The \texttt{rtkinenc} package

Lars Hellström∗

2000/01/24

Abstract

The \texttt{rtkinenc} package is functionally similar to the standard \LaTeX{} package \texttt{inputenc}—both set up active characters so that input character outside 7-bit ASCII are converted to the corresponding \LaTeX{} commands. Names of commands in \texttt{rtkinenc} have been selected so that it can read \texttt{inputenc} encoding definition files, and the aim is that \texttt{rtkinenc} should be backwards compatible with \texttt{inputenc}. \texttt{rtkinenc} is not a new version of \texttt{inputenc} though, nor is it part of standard \LaTeX{}.

The main difference between the two packages lies in the view on the input. With \texttt{inputenc}, the non-ASCII characters in the input are considered as shorthand representations of the “true” document contents (usually one or several commands), and the conversion is therefore irreversible. With \texttt{rtkinenc} the input file itself is considered the true document, and the conversion of non-ASCII characters to \LaTeX{} commands is merely done because it is the first step on the preferred route to typeset output. If the command is for an unavailable text symbol, then it is possible to return to the raw input and try some fallback method of typesetting the character.

The \texttt{inputenc} approach is natural for normal \LaTeX{} documents, but the \texttt{rtkinenc} approach is advantageous for program source code, where the true meaning of a file is not defined by \TeX{}, but by the compiler, interpreter, or whatever.

1 Implementation

\begin{verbatim}{*\pkg}\\NeedsTeXFormat{LaTeX2e}[1995/12/01]\ProvidesPackage{rtkinenc}[2000/01/24 v1.0 Rethinking input encoding package]\end{verbatim}

1.1 Basic mechanisms

\begin{verbatim}{*\RIE@last@char}\RIE@last@char\end{verbatim} is a \texttt{\count} register for storing the code of the raw character currently being typeset. It is minus one if no raw character is being typeset. This register should always be set globally.

∗E-mail: Lars.Hellstrom@math.umu.se
The syntax for the \RieC command is
\RieC\{⟨code⟩\}\{⟨definition⟩\}

Here \textit{⟨code⟩} is assumed to be a sequence of digits giving some raw character code, and \textit{⟨definition⟩} is assumed to be robust \LaTeX code for typesetting (some representation of) that raw character. \RieC is used as \texttt{inputenc}'s \TeX command—an \RieC with arguments form the definition of an active character—but it carries extra information in the \textit{⟨code⟩} argument.

Depending on what the control sequences \protect and \Rie@char are there are three different things this can do.

- It can execute the \textit{⟨definition⟩} straight off. This happens when \protect is \@typeset@protect and \Rie@char is \Rie@text@char.

- It can set \Rie@last@char to \textit{⟨code⟩}, then execute the \textit{⟨definition⟩}, and finally set \Rie@last@char to minus one. This happens when \protect is \@typeset@protect and \Rie@char is \Rie@code@char.

  Setting \Rie@last@char like that has the side-effect of prohibiting kerns and ligatures between the \textit{⟨definition⟩} and what surrounds it. Therefore it is inappropriate to have \Rie@char equal to \Rie@code@char in many types of text, and by default it will not be.

- It can expand to itself.\footnote{Unless some command in the \textit{⟨definition⟩} was defined using \texttt{DeclareRobustCommand}, in which case it is the one level expansion of that command that will expand to itself. It all works out right in the end anyway.} This happens when \protect is not \@typeset@protect, e.g. when writing to a file.

\RieC cannot be defined using \texttt{DeclareRobustCommand}, since that would insert \texttt{protect}s that would prohibit normal kerning and ligaturing. Therefore the command robustness is maintained through an ad hoc definition (the \texttt{def}). The reason for starting with an \texttt{\@ifundefined} is that the user shouldn’t get less info about a redefinition than he/she would with a \texttt{DeclareRobustCommand}.

\begin{verbatim}
7 \@ifundefined{RieC}{\%
8 \PackageError{rtkinenc}{Redefining \protect\RieC}{\%
9 I had expected \protect\RieC\space{space to be undefined.\MessageBreak
10 Since it wasn’t, there’s a chance I have\MessageBreak
11 broken something.\MessageBreak\@ehc
12 }
13 }
14 \def\RieC{%
15 \ifx \protect\@typeset@protect
16 \expandafter\Rie@char
17 \else
18 \noexpand\RieC
19 \fi
\end{verbatim}
RieBC The syntax for the \RieBC command is
\RieBC{⟨code⟩}{⟨text definition⟩}{⟨math definition⟩}

Here \textit{(code)} is assumed to be a sequence of digits giving some raw character code, whereas \textit{(text definition)} and \textit{(math definition)} are assumed to be robust \LaTeX code for typesetting (some representation of) that raw character in text and math mode, respectively. \RieBC is used like \RieC, but offers the possibility of alternative definitions for increased typesetting quality.

The only way to know whether a particular text command could be rendered as intended or not is to seize control of the standard \LaTeX command \TextSymbolUnavailable. This command will then be given a new definition which selects whether some raw character fallback macro or the standard \LaTeX error message should be given. The raw character fallback macro can be set using the \SetUnavailableAction command.

Before seizing control of \TextSymbolUnavailable, one must make sure that it does not have its autoload definition. Then the \LaTeX definition is saved away in \@@TextSymbolUnavailable.

\begin{verbatim}
def\@tempa{\@autoerr\TextSymbolUnavailable}
\ifx \@tempa\TextSymbolUnavailable
\@autoerr\relax
\fi
\let\@@TextSymbolUnavailable=\TextSymbolUnavailable
\end{verbatim}

Then the new definition is given. It is pretty straightforward.

\begin{verbatim}
def\TextSymbolUnavailable{\ifnum \m@ne<\RIE@last@char
\expandafter\RIE@symbol@unavailable \expandafter\RIE@last@char
\else
\expandafter\@@TextSymbolUnavailable
\fi}
\PackageInfo{rtkinenc}{Redefining \protect\TextSymbolUnavailable}
\end{verbatim}

The \SetUnavailableAction command locally defines the \RIE@symbol@unavailable macro, which is executed instead of the standard \LaTeX \TextSymbolUnavailable when the text symbol in case was the representation of a raw input character. \SetUnavailableAction is used as

\begin{verbatim}
\SetUnavailableAction{⟨definition⟩}
\end{verbatim}

where ⟨definition⟩ is like the last argument of \newcommand. The ⟨definition⟩ may contain #1 and #2, where #1 will contain the current raw character number and #2 will contain the text command for which no definition could be found.

\begin{verbatim}
\newcommand\SetUnavailableAction{\def\RIE@symbol@unavailable{⟨definition⟩}}
\end{verbatim}

The default fallback action is to call \@@TextSymbolUnavailable, but most of the definition deals with recognizing and handling the case that the input character hasn’t been declared, rather than that its definition is not available. Normally, this shouldn’t be used at all; instead the user should have installed another fallback.

\begin{verbatim}
def\SetUnavailableAction{%
\ifx #2\relax
\begingroup
\let\RIE@char=\RIE@text@char
\RIE@undefined{#1}%
\endgroup
\else
\@@TextSymbolUnavailable{#2}%
\fi}
\end{verbatim}

\RIE@undefined The \RIE@undefined macro is used in the definition of input characters which are not defined in the current input encoding. It takes one argument, namely the code for the character in question. In text mode, this is an error. In code mode, it is passed on to the unavailable-action macro \RIE@symbol@unavailable.  

4
The default value of \RIE@symbol@unavailable recognizes the \relax passed as second argument below as a flag that in reality it’s the input character that hasn’t been defined.

75 \def\RIE@undefined#1{%
76 \ifx \RIE@char\RIE@text@char
77 \PackageError{rtkinenc}{%
78 Input character \texttt{#1} is undefined\MessageBreak
79 in inputencoding \RIE@encoding}@eha
80 \else
81 \RIE@symbol@unavailable(#1)\relax
82 \fi
83 }

The \InputModeCode and \InputModeText commands switch to the ‘code’ and ‘text’ respectively modes for the rtkinenc package. They both act locally, since it is often convenient to have the previous mode restored at the end of an environment.

The \IfInputModeCode command can be used to test which mode the rtkinenc package currently is in. Is is used as

\IfInputModeCode{(code)}{(text)}

This will expand to \texttt{(code)} or \texttt{(text)} when the current mode is code mode or text mode, respectively.

84 \newcommand\InputModeCode{\let\RIE@char=\RIE@code@char}
85 \newcommand\InputModeText{\let\RIE@char=\RIE@text@char}
86 \newcommand\IfInputModeCode{\ifx \RIE@char\RIE@code@char
87 \expandafter\@firstoftwo
88 \else
89 \expandafter\@secondoftwo
90 \fi
91 }

1.2 Setting the input encoding

The first two commands are identical; the duplication is only for being compatible with inputenc. The reason that there are two different commands in inputenc is that \DeclareInputMath saves a little memory by not taking special measures to see to that a space (if there is any) that follows the input character is not gobbled in case it is written to a temporary file and subsequently read back. Saving that small amount of memory is not the aim for rtkinenc, which is instead using up even more memory by including the character code in the definition.

These two commands are used as

\DeclareInputText{(slot)}{(definition)}
\DeclareInputMath{(slot)}{(definition)}

This makes the active character whose character code is \texttt{(slot)} a parameterless macro whose expansion is
\RieC{(slot (sanitized))}\{definition}\}

(slot (sanitized)) has the same numerical value as (slot), but consists only of decimal digits. \{definition\} is the same in both cases.

\DeclareInputBoth The \DeclareInputBoth command is similar to \DeclareInputText and \DeclareInputMath commands, but it offers an extra feature—the text and math definitions of a character can be different. \DeclareInputBoth is used as

\DeclareInputBoth{(slot)}\{text\}\{math\}

where \{text\} and \{math\} are the text and math mode definitions respectively.

\DeclareInputBoth
The \inputencoding command sets the current input encoding to be the one specified in its only argument. First all characters are set to be active and defined as \RIE@undefined{⟨slot⟩}, except for Null ("\^\@"), tab ("\^\I"), line feed ("\^\J"), form feed ("\^\L"), carriage return ("\^\M"), and space– which are left as they were. Then #1.def is inputed; this file is expected to contain all the \DeclareInput... commands that are needed for the wanted input encoding.

Besides that, \inputencoding also does a check to see that the file #1.def actually did execute some \DeclareInput... command. Since it wouldn’t at all surprise me if someone likes to tinker with this test, it is done exactly as in inputenc.

\inputencoding should not be used in horizontal mode since space tokens within the file inputed will produce unwanted spaces in the output.

\RIE@encoding stores the name of the current input encoding. It is used in an error message by \RIE@undefined.

\begin{verbatim}
\def\inputencoding#1{\protect\inputencoding{#1}\
\gdef\@inpenc@test{\global\let\@inpenc@test\relax}\
\protected@edef\RIE@encoding{#1}\
\ifvmode\
\RIE@loop\^^A\^^H\
\RIE@loop\^^K\^^K\
\RIE@loop\^^N\^^_\
\RIE@loop\^^?\^^ff\
\input{#1.def}\
\fi\
\ifx\@inpenc@test\relax\else\
\PackageError{rtkinenc}{No characters defined by input encoding change to '#1'}\fi
\end{verbatim}

\RIE@loop The \RIE@loop command makes characters #1 to #2 inclusive active and undefined.

\begin{verbatim}
\begingroup\
\catcode\z@=active\
\gdef\RIE@loop\#1\#2{\
\@tempcnta=\#1\relax\ifx\@tempcnta\#1\relax\
\count@=\uccode\z@\
\loop\
\catcode\@tempcnta=active\
\uccode\z@=\@tempcnta\
\fi\
\loop\
\endgroup
\end{verbatim}
1.3 Miscellanea

In many computer languages, special character escape sequences based on character codes must be written with the character code in hexadecimal or octal notation. These commands take a \TeX\ number in the interval 0–255 as argument and typesets it using the hexadecimal (figures 0–9 and a–f) or octal (figures 0–7) notation. No font changes or \TeX\ mode changes are made. \TypesetHexNumber\ always typesets two characters, \TypesetOctalNumber\ always typesets three characters.

Care has been taken to see to that every count register can be used as the argument of these two macros, and they only make local assignments. They do not check that the argument is in range, though.

\begin{verbatim}
172 \newcommand\TypesetHexNumber[1]{%
173 \begingroup
174 \count@=#1\relax
175 \chardef\@tempa=\count@
176 \divide \count@ \sixt@@n
177 \ifcase\count@
178 0\or 1\or 2\or 3\or 4\or 5\or 6\or 7\or 8\or 9\or a\or b\or
179 c\or d\or e\else f%
180 \fi
181 \multiply \count@ \sixt@@n
182 \advance \count@ \@tempa
183 \ifcase\count@
184 0\or 1\or 2\or 3\or 4\or 5\or 6\or 7\or 8\or 9\or a\or b\or
185 c\or d\or e\else f%
186 \fi
187 \endgroup
188 }
\end{verbatim}

\begin{verbatim}
189 \newcommand\TypesetOctalNumber[1]{%
190 \begingroup
191 \count@=#1\relax
192 \chardef\@tempa=\count@
193 \divide \count@ 64\relax
194 \the\count@
195 \multiply \count@ -64%
196 \advance \count@ \@tempa
197 \endgroup
198 }
\end{verbatim}
The `\verifycharcode` command is used as

```latex
\verifycharcode⟨character⟩⟨code⟩
```

Here ⟨character⟩ can be any ⟨character token⟩ (as defined in The TeXbook); e.g. a control sequence whose name consists of one character. ⟨code⟩ can be any valid ⟨number⟩. If ⟨code⟩ is not the character code of the ⟨character⟩, then `\verifycharcode` makes a warning about this.

The purpose of this command is to detect when the code of some character used in a document is changed. Today these things happen mainly when transferring a document between two systems which use different encodings, and it is usually the right thing to do. Some computer programs (and now I mean the source) do however rely on the exact character codes used in them, and documents containing such programs may use the `\verifycharcode` to test that none of the critical characters have been altered.

Each option is interpreted as the name of an input encoding. That input encoding definition file is inputed.