

The lt3rawobjects package

Paolo De Donato

Released on 2022/12/27 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	Put objects inside objects	4
	5.1 Put a pointer variable	4
	5.2 Clone the inner structure	5
	5.3 Embedded objects	6
6	Library functions	6
	6.1 Common functions	6
	6.2 Base object functions	7
	6.3 Members	8
	6.4 Methods	9
	6.5 Constant member creation	10
	6.6 Macros	11
	6.7 Proxy utilities and object creation	11
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 Instead changes introduced in minor and patch version updates are always backward compatible, and any withdrawn function is declared deprecated instead of being removed.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.

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

4.2 Methods

Methods are L^AT_EX3 functions that can't be changed once they're created. Like constant, methods could be near or remote.

4.3 Members

Members are just mutable L^AT_EX3 variables. Members can be manually created inside objects or can be automatically created during object creation if their definitions are stored inside the generating proxy through the `\proxy_push_member` function. These members automatically created by proxies are called *tracked* since the generator knows about them. Moreover, you don't need to specify the type of a tracked member since it can be inferred from the generator and the `assign` functions automatically copy all the tracked members in the new object.

If the object is local/global then all its members are automatically local/global.

5 Put objects inside objects

Sometimes it's necessary to include other objects inside an object, and since objects are structured data types you can't put them directly inside a variable. However `lt3rawobjects` provides some workarounds that allows you to include objects inside other objects, each with its own advantages and disadvantages.

In the following examples we're in module `mymod` and we want to put inside object `A` another object created with proxy `prx`.

5.1 Put a pointer variable

A simple solution is creating that object outside `A` with `\object_create`

```
\object_create:nnnNN
{ \object_address:nn{ mymod }{ prx } }{ mymod }{ B } ....
```

and then creating a pointer variable inside `A` (usually of type `tl` or `str`) holding the newly created address:

```
\object_new_member:nnn
{
  \object_address:nn{ mymod }{ A }
  }{ pointer }{ tl }

\tl_(g)set:cn
```

```

{
  \object_new_member:nnn
  {
    \object_address:nn{ mymod }{ A }
    }{ pointer }{ t1 }
  }
  {
    \object_address:nn{ mymod }{ B }
  }
}

```

you can access the pointed object by calling `\object_member_use` on `pointer` member.

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;
- creating objects also creates additional hidden variables, taking so (little) additional space.

5.2 Clone the inner structure

Instead of referring a complete object you can just clone the inner structure of `prx` and put inside `A`. For example if `prx` declares member `x` of type `str` and member `y` of type `int` then you can do

```

\object_new_member:nnn
{
  \object_address:nn{ mymod }{ A }
  }{ prx-x }{ str }
\object_new_member:nnn
{
  \object_address:nn{ mymod }{ A }
  }{ prx-y }{ int }

```

and then use `prx-x`, `prx-y` as normal members of `A`.

Advantages

- Simple and no additional function needed to create and manage included objects;
- you can put these specifications inside a proxy so that every object created with it will have the required members/methods;
- no hidden variable created, lowest overhead among the proposed solutions.

Disadvantages

- Cloning the inner structure doesn't create any object, so you don't have any object address nor you can share the included "object" unless you share the container object too.

5.3 Embedded objects

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

```
\embedded_create:nnn
{
  \object_address:nn{ mymod }{ A }
}{ prx }{ B }
```

and addresses of embedded 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

```
\rwobj_address_f:n ★ \rwobj_address_f:n {<address>}
```

Fully expand an address in an f-type argument.

From: 2.3

6.2 Base object functions

<hr/> <u>\object_address:nn</u> ☆	\object_address:nn {<module>} {<id>}
	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
<hr/> <u>\object_address_set:Nnn</u> <u>\object_address_gset:Nnn</u>	\object_address_set:nn <str var> {<module>} {<id>}
	Stores the address of selected object inside the string variable <str var>. From: 1.1
<hr/> <u>\object_embedded_adr:nn</u> ☆ <u>\object_embedded_adr:Vn</u> ☆	\object_embedded_adr:nn {<address>} {<id>}
	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
<hr/> <u>\object_if_exist_p:n</u> ★ <u>\object_if_exist_p:V</u> ★ <u>\object_if_exist:nTF</u> ★ <u>\object_if_exist:VTF</u> ★	\object_if_exist_p:n {<address>} \object_if_exist:nTF {<address>} {<true code>} {<false code>}
	Tests if an object was instantiated at the specified address. From: 1.0
<hr/> <u>\object_get_module:n</u> ★ <u>\object_get_module:V</u> ★ <u>\object_get_proxy_adr:n</u> ★ <u>\object_get_proxy_adr:V</u> ★	\object_get_module:n {<address>} \object_get_proxy_adr:n {<address>}
	Get the object module and its generator. From: 1.0
<hr/> <u>\object_if_local_p:n</u> ★ <u>\object_if_local_p:V</u> ★ <u>\object_if_local:nTF</u> ★ <u>\object_if_local:VTF</u> ★ <u>\object_if_global_p:n</u> ★ <u>\object_if_global_p:V</u> ★ <u>\object_if_global:nTF</u> ★ <u>\object_if_global:VTF</u> ★	\object_if_local_p:n {<address>} \object_if_local:nTF {<address>} {<true code>} {<false code>}
	Tests if the object is local or global. From: 1.0
<hr/> <u>\object_if_public_p:n</u> ★ <u>\object_if_public_p:V</u> ★ <u>\object_if_public:nTF</u> ★ <u>\object_if_public:VTF</u> ★ <u>\object_if_private_p:n</u> ★ <u>\object_if_private_p:V</u> ★ <u>\object_if_private:nTF</u> ★ <u>\object_if_private:VTF</u> ★	\object_if_public_p:n {<address>} \object_if_public:nTF {<address>} {<true code>} {<false code>}
	Tests if the object is public or private. From: 1.0

6.3 Members

<code>\object_member_adr:nnn</code>	☆	<code>\object_member_adr:nnn {<address>} {<member name>} {<member type>}</code>
<code>\object_member_adr:(Vnn nnv)</code>	☆	<code>\object_member_adr:nn {<address>} {<member name>}</code>
<code>\object_member_adr:nn</code>	☆	
<code>\object_member_adr:Vn</code>	☆	

Fully expands to the address of specified member variable. If type is not specified it'll be retrieved from the generator proxy, but only if member is specified in the generator.

From: 1.0

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

Tests if the specified member exist.

From: 2.0

<code>\object_member_type:nn</code>	★	<code>\object_member_type:nn {<address>} {<member name>}</code>
<code>\object_member_type:Vn</code>	★	

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

<code>\object_new_member:nnn</code>		<code>\object_new_member:nnn {<address>} {<member name>} {<member type>}</code>
<code>\object_new_member:(Vnn nnv)</code>		

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

<code>\object_member_use:nnn</code>	★	<code>\object_member_use:nnn {<address>} {<member name>} {<member type>}</code>
<code>\object_member_use:(Vnn nnv)</code>	★	<code>\object_member_use:nn {<address>} {<member name>}</code>
<code>\object_member_use:nn</code>	★	
<code>\object_member_use:Vn</code>	★	

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

<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

6.4 Methods

Currentlu only constant methods (near and remote) are implemented in `lt3rawobjects` as explained before.

<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>*</code>	<code>\object_ncmethod_if_exist_p:nnn</code>	<code>{\langle address \rangle}</code>	<code>{\langle method name \rangle}</code>	<code>{\langle method variant \rangle}</code>
<code>\object_ncmethod_if_exist_p:Vnn</code>	<code>*</code>				
<code>\object_ncmethod_if_exist:nnnTF</code>	<code>*</code>	<code>\object_ncmethod_if_exist:nnnTF</code>	<code>{\langle address \rangle}</code>	<code>{\langle method name \rangle}</code>	<code>{\langle method variant \rangle}</code>
<code>\object_ncmethod_if_exist:VnnTF</code>	<code>*</code>				
<code>\object_rcmethod_if_exist_p:nnn</code>	<code>*</code>				
<code>\object_rcmethod_if_exist_p:Vnn</code>	<code>*</code>				
<code>\object_rcmethod_if_exist:nnnTF</code>	<code>*</code>				
<code>\object_rcmethod_if_exist:VnnTF</code>	<code>*</code>				

Tests if the specified method constant exist.

From: 2.0

<code>\object_new_cmethod:nnnn</code>	<code>\object_new_cmethod:nnnn</code>	<code>{\langle address \rangle}</code>	<code>{\langle method name \rangle}</code>	<code>{\langle method arguments \rangle}</code>	<code>{\langle code \rangle}</code>
<code>\object_new_cmethod:Vnnn</code>					

Creates a new method with specified name and argument types. The `{\langle method arguments \rangle}` 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>*</code>	<code>\object_ncmethod_call:nnn</code>	<code>{\langle address \rangle}</code>	<code>{\langle method name \rangle}</code>	<code>{\langle method variant \rangle}</code>
<code>\object_ncmethod_call:Vnn</code>	<code>*</code>				
<code>\object_rcmethod_call:nnn</code>	<code>*</code>				
<code>\object_rcmethod_call:Vnn</code>	<code>*</code>				

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

From: 2.0

6.5 Constant member creation

Unlike normal variables, constant variables in L^AT_EX3 are created in different ways depending on the specified type. So we dedicate a new section only to collect some of these functions readapted for near constants (remote constants are simply near constants created on the generator proxy).

<code>\object_newconst_tl:nnn</code>	<code>\object_newconst_⟨type⟩:nnn</code>	<code>{\langle address \rangle}</code>	<code>{\langle constant name \rangle}</code>	<code>{\langle value \rangle}</code>
<code>\object_newconst_tl:Vnn</code>				
<code>\object_newconst_str:nnn</code>				
<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>				

Creates a constant variable with type `⟨type⟩` and sets its value to `⟨value⟩`.

From: 1.1

<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>
------------------------------------	---

Expands to `\⟨type⟩_const:cn {⟨address⟩} {⟨value⟩}`, use it if you need to create simple constants with custom types.
From: 2.1

6.6 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.

6.7 Proxy utilities and object creation

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

<code>\object_test_proxy_p:nn *</code> <code>\object_test_proxy_p:Vn *</code> <code>\object_test_proxy:nnTF *</code> <code>\object_test_proxy:VnTF *</code>	<code>\object_test_proxy_p:nn {⟨object address⟩} {⟨proxy address⟩}</code> <code>\object_test_proxy:nnTF {⟨object address⟩} {⟨proxy address⟩} {⟨true code⟩} {⟨false code⟩}</code>
--	---

Test if the specified object is generated by the selected proxy, where *⟨proxy variable⟩* 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>\object_test_proxy_p:VN *</code> <code>\object_test_proxy:nNTF *</code> <code>\object_test_proxy:VNTF *</code>	<code>\object_test_proxy_p:nN {⟨object address⟩} {⟨proxy variable⟩}</code> <code>\object_test_proxy:nNTF {⟨object address⟩} {⟨proxy variable⟩} {⟨true code⟩} {⟨false code⟩}</code>
--	---

Test if the specified object is generated by the selected proxy, where *⟨proxy variable⟩* is a string variable holding the proxy address. The `:nN` variant don't use `e` expansion, instead of `:nn` 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 `rawobjects` module.

From: 1.0

<code>\object_create:nnnNN</code> <code>\object_create:VnnNN</code>	<code>\object_create:nnnNN {⟨proxy address⟩} {⟨module⟩} {⟨id⟩} {⟨scope⟩} {⟨visibility⟩}</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:VnnN</code> <code>\object_create:nnn</code> <code>\object_create:Vnn</code>	<code>\object_create:nnnN {⟨proxy address⟩} {⟨module⟩} {⟨id⟩} {⟨scope⟩}</code> <code>\object_create:nnn {⟨proxy address⟩} {⟨module⟩} {⟨id⟩}</code>
--	---

Same as `\object_create:nnnNN` but both create only public objects, and the `:nnn` version only global ones. Always use these two function instead of `\object_create:nnnNN` unless you strictly need private objects.

From: 2.3

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

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

From: 2.2

<code>\c_object_local_str</code> <code>\c_object_global_str</code>	
---	--

Possible values for *⟨scope⟩* parameter.

From: 1.0

<code>\c_object_public_str</code> <code>\c_object_private_str</code>	
---	--

Possible values for *⟨visibility⟩* parameter.

From: 1.0

\object_create_set:NnnnNN \object_create_set:(NVnnNN NnnfNN) \object_create_gset:NnnnNN \object_create_gset:(NVnnNN NnnfNN)	\object_create_set:NnnnNN <str var> {\proxy address} {\module} {\id} <scope> <visibility>
--	--

Creates an object and sets its fully expanded address inside <str var>.

From: 1.0

\object_allocate_incr:NnnnNN \object_allocate_incr:NNVnNN \object_gallocate_incr:NnnnNN \object_gallocate_incr:NNVnNN \object_allocate_gincr:NnnnNN \object_allocate_gincr:NNVnNN \object_gallocate_gincr:NnnnNN \object_gallocate_gincr:NNVnNN	\object_allocate_incr:NnnnNN <str var> <int var> {\proxy address} {\module} <scope> <visibility>
--	---

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

From: 1.1

\proxy_create:nnN \proxy_create_set:NnnN \proxy_create_gset:NnnN	\proxy_create:nnN {\module} {\id} <visibility> \proxy_create_set:NnnN <str var> {\module} {\id} <visibility>
--	---

These commands are deprecated because proxies should be global and public. Use instead `\proxy_create:nn`, `\proxy_create_set:Nnn` and `\proxy_create_gset:Nnn`.

From: 1.0

Deprecated in: 2.3

\proxy_create:nn \proxy_create_set:Nnn \proxy_create_gset:Nnn	\proxy_create:nn {\module} {\id} \proxy_create_set:Nnn <str var> {\module} {\id}
---	---

Creates a global public proxy object.

From: 2.3

\proxy_push_member:nnn \proxy_push_member:Vnn	\proxy_push_member:nnn {\proxy address} {\member name} {\member type}
--	---

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 `\object_member_type` functions.

From: 1.0

\proxy_push_embedded:nnn \proxy_push_embedded:Vnn	\proxy_push_embedded:nnn {\proxy address} {\embedded object name} {\embedded object proxy}
--	--

Updates a proxy object with a new embedded object specification.

From: 2.2

```
\proxy_add_initializer:nN
\proxy_add_initializer:VN
```

```
\proxy_add_initializer:nN {⟨proxy address⟩} {⟨initializer⟩}
```

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:

- the full expanded address of used proxy as an **n** argument;
- the module name as an **n** argument;
- the full expanded address of created object as an **n** argument.

Initializer will be executed in the same order they're added.

```
\object_assign:nn
\object_assign:(Vn|nV|VV)
```

```
\object_assign:nn {⟨to address⟩} {⟨from address⟩}
```

Assigns the content of each variable of object at *⟨from address⟩* to each corresponsive variable in *⟨to address⟩*. 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`.

```
\str_new:N \g_myproxy_str
\proxy_create_gset:Nnn \g_myproxy_str { example }{ myproxy }
\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.

```
\str_new:N \g_myobj_str
\object_create_gset:NVnn \g_myobj_str \g_myproxy_str
{ example }{ myobj }
\tl_gset:cn
{
  \object_member_adr:Vn \g_myobj_str { myvar }
}
{ \c_dollar_str{ } ~ dollar ~ \c_dollar_str{ } }
\object_member_use:Vn \g_myobj_str { myvar }
```

Output: \$ dollar \$

If you don't want to specify an object identifier you can also do

```
\int_new:N \g_intc_int
\object_gallocate_gincr:NNVnNN \g_myobj_str \g_intc_int \g_myproxy_str
{ example } \c_object_local_str \c_object_public_str
\tl_gset:cn
{
  \object_member_adr:Vn \g_myobj_str { myvar }
```

```

    }
    { \c_dollar_str{} ~ dollar ~ \c_dollar_str{} }
\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

```

\proxy_create:nn{ mymod }{ INT }
\proxy_push_member:nnn
{
  \object_address:nn{ mymod }{ INT }
}{ var }{ t1 }

```

Container proxy

```

\proxy_create:nn{ mymod }{ EXT }
\proxy_push_embedded:nnn
{
  \object_address:nn{ mymod }{ EXT }
}
{ emb }
{
  \object_address:nn{ mymod }{ INT }
}

```

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

```

\str_new:N \g_EXTobj_str
\int_new:N \g_intcount_int
\object_gallocate_gincr:NNnnNN
  \g_EXTobj_str \g_intcount_int
{
  \object_address:nn{ mymod }{ EXT }
}
{ mymod }
\c_object_local_str \c_object_public_str

```

and use the embedded object in the following way:

```

\object_member_set:nnn
{
  \object_embedded_adr:Vn \g_EXTobj_str { emb }
}{ var }{ Hi }
\object_member_use:nn
{
  \object_embedded_adr:Vn \g_EXTobj_str { emb }
}{ var }

```

Output: Hi

8 Implementation

```

1 <*package>
2 <@@=rawobjects>
3
4 Deprecation message
5
6 \msg_new:nnn { rawobjects }{ deprecate }
7 {
8   Command ~ #1 ~ is ~ deprecated. ~ Use ~ instead ~ #2
9 }
10
11 \cs_new_protected:Nn \__rawobjects_launch_deprecate:NN
12 {
13   \msg_warning:nnnn{ rawobjects }{ deprecate }{ #1 }{ #2 }
14 }
15
16 \rwobj_address_f:n It just performs a c expansion before passing it to \cs_to_str:N.
17
18 \cs_new:Nn \rwobj_address_f:n
19 {
20   \exp_args:Nc \cs_to_str:N { #1 }
21 }
22
23 (End definition for \rwobj_address_f:n. This function is documented on page 6.)
24
25
26 \c_object_local_str
27 \c_object_global_str
28 \c_object_public_str
29 \c_object_private_str
30
31 \str_const:Nn \c_object_local_str {l}
32 \str_const:Nn \c_object_global_str {g}
33 \str_const:Nn \c_object_public_str {_}
34 \str_const:Nn \c_object_private_str {__}
35
36
37 \cs_new:Nn \__rawobjects_scope:N
38 {
39   \str_use:N #1
40 }
41
42 \cs_new:Nn \__rawobjects_scope_pfx:N
43 {
44   \str_if_eq:NNF #1 \c_object_local_str
45   { g }
46 }
47
48 \cs_generate_variant:Nn \__rawobjects_scope_pfx:N { c }
49
50 \cs_new:Nn \__rawobjects_scope_pfx_cl:n
51 {
52   \__rawobjects_scope_pfx:c{
53     \object_ncmember_adr:nnn
54     {
55       \object_embedded_adr:nn { #1 }{ /_I_/ }
56     }
57   }
58 }

```



```

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 12.)

`\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.)

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

```

267
268 \cs_new:Nn \object_member_type:nn
269 {
270     \object_rcmember_use:nnn { #1 }
271     { #2 _ type }{ str }
272 }

```

273

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

274

```
275 \msg_new:nnnn { rawobjects }{ noerr }{ Unspecified ~ scope }
276 {
277   Object ~ #1 ~ hasn't ~ a ~ scope ~ variable
278 }
```

279

```
280 \msg_new:nnnn { rawobjects }{ scoperr }{ Nonstandard ~ scope }
281 {
282   Operation ~ not ~ permitted ~ on ~ object ~ #1 ~
283   ~ since ~ it ~ wasn't ~ declared ~ local ~ or ~ global
284 }
```

285

```
286 \cs_new_protected:Nn \__rawobjects_force_scope:n
287 {
288   \cs_if_exist:cTF
289   {
290     \object_ncmember_adr:nnn
291     {
292       \object_embedded_adr:nn{ #1 }{ /_I_/ }
293     }
294     { S }{ str }
295   }
296   {
297     \bool_if:nF
298     {
299       \object_if_local_p:n { #1 } || \object_if_global_p:n { #1 }
300     }
301     {
302       \msg_error:nnx { rawobjects }{ scoperr }{ #1 }
303     }
304   }
305   {
306     \msg_error:nnx { rawobjects }{ noerr }{ #1 }
307   }
308 }
309
```

\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

310

```
311 \prg_new_conditional:Nnn \object_member_if_exist:nnn {p, T, F, TF }
312 {
313   \cs_if_exist:cTF
314   {
315     \object_member_adr:nnn { #1 }{ #2 }{ #3 }
316   }
317   {
318     \prg_return_true:
319   }
320   {
321     \prg_return_false:
322   }
```

```

323 }
324
325 \prg_new_conditional:Nnn \object_member_if_exist:nn {p, T, F, TF }
326 {
327   \cs_if_exist:cTF
328   {
329     \object_member_adr:nn { #1 }{ #2 }
330   }
331   {
332     \prg_return_true:
333   }
334   {
335     \prg_return_false:
336   }
337 }
338
339 \prg_generate_conditional_variant:Nnn \object_member_if_exist:nnn
340 { Vnn }{ p, T, F, TF }
341 \prg_generate_conditional_variant:Nnn \object_member_if_exist:nn
342 { Vn }{ p, T, F, TF }
343

```

(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

```

344
345 \msg_new:nnnn{ rawobjects }{ none }{ Invalid ~ basic ~ type }{ Basic ~ type ~ #1 ~ doesn't
346
347 \cs_new_protected:Nn \object_new_member:nnn
348 {
349   \cs_if_exist_use:cTF { #3 _ new:c }
350   {
351     { \object_member_adr:nnn { #1 }{ #2 }{ #3 } }
352   }
353   {
354     \msg_error:nnn{ rawobjects }{ none }{ #3 }
355   }
356 }
357
358 \cs_generate_variant:Nn \object_new_member:nnn { Vnn, nnv }
359

```

(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
360
361 \cs_new:Nn \object_member_use:nnn
362 {
363   \cs_if_exist_use:cT { #3 _ use:c }
364   {
365     { \object_member_adr:nnn { #1 }{ #2 }{ #3 } }
366   }
367 }
368

```

```

369 \cs_new:Nn \object_member_use:nn
370 {
371   \object_member_use:nnv { #1 }{ #2 }
372   {
373     \object_rcmember_adr:nnn { #1 }
374     { #2 _ type }{ str }
375   }
376 }
377
378 \cs_generate_variant:Nn \object_member_use:nnn { Vnn, vnn, nnv }
379 \cs_generate_variant:Nn \object_member_use:nn { Vn }
380

```

(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_eq:nnn`

```

381
382 \cs_new_protected:Nn \object_member_set:nnnn
383 {
384   \cs_if_exist_use:cT
385   {
386     #3 _ \__rawobjects_scope_pfx_cl:n{ #1 } set:cn
387   }
388   {
389     { \object_member_adr:nnn { #1 }{ #2 }{ #3 } }
390     { #4 }
391   }
392 }
393
394 \cs_generate_variant:Nn \object_member_set:nnnn { Vnnn, nnvn }
395
396 \cs_new_protected:Nn \object_member_set:nnn
397 {
398   \object_member_set:nnvn { #1 }{ #2 }
399   {
400     \object_rcmember_adr:nnn { #1 }
401     { #2 _ type }{ str }
402   } { #3 }
403 }
404
405 \cs_generate_variant:Nn \object_member_set:nnn { Vnn }
406

```

(End definition for `\object_member_set:nnnn` and `\object_member_set_eq:nnn`. These functions are documented on page 8.)

`\object_member_set_eq:nnnN` Make a member equal to another variable.
`\object_member_set_eq:nnN`

```

407
408 \cs_new_protected:Nn \object_member_set_eq:nnnN
409 {
410   \__rawobjects_force_scope:n { #1 }
411   \cs_if_exist_use:cT
412   {
413     #3 _ \__rawobjects_scope_pfx:n { #1 } set _ eq:cn

```



```

414     }
415     {
416         { \object_member_adr:nnn { #1 }{ #2 }{ #3 } } #4
417     }
418 }
419
420 \cs_generate_variant:Nn \object_member_set_eq:nnnN { VnnN, nnnnc, Vnnnc, nnvN }
421
422 \cs_new_protected:Nn \object_member_set_eq:nnN
423 {
424     \object_member_set_eq:nnvN { #1 }{ #2 }
425     {
426         \object_rcmember_adr:nnn { #1 }
427         { #2 _ type }{ str }
428     } #3
429 }
430
431 \cs_generate_variant:Nn \object_member_set_eq:nnN { VnN, nnc, Vnc }
432

```

(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

```

433
434 \cs_new:Nn \object_ncmember_adr:nnn
435 {
436     \tl_to_str:n{ c _ } #1 \tl_to_str:n { _ CONST _ #2 _ #3 }
437 }
438
439 \cs_generate_variant:Nn \object_ncmember_adr:nnn { Vnn, vnn }
440

```

(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.

```

441
442 \cs_new:Nn \object_rcmember_adr:nnn
443 {
444     \object_ncmember_adr:vnn
445     {
446         \object_ncmember_adr:nnn
447         {
448             \object_embedded_adr:nn{ #1 }{ /_I_/ }
449         }
450         { P }{ str }
451     }
452     { #2 }{ #3 }
453 }
454
455 \cs_generate_variant:Nn \object_rcmember_adr:nnn { Vnn }

```

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

`\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.

```

456
457 \prg_new_conditional:Nnn \object_ncmember_if_exist:nnn {p, T, F, TF }
458 {
459   \cs_if_exist:cTF
460   {
461     \object_ncmember_adr:nnn { #1 }{ #2 }{ #3 }
462   }
463   {
464     \prg_return_true:
465   }
466   {
467     \prg_return_false:
468   }
469 }
470
471 \prg_new_conditional:Nnn \object_rcmember_if_exist:nnn {p, T, F, TF }
472 {
473   \cs_if_exist:cTF
474   {
475     \object_rcmember_adr:nnn { #1 }{ #2 }{ #3 }
476   }
477   {
478     \prg_return_true:
479   }
480   {
481     \prg_return_false:
482   }
483 }
484
485 \prg_generate_conditional_variant:Nnn \object_ncmember_if_exist:nnn
486 { Vnn }{ p, T, F, TF }
487 \prg_generate_conditional_variant:Nnn \object_rcmember_if_exist:nnn
488 { Vnn }{ p, T, F, TF }
489

```

(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`
`\object_rcmember_use:nnn`

Uses a near/remote constant.

```

490
491 \cs_new:Nn \object_ncmember_use:nnn
492 {
493   \cs_if_exist_use:cT { #3 _ use:c }
494   {
495     { \object_ncmember_adr:nnn { #1 }{ #2 }{ #3 } }
496   }
497 }
498
499 \cs_new:Nn \object_rcmember_use:nnn
500 {
501   \cs_if_exist_use:cT { #3 _ use:c }
502   {
503     { \object_rcmember_adr:nnn { #1 }{ #2 }{ #3 } }

```

```

504     }
505 }
506
507 \cs_generate_variant:Nn \object_ncmember_use:nnn { Vnn }
508 \cs_generate_variant:Nn \object_rcmember_use:nnn { Vnn }
509

```

(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

```

510
511 \cs_new_protected:Nn \object_newconst:nnnn
512 {
513   \use:c { #3 _ const:cn }
514   {
515     \object_ncmember_adr:nnn { #1 } { #2 } { #3 }
516   }
517   { #4 }
518 }
519

```

(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`

```

520
521 \cs_new_protected:Nn \object_newconst_tl:nnn
522 {
523   \object_newconst:nnnn { #1 } { #2 } { tl } { #3 }
524 }
525 \cs_new_protected:Nn \object_newconst_str:nnn
526 {
527   \object_newconst:nnnn { #1 } { #2 } { str } { #3 }
528 }
529 \cs_new_protected:Nn \object_newconst_int:nnn
530 {
531   \object_newconst:nnnn { #1 } { #2 } { int } { #3 }
532 }
533 \cs_new_protected:Nn \object_newconst_clist:nnn
534 {
535   \object_newconst:nnnn { #1 } { #2 } { clist } { #3 }
536 }
537 \cs_new_protected:Nn \object_newconst_dim:nnn
538 {
539   \object_newconst:nnnn { #1 } { #2 } { dim } { #3 }
540 }
541 \cs_new_protected:Nn \object_newconst_skip:nnn
542 {
543   \object_newconst:nnnn { #1 } { #2 } { skip } { #3 }
544 }
545 \cs_new_protected:Nn \object_newconst_fp:nnn
546 {
547   \object_newconst:nnnn { #1 } { #2 } { fp } { #3 }
548 }
549

```

```

550 \cs_generate_variant:Nn \object_newconst_tl:nnn { Vnn }
551 \cs_generate_variant:Nn \object_newconst_str:nnn { Vnn }
552 \cs_generate_variant:Nn \object_newconst_int:nnn { Vnn }
553 \cs_generate_variant:Nn \object_newconst_clist:nnn { Vnn }
554 \cs_generate_variant:Nn \object_newconst_dim:nnn { Vnn }
555 \cs_generate_variant:Nn \object_newconst_skip:nnn { Vnn }
556 \cs_generate_variant:Nn \object_newconst_fp:nnn { Vnn }
557
558
559 \cs_generate_variant:Nn \object_newconst_str:nnn { nnx }
560 \cs_generate_variant:Nn \object_newconst_str:nnn { nnV }
561

```

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

`\object_newconst_seq_from_clist:nnn` Creates a `seq` constant.

```

562
563 \cs_new_protected:Nn \object_newconst_seq_from_clist:nnn
564 {
565   \seq_const_from_clist:cn
566   {
567     \object_ncmember_adr:nnn { #1 } { #2 } { seq }
568   }
569   { #3 }
570 }
571
572 \cs_generate_variant:Nn \object_newconst_seq_from_clist:nnn { Vnn }
573

```

(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.

```

574
575 \cs_new_protected:Nn \object_newconst_prop_from_keyval:nnn
576 {
577   \prop_const_from_keyval:cn
578   {
579     \object_ncmember_adr:nnn { #1 } { #2 } { prop }
580   }
581   { #3 }
582 }
583
584 \cs_generate_variant:Nn \object_newconst_prop_from_keyval:nnn { Vnn }
585

```

(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`

```

586
587 \cs_new:Nn \object_ncmethod_adr:nnn
588 {
589   #1 \tl_to_str:n { _ CMETHOD _ #2 : #3 }
590 }
591
592 \cs_generate_variant:Nn \object_ncmethod_adr:nnn { Vnn , vnn }

```

```

593
594 \cs_new:Nn \object_rcmethod_adr:nnn
595 {
596   \object_ncmethod_adr:vnn
597   {
598     \object_ncmember_adr:nnn
599     {
600       \object_embedded_adr:nn{ #1 }{ /_I_/ }
601     }
602     { P }{ str }
603   }
604   { #2 }{ #3 }
605 }
606
607 \cs_generate_variant:Nn \object_ncmethod_adr:nnn { Vnn , vnn }
608 \cs_generate_variant:Nn \object_rcmethod_adr:nnn { Vnn }
609

```

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

```

\object_ncmethod_if_exist_p:nnn
\object_ncmethod_if_exist:nnnTF
\object_rcmethod_if_exist_p:nnn
\object_rcmethod_if_exist:nnnTF
Tests if the specified member constant exists.
610
611 \prg_new_conditional:Nnn \object_ncmethod_if_exist:nnn {p, T, F, TF }
612 {
613   \cs_if_exist:cTF
614   {
615     \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
616   }
617   {
618     \prg_return_true:
619   }
620   {
621     \prg_return_false:
622   }
623 }
624
625 \prg_new_conditional:Nnn \object_rcmethod_if_exist:nnn {p, T, F, TF }
626 {
627   \cs_if_exist:cTF
628   {
629     \object_rcmethodr_adr:nnn { #1 }{ #2 }{ #3 }
630   }
631   {
632     \prg_return_true:
633   }
634   {
635     \prg_return_false:
636   }
637 }
638
639 \prg_generate_conditional_variant:Nnn \object_ncmethod_if_exist:nnn
640 { Vnn }{ p, T, F, TF }
641 \prg_generate_conditional_variant:Nnn \object_rcmethod_if_exist:nnn

```

```

642 { Vnn }{ p, T, F, TF }
643

```

(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

```

644
645 \cs_new_protected:Nn \object_new_cmethod:nnnn
646 {
647   \cs_new:cn
648   {
649     \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
650   }
651   { #4 }
652 }
653
654 \cs_generate_variant:Nn \object_new_cmethod:nnnn { Vnnn }
655

```

(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`

```

656
657 \cs_new:Nn \object_ncmethod_call:nnn
658 {
659   \use:c
660   {
661     \object_ncmethod_adr:nnn { #1 }{ #2 }{ #3 }
662   }
663 }
664
665 \cs_new:Nn \object_rcmethod_call:nnn
666 {
667   \use:c
668   {
669     \object_rcmethod_adr:nnn { #1 }{ #2 }{ #3 }
670   }
671 }
672
673 \cs_generate_variant:Nn \object_ncmethod_call:nnn { Vnn }
674 \cs_generate_variant:Nn \object_rcmethod_call:nnn { Vnn }
675

```

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

```

676
677 \cs_new_protected:Nn \__rawobjects_initproxy:nnn
678 {
679   \object_newconst:nnnn
680   {
681     \object_embedded_adr:nn{ #3 }{ /_I_/ }
682   }
683   { ifprox }{ bool }{ \c_true_bool }
684 }

```

```

685 \cs_generate_variant:Nn \__rawobjects_initproxy:nnn { VnV }
686

```

\object_if_proxy_p:n Test if an object is a proxy.

\object_if_proxy:nTF

```

687
688 \cs_new:Nn \__rawobjects_bol_com:N
689 {
690   \cs_if_exist_p:N #1 && \bool_if_p:N #1
691 }
692
693 \cs_generate_variant:Nn \__rawobjects_bol_com:N { c }
694
695 \prg_new_conditional:Nnn \object_if_proxy:n {p, T, F, TF}
696 {
697   \cs_if_exist:cTF
698   {
699     \object_ncmember_adr:nnn
700     {
701       \object_embedded_adr:nn{ #1 }{ /_I_/ }
702     }
703     { ifprox }{ bool }
704   }
705   {
706     \bool_if:cTF
707     {
708       \object_ncmember_adr:nnn
709       {
710         \object_embedded_adr:nn{ #1 }{ /_I_/ }
711       }
712       { ifprox }{ bool }
713     }
714     {
715       \prg_return_true:
716     }
717     {
718       \prg_return_false:
719     }
720   }
721   {
722     \prg_return_false:
723   }
724 }
725

```

(End definition for \object_if_proxy:nTF. This function is documented on page 11.)

\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

```

726
727 \prg_generate_conditional_variant:Nnn \str_if_eq:nn { ve }{ TF }
728
729 \prg_new_conditional:Nnn \object_test_proxy:nn {p, T, F, TF}
730 {
731   \str_if_eq:veTF
732   {

```

```

733         \object_ncmember_adr:nnn
734         {
735             \object_embedded_adr:nn{ #1 }{ /_I_/ }
736         }
737         { P }{ str }
738     }
739 { #2 }
740 {
741     \prg_return_true:
742 }
743 {
744     \prg_return_false:
745 }
746 }
747
748 \prg_new_conditional:Nnn \object_test_proxy:nN {p, T, F, TF}
749 {
750     \str_if_eq:cNTF
751     {
752         \object_ncmember_adr:nnn
753         {
754             \object_embedded_adr:nn{ #1 }{ /_I_/ }
755         }
756         { P }{ str }
757     }
758     #2
759     {
760         \prg_return_true:
761     }
762     {
763         \prg_return_false:
764     }
765 }
766
767 \prg_generate_conditional_variant:Nnn \object_test_proxy:nn
768 { Vn }{p, T, F, TF}
769 \prg_generate_conditional_variant:Nnn \object_test_proxy:nN
770 { VN }{p, T, F, TF}
771

```

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

```

\object_create:nnnNN Creates an object from a proxy.
\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
\object_create:nnn
\cs_new_protected:Nn \__rawobjects_force_proxy:n
{
\object_if_proxy:nF { #1 }
{

```



```

782         \msg_error:nnn { rawobjects }{ notproxy }{ #1 }
783     }
784 }
785
786 \cs_new_protected:Nn \__rawobjects_create_anon:nnnNN
787 {
788     \tl_if_empty:nF{ #1 }
789     {
790
791         \__rawobjects_force_proxy:n { #1 }
792
793
794         \object_newconst_str:nnn
795         {
796             \object_embedded_adr:nn{ #3 }{ /_I_/ }
797         }
798         { M }{ #2 }
799         \object_newconst_str:nnn
800         {
801             \object_embedded_adr:nn{ #3 }{ /_I_/ }
802         }
803         { P }{ #1 }
804         \object_newconst_str:nnV
805         {
806             \object_embedded_adr:nn{ #3 }{ /_I_/ }
807         }
808         { S } #4
809         \object_newconst_str:nnV
810         {
811             \object_embedded_adr:nn{ #3 }{ /_I_/ }
812         }
813         { V } #5
814
815         \seq_map_inline:cn
816         {
817             \object_member_adr:nnn { #1 }{ varlist }{ seq }
818         }
819         {
820             \object_new_member:nnv { #3 }{ ##1 }
821             {
822                 \object_ncmember_adr:nnn { #1 }{ ##1 _ type }{ str }
823             }
824         }
825
826         \seq_map_inline:cn
827         {
828             \object_member_adr:nnn { #1 }{ objlist }{ seq }
829         }
830         {
831             \embedded_create:nvn
832             { #3 }
833             {
834                 \object_ncmember_adr:nnn { #1 }{ ##1 _ proxy }{ str }
835             }

```

```

836         { ##1 }
837     }
838
839     \tl_map_inline:cn
840     {
841         \object_member_adr:nnn { #1 }{ init }{ t1 }
842     }
843     {
844         ##1 { #1 }{ #2 }{ #3 }
845     }
846
847 }
848 }
849
850 \cs_generate_variant:Nn \__rawobjects_create_anon:nnnNN { xnxNN, xvxcc }
851
852 \cs_new_protected:Nn \object_create:nnnNN
853 {
854     \__rawobjects_create_anon:xnxNN { #1 }{ #2 }
855     { \object_address:nn { #2 }{ #3 } }
856     #4 #5
857 }
858
859 \cs_generate_variant:Nn \object_create:nnnnNN { VnnNN }
860
861 \cs_new_protected:Nn \object_create_set:NnnnnNN
862 {
863     \object_create:nnnnNN { #2 }{ #3 }{ #4 } #5 #6
864     \str_set:Nx #1 { \object_address:nn { #3 }{ #4 } }
865 }
866
867 \cs_new_protected:Nn \object_create_gset:NnnnnNN
868 {
869     \object_create:nnnnNN { #2 }{ #3 }{ #4 } #5 #6
870     \str_gset:Nx #1 { \object_address:nn { #3 }{ #4 } }
871 }
872
873 \cs_generate_variant:Nn \object_create_set:NnnnnNN { NVnnNN, NnnfNN }
874 \cs_generate_variant:Nn \object_create_gset:NnnnnNN { NVnnNN, NnnfNN }
875
876
877
878 \cs_new_protected:Nn \object_create:nnnN
879 {
880     \object_create:nnnnNN { #1 }{ #2 }{ #3 } #4 \c_object_public_str
881 }
882
883 \cs_generate_variant:Nn \object_create:nnnN { VnnN }
884
885 \cs_new_protected:Nn \object_create_set:NnnnnN
886 {
887     \object_create_set:NnnnnNN #1 { #2 }{ #3 }{ #4 } #5 \c_object_public_str
888 }
889

```

```

890 \cs_new_protected:Nn \object_create_gset:NnnnN
891 {
892   \object_create_gset:NnnnNN #1 { #2 }{ #3 }{ #4 } #5 \c_object_public_str
893 }
894
895 \cs_generate_variant:Nn \object_create_set:NnnnN { NVnnN }
896 \cs_generate_variant:Nn \object_create_gset:NnnnN { NVnnN }
897
898 \cs_new_protected:Nn \object_create:nnn
899 {
900   \object_create:nnnNN { #1 }{ #2 }{ #3 }
901   \c_object_global_str \c_object_public_str
902 }
903
904 \cs_generate_variant:Nn \object_create:nnn { Vnn }
905
906 \cs_new_protected:Nn \object_create_set:Nnnn
907 {
908   \object_create_set:NnnnNN #1 { #2 }{ #3 }{ #4 }
909   \c_object_global_str \c_object_public_str
910 }
911
912 \cs_new_protected:Nn \object_create_gset:Nnnn
913 {
914   \object_create_gset:NnnnNN #1 { #2 }{ #3 }{ #4 }
915   \c_object_global_str \c_object_public_str
916 }
917
918 \cs_generate_variant:Nn \object_create_set:Nnnn { NVnn }
919 \cs_generate_variant:Nn \object_create_gset:Nnnn { NVnn }
920
921
922
923
924 \cs_new_protected:Nn \embedded_create:nnn
925 {
926   \__rawobjects_create_anon:xvxc { #2 }
927   {
928     \object_ncmember_adr:nnn
929     {
930       \object_embedded_adr:nn{ #1 }{ /_I_/ }
931     }
932     { M }{ str }
933   }
934   {
935     \object_embedded_adr:nn
936     { #1 }{ #3 }
937   }
938   {
939     \object_ncmember_adr:nnn
940     {
941       \object_embedded_adr:nn{ #1 }{ /_I_/ }
942     }
943     { S }{ str }

```

```

944     }
945     {
946         \object_ncmember_adr:nnn
947         {
948             \object_embedded_adr:nn{ #1 }{ /_I_/ }
949         }
950         { V }{ str }
951     }
952 }
953
954 \cs_generate_variant:Nn \embedded_create:nnn { nvn, Vnn }
955

```

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

<code>\proxy_create:nn</code> <code>\proxy_create_set:Nnn</code> <code>\proxy_create_gset:Nnn</code>	<p>Creates a new proxy object</p> <pre> 956 957 \cs_new_protected:Nn \proxy_create:nn 958 { 959 \object_create:VnnNN \c_proxy_address_str { #1 }{ #2 } 960 \c_object_global_str \c_object_public_str 961 } 962 963 \cs_new_protected:Nn \proxy_create_set:Nnn 964 { 965 \object_create_set:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 } 966 \c_object_global_str \c_object_public_str 967 } 968 969 \cs_new_protected:Nn \proxy_create_gset:Nnn 970 { 971 \object_create_gset:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 } 972 \c_object_global_str \c_object_public_str 973 } 974 975 976 977 \cs_new_protected:Nn \proxy_create:nnN 978 { 979 __rawobjects_launch_deprecate:NN \proxy_create:nnN \proxy_create:nn 980 \object_create:VnnNN \c_proxy_address_str { #1 }{ #2 } 981 \c_object_global_str #3 982 } 983 984 \cs_new_protected:Nn \proxy_create_set:NnnN 985 { 986 __rawobjects_launch_deprecate:NN \proxy_create_set:NnnN \proxy_create_set:Nnn 987 \object_create_set:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 } 988 \c_object_global_str #4 989 } 990 991 \cs_new_protected:Nn \proxy_create_gset:NnnN 992 { 993 __rawobjects_launch_deprecate:NN \proxy_create_gset:NnnN \proxy_create_gset:Nnn </pre>
--	--

```

994     \object_create_gset:NVnnNN #1 \c_proxy_address_str { #2 }{ #3 }
995     \c_object_global_str #4
996 }
997

```

(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.

```

998
999 \cs_new_protected:Nn \proxy_push_member:nnn
1000 {
1001   \object_newconst_str:nnn { #1 }{ #2 _ type }{ #3 }
1002   \seq_gput_left:cn
1003   {
1004     \object_member_adr:nnn { #1 }{ varlist }{ seq }
1005   }
1006   { #2 }
1007 }
1008
1009 \cs_generate_variant:Nn \proxy_push_member:nnn { Vnn }
1010

```

(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.

```

1011
1012 \cs_new_protected:Nn \proxy_push_embedded:nnn
1013 {
1014   \object_newconst_str:nnx { #1 }{ #2 _ proxy }{ #3 }
1015   \seq_gput_left:cn
1016   {
1017     \object_member_adr:nnn { #1 }{ objlist }{ seq }
1018   }
1019   { #2 }
1020 }
1021
1022 \cs_generate_variant:Nn \proxy_push_embedded:nnn { Vnn }
1023

```

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

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

```

1024
1025 \cs_new_protected:Nn \proxy_add_initializer:nN
1026 {
1027   \tl_gput_right:cn
1028   {
1029     \object_member_adr:nnn { #1 }{ init }{ tl }
1030   }
1031   { #2 }
1032 }
1033
1034 \cs_generate_variant:Nn \proxy_add_initializer:nN { VN }
1035

```

(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.

```

1036
1037 \str_const:Nx \c_proxy_address_str
1038   { \object_address:nn { rawobjects }{ proxy } }
1039
1040 \object_newconst_str:nnn
1041   {
1042     \object_embedded_adr:Vn \c_proxy_address_str { /_I/ }
1043   }
1044   { M }{ rawobjects }
1045
1046 \object_newconst_str:nnV
1047   {
1048     \object_embedded_adr:Vn \c_proxy_address_str { /_I/ }
1049   }
1050   { P } \c_proxy_address_str
1051
1052 \object_newconst_str:nnV
1053   {
1054     \object_embedded_adr:Vn \c_proxy_address_str { /_I/ }
1055   }
1056   { S } \c_object_global_str
1057
1058 \object_newconst_str:nnV
1059   {
1060     \object_embedded_adr:Vn \c_proxy_address_str { /_I/ }
1061   }
1062   { V } \c_object_public_str
1063
1064
1065 \__rawobjects_initproxy:VnV \c_proxy_address_str { rawobjects } \c_proxy_address_str
1066
1067 \object_new_member:Vnn \c_proxy_address_str { init }{ tl }
1068
1069 \object_new_member:Vnn \c_proxy_address_str { varlist }{ seq }
1070
1071 \object_new_member:Vnn \c_proxy_address_str { objlist }{ seq }
1072
1073 \proxy_push_member:Vnn \c_proxy_address_str
1074   { init }{ tl }
1075 \proxy_push_member:Vnn \c_proxy_address_str
1076   { varlist }{ seq }
1077 \proxy_push_member:Vnn \c_proxy_address_str
1078   { objlist }{ seq }
1079
1080 \proxy_add_initializer:VN \c_proxy_address_str
1081   \__rawobjects_initproxy:nnn
1082

```

(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

```

```

1083
1084 \cs_new:Nn \__rawobjects_combine_aux:nnn
1085 {
1086     anon . #3 . #2 . #1
1087 }
1088
1089 \cs_generate_variant:Nn \__rawobjects_combine_aux:nnn { Vnf }
1090
1091 \cs_new:Nn \__rawobjects_combine:Nn
1092 {
1093     \__rawobjects_combine_aux:Vnf #1 { #2 }
1094     {
1095         \cs_to_str:N #1
1096     }
1097 }
1098
1099 \cs_new_protected:Nn \object_allocate_incr:NNnnNN
1100 {
1101     \object_create_set:NnnfNN #1 { #3 }{ #4 }
1102     {
1103         \__rawobjects_combine:Nn #2 { #3 }
1104     }
1105     #5 #6
1106
1107     \int_incr:N #2
1108 }
1109
1110 \cs_new_protected:Nn \object_gallocate_incr:NNnnNN
1111 {
1112     \object_create_gset:NnnfNN #1 { #3 }{ #4 }
1113     {
1114         \__rawobjects_combine:Nn #2 { #3 }
1115     }
1116     #5 #6
1117
1118     \int_incr:N #2
1119 }
1120
1121 \cs_generate_variant:Nn \object_allocate_incr:NNnnNN { NNvNN }
1122
1123 \cs_generate_variant:Nn \object_gallocate_incr:NNnnNN { NNvNN }
1124
1125 \cs_new_protected:Nn \object_allocate_gincr:NNnnNN
1126 {
1127     \object_create_set:NnnfNN #1 { #3 }{ #4 }
1128     {
1129         \__rawobjects_combine:Nn #2 { #3 }
1130     }
1131     #5 #6
1132
1133     \int_gincr:N #2
1134 }
1135
1136 \cs_new_protected:Nn \object_gallocate_gincr:NNnnNN

```

```

1137 {
1138   \object_create_gset:NnnfNN #1 { #3 }{ #4 }
1139   {
1140     \__rawobjects_combine:Nn #2 { #3 }
1141   }
1142   #5 #6
1143
1144   \int_gincr:N #2
1145 }
1146
1147 \cs_generate_variant:Nn \object_allocate_gincr:NNnnNN { NNvNNN }
1148
1149 \cs_generate_variant:Nn \object_gallocate_gincr:NNnnNN { NNvNNN }
1150

```

(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.

```

1151 \cs_new_protected:Nn \object_assign:nn
1152 {
1153   \seq_map_inline:cn
1154   {
1155     \object_member_adr:vnn
1156     {
1157       \object_ncmember_adr:nnn
1158       {
1159         \object_embedded_adr:nn{ #1 }{ /_I_/ }
1160       }
1161       { P }{ str }
1162     }
1163     { varlist }{ seq }
1164   }
1165   {
1166     \object_member_set_eq:nnc { #1 }{ ##1 }
1167     {
1168       \object_member_adr:nn{ #2 }{ ##1 }
1169     }
1170   }
1171 }
1172
1173 \cs_generate_variant:Nn \object_assign:nn { nV, Vn, VV }

```

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

```

1174 </package>

```