The bytefield package[*]

Scott Pakin
scott+bf@pakin.org

September 24, 2023

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.

WARNING: bytefield version 2.x breaks compatibility with older versions of the package. See Section 2.7 for help porting documents to the new interface.

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, 8, 16, or 32 bits), and each column represents a bit within a word. The bytefield package makes it easy to typeset these sorts of figures. bytefield facilitates drawing 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, and
- "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 or PDF 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 document corresponds to bytefield v2.8, dated 2023/09/24.

2 Usage

2.1 A first example

The Internet Engineering Task Force's Request for Comments (RFC) number 3016 includes the following ASCII-graphics illustration of the RTP packetization of an MPEG-4 Visual bitstream:

```
0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|V=2|P|X|  CC   |M|     PT      |       sequence number         | RTP
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           timestamp                           | Header
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           synchronization source (SSRC) identifier            |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|            contributing source (CSRC) identifiers             |
|                             ....                              |
+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+=+
|                                                               | RTP
|       MPEG-4 Visual stream (byte aligned)                     | Pay-
|                                                               | load
|                               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                               :...OPTIONAL RTP padding        |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The following LaTeX code shows how straightforward it is to typeset that illustration using the bytefield package:

```
\begin{bytefield}[bitwidth=1.1em]{32}
  \bitheader{0-31} \\
  \begin{rightwordgroup}{RTP \\ Header}
    \bitbox{2}{V=2} & \bitbox{1}{P} & \bitbox{1}{X}
    & \bitbox{4}{CC} & \bitbox{1}{M} & \bitbox{7}{PT}
    & \bitbox{16}{sequence number} \\
    \bitbox{32}{timestamp}
  \end{rightwordgroup} \\
  \bitbox{32}{synchronization source (SSRC) identifier} \\
  \wordbox[tlr]{1}{contributing source (CSRC) identifiers} \\
  \wordbox[blr]{1}{$\cdots$} \\
  \begin{rightwordgroup}{RTP \\ Payload}
    \wordbox[tlr]{3}{MPEG-4 Visual stream (byte aligned)} \\
    \bitbox[blr]{16}{}
    & \bitbox{16}{\dots\emph{optional} RTP padding}
  \end{rightwordgroup}
\end{bytefield}
```

Figure 1 presents the typeset output of the preceding code. Sections 2.2 and 2.3 explain each of the environments, macros, and arguments that were utilized plus many additional features of the bytefield package.

2.2 Basic commands

This section explains how to use the bytefield package. It lists all of the exported macros and environments in approximately decreasing order of usefulness.

```
\begin{bytefield} [⟨options⟩] {⟨bit-width⟩}
⟨fields⟩
\end{bytefield}
```

The bytefield package's top-level environment is called, not surprisingly, "`bytefield`". It takes one mandatory argument, which is the number of bits in each word, and one optional argument, which is a comma-separated list of ⟨*key*⟩=⟨*value*⟩ pairs, described in Section 2.3, for formatting the bit-field's layout. One can think of a `bytefield` as being analogous to a `tabular`: words are separated by "`\\`", and fields within a word are separated by "`&`". As in a `tabular`, "`\\`" accepts a ⟨*length*⟩ as an optional argument, and this specifies the amount of additional vertical whitespace to include after the current word is typeset.

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

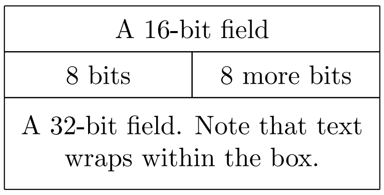
The two main commands one uses within a bytefield environment are `\bitbox` and `\wordbox`. The former typesets a field that is one or more bits wide and a single word tall. The latter typesets a field that is an entire word wide and one or more words tall.

The first, 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). The second, required, argument is the width in bits of a bit box or the height in words of a word box. The third argument is an optional, comma-separated list of ⟨*key*⟩=⟨*value*⟩ pairs, described in Section 2.3. The fourth, required, argument is the text to typeset within the box. 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 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}
```

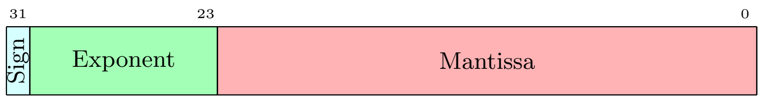
The resulting bit field looks like this:



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); bytefield does not currently check for under- or overruns.

Here's an example of using the `bgcolor` option to fill each box with a different color:

```
\definecolor{lightcyan}{rgb}{0.84,1,1}
\definecolor{lightgreen}{rgb}{0.64,1,0.71}
\definecolor{lightred}{rgb}{1,0.7,0.71}
\begin{bytefield}[bitheight=\widthof{~Sign~},
                  boxformatting={\centering\small}]{32}
  \bitheader[endianness=big]{31,23,0} \\
  \bitbox{1}[bgcolor=lightcyan]{\rotatebox{90}{Sign}} &
  \bitbox{8}[bgcolor=lightgreen]{Exponent} &
  \bitbox{23}[bgcolor=lightred]{Mantissa}
\end{bytefield}
```

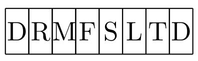


Within a `\bitbox` or `\wordbox`, the bytefield package defines `\height`, `\depth`, `\totalheight`, and `\width` to the corresponding dimensions of the box. Section 2.4 gives an example of how these lengths may be utilized.

`\bitboxes` [⟨*sides*⟩] {⟨*width*⟩} [⟨*options*⟩] {⟨*tokens*⟩}
`\bitboxes*` [⟨*sides*⟩] {⟨*width*⟩} [⟨*options*⟩] {⟨*tokens*⟩}

The `\bitboxes` command provides a shortcut for typesetting a sequence of fields of the same width. It takes essentially the same arguments as `\bitbox` but interpets these differently. Instead of representing a single piece of text to typeset within a field of width ⟨*width*⟩, `\bitboxes`'s ⟨*tokens*⟩ argument represents a list of tokens (e.g, individual characters), each of which is typeset within a separate box of width ⟨*width*⟩. Consider, for example, the following sequence of `\bitbox` commands:

```
\begin{bytefield}{8}
  \bitbox{1}{D} & \bitbox{1}{R} & \bitbox{1}{M} & \bitbox{1}{F} &
  \bitbox{1}{S} & \bitbox{1}{L} & \bitbox{1}{T} & \bitbox{1}{D}
\end{bytefield}
```

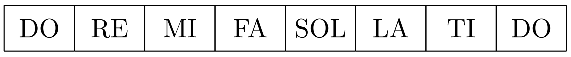


With `\bitboxes` this can be abbreviated to

```
\begin{bytefield}{8}
  \bitboxes{1}{DRMFSLTD}
\end{bytefield}
```

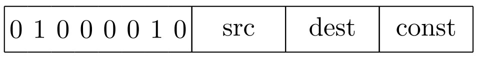
Spaces are ignored within `\bitboxes`'s ⟨*text*⟩ argument, and curly braces can be used to group multiple characters into a single token:

```
\begin{bytefield}{24}
  \bitboxes{3}{{DO} {RE} {MI} {FA} {SOL} {LA} {TI} {DO}}
\end{bytefield}
```



The starred form of `\bitboxes` is identical except that it suppresses all internal vertical lines. It can therefore be quite convenient for typesetting binary constants:

```
\begin{bytefield}{16}
  \bitboxes*{1}{01000010} & \bitbox{4}{src\strut} &
  \bitbox{4}{dest\strut} & \bitbox{4}{const\strut}
\end{bytefield}
```



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

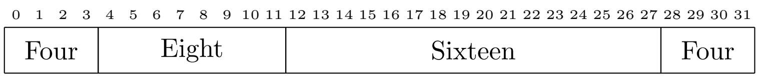
To make the bit field more readable, it helps to label bit positions across the top. The `\bitheader` command provides a flexible way to do that. The optional argument is a comma-separated list of ⟨*key*⟩=⟨*value*⟩ pairs from the set described in Section 2.3. In practice, the only parameters that are meaningful in the context of `\bitheader` are `bitformatting`, `endianness`, and `lsb`. See Section 2.3 for descriptions and examples of those parameters.

`\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 the `endianness` parameter 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. Ranges and single numbers can even 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 bit field looks like this:



```
\begin{rightwordgroup} [⟨options⟩] {⟨text⟩}
⟨rows of bit boxes and word boxes⟩
\end{rightwordgroup}

\begin{leftwordgroup} [⟨options⟩] {⟨text⟩}
⟨rows of bit boxes and word boxes⟩
\end{leftwordgroup}
```

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

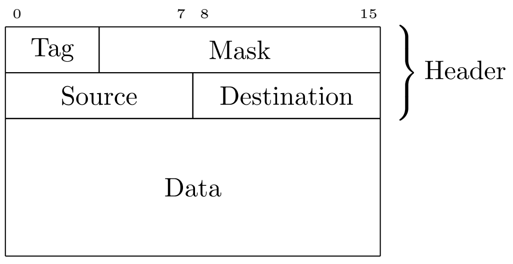
The optional argument is a comma-separated list of ⟨*key*⟩=⟨*value*⟩ pairs from the set described in Section 2.3. In practice, only `curlystyle`, `leftcurlystyle`, and `rightcurlystyle`, make sense within the context of a `\begin{`⟨*side*⟩`wordgroup}`.

Unlike other LaTeX environments, `rightwordgroup` and `leftwordgroup` do not have to nest properly with each other. However, they cannot overlap themselves. In other words, `\begin{rightwordgroup}`... `\begin{leftwordgroup}`...`\end{rightwordgroup}`...`\end{leftwordgroup}` is a valid sequence, but `\begin{rightwordgroup}`...`\begin{rightwordgroup}`... `\end{rightwordgroup}`...`\end{rightwordgroup}` is not.

The following example presents the basic usage of `\begin{rightwordgroup}` and `\end{rightwordgroup}`:

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

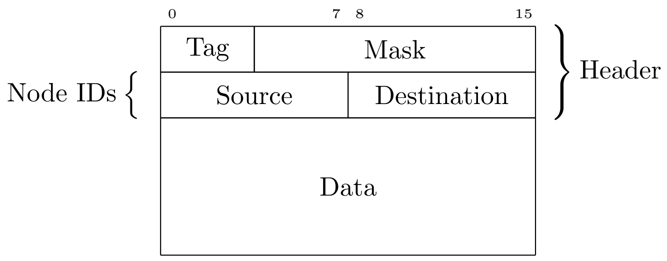
Note the juxtaposition of "\\" to the `\begin{rightwordgroup}` and the `\end{rightwordgroup}` in the above. The resulting bit field looks like this:



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

```
\begin{bytefield}{16}
```

```
  \bitheader{0,7,8,15} \\
  \begin{rightwordgroup}{Header}
    \bitbox{4}{Tag} & \bitbox{12}{Mask} \\
    \begin{leftwordgroup}{Node IDs}
      \bitbox{8}{Source} & \bitbox{8}{Destination}
    \end{leftwordgroup}
  \end{rightwordgroup} \\
  \wordbox{3}{Data}
\end{bytefield}
```

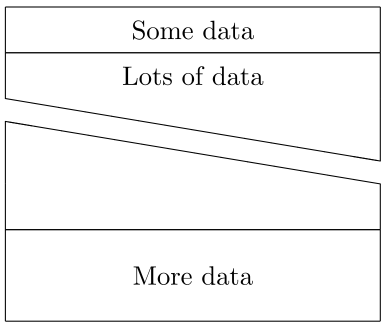


Because `rightwordgroup` and `leftwordgroup` are not required to nest properly, the resulting bit field would look the same if the `\end{leftwordgroup}` and `\end{rightwordgroup}` were swapped. Again, note the justaposition of "`\\`" to the various word-grouping commands in the above.

`\skippedwords`

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

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



```
\bytefieldsetup {⟨options⟩}
```

Alter the formatting of all subsequent bit fields. Section 2.3 describes the possible values for each ⟨*key*⟩=⟨*value*⟩ pair in the comma-separated list that `\bytefieldsetup` accepts as its argument. Note that changes made with `\bytefieldsetup` are local to their current scope. Hence, if used within an environment (e.g., `figure`), `\bytefieldsetup` does not impact bit fields drawn outside that environment.

2.3 Formatting options

A document author can customize many of the bytefield package's figure-formatting parameters, either globally or on a per-figure basis. The parameters described below can be specified in six locations:

- as package options (i.e., in the `\usepackage[`⟨*options*⟩`]{bytefield}` line), which affects all `bytefield` environments in the entire document,
- anywhere in the document using the `\bytefieldsetup` command, which affects all subsequent `bytefield` environments in the current scope,
- as the optional argument to a `\begin{bytefield}`, which affects only that single bit-field figure, or
- as the optional argument to a `\bitheader`, which affects only that particular header. (Only a few parameters are meaningful in this context.)
- as the optional argument to a `\begin{leftwordgroup}` or `\begin{rightwordgroup}`, which affects only that particular word group. (Only a few parameters are meaningful in this context.)
- as the second optional argument to a `\bitbox`, `\wordbox`, or `\bitboxes`, which affects only that particular box. (Only a few parameters are meaningful in this context.)

Unfortunately, LaTeX tends to abort with a "`TeX capacity exceeded`" or "`Missing \endcsname inserted`" error when a control sequence (i.e., \⟨*name*⟩) or \⟨*symbol*⟩) is encountered within the optional argument to `\usepackage`. Hence, parameters that typically expect a control sequence in their argument—in particular, `bitformatting`, `boxformatting`, `leftcurly`, and `rightcurly`—should best be avoided within the `\usepackage[`⟨*options*⟩`]{bytefield}` line.

```
bitwidth = ⟨length⟩
bitheight = ⟨length⟩
```

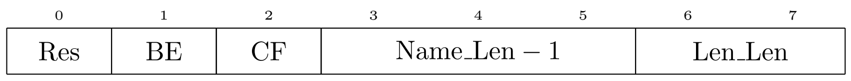
The above parameters represent the width and height of each bit in a bit field. The default value of `bitwidth` is the width of "`{\tiny 99i}`", i.e., the width of a two-digit number plus a small amount of extra space. This enables `\bitheader` to show two-digit numbers without overlap. The default value of `bitheight` is `2ex`, which should allow a normal piece of text to appear within a `\bitbox` or `\wordbox` without abutting the box's top or bottom edge.

As a special case, if `bitwidth` is set to the word "`auto`", it will be set to the width of "`99i`" in the current bit-number formatting (see `bitformatting` below). This feature provides a convenient way to adjust the bit width after a formatting change.

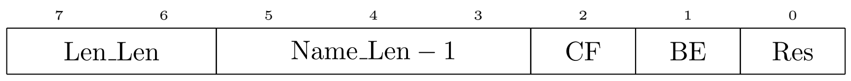
`endianness` = `little` *or* `big`

Specify either little-endian (left-to-right) or big-endian (right-to-left) ordering of the bit numbers. The default is little-endian numbering. Contrast the following two examples. The first formats a bit field in little-endian ordering using an explicit `endianness=little`, and the second formats the same bit field in big-endian ordering using `endianness=big`.

```
\begin{bytefield}[endianness=little,bitwidth=0.11111\linewidth]{8}
  \bitheader{0-7} \\
  \bitbox{1}{Res} & \bitbox{1}{BE} & \bitbox{1}{CF}
  & \bitbox{3}{$\mbox{Name\_Len}-1$} & \bitbox{2}{Len\_Len} \\
\end{bytefield}
```



```
\begin{bytefield}[endianness=big,bitwidth=0.11111\linewidth]{8}
  \bitheader{0-7} \\
  \bitbox{2}{Len\_Len} & \bitbox{3}{$\mbox{Name\_Len}-1$}
  & \bitbox{1}{CF} & \bitbox{1}{BE} & \bitbox{1}{Res} \\
\end{bytefield}
```



`bitformatting` = ⟨*command*⟩ *or* `{`⟨*commands*⟩`}`

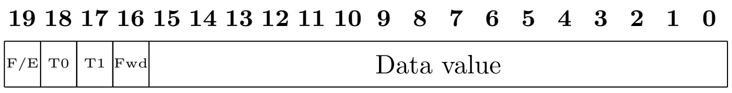
The numbers that appear in a bit header are typeset in the `bitformatting` style, which defaults to `\tiny`. To alter the style of bit numbers in the bit header, set `bitformatting` to a macro that takes a single argument (like `\textbf`) or no arguments (like `\small`). Groups of commands (e.g., `{\large\itshape}`) are also acceptable.

When `bitformatting` is set, `bitwidth` usually needs to be recalculated as well to ensure that a correct amount of spacing surrounds each number in the bit header. As described above, setting `bitwidth=auto` is a convenient shortcut for recalculating the bit-width in the common case of bit fields containing no more than 99 bits per line and no particularly wide labels in bit boxes that contain only a few bits.

The following example shows how to use `bitformatting` and `bitwidth` to format a bit header with small, boldface text:

```
\begin{bytefield}[bitformatting={\small\bfseries},
                  bitwidth=auto,
                  endianness=big]{20}
  \bitheader{0-19} \\
  \bitbox{1}{\tiny F/E} & \bitbox{1}{\tiny T0} & \bitbox{1}{\tiny T1}
  & \bitbox{1}{\tiny Fwd} & \bitbox{16}{Data value} \\
\end{bytefield}
```

The resulting bit field looks like this:



`boxformatting` = ⟨*command*⟩ *or* {⟨*commands*⟩}

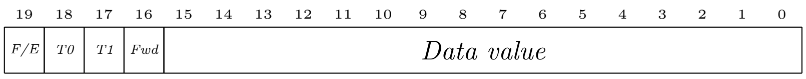
The text that appears in a `\bitbox` or `\wordbox` is formatted in the `boxformatting` style, which defaults to `\centering`. To alter the style of bit numbers in the bit header, set `boxformatting` to a macro that takes a single argument (like `\textbf` but not `\textbf`—see below) or no arguments (like `\small`). Groups of commands (e.g., `{\large\itshape}`) are also acceptable.

If `boxformatting` is set to a macro that takes an argument, the macro must be defined as a "long" macro, which means it can accept more than one paragraph as an argument. Commands defined with `\newcommand` are automatically made long, but commands defined with `\newcommand*` are not. LaTeX's `\text`... formatting commands (e.g., `\textbf`) are not long and therefore cannot be used directly in `boxformatting`; use the zero-argument versions (e.g., `\bfseries`) instead.

The following example shows how to use `boxformatting` to format the text within each box horizontally centered and italicized:

```
\begin{bytefield}[boxformatting={\centering\itshape},
                  bitwidth=1.5em,
                  endianness=big]{20}
  \bitheader{0-19} \\
  \bitbox{1}{\tiny F/E} & \bitbox{1}{\tiny T0} & \bitbox{1}{\tiny T1}
  & \bitbox{1}{\tiny Fwd} & \bitbox{16}{Data value} \\
\end{bytefield}
```

The resulting bit field looks like this:



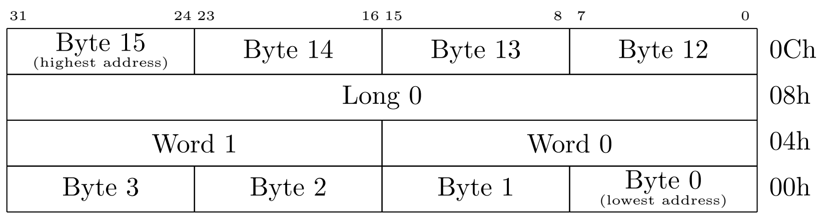
`bgcolor` = ⟨*color*⟩

Bit and word boxes are normally left unfilled. The `bgcolor` option fills them with a specified background color. A document will need to include the color, xcolor, or similar package to expose color names to bytefield. The `boxformatting` option described above can be used to set the foreground color.

`leftcurly` = ⟨*delimiter*⟩
`rightcurly` = ⟨*delimiter*⟩

Word groups are normally indicated by a curly brace spanning all of its rows. However, the curly brace can be replaced by any other extensible math delimiter (i.e., a symbol that can meaningfully follow `\left` or `\right` in math mode) via a suitable redefinition of `leftcurly` or `rightcurly`. As in math mode, "`.`" means "no symbol", as in the following example (courtesy of Steven R. King):

```
\begin{bytefield}[rightcurly=., rightcurlyspace=0pt]{32}
  \bitheader[endianness=big]{0,7,8,15,16,23,24,31} \\
  \begin{rightwordgroup}{0Ch}
    \bitbox{8}{Byte 15 \\ \tiny (highest address)}
    & \bitbox{8}{Byte 14}
    & \bitbox{8}{Byte 13}
    & \bitbox{8}{Byte 12}
  \end{rightwordgroup} \\
  \begin{rightwordgroup}{08h}
    \bitbox{32}{Long 0}
  \end{rightwordgroup} \\
  \begin{rightwordgroup}{04h}
    \bitbox{16}{Word 1} & \bitbox{16}{Word 0}
  \end{rightwordgroup} \\
  \begin{rightwordgroup}{00h}
    \bitbox{8}{Byte 3}
    & \bitbox{8}{Byte 2}
    & \bitbox{8}{Byte 1}
    & \bitbox{8}{Byte 0 \\ \tiny (lowest address)}
  \end{rightwordgroup}
\end{bytefield}
```



`leftcurlyspace` = ⟨*length*⟩
`rightcurlyspace` = ⟨*length*⟩
`curlyspace` = ⟨*length*⟩

`leftcurlyspace` and `rightcurlyspace` specify the space to insert between the bit field and the curly brace in a left or right word group (default: `1ex`). Setting `curlyspace` is a shortcut for setting both `leftcurlyspace` and `rightcurlyspace` to the same value.

`leftlabelspace` = ⟨*length*⟩
`rightlabelspace` = ⟨*length*⟩
`labelspace` = ⟨*length*⟩

`leftlabelspace` and `rightlabelspace` specify the space to insert between the curly brace and the text label in a left or right word group (default: `0.5ex`). Setting `labelspace` is a shortcut for setting both `leftlabelbrace` and `rightlabelspace` to the same value.

Figure 2 illustrates the juxtaposition of `rightcurlyspace` and `rightlabelspace` to a word group and its label. The `leftcurlyspace` and `leftlabelspace` parameters are symmetric.

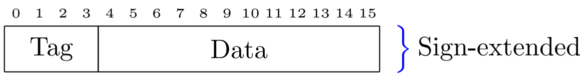
`leftcurlyshrinkage` = ⟨*length*⟩
`rightcurlyshrinkage` = ⟨*length*⟩
`curlyshrinkage` = ⟨*length*⟩

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. `leftcurlyshrinkage`/`rightcurlyshrinkage` is an amount by which to reduce the height of the curly brace in a left/right word group's label. Setting `curlyshrinkage` is a shortcut for setting both `leftcurlyshrinkage` and `rightcurlyshrinkage` to the same value. Shrinkages default to `5pt`, and it is extremely unlikely that one would ever need to change them. Nevertheless, these parameters are included here in case a document is typeset with a math font containing radically different curly braces from the ones that come with TEX/LATEX or that replaces the curly braces (using `leftcurly`/`rightcurly`, described above) with symbols of substantially different heights.

`leftcurlystyle` = ⟨*command*⟩
`rightcurlystyle` = ⟨*command*⟩
`curlystyle` = ⟨*command*⟩

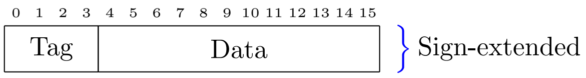
Provide a macro that will be invoked before the code that draws left, right, or both curly braces. The macro must accept either zero or one argument. It can be used, for example, to color the curly brace:

```
\begin{bytefield}[curlystyle=\color{blue}]{16}
  \bitheader{0-15} \\
  \begin{rightwordgroup}{Sign-extended}
      \bitbox{4}{Tag} & \bitbox{12}{Data}
  \end{rightwordgroup} \\
\end{bytefield}
```



or

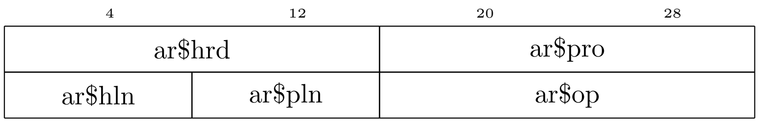
```
\begin{bytefield}{16}
  \bitheader{0-15} \\
  \begin{rightwordgroup}[curlystyle=\color{blue}]{Sign-extended}
      \bitbox{4}{Tag} & \bitbox{12}{Data}
  \end{rightwordgroup} \\
\end{bytefield}
```



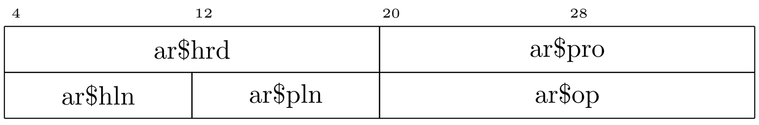
`lsb` = ⟨*integer*⟩

Designate the least significant bit (LSB) in the bit header. By default, the LSB is zero, which means that the first bit position in the header corresponds to bit 0. Specifying a different LSB shifts the bit header such that the first bit position instead corresponds to ⟨*integer*⟩. Note that the `lsb` option affects bit *positions* regardless of whether these positions are labeled, as demonstrated by the following two examples:

```
\begin{bytefield}{32}
  \bitheader[lsb=0]{4,12,20,28} \\
  \bitbox{16}{ar\$hrd} & \bitbox{16}{ar\$pro} \\
  \bitbox{8}{ar\$hln} & \bitbox{8}{ar\$pln} & \bitbox{16}{ar\$op} \\
\end{bytefield}
```



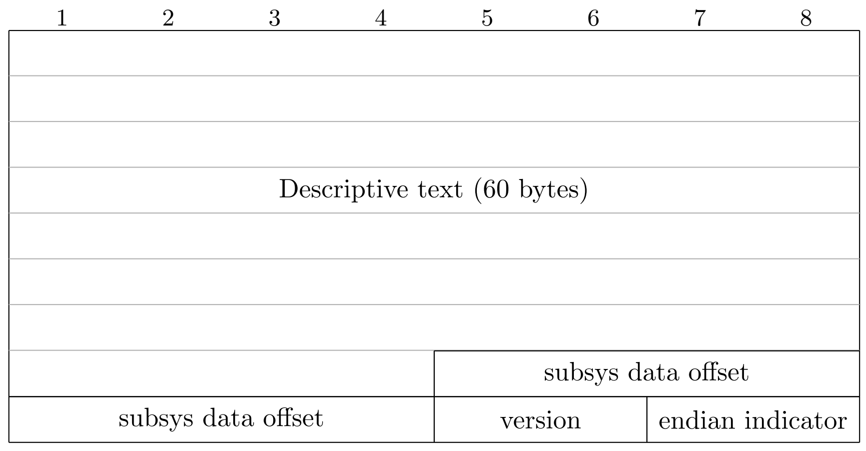
```
\begin{bytefield}{32}
  \bitheader[lsb=4]{4,12,20,28} \\
  \bitbox{16}{ar\$hrd} & \bitbox{16}{ar\$pro} \\
  \bitbox{8}{ar\$hln} & \bitbox{8}{ar\$pln} & \bitbox{16}{ar\$op} \\
\end{bytefield}
```



`perword` = ⟨*command*⟩

Provide a macro that will be invoked once for each word in a word box after the regular content is rendered. The macro will be passed two arguments: the word number (starting from 0) and the total number of words in the word box. Furthermore, the macro will be invoked within a one-word-wide box positioned at the base of the word. `perword` can therefore be used for delineating words within a word box, numbering words, or performing other such annotations. As a simple example, the following code draws a gray line at the bottom of each word in the "Descriptive text" word box:

```
\newcommand{\wordline}[2]{\color[rgb]{0.7,0.7,0.7}\hrulefill}
\begin{bytefield}[bitwidth=4em]{8}
  \bitheader[lsb=1,bitformatting=\small]{1-8} \\
  \wordbox[lrt]{7}[perword=\wordline]{Descriptive text (60 bytes)} \\
  \bitbox[lrb]{4}{} & \bitbox{4}{subsys data offset} \\
  \bitbox{4}{subsys data offset} & \bitbox{2}{version} &
    \bitbox{2}{endian indicator} \\
\end{bytefield}
```

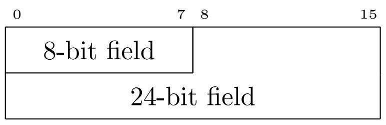


2.4 Common tricks

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

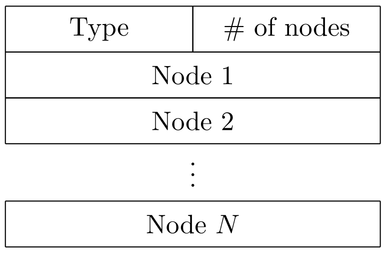
Odd-sized fields To produce a field that is, say, $1\frac{1}{2}$ words long, use a `\bitbox` for the fractional part and specify appropriate values for the various ⟨*sides*⟩ parameters. For instance:

```
\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 that appear the middle of enumerated data, put some `\vdots` in a `\wordbox` with empty ⟨*sides*⟩:

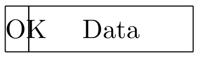
```
\begin{bytefield}{16}
  \bitbox{8}{Type} & \bitbox{8}{\# of nodes} \\
  \wordbox{1}{Node~1} \\
  \wordbox{1}{Node~2} \\
  \wordbox[]{1}{$\vdots$} \\[1ex]
  \wordbox{1}{Node~$N$}
\end{bytefield}
```



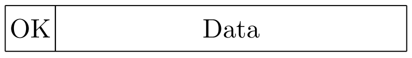
The extra `1ex` of vertical space helps vertically center the `\vdots` a bit better.

Narrow fields There are a number of options for labeling a narrow field (e.g., one occupying a single bit):

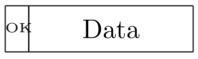
Default:



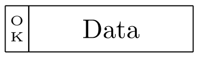
`\bytefieldsetup{%`
`  bitwidth=\widthof{OK~}}`:



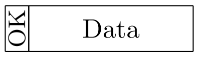
`\tiny OK`:



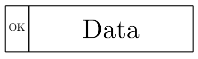
`\tiny O \\ K`:



`\rotatebox{90}{\small OK}`:

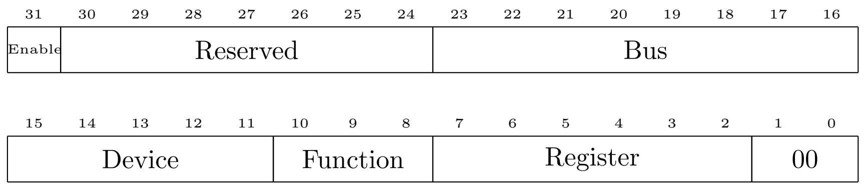


`\let\bw=\width`
`\resizebox{\bw}{!}{~OK~}`:



Multi-line bit fields Presentations of wide registers are often easier to read when split across multiple lines. (This capability was originally requested by Chris L'Esperance and is currently implemented in bytefield based on code provided by Renaud Pacalet.) The trick behind the typesetting of multi-line bit fields is to pass the `lsb` option to `\bitheader` to change the starting bit number used in each bit header:

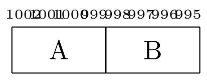
```
\begin{bytefield}[endianness=big,bitwidth=2em]{16}
  \bitheader[lsb=16]{16-31} \\
  \bitbox{1}{\tiny Enable} & \bitbox{7}{Reserved}
  & \bitbox{8}{Bus} \\[3ex]
  \bitheader{0-15} \\
  \bitbox{5}{Device} & \bitbox{3}{Function} & \bitbox{6}{Register}
  & \bitbox{2}{00}
\end{bytefield}
```



Note the use of the optional argument to `\\` to introduce three x-heights of additional whitespace between the two rows of bits.

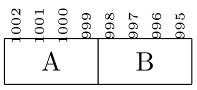
Rotated bit labels A problem with using very large bit numbers is that the labels run into each other, as in the following example:

```
\begin{bytefield}[endianness=big]{8}
  \bitheader[lsb=995]{995-1002} \\
  \bitbox{4}{A} & \bitbox{4}{B}
\end{bytefield}
```



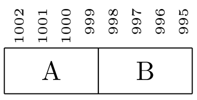
One solution is to use the `bitformatting` option and the graphicx package's `\rotatebox` command to rotate each bit label by 90°. Unfortunately, the naive use of `bitformatting` and `\rotatebox` does not typeset nicely:

```
\begin{bytefield}[endianness=big]{8}
  \bitheader[lsb=995,
             bitformatting={\tiny\rotatebox[origin=B]{90}}]{995-1002} \\
  \bitbox{4}{A} & \bitbox{4}{B}
\end{bytefield}
```



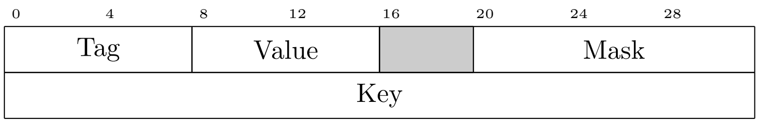
The two problems are that (1) the numbers are left-justified, and (2) the numbers touch the top margin of the word box. To address these problems we use `\makebox` to construct a right-justified region that is sufficiently wide to hold our largest number plus some additional space to shift the rotated numbers upwards:

```
\newlength{\bitlabelwidth}
\newcommand{\rotbitheader}[1]{%
  \tiny
  \settowidth{\bitlabelwidth}{\quad 9999}%
  \rotatebox[origin=B]{90}{\makebox[\bitlabelwidth][r]{#1}}%
}

\begin{bytefield}[endianness=big]{8}
  \bitheader[lsb=995,bitformatting=\rotbitheader]{995-1002} \\
  \bitbox{4}{A} & \bitbox{4}{B}
\end{bytefield}
```



Unused bits The `bgcolor` option can be used to represent unused bits by specifying a background fill color—light gray looks nice—and empty text:

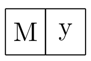
```
\definecolor{lightgray}{gray}{0.8}
\begin{bytefield}{32}
  \bitheader{0,4,8,12,16,20,24,28} \\
  \bitbox{8}{Tag} & \bitbox{8}{Value} &
  \bitbox{4}[bgcolor=lightgray]{} &
  \bitbox{12}{Mask} \\
  \wordbox{1}{Key}
\end{bytefield}
```



Aligning text on the baseline Because bytefield internally uses LaTeX's `picture` environment and that environment's `\makebox` command to draw bit boxes and word boxes, the text within a box is centered vertically with no attention paid to the text's baseline. As a result, some bit-field labels appear somewhat askew:

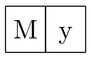
```
\begin{bytefield}[bitwidth=1.5em]{2}
```

```
  \bitbox{1}{M} & \bitbox{1}{y}
\end{bytefield}
```



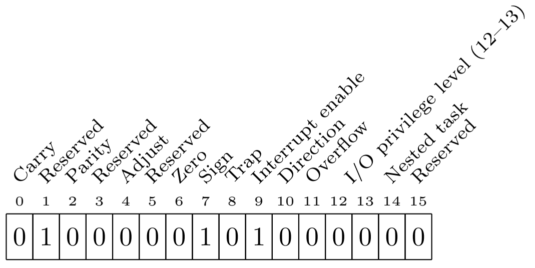
A solution is to use the `boxformatting` option to trick `\makebox` into thinking that all text has the same height and depth. Here we use `\raisebox` to indicate that all text is as tall as a "`W`" and does not descend at all below the baseline:

```
\newlength{\maxheight}
\setlength{\maxheight}{\heightof{W}}

\newcommand{\baselinealign}[1]{%
  \centering
  \raisebox{0pt}[\maxheight][0pt]{#1}%
}

\begin{bytefield}[boxformatting=\baselinealign,
                  bitwidth=1.5em]{2}
  \bitbox{1}{M} & \bitbox{1}{y}
\end{bytefield}
```



Register contents Sometimes, rather than listing the *meaning* of each bit field within each `\bitbox` or `\wordbox`, it may be desirable to list the *contents*, with the meaning described in an additional label above each bit number in the bit header. Although the register package is more suited to this form of layout, bytefield can serve in a pinch with the help of the `\turnbox` macro from the rotating package:

```
\newcommand{\bitlabel}[2]{%
  \bitbox[]{#1}{%
    \raisebox{0pt}[4ex][0pt]{%
      \turnbox{45}{\fontsize{7}{7}\selectfont#2}%
    }%
  }%
}

\begin{bytefield}[bitwidth=1em]{16}
   \bitlabel{1}{Carry} & \bitlabel{1}{Reserved} &
   \bitlabel{1}{Parity} & \bitlabel{1}{Reserved} &
   \bitlabel{1}{Adjust} & \bitlabel{1}{Reserved} &
   \bitlabel{1}{Zero} & \bitlabel{1}{Sign} &
   \bitlabel{1}{Trap} & \bitlabel{1}{Interrupt enable} &
   \bitlabel{1}{Direction} & \bitlabel{1}{Overflow} &
   \bitlabel{2}{I/O privilege level (12--13)} &
   \bitlabel{1}{Nested task} & \bitlabel{1}{Reserved} \\
   \bitheader{0-15} \\
   \bitbox{1}{0} & \bitbox{1}{1} & \bitbox{1}{0} & \bitbox{1}{0} &
   \bitbox{1}{0} & \bitbox{1}{0} & \bitbox{1}{0} & \bitbox{1}{1} &
   \bitbox{1}{0} & \bitbox{1}{1} & \bitbox{1}{0} & \bitbox{1}{0} &
   \bitbox{1}{0} & \bitbox{1}{0} & \bitbox{1}{0} & \bitbox{1}{0}
 \end{bytefield}
```

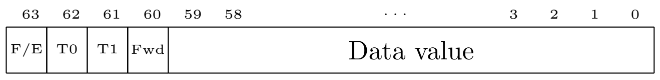


2.5 Not-so-common tricks

Omitted bit numbers It is occasionally convenient to show a wide bit field in which the middle numbers are replaced with an ellipsis. The trick to typesetting such a thing with bytefield is to point the `bitformatting` option to a macro that conditionally modifies the given bit number before outputting it. One catch is that bytefield measures the height of the string "1234567890" using the current bit formatting, so that needs to be a valid input. (If `bitwidth` is set to "`auto`", then "`99i`" also has to be a valid input, but we're not using "`auto`" here.) The following example shows how to *conditionally* modify the bit number: If the number is 1234567890, it is used as is; numbers greater than 9 are increased by 48; numbers less than 4 are unmodified; the number 6 is replaced by an ellipsis; and all other numbers are discarded.

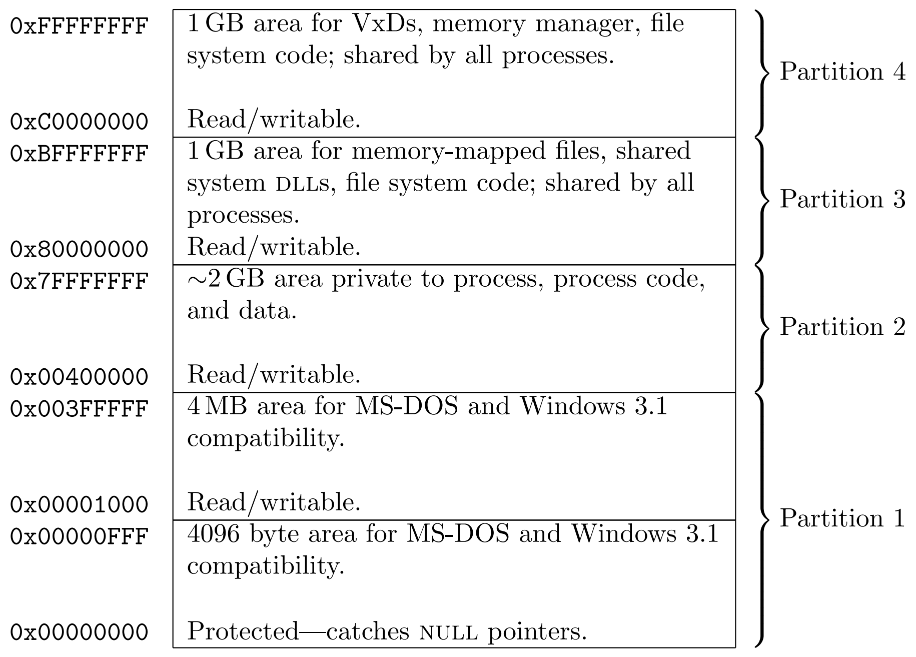
```
\newcommand{\fakesixtyfourbits}[1]{%
  \tiny
  \ifnum#1=1234567890
    #1
  \else
    \ifnum#1>9
      \count32=#1
      \advance\count32 by 48
      \the\count32%
    \else
      \ifnum#1<4
        #1%
      \else
        \ifnum#1=6
          $\cdots$%
        \fi
      \fi
    \fi
  \fi
}
\begin{bytefield}[%
```

```
    bitwidth=\widthof{\tiny Fwd~},
    bitformatting=\fakesixtyfourbits,
    endianness=big]{16}
  \bitheader{0-15} \\
  \bitbox{1}{\tiny F/E} & \bitbox{1}{\tiny T0} & \bitbox{1}{\tiny T1}
  & \bitbox{1}{\tiny Fwd} & \bitbox{12}{Data value}
\end{bytefield}
```



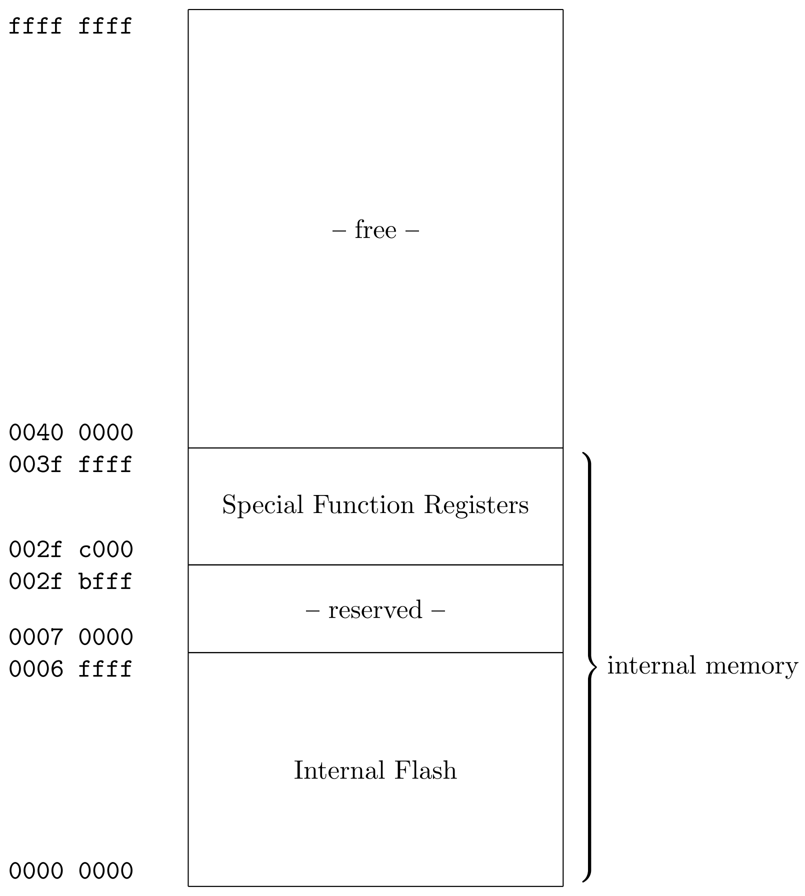
Memory-map diagrams While certainly not the intended purpose of the bytefield package, one can utilize word boxes with empty ⟨*sides*⟩ and word labels to produce memory-map diagrams:

```
\newcommand{\descbox}[2]{\parbox[c][3.8\baselineskip]{0.95\width}{%
  \raggedright #1\vfill #2}}
\begin{bytefield}[bitheight=4\baselineskip]{32}
  \begin{rightwordgroup}{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.}}
  \end{rightwordgroup} \\
  \begin{rightwordgroup}{Partition 3}
    \bitbox[]{8}{\texttt{0xBFFFFFFF} \\[2\baselineskip]
      \texttt{0x80000000}} &
    \bitbox{24}{\descbox{1\,GB area for memory-mapped files,
      shared system \textsc{dll}s, file system code; shared by all
      processes.}{Read/writable.}}
  \end{rightwordgroup} \\
  \begin{rightwordgroup}{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.}}
  \end{rightwordgroup} \\
  \begin{rightwordgroup}{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 \textsc{null}
      pointers.}}
  \end{rightwordgroup}
\end{bytefield}
```


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

The following variation uses variable-height regions in the memory map:

```
\newcommand{\memsection}[4]{%
  % define the height of the memsection
  \bytefieldsetup{bitheight=#3\baselineskip}%
  \bitbox[]{10}{%
    \texttt{#1}%      print end address
    \\
    %   do some spacing
    \vspace{#3\baselineskip}
    \vspace{-2\baselineskip}
    \vspace{-#3pt}
    \texttt{#2}%      print start address
  }%
  \bitbox{16}{#4}%    print box with caption
}

\begin{bytefield}{24}
        \memsection{ffff ffff}{0040 0000}{15}{-- free --}\\
        \begin{rightwordgroup}{internal memory}
                \memsection{003f ffff}{002f c000}{4}{Special Function
                  Registers}\\
                \memsection{002f bfff}{0007 0000}{3}{-- reserved --}\\
                \memsection{0006 ffff}{0000 0000}{8}{Internal Flash}
        \end{rightwordgroup}\\
\end{bytefield}
```



2.6 Putting it all together

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

```
\begin{bytefield}[bitheight=2.5\baselineskip]{32}
  \bitheader{0,7,8,15,16,23,24,31} \\
  \begin{rightwordgroup}{\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}
  \end{rightwordgroup} \\
  \wordbox[lrt]{1}{Data octets} \\
  \skippedwords \\
  \wordbox[lrb]{1}{} \\
  \begin{leftwordgroup}{\parbox{6em}{\raggedright Note that we can display,
    for example, a misaligned 64-bit value with clever use of the
```

```
      optional argument to \texttt{\string\wordbox} and
      \texttt{\string\bitbox}.}}
      \bitbox{8}{Source} & \bitbox{8}{Destination} &
        \bitbox[lrt]{16}{} \\
      \wordbox[lr]{1}{Timestamp} \\
      \begin{rightwordgroup}{\parbox{6em}{\raggedright Why two Length fields?
        No particular reason.}}
        \bitbox[lrb]{16}{} & \bitbox{16}{Length}
    \end{leftwordgroup} \\
        \bitbox{6}{Key} & \bitbox{6}{Value} & \bitbox{4}{Unused} &
          \bitbox{16}{Length}
      \end{rightwordgroup} \\
    \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}{\[ \mbox{A5A5}_{\mbox{\scriptsize H}} \oplus
      \left(\sum_{i=1}^N \mbox{Data}_i \right) \bmod 2^{20} \]} &
      \bitboxes*{1}{000010 000110} \\
    \wordbox{2}{64-bit random number}
  \end{bytefield}
```

Figure 3 shows the resulting protocol diagram.

2.7 Upgrading from older versions

bytefield's user interface changed substantially with the introduction of version 2.0. Because documents written for bytefield v1.x will not build properly under later versions of the package, this section explains how to convert documents to the new interface.

```
\wordgroupr
\endwordgroupr
```

These have been replaced with the `rightwordgroup` environment to make their invocation more LaTeX-like. Use `\begin{rightwordgroup}` instead of `\wordgroupr` and `\end{rightwordgroup}` instead of `\endwordgroupr`.

```
\wordgroupl
\endwordgroupl
```

These have been replaced with the `leftwordgroup` environment to make their invocation more LaTeX-like. Use `\begin{leftwordgroup}` instead of `\wordgroupl` and `\end{leftwordgroup}` instead of `\endwordgroupl`.

```
\bitwidth
```

Instead of changing bit widths with `\setlength{\bitwidth}{`⟨*width*⟩`}`, use

`\bytefieldsetup{bitwidth=`⟨*width*⟩`}`.

```
\byteheight
```

Instead of changing bit heights with `\setlength{\byteheight}{`⟨*height*⟩`}`, use `\bytefieldsetup{bitheight=`⟨*height*⟩`}` (and note the change from "`byte`" to "`bit`" for consistency with `bitwidth`).

```
\curlyspace
\labelspace
```

Instead of using `\setlength{\curlyspace}{`⟨*dist*⟩`}` and `\setlength{\labelspace}{`⟨*dist*⟩`}` to alter the horizontal space that appears before and after a curly brace, use `\bytefieldsetup{curlyspace=`⟨*dist*⟩`}` and `\bytefieldsetup{labelspace=`⟨*dist*⟩`}`. Note that, as described in Section 2.2, left and right spacing can be set independently if desired.

```
\curlyshrinkage
```

Instead of using `\setlength{\curlyshrinkage}{`⟨*dist*⟩`}` to reduce the vertical space occupied by a curly brace, use `\bytefieldsetup{curlyshrinkage=`⟨*dist*⟩`}`. Note that, as described in Section 2.2, left and right curly-brace height can be reduced independently if desired.

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

The meaning of `\bitwidth`'s optional argument changed with bytefield v2.1. In older versions of the package, the optional argument was one of "`l`" or "`b`" for, respectively, little-endian or big-endian bit ordering. Starting with version 2.1, the optional argument can be any of the parameters described in Section 2.3 (but practically only `bitformatting`, `endianness`, and `lsb`). Hence, "`l`" should be replaced with `endianness=little` and "`b`" should be replaced with `endianness=big`. Although more verbose, these new options can be specified once for the entire document by listing them as package options or as arguments to `\bytefieldsetup`.

As a crutch to help build older documents with minimal modification, bytefield provides a `compat1` package option that restores the old interface. This option, invoked with `\usepackage[compat1]{bytefield}`, may disappear in a future version of the package and should therefore not be relied upon as a long-term approach to using bytefield.

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 Section 2). All other macros are used only internally by bytefield.

3.1 Required packages

Although `\widthof` and `\heightof` were introduced in June 1998, teTeX 2.0—still in widespread use at the time of this writing (2005)—ships with an earlier `calc.sty` in the `source` directory. Because a misconfigured system may find the `source` version of `calc.sty` we explicitly specify a later date when loading the `calc` package.

```
\RequirePackage{calc}[1998/07/07]
\RequirePackage{keyval}
```

3.2 Utility macros

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

`\bf@newdimen` `\allocationnumber` `\newdimen` defines new ⟨*dimen*⟩s globally. `\bf@newdimen` defines them locally. It simply merges LaTeX 2$_\varepsilon$'s `\newdimen` and `\alloc@` macros while omitting `\alloc@`'s "`\global`" declaration.

```
\def\bf@newdimen#1{\advance\count11 by 1
  \ch@ck1\insc@unt\dimen
  \allocationnumber=\count11
  \dimendef#1=\allocationnumber
  \wlog{\string#1=\string\dimen\the\allocationnumber\space (locally)}%
}
```

`\bf@newdimen` ε-TeX provides many more ⟨*dimen*⟩s than the original TeX's 255. When running newer versions of ε-TeX we rebind `\bf@newdimen` to `\newdimen`. If the etex package is loaded, however, we instead rebind `\bf@newdimen` to `\locdimen` to keep the allocation local. Finally, if we're not running ε-TeX we leave `\bf@newdimen` defined as above to help reduce register pressure when only 255 ⟨*dimen*⟩s are available.

```
\AtBeginDocument{%
  \expandafter\ifx\csname e@alloc\endcsname\relax
    \expandafter\ifx\csname locdimen\endcsname\relax
    \else
      \let\bf@newdimen=\locdimen
    \fi
  \else
    \let\bf@newdimen=\newdimen
  \fi
}
```

`\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
}
```

3.3 Top-level environment

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

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

★ bytefield (*env.*) \bits@wide \old@nl \amp The bytefield environment contains the layout of bits in a sequence of words. This is the main environment defined by the bytefield package. 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}[2][]{%
  \bf@bytefieldsetup{#1}%
  \renewcommand{\baselinestretch}{}%
  \selectfont
  \def\bits@wide{#2}%
  \let\old@nl=\\%
  \let\amp=&%
  \catcode`\&=10
  \openup -1pt
  \setlength{\bytefield@height}{0pt}%
  \setlength{\unitlength}{1pt}%
  \global\counting@wordstrue
  \begin{lrbox}{\entire@bytefield@picture}%
```

★ \\ We redefine \\ within the bytefield environment to make it aware of curly braces that surround the protocol diagram.

```
  \renewcommand{\\}[1][0pt]{%
    \unskip
    \vspace{##1}%
    \amp\show@wordlabelr\cr
    \ignorespaces\global\counting@wordstrue\make@lspace\amp}%

  \vbox\bgroup\ialign\bgroup##\amp##\amp##\cr\amp
}{%
  \unskip
  \amp\show@wordlabelr\cr\egroup\egroup
  \end{lrbox}%
  \usebox{\entire@bytefield@picture}%
}
```

3.4 Box-drawing macros

3.4.1 Drawing (proper)

`\bf@bitformatting` Format a bit number in the bit header. `\bf@bitformatting` may be redefined to take either a single argument (à la `\textbf`) or no argument (à la `\small`).

```
\newcommand*{\bf@bitformatting}{\tiny}
```

`\bf@boxformatting` Format the text within a bit box or word box. `\bf@boxformatting` takes either a single argument (à la `\textbf`) or no argument (à la `\small`). The text that follows `\bf@boxformatting` is guaranteed to be a group that ends in `\par`, so if `\bf@boxformatting` accepts an argument, the macro should be defined with `\long` (e.g., with `\newcommand` but not with `\newcommand*`).

```
\newcommand*{\bf@boxformatting}{\centering}
```

`\bf@bitwidth` Define 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{\bf@bitwidth}
\settowidth{\bf@bitwidth}{\bf@bitformatting{99i}}
```

`\bf@bitheight` This is the height of a single bit within the bit field.

```
\newlength{\bf@bitheight}
\setlength{\bf@bitheight}{4ex}
```

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

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

`\bf@call@box@cmd`
`\bf@call@box@func` Define any box-drawing macro that accepts the same set of four arguments. It takes as input the name of a macro that is defined with formal parameters `[#1]#2[#3]#4`. `\bf@call@box@cmd` then invokes that macro, passing it a set of lines to draw out of the set "`lrtbLRTB`" (`#1`), a number of bits or words (`#2`), a list of key/value pairs (`#3`), and arbitrary text to typeset (`#4`).

```
\newcommand*{\bf@call@box@cmd}[1]{%
  \def\bf@call@box@func{#1}%
  \bf@call@box@cmd@i
}
```

`\bf@call@box@cmd@i`
`\bf@call@box@arg@i`
`\bf@call@box@arg@ii` Store the set of lines and the bit/word count and invoke `\bf@call@box@cmd@ii`.

```
\newcommand*{\bf@call@box@cmd@i}[2][lrtb]{%
  \def\bf@call@box@arg@i{#1}%
  \def\bf@call@box@arg@ii{#2}%
  \bf@call@box@cmd@ii
}
```

`\bf@call@box@cmd@ii`
`\bf@call@box@arg@iii`
`\bf@call@box@arg@iv` Store the key/value parameters and the text to typeset then invoke the macro originally passed to `\bf@call@box@cmd`.

```
\newcommand*{\bf@call@box@cmd@ii}[2][]{%
  \def\bf@call@box@arg@iii{#1}%
  \def\bf@call@box@arg@iv{#2}%
  \bf@call@box@func
}
```

★ `\bitbox` Put some text (`#4`) 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`). Additional drawing parameters can be provided via another optional argument (`#3`).

```
\DeclareRobustCommand{\bitbox}{\bf@call@box@cmd{\bf@bitbox}}
```

`\bf@bitbox` Implement all of the `\bitbox` logic.

```
\def\bf@bitbox{%
  \bgroup
    \expandafter\bf@parse@bitbox@arg\expandafter{\bf@call@box@arg@i}%
    \setlength{\units@wide}{\bf@bitwidth * \bf@call@box@arg@ii}%
    \expandafter\bf@bytefieldsetup\expandafter{\bf@call@box@arg@iii}%
    \@ifundefined{bf@bgcolor}{%
    }{%
```

If `bgcolor` was specified, draw a colored rule of the full size of the box.

```
      \rlap{%
        \draw@bit@picture{\strip@pt\units@wide}{\strip@pt\bf@bitheight}{%
          \color{\bf@bgcolor}%
          \rule{\width}{\height}%
        }%
      }%
    }%
```

Draw the user-provided text on top of the rule (if any).

```
    \draw@bit@picture{\strip@pt\units@wide}{\strip@pt\bf@bitheight}{%
      \bf@call@box@arg@iv
    }%
  \egroup
  \ignorespaces
}
```

★ `\wordbox` Put some text (`#4`) 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`). Additional drawing parameters can be provided via another optional argument (`#3`).

```
\DeclareRobustCommand{\wordbox}{\bf@call@box@cmd{\bf@wordbox}}
```

`\bf@wordbox` Implement all of the `\wordbox` logic.

```
\def\bf@wordbox{%
  \bgroup
    \expandafter\bf@parse@bitbox@arg\expandafter{\bf@call@box@arg@i}%
    \setlength{\units@wide}{\bf@bitwidth * \bits@wide}%
    \setlength{\units@tall}{\bf@bitheight * \bf@call@box@arg@ii}%
    \expandafter\bf@bytefieldsetup\expandafter{\bf@call@box@arg@iii}%
    \@ifundefined{bf@bgcolor}{%
    }{%
```

If `bgcolor` was specified, draw a colored rule of the full size of the box.

```
      \rlap{%
        \draw@bit@picture{\strip@pt\units@wide}{\strip@pt\units@tall}{%
          \color{\bf@bgcolor}%
          \rule{\width}{\height}%
        }%
      }%
    }%
```

Draw the user-provided text on top of the rule (if any).

```
    \draw@bit@picture{\strip@pt\units@wide}{\strip@pt\units@tall}{%
      \bf@call@box@arg@iv
    }%
```

Invoke the user-provided `\bf@per@word` macro once per word.

```
    \@ifundefined{bf@per@word}{}{\bf@invoke@per@word{\bf@call@box@arg@ii}}%
  \egroup
  \ignorespaces
}
```

`\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{#1\unitlength}{%
      \bf@set@user@dimens{#1}{#2}%
      \bf@boxformatting{#3\par}}}}%
```

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}}%
    \global\counting@wordsfalse
  \fi
}
```

\bf@invoke@per@word Invoke \bf@per@word once per word, passing it the (0-indexed) word number and total number of words.

```
\newcommand{\bf@invoke@per@word}[1]{%
  \begin{picture}(0,0)%
    \@tempcnta=0
    \@tempdima=#1\bf@bitheight
```

Make various useful dimensions available to \bf@per@word.

```
    \bf@set@user@dimens{\strip@pt\units@wide}{\strip@pt\units@tall}%
    \loop
      \advance\@tempdima by -\bf@bitheight
      \bgroup
        \put(-\strip@pt\units@wide, \strip@pt\@tempdima){%
          \expandafter\bf@per@word\expandafter{\the\@tempcnta}{#1}%
        }%
      \egroup
      \advance\@tempcnta by 1\relax
      \ifnum#1>\@tempcnta
    \repeat
  \end{picture}%
}
```

\bf@set@user@dimens

★ \width
★ \height
★ \depth
★ \totalheight
★ \widthunits
★ \heightunits

Given a width in bits (#1) and a height in words (#2), make a number of box dimensions available to the author: \width, \height, \depth, \totalheight. Additionally, make the arguments available to the author via the \widthunits and \heightunits macros.

```
\newcommand{\bf@set@user@dimens}[2]{%
  \bf@newdimen\width
  \bf@newdimen\height
  \bf@newdimen\depth
  \bf@newdimen\totalheight
  \width=#1\unitlength
  \height=#2\unitlength
  \depth=0pt%
  \totalheight=#2\unitlength
  \def\widthunits{#1}%
  \def\heightunits{#2}%
}
```

★ \bitboxes
★ \bitboxes*

Put each token in #3 into 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). The *-form of the command omits interior left and right lines.

```
\DeclareRobustCommand{\bitboxes}{%
  \@ifstar
     {\bf@call@box@cmd{\bf@bitboxes@star}}%
     {\bf@call@box@cmd{\bf@bitboxes@no@star}}%
}
```

\bf@relax Define a macro that expands to \relax for use with \ifx tests against \bf@bitboxes@arg, which can contain either tokens to typeset or \relax.

```
\def\bf@relax{\relax}
```

`\bf@bitboxes@no@star` Implement the unstarred version of `\bitboxes`. This macro simply expands its text argument into a list of tokens followed by `\relax` then invokes `\bf@bitboxes@no@star@i`.

```
\def\bf@bitboxes@no@star{%
  \expandafter\bf@bitboxes@no@star@i\bf@call@box@arg@iv\relax
  \ignorespaces
}
```

`\bf@bitboxes@no@star@i` Walk the subsequent tokens one-by-one until `\relax` is encountered. For each token, invoke `\bf@bitbox` (the internal version of `\bitbox` for which `\bf@call@box@arg@`⟨*number*⟩ are all defined.

```
\def\bf@bitboxes@no@star@i#1{%
  \def\bf@call@box@arg@iv{#1}%
  \ifx\bf@call@box@arg@iv\bf@relax
    \let\next=\relax
  \else
    \bf@bitbox
    \let\next=\bf@bitboxes@no@star@i
  \fi
  \next
}
```

`\bf@bitboxes@star` `\bf@bitboxes@sides` Implement the starred version of `\bitboxes`. This macro simply stores the original ⟨*sides*⟩ argument in `\bf@bitboxes@sides`, expands its text argument into a list of tokens followed by two `\relax`es, and invokes `\bf@bitboxes@star@i`.

```
\def\bf@bitboxes@star{%
  \edef\bf@bitboxes@sides{\bf@call@box@arg@i}%
  \expandafter\bf@bitboxes@star@i\bf@call@box@arg@iv\relax\relax
  \ignorespaces
}
```

`\bf@bitboxes@star@i` `\bf@call@box@arg@iv` `\bf@bitboxes@arg@ii` `\next` Process the first token in the text argument passed to `\bitboxes*`. If it's also the last token (indicated by its being followed by `\relax`), draw an ordinary bit box with all sides present. If it's not the last token, draw a bit box with the right side suppressed and invoke `\bf@bitboxes@star@ii` on the remaining tokens.

```
\def\bf@bitboxes@star@i#1#2{%
  \def\bf@call@box@arg@iv{#1}%
  \def\bf@bitboxes@arg@ii{#2}%
  \ifx\bf@bitboxes@arg@ii\bf@relax
    \bf@bitbox
    \let\next=\relax
  \else
    \edef\bf@call@box@arg@i{\bf@bitboxes@sides R}%
    \bf@bitbox
    \def\next{\bf@bitboxes@star@ii{#2}}%
  \fi
  \next
}
```

`\bf@bitboxes@star@ii` `\bf@call@box@arg@iv` `\bf@bitboxes@arg@ii` `\next` Process the second and subsequent tokens in the text argument passed to `\bitboxes*`. If the next token in the stream is the final one (indicated by its being followed by `\relax`), draw a bit box with the left side suppressed. If it's not the final token, draw a bit box with both the left and right sides suppressed and invoke itself recursively on the remaining tokens.

```
\def\bf@bitboxes@star@ii#1#2{%
  \def\bf@call@box@arg@iv{#1}%
  \def\bf@bitboxes@arg@ii{#2}%
  \ifx\bf@bitboxes@arg@ii\bf@relax
    \edef\bf@call@box@arg@i{\bf@bitboxes@sides L}%
  \else
    \edef\bf@call@box@arg@i{\bf@bitboxes@sides LR}%
  \fi
  \ifx\bf@call@box@arg@iv\bf@relax
    \let\next=\relax
  \else
    \bf@bitbox
    \def\next{\bf@bitboxes@star@ii{#2}}%
  \fi
  \next
}
```

3.4.2 Parsing arguments

The macros in this section are used to parse the optional argument to `\bitbox`, `\wordbox`, and `\bitboxes`, which is some subset of {`l`, `r`, `t`, `b`, `L`, `R`, `T`, `B`} and defaults to "`lrtb`" for all three user macros. If the argument is empty, no lines are drawn. Lowercase letters in the argument display, respectively, the left, right, top, or bottom side of a box. Uppercase letters undo the effect of the corresponding, prior, lowercase letter and are used internally by `\bitboxes` to suppress internal left and right lines.

`\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
```

`\bf@parse@bitbox@arg`

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

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

`\bf@parse@bitbox@sides`

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

```
\def\bf@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
     \else
      \ifx#1T%
       \bitbox@topfalse
      \else
       \ifx#1B%
        \bitbox@bottomfalse
       \else
        \ifx#1L%
         \bitbox@leftfalse
        \else
         \ifx#1R%
          \bitbox@rightfalse
         \else
          \PackageWarning{bytefield}{Unrecognized box side ‘#1’}%
         \fi
        \fi
       \fi
      \fi
     \fi
    \fi
   \fi
  \fi
  \expandafter\bf@parse@bitbox@sides
 \fi
}
```

3.5 Skipped words

`\units@high` This is 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}{\bf@bitwidth * \bits@wide}%
 \setlength{\units@high}{1pt * \ratio{\units@wide}{6.0pt}}%
 \setlength{\units@tall}{#1 + \units@high}%
 \edef\num@wide{\strip@pt\units@wide}%
 \edef\num@tall{\strip@pt\units@tall}%
 \edef\num@high{\strip@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}}%
    \global\counting@wordsfalse
  \fi
}
```

3.6 Bit-position labels

`\bf@bit@endianness` bytefield can label bit headers in either little-endian $(0, 1, 2, \ldots, N-1)$ or big-endian $(N-1, N-2, N-3, \ldots, 0)$ fashion. The `\bf@bit@endianness` macro specifies which to use, either "`l`" for little-endian (the default) or "`b`" for big-endian.

```
\newcommand*{\bf@bit@endianness}{l}
```

`\bf@first@bit` Normally, bits are numbered starting from zero. However, `\bf@first@bit` can be altered (usually locally) to begin numbering from a different value.

```
\newcommand*{\bf@first@bit}{0}
```

★ `\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 `lsb` option to reverse the labels' direction.

```
\DeclareRobustCommand{\bitheader}[2][]{%
  \bf@parse@bitbox@arg{lrtb}%
  \setlength{\units@wide}{\bf@bitwidth * \bits@wide}%
  \setlength{\units@tall}{\heightof{\bf@bitformatting{1234567890}}}%
  \setlength{\units@high}{\units@tall * -1}%
  \bf@process@bitheader@opts{#1}%
  \begin{picture}(\strip@pt\units@wide,\strip@pt\units@tall)%
                 (0,\strip@pt\units@high)
    \bf@parse@range@list#2,X,
  \end{picture}%
  \ifcounting@words
    \inc@bytefield@height{\unitlength * \real{\strip@pt\units@tall}}%
    \global\counting@wordsfalse
  \fi
  \ignorespaces
}
```

`\bf@parse@range@list` This is helper function #1 for `\bitheader`. It parses a comma-separated list of ranges, calling `\bf@parse@range` on each range.

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

`\header@xpos`
`header@val`
`max@header@val`

Define some miscellaneous variables to be used internally by `\bf@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}
```

`\bf@parse@range`

This is helper function #2 for `\bitheader`. It parses a hyphen-separated pair of numbers (or a single number) and displays the number at the correct bit position.

```
\def\bf@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\bf@bit@endianness b%
        \setlength{\header@xpos}{%
          \bf@bitwidth * (\bits@wide - \value{header@val} + \bf@first@bit - 1)}%
      \else
        \setlength{\header@xpos}{\bf@bitwidth * (\value{header@val} - \bf@first@bit)}%
      \fi
      \put(\strip@pt\header@xpos,0){%
        \makebox(\strip@pt\bf@bitwidth,\strip@pt\units@tall){%
          \bf@bitformatting{\theheader@val}}}
      \addtocounter{header@val}{1}
  \repeat
}
```

`\bf@process@bitheader@opts`
`\KV@bytefield@l`
`\KV@bytefield@b`
`\KV@bytefield@l@default`
`\KV@bytefield@b@default`

This is helper function #3 for `\bitheader`. It processes the optional argument to `\bitheader`.

```
\newcommand*{\bf@process@bitheader@opts}{%
  \let\KV@bytefield@l=\KV@bitheader@l
  \let\KV@bytefield@b=\KV@bitheader@b
  \let\KV@bytefield@l@default=\KV@bitheader@l@default
  \let\KV@bytefield@b@default=\KV@bitheader@b@default
  \setkeys{bytefield}%
}
```

`\KV@bitheader@l`
`\KV@bitheader@b`

For backwards compatibility we also accept the (now deprecated) `l` as a synonym for `endianness=little` and `b` as a synonym for `endianness=big`. A typical document will specify an `endianness` option not as an argument to `\bitheader` but rather as a package option that applies to the entire document. If the `compat1` option was provided to bytefield (determined below by the existence of the `\curlyshrinkage` control word), we suppress the deprecation warning message.

```
\define@key{bitheader}{l}[true]{%
  \expandafter\ifx\csname curlyshrinkage\endcsname\relax
    \PackageWarning{bytefield}{%
      The "l" argument to \protect\bitheader\space is deprecated.\MessageBreak
      Instead, please use "endianness=little", which can\MessageBreak
      even be declared globally for the entire document.\MessageBreak
      This warning occurred}%
  \fi
  \def\bf@bit@endianness{l}%
}
```

```
\define@key{bitheader}{b}[true]{%
  \expandafter\ifx\csname curlyshrinkage\endcsname\relax
    \PackageWarning{bytefield}{%
      The "b" argument to \protect\bitheader\space is deprecated.\MessageBreak
      Instead, please use "endianness=big", which can\MessageBreak
      even be declared globally for the entire document.\MessageBreak
      This warning occurred}%
  \fi
  \def\bf@bit@endianness{b}%
}
```

3.7 Word labels

3.7.1 Curly-brace manipulation

`\bf@leftcurlyshrinkage` `\bf@rightcurlyshrinkage`

Reduce the height of a left (right) curly brace by `\bf@leftcurlyshrinkage` (`\bf@rightcurlyshrinkage`) 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, `\bf@leftcurlyshrinkage` and `\bf@rightcurlyshrinkage` can be modified.

```
\def\bf@leftcurlyshrinkage{5pt}
\def\bf@rightcurlyshrinkage{5pt}
```

`\bf@leftcurlyspace` `\bf@rightcurlyspace` `\bf@leftlabelspace` `\bf@rightlabelspace`

Define the amount of space to insert before a curly brace and before a word label (i.e., after a curly brace).

```
\def\bf@leftcurlyspace{1ex}
\def\bf@rightcurlyspace{1ex}
\def\bf@leftlabelspace{0.5ex}
\def\bf@rightlabelspace{0.5ex}
```

`\bf@leftcurly` `\bf@rightcurly`

Define the symbols to use as left and right curly braces. These symbols must be extensible math symbols (i.e., they will immediately follow `\left` or `\right` in math mode).

```
\let\bf@leftcurly=\{
\let\bf@rightcurly=\}
```

`\bf@leftcurlystyle` `\bf@rightcurlystyle`

Define the default formatting for left and right curly braces as "do nothing special".

```
\let\bf@leftcurlystyle=\relax
\let\bf@rightcurlystyle=\relax
```

`\curly@box`

Define a box in which to temporarily store formatted curly braces.

```
\newbox{\curly@box}
```

`\store@rcurly` `\curly@height` `\half@curly@height` `\curly@shift` `\old@axis`

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{%
  \begingroup
    \bf@newdimen\curly@height
    \setlength{\curly@height}{#2 - \bf@rightcurlyshrinkage}%
    \bf@newdimen\half@curly@height
    \setlength{\half@curly@height}{0.5\curly@height}%
    \bf@newdimen\curly@shift
    \setlength{\curly@shift}{\bf@rightcurlyshrinkage}%
    \setlength{\curly@shift}{\half@curly@height + 0.5\curly@shift}%
    \addtolength{\curly@shift}{-\fontdimen22\textfont2}%
    \global\sbox{#1}{\raisebox{\curly@shift}{%
      \bf@rightcurlystyle{%
        $\left.
        \vrule height\half@curly@height
               width 0pt
               depth\half@curly@height\right\bf@rightcurly$%
      }%
    }}%
  \endgroup
}
```

`\store@lcurly` `\curly@height` `\half@curly@height` `\curly@shift` These are the same as `\store@rcurly`, etc. but using a "{" instead of a "}".

```
\def\store@lcurly#1#2{%
  \begingroup
    \bf@newdimen\curly@height
    \setlength{\curly@height}{#2 - \bf@leftcurlyshrinkage}%
    \bf@newdimen\half@curly@height
    \setlength{\half@curly@height}{0.5\curly@height}%
    \bf@newdimen\curly@shift
    \setlength{\curly@shift}{\bf@leftcurlyshrinkage}%
    \setlength{\curly@shift}{\half@curly@height + 0.5\curly@shift}%
    \addtolength{\curly@shift}{-\fontdimen22\textfont2}%
    \global\sbox{#1}{\raisebox{\curly@shift}{%
      \bf@leftcurlystyle{%
        $\left\bf@leftcurly
        \vrule height\half@curly@height
               width 0pt
               depth\half@curly@height\right.$%
      }%
    }}%
  \endgroup
}
```

3.7.2 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 `\end{rightwordgroup}` defines it to output the word label and then reset itself to a no-op.

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

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

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

★ `rightwordgroup` *(env.)* Label the words defined between `\begin{rightwordgroup}` and `\end{rightwordgroup}` on the right side of the bit field. The first, optional, argument is a list of parameters, as defined in Section 2.3. The second, mandatory, 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{rightwordgroup}[2][]{%
```

We begin by ending the group that `\begin{rightwordgroup}` created. This lets the `rightwordgroup` environment span rows (because we're technically no longer within the environment).

```
  \endgroup
```

`\wordlabelr@start` `\wordlabelr@params` `\wordlabelr@text`

`\begin{rightwordgroup}` merely stores the starting height in `\wordlabelr@start` and the user-supplied text in `\wordlabelr@text`. `\end{rightwordgroup}` does most of the work.

```
  \global\wordlabelr@start=\bytefield@height
  \gdef\wordlabelr@params{#1}%
  \gdef\wordlabelr@text{#2}%
  \ignorespaces
}{%
```

`\wordlabelr@end`

Because we already ended the group that `\begin{rightwordgroup}` created we now have to begin a group for `\end{rightwordgroup}` to end.

```
  \begingroup
  \global\wordlabelr@end=\bytefield@height
```

`\show@wordlabelr`

Redefine `\show@wordlabelr` to output `\bf@rightcurlyspace` space, followed by a large curly brace (in `\curlybox`), followed by `\bf@rightlabelspace` 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}%
```

Evaluate any parameters passed to `\begin{rightwordgroup}` right before we render the curly brace.

```
    \expandafter\bf@bytefieldsetup\expandafter{\wordlabelr@params}%
    \store@rcurly{\curly@box}{\label@box@height}%
    \bf@newdimen\total@box@width
    \setlength{\total@box@width}{%
      \bf@rightcurlyspace +
      \widthof{\usebox{\curly@box}} +
      \bf@rightlabelspace +
      \label@box@width
    }%
    \begin{picture}(\strip@pt\total@box@width,0)
      \put(0,0){%
        \hspace*{\bf@rightcurlyspace}%
        \usebox{\curly@box}%
        \hspace*{\bf@rightlabelspace}%
        \makebox(\strip@pt\label@box@width,\strip@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{}}%
```

`\@currenvir` Because of our meddling with `\begingroup` and `\endgroup`, the current environment is all messed up. We therefore force the `\end{rightwordgroup}` to succeed, even if it doesn't match the preceding `\begin`.

```
  \def\@currenvir{rightwordgroup}%
  \ignorespacesafterend
}
```

3.7.3 Left-side labels

`\wordlabell@start` `\wordlabell@end` Declare the starting and ending height (in points) of the set of rows to be labeled on the left.

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

`\total@box@width` Declare the total width of the next label to typeset on the left of the bit field, 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 `\halign`ed bytefield table. It's normally a no-op, but `\begin{leftwordgroup}` defines it to output enough space for the next word label and then reset itself to a no-op.

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

★ `leftwordgroup` (*env.*) This environment is essentially the same as the `rightwordgroup` environment but puts the label on the left. However, the following code is not symmetric to that of `rightwordgroup`. The problem is that we encounter `\begin{leftwordgroup}` 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 `\end{leftwordgroup}`, 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. `\end{leftwordgroup}` outputs the label immediately. Even though `\end{leftwordgroup}` is called at the end of the *second* column, it `\put`s the label at 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 bit field or extend beyond the bit-field boundaries.

```
\newenvironment{leftwordgroup}[2][]{%
```

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

`\wordlabell@start` `\wordlabell@params` `\wordlabell@text` We store the starting height, optional parameters (see Section 2.3), and label text, all of which are needed by the `\end{leftwordgroup}`. We immediately parse the parameters because they may affect the `\store@lcurly` invocation below.

```
\global\wordlabell@start=\bytefield@height
\gdef\wordlabell@params{#1}%
\gdef\wordlabell@text{#2}%
\bf@bytefieldsetup{#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 `\end{leftwordgroup}`, 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}}%
\store@lcurly{\curly@box}{5cm}%
\setlength{\total@lbox@width}{%
  \bf@leftcurlyspace +
  \widthof{\usebox{\curly@box}} +
  \bf@leftlabelspace +
  \label@box@width}%
\global\total@lbox@width=\total@lbox@width
```

`\make@lspace` 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{}%
}%
```

We now end the group that `\begin{rightwordgroup}` created. This lets the `leftwordgroup` environment span rows (because we're technically no longer within the environment).

```
\endgroup
\ignorespaces
}{%
```

Because we already ended the group that `\begin{leftwordgroup}` created we have to start the `\end{leftwordgroup}` by beginning a group for `\end{leftwordgroup}` to end.

```
\begingroup
```

The `\end{leftwordgroup}` code is comparatively straightforward. We calculate the final height of the word group, and then output the label text, followed by `\bf@leftlabelspace` space, followed by a curly brace (now that we know how tall it's supposed to be), followed by `\bf@leftcurlyspace` 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. Before typesetting the curly brace we re-parse the optional parameters because we're in a new group from the one in which we parsed them before, and the parameters can affect the second `\store@lcurly` invocation just they could have affected the first.

```
  \global\wordlabell@end=\bytefield@height
  \bf@newdimen\starting@point
  \setlength{\starting@point}{%
    -\total@lbox@width - \bf@bitwidth*\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}%
  \expandafter\bf@bytefieldsetup\expandafter{\wordlabell@params}%
  \store@lcurly{\curly@box}{\label@box@height}%
  \begin{picture}(0,0)
    \put(\strip@pt\starting@point,0){%
      \makebox(\strip@pt\label@box@width,\strip@pt\label@box@height){%
        \usebox{\word@label@box}}%
      \hspace*{\bf@leftlabelspace}%
      \usebox{\curly@box}%
      \hspace*{\bf@leftcurlyspace}}
  \end{picture}%
```

`\@currenvir` Because of our meddling with `\begingroup` and `\endgroup`, the current environment is all messed up. We therefore force the `\end{leftwordgroup}` to succeed, even if it doesn't match the preceding `\begin`.

```
  \def\@currenvir{leftwordgroup}%
  \ignorespacesafterend
}
```

3.7.4 Scratch space

`\label@box@width` `\label@box@height` `\word@label@box` Declare some 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}
```

3.8 Compatibility mode

`\bf@enter@compatibility@mode@i` bytefield's interface changed substantially with the move to version 2.0. To give version 1.x users a quick way to build their old documents, we provide a version 1.x compatibility mode. We don't enable this by default because it exposes a number of extra length registers (a precious resource) and because we want to encourage users to migrate to the new interface.

```
\newcommand{\bf@enter@compatibility@mode@i}{%
```

`\bitwidth` `\byteheight` `\curlyspace` `\labelspace` `\curlyshrinkage` Define a handful of lengths that the user was allowed to `\setlength` explicitly in bytefield 1.x.

```
  \PackageInfo{bytefield}{Entering version 1 compatibility mode}%
  \newlength{\bitwidth}%
  \newlength{\byteheight}%
```

```
\newlength{\curlyspace}%
\newlength{\labelspace}%
\newlength{\curlyshrinkage}%
```

```
\settowidth{\bitwidth}{\tiny 99i}%
\setlength{\byteheight}{4ex}%
\setlength{\curlyspace}{1ex}%
\setlength{\labelspace}{0.5ex}%
\setlength{\curlyshrinkage}{5pt}%
```

`\newbytefield` `\endnewbytefield` `bytefield` (*env.*) Redefine the `bytefield` environment in terms of the existing (new-interface) `bytefield` environment. The difference is that the redefinition utilizes all of the preceding lengths.

```
\let\newbytefield=\bytefield
\let\endnewbytefield=\endbytefield
\renewenvironment{bytefield}[1]{%
  \begin{newbytefield}[%
    bitwidth=\bitwidth,
    bitheight=\byteheight,
    curlyspace=\curlyspace,
    labelspace=\labelspace,
    curlyshrinkage=\curlyshrinkage]{##1}%
}{%
  \end{newbytefield}%
}
```

`\wordgroupr` `\endwordgroupr` `\wordgroupl` `\endwordgroupl` Define `\wordgroupr`, `\endwordgroupr`, `\wordgroupl`, and `\endwordgroupl` in terms of the new `rightwordgroup` and `leftwordgroup` environments.

```
\def\wordgroupr{\begin{rightwordgroup}}
\def\endwordgroupr{\end{rightwordgroup}}
\def\wordgroupl{\begin{leftwordgroup}}
\def\endwordgroupl{\end{leftwordgroup}}
```

`\bytefieldsetup` Disable `\bytefieldsetup` in compatibility mode because it doesn't work as expected. (Every use of the compatibility-mode `bytefield` environment overwrites all of the figure-formatting values.)

```
\renewcommand{\bytefieldsetup}[1]{%
  \PackageError{bytefield}{%
    The \protect\bytefieldsetup\space macro is not available in\MessageBreak
    version 1 compatibility mode%
  }{%
    Remove [compat1] from the \protect\usepackage{bytefield} line to
    make \protect\bytefieldsetup\MessageBreak
    available to this document.\space\space (The document may also need
    to be modified to use\MessageBreak
    the new bytefield interface.)
  }%
}%
}
```

`\wordgroupr` `\endwordgroupr` `\wordgroupl` `\endwordgroupl` Issue a helpful error message for the commands that were removed in bytefield v2.0. While this won't help users whose first invalid action is to modify a no-longer-extant length register such as `\bitwidth` or `\byteheight`, it may benefit at least a few users who didn't realize that the bytefield interface has changed substantially with version 2.0.

```
\newcommand{\wordgroupr}{%
  \PackageError{bytefield}{%
    Macros \protect\wordgroupr, \protect\wordgroupl, \protect\endwordgroupr,
    \MessageBreak
    and \protect\endwordgroupl\space no longer exist%
  }{%
    Starting with version 2.0, bytefield uses \protect\begin{wordgroupr}...
    \MessageBreak
    \protect\end{wordgroupr} and \protect\begin{wordgroupl}...%
    \protect\end{wordgroupl}\MessageBreak
    to specify word groups and a new \protect\bytefieldsetup\space macro to
    \MessageBreak
    change bytefield's various formatting parameters.%
  }%
}
\let\endwordgroupr=\wordgroupr
\let\wordgroupl=\wordgroupr
\let\endwordgroupl=\wordgroupr
```

3.9 Option processing

We use the keyval package to handle option processing. Because all of bytefield's options have local impact, options can be specified either as package arguments or through the use of the `\bytefieldsetup` macro.

`\KV@bytefield@bitwidth` `\bf@bw@arg` `\bf@auto`

Specify the width of a bit number in the bit header. If the special value "`auto`" is given, set the width to the width of a formatted "`99i`".

```
\define@key{bytefield}{bitwidth}{%
  \def\bf@bw@arg{#1}%
  \def\bf@auto{auto}%
  \ifx\bf@bw@arg\bf@auto
    \settowidth{\bf@bitwidth}{\bf@bitformatting{99i}}%
  \else
    \setlength{\bf@bitwidth}{#1}%
  \fi
}
```

`\KV@bytefield@bf@bitheight`

Specify the height of a bit in a `\bitbox` or `\wordbox`.

```
\define@key{bytefield}{bitheight}{\setlength{\bf@bitheight}{#1}}
```

`\KV@bytefield@bitformatting`

Specify the style of a bit number in the bit header. This should be passed an expression that takes either one argument (e.g., `\textit`) or no arguments (e.g., `{\small\bfseries}`).

```
\define@key{bytefield}{bitformatting}{\def\bf@bitformatting{#1}}
```

`\KV@bytefield@boxformatting`

Specify a style to be applied to the contents of every bit box and word box. This should be passed an expression that takes either one argument (e.g., `\textit`) or no arguments (e.g., `{\small\bfseries}`).

```
\define@key{bytefield}{boxformatting}{\def\bf@boxformatting{#1}}
```

\KV@bytefield@leftcurly
\KV@bytefield@rightcurly
\bf@leftcurly
\bf@rightcurly

Specify the symbol to use for bracketing a left or right word group. This must be an extensible math delimiter (i.e., something that can immediately follow `\left` or `\right` in math mode).

```
\define@key{bytefield}{leftcurly}{\def\bf@leftcurly{#1}}
\define@key{bytefield}{rightcurly}{\def\bf@rightcurly{#1}}
```

\KV@bytefield@leftcurlyspace
\KV@bytefield@rightcurlyspace
\KV@bytefield@curlyspace
\bf@leftcurlyspace
\bf@rightcurlyspace

Specify the amount of space between the bit fields in a word group and the adjacent left or right curly brace. The `curlyspace` option is a shortcut that puts the same space before both left and right curly braces.

```
\define@key{bytefield}{leftcurlyspace}{\def\bf@leftcurlyspace{#1}}
\define@key{bytefield}{rightcurlyspace}{\def\bf@rightcurlyspace{#1}}
\define@key{bytefield}{curlyspace}{%
  \def\bf@leftcurlyspace{#1}%
  \def\bf@rightcurlyspace{#1}%
}
```

\KV@bytefield@leftlabelspace
\KV@bytefield@rightlabelspace
\KV@bytefield@labelspace
\bf@leftlabelspace
\bf@rightlabelspace

Specify the amount of space between a left or right word group's curly brace and the associated label text. The `labelspace` option is a shortcut that puts the same space after both left and right curly braces.

```
\define@key{bytefield}{leftlabelspace}{\def\bf@leftlabelspace{#1}}
\define@key{bytefield}{rightlabelspace}{\def\bf@rightlabelspace{#1}}
\define@key{bytefield}{labelspace}{%
  \def\bf@leftlabelspace{#1}%
  \def\bf@rightlabelspace{#1}%
}
```

V@bytefield@leftcurlyshrinkage
@bytefield@rightcurlyshrinkage
\KV@bytefield@curlyshrinkage
\bf@leftcurlyshrinkage
\bf@rightcurlyshrinkage

Specify the number of points by which to reduce the height of a curly brace (left, right, or both) so its ends don't overlap whatever's above or below it.

```
\define@key{bytefield}{leftcurlyshrinkage}{\def\bf@leftcurlyshrinkage{#1}}
\define@key{bytefield}{rightcurlyshrinkage}{\def\bf@rightcurlyshrinkage{#1}}
\define@key{bytefield}{curlyshrinkage}{%
  \def\bf@leftcurlyshrinkage{#1}%
  \def\bf@rightcurlyshrinkage{#1}%
}
```

\KV@bytefield@leftcurlystyle
\KV@bytefield@rightcurlystyle
\KV@bytefieldcurlystyle
\bf@leftcurlystyle
\bf@rightcurlystyle

Specify a macro that takes either zero or one argument and that precedes the text that draws a left curly brace, right curly brace, or either curly brace.

```
\define@key{bytefield}{leftcurlystyle}{\def\bf@leftcurlystyle{#1}}
\define@key{bytefield}{rightcurlystyle}{\def\bf@rightcurlystyle{#1}}
\define@key{bytefield}{curlystyle}{%
  \def\bf@leftcurlystyle{#1}%
  \def\bf@rightcurlystyle{#1}%
}
```

\KV@bytefield@endianness
\bf@parse@endianness

Set the default endianness to either little endian or big endian.

```
\define@key{bytefield}{endianness}{\bf@parse@endianness{#1}}
```

```
\newcommand{\bf@parse@endianness}[1]{%
  \def\bf@little{little}%
  \def\bf@big{big}%
  \def\bf@arg{#1}%
  \ifx\bf@arg\bf@little
    \def\bf@bit@endianness{l}%
  \else
    \ifx\bf@arg\bf@big
      \def\bf@bit@endianness{b}%
    \else
      \PackageError{bytefield}{%
        Invalid argument "#1" to the endianness option%
      }{%
        The endianness option must be set to either "little" or
        "big".\MessageBreak
        Please specify either endianness=little or endianness=big.
      }%
    \fi
  \fi
}
```

\KV@bytefield@lsb Specify a numerical value for the least significant bit of a word.

```
\define@key{bytefield}{lsb}{\def\bf@first@bit{#1}}
```

\bf@bgcolor \KV@bytefield@bgcolor Specify a background color for a bit box or word box.

```
\define@key{bytefield}{bgcolor}{\def\bf@bgcolor{#1}}
```

\bf@per@word \KV@bytefield@per@word Specify a macro to invoke for each word of a word box. The macro must take two arguments: the word number (0-indexed) and the total number of words.

```
\define@key{bytefield}{perword}{\def\bf@per@word{#1}}
```

★ \bytefieldsetup \bf@bytefieldsetup Reconfigure values for various bytefield parameters. Internally to the package we use the `\bf@bytefieldsetup` macro instead of `\bytefieldsetup`. This enables us to redefine `\bytefieldsetup` when entering version 1 compatibility mode without impacting the rest of bytefield.

```
\newcommand{\bf@bytefieldsetup}{\setkeys{bytefield}}
\let\bytefieldsetup=\bf@bytefieldsetup
```

We define only a single option that can be used only as a package option, not as an argument to `\bytefieldsetup`: `compat1` instructs bytefield to enter version 1 compatibility mode—at the cost of a number of additional length registers and the inability to specify parameters in the argument to the `bytefield` environment.

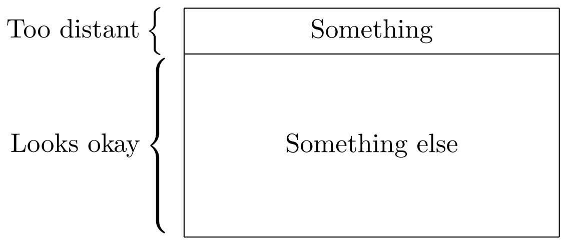
```
\DeclareOption{compat1}{\bf@enter@compatibility@mode@i}
```

\bf@package@options \next We want to use `\bf@bytefieldsetup` to process bytefield package options. Unfortunately, `\DeclareOption` doesn't handle ⟨*key*⟩=⟨*value*⟩ arguments. Hence, we use `\DeclareOption*` to catch *all* options, each of which it appends to `\bf@package@options`. `\bf@package@options` is passed to `\bf@bytefieldsetup` only at the beginning of the document so that the options it specifies (a) can refer to ex-heights and (b) override the default values, which are also set at the beginning of the document.

```
\def\bf@package@options{}
\DeclareOption*{%
  \edef\next{%
    \noexpand\g@addto@macro\noexpand\bf@package@options{,\CurrentOption}%
  }%
  \next
}
\ProcessOptions\relax
\expandafter\bf@bytefieldsetup\expandafter{\bf@package@options}
```

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. For example, bytefield is somewhat wasteful of ⟨*dimen*⟩ registers (although it did get a lot better with version 1.1 and again with version 1.3). The package should really get a major overhaul now that I've gotten better at TeX/LaTeX programming. One minor improvement I'd like to make in the package is to move left, small curly braces closer to the bit field. In the following figure, notice how distant the small curly appears from the bit-field body:



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

Change History

v1.0
  General: Initial version 1
v1.1
  General: Restructured the `.dtx` file 1
  `\allocationnumber`: Bug fix: Added `\bf@newdimen` to greatly reduce the likelihood of "`No room for a new \dimen`" errors (reported by Vitaly A. Repin) 25
  `\bf@parse@range@list`: Bug fix: Swapped order of arguments to `\ifx` test (suggested by Hans-Joachim Widmaier) . . . 34
v1.2
  `\curly@box`: Bug fix: Defined `\curly@box` globally (suggested by Stefan Ulrich) 36
v1.2a
  General: Specified an explicit package date when loading the calc package to avoid loading an outdated version. Thanks to Kevin Quick for discovering that outdated versions of calc are still being included in TeX distributions. 25
v1.3
  `\bf@newdimen`: Added support for ε-TeX's larger local ⟨*dimen*⟩ pool (code provided by Heiko Oberdiek) 25
v1.4
  General: Made assignments to `\counting@words` global to prevent vertical-spacing problems with back-to-back word groups (bug fix due to Steven R. King) 1
   Split `\curlyspace`, `\labelspace`, and `\curlyshrinkage` into `left` and `right` versions 1
  `\bf@bitformatting`: Introduced this macro at Steven R. King's request to enable users to alter the bit header's font size 27
v2.0
  General: Made a number of non-backwards-compatible changes, including replacing `\wordgroupr` and

\endwordgroupr with a rightwordgroup environment and \wordgroupl and \endwordgroupl with a leftwordgroup environment and also replacing a slew of user-visible lengths and macros with a single \bytefieldsetup macro 1

\bytefieldsetup: Introduced this macro to provide a more convenient way of configuring bytefield's parameters 45

v2.1

General: Included in the documentation a variable-height memory-map example suggested by Martin Demling 21

\\: Augmented the definition of \\ to accept an optional argument, just like in a tabular environment 26

\bf@parse@range: Added code due to Renaud Pacalet for shifting the bit header by a distance corresponding to \bf@first@bit, used for typesetting registers split across rows 35

\bitheader: Changed the optional argument to accept ⟨*key*⟩=⟨*value*⟩ pairs instead of just "l" and "b" 34

v2.2

\bitboxes: Added this macro based on an idea proposed by Andrew Mertz 30

v2.3

\bf@newdimen: Rewrote the macro based on discussions with David Carlisle to avoid producing "No room for a new \dimen" errors in newer versions of ε-TEX (see http://tex.stackexchange.com/q/275042) 25

v2.4

bytefield: Make the code resilient to changes in \baselinestretch. Thanks to Karst Koymans for the bug report 26

v2.5

General: Accept a new bgcolor option to set a bit's background color 1

Accept a new perword option to execute a macro for each word in a word box. This addresses a feature request by Victor Toni . 1

Redefine the \bitbox, \wordbox, and \bitboxes commands additionally to accept key/value options 1

v2.6

General: Accept new curlystyle, leftcurlystyle, and rightcurlystyle options to control the styling (e.g., color) of curly braces. This addresses a feature request by Victor Toni 1

leftwordgroup: Accept key/value options 39

Suppress spaces following the \end{leftwordgroup} 39

rightwordgroup: Accept key/value options 38

Suppress spaces following the \end{rightwordgroup} 38

v2.7

\store@lcurly: Properly align left curly braces under LuaLATEX (bug reported by Georgi Nikiforov) 37

\store@rcurly: Properly align right curly braces under LuaLATEX (bug reported by Georgi Nikiforov) 36

v2.8

General: Corrected the documentation. Thanks to Ömer Faruk Birgül for pointing out an error in one of the examples 1

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.

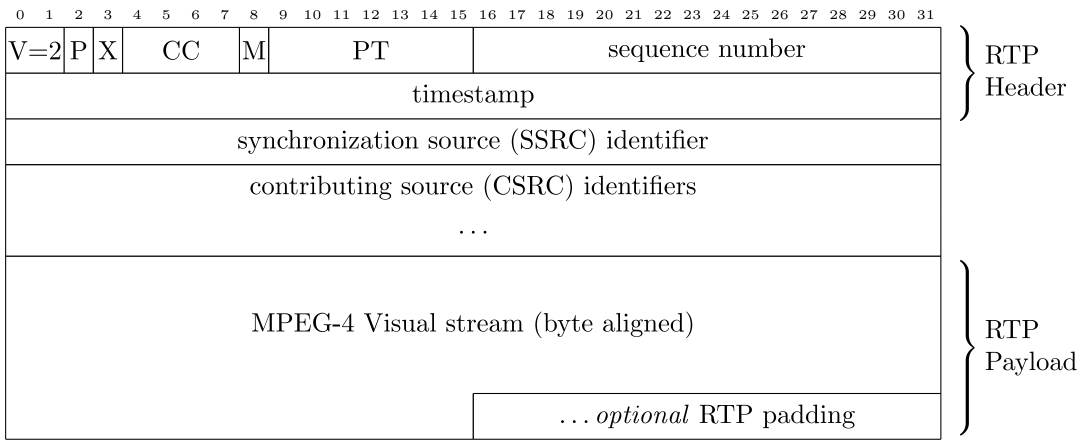


Figure 1: Sample bytefield output

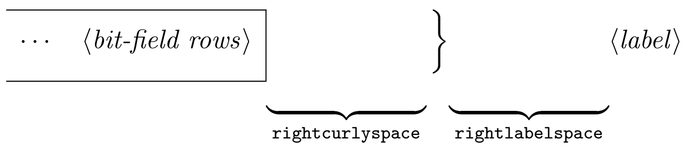


Figure 2: Role of `rightcurlyspace` and `rightlabelspace`

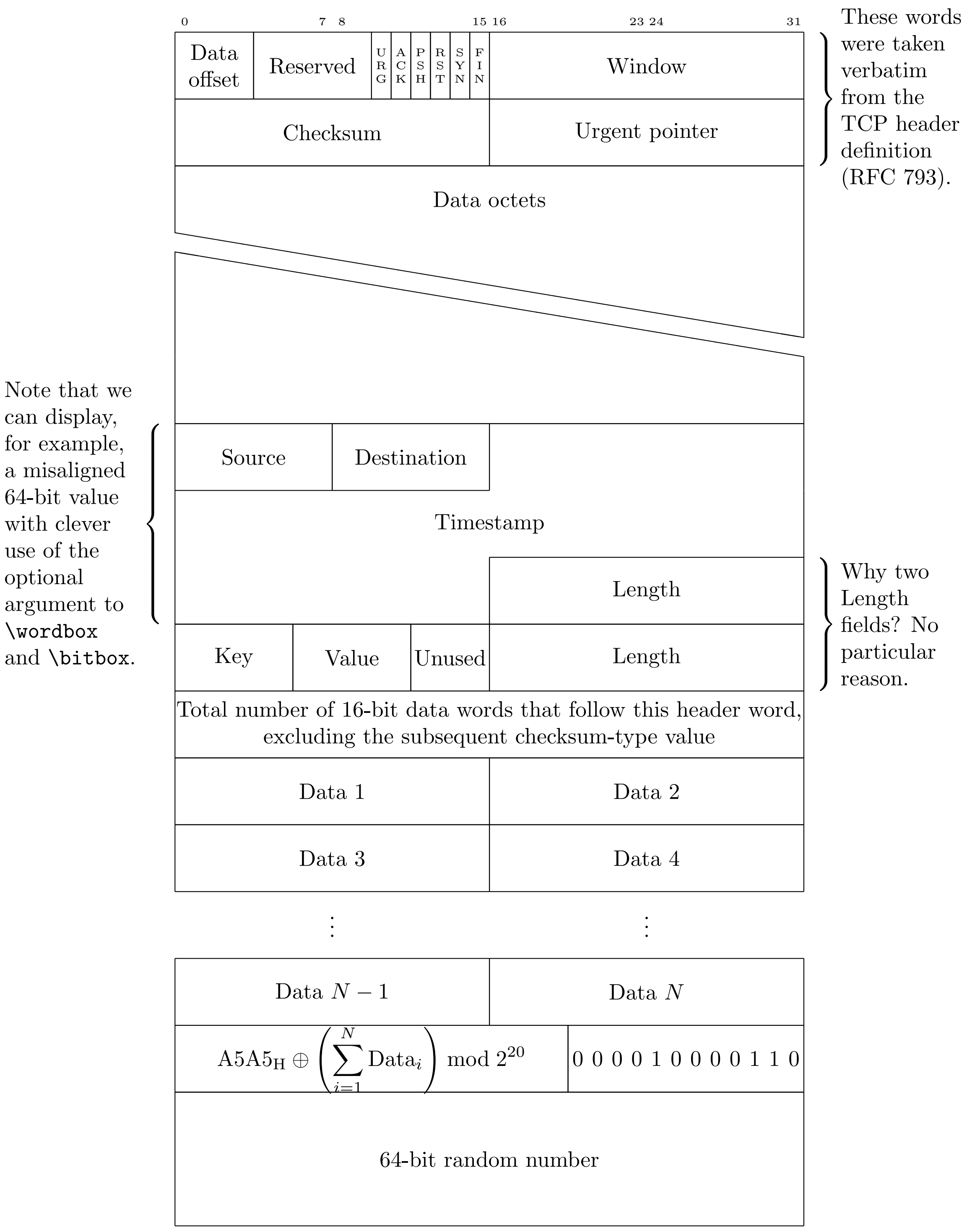

0
7 8
15 16
23 24
31
Data offset
Reserved
URG
ACK
PSH
RST
SYN
FIN
Window
Checksum
Urgent pointer
Data octets
These words were taken verbatim from the TCP header definition (RFC 793).
Note that we can display, for example, a misaligned 64-bit value with clever use of the optional argument to \wordbox and \bitbox.
Source
Destination
Timestamp
Length
Why two Length fields? No particular reason.
Key
Value
Unused
Length
Total number of 16-bit data words that follow this header word, excluding the subsequent checksum-type value
Data 1
Data 2
Data 3
Data 4
Data $N-1$
Data N
$\mathrm{A5A5_H} \oplus \left(\sum_{i=1}^{N} \mathrm{Data}_i \right) \bmod 2^{20}$
0 0 0 0 1 0 0 0 0 1 1 0
64-bit random number

Figure 3: Complex protocol diagram drawn with the bytefield package