HiTEX: Reflowable Output for TEX

HiTEX:

Reflowable Output for TEX

Für Beatriz

MARTIN RUCKERT *Munich University of Applied Sciences*

Date: 2020-03-05 19:29:50 +0100 (Thu, 05 Mar 2020)

Revision: 1873

Ruckert, Martin.
 HiTeX: TeX for Reflowable Output
 Includes index.
 ISBN 0-000-00000-0

Internet page https://w3-o.cs.hm.edu/~ruckert/hint/ may contain current information about this book, downloadable software, and news.

ruckert@cs.hm.edu

ISBN-10: 0-000-00000-0
ISBN-13: 000-00000000000

First printing, August 2019

Preface

To be written

München
August 20, 2018 *Martin Ruckert*

Contents

1 Introduction

HiTEX is a modified version of TEXthat replaces the DVI (device independent) output format by the HINT format. The HINT format, described in [12], was designed with two objectives: being able to support reflowing pages using the TEX typesetting engine and making it simple to use it as output format of the TEX programm. The present book tries to convince its readers that the latter claim is true. To this end, this book implements a version of TEX that produces HINT output—to be precise: a short format HINT file.

The book is written as a literate program[8] using "The CWEB System of Structured Documentation"[9]. The largest part of this program is, of course, TEX[7] itself. Therefore a significant part of HiTEX consists of modifications of the TEX source code, which itself is written as a literate program using "The WEB System of Structured Documentation"[6]. The tiny, but significant, difference between WEB and CWEB is the transition from Pascal to C as target language. To bridge the gap between WEB and CWEB, HiTEX is not based on the original WEB implementation of TEX but on the CWEB implementation of TEX as described in [10] and [11].

To accomplish modifications of a CWEB file, the CWEB tools support the use of "change files". These change files are lists of code changes optionally embellished with explanatory text. Each code change consists of two parts: a literal copy of the original source code followed by the replacement text. When a CWEB tool applies such a change file to a cweb file, it will read both files sequentially and applys the code changes in the order given: Whenever it finds a section in the CWEB file that matches the first part of the current code change, it will replace it by the second part of the code change, and advance to the next code change.

The present book makes an attempt to present these code changes in "literate programming style" and had to overcome two obstacles: First, in a literate program, the exposition determines the order of appearance of the code sections; and second, the tools that generate the documentation from a CWEB file are not able to generate documentation from a change file.

The first problem can be solved by using the program tie[3][2]. It allows splitting a single change file into multiple change files, and while within each change file, the order of changes is still determined by the original cweb file, changes that belong together can be grouped into separate files.

The second problem is solved by a simple preprocessor, that converts change files into cweb files which then can be converted into TEX. With the help of some modifications of the macros in cwebmac.tex these TEX files are used to present change files in this book.

Let's look at two examples to explain and illustrate the presentation of code changes in this book: the `display_node.ch` and `types.ch` change files.

1.1 The Function *display_node*

It is the purpose of the `display_node.ch` change file to make the TEX code contained in the section "⟨ Display node p ⟩" available to other parts of HiTEX as a function. The function is then used to display debugging output.

This requires the following changes:

First, we prevent the code expansion for this section in `ctex.w` and insert the function call instead. We replace

old

```
⟨ Display node p ⟩;
```

by

new

```
display_node(p);
```

Then we replace a few lines later the line that defines the code section

old

```
⟨ Display node p ⟩ ≡
```

by the function header and the initial "{" :

new

```
void display_node(pointer p)
{
```

Finally, we insert the closing brace for the function by adding it after the closing brace of the code section:

old

```
}
```

becomes

new

```
} }
```

As you can see, the two parts of a code change are enclosed in square brackets where the opening top bracket is labeled *old* or *new*. While a full understanding of these code changes requires a look at the original TEX code, at least they document what was done and why.

1.2 The textypes.h Header File

The HiTEX program will add a bunch of new functions to TEX that will need access to TEX's internals and the most readable and convenient way to gain this access is using the numerous macros that TEX defines for this purpose. The types.ch change files will take care of writing these macros into the header file textypes.h. There are two related header files: texvars.h and texfuncs.h that contain prototypes for the global variables and functions of TEX. These are not generated from the TEX sources but listed in sections A.4 and A.5 of the Appendix. But back to the generation of textypes.h.

This is accomplished by replacing

old

```
⟨ Compiler directives ⟩          /* all file names are defined dynamically */
  ⟨ Labels in the outer block ⟩
  ⟨ Constants in the outer block ⟩
  ⟨ Types in the outer block ⟩
```

by

new

```
⟨ textypes.h ⟩ ≡
#ifndef _TEX_TYPES_H_
#define _TEX_TYPES_H_
#include "basetypes.h"
  ⟨ Compiler directives ⟩
  ⟨ Constants in the outer block ⟩
  ⟨ Types in the outer block ⟩
#endif
```

As a replacement for the type and macro definitions, the header file textypes.h is included. The header file texfuncs.h contains a prototype for the new function *display_node*.

```
#include <math.h>
#include "textypes.h"
#include "texfuncs.h"
```

It remains to clean up some infelicities: While *pool_name* is a constant, it's a constant variable, and it should not be in textypes.h which is included multiple times. So we delete it from the section ⟨ Constants in the outer block ⟩ and add it to the section ⟨ Global variables ⟩:

old

```
  const char *pool_name =
      "TeXformats:TEX.POOL␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣";
```

becomes

new

```
⟨ Global variables ⟩ ≡
  const char *pool_name =
      "TeXformats:TEX.POOL␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣";
```

2 Modifying TEX

In this section, we explain the different change files that modify TEX. Larger changes are accomplished by replacing entire functions.

2.1 The `kpathsearch` Library

Here we collect the changes made to `ctex.w` to use the kpathsearch library.

To read text files, TEX uses *a_open_in*

old

```
bool a_open_in(alpha_file * f)                /* open a text file for input */
{ reset((*f), name_of_file, "r"); return reset_OK((*f));
}
```

new

```
extern FILE *ktex_open_txt(uint8_t *str);
bool a_open_in(alpha_file * f)                /* open a text file for input */
{ f->f = ktex_open_txt(name_of_file + 1);
  if (f->f != NULL) get(*f);
  return reset_OK((*f));
}
```

To read TEX's font metric files, TEX uses *b_open_in*

old

```
bool b_open_in(byte_file * f)                 /* open a binary file for input */
{ reset((*f), name_of_file, "rb"); return reset_OK((*f));
}
```

new

```
extern FILE *ktex_open_tfm(uint8_t *str);
bool b_open_in(byte_file * f)                 /* open a binary file for input */
{ f->f = ktex_open_tfm(name_of_file + 1);
  if (f->f != NULL) get(*f);
  return reset_OK((*f));
}
```

To read a format file, TEX uses *w_open_in*.

old

```
bool w_open_in(word_file * f)                /* open a word file for input */
{ reset((*f), name_of_file, "rb"); return reset_OK((*f));
}
```

new

```
extern FILE *ktex_open_fmt(uint8_t *str);
bool w_open_in(word_file * f)                /* open a word file for input */
{ f→f = ktex_open_fmt(name_of_file + 1);
  if (f→f ≠ NULL) get(*f);
  return reset_OK((*f));
}
```

To handle the command line, we need a function to add filenames to TEX's input buffer. So we duplicate part of the *input_ln* function and package it into the *input_add_arg* function. The function adds a separating space between filenames and assumes that the filenames do not contain spaces.

old

```
bool input_ln(alpha_file * f, bool bypass_eoln)
```

new

```
extern void ktex_program_name(char *p);
extern bool ktex_init(int argc, char *argv[]);
static void input_add_char(unsigned int c)
{
  if (last ≥ max_buf_stack) { max_buf_stack = last + 1;
    if (max_buf_stack ≡ buf_size)
      ⟨ Report overflow of the input buffer, and abort ⟩;
  }
  buffer[last] = xord[c]; incr(last);
}
void input_add_arg(char *str)
{
  if (last > first ∧ buffer[last − 1] ≠ '␣') input_add_char('␣');
  while (*str ≠ 0) input_add_char(*str++);
}
bool input_ln(alpha_file * f, bool bypass_eoln)
```

The initialization of HiTEX should handle the command line and initialize the kpathsearch library. So we change

old

```
if (¬init_terminal()) exit(0);
```

to

new

```
loc = last = first; ktex_program_name(argv[0]); argv++; argc--;
if (¬ktex_init(argc, argv) ∧ ¬init_terminal( )) exit(0);
```

TEX provides some defaults when searching for an input file that are superseded by the kpathsearch library and can be omitted.

old

```
pack_buffered_name(format_area_length, loc, j − 1);
                                        /* now try the system format file area */
if (w_open_in(&fmt_file)) goto found;
wake_up_terminal;
wterm_ln("Sorry, I can't find that format; will try PLAIN.");
update_terminal;
}                /* now pull out all the stops: try for the system plain file */
pack_buffered_name(format_default_length − format_ext_length, 1, 0); if
    (¬w_open_in(&fmt_file)) { wake_up_terminal;
wterm_ln("I can't find the PLAIN format file!");
```

new

```
wake_up_terminal; wterm_ln("Sorry, I can't find that format.");
update_terminal;
```

old

```
if (cur_ext ≡ empty_string) cur_ext = ⟨ ".tex" ⟩;
```

old

```
if (cur_area ≡ empty_string) {
  pack_file_name(cur_name, TEX_area, cur_ext);
  if (a_open_in(&cur_file)) goto done;
}
```

When TEX opens a font metric file, it used to supply the *TEX_font_area*. The kpathserach library does a better job.

old

```
if (aire ≡ empty_string)
  pack_file_name(nom, TEX_font_area, ⟨ ".tfm" ⟩);
else pack_file_name(nom, aire, ⟨ ".tfm" ⟩);
```

new

```
pack_file_name(nom, empty_string, ⟨ ".tfm" ⟩);
```

Last not least we have to take care of HiTEX's command line arguments.

old

```
int main(void){                                        /* start_here */
```

new

```
int main(int argc, char *argv[]){                    /* start_here */
```

⟨ ktex.c 1 ⟩ ≡ (19)

```
#include "basetypes.h"
#include "error.h"
#include "hformat.h"
#include <kpathsea/kpathsea.h>
  extern void input_add_arg(char *str);
  int dpi = 600;
  char *mf_mode = NULL;
  char *alt_font = NULL;
  char *program_name = NULL;
  void ktex_program_name(char *p)
  { kpse_set_program_name(p, NULL); program_name = kpse_program_name;
  }
  bool ktex_init(int argc, char *argv[])
  { int i;
    kpse_init_prog("HITEX", dpi, mf_mode, alt_font);
    if (argc ≡ 0) return false;          /* no input file on the command line */
    if (argv[0][0] ≠ '&') {
      if (strncmp(kpse_program_name, "hitex", 5) ≡ 0)
        input_add_arg("&hitex");
      else if (strncmp(kpse_program_name, "ctex", 4) ≡ 0)
        input_add_arg("&ctex");
      else if (strncmp(kpse_program_name, "hilatex", 7) ≡ 0)
        input_add_arg("&hilatex");
      else if (strncmp(kpse_program_name, "clatex", 6) ≡ 0)
        input_add_arg("&clatex");
    }
    for (i = 0; i < argc; i++) input_add_arg(argv[i]);
    return true;
  }
  FILE *ktex_open_txt(const_string filename)      /* open a text file for input */
  { char *fname = NULL;
    FILE *f = NULL;
    fname = kpse_find_file(filename, kpse_tex_format, true);
    if (fname ≠ NULL) { f = fopen(fname, "r"); free(fname);
    }
    else QUIT("Unable to find file %s", filename);
    if (f ≡ NULL) QUIT("Unable to open file %s", filename);
    return f;
  }
```

```
char *ktex_find_tfm(const_string filename)  /* find a font metric file for input */
{ char *fname = NULL;
  fname = kpse_find_file(filename, kpse_tfm_format, true);
  if (fname == NULL) {
    kpse_set_program_enabled(kpse_tfm_format, true, kpse_src_texmf_cnf);
    fname = kpse_find_file(filename, kpse_tfm_format, true);
  }
  if (fname == NULL) fprintf(stderr,
        "Unable to find or load font metric file for font %s\n",
        filename), exit(1);
  return fname;
}
FILE *ktex_open_tfm(const_string filename)
                                        /* open a font metric file for input */
{ char *fname = NULL;
  FILE *f = NULL;
  fname = ktex_find_tfm(filename);  f = fopen(fname, "rb");  free(fname);
  return f;
}
FILE *ktex_open_fmt(const_string filename)   /* open a format file for input */
{ char *fname = NULL;
  FILE *f = NULL;
  fname = kpse_find_file(filename, kpse_fmt_format, true);
  if (fname != NULL) { f = fopen(fname, "rb");  free(fname);
  }
  return f;
}
char *ktex_find_pk(const_string filename)          /* find a pk file for input */
{ char *fname = NULL;
  kpse_glyph_file_type file_ret;
  fname = kpse_find_glyph(filename, dpi, kpse_pk_format, &file_ret);
  if (fname == NULL) {
    kpse_set_program_enabled(kpse_pk_format, true, kpse_src_texmf_cnf);
    fname = kpse_find_glyph(filename, dpi, kpse_pk_format, &file_ret);
  }
  if (fname == NULL)
    fprintf(stderr, "Unable to find or load font %s\n", filename), exit(1);
  return fname;
}
```

2.2 The *main* Program

We add three function calls to TEX's main program: at the very beginning we replace the function *init_terminal* by *hinit_terminal*, a function that will process the command line; later we add functions to open and to close the HINT output file just before and after the *main_control* loop.

We remove the program name and all options from the argument vector; of course not without setting the options.

old

```
ktex_program_name(argv[0]); argv++; argc--;
```

new

```
ktex_program_name(argv[0]); argv++; argc--;
while (argc > 0) { int i;
  if (argv[0][0] ≠ '-') break;
  i = hset_option(argv[0] + 1, argv[1]); argv += i; argc -= i;
}
```

Finaly, we enclose the main loop in two functions to open and close the HINT file.

old

```
main_control( );                                   /* come to life */
```

becomes

new

```
hhsize = hsize; hvsize = vsize; hint_open( ); main_control( );
hint_close( );                                     /* come to life */
```

The functions to open and close the HINT file follow in section 7. Here is the function *hinit_terminal*:

⟨ HiTEX routines 2 ⟩ ≡ (5)

```
extern int dpi;
extern char *mf_mode;
extern char *program_name;
int hset_option(char *c, char *d)
{
  if (c[0] ≡ 'c') { option_compress = true; return 1;
  }
#ifdef DEBUG
  else
    if (c[0] ≡ 'd') {
      if (c[1] ≠ 0) { debugflags = strtol(c + 1, NULL, 16); return 1;
      }
      else { debugflags = strtol(d, NULL, 16); return 2;
      }
    }
```

```
#endif
    else
      if (c[0] ≡ 'r') {
        if (c[1] ≠ 0) { dpi = strtol(c + 1, NULL, 10); return 1;
        }
        else { dpi = strtol(d, NULL, 10); return 2;
        }
      }
      else if (c[0] ≡ 'm') {
        if (c[1] ≠ 0) { mf_mode = c + 1; return 1;
        }
        else { mf_mode = d; return 2;
        }
      }
      else ⟨ explain usage 4 ⟩
  }
```

Used in 107.

Handling the command line is done in two steps: First we handle command line options and then we copy the remainder to the input buffer. This allows to start HiTEX with a command line like "`hinitex -c &plain hello.tex`" or just "`hinitex hello.tex`".

```
⟨ explain usage 4 ⟩ ≡                                                    (4)
  { fprintf(stderr, "Usage: %s [options] [&format] inputfile\n",
        program_name);
    fprintf(stderr, "Options:\n"
    "\t -c          \t enable compression of section 1 and 2\n"
    "\t -r <number>\t set the resolution to <number> dpi\n"
    "\t -m <mode>  \t set the METAFONT mode\n");
#ifdef DEBUG
    fprintf(stderr, "\t -d XX       \t hexadecimal value. OR\
        \t together these values:\n");
    fprintf(stderr, "\t\t\t XX=%03X \t basic debugging\n", DBGBASIC);
    fprintf(stderr, "\t\t\t XX=%03X \t tag debugging\n", DBGTAGS);
    fprintf(stderr, "\t\t\t XX=%03X \t node debugging\n", DBGNODE);
    fprintf(stderr, "\t\t\t XX=%03X \t definition debugging\n", DBGDEF);
    fprintf(stderr, "\t\t\t XX=%03X \t directory debugging\n", DBGDIR);
    fprintf(stderr, "\t\t\t XX=%03X \t range debugging\n", DBGRANGE);
    fprintf(stderr, "\t\t\t XX=%03X \t float debugging\n", DBGFLOAT);
    fprintf(stderr, "\t\t\t XX=%03X \t compression debugging\n",
        DBGCOMPRESS);
    fprintf(stderr, "\t\t\t XX=%03X \t buffer debugging\n", DBGBUFFER);
    fprintf(stderr, "\t\t\t XX=%03X \t TeX debugging\n", DBGTEX);
    fprintf(stderr, "\t\t\t XX=%03X \t font debugging\n", DBGFONT);
#endif
    exit(1);
  }
```

Used in 2.

2.3 The Page Builder

The point where HiTEX goes an entirely different path than TEX is the page builder: Instead of building a page, HiTEX writes a HINT file.

We remove the *build_page* routine entirely from TEX:

old

```
void build_page(void)    /* append contributions to the current page */
{ pointer p;                                  /* the node being appended */
  pointer q, r;                               /* nodes being examined */
  int b, c;                                   /* badness and cost of current page */
  int pi;                                     /* penalty to be added to the badness */
  uint8_t n;                                  /* insertion box number */
  scaled delta, h, w;                         /* sizes used for insertion calculations */
  if ((link(contrib_head) ≡ null) ∨ output_active) return;
  do {
  resume: p = link(contrib_head);
    ⟨ Update the values of last_glue, last_penalty, and last_kern ⟩;
    ⟨ Move node p to the current page; if it is time for a page break, put
        the nodes following the break back onto the contribution list,
        and return to the user's output routine if there is one ⟩;
  } while (¬(link(contrib_head) ≡ null)); ⟨ Make the contribution list
        empty by setting its tail to contrib_head ⟩;
}
```

The new *build_page* routine of HiTEX is described in section 4.2.

We add one more change here: When TEX invokes the *its_all_over* function—you can guess when—it appends an empty line and a very large neagtive penalty to the page. The new line now has an extended dimension as its width and TEX's penalty is so large that it is not even a legal penalty in a HINT file. The additional information that such a huge penalty contains for an output routine is not needed in HiTEX because in HiTEX there is no output routine. So we reduce this penalty to a normal *eject_penalty*.

old

```
tail_append(new_null_box()); width(tail) = hsize;
```

becomes

new

```
tail_append(new_set_node());
set_extent(tail) = hget_xdimen_no(dimen_par(hsize_code),
    dimen_par_hfactor(hsize_code), dimen_par_vfactor(hsize_code));
```

old

```
tail_append(new_penalty(−°10000000000));
```

becomes

new

```
tail_append(new_penalty(eject_penalty));
```

2.4 Adding new whatsit Nodes

TEX has a mechanism to extend it: the whatsit nodes. For new concepts like baseline specifications, paragraphs, and boxes with unknown dimensions, we define now special whatsit nodes. The new node definitions are supplemented by procedures to print them, copy them, delete them, and handle them in various contexts.

2.4.1 Definitions

But let's start at the beginning. Because we want to convert some of the new nodes into regular box nodes, we have to make sure that they have the same size. This is achived by increasing the size of box nodes.

old

```
#define box_node_size 7 /* number of words to allocate for a box node */
```

new

```
#define box_node_size 9
    /* number of words to allocate for a box, set, or pack node */
```

We add the definitions for the new nodes after TEX's last entry for the whatsit node, the *open_name* node. We define *par_node* for parameters, *graf_node* for paragraphs, *disp_node* for displayed equations, *baseline_node* for baseline specifications, *image_node* for images, *hpack_node* and *vpack_node* for boxes that still need to be passed to *hpack* and *vpack* respectively, and *hset_node* and *vset_node* that still need setting the glue. The node type *align_node* is about to disappear in the final version of HiTEX.

old

```
#define open_ext(X) link(X+2)
                    /* string number of file extension for open_name */
```

becomes

new

```
#define open_ext(X) link(X+2)
                    /* string number of file extension for open_name */
#define par_node 6 /* subtype that records the change of a parameter */
#define par_node_size 3 /* number of memory words in a par_node */
#define par_type(X) type(X+1)                    /* type of parameter */
#define int_type 0                               /* type of an int_par node */
#define dimen_type 1                             /* type of an dimen_par node */
#define glue_type 2                              /* type of an glue_par node */
#define par_number(X) subtype(X+1)               /* the parameter number */
#define par_value(X) mem[X+2]                    /* the parameter value */
```

```
#define graf_node 7 /* subtype that records a paragraph */
#define graf_node_size 6 /* number of memory words in a graf_node */
#define graf_penalty(X) mem[X+1].i /* the final penalty */
#define graf_continue(X) type(X+2) /* keep prev_graf */
#define graf_first_line(X) link(X+2) /* line number of first line */
#define graf_params(X) link(X+3) /* list of parameter nodes */
#define graf_list_offset 4
#define graf_list(X) link(X+graf_list_offset) /* list of content nodes */
#define graf_extent(X) mem[X+5].i /* the extent index */

#define disp_node 8 /* subtype that records a math display */
#define disp_node_size 3 /* number of memory words in a disp_node */
#define display_left(X) type(X+1) /* 1=left 0=right */
#define display_no_bs(X) subtype(X+1)
                                  /* prev_depth ≡ ignore_depth */
#define display_params(X) link(X+1) /* list of parameter nodes */
#define display_formula(X) link(X+2) /* formula list */
#define display_eqno(X) info(X+2) /* box with equation number */

#define baseline_node 9 /* subtype that records a baseline_skip */
#define baseline_node_size small_node_size
  /* This is 2; we will convert baseline nodes to glue nodes */
#define baseline_node_no(X) mem[X+1].i /* baseline reference */

#define image_node 10 /* subtype that records an image */
#define image_node_size 9
                  /* width, depth, height, shift_amount like a hbox */
#define image_width(X) width(X) /* width of image */
#define image_height(X) height(X) /* height of image */
#define image_depth(X) depth(X) /* depth of image==zero */
#define image_no(X) mem[X+4].i /* the section number */
#define image_stretch(X) mem[X+5].sc /* stretchability of image */
#define image_shrink(X) mem[X+6].sc /* shrinkability of image */
#define image_stretch_order(X) stretch_order(X+7)
#define image_shrink_order(X) shrink_order(X+7)
#define image_name(X) link(X+7) /* string number of file name */
#define image_area(X) info(X+8) /* string number of file area */
#define image_ext(X) link(X+8) /* string number of file extension */

#define hpack_node 12 /* a hlist that needs to go to hpack */
#define vpack_node 13 /* a vlist that needs to go to vpackage */
#define pack_node_size box_node_size /* a box node up to list_ptr */
#define pack_m(X) type(X+list_offset)
                                  /* either additional or exactly */
#define pack_limit(X) mem[(X)+glue_offset].sc
                                  /* depth limit in vpack */
#define pack_extent(X) mem[(X)+1+glue_offset].i /* reference */
```

```
#define hset_node 14 /* represents a hlist that needs glue_set */
#define vset_node 15 /* represents a vlist that needs glue_set */
#define set_node_size box_node_size /* up to list_ptr like a box node */
#define set_stretch_order glue_sign
#define set_shrink_order glue_order
#define set_stretch(X) mem[(X) + glue_offset].sc
                                        /* replaces glue_set */
#define set_extent(X) pack_extent(X) /* reference */
#define set_shrink(X) mem[(X) + 2 + glue_offset].sc
#define align_node 16 /* represents an alignment */
#define align_node_size 4
#define align_preamble(X) info(X + 1) /* the preamble */
#define align_list(X) link(X + 1) /* the unset rows/columns */
#define align_extent(X) mem[X + 2].i /* the extent of the alignment */
#define align_m(X) type(X + 3) /* either additional or exactly */
#define align_v(X) subtype(X + 3) /* true if vertical */
```

2.4.2 Printing command codes

Next are modifications to the *print_cmd_chr* routine which prints a symbolic interpretation of a command code and its modifier. We add the new cases after the *special_node* case.

old

```
case special_node: print_esc(⟨ "special" ⟩); break;
```

becomes

new

```
case special_node: print_esc(⟨ "special" ⟩); break;
case par_node: print_str("parameter"); break;
case graf_node: print_str("paragraf"); break;
case disp_node: print_str("display"); break;
case baseline_node: print_str("baselineskip"); break;
case hpack_node: print_str("hpack"); break;
case vpack_node: print_str("vpack"); break;
case hset_node: print_str("hset"); break;
case vset_node: print_str("vset"); break;
case image_node: print_str("image"); break;
case align_node: print_str("align"); break;
```

2.4.3 Executing

When an *extension* command occurs in *main_control*, the *do_extension* routine is called. It needs cases for the new node types.

old

```
case special_node: ⟨ Implement \special ⟩ break;
```

becomes

new

```
case special_node: ⟨ Implement \special ⟩ break;
case par_node: case graf_node: case disp_node: case baseline_node:
  case hpack_node: case vpack_node: case hset_node: case vset_node:
  case align_node: break;
case image_node: break;                    /* see section 2.10, page 46 */
```

2.4.4 Displaying

To display the new nodes, TEX's case statement for whatsit nodes needs to be extended.

old

```
default: print_str("whatsit?");
  }
```

becomes

new

```
case par_node: print_str("\\parameter␣"); print_int(par_type(p));
  print_char(','); print_int(par_number(p)); print_char(':');
  print_int(par_value(p).i); break;
case graf_node: print_str("\\paragraf("); print_int(graf_penalty(p));
  print_char('␣'); print_int(graf_continue(p)); print_char('␣');
  print_int(graf_first_line(p)); print_char(')');
  node_list_display(graf_params(p)); node_list_display(graf_list(p)); break;
case disp_node: print_str("\\display␣");
  node_list_display(display_eqno(p));
  if (display_left(p)) print_str("left␣");
  else print_str("right␣");
  node_list_display(display_formula(p));
  node_list_display(display_params(p)); break;
case baseline_node: print_str("\\baselineskip␣");
  print_baseline_skip(baseline_node_no(p)); break;
case hset_node: case vset_node: print_char('\\');
  print_char(subtype(p) ≡ hset_node ? 'h' : 'v'); print_str("set(");
  print_scaled(height(p)); print_char('+'); print_scaled(depth(p));
  print_str(")x"); print_scaled(width(p));
  if (shift_amount(p) ≠ 0) { print_str(",␣shifted␣");
    print_scaled(shift_amount(p));
  }
  if (set_stretch(p) ≠ 0) { print_str(",␣stretch␣");
    print_glue(set_stretch(p), set_stretch_order(p), ⟨ "pt" ⟩);
```

```
    }
    if (set_shrink(p) ≠ 0) { print_str(", shrink "); print_glue(set_shrink(p), set_shrink_order(p), ⟨ "pt" ⟩); }
    print_str(", extent "); print_xdimen(set_extent(p));
    node_list_display(list_ptr(p)); /* recursive call */
    break;
  case hpack_node: case vpack_node: print_char('\\');
    print_char(subtype(p) ≡ hpack_node ? 'h' : 'v'); print_str("pack(");
    print_str(pack_m(p) ≡ exactly ? "exactly " : "additional ");
    print_xdimen(pack_extent(p));
    if (subtype(p) ≡ vpack_node ∧ pack_limit(p) ≠ max_dimen) {
      print_str(", limit "); print_scaled(pack_limit(p));
    }
    print_char(')'); node_list_display(list_ptr(p)); break;
  case image_node: print_str("\\image("); print_char('(');
    print_scaled(image_height(p)); print_char('+');
    print_scaled(image_depth(p)); print_str(")x");
    print_scaled(image_width(p));
    if (image_stretch(p) ≠ 0) { print_str(" plus "); print_glue(image_stretch(p), image_stretch_order(p), ⟨ "pt" ⟩); }
    if (image_shrink(p) ≠ 0) { print_str(" minus "); print_glue(image_shrink(p), image_shrink_order(p), ⟨ "pt" ⟩); }
    print_str(", section "); print_int(image_no(p));
    if (image_name(p) ≠ 0) { print_str(", "); print(image_name(p)); }
    break;
  case align_node: print_str("\\align(");
    print_str(align_m(p) ≡ exactly ? "exactly " : "additional ");
    print_xdimen(align_extent(p)); print_char(')');
    node_list_display(align_preamble(p)); print_char(':');
    node_list_display(align_list(p)); break;
  default: print_str("whatsit?");
    }
```

We add declarations for the procedures to print extended dimensions and baseline skips.

```
⟨ Basic printing ⟩ ≡
  extern void print_xdimen(int i);
  extern void print_baseline_skip(int i);
```

2.4.5 Copying

Next is the extension of TEX's case statement for copying nodes:

old

```
default: confusion(⟨ "ext2" ⟩);
```

becomes

new

```
case par_node:
  { r = get_node(par_node_size);
    if (par_type(p) ≡ glue_type) add_glue_ref(par_value(p).i);
    words = par_node_size;
  } break;
case graf_node:
  { r = get_node(graf_node_size);
    graf_params(r) = copy_node_list(graf_params(p));
    graf_list(r) = copy_node_list(graf_list(p)); words = graf_node_size − 1;
  } break;
case disp_node:
  { r = get_node(disp_node_size); display_left(r) = display_left(p);
    display_eqno(r) = copy_node_list(display_eqno(p));
    display_formula(r) = copy_node_list(display_formula(p));
    display_params(r) = copy_node_list(display_params(p));
    words = disp_node_size − 2;
  } break;
case baseline_node:
  { r = get_node(baseline_node_size); words = baseline_node_size;
  } break;
case hpack_node: case vpack_node:
  { r = get_node(pack_node_size); pack_m(r) = pack_m(p);
    list_ptr(r) = copy_node_list(list_ptr(p));
    pack_extent(r) = pack_extent(p); pack_limit(r) = pack_limit(p);
    words = pack_node_size − 3;
  } break;
case hset_node: case vset_node:
  { r = get_node(set_node_size); mem[r + 8] = mem[p + 8];
    mem[r + 7] = mem[p + 7]; mem[r + 6] = mem[p + 6];
    mem[r + 5] = mem[p + 5];                    /∗ copy the last four words ∗/
    list_ptr(r) = copy_node_list(list_ptr(p));   /∗ this affects mem[r + 5] ∗/
    words = 5;
  } break;
case image_node: r = get_node(image_node_size);
  words = image_node_size; break;
case align_node:
  { r = get_node(align_node_size);
    align_preamble(r) = copy_node_list(align_preamble(p));
    align_list(r) = copy_node_list(align_list(p));
    words = align_node_size − 1;
```

```
  } break;
default: confusion(⟨ "ext2" ⟩);
```

2.4.6 Deleting

To delete a node, we add new cases:

old

```
case close_node: case language_node: free_node(p, small_node_size); break;
```

becomes

new

```
case close_node: case language_node: free_node(p, small_node_size); break;
case par_node:
  if (par_type(p) ≡ glue_type) fast_delete_glue_ref(par_value(p).i);
  free_node(p, par_node_size); break;
case graf_node: flush_node_list(graf_params(p));
  flush_node_list(graf_list(p)); free_node(p, graf_node_size); break;
case disp_node: flush_node_list(display_eqno(p));
  flush_node_list(display_formula(p)); flush_node_list(display_params(p));
  free_node(p, disp_node_size); break;
case baseline_node: free_node(p, baseline_node_size); break;
case hpack_node: case vpack_node: flush_node_list(list_ptr(p));
  free_node(p, pack_node_size); break;
case hset_node: case vset_node: flush_node_list(list_ptr(p));
  free_node(p, set_node_size); break;
case image_node: free_node(p, image_node_size); break;
case align_node: flush_node_list(align_preamble(p));
  flush_node_list(align_list(p)); free_node(p, align_node_size); break;
```

When shiping out data in *hlist_out* and *vlist_out*, TEX uses the following code. We can probably ignore the change.

old

```
default: confusion(⟨ "ext4" ⟩);
```

becomes

new

```
case par_node: case graf_node: case disp_node: case baseline_node:
  case hpack_node: case vpack_node: case hset_node: case vset_node:
  case image_node: case align_node: break;
default: confusion(⟨ "ext4" ⟩);
```

2.4.1 Freeing

Because some of the new nodes occur at places where TEX originaly only handles box nodes, there are many places in TEX where we can no longer just say *free_node*(*b*, *box_node_size*) to free a (box) node. Instead, we have to use the more general routine *flush_node_list*. This leads to a long series of simple replacements:

In *rebox*

old

free_node(*b*, *box_node_size*);

becomes

new

list_ptr(*b*) = *null*; *flush_node_list*(*b*);

In ⟨ Process node-or-noad ⟩

old

free_node(*z*, *box_node_size*);

becomes

new

list_ptr(*z*) = *null*; *flush_node_list*(*z*);

In ⟨ Attach the limits ... ⟩ (twice)

old

{ *free_node*(*x*, *box_node_size*); *list_ptr*(*v*) = *y*;

becomes

new

{ *list_ptr*(*x*) = *null*; *flush_node_list*(*x*); *list_ptr*(*v*) = *y*;

and

old

if (*math_type*(*subscr*(*q*)) ≡ *empty*) *free_node*(*z*, *box_node_size*);

becomes

new

if (*math_type*(*subscr*(*q*)) ≡ *empty*)
{ *list_ptr*(*z*) = *null*; *flush_node_list*(*z*); }

In *make_scripts*

old

free_node(*z*, *box_node_size*);

becomes

new

list_ptr(*z*) = *null*; *flush_node_list*(*z*);

In *vsplit*

old

$q = prune\_page\_top(q);\ \ p = list\_ptr(v);\ \ free\_node(v, box\_node\_size);$

becomes

new

$q = prune\_page\_top(q);\ \ p = list\_ptr(v);\ \ list\_ptr(v) = null;$
$flush\_node\_list(v);$

In ⟨ Wrap up the box specified by node r ... ⟩ (twice)

old

$free\_node(temp\_ptr, box\_node\_size);\ \ wait = true;$

becomes

new

$list\_ptr(temp\_ptr) = null;\ \ flush\_node\_list(temp\_ptr);\ \ wait = true;$

and

old

$free\_node(box(n), box\_node\_size);$

becomes

new

$list\_ptr(box(n)) = null;\ \ flush\_node\_list(box(n));$

In ⟨ Cases of *handle_right_brace* ... ⟩

old

$free\_node(p, box\_node\_size);$

becomes

new

$list\_ptr(p) = null;\ \ flush\_node\_list(p);$

In *unpackage*

old

$free\_node(p, box\_node\_size);$

becomes

new

$list\_ptr(p) = null;\ \ flush\_node\_list(p);$

In ⟨ Squeeze the equation ... ⟩ (twice)

old

$\{\ \ free\_node(b, box\_node\_size);$

becomes

new

```
{ list_ptr(b) = null; flush_node_list(b);
```

and

old

```
{ free_node(b, box_node_size);
```

becomes

new

```
{ list_ptr(b) = null; flush_node_list(b);
```

2.4.2 Unpacking

In the function *unpackage* the pointer p might now point to an hset, vset, hpack or vpack node instead of a vbox or hbox node. We have to adapte the following test to this new situation.

old

```
if ((abs(mode) ≡ mmode) ∨ ((abs(mode) ≡ vmode) ∧ (type(p) ≠
    vlist_node)) ∨ ((abs(mode) ≡ hmode) ∧ (type(p) ≠ hlist_node)))
```

becomes

new

```
if ((abs(mode) ≡ mmode) ∨
    ((abs(mode) ≡ vmode) ∧ (type(p) ≠ vlist_node) ∧ (type(p) ≠
    whatsit_node ∨ (subtype(p) ≠ vset_node ∧ subtype(p) ≠ vpack_node))) ∨
    ((abs(mode) ≡ hmode) ∧ (type(p) ≠ hlist_node) ∧ (type(p) ≠
    whatsit_node ∨ (subtype(p) ≠ hset_node ∧ subtype(p) ≠ hpack_node))))
```

We continue to define auxiliar functions to create the new nodes.

2.4.3 Creating

The following functions create nodes for paragraphs, displayed equations, baseline skips, hpack nodes, vpack nodes, hset nodes, vset nodes, and image nodes.

⟨ HiTEX routines 2 ⟩ +≡ (3)

```
pointer new_graf_node(void)
{ pointer p;
  p = get_node(graf_node_size); type(p) = whatsit_node;
  subtype(p) = graf_node; graf_params(p) = null; graf_list(p) = null; return p;
}
pointer new_disp_node(void)
{ pointer p;
  p = get_node(disp_node_size); type(p) = whatsit_node;
  subtype(p) = disp_node; display_params(p) = null;
  display_formula(p) = null; display_eqno(p) = null; return p;
}
```

```
pointer new_baseline_node(pointer bs, pointer ls, scaled lsl)
{ pointer p;
  p = get_node(baseline_node_size); type(p) = whatsit_node;
  subtype(p) = baseline_node;
  baseline_node_no(p) = hget_baseline_no(bs, ls, lsl); return p;
}
pointer new_pack_node(void)
{ pointer p;
  p = get_node(pack_node_size); type(p) = whatsit_node;
  subtype(p) = hpack_node;
  width(p) = depth(p) = height(p) = shift_amount(p) = 0;
  pack_limit(p) = max_dimen; list_ptr(p) = null; return p;
}
pointer new_set_node(void)
{ pointer p;
  p = get_node(set_node_size); type(p) = whatsit_node; subtype(p) = hset_node;
  width(p) = depth(p) = height(p) = shift_amount(p) = set_stretch(p) =
      set_shrink(p) = set_extent(p) = 0; list_ptr(p) = null; return p;
}
pointer new_image_node(str_numbern, str_numbera, str_numbere)
{ pointer p;
  int i;
  char *fn;
  int l;
  p = get_node(image_node_size); type(p) = whatsit_node;
  subtype(p) = image_node; image_name(p) = n; image_area(p) = a;
  image_ext(p) = e; fn = hfile_name(n, a, e); l = strlen(fn);
  if (l ≥ file_name_size) QUIT("Filename␣of␣image␣file␣too␣long");
  i = hnew_file_section(fn); image_no(p) = i;
  image_width(p) = image_height(p) = image_stretch(p) = image_shrink(p) = 0;
  image_shrink_order(p) = image_stretch_order(p) = normal;
  return p;
}
```

2.4.4 Parameter nodes

Parameter nodes are added to the current list using the *add_par_node* function.

⟨ Create the parameter node 6 ⟩ ≡ (6)

```
p = get_node(par_node_size); type(p) = whatsit_node; subtype(p) = par_node;
par_type(p) = t; par_number(p) = n;
```

Used in 8.

⟨ Initialize the parameter node 7 ⟩ ≡ (7)

```
if (t ≡ int_type) par_value(p).i = v;
else if (t ≡ dimen_type) par_value(p).sc = v;
else if (t ≡ glue_type) { par_value(p).i = v; add_glue_ref(par_value(p).i); }
```

```
else { free_node(p, par_node_size); QUIT("Undefined␣parameter␣type␣%d", t);
}
```
Used in 8.

⟨ HiTEX routines 2 ⟩ +≡ (8)

```
void add_par_node(uint8_t t, uint8_t n, int v)
{ pointer p;
  ⟨ Create the parameter node 6 ⟩
  ⟨ Initialize the parameter node 7 ⟩
  link(p) = link(temp_head); link(temp_head) = p;
}
```

2.5 Extended Dimensions

An extended dimension is a linar function of `hsize` and `vsize`, and whenever TEX works with a dimension, we want it to be able to deal with an extended dimension. This implies many changes throughout TEX's sources. So let's get started.

Dimensions are stored in the table of equivalents, the *eqtb* array. We use two parallel arrays *hfactor_eqtb* and *vfactor_eqtb* to be able to work with extended dimensions. We start with providing access macros.

old

```
#define dimen(X) eqtb[scaled_base + X].sc
#define dimen_par(X) eqtb[dimen_base + X].sc /* a scaled quantity */
```

becomes

new

```
#define dimen(X) eqtb[scaled_base + X].sc
#define dimen_par(X) eqtb[dimen_base + X].sc /* a scaled quantity */
#define dimen_hfactor(X) hfactor_eqtb[scaled_base + X].sc
#define dimen_vfactor(X) vfactor_eqtb[scaled_base + X].sc
#define dimen_par_hfactor(X) hfactor_eqtb[dimen_base + X].sc
#define dimen_par_vfactor(X) vfactor_eqtb[dimen_base + X].sc
```

The new arrays are initialized with zero.

old

```
for (k = dimen_base; k ≤ eqtb_size; k++) eqtb[k].sc = 0;
```

becomes

new

```
for (k = dimen_base; k ≤ eqtb_size; k++)
  hfactor_eqtb[k].sc = vfactor_eqtb[k].sc = eqtb[k].sc = 0;
```

The definitions of *hfactor_eqtb* and *vfactor_eqtb* complement the definition of *eqtb*. Two special variables are added to keep track of changes to *hsize* and *vsize* before these values are finaly fixed after calling *freeze_page_specs*.

old

```
memory_word eqtb0[eqtb_size − active_base + 1],
    ∗const eqtb = eqtb0 − active_base;
```

becomes

new

```
memory_word eqtb0[eqtb_size − active_base + 1],
    ∗const eqtb = eqtb0 − active_base;
memory_word hfactor_eqtb0[dimen_pars + 256] = {{{0}}},
    ∗const hfactor_eqtb = hfactor_eqtb0 − dimen_base;
memory_word vfactor_eqtb0[dimen_pars + 256] = {{{0}}},
    ∗const vfactor_eqtb = vfactor_eqtb0 − dimen_base;
scaled hhsize = 0, hvsize = 0;
```

Because dimensions might go on the save stack, we triplicate it as well.

old

```
memory_word save_stack[save_size + 1];
```

becomes

new

```
memory_word save_stack[save_size + 1];
memory_word save_hfactor[save_size + 1];
memory_word save_vfactor[save_size + 1];
```

We extend the *saved* macro and adapt the *eq_save* function.

old

```
#define saved(X) save_stack[save_ptr + X].i
```

becomes

new

```
#define saved(X) save_stack[save_ptr + X].i
#define saved_hfactor(X) save_hfactor[save_ptr + X].i
#define saved_vfactor(X) save_vfactor[save_ptr + X].i
```

and

old

```
else { save_stack[save_ptr] = eqtb[p]; incr(save_ptr);
```

becomes

new

```
else { save_stack[save_ptr] = eqtb[p];
if (p ≥ dimen_base) { save_hfactor[save_ptr] = hfactor_eqtb[p];
  save_vfactor[save_ptr] = vfactor_eqtb[p];
}
incr(save_ptr);
```

We store *hhsize* and *hvsize* in the *dimen_defined* array once we have started to build pages. We keep track of changes to *hsize* and *vsize* in these variables, but we prevent assignments to `hsize` and `vsize` on the global level; these values are determined by the HINT viewer. We simply ignore such assignments, because HINT should be able to process any TEX file, and modifications of `hsize` or `vsize` are quite common.

In the function *eq_word_define*

old

```
{ if (xeq_level[p] ≠ cur_level)
```

becomes

new

```
{ if (cur_level ≡ level_one) {
    if (p ≡ dimen_base + hsize_code) { hhsize = w; return; }
    if (p ≡ dimen_base + vsize_code) { hvsize = w; return; }
  }
  if (xeq_level[p] ≠ cur_level)
```

On the local level, we store dimensions always together with their hfactor and vfactor.

old

```
eqtb[p].i = w;
```

becomes

new

```
eqtb[p].i = w;
if (p ≥ dimen_base) { hfactor_eqtb[p].i = cur_hfactor;
  vfactor_eqtb[p].i = cur_vfactor;
}
```

In *geq_word_define* which handles global definitions, we need similar changes:

old

```
{ eqtb[p].i = w; xeq_level[p] = level_one;
```

becomes

new

```
{ xeq_level[p] = level_one;
  if (p ≡ dimen_base + hsize_code) hhsize = w;
  else if (p ≡ dimen_base + vsize_code) hvsize = w;
  else { eqtb[p].i = w;
    if (p ≥ dimen_base) { hfactor_eqtb[p].i = cur_hfactor;
      vfactor_eqtb[p].i = cur_vfactor;
    }
  }
```

and

old

```
{ eqtb[p] = save_stack[save_ptr]; xeq_level[p] = l;
```

becomes

new

```
{ eqtb[p] = save_stack[save_ptr];
  if (p ≥ dimen_base) { hfactor_eqtb[p] = save_hfactor[save_ptr];
    vfactor_eqtb[p] = save_vfactor[save_ptr];
  }
  xeq_level[p] = l;
```

Now we can look at TEX's computations with extended dimensions.

We start with a test macro that determines if *cur_val* has an associated nonzero *cur_hfactor* or *cur_vfactore*.

old

```
#define tok_val 5 /* token lists */
```

becomes

new

```
#define tok_val 5 /* token lists */
#define has_factor (cur_hfactor ≠ 0 ∨ cur_vfactor ≠ 0)
```

We supplement the *cur_val* variable with variables for the current factors.

old

```
int cur_val; /* value returned by numeric scanners */
```

becomes

new

```
int cur_val, cur_hfactor, cur_vfactor;
                /* value returned by numeric scanners */
```

Here comes TEX's code to handle an `assign`:

old

```
case assign_dimen: scanned_result(eqtb[m].sc)(dimen_val) break;
```

It becomes

new

```
case assign_dimen: scanned_result(eqtb[m].sc)(dimen_val);
  if (m ≥ dimen_base)
  { cur_hfactor = hfactor_eqtb[m].sc; cur_vfactor = vfactor_eqtb[m].sc; }
  else cur_hfactor = cur_vfactor = 0; break;
```

The scanning of an extended dimension requires this change:

old

```
case dimen_val: cur_val = dimen(cur_val); break;
```

becomes

new

```
case dimen_val: cur_hfactor = dimen_hfactor(cur_val);
  cur_vfactor = dimen_vfactor(cur_val); cur_val = dimen(cur_val); break;
```

Negation is simple:

old

```
  else negate(cur_val);
```

becomes

new

```
  else { negate(cur_val); negate(cur_hfactor); negate(cur_vfactor); }
```

The function *scan_dimen* starts with initializing variables. We extend them with the initialization of *cur_hfactor* and *cur_vfactor*.

old

```
  f = 0; arith_error = false; cur_order = normal; negative = false;
```

becomes

new

```
  f = 0; arith_error = false; cur_order = normal; negative = false;
  cur_hfactor = cur_vfactor = 0;
```

At the end of *scan_dimen*, before attaching the sign, TEX checks for overflow:

old

```
  attach_sign: if (arith_error ∨ (abs(cur_val) ≥ °10000000000))
```

becomes

new

```
  attach_sign: if (arith_error ∨ (abs(cur_val) ≥
      °10000000000) ∨ (abs(cur_hfactor) ≥
      °10000000000) ∨ (abs(cur_vfactor) ≥ °10000000000))
```

And now, the sign is attached to all three components of the extende dimension:

old

```
if (negative) negate(cur_val);
```

becomes

new

```
if (negative)
{ negate(cur_val); negate(cur_hfactor); negate(cur_vfactor); }
```

When we implement multiplication of extended dimension, we check that extended dimensions are not multiplied by extended dimensions:

old

```
save_cur_val = cur_val;
```

becomes

new

```
save_cur_val = cur_val;
if (has_factor) {
  print_err("Factor is not constant. Linear component ignored");
  cur_hfactor = cur_vfactor = 0;
}
```

Finally, we multiply all components of the extended dimension by the common factor.

old

```
found: cur_val = nx_plus_y(save_cur_val, v, xn_over_d(v, f, °200000));
```

becomes

new

```
found:
  if (has_factor) { cur_hfactor = nx_plus_y(save_cur_val, cur_hfactor,
        xn_over_d(cur_hfactor, f, unity));
    cur_vfactor = nx_plus_y(save_cur_val, cur_vfactor,
        xn_over_d(cur_vfactor, f, unity));
  }
  cur_val = nx_plus_y(save_cur_val, v, xn_over_d(v, f, unity));
```

A common use of a dimension is passing it to the *hpack* or *vpack* procedures. These procedures will be extended as shown below. For now, we just add the required extra parameters for the `hfactor` and `vfactor`

old

```
#define natural 0, additional
                    /* shorthand for parameters to hpack and vpack */
```

becomes

new

```
#define natural 0,0,0,additional
                    /* shorthand for parameters to hpack and vpack */
```

The *scan_spec* soubroutine scans `hbox` or `vbox` constructions in the users input before *hpack* or *vpack* is called. The initialization of *cur_val* in *scan_spec* is supplemented by an initialization of *cur_hfactor* and *cur_vfactor*.

old

```
else { spec_code = additional; cur_val = 0;
```

becomes

new

```
else { spec_code = additional; cur_val = cur_hfactor = cur_vfactor = 0;
```

The function *scan_spec* ends with saving the new dimension on the save stack.

old

```
saved(0) = spec_code; saved(1) = cur_val; save_ptr = save_ptr + 2;
```

becomes

new

```
saved(0) = spec_code; saved(1) = cur_val;
saved_hfactor(1) = cur_hfactor; saved_vfactor(1) = cur_vfactor;
save_ptr = save_ptr + 2;
```

The new *hpack* and *vpack* routines require significant changes. So be do not construct them using the change file, but we provide them as **extern** functions. Therefore, the function definition of TEX are completely removed.

old

```
pointer hpack(pointer p, scaled w, small_number m)
{ pointer r;                          /* the box node that will be returned */
  pointer q;                                        /* trails behind p */
  scaled h, d, x;                       /* height, depth, and natural width */
  scaled s;                                          /* shift amount */
  pointer g;                          /* points to a glue specification */
  glue_ord o;                                       /* order of infinity */
  internal_font_number f;               /* the font in a char_node */
  four_quarters i;                    /* font information about a char_node */
  eight_bits hd;                 /* height and depth indices for a character */
  last_badness = 0; r = get_node(box_node_size); type(r) = hlist_node;
  subtype(r) = min_quarterword; shift_amount(r) = 0;
  q = r + list_offset; link(q) = p;
  h = 0; ⟨ Clear dimensions to zero ⟩;
  while (p ≠ null) ⟨ Examine node p in the hlist, taking account of its
        effect on the dimensions of the new box, or moving it to the
        adjustment list; then advance p to the next node ⟩;
```

```
  if (adjust_tail ≠ null) link(adjust_tail) = null;
  height(r) = h; depth(r) = d;
  ⟨ Determine the value of width(r) and the appropriate glue setting;
      then return or goto common_ending ⟩;
common_ending: ⟨ Finish issuing a diagnostic message for an overfull or
      underfull hbox ⟩;
end: return r;
}
```

Since *vpack* is actually a shortcut for a call to *vpackage*, we repeat the previous step for *vpackage*.

old

```
pointer vpackage(pointer p, scaled h, small_number m, scaled l)
{ pointer r;                          /* the box node that will be returned */
  scaled w, d, x;                     /* width, depth, and natural height */
  scaled s;                                          /* shift amount */
  pointer g;                          /* points to a glue specification */
  glue_ord o;                                    /* order of infinity */
  last_badness = 0; r = get_node(box_node_size); type(r) = vlist_node;
  subtype(r) = min_quarterword; shift_amount(r) = 0; list_ptr(r) = p;
  w = 0; ⟨ Clear dimensions to zero ⟩;
  while (p ≠ null)
    ⟨ Examine node p in the vlist, taking account of its effect on the
        dimensions of the new box; then advance p to the next node ⟩;
  width(r) = w;
  if (d > l) { x = x + d − l; depth(r) = l;
  }
  else depth(r) = d;
  ⟨ Determine the value of height(r) and the appropriate glue setting;
      then return or goto common_ending ⟩;
common_ending: ⟨ Finish issuing a diagnostic message for an overfull or
      underfull vbox ⟩;
end: return r;
}
```

Next a series of function calls with obvious changes:

old

```
return hpack(b, w, exactly);
```

becomes

new

```
return hpack(b, w, 0, 0, exactly);
```

old

$p = hpack(preamble, saved(1), saved(0));\quad overfull\_rule = rule\_save;$

becomes

new

$p = hpack(preamble, saved(1), saved\_hfactor(1), saved\_vfactor(1),$
$\quad saved(0));\quad overfull\_rule = rule\_save;$

old

$p = vpack(preamble, saved(1), saved(0));$

becomes

new

$p = vpack(preamble, saved(1), saved\_hfactor(1), saved\_vfactor(1),$
$\quad saved(0));$

old

$adjust\_tail = adjust\_head;\quad just\_box = hpack(q, cur\_width, exactly);$

becomes

new

$adjust\_tail = adjust\_head;\quad just\_box = hpack(q, cur\_width, 0, 0, exactly);$

old

return $vpackage(p, h, exactly, split\_max\_depth);$

becomes

new

return $vpackage(p, h, 0, 0, exactly, split\_max\_depth);$

and

old

$box(255) = vpackage(link(page\_head), best\_size, exactly, page\_max\_depth);$

becomes

new

$box(255) = vpackage(link(page\_head), best\_size, 0, 0, exactly,$
$\quad page\_max\_depth);$

This one is slighly more complicated: *hpack* is called after a box is completed using the infomation on the save stack:

old

if $(mode \equiv -hmode)\ \ cur\_box = hpack(link(head), saved(2), saved(1));$
else $\{\ \ cur\_box = vpackage(link(head), saved(2), saved(1), d);$

becomes

new

```
if (mode ≡ −hmode) cur_box = hpack(link(head), saved(2),
        saved_hfactor(2), saved_vfactor(2), saved(1));
else { cur_box = vpackage(link(head), saved(2), saved_hfactor(2),
      saved_vfactor(2), saved(1), d);
```

And we continue with replacing function calls:

old

```
p = vpack(link(head), saved(1), saved(0)); pop_nest( );
```

becomes

new

```
p = vpack(link(head), saved(1), saved_hfactor(1), saved_vfactor(1),
      saved(0)); pop_nest( );
```

old

```
b = hpack(p, z − q, exactly);
```

becomes

new

```
b = hpack(p, z − q, 0, 0, exactly);
```

and

old

```
b = hpack(p, z, exactly);
```

becomes

new

```
b = hpack(p, z, 0, 0, exactly);
```

At the end of processing an `advance` command, TEX now takes into account extended dimensions.

old

```
if (q ≡ advance) cur_val = cur_val + eqtb[l].i;
```

becomes

new

```
if (q ≡ advance) { cur_val = cur_val + eqtb[l].i;
  if (l ≥ dimen_base) { cur_hfactor += hfactor_eqtb[l].sc;
    cur_vfactor += vfactor_eqtb[l].sc;
  }
}
```

This concludes the necessary changes to teach TEX about extended dimensions.

2.6 Hyphenation

If the breaking of a paragraph into lines must be postponed because `hsize` is not known, hyphenation must be done for all words because we want to keep hyphenation out of the viewer. TEX distingishes between discretionary breaks inserted by the author of a text and discretionary breaks discovered by the hyphenation routine: the latter are called here "automatic" and are used only in pass two of the line breaking routine.

The change file that follows below contains three types of changes:

- HiTEX restricts the *replace_count* to seven bit to make room for an extra bit to mark automatic discretionary breaks. Assignments to the replace count are therefore replaced by a macro that protects the automatic bit.
- The TEX code for "⟨ Try to hyphenate the following word ⟩" is packaged into a function *hyphenate_word*, so that it can be invoked as needed.
- And calls of the function *line_break* are replaced by calls to *hline_break* to have a hook for inserting HiTEXspecific changes.

old

```
#define replace_count  subtype
                          /* how many subsequent nodes to replace */
```

becomes

new

```
#define replace_count(X)  (mem[X].hh.b1 & #7F)
                          /* how many subsequent nodes to replace */
#define set_replace_count(X,Y)  (mem[X].hh.b1 = (Y) & #7F)
#define set_auto_disc(X)  (mem[X].hh.b1 |= #80)
#define is_auto_disc(X)  (mem[X].hh.b1 & #80)
```

Assignments to *replace_count* are now done using the *set_replace_count* macro.

old

```
replace_count(p) = 0;  pre_break(p) = null;  post_break(p) = null;
```

becomes

new

```
set_replace_count(p,0);  pre_break(p) = null;  post_break(p) = null;
```

The TEX code to hyphenate a word is packaged into a function, so we replace the corresponding section by a function call.

old

```
⟨ Try to hyphenate the following word ⟩;
```

becomes

new

```
hyphenate_word( );
```

Here is another assignment to the replace count:

old

```
flush_node_list(link(q)); replace_count(q) = 0;
```

becomes

new

```
flush_node_list(link(q)); set_replace_count(q, 0);
```

For the definition of the *hyphenate_word* procedure, we insert the function header and definitions for the local variables.

old

```
⟨ Try to hyphenate the following word ⟩ ≡
  { prev_s = cur_p; s = link(prev_s);
```

becomes

new

```
void hyphenate_word(void)
{ pointer q, s, prev_s; /* miscellaneous nodes of temporary interest */
  small_number j; /* an index into hc or hu */
  uint8_t c; /* character being considered for hyphenation */
  prev_s = cur_p; s = link(prev_s);
```

Another assignment, this time we set the automatic bit.

old

```
else { link(s) = r; replace_count(r) = r_count;
```

becomes

new

```
else { link(s) = r; set_replace_count(r, r_count); set_auto_disc(r);
```

We replace the original *line_break* function by *hline_break* to have a hook where we can insert special code for HiTEX.

old

```
else line_break(widow_penalty);
```

becomes

new

```
else hline_break(widow_penalty);
```

Another assignment that constructs explicit discretionarys.

old

```
if (n ≤ max_quarterword) replace_count(tail) = n;
```

becomes

new

```
if (n ≤ ^7F) set_replace_count(tail, n);
```

The function *hline_break* follows: (It seems like the first word in a paragraph does not get hyphenated.)

⟨ HiTEX routines 2 ⟩ +≡ (17)

```
void hline_break(int final_widow_penalty)
{ bool auto_breaking;                    /* is node cur_p outside a formula? */
  pointer r, s;                          /* miscellaneous nodes of temporary interest */
  pointer pp;
  bool par_shape_fix = false;
  if (dimen_par_hfactor(hsize_code) ≡ 0 ∧ dimen_par_vfactor(hsize_code) ≡ 0) {
    line_break(final_widow_penalty);                    /* the easy case */
    return;
  }                                                     /* Get ready to start line breaking */
  pp = new_graf_node( ); graf_penalty(pp) = final_widow_penalty;
  if (par_shape_ptr ≡ null)
    graf_extent(pp) = hget_xdimen_no(dimen_par(hsize_code),
        dimen_par_hfactor(hsize_code), dimen_par_vfactor(hsize_code));
  else { /* here we make a quick fix to cover LaTeXs implementation of
        trivlist using parshape=1 indent length */
    last_special_line = info(par_shape_ptr) − 1;
    if (last_special_line ≠ 0)
      wlog("Warning␣parshape␣with␣n=%d␣not␣yet␣implemented",
          info(par_shape_ptr));
    second_width = mem[par_shape_ptr + 2 ∗ (last_special_line + 1)].sc;
        /* since hsize==0 this is what was subtracted to get the length */
    second_indent = mem[par_shape_ptr + 2 ∗ last_special_line + 1].sc;
    graf_extent(pp) = hget_xdimen_no(dimen_par(hsize_code) + second_width +
        second_indent, dimen_par_hfactor(hsize_code),
        dimen_par_vfactor(hsize_code)); par_shape_fix = true;
  }
  link(temp_head) = link(head);
#ifdef DEBUG
  wlog("\nCalling␣line_break:\n");
  wlog("hang_indent=0x%08X␣hang_after=%d", hang_indent, hang_after);
#if 0
  if (left_skip ≠ zero_glue) { print_str("␣left␣skip="); print_spec(left_skip, 0);
  }
  if (right_skip ≠ zero_glue) { print_str("␣right␣skip=");
    print_spec(right_skip, 0);
  }
#endif
  if (line_skip_limit ≠ 0) wlog("␣line_skip_limit=0x%08X", line_skip_limit);
  wlog("␣prev_graf=0x%08X", prev_graf);
```

```
#endif
    init_cur_lang = prev_graf % °200000; init_l_hyf = prev_graf /°20000000;
    init_r_hyf = (prev_graf /°200000) % °100; pop_nest( );
    graf_continue(pp) = (prev_graf ≠ 0);
#ifdef DEBUG
    wlog("␣prev_graf=0x%08X", prev_graf);
#endif                                          /* Initialize for hyphenating... */
#ifdef INIT
    if (trie_not_ready) init_trie( );
#endif
    cur_lang = init_cur_lang; l_hyf = init_l_hyf; r_hyf = init_r_hyf;
    cur_p = link(temp_head); auto_breaking = true;
#if 0
    prev_p = cur_p;                  /* glue at beginning is not a legal breakpoint */
#endif
    while (cur_p ≠ null) {    /* Call try_break if cur_p is a legal breakpoint... */
      if (is_char_node(cur_p)) {
                    /* Advance cur_p to the node following the present string... */
#if 0
        prev_p = cur_p;
#endif
        do { f = font(cur_p); cur_p = link(cur_p);
        } while (is_char_node(cur_p));
        if (cur_p ≡ null) goto done5; /* mr: no glue and penalty at the end */
      }
      switch (type(cur_p)) {
      case whatsit_node: adv_past(cur_p); break;
      case glue_node:
        if (auto_breaking)                  /* Try to hyphenate the following word */
          hyphenate_word( );
        break;
      case ligature_node: f = font(lig_char(cur_p)); break;
      case disc_node:          /* Try to break after a discretionary fragment... */
        r = replace_count(cur_p); s = link(cur_p);
        while (r > 0) { decr(r); s = link(s);
        }
#if 0
        prev_p = cur_p;
#endif
        cur_p = s; goto done5;
      case math_node: auto_breaking = (subtype(cur_p) ≡ after); break;
      default: break;
      }
#if 0
      prev_p = cur_p;
#endif
```

```
    cur_p = link(cur_p);
  done5: ;
  }
  graf_list(pp) = link(temp_head);                    /* adding parameter nodes */
  link(temp_head) = null;
  add_par_node(int_type, pretolerance_code, pretolerance);
  add_par_node(int_type, tolerance_code, tolerance);
  add_par_node(int_type, line_penalty_code, line_penalty);
  add_par_node(int_type, hyphen_penalty_code, hyphen_penalty);
  add_par_node(int_type, ex_hyphen_penalty_code, ex_hyphen_penalty);
  add_par_node(int_type, club_penalty_code, club_penalty);
  add_par_node(int_type, widow_penalty_code, widow_penalty);
  add_par_node(int_type, broken_penalty_code, broken_penalty);
  add_par_node(int_type, inter_line_penalty_code, inter_line_penalty);
  add_par_node(int_type, double_hyphen_demerits_code, double_hyphen_demerits);
  add_par_node(int_type, final_hyphen_demerits_code, final_hyphen_demerits);
  add_par_node(int_type, adj_demerits_code, adj_demerits);
  add_par_node(int_type, looseness_code, looseness);
  if (par_shape_fix) { add_par_node(int_type, hang_after_code, 0);
    add_par_node(dimen_type, hang_indent_code, second_indent);
  }
  else { add_par_node(int_type, hang_after_code, hang_after);
    add_par_node(dimen_type, hang_indent_code, hang_indent);
  }
  add_par_node(dimen_type, line_skip_limit_code, line_skip_limit);
  add_par_node(dimen_type, emergency_stretch_code, emergency_stretch);
  add_par_node(glue_type, line_skip_code, line_skip);
  add_par_node(glue_type, baseline_skip_code, baseline_skip);
  add_par_node(glue_type, left_skip_code, left_skip);
  add_par_node(glue_type, right_skip_code, right_skip);
  add_par_node(glue_type, par_fill_skip_code, par_fill_skip);
                                          /* par_shape is not yet supported */
  graf_params(pp) = link(temp_head); link(temp_head) = null;
  append_to_vlist(pp);
}
```

2.7 Baseline Skips

TEX will automatically insert a baseline skip between two boxes in a vertical list. The baseline skip is computed to make the distance between the baselines of the two boxes exactly equal to the value of `\baselineskip`. And if that is not possible, because it would make the distance between the descenders of the upper box and the ascenders of the lower box smaller than `\lineskiplimit`, it will insert at least a small amount of glue of size `\lineskip`. As a further complication, if the depth of the upper box has the special value *ignore_depth* the insertion of a baseline skip is suppressed. It is also common practice that authors manipulate the values

of `\baselineskip`, `\lineskiplimit`, `\lineskip`, and `\prevdepth` to change the baseline calculations of TEX.
Of course a prerequisite of the whole computation of a baseline skip is the knowledge of the depth of the upper box and the height of the lower box. With the new types of boxes introduced by HiTEX neither of them might be known. In these cases, the baseline skip computation must be defered to the viewer by inserting a whatsit node of subtype *baseline_node*.

To be able to signal an unknown depth to the code that inserts baseline skips, we use the special depth value *unknown_depth*.

old

```
#define ignore_depth −65536000 /* prev_depth value that is ignored */
```

becomes

new

```
#define ignore_depth (−1000 * unity)
                                    /* prev_depth value that is ignored */
#define unknown_depth (−2000 * unity)
                                    /* prev_depth value that is unknown */
```

This special value is also recognized in the function *show_activities*.

old

```
if (a.sc ≤ ignore_depth) print_str("ignored");
```

becomes

new

```
if (a.sc ≤ ignore_depth) {
  if (a.sc ≤ unknown_depth) print_str("unknown");
  else print_str("ignored");
}
```

The append to vlist procedure inserts baseline skips. We test for cases where the height and depth is known, and if so, execute the ususal computation. Otherwise we create a new baseline node.

old

```
void append_to_vlist(pointer b){ scaled d;
                                /* deficiency of space between baselines */
    pointer p;                                        /* a new glue node */
    if (prev_depth > ignore_depth)
```

becomes

new

```
void append_to_vlist(pointer b)
{ bool height_known;
  height_known = (type(b) ≡ hlist_node ∨ type(b) ≡ vlist_node ∨
      (type(b) ≡ whatsit_node ∧ subtype(b) ≡ hset_node) ∨
      (type(b) ≡ whatsit_node ∧ subtype(b) ≡ image_node));
  if (prev_depth > ignore_depth ∧ height_known)
  { scaled d;                    /* deficiency of space between baselines */
    pointer p;                                        /* a new glue node */
```

and

old

```
  } link(tail) = b;  tail = b;  prev_depth = depth(b);
  }
```

becomes

new

```
  } } else if (prev_depth ≤ unknown_depth ∨ prev_depth > ignore_depth)
  { pointer p;
    p = new_baseline_node(baseline_skip, line_skip, line_skip_limit);
    link(tail) = p;  tail = p;
  }
link(tail) = b;  tail = b;
if (height_known) prev_depth = depth(b);  /* then also depth is known */
else prev_depth = unknown_depth;
}
```

2.8 Displayed Formulas

TEX enters into math mode when it finds a math shift character "$". Then it calls *init_math* which checks for an aditional math shift character to call either ⟨ Go into ordinary math mode ⟩ or ⟨ Go into display math mode ⟩. We remove most of the differences from the latter because we will treat both cases very similar here and will consider the necessary differences in the HINT viewer.

When the shift character that terminats math more is encountered, TEX calls *after_math* which ends with either ⟨ Finish math in text ⟩ or ⟨ Finish displayed math ⟩ . We need to modify the latter, because positioning a displayed equation usually depends on `hsize`, and must be postponed until the viewer knows its value. So all HiTEX can do in this case is create a new whatsit node with subtype *disp_node* and insert it proceeding as if the formula had occured in ordinary math mode.

old

```
⟨ Go into display math mode ⟩ ≡
  { if (head ≡ tail)                    /* '\noindent$$' or '$$ $$' */
    { pop_nest( ); w = −max_dimen;
    }
    else { line_break(display_widow_penalty);
      ⟨ Calculate the natural width, w, by which the characters of the final
          line extend to the right of the reference point, plus two ems; or
          set w: = max_dimen if the non-blank information on that line
          is affected by stretching or shrinking ⟩;
    } /* now we are in vertical mode, working on the list that will contain
          the display */
    ⟨ Calculate the length, l, and the shift amount, s, of the display lines ⟩;
    push_math(math_shift_group); mode = mmode;
    eq_word_define(int_base + cur_fam_code, −1);
    eq_word_define(dimen_base + pre_display_size_code, w);
    eq_word_define(dimen_base + display_width_code, l);
    eq_word_define(dimen_base + display_indent_code, s);
    if (every_display ≠ null)
      begin_token_list(every_display, every_display_text);
    if (nest_ptr ≡ 1) build_page( );
  }
```

is simplified to

new

```
⟨ Go into display math mode ⟩ +≡
  { push_math(math_shift_group); mode = mmode;
    eq_word_define(int_base + cur_fam_code, −1);
    if (every_display ≠ null)
      begin_token_list(every_display, every_display_text);
  }
```

Now lets look at ending display math mode:

We start by removing the local variables of *after_math* that are needed only for ⟨ Finish displayed math ⟩.

old

```
⟨ Local variables for finishing a displayed formula ⟩
```

Now we can replace most of the code of ⟨ Finish displayed math ⟩ by the creation of a new whatsit node with subtype *disp_node*.

old

```
⟨ Finish displayed math ⟩ = cur_mlist = p; cur_style = display_style;
  mlist_penalties = false; mlist_to_hlist( ); p = link(temp_head);
  adjust_tail = adjust_head; b = hpack(p, natural); p = list_ptr(b);
  t = adjust_tail; adjust_tail = null;
  w = width(b); z = display_width; s = display_indent;
```

```
if ((a ≡ null) ∨ danger) { e = 0; q = 0;
}
else { e = width(a); q = e + math_quad(text_size);
}
if (w + q > z)
  ⟨ Squeeze the equation as much as possible; if there is an equation
      number that should go on a separate line by itself, set e: = 0 ⟩;
⟨ Determine the displacement, d, of the left edge of the equation, with
    respect to the line size z, assuming that l = false ⟩;
⟨ Append the glue or equation number preceding the display ⟩;
⟨ Append the display and perhaps also the equation number ⟩;
⟨ Append the glue or equation number following the display ⟩;
resume_after_display( )
```

becomes

new

```
⟨ Finish displayed math ⟩ = cur_mlist = p; cur_style = display_style;
  mlist_penalties = false; mlist_to_hlist( ); p = link(temp_head);
  link(temp_head) = null;
{ pointer q;
  q = new_disp_node( ); display_eqno(q) = a; display_left(q) = l;
  display_formula(q) = p;                    /* adding parameter nodes */
  if (hang_indent ≠ 0) {
    add_par_node(dimen_type, hang_indent_code, hang_indent);
    if (hang_after ≠ 1)
      add_par_node(int_type, hang_after_code, hang_after);
  }
  display_params(q) = link(temp_head); link(temp_head) = null;
  display_no_bs(q) = prev_depth ≤ ignore_depth; tail_append(q);
}                                  /* this is from resume_after_display */
if (cur_group ≠ math_shift_group) confusion(⟨ "display" ⟩);
unsave( ); prev_graf = prev_graf + 3; space_factor = 1000;
⟨ Scan an optional space ⟩;
```

2.9 Alignments

Instead of a single unset node type, we will need three to be able to convert unset nodes back into set or pack nodes.

old

```
#define unset_node 13                    /* type for an unset node */
```

new

```
#define unset_node 13                    /* type for an unset node */
#define unset_set_node 32                /* type for an unset set_node */
#define unset_pack_node 33               /* type for an unset pack_node */
```

old

```
case unset_node: print_str("[]"); break;
```

new

```
case unset_node: case unset_set_node: case unset_pack_node:
  print_str("[]"); break;
```

old

```
case hlist_node: case vlist_node: case unset_node: ⟨ Display box p ⟩ break;
```

new

```
case hlist_node: case vlist_node: case unset_node: case unset_set_node:
  case unset_pack_node: ⟨ Display box p ⟩ break;
```

old

```
  if (type(p) ≡ unset_node)
```

new

```
  if (type(p) ≡ unset_set_node) print_str("set");
  else if (type(p) ≡ unset_pack_node) print_str("pack");
  else if (type(p) ≡ unset_node)
```

old

```
  case hlist_node: case vlist_node: case unset_node: {
  flush_node_list(list_ptr(p));
```

new

```
  case hlist_node: case vlist_node: case unset_node: case unset_set_node:
  case unset_pack_node: { flush_node_list(list_ptr(p));
```

old

```
  case hlist_node: case vlist_node: case unset_node: {
  r = get_node(box_node_size);
```

new

```
  case hlist_node: case vlist_node: case unset_node: case unset_set_node:
  case unset_pack_node: { r = get_node(box_node_size);
```

old

```
  case hlist_node: case vlist_node: case rule_node: case unset_node:
```

new

```
  case hlist_node: case vlist_node: case rule_node: case unset_node:
  case unset_set_node: case unset_pack_node:
```

old

case *hlist_node*: **case** *vlist_node*: **case** *rule_node*: **case** *unset_node*:

new

case *hlist_node*: **case** *vlist_node*: **case** *rule_node*: **case** *unset_node*:
case *unset_set_node*: **case** *unset_pack_node*:

When we record width of columns, we have to take into account that the width of a column might be an extended dimension. Since we can not compute the maximum of two extended dimensions, we mark such columns with a the maximum possible width *max_dimen* plus 1.

So this is what we do, when we ⟨ Package an unset box for the current column and record its width ⟩:

old

{ $adjust\_tail = cur\_tail$; $u = hpack(link(head), natural)$; $w = width(u)$;

new

{ $adjust\_tail = cur\_tail$; $u = hpack(link(head), natural)$;
if $(type(u) \equiv hlist\_node)$ $w = width(u)$;
else $w = max\_dimen + 1$;

and similar

old

else { $u = vpackage(link(head), natural, 0)$; $w = height(u)$;

new

else { $u = vpackage(link(head), natural, 0)$;
if $(type(u) \equiv vlist\_node)$ $w = height(u)$;
else $w = max\_dimen + 1$;

We can not convert all nodes to unset nodes but we need to keep the information about the fifferent node types.

old

$type(u) = unset\_node$; $span\_count(u) = n$;
⟨ Determine the stretch order ⟩;
$glue\_order(u) = o$; $glue\_stretch(u) = total\_stretch[o]$;
⟨ Determine the shrink order ⟩;
$glue\_sign(u) = o$; $glue\_shrink(u) = total\_shrink[o]$;

new

```
if (type(u) ≡ whatsit_node) {
  if (subtype(u) ≡ hset_node ∨ subtype(u) ≡ vset_node)
    type(u) = unset_set_node;
  else type(u) = unset_pack_node;
  span_count(u) = n;
}
else if (type(u) ≡ hlist_node ∨ type(u) ≡ vlist_node) {
  type(u) = unset_node; span_count(u) = n;
  ⟨ Determine the stretch order ⟩;
  glue_order(u) = o; glue_stretch(u) = total_stretch[o];
  ⟨ Determine the shrink order ⟩;
  glue_sign(u) = o; glue_shrink(u) = total_shrink[o];
}
```

We need an indicator x for extended alignments.

old

```
scaled t, w;                                   /* width of column */
```

new

```
scaled t, w;                                   /* width of column */
bool x = false;                        /* indicats an extende alignment */
```

In the last part of function *fin_align*, we handle extended alignments separately:

old

```
⟨ Package the preamble list, to determine the actual tabskip glue
    amounts, and let p point to this prototype box ⟩;
⟨ Set the glue in all the unset boxes of the current list ⟩;
flush_node_list(p); pop_alignment( );
```

new

```
if (x) { ⟨ Handle an alignment that depends on hsize or vsize ⟩
  pop_alignment( );
}
else { ⟨ Package the preamble list, to determine the actual tabskip glue
      amounts, and let p point to this prototype box ⟩;
  ⟨ Set the glue in all the unset boxes of the current list ⟩;
  flush_node_list(p); pop_alignment( );
}
```

After the end of *fin_align* we add the new code to handle extended alignments. We replace the preamble

old

```
⟨ Declare the procedure called align_peek ⟩
```

new

```
⟨ Declare the procedure called align_peek ⟩
  ⟨ Handle an alignment that depends on hsize or vsize ⟩ ≡
  pointer r = get_node(align_node_size);
  save_ptr = save_ptr − 2; pack_begin_line = −mode_line;
  type(r) = whatsit_node; subtype(r) = align_node;
  align_preamble(r) = preamble; align_list(r) = link(head);
  align_extent(r) = hget_xdimen_no(saved(1), saved_hfactor(1),
      saved_vfactor(1)); align_m(r) = saved(0);
  align_v(r) = (mode ≠ −vmode); link(head) = r; tail = r;
  pack_begin_line = 0;
```

After making an unset node with the maximum column width, we can check for columns containing extended boxes

old

```
  } while (¬(q ≡ null))
```

new

```
  if (width(q) > max_dimen) x = true;
  } while (¬(q ≡ null))
```

2.10 Implementing HiTEX Specific Primitives

Now we implement two new TEX primitives: one to include images and one to test whether the current instance of TEX is the new HiTEX. To implement a primitive, we call the *primitive* function. Since it needs the name of the new primitive as a TEX string, we need to add the string to the TEX string pool.

Note that primitives are put into TeX's memory when a TEX format file is read. So adding or changing the code below implies regenerating the format files. To generate a format file run a command like

```
hinitex
**plain \dump
```

and then place the new format file in the TeXformats directory.

After

old

```
  primitive(⟨ "setlanguage" ⟩, extension, set_language_code);
```

we add

new

```
  primitive(⟨ "setlanguage" ⟩, extension, set_language_code);
  primitive(⟨ "image" ⟩, extension, image_node);
  primitive(⟨ "@HINT" ⟩, relax, 256);                    /∗ cf. relax ∗/
```

The code associated with the `image` primitive is defined in the *do_extension* function (see also section 2.4 on page 15).

old

```
case image_node: break;                    /* see section 2.10, page 46 */
```

becomes

new

```
case image_node:
  { pointer p;
    scan_optional_equals( ); scan_file_name( );
    p = new_image_node(cur_name, cur_area, cur_ext);
    loop
    {
      if (scan_keyword(⟨ "width" ⟩)) { scan_normal_dimen;
        image_width(p) = cur_val;
      }
      else if (scan_keyword(⟨ "height" ⟩)) { scan_normal_dimen;
        image_height(p) = cur_val;
      }
      else if (scan_keyword(⟨ "plus" ⟩)) { scan_dimen(false, true, false);
        image_stretch(p) = cur_val; image_stretch_order(p) = cur_order;
      }
      else if (scan_keyword(⟨ "minus" ⟩)) { scan_dimen(false, true, false);
        image_shrink(p) = cur_val; image_shrink_order(p) = cur_order;
      }
      else break;
    }
    hget_image_information(p);
    if (abs(mode) ≡ vmode) append_to_vlist(p);
              /* image nodes have height, width, and depth like boxes */
    else tail_append(p);
  }
```

Now we show how to add new strings numbered 668, 669 after 667 to the string pool. First we define the strings

old

```
#define str_667 "ext4"
⟨ "ext4" ⟩ ≡ 667
```

becomes

new

```
#define str_667 "ext4"
⟨ "ext4" ⟩ +≡ 667
#define str_668 "image"
⟨ "image" ⟩ ≡ 668
#define str_669 "@HINT"
```

⟨ "@HINT" ⟩ ≡ 669

Next we add the strings to the string pool:

old

str_664 str_665 str_666 str_667

becomes

new

str_664 str_665 str_666 str_667 str_668 str_669

Then we add the position to the *str_start* array

old

str_start_668

becomes

new

str_start_668 , *str_start_669* , *str_start_670*

and we define the string start position and adjust the initial values of *str_ptr* and *pool_ptr*.

old

str_start_668 = *str_start_667* + **sizeof** (*str_667*) − 1,
str_start_end
}
str_starts;
⟨ *pool_ptr* initialization ⟩ ≡
str_start_668
⟨ *str_ptr* initialization ⟩ ≡
668

becomes

new

str_start_668 = *str_start_667* + **sizeof** (*str_667*) − 1,
str_start_669 = *str_start_668* + **sizeof** (*str_668*) − 1,
str_start_670 = *str_start_669* + **sizeof** (*str_669*) − 1,
str_start_end
}
str_starts;
⟨ *pool_ptr* initialization ⟩ +≡
str_start_670
⟨ *str_ptr* initialization ⟩ +≡
670

3 Miscellaneous Changes

3.1 Character Set

The TEX processor converts between ASCII code and the user's external character set by means of arrays *xord* and *xchr* that are analogous to Pascal's *ord* and *chr* functions. We change the definitions of *xchr* to the identity mapping like in texlife. Therefore we replace

old

```
for (i = 0; i ≤ °37; i++) xchr[i] = '␣';
for (i = °177; i ≤ °377; i++) xchr[i] = '␣';
```

by

new

```
for (i = 0; i ≤ °37; i++) xchr[i] = i;
for (i = °177; i ≤ °377; i++) xchr[i] = i;
```

3.2 File IO

Just for convenience, I add an *fflush* to be able to see debugging output immediately in the log file. This should be removed in the final version. In line 1472

old

```
#define wlog_cr pascal_write(log_file, "\n")
```

becomes

new

```
#define wlog_cr
  (pascal_write(log_file, "\n"), ((log_file).f ? fflush((log_file).f) : 0))
```

3.3 Compiler Warning

In line 1680, the Microsoft C comiler complains

```
warning C4018:  '>=' :  signed/unsigned mismatch
```

We replace

old

```
loop { while (n ≥ v)
```

by

new

```
loop { while (n ≥ (int) v)
```

In line 2333 we have

```
warning C4146:
unary minus operator applied to unsigned type, result still unsigned
```

old

```
else { xn_over_d = −u; rem = −(v % d);
```

becomes

new

```
else { xn_over_d = −(int) u; rem = −(int)(v % d);
```

3.4 References

In the definition of a **memory_word** we need also an **unsigned int** as union member to store text offsets in a passive node or the image position in a stream.

old

```
int i;
```

becomes

new

```
int32_t i;
uint32_t u;
```

3.5 Debugging

line 2826 check unreasonable *free_node* calls

old

```
node_size(p) = s; link(p) = empty_flag;
```

becomes

new

```
#ifdef MEM_DEBUG
  if (subtype(p) ≡ 12) fprintf(log_file.f,
        "\nfree␣type=%d,␣subtype=%d,␣size=%d␣␣p=0x%04x,\t",
        type(p), subtype(p), s, p);
#endif
  if (s > 11)
    error ( ) ;
  if (p > lo_mem_max)
    error ( ) ;
  node_size(p) = s; link(p) = empty_flag;
```

3.6 Compiler Warning

The *show_node_list* declares a variable that is never used. We delete line 3732.

old

```
double g;                    /* a glue ratio, as a floating point number */
```

3.7 Space Glue

Because HiTeX stores the space glue with every font, we package the code to compute it in a function and make it externaly available.

line 20352

old

```
⟨ Find the glue specification ⟩ ≡
  { main_p = font_glue[cur_font];
    if (main_p ≡ null) { main_p = new_spec(zero_glue);
      main_k = param_base[cur_font] + space_code;
      width(main_p) = font_info[main_k].sc;   /* that's space(cur_font) */
      stretch(main_p) = font_info[main_k + 1].sc;
                                          /* and space_stretch(cur_font) */
      shrink(main_p) = font_info[main_k + 2].sc;
                                          /* and space_shrink(cur_font) */
      font_glue[cur_font] = main_p;
    }
  }
```

and after

new

```
⟨ Find the glue specification ⟩ +≡
  main_p = find_space_glue(cur_font);
  pointer find_space_glue(internal_font_number f)
  { main_p = font_glue[f];
    if (main_p ≡ null) { main_p = new_spec(zero_glue);
      main_k = param_base[f] + space_code;
      width(main_p) = font_info[main_k].sc;          /* that's space(f) */
      stretch(main_p) = font_info[main_k + 1].sc;
                                              /* and space_stretch(f) */
      shrink(main_p) = font_info[main_k + 2].sc;
                                              /* and space_shrink(f) */
      font_glue[f] = main_p;
    }
    return main_p;
  }
```

4 HiTEX

4.1 Images

The handling of images is an integral part of HiTEX. In section 2.10 on page 46 the `image` primitive was defined. It requires a filename for the image and allows the specification, of width, height, stretch and shrink of an image. If either width or height is not given, HiTEX tries to extract this information from the image file itself calling the function *hget_image_information* with the pointer p to the image node as parameter.

⟨ HiTEX routines 2 ⟩ +≡ (18)

```
void hget_image_information(pointer p)
{ char *fn;
  FILE *f;
  if (image_width(p) ≠ 0 ∧ image_height(p) ≠ 0) return;
  fn = dir[image_no(p)].file_name; f = fopen(fn, "rb");
  if (f ≡ NULL) QUIT("Unable␣to␣open␣image␣file␣%s.", fn);
  MESSAGE("(%s", fn); img_buf_size = 0;
  if (¬get_BMP_info(f, fn, p)∧¬get_PNG_info(f, fn, p)∧¬get_JPG_info(f, fn, p))

    QUIT("Unable␣to␣obtain␣width␣and␣height␣infor\
        mation␣for␣image␣%s", fn);
  fclose(f);
  MESSAGE("␣width=␣%fpt␣height=␣%fpt)", image_width(p)/(double)
      ONE, image_height(p)/(double) ONE);
}
```

When we have found the width and height of the stored image, we can supplement the information given in the TEX file. Occasionaly, the image file will not specify the absolute dimensions of the image. In this case we can still compute the aspect ratio and supplement either the width based on the height or vice versa. This is accomplished by calling the *set_image_dimensions* function. It returns true on success and false otherwise.

⟨ HiTEX auxiliar routines 11 ⟩ ≡ (11)

```
static bool set_image_dimensions(pointer p, double w, double h, bool
      absolute)
{
  if (image_width(p) ≠ 0) { double aspect = h/w;
```

```
    image_height(p) = round(image_width(p) * aspect);
  }
  else if (image_height(p) ≠ 0) { double aspect = w/h;
    image_width(p) = round(image_height(p) * aspect);
  }
  else {
    if (¬absolute) return false;
    image_width(p) = round(unity * w); image_height(p) = round(unity * h);
  }
  return true;
}
```

Used in 107.

We call the following routines with the image buffer partly filled with the start of the image file.

```
⟨ HiTEX auxiliar routines 11 ⟩ +≡                                    (12)
#define IMG_BUF_MAX 54
#define IMG_HEAD_MAX 2
  static unsigned char img_buf[IMG_BUF_MAX];
  static int img_buf_size;
#define LittleEndian32(X) (img_buf[(X)] + (img_buf[(X) + 1] ≪
      8) + (img_buf[(X) + 2] ≪ 16) + (img_buf[(X) + 3] ≪ 24))
#define BigEndian16(X) (img_buf[(X) + 1] + (img_buf[(X)] ≪ 8))
#define BigEndian32(X)
  (img_buf[(X) + 3] + (img_buf[(X) + 2] ≪ 8) + (img_buf[(X) + 1] ≪
      16) + (img_buf[(X)] ≪ 24))
#define Match2 (X, A, B) ((img_buf[(X)] ≡ (A)) ∧ (img_buf[(X) + 1] ≡ (B)))
#define Match4 (X, A, B, C, D) (Match2(X, A, B) ∧ Match2((X) + 2, C, D))
#define GET_IMG_BUF(X)
  if (img_buf_size < X) {
    int i = fread(img_buf + img_buf_size, 1, (X) − img_buf_size, f);
    if (i < 0) QUIT("Unable to read image %s", fn);
    else if (i ≡ 0) QUIT("Unable to read image header %s", fn);
    else img_buf_size += i;
  }
```

We start with Windows Bitmaps. A Windows bitmap file usually has the extension `.bmp` but the better way to check for a Windows bitmap file ist to examine the first two byte of the file: the ASCII codes for 'B' and 'M'. Once we have verified the file type, we find the width and height of the bitmap in pixels at offsets `#12` and `#16` stored as little-endian 32 bit integers. At offsets `#26` and `#2A`, we find the horizontal and vertical resolution in pixel per meter stored in the same format. This is sufficient to compute the true width and height of the image in scaled points. If either the width or the height was given in the TEX file, we just compute the aspect ratio and compute the missing value.

The Windows Bitmap format is easy to process but not very efficient. So the support for this format in HiTEX is deprecated and will disappear. You should use one of the formats described next.

⟨ HiTEX auxiliar routines 11 ⟩ +≡ (13)

```
static bool get_BMP_info(FILE *f, char *fn, pointer p)
{ double w, h;
  double xppm, yppm;
  GET_IMG_BUF(2);
  if (¬Match2(0, 'B', 'M')) return false;
  GET_IMG_BUF(#2E); w = (double) LittleEndian32(#12); /* width in pixel */
  h = (double) LittleEndian32(#16);                   /* height in pixel */
  xppm = (double) LittleEndian32(#26);      /* horizontal pixel per meter */
  yppm = (double) LittleEndian32(#2A);      /* vertical pixel per meter */
  return set_image_dimensions(p, (72.27 * 1000.0/25.4) * w/xppm,
      (72.27 * 1000.0/25.4) * h/yppm, true);
}
```

Now we repeat this process for image files in using the Portable Network Graphics[1] file format. This file format is well suited to simple graphics that do not use color gradients. These images usually have the extension `.png` and start with an eight byte signature: #89 followed by the ASCII Codes 'P', 'N', 'G', followd by a carriage return (#0D and line feed (#0A), an DOS end-of-file character (#1A) and final line feed (#0A). After the signature follows a list of chunks. The first chunk is the image header chunk. Each chunk starts with the size of the chunk stored as big-endian 32 bit integer, followed by the chunk name stored as four ASCII codes followed by the chunk data and a CRC. The size as stored in the chunk does not include the size itself, the name or the CRC. The first chunk is the IHDR chunk. The chunk data of the IHDR chunk starts with the width and the height of the image in pixels stored as 32 bit big-endian integers.

Finding the image resolution takes some more effort. The image resolution is stored in an optional chunk named "pHYs" for the physical pixel dimensions. All we know is that this chunk, if it exists, will appear after the IHDR chunk and before the (required) IDAT chunk. The pHYs chunk contains two 32 bit big-endian integers, giving the horizontal and vertical pixels per unit, and a one byte unit specifier, which is either 0 for an undefined unit or 1 for the meter as unit. With an undefined unit only the aspect ratio of the pixels anh hence the aspect ratio of the image can be determined.

⟨ HiTEX auxiliar routines 11 ⟩ +≡ (14)

```
static bool get_PNG_info(FILE *f, char *fn, pointer p)
{ int pos, size;
  double w, h;
  double xppu, yppu;
  int unit;
  GET_IMG_BUF(24);
  if (¬Match4(0, #89, 'P', 'N', 'G') ∨ ¬Match4(4, #0D, #0A, #1A, #0A))
    return false;
  size = BigEndian32(8);
  if (¬Match4(12, 'I', 'H', 'D', 'R')) return false;
```

```
    w = (double) BigEndian32(16); h = (double) BigEndian32(20);
    pos = 20 + size;
    while (true) {
      if (fseek(f, pos, SEEK_SET) ≠ 0) return false;
      img_buf_size = 0; GET_IMG_BUF(17); size = BigEndian32(0);
      if (Match4(4, 'p', 'H', 'Y', 's')) { xppu = (double) BigEndian32(8);
        yppu = (double) BigEndian32(12); unit = img_buf[16];
        if (unit ≡ 0) return set_image_dimensions(p, w/xppu, h/yppu, false);
        else if (unit ≡ 1) return set_image_dimensions(p,
              (72.27/0.0254) * w/xppu, (72.27/0.0254) * h/yppu, true);
        else return false;
      }
      else if (Match4(4, 'I', 'D', 'A', 'T'))
        return set_image_dimensions(p, w, h, false);
      else pos = pos + 12 + size;
    }
    return false;
  }
```

For photographs, the JPEG File Interchange Format (JFIF)[4][5] is more appropriate. JPEG files come with all sorts of file extensions like `.jpg`, `.jpeg`, or `.jfif`. We check the file siganture: it starts with the the SOI (Start of Image) marker #FF, #D8 followed by the JIFI-Tag. The JIFI-Tag starts with the segment marker APP0 (#FF, #E0) followed by the 2 byte segment size, followed by the ASCII codes 'J', 'F', 'I', 'F' followed by a zero byte. Next is a two byte version number which we do not read. Before the resolution proper there is a resolution unit indicator byte (0 = no units, 1 = dots per inch, 2 = dots per cm) and then comes the horizontal and vertical resolution both as 16 Bit big-endian integers. To find the actual width and height, we have to search for a start of frame marker (#FF, #C0+n with $0 \le n \le 15$). Which is followed by the 2 byte segment size, the 1 byte sample precission, the 2 byte height and the 2 byte width.

⟨ HiTEX auxiliar routines 11 ⟩ +≡ (15)

```
  static bool get_JPG_info(FILE *f, char *fn, pointer p)
  { int pos, size;
    double w, h;
    double xppu, yppu;
    int unit;
    GET_IMG_BUF(18);
    if (¬Match4(0, #FF, #D8, #FF, #E0)) return false;
    size = BigEndian16(4);
    if (¬Match4(6, 'J', 'F', 'I', 'F')) return false;
    if (img_buf[10] ≠ 0) return false;
    unit = img_buf[13]; xppu = (double) BigEndian16(14);
    yppu = (double) BigEndian16(16); pos = 4 + size;
    while (true) {
      if (fseek(f, pos, SEEK_SET) ≠ 0) return false;
```

```
    img_buf_size = 0; GET_IMG_BUF(10);
    if (img_buf[0] ≠ #FF) return false;      /* Not the start of a segment */
    if ((img_buf[1] & #F0) ≡ #C0)                    /* Start of Frame */
    { h = (double) BigEndian16(5); w = (double) BigEndian16(7);
      if (unit ≡ 0) return set_image_dimensions(p, w/xppu, h/yppu, false);
      else if (unit ≡ 1)
        return set_image_dimensions(p, 72.27 * w/xppu, 72.27 * h/yppu, true);
      else if (unit ≡ 2) return set_image_dimensions(p,
            (72.27/2.54) * w/xppu, (72.27/2.54) * h/yppu, true);
      else return false;
    }
    else { size = BigEndian16(2); pos = pos + 2 + size;
    }
  }
  return false;
}
```

There is still one image format missing, scalable vector graphics.

4.2 The New Page Builder

.

Here is the new *build_page* routine of HiTEX:

⟨ HiTEX routines 2 ⟩ +≡ (16)

```
void build_page(void)
{ pointer p;
#if 0
  static int cur_template = 0;
#endif
  if ((link(contrib_head) ≡ null) ∨ output_active) return;
  if (page_contents ≡ empty)
                /* accumulate page_content until we determine the defaults */
  { p = link(contrib_head); do {
      if (type(p) ≡ penalty_node ∧ penalty(p) ≤ eject_penalty) {
          /* we have an empty page and an eject penalty. We delete leading
           empty pages */
        pointer q;
        do { q = link(contrib_head); link(contrib_head) = link(q);
          link(q) = null; flush_node_list(q);
        } while (q ≠ p); p = link(contrib_head);
        if (p ≡ null) {
          if (nest_ptr ≡ 0) tail = contrib_head;
          else contrib_tail = contrib_head;
          return;
        }
      }
```

```
        if (type(p) ≡ hlist_node ∨ (type(p) ≡ vlist_node ∧ list_ptr(p) ≠
            null) ∨ type(p) ≡ rule_node ∨ (type(p) ≡ whatsit_node ∧ (subtype(p) ≡
            image_node ∨ subtype(p) ≡ align_node ∨ subtype(p) ≡
            graf_node ∨ subtype(p) ≡ disp_node ∨ subtype(p) ≡
            hset_node ∨ subtype(p) ≡ vset_node ∨ subtype(p) ≡
            hpack_node ∨ subtype(p) ≡ vpack_node)) ∨ type(p) ≡ ins_node
                                        /* not sure if this is needed here */
        ) { freeze_page_specs(box_there); hfix_defaults( ); /* set cur_template ? */
          break;
        }
        p = link(p);
      } while (p ≠ null);
      if (page_contents ≡ empty) return;
    }
#ifdef DEBUG
    if (debugflags & DBGFLEX) { selector = term_and_log; depth_threshold = 30;
      breadth_max = 30; show_node_list(link(contrib_head));
    }
#endif
    page_goal = max_dimen; page_goal = #11110000;
    /* page_goal = °10000000000 − 1; maximum value without arithmetic errors
        */
    if (box(255) ≠ null) { get_page(box(255)); box(255) = null;
    }
    do { p = link(contrib_head); hout_node(p);
                                /* nodes that get deleted at the start of a page */
      link(contrib_head) = link(p); link(p) = null; flush_node_list(p);
    } while (¬(link(contrib_head) ≡ null));
    if (nest_ptr ≡ 0) tail = contrib_head;
    else contrib_tail = contrib_head;
    DBG(DBGBUFFER, "after␣build␣page␣dyn_used=␣%d\n", dyn_used);
  }
```

An important feature of the new routine is the call to *hfix_defaults*. It occurs when the first "visible mark" is placed on the page. At that point we record the current values of TEX's parameters which we will use to generate the definition section of the HINT file. It is still possible to specify alternative values for these paramaters by using parameter lists but only at an additional cost in space and time.

Furthermore, this is the point where we freeze the definition of *hsize* and *vsize*. The current values will be regarded as the sizes as recommended by the author. From then on *hsize* and *vsize* are replaced by the equivalent extended dimensions and any attempt to modify them on the global level will be ignored.

```
⟨ Switch hsize and vsize to extended dimensions 17 ⟩ ≡                    (17)
  hsize = 0; vsize = 0; dimen_par_hfactor(hsize_code) = unity;
  dimen_par_vfactor(vsize_code) = unity;                          Used in 30.
```

4.3 Replacing hpack and vpack

```
⟨ HiTEX routines 2 ⟩ +≡                                                    (18)
  pointer hpack(pointer p, scaled w, scaled hf, scaled vf, small_number
      m){ pointer r;                    /* the box node that will be returned */
    pointer prev_p;                                        /* trails behind p */
    scaled h, d, x;                      /* height, depth, and natural width */
    scaled s;                                              /* shift amount */
    pointer g;                           /* points to a glue specification */
    glue_ord sto, sho;                                     /* order of infinity */
    internal_font_number f;              /* the font in a char_node */
    four_quarters i;                     /* font information about a char_node */
    eight_bits hd;                       /* height and depth indices for a character */
    last_badness = 0; r = get_node(box_node_size); type(r) = hlist_node;
    subtype(r) = min_quarterword; shift_amount(r) = 0;
    prev_p = r + list_offset; link(prev_p) = p; h = 0; d = 0; x = 0;
    total_stretch[normal] = 0; total_shrink[normal] = 0; total_stretch[fil] = 0;
    total_shrink[fil] = 0; total_stretch[fill] = 0; total_shrink[fill] = 0;
    total_stretch[filll] = 0; total_shrink[filll] = 0;
    while (p ≠ null) {
    reswitch:
      while (is_char_node(p)) { f = font(p); i = char_info(f)(character(p));
        hd = height_depth(i); x = x + char_width(f)(i);
        s = char_height(f)(hd);
        if (s > h) h = s;
        s = char_depth(f)(hd);
        if (s > d) d = s;
        p = link(p);
      }
      if (p ≠ null) {
        switch (type(p)) {
        case hlist_node: case vlist_node: case rule_node: case unset_node:
          { x = x + width(p);
            if (type(p) ≥ rule_node) s = 0;
            else s = shift_amount(p);
            if (height(p) − s > h) h = height(p) − s;
            if (depth(p) + s > d) d = depth(p) + s;
          }
          break;
        case ins_node: case mark_node: case adjust_node:
          if (adjust_tail ≠ null) {
            while (link(prev_p) ≠ p) prev_p = link(prev_p);
            if (type(p) ≡ adjust_node) { link(adjust_tail) = adjust_ptr(p);
              while (link(adjust_tail) ≠ null) adjust_tail = link(adjust_tail);
              p = link(p); free_node(link(prev_p), small_node_size);
            }
```

```
        else { link(adjust_tail) = p; adjust_tail = p; p = link(p);
        }
        link(prev_p) = p; p = prev_p;
      }
      break;
    case whatsit_node:
      if (subtype(p) ≡ graf_node) goto repack;
      else if (subtype(p) ≡ disp_node) goto repack;
      else if (subtype(p) ≡ vpack_node) goto repack;
      else if (subtype(p) ≡ hpack_node) goto repack;
      else if (subtype(p) ≡ hset_node) goto repack;
      else if (subtype(p) ≡ vset_node) goto repack;
      else if (subtype(p) ≡ image_node) { glue_ord o;
        if (image_height(p) > h) h = image_height(p);
        x = x + image_width(p); o = image_stretch_order(p);
        total_stretch[o] = total_stretch[o] + image_stretch(p);
        o = image_shrink_order(p);
        total_shrink[o] = total_shrink[o] + image_shrink(p);
      }
      break; break;
    case glue_node:
      { glue_ord o;
        g = glue_ptr(p); x = x + width(g); o = stretch_order(g);
        total_stretch[o] = total_stretch[o] + stretch(g);
        o = shrink_order(g); total_shrink[o] = total_shrink[o] + shrink(g);
        if (subtype(p) ≥ a_leaders) { g = leader_ptr(p);
          if (height(g) > h) h = height(g);
          if (depth(g) > d) d = depth(g);
        }
      }
      break;
    case kern_node: case math_node: x = x + width(p); break;
    case ligature_node:
      { mem[lig_trick] = mem[lig_char(p)]; link(lig_trick) = link(p);
        p = lig_trick; goto reswitch;
      }
    default: do_nothing;
    }
    p = link(p);
  }
}
if (adjust_tail ≠ null) link(adjust_tail) = null;
height(r) = h; depth(r) = d;
if (total_stretch[filll] ≠ 0) sto = filll;
else if (total_stretch[fill] ≠ 0) sto = fill;
```

```
else if (total_stretch[fil] ≠ 0) sto = fil;
else sto = normal;
if (total_shrink[filll] ≠ 0) sho = filll;
else if (total_shrink[fill] ≠ 0) sho = fill;
else if (total_shrink[fil] ≠ 0) sho = fil;
else sho = normal;
if (hf ≠ 0 ∨ vf ≠ 0)                                  /* convert to a hset node */
{ pointer q;
  q = new_set_node( ); subtype(q) = hset_node; height(q) = h;
  depth(q) = d; width(q) = x;                          /* the natural width */
  shift_amount(q) = shift_amount(r); list_ptr(q) = list_ptr(r);
  list_ptr(r) = null; free_node(r, box_node_size);
  if (m ≡ exactly) set_extent(q) = hget_xdimen_no(w, hf, vf);
  else set_extent(q) = hget_xdimen_no(x + w, hf, vf);
  set_stretch_order(q) = sto; set_shrink_order(q) = sho;
  set_stretch(q) = total_stretch[sto]; set_shrink(q) = total_shrink[sho];
  return q;
}
if (m ≡ additional) w = x + w;
width(r) = w; x = w − x;                  /* now x is the excess to be made up */
if (x ≡ 0) { glue_sign(r) = normal; glue_order(r) = normal;
  set_glue_ratio_zero(glue_set(r)); goto end;
}
else if (x > 0) { glue_order(r) = sto; glue_sign(r) = stretching;
  if (total_stretch[sto] ≠ 0)
    glue_set(r) = unfloat(x/(double) total_stretch[sto]);
  else { glue_sign(r) = normal; set_glue_ratio_zero(glue_set(r));
  }
  if (sto ≡ normal) {
    if (list_ptr(r) ≠ null) {
      last_badness = badness(x, total_stretch[normal]);
      if (last_badness > hbadness) { print_ln( );
        if (last_badness > 100) print_nl("Underfull");
        else print_nl("Loose");
        print_str("␣\\hbox␣(badness␣"); print_int(last_badness);
        goto common_ending;
      }
    }
  }
  goto end;
}
else { glue_order(r) = sho; glue_sign(r) = shrinking;
  if (total_shrink[sho] ≠ 0)
    glue_set(r) = unfloat((−x)/(double) total_shrink[sho]);
  else { glue_sign(r) = normal; set_glue_ratio_zero(glue_set(r));
```

```
    }
    if ((total_shrink[sho] < −x) ∧ (sho ≡ normal) ∧ (list_ptr(r) ≠ null)) {
      last_badness = 1000000; set_glue_ratio_one(glue_set(r));
      if ((−x − total_shrink[normal] > hfuzz) ∨ (hbadness < 100)) {
        if ((overfull_rule > 0) ∧ (−x − total_shrink[normal] > hfuzz)) {
          while (link(prev_p) ≠ null) prev_p = link(prev_p);
          link(prev_p) = new_rule( ); width(link(prev_p)) = overfull_rule;
        }
        print_ln( ); print_nl("Overfull␣\\hbox␣("); print_scaled(−x − total_shrink[normal]); print_str("pt␣too␣wide");
        goto common_ending;
      }
    }
    else if (sho ≡ normal) {
      if (list_ptr(r) ≠ null) {
        last_badness = badness(−x, total_shrink[normal]);
        if (last_badness > hbadness) { print_ln( );
          print_nl("Tight␣\\hbox␣(badness␣"); print_int(last_badness);
          goto common_ending;
        }
      }
    }
    goto end;
  }
  common_ending:
  if (output_active)
    print_str(")␣has␣occurred␣while␣\\output␣is␣active");
  else {
  if (pack_begin_line ≠ 0) {
    if (pack_begin_line > 0) print_str(")␣in␣paragraph␣at␣lines␣");
    else print_str(")␣in␣alignment␣at␣lines␣");
    print_int(abs(pack_begin_line)); print_str("--");
  }
  else print_str(")␣detected␣at␣line␣");
  print_int ( line ) ;
  }
  print_ln( ); font_in_short_display = null_font; short_display(list_ptr(r));
  print_ln( ); begin_diagnostic( ); show_box(r); end_diagnostic(true);
end: return r;
repack:
  {                                        /* convert the box to a hpack_node */
    pointer q;
    q = new_pack_node( ); height(q) = h; depth(q) = d; width(q) = x;
    subtype(q) = hpack_node; list_ptr(q) = list_ptr(r); list_ptr(r) = null;
```

```
    free_node(r, box_node_size); pack_limit(q) = max_dimen;
                                                    /* no limit, not used */
    pack_m(q) = m; pack_extent(q) = hget_xdimen_no(w, hf, vf); return q;
  }
  }
```

⟨ HiTEX routines 2 ⟩ +≡ (19)

```
  pointer vpackage(pointer p, scaled h, scaled hf, scaled vf, small_number
      m, scaled l){ pointer r;      /* the box node that will be returned */
    scaled w, d, x;                  /* width, depth, and natural height */
    scaled s = 0;                                    /* shift amount */
    pointer g;                          /* points to a glue specification */
    glue_ord sho, sto;                              /* order of infinity */
#if 0
    bool height_unknown, width_unknown;
    height_unknown = width_unknown = false;
#endif
    last_badness = 0; r = get_node(box_node_size); type(r) = vlist_node;
    subtype(r) = min_quarterword; shift_amount(r) = 0; list_ptr(r) = p;
    w = 0; d = 0; x = 0; total_stretch[normal] = 0; total_shrink[normal] = 0;
    total_stretch[fil] = 0; total_shrink[fil] = 0; total_stretch[fill] = 0;
    total_shrink[fill] = 0; total_stretch[filll] = 0; total_shrink[filll] = 0;
    while (p ≠ null) {
      if (is_char_node(p)) confusion(519);
      else
        switch (type(p)) {
        case hlist_node: case vlist_node: case rule_node: case unset_node:
          x = x + d + height(p); d = depth(p);
          if (type(p) ≥ rule_node) s = 0;
          else s = shift_amount(p);
          if (width(p) + s > w) w = width(p) + s;
          break;
        case whatsit_node:
          if (subtype(p) ≡ graf_node) goto repack;
          else if (subtype(p) ≡ disp_node) goto repack;
          else if (subtype(p) ≡ vpack_node) goto repack;
          else if (subtype(p) ≡ hpack_node) goto repack;
          else if (subtype(p) ≡ hset_node) goto repack;
          else if (subtype(p) ≡ vset_node) goto repack;
          else if (subtype(p) ≡ image_node) { glue_ord o;
            if (image_width(p) > w) w = image_width(p);
            x = x + d + image_height(p); d = 0; o = image_stretch_order(p);
            total_stretch[o] = total_stretch[o] + image_stretch(p);
            o = image_shrink_order(p);
            total_shrink[o] = total_shrink[o] + image_shrink(p);
```

```
        }
        break;
      case glue_node:
        { glue_ord o;
          x = x + d; d = 0; g = glue_ptr(p); x = x + width(g);
          o = stretch_order(g);
          total_stretch[o] = total_stretch[o] + stretch(g);
          o = shrink_order(g); total_shrink[o] = total_shrink[o] + shrink(g);
          if (subtype(p) ≥ a_leaders) { g = leader_ptr(p);
            if (width(g) > w) w = width(g);
          }
        }
        break;
      case kern_node: x = x + d + width(p); d = 0; break;
      default: do_nothing;
      }
    p = link(p);
  }
  width(r) = w;
  if (total_stretch[filll] ≠ 0) sto = filll;
  else if (total_stretch[fill] ≠ 0) sto = fill;
  else if (total_stretch[fil] ≠ 0) sto = fil;
  else sto = normal;
  if (total_shrink[filll] ≠ 0) sho = filll;
  else if (total_shrink[fill] ≠ 0) sho = fill;
  else if (total_shrink[fil] ≠ 0) sho = fil;
  else sho = normal;
  if (hf ≠ 0 ∨ vf ≠ 0)                          /* convert to a vset node */
  { pointer q;
    q = new_set_node( ); subtype(q) = vset_node; width(q) = w;
    if (d > l) { x = x + d − l; depth(r) = l;
    }
    else depth(r) = d;
    height(q) = x; depth(q) = d; shift_amount(q) = shift_amount(r);
    list_ptr(q) = list_ptr(r); list_ptr(r) = null; free_node(r, box_node_size);
    if (m ≡ exactly) set_extent(q) = hget_xdimen_no(h, hf, vf);
    else set_extent(q) = hget_xdimen_no(x + h, hf, vf);
    set_stretch_order(q) = sto; set_shrink_order(q) = sho;
    set_stretch(q) = total_stretch[sto]; set_shrink(q) = total_shrink[sho];
    return q;
  }
  if (d > l) { x = x + d − l; depth(r) = l;
  }
  else depth(r) = d;
  if (m ≡ additional) h = x + h;
```

```
height(r) = h; x = h − x;                    /* now x is the excess to be made up */
if (x ≡ 0) { glue_sign(r) = normal; glue_order(r) = normal;
  set_glue_ratio_zero(glue_set(r)); goto end;
}
else if (x > 0) { glue_order(r) = sto; glue_sign(r) = stretching;
  if (total_stretch[sto] ≠ 0)
    glue_set(r) = unfloat(x/(double) total_stretch[sto]);
  else { glue_sign(r) = normal; set_glue_ratio_zero(glue_set(r));
  }
  if (sto ≡ normal) {
    if (list_ptr(r) ≠ null) {
      last_badness = badness(x, total_stretch[normal]);
      if (last_badness > vbadness) { print_ln( );
        if (last_badness > 100) print_nl("Underfull");
        else print_nl("Loose");
        print_str("␣\\vbox␣(badness␣"); print_int(last_badness);
        goto common_ending;
      }
    }
  }
  goto end;
}
else                                                    /* if (x¡0) */
{ glue_order(r) = sho; glue_sign(r) = shrinking;
  if (total_shrink[sho] ≠ 0)
    glue_set(r) = unfloat((−x)/(double) total_shrink[sho]);
  else { glue_sign(r) = normal; set_glue_ratio_zero(glue_set(r));
  }
  if ((total_shrink[sho] < −x) ∧ (sho ≡ normal) ∧ (list_ptr(r) ≠ null)) {
    last_badness = 1000000; set_glue_ratio_one(glue_set(r));
    if ((−x − total_shrink[normal] > vfuzz) ∨ (vbadness < 100)) {
      print_ln( ); print_nl("Overfull␣\\vbox␣(");
      print_scaled(−x − total_shrink[normal]); print_str("pt␣too␣high");
      goto common_ending;
    }
  }
  else if (sho ≡ normal) {
    if (list_ptr(r) ≠ null) {
      last_badness = badness(−x, total_shrink[normal]);
      if (last_badness > vbadness) { print_ln( );
        print_nl("Tight␣\\vbox␣(badness␣"); print_int(last_badness);
        goto common_ending;
      }
    }
  }
  goto end;
```

```
    }
    common_ending:
    if (output_active)
      print_str(") has occurred while \\output is active");
    else {
    if (pack_begin_line ≠ 0) { print_str(") in alignment at lines ");
      print_int(abs(pack_begin_line)); print_str("--");
    }
    else print_str(") detected at line ");
    print_int ( line ) ; print_ln( );
    }
    begin_diagnostic( ); show_box(r); end_diagnostic(true);
  end: return r;
  repack:
    { /* convert the box to a vpack_node */
      pointer q;
      q = new_pack_node( ); subtype(q) = vpack_node; height(q) = x;
      depth(q) = d; width(q) = w; list_ptr(q) = list_ptr(r); list_ptr(r) = null;
      free_node(r, box_node_size); pack_limit(q) = l; pack_m(q) = m;
      pack_extent(q) = hget_xdimen_no(h, hf , vf ); return q;
    }
    }
```

4.4 Streams

HINT numbers inserts starting at 0 for the main text and continues upwards.

⟨ HiTEX routines 2 ⟩ +≡ (20)

```
  typedef struct {
    int f;     /* magnification factor: inserting a box of height h will contribute
        h*f/1000 to the main page */
    int e; /* maximum extent: this extent gives the maximum size per page for
        this insertion */
    pointer b; /* before: material that is inserted before the inserted material */
    pointer a;  /* after: material that is inserted after the inserted material */
    int p; /* prefer: if nonnegative, move the insert to this stream if possible */
    int n; /* next: if nonnegative, move the insert to this stream if it can not be
        accomodated otherwise */
    int r;              /* split ratio: if positive split the final contribution of this
        streams between p and n in the ratio r/1000 for p and 1-r/1000 for n
        before contributing p and r to the page*/
    bool u;                          /* used: indicates that this stream is used */
    pointer g;                                  /* deprecated top skip glue */
    pointer h; /* derived value: the natural height + depth, shrink and stretch
        of list a and b */
  } stream_def;
```

```
int inserts_defined[#100];   /* maps TEX insert numbers to HINT numbers */
int streams_used = 1;                          /* stream 0 is the main text */
stream_def streams[#100] = {{0}};
                                  /* insert definitions indexed by HINT number */
int get_stream(int n)   /* given the subtype of an insert node return a stream
      number for it allocate the stream if needed */
{ pointer p, q;
  int k = inserts_defined[n];
  if (k ≥ 0) return k;
  k = streams_used; streams_used++;
          /* no overflow test necessary since TeX knowns only 255 inserts */
  inserts_defined[n] = k; streams[k].f = count(n);
  streams[k].e = hget_xdimen_no(dimen(n), dimen_hfactor(n),
      dimen_vfactor(n)); p = box(n); box(n) = null;
  if (p ≡ null) streams[k].b = streams[k].a = null;
  else { q = list_ptr(p); free_node(p, box_node_size);
                                  /* first level of boxing needed for box register */
    p = q;
    if (p ≡ null) streams[k].b = streams[k].a = null;
    else { streams[k].b = p; p = link(p); streams[k].a = p;
      if (p ≠ null) { p = link(p);
                          /* now p points to the unused remainder of the list */
        if (p ≠ null) flush_node_list(p);
      }
      if (streams[k].b ≠ null) { p = streams[k].b; link(p) = null;
        if (type(p) ≡ hlist_node ∨ type(p) ≡ vlist_node)
                    /* second level of boxing needed for before and after list */
        {           /* remove one level of boxing to allow merging the glue */
          q = list_ptr(p); free_node(p, box_node_size); streams[k].b = q;
        }
      }
      if (streams[k].a ≠ null) { p = streams[k].a; link(p) = null;
        if (type(p) ≡ hlist_node ∨ type(p) ≡ vlist_node)
                    /* second level of boxing needed for before and after list */
        {           /* remove one level of boxing to allow merging the glue */
          q = list_ptr(p); free_node(p, box_node_size); streams[k].a = q;
        }
      }
    }
  }
  streams[k].g = zero_glue; add_glue_ref(zero_glue); streams[k].p = −1;
  streams[k].n = −1; streams[k].r = −1; streams[k].u = true; return k;
}
```

4.5 Pages

⟨ HiTEX routines 2 ⟩ +≡ (21)

```
pointer pages_defined[#100] = {null};
int pages_used = 1;                /* the shipout box 255 page is always defined */
int get_page(pointer p)
        /* pointer p should be a vbox that is used as a template for pages */
{ int k;
  k = pages_used;
  if (p ≡ null) QUIT("Page template expected");
  if (type(p) ≠ vlist_node ∧ (type(p) ≠ whatsit_node ∨ (subtype(p) ≠
        vset_node ∧ subtype(p) ≠ vpack_node)))
    QUIT("Page template must be a vlist");
  if (pages_used ≥ #100)
    QUIT("Too many page templates %d", pages_used + 1);
  else pages_used++;
  pages_defined[k] = p; return k;
}
static pointer hsimple_page(void)
{ pointer p, q;
  p = new_null_box( ); type(p) = vlist_node;
  list_ptr(p) = q = get_node(ins_node_size); type(q) = ins_node;
  subtype(q) = 0; /* insertion nodes with subtype 255 are forbidden, we use 0
      for the main page */
  return p;
}
```

5 HINT Output

5.1 Initialization

```
⟨ HiTEX routines 2 ⟩ +≡                                                    (22)
  static void hout_init(void)
  { new_directory(dir_entries); new_output_buffers(); max_section_no = 2;
    hdef_init(); hput_content_start();
  }
  void hint_open(void)
  { char ext[] = ".hnt";
    str_number s;
    int k, j;
    j = (int) strlen(ext); str_room(j);
    for (k = 0; k < j; k++) append_char(xord[(int) ext[k]]);
    s = make_string(); pack_job_name(s);
    hout = fopen((const char *) name_of_file + 1, "wb");
    if (hout ≡ NULL) {
      fprintf(stderr, "Unable to open %s\n", name_of_file + 1); exit(1);
    }
    hout_init(); option_global = true; hlog = stderr;
  }
```

5.2 Termination

```
⟨ HiTEX routines 2 ⟩ +≡                                                    (23)
  static void hout_terminate(void)
  { hput_content_end(); hput_definitions(); hput_directory();
    hput_hint("created by HiTeX");
  }
  void hint_close(void)
  { hout_terminate();
    if (hout ≠ NULL) fclose(hout);
    hout = NULL;
  }
```

5.3 HINT Directory

There is not much to do here: some code to find a new or existing directory entry, a variable to hold the number of directorie entries allocated, a function to allocate a new file section, and an auxiliar function to convert TEX's file names to ordinary C strings.

⟨ Find an existing directory entry 24 ⟩ ≡ (24)
```
for (i = 3; i ≤ max_section_no; i++)
  if (dir[i].file_name ≠ NULL ∧ strcmp(dir[i].file_name, file_name) ≡ 0)
    return i;
```
Used in 28.

⟨ Allocate a new directory entry 25 ⟩ ≡ (25)
```
i = max_section_no; i++;
if (i > #FFFF) QUIT("Too many file sections");
if (i ≥ dir_entries) RESIZE(dir, dir_entries, entry_t);
max_section_no = i;
if (max_section_no > #FFFF) QUIT("Too many sections");
dir[i].section_no = i;
```
Used in 28.

⟨ HiTEX macros 26 ⟩ ≡ (26)
```
#define RESIZE(P, S, T)
  { int _n = (S) * 1.4142136 + 0.5;
    if (_n < 32) _n = 32;
    { REALLOCATE(P, _n, T); memset((P) + (S), 0, (_n − (S)) * sizeof (T));
      (S) = _n;
    }
  }
```
Used in 107.

⟨ HiTEX variables 27 ⟩ ≡ (27)
```
static int dir_entries = 4;
```
Used in 107.

⟨ HiTEX auxiliar routines 11 ⟩ +≡ (28)
```
static uint16_t hnew_file_section(char *file_name)
{ uint16_t i;
  ⟨ Find an existing directory entry 24 ⟩
  ⟨ Allocate a new directory entry 25 ⟩
  dir[i].file_name = strdup(file_name); return i;
}
static char *hfile_name(str_numbern, str_numbera, str_numbere)
                                  /* convert a TeX filename to a C filename */
{ pack_file_name(n, a, e); return (char *) name_of_file + 1;
}
```

6 HINT Definitions

Definitions are used for two reasons: they provide default values for the parameters that drive TEX's algorithms running in the HINT viewer, and they provide a compact notation for HINT content nodes.

To find the optimal coding for a HINT file, a global knowledge of the HINT file is necessary. This would require a two pass process: in the first pass HiTEX could gather statistics on the use of parameter values and content nodes as a basis for making definitions and in the second pass it could encode the content using these definitions. I consider it, however, more reasonable to write such a two pass optimizer as a separate program which can be reused on any HINT file. Hence HiTEX uses a much simpler one pass approach:

- HiTEX generates definitions for TEX-parameters using the values they have when the first non discardable item appears in *build_page*. This is usually the case after initial style files have been processed and we can expect that they set useful default values.

  The procedure that generates these definitions is called *hfix_defaults*:

```
⟨ HiTEX routines 2 ⟩ +≡                                              (29)
  void hfix_defaults(void)
  { int i;
    DBG(DBGDEF, "Freezing␣HINT␣file␣defaults\n");
    ⟨ Fix definitions for integer parameters 34 ⟩
    ⟨ Fix definitions for dimension parameters 40 ⟩
    ⟨ Fix definitions for glue parameters 50 ⟩
  }
```

- HiTEX generates definitions to be used in content nodes on the fly: Whenever a routine outputs an item for which a definition might be avalable, it calls a *hget_..._no* function. This function returns, if possible, the reference number of a suitable definition. If no definition is available, the function will try to allocate a new one, only if all reference numbers from 0 to #FF are already in use, a −1 is returned to indicate failure.

  There are two possible problems with this approach: We might miss a very common item because it occurs for the first time late in the input when all reference numbers are already in use. For example an index might repeat a certain pattern for each entry. And second, we might make a definition for an

item that occurs only once. Taken together the definition plus the reference to it requires more space than the same item without a definition.

We can hope that the first effect does not occur to often, especially if the TEX file is short, and we know that the second effect is limited by the total number of definitions we can make times four byte of overhead per instance.

Here we initialize the necessary data structures for definitions.

```
⟨ HiTEX routines 2 ⟩ +≡                                                (30)
  static void hdef_init(void)
  { int i;
    ⟨ Switch hsize and vsize to extended dimensions 17 ⟩
    ⟨ Initialize definitions for extended dimensions 44 ⟩
    ⟨ Initialize definitions for baseline skips 56 ⟩
    ⟨ Initialize definitions for fonts 70 ⟩
#if 0
    overfull_rule = 0;                          /* no overfull rules please */
#endif
    for (i = 0; i < #100; i++) inserts_defined[i] = −1;
    inserts_defined[0] = 0;  /* TeX's insert0 maps to the main page stream 0 */
    streams[0].f = 1000; streams[0].e = vsize_xdimen_no;          /* vsize */
    streams[0].g = top_skip; add_glue_ref(top_skip); streams[0].u = true;
    streams[0].p = −1; streams[0].n = −1; streams[0].r = −1;
    streams_used = 1; pages_defined[0] = hsimple_page(); pages_used = 1;
  }
```

After all definitions are ready, we write them using the function *hput_definitions*.

```
⟨ HiTEX routines 2 ⟩ +≡                                                (31)
  static void hput_definitions()
                          /* write the definitions into the definitions buffer */
  { int i;
    hput_definitions_start(); hput_max_definitions();
    ⟨ Output integer definitions 37 ⟩
    ⟨ Output dimension definitions 42 ⟩
    ⟨ Output extended dimension definitions 46 ⟩
    ⟨ Output glue definitions 53 ⟩
    ⟨ Output baseline skip definitions 59 ⟩
    ⟨ Output hyphen definitions 64 ⟩
    ⟨ Output parameter list definitions 68 ⟩
    ⟨ Output font definitions 73 ⟩
#if 0
    hprint_streams(); hprint_pages(); hprint_files();
#endif
    hput_definitions_end(); hput_range_defs();
                          /* expects the definitions section to be ended */
  }
```

In the following, we present for each node type the code to generate the definitions, using a common schema: We define a data structure called *..._defined*, to hold the definitions; we define, if applicable, the TEX-parameters; we add an *hget_..._no* function to allocate new definitions; and we finish with the code to output the collected definitions.

Lets start with the most simple case: integers.

6.1 Integers

6.1.1 Data

The data structure to hold the integer definitions is a simple array with $^{\#}100$ entries. A more complex data structure, for example a hash table, could speed up searching for existing definitions but lets keep things simple for now.

```
⟨ HiTEX variables 27 ⟩ +≡                                                    (32)
  static int32_t int_defined[#100] = {0};
```

6.1.2 Mapping

Before we can generate definitions for TEX-parameters, we have to map TEX's parameter numbers to HINT definition numbers. While it seems more convenient here to have the reverse mapping, we need the mapping only once to record parameter definitions, but we will need it repeatedly in the function *hdef_param_node* and the overhead here does not warant having the mapping in both directions.

```
⟨ HiTEX variables 27 ⟩ +≡                                                    (33)
  static const int hmap_int[] = {
  pretolerance_no,                                  /* pretolerance_code 0 */
  tolerance_no,                                        /* tolerance_code 1 */
  line_penalty_no,                                   /* line_penalty_code 2 */
  hyphen_penalty_no,                               /* hyphen_penalty_code 3 */
  ex_hyphen_penalty_no,                         /* ex_hyphen_penalty_code 4 */
  club_penalty_no,                                   /* club_penalty_code 5 */
  widow_penalty_no,                                 /* widow_penalty_code 6 */
  display_widow_penalty_no,                 /* display_widow_penalty_code 7 */
  broken_penalty_no,                               /* broken_penalty_code 8 */
  −1,                                                /* bin_op_penalty_code 9 */
  −1,                                                 /* rel_penalty_code 10 */
  pre_display_penalty_no,                     /* pre_display_penalty_code 11 */
  post_display_penalty_no,                   /* post_display_penalty_code 12 */
  inter_line_penalty_no,                       /* inter_line_penalty_code 13 */
  double_hyphen_demerits_no,           /* double_hyphen_demerits_code 14 */
  final_hyphen_demerits_no,             /* final_hyphen_demerits_code 15 */
  adj_demerits_no,                                 /* adj_demerits_code 16 */
  −1,                                                       /* mag_code 17 */
  −1,                                        /* delimiter_factor_code 18 */
  looseness_no,                                       /* looseness_code 19 */
  time_no,                                                 /* time_code 20 */
```

```
  day_no,                                          /* day_code 21 */
  month_no,                                        /* month_code 22 */
  year_no,                                         /* year_code 23 */
  -1,                                              /* show_box_breadth_code 24 */
  -1,                                              /* show_box_depth_code 25 */
  -1,                                              /* hbadness_code 26 */
  -1,                                              /* vbadness_code 27 */
  -1,                                              /* pausing_code 28 */
  -1,                                              /* tracing_online_code 29 */
  -1,                                              /* tracing_macros_code 30 */
  -1,                                              /* tracing_stats_code 31 */
  -1,                                              /* tracing_paragraphs_code 32 */
  -1,                                              /* tracing_pages_code 33 */
  -1,                                              /* tracing_output_code 34 */
  -1,                                              /* tracing_lost_chars_code 35 */
  -1,                                              /* tracing_commands_code 36 */
  -1,                                              /* tracing_restores_code 37 */
  -1,                                              /* uc_hyph_code 38 */
  -1,                                              /* output_penalty_code 39 */
  -1,                                              /* max_dead_cycles_code 40 */
  hang_after_no                                    /* hang_after_code 41 */
  };
```

6.1.3 Parameters

Now we can generate the definitions for integer parameters:

```
⟨ Fix definitions for integer parameters 34 ⟩ ≡                    (34)
  int_defined[zero_int_no] = 0;
  for (i = pretolerance_code; i ≤ hang_after_code; i++)
    if (hmap_int[i] ≥ 0) int_defined[hmap_int[i]] = int_par(i);
  max_ref[int_kind] = MAX_INT_DEFAULT;                    Used in 29.
```

6.1.4 Allocation

The function *hget_int_no* tries to allocate a predefined integer number; if not successful, it returns −1.

```
⟨ HiTEX routines 2 ⟩ +≡                                            (35)
  int hget_int_no(int32_t n)
  { int i;
    int m = max_ref[int_kind];
    for (i = 0; i ≤ m; i++)
      if (n ≡ int_defined[i]) return i;
    if (m < #FF) { m = ++max_ref[int_kind]; int_defined[m] = n; return m;
    }
    else return -1;
  }
```

6.1.5 Output

Before we give the code to output an integer definition, we declare a macro that is usefull for all the definitions. `HPUTDEF` takes a function F and a reference number R. It is assumed that F writes a definition into the output and returns a tag. The macro will then add the reference number and both tags to the output.

⟨ HiTEX macros 26 ⟩ +≡ (36)

```
#define HPUTDEF (F, R)
  { uint32_t _p;
    uint8_t _t;
    HPUTNODE;                                              /* allocate */
    _p = hpos − hstart; HPUT8(0);                              /* tag */
    HPUT8(R);                                              /* reference */
    _t = F; hstart[_p] = _t; DBGTAG(_t, hstart + _p); DBGTAG(_t, hpos); HPUT8(_t);
  }
```

Definitions are written to the output only if they differ from HiTEX's built in defaults.

⟨ Output integer definitions 37 ⟩ ≡ (37)

```
  DBG(DBGDEF, "Maximum␣int␣reference:␣%d\n", max_ref[int_kind]);
  for (i = max_fixed[int_kind] + 1; i ≤ max_default[int_kind]; i++) {
    if (int_defined[i] ≠ int_defaults[i])
      HPUTDEF(hput_int(int_defined[i]), i);
  }
  for ( ; i ≤ max_ref[int_kind]; i++)
    HPUTDEF(hput_int(int_defined[i]), i);
```

Used in 31.

6.2 Dimensions

We proceed as we did for integers, starting with the array that holds the defined dimensions.

6.2.1 Data

⟨ HiTEX variables 27 ⟩ +≡ (38)

```
  static scaled dimen_defined[#100] = {0};
```

6.2.2 Mapping

⟨ HiTEX variables 27 ⟩ +≡ (39)

```
  static const int hmap_dimen[ ] = {
  −1,                                          /* par_indent_code 0 */
  −1,                                       /* math_surround_code 1 */
  line_skip_limit_no,                        /* line_skip_limit_code 2 */
  hsize_dimen_no,                                  /* hsize_code 3 */
  vsize_dimen_no,                                  /* vsize_code 4 */
  max_depth_no,                               /* max_depth_code 5 */
  −1,                                      /* split_max_depth_code 6 */
  −1,                                       /* box_max_depth_code 7 */
```

```
  −1,                                          /* hfuzz_code 8 */
  −1,                                          /* vfuzz_code 9 */
  −1,                                  /* delimiter_shortfall_code 10 */
  −1,                                /* null_delimiter_space_code 11 */
  −1,                                     /* script_space_code 12 */
  −1,                                  /* pre_display_size_code 13 */
  −1,                                    /* display_width_code 14 */
  −1,                                   /* display_indent_code 15 */
  −1,                                     /* overfull_rule_code 16 */
  hang_indent_no,                         /* hang_indent_code 17 */
  −1,                                          /* h_offset_code 18 */
  −1,                                          /* v_offset_code 19 */
  emergency_stretch_no                 /* emergency_stretch_code 20 */
  };
```

6.2.3 Parameters

⟨ Fix definitions for dimension parameters 40 ⟩ ≡ (40)

```
  dimen_defined[zero_dimen_no] = 0;
  for (i = par_indent_code; i ≤ emergency_stretch_code; i++)
    if (hmap_dimen[i] ≥ 0) dimen_defined[hmap_dimen[i]] = dimen_par(i);
  dimen_defined[hsize_dimen_no] = hhsize;
  dimen_defined[vsize_dimen_no] = hvsize;
  dimen_defined[quad_no] = quad(cur_font);
  dimen_defined[math_quad_no] = math_quad(text_size);
  max_ref[dimen_kind] = MAX_DIMEN_DEFAULT;
```

Used in 29.

6.2.4 Allocation

⟨ HiTEX routines 2 ⟩ +≡ (41)

```
  int hget_dimen_no(scaled s)          /* tries to allocate a predefined dimension
        number in the range 0 to 0xFF if not successful return -1 */
  { int i;
    int m = max_ref[dimen_kind];
    for (i = 0; i ≤ m; i++)
      if (s ≡ dimen_defined[i]) return i;
    if (m < #FF) { m = ++max_ref[dimen_kind]; dimen_defined[m] = s;
      return m;
    }
    else return −1;
  }
```

6.2.5 Output

⟨ Output dimension definitions 42 ⟩ ≡ (42)
```
DBG(DBGDEF, "Maximum␣dimen␣reference:␣%d\n", max_ref[dimen_kind]);
for (i = max_fixed[dimen_kind] + 1; i ≤ max_default[dimen_kind]; i++) {
  if (dimen_defined[i] ≠ dimen_defaults[i])
    HPUTDEF(hput_dimen(dimen_defined[i]), i);
}
for ( ; i ≤ max_ref[dimen_kind]; i++)
  HPUTDEF(hput_dimen(dimen_defined[i]), i);
```
Used in 31.

6.3 Extended Dimensions

Since HiTEX's nodes never store an extended dimension directly but always use a preference number. We can not allocat a fixed array with 256 entries to store the extended dimensions. Instead we need a variable size array that grows as needed to allocate all extended dimensions needed.

6.3.1 Data

⟨ HiTEX variables 27 ⟩ +≡ (43)
```
static xdimen_t*xdimen_defined = NULL;
static int xdimen_used = 0, xdimen_allocated = 0;
```

6.3.2 Initialization

⟨ Initialize definitions for extended dimensions 44 ⟩ ≡ (44)
```
xdimen_allocated = max_fixed[xdimen_kind] + 1;
ALLOCATE(xdimen_defined, xdimen_allocated, xdimen_t);
DBG(DBGDEF, "Maximum␣xdimen␣reference:␣%d\n", max_ref[xdimen_kind]);
for (i = 0; i ≤ max_fixed[xdimen_kind]; i++)
  xdimen_defined[i] = xdimen_defaults[i];
xdimen_used = max_fixed[xdimen_kind] + 1;
```
Used in 30.

6.3.3 Allocation

To obtain a reference number for an extended dimension, we search the array and if no match was found, we allocate a new entry, reallocating the array if needed. We use the variable *rover* to mark the place where the last entry was inserted, because quite often we repeatedly search for the same values.

⟨ HiTEX routines 2 ⟩ +≡ (45)
```
int hget_xdimen_no(dimen_t w, scaled h, scaled v)
{ static int rover = 0;
  int i;
  float32_t fh, fv;
  fh = h/(double) ONE; fv = v/(double) ONE;
  for (i = 0; i < xdimen_used; i++)          /* search for an existing spec */
  { xdimen_t * q = &(xdimen_defined[rover]);
    if (w ≡ q→w ∧ fh ≡ q→h ∧ fv ≡ q→v) return rover;
```

```
    else if (rover ≡ 0) rover = xdimen_used − 1;
    else rover−−;
  }
  if (xdimen_used ≥ xdimen_allocated)
    RESIZE(xdimen_defined, xdimen_allocated, xdimen_t);
  rover = xdimen_used++;
  if (rover < #100) max_ref[xdimen_kind] = rover;
  xdimen_defined[rover].w = w; xdimen_defined[rover].h = fh;
  xdimen_defined[rover].v = fv; return rover;
}
```

6.3.4 Output

⟨ Output extended dimension definitions 46 ⟩ ≡ (46)

```
DBG(DBGDEF, "Maximum␣xdimen␣reference:␣%d\n", max_ref[xdimen_kind]);
for (i = max_fixed[xdimen_kind] + 1; i ≤ max_default[xdimen_kind]; i++) {
  if (xdimen_defined[i].w ≠ xdimen_defaults[i].w ∨ xdimen_defined[i].h ≠
      xdimen_defaults[i].h ∨ xdimen_defined[i].v ≠ xdimen_defaults[i].v)
    HPUTDEF(hput_xdimen(&xdimen_defined[i]), i);
}
for ( ; i ≤ max_ref[xdimen_kind]; i++)
  HPUTDEF(hput_xdimen(&xdimen_defined[i]), i);
```

Used in 31.

6.3.5 Printing

The following function is needed in HiTEX to produce debugging output if needed.

⟨ HiTEX routines 2 ⟩ +≡ (47)

```
void print_xdimen(int i)
{
  if (0 ≤ i ∧ i < xdimen_used) { xdimen_t * e = &xdimen_defined[i];
    print_scaled(e→w);
    if (e→h ≠ 0.0) { print_char('+'); print_scaled(ROUND(e→h * ONE));
      print_str("*hsize");
    }
    if (e→v ≠ 0.0) { print_char('+'); print_scaled(ROUND(e→v * ONE));
      print_str("*vsize");
    }
  }
  else print_str("unknown");
}
```

6.4 Glues

In general there are two choices on how to store a definition: We can use the data structures used by TEX or we can use the data structures defined by HINT. If we are lucky, both of them are the same as we have seen for integers and dimensions. For extended dimensions, we had to use the HINT data type *xdimen_t* because TEX has no corresponding data type and uses only reference numbers. In the case of glue, we definitely have a choice. We decide to use TEX's pointers to glue specifications in the hope to save some work when comparing glues for equality, because TEX already reuses glue specifications and often a simple comparison of pointers might suffice.

6.4.1 Data

```
⟨ HiTEX variables 27 ⟩ +≡                                                (48)
  static pointer glue_defined[#100];
```

6.4.2 Mapping

```
⟨ HiTEX variables 27 ⟩ +≡                                                (49)
  static int hmap_glue[] = {line_skip_no,                    /* line_skip_code 0 */
  baseline_skip_no,                                          /* baseline_skip_code 1 */
  −1,                                                        /* par_skip_code 2 */
  above_display_skip_no,                                     /* above_display_skip_code 3 */
  below_display_skip_no,                                     /* below_display_skip_code 4 */
  above_display_short_skip_no,                               /* above_display_short_skip_code 5 */
  below_display_short_skip_no,                               /* below_display_short_skip_code 6 */
  left_skip_no,                                              /* left_skip_code 7 */
  right_skip_no,                                             /* right_skip_code 8 */
  top_skip_no,                                               /* top_skip_code 9 */
  split_top_skip_no,                                         /* split_top_skip_code 10 */
  tab_skip_no,                                               /* tab_skip_code 11 */
  −1,                                                        /* space_skip_code 12 */
  −1,                                                        /* xspace_skip_code 13 */
  par_fill_skip_no                                           /* par_fill_skip_code 14 */
  };
```

6.4.3 Parameters

```
⟨ Fix definitions for glue parameters 50 ⟩ ≡                              (50)
  glue_defined[zero_skip_no] = zero_glue; incr(glue_ref_count(zero_glue));
  for (i = line_skip_code; i ≤ par_fill_skip_code; i++)
    if (hmap_glue[i] ≥ 0) { glue_defined[hmap_glue[i]] = glue_par(i);
      incr(glue_ref_count(glue_par(i)));
    }
  max_ref[glue_kind] = MAX_GLUE_DEFAULT;                                 Used in 29.
```

6.4.4 Allocation

Next we define some auxiliar routines to compare glues for equality and to convert glues beween the different representations.

⟨ HiTEX auxiliar routines 11 ⟩ +≡ (51)

```
int glue_spec_equal(pointer p, pointer q)
{ return (width(q) ≡ width(p) ∧ stretch(q) ≡ stretch(p) ∧ shrink(q) ≡
      shrink(p) ∧ (stretch_order(q) ≡ stretch_order(p) ∨ stretch(q) ≡
      0) ∧ (shrink_order(q) ≡ shrink_order(p) ∨ shrink(q) ≡ 0));
}
int glue_equal(pointer p, pointer q)
{ return p ≡ q ∨ glue_spec_equal(p, q);
}
int glue_t_equal(glue_t *p, glue_t *q)
{ return (p→w.w ≡ q→w.w ∧ p→w.h ≡ q→w.h ∧ p→w.v ≡ q→w.v ∧ p→p.f ≡
      q→p.f ∧ p→m.f ≡ q→m.f ∧ (p→p.o ≡ q→p.o ∨ p→p.f ≡ 0.0) ∧ (p→m.o ≡
      q→m.o ∨ q→m.f ≡ 0.0));
}
```

To find a matching glue we make two passes over the defined glues: on the first pass we just compare pointers and on the second pass we also compare values. An alternative approach to speed up searching is used in section 6.7 below.

⟨ HiTEX routines 2 ⟩ +≡ (52)

```
static int hget_glue_no(pointer p)
{ static int rover = 0;
  int i;
  for (i = 0; i ≤ max_ref[glue_kind]; i++) {
    if (p ≡ glue_defined[rover]) return rover;
    else if (rover ≡ 0) rover = max_ref[glue_kind];
    else rover−−;
  }
  for (i = 0; i ≤ max_ref[glue_kind]; i++) { pointer q = glue_defined[rover];
    if (glue_spec_equal(p, q)) return rover;
    else if (rover ≡ 0) rover = max_ref[glue_kind];
    else rover−−;
  }
  if (max_ref[glue_kind] < #FF) { rover = ++max_ref[glue_kind];
    glue_defined[rover] = p; incr(glue_ref_count(p));
    DBG(DBGDEF, "Defining␣new␣glue␣%d\n", rover); return rover;
  }
  else return −1;
}
```

6.4.5 Output

⟨ Output glue definitions 53 ⟩ ≡ (53)
```
DBG(DBGDEF, "Maximum glue reference: %d\n", max_ref[glue_kind]);
for (i = max_fixed[glue_kind] + 1; i ≤ max_default[glue_kind]; i++) { glue_t g;
  to_glue_t(glue_defined[i], &g);
  if (¬glue_t_equal(&g, &glue_defaults[i])) HPUTDEF(hput_glue(&g), i);
}
for ( ; i ≤ max_ref[glue_kind]; i++) HPUTDEF(hout_glue_spec(glue_defined[i]), i);
```
Used in 31.

The above code uses the following conversion routine:

⟨ HiTEX auxiliar routines 11 ⟩ +≡ (54)
```
void to_glue_t(pointer p, glue_t *g)
{ g→w.w = width(p); g→w.h = g→w.v = 0.0;
  g→p.f = stretch(p)/(double) ONE; g→p.o = stretch_order(p);
  g→m.f = shrink(p)/(double) ONE; g→m.o = shrink_order(p);
}
```

6.5 Baseline Skips

TeX's baseline nodes just store a baseline skip reference number. We have seen this sitation before when dealing with extended dimensions and the solution here is the same: a dynamicaly allocated array.

6.5.1 Data

⟨ HiTEX variables 27 ⟩ +≡ (55)
```
typedef struct {
  pointer ls, bs;                    /* line skip and baselineskip gluespecs */
  scaled lsl;                                              /* lineskip limit */
} bl_defined_t;
static bl_defined_t *bl_defined = NULL;
static int bl_used = 0, bl_allocated = 0;
```

6.5.2 Initialization

The zero baseline skip is predefined which prevents an ambiguous info value of zero in a baseline node.

⟨ Initialize definitions for baseline skips 56 ⟩ ≡ (56)
```
bl_allocated = 8; ALLOCATE(bl_defined, bl_allocated, bl_defined_t);
bl_defined[zero_baseline_no].bs = zero_glue; incr(glue_ref_count(zero_glue));
bl_defined[zero_baseline_no].ls = zero_glue; incr(glue_ref_count(zero_glue));
bl_defined[zero_baseline_no].lsl = 0; bl_used = MAX_BASELINE_DEFAULT + 1;
max_ref[baseline_kind] = MAX_BASELINE_DEFAULT;
```
Used in 30.

6.5.3 Allocation

```
⟨ HiTEX routines 2 ⟩ +≡                                                    (57)
  int hget_baseline_no(pointer bs, pointer ls, scaled lsl)
  { static int rover = 0;
    int i;
    for (i = 0; i < bl_used; i++)                    /* search for an existing spec */
    { bl_defined_t *q = &(bl_defined[rover]);
      if (glue_equal(bs, q→bs) ∧ glue_equal(ls, q→ls) ∧ lsl ≡ q→lsl) return rover;
      else if (rover ≡ 0) rover = bl_used − 1;
      else rover−−;
    }
    if (bl_used ≥ bl_allocated) RESIZE(bl_defined, bl_allocated, bl_defined_t);
    rover = bl_used++;
    if (rover < #100) max_ref[baseline_kind] = rover;
    if (glue_equal(bs, zero_glue)) { bl_defined[rover].bs = zero_glue;
      incr(glue_ref_count(zero_glue));
    }
    else { bl_defined[rover].bs = bs; incr(glue_ref_count(bs));
    }
    if (glue_equal(ls, zero_glue)) { bl_defined[rover].ls = zero_glue;
      incr(glue_ref_count(zero_glue));
    }
    else { bl_defined[rover].ls = ls; incr(glue_ref_count(ls));
    }
    bl_defined[rover].lsl = lsl; return rover;
  }
```

6.5.4 Output

The following routine does not allocate a new glue definition, because the baselinedefinitions are output after the glue definitions. This is not perfect.

```
⟨ HiTEX auxiliar routines 11 ⟩ +≡                                          (58)
  uint8_t hout_baselinespec(int n)
  { info_t i = b000;
    pointer p;
    scaled s;
    p = bl_defined[n].bs;
    if (p ≠ zero_glue) { uint8_t *pos;
      uint8_t tag;
      HPUTNODE;                                                  /* allocate */
      pos = hpos; hpos++;                                            /* tag */
      tag = hout_glue_spec(p); *pos = tag; DBGTAG(tag, pos);
      DBGTAG(tag, hpos); HPUT8(tag); i |= b100;
    }
    p = bl_defined[n].ls;
```

```
  if (p ≠ zero_glue) { uint8_t *pos;
    uint8_t tag;
    HPUTNODE;                                              /* allocate */
    pos = hpos; hpos++;                                    /* tag */
    tag = hout_glue_spec(p); *pos = tag; DBGTAG(tag, pos);
    DBGTAG(tag, hpos); HPUT8(tag); i |= b010;
  }
  s = bl_defined[n].lsl;
  if (s ≠ 0) { HPUT32(s); i |= b001;
  }
  return TAG(baseline_kind, i);
}
```

⟨ Output baseline skip definitions 59 ⟩ ≡ (59)

```
DBG(DBGDEF, "Defining␣%d␣baseline␣skips\n", max_ref[baseline_kind]);
for (i = 1; i ≤ max_ref[baseline_kind]; i++) { uint32_t pos = hpos − hstart;
  uint8_t tag;
  hpos++;                                          /* space for the tag */
  HPUT8(i);                                                /* reference */
  tag = hout_baselinespec(i); hstart[pos] = tag; HPUT8(tag);
}
```

Used in 31.

6.5.5 Printing

The following function is needed in HiTEX to produce debugging output if needed.

⟨ HiTEX routines 2 ⟩ +≡ (60)

```
void print_baseline_skip(int i)
{
  if (0 ≤ i ∧ i < bl_used) { print_spec(bl_defined[i].bs, 0); print_char(',');
    print_spec(bl_defined[i].ls, 0); print_char(',');
    print_scaled(bl_defined[i].lsl);
  }
  else print_str("unknown");
}
```

6.6 Hyphenation

6.6.1 Data

For discretionary hyphens, we use again the pointer representation.

⟨ HiTEX variables 27 ⟩ +≡ (61)

```
static pointer hy_defined[#100];
```

6.6.2 Allocation

There are no predefined hyphens and so we start with two auxiliar functions and the function to get a hyphen number.

```
⟨ HiTEX auxiliar routines 11 ⟩ +≡                                    (62)
  static bool list_equal(pointer p, pointer q)
                                  /* a simple list compare for use in hget_hyphen */
  {
    while (true)
      if (p ≡ q) return true;
      else if (p ≡ null ∨ q ≡ null) return false;
      else if (is_char_node(p)∧is_char_node(q)∧font(p) ≡ font(q)∧character(p) ≡
            character(q)) { p = link(p); q = link(q);
      }
      else return false;
  }
  static pointer copy_disc_node(pointer p)
  { pointer q;
    q = get_node(small_node_size); pre_break(q) = copy_node_list(pre_break(p));
    post_break(q) = copy_node_list(post_break(p)); type(q) = type(p);
    subtype(q) = subtype(p);                    /* replace count and explicit bit */
    return q;
  }
⟨ HiTEX routines 2 ⟩ +≡                                              (63)
  int hget_hyphen_no(pointer p)
  { static int rover = 0;
    int i;
    for (i = 0; i ≤ max_ref[hyphen_kind]; i++) { pointer q = hy_defined[rover];
      if (is_auto_disc(p) ≡ is_auto_disc(q) ∧ replace_count(p) ≡
            replace_count(q) ∧ list_equal(pre_break(p),
            pre_break(q)) ∧ list_equal(post_break(p), post_break(q)))
        return rover;
      else if (rover ≡ 0) rover = max_ref[hyphen_kind];
      else rover−−;
    }
    if (max_ref[hyphen_kind] ≥ #FF) return −1;
    rover = ++max_ref[hyphen_kind]; hy_defined[rover] = copy_disc_node(p);
    return rover;
  }
```

6.6.3 Output

```
⟨ Output hyphen definitions 64 ⟩ ≡                                   (64)
  DBG(DBGDEF, "Maximum␣hyphen␣reference:␣%d\n", max_ref[hyphen_kind]);
  for (i = 0; i ≤ max_ref[hyphen_kind]; i++)
    HPUTDEF(hout_hyphen(hy_defined[i]), i);                    Used in 31.
```

6.7 Parameter Lists

6.7.1 Data

We store predefined parameter lists in a hash table in order to speed up finding existing parameter lists. The parameter list itself is stored as a byte sequence using the short HINT file format. We link the table entries in order of increasing reference numbers to be able to output them in a more "orderly" fashion.

```
⟨ HiTeX variables 27 ⟩ +≡                                              (65)
#define PLH_SIZE 313                          /* a prime number ≈ 2^8 × 1.2. */
  struct {
    int l;                                                      /* link */
    uint32_t h;                                                 /* hash */
    uint32_t n;                                               /* number */
    uint32_t s;                                                 /* size */
    uint8_t *p;                                               /* pointer */
  } pl_defined[PLH_SIZE] = {{0}};
  int pl_head = -1, *pl_tail = &pl_head;
```

6.7.2 Allocation

Next we define three short auxiliar routines and the *hget_param_list_no* function.

```
⟨ HiTeX routines 2 ⟩ +≡                                                 (66)
  static uint32_t hparam_list_hash(list_t *l)
  { uint32_t h = 0;
    uint32_t i;
    for (i = 0; i < l→s; i++) h = 3 * h + hstart[l→p + i];
    return i;
  }
  static bool pl_equal(list_t *l, uint8_t *p)
  { uint8_t *q = hstart + l→p;
    uint32_t i;
    for (i = 0; i < l→s; i++)
      if (q[i] ≠ p[i]) return false;
    return true;
  }
  static void pl_copy(list_t *l, uint8_t *p)
  { uint8_t *q = hstart + l→p;
    memcpy(p, q, l→s);
  }
  int hget_param_list_no(list_t *l)
  { uint32_t h;
    int i;
```

```
  if (l→s ≤ 0) return −1;
  h = hparam_list_hash(l); i = h % PLH_SIZE;
  while (pl_defined[i].p ≠ NULL) {
    if (pl_defined[i].h ≡ h ∧ pl_equal(l, pl_defined[i].p)) return pl_defined[i].n;
    i = i + 199;                                   /* some other prime */
    if (i ≥ PLH_SIZE) i = i − PLH_SIZE;
  }
  if (max_ref[param_kind] ≥ #FF) return −1;
  pl_defined[i].n = ++max_ref[param_kind]; *pl_tail = i;
  pl_tail = &(pl_defined[i].l); pl_defined[i].l = −1; pl_defined[i].h = h;
  pl_defined[i].s = l→s; ALLOCATE(pl_defined[i].p, l→s, uint8_t);
  pl_copy(l, pl_defined[i].p); return pl_defined[i].n;
}
```

6.7.3 Output

To output parameter lists, we need a function to output a parameter node:

⟨ HiTEX routines 2 ⟩ +≡ (67)

```
void hdef_param_node(int ptype, int pnumber, int pvalue)
{
  if (ptype ≡ int_type) {
    if (pvalue ≡ int_defined[hmap_int[pnumber]]) return;
    else HPUTDEF(hput_int(pvalue), hmap_int[pnumber]);
  }
  else if (ptype ≡ dimen_type) {
    if (pvalue ≡ dimen_defined[hmap_dimen[pnumber]]) return;
    else HPUTDEF(hput_dimen(pvalue), hmap_dimen[pnumber]);
  }
  else if (ptype ≡ glue_type) {
    if (glue_equal(pvalue, glue_defined[hmap_glue[pnumber]])) return;
    else HPUTDEF(hout_glue_spec((pointer) pvalue), hmap_glue[pnumber]);
  }                                          /* I need to check for xdimens here
        HPUTDEF(hput_xdimen(&xdimen_defined[i]), i); */
  else QUIT("Unexpected parameter type %d", ptype);
}
```

Now we use the linked list starting with *pl_head* to output the predefined parameter lists sorted by their reference number.

⟨ Output parameter list definitions 68 ⟩ ≡ (68)

```
DBG(DBGDEF, "Defining %d parameter lists\n", max_ref[param_kind] + 1);
for (i = pl_head; i ≥ 0; i = pl_defined[i].l) { int j;
  DBG(DBGDEF, "Defining parameter list %d, size 0x%x\n", i,
      pl_defined[i].s); j = hsize_bytes(pl_defined[i].s);
  HPUTX(1 + 1 + j + 1 + pl_defined[i].s + 1 + j + 1); HPUTTAG(param_kind, j + 1);
  HPUT8(pl_defined[i].n); hput_list_size(pl_defined[i].s, j); HPUT8(#100 − j);
  memcpy(hpos, pl_defined[i].p, pl_defined[i].s); hpos = hpos + pl_defined[i].s;
```

```
    HPUT8(#100 − j); hput_list_size(pl_defined[i].s, j);
    HPUTTAG(param_kind, j + 1);
  }
```
Used in 31.

6.8 Fonts

6.8.1 Data

Last not least we consider fonts. To store a font definition, we define the data type *font_t* and an array *hfonts* of pointers indexed by HINT font numbers. To map TEX font numbers to HINT font numbers we use the array *hmap_font*.

```
⟨ HiTEX variables 27 ⟩ +≡                                                    (69)
#define MAX_FONTS #100
  typedef struct {
    uint8_t i;                  /* maps HINT font number to TEX font number */
    pointer g;                                                  /* space glue */
    pointer h;                                                  /* default hyphen */
    pointer p[MAX_FONT_PARAMS];                                 /* font parameters */
    uint16_t m;                                 /* section number of font tfm file */
    uint16_t y;                                 /* section number of font pk file */
  } font_t;
  static font_t *hfonts[MAX_FONTS] = {NULL};
  static int hmap_font[MAX_FONTS];
```

6.8.2 Mapping

```
⟨ Initialize definitions for fonts 70 ⟩ ≡                                    (70)
  for (i = 0; i < #100; i++) hmap_font[i] = −1;
  max_ref[font_kind] = −1;
```
Used in 30.

6.8.3 Allocation

```
⟨ HiTEX auxiliar routines 11 ⟩ +≡                                            (71)
  static pointer hget_font_space(uint8_t f)
  { pointer p;
    if (space_skip ≡ zero_glue) p = find_space_glue(f);
    else p = glue_par(space_skip_code);
    add_glue_ref(p); return p;
  }
  static pointer hget_font_hyphen(uint8_t f)
  { pointer p;
    int c;
    p = new_disc( ); c = hyphen_char[f];
    if (c ≥ 0 ∧ c < 256) pre_break(p) = new_character(f, c);
    return p;
  }
```

```
  static void hdef_font_params(pointer p[MAX_FONT_PARAMS])
  { }
⟨ HiTEX routines 2 ⟩ +≡ (72)
  uint8_t hget_font_no(uint8_t f)
                        /* f a tex internal font number, g a hint font number */
  { int g;
    char *n, *fn;
    int l;
    g = hmap_font[f]; DBG(DBGDEF, "Trying TeX font %d->%d\n", f, g);
    if (g ≥ 0) return g;
    DBG(DBGDEF, "New TeX font %d\n", f);
    if (max_ref[font_kind] ≥ #100) QUIT("too many fonts in use");
    if (f ≡ 0)                    /* replace the nullfont by some default font */
      return 0;
    g = ++(max_ref[font_kind]); ALLOCATE(hfonts[g], 1, font_t);
    hfonts[g]→i = f; hmap_font[f] = g; hfonts[g]→g = hget_font_space(f);
    hfonts[g]→h = hget_font_hyphen(f);
    fn = hfile_name(font_name[f], empty_string, empty_string);
    n = ktex_find_tfm(fn); l = strlen(n);
    if (l ≥ file_name_size) QUIT("Filename of font metric file too long");
    hfonts[g]→m = hnew_file_section(n); free(n); n = ktex_find_pk(fn);
    l = strlen(n);
    if (l ≥ file_name_size) QUIT("Filename of font pk file too long");
    hfonts[g]→y = hnew_file_section(n);        /* for now, we use 600pk fonts */
    free(n); return g;
  }
```

6.8.4 Output

```
⟨ Output font definitions 73 ⟩ ≡ (73)
  { int f;
    DBG(DBGDEF, "Defining %d fonts\n", max_ref[font_kind] + 1);
    for (f = 0; f ≤ max_ref[font_kind]; f++) { font_t *hf = hfonts[f];
      internal_font_number g = hf→i;
      uint32_t pos = hpos − hstart;
      info_t i = b000;
      DBG(DBGDEF, "Defining font %d size 0x%x\n", f, font_size[f]); hpos++;
      HPUTNODE;                                  /* space for the tag and the node */
      HPUT8(f);                                                    /* reference */
      hout_string(font_id_text(g));
      if (font_size[g] ≠ 0) { HPUT32(font_size[g]); i = b001;
      }
      HPUT16(hf→m); HPUT16(hf→y);
      DBG(DBGDEF, "Defining font space\n");
      HPUTCONTENT(hout_glue_spec, hf→g);
```

```
    DBG(DBGDEF, "Defining␣font␣hyphen\n");
    HPUTCONTENT(hout_hyphen, hf→h); hdef_font_params(hf→p);
    DBG(DBGDEF, "End␣of␣font␣%d\n", f); hput_tags(pos, TAG(font_kind, i));
  }
}
```

Used in 31.

We used the following function to write a TEX string to the HINT file:

⟨ HiTEX auxiliar routines 11 ⟩ +≡ (74)

```
void hout_string(int s)                    /* put a TeX string into a hint file */
{ pool_pointer j;
  uint8_t c;
  j = str_start[s];
  while (j < str_start[s + 1]) { c = so(str_pool[j++]);
    if (c ≡ '%' ∨ c < #20 ∨ c ≥ #7F) { char str[4];
      snprintf(str, 4, "%%%02X", c);       /* convert to printable ASCII */
      HPUTX(3); HPUT8(str[0]); HPUT8(str[1]); HPUT8(str[2]);
    }
    else { HPUTX(1); HPUT8(c);
    }
  }
  HPUT8(0);
}
```

We used the following macro to add tags around the font glue and the font hyphen:

⟨ HiTEX macros 26 ⟩ +≡ (75)

```
#define HPUTCONTENT (F, D)
  { uint8_t *_p;
    uint8_t _t;
    HPUTNODE;                                                  /* allocate */
    _p = hpos++;                                                   /* tag */
    _t = F(D); *_p = _t; DBGTAG(_t, _p); DBGTAG(_t, hpos); HPUT8(_t);
  }
```

7 HINT Content

TEX puts content nodes on the contribution list and once in a while calls *build_page* to move nodes from the contribution list to the current page. HiTEX has a special version of *build_page* that will simply remove nodes from the contribution list and passe them to the function *hout_node*.

```
⟨ HiTEX routines 2 ⟩ +≡                                                    (76)
  void hout_node(pointer p)
  { uint32_t pos = hpos − hstart;
    uint8_t tag;
    HPUTNODE; hpos++;
    if (is_char_node(p)) ⟨ output a character node 77 ⟩
    else
      switch (type(p))
      { ⟨ cases to output content nodes 78 ⟩
        default: MESSAGE("\nOutput␣of␣node␣type=%d␣subtype=%d␣not\
            ␣implemented\n", type(p), subtype(p)); display_node(p);
          MESSAGE("End␣of␣node"); hpos−−; return;
        }
      hput_tags(pos, tag);
    }
```

To output HINT nodes, we use the functions defined in `hput.c` from the `shrink` program (see [12]).

Let's start with character nodes.

7.1 Characters

The processing of a character node consist of three steps: checking for definitions, converting the TEX node pointed to by p to a HINT data type, here a **glyph_t**, and using the corresponding `hput_...` function to output the node and return the *tag*. In the following, we will see the same approach in many small variations for all kinds of nodes.

```
⟨ output a character node 77 ⟩ ≡                                          (77)
  { glyph_t g;
    g.f = hget_font_no(font(p)); g.c = character(p); tag = hput_glyph(&g);
  }                                                                  Used in 76.
```

7.2 Penalties

Integer nodes, which as content nodes are used for penalties, come next. Except for the embedding between **case** and **break**, the processing of penalty nodes follows the same pattern we have just seen.

```
⟨ cases to output content nodes 78 ⟩ ≡                                    (78)
case penalty_node:
  { int n;
    n = hget_int_no(penalty(p));
    if (n < 0) tag = hput_int(penalty(p));
    else { HPUT8(n); tag = TAG(penalty_kind, 0);
    }
  }
  break;                                                          Used in 76.
```

7.3 Kerns

The kern nodes of TEX contain a single dimension and a flag to mark "explicit" kerns.

```
⟨ cases to output content nodes 78 ⟩ +≡                                   (79)
case kern_node:
  { int n;
    n = hget_dimen_no(width(p));
    if (n < 0) { kern_tk; k.x = (subtype(p) ≡ explicit); k.d.w = width(p);
      k.d.h = k.d.v = 0.0; tag = hput_kern(&k);
    }
    else { HPUT8(n);
      if (subtype(p) ≡ explicit) tag = TAG(kern_kind, b100);
      else tag = TAG(kern_kind, b000);
    }
  }
  break;
```

7.4 Extended Dimensions

Extended dimensions do not consitute content on their own, but nodes containing an extended dimension are part of other nodes. Here we define an auxiliar function that checks for a predefined extended dimension and if found outputs a the reference number and returns false; otherwise it outputs the extended dimension and returns true.

```
⟨ HiTEX auxiliar routines 11 ⟩ +≡                                         (80)
  bool hout_xdimen(int n)
  { if (n ≥ xdimen_used)
      QUIT("xdimen number out of range %d>%d", n, xdimen_used − 1);
    if (n > max_ref[xdimen_kind])
    { hput_xdimen_node(&xdimen_defined[n]); return true; }
    else { HPUT8(n); return false; }
```

```
}
```

7.5 Mathematics

```
⟨ cases to output content nodes 78 ⟩ +≡                                    (81)
case math_node:
  { math_t m;
    m.on = (subtype(p) ≡ before); m.w = width(p); tag = hput_math(&m);
  }
  break;
```

7.6 Glue and Leaders

Because glue specifications and glue nodes are sometimes part of other nodes, we start with three auxiliar functions: The first simply converts a HiTEX glue node into a HINT **glue_t**, outputs it and returns the tag; the second checks for predefined glues, and the third outputs a complete glue node including tags.

```
⟨ HiTEX auxiliar routines 11 ⟩ +≡                                          (82)
  static uint8_t hout_glue_spec(pointer p)
  { glue_t g;
    to_glue_t(p, &g); return hput_glue(&g); }
  static uint8_t hout_glue(pointer p)
  { int n;
    n = hget_glue_no(p);
    if (n < 0) return hout_glue_spec(p);
    else { HPUT8(n); return TAG(glue_kind, 0); }
  }
  static void hout_glue_node(pointer p)
  { uint8_t *pos;
    uint8_t tag;
    HPUTNODE;                                                  /* allocate */
    pos = hpos; hpos++;                                            /* tag */
    tag = hout_glue(p); *pos = tag; DBGTAG(tag, pos); DBGTAG(tag, hpos);
    HPUT8(tag);
  }
```

Since TEX implements leaders as a kind of glue, we have one case statement covering glue and leaders.

```
⟨ cases to output content nodes 78 ⟩ +≡                                    (83)
case glue_node:
  if (subtype(p) < cond_math_glue)                          /* normal glue */
    tag = hout_glue(glue_ptr(p));
  else if (a_leaders ≤ subtype(p) ∧ subtype(p) ≤ x_leaders)      /* leaders */
  { hout_glue_node(glue_ptr(p)); hout_node(leader_ptr(p));
    tag = TAG(leaders_kind, subtype(p) − a_leaders + 1);
  }
```

```
else QUIT("glue subtype %d not implemented\n", subtype(p));
break;
```

7.7 Hyphenation

Hyphens are needed in font descriptions (see section). Therefore we define a function that converts TEX's *disc_node* pointers to HINTs **hyphen_t**, outputs the hyphen, and returns the tag.

⟨ HiTEX auxiliar routines 11 ⟩ +≡ (84)

```
uint8_t hout_hyphen(pointer p)
{ hyphen_t h;
  h.x = ¬is_auto_disc(p);
  if (pre_break(p) ≡ null) h.p.s = 0;
  else { uint32_t lpos;
    lpos = hpos − hstart; h.p.k = list_kind;
    hout_list_node(pre_break(p), lpos, &(h.p));
  }
  if (post_break(p) ≡ null) h.q.s = 0;
  else { uint32_t lpos;
    lpos = hpos − hstart; h.q.k = list_kind;
    hout_list_node(post_break(p), lpos, &(h.q));
  }
  h.r = replace_count(p); return hput_hyphen(&h);
}
```

⟨ cases to output content nodes 78 ⟩ +≡ (85)

```
case disc_node:
  { int n;
    n = hget_hyphen_no(p);
    if (n < 0) tag = hout_hyphen(p);
    else { HPUT8(n); tag = TAG(hyphen_kind, 0);
    }
  }
  break;
```

7.8 Ligatures

The subtype giving information on left and right boundary characters is ignored since the HINT viewer will not do ligature or kerning programms and neither attempt hyphenation.

⟨ cases to output content nodes 78 ⟩ +≡ (86)

```
case ligature_node:
  { lig_t l;
    pointer q;
```

```
    l.f = hget_font_no(font(lig_char(p))); HPUT8(l.f); l.l.p = hpos − hstart;
    hput_utf8(qo(character(lig_char(p)))); q = lig_ptr(p);
    while (q > null) { hput_utf8(qo(character(q))); q = link(q);
    }
    l.l.s = (hpos − hstart) − l.l.p; tag = hput_ligature(&l);
  }
  break;
```

7.9 Rules

```
⟨ cases to output content nodes 78 ⟩ +≡ (87)
  case rule_node:
    { rule_t r;
      if (is_running(height(p))) r.h = RUNNING_DIMEN;
      else r.h = height(p);
      if (is_running(depth(p))) r.d = RUNNING_DIMEN;
      else r.d = depth(p);
      if (is_running(width(p))) r.w = RUNNING_DIMEN;
      else r.w = width(p);
      tag = hput_rule(&r);
    }
    break;
```

7.10 Boxes

```
⟨ cases to output content nodes 78 ⟩ +≡ (88)
  case hlist_node: case vlist_node:
    if (type(p) ≡ hlist_node) tag = TAG(hbox_kind, 0);
    else tag = TAG(vbox_kind, 0);
    tag |= hput_box_dimen(height(p), depth(p), width(p));
      tag |= hput_box_shift(shift_amount(p));
      tag |= hput_box_glue_set((glue_sign(p) ≡ stretching) ? +1 : −1, glue_set(p),
          glue_order(p));
    { list_t l;
      uint32_t pos;
      pos = hpos − hstart; l.k = list_kind; hout_list_node(list_ptr(p), pos, &l);
    }
    break;
```

7.11 Adjustments

```
⟨ cases to output content nodes 78 ⟩ +≡ (89)
  case adjust_node:
    { list_t l;
      l.k = adjust_kind; HPUT8(0);                                  /* size */
      tag = hout_list(adjust_ptr(p), pos + 1, &l);
    }
    break;
```

7.12 Marks

We currently ignore Marks.

⟨ cases to output content nodes 78 ⟩ +≡ (90)
```
case mark_node: hpos--; return;
```

7.13 Whatsit Nodes

We have added custom whatsit nodes and now we switch based on the subtype.

⟨ cases to output content nodes 78 ⟩ +≡ (91)
```
case whatsit_node:
  switch (subtype(p)) { ⟨ cases to output whatsit content nodes 92 ⟩
  default:
    MESSAGE("\nOutput␣of␣whatsit␣nodes␣subtype=%d␣not␣implemented\n",
        subtype(p)); display_node(p); MESSAGE("End␣of␣node"); hpos--;
    return;
  }
  break;
```

7.14 TEX's Whatsit Nodes

Here we remove the tag from the output and call the standard TEXroutine or we ignore the node.

⟨ cases to output whatsit content nodes 92 ⟩ ≡ (92)
```
case open_node: case write_node: case close_node:
                                  /* out_what(p); this does not work for LaTeX */
  case special_node: case language_node: hpos--; return;
```
Used in 91.

7.15 Paragraphs

⟨ cases to output whatsit content nodes 92 ⟩ +≡ (93)
```
case graf_node:
  { uint32_t pos;
    list_t l;
    int n;
    info_t i = b000;
    n = graf_extent(p);
    if (hout_xdimen(n)) i |= b100;
    pos = hpos - hstart; l.k = param_kind;
    n = hout_param_list(graf_params(p), pos, &l);
    if (n ≥ 0) HPUT8(n);
    else i |= b010;
    pos = hpos - hstart; l.k = list_kind; hout_list_node(graf_list(p), pos, &l);
    tag = TAG(par_kind, i);
  }
  break;
#if 0
```

```
  hprintf("graf"); hprint_bits(0, graf_continue(p) ≠ 0,
      graf_penalty(p) ≠ default_int[widow_penalty_code].pv);
                    /* whether to use final_widow_penalty or widow_penalty */
  if (graf_continue(p) ≠ 0) hprintf("(continued)␣");
  if (graf_penalty(p) ≠ default_int[widow_penalty_code].pv)
    hprintf("(final_widow_penalty)␣%d␣", graf_penalty(p));
  hprintf("(extent)␣%d␣", graf_extent(p));
  hprint_node_list("parameter", graf_params(p)); hprint_text(graf_list(p)); break;
#endif
```

7.16 Baseline Skips

```
⟨ cases to output whatsit content nodes 92 ⟩ +≡                    (94)
  case baseline_node:
    { int n;
      n = baseline_node_no(p);
      if (n > #FF) tag = hout_baselinespec(n);
      else { HPUT8(n); tag = TAG(baseline_kind, b000);
      }
    }
    break;
```

7.17 Displayed Equations

```
⟨ cases to output whatsit content nodes 92 ⟩ +≡                    (95)
  case disp_node:
    { uint32_t pos;
      list_t l;
      int n;
      info_t i = b000;
      pos = hpos − hstart; l.k = param_kind;
      n = hout_param_list(display_params(p), pos, &l);
      if (n ≥ 0) HPUT8(n);
      else i |= b100;
      if (display_eqno(p) ≠ null ∧ display_left(p)) { hout_node(display_eqno(p));
        i |= b010;
      }
      pos = hpos − hstart; l.k = list_kind;
      hout_list_node(display_formula(p), pos, &l);
      if (display_eqno(p) ≠ null ∧ ¬display_left(p)) { hout_node(display_eqno(p));
        i |= b001;
      }
      tag = TAG(display_kind, i);
                /* the display_no_bs(p) tells whether the baseline skip is ignored */
    }
    break;
```

7.18 Extended Boxes

⟨ cases to output whatsit content nodes 92 ⟩ +≡ (96)

```
case hset_node: case vset_node:
  { kind_t k = subtype(p) ≡ hset_node ? hset_kind : vset_kind;
    info_t i = b000;
    stretch_t s;
    int n = set_extent(p);
    if (hout_xdimen(n)) i |= b001;
    i |= hput_box_dimen(height(p), depth(p), width(p));
      i |= hput_box_shift(shift_amount(p)); s.f = set_stretch(p)/(double) ONE;
      s.o = set_stretch_order(p); hput_stretch(&s); s.f = set_shrink(p)/(double)
          ONE; s.o = set_shrink_order(p); hput_stretch(&s);
    { list_t l;
      uint32_t pos;
      pos = hpos − hstart; l.k = list_kind; hout_list_node(list_ptr(p), pos, &l);
    }
    tag = TAG(k, i);
  }
  break;
```

⟨ cases to output whatsit content nodes 92 ⟩ +≡ (97)

```
case hpack_node:
  { info_t i = b000;
    int n = pack_extent(p);
    if (hout_xdimen(n)) i |= b100;
    if (pack_m(p) ≡ exactly) i = b000;
    else i = b010;
    { list_t l;
      uint32_t pos;
      pos = hpos − hstart; l.k = list_kind; hout_list_node(list_ptr(p), pos, &l);
    }
    tag = TAG(hpack_kind, i);
  }
  break;
case vpack_node:
  { info_t i = b000;
    int n = pack_extent(p);
    if (hout_xdimen(n)) i |= b100;
    if (pack_m(p) ≡ exactly) i = b000;
    else i = b010;
    if (pack_limit(p) ≠ max_dimen) { HPUT32(pack_limit(p)); i |= b001;
    }
    { list_t l;
      uint32_t pos;
```

```
      pos = hpos − hstart; l.k = list_kind; hout_list_node(list_ptr(p), pos, &l);
    }
    tag = TAG(vpack_kind, i);
  }
  break;
```

7.19 Extended Alignments

⟨ cases to output whatsit content nodes 92 ⟩ +≡ (98)

```
case align_node:
  { info_t i = b000;
    if (align_m(p) ≡ additional) i |= b010;
    if (align_v(p)) i |= b001;
    if (hout_xdimen(align_extent(p))) i |= b100;
    hout_preamble(align_preamble(p)); hout_align_list(align_list(p), align_v(p));
    tag = TAG(table_kind, i);
  }
  break;
```

In the preamble we remove the unset nodes and retain only the list of tabskip glues.

⟨ HiTEX auxiliar routines 11 ⟩ +≡ (99)

```
void hout_preamble(pointer p)
{ pointer q, r;
  list_t l;
  DBG(DBGBASIC, "Writing␣Preamble\n"); q = p;
  if (q ≠ null) r = link(q);
  else r = null;
  while (r ≠ null) {
    if (type(r) ≡ unset_node) { link(q) = link(r); link(r) = null;
      flush_node_list(r);
    }
    else q = r;
    r = link(q);
  }
  l.k = list_kind; hout_list_node(p, hpos − hstart, &l);
  DBG(DBGBASIC, "End␣Preamble\n");
}
```

In the *align_list* we have to convert the unset nodes back to box nodes or extended box nodes packaged inside an item node. When the viewer reads an item node, it will package the extended boxes to their natural size. This is the size that is needed to compute the maximum width of a column.

⟨ HiTEX auxiliar routines 11 ⟩ +≡ (100)

```
static void hout_item(pointer p, uint8_t t, uint8_t s)
{ info_t i = b000;
  uint8_t n;
```

```
  n = span_count(p) + 1;
  DBG(DBGBASIC, "Writing␣Item␣%d/%d->%d/%d\n", type(p), n, t, s);
  display_node(p);
  if (n ≡ 0) QUIT("Span␣count␣of␣item␣must␣be␣positive");
  if (n < 7) i = n;
  else i = 7;
  HPUTTAG(item_kind, i); type(p) = t; subtype(p) = s; hout_node(p);
  if (i ≡ 7) HPUT8(n);
  HPUTTAG(item_kind, i); DBG(DBGBASIC, "End␣Item\n");
}
static void hout_item_list(pointer p, bool v)
{ list_t l;
  uint32_t pos;
  DBG(DBGBASIC, "Writing␣Item␣List\n"); l.k = list_kind;
  HPUTTAG(item_kind, b000); pos = hpos − hstart; HPUTX(2); HPUT8(0);
                                                  /* space for the list tag */
  HPUT8(0);                                       /* space for the list size */
  l.p = hpos − hstart;
  while (p > mem_min) {
    if (is_char_node(p)) hout_node(p);
    else if (type(p) ≡ unset_node) hout_item(p, v ? vlist_node : hlist_node, 0);
    else if (type(p) ≡ unset_set_node)
      hout_item(p, whatsit_node, v ? vset_node : hset_node);
    else if (type(p) ≡ unset_pack_node)
      hout_item(p, whatsit_node, v ? vpack_node : hpack_node);
    else hout_node(p);
    p = link(p);
  }
  l.s = (hpos − hstart) − l.p; hput_tags(pos, hput_list(pos + 1, &l));
  HPUTTAG(item_kind, b000); DBG(DBGBASIC, "End␣Item␣List\n");
}
void hout_align_list(pointer p, bool v)
{ list_t l;
  uint32_t pos;
  DBG(DBGBASIC, "Writing␣Align␣List\n"); l.k = list_kind;
  pos = hpos − hstart; HPUTX(2); HPUT8(0);        /* space for the tag */
  HPUT8(0);                                       /* space for the list size */
  l.p = pos + 2;
  while (p > mem_min) {
    if (¬is_char_node(p) ∧ (type(p) ≡ unset_node ∨ type(p) ≡
        unset_set_node ∨ type(p) ≡ unset_pack_node))
      hout_item_list(list_ptr(p), v);
    else hout_node(p);
    p = link(p);
  }
```

```
    l.s = (hpos − hstart) − l.p; hput_tags(pos, hput_list(pos + 1, &l));
    DBG(DBGBASIC, "End␣Align␣List\n");
  }
```

Inside the alignment list we will find various types of unset nodes, we convert them back to regular nodes and put them inside an item node.

```
⟨ cases to output content nodes 78 ⟩ +≡ (101)
  case unset_node: case unset_set_node: case unset_pack_node:
```

7.20 Images

```
⟨ cases to output whatsit content nodes 92 ⟩ +≡ (102)
  case image_node:
    { image_t i;
      i.n = image_no(p); i.w = image_width(p); i.h = image_height(p);
      i.p.f = image_stretch(p)/(double) ONE; i.p.o = image_stretch_order(p);
      i.m.f = image_shrink(p)/(double) ONE; i.m.o = image_shrink_order(p);
      tag = hput_image(&i);
    }
    break;
```

7.21 Lists

Two functions are provided here: *hout_list* will write a list given by the pointer p to the output at the current position *hpos*. After the list has finished, it will move the list, if necessary, and add the size information so that the final list will be at position *pos*; *hout_list_node* uses *hout_list* but adds the tags to form a complete node.

```
⟨ HiTEX routines 2 ⟩ +≡ (103)
  static uint8_t hout_list(pointer p, uint32_t pos, list_t *l)
  { l→p = hpos − hstart;
    while (p > mem_min) { hout_node(p); p = link(p);
    }
    l→s = (hpos − hstart) − l→p; return hput_list(pos, l);
  }
  static void hout_list_node(pointer p, uint32_t pos, list_t *l)
      /* p is a pointer to a node list, output the node list at position pos (thats
        where the tag goes) using l-¿k as list kind, and set l-¿p and l-¿s. */
  { hpos = hstart + pos; HPUTX(3); HPUT8(0);          /* space for the tag */
    HPUT8(0);                                        /* space for the list size */
    HPUT8(0);                                /* space for the size boundary byte */
    hput_tags(pos, hout_list(p, pos + 1, l));
  }
```

7.22 Parameter Lists

The next function is like *hout_list_node* but restricted to parameter nodes.

⟨ HiTEX routines 2 ⟩ +≡ (104)

```
static int hout_param_list(pointer p, uint32_t pos, list_t *l)
    /* p is a pointer to a param node list, either find a reference number to a
      predefined parameter list and return the reference number or output the
      node list at position pos (thats where the tag goes) and set l-¿k, l-¿p and
      l-¿s, and return -1; */
{ int n;
  if (p ≡ null) return −1;                    /* omit empty parameter lists */
  hpos = hstart + pos; HPUTX(3); HPUT8(0);          /* space for the tag */
  HPUT8(0);                                    /* space for the list size */
  HPUT8(0);                                /* space for the size boundary byte*/
  l→p = hpos − hstart;
  while (p > mem_min) {
    hdef_param_node(par_type(p), par_number(p), par_value(p).i); p = link(p);
  }
  l→s = (hpos − hstart) − l→p; n = hget_param_list_no(l);
  if (n ≥ 0) hpos = hstart + pos;
  else hput_tags(pos, hput_list(pos + 1, l));
  return n;
}
```

7.23 Text

The routines in this section are not yet ready.

⟨ HiTEX routines 2 ⟩ +≡ (105)

```
#if 0
  static void hchange_text_font(internal_font_number f)
  { uint8_t g;
    if (f ≠ hfont) { g = get_font_no(f);
      if (g < 8) hputcc(FONTO_CHAR + g);
      else { hputcc(FONTN_CHAR); hputcc(g);
      }
      hfont = f;
    }
  }
  static void hprint_text_char(pointer p)
  { uint8_t f, c;
    f = font(p); c = character(p); hchange_text_font(f);
    if (c ≤ SPACE_CHAR) hputcc(ESC_CHAR);
    hputcc(c);
  }
```

```
static void hprint_text_node(pointer p)
{
  switch (type(p)) {
  case hlist_node:                    /* this used to be the par_indent case */
    goto nodex;
  case glue_node:
    if (subtype(p) ≥ cond_math_glue) goto nodex;
    else { pointer q = glue_ptr(p);
      int i;
      if (glue_equal(f_space_glue[hfont], q)) { hputc(SPACE_CHAR); return;
      }
      if (glue_equal(f_xspace_glue[hfont], q)) { hputcc(XSPACE_CHAR); return;
      }
      if (f_1_glue[hfont] ≡ 0 ∧ (subtype(p) − 1 ≡ space_skip_code)) {
        pointer r = glue_par(subtype(p) − 1);
        add_glue_ref(r); f_1_glue[hfont] = r;
      }
      if (f_1_glue[hfont] ≠ 0 ∧ glue_equal(f_1_glue[hfont], q)) {
        hputcc(GLUE1_CHAR); return;
      }
      if (f_2_glue[hfont] ≡ 0∧(subtype(p)−1 ≡ space_skip_code ∨ subtype(p)−1 ≡
            xspace_skip_code)) { pointer r = glue_par(subtype(p) − 1);
        add_glue_ref(r); f_2_glue[hfont] = r;
      }
      if (f_2_glue[hfont] ≠ 0 ∧ glue_equal(f_2_glue[hfont], q)) {
        hputcc(GLUE2_CHAR); return;
      }
      if (f_3_glue[hfont] ≡ 0) { f_3_glue[hfont] = q; add_glue_ref(q);
      }
      if (f_3_glue[hfont] ≠ 0 ∧ glue_equal(f_3_glue[hfont], q)) {
        hputcc(GLUE3_CHAR); return;
      }
      i = hget_glue_no(q);
      if (i ≥ 0) { hputcc(GLUEN_CHAR); hputcc(i); return;
      }
    }
    break;
  case ligature_node:
    { int n;
      pointer q;
      for (n = 0, q = lig_ptr(p); n < 5 ∧ q ≠ null; n++, q = link(q)) continue;
      if (n ≡ 2) hputcc(LIG2_CHAR);
      else if (n ≡ 3) hputcc(LIG3_CHAR);
      else if (n ≡ 0) hputcc(LIG0_CHAR);
      else goto nodex;
```

```
        hprint_text_char(lig_char(p));
        for (q = lig_ptr(p); q ≠ null; q = link(q)) hprint_text_char(q);
        return;
      }
    case disc_node:
      if (post_break(p) ≡ null ∧ pre_break(p) ≠ null ∧ replace_count(p) ≡ 0) {
        pointer q;
        q = pre_break(p);
        if (is_char_node(q) ∧ link(q) ≡ null ∧ font(q) ≡ hfont ∧ character(q) ≡
            hyphen_char[hfont]) {
          if (is_auto_disc(p)) hputcc(DISC1_CHAR);
          else hputcc(DISC2_CHAR);
          return;
        }
      }
      else if (post_break(p) ≡ null ∧ pre_break(p) ≡ null ∧ replace_count(p) ≡
          0 ∧ ¬is_auto_disc(p)) { hputcc(DISC3_CHAR); return;
      }
      break;
    case math_node:
      if (width(p) ≠ 0) goto nodex;
      if (subtype(p) ≡ before) hputcc(MATHON_CHAR);
      else hputcc(MATHOFF_CHAR);
      return;
    default: break;
    }
  nodex: hout_node(p);
  }
  static void hprint_text(pointer p)
  { internal_font_number f = hfont;
    nesting++; hprint_nesting( ); hprintf("<text␣");
    while (p > mem_min) {
      if (is_char_node(p)) hprint_text_char(p);
      else hprint_text_node(p);
      p = link(p);
    }
    hchange_text_font(f); hprintf(">\n"); nesting--;
  }
#endif
```

7.24 Streams

Streams and inserts are not yet supported.

⟨ HiTEX routines 2 ⟩ +≡ (106)

```
#if 0
```

```
  static void hprint_streams(void)
  { int k;
    for (k = 1; k < streams_used; k++) { hprint_nesting( );
      hprintf("<streamdef");
      hprint_bits(streams[k].p ≥ 0, streams[k].n ≥ 0, streams[k].r > 0);
      hprintf("␣%d␣(factor)␣%d␣␣(extent)␣%d␣", k, streams[k].f, streams[k].e);
      nesting++;
      if (streams[k].p ≥ 0) { hprint_nesting( );
        hprintf("(prefere)␣%d␣", streams[k].p);
      }
      if (streams[k].n ≥ 0) { hprint_nesting( );
        hprintf("(next)␣%d␣", streams[k].n);
      }
      if (streams[k].r > 0) { hprint_nesting( );
        hprintf("(split␣ratio)␣%d␣", streams[k].r);
      }
      hprint_node_list("pre", streams[k].b); hprint_nesting( );
      hprintf("<glue␣(top_skip)␣"); hprint_glue_spec(streams[k].g);
      hprintf(">"); hprint_node_list("post", streams[k].a); nesting--;
      hprintf(">");
    }
  }
#endif
#if 0
  static bool is_template(pointer p)          /* check whether p contains nodes
        that need repackaging. If yes, its a template return true, else its not a
        template return false. This function list all node types in a template that
        may contain streams and need repackaging upon output. */
  {
    while (p ≠ null) {
      switch (type(p)) {
      case hlist_node: case vlist_node: case unset_node:
        if (is_template(list_ptr(p))) return true;
        break;
      case adjust_node:
        if (is_template(adjust_ptr(p))) return true;
        break;
      case ins_node: return true;
      case whatsit_node:
        switch (subtype(p)) {
        case hset_node: case vset_node: case hpack_node: case vpack_node:
          return true;
        case align_node:
          if (is_template(align_list(p))) return true;
          break;
```

```
      default: break;
      }
    default: break;
    }
    p = link(p);
  }
  return false;
}
static void hprint_template_list(pointer p);
static void hprint_template(pointer p, int e, bool fixed, scaled d, scaled
    s, bool vertical, int m)
                          /* print a vertical or horizontal template with list p */
{ hprintf("<%ctemplate␣", vertical ? 'v' : 'h');
  hprint_bits(fixed, vertical ∧ d ≠ max_dimen, s ≠ 0);
  if (fixed) { hprintf("␣(additional)␣%d␣(fixed)␣", m); hprint_scaled(e);
  }
  else hprintf("␣(extent)␣%d", e);
  if (vertical ∧ d ≠ max_dimen) { hprintf("␣(depth␣limit)␣");
    hprint_scaled(d);
  }
  if (s ≠ 0) { hprintf("␣(shifted)␣"); hprint_scaled(s);
  }
  hprint_template_list(p); hprintf(">");
}
static void hprint_template_list(pointer p)
                                        /* print the node list p as a template */
{
  if (p ≡ null) { hprintf("<>"); return;
  }
  nesting++; hprint_nesting(); hprintf("<"); goto first_line;
  while (p ≠ null) { hprint_nesting();
  first_line:
    switch (type(p)) {
    case hlist_node:
      if (is_template(list_ptr(p))) hprint_template(list_ptr(p), width(p), true,
          max_dimen, shift_amount(p), false, exactly);
      else hout_node(p);
      break;
    case vlist_node:
      if (is_template(list_ptr(p))) hprint_template(list_ptr(p), height(p), true,
          max_dimen, shift_amount(p), true, exactly);
      else hout_node(p);
      break;
    case unset_node:
```

```
        if (is_template(list_ptr(p)))
          QUIT("Template␣unset␣nodes␣not␣yet␣implemented");
        else hout_node(p);
        break;
      case ins_node: hprintf("<stream␣[000]␣%d>", get_stream(subtype(p)));
                                                  /* get the stream number */
        break;
      case whatsit_node:
        switch (subtype(p)) {
        case hset_node: hprint_template(list_ptr(p), set_extent(p), false,
            max_dimen, shift_amount(p), false, exactly); break;
        case vset_node: hprint_template(list_ptr(p), set_extent(p), false,
            max_dimen, shift_amount(p), true, exactly); break;
        case hpack_node: hprint_template(list_ptr(p), pack_extent(p), false,
            pack_limit(p), shift_amount(p), false, pack_m(p)); break;
        case vpack_node: hprint_template(list_ptr(p), pack_extent(p), false,
            pack_limit(p), shift_amount(p), true, pack_m(p)); break;
        default: hprint_nesting( ); hout_node(p); break;
        }
        break;
      default: hprint_nesting( ); hout_node(p); break;
      }
      p = link(p);
    }
    nesting −−; hprintf(">");
  }
#endif
#if 0
  static void hprint_pages(void)
  { int k;
    for (k = 1; k < pages_used; k++) { hprint_nesting( ); hprintf("<pagedef");
      hprint_bits(0, 0, 0); hprintf("%d␣", k);
      hprint_template_list(list_ptr(pages_defined[k])); hprintf(">");
    }
  }
#endif
```

Appendix

A Source Files

A.1 Basic types

To define basic types we use the file `bastypes.h` from [12].

A.2 HiTEX routines: `hitex.c`

(107)

```
#include <stdio.h>
#include "basetypes.h"
#include "error.h"
#include "hformat.h"
#include "textypes.h"
#include "texvars.h"
#include "texfuncs.h"
#include "hput.h"
#include "hitex.h"
  extern scaled hhsize, hvsize;
  extern bool option_compress;
  extern bool option_global;
  ⟨ HiTEX macros 26 ⟩
  ⟨ HiTEX variables 27 ⟩
  ⟨ HiTEX auxiliar routines 11 ⟩
  ⟨ HiTEX routines 2 ⟩
```

A.3 HiTEX prototypes: hitex.h

⟨ hitex.h 108 ⟩ ≡ (108)

```
extern void hint_open(void);
extern void hint_close(void);
extern void hout_node(pointer p);
extern void hfix_defaults(void);
extern int hget_xdimen_no(dimen_tw, scaled h, scaled v);
extern void hget_image_information(pointer p);
static void hout_string(int s);
static uint8_t hout_hyphen(pointer p);
static uint8_t hout_glue_spec(pointer p);
static void hout_glue_node(pointer p);
static int hget_baseline_no(pointer bs, pointer ls, scaled lsl);
static int get_page(pointer p);
static int hget_glue_no(pointer p);
static uint8_t hout_list(pointer p, uint32_t pos, list_t *l);
static int hout_param_list(pointer p, uint32_t pos, list_t *l);
static void hout_list_node(pointer p, uint32_t pos, list_t *l);
static void hdef_init(void);
static void hput_definitions( );
```

A.4 TEX variables: texvars.h

To be able to use the global variables of TEX, we list them in a header file.

⟨ texvars.h 109 ⟩ ≡ (109)

```
#ifndef _TEX_VARS_H
#define _TEX_VARS_H_
  extern int bad;
  extern ASCII_code xord[256];
  extern uint8_t xchr[256];
  extern uint8_t name_of_file0[file_name_size + 1], *const name_of_file;
  extern uint8_t name_length;
  extern ASCII_code buffer[buf_size + 1];
  extern uint16_t first;
  extern uint16_t last;
  extern uint16_t max_buf_stack;
  extern alpha_file term_in;
  extern alpha_file term_out;
  extern packed_ASCII_code str_pool[pool_size + 1];
  extern const char *pool_name;
  extern pool_pointer str_start[max_strings + 1];
  extern pool_pointer pool_ptr;
  extern str_number str_ptr;
  extern pool_pointer init_pool_ptr;
  extern str_number init_str_ptr;
#ifdef INIT
```

```
  extern alpha_file pool_file;
#endif
  extern alpha_file log_file;
  extern uint8_t selector;
  extern uint8_t dig[23];
  extern int tally;
  extern uint8_t term_offset;
  extern uint8_t file_offset;
  extern ASCII_code trick_buf[error_line + 1];
  extern int trick_count;
  extern int first_count;
  extern uint8_t interaction;
  extern bool deletions_allowed;
  extern bool set_box_allowed;
  extern uint8_t history;
  extern int8_t error_count;
  extern char *help_line[6];
  extern uint8_t help_ptr;
  extern bool use_err_help;
  extern int interrupt;
  extern bool OK_to_interrupt;
  extern bool arith_error;
  extern scaled rem;
  extern pointer temp_ptr;
  extern memory_word mem0[mem_max - mem_min + 1], *const mem;
  extern pointer lo_mem_max;
  extern pointer hi_mem_min;
  extern int var_used, dyn_used;
#ifdef DEBUG
#define incr_dyn_used incr (dyn_used)
#define decr_dyn_used decr (dyn_used)
#else
#define incr_dyn_used
#define decr_dyn_used
#endif
  extern pointer avail;
  extern pointer mem_end;
  extern pointer rover;
#ifdef DEBUG
  extern bool is_free0[mem_max - mem_min + 1], *const is_free;
  extern bool was_free0[mem_max - mem_min + 1], *const was_free;
  extern pointer was_mem_end, was_lo_max, was_hi_min;
  extern bool panicking;
#endif
  extern int font_in_short_display;
  extern int depth_threshold;
```

```
extern int breadth_max;
extern list_state_record nest[nest_size + 1];
extern uint8_t nest_ptr;
extern uint8_t max_nest_stack;
extern list_state_record cur_list;
extern int16_t shown_mode;
extern uint8_t old_setting;
extern memory_word eqtb0[eqtb_size − active_base + 1], *const eqtb;
extern memory_word hfactor_eqtb0[dimen_pars + 256], *const hfactor_eqtb;
extern memory_word vfactor_eqtb0[dimen_pars + 256], *const vfactor_eqtb;
extern quarterword xeq_level0[eqtb_size − int_base + 1], *const xeq_level;
extern two_halves hash0[undefined_control_sequence − hash_base], *const hash;
extern pointer hash_used;
extern bool no_new_control_sequence;
extern int cs_count;
extern memory_word save_stack[save_size + 1];
extern memory_word save_hfactor[save_size + 1];
extern memory_word save_vfactor[save_size + 1];
extern uint16_t save_ptr;
extern uint16_t max_save_stack;
extern quarterword cur_level;
extern group_code cur_group;
extern uint16_t cur_boundary;
extern int mag_set;
extern eight_bits cur_cmd;
extern halfword cur_chr;
extern pointer cur_cs;
extern halfword cur_tok;
extern in_state_record input_stack[stack_size + 1];
extern uint8_t input_ptr;
extern uint8_t max_in_stack;
extern in_state_record cur_input;
extern uint8_t in_open;
extern uint8_t open_parens;
extern alpha_file input_file0[max_in_open], *const input_file; extern int line ;
extern int line_stack0[max_in_open], *const line_stack;
extern uint8_t scanner_status;
extern pointer warning_index;
extern pointer def_ref;
extern pointer param_stack[param_size + 1];
extern uint8_t param_ptr;
extern int max_param_stack;
extern int align_state;
extern uint8_t base_ptr;
extern pointer par_loc;
```

```
extern halfword par_token;
extern bool force_eof;
extern pointer cur_mark0[split_bot_mark_code − top_mark_code + 1],
    ∗const cur_mark;
extern uint8_t long_state;
extern pointer pstack[9];
extern int cur_val, cur_hfactor, cur_vfactor;
extern uint8_t cur_val_level;
extern small_number radix;
extern glue_ord cur_order;
extern alpha_file read_file[16];
extern uint8_t read_open[17];
extern pointer cond_ptr;
extern uint8_t if_limit;
extern small_number cur_if;
extern int if_line;
extern int skip_line;
extern str_number cur_name;
extern str_number cur_area;
extern str_number cur_ext;
extern pool_pointer area_delimiter;
extern pool_pointer ext_delimiter;
extern ASCII_code TEX_format_default[1 + format_default_length + 1];
extern bool name_in_progress;
extern str_number job_name;
extern bool log_opened;
extern byte_file dvi_file;
extern str_number output_file_name;
extern str_number log_name;
extern byte_file tfm_file;
extern memory_word font_info[font_mem_size + 1];
extern font_index fmem_ptr;
extern internal_font_number font_ptr;
extern four_quarters font_check0[font_max − font_base + 1],
    ∗const font_check;
extern scaled font_size0[font_max − font_base + 1], ∗const font_size;
extern scaled font_dsize0[font_max − font_base + 1], ∗const font_dsize;
extern font_index font_params0[font_max − font_base + 1], ∗const font_params;
extern str_number font_name0[font_max − font_base + 1], ∗const font_name;
extern str_number font_area0[font_max − font_base + 1], ∗const font_area;
extern eight_bits font_bc0[font_max − font_base + 1], ∗const font_bc;
extern eight_bits font_ec0[font_max − font_base + 1], ∗const font_ec;
extern pointer font_glue0[font_max − font_base + 1], ∗const font_glue;
extern bool font_used0[font_max − font_base + 1], ∗const font_used;
extern int hyphen_char0[font_max − font_base + 1], ∗const hyphen_char;
extern int skew_char0[font_max − font_base + 1], ∗const skew_char;
```

```
extern font_index bchar_label0[font_max − font_base + 1], ∗const bchar_label;
extern uint16_t font_bchar0[font_max − font_base + 1], ∗const font_bchar;
extern uint16_t font_false_bchar0[font_max − font_base + 1],
    ∗const font_false_bchar;
extern int char_base0[font_max − font_base + 1], ∗const char_base;
extern int width_base0[font_max − font_base + 1], ∗const width_base;
extern int height_base0[font_max − font_base + 1], ∗const height_base;
extern int depth_base0[font_max − font_base + 1], ∗const depth_base;
extern int italic_base0[font_max − font_base + 1], ∗const italic_base;
extern int lig_kern_base0[font_max − font_base + 1], ∗const lig_kern_base;
extern int kern_base0[font_max − font_base + 1], ∗const kern_base;
extern int exten_base0[font_max − font_base + 1], ∗const exten_base;
extern int param_base0[font_max − font_base + 1], ∗const param_base;
extern four_quarters null_character;
extern int total_pages;
extern scaled max_v;
extern scaled max_h;
extern int max_push;
extern int last_bop;
extern int dead_cycles;
extern bool doing_leaders;
extern quarterword c, f;
extern scaled rule_ht, rule_dp, rule_wd;
extern pointer g;
extern int lq, lr;
extern eight_bits dvi_buf[dvi_buf_size + 1];
extern dvi_index half_buf;
extern dvi_index dvi_limit;
extern dvi_index dvi_ptr;
extern int dvi_offset;
extern int dvi_gone;
extern pointer down_ptr, right_ptr;
extern scaled dvi_h, dvi_v;
extern scaled cur_h, cur_v;
extern internal_font_number dvi_f;
extern int cur_s;
extern scaled total_stretch0[filll − normal + 1], ∗const total_stretch;
extern scaled total_shrink0[filll − normal + 1], ∗const total_shrink;
extern int last_badness;
extern pointer adjust_tail;
extern int pack_begin_line;
extern two_halves empty_field;
extern four_quarters null_delimiter;
extern pointer cur_mlist;
extern small_number cur_style;
extern small_number cur_size;
```

```
extern scaled cur_mu;
extern bool mlist_penalties;
extern internal_font_number cur_f;
extern quarterword cur_c;
extern four_quarters cur_i;
extern int magic_offset;
extern pointer cur_align;
extern pointer cur_span;
extern pointer cur_loop;
extern pointer align_ptr;
extern pointer cur_head, cur_tail;
extern pointer just_box;
extern pointer passive;
extern pointer printed_node;
extern halfword pass_number;
extern scaled active_width0[6], *const active_width;
extern scaled cur_active_width0[6], *const cur_active_width;
extern scaled background0[6], *const background;
extern scaled break_width0[6], *const break_width;
extern bool no_shrink_error_yet;
extern pointer cur_p;
extern bool second_pass;
extern bool final_pass;
extern int threshold;
extern int minimal_demerits0[tight_fit - very_loose_fit + 1],
    *const minimal_demerits;
extern int minimum_demerits;
extern pointer best_place0[tight_fit - very_loose_fit + 1], *const best_place;
extern halfword best_pl_line0[tight_fit - very_loose_fit + 1], *const best_pl_line;
extern scaled disc_width;
extern halfword easy_line;
extern halfword last_special_line;
extern scaled first_width;
extern scaled second_width;
extern scaled first_indent;
extern scaled second_indent;
extern pointer best_bet;
extern int fewest_demerits;
extern halfword best_line;
extern int actual_looseness;
extern int line_diff;
extern uint16_t hc[66];
extern small_number hn;
extern pointer ha, hb;
extern internal_font_number hf;
extern uint16_t hu[64];
```

```
  extern int hyf_char;
  extern ASCII_code cur_lang, init_cur_lang;
  extern int l_hyf, r_hyf, init_l_hyf, init_r_hyf;
  extern halfword hyf_bchar;
  extern uint8_t hyf[65];
  extern pointer init_list;
  extern bool init_lig;
  extern bool init_lft;
  extern small_number hyphen_passed;
  extern halfword cur_l, cur_r;
  extern pointer cur_q;
  extern pointer lig_stack;
  extern bool ligature_present;
  extern bool lft_hit, rt_hit;
  extern two_halves trie[trie_size + 1];
  extern small_number hyf_distance0[trie_op_size], *const hyf_distance;
  extern small_number hyf_num0[trie_op_size], *const hyf_num;
  extern quarterword hyf_next0[trie_op_size], *const hyf_next;
  extern uint16_t op_start[256];
  extern str_number hyph_word[hyph_size + 1];
  extern pointer hyph_list[hyph_size + 1];
  extern hyph_pointer hyph_count;
#ifdef INIT
  extern uint16_t trie_op_hash0[trie_op_size + trie_op_size + 1],
      *const trie_op_hash;
  extern quarterword trie_used[256];
  extern ASCII_code trie_op_lang0[trie_op_size], *const trie_op_lang;
  extern quarterword trie_op_val0[trie_op_size], *const trie_op_val;
  extern uint16_t trie_op_ptr;
  extern packed_ASCII_code trie_c[trie_size + 1];
  extern quarterword trie_o[trie_size + 1];
  extern trie_pointer trie_l[trie_size + 1];
  extern trie_pointer trie_r[trie_size + 1];
  extern trie_pointer trie_ptr;
  extern trie_pointer trie_hash[trie_size + 1];
  extern bool trie_taken0[trie_size], *const trie_taken;
  extern trie_pointer trie_min[256];
  extern trie_pointer trie_max;
  extern bool trie_not_ready;
#endif
  extern scaled best_height_plus_depth;
  extern pointer page_tail;
  extern uint8_t page_contents;
  extern scaled page_max_depth;
  extern pointer best_page_break;
  extern int least_page_cost;
```

```
  extern scaled best_size;
  extern scaled page_so_far[8];
  extern pointer last_glue;
  extern int last_penalty;
  extern scaled last_kern;
  extern int insert_penalties;
  extern bool output_active;
  extern internal_font_number main_f;
  extern four_quarters main_i;
  extern four_quarters main_j;
  extern font_index main_k;
  extern pointer main_p;
  extern int main_s;
  extern halfword bchar;
  extern halfword false_bchar;
  extern bool cancel_boundary;
  extern bool ins_disc;
  extern pointer cur_box;
  extern halfword after_token;
  extern bool long_help_seen;
  extern str_number format_ident;
  extern word_file fmt_file;
  extern int ready_already;
  extern alpha_file write_file[16];
  extern bool write_open[18];
  extern pointer write_loc;
#endif
```

A.5 TEX Functions: texfuncs.h

To be able to use the functions of TEX, we put the necessary function prototypes in a header file.

⟨ texfuncs.h 110 ⟩ ≡ (110)

```
#ifndef _TEX_FUNCS_H_
#define _TEX_FUNCS_H_
  extern void input_add_arg(char *str);
  extern bool input_ln(alpha_file * f, bool bypass_eoln);
  extern bool init_terminal(void);
  extern str_number make_string(void);
  extern bool str_eq_buf(str_number s, int k);
  extern bool str_eq_str(str_number s, str_number t);
  extern void print_two(int n);
  extern void print_hex(int n);
  extern void print_roman_int(int n);
  extern void print_current_string(void);
  extern void term_input(void);
```

```
extern void int_error(int n);
extern void normalize_selector(void);
extern void pause_for_instructions(void);
extern int half(int x);
extern scaled round_decimals(small_number k);
extern void print_scaled(scaled s);
extern scaled mult_and_add(int n, scaled x, scaled y, scaled max_answer);
extern scaled x_over_n(scaled x, int n);
extern scaled xn_over_d(scaled x, int n, int d);
extern halfword badness(scaled t, scaled s);
extern pointer get_avail(void);
extern void flush_list(pointer p);
extern pointer get_node(int s);
extern void free_node(pointer p, halfword s);
extern pointer new_null_box(void);
extern pointer new_rule(void);
extern pointer new_ligature(quarterword f, quarterword c, pointer q);
extern pointer new_disc(void);
extern pointer new_math(scaled w, small_number s);
extern pointer new_spec(pointer p);
extern pointer new_param_glue(small_number n);
extern pointer new_glue(pointer q);
extern pointer new_skip_param(small_number n);
extern pointer new_kern(scaled w);
extern pointer new_penalty(int m);
extern void short_display(int p);
extern void print_font_and_char(int p);
extern void print_glue(scaled d, int order, str_number s);
extern void print_spec(int p, str_number s);
extern void show_node_list(int p);
extern void show_box(pointer p);
extern void delete_token_ref(pointer p);
extern void delete_glue_ref(pointer p);
extern void flush_node_list(pointer p);
extern pointer copy_node_list(pointer p);
extern void print_mode(int m);
extern void push_nest(void);
extern void pop_nest(void);
extern void print_totals(void);
;
extern void print_param(int n);
extern void fix_date_and_time(void);
extern void begin_diagnostic(void);
extern void print_length_param(int n);
extern pointer id_lookup(int j, int l);
```

```
extern void new_save_level(group_code c);
extern void eq_destroy(memory_word w);
extern void eq_save(pointer p, quarterword l);
extern void eq_define(pointer p, quarterword t, halfword e);
extern void eq_word_define(pointer p, int w);
extern void geq_define(pointer p, quarterword t, halfword e);
extern void save_for_after(halfword t);
extern void prepare_mag(void);
extern void token_show(pointer p);
extern void print_meaning(void);
extern void show_cur_cmd_chr(void);
extern void show_context(void);
extern void begin_token_list(pointer p, quarterword t);
extern void end_token_list(void);
extern void back_input(void);
extern void back_error(void);
extern void begin_file_reading(void);
extern void end_file_reading(void);
extern void clear_for_error_prompt(void);
extern void check_outer_validity(void);
extern void firm_up_the_line(void);
extern void get_next(void);
extern void firm_up_the_line(void);
extern void get_token(void);
extern void get_x_token(void);
extern void x_token(void);
extern void scan_left_brace(void);
extern void scan_optional_equals(void);
extern bool scan_keyword(str_number s);
extern void mu_error(void);
extern void scan_int(void);
extern void scan_something_internal(small_number level, bool negative);
extern void scan_int(void);
extern void scan_dimen(bool mu, bool inf, bool shortcut);
extern void scan_glue(small_number level);
extern pointer scan_rule_spec(void);
extern pointer str_toks(pool_pointer b);
extern pointer the_toks(void);
extern void ins_the_toks(void);
extern void conv_toks(void);
extern pointer scan_toks(bool macro_def, bool xpand);
extern void read_toks(int n, pointer r);
extern void pass_text(void);
extern void change_if_limit(small_number l, pointer p);
extern void conditional(void);
extern void begin_name(void);
```

```
extern bool more_name(ASCII_code c);
extern void end_name(void);
extern void pack_file_name(str_number n, str_number a, str_number e);
extern void pack_buffered_name(small_number n, int a, int b);
extern str_number make_name_string(void);
extern void scan_file_name(void);
extern void pack_job_name(str_number s);
extern void prompt_file_name(char *s, str_number e);
extern void open_log_file(void);
extern void start_input(void);
extern internal_font_number read_font_info(pointer u, str_number nom,
    str_number aire, scaled s);
extern void char_warning(internal_font_number f, eight_bits c);
extern pointer new_character(internal_font_number f, eight_bits c);
extern void write_dvi(dvi_index a, dvi_index b);
extern void dvi_swap(void);
extern void dvi_four(int x);
extern void dvi_pop(int l);
extern void dvi_font_def(internal_font_number f);
extern void movement(scaled w, eight_bits o);
extern void prune_movements(int l);
extern void vlist_out(void);
extern void vlist_out(void);
extern void ship_out(pointer p);
extern void scan_spec(group_code c, bool three_codes);
extern pointer hpack(pointer p, scaled w, scaled hf, scaled
    vf, small_number m);
extern void append_to_vlist(pointer b);
extern pointer new_noad(void);
extern pointer new_style(small_number s);
extern pointer new_choice(void);
extern void show_info(void);
extern pointer fraction_rule(scaled t);
extern pointer overbar(pointer b, scaled k, scaled t);
extern pointer rebox(pointer b, scaled w);
extern pointer math_glue(pointer g, scaled m);
extern void math_kern(pointer p, scaled m);
extern void flush_math(void);
extern void mlist_to_hlist(void);
extern void fetch(pointer a);
extern void push_alignment(void);
extern void init_col(void);
extern bool fin_col(void);
extern void fin_row(void);
extern void do_assignments(void);
extern void new_hyph_exceptions(void);
```

```
  extern pointer prune_page_top(pointer p);
  extern pointer vert_break(pointer p, scaled h, scaled d);
  extern pointer vsplit(eight_bits n, scaled h);
  extern void print_totals(void);
  extern void freeze_page_specs(small_number s);
  extern void box_error(eight_bits n);
  extern void ensure_vbox(eight_bits n);
  extern void give_err_help(void);
  extern void initialize(void);
  extern void b_close(byte_file * f);
  extern void w_close(word_file * f);
  extern void print_ln(void);
  extern void print_char(ASCII_code s);
  extern void print(int s);
  extern void print_str(char *s);
  extern void slow_print(int s);
  extern void print_nl(char *s);
  extern void print_esc(str_number s);
  extern void print_the_digs(eight_bits k);
  extern void print_int(int n);
  extern void normalize_selector(void);
  extern void get_token(void);
  extern void term_input(void);
  extern void show_context(void);
  extern void begin_file_reading(void);
  extern void open_log_file(void);
  extern void close_files_and_terminate(void);
  extern void clear_for_error_prompt(void);
  extern void give_err_help(void);
#ifdef DEBUG
  extern void debug_help(void);
#else
#define debug_help ( ) do_nothing
#endif
  extern void jump_out(void); extern void
  error (void) ;
  extern void fatal_error(char *s);
  extern void overflow(char *s, int n);
  extern void confusion(str_number s);
  extern void print_word(memory_word w);
  extern void sort_avail(void);
  extern void check_mem(bool print_locs);
  extern void search_mem(pointer p);
  extern void print_mark(int p);
  extern void print_rule_dimen(scaled d);
```

```
extern void show_activities(void);
extern void print_skip_param(int n);
extern void end_diagnostic(bool blank_line);
extern void show_eqtb(pointer n);
extern void print_cs(int p);
extern void sprint_cs(pointer p);
extern void primitive(str_number s, quarterword c, halfword o);
extern void geq_word_define(pointer p, int w);
extern void back_input(void);
extern void unsave(void);
extern void restore_trace(pointer p, str_number s);
extern void show_token_list(int p, int q, int l);
extern void print_cmd_chr(quarterword cmd, halfword chr_code);
extern void runaway(void);
extern void ins_error(void);
extern void pass_text(void);
extern void start_input(void);
extern void conditional(void);
extern void get_x_token(void);
extern void conv_toks(void);
extern void ins_the_toks(void);
extern void expand(void);
extern void insert_relax(void);
extern void macro_call(void);
extern void scan_eight_bit_int(void);
extern void scan_char_num(void);
extern void scan_four_bit_int(void);
extern void scan_fifteen_bit_int(void);
extern void scan_twenty_seven_bit_int(void);
extern void print_file_name(int n, int a, int e);
extern void scan_font_ident(void);
extern void find_font_dimen(bool writing);
extern void hlist_out(void);
extern void print_fam_and_char(pointer p);
extern void print_delimiter(pointer p);
extern void show_info(void);
extern void print_subsidiary_data(pointer p, ASCII_code c);
extern void print_style(int c);
extern void print_size(int s);
extern void stack_into_box(pointer b, internal_font_number
    f, quarterword c);
extern void make_over(pointer q);
extern void make_under(pointer q);
extern void make_vcenter(pointer q);
extern void make_radical(pointer q);
extern void make_math_accent(pointer q);
```

```
extern void make_fraction(pointer q);
extern void make_ord(pointer q);
extern void make_scripts(pointer q, scaled delta);
extern void pop_alignment(void);
extern void align_peek(void);
extern void normal_paragraph(void);
extern void init_align(void);
extern void get_preamble_token(void);
extern void align_peek(void);
extern void init_row(void);
extern void init_span(pointer p);
extern void resume_after_display(void);
extern void fin_align(void);
extern void line_break(int final_widow_penalty);
extern void try_break(int pi, small_number break_type);
extern void post_line_break(int final_widow_penalty);
extern void hyphenate(void);
extern void first_fit(trie_pointer p);
extern void trie_pack(trie_pointer p);
extern void trie_fix(trie_pointer p);
extern void new_patterns(void);
extern void init_trie(void);
extern void build_page(void);
extern void fire_up(pointer c);
extern void main_control(void);
extern void app_space(void);
extern void insert_dollar_sign(void);
extern void you_cant(void);
extern void report_illegal_case(void);
extern void append_glue(void);
extern void append_kern(void);
extern void off_save(void);
extern void handle_right_brace(void);
extern void extra_right_brace(void);
extern void normal_paragraph(void);
extern void box_end(int box_context);
extern void begin_box(int box_context);
extern void scan_box(int box_context);
extern void package(small_number c);
extern void new_graf(bool indented);
extern void indent_in_hmode(void);
extern void head_for_vmode(void);
extern void end_graf(void);
extern void begin_insert_or_adjust(void);
extern void make_mark(void);
extern void append_penalty(void);
```

```
extern void delete_last(void);
extern void unpackage(void);
extern void append_italic_correction(void);
extern void append_discretionary(void);
extern void build_discretionary(void);
extern void make_accent(void);
extern void align_error(void);
extern void no_align_error(void);
extern void omit_error(void);
extern void do_endv(void);
extern void cs_error(void);
extern void push_math(group_code c);
extern void init_math(void);
extern void start_eq_no(void);
extern void scan_math(pointer p);
extern void set_math_char(int c);
extern void math_limit_switch(void);
extern void scan_delimiter(pointer p, bool r);
extern void math_radical(void);
extern void math_ac(void);
extern void append_choices(void);
extern void build_choices(void);
extern void sub_sup(void);
extern void math_fraction(void);
extern void math_left_right(void);
extern void after_math(void);
extern void resume_after_display(void);
extern void prefixed_command(void);
extern void get_r_token(void);
extern void trap_zero_glue(void);
extern void do_register_command(small_number a);
extern void alter_aux(void);
extern void alter_prev_graf(void);
extern void alter_page_so_far(void);
extern void alter_integer(void);
extern void alter_box_dimen(void);
extern void new_font(small_number a);
extern void new_interaction(void);
extern void do_assignments(void);
extern void open_or_close_in(void);
extern void issue_message(void);
extern void shift_case(void);
extern void show_whatever(void);
extern void store_fmt_file(void);
extern void final_cleanup(void);
extern void do_extension(void);
```

```
  extern void new_whatsit(small_number s, small_number w);
  extern void new_write_whatsit(small_number w);
  extern void print_write_whatsit(str_number s, pointer p);
  extern void special_out(pointer p);
  extern void write_out(pointer p);
  extern void out_what(pointer p);
  extern void fix_language(void);                    /* added manualy */
  extern bool open_fmt_file(void);
  extern bool load_fmt_file(void);
  extern bool its_all_over(void);
  extern bool privileged(void);
  extern bool w_open_out(word_file * f);
  extern bool b_open_out(byte_file * f);
  extern bool a_open_out(alpha_file * f);
  extern str_number w_make_name_string(word_file * f);
  extern str_number b_make_name_string(byte_file * f);
#define vpack(...) vpackage (__VA_ARGS__, max_dimen)
  extern pointer vpackage(pointer p, scaled h, scaled hf, scaled
      vf, small_number m, scaled l);
  extern pointer new_lig_item(quarterword c);
  extern small_number norm_min(int h);
  extern pointer finite_shrink(pointer p);           /* new procedures */
  extern void check_constants(void);
  extern void get_the_first_line(void);
  extern void fill_box_255(pointer c);
  extern void end_of_output(void);
  extern void open_dump_file(void);
  extern void close_dump_file(void);
  extern pointer find_space_glue(internal_font_number f);
  extern void hyphenate_word(void);
  extern void print_xdimen(int i);
  extern void print_baseline_skip(int i);
  extern void hline_break(int final_widow_penalty);
  extern pointer new_image_node(str_number n, str_number a, str_number e);
  extern pointer new_set_node(void);
  extern pointer new_pack_node(void);
  extern pointer new_baseline_node(pointer bs, pointer ls, scaled lsl);
  extern pointer new_disp_node(void);
  extern void add_par_node(uint8_t t, uint8_t n, int v);
  extern void display_node(pointer p);          /* generated by htex.ch */
  extern void hint_open(void);
  extern void hint_close(void);
  extern int hget_xdimen_no(scaled w, scaled h, scaled v);
  extern void hget_image_information(pointer p);
  extern bool ktex_init(int argc, char *argv[]);
  extern char *ktex_find_tfm(const char *filename);
```

```
  extern char *ktex_find_pk(const char *filename);
  extern int hset_option(char *c, char *d);
#endif
```

Crossreference of Code

⟨ Allocate a new directory entry ⟩ Defined in section 25. Used in section 28.
⟨ Create the parameter node ⟩ Defined in section 6. Used in section 8.
⟨ Find an existing directory entry ⟩ Defined in section 24. Used in section 28.
⟨ Fix definitions for dimension parameters ⟩ Defined in section 40. Used in section 29.
⟨ Fix definitions for glue parameters ⟩ Defined in section 50. Used in section 29.
⟨ Fix definitions for integer parameters ⟩ Defined in section 34. Used in section 29.
⟨ HiTEX auxiliar routines ⟩ Defined in section 11, 12, 13, 14, 15, 28, 51, 54, 58, 62, 71, 74, 80, 82, 84, 99, and 100. Used in section 107.
⟨ HiTEX macros ⟩ Defined in section 26, 36, and 75. Used in section 107.
⟨ HiTEX routines ⟩ Defined in section 2, 5, 8, 9, 10, 16, 18, 19, 20, 21, 22, 23, 29, 30, 31, 35, 41, 45, 47, 52, 57, 60, 63, 66, 67, 72, 76, 103, 104, 105, and 106. Used in section 107.
⟨ HiTEX variables ⟩ Defined in section 27, 32, 33, 38, 39, 43, 48, 49, 55, 61, 65, and 69. Used in section 107.
⟨ Initialize definitions for baseline skips ⟩ Defined in section 56. Used in section 30.
⟨ Initialize definitions for extended dimensions ⟩ Defined in section 44. Used in section 30.
⟨ Initialize definitions for fonts ⟩ Defined in section 70. Used in section 30.
⟨ Initialize the parameter node ⟩ Defined in section 7. Used in section 8.
⟨ Output baseline skip definitions ⟩ Defined in section 59. Used in section 31.
⟨ Output dimension definitions ⟩ Defined in section 42. Used in section 31.
⟨ Output extended dimension definitions ⟩ Defined in section 46. Used in section 31.
⟨ Output font definitions ⟩ Defined in section 73. Used in section 31.
⟨ Output glue definitions ⟩ Defined in section 53. Used in section 31.
⟨ Output hyphen definitions ⟩ Defined in section 64. Used in section 31.
⟨ Output integer definitions ⟩ Defined in section 37. Used in section 31.
⟨ Output parameter list definitions ⟩ Defined in section 68. Used in section 31.
⟨ Switch *hsize* and *vsize* to extended dimensions ⟩ Defined in section 17. Used in section 30.
⟨ cases to output content nodes ⟩ Defined in section 78, 79, 81, 83, 85, 86, 87, 88, 89, 90, 91, and 101. Used in section 76.
⟨ cases to output whatsit content nodes ⟩ Defined in section 92, 93, 94, 95, 96, 97, 98, and 102. Used in section 91.
⟨ explain usage ⟩ Defined in section 4. Used in section 2.
⟨ `hitex.h` ⟩ Defined in section 108.
⟨ `ktex.c` ⟩ Defined in section 1.

⟨ output a character node ⟩ Defined in section 77. Used in section 76.
⟨ `texfuncs.h` ⟩ Defined in section 110.
⟨ `texvars.h` ⟩ Defined in section 109.

References

[1] David Duce. Portable network graphics (png) specification (second edition). World Wide Web Consortium, Recommendation REC-PNG-20031110, November 2003.

[2] Julian Gilbey. The CTIE processor. Technical report, 2003.

[3] Klaus Guntermann. The TIE processor. Technical report, TH Darmstadt, Fachbereich Informatik, Institut für Theoretische Informatik, 1989.

[4] Eric Hamilton. Jpeg file interchange format. Technical report, C-Cube Microsystems, Milpitas, CA, USA, 9 1992.

[5] ITU. Jpeg iso/iec 10918-1 : 1993(e) ccit recommendation t.81, 1993.

[6] Donald E. Knuth. *The WEB system of structured documentation.* Stanford University, Computer Science Dept., Stanford, CA, 1983. STAN-CS-83-980. `https://ctan.org/pkg/cweb`.

[7] Donald E. Knuth. *TEX: The Program.* Computers & Typesetting, Volume B. Addison-Wesley, 1986.

[8] Donald E. Knuth. *Literate Programming.* CSLI Lecture Notes Number 27. Center for the Study of Language and Information, Stanford, CA, 1992.

[9] Donald E. Knuth and Silvio Levy. *The CWEB System of Structured Documentation.* Addison Wesley, 1994. `https://ctan.org/pkg/cweb`.

[10] Martin Ruckert. *WEB to cweb.* CreateSpace, 2017. ISBN 1-548-58234-4. `https://amazon.com/dp/1548582344`.

[11] Martin Ruckert. `web2w`*: Converting TEX from WEB to cweb.* `https://ctan.org/pkg/web2w`, 2017.

[12] Martin Ruckert. *HINT: The File Format.* CreateSpace, 2019. ISBN 0-000-00000-0. `https://amazon.com/dp/0000000`.

Index