The rtkinenc package

Lars Hellström*

2000/01/24

Abstract

The rtkinenc package is functionally similar to the standard LaTeX package 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 rtkinenc have been selected so that it can read inputenc encoding definition files, and the aim is that rtkinenc should be backwards compatible with inputenc. rtkinenc is not a new version of inputenc though, nor is it part of standard LaTeX.

The main difference between the two packages lies in the view on the input. With 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 rtkinenc the input file itself is considered the true document, and the conversion of non-ASCII characters to LaTeX commands is merely done becuase 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 inputenc approach is natural for normal LaTeX documents, but the 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

```
1 ⟨*pkg⟩
2 \NeedsTeXFormat{LaTeX2e}[1995/12/01]
3 \ProvidesPackage{rtkinenc}
4    [2000/01/24 v1.0 Rethinking input encoding package]
```

1.1 Basic machanisms

`\RIE@last@char` `\RIE@last@char` is a `\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

```
\newcount\RIE@last@char
\global\RIE@last@char=\m@ne
```

\RieC
\RIE@char
\RIE@text@char
\RIE@code@char

The syntax for the \RieC command is

\RieC{⟨*code*⟩}{⟨*definition*⟩}

Here ⟨*code*⟩ is assumed to be a sequence of digits giving some raw character code, and ⟨*definition*⟩ is assumed to be robust LaTeX code for typesetting (some representation of) that raw character. \RieC is used as inputenc's \IeC command—an \RieC with arguments form the definition of an active character—but it carries extra information in the ⟨*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 ⟨*definition*⟩ straight off. This happens when \protect is \@typeset@protect and \RIE@char is \RIE@text@char.
- It can set \RIE@last@char to ⟨*code*⟩, then execute the ⟨*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 ⟨*defintion*⟩ and what surrounds it. Therefore it is inapproriate 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.[1] This happens when \protect is not \@typeset@protect, e.g. when writing to a file.

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

```
\@ifundefined{RieC}{}{%
   \PackageError{rtkinenc}{Redefining \protect\RieC}{%
      I had expected \protect\RieC\space to be undefined.\MessageBreak
      Since it wasn't, there's a chance I have\MessageBreak
      broken something.\MessageBreak\@ehc
   }
}
\def\RieC{%
   \ifx \protect\@typeset@protect
      \expandafter\RIE@char
   \else
      \noexpand\RieC
```

[1] Unless some command in the ⟨*definition*⟩ was defined using \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.

```
    \fi
}
\let\RIE@text@char=\@secondoftwo
\let\RIE@char=\RIE@text@char
\def\RIE@code@char#1#2{%
    \global\RIE@last@char=#1
    #2%
    \global\RIE@last@char=\m@ne
}
```

RieBC
RIE@both@char

The syntax for the \RieBC command is

\RieBC{⟨*code*⟩}{⟨*text definition*⟩}{⟨*math definition*⟩}

Here ⟨*code*⟩ is assumed to be a sequence of digits giving some raw character code, whereas ⟨*text definition*⟩ and ⟨*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.

```
\@ifundefined{RieBC}{}{%
    \PackageError{rtkinenc}{Redefining \protect\RieBC}{%
        I had expected \protect\RieBC\space to be undefined.\MessageBreak
        Since it wasn't, there's a chance I have\MessageBreak
        broken something.\MessageBreak\@ehc
    }
}
\def\RieBC{%
    \ifx \protect\@typeset@protect
        \expandafter\RIE@both@char
    \else
        \noexpand\RieBC
    \fi
}
\def\RIE@both@char#1#2#3{%
    \ifx \RIE@char\RIE@code@char
        \global\RIE@last@char=#1
    \fi
    \ifmmode #3\else #2\fi
    \ifx \RIE@char\RIE@code@char
        \global\RIE@last@char=\m@ne
    \fi
}
```

\TextSymbolUnavailable
\@@TextSymbolUnavailable
\SetUnavailableAction
\RIE@symbol@unavailable

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`.

```
\def\@tempa{\@autoerr\TextSymbolUnavailable}
\ifx \@tempa\TextSymbolUnavailable
   \@autoerr\relax
\fi
\let\@@TextSymbolUnavailable=\TextSymbolUnavailable
```

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

```
\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}
```

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

`\SetUnavailableAction{`⟨*definition*⟩`}`

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.

```
\newcommand\SetUnavailableAction{\def\RIE@symbol@unavailable##1##2}
```

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.

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

`\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`.

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.

```
\def\RIE@undefined#1{%
   \ifx \RIE@char\RIE@text@char
      \PackageError{rtkinenc}{%
         Input character #1 is undefined\MessageBreak
         in inputencoding \RIE@encoding}\@eha
   \else
      \RIE@symbol@unavailable{#1}\relax
   \fi
}
```

`\InputModeCode` `\InputModeText` `\IfInputModeCode`

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 ⟨*code*⟩ or ⟨*text*⟩ when the current mode is code mode or text mode, respectively.

```
\newcommand\InputModeCode{\let\RIE@char=\RIE@code@char}
\newcommand\InputModeText{\let\RIE@char=\RIE@text@char}
\newcommand\IfInputModeCode{%
   \ifx \RIE@char\RIE@code@char
      \expandafter\@firstoftwo
   \else
      \expandafter\@secondoftwo
   \fi
}
```

1.2 Setting the input encoding

The first two commands are identical; the duplication is only for being compabile 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.

`\DeclareInputText` `\DeclareInputMath`

These two commands are used as

`\DeclareInputText{`⟨*slot*⟩`}{`⟨*definition*⟩`}`
`\DeclareInputMath{`⟨*slot*⟩`}{`⟨*definition*⟩`}`

This makes the active character whose character code is ⟨*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.

```
\expandafter\ifx \csname DeclareInputText\endcsname\relax
   \begingroup
      \catcode\z@=13
      \gdef\DeclareInputText#1#2{%
         \@inpenc@test
         \begingroup
            \uccode\z@=#1%
            \uppercase{%
         \endgroup
            \expandafter\def \expandafter^^@%
         }\expandafter{\expandafter\RieC \expandafter{\number#1}{#2}}%
      }%
   \endgroup
\else
   \PackageError{rtkinenc}{\protect\DeclareInputText\space
      already defined}{\@ehd\MessageBreak
      Likely cause: you are already using the inputenc package.}
\fi
\@ifundefined{DeclareInputMath}{%
   \let\DeclareInputMath=\DeclareInputText
}{%
   \PackageError{rtkinenc}{\protect\DeclareInputMath\space
      already defined}{\@ehd\MessageBreak
      Likely cause: you are already using the inputenc package.}
}
```

\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.

```
\expandafter\ifx \csname DeclareInputBoth\endcsname\relax
   \begingroup
      \catcode\z@=13
      \gdef\DeclareInputBoth#1#2#3{%
         \@inpenc@test
         \begingroup
            \uccode\z@=#1%
            \uppercase{%
         \endgroup
            \expandafter\def \expandafter^^@%
         }\expandafter{\expandafter\RieBC \expandafter{\number#1}%
            {#2}{#3}%
```

```
        }%
      }%
   \endgroup
\else
   \PackageError{rtkinenc}{\protect\DeclareInputBoth\space
      already defined}\@ehd
\fi
```

`\inputencoding` `\@inpenc@test` `\RIE@encoding`

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`.

```
\def\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
      \PackageWarning{rtkinenc}%
              {No characters defined\MessageBreak
               by input encoding change to `#1'}%
   \fi
}
```

`\RIE@loop`

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

```
\begingroup
   \catcode\z@=\active
   \gdef\RIE@loop#1#2{%
      \@tempcnta=`#1\relax
      \count@=\uccode\z@
      \loop
         \catcode\@tempcnta\active
         \uccode\z@=\@tempcnta
```

```
        \uppercase{%
          \expandafter\def \expandafter^^@\expandafter{%
            \expandafter\RIE@undefined\expandafter{\the\@tempcnta}%
          }%
        }%
      \ifnum `#2>\@tempcnta
        \advance \@tempcnta \@ne
      \repeat
      \uccode\z@=\count@
  }
\endgroup
```

1.3 Miscellanea

`\TypesetHexNumber`
`\TypesetOctalNumber`

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.

```
\newcommand\TypesetHexNumber[1]{%
  \begingroup
    \count@=#1\relax
    \chardef\@tempa=\count@
    \divide \count@ \sixt@@n
    \ifcase\count@
      0\or 1\or 2\or 3\or 4\or 5\or 6\or 7\or 8\or 9\or a\or b\or
      c\or d\or e\else f%
    \fi
    \multiply \count@ -\sixt@@n
    \advance \count@ \@tempa
    \ifcase\count@
      0\or 1\or 2\or 3\or 4\or 5\or 6\or 7\or 8\or 9\or a\or b\or
      c\or d\or e\else f%
    \fi
  \endgroup
}
\newcommand\TypesetOctalNumber[1]{%
  \begingroup
    \count@=#1\relax
    \chardef\@tempa=\count@
    \divide \count@ 64\relax
    \the\count@
    \multiply \count@ -64%
    \advance \count@ \@tempa
```

```
      \chardef\@tempa=\count@
      \divide \count@ 8\relax
      \the\count@
      \multiply \count@ -8%
      \advance \count@ \@tempa
      \the\count@
   \endgroup
}
```

`\verifycharcode` The `\verifycharcode` command is used as

`\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 charater. ⟨*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 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.

```
\newcommand\verifycharcode[2]{%
   \ifnum `#1=#2 \else
      \PackageWarning{rtkinenc}{%
         Input character with code \number#2\MessageBreak
         should be the character with code \number`#1}%
   \fi
}
```

1.4 Option processing

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

```
\DeclareOption*{\inputencoding{\CurrentOption}}
\ProcessOptions
⟨/pkg⟩
```