The bytefield package*

Scott Pakin
pakin@uiuc.edu

2000/07/02

Abstract

The `bytefield` package helps the user create illustrations for network protocol specifications and anything else that utilizes fields of data. These illustrations show how the bits and bytes are laid out in a packet or in memory.

1 Introduction

Network protocols are usually specified in terms of a sequence of bits and bytes arranged in a field. This is portrayed graphically as a grid of boxes. Each row in the grid represents one word (frequently, 32 bits), and each column represents a bit within a word. The `bytefield` package makes it easy to typeset these sorts of figures.

`bytefield` lets one draw protocol diagrams that contain:

- Words of any arbitrary number of bits
- Column headers showing bit positions
- Multiword fields—even non-word-aligned and even if the total number of bits is not a multiple of the word length
- Word labels on either the left or right of the figure
- "Skipped words" within fields

Because `bytefield` draws its figures using only the LaTeX `picture` environment, these figures are not specific to any particular backend, do not require PostScript support, and do not need support from external programs. Furthermore, unlike an imported graphic, `bytefield` pictures can include arbitrary LaTeX constructs, such as mathematical equations, `\ref`s and `\cite`s to the surrounding document, and macro calls.

*This file has version number v1.00, last revised 2000/07/02.

2 Usage

2.1 Basic commands

This section explains how to use the `bytefield` package. It lists all the exported environments, commands, and variables in decreasing order of importance.

`\begin{bytefield}` `{`⟨*bit-width*⟩`}`
⟨*fields*⟩
`\end{bytefield}`

The top-level environment is called, not surprisingly, "`bytefield`". It takes one (mandatory) argument, which is the number of bits in each word. One can think of a `bytefield` as being analogous to a `tabular`: words are separated by `\\`, and fields within a word are separated by `&`.

`\wordbox` `[`⟨*sides*⟩`]` `{`⟨*height*⟩`}` `{`⟨*text*⟩`}`
`\bitbox` `[`⟨*sides*⟩`]` `{`⟨*width*⟩`}` `{`⟨*text*⟩`}`

The two main commands one uses within a `bytefield` environment are `\wordbox` and `\bitbox`. The former typesets a field that is one or more words tall and an entire word wide. The latter typesets a field that is one or more bits wide and a single word tall.

The optional argument, ⟨*sides*⟩, is a list of letters specifying which sides of the field box to draw—`[l]`eft, `[r]`ight, `[t]`op, and/or `[b]`ottom. The default is "`lrtb`" (i.e., all sides are drawn). ⟨*text*⟩ is the text to include within the `\wordbox` or `\bitbox`. It is typeset horizontally centered within a vertically centered `\parbox`. Hence, words will wrap, and `\\` can be used to break lines manually.

The following example shows how to produce a simple 16-bit-wide byte field:

```
\begin{bytefield}{16}
  \wordbox{1}{A 16-bit field} \\
  \bitbox{8}{8 bits} & \bitbox{8}{8 more bits} \\
  \wordbox{2}{A 32-bit field.  Note that text wraps within the box.}
\end{bytefield}
```

The resulting figure looks like this:

<table>
<tr><td colspan="2">A 16-bit field</td></tr>
<tr><td>8 bits</td><td>8 more bits</td></tr>
<tr><td colspan="2">A 32-bit field. Note that text wraps within the box.</td></tr>
</table>

It is the user's responsibility to ensure that the total number of bits in each row adds up to the number of bits in a single word (the mandatory argument to the `bytefield` environment).

Within a `\wordbox` or `\bitbox`, the `bytefield` package defines `\height`, `\depth`, `\totalheight`, and `\width` to the corresponding dimensions of the box. Section 2.2 gives an example of how these lengths may be utilized.

`\bitheader [`⟨*endianness*⟩`] {`⟨*bit-positions*⟩`}`

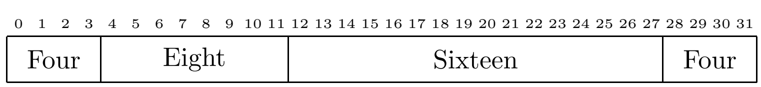
To make the figure more readable, it helps to label bit positions across the top. The `\bitheader` command provides a flexible way to do that. The optional argument, ⟨*endianness*⟩ is one of "`b`" or "`l`" and specifies whether the bits in each word are numbered in big-endian style (right to left) or little-endian style (left to right). The default is little-endian (`l`).

`\bitheader`'s mandatory argument, ⟨*bit-positions*⟩, is a comma-separated list of bit positions to label. For example, "`0,2,4,6,8,10,12,14`" means to label those bit positions. The numbers must be listed in increasing order. (Use ⟨*endianness*⟩ to display the header in reverse order.) Hyphen-separated ranges are also valid. For example, "`0-15`" means to label all bits from 0 to 15, inclusive. While not particularly useful, ranges and single numbers can be intermixed, as in "`0-3,8,12-15`".

The following example shows how `\bitheader` may be used:

```
\begin{bytefield}{32}
  \bitheader{0-31} \\
  \bitbox{4}{Four} & \bitbox{8}{Eight} &
    \bitbox{16}{Sixteen} & \bitbox{4}{Four}
\end{bytefield}
```

The resulting figure looks like this:



`\wordgroupr {`⟨*text*⟩`}`
`\endwordgroupr`

`\wordgroupl {`⟨*text*⟩`}`
`\endwordgroupl`

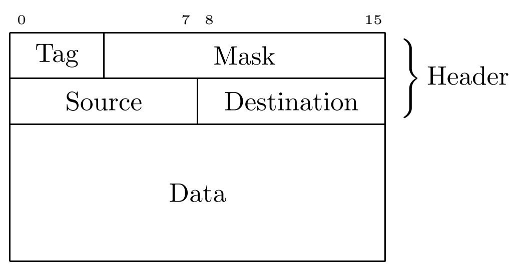
When a set of words functions as a single, logical unit, it helps to group these words together visually. All words defined between `\wordgroupr` and `\endwordgroupr` will be labeled on the right with ⟨*text*⟩. Similarly, all words defined between `\wordgroupl` and `\endwordgroupl` will be labeled on the left with ⟨*text*⟩. `\wordgroup`*x* must lie at the beginning of a row (i.e., right after a `\\`), and `\endwordgroup`*x* must lie right *before* the end of the row (i.e., right before a `\\`).

`\wordgroupr`...`\endwordgroupr` and `\wordgroupl`...`\endwordgroupl` can overlap each other. However, they cannot overlap themselves. In other words, `\wordgroupr`...`\wordgroupl`...`\endwordgroupr`...`\endwordgroupl` is a valid sequence, but `\wordgroupr`...`\wordgroupr`...`\endwordgroupr`...`\endwordgroupr` is not.

The following example shows how to use `\wordgroupr` and `\endwordgroupr`:

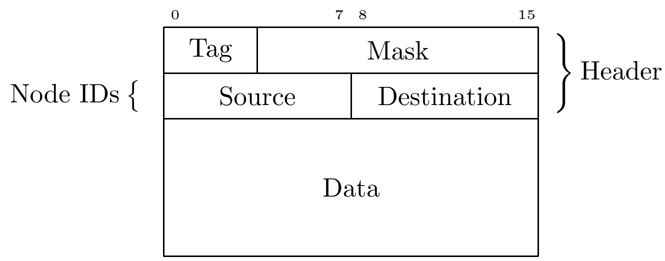
```
\begin{bytefield}{16}
  \bitheader{0,7,8,15} \\
  \wordgroupr{Header}
    \bitbox{4}{Tag} & \bitbox{12}{Mask} \\
    \bitbox{8}{Source} & \bitbox{8}{Destination}
  \endwordgroupr \\
  \wordbox{3}{Data}
\end{bytefield}
```

Note the justaposition of `\\` to `\wordgroupr` and `\endwordgroupr` in the above. The resulting figure looks like this:



As a more complex example, the following nests left and right labels:

```
\begin{bytefield}{16}
  \bitheader{0,7,8,15} \\
  \wordgroupr{Header}
    \bitbox{4}{Tag} & \bitbox{12}{Mask} \\
    \wordgroupl{Node IDs}
      \bitbox{8}{Source} & \bitbox{8}{Destination}
    \endwordgroupl
  \endwordgroupr \\
  \wordbox{3}{Data}
\end{bytefield}
```

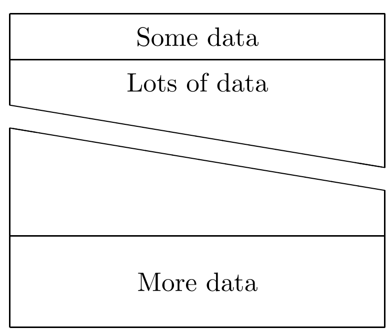


Again, note the justaposition of `\\` to the various word-grouping commands in the above.

`\skippedwords`

Draw a graphic representing a number of words that are not shown. `\skippedwords` is intended to work with the ⟨*sides*⟩ argument to `\wordbox`. For example:

```
\begin{bytefield}{16}
  \wordbox{1}{Some data} \\
  \wordbox[lrt]{1}{Lots of data} \\
  \skippedwords \\
  \wordbox[lrb]{1}{} \\
  \wordbox{2}{More data}
\end{bytefield}
```



`\bitwidth`
`\byteheight`

The above variables represent the width of each bit and height of each byte in the figure. Change them with `\setlength` to adjust the size of the figure. The default value of `\byteheight` is `2ex`, and the default value of `\bitwidth` is the width of “`{\tiny 99i}`”, i.e., the width of a two-digit number plus a small amount of extra space. This enables `\bitheader` to show two-digit numbers without overlap.

```
\curlyspace
\labelspace
```

`\curlyspace` is the space to insert between the figure and the curly brace preceding a word group (default: `1ex`). `\labelspace` is the space to insert between the curly brace and the label (default: `0.5ex`). Change these with `\setlength` to adjust the spacing.

```
\curlyshrinkage
```

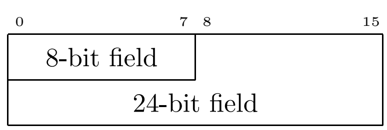
In TEX/LATEX, the height of a curly brace does not include the tips. Hence, in a word group label, the tips of the curly brace will extend beyond the height of the word group. `\curlyshrinkage` is an amount by which to reduce the height of curly braces in labels. It is set to `5pt`, and it is extremely unlikely that one would ever need to change it. Nevertheless, it is documented here in case the document is typeset with a math font containing radically different curly braces from the ones that come with TEX/LATEX.

2.2 Common tricks

This section shows some clever ways to use `bytefield`'s commands to produce some useful effects.

Odd-sized fields To produce a field that is, say, $1\frac{1}{2}$ words long, use a `\bitbox` for the fractional part and specify appropriate values for the various ⟨*sides*⟩ parameters. For instance:

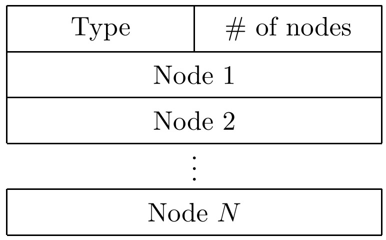
```
\begin{bytefield}{16}
  \bitheader{0,7,8,15} \\
  \bitbox{8}{8-bit field} & \bitbox[lrt]{8}{} \\
  \wordbox[lrb]{1}{24-bit field}
\end{bytefield}
```



Ellipses To skip words from the middle of enumerated data, put some `\vdots` in a `\wordbox` with empty ⟨*sides*⟩:

```
\begin{bytefield}{16}
  \bitbox{8}{Type} & \bitbox{8}{\# of nodes} \\
  \wordbox{1}{Node~1} \\
  \wordbox{1}{Node~2} \\
  \wordbox[]{1}{$\vdots$ \\[1ex]} \\
  \wordbox{1}{Node~$N$} \\
```

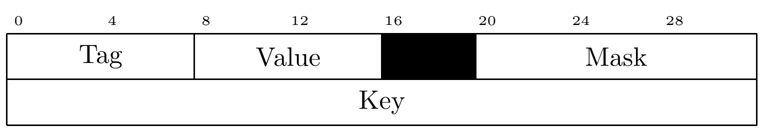
```
\end{bytefield}
```



The extra `1ex` of vertical space helps center the `\vdots` a bit better.

Unused bits Because `\width` and `\height` are defined within `\bitbox`es (also `\wordbox`es), we can represent unused bits by filling a `\bitbox` with a rule of size `\width`×`\height`.

```
\begin{bytefield}{32}
  \bitheader{0,4,8,12,16,20,24,28} \\
  \bitbox{8}{Tag} & \bitbox{8}{Value} &
    \bitbox{4}{\rule{\width}{\height}} & \bitbox{12}{Mask} \\
  \wordbox{1}{Key}
\end{bytefield}
```



The effect is much better when the `color` package is used to draw the unused bits in color. (Gray looks nice.)

2.3 Not-so-common tricks

While certainly not the intended purpose of the `bytefield` package, one can utilize `\wordbox`es with empty ⟨*sides*⟩ and word labels to produce memory-map diagrams:

```
\setlength{\byteheight}{4\baselineskip}
\newcommand{\descbox}[2]{\parbox[c][3.8\baselineskip]{0.95\width}{%
  \raggedright #1\vfill #2}}
\begin{bytefield}{32}
  \wordgroupr{Partition 4}
    \bitbox[]{8}{\texttt{0xFFFFFFFF} \\[2\baselineskip]
      \texttt{0xC0000000}} &
    \bitbox{24}{\descbox{1\,GB area for VxDs, memory manager,
      file system code; shared by all processes.}{Read/writable.}}
  \endwordgroupr \\
```

```
\wordgroupr{Partition 3}
  \bitbox[]{8}{\texttt{0xBFFFFFFF} \\[2\baselineskip]
    \texttt{0x80000000}} &
  \bitbox{24}{\descbox{1\,GB area for memory-mapped files,
    shared system DLLs, file system code; shared by all
    processes.}{Read/writable.}}
\endwordgroupr \\
\wordgroupr{Partition 2}
  \bitbox[]{8}{\texttt{0x7FFFFFFF} \\[2\baselineskip]
    \texttt{0x00400000}} &
  \bitbox{24}{\descbox{$\sim$2\,GB area private to process, process
    code, and data.}{Read/writable.}}
\endwordgroupr \\
\wordgroupr{Partition 1}
  \bitbox[]{8}{\texttt{0x003FFFFF} \\[2\baselineskip]
    \texttt{0x00001000}} &
  \bitbox{24}{\descbox{4\,MB area for MS-DOS and Windows~3.1
    compatibility.}{Read/writable.}} \\
  \bitbox[]{8}{\texttt{0x00000FFF} \\[2\baselineskip]
    \texttt{0x00000000}} &
  \bitbox{24}{\descbox{4096~byte area for MS-DOS and Windows~3.1
    compatibility.}{Protected---catches {\small NULL} pointers.}}
\endwordgroupr \\
```

`0xFFFFFFFF` `0xC0000000`	1 GB area for VxDs, memory manager, file system code; shared by all processes. Read/writable.	Partition 4
`0xBFFFFFFF` `0x80000000`	1 GB area for memory-mapped files, shared system DLLs, file system code; shared by all processes. Read/writable.	Partition 3
`0x7FFFFFFF` `0x00400000`	∼2 GB area private to process, process code, and data. Read/writable.	Partition 2
`0x003FFFFF` `0x00001000`	4 MB area for MS-DOS and Windows 3.1 compatibility. Read/writable.	Partition 1
`0x00000FFF` `0x00000000`	4096 byte area for MS-DOS and Windows 3.1 compatibility. Protected—catches NULL pointers.	

2.4 Putting it all together

The following code showcases most of `bytefield`'s features in a single figure.

```
\setlength{\byteheight}{2.5\baselineskip}
\begin{bytefield}{32}
  \bitheader{0,7,8,15,16,23,24,31} \\
  \wordgroupr{\parbox{6em}{\raggedright These words were taken
    verbatim from the TCP header definition (RFC~793).}}
    \bitbox{4}{Data offset} & \bitbox{6}{Reserved} &
      \bitbox{1}{\tiny U\\R\\G} & \bitbox{1}{\tiny A\\C\\K} &
      \bitbox{1}{\tiny P\\S\\H} & \bitbox{1}{\tiny R\\S\\T} &
      \bitbox{1}{\tiny S\\Y\\N} & \bitbox{1}{\tiny F\\I\\N} &
      \bitbox{16}{Window} \\
    \bitbox{16}{Checksum} & \bitbox{16}{Urgent pointer}
  \endwordgroupr \\
  \wordbox[lrt]{1}{Data octets} \\
  \skippedwords \\
  \wordbox[lrb]{1}{} \\
  \wordgroupl{\parbox{6em}{\raggedright Note that we can display,
    for example, a misaligned 64-bit value with clever use of the
    optional argument to \texttt{\textbackslash wordbox} and
    \texttt{\textbackslash bitbox}.}}
    \bitbox{8}{Source} & \bitbox{8}{Destination} & \bitbox[lrt]{16}{} \\
    \wordbox[lr]{1}{Timestamp} \\
    \wordgroupr{\parbox{6em}{\raggedright Why two Length fields?
      No particular reason.}}
      \bitbox[lrb]{16}{} & \bitbox{16}{Length}
  \endwordgroupl \\
      \bitbox{6}{Key} & \bitbox{6}{Value} & \bitbox{4}{Unused} &
        \bitbox{16}{Length}
    \endwordgroupr \\
  \wordbox{1}{Total number of 16-bit data words that follow this
    header word, excluding the subsequent checksum-type value} \\
  \bitbox{16}{Data~1} & \bitbox{16}{Data~2} \\
  \bitbox{16}{Data~3} & \bitbox{16}{Data~4} \\
  \bitbox[]{16}{$\vdots$ \\[1ex]} & \bitbox[]{16}{$\vdots$ \\[1ex]} \\
  \bitbox{16}{Data~$N-1$} & \bitbox{16}{Data~$N$} \\
  \bitbox{20}{\[ \mbox{A5A5}_{\mbox{\scriptsize H}} \oplus
    \left(\sum_{i=1}^N \mbox{Data}_i \right) \bmod 2^{20} \]} &
    \bitbox{12}{Command} \\
  \wordbox{2}{64-bit random number}
\end{bytefield}
```

Figure 1 shows the resulting protocol diagram.

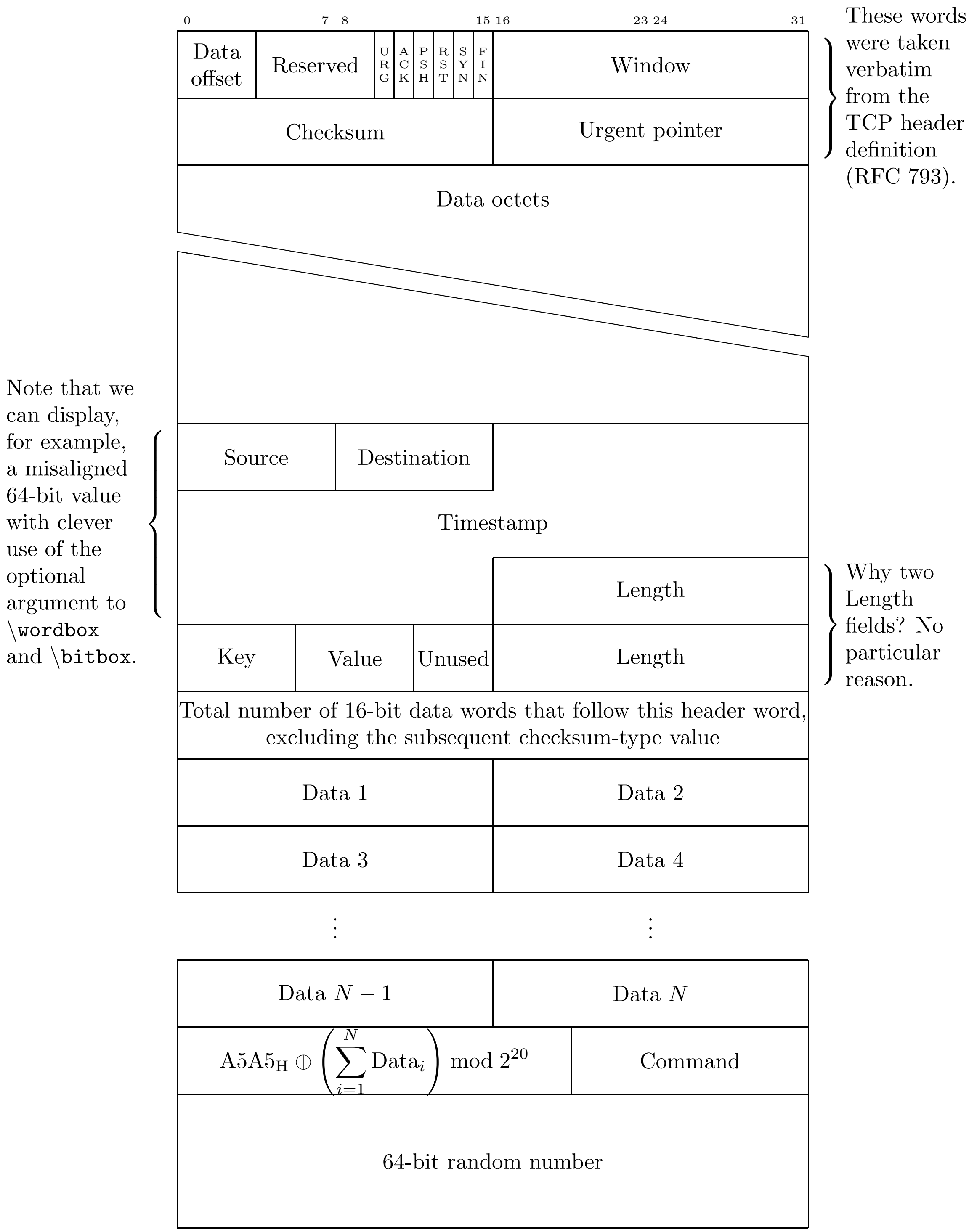


Figure 1: Complex protocol diagram drawn with the `bytefield` package

3 Implementation

This section contains the complete source code for `bytefield`. Most users will not get much out of it, but it should be of use to those who need more precise documentation and those who want to extend (or debug ☺) the `bytefield` package.

In this section, macros marked in the margin with a "★" are intended to be called by the user (and were described in the previous section). All other macros are used only internally by `bytefield`.

```
⟨*package⟩
```

3.1 Utility macros

`\remove@dim` Remove "`pt`" from the end of a dimen (e.g., `12.34pt` ↦ "`12.34`"). I stole this from Hideki Isozaki's `ecltree` package.

```
{\catcode`\p=12 \catcode`\t=12 \gdef\remove@dim#1pt{#1}}
```

`\no@pt` Make `\remove@dim` a little more user-friendly.

```
\def\no@pt#1{\expandafter\remove@dim\the#1}
```

The following macros in this section are used by the box-drawing macros and the "skipped words"-drawing macros.

`\bytefield@height`
`\ifcounting@words` When `\ifcounting@words` is TRUE, add the height of the next `picture` environment to `\bytefield@height`. We set `\counting@wordstrue` at the beginning of each word, and `\counting@wordsfalse` after each `\bitbox`, `\wordbox`, or `\skippedwords` picture.

```
\newlength{\bytefield@height}
\newif\ifcounting@words
```

`\inc@bytefield@height` We have to define a special macro to increment `\bytefield@height` because the `calc` package's `\addtolength` macro doesn't seem to see the global value. So we `\setlength` a temporary (to get `calc`'s nice infix features) and `\advance` `\bytefield@height` by that amount.

```
\newlength{\bytefield@height@increment}
\DeclareRobustCommand{\inc@bytefield@height}[1]{%
  \setlength{\bytefield@height@increment}{#1}%
  \global\advance\bytefield@height by \bytefield@height@increment}
```

3.2 Top-level environment

`bits@wide` The number of bits in each word (i.e., the argument to the `\bytefield` environment).

```
\newcounter{bits@wide}
```

`\entire@bytefield@picture` A box containing the entire bytefield. By storing everything in a box and then typesetting it later (at the `\end{bytefield}`), we can center the bytefield, put a box around it, and do other operations on the entire figure.

```
\newsavebox{\entire@bytefield@picture}
```

★ `bytefield` Environment containing the layout of bits in a sequence of bytes. This is the main environment defined by the `bytefield` pacakge. The argument is the number of bits wide the bytefield should be. We turn `&` into a space character so the user can think of a `bytefield` as being analogous to a `tabular` environment, even though we're really setting the bulk of the picture in a single column. (Row labels go in separate columns, however.)

```
\newenvironment{bytefield}[1]{%
  \setcounter{bits@wide}{#1}%
  \let\old@nl=\\%
  \let\amp=&%
  \catcode`\&=10
  \openup -1pt
  \setlength{\bytefield@height}{0pt}%
  \setlength{\unitlength}{1pt}%
  \counting@wordstrue
  \begin{lrbox}{\entire@bytefield@picture}%
  \renewcommand{\\}{%
    \amp\show@wordlabelr\cr\ignorespaces\counting@wordstrue\make@lspace\amp}%
  \vbox\bgroup\ialign\bgroup##\amp##\amp##\cr\amp%
}{%
  \amp\show@wordlabelr\cr\egroup\egroup%
  \end{lrbox}%
  \usebox{\entire@bytefield@picture}}
```

3.3 Box-drawing macros

3.3.1 Drawing (proper)

★ `\bitwidth` The width of a single bit. Note that this is wide enough to display a two-digit number without it running into adjacent numbers. For larger words, be sure to `\setlength` this larger.

```
\newlength{\bitwidth}
\AtBeginDocument{\settowidth{\bitwidth}{\tiny 99i}}
```

★ `\byteheight` The height of a single byte.

```
\newlength{\byteheight}
\AtBeginDocument{\setlength{\byteheight}{4ex}}
```

`\units@wide`
`\units@tall` Scratch variables for storing the width and height (in points) of the box we're about to draw.

```
\newlength{\units@wide}
\newlength{\units@tall}
```

`\bitbox` Put some text (#3) in a box that's a given number of bits (#2) wide and one byte tall. An optional argument (#1) specifies which lines to draw—[l]eft, [r]ight, [t]op, and/or [b]ottom (default: lrtb).

```
\DeclareRobustCommand{\bitbox}[3][lrtb]{%
  \setlength{\units@wide}{\bitwidth * #2}%
  \parse@bitbox@arg{#1}%
  \draw@bit@picture{\no@pt{\units@wide}}{\no@pt{\byteheight}}{#3}}
```

`\wordbox` Put some text (#3) in a box that's a given number of bytes (#2) tall and one word (bits@wide bits) wide. An optional argument (#1) specifies which lines to draw—[l]eft, [r]ight, [t]op, and/or [b]ottom (default: lrtb).

```
\DeclareRobustCommand{\wordbox}[3][lrtb]{%
  \setlength{\units@wide}{\bitwidth * \value{bits@wide}}%
  \setlength{\units@tall}{\byteheight * #2}%
  \parse@bitbox@arg{#1}%
  \draw@bit@picture{\no@pt{\units@wide}}{\no@pt{\units@tall}}{#3}}
```

`\height` `\depth` `\totalheight` `\width` Box sizes to make available to the user within a `\bitbox` or `\wordbox`. These *should* be local to the `\parbox` in `\draw@bit@picture`, but when I moved the `\newdimen`s into `\draw@bit@picture`, I kept running out of dimens for documents containing many boxes. What's the right way to do this sort of local variable in TeX?

```
\newdimen\height
\newdimen\depth
\newdimen\totalheight
\newdimen\width
```

`\draw@bit@picture` Put some text (#3) in a box that's a given number of units (#1) wide and a given number of units (#2) tall. We format the text with a `\parbox` to enable word-wrapping and explicit line breaks. In addition, we define `\height`, `\depth`, `\totalheight`, and `\width` (à la `\makebox` and friends), so the user can utilize those for special effects (e.g., a `\rule` that fills the entire box). As an added bonus, we define `\widthunits` and `\heightunits`, which are the width and height of the box in multiples of `\unitlength` (i.e., #1 and #2, respectively).

```
\DeclareRobustCommand{\draw@bit@picture}[3]{%
  \begin{picture}(#1,#2)
```

First, we plot the user's text, with all sorts of useful lengths predefined.

```
    \put(0,0){\makebox(#1,#2){\parbox[c]{#1\unitlength}{%
      \height=#2\unitlength%
      \depth=0pt%
      \totalheight=#2\unitlength%
      \width=#1\unitlength%
      \def\widthunits{#1}%
      \def\heightunits{#2}%
      \centering #3}}}
```

Next, we draw each line individually. I suppose we could make a special case for "all lines" and use a `\framebox` above, but the following works just fine.

```
    \ifbitbox@top
      \put(0,#2){\line(1,0){#1}}
    \fi
    \ifbitbox@bottom
      \put(0,0){\line(1,0){#1}}
    \fi
    \ifbitbox@left
      \put(0,0){\line(0,1){#2}}
    \fi
    \ifbitbox@right
      \put(#1,0){\line(0,1){#2}}
    \fi
  \end{picture}%
```

Finally, we indicate that we're no longer at the beginning of a word. The following code structure (albeit with different arguments to `\inc@bytefield@height`) is repeated in various places throughout this package. We document it only here, however.

```
  \ifcounting@words
    \inc@bytefield@height{\unitlength * \real{#2}}%
    \counting@wordsfalse
  \fi
  \ignorespaces}
```

3.3.2 Parsing arguments

The macros in this section are used to parse the optional argument to `\bitbox` or `\wordbox`, which is some subset of $\{$`l`, `r`, `t`, `b`$\}$.

`\ifbitbox@top`
`\ifbitbox@bottom`
`\ifbitbox@left`
`\ifbitbox@right`

These macros are set to TRUE if we're to draw the corresponding edge on the subsequent `\bitbox` or `\wordbox`.

```
\newif\ifbitbox@top
\newif\ifbitbox@bottom
\newif\ifbitbox@left
\newif\ifbitbox@right
```

`\parse@bitbox@arg`

This main parsing macro merely resets the above conditionals and calls a helper function, `\parse@bitbox@sides`.

```
\def\parse@bitbox@arg#1{%
  \bitbox@topfalse
  \bitbox@bottomfalse
  \bitbox@leftfalse
  \bitbox@rightfalse
  \parse@bitbox@sides#1X}
```

`\parse@bitbox@sides`

The helper function for `\parse@bitbox@arg` parses a single letter, sets the appropriate conditional to TRUE, and calls itself tail-recursively until it sees an "`X`".

```
\def\parse@bitbox@sides#1{%
  \ifx#1X%
  \else
    \ifx#1t%
      \bitbox@toptrue
    \else
      \ifx#1b%
        \bitbox@bottomtrue
      \else
        \ifx#1l%
          \bitbox@lefttrue
        \else
          \ifx#1r%
            \bitbox@righttrue
          \fi
        \fi
      \fi
    \fi
    \expandafter\parse@bitbox@sides
  \fi}
```

3.4 Skipped words

`\units@high` The height of each diagonal line in the `\skippedwords` graphic. Note that `\units@high` = `\units@tall` − *optional argument to* `\skippedwords`.

```
\newlength{\units@high}
```

★ `\skippedwords` Output a fancy graphic representing skipped words. The optional argument is the vertical space between the two diagonal lines (default: `2ex`).

```
\DeclareRobustCommand{\skippedwords}[1][2ex]{%
  \setlength{\units@wide}{\bitwidth * \value{bits@wide}}%
  \setlength{\units@high}{1pt * \ratio{\units@wide}{6.0pt}}%
  \setlength{\units@tall}{#1 + \units@high}%
  \edef\num@wide{\no@pt{\units@wide}}%
  \edef\num@tall{\no@pt{\units@tall}}%
  \edef\num@high{\no@pt{\units@high}}%
  \begin{picture}(\num@wide,\num@tall)
    \put(0,\num@tall){\line(6,-1){\num@wide}}
    \put(\num@wide,0){\line(-6,1){\num@wide}}
    \put(0,0){\line(0,1){\num@high}}
    \put(\num@wide,\num@tall){\line(0,-1){\num@high}}
  \end{picture}%
  \ifcounting@words
    \inc@bytefield@height{\unitlength * \real{\num@tall}}%
    \counting@wordsfalse
  \fi}
```

3.5 Bit-position labels

`\bitheader` Output a header of numbered bit positions. The optional argument (`#1`) is “`l`” for little-endian (default) or “`b`” for big-endian. The required argument (`#2`) is a list of bit positions to label. It is composed of comma-separated ranges of numbers, for example, “`0-31`”, “`0,7-8,15-16,23-24,31`”, or even something odd like “`0-7,15-23`”. Ranges must be specified in increasing order; use the optional argument to `\bitheader` to reverse the labels’ direction.

```
\DeclareRobustCommand{\bitheader}[2][l]{%
  \parse@bitbox@arg{lrtb}%
  \setlength{\units@wide}{\bitwidth * \value{bits@wide}}%
  \setlength{\units@tall}{\heightof{\tiny 9}}%
  \setlength{\units@high}{\units@tall * -1}%
  \def\bit@endianness{#1}%
  \begin{picture}(\no@pt{\units@wide},\no@pt{\units@tall})(0,\no@pt{\units@high})
    \parse@range@list#2,X,
  \end{picture}%
  \ifcounting@words
    \inc@bytefield@height{\unitlength * \real{\no@pt{\units@tall}}}%
    \counting@wordsfalse
  \fi
  \ignorespaces}
```

`\parse@range@list` Helper function #1 for `\bitheader`—parse a comma-separated list of ranges, calling `\parse@range` on each range.

```
\def\parse@range@list#1,{%
  \ifx#1X
  \else
    \parse@range#1-#1-#1\relax
    \expandafter\parse@range@list
  \fi}
```

`\header@xpos`
`header@val`
`max@header@val`
Miscellaneous variables used internally by `\parse@range`—x position of header, current label to output, and maximum label to output (+1).

```
\newlength{\header@xpos}
\newcounter{header@val}
\newcounter{max@header@val}
```

`\parse@range` Helper function #2 for `\bitheader`—parse a hyphen-separated pair of numbers (or a single number) and display the number at the correct bit position.

```
\def\parse@range#1-#2-#3\relax{%
  \setcounter{header@val}{#1}
  \setcounter{max@header@val}{#2 + 1}
  \loop
    \ifnum\value{header@val}<\value{max@header@val}%
      \if\bit@endianness b%
        \setlength{\header@xpos}{\bitwidth * (\value{bits@wide}-\value{header@val}-1)}
      \else
        \setlength{\header@xpos}{\bitwidth * \value{header@val}}
```

```
    \fi
    \put(\no@pt{\header@xpos},0){%
      \makebox(\no@pt{\bitwidth},\no@pt{\units@tall}){\tiny \theheader@val}}
    \addtocounter{header@val}{1}
\repeat}
```

3.6 Word labels

3.6.1 Curly-brace manipulation

★ `\curlyshrinkage` Reduce the height of a curly brace by `\curlyshrinkage` so its ends don't overlap whatever is above or below it. The default value (5 pt.) was determined empirically and shouldn't need to be changed. However, on the off-chance the user employs a math font with very different curly braces from Computer Modern's, `\curlyshrinkage` can be modified.

```
\newlength{\curlyshrinkage}
\setlength{\curlyshrinkage}{5pt}
```

★ `\curlyspace`
★ `\labelspace` Space to insert before a curly brace and before a word label (i.e., after a curly brace). Because the default values are specified in terms of x heights, we wait until the `\begin{document}` to set them, after the default font has been selected.

```
\newlength{\curlyspace}
\AtBeginDocument{\setlength{\curlyspace}{1ex}}
\newlength{\labelspace}
\AtBeginDocument{\setlength{\labelspace}{0.5ex}}
```

`\store@rcurly` Store a "}" that's `#2` tall in box `#1`. The only unintuitive thing here is that we have to redefine `\fontdimen22`—axis height—to 0 pt. before typesetting the curly brace. Otherwise, the brace would be vertically off-center by a few points. When we're finished, we reset it back to its old value.

```
\def\store@rcurly#1#2{%
  \newdimen\curly@height%
  \setlength{\curly@height}{#2 - \curlyshrinkage}%
  \newdimen\half@curly@height%
  \setlength{\half@curly@height}{0.5\curly@height}%
  \newdimen\curly@shift%
  \setlength{\curly@shift}{\half@curly@height + 0.5\curlyshrinkage}%
  \sbox{#1}{\raisebox{\curly@shift}{%
    $\xdef\old@axis{\the\fontdimen22\textfont2}$%
    $\fontdimen22\textfont2=0pt%
     \left.\vrule height\half@curly@height width0pt depth\half@curly@height\right\}$%
    $\fontdimen22\textfont2=\old@axis$}}%
}
```

`\store@lcurly` Same as `\store@rcurly`, but using a "{" instead of a "}".

```
\def\store@lcurly#1#2{%
  \newdimen\curly@height%
  \setlength{\curly@height}{#2 - \curlyshrinkage}%
```

```
\newdimen\half@curly@height%
\setlength{\half@curly@height}{0.5\curly@height}%
\newdimen\curly@shift%
\setlength{\curly@shift}{\half@curly@height + 0.5\curlyshrinkage}%
\sbox{#1}{\raisebox{\curly@shift}{%
  $\xdef\old@axis{\the\fontdimen22\textfont2}$%
  $\fontdimen22\textfont2=0pt%
   \left\{\vrule height\half@curly@height width0pt depth\half@curly@height\right.$%
  $\fontdimen22\textfont2=\old@axis$}}%
}
```

3.6.2 Right-side labels

`\show@wordlabelr` This macro is output in the third column of every row of the `\ialign`ed bytefield table. It's normally a no-op, but `\endwordgroupr` defines it to output the word label and then reset itself to a no-op.

```
\def\show@wordlabelr{}
```

`\wordlabelr@start` `\wordlabelr@end` The starting and ending height (in points) of the set of rows to be labelled on the right.

```
\newlength{\wordlabelr@start}
\newlength{\wordlabelr@end}
```

★ ★ `\wordgroupr` `\endwordgroupr` Label the words defined between `\wordgroupr` and `\endwordgroupr` on the right side of the figure. The argument is the text of the label. The label is typeset to the right of a large curly brace, which groups the words together.

```
\newenvironment{wordgroupr}[1]{%
```

`\wordgroupr` merely stores the starting height in `\wordlabelr@start` and the user-supplied text in `\wordlabelr@text`. `\endwordgroupr` does most of the work.

```
  \global\wordlabelr@start=\bytefield@height
  \gdef\wordlabelr@text{#1}%
  \ignorespaces%
}{%
  \global\wordlabelr@end=\bytefield@height
```

Redefine `\show@wordlabelr` to output `\curlyspace` space, followed by a large curly brace (in `\curlybox`), followed by `\labelspace` space, followed by the user's text (previously recorded in `\wordlabelr@text`). We typeset `\wordlabelr@text` within a `tabular` environment, so LaTeX will calculate its width automatically.

```
  \gdef\show@wordlabelr{%
    \sbox{\word@label@box}{\begin{tabular}[b]{@{}l@{}}\wordlabelr@text\end{tabular}}%
    \settowidth{\label@box@width}{\usebox{\word@label@box}}%
    \setlength{\label@box@height}{\wordlabelr@end-\wordlabelr@start}%
    \newbox{\curly@box}%
    \store@rcurly{\curly@box}{\label@box@height}%
    \newdimen\total@box@width%
    \setlength{\total@box@width}{%
      \curlyspace +
```

```
      \widthof{\usebox{\curly@box}} +
      \labelspace +
      \label@box@width}%
    \begin{picture}(\no@pt{\total@box@width},0)
      \put(0,0){%
        \hspace*{\curlyspace}%
        \usebox{\curly@box}%
        \hspace*{\labelspace}%
        \makebox(\no@pt{\label@box@width},\no@pt{\label@box@height}){%
          \usebox{\word@label@box}}}
    \end{picture}%
```

The last thing `\show@wordlabelr` does is redefine itself back to a no-op.

```
    \gdef\show@wordlabelr{}}%
  \ignorespaces}
```

3.6.3 Left-side labels

`\wordlabell@start`
`\wordlabell@end`

The starting and ending height (in points) of the set of rows to be labelled on the left.

```
\newlength{\wordlabell@start}
\newlength{\wordlabell@end}
```

`\total@box@width`

The total width of the next label to typeset on the left of the figure, that is, the aggregate width of the text box, curly brace, and spaces on either side of the curly brace.

```
\newlength{\total@lbox@width}
```

`\make@lspace`

This macro is output in the first column of every row of the `\ialign`ed bytefield table. It's normally a no-op, but `\wordgroupl` defines it to output enough space for the next word label and then reset itself to a no-op.

```
\gdef\make@lspace{}
```

★ `\wordgroupl`
★ `\endwordgroupl`

Same as `\wordgroupr` and `\endwordgroupr`, but put the label on the left. However, the following code is not symmetric to that of `\wordgroupr` and `\endwordgroupr`. The problem is that we encounter `\wordgroupl` after entering the second (i.e., figure) column, which doesn't give us a chance to reserve space in the first (i.e., left label) column. When we reach the `\endwordgroupl`, we know the height of the group of words we wish to label. However, if we try to label the words in the subsequent first column, we won't know the vertical offset from the "cursor" at which to start drawing the label, because we can't know the height of the subsequent row until we reach the second column.[1]

Our solution is to allocate space for the box the next time we enter a first column. As long as space is eventually allocated, the column will expand to fit that space. `\endwordgroupl` outputs the label immediately. Even though `\endwordgroupl` is called at the end of the *second* column, it `\put`s the label at

[1]Question: Is there a way to push the label up to the *top* of the subsequent row, perhaps with `\vfill`?

a sufficiently negative x location for it to overlap the first column. Because there will eventually be enough space to accomodate the label, we know that the label won't overlap the figure or extend beyond the figure boundaries.

```
\newenvironment{wordgroupl}[1]{%
```

First, we store the starting height and label text, which are needed by `\endwordgroupl`.

```
  \global\wordlabell@start=\bytefield@height
  \gdef\wordlabell@text{#1}%
```

Next, we typeset a draft version of the label into `\word@label@box`, which we measure (into `\total@lbox@width`) and then discard. We can't typeset the final version of the label until we reach the `\endwordgroupl`, because that's when we learn the height of the word group. Without knowing the height of the word group, we don't how how big to make the curly brace. In the scratch version, we make the curly brace 5 cm. tall. This should be more than large enough to reach the maximum curly-brace width, which is all we really care about at this point.

```
  \sbox{\word@label@box}{\begin{tabular}[b]{@{}l@{}}\wordlabell@text\end{tabular}}%
  \settowidth{\label@box@width}{\usebox{\word@label@box}}%
  \newbox{\curly@box}%
  \store@lcurly{\curly@box}{5cm}%
  \setlength{\total@lbox@width}{%
    \curlyspace +
    \widthof{\usebox{\curly@box}} +
    \labelspace +
    \label@box@width}%
  \global\total@lbox@width=\total@lbox@width
```

Now we know how wide the box is going to be (unless, of course, the user is using some weird math font that scales the width of a curly brace proportionally to its height). So we redefine `\make@lspace` to output `\total@lbox@width`'s worth of space and then redefine itself back to a no-op.

```
  \gdef\make@lspace{%
    \hspace*{\total@lbox@width}%
    \gdef\make@lspace{}}%
  \ignorespaces%
}{%
```

`\endwordgroupl` is comparatively straightforward. We calculate the final height of the word group, and then output the label text, followed by `\labelspace` space, followed by a curly brace (now that we know how tall it's supposed to be), followed by `\curlyspace` space. The trick, as described earlier, is that we typeset the entire label in the second column, but in a 0×0 `picture` environment and with a negative horizontal offset (`\starting@point`), thereby making it overlap the first column.

```
  \global\wordlabell@end=\bytefield@height
  \newdimen\starting@point
  \setlength{\starting@point}{%
    -\total@lbox@width - \bitwidth*\value{bits@wide}}%
  \sbox{\word@label@box}{\begin{tabular}[b]{@{}l@{}}\wordlabell@text\end{tabular}}%
  \settowidth{\label@box@width}{\usebox{\word@label@box}}%
```

```
\setlength{\label@box@height}{\wordlabell@end-\wordlabell@start}%
\newbox{\curly@box}%
\store@lcurly{\curly@box}{\label@box@height}%
\begin{picture}(0,0)
  \put(\no@pt{\starting@point},0){%
    \makebox(\no@pt{\label@box@width},\no@pt{\label@box@height}){%
      \usebox{\word@label@box}}%
    \hspace*{\labelspace}%
    \usebox{\curly@box}%
    \hspace*{\curlyspace}}
\end{picture}%
\ignorespaces}
```

3.6.4 Scratch space

`\label@box@width` `\label@box@height` `\word@label@box`

Scratch storage for the width, height, and contents of the word label we're about to output.

```
\newlength{\label@box@width}
\newlength{\label@box@height}
\newsavebox{\word@label@box}
```

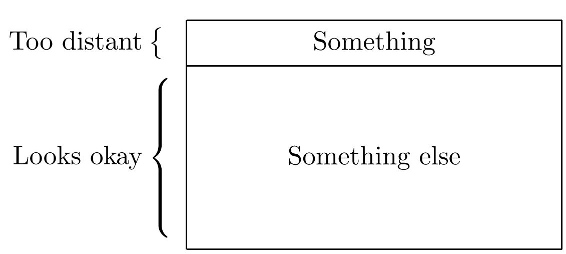
⟨/package⟩

4 Future work

`bytefield` is my first LaTeX package, and, as such, there are a number of macros that could probably have been implemented a lot better. The package should really get a major rewrite. If I were to do it all over again, I would probably not use an `\ialign` for the main `bytefield` environment. The problem—as I discovered too late—is that `\begin`...`\end` blocks are unable to cross cells of an `\ialign` (or `tabular` environment, for that matter).

That said, I'd like the next major release of `bytefield` to let the user use `\begin{wordgroup}[r]`...`\end{wordgroup}` instead of `\wordgroupr`... `\endwordgroupr` and `\begin{wordgroup}[l]`...`\end{wordgroup}` instead of `\wordgroupl`...`\endwordgroupl`. That would make the word-grouping commands a little more LaTeX-ish.

Finally, a minor improvement I'd like to make in the package is to move left, small curly braces closer to the figure. In the following figure, notice how distant the small curly appears from the figure body:



The problem is that the curly braces are left-aligned relative to each other, while they should be right-aligned.