# The xargs package

Manuel Pégourié-Gonnard
mpg@elzevir.fr

v1.1 (2008/03/22)

## Contents

*Important note for French users*: a French version of the user documentation is included in the `xargs-fr.pdf` file.

## 1 Introduction

Defining commands with an optional argument is easy in LaTeX 2ε. There is, however, two limitations: you can make only one argument optional and it must be the first one. The xargs package provide extended variants of `\newcommand` & friends, for which these limitations no longer hold: It is now easy to define commands with many (and freely placed) optional arguments, using a nice ⟨*key*⟩=⟨*value*⟩ syntax. For example, the following defines a command with two optional arguments.

```
\newcommandx*\coord[3][1=1, 3=n]{(#2_{#1},\ldots,#2_{#3})}
```

| | |
|---|---|
| `$\coord{x}$` | $(x_1, \ldots, x_n)$ |
| `$\coord[0]{y}$` | $(y_0, \ldots, y_n)$ |
| `$\coord{z}[m]$` | $(z_1, \ldots, z_m)$ |
| `$\coord[0]{t}[m]$` | $(t_0, \ldots, t_m)$ |

## 2 Usage

### 2.1 Basics

The xargs package defines an extended variant for every LaTeX macro related to macro definition. xargs's macro are named after their LaTeX counterparts, just adding an `x` at end (see the list in the margin). Here is the complete list:

```
\newcommandx       \renewcommandx
\newenvironmentx   \renewenvironmentx
\providecommandx   \DeclareRobustCommandx
\CheckCommandx
```

If you are not familiar with all of them, you can either just keep using the commands you already know, or check Lamport's book or the LaTeX Companion (or any LaTeX 2$_\varepsilon$ manual) to learn the others.

Since these commands all share the same syntax, I'll always use `\newcommandx` in the following, but remember it works the same for all seven commands. (Of course, command about environments take one more argument, for the end definition.) Here is `\newcommandx`'s complete syntax.

`\newcommandx`⟨*∗*⟩`{`⟨*command*⟩`}[`⟨*number*⟩`][`⟨*list*⟩`]{`⟨*definition*⟩`}`

Everything here is the same as the usual `\newcommand` syntax, except ⟨*list*⟩. Let's recall this briefly. The optional ∗ make LaTeX define a "short" macro, that is a macro that won't accept a paragraph break (`\par` or an empty line) inside its argument; if you don't put it, the macro will be long. ⟨*command*⟩ is any control sequence, and can but need not be enclosed in braces, as you like. The ⟨*number*⟩ specifies how many arguments your macro will take (including optional ones): it should be a non-negative integer, and at most 9. The macro's ⟨*definition*⟩ is a balanced text, where every `#` sign must be followed by a number, thus representing an argument, or by another `#` sign. The two arguments ⟨*number*⟩ and ⟨*list*⟩ are optionals.

Now comes the new and funny part. ⟨*list*⟩ is a coma-separated list of element ⟨*digit*⟩=⟨*value*⟩. Here, ⟨*digit*⟩ should be non-zero, and at most ⟨*number*⟩ (the total number of arguments). The ⟨*value*⟩ is any balanced text, and can be empty. If so, the `=` sign becomes optional: You only need to write ⟨*digit*⟩ if you want the ⟨*digit*⟩th argument to be optional, with empty default value. Of course, every argument whose number is a ⟨*digit*⟩ in the ⟨*list*⟩ becomes optional, with ⟨*value*⟩ as its default value.

If you are not very familiar with some aspects of the syntax provided by the xkeyval package, you may be interested in the following remarks about the syntax of ⟨*list*⟩. Since ⟨*list*⟩ is coma-separated, if you want to use a coma inside a ⟨*value*⟩, you need to enclose it (either the coma or the whole ⟨*value*⟩) in braces. The same applies if you want to use a closing square bracket inside the ⟨*list*⟩. Don't worry about those unwanted braces, they will be removed later. Actually, xkeyval removes up to 3 braces set: If you really want braces around a value, you need to type something like `1={{{{\large stuff}}}}`.

That's all for the basics: you are now ready to use xargs, and can stop reading this doc now if you want. If, however, you are wondering about what happens if you have many successive optional arguments, or care about doing global definitions, or even need to know precisely the limitations of xargs, go on with the next subsections.

## 2.2 The `usedefault` key

So, what happens with many successive optional arguments? The default behaviour is that of LaTeX's commands like `\makebox` or `\parbox`: you can't specify the third argument if you didn't specify the first two ones. For example, in my first example, please notice how I used the mandatory argument to separate the two optional ones.

However, maybe you don't like this and prefer choosing your argument's order as you want, according to their logical meaning. Ok. That's exactly what the `usedefault` key is for. Just include it in the ⟨*list*⟩, and you can now use `[]` to skip one optional argument (using its default value) and go to the next one.

```
\newcommandx*\coord[3][2=1,3=n,usedefault]{(#2_{#1},\ldots,#2_{#3})}
```

| | |
|---|---|
| `$\coord{x}$` | $(1_x,\ldots,1_n)$ |
| `$\coord{y}[0]$` | $(0_y,\ldots,0_n)$ |
| `$\coord{z}[][m]$` | $(1_z,\ldots,1_m)$ |
| `$\coord{t}[0][m]$` | $(0_t,\ldots,0_m)$ |

Of course, on this simple example, this is merely a matter of taste, but sometimes the `usedefault` key can save you a lot of typing, since the optional value for an argument can be rather long, and you don't always have enough mandatory arguments to separate the optional ones.

This simple way of using `usedefault` has one problem: you can no more specify an empty value for an optional argument. Of course you need a special value of the argument to mean "please use the default value there", but it doesn't need to be always the empty string. Actually, you can say `usedefaut=`⟨*flag*⟩ to choose this special value. The following example has no other purpose that illustrating this.

```
\newcommandx*\test[2][1=A, 2=B, usedefault=@]{(#1,#2)}
```

| | |
|---|---|
| `\test[b]` | (b,B) |
| `\test[][b]` | (,b) |
| `\test[@][b]` | (A,b) |

## 2.3 Using a prefix

Standard `\newcommand` allows you to define so-called "long" (i. e. the arguments may contain a `\par` token) or "short" (they may not) macros with the optional star. This is part of what TeX calls a "prefix" for the definition, namely the `\long` prefix. Other components of the prefix are `\global`, `\outer`, and $\varepsilon$-TeX's `\protected`. There is no way to use them with `\newcommand`, though `\global` can be specially interesting in order to avoid definitions made inside a group (e. g. an environment) "disappear" at the end of the group. (For details about the other possible components, see the TeXbook and $\varepsilon$-TeX's manual.)

With `xargs`, you can use the `addprefix` key, *except* for the `\outer` prefix, which is not and will not be supported (and not used anywhere I know in LaTeX 2$_\varepsilon$, either). Please note that it *adds* a prefix to the current one, which by default is `\long` for the unstarred form, and empty for the starred form. You can also use this key many times: all prefixes will be merged together. For example, the following two instructions do the exactly the same thing.

```
\newcommandx*\foo[0][addprefix=\global, addprefix=\long,
  addprefix=\protected]{bar}
\newcommandx\foo[0][addprefix=\global\protected]{bar}
```

By the way, macros with at least one optional argument are already robust in LaTeX 2$_\varepsilon$'s meaning of the word, so I don't know if the `\protected` prefix is very interesting there. Maybe the ability to perform global definitions is the main use of the `addprefix` key.

### 2.4 Compatibility and known limitations

Okay, let's see the bad things (the limitations) first. There is essentially one: you cannot use in the ⟨*list*⟩ some elements, because xkeyval won't handle them properly. Namely, hash signs (tokens with `\catcode` 6), and `\par` tokens are forbidden, and any part of the list should look properly `\if`-balanced to TEX. Only the first limitation is shared by the standard `\newcommand`: it accepts no hash signs in a default value. Apart from this, you can use anything you want, everywhere you want, as far as I know.

Now the "good" features. I've tried hard to make macros defined with xargs as much similar as possible with those defined with standard LATEX's commands. Actually, when `\newcommandx` is asked to perform a definition which `\newcommand` can do, the resulting macro will be defined exactly as the latter would have done. More precisely, the following code (and similar tests) issues no warning.

```
\newcommandx\foo[2][1=default]{def-foo}
\CheckCommand\foo[2][default]{def-foo}
\newcommand\baz{def-baz}
\CheckCommandx*\baz[0][addprefix=\long]{def-baz}
```

Moreover, there are only three points (to my knowledge) where xargs's commands differ from the kernel ones. The first one was already mentioned, it is due to using xkeyval for precessing the ⟨*list*⟩. The second and third points are meant to be good one. Second point is: There is a bug[1] in the current implementation of `\CheckCommand`, that I obviously tried to avoid.

Last, I don't use kernel's (nor amsmath's) version of `\@ifnextchar`. Indeed, a problem arises when the last argument of a command is optional: we have to make a choice about what to do with spaces while scanning ahead for a left square bracket. I chose to scan over them, and the put them back in the text in there were no optional argument. I'm no more sure it is the right thing to do, and I'll probably make an option to let the user decide in a future version.

## 3 Implementation

First, I'd like to recall the way LATEX 2$_\varepsilon$ handles optional arguments. For example, a command `\foo` defined with `\newcommand*\foo[2][bar]{baz}` is implemented as the pair:

```
\foo=macro:->\@protected@testopt\foo\\foo{bar}
\\foo=macro:[#1]#2->baz
```

There is one "external" macro `\foo`, which is merely an argument grabber or a parser, and an internal macro `\\foo`, which is the "real" one. In xargs this is quite the same, except that the external macro is a bit more sophisticated. This idea of the external macro being a parser, giving the arguments to the internal one in a

---

[1] http://www.latex-project.org/cgi-bin/ltxbugs2html?pr=latex/3971

standard form, reflects in the name of LaTeX3's experimental package for extended argument syntax: xparse.

Here the approach is a bit different. Of course, the idea is still to build a parser for the specified syntax, but since `\newcommandx` user syntax is based on xkeyval, we also have to care about keys and there default values, and to put the whole thing in the correct order before we build up the parser. We also seek for compatibility with existing LaTeX 2$_\varepsilon$ commands, which adds a few tests. The organisation is as follows.

Before we really begin, just a few preliminaries. First, load the xkeyval package for it's nice key=value syntax.

```
\RequirePackage{xkeyval}
```

`\xargs@max`
`\xargs@temp`
`\xargs@toksa`
`\xargs@toksb`

Then allocate a few registers and make sure the name of our private scratch macro is free for use. Note that for certain uses, we really need a `\toks` register because the string used can possibly contain `#` characters. Sometimes I also use a `\toks` register instead of a macro just for ease of use (writing less `\expandafter`s).

```
\@ifdefinable\xargs@max{\newcount\xargs@max}

\@ifdefinable\xargs@temp\relax
\@ifdefinable\xargs@toksa{\newtoks\xargs@toksa}
\@ifdefinable\xargs@toksb{\newtoks\xargs@toksb}
```

## 3.1 Parser

Let's begin with a simple, concrete example. Recall that we defined a command `\vect` with `\newcommandx\vect[3]{1=0,3=n}{(#2_{#1},\ldots,#2_{#3}}` and have a look at its implementation.

```
\vect=macro:->\@protected@testopt@xargs\vect\\vect
  {\xargs@test@opt{0},\xargs@put@arg,\xargs@test@opt{n},}
\\vect=macro:[#1]#2[#3]->(#2_{#1},\ldots ,#2_{#3})
```

As you can see, the parser is stored as a coma-separated list of "actions". Here the only actions are to grab a mandatory argument or to check for the presence of of an optional one. In this case, special care is taken about spaces. Actually, there can be one more action, associated with the `usedefaut` user key: `\xargs@setdefflag`, which specifies the flag that says "use the default value".

The parsing is done by a loop that read and executes each action from the originating list, and concurrently builds an argument list such as `[0]{x}[m]` to be passed to `\\vect` for example. All of this happens inside a group.

`\@protected@testop@xargs`

This first macro closely resembles kernel's `\@protected@testopt` (similarity in their names is intentional, see `\CheckCommandx`). It just checks the protection context and call the real argument grabbing macro.

```
\newcommand*\@protected@testopt@xargs[1]{%
  \ifx\protect\@typeset@protect
    \expandafter\xargs@read
  \else
    \@x@protect#1%
  \fi}
```

`\xargs@read`

Initiate the loop. `\xargs@toksa` will become the call to the internal macro with all arguments, `\xargs@toksb` contains the actions list for arguments grabbing.

```
\newcommand*\xargs@read[2]{%
  \begingroup
  \xargs@toksa{#1}%
  \xargs@toksb{#2}%
  \xargs@continue}
```

`\xargs@continue`
`\xargs@pick@next`

Each iteration of the loop consist of two steps: pick the next action (and remove it from the list), and execute it. When there is no more action in the list, it means the arguments grabbing stage is over, and it's time to execute the internal macro by expanding the contents of `\xargs@toksa`.

```
\newcommand\xargs@continue{%
  \expandafter\xargs@pick@next\the\xargs@toksb,\@nil
  \xargs@temp}
```

```
\@ifdefinable\xargs@pick@next{%
  \def\xargs@pick@next#1,#2\@nil{%
    \def\xargs@temp{#1}%
    \xargs@toksb{#2}%
    \ifx\xargs@temp\empty
      \def\xargs@temp{\expandafter\endgroup\the\xargs@toksa}%
    \fi}}
```

`\xargs@set@defflag`

Let's begin with the most simple action.

```
\newcommand*\xargs@set@defflag[1]{%
  \def\xargs@default@flag{#1}}
```

`\xargs@put@arg`
`\xargs@test@opt`
`\xargs@put@opt`

Now have a look at the argument grabbing macros. The first one, `\xargs@put@arg`, just reads an undelimited argument in the input stack and add it to the arguments list. `\xargs@testopt` checks if the next non-space token is a square bracket to decide if it have to read an argument from the input or use the default value, and takes care to enclose it in square brackets.

```
\newcommand\xargs@put@arg[1]{%
  \xargs@toksa\expandafter{\the\xargs@toksa{#1}}%
  \xargs@continue}
```

```
\newcommand*\xargs@test@opt[1]{%
  \xargs@ifnextchar[%]
    {\xargs@grab@opt{#1}}%
    {\xargs@put@opt{#1}}}
```

```
\newcommand\xargs@put@opt[1]{%
    \xargs@toksa\expandafter{\the\xargs@toksa[{#1}]}%
  \xargs@continue}
```

```
\@ifdefinable\xargs@grab@opt{%
  \long\def\xargs@grab@opt#1[#2]{%
    \toks@{#2}\edef\xargs@temp{\the\toks@}%
    \ifx\xargs@temp\xargs@default@flag
      \expandafter\@firstoftwo
    \else
      \expandafter\@secondoftwo
    \fi{%
      \xargs@put@opt{#1}%
      }{%
      \xargs@put@opt{#2}}}}
```

`\xargs@ifnextchar`
`\xargs@ifnch`
`\xargs@xifnch`

Here comes a modified version of `\@ifnextchar`, that works like kernel's one, except that it remembers how many spaces it gobbles and puts them back in case the next non-space character isn't a match. Not sure whether this is the better way to do, may change in future versions.

```
\newcommand\xargs@ifnextchar[3]{%
  \let\xargs@temp\empty
  \let\reserved@d=#1%
  \def\reserved@a{#2}%
  \def\reserved@b{#3}%
  \futurelet\@let@token\xargs@ifnch}
```

```
\newcommand\xargs@ifnch{%
  \ifx\@let@token\@sptoken
    \edef\xargs@temp{\xargs@temp\space}%
    \let\reserved@c\xargs@xifnch
  \else
    \ifx\@let@token\reserved@d
      \let\reserved@c\reserved@a
    \else
      \def\reserved@c{\expandafter\reserved@b\xargs@temp}%
    \fi
  \fi
  \reserved@c}
```

```
\@ifdefinable\xargs@xifnch{%
  \expandafter\def\expandafter\xargs@xifnch\space{%
    \futurelet\@let@token\xargs@ifnch}}
```

## 3.2 Keys

Okay, we are done with the parsing related macros. Now define stuff for the definition of macros. In this part we use xkeyval. Let's start with the particular keys for options `addprefix` and `default`. Like all `xargs` key, we use the prefix `xargs` and the familly `key`. The `addprefix` key can be used many times : each value is appended at the end of the current prefix. Actually, we also construct a "short" prefix (without any `\long`), for the external macro. We define them globally, since key processing will happen inside a group, and the definition outside.

```
\@ifdefinable\xargs@key@addprefix{%
  \define@key[xargs]{key}{addprefix}[]{%
    \global\expandafter\def\expandafter\xargs@prefix\expandafter{%
      \xargs@prefix#1}%
    \xargs@makeshort#1\long\@nil}}
```

The `\long` tokens are removed from the prefix in a fast and easy way, assuming the input is a correct prefix. (It will crash e.g. if the input contains an undefined CS or braces, but this will make all crash later anyway. By the way, we also assume the prefix contains no macro parameter token. . . )

```
\@ifdefinable\xargs@makeshort{%
  \def\xargs@makeshort#1\long#2{%
```

```
    \expandafter\gdef\expandafter\xargs@shortpref\expandafter{%
      \xargs@shortpref#1}%
    \ifx#2\@nil \else
      \expandafter\xargs@makeshort\expandafter#2%
    \fi}}
```

The initial prefixes will be fixed by `\newcommandx` and its friends when they check the star: empty in the stared version, `\long` otherwise. For this, they use xargs's variant or `\@star@or@long`:

```
\newcommand\xargs@star@or@long[1]{%
  \global\let\xargs@shortpref\@empty
  \@ifstar{\gdef\xargs@prefix{}#1}{\gdef\xargs@prefix{\long}#1}}
```

Now, another particular key is the `usedefault` key. When used, it just sets `\xargs@default@flag` and the corresponding boolean. Later on, this will be used to possibly introduce a `\xargs@set@default` action at the beginning of the actions list.

```
\define@key[xargs]{key}{usedefault}[]{%
  \xargs@toksa{#1}\edef\xargs@default@flag{\the\xargs@toksa}}
```

Let's continue with the more important keys. We have to collect through xkeyval at most 9 actions numbered 1 to `\xargs@max` (the total number of arguments), each of them being `\xargs@test@opt` or `\xargs@put@arg`. Latter, we will use them to build up the parser.

`\@namenewc`
`\xargs@action@1`
`\xargs@action@2`
`\xargs@action@3`
`\xargs@action@4`
`\xargs@action@5`
`\xargs@action@6`
`\xargs@action@7`
`\xargs@action@8`
`\xargs@action@9`

So our first task is to define container macros for the at most nine actions which represent arguments parsing, with default value `\xargs@put@arg` since every argument is mandatory unless specified.

```
\providecommand\@namenewc[1]{%
  \expandafter\newcommand\csname #1\endcsname}
```

```
\@namenewc{xargs@action@1}{\xargs@put@arg}
\@namenewc{xargs@action@2}{\xargs@put@arg}
\@namenewc{xargs@action@3}{\xargs@put@arg}
\@namenewc{xargs@action@4}{\xargs@put@arg}
\@namenewc{xargs@action@5}{\xargs@put@arg}
\@namenewc{xargs@action@6}{\xargs@put@arg}
\@namenewc{xargs@action@7}{\xargs@put@arg}
\@namenewc{xargs@action@8}{\xargs@put@arg}
\@namenewc{xargs@action@9}{\xargs@put@arg}
```

`\xargs@def@key`

The next macro will define the keys. Its first argument is the key's number. The second argument will be discussed later.

```
\newcommand*\xargs@def@key[2]{%
  \expandafter\@ifdefinable\csname xargs@key@#1\endcsname{%
    \define@key[xargs]{key}{#1}[]{%
```

The first thing do to, before setting any action, is to check whether this key can be used for this command, and complain if not.

```
      \ifnum\xargs@max<#1
        \PackageError{xargs}{%
          Illegal argument label in\MessageBreak
          optional arguments description%
          }{%
          You are trying to make optional an argument whose label (#1)
          \MessageBreak is higher than the total number (\the\xargs@max)
          of parameters. \MessageBreak This can't be done and your
          demand will be ignored.}%
      \else
```

If the key number is correct, it may be that the user is trying to use it twice for the same command. Since it's probably a mistake, issue a warning in such case.

```
        \expandafter\expandafter\expandafter
        \ifx\csname xargs@action@#1\endcsname\xargs@put@arg \else
          \PackageWarning{xargs}{%
            Argument #1 was allready given a default value.\MessageBreak
            Previous value will be overriden.\MessageBreak}%
        \fi
```

If everything looks okay, define the action to be `\xargs@test@opt` with the given value, and execute the (for now) mysterious second argument.

```
        \@namedef{xargs@action@#1}{\xargs@test@opt{##1}}%
        #2%
      \fi}}}
```

`\ifxargs@firstopt@`
`\ifxargs@otheropt@`
`\xargs@key@1`
`\xargs@key@2`
`\xargs@key@3`
`\xargs@key@4`
`\xargs@key@5`
`\xargs@key@6`
`\xargs@key@7`
`\xargs@key@8`
`\xargs@key@9`

The second argument is used to set the value for some `\if` which will keep track of the existence of an optional argument other than the first one, and the of the possibly optional nature of the first. Such information will be useful when we will have to decide if we use the $\LaTeX\,2_\varepsilon$ standard way or xargs custom one to define the macro.

```
\newif\ifxargs@firstopt@
\newif\ifxargs@otheropt@
```

Now actually define the keys.

```
\xargs@def@key1\xargs@firstopt@true
\xargs@def@key2\xargs@otheropt@true \xargs@def@key3\xargs@otheropt@true
\xargs@def@key4\xargs@otheropt@true \xargs@def@key5\xargs@otheropt@true
\xargs@def@key6\xargs@otheropt@true \xargs@def@key7\xargs@otheropt@true
\xargs@def@key8\xargs@otheropt@true \xargs@def@key9\xargs@otheropt@true
```

`\xargs@setkeys`
`\xargs@check@keys`

We set the keys with the starred version of `\setkeys`, so we can check if there were some strange keys we cannot handle, and issue a meaningful warning if there are some.

```
\newcommand\xargs@setkeys[1]{%
  \setkeys*[xargs]{key}{#1}%
  \xargs@check@keys}
```

```
\newcommand\xargs@check@keys{%
  \ifx\XKV@rm\empty \else
    \xargs@toksa\expandafter{\XKV@rm}%
    \PackageError{xargs}{%
      Illegal key or argument label in\MessageBreak
      optional arguments description%
      }{%
      You can only use non-zero digits as argument labels.\MessageBreak
      Other allowed keys are usedefault and addprefix.\MessageBreak
      You wrote: "\the\xargs@toksa".\MessageBreak
      I can't understand this and I'm going to ignore it.}%
  \fi}
```

## 3.3 Definition

`\xargs@add@args`

Now our goal is to build two lists from our up to nine argument grabbers, the special action `\xargs@setdefflag` and not forgetting the prefix. The first list is the coma-separated list of actions already discussed. The second is the parameter text for use in the definition on the internal macro, for example `[#1]#2[#3]`. The next macro takes the content of a `\xargs@action@X` macro for argument and adds the corresponding items to this lists. It checks if the first token of its parameter is `\xargs@testopt` in order to know if the `#n` has to be enclosed in square brackets.

```
\newcommand\xargs@add@args[1]{%
  \xargs@toksa\expandafter{\the\xargs@toksa #1,}%
  \expandafter
  \ifx\@car#1\@nil\xargs@put@arg
    \xargs@toksb\expandafter\expandafter\expandafter{%
      \the\expandafter\xargs@toksb\expandafter##\the\count@}%
  \else
    \xargs@toksb\expandafter\expandafter\expandafter{%
      \the\expandafter\xargs@toksb\expandafter
      [\expandafter##\the\count@]}%
  \fi}
```

`\xargs@process@keys`

Here comes the main input processing macro, which prepares the information needed to define the final macro, and expands it to the defining macro.

```
\@ifdefinable\xargs@process@keys{%
  \long\def\xargs@process@keys#1[#2]{%
```

Some initialisations. We work inside a group so that the default values for the `\xargs@action@X` macros and the `\xargs@XXXopt@` be automatically restored for the next time.

```
    \begingroup
    \xargs@setkeys{#2}%
    \xargs@toksa{}\xargs@toksb{}%
```

Let's begin with the `usedefault` part.

```
    \@ifundefined{xargs@default@flag}{}{%
      \xargs@toksa\expandafter{%
        \expandafter\xargs@set@defflag\expandafter{%
          \xargs@default@flag}}}
```

Then the main loop actually builds up the two lists in the correct order.

```
    \count@\z@
    \@whilenum\xargs@max>\count@ \do{%
      \advance\count@\@ne
      \expandafter\expandafter\expandafter\xargs@add@args
      \expandafter\expandafter\expandafter{%
        \csname xargs@action@\the\count@\endcsname}}%
```

Then we need to address a special case: if only the first argument is optional, we use LaTeX 2$_\varepsilon$'s standard `\newcommand` construct, and we dont need an actions list like the one just build, but only the default value for the first argument. In this case, we extract this value from `\xargs@action@1` by expanding it two times with a modified `\xargs@testopt` (and one more expansion step for the `\csname` gives 3, hence the $2^3 - 1 = 7$ `\expandafter`s).

```
    \ifxargs@otheropt@ \else
      \ifxargs@firstopt@
```

```
        \let\xargs@test@opt\@firstofone
        \xargs@toksa\expandafter\expandafter\expandafter
        \expandafter\expandafter\expandafter\expandafter{%
          \csname xargs@action@1\endcsname}%
      \fi
    \fi
```

Finally expand the stuff to the next macro and, while we're at it, choose the next macro : depending of the existence and place of an optional argument, use LaTeX's or xargs's way. In the LaTeX case, however, we don't use `\@argdef` or `\xargdef` since we want to be able to use a prefix (and we have more work done allready, too).

```
    \edef\xargs@temp{%
      \ifxargs@otheropt@ \noexpand\xargs@xargsdef \else
        \ifxargs@firstopt@ \noexpand\xargs@xargdef \else
          \noexpand\xargs@argdef
      \fi\fi
      \noexpand#1%
      \expandafter\noexpand\csname\string#1\endcsname
      {\the\xargs@toksa}{\the\xargs@toksb}}%
```

Now we can close the group and forget all about key values, etc. Time to conclude and actually define the macro. (The only thing not passed as an argument is the prefix, which is globally set.) We also take care to execute `\xargs@drc@hook` just outside the group.

```
    \expandafter\endgroup
    \expandafter\xargs@drc@hook
    \xargs@temp}}
```

`\xargs@argdef`
`\xargs@xargdef`
`\xargs@xargsdef`

The first two next macros are modified versions of kernel's `\@argdef` and `\@xargdef`, that do the same work, but use the prefix we built, and also are simpler since they get the internal name as an argument. The last one is the only new macro.

```
\newcommand\xargs@argdef[5]{%
  \@ifdefinable#1{%
    \xargs@prefix\def#1#4{#5}}}
```

```
\newcommand\xargs@xargdef[5]{%
  \@ifdefinable#1{%
    \xargs@shortpref\def#1{\@protected@testopt#1#2{#3}}%
    \xargs@prefix\def#2#4{#5}}}
```

```
\newcommand\xargs@xargsdef[5]{%
  \@ifdefinable#1{%
    \xargs@shortpref\def#1{\@protected@testopt@xargs#1#2{#3}}%
    \xargs@prefix\def#2#4{#5}}}
```

## 3.4 User macros

`\newcommandx`
`\xargs@newc`

All the internal macros are ready. It's time to define the user commands, beginning with `\newcommandx`. Like its standard version, it just checks the star and call the next macro wich grabs the number of arguments.

```
\newcommand\newcommandx{%
  \xargs@star@or@long\xargs@newc}
```

```
\newcommand*\xargs@newc[1]{%
  \@testopt{\xargs@set@max{#1}}{0}}
```

`\xargs@set@max` Set the value of `\xargs@max`. If no optional arguments description follows, simply call `\argdef` because all the complicated stuff is useless here.

```
\@ifdefinable\xargs@set@max{%
  \def\xargs@set@max#1[#2]{%
    \kernel@ifnextchar[%]
      {\xargs@max=#2 \xargs@check@max{#1}}%
      {\@argdef#1[#2]}}}
```

`\xargs@check@max` To avoid possible problems later, check right now that `\xargs@max` value is valid. If not, warn the user and treat this value as zero. Then begin the key processing.

```
\newcommand\xargs@check@max{%
  \ifcase\xargs@max \or\or\or\or\or\or\or\or\or\else
    \PackageError{xargs}{Illegal number, treated as zero}{The total
      number of arguments must be in the 0..9 range.\MessageBreak
      Since your value is illegal, i’m going to use 0 instead.}
    \xargs@max0
  \fi
  \xargs@process@keys}
```

The other macros (`\renewcommandx` etc.) closely resemble their kernel homologues, since they are mostly wrappers around some call to `\xargs@newc`. There is however two exceptions: `\CheckCommand` and `\DeclareRobustCommandx`. Indeed, the current implementation of `\CheckCommand` in the kernel suffers from two bugs (see PR/3971) which I’m trying to avoid. For `\DeclareRobustCommandx`, the problem is to handle the prefix correctly: for that we use a hook, in order to delay the external macro’s definition until we get the prefix right. So, let’s see those two commands first.

`\CheckCommandx` We begin as usual detecting the possible star.

```
\newcommand\CheckCommandx{%
  \xargs@star@or@long\xargs@CheckC}
\@onlypreamble\CheckCommandx
```

`\xargs@CheckC`
`\xargs@check@a`
`\xargs@check@b`
First, we don’t use the `#2#` trick from the kernel, since it can fail if there are braces in the default values. Instead, we follow the argument grabing method used for `\new@environment`, ie calling `\kernel@ifnextchar` explicitly.

```
\newcommand\xargs@CheckC[1]{%
  \@testopt{\xargs@check@a#1}0}
\@onlypreamble\xargs@CheckC
```

```
\@ifdefinable\xargs@check@a{%
  \def\xargs@check@a#1[#2]{%
    \kernel@ifnextchar[%]
      {\xargs@check@b#1[#2]}%
      {\xargs@check@c#1{[#2]}}}}
\@onlypreamble\xargs@check@a
```

```
\@ifdefinable\xargs@check@b{%
  \def\xargs@check@b#1[#2][#3]{%
    \xargs@check@c{#1}{[#2][{#3}]}}}
\@onlypreamble\xargs@check@b
```

`\xargs@CheckC`

Here comes the major difference with the kernel version. If `\\reserved@a` is defined, we not only check that it is equal to `\\foo` (assuming `\foo` is the macro being tested), we also check that `\foo` makes something sensible, with `\xargs@check@d`.

```
\newcommand\xargs@check@c[3]{%
  \xargs@toksa{#1}%
  \expandafter\let\csname\string\reserved@a\endcsname\relax
  \xargs@renewc\reserved@a#2{#3}%
  \@ifundefined{\string\reserved@a}{%
    \ifx#1\reserved@a \else
      \xargs@check@complain
    \fi
    }{%
    \expandafter
    \ifx\csname\string#1\expandafter\endcsname
        \csname\string\reserved@a\endcsname
      \xargs@check@d
    \else
      \xargs@check@complain
    \fi}}
\@onlypreamble\xargs@check@c
```

So, what do we want `\foo` to do? If `\\foo` is defined, `\foo` should begin with one of the followings:

```
\@protected@testopt \foo \\foo
\@protected@testopt@xargs \foo \\foo
```

Since I'm to lazy to really check this, the `\xargs@check@d` macro only checks if the `\meaning` of `\foo` begins with `\@protected@test@opt` (without a space after it). It does this using a macro with delimited argument. Here are preliminaries to this definition: We need to have this string in `\catcode` 12 tokens.

```
\def\xargs@temp{\@protected@testopt}
\expandafter\xargs@toksa\expandafter{\meaning\xargs@temp}
\def\xargs@temp#1 {\def\xargs@temp{#1}}
\expandafter\xargs@temp\the\xargs@toksa
```

`\xargs@check@d`
`\xargs@check@e`

Now, `\xargs@check@c` just pass the `\meaning` of the command `\foo` being checked to the allready mentionned macro with delimited arguments, which will check if its first argument is empty (ie, if `\foo`'s `\meaning` starts with what we want) and complain otherwise.

```
\expandafter\newcommand\expandafter\xargs@check@d\expandafter{%
  \expandafter\expandafter\expandafter\xargs@check@e
  \expandafter\meaning\expandafter\reserved@a\xargs@temp\@nil}
\@onlypreamble\xargs@check@d
```

```
\@ifdefinable\xargs@check@e{%
  \expandafter\def\expandafter\xargs@check@e
  \expandafter#\expandafter1\xargs@temp#2\@nil{%
    \ifx\empty#1\empty \else
      \xargs@check@complain
    \fi}}
\@onlypreamble\xargs@check@e
```

`\xargs@check@complain`

The complaining macro uses the name saved by `\xargs@check@c` in `\xargs@toksa` in order to complain about the correct macro.

```
\newcommand\xargs@check@complain{%
  \PackageWarningNoLine{xargs}{Command \the\xargs@toksa has changed.
    \MessageBreak Check if current package is valid}}
\@onlypreamble\xargs@check@complain
```

`\DeclareRobustCommandx`
`\xargs@DRC`

The xargs version of `\DeclareRobustCommand`, and related internal macros.

```
\newcommand\DeclareRobustCommandx{%
  \xargs@star@or@long\xargs@DRC}
```

```
\newcommand*\xargs@DRC[1]{%
  \ifx#1\@undefined\else\ifx#1\relax\else
    \PackageInfo{xargs}{Redefining \string#1}%
  \fi\fi
  \edef\reserved@a{\string#1}%
  \def\reserved@b{#1}%
  \edef\reserved@b{\expandafter\strip@prefix\meaning\reserved@b}%
```

Here is the difference from kernel's code: instead of doing the definition of the user macro now, we just set the hook to do it latter, when the correct prefix will be set, then disable itself for next time.

```
  \edef\xargs@drc@hook{%
    \noexpand\xargs@shortpref\def\noexpand#1{%
      \ifx\reserved@a\reserved@b
        \noexpand\x@protect
        \noexpand#1%
      \fi
      \noexpand\protect
      \expandafter\noexpand\csname
      \expandafter\@gobble\string#1 \endcsname}%
    \expandafter\let\noexpand\xargs@drc@hook\relax}%
  \let\@ifdefinable\@rc@ifdefinable
  \expandafter\xargs@newc\csname
  \expandafter\@gobble\string#1 \endcsname}
```

And finally set a default empty hook.

```
\let\xargs@drc@hook\relax
```

From now on, there is absolutely nothing to comment, since the next macros are mainly wrappers around `\xargs@newc`, just as kernel's ones are wrappers around `\new@command`. So the code below is only copy/paste with search&replace from the kernel code.

`\renewcommandx`
`\xargs@renewc`

The xargs version of `\renewcommand`, and it's related internal macro.

```
\newcommand\renewcommandx{%
  \xargs@star@or@long\xargs@renewc}
```

```
\newcommand*\xargs@renewc[1]{%
  \begingroup\escapechar\m@ne
    \xdef\@gtempa{{\string#1}}%
  \endgroup
  \expandafter\@ifundefined\@gtempa{%
    \PackageError{xargs}{\noexpand#1undefined}{%
      Try typing \space <return> \space to proceed.\MessageBreak
      If that doesn't work, type \space X <return> \space to quit.}}%
    \relax
  \let\@ifdefinable\@rc@ifdefinable
  \xargs@newc#1}
```

\providecommandx
\xargs@providec

The xargs version of `\providecommand`, and the related internal macro.

```
\newcommand\providecommandx{%
  \xargs@star@or@long\xargs@providec}
```

```
\newcommand*\xargs@providec[1]{%
  \begingroup\escapechar\m@ne
    \xdef\@gtempa{{\string#1}}%
  \endgroup
  \expandafter\@ifundefined\@gtempa
    {\def\reserved@a{\xargs@newc#1}}%
    {\def\reserved@a{\renew@command\reserved@a}}%
  \reserved@a}
```

\newenvironment
\xargs@newenv
\xargs@newenva
\xargs@newenvb
\xargs@new@env

The xargs version of `\newenvironment`, and related internal macros.

```
\newcommand\newenvironmentx{%
  \xargs@star@or@long\xargs@newenv}
```

```
\newcommand*\xargs@newenv[1]{%
  \@testopt{\xargs@newenva#1}0}
```

```
\@ifdefinable\xargs@newenva{%
  \def\xargs@newenva#1[#2]{%
    \kernel@ifnextchar[%]
      {\xargs@newenvb#1[#2]}%
      {\xargs@new@env{#1}{[#2]}}}}
```

```
\@ifdefinable\xargs@newenvb{%
  \def\xargs@newenvb#1[#2][#3]{%
    \xargs@new@env{#1}{[#2][{#3}]}}}
```

```
\newcommand\xargs@new@env[4]{%
  \@ifundefined{#1}{%
    \expandafter\let\csname#1\expandafter\endcsname
    \csname end#1\endcsname}%
    \relax
  \expandafter\xargs@newc
    \csname #1\endcsname#2{#3}%
  \xargs@shortpref\expandafter\def\csname end#1\endcsname{#4}}
```

\renewenvironment
\xargs@renewenv

The xargs version of `\renewenvironment`, and the related internal macro.

```
\newcommand\renewenvironmentx{%
  \xargs@star@or@long\xargs@renewenv}
```

```
\newcommand*\xargs@renewenv[1]{%
  \@ifundefined{#1}{%
    \PackageError{xargs}{\noexpand#1undefined}{%
      Try typing \space <return> \space to proceed.\MessageBreak
      If that doesn't work, type \space X <return> \space to quit.}}%
    \relax
  \expandafter\let\csname#1\endcsname\relax
  \expandafter\let\csname end#1\endcsname\relax
  \xargs@newenv{#1}}
```

That's all folks!
Happy TEXing!