Writing DDD Themes

User's Guide and Reference Manual
First Edition, for DDD Version 3.3.12-test5
Last updated 2001-02-01

Andreas Zeller

Writing DDD Themes
User's Guide and Reference Manual

Distributed by
Free Software Foundation, Inc.
59 Temple Place – Suite 330
Boston, MA 02111-1307
USA

DDD and this manual are available via
the DDD WWW page.

Send questions, comments, suggestions, etc. to ddd@gnu.org.
Send bug reports to bug-ddd@gnu.org.

Short Contents

Table of Contents

Welcome

Welcome to *Writing* DDD *Themes*! In this manual, we will sketch how data visualization in DDD works. (DDD, the Data Display Debugger, is a debugger front-end with data visualization. For details, see section "Summary of DDD" in *Debugging with DDD*.)

1 Creating Displays

We begin with a short discussion of how DDD actually creates displays from data.

1.1 Handling Boxes

All data displayed in the DDD data window is maintained by the inferior debugger. GDB, for instance, provides a *display list*, holding symbolic expressions to be evaluated and printed on standard output at each program stop. The GDB command '`display tree`' adds '`tree`' to the display list and makes GDB print the value of '`tree`' as, say, '`tree = (Tree *) 0x20e98`', at each program stop. This GDB output is processed by DDD and displayed in the data window.

Each element of the display list, as transmitted by the inferior debugger, is read by DDD and translated into a *box*. Boxes are rectangular entities with a specific content that can be displayed in the data window. We distinguish *atomic* boxes and *composite* boxes. An atomic box holds white or black space, a line, or a string. Composite boxes are horizontal or vertical alignments of other boxes. Each box has a size and an extent that determines how it fits into a larger surrounding space.

Through construction of larger and larger boxes, DDD constructs a graph node from the GDB data structure in a similar way a typesetting system like TEX builds words from letters and pages from paragraphs.

Such constructions are easily expressed by means of functions mapping boxes onto boxes. These *display functions* can be specified by the user and interpreted by DDD, using an applicative language called VSL for *visual structure language*. VSL functions can be specified by the DDD user, leaving much room for extensions and customization. A VSL display function putting a frame around its argument looks like this:

```
// Put a frame around TEXT
frame(text) = hrule()
  | vrule() & text & vrule()
  | hrule();
```

Here, `hrule()` and `vrule()` are primitive functions returning horizontal and vertical lines, respectively. The '`&`' and '`|`' operators construct horizontal and vertical alignments from their arguments.

VSL provides basic facilities like pattern matching and variable numbers of function arguments. The `halign()` function, for instance, builds a horizontal alignment from an arbitrary number of arguments, matched by three dots ('`...`'):

```
// Horizontal alignment
halign(x) = x;
halign(x, ...) = x & halign(...);
```

Frequently needed functions like `halign()` are grouped into a standard VSL library.

1.2 Building Boxes from Data

To visualize data structures, each atomic type and each type constructor from the programming language is assigned a VSL display function. Atomic values like numbers, characters, enumerations, or character strings are displayed using string boxes holding their value; the VSL function to display them leaves them unchanged:

```
// Atomic Values
simple_value(value) = value;
```

Composite values require more attention. An array, for instance, may be displayed using a horizontal alignment:

```
// Array
array(...) = frame(halign(...));
```

When GDB sends DDD the value of an array, the VSL function '`array()`' is invoked with array elements as values. A GDB array expression '`{1, 2, 3}`' is thus evaluated in VSL as

```
array(simple_value("1"), simple_value("2"), simple_value("3"))
```

which equals

```
"1" & "2" & "3"
```

a composite box holding a horizontal alignment of three string boxes. The actual VSL function used in DDD also puts delimiters between the elements and comes in a vertical variant as well.

Nested structures like multi-dimensional arrays are displayed by applying the `array()` function in a bottom-up fashion. First, `array()` is applied to the innermost structures; the resulting boxes are then passed as arguments to another `array()` invocation. The GDB output

```
{{"A", "B", "C"}, {"D", "E", "F"}}
```

representing a 2 * 3 array of character strings, is evaluated in VSL as

```
array(array("A", "B", "C"), array("A", "B", "C"))
```

resulting in a horizontal alignment of two more alignments representing the inner arrays.

Record structures are built in a similar manner, using a display function `struct\_member` rendering the record members. Names and values are separated by an equality sign:

```
// Member of a record structure
struct_member (name, value) =
  name & " = " & value;
```

The display function `struct` renders the record itself, using the `valign()` function.[1]

```
// Record structure
struct(...) = frame(valign(...));
```

This is a simple example; the actual VSL function used in DDD takes additional effort to align the equality signs; also, it ensures that language-specific delimiters are used, that collapsed structs are rendered properly, and so on.

[1] `valign()` is similar to `halign()`, but builds a vertical alignment.

2 Writing Themes

The basic idea of a *theme* is to customize one or more aspects of the visual appearance of data. This is done by *modifying* specific VSL definitions.

2.1 Example: Changing the Display Title Color

As a simple example, consider the following task: You want to display display titles in blue instead of black. The VSL function which handles the colors of display titles is called ‘`title_color`’ (see Section A.2 [Displaying Colors], page 10). It is defined as

```
title_color(box) = color(box, "black");
```

All you’d have to do to change the color is to provide a new definition:

```
title_color(box) = color(box, "blue");
```

How do you do this? You create a *data theme* which modifies the definition.

Using your favourite text editor, you create a file named, say, ‘`blue-title.vsl`’ in the directory ‘`~/.ddd/themes/`’.

The file ‘`blue-title.vsl`’ has the following content:

```
#pragma replace title_color
title_color(box) = color(box, "blue");
```

In DDD, select ‘`Data` ⇒ `Themes`’. You will find ‘`blue-title.vsl`’ in a line on its own. Set the checkbox next to ‘`blue-title.vsl`’ in order to activate it. Whoa! All display titles will now appear in blue.

2.2 The General Scheme

The general scheme for writing a theme is:

- *Find the appropriate VSL function.*

  Find out which VSL function *function* is responsible for a specific task. See Appendix A [DDD VSL Functions], page 9, for details on the VSL functions used by DDD.

- *Replace it by your own definition.*

  Write a theme (a text file) with the following content:

  ```
  #pragma replace function
  function(args) = definition;
  ```

  This will replace the existing definition of *function* by your new definition *definition*. It is composed of two parts:

  - The ‘`#pragma replace`’ declaration removes the original definition of *function*. See Section C.6.4 [VSL Redefining Functions], page 34, for details.
  - The following line provides a new *definition* for *function*.

  Please note: If the function *function* is marked as ‘`Global VSL Function`’, it must be (re-)defined using ‘`->`’ instead of ‘`=`’; See Section C.6 [VSL Function Definitions], page 32, for details. You may also want to consider ‘`#pragma override`’ instead; See Section 2.3 [Overriding vs. Replacing], page 6, for details.

- *Install the theme in a place where* DDD *can find it.*

  For your personal use, this is normally the directory '`~/.ddd/themes/`'.

  Besides your personal directory, DDD also searches for themes in its theme directory, typically '`/usr/local/share/ddd-3.3.12-test5/themes/`'.

  The DDD '`vslPath`' resource controls the actual path where DDD looks for themes. See section "VSL Resources" in *Debugging with DDD*, for details.

- *In* DDD*, invoke* '`Data ⇒ Themes`' *to apply the theme.*

  You're done!

2.3 Overriding vs. Replacing

In certain cases, you may not want to replace the original definition by your own, but rather *extend* the original definition.

As an example, consider the '`value_box`' function (see Section A.4 [Displaying Data Displays], page 11). It is applied to every single value displayed. By default, it does nothing. So we could write a theme that leaves a little white space around values:

```
#pragma replace value_box
value_box(box) -> whiteframe(box);
```

or another theme that changes the color to black on yellow:

```
#pragma replace value_box
value_box(box) -> color(box, "black", "yellow");
```

However, we cannot apply both themes at once (say, to create a green-on-yellow scheme). This is because each of the two themes replaces the previous definition—the theme that comes last wins.

The solution to this problem is to set up the theme in such a way that it *extends* the original definition rather than to replace it. To do so, VSL provides an alternative to '`#pragma replace`', namely '`#pragma override`' (see Section C.6.6 [VSL Overriding Functions], page 35).

Like '`#pragma replace`', the '`#pragma override`' declaration allows for a new definition of a function. In contrast to '`#pragma replace`', though, uses of the function prior to '`#pragma override`' are not affected—they still refer to the old definition.

Here's a better theme that changes the color to black on yellow. First, it makes the old definition of '`value_box`' accessible as '`old_value_box`'. Then, it provides a new definition for '`value_box`' which refers to the old definition, saved in '`old_value_box`'.

```
#pragma override old_value_box
old_value_box(...) = value_box(...);

#pragma override value_box
value_box(value) -> color(old_value_box(value),
                          "black", "yellow");
```

Why do we need a '`#pragma override`' for '`old_value_box`', too? Simple: to avoid name clashes between multiple themes. VSL has no scopes or name spaces for definitions, so we must resort to this crude, but effective scheme.

2.4 A Complex Example

As a more complex example, we define a theme that highlights all null pointers. First, we need a predicate 'is_null' that tells us whether a pointer value is null:

```
// True if S1 ends in S2
ends_in(s1, s2) =
    let s1c = chars(s1),
        s2c = chars(s2) in suffix(s2c, s1c);

// True if null value
is_null(value) =
    (ends_in(value, "0x0") or ends_in(value, "nil"));
```

The 'null_pointer' function tells us how we actually want to render null values:

```
// Rendering of null values
null_pointer(value) -> color(value, "red");
```

Now we go and redefine the 'pointer_value' function such that 'null_pointer' is applied only to null values:

```
#pragma override old_pointer_value
old_pointer_value(...) -> pointer_value(...);

#pragma override pointer_value

// Ordinary pointers
pointer_value (value) ->
    old_pointer_value(v)
    where v = (if (is_null(value)) then
                   null_pointer(value)
               else
                   value
               fi);
```

All we need now is the same definition for dereferenced pointers (that is, overriding the 'dereferenced_pointer_value' function), and here we go!

2.5 Future Work

With the information in this manual, you should be able to set up your own themes. If you miss anything, please let us know: simply write to ddd@gnu.org.

If there is sufficient interest, DDD's data themes will be further extended. Among the most wanted features is the ability to access and parse debuggee data from within VSL functions; this would allow user-defined processing of debuggee data. Let us know if you're interested—and keep in touch!

Appendix A DDD VSL Functions

This appendix describes how DDD invokes VSL functions to create data displays.

The functions in this section are predefined in the library 'ddd.vsl'. They can be used and replaced by DDD themes.

Please note: Functions marked as 'Global VSL Function' must be (re-)defined using '->' instead of '='. See Section C.6 [VSL Function Definitions], page 32, for details.

A.1 Displaying Fonts

These are the function DDD uses for rendering boxes in different fonts:

small_rm (*box*) VSL Function
small_bf (*box*) VSL Function
small_it (*box*) VSL Function
small_bi (*box*) VSL Function

Returns *box* in small roman / bold face / italic / bold italic font.

small_size () VSL Function

Default size for small fonts.[1]

tiny_rm (*box*) VSL Function
tiny_bf (*box*) VSL Function
tiny_it (*box*) VSL Function
tiny_bi (*box*) VSL Function

Returns *box* in tiny roman / bold face / italic / bold italic font.

tiny_size () VSL Function

Default size for tiny fonts.[2]

title_rm (*box*) VSL Function
title_bf (*box*) VSL Function
title_it (*box*) VSL Function
title_bi (*box*) VSL Function

Returns *box* (a display title) in roman / bold face / italic / bold italic font.

value_rm (*box*) VSL Function
value_bf (*box*) VSL Function
value_it (*box*) VSL Function
value_bi (*box*) VSL Function

Returns *box* (a display value) in roman / bold face / italic / bold italic font.

---

[1] DDD replaces this as set in the DDD font preferences. Use 'ddd --fonts' to see the actual definitions.

[2] DDD replaces this as set in the DDD font preferences. Use 'ddd --fonts' to see the actual definitions.

A.2 Displaying Colors

display_color (*box*) VSL Function

Returns *box* in the color used for displays. Default definition is

```
display_color(box) = color(box, "black", "white");
```

title_color (*box*) VSL Function

Returns *box* in the color used for display titles. Default definition is

```
title_color(box) = color(box, "black");
```

disabled_color (*box*) VSL Function

Returns *box* in the color used for disabled displays. Default definition is

```
disabled_color(box) = color(box, "white", "grey50");
```

simple_color (*box*) VSL Function

Returns *box* in the color used for simple values. Default definition is

```
simple_color(box) = color(box, "black");
```

text_color (*box*) VSL Function

Returns *box* in the color used for multi-line texts. Default definition is

```
text_color(box) = color(box, "black");
```

pointer_color (*box*) VSL Function

Returns *box* in the color used for pointers. Default definition is

```
pointer_color(box) = color(box, "blue4");
```

struct_color (*box*) VSL Function

Returns *box* in the color used for structs. Default definition is

```
struct_color(box) = color(box, "black");
```

list_color (*box*) VSL Function

Returns *box* in the color used for lists. Default definition is

```
list_color(box) = color(box, "black");
```

array_color (*box*) VSL Function

Returns *box* in the color used for arrays. Default definition is

```
array_color(box) = color(box, "blue4");
```

reference_color (*box*) VSL Function

Returns *box* in the color used for references. Default definition is

```
reference_color(box) = color(box, "blue4");
```

changed_color (*box*) VSL Function

Returns *box* in the color used for changed values. Default definition is

```
changed_color(box) = color(box, "black", "#ffffcc");
```

shadow_color (*box*) VSL Function

Returns *box* in the color used for display shadows. Default definition is

```
shadow_color(box) = color(box, "grey");
```

A.3 Displaying Shadows

shadow (*box*) VSL Function

Return *box* with a shadow around it.

A.4 Displaying Data Displays

DDD uses these functions to create data displays.

title (*display_number*, *name*) Global VSL Function
title (*name*) Global VSL Function

Returns a box for the display title. If *display_number* (a string) is given, this is prepended to the title.

annotation (*name*) Global VSL Function

Returns a box for an edge annotation. This typically uses a tiny font.

disabled () Global VSL Function

Returns a box to be used as value for disabled displays.

none () Global VSL Function

Returns a box for "no value" (i.e. undefined values). Default: an empty string.

value_box (*value*) Global VSL Function

Returns *value* in a display box. Default: leave unchanged.

display_box (*title*, *value*) Global VSL Function
display_box (*value*) Global VSL Function

Returns the entire display box. *title* comes from `title()`, *value* from `value_box()`.

A.5 Displaying Simple Values

DDD uses these functions to display simple values.

simple_value (*value*) Global VSL Function

Returns a box for a simple non-numeric value (characters, strings, constants, . . .). This is typically aligned to the left.

numeric_value (*value*) Global VSL Function

Returns a box for a simple numeric value. This is typically aligned to the right.

collapsed_simple_value () Global VSL Function

Returns a box for a collapsed simple value.

A.6 Displaying Pointers

DDD uses these functions to display pointers.

pointer_value (*value*) Global VSL Function
Returns a box for a pointer value.

dereferenced_pointer_value (*value*) Global VSL Function
Returns a box for a dereferenced pointer value.

collapsed_pointer_value () Global VSL Function
Returns a box for a collapsed pointer.

A.7 Displaying References

DDD uses these functions to display references.

reference_value (*value*) Global VSL Function
Returns a box for a reference value.

collapsed_reference_value () Global VSL Function
Returns a box for a collapsed reference.

A.8 Displaying Arrays

DDD uses these functions to display arrays.

horizontal_array (*values*. . .) Global VSL Function
Returns a box for a horizontal array containing *values*.

vertical_array (*values*. . .) Global VSL Function
Returns a box for a vertical array containing *values*.

empty_array () Global VSL Function
Returns a box for an empty array.

collapsed_array () Global VSL Function
Returns a box for a collapsed array.

twodim_array (*rows*. . .) Global VSL Function
Returns a box for a two-dimensional array. Argument is a list of rows, suitable for use with `tab()` or `dtab()`.

twodim_array_elem (*value*) Global VSL Function
Returns a box for an element in a two-dimensional array.

A.9 Displaying Structs

A struct is a set of (*name*, *value*) pairs, and is also called "record" or "object". DDD uses these functions to display structs.

struct_value (*members*. . .) Global VSL Function
Returns a box for a struct containing *members*.

collapsed_struct_value () Global VSL Function
Returns a box for a collapsed struct.

empty_struct_value () Global VSL Function
Returns a box for an empty struct.

struct_member_name (*name*) Global VSL Function
Returns a box for a member name.

struct_member (*name*, *sep*, *value*, *name_width*) Global VSL Function
Returns a box for a struct member. *name* is the member name, typeset with `struct_member_name()`, *sep* is the separator (as determined by the current programming language), *value* is the typeset member value, and *name_width* is the maximum width of all member names.

horizontal_unnamed_struct () Global VSL Function
vertical_unnamed_struct () Global VSL Function
Returns a box for a horizontal / vertical unnamed struct, where member names are suppressed.

struct_member (*value*) Global VSL Function
Returns a box for a struct member in a struct where member names are suppressed.

A.10 Displaying Lists

A list is a set of (*name*, *value*) pairs not defined by the specific programming language. DDD uses this format to display variable lists.

list_value (*members*. . .) Global VSL Function
Returns a box for a list containing *members*.

collapsed_list_value () Global VSL Function
Returns a box for a collapsed list.

empty_list_value () Global VSL Function
Returns a box for an empty list.

list_member_name (*name*) Global VSL Function
Returns a box for a member name.

list_member (*name*, *sep*, *value*, *name_width*) Global VSL Function
Returns a box for a list member. *name* is the member name, typeset with `list_member_name()`, *sep* is the separator (as determined by the current programming language), *value* is the typeset member value, and *name_width* is the maximum width of all member names.

horizontal_unnamed_list () Global VSL Function
vertical_unnamed_list () Global VSL Function
Returns a box for a horizontal / vertical unnamed list, where member names are suppressed.

list_member (*value*) Global VSL Function
Returns a box for a list member in a list where member names are suppressed.

A.11 Displaying Sequences

Sequences are lists of arbitrary, unstructured values.

sequence_value (*values*. . .) Global VSL Function
Returns a box for a list of values.

collapsed_sequence_value () Global VSL Function
Returns a box for a collapsed sequence.

A.12 Displaying Multi-Line Texts

DDD uses these functions to display multi-line texts, such as status displays.

text_value (*lines*. . .) Global VSL Function
Returns a box for a list of lines (typically in a vertical alignment).

collapsed_text_value () Global VSL Function
Returns a box for a collapsed text.

A.13 Displaying Extra Properties

DDD uses these functions to display additional properties.

repeated_value (*value*, *n*) Global VSL Function
Returns a box for a *value* that is repeated *n* times. Note: *n* is a number, not a string.

changed_value (*value*) Global VSL Function
Returns a box for a *value* that has changed since the last display. Typically, this invokes `changed_color(`*value*`)`.

Appendix B VSL Library

This appendix describes the VSL functions available in the standard VSL library.

Unless otherwise stated, all following functions are defined in '`std.vsl`'.

For DDD themes, '`std.vsl`' need not be included explicitly.

B.1 Conventions

Throughout this document, we write *a* = (*a1*, *a2*) to refer to individual box sizes. *a1* stands for the horizontal size of *a*, and *a2* stands for the vertical size of *a*.

B.2 Space Functions

B.2.1 Empty Space

fill () VSL Function

Returns an empty box of width 0 and height 0 which stretches in both horizontal and vertical directions.

hfill () VSL Function

Returns a box of height 0 which stretches horizontally.

vfill () VSL Function

Returns a box of width 0 which stretches vertically.

B.2.2 Black Lines

rule () VSL Function

Returns a black box of width 0 and height 0 which stretches in both horizontal and vertical directions.

hrule ([*thickness*]) VSL Function

Returns a black box of width 0 and height *thickness* which stretches horizontally. *thickness* defaults to `rulethickness()` (typically 1 pixel).

vrule ([*thickness*]) VSL Function

Returns a black box of width *thickness* and height 0 which stretches vertically. *thickness* defaults to `rulethickness()` (typically 1 pixel).

rulethickness () VSL Function

Returns the default thickness for black rules (default: 1).

B.2.3 White Space

hwhite ([*thickness*]) VSL Function

Returns a black box of width 0 and height *thickness* which stretches horizontally. *thickness* defaults to `whitethickness()` (typically 2 pixels).

vwhite ([*thickness*]) VSL Function

Returns a black box of width *thickness* and height 0 which stretches vertically. *thickness* defaults to `whitethickness()` (typically 2 pixels).

whitethickness () VSL Function

Returns the default thickness for white rules (default: 2).

B.2.4 Controlling Stretch

hfix (*a*) VSL Function

Returns a box containing *a*, but not stretchable horizontally.

vfix (*a*) VSL Function

Returns a box containing *a*, but not stretchable vertically.

fix (*a*) VSL Function

Returns a box containing *a*, but not stretchable in either direction.

B.2.5 Box Dimensions

hspace (*a*) VSL Function

If $a = (a1, a2)$, create a square empty box with a size of $(a1, a1)$.

vspace (*a*) VSL Function

If $a = (a1, a2)$, create a square empty box with a size of $(a2, a2)$.

square (*a*) VSL Function

If $a = (a1, a2)$, create a square empty box with a size of $\max(a1, a2)$.

box (*n*, *m*) VSL Function

Returns a box of size (n, m).

B.3 Composition Functions

B.3.1 Horizontal Composition

(&) (*a*, *b*) VSL Function
(&) (*boxes*. . .) VSL Function
halign (*boxes*. . .) VSL Function

Returns a horizontal alignment of *a* and *b*; *a* is placed left of *b*. Typically written in inline form '`a & b`'.

The alternative forms (available in function-call form only) return a horizontal left-to-right alignment of their arguments.

hralign (*boxes*. . .) VSL Function

Returns a right-to-left alignment of its arguments.

B.3.2 Vertical Composition

(|) (*a*, *b*) VSL Function
(|) (*boxes*. . .) VSL Function
valign (*boxes*. . .) VSL Function

Returns a vertical alignment of *a* and *b*; *a* is placed above *b*. Typically written in inline form '`a | b`'.

The alternative forms (available in function-call form only) return a vertical top-to-bottom alignment of their arguments.

vralign (*boxes*. . .) VSL Function

Returns a bottom-to-top alignment of its arguments.

vlist (*sep*, *boxes*. . .) VSL Function

Returns a top-to-bottom alignment of *boxes*, where any two boxes are separated by *sep*.

B.3.3 Textual Composition

(~) (*a*, *b*) VSL Function
(~) (*boxes*. . .) VSL Function
talign (*boxes*. . .) VSL Function

Returns a textual concatenation of *a* and *b*. *b* is placed in the lower right unused corner of *a*. Typically written in inline form '`a ~ b`'.

The alternative forms (available in function-call form only) return a textual concatenation of their arguments.

tralign (*boxes*. . .) VSL Function

Returns a textual right-to-left concatenation of its arguments.

tlist (*sep*, *boxes*. . .) VSL Function

Returns a textual left-to-right alignment of *boxes*, where any two boxes are separated by *sep*.

commalist (*boxes*. . .) VSL Function

Shorthand for '`tlist(", ", `*boxes*`...)`'.

semicolonlist (*boxes*. . .) VSL Function

Shorthand for '`tlist("; ", `*boxes*`...)`'.

B.3.4 Overlays

(^) (*a*, *b*) VSL Function
(^) (*boxes*. . .) VSL Function

Returns an overlay of *a* and *b*. *a* and *b* are placed in the same rectangular area, which is the maximum size of *a* and *b*; first, *a* is drawn, then *b*. Typically written in inline form '*a* ^ *b*'.

The second form (available in function-call form only) returns an overlay of its arguments.

B.4 Arithmetic Functions

(+) (*a*, *b*) VSL Function
(+) (*boxes*. . .) VSL Function

Returns the sum of *a* and *b*. If $a = (a1, a2)$ and $b = (b1, b2)$, then $a + b = (a1 + a2, b1 + b2)$. Typically written in inline form '*a* + *b*'.

The second form (available in function-call form only) returns the sum of its arguments.

The special form '+*a*' is equivalent to '*a*'.

(-) (*a*, *b*) VSL Function

Returns the difference of *a* and *b*. If $a = (a1, a2)$ and $b = (b1, b2)$, then $a - b = (a1 - a2, b1 - b2)$. Typically written in inline form '*a* − *b*'.

The special form '−*a*' is equivalent to '`0−`*a*'.

(*) (*a*, *b*) VSL Function
(*) (*boxes*. . .) VSL Function

Returns the product of *a* and *b*. If $a = (a1, a2)$ and $b = (b1, b2)$, then $a * b = (a1 * a2, b1 * b2)$. Typically written in inline form '*a* * *b*'.

The second form (available in function-call form only) returns the product of its arguments.

(/) (*a*, *b*) VSL Function

Returns the quotient of *a* and *b*. If $a = (a1, a2)$ and $b = (b1, b2)$, then $a / b = (a1 / a2, b1 / b2)$. Typically written in inline form '*a* / *b*'.

(%) (*a*, *b*) VSL Function

Returns the remainder of *a* and *b*. If $a = (a1, a2)$ and $b = (b1, b2)$, then $a \% b = (a1 \% a2, b1 \% b2)$. Typically written in inline form '*a* % *b*'.

B.5 Comparison Functions

(=) (*a*, *b*) VSL Function

Returns true (‘`1`’) if $a = b$, and false (‘`0`’), otherwise. $a = b$ holds if *a* and *b* have the same size, the same structure, and the same content. Typically written in inline form ‘*a* / *b*’.

(<>) (*a*, *b*) VSL Function

Returns false (‘`0`’) if $a = b$, and true (‘`1`’), otherwise. $a = b$ holds if *a* and *b* have the same size, the same structure, and the same content. Typically written in inline form ‘*a* / *b*’.

(<) (*a*, *b*) VSL Function

If $a = (a1, a2)$ and $b = (b1, b2)$, then this function returns true (‘`1`’) if $a1 < b1$ or $a2 < b2$ holds; false (‘`0`’), otherwise. Typically written in inline form ‘$a < b$’.

(<=) (*a*, *b*) VSL Function

If $a = (a1, a2)$ and $b = (b1, b2)$, then this function returns true (‘`1`’) if $a1 <= b1$ or $a2 <= b2$ holds; false (‘`0`’), otherwise. Typically written in inline form ‘$a <= b$’.

(>) (*a*, *b*) VSL Function

If $a = (a1, a2)$ and $b = (b1, b2)$, then this function returns true (‘`1`’) if $a1 > b1$ or $a2 > b2$ holds; false (‘`0`’), otherwise. Typically written in inline form ‘$a > b$’.

(>=) (*a*, *b*) VSL Function

If $a = (a1, a2)$ and $b = (b1, b2)$, then this function returns true (‘`1`’) if $a1 >= b1$ or $a2 >= b2$ holds; false (‘`0`’), otherwise. Typically written in inline form ‘$a >= b$’.

B.5.1 Maximum and Minimum Functions

max (*b1*, *b2*, . . .) VSL Function

Returns the maximum of its arguments; that is, the one box *b* in its arguments for which $b > b1$, $b > b2$, . . . holds.

min (*b1*, *b2*, . . .) VSL Function

Returns the maximum of its arguments; that is, the one box *b* in its arguments for which $b < b1$, $b < b2$, . . . holds.

B.6 Negation Functions

(not) (*a*) VSL Function

Returns true (‘`1`’) if *a* is false, and false (‘`0`’), otherwise. Typically written in inline form ‘`not` *a*’.

See Section C.3.5 [VSL Boolean Operators], page 31, for `and` and `or`.

B.7 Frame Functions

ruleframe (*a*[, *thickness*]) VSL Function

Returns *a* within a black rectangular frame of thickness *thickness*. *thickness* defaults to `rulethickness()` (typically 1 pixel).

whiteframe (*a*[, *thickness*]) VSL Function

Returns *a* within a white rectangular frame of thickness *thickness*. *thickness* defaults to `whitethickness()` (typically 2 pixels).

frame (*a*) VSL Function

Returns *a* within a rectangular frame. Equivalent to '`ruleframe(whiteframe(`*a*`)`'.

doubleframe (*a*) VSL Function

Shortcut for '`frame(frame(`*a*`))`'.

thickframe (*a*) VSL Function

Shortcut for '`ruleframe(frame(`*a*`))`'.

B.8 Alignment Functions

B.8.1 Centering Functions

hcenter (*a*) VSL Function

Returns box *a* centered horizontally within a (vertical) alignment.

Example: In '*a* `| hcenter(`*b*`) |` *c*', *b* is centered relatively to *a* and *c*.

vcenter (*a*) VSL Function

Returns box *a* centered vertically within a (horizontal) alignment.

Example: In '*a* `& vcenter(`*b*`) &` *c*', *b* is centered relatively to *a* and *c*.

center (*a*) VSL Function

Returns box *a* centered vertically and horizontally within an alignment.

Example: In '`100 ^ center(`*b*`)`', *b* is centered within a square of size 100.

B.8.2 Flushing Functions

n_flush (*box*) VSL Function
s_flush (*box*) VSL Function
w_flush (*box*) VSL Function
e_flush (*box*) VSL Function

Within an alignment, Flushes box to the center of a side.

Example: In '`100 ^ s_flush(`*b*`)`', *b* is centered on the bottom side of a square of size 100.

nw_flush (*box*) VSL Function
sw_flush (*box*) VSL Function
ne_flush (*box*) VSL Function
se_flush (*box*) VSL Function

Within an alignment, Flushes box to a corner.

Example: In '`100 ^ se_flush(b)`', *b* is placed in the lower right corner of a square of size 100.

B.9 Emphasis Functions

underline (*a*) VSL Function

Returns *a* with a line underneath.

overline (*a*) VSL Function

Returns *a* with a line above it.

crossline (*a*) VSL Function

Returns *a* with a horizontal line across it.

doublestrike (*a*) VSL Function

Returns *a* in "poor man's bold": it is drawn two times, displaced horizontally by one pixel.

B.10 Indentation Functions

indent (*box*) VSL Function

Return a box where white space of width `indentamount()` is placed left of *box*.

indentamount () VSL Function

Indent amount to be used in `indent()`; defaults to '`"  "`' (two spaces).

B.11 String Functions

To retrieve the string from a composite box, use `string()`:

string (*box*) VSL Function

Return the string (in left-to-right, top-to-bottom order) within *box*.

To convert numbers to strings, use `num()`:

num (*a* [, *varbase*]) VSL Function

For a square box *a* = (*a1*, *a1*), returns a string containing a textual representation of *a1*. *base* must be between 2 and 16; it defaults to '`10`'. Example: `num(25)` ⇒ `"25"`)

dec (*a*) VSL Function
oct (*a*) VSL Function
bin (*a*) VSL Function
hex (*a*) VSL Function

Shortcut for '`num(a, 10)`', '`num(a, 8)`', '`num(a, 2)`', '`num(a, 16)`', respectively.

B.12 List Functions

The functions in this section require inclusion of the library '`list.vsl`'.
For themes, '`list.vsl`' need not be included explicitly.

B.12.1 Creating Lists

(::) (*list1*, *list2*, ...) VSL Function
Return the concatenation of the given lists. Typically written in inline form: `[1] :: [2] :: [3]` ⇒ `[1, 2, 3]`.

append (*list*, *elem*) VSL Function
Returns *list* with *elem* appended at the end: `append([1, 2, 3], 4)` ⇒ `[1, 2, 3, 4]`

B.12.2 List Properties

isatom (*x*) VSL Function
Returns True (1) if *x* is an atom; False (0) if *x* is a list.

islist (*x*) VSL Function
Returns True (1) if *x* is a list; False (0) if *x* is an atom.

member (*x*, *list*) VSL Function
Returns True (1) if *x* is an element of *list*; False (0) if not: `member(1, [1, 2, 3])` ⇒ `true`

prefix (*sublist*, *list*) VSL Function
suffix (*sublist*, *list*) VSL Function
sublist (*sublist*, *list*) VSL Function
Returns True (1) if *sublist* is a prefix / suffix / sublist of *list*; False (0) if not: `prefix([1], [1, 2])` ⇒ `true`, `suffix([3], [1, 2])` ⇒ `false`, `sublist([2, 2], [1, 2, 2, 3])` ⇒ `true`,

length (*list*) VSL Functions
Returns the number of elements in *list*: `length([1, 2, 3])` ⇒ `3`

B.12.3 Accessing List Elements

car (*list*) VSL Function
head (*list*) VSL Function
Returns the first element of *list*: `car([1, 2, 3])` ⇒ `1`

cdr (*list*) VSL Function
tail (*list*) VSL Function
Returns *list* without its first element: `cdr([1, 2, 3])` ⇒ `[2, 3]`

elem (*list*, *n*) VSL Function

Returns the *n*-th element (starting with 0) of *list*: `elem([4, 5, 6], 0)` ⇒ `4`

pos (*elem*, *list*) VSL Function

Returns the position of *elem* in *list* (starting with 0): `pos(4, [1, 2, 4])` ⇒ `2`

last (*list*) VSL Function

Returns the last element of *list*: `last([4, 5, 6])` ⇒ `6`

B.12.4 Manipulating Lists

reverse (*list*) VSL Function

Returns a reversed *list*: `reverse([3, 4, 5])` ⇒ `[5, 4, 3]`

delete (*list*, *elem*) VSL Function

Returns *list*, with all elements *elem* removed: `delete([4, 5, 5, 6], 5)` ⇒ `[4, 6]`

select (*list*, *elem*) VSL Function

Returns *list*, with the first element *elem* removed: `select([4, 5, 5, 6], 5)` ⇒ `[4, 5, 6]`

flat (*list*) VSL Function

Returns flattened *list*: `flat([[3, 4], [[5], [6]]])` ⇒ `[3, 4, 5, 6]`

sort (*list*) VSL Function

Returns sortened *list* (according to box size): `sort([7, 4, 9])` ⇒ `[4, 7, 9]`

B.12.5 Lists and Strings

chars (*s*) VSL Function

Returns a list of all characters in the box *s*: `chars("abc")` ⇒ `["a", "b", "c"]`

list (*list*) VSL Function

Returns a string, pretty-printing the *list*: `list([4, 5, 6])` ⇒ `"[4, 5, 6]"`

B.13 Table Functions

The functions in this section require inclusion of the library '`tab.vsl`'.

For themes, '`tab.vsl`' need not be included explicitly.

tab (*table*) VSL Function

Return *table* (a list of lists) aligned in a table: `tab([[1, 2, 3], [4, 5, 6], [7, 8]])` ⇒

```
1 2 3
4 5 6
7 8
```

dtab (*table*) VSL Function

Like `tab`, but place delimiters (horizontal and vertical rules) around table elements.

tab_elem (*x*) VSL Function

Returns padded table element *x*. Its default definition is:

```
tab_elem([]) = tab_elem(0);      // empty table
tab_elem(x)  = whiteframe(x);    // padding
```

B.14 Font Functions

The functions in this section require inclusion of the library '`fonts.vsl`'.

For themes, '`fonts.vsl`' need not be included explicitly.

B.14.1 Font Basics

font (*box*, *font*) VSL Function

Returns *box*, with all strings set in *font* (a valid X11 font description)

B.14.2 Font Name Selection

weight_bold () VSL Function
weight_medium () VSL Function

Font weight specifier in `fontname()` (see below).

slant_unslanted () VSL Function
slant_italic () VSL Function

Font slant Specifier in `fontname()` (see below).

family_times () VSL Function
family_courier () VSL Function
family_helvetica () VSL Function
family_new_century () VSL Function
family_typewriter () VSL Function

Font family specifier in `fontname()` (see below).

fontname ([*weight*, [*slant*, [*family*, [*size*]]]]) VSL Function

Returns a fontname, suitable for use with `font()`.

- *weight* defaults to `stdfontweight()` (see below).
- *slant* defaults to `stdfontslant()` (see below).
- *family* defaults to `stdfontfamily()` (see below).
- *size* is a pair (*pixels*, *points*) where *pixels* being zero means to use *points* instead and vice versa. defaults to `stdfontsize()` (see below).

B.14.3 Font Defaults

stdfontweight () VSL Function

Default font weight: `weight_medium()`.

stdfontslant () VSL Function

Default font slant: `slant_unslanted()`.

stdfontfamily () VSL Function

Default font family: `family_times()`.

DDD replaces this as set in the DDD font preferences. Use '`ddd --fonts`' to see the actual definitions.

stdfontsize () VSL Function

Default font size: `(stdfontpixels(), stdfontpoints())`.

DDD replaces this as set in the DDD font preferences. Use '`ddd --fonts`' to see the actual definitions.

stdfontpixels () VSL Function

Default font size (in pixels): 0, meaning to use `stdfontpoints()` instead.

stdfontpoints () VSL Function

Default font size (in 1/10 points): 120.

B.14.4 Font Selection

rm (*box* [, *family* [, *size*]]) VSL Function
bf (*box* [, *family* [, *size*]]) VSL Function
it (*box* [, *family* [, *size*]]) VSL Function
bi (*box* [, *family* [, *size*]]) VSL Function

Returns *box* in roman / bold face / italic / bold italic. *family* specifies one of the font families; it defaults to `stdfontfamily()` (see above). *size* specifies a font size; it defaults to `stdfontsize()` (see above).

B.15 Color Functions

The functions in this section require inclusion of the library '`colors.vsl`'.

For themes, '`colors.vsl`' need not be included explicitly.

color (*box*, *foreground* [, *background*]]) VSL Function

Returns *box*, where the foreground color will be drawn using the *foreground* color. If *background* is specified as well, it will be used for drawing the background. Both *foreground* and *background* are strings specifying a valid X11 color.

B.16 Arc Functions

The functions in this section require inclusion of the library '`arcs.vsl`'.

For themes, '`arcs.vsl`' *must* be included explicitly, using a line

```
#include <arcs.vsl>
```

at the beginning of the theme.

B.16.1 Arc Basics

arc (*start*, *length* [, *thickness*]) VSL Function

Returns a stretchable box with an arc of *length*, starting at angle *start*. *start* and *length* must be multiples of 90 (degrees). The angle of *start* is specified clockwise relative to the 9 o'clock position. *thickness* defaults to `arcthickness()` (see below).

arcthickness () VSL Function

Default width of arcs. Defaults to `rulethickness()`.

B.16.2 Custom Arc Functions

oval (*box*) VSL Function

Returns an oval containing *box*. Example: `oval("33")`.

ellipse (*box*) VSL Function
ellipse () VSL Function

Returns an ellipse containing *box*. Example: `ellipse("START")`. If *box* is omitted, the ellipse is stretchable and expands to the available space.

circle (*box*) VSL Function

Returns a circle containing *box*. Example: `circle(10)`.

B.17 Slope Functions

The functions in this section require inclusion of the library '`slopes.vsl`'.

For themes, '`slopes.vsl`' *must* be included explicitly, using a line

```
#include <slopes.vsl>
```

at the beginning of the theme.

B.17.1 Slope Basics

rise ([*thickness*]) VSL Function

Create a stretchable box with a line from the lower left to the upper right corner. *thickness* defaults to `slopethickness()` (see below).

fall ([*thickness*]) VSL Function

Create a stretchable box with a line from the upper left to the lower right corner. *thickness* defaults to `slopethickness()` (see below).

slopethickness () VSL Function

Default thickness of slopes. Defaults to `rulethickness()`.

B.17.2 Arrow Functions

n_arrow () VSL Function
w_arrow () VSL Function
s_arrow () VSL Function
e_arrow () VSL Function

Returns a box with an arrow pointing to the upper, left, lower, or right side, respectively.

nw_arrow () VSL Function
ne_arrow () VSL Function
sw_arrow () VSL Function
se_arrow () VSL Function

Returns a box with an arrow pointing to the upper left, upper right, lower left, or lower right side, respectively.

B.17.3 Custom Slope Functions

punchcard (*box*) VSL Function

Returns a punchcard containing *box*.

rhomb (*box*) VSL Function

Returns a rhomb containing *box*.

octogon (*box*) VSL Function

Returns an octogon containing *box*.

Appendix C VSL Reference

This appendix describes the VSL language.

C.1 Boxes

VSL knows two data types. The most common data type is the *box*. A box is a rectangular area with a *content*, a *size*, and a *stretchability*.

Boxes are either *atomic* or *composite*. A composite box is built from two or more other boxes. These boxes can be aligned horizontally, vertically, or otherwise.

Boxes have a specific minimum *size*, depending on their content. We say 'minimum' size here, because some boxes are *stretchable*—that is, they can fill up the available space.

If you have a vertical alignment of three boxes *A*, *B*, and *C*, like this:

```
AAAAAA
AAAAAA
  B
  B
CCCCCC
CCCCCC
```

and *B* is stretchable horizontally, then *B* will fill up the available horizontal space:

```
AAAAAA
AAAAAA
BBBBBB
BBBBBB
CCCCCC
CCCCCC
```

If two or more boxes compete for the same space, the space will be distributed in proportion to their stretchability.

An atomic stretchable box has a stretchability of 1. An alignment of multiple boxes stretchable in the direction of the alignment boxes will have a stretchability which is the sum of all stretchabilities.

If you have a vertical alignment of three boxes *A*, *B*, *C*, *D*, and *E*, like this:

```
AAAAAA
AAAAAA
BC   D
BC   D
EEEEEE
EEEEEE
```

and *B*, *C*, and *D* are stretchable horizontally (with a stretchability of 1), then the horizontal alignment of *B* and *C* will have a stretchability of 2. Thus, the alignment of *B* and *C* gets two thirds of the available space; *D* gets the remaining third.

```
AAAAAA
AAAAAA
BBCCDD
BBCCDD
EEEEEE
EEEEEE
```

C.2 Lists

Besides boxes, VSL knows *lists*. A list is not a box—it has no size or stretchability. A list is a simple means to structure data.

VSL lists are very much like lists in functional languages like Lisp or Scheme. They consist of a head (typically a list element) and a tail (which is either a list remainder or the empty list).

C.3 Expressions

C.3.1 String Literals

The expression '`"`*text*`"`' returns a box containing *text*. *text* is parsed according to C syntax rules.

Multiple string expressions may follow each other to form a larger constant, as in C++. '`"`*text1*`"` `"`*text2*`"`' is equivalent to '`"`*text1 text2*`"`'

Strings are not stretchable.

C.3.2 Number Literals

Any constant integer *n* evaluates to a *number*—that is, a non-stretchable empty square box with size (*n*, *n*).

C.3.3 List Literals

The expression '`[`*a*`,` *b*`, ...]`' evaluates to a *list* containing the element *a*, *b*, ... '`[]`' is the empty list.

The expression '`[`*head* `:` *tail*`]`' evaluates to a list whose first element is *head* and whose remainder (typically a list) is *tail*.

In most contexts, round parentheses can be used as alternatives to square brackets. Thus, '`(`*a*`,` *b*`)`' is a list with two elements, and '`()`' is the empty list.

Within an expression, though, square parentheses must be used to create a list with one element. In an expression, the form '`(`*a*`)`' is not a list, but an alternative notation for *a*.

C.3.4 Conditionals

A box *a* = (*a1*, *a2*) is called *true* if *a1* or *a2* is non-zero. It is called *false* if both *a1* or *a2* are zero.

The special form

```
if a then b else c fi
```

returns *b* if *a* is true, and *c* otherwise. Only one of *b* or *c* is evaluated.

The special form

```
elsif a2 then b2 else c fi
```

is equivalent to

```
else if a2 then b2 else c fi fi
```

C.3.5 Boolean Operators

The special form

```
a and b
```

is equivalent to

```
if a then b else 0 fi
```

The special form

```
a or b
```

is equivalent to

```
if a then 1 else b fi
```

The special form

```
not a
```

is equivalent to

```
if a then 0 else 1 fi
```

Actually, 'not' is realized as a function; See Section B.6 [Negation Functions], page 19, for details.

C.3.6 Local Variables

You can introduce local variables using 'let' and 'where':

```
let v1 = e1 in e
```

makes *v1* available as replacement for *e1* in the expression *e*.

Example:

```
let pi = 3.1415 in 2 * pi ⇒ 6.2830
```

The special form

```
let v1 = e1, v2 = e2, ... in e
```

is equivalent to

```
let v1 = e1 in let v2 = e2 in let ... in e
```

As an alternative, you can also use the where form:

```
e where v1 = e1
```

is equivalent to

```
let v1 = e1 in e
```

Example:

```
("here lies" | name) where
    name = ("one whose name" | "was writ in water")
```

The special form

```
e where v1 = e1, v2 = e2, ...
```

is equivalent to

```
let v1 = e1, v2 = e2, ... in e
```

C.3.7 Let Patterns

You can access the individual elements of a list or some composite box by giving an appropriate *pattern*:

```
let (left, right) = pair in expr
```

If `pair` has the value, say, `(3, 4)`, then `left` will be available as a replacement for `3`, and `right` will be available as a replacement for `4` in *expr*.

A special pattern is available for accessing the head and the tail of a list:

```
let [head : tail] = list in expr
```

If `expr` has the value, say, `[3, 4, 5]`, then `head` will be `3`, and `tail` will be `[4, 5]` in *expr*.

C.4 Function Calls

A function call takes the form

name list

which invokes the (previously declared or defined) function with an argument of *list*. Normally, *list* is a list literal (see Section C.3.3 [VSL List Literals], page 30) written with round brackets.

C.5 Constant Definitions

A VSL file consists of a list of *definitions*.

A constant definition takes the form

name = *expression*;

Any later definitions can use *name* as a replacement for *expression*.

Example:

```
true = 1;
false = 0;
```

C.6 Function Definitions

In VSL, all functions either map a *list* to a *box* or a *list* to a *list*. A function definition takes the form

name list = *expression*;

where *list* is a list literal (see Section C.3.3 [VSL List Literals], page 30).

The list literal is typically written in round parentheses, making the above form look like this:

name (*param1*, *param2*, ...) = *expression*;

The '=' is replaced by '->' if *name* is a *global* definition—that is, *name* can be called from a library client such as DDD. A *local* definition (with '=') can be called only from other VSL functions.[1]

[1] The distinction into global and local definitions is useful when optimizing the library: local definitions that are unused within the library can be removed, while global definitions cannot.

C.6.1 Function Parameters

The parameter list *list* may contain names of formal parameters. Upon a function call, these are bound to the actual arguments.

If the function

```
sum(a, b) = a + b;
```

is called as

```
sum(2. 3)
```

then `a` will be bound to `2` and `b` will be bound to `3`, evaluating to `5`.

C.6.1.1 VSL Unused Parameters

Unused parameters cause a warning, as in this example:

```
first_arg(a, dummy) = a;          // Warning
```

If a parameter has the name '_', it will not be bound to the actual argument (and can thus not be used). Use '_' as parameter name for unused arguments:

```
first_arg(a, _) = a;              // No warning
```

'_' can be used multiple times in a parameter list.

C.6.2 Function Patterns

A VSL function may have multiple definitions, each with a specific *pattern*. The first definition whose pattern *matches* the actual argument is used.

What does 'matching' mean? Within a pattern,

- An ordinary formal parameter matches any single value
- A formal parameter whose name is '`...`' or ends in '`...`' matches a single value or a list or a list remainder
- A constant matches exactly the same value
- A composite box or list matches a composite box or list if
  - the composites have the same type
  - the composites have the same number of elements
  - the elements match each other.

Here are some examples. The `num()` function (see Section B.11 [String Functions], page 21) can take either one or two arguments. The one-argument definition simply invokes the two-argument definition:

```
num(a, base) = ...;
num(a) = num(a, 10);
```

Here's another example: The `digit` function returns a string representation for a single number. It has multiple definitions, all dependent on the actual argument:

```
digit(0) = "0";
digit(1) = "1";
digit(2) = "2";
digit(3) = "3";
```

```
digit(4) = "4";
digit(5) = "5";
digit(6) = "6";
digit(7) = "7";
digit(8) = "8";
digit(9) = "9";
digit(10) = "a";
digit(11) = "b";
digit(12) = "c";
digit(13) = "d";
digit(14) = "e";
digit(15) = "f";
digit(_) = fail("invalid digit() argument");
```

Formal parameters ending in '`...`' are useful for defining *aliases* of functions. The definition

```
roman(...) = rm(...);
```

makes `roman` an alias of `rm`—any parameters (regardless how many) passed to `roman` will be passed to `rm`.

Here's an example of how formal parameters ending in '`...`' can be used to realize *variadic functions*, taking any number of arguments (see Section B.5.1 [Maximum and Minimum Functions], page 19):

```
max(a) = a;
max(a, b, ...) = if a > b then max(a, ...) else max(b, ...) fi;
min(a) = a;
min(a, b, ...) = if a < b then min(a, ...) else min(b, ...) fi;
```

C.6.3 Declaring Functions

If you want to use a function before it has been defined, just write down its signature without specifying a body. Here's an example:

```
num(a, base);                        // declaration
num(a) = num(a, 10);
```

Remember to give a definition later on, though.

C.6.4 Redefining Functions

You can redefine a VSL function even *after* its original definition. You can

replace the original definition, thus making all previous definitions refer to your new definition;

override the original definition, thus making only later definitions refer to your new definition.

C.6.5 Replacing Functions

To remove an original definition, use

`#pragma replace` *name*

This removes all previous definitions of *name*. Be sure to provide your own definitions, though.

'`#pragma replace`' is typically used to change defaults:

```
#include "fonts.vsl"            // defines stdfontsize()

#pragma replace stdfontsize()   // replace def
stdfontsize() = 20;
```

All existing function calls will now refer to the new definition.

C.6.6 Overriding Functions

To override an original definition, use

```
#pragma override name
```

This makes all later definitions use your new definition of *name*. Earlier definitions, however, still refer to the old definition.

'`#pragma override`' is typically used if you want to redefine a function while still refering to the old definition:

```
#include "fonts.vsl"            // defines stdfontsize()

// Save old definition
old_stdfontsize() = stdfontsize();

#pragma override stdfontsize()

// Refer to old definition
stdfontsize() = old_stdfontsize() * 2;
```

Since we used '`#pragma override`', we can use `old_stdfontsize()` to refer to the original definition of `stdfontsize()`.

C.7 Includes

In a VSL file, you can include at any part the contents of another VSL file, using one of the special forms

```
#include "file"
#include <file>
```

The form '`<file>`' looks for VSL files in a number of standard directories; the form '`"file"`' first looks in the directory where the current file resides.

Any included file is included only once.

In DDD, you can set these places using the '`vslPath`' resource. See section "Customizing Display Appearance" in *Debugging with DDD*, for details.

C.8 Operators

VSL comes with a number of *inline operators,* which can be used to compose boxes. With raising precedence, these are:

```
or
and
= <>
```

```
<= < >= >
::
|
^
~
&
+ -
* / %
not
```

Except for `or` and `and`, these operators are mapped to function calls. Each invocation of an operator ‘`@`’ in the form ‘*a* `@` *b*’ gets translated to a call of the VSL function with the special name ‘`(@)`’. This VSL function can be defined just like any other VSL function.

For instance, the expression *a* + *b* gets translated to a function call `(+)(a, b)`; *a* `&` *b* invokes `(&)(a, b)`.

In the file ‘`builtin.vsl`’, you can actually find definitions of these functions:

```
(&)(...) = __op_halign(...);
(+)(...) = __op_plus(...);
```

The functions `__op_halign` and `__op_plus` are the names by which the ‘`(&)`’ and ‘`(+)`’ functions are implemented. In this document, though, we will not look further at these internals.

Here are the places where the operator functions are described:

For ‘=’ and ‘<>’, See Section B.5 [Comparison Functions], page 19.

For ‘<=’, ‘<’, ‘>=’, and ‘>’, See Section B.5 [Comparison Functions], page 19.

For ‘`::`’, See Section B.12 [List Functions], page 22.

For ‘`|`’, ‘`^`’, ‘`~`’, and ‘`&`’, See Section B.3 [Composition Functions], page 17.

For ‘+’, ‘−’, ‘`*`’, ‘`/`’, and ‘`%`’, See Section B.4 [Arithmetic Functions], page 18.

For ‘`not`’, See Section B.6 [Negation Functions], page 19.

C.9 Syntax Summary

The following file summarizes the syntax of VSL files.

```
/*** VSL file ***/

file                        :       item_list

item_list                   :       /* empty */
                            |       item_list item

item                        :       function_declaration ';'
                            |       function_definition ';'
                            |       override_declaration
                            |       replace_declaration
                            |       include_declaration
                            |       line_declaration
                            |       ';'
                            |       error ';'
```

```
/*** functions ***/

function_declaration    :       function_header

function_header         :       function_identifier function_argument
                        |       function_identifier

function_identifier     :       identifier
                        |       '(' '==' ')'
                        |       '(' '<>' ')'
                        |       '(' '>' ')'
                        |       '(' '>=' ')'
                        |       '(' '<' ')'
                        |       '(' '<=' ')'
                        |       '(' '&' ')'
                        |       '(' '|' ')'
                        |       '(' '^' ')'
                        |       '(' '~' ')'
                        |       '(' '+' ')'
                        |       '(' '-' ')'
                        |       '(' '*' ')'
                        |       '(' '/' ')'
                        |       '(' '%' ')'
                        |       '(' '::' ')'
                        |       '(' 'not' ')'

identifier              :       IDENTIFIER

function_definition     :       local_definition
                        |       global_definition

local_definition        :       local_header function_body

local_header            :       function_header '='

global_definition       :       global_header function_body

global_header           :       function_header '->'

function_body           :       box_expression_with_defs


/*** expressions ***/

/*** let, where ***/

box_expression_with_defs:       box_expression_with_wheres
```

```
                        |       'let' var_definition in_box_expression

in_box_expression       :       'in' box_expression_with_defs
                        |       ',' var_definition in_box_expression

box_expression_with_wheres:     box_expression
                        |       box_expression_with_where

box_expression_with_where:      box_expression_with_wheres
                                'where' var_definition
                        |       box_expression_with_where
                                ',' var_definition

var_definition          :       box_expression '=' box_expression


/*** basic expressions ***/

box_expression          :       '(' box_expression_with_defs ')'
                        |       list_expression
                        |       const_expression
                        |       binary_expression
                        |       unary_expression
                        |       cond_expression
                        |       function_call
                        |       argument_or_function

list_expression         :       '[' ']'
                        |       '[' box_expression_list ']'
                        |       '(' ')'
                        |       '(' multiple_box_expression_list ')'

box_expression_list     :       box_expression_with_defs
                        |       multiple_box_expression_list

multiple_box_expression_list:   box_expression ':' box_expression
                        |       box_expression ',' box_expression_list
                        |       box_expression '...'
                        |       '...'

const_expression        :       string_constant
                        |       numeric_constant

string_constant         :       STRING
                        |       string_constant STRING

numeric_constant        :       INTEGER

function_call           :       function_identifier function_argument
```

```
unary_expression        :       'not' box_expression
                        |       '+' box_expression
                        |       '-' box_expression


/*** operators ***/

binary_expression       :       box_expression '=' box_expression
                        |       box_expression '<>' box_expression
                        |       box_expression '>' box_expression
                        |       box_expression '>=' box_expression
                        |       box_expression '<' box_expression
                        |       box_expression '<=' box_expression
                        |       box_expression '&' box_expression
                        |       box_expression '|' box_expression
                        |       box_expression '^' box_expression
                        |       box_expression '~' box_expression
                        |       box_expression '+' box_expression
                        |       box_expression '-' box_expression
                        |       box_expression '*' box_expression
                        |       box_expression '/' box_expression
                        |       box_expression '%' box_expression
                        |       box_expression '::' box_expression
                        |       box_expression 'or' box_expression
                        |       box_expression 'and' box_expression

cond_expression         :       'if' box_expression
                                'then' box_expression_with_defs
                                else_expression
                                'fi'

else_expression         :       'elsif' box_expression
                                'then' box_expression_with_defs
                                else_expression
                        |       'else' box_expression_with_defs

function_argument       :       list_expression
                        |       '(' box_expression_with_defs ')'

argument_or_function    :       identifier


/*** directives ***/

override_declaration    :       '#pragma' 'override' override_list

override_list           :       override_identifier
                        |       override_list ',' override_identifier
```

```
override_identifier     :       function_identifier

replace_declaration     :       '#pragma' 'replace' replace_list

replace_list            :       replace_identifier
                        |       replace_list ',' replace_identifier

replace_identifier      :       function_identifier

include_declaration     :       '#include' '"' SIMPLE_STRING '"'
                        |       '#include' '<' SIMPLE_STRING '>'

line_declaration        :       '#line' INTEGER
                        |       '#line' INTEGER STRING
```

Appendix D GNU Free Documentation License

Version 1.1, March 2000

0. PREAMBLE

   The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

   This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

   We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

   This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

   A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

   A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

   The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

   The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

   A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

   You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

   You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

   If you publish printed copies of the Document numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

   If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

   If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section as "Endorsements" or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

   You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

   The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

   In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

   You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

   A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document, provided no compilation copyright is claimed for the compilation. Such a compilation is called an "aggregate", and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

   If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document's Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

   Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

9. TERMINATION

   You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

    The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

    Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

```
Copyright (C)  year  your name.
Permission is granted to copy, distribute and/or modify this document
under the terms of the GNU Free Documentation License, Version 1.1
or any later version published by the Free Software Foundation;
with the Invariant Sections being list their titles, with the
Front-Cover Texts being list, and with the Back-Cover Texts being list.
A copy of the license is included in the section entitled ``GNU
Free Documentation License''.
```

If you have no Invariant Sections, write "with no Invariant Sections" instead of saying which ones are invariant. If you have no Front-Cover Texts, write "no Front-Cover Texts" instead of "Front-Cover Texts being *list*"; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.

Index

H

I

L

M

N

O

P

R

S