The \texttt{xargs} package

Manuel Pégourié-Gonnard  
mpg@elzevir.fr

v1.1 (2008/03/22)

Contents

1 Introduction 1
2 Usage 1
3 Implementation 4

\textit{Important note for French users}: a French version of the user documentation is included in the \texttt{xargs-fr.pdf} file.

1 Introduction

Defining commands with an optional argument is easy in \LaTeX{}. There is, however, two limitations: you can make only one argument optional and it must be the first one. The \texttt{xargs} package provide extended variants of \texttt{\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 \texttt{⟨key⟩=⟨value⟩} syntax. For example, the following defines a command with two optional arguments.

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

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

2 Usage

2.1 Basics

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

\begin{verbatim}
\newcommandx \renewcommandx
\newenvironmentx \renewenvironmentx
\providecommandx \DeclareRobustCommandx
\CheckCommandx
\end{verbatim}
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\ \texttt{2ε} manual) to learn the others.

Since these commands all share the same syntax, I’ll always use \texttt{\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 \texttt{\newcommandx}'s complete syntax.

\begin{verbatim}
\newcommandx\langle∗\rangle\{(\langlecommand\rangle)\}\[(\langlenumber\rangle)\]\[(\langlelist\rangle)\] {\langledefinition\rangle}
\end{verbatim}

Everything here is the same as the usual \texttt{\newcommand} syntax, except \texttt{⟨list⟩}. Let’s recall this briefly. The optional \texttt{*} make \LaTeX\ define a “short” macro, that is a macro that won’t accept a paragraph break (\texttt{\par} or an empty line) inside its argument; if you don’t put it, the macro will be long. \texttt{⟨command⟩} is any control sequence, and can but need not be enclosed in braces, as you like. The \texttt{⟨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 \texttt{⟨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 \texttt{⟨number⟩} and \texttt{⟨list⟩} are optionals.

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

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

That’s all for the basics: you are now ready to use \texttt{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 \texttt{xargs}, go on with the next subsections.

2.2 The \texttt{usedefault} key

So, what happens with many successive optional arguments? The default behaviour is that of \LaTeX\’s commands like \texttt{\makebox} or \texttt{\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 \texttt{usedefault} key is for. Just include it in the \texttt{⟨list⟩}, and you can now use \[\] to skip one optional argument (using its default value) and go to the next one.

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

\[
\begin{align*}
\coord{x} & \quad (1_x,\ldots,1_n) \\
\coord{y}[0] & \quad (0_y,\ldots,0_n) \\
\coord{z}[1][m] & \quad (1_z,\ldots,1_m) \\
\coord{t}[0][m] & \quad (0_t,\ldots,0_m)
\end{align*}
\]

Of course, on this simple example, this is merely a matter of taste, but sometimes the \texttt{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 \texttt{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 \texttt{usedefault=⟨flag⟩} to choose this special value. The following example has no other purpose that illustrating this.

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

\[
\begin{align*}
\test[b] & \quad (b,B) \\
\test[\@][b] & \quad (,b) \\
\test[A][b] & \quad (A,b)
\end{align*}
\]

### 2.3 Using a prefix

Standard \texttt{\newcommand} allows you to define so-called “long” (i.e. the arguments may contain a \texttt{\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 \texttt{\long} prefix. Other components of the prefix are \texttt{\global}, \texttt{\outer}, and \eTeX{}’s \texttt{\protected}. There is no way to use them with \texttt{\newcommand}, though \texttt{\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 \TeX{}book and \eTeX{}’s manual.)

With \texttt{xargs}, you can use the \texttt{addprefix} key, \textit{except} for the \texttt{\outer} prefix, which is not and will not be supported (and not used anywhere I know in \texttt{\@}\TeX{}2\e, either). Please note that it \textit{adds} a prefix to the current one, which by default is \texttt{\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.

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

By the way, macros with at least one optional argument are already robust in \texttt{\@}\TeX{}2\e’s meaning of the word, so I don’t know if the \texttt{\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 `T\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 `E\TeX`’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.

```latex
\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 `E\TeX` handles optional arguments. For example, a command `\foo` defined with `\newcommand\foo[2]{baz}` is implemented as the pair:

```latex
\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](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: \texttt{xparse}.

Here the approach is a bit different. Of course, the idea is still to build a parser for the specified syntax, but since \texttt{\newcommandx} user syntax is based on \texttt{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 \LaTeX2ε commands, which adds a few tests. The organisation is as follows.

3.1 Parser  
3.2 Keys  
3.3 Definition  
3.4 User macros

Before we really begin, just a few preliminaries. First, load the \texttt{xkeyval} package for it’s nice key=value syntax.

\begin{verbatim}
\RequirePackage{xkeyval}
\xargs@max
\xargs@temp
\xargs@toksa
\xargs@toksb
\end{verbatim}

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 \texttt{\toks} register because the string used can possibly contain # characters. Sometimes I also use a \texttt{\toks} register instead of a macro just for ease of use (writing less \texttt{\expandafter}s).

\begin{verbatim}
\@ifdefinable\xargs@max{\newcount\xargs@max}
\@ifdefinable\xargs@temp\relax
\@ifdefinable\xargs@toksa{\newtoks\xargs@toksa}
\@ifdefinable\xargs@toksb{\newtoks\xargs@toksb}
\end{verbatim}

3.1 Parser

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

\begin{verbatim}
\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})
\end{verbatim}

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 an optional one. In this case, special care is taken about spaces. Actually, there can be one more action, associated with the \texttt{\usedefault} user key: \texttt{\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 \texttt{\vect} for example. All of this happens inside a group.

\begin{verbatim}
\@protected@testopt@xargs
\newcommand*\@protected@testopt@xargs[1]{%
\if\protect\@typeset@protect
\expandafter\xargs@read
\else
\expandafter\xargs@read
\fi

\end{verbatim}

This first macro closely resembles kernel’s \texttt{@protected@testopt} (similarity in their names is intentional, see \texttt{\CheckCommandx}). It just checks the protection context and call the real argument grabbing macro.
\xargs@read
\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]{%\egingroup\xargs@toksa{#1}\xargs@toksb{#2}\xargs@continue}
\xargs@continue 
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}{\long\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
\newcommand*{\xargs@set@defflag}[1]{\def{\xargs@default@flag{#1}}}
\xargs@put@arg
\newcommand{\xargs@put@arg}[1]{\xargs@toksa{\expandafter{\the\xargs@toksa{#1}}}\xargs@continue}
\xargs@test@opt
\newcommand{\xargs@test@opt}[1]{\xargs@ifnextchar{[}{\xargs@grab@opt{#1}}{\xargs@put@opt{#1}}}
\xargs@put@opt
\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}}
\xargs@put@opt[1]{%\xargs@toksa{\expandafter{\the\xargs@toksa[#1]}}\xargs@continue}
\xargs@test@opt[1]{%\xargs@toksa{\expandafter{\the\xargs@toksa[#1]}}\xargs@continue\long\def{\xargs@grab@opt[#2]}{\toks@{#2}\edef{\xargs@temp{\the\toks@}}\ifx{\xargs@temp\xargs@default@flag\expandafter{\@firstoftwo}}\else\expandafter{\@secondoftwo}}
\xargs@@continue
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
\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 family 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}}%
\@ifdefinable\xargs@makeshort{%
\def\xargs@makeshort#1\long#2{%

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

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.
\begin{verbatim}
\define@key[xargs]{key}{usedefault}{[]}{% 
\xargs@toksa{#1}\edef\xargs@default@flag{\the\xargs@toksa}}
\end{verbatim}

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.
\begin{verbatim}
\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}
\end{verbatim}

The next macro will define the keys. Its first argument is the key's number. The second argument will be discussed later.
\begin{verbatim}
\newcommand*\xargs@def@key[2]{% 
\expandafter@if\@ifdefinable\csname xargs@key@#1\endcsname{% 
\define@key[xargs]{key}{#1}{[#2]}}{% 
\PackageError{xargs}{% 
Illegal argument label in\MessageBreak 
optional arguments description% 
}{}% 
\MessageBreak is higher than the total number \textbackslash the\xargs@max}
\end{verbatim}
of parameters. This can’t be done and your demand will be ignored.}%
\else
  \PackageWarning{xargs}{% Argument #1 was already given a default value. Previous value will be overridden.%}
\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
\newcommand\xargs@setkeys[1]{% 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.
\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% optional arguments description%}
  }%
\fi}
\newif\ifxargs@firstopt@
\newif\ifxargs@otheropt@
\newcommand\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
Now actually define the keys.
\xargs@key@7
\xargs@key@8
\xargs@key@9
\xargs@setkeys
\xargs@check@keys
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{} standard way or \xargs{} custom one to define the macro.
\section*{3.3 Definition}

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

\begin{verbatim}
\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 s{\expandafter##\the\count@}}% 
  \fi}
\end{verbatim}

\texttt{\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.

\begin{verbatim}
\@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 \texttt{\xargs@action@X} macros and the \texttt{\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{\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\epsilon's standard \texttt{\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 \texttt{\xargs@action@1} by expanding it two times with a modified \texttt{\xargs@testopt} (and one more expansion step for the \texttt{\csname} gives 3, hence the $2^3 - 1 = 7$ \texttt{\expandafters}). 
    \ifxargs@otheropt@ \else 
    \ifxargs@firstopt@ 
    \fi 
  \endgroup }
\end{verbatim}

10
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 \@argdef or \xargs’s way. In the \LaTeX{} case, however, we don’t use \@argdef or \xargs since we want to be able to use a prefix (and we have more work done already, too).

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.

\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}}

\subsection{User macros}

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 which 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[#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\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.}%\fi%\xargs@max0%\xargs@process@keys}

The other macros (\renewcommand\etc) closely resemble their kernel homologues, since they are mostly wrappers around some call to \xargs@newc. There is however two exceptions: \CheckCommand and \DeclareRobustCommand. Indeed, the current implementation of \CheckCommand in the kernel suffers from two bugs (see PR/3971) which I’m trying to avoid. For \DeclareRobustCommand, 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.
\CheckCommand
We begin as usual detecting the possible star.
\newcommand{\CheckCommand}{%  \xargs@star@or@long{\xargs@CheckC}}%\@onlypreamble{\CheckCommand} \xargs@CheckC
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 grabbing method used for \newenvironment, ie calling \kernel@ifnextchar explicitly.
\newcommand{\xargs@CheckC}[1]{%  \@testopt{\xargs@check@a#1}0}%\@onlypreamble{\xargs@CheckC} \xargs@check@a
\newcommand{\xargs@check@a}{%  \@ifdefinable{\xargs@check@a}{%  \def{\xargs@check@a#1[#2]}{%  \kernel@ifnextchar[]%  \{\xargs@check@c{#1}{#2}%  \{\xargs@check@c{#1}{#2}}%\@onlypreamble{\xargs@check@a} %\@ifdefinable{\xargs@check@c}{%  \def{\xargs@check@c[#1][#2]}{%  \xargs@check@c{#1}{#2}\{#3}%\@onlypreamble{\xargs@check@c} %\@ifdefinable{\xargs@check@c}{%  \def{\xargs@check@c#1[#2][#3]}{%  \xargs@check@c#1#2\{#3}%\@onlypreamble{\xargs@check@c} %\@ifdefinable{\xargs@check@c}{%  \def{\xargs@check@c#1[#2][#3]}{%  \xargs@check@c#1#2\{#3}%\@onlypreamble{\xargs@check@c} %\@ifdefinable{\xargs@check@c}{%  \def{\xargs@check@c#1[#2][#3]}{%  \xargs@check@c#1#2\{#3}%\@onlypreamble{\xargs@check@c} %\@ifdefinable{\xargs@check@c}{%  \def{\xargs@check@c#1[#2][#3]}{%  \xargs@check@c#1#2\{#3}%\@onlypreamble{\xargs@check@c} %\@ifdefinable{\xargs@check@c}{%  \def{\xargs@check@c#1[#2][#3]}{%  \xargs@check@c#1#2\{#3}%\@onlypreamble{\xargs@check@c} %\@ifdefinable{\xargs@check@c}{%  \def{\xargs@check@c#1[#2][#3]}{%  \xargs@check@c#1#2\{#3}%\@onlypreamble{\xargs@check@c} %\@ifdefinable{\xargs@check@c}{%  \def{\xargs@check@c#1[#2][#3]}{%  \xargs@check@c#1#2\{#3}%\@onlypreamble{\xargs@check@c} %\@ifdefinable{\xargs@check@c}{%  \def{\xargs@check@c#1[#2][#3]}{%  \xargs@check@c#1#2\{#3}%\@onlypreamble{\xargs@check@c} }
\checkC Here comes the major difference with the kernel version. If \reservedA 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 \checkD.

232 \newcommand\checkC[3]{%  
233 \xargs@toksa{#1}  
234 \expandafter\let\csname\string\reservedA\endcsname\relax  
235 \xargs@renewc\reservedA#2{#3}%  
236 \@ifundefined{\string\reservedA} {%  
237 \@ifx#1\reservedA \else  
238 \xargs@check@complain  
239 \fi  
240 }{%  
241 \expandafter\ifx\csname\string#1\expandafter\endcsname\reservedA\expandafter\relax  
242 \else  
243 \xargs@check@complain  
244 \fi
245 \@onlypreamble\checkC

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 \checkD macro only checks if the \meaning of \foo begins with \\protected@testopt (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.

249 \def\xargs@temp{\\protected@testopt\foo \foo}  
250 \\expandafter\meaning\reservedA\xargs@temp\@nil}  
251 \@onlypreamble\checkC

\checkD \checkE

Now, \checkC just pass the \meaning of the command \foo being checked to the already mentioned 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.

253 \expandafter\newcommand\expandafter\xargs@checkD\expandafter\xargs@checkE\expandafter\{%  
254 \expandafter\expandafter\expandafter\xargs@checkE\expandafter\xargs@checkD\expandafter\{%  
255 \expandafter\meaning\expandafter\reservedA\xargs@checkD\xargs@checkE\xargs@checkD\xargs@checkE\xargs@checkD\%  
256 \@onlypreamble\xargs@checkD
257 \@ifdefinable\xargs@checkE{  
258 \expandafter\def\expandafter\xargs@checkE\expandafter\{%  
259 \expandafter\expandafter\expandafter\xargs@checkE\expandafter\{%  
260 \expandafter\ifx\empty\empty \else  
261 \xargs@check@complain  
262 \fi}\}  
263 \@onlypreamble\xargs@checkE

\checkE The complaining macro uses the name saved by \checkC in \xargs@toksa in order to complain about the correct macro.
\newcommand\xargs@check@complain{%
\PackageWarningNoLine{xargs}{Command \the\xargs@toks a has changed.}
\MessageBreak Check if current package is valid}}
\onlypreamble\xargs@check@complain

\DeclareRobustCommand\xargs@DRC{The \texttt{xargs} version of \texttt{\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 \texttt{\xargs@newc}, just as kernel's ones are wrappers around \texttt{\new@command}. So the code below is only copy/paste with search&replace from the kernel code.
\newcommand\xargs@renewc{The \texttt{xargs} version of \texttt{\renewcommand}, and it's related internal macro.}
\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
\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
\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]{}%
{\xargs@new@env{#1}{[#2][#3]}}}
\@ifdefinable\xargs@newenvb{%
\def\xargs@newenvb#1[#2][#3]{}%
\xargs@new@env{#1}{[#2][#3]}}
\newcommand\xargs@new@env[4] {%
\@ifundefined{#1}{}%
\@ifdefinable{#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]
\relax
\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]
\relax
\xargs@newenv{#1}}

\renewenvironment \xargs@renewenv
\xargs@renewenva
\xargs@renewenvb
\xargs@renew@env
\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\expandafter\endcsname
\csname end#1\endcsname}%
\relax
\expandafter\let\csname#1\expandafter\endcsname
\csname end#1\endcsname}%
\xargs@shortpref\expandafter\def\csname end#1[endcsname#1]
\relax
\xargs@renewenv{#1}}

That’s all folks!
Happy \TeX-ing!