976     \cs_generate_variant:Nn \object_create_gset:Nnnn { NVnn }
977
978
979
980
981     \cs_new_protected:Nn \embedded_create:nnn
982     {
983         \__rawobjects_create_anon:xvxc { #2 }
984         {
985             \object_ncmember_adr:nnn
986             {
987                 \object_embedded_adr:nn{ #1 }{ /_I_/ }
988             }
989             { M }{ str }
990         }
991         {
992             \object_embedded_adr:nn
993             { #1 }{ #3 }
994         }
995         {
996             \object_ncmember_adr:nnn
997             {
998                 \object_embedded_adr:nn{ #1 }{ /_I_/ }
999             }
1000             { S }{ str }
1001         }
1002         {
1003             \object_ncmember_adr:nnn
1004             {
1005                 \object_embedded_adr:nn{ #1 }{ /_I_/ }
1006             }
1007             { V }{ str }
1008         }
1009     }
1010
1011     \cs_generate_variant:Nn \embedded_create:nnn { nvnn, Vnn }
1012

```

(End definition for \object_create:nnnNN and others. These functions are documented on page 12.)

\proxy_create:nn
 \proxy_create_set:Nnn
 \proxy_create_gset:Nnn

Creates a new proxy object

```

1013
1014     \cs_new_protected:Nn \proxy_create:nn
1015     {
1016         \object_create:VnnNN \c_proxy_address_str { #1 }{ #2 }
1017         \c_object_global_str \c_object_public_str
1018     }
1019
1020     \cs_new_protected:Nn \proxy_create_set:Nnn
1021     {
1022         \object_create_set:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 }

```

```

1023     \c_object_global_str \c_object_public_str
1024 }
1025
1026 \cs_new_protected:Nn \proxy_create_gset:Nnn
1027 {
1028     \object_create_gset:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 }
1029     \c_object_global_str \c_object_public_str
1030 }
1031
1032
1033
1034 \cs_new_protected:Nn \proxy_create:nnN
1035 {
1036     \__rawobjects_launch_deprecate:NN \proxy_create:nnN \proxy_create:nn
1037     \object_create:VnnNN \c_proxy_address_str { #1 }{ #2 }
1038     \c_object_global_str #3
1039 }
1040
1041 \cs_new_protected:Nn \proxy_create_set:NnnN
1042 {
1043     \__rawobjects_launch_deprecate:NN \proxy_create_set:NnnN \proxy_create_set:Nnn
1044     \object_create_set:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 }
1045     \c_object_global_str #4
1046 }
1047
1048 \cs_new_protected:Nn \proxy_create_gset:NnnN
1049 {
1050     \__rawobjects_launch_deprecate:NN \proxy_create_gset:NnnN \proxy_create_gset:Nnn
1051     \object_create_gset:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 }
1052     \c_object_global_str #4
1053 }
1054

```

(End definition for `\proxy_create:nn`, `\proxy_create_set:Nnn`, and `\proxy_create_gset:Nnn`. These functions are documented on page 13.)

`\proxy_push_member:nnn` Push a new member inside a proxy.

```

1055
1056 \cs_new_protected:Nn \proxy_push_member:nnn
1057 {
1058     \object_newconst_str:nnn { #1 }{ #2 _ type }{ #3 }
1059     \seq_gput_left:cn
1060     {
1061         \object_member_adr:nnn { #1 }{ varlist }{ seq }
1062     }
1063     { #2 }
1064 }
1065
1066 \cs_generate_variant:Nn \proxy_push_member:nnn { Vnn }
1067

```

(End definition for `\proxy_push_member:nnn`. This function is documented on page 13.)

`\proxy_push_embedded:nnn` Push a new embedded object inside a proxy.

```

1068

```

```

1069 \cs_new_protected:Nn \proxy_push_embedded:nnn
1070 {
1071   \object_newconst_str:nnx { #1 }{ #2 _ proxy }{ #3 }
1072   \seq_gput_left:cn
1073   {
1074     \object_member_adr:nnn { #1 }{ objlist }{ seq }
1075   }
1076   { #2 }
1077 }
1078
1079 \cs_generate_variant:Nn \proxy_push_embedded:nnn { Vnn }
1080

```

(End definition for `\proxy_push_embedded:nnn`. This function is documented on page 14.)

`\proxy_add_initializer:nN` Push a new embedded object inside a proxy.

```

1081
1082 \cs_new_protected:Nn \proxy_add_initializer:nN
1083 {
1084   \tl_gput_right:cn
1085   {
1086     \object_member_adr:nnn { #1 }{ init }{ tl }
1087   }
1088   { #2 }
1089 }
1090
1091 \cs_generate_variant:Nn \proxy_add_initializer:nN { VN }
1092

```

(End definition for `\proxy_add_initializer:nN`. This function is documented on page 14.)

`\c_proxy_address_str` Variable containing the address of the proxy object.

```

1093
1094 \str_const:Nx \c_proxy_address_str
1095 { \object_address:nn { rawobjects }{ proxy } }
1096
1097 \object_newconst_str:nnn
1098 {
1099   \object_embedded_adr:Vn \c_proxy_address_str { /_I/ }
1100 }
1101 { M }{ rawobjects }
1102
1103 \object_newconst_str:nnV
1104 {
1105   \object_embedded_adr:Vn \c_proxy_address_str { /_I/ }
1106 }
1107 { P } \c_proxy_address_str
1108
1109 \object_newconst_str:nnV
1110 {
1111   \object_embedded_adr:Vn \c_proxy_address_str { /_I/ }
1112 }
1113 { S } \c_object_global_str
1114
1115 \object_newconst_str:nnV

```

```

1116 {
1117     \object_embedded_adr:Vn \c_proxy_address_str { /_I_/ }
1118 }
1119 { V } \c_object_public_str
1120
1121
1122 \__rawobjects_initproxy:VnV \c_proxy_address_str { rawobjects } \c_proxy_address_str
1123
1124 \object_new_member:Vnn \c_proxy_address_str { init }{ t1 }
1125
1126 \object_new_member:Vnn \c_proxy_address_str { varlist }{ seq }
1127
1128 \object_new_member:Vnn \c_proxy_address_str { objlist }{ seq }
1129
1130 \proxy_push_member:Vnn \c_proxy_address_str
1131     { init }{ t1 }
1132 \proxy_push_member:Vnn \c_proxy_address_str
1133     { varlist }{ seq }
1134 \proxy_push_member:Vnn \c_proxy_address_str
1135     { objlist }{ seq }
1136
1137 \proxy_add_initializer:VN \c_proxy_address_str
1138     \__rawobjects_initproxy:nnn
1139

```

(End definition for \c_proxy_address_str. This variable is documented on page 12.)

```

\object_allocate_incr:NNnnNN Create an address and use it to instantiate an object
    \object_gallocate_incr:NNnnNN
    \object_allocate_gincr:NNnnNN
    \object_gallocate_gincr:NNnnNN
1140
1141 \cs_new:Nn \__rawobjects_combine_aux:nnn
1142 {
1143     anon . #3 . #2 . #1
1144 }
1145
1146 \cs_generate_variant:Nn \__rawobjects_combine_aux:nnn { Vnf }
1147
1148 \cs_new:Nn \__rawobjects_combine:Nn
1149 {
1150     \__rawobjects_combine_aux:Vnf #1 { #2 }
1151     {
1152         \cs_to_str:N #1
1153     }
1154 }
1155
1156 \cs_new_protected:Nn \object_allocate_incr:NNnnNN
1157 {
1158     \object_create_set:NnnfNN #1 { #3 }{ #4 }
1159     {
1160         \__rawobjects_combine:Nn #2 { #3 }
1161     }
1162     #5 #6
1163
1164     \int_incr:N #2
1165 }

```



```

1166
1167 \cs_new_protected:Nn \object_gallocate_incr:NNnnNN
1168 {
1169   \object_create_gset:NnnfNN #1 { #3 }{ #4 }
1170   {
1171     \__rawobjects_combine:Nn #2 { #3 }
1172   }
1173   #5 #6
1174
1175   \int_incr:N #2
1176 }
1177
1178 \cs_generate_variant:Nn \object_allocate_incr:NNnnNN { NNVnNN }
1179
1180 \cs_generate_variant:Nn \object_gallocate_incr:NNnnNN { NNVnNN }
1181
1182 \cs_new_protected:Nn \object_allocate_gincr:NNnnNN
1183 {
1184   \object_create_set:NnnfNN #1 { #3 }{ #4 }
1185   {
1186     \__rawobjects_combine:Nn #2 { #3 }
1187   }
1188   #5 #6
1189
1190   \int_gincr:N #2
1191 }
1192
1193 \cs_new_protected:Nn \object_gallocate_gincr:NNnnNN
1194 {
1195   \object_create_gset:NnnfNN #1 { #3 }{ #4 }
1196   {
1197     \__rawobjects_combine:Nn #2 { #3 }
1198   }
1199   #5 #6
1200
1201   \int_gincr:N #2
1202 }
1203
1204 \cs_generate_variant:Nn \object_allocate_gincr:NNnnNN { NNVnNN }
1205
1206 \cs_generate_variant:Nn \object_gallocate_gincr:NNnnNN { NNVnNN }
1207

```

(End definition for `\object_allocate_incr:NNnnNN` and others. These functions are documented on page 13.)

\object_assign:nn Copy an object to another one.

```

1208 \cs_new_protected:Nn \object_assign:nn
1209 {
1210   \seq_map_inline:cn
1211   {
1212     \object_member_adr:vnn
1213     {
1214       \object_ncmember_adr:nnn

```

```

1215         {
1216         \object_embedded_adr:nn{ #1 }{ /_I_/ }
1217         }
1218         { P }{ str }
1219     }
1220     { varlist }{ seq }
1221 }
1222 {
1223 \object_member_set_eq:nnc { #1 }{ ##1 }
1224 {
1225     \object_member_adr:nn{ #2 }{ ##1 }
1226 }
1227 }
1228 }
1229
1230 \cs_generate_variant:Nn \object_assign:nn { nV, Vn, VV }

```

(End definition for \object_assign:nn. This function is documented on page 14.)

```

1231 </package>

```