The xargs package

Manuel Pégourié-Gonnard
mpg@elzevir.fr

1.09 (2008/03/08)

Contents

1 Introduction

Defining commands with an optional argument is easy in LaTeX 2_ε. There is, however, two limitations: there can be at most one optional argument and it must be the first one. The xargs package provide extended variants of `\newcommand` & friends, for which these limitations no longer hold.

You may know some tricks in order to define commands with many optional arguments, or with last argument optional, etc. Theses tricks are subject to a few problems (using arguments in arbitrary order can be difficult, sometimes space is gobbled where it should not), which can be difficult to solve. Anyway, you don't want to worry about such tricky things while writing a document.

The xargs package provides you with an easy and (hopefully) robust way to define such commands, using a nice ⟨*key*⟩=⟨*value*⟩ syntax.

2 Usage

The xargs package defines an extended variant for every LaTeX macro related to macro definition. xargs's macro are named after their LaTeX couterparts, just adding an `x` at end, e.g. `\newcommandx`, `\renewcommandx` or `\newenvironmentx`. Since they all share the same syntax (closely ressembling `\newcommand`'s one), I will only explain with `\newcommandx`.

Let's begin with an example. After the following definitions,

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

you can use `\vect` like this:

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

You surely understood `\vect` is now a macro with 3 arguments, the first and third being optional. They both have their own default value (resp. `1` and `n`). You maybe noticed how `\newcommandx`'s syntax closely ressembles `\newcommand`'s syntax: the only difference is that, instead of the default value of the only optional argument, you have to specifiy the number of the optional arguments, followed by a = and their default value.

Now let's have a closer look at `\newcommandx`'s syntax, shared by all other xargs commands. (While dealing with details, here is the complete list of those: `\newcommandx` and `\renewcommandx` for simple macros, `\providecommandx` to make sure a macro exists, `\DeclareRobustCommandx` to (re)define a robust macro, `\CheckCommandx` to check if a macro has the correct meaning, `\newenvironmentx` and `\renewenvironmentx` for environments.)

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

Everything here is the same as 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, otherwise 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 with a number, thus representing an argument, or with 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.

While using a macro with many optional arguments, keep in mind the following fact. If arguments, say 1 and 2, are optional, then if you specify a value for only one optional argument, it will be used for argument 1, and argument 2 will be considered non-specified (thus its default value will be used). This behaviour is consistent with existing LaTeX's command, like `\makebox`. It isn't a technical limitation, I just couldn't imagine a better way to do. By the way, please notice the way I separated the two optional arguments from the example above in order to be able to use them independantly.

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}}}}`.

The last thing you (maybe) need to know about ⟨*list*⟩ is a little limitation of xargs, inherited from xkeyval, which I didn't managed to work around (actually, I

know a way to do it, but it fails in rare cases involving \letting an active character equal to a brace, so I decided not to include it). So what is this problem? It's just you cannot use \par (or an empty line) in the ⟨*list*⟩. If you need a paragraph break in a ⟨*value*⟩, try \endgraf. If this issue really bothers you, please let me know.

There's only three features of xargs not yet discussed. Since they are all "good" features, you may not need to read the end of the documentation, but since I made the effort to implement it, I want to talk about it. First one is that macros are made in a minimalistic way: If you use \newcommandx to create a macro you could have made with \newcommand, xargs will notice and use \newcommand internally. So, you can allways use \newcommandx without bothering.

Second feature consist in avoiding a possible problem with spaces after a macro whose last argument is optional. If defined in a naive way, such macros would gobble spaces after them when the optional argument is not specified, but macros created with xargs don't, so you don't need to take any special care about spaces.

Last, xargs macros try to behave exactly as standard LaTeX macros do. As far as I know, there are only two exceptions. I allready mentionned the first, which is the problem with \par in default value, due to xkeyval. The other one is that, right now, the current implementation of \CheckCommand has some problems (see PR/3971). I tried not to reproduce them in \CheckCommandx.

3 Implementation

If you are familiar with the way LaTeX 2_ε manages macros with an optional argument, then you will probably not be surprised by the xargs way. Indeed, with LaTeX 2_ε, a command \foo defined with, say \newcommand*\foo[2][bar]{baz} is implemented as the pair:

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

With xargs's \newcommandx, a macro \vect as above is implemented as:

```
\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})
```

This is very similar: the "real" macro is the one whose name begins with a \ and the macro called by the user just checks the protection context and collects the arguments for the internal macro, using the default value if none is given for the optional argument(s). However, the analogy ends here, since in "normal" LaTeX there is only one optional argument, but xargs commands need more information about optional arguments, namely their position, and not only the default values.

This information is stored as a coma-separated list of "actions", each action consisting of either the single command \xargs@put@arg, which denotes a mandatory

argument and makes LaTeX grab it and add it to the list of arguments to be passed to the internal macro, or `\xargs@test@opt` with argument the default value, which denotes an optional argument. In the later case, the presence of the optional argument is checked in a way slightly differing from LaTeX 2_ε's `\@ifnextchar`, then the relevant value added to the arguments list.

3.1 Parser

All this argument grabing job is done with 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. The first part of the code consists of those macros used for the argument grabbing and execution of internal command process.

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}
```

`\@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}}
\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`

You probably noticed that `\xargs@testopt` doesn't use kernel's `\@ifnextchar`. The reason is, I don't want macros to gobble space if their last argument is optional and not specified. Indeed, it would be strange to have spaces after `\vect[0]{x}` gobbled. So the modified version of `\@ifnextchar` below works like kernel's one, except that it remembers how many spaces it gobbles and restitutes them in case the next non-space character isn't a match.

```
\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 execution related macros. Now let's start with 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 prefix `xargs` and 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@prefix{%
  \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. The idea is to collect through xkeyval at most 9 actions numbered 1 to `\xargs@max` (the total number of arguments) of the type seen above, then structure them in a coma-separated list for use in the definition of the user macro. Special care is taken to define simpler macros in the two special cases where all arguments, possibly except the first one, are mandatory (standard LaTeX 2_ε cases).

`\@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 for us. 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 wether 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 just consist in setting the value for some `\if` wich will keep track of the existence of an optional argument other than the first one, and the of the possibly otpional nature of the first. Such information will be usefull when we will have to decide if we use the LaTeX 2_ε 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 meaningfull 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 argument label in\MessageBreak
      optional arguments description%
      }{%
      You can only use non-zero digits as argument labels.\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 actions macros. The first is the coma-separated list of actions allready 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{}%
```

Now 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_ε'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 three times with a modified `\xargs@testopt`.

```
    \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
```

Before we do the definitions, remember to execute `\xargs@drc@hook`, since the next macros will only define the internal macro(s) with a `\DeclareRobustCommandx`, and the hook defines the user macro, with the correct prefix now.

```
    \xargs@drc@hook
```

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.)

```
    \expandafter\endgroup
    \xargs@temp}}
\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 (`\renewcommand` etc) closely resemble their kernel couterpart, since they are mostly wrappers around some call to `\xargs@newc`. There is however an exception, `\CheckCommand`, which I will treat first. Here my way differs from the kernel's one, since current implemetation of `\CheckCommand` in the kernel suffers from two bugs (see PR/3971).

\CheckCommandx We begin as usual detecting the possible star.

```
\@ifdefinable\CheckCommandx{%
  \def\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.

```
\@ifdefinable\xargs@CheckC{%
  \def\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@chech@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.

```
\@ifdefinable\xargs@check@d{%
  \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
```

From now on, there is absolutely nothing to comment on, 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 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 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}
```

\DeclareRobustCommandx
\xargs@DRC

The xargs version of \DeclareRobustCommand, and related internal macro.

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

\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 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!