ESK:
Encapsulated Sketch for LaTeX*

Tom Kazimiers†

May 5, 2010

Abstract

The ESK package allows to encapsulate Sketch files in LaTeX sources. This is very useful for keeping illustrations in sync with the text. It also frees the user from inventing descriptive names for LaTeX files that fit into the confines of file system conventions.

Copying

*This is `esk.dtx`, version v1.0, revision 1.0, date 2010/05/05.

†e-mail: `tom@voodoo-arts.net`

1 Introduction

When adding illustrations to documents, one faces two bookkeeping problems:

- How to encourage oneself to keep the illustrations in sync with the text, when the document is updated?
- How to make sure that the illustrations appear on the right spot?

For both problems, the best solution is to encapsulate the figures in the LaTeX source:

- It is much easier to remember to update an illustration if one doesn't have to switch files in the editor.
- One does not have to invent illustrative file names, if the computer keeps track of them.

Therefore ESK was written to allow to encapsulate Sketch [1] into LaTeX [2, 3]. It is based on `emp` [4] since it follows a similar approach for METAPOST [5]. Nevertheless, it is arguable that complex Sketch figures may be easier handled in a separate file. That is because it does not directly improve readability for ones source code to have the Sketch code mixed with LaTeX. But that's purely a matter of taste and complexity. To have Sketch code in separate files be included in your build process you could do the following:

1. have your Sketch code in a file, e.g. *nice_scene.sk*
2. include the file *nice_scene.sk.tex* in your document source
3. configure your build in a way to automatically call Sketch on all $*.sk$ files, e.g in a Makefile:
   `for i in ‘ls *.sk‘; do sketch -o "$$i.tex" "$$i"; done`

At least for less complex graphics it is more convenient to use ESK and thus stay consistent more easily.

2 Usage

This chapter describes the different macros and environments provided by the ESK package. The `esk` environment is the one that actually generates printable source code. Depending on what options have been specified with `\eskglobals` and `\eskaddtoglobals` this is TikZ or PSTricks code. If an `esk` environment is encountered, it gets processed the following way:

1. Create a file name for the current figure out of the base name and a running figure number: ⟨*name*⟩.⟨*number*⟩.sk (E. g. pyramid.1.sk)
2. (a) If a file exists that is named like written in 1 but with an additional *.tex* at the end (e.g. pyramid.1.sk.tex) it is treated as a Sketch processed result file. Thus, it is included as a replacement for the environments content.
   (b) If such an item as in 2a is not found a Sketch file with the contents of the environment is saved to a file with the name generated in 1.

In contrast to METAPOST Sketch can't produce different output files out of one source file. This means every Sketch figure has to be put into its own Sketch file. As a consequence one has to process all generated Sketchfiles with Sketchand one can not have one generated file for the whole document. A possible way of managing the build (within a Makefile) of a document then could be:

1. Call `latex` on the document source
2. Process all Sketch files and stick to naming convention:
   `for i in ‘ls *.sk‘; do sketch -o "$$i.tex" "$$i"; done`
3. Call either `latex` and `dvips` or `pdflatex` on the document source to actually see TikZ or PSTricks figures.

2.1 Commands and Environments

`esk` The `esk` environment contains the description of a single figure that will be placed at the location of the environment. The macro has two optional arguments. The first is the name of the figure and defaults to `\jobname`. It is used as the base name for file names. The second one consists of a comma separated list of names previously defined with `\eskdef`. Note that the names have to be put in parentheses (not brackets or braces). Those definitions will be prepended to the Sketch-commands.

```
\begin{esk}[⟨name⟩](⟨def 1⟩,⟨def 2⟩,...)
    ⟨Sketch-commands⟩
\end{esk}
```

`eskdef` The `eskdef` environment acts as a container for Sketch-commands. In contrast to `esk` nothing is written to a file or drawn, but kept in a token list register to recall it later on. Thus, reoccurring patterns can be factored out and used as argument in an `esk` environment. This is useful, because these environments use the `verbatim` package and can therefore *not* be used directly as an argument to other macros.

```
\begin{eskdef}{⟨name⟩}
    ⟨Sketch-commands⟩
\end{eskdef}
```

`\eskprelude` Define a Sketch prelude to be written to the top of every Sketch file. The default is an empty prelude. Keep in mind that verbatim arguments are not allowed.

`\eskaddtoprelude` Add to the Sketch prelude. E. g. `\eskaddtoprelude{def O (0,0,0)}` makes sure the variable O is available in all `esk` environments (and thus in every generated Sketch file). Of cause, this could also be achieved with `Eskimo`.

`\eskglobals` Define global Sketch properties that get passed to the `global {...}` method of Sketch. This defaults to `language tikz`.

`\eskaddtoglobals` Add something to the global parameters of Sketch.

2.2 Examples

For a simple example, let's draw a pyramid in a coordinate system. Since our scene should be a composition of coordinate axes and the geometry, we prepare

definitions for the single parts. In that way the parts will be reusable. First the coordinate system:

```
⟨*sample⟩
\begin{eskdef}{axes}
  def three_axes {
    % draw the axes
    def ax (dx,0,0)
    def ay (0,dy,0)
    def az (0,0,dz)
    line[arrows=<->,line width=.4pt](ax)(O)(ay)
    line[arrows=->,line width=.4pt](O)(az)
    % annotate axes
    special |\path #1 node[left] {$z$}
                   #2 node[below] {$x$}
                   #3 node[above] {$y$};|(az)(ax)(ay)
  }
\end{eskdef}
```

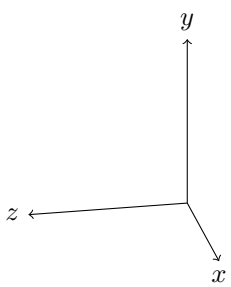
Now the pyramid:

```
\begin{eskdef}{pyramid}
  def pyramid {
    def p0 (0,2)
    def p1 (1.5,0)
    def N 4
    def seg_rot rotate(360 / N, [J])
    % draw the pyramid by rotating a line about the J axis
    sweep[fill=red!20, fill opacity=0.5] { N<>, [[seg_rot]] }
       line[cull=false,fill=blue!20,fill opacity=0.5](p0)(p1)
  }
\end{eskdef}
```

In the definitions some variable have been used that have not been declared so far (`O`, `dx`, `dy`, `dz`, `J`). They have been introduced to make the definitions more versatile. In order to draw the scene their declaration has to be prepended to our output:

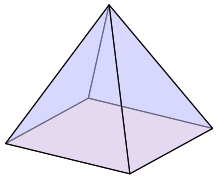
```
\eskaddtoprelude{def O (0,0,0)}
\eskaddtoprelude{def dx 2.3}
\eskaddtoprelude{def dy 2.5}
\eskaddtoprelude{def dz dx}
\eskaddtoprelude{def J [0,1,0]}
```

Now the previously created definitions can be used to do the actual drawing. First, the coordinate system on its own:



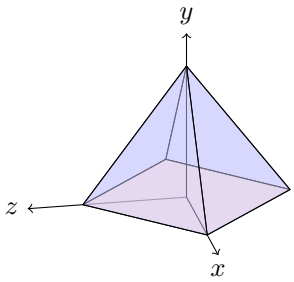
```
\begin{esk}(axes)
  def scene {
    {three_axes}
  }
  put { view((10,4,2)) } {scene}
\end{esk}
```

Now the pyramid (note, the transparency effect will only be visible in a pdf):



```
\begin{esk}(pyramid)
  def scene {
    {pyramid}
  }
  put { view((10,4,2)) } {scene}
\end{esk}
```

Finally both:




```
\begin{esk}(axes,pyramid)
  def scene {
    {pyramid}
    {three_axes}
  }
  put { view((10,4,2)) } {scene}
\end{esk}
⟨/sample⟩
```

With permission of Kjell Magne Fauske, the source code for this example scene has been taken from [6].

References

[1] Eugene K. Ressler, Sketch, 2010/04/24, http://www.frontiernet.net/ eugene.ressler/

[2] Leslie Lamport, *LaTeX — A Documentation Preparation System*, Addison-Wesley, Reading MA, 1985.

[3] Donald E. Knuth, *The TEXbook*, Addison-Wesley, 1996

[4] Thorsten Ohl, `emp`,Encapsulated MetaPost, 1997, available from CTAN

[5] John D. Hobby, *A User's Manual for METAPOST*, Computer Science Report #162, AT&T Bell Laboratories, April 1992.

[6] Kjell Magnus Fauske, An introduction to Sketch, 2010/04/24, http://www.fauskes.net/nb/introduction-to-sketch/

Distribution

ESK is available by anonymous internet ftp from any of the Comprehensive TEX Archive Network (CTAN) hosts

`ftp.tex.ac.uk`, `ftp.dante.de`

in the directory

`macros/latex/contrib/esk`

It is available from host

`www.voodoo-arts.net`

in the directory

`pub/tex/esk`

A work in progress under git control is available from

`http://github.com/tomka/esk`

3 Implementation

This project is greatly inspired and based on EMP. EMP is a LaTeX package to provide a convenient way to work with metapost files and code from inside LaTeX documents.
It's is good practice to identify this version of the document style option. We do this by parsing an RCS `Id` string and storing the result in the conventional TEX control sequences. The parsing macro is only visible locally (within the surrounding scope), but generated control sequences like `\filename` are defined globally (due to the use of `\gdef`):

```
⟨*style⟩
\def\fileversion{v1.0}
{\def\RCS#1#2\endRCS{%
  % is the first parameter a "$%?
  \ifx$#1%
    \@RCS $#2 \endRCS
  \else
    \@RCS $*: #1#2$ \endRCS
  \fi}%
 \def\@RCS $#1: #2,v #3 #4 #5 #6 #7$ \endRCS{%
   % global defines (independent of current scope) of file attributes
```

```
    \gdef\filename{#2}%
    \gdef\filerevision{#3}%
    \gdef\filedate{#4}%
    \gdef\filemaintainer{#6}}%
\RCS $Id: esk.dtx,v 1.0 2010/05/05 01:23:42 kazimiers Exp $ \endRCS}%
```

Make clear what LaTeX version is needed:

```
\NeedsTeXFormat{LaTeX2e}
```

And now the standard procedure:

```
\ProvidesPackage{esk}[\filedate\space\fileversion\space
  Encapsulated Sketch LaTeX Package (\filemaintainer)]
```

We do not declare options for this package, so we do not need to invoke processing for found ones. Some other packages needed by ESK, partly of a minimum version, get specified.

```
\RequirePackage{verbatim}
\RequirePackage{kvsetkeys}[2007/09/29]
```

The characters "%", "{" and "}" are somewhat special to TEX. More precisely are they used for comments and grouping respectively. Sketch uses them as well, for the same purposes. To allow convenient code generation by using macros to produce the symbols, three macros get defined:

`\p@rcent` That macro is used for creating comments. The % sign is locally defined as a common letter (category code 11) and create a global macro using it. The `@` in the name of the control sequence makes it only visible from inside the package[1].

```
{\catcode`\%=11\gdef\p@rcent{%}}
```

`\lc@rly` `\rc@rly` Curly braces are used for scope and group definitions in Sketch. Just like with the `\p@rcent` macro, we need to make them a common letter. Unfortunately we need the curly braces to define a local scope for TEX. To come around this the characters > and < are locally defined to be grouping characters (category code 1). To allow normal scope closing we finally make the curly braces grouping characters again.

```
{\catcode`\>=1  \catcode`\<=2
 \catcode`\{=11 \catcode`\}=11
 \gdef\lc@rly>{<
 \gdef\rc@rly>}<
 \catcode`\{=1  \catcode`\}=2
}
```

`\eskwrite` Define a macro to write the contents of its first argument to a file. If the Boolean toggle `@eskio` is set to true, the passed argument is written[2] out to the file referenced in `@outesk`. Normally TEX does the actual writing during a `\shipout` operation, but we force it to do it immediately[3]. All directly following spaces on the input will be eaten[4].

```
\def\eskwrite#1{%
  \if@eskio
    \immediate\write\@outesk{#1}%
```

[1] see http://de.wikibooks.org/wiki/LaTeX-Wrterbuch:_

[2] see http://www.tug.org/utilities/plain/cseq.html#write-rp

[3] see http://www.tug.org/utilities/plain/cseq.html#immediate-rp

[4] see http://en.wikibooks.org/wiki/TeX/ignorespaces

```
  \fi
  \ignorespaces}
```

`\eskwritetoken` If a token list register should be put into a file, this macro should be used. It will expand the token variable to its current contents.

```
\def\eskwritetoken#1{
  \eskwrite{\the#1}}
```

Next a new private Boolean toggle is defined. It is used to reflect if file writing is enabled and set it to true.

```
\newif\if@eskio
\@eskiotrue
```

The next free output file handle will be referenced by the private macro `\@outesk`. At this point no file is opened, but just an output channel defined[5]. An example file open could now look like: `\openout\@outesk=TEXTFILE.TXT`.

```
\newwrite\@outesk
```

`\eskfile` This environment encloses each Sketch input file. The single optional argument gives the name of the file and defaults to `\jobname`. This will probably not be used explicitly when defining esk figures. It is invoked automatically with an appropriate name for a figure. The macro `\theeskfile` gets locally defined and stores the base name for a file.

```
\newcommand{\eskfile}[1][\jobname]{%
  \def\theeskfile{#1}%
```

Open the Sketch file. If output is enabled, check if we're running under AMS-LaTeX and if that is the case turn off I/O during the first pass over equation environments. This is done by looking for `\ifmeasuring@` of AMS-LaTeX and, if found, replacing all `\if@eskio` checks with it.

```
  \if@eskio
    \@ifundefined{ifmeasuring@}%
      {}%
      {\def\if@eskio{\ifmeasuring@\else}}%
```

A a new output file is linked to our `\@outesk` file number. The name of that file is the content of `\theeskfile` with extension *.sk*. Afterwards a start comment is written to the new file.

```
    \immediate\openout\@outesk=\theeskfile.sk\relax
    \eskwrite{\p@rcent\p@rcent\p@rcent\space \theeskfile.sk -- %
              do not edit, generated automatically by \jobname.tex}%
```

The `esk@prelude` token list register stores a prelude that should be put at the beginning of the new file. If the register is empty, the expansion of it (`\the\esk@prelude`) will be empty. This means the `\ifx` condition is met, because the actual check is now if `*` equals `*`. Hence the `\else` branch will *not* be called. If the token register is not empty (and does not start with an asterisk) the `\else` branch is used..

```
    \expandafter\ifx\expandafter*\the\esk@prelude*\else
      \eskwrite{\the\esk@prelude}%
    \fi
  \fi}
```

[5] see *A TEX primer for scientists* by Stanley A. Sawyer,Steven George Krantz on p. 283

Define `\theeskfile`, later redefined with the name of the currently opened file, to be `\relax` (i.e. stop reading tokens). This should be the value if no file is opened.

```
\let\theeskfile\relax
```

Define a new counter `\eskfig` to count the single esk figures. It is initialized with 0.

```
\newcounter{eskfig}
```

Let TEX create a new token list register alias `\esk@prelude`. It stores an optional prelude for the files written out. If the indirect alias creation `\newtoks` is used, TEX selects a free register and hides the technical detail from us.

```
\newtoks\esk@prelude
```

`\eskprelude` `\eskaddtoprelude`

Define a public `\eskprelude` macro that replaces the contents of the internal token list register `\esk@prelude` with the argument passed.

```
\def\eskprelude#1{\esk@prelude={#1}}
```

Define a public macro that appends its argument to the internal `\esk@prelude` token list register. The text is added on a new line. This is accomplished by using `^^J`, a replacement ASCII representation for LF (line feed)[6].

```
\def\eskaddtoprelude#1{\esk@prelude=\expandafter{\the\esk@prelude^^J#1}}
```

The token list register storing the global settings of Sketch is called `\esk@globals` and defaults to *language tikz*.

```
\newtoks\esk@globals
\esk@globals={language tikz}
```

`\eskglobals` `\eskaddtoglobals`

The macros `\eskglobals` and `\eskaddtoglobals` are there to set and modify the private token list register `\esk@globals`. With them one has control over the general settings of Sketch. On adding, new settings will be delimited by a comma.

```
\def\eskglobals#1{\esk@globals={#1}}
\def\eskaddtoglobals#1{\esk@globals=\expandafter{\the\esk@globals,#1}}
```

`\endeskfile`

And here is how the `empfile` environment is closed. If there are global settings they are written out. The last line of the generated file will be an end statement in form of a comment. followed by a line break. As a convention the macro keeping the base name of the file, `\theeskfile`, is set to `\relax`. That indicates that no file is open. To make that true, the currently opened file (if any) is finally closed.

```
\def\endeskfile{%
  \expandafter\ifx\expandafter*\the\esk@globals*\else
    \eskwrite{global \lc@rly\the\esk@globals\rc@rly }%
  \fi
  \eskwrite{\p@rcent\p@rcent\p@rcent\space the end.^^J}%
  \let\theeskfile\relax
  \if@eskio
    \immediate\closeout\@outesk
  \fi}
```

[6] e.g. see: http://www.torsten-horn.de/techdocs/ascii.htm

`\esk` The `esk` environment encloses Sketch code that will be put into a file for being later processed by Sketch. First it (re-)defines the macro `\esk@@name` with the environments argument. That argument is used as the base name for the corresponding file and defaults to `\jobname`. Then the internal macro `\esk@` produces a single esk graphic.

```
\newcommand{\esk}[1][\jobname]{%
  \def\esk@@name{#1}%
  \esk@}
```

Since the `esk` environment allows two optional parameters and only one can have brackets, the second parameter is surrounded by parentheses. A macro for an opening parenthesis is defined:

```
\let\leftparanthesis=(
```

`\esk@` The private `\esk@` macro stores the immediately following token in the macro `\next` and invokes `\esk@impl`.

```
\def\esk@{
  \futurelet\next\esk@impl}
```

`\esk@impl` Now that the following token is known in `\next` it is checked if the second optional argument got passed. This is done by testing if the next token is an opening parenthesis and depending on its occurrence `\esk@impl@Arg` or `\esk@impl@NoArg` is invoked. Since we want to work with the content of the environment verbatim, we have to get rid of `\else` and `\fi` in the input stream. This can be achieved by just expanding them before calling the verbatim handling macros with `\expandafter`.

```
\def\esk@impl{%
  \ifx\next\leftparanthesis
    \expandafter\esk@impl@Arg
  \else
    \expandafter\esk@impl@NoArg
  \fi}
```

`\esk@impl@NoArg` `\esk@impl@Arg` The macro `\esk@impl@NoArg` just calls `\esk@impl@Arg` with an empty argument. It is mainly there for readability.

```
\def\esk@impl@NoArg{\esk@impl@Arg()}
```

The following macro, `\esk@impl@Arg`, expects one argument surrounded by parentheses, namely a list of `eskdef` names. It makes sure some preconditions are met by invoking `\esk@start`. Afterwards `\esk@includegraphics` checks if a Sketch file should be generated or a LaTeX file be included. Finally the argument is parsed as a comma separated list to call `\esk@def@processor` for each element found and the actual Sketch code verbatim processing is started with `\esk@cmds`. As the verbatim line processing macro name "eskwritetoken" is passed as an argument.

```
\def\esk@impl@Arg(#1){%
  \esk@start%
  \esk@includegraphics{\theeskfile}%
  \comma@parse{#1}{\esk@def@processor}%
  \esk@cmds{eskwritetoken}}
```

esk@def@processor

The macro `\esk@def@processor` gets expanded for every element of the second optional argument of the `esk` environment. Here every `eskdef` name of that list will be included in the current file by invoking `\eskuse` for it.

```
\def\esk@def@processor#1{
  \esk@use{#1}}
```

\esk@start

A macro for preparing for a new Sketch figure.

```
\def\esk@start{%
```

We can't use `\stepcounter` because of the `amstext` option of AMS-LaTeX disables it sometimes. Instead we globally advance the eskfig counter manually by one. Afterwards we call `\esk@checkfile` to make sure a file is open. Finally we invoke `\esk@@def` with our previously defined temporary esk file name to generate new `\theeskfile` and `\theeskfig` alias macros for the current figure.

```
  \global\expandafter\advance\csname c@eskfig\endcsname \@ne
  \esk@checkfile
  \esk@@def{\esk@@name}}
```

\esk@checkfile

Make sure that a Sketch file is open, otherwise *really* obscure error messages are possible. This is done by checking if `\theeskfile` is the same as `\relax` (as defined during initialization and file closing). If so, try to open a file (again) and do the test again. If it still fails print and produce an error.

```
\def\esk@checkfile{%
  \ifx\theeskfile\relax
    \eskfile[\esk@@name.\arabic{eskfig}]
  \fi
  \ifx\theeskfile\relax
    \errmessage{Could not open file "\esk@@name.\arabic{eskfig}.sk"!}
  \fi}
```

\esk@includegraphics

If a file having *.sk.tex* added to the base name exists this macro will include it. To start a new paragraph if we are in vertical mode and switch to horizontal mode `\leaveemode` is called at the beginning. Then, if the file exists, pass its name as an argument to `\input` (which expects the file to end with *.tex*). If there is no such file a message is typed out to tell the user that the sketch files might need the actual processing.

```
\def\esk@includegraphics#1{%
  \leavevmode
  \IfFileExists{#1.sk.tex}%
    {\input{#1.sk.tex}}%
    {\typeout{%
      esk: File #1.sk.tex\space not found:^^J%
      esk: Process #1.sk with Sketch (-o #1.sk.tex) and then %
           reprocess this file.}}}
```

\esk@cmds

The macro `\esk@cmds` gets the `esk` environments content by using the verbatim package. Each line is processed by a macro which name is passed as an argument. That is done to reuse the macro for `esk` and `eskdef` environments. The macros in use for the line processing are `eskwrite` and `esk@def@verb@proc` respectively. Due to the use of `\begingroup` TeX enters a group that has to be terminated by `\endgroup` and not by }.

```
\newcommand{\esk@cmds}[1]{%
  \begingroup
```

The macros `\@bsphack` ... `\@esphack` are used by macros such as `\index` and `\begin{@float}` ... `\end{@float}` that want to be invisible – i.e. not leave any extra space when used in the middle of text. Such a macro should begin with `\@bsphack` and end with `\@esphack`. The macro in question should not create any text, nor change the mode.

```
\@bsphack
```

The next thing to do is to defuse LaTeX' special characters: `\dospeciels` expands to a list of special characters of the form `\do\ \do\\ \do\{ \do\}`.... If one (re-)defines the "`\do`" macro one can execute a macro on all of them. In our case we define `\do` to be `\@makeother`. That assigns category code 12 (non-letter) to all special characters, thus they get normal characters without any special meaning. Due to the environment those changes are local.

```
\let\do\@makeother\dospecials
```

`^^M` is the ASCII representation of CR (carriage return). With the following line we make it an active character. Thus a macro with the name `^^M` can now be defined.

```
\catcode`\^^M\active
```

Since we use the verbatim package `\verbatim@processline` is called after each line. We redefine it to do what we would like it to do: Process the line with the macro with the name passed as argument.. The current line is available in `\verbatim@line`, a token register [7].

```
\def\verbatim@processline{\csname#1\endcsname{\verbatim@line}}%
```

Enter the real verbatim mode. From here on *all* characters have lost their special meaning (if they had any).

```
\verbatim@start}%
```

`\endesk@cmds` To end the invisible environment and the group started by `\esk@cmds`, this macro has to be used.

```
\def\endesk@cmds{%
    \@esphack
  \endgroup}
```

`\endesk` This macro triggers the termination of the verbatim reading and closes the current file.

```
\def\endesk{%
  \endesk@cmds
  \endeskfile}
```

`\eskdef` An `eskdef` environment allows to store blocks of Sketch code in token list registers for using them in `esk` environments. The macro has one parameter, the name of the definition. First a private and local name for the new block is defined. Then `\esk@def` checks if the name is already there and does the rest.

```
\newcommand{\eskdef}[1]{%
  %% Define a new name
  \def\esk@@def@name{esk@def:#1}%
  \esk@def}
```

[7] see: Latex hacks by Anselm Lingnau, p. 43

`\esk@def` The `\esk@def` macro relies on `\esk@@def@name` being defined previously. At the beginning it checks if that name is already registered by looking for a control sequence with the defined name. If so, an error message is produced.

```
\def\esk@def{%
  \expandafter\ifcsname\esk@@def@name\endcsname
    \errmessage{"\esk@@def@name" is already defined!}
  \fi
```

If a new `eskdef` name is given a new token list register is created and named like the expansion of `\esk@@def@name`. It is not necessary to tell TEX that the new register will be global, because *new...* tokens act always globally. Unfortunately `\newtoks` is an `\outer` macro and we use the wrapper `\tok@newtoks` (see below) to call it.

```
  \expandafter\tok@newtoks\csname\esk@@def@name\endcsname
```

Create or override a global definition `\esk@@def@reg` containing our new token register. Unfortunately, this works only with a global definition. Afterwards the verbatim reading of the environment is started with a different verbatim line processor as before. Finally the macro ends with removing the previously defined alias for the new token list register.

```
  \global\edef\esk@@def@reg{\csname\esk@@def@name\endcsname}
  \esk@cmds{esk@def@verb@proc}}
  \global\def\esk@@def@reg{}
```

`\esk@def@verb@proc` The verbatim line processor for the `eskdef` environment first creates a local alias for the new token list register. This is done to make the code more readable.

```
\def\esk@def@verb@proc#1{%
  \expandafter\let\expandafter\token@reg\esk@@def@reg
```

If the token list register is empty it is filled with the current verbatim line.

```
  \expandafter\ifx\expandafter*\the\token@reg*
    \global\esk@@def@reg=\expandafter{\the#1}
```

If not, the verbatim line is appended on a new line.

```
  \else
    \global\esk@@def@reg=\expandafter{%
      \the\expandafter\token@reg\expandafter^^J\the#1}
  \fi}
```

`\endeskdef` On ending an `eskdef` environment `\endeskdef` is expanded. Here, its only purpose is to invoke the macro ending the verbatim input environment.

```
\def\endeskdef{
  \endesk@cmds}
```

`\esk@@def` A macro which takes a file name as argument to globally define new macros `\esk@k:f:<arg>` and `\esk@k:c:<arg>` which expand to `\theeskfile` and `\theeskfig` respectively.

```
\def\esk@@def#1{%
  \global\e@namedef{esk@k:f:#1}{\theeskfile}%
  \global\e@namedef{esk@k:c:#1}{\theeskfig}}
```

`\e@namedef` A macro which defines a new macro with the name of the argument. This is done in use of `\expandafter` and `\csname`...`\endcsname`. The new macro

expands to the following group, i.e. the new macros body. Due to the use of `\edef` this happens dynamically.

```
\def\e@namedef#1{%
  \expandafter\edef\csname #1\endcsname}
```

`\esk@use` The `\esk@use` macro appends an ESK Sketch code definition defined by `\eskdef`, into the currently defined file. After creating an alias macro for the argument passed is is made sure that the definition actually exists. If not an error message is produced. If there is a token list register, named like passed as argument, a short describing comment is written. Further, the register is written as token to the file. The file writing is finished with a new line.

```
\def\esk@use#1{%
  \def\esk@@def@name{esk@def:#1}%
  \expandafter\ifcsname\esk@@def@name\endcsname
    \eskwrite{\p@rcent\p@rcent\space included definition: #1}%
    \expandafter\eskwritetoken\expandafter{%
     \expandafter\csname\esk@@def@name\endcsname}
    \eskwrite{^^J}
  \else
    \errmessage{esk: "#1" is undefined!}
  \fi
}
```

`\tok@newtoks` Since `\newtoks` is an `\outer` macro, it is not allowed in definitions. Because we are in the need of creating token list registers on the fly, we define a wrapper. It lets TEX construct the `\newtoks` call:

```
\def\tok@newtoks{
  \csname newtoks\endcsname}
```

`\futurespacelet` A special version of the `\futurelet` macro. It is taken from Donald. E. Knuths *TEXbook* and behaves like `\futurelet`, but ignores spaces.

```
\def\futurenospacelet#1{\def\cs{#1}%
  \afterassignment\stepone\let\nexttoken= }
```

Let `\stoken` be a space token:

```
\def\\{\let\stoken= } \\
```

And define the stepwise look-ahead macros:

```
\def\stepone{\expandafter\futurelet\cs\steptwo}
\def\steptwo{\expandafter\ifx\cs\stoken\let\next=\stepthree
  \else\let\next=\nexttoken\fi \next}
\def\stepthree{\afterassignment\stepone\let\next= }
⟨/style⟩
```

Index

Numbers written in italic refer to the page where the corresponding entry is described; numbers underlined refer to the code line of the definition; numbers in roman refer to the code lines where the entry is used.

Symbols

Change History

v1.0

General: Version 1.0 Release 6

A Driver File

221 ⟨*driver⟩

```
\documentclass[a4paper]{article}
\usepackage{doc}
\usepackage{amsmath}
```

The logos would come out much nicer if `mflogo` would support some more letters (i.e. k and K). We don't have that and so we define the logos the following way:

```
\def\ESK{\textsf{ESK}}%
\def\SK{\textsf{Sketch}}%
\def\MP{\textsf{META}\-\textsf{POST}}%

```

Protect against certain outdated versions of the `kvsetkeys` package:

```
\usepackage{kvsetkeys}[2007/09/29]
\usepackage{tikz}
\usepackage{esk}

\setlength{\parindent}{0pt}
\def\manindex#1{\SortIndex{#1}{#1}}
⟨manual⟩\OnlyDescription
\EnableCrossrefs
\RecordChanges
\CodelineIndex
\DoNotIndex{\def,\gdef,\long,\let,\begin,\end,\if,\ifx,\else,\fi}
\DoNotIndex{\immediate,\write,\newwrite,\openout,\closeout,\typeout}
\DoNotIndex{\font,\jobname,\documentclass,\char,\catcode,\ }
\DoNotIndex{\CodelineIndex,\DocInput,\DoNotIndex,\EnableCrossrefs}
\DoNotIndex{\filedate,\filename,\fileversion,\logo,\manfnt}
\DoNotIndex{\NeedsTeXFormat,\ProvidesPackage,\RecordChanges,\space}
\DoNotIndex{\begingroup,\csname,\edef,\endcsname,\expandafter}
\DoNotIndex{\usepackage,\@ifundefined,\ignorespaces,\item,\leavevmode}
\DoNotIndex{\newcounter,\newif,\par,\parindent}
\DoNotIndex{\relax,\setcounter,\stepcounter,\the,\advance}
\DoNotIndex{\CurrentOption,\DeclareOption,\documentstyle}
\DoNotIndex{\endgroup,\global,\hfuzz,\LaTeX,\LaTeXe}
\DoNotIndex{\macrocode,\OnlyDescription,\PassOptionsToPackage}
\DoNotIndex{\ProcessOptions,\RequirePackage,\string,\textsf,\unitlength}
\DoNotIndex{\@bsphack,\@esphack,\@nameuse,\@ne,\active,\do,\dospecials}
\DoNotIndex{\errhelp,\errmessage,\ifcase,\IfFileExists,\includegraphics}
\DoNotIndex{\manindex,\SortIndex,\newcommand,\newtoks,\or,\origmacrocode}
\DoNotIndex{\alpha,\displaystyle,\frac,\sin,\texttt}
```

Cut the line breaking some slack for macro code which might contain long lines (it doesn't really hurt if they stick out a bit).

```
\let\origmacrocode\macrocode
\def\macrocode{\hfuzz 5em\origmacrocode}
\begin{document}
  \DocInput{esk.dtx}
\end{document}
⟨/driver⟩
```