1 Minim

minim.opm

```
\_def\_optexminim_version{0.1}
\_codedecl \optexminim_loaded  {Minim compatibility for \OpTeX/ (v\_optexminim_version)}
```

When we want to use minim with OpTEX, we need to accomodate for their differences in allocations and callbacks. This package tries to do that in a way that works, but is not necessarily the nicest – this are really core routines we are talking about, and both formats have their own ways, which in certain parts (don't) try to keep backwards compatibility with older TEX formats.

OpTEX defines most allocation macros in `alloc.opm` and some Lua allocation functions in `optex.lua`. Minim "packages" are not standalone, they all depend on core routines defined in `minim-alloc.tex` and `minim-alloc.lua` respectively. Minim as a format preloads a stripped version of `etex.src` so the Lua code makes some assumptions about that (i.e. expects local allocators).

Both OpTEX and minim want to make it possible to register more functions for a single callback, by chaining their calls and `callback.register()`ing only a proxy function. While minim stays close to the `callback` interface from LuaTEX, OpTEX is a subset of the LATEX `luatexbase` interface.

Because we only change what is defined by others, we actually need a dummy macro for `\_codedecl`.

minim.opm

```
\_catcode`\@=11
% dummy macro to signalize that we are loaded
\_let\optexminim_loaded=\empty
```

In general, there are four allocator types expected by minim:

- Knuth allocators from `plain.tex` (like `\newcount`). These are already defined by OpTEX (except for `\newlanguage`, which doesn't concern us). Although minim itself sets the old `\allocationnumber` counter, which is not even defined in OpTEX.
- Global allocators from `etex.src` (like `\globcount`). These are not defined by OpTEX, since they no longer make sense (LuaTEX doesn't use sparse arrays for registers). Minim defines them to be the classic Knuth allocators if it doesn't find them on the TeX side, but expects them from the Lua side.
- Local allocators from `etex.src` (like `\loccount`). Concept of local allocators is completely missing in OpTEX. The semantics of local allocation in Lua is weird too, so we try to avoid these, since minim also doesn't use them.
- LuaTEX allocators from `ltluatex.tex` (like `\newattribute`). Subset of these is in OpTEX (only atributes, which are also allocatable in Lua, and catcode tables). But minim tries to be compatible with LATEX and patches its routines if it detects them.

For defining TEX commands implemented in Lua, OpTEX has `define_lua_command`, which actually does the allocation and definition at the same time, and allows to do so only from Lua end.

Historically (in my opinion unfortunately) LATEX made the allocations of these functions available from TEX end, and the "lua define" operation is thus a two step process which involves synchronization with TEX.

`minim-alloc` actually defines a `luadef` function which is like `define_lua_command`, but is backed by the minim allocator. To make this work, we just need to set the LATEX register to the index of last allocated function, since it allocates at counter plus one. Then minim will start where OpTEX stopped, and we will later define `define_lua_command` to be just minim's `luadef`.

We tell the number of allocated function by going through the table of actually used functions. This is not that robust, because while `define_lua_command` allocates sequentially, the provided functions may be `nil`, which breaks the code below.

minim.opm

```
\_newcount\allocationnumber

% for synchronisation of allocated Lua functions
\_ea\_newcount\_csname e@alloc@luafunction@count\_endcsname

\directlua{
    local function_table = lua.get_functions_table()
    local i = 1
    while function_table[i] ~= nil do
        i = i + 1
    end
    % minim allocates at count + 1 for "new" allocators
    tex.setcount("global", "e@alloc@luafunction@count", i - 1)
}
```

Callbackwise, although minim's approach is simpler, it has a fatal flaw – there isn't real support for removing functions from callbacks. Of course, the individual functions could have some switches to turn them off, but the problem is that the callbacks should have their implicit behaviour when no callback is registered. That is why we have to keep OpTEX's higher level interface, and just implement minim on top.

We do this by hiding the `luatexbase` namespace temporarily (so that minim doesn't take it into account) and replacing the LuaTEX functions by proxies that call the OpTEX mechanism.

Minim reexports theses three functions, so they should be reasonably functional. Though currently, there are a couple of exceptions:

- Replacing a registered function removes the old function and adds a new one. This can mess with the order of functions, which may or may not be fine, but there isn't any high level interface for actually deciding the order of functions anyways, so it will have to do.
- `list` returns only those callbacks that are currently registered by minim. This is fine for the current use by minim (which just saves all registered callbacks when it is loaded, initializes its mechanism and reinserts the callbacks with its functions), but may not be for general use.
- Disabling callbacks is not supported at all. This is also case in OpTEX so unless need arises this should be fine.

minim.opm

```
\directlua{
    local lb = luatexbase
    luatexbase = nil
    local registered = {}
    function callback.register(cb, fn)
        if fn == false then % disable the callback
            % not supported
        elseif fn == nil then % disable the anonymous function
            registered[cb] = nil
            lb.remove_from_callback(cb, "minim")
        else % register the anonymous function
            if registered[cb] then
                % already registered, to replace remove the old
                lb.remove_from_callback(cb, "minim")
            end
            registered[cb] = fn
            lb.add_to_callback(cb, fn, "minim")
        end
    end

    % should return list of all callbacks, but we don't have access to that
    function callback.list(cb, fn)
        % return copy of the list
        local t = {}
        for k, _ in ipairs(registered) do
            t[k] = true
        end
        return t
    end

    function callback.find(cb, fn)
        return registered[cb]
    end

    callback.luatexbase = lb
}
```

Minim hooks into language mechanism with standard ε-TEX `\uselanguage@hook`. OpTEX doesn't have the hook, so we add it.

minim.opm

```
\_def\uselanguage@hook#1{}
\_let\.uselang=\_uselang
\_def\_uselang#1#2#3#4{%
   \.uselang{#1}{#2}{#3}{#4}
   \_ea\uselanguage@hook\_expanded{{\_cs{_nlan:#1}}}
}
```

The preparations are over. We load `minim-alloc.tex`.

minim.opm

```
\input minim-alloc
```

Both LaTeX and the minim inspired catcode table allocators initialize the catcode tables with `\initcatcodetable` (i.e. iniTeX catcodes). OpTeX merely allocates the registers. LuaTeX doesn't allow to activate unitialized catcode table, therefore activation with either `\initcatcodetable` or `\savecatcodetable` is necessary before use. To ensure compatibility with foreign macros, we also issue `\initcatcodetable` on allocation in the public version of `\newcatcodetable`.

minim.opm

```
\_def\newcatcodetable#1{\_newcatcodetable#1\_initcatcodetable#1}
```

By now, the Knuthian allocators are dealt with. ε-TeX global and local allocators are still undefined, but are expected in minim's Lua code with their hardcoded counter register numbers. This is unacceptable, since in this range (`\count260` to `\count276`) OpTeX has already made allocations. Thus we need to replace these Lua functions with similar definitions. For some, OpTeX also has a different idea whether the counter represent the last or next allocated register number, so we correct that as well.

We simply delete minim's Lua functions for local allocations. It actually doesn't use them (just like in the TeX case) and the semantics are just weird, so it is easier to simply get rid of them, and not implement them.

We also don't forget to actually set `define_lua_command` to be minim's `luadef` and to restore the `luatexbase` namespace.

minim.opm

```
\directlua{
    luatexbase = callback.luatexbase
    callback.luatexbase = nil

    local minimalloc = require("minim-alloc")

    define_lua_command = minimalloc.luadef

    % these are allocators already defined in OpTeX that we need to repair
    local toreplace = {
        "count",
        "dimen",
        "skip",
        "muskip",
        "box",
        "toks",
        "marks",
        "attribute",
        "catcodetable",
    }

    for _, alloc in ipairs(toreplace) do
        local cache = {}
        local countername = string.format("_\_pcent salloc", alloc)
        minimalloc["new_"..alloc] = function(id)
            local n = cache[id]
            if not n then
                n = tex.getcount(countername) + 1
                tex.setcount("global", countername, n)
                if id then
                    cache[id] = n
                end
                minimalloc.log(
                    "\_nbb\_pcent s\_pcent d : \_pcent s", alloc, n, id or "<unnamed>")
            end
            return n
        end
    end
}
```

We also need to do something about `minim-hooks.tex`, which hooks into `\shipout`, but the default OpTeX output routine (and perhaps also the user ones) use `\_shipout`.

Minim also adds to `\everypar`, but that is fine.

minim.opm

```
\_let\shipout\_shipout

```

```
\input minim-hooks

\_catcode`\:=11
\_let\_shipout\minim:shipout:new

% catcodes changes don't propage, since this file is loaded with \opinput
```

2 Minim-mp

minim-mp.opm

```
\_codedecl \_optexminimmp_used {Minim-PDF for \OpTeX/}
\_namespace{optexminimmp}

\_def\.used{}

\_load[minim]

\_input minim-mp

\_endnamespace
```

3 Minim-PDF

minim-pdf.opm

```
\_codedecl \nohyphlang {Minim-PDF for \OpTeX/}
\_namespace{optexminimpdf}
```

Before loading `minim-pdf` we do a few preparations. Most importantly adjusting core of minim, which is done in `minim.opm`.

minim-pdf.opm

```
\_load[minim]

\_catcode`\@=11
```

If not detected, a few "dummy" languages would be (in erroneous ways) defined by minim: like "nohyph" and "undetermined". We define a few dummy control sequences, to make minim not define them, since we define them ourselves below. They are used in standard way, but their "ISO codes" are weird:

```
\nohyphlang
\nolanglang
\uncodedlang
\undeterminedlang
```

minim-pdf.opm

```
\_def\_tmp{}
\_let\lang@nohyph=\_tmp
\_let\lang@nolang=\_tmp
\_let\lang@uncoded=\_tmp
\_let\lang@undetermined=\_tmp

% OpTeX doesn't use language numbers up until 100, so we start at
% 20 (leave a few for temporary assignments, like usual)
\_preplang nohyph       nohyph       {} 1 11
\_preplang nolang       nolang       {} 2 11
\_preplang uncoded      uncoded      {} 3 11
\_preplang undetermined undetermined {} 4 11
```

Now we actually load `minim-pdf`.

minim-pdf.opm

```
\_input minim-pdf
```

Users aren't supposed to define custom languages in OpTEX, forbid that.

minim-pdf.opm

```
\_def\_tmp{\errmessage{don't use this command with OpTeX}}
\_let\newnamedddialect=\_tmp
\_let\newnameddllanguage=\_tmp
```

Since a language may already be set (at least the default Knuth english), then we need to tell minim about it, by reexecuting the language command (like `\enlang`), thus calling into minim through the above mentioned hook.

minim-pdf.opm

```
% set the current language again to let minim know what it is
\_cs{\_cs{_lan:\_the\_language}lang}

% catcodes changes don't propage, since this file is loaded with \opinput
\_endnamespace
```