HiTEX: Reflowable Output for TEX

HiTEX

Reflowable Output for TEX

Für Beatriz

Version 1.1 (Draft)

MARTIN RUCKERT *Munich University of Applied Sciences*

Date: 2020-09-24 18:10:56 +0200 (Thu, 24 Sep 2020) , Revision: 2059

The author has taken care in the preparation of this book, but makes no expressed or implied warranty of any kind and assumes no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

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

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

Copyright © 2018 by Martin Ruckert

All rights reserved. Printed using CreateSpace. This publication is protected by copyright, and permission must be obtained from the publisher prior to any prohibited reproduction, storage in a retrieval system, or transmission in any form or by any means, electronic, mechanical, photocopying, recording, or likewise. To obtain permission to use material from this work, please submit a written request to Martin Ruckert, Hochschule München, Fakultät für Informatik und Mathematik, Lothstrasse 64, 80335 München, Germany.

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 [14], 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[10] using "The CWEB System of Structured Documentation"[11]. The largest part of this program is, of course, TEX[9] 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"[7]. 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 [12] and [13].

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[4][3]. 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 file 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 "textypes.h"
#include "texfuncs.h"
```

2 TEX Live

HiTEX aspires to become a member of the TEX Live family of programs. To reach this goal, three major accomplishments are necessary:

HiTEX must

- respect the TEX Live conventions for command line parameters,
- find its input files using the `kpathsearch` library, and
- implement TEX primitives to support LATEX.

Therefore in a first step, we implement changes to TEX and define a collection of supporting functions to add the necessary functionality to TEX. The result is the `cinitex` program. It is supposed to be compatible with TEX Live, but it is still a regular TEX engine producing ordinary `DVI` files as output.

Naturaly, the functions that follow now are taken, with small modifications, from the TEX Live sources[1]. What is added here, or rather subtracted here, are the parts that are specific to some of the other TEX engines included in TEX Live. New is also that the code is presented in literate programming style.

Most modifications that turn TEX into HiTEX will have to wait until section 3, but a few HiTEX specific snippets are already found in this section and are bracketed with an "# **ifdef** `HITEX` ... # **endif**".

2.1 Command Line Parameters

To get an idea how a command line is structured, we first look at the help text that is displayed if the user asks for it (or if HiTEX decides that the user needs it). The help text is produced by the function *usage_help*.

```
⟨ TEX Live support functions 1 ⟩ ≡                                    (4)
  static void usage_help(void)
  { ⟨ explain command line 2 ⟩
    ⟨ explain options 3 ⟩
    fprintf(stderr, "\nEmail bug reports to ruckert@cs.hm.edu.\n");
    exit(0);
  }                                                                  Used in 34.
```

The command line commes in three slightly different versions:

```
⟨ explain command line 2 ⟩ ≡                                          (2)
  fprintf(stderr,
  "Usage:  %s [OPTION]...  [TEXNAME[.tex]] [COMMANDS]\n"
```

```
  "   or:  %s [OPTION]...  \\FIRST-LINE\n"
  "   or:  %s [OPTION]...  &FMT ARGS\n\n",
  argv[0], argv[0], argv[0]);
  fprintf(stderr,
  "  Run TeX on TEXNAME. Any remaining COMMANDS are processed\n"
  "  as TeX input after TEXNAME is read.\n"
  "  If the first line of TEXNAME starts with %%&FMT, and FMT is\n"
  "  an existing .fmt file, use it.  Else use 'NAME.fmt', where\n"
  "  NAME is the program invocation name.\n"
  "\n"
  "  Alternatively, if the first non-option argument begins\n"
  "  with a backslash, interpret all non-option arguments as\n"
  "  a line of TeX input.\n"
  "\n"
  "  Alternatively, if the first non-option argument begins\n"
  "  with a &, the next word is taken as the FMT to read,\n"
  "  overriding all else.    Any remaining arguments are\n"
  "  processed as above.\n"
  "\n"
  "  If no arguments or options are specified, prompt for input.\n"
  "\n");
```

Used in 1.

Next commes a list of possible options and their explanation:

⟨ explain options 3 ⟩ ≡ (3)

```
  fprintf(stderr,"Options:\n"
  " -help                 "
      "\t display this help and exit\n"
  " -version              "
      "\t output version information and exit\n"
  " -ini                  "
      "\t be initex for dumping formats;  this is\n"
      "\t\t\t also true if the program name is '(h|c)initex'\n"
  " -progname=STRING      "
      "\t set program (and fmt) name to STRING\n"
  " -fmt=FMTNAME          "
      "\t use FMTNAME instead of program name or a %%& line\n"
  " -output-directory=DIR "
      "\t use DIR as the directory to write files to\n"
  " -jobname=STRING       "
      "\t set the TeX \\jobname to STRING\n"
  " [-no]-mktex=FMT       "
      "\t disable/enable mktexFMT generation (FMT=tex/tfm/pk)\n"
  " -interaction=STRING   "
      "\t set interaction mode (STRING=batchmode/\n"
      "\t\t\t nonstopmode/scrollmode/errorstopmode)\n"
  " -kpathsea-debug=NUMBER"
```

```
      "\t set path searching debugging flags according\n"
      "\t\t\t to the bits of NUMBER\n"
  " [-no]-parse-first-line"
      "\t disable/enable parsing of the first line of\n"
      "\t\t\t the input file\n"
  " [-no]-file-line-error-style\n"
  " [-no]-file-line-error"
      "\t Disable/Enable file:line:error style\n"
#ifdef HITEX
  " -compress             "
      "\t enable compression of section 1 and 2\n"
  " -no-empty-page        "
      "\t suppress empty pages\n"
  " -hyphenate-first-word "
      "\t hyphenate the first word of a paragraph\n"
  " -resolution=NUMBER    "
      "\t set the resolution to NUMBER dpi\n"
  " -mfmode=MODE          "
      "\t set the METAFONT mode to MODE\n"
#endif
  );
```

Used in 1.

If the program was compiled with debugging enabled there is an additional debug option:

⟨ explain options 3 ⟩ +≡ (4)

```
#ifdef DEBUG
#ifdef HITEX
  fprintf(stderr,
  " -debug=XX             "
      "\t XX is a hexadecimal value.  OR together these values:\n");
  fprintf(stderr,"\t\t\t XX=%04X \t basic debugging\n",DBGBASIC);
  fprintf(stderr,"\t\t\t XX=%04X \t tag debugging\n",DBGTAGS);
  fprintf(stderr,"\t\t\t XX=%04X \t node debugging\n",DBGNODE);
  fprintf(stderr,"\t\t\t XX=%04X \t definition debugging\n",DBGDEF);
  fprintf(stderr,"\t\t\t XX=%04X \t directory debugging\n",DBGDIR);
  fprintf(stderr,"\t\t\t XX=%04X \t range debugging\n",DBGRANGE);
  fprintf(stderr,"\t\t\t XX=%04X \t float debugging\n",DBGFLOAT);
  fprintf(stderr,"\t\t\t XX=%04X \t compression debugging\n",
      DBGCOMPRESS);
  fprintf(stderr,"\t\t\t XX=%04X \t buffer debugging\n",DBGBUFFER);
  fprintf(stderr,"\t\t\t XX=%04X \t TeX debugging\n",DBGTEX);
  fprintf(stderr,"\t\t\t XX=%04X \t page debugging\n",DBGPAGE);
  fprintf(stderr,"\t\t\t XX=%04X \t font debugging\n",DBGFONT);
#endif
#endif
```

Special care is needed if option arguments are quoted and contain spaces. The function *normalize_quotes* makes shure that arguments containing spaces get quotes around them and it checks for unballanced quotes.

⟨ TEX Live support functions 1 ⟩ +≡ (5)

```
static char *normalize_quotes(const char *name, const char *mesg)
{ bool quoted = false;
  bool must_quote = (strchr(name, ' ') ≠ NULL);
  char *ret = xmalloc(strlen(name) + 3); /* room for two quotes and NUL */
  char *p = ret;
  const char *q;
  if (must_quote) *p++ = '"';
  for (q = name; *q; q++)
    if (*q ≡ '"') quoted = ¬quoted; else *p++ = *q;
  if (must_quote) *p++ = '"';
  *p = '\0';
  if (quoted) {
    fprintf(stderr, "! Unbalanced quotes in %s %s\n", mesg, name);
    exit(1);
  }
  return ret;
}
```

Parsing the command line options is accomplished with the *parse_options* function which in turn uses the *getopt_long_only* function from the C library. This function returns 0 and sets the *option_index* parameter to the option found, or it returns −1 if the end of all options is reached.

⟨ TEX Live support functions 1 ⟩ +≡ (6)

```
static void parse_options(int argc, char *argv[])
{
  while (true) { int option_index;
    int g = getopt_long_only(argc, argv, "+", long_options, &option_index);
    if (g ≡ 0) { ⟨ handle the option at option_index 9 ⟩ }
    else if (g ≡ −1) return;
  }
}
```

The following variables may be set using the options. For flags, we use −1 for an undefined value, 0 or *false* for false, and 1 or *true* for true.

⟨ TEX Live variables 7 ⟩ ≡ (7)

```
int iniversion = 0;
static int parsefirstlinep = −1;
int filelineerrorstylep = −1;
static const char *user_progname = NULL, *output_directory = NULL,
    *c_job_name = NULL;
char *dump_name = NULL;
```

```
#ifdef HITEX
  int option_no_empty_page = false, option_hyphen_first = false;
  int option_dpi = 600;
  const char *option_mfmode = "ljfour", *option_dpi_str = "600";
#endif
```
Used in 34.

In addition to these variables, the *interaction* variable of TEX as well as the *option_compress* and *debugflags* variable of HINT can be set from the commandline.

For all options, the *long_options* array contains the name, whether the option takes a parameter, the (optional) address of a flag, and the value to store in the flag variable or return.

```
⟨ TEX Live variables 7 ⟩ +≡                                              (8)
  static struct option long_options[] = {
  {"help",0,0,0},
  {"version",0,0,0},
  {"ini",0,&iniversion,1},
  {"progname",1,0,0},
  {"fmt",1,0,0},
  {"output-directory",1,0,0},
  {"jobname",1,0,0},
  {"mktex",1,0,0},
  {"no-mktex",1,0,0},
  {"interaction",1,0,0},
  {"kpathsea-debug",1,0,0},
  {"parse-first-line",0,&parsefirstlinep,1},
  {"no-parse-first-line",0,&parsefirstlinep,0},
  {"file-line-error-style",0,&filelineerrorstylep,1},
      {"no-file-line-error-style",0,&filelineerrorstylep,0},
      {"file-line-error",0,&filelineerrorstylep,1},{"no-file-line-error",
      0,&filelineerrorstylep,0},
#ifdef HITEX
  {"compress",0,&option_compress,1},
  {"no-empty-page",0,&option_no_empty_page,1},
  {"hyphenate-first-word",0,&option_hyphen_first,1},
  {"resolution",1,0,0},
  {"mfmode",1,0,0},
#ifdef DEBUG
  {"debug",1,0,0},
#endif
#endif
  {0,0,0,0}};
```

To handle the options, we compare the name at the given *option_index* with the different option names. This is not a very efficient method, but the impact is low and it's simple to write.

We start with an auxiliar macro using `STREQ` from the `kpathsearch` library and two simple options:

⟨ handle the option at *option_index* 9 ⟩ ≡ (9)
```
#define ARGUMENT_IS (X) STREQ (long_options[option_index].name, X)
  if (ARGUMENT_IS("help")) usage_help( );
  else if (ARGUMENT_IS("version")) { printf("Version␣"HITEX_VERSION"\n");
    exit(0);
  }
```
Used in 6.

Next some options that take a string as argument:

⟨ handle the option at *option_index* 9 ⟩ +≡ (10)
```
  else if (ARGUMENT_IS("progname")) user_progname = optarg;
  else if (ARGUMENT_IS("fmt")) dump_name = optarg;
  else if (ARGUMENT_IS("output-directory")) output_directory = optarg;
  else if (ARGUMENT_IS("jobname"))
    c_job_name = normalize_quotes(optarg, "jobname");
```

The *c_job_name* is used in the *get_job_name* function, to overwrite a given job name when setting TEX's *job_name* variable.

⟨ TEX Live functions 11 ⟩ ≡ (11)
```
  int get_job_name(int s)
  {
    if (c_job_name ≠ NULL) s = make_tex_string(c_job_name);
    return s;
  }
```
Used in 34.

The next two options simply pass the option argument to a function from the `kpathsearch` library.

⟨ handle the option at *option_index* 9 ⟩ +≡ (12)
```
  else if (ARGUMENT_IS("mktex")) kpse_maketex_option(optarg, true);
  else if (ARGUMENT_IS("no-mktex")) kpse_maketex_option(optarg, false);
```

The "interaction" option needs to set TEX's *interaction* variable. While TEX defines macros for the values, using the corresponding include file might produce name conflicts. So the following code uses the numeric values directly.

⟨ handle the option at *option_index* 9 ⟩ +≡ (13)
```
  else if (ARGUMENT_IS("interaction")) {
    if (STREQ(optarg, "batchmode")) interaction = 0;
    else if (STREQ(optarg, "nonstopmode")) interaction = 1;
    else if (STREQ(optarg, "scrollmode")) interaction = 2;
    else if (STREQ(optarg, "errorstopmode")) interaction = 3;
    else WARNING1("Ignoring␣unknown␣argument␣`%s'␣to␣--interaction",
          optarg);
  }
```

To debug the searching done by the `kpathsearch` library you can use the following option; the value 3 is a good choice to start with.

⟨ handle the option at *option_index* 9 ⟩ +≡ (14)
```
  else if (ARGUMENT_IS("kpathsea-debug")) kpathsea_debug |= atoi(optarg);
```

We conclude with the HiTEX specific options:

⟨ handle the option at *option_index* 9 ⟩ +≡ (15)

```
#ifdef HITEX
  else if (ARGUMENT_IS("resolution")) { option_dpi_str = optarg;
      option_dpi = strtol(option_dpi_str, NULL, 10);
  }
  else if (ARGUMENT_IS("mfmode")) option_mfmode = optarg;
#ifdef DEBUG
  else if (ARGUMENT_IS("debug")) debugflags = strtol(optarg, NULL, 16);
#endif
#endif
```

Parsing the command line options is done in the function *tl_maininit* which is the first function called in the main program of a TEX engine belonging to TEX Live. Before doing so, we make copies of argument count and argument vector.

⟨ TEX Live variables 7 ⟩ +≡ (16)

```
  static char **argv;
  static int argc;
```

⟨ TEX Live functions 11 ⟩ +≡ (17)

```
  void tl_maininit(int ac, char *av[])
  { char *main_input_file;
    argc = ac; argv = av; interaction = 3; /* error_stop_mode */
    ⟨ parse options 18 ⟩
    ⟨ set the program name 19 ⟩
    xputenv("engine", "hitex");
    ⟨ set the input file name 21 ⟩
    ⟨ set defaults from the texmf.cfg file 22 ⟩
    ⟨ set the format name 24 ⟩
    ⟨ enable the generation of input files 26 ⟩
  }
```

Before we can call the *parse_options* function, we might need some special preparations for Windows.

⟨ parse options 18 ⟩ ≡ (18)

```
#if defined (WIN32)
  { char *enc;
    kpse_set_program_name(argv[0], NULL);
    enc = kpse_var_value("command_line_encoding");
    get_command_line_args_utf8(enc, &argc, &argv);
    parse_options(argc, argv);
  }
#else
  parse_options(ac, av);
#endif
```

Used in 17.

With some luck, we can now inform the `kpathsearch` library about the program name.

```
⟨ set the program name 19 ⟩ ≡                                    (19)
  if (¬user_progname) user_progname = dump_name;
#if defined (WIN32)
  if (user_progname) kpse_reset_program_name(user_progname);
#else
  kpse_set_program_name(argv[0], user_progname);
#endif                                                  Used in 17.
```

Getting a hold of the input file name is more complicated. We look at the first argument after the options: If it does not start with an "`&`" and neither with a "\", it's a simple file name. Under Windows, however, filenames might start with a drive letter followed by a colon and a "\" which is used to separate directory names. Finaly, if the filename is a quoted string, we need to remove the quotes before we use the `kpathsearch` library to find it and reattach the quotes afterwards.

```
⟨ TEX Live support functions 1 ⟩ +≡                              (20)
#ifdef WIN32
  static void clean_windows_filename(char *filename)
  {
    if (strlen(filename) > 2 ∧ isalpha(filename[0]) ∧ filename[1] ≡
          ':' ∧ filename[2] ≡ '\\') { char *pp;
      for (pp = filename; *pp; pp++)
        if (*pp ≡ '\\') *pp = '/';
    }
  }
#endif
  static char *tl_find_file(char *fname, kpse_file_format_type t, bool mx)
  { char *filename;
    int final_quote = strlen(fname) − 1;
    bool quoted = final_quote > 1 ∧ fname[0] ≡ '"' ∧ fname[final_quote] ≡ '"';
    if (quoted) {                  /* Overwrite last quote and skip first quote. */
      fname[final_quote] = '\0'; fname++;
    }
    filename = kpse_find_file(fname, t, mx);
    if (quoted) {                                          /* Undo modifications */
      fname−−; fname[final_quote] = '"';
    }
    return filename;
  }
  static char *get_input_file_name(void)
  { char *input_file_name = NULL;
    if (argv[optind] ∧ argv[optind][0] ≠ '&' ∧ argv[optind][0] ≠ '\\') {
#ifdef WIN32
```

```
      clean_windows_filename(argv[optind]);
#endif
      argv[optind] = normalize_quotes(argv[optind], "argument");
      input_file_name = tl_find_file(argv[optind], kpse_tex_format, false);
    }
    return input_file_name;
  }
```

After we called *get_input_file_name*, we might need to look at *argv*[*argc* − 1] in case we run under Windows.

```
⟨ set the input file name 21 ⟩ ≡                                    (21)
  main_input_file = get_input_file_name( );
#ifdef WIN32                                  /* Were we given a simple filename? */
  if (main_input_file ≡ NULL) { char *name = argv[argc − 1];
    if (name ∧ name[0] ≠ '-' ∧ name[0] ≠ '&' ∧ name[0] ≠ '\\') {
      clean_windows_filename(name);
      name = normalize_quotes(name, "argument");
      main_input_file = tl_find_file(name, kpse_tex_format, false);
      argv[argc − 1] = name;
    }
  }
#endif                                                          Used in 17.
```

After we have an input file, we make an attempt at filling in options from the `texmf.cfg` file.

```
⟨ set defaults from the texmf.cfg file 22 ⟩ ≡                        (22)
  if (filelineerrorstylep < 0)
    filelineerrorstylep = texmf_yesno("file_line_error_style");
  if (parsefirstlinep < 0) parsefirstlinep = texmf_yesno("parse_first_line");
                                                                Used in 17.
```

We needed:

```
⟨ TEX Live support functions 1 ⟩ +≡                                  (23)
  static bool texmf_yesno(const char *var)
  { char *value = kpse_var_value(var);
    return value ∧ (*value ≡ 't' ∨ *value ≡ 'y' ∨ *value ≡ '1');
  }
```

To set the format name, we first check if the format name was given on the command line with an `&` prefix, second we might check the first line of the input file, and last, we check if the program is an initex or virtex program.

If we still don't have a format, we use a plain format if running as a virtex, otherwise the program name is our best guess. There is no need to check for an extension, because the `kpathsearch` library will take care of that. We store the format file name in *dump_name* which is used in the function *tl_open_fmt* below.

```
⟨ set the format name 24 ⟩ ≡                                         (24)
  if (parsefirstlinep ∧ ¬dump_name) parse_first_line(main_input_file);
```

```
  if (¬main_input_file ∧ argv[1] ∧ argv[1][0] ≡ '&') dump_name = argv[1] + 1;
#ifdef HITEX
  if (strcmp(kpse_program_name, "hinitex") ≡ 0) iniversion = true;
  else if (strcmp(kpse_program_name, "hvirtex") ≡ 0 ∧ ¬dump_name)
    dump_name = "hitex";
#else
  if (strcmp(kpse_program_name, "cinitex") ≡ 0) iniversion = true;
  else if (strcmp(kpse_program_name, "cvirtex") ≡ 0 ∧ ¬dump_name)
    dump_name = "ctex";
#endif
  if (¬dump_name) dump_name = kpse_program_name;
  if (¬dump_name) { fprintf(stderr, "Unable␣to␣determine␣format␣name\n");
    exit(1);
  }
```

Used in 17.

Here is the function *parse_first_line*. It searches the first line of the file for a TEX comment of the form "%&format"[1]. If found we will use the format given there.

⟨ TEX Live support functions 1 ⟩ +≡ (25)

```
  static void parse_first_line(char *filename)
  { FILE *f = NULL;
    if (filename ≡ NULL) return;
    f = tl_open_tex(filename);
    if (f ≠ NULL) { char *r, *s, *t = read_line(f);
      xfclose(f, filename);
      if (t ≡ NULL) return;
      s = t;
      if (s[0] ≡ '%' ∧ s[1] ≡ '&') { s = s + 2;
        while (ISBLANK(*s)) ++s;
        r = s;
        while (*s ≠ 0 ∧ *s ≠ '␣' ∧ *s ≠ '\r' ∧ *s ≠ '\n') s++;
        *s = 0;
        if (dump_name ≡ NULL) { char *f_name = concat(r, ".fmt");
          char *d_name = kpse_find_file(f_name, kpse_fmt_format, false);
          if (d_name ∧ kpse_readable_file(d_name)) { dump_name = xstrdup(r);
            kpse_reset_program_name(dump_name);
          }
          free(f_name);
        }
      }
      free(t);
    }
  }
```

[1] The idea of using this format came from Wlodzimierz Bzyl.

The TEX Live infrastructure is able to generate format files, tex files, font metric files, and pk files if required.

```
⟨ enable the generation of input files 26 ⟩ ≡                                (26)
  kpse_set_program_enabled(kpse_tfm_format, MAKE_TEX_TFM_BY_DEFAULT,
      kpse_src_compile);
  kpse_set_program_enabled(kpse_tex_format, MAKE_TEX_TEX_BY_DEFAULT,
      kpse_src_compile);
  kpse_set_program_enabled(kpse_fmt_format, MAKE_TEX_FMT_BY_DEFAULT,
      kpse_src_compile);
#ifdef HITEX
  kpse_set_program_enabled(kpse_pk_format, MAKE_TEX_PK_BY_DEFAULT,
      kpse_src_compile); xputenv("MAKETEX_BASE_DPI", option_dpi_str);
  xputenv("MAKETEX_MODE", option_mfmode);
#endif                                                          Used in 17.
```

In the old days, TEX was a Pascal program, and standard Pascal did say nothing about a command line. So TEX would open the terminal file for input and read all the information from the terminal. If you don't give TEX command line arguments, this is still true today. In our present time, people got so much used to control the behaviour of a program using command line arguments—especialy when writing scripts—that TEX Live allows the specification of commands on the command line which the TEX engine would normaly expect on the first line of its terminal input. So our next task is the writing a function to add the command line to the input buffer.

```
⟨ TEX Live functions 11 ⟩ +≡                                                 (27)
  void input_add_argv(void)
  { while (optind < argc) input_add_str(argv[optind++]); }
```

2.2 Opening Files with the kpathsearch Library

In ctex.w the standard function *fopen*, packaged into TEX's *reset* macro, is used to open files. The kpathsearch library uses different search pathes for different types of files and therefore different functions are needed to open these files.

We define four functions: one for text files containing TEX sources and three for binary files containing formats, font metrics, and PK pixel data.

Here is the function to open a TEX input file. It is used to open TEX source files and TEX font metric files. They are very similar.

```
⟨ TEX Live support functions 1 ⟩ +≡                                          (28)
  static FILE *tl_open_in(char *filename, kpse_file_format_type t, const
      char *mode)
  { char *fname = NULL;
    FILE *f = NULL;
    fname = tl_find_file(filename, t, true);
    if (fname ≠ NULL) { f = fopen(fname, mode); free(fname); }
    return f;
  }
```

```
FILE *tl_open_tex(char *filename)
{ return tl_open_in(filename, kpse_tex_format, "r"); }
FILE *tl_open_tfm(const char *filename)
{ return kpse_open_file(filename, kpse_tfm_format);
}
```

Since font metric files and glyph data is also needed for the directory section of a HINT file, we need the next three functions. To find outline fonts, it is actually necessary to find and consult the `psfonts.map` or `ps2pk.map` files to obtain the correct names of Type1, TrueType, or OpenType fonts (and their encoding). This is still omitted here.

⟨ TEX Live functions 11 ⟩ +≡ (29)

```
#ifdef HITEX
  char *tl_find_tfm(char *filename)
  { char *fname = NULL;
    fname = tl_find_file(filename, kpse_tfm_format, true);
    if (fname ≡ NULL)
      fprintf(stderr, "Unable␣to␣find␣font␣metric␣file␣for␣font␣%s\n",
          filename), exit(1);
    return fname;
  }
  char *tl_find_glyph(char *filename)
  { char *fname = NULL;
    kpse_glyph_file_type file_ret;
    fname = tl_find_file(filename, kpse_type1_format, true);
    if (fname ≡ NULL) fname = tl_find_file(filename, kpse_truetype_format, true);
    if (fname ≡ NULL) fname = tl_find_file(filename, kpse_opentype_format, true);
    if (fname ≡ NULL)
      fname = kpse_find_glyph(filename, option_dpi, kpse_pk_format, &file_ret);
    if (fname ≡ NULL) fprintf(stderr,
          "Unable␣to␣find␣glyph␣data␣for␣font␣%s\n", filename), exit(1);
    return fname;
  }
#endif
```

TEX's *open_fmt_file* function will call the following function either with the name of a format file as given with an "`&`" prefix in the input or with `NULL` if no such name was specified. The function will try *dump_name* as a last resort before returning `NULL`.

⟨ TEX Live functions 11 ⟩ +≡ (30)

```
FILE *tl_open_fmt(char *file_name)
{ FILE *f;
  if (file_name ≠ NULL ∧ file_name[0] ≠ 0) {
    f = tl_open_in(file_name, kpse_fmt_format, "rb");
    if (f ≠ NULL) return f;
```

```
    }
    f = tl_open_in(dump_name, kpse_fmt_format, "rb"); return f;
  }
```

When we open an output file, there is usually no searching necessary. In the best case, we have an absolute path and can open it. If the path is relative, we try in this order:

- the *file_name* prefixed by the *output_directory*,
- the *file_name* as is, and
- the *file_name* prefixed with the environment variable TEXMFOUTPUT.

If we were successful with one of the modified names, we update *name_of_file* using *pack_buffered_name*.

⟨ TEX Live functions 11 ⟩ +≡ (31)

```
  FILE *tl_open_out(const char *file_name, const char *file_mode)
  { FILE *f = NULL;
    char *new_name = NULL;
    bool absolute = kpse_absolute_p(file_name, false);
    if (absolute) { f = fopen(file_name, file_mode);
      if (f ≠ NULL) return f;
    }
    else {
      if (output_directory) { char *new_name = concat3(output_directory,
            DIR_SEP_STRING, file_name);
        f = fopen(new_name, file_mode);
      }
      if (f ≡ NULL) { free(new_name); new_name = NULL;
        f = fopen(file_name, file_mode);
      }
      if (f ≡ NULL) {
        const char *texmfoutput = kpse_var_value("TEXMFOUTPUT");
        if (texmfoutput ≠ NULL ∧ texmfoutput[0] ≠ 0) {
          new_name = concat3(texmfoutput, DIR_SEP_STRING, file_name);
          f = fopen(new_name, file_mode);
          if (f ≡ NULL) { free(new_name); new_name = NULL;
          }
        }
      }
    }
    if (f ≠ NULL ∧ new_name ≠ NULL)
      pack_buffered_name(new_name, 0, strlen(new_name) − 1);
    free(new_name); return f;
  }
```

2.2.1 Date and Time

We provide a function that can be used to initialize TEX's date and time information.

⟨ TEX Live functions 11 ⟩ +≡ (32)

```
#include <time.h>
  void tl_date_and_time(int *t, int *d, int *m, int *y)
  { time_t now = time(NULL);
    struct tm *gmt = gmtime(&now);
    *t = gmt→tm_hour * 60 + gmt→tm_min; *d = gmt→tm_mday;
    *m = gmt→tm_mon + 1; *y = gmt→tm_year + 1900;
  }
```

2.3 From TEX to TEX Live

We put the TEX Live functions into a separate C file `tltex.c` because we will use these functions not only for `hitex` but also for `ctex` the regular version of TEX that we use mainly as a reference to comare its output against the output of HiTEX. For all variables and function that must be shared between `tltex.c` and `ctex.c`, which we generate from `ctex.w`, we put a declaration in the `tltex.h` header file.

⟨ tltex.h 33 ⟩ ≡ (33)

```
#define HITEX_VERSION "1.1"
  extern int option_no_empty_page, option_hyphen_first, option_compress;
  extern bool option_global;
  extern int option_dpi;
  extern const char *option_mfmode, *option_dpi_str;
  extern char *dump_name;
  extern uint8_t interaction;
  extern int ready_already;
  extern int iniversion, filelineerrorstylep;
  extern FILE *tl_open_out(const char *file_name, const char *file_mode);
  extern FILE *tl_open_tex(char *file_name);
  extern FILE *tl_open_tfm(const char *file_name);
  extern FILE *tl_open_fmt(char *file_name);
  extern void tl_date_and_time(int *t, int *d, int *m, int *y);
  extern void tl_maininit(int argc, char *argv[]);
  extern void input_add_argv(void);
  extern void pack_buffered_name(const char *buffer, int a, int b);
  extern int make_tex_string(const char *str);
  extern int get_job_name(int s);
  extern void input_add_str(const char *str);
```

⟨ tltex.c 34 ⟩ ≡ (34)

```
#include "basetypes.h"
#include "error.h"
#include "hformat.h"
#include "tltex.h"
```

```
#include <kpathsea/kpathsea.h>
  extern void input_add_arg(char *str);
```

⟨ TEX Live variables 7 ⟩
⟨ TEX Live support functions 1 ⟩
⟨ TEX Live functions 11 ⟩

And now we are ready to incorporate the new functions into TEX.

2.3.1 iniTEX

TEX's table entries should not be initialized unless *iniversion* is true. Especially because *format_ident* would be set to `"(INITEX)"` which would later prevent the loading of a format file.

old

```
⟨ Initialize table entries (done by INITEX only) ⟩
```

becomes

new

```
if (iniversion) { ⟨ Initialize table entries (done by INITEX only) ⟩
}
```

2.3.2 Include files

Next we add the `tltex.h` include file after the other include files.

old

```
#include <math.h>
```

becomes

new

```
#include <math.h>
#include "tltex.h"
```

2.3.3 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;
```

2.3.4 Opening input files

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));
}
```

It now uses *tl_open_tex*:

new

```
bool a_open_in(alpha_file * f)          /* open a text file for input */
{ f→f = tl_open_tex((char *) 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));
}
```

It now uses *tl_open_tfm*:

new

```
bool b_open_in(byte_file * f)           /* open a binary file for input */
{ f→f = tl_open_tfm((char *) 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));
}
```

It now uses *tl_open_fmt*:

new

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

2.3.5 The command line

To handle the command line, we need a function to add strings to TEX's input buffer. So we duplicate part of the *input_ln* function and package it into the *input_add_str* function. We skip initial spaces and replace trailing space and line endings by a single space. We insert the new code before the function header of *input_ln*.

old

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

The new *input_add_str* function uses *input_add_char*.

new

```
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_str(const char *str)
{ int prev_last;
  while (*str ≡ '␣') str++;
  prev_last = last;
  while (*str ≠ 0) input_add_char(*str++);
  for (--last; last ≥ first; --last) { char c = buffer[last];
    if ((c) ≠ '␣' ∧ (c) ≠ '\r' ∧ (c) ≠ '\n') break;
  }
  last++;
  if (last > prev_last) input_add_char('␣');
}
bool input_ln(alpha_file * f, bool bypass_eoln)
```

As we will see below, we call the function *input_add_argv* immediately before calling *init_terminal*. As a consequence, there might be some input already when we enter *init_terminal* and we have to account for that. So we change the function *init_terminal*.

old

```
t_open_in;
```

becomes

new

```
t_open_in; last = first; input_add_argv( ); loc = first;
if (loc < last) return true;
```

2.3.6 Output files

We need to change TEX's *rewrite* mcaro, redirecting it to the *tl_open_ouput* function.

old

```
#define rewrite (file, name, mode)
  ((file).f = fopen((char *)(name) + 1, mode))
```

becomes

new

```
#define rewrite (file, name, mode)
  ((file).f = tl_open_out((char *)(name) + 1, mode))
```

If there is no *format_ident* then we use the *dump_name* as the format.

old

```
if (format_ident ≡ 0) wterm_ln("␣(no␣format␣preloaded)");
```

becomes

new

```
if (format_ident ≡ 0)
  wterm_ln("␣(preloaded␣format=%s)", dump_name);
```

Next we move the content of the *print_err* macro into a function. We need to make sure that *print_err* remains a compound statement because otherwise ocasionally a trailing semicolon would be missing. Further we add a call to *print_file_line* in case the "file:line:error" style was selected.

old

```
#define print_err(X)
        { if (interaction ≡ error_stop_mode) wake_up_terminal;
          print_nl("!␣"); print_str(X);
        }
```

becomes

new

```
#define print_err(X)
        { print_err_str(X);
        }
⟨ Basic print ⟩ ≡
  static void print_file_line(void){ int level = in_open;
      while (level > 0 ∧ source_filename_stack[level] ≡ 0) level−−;
      if (level ≡ 0) print_nl("! ␣");
      else { print_nl(""); print(source_filename_stack[level]);
      print_char(':'); if (level ≡ in_open) print_int ( line ) ;
      else print_int(line_stack[level]);
      print_str(": ␣");
      }
      }
      void print_err_str(char *str)
      { if (interaction ≡ error_stop_mode) wake_up_terminal;
        if (filelineerrorstylep) print_file_line( );
        else print_nl("! ␣");
        print_str(str);
      }
```

2.3.7 Setting time and date

We use the *gmtime* function to implement *fix_data_and_time*, because for a timestamp embedded into a file, a time that does not depend on the (unknown) location where the file was produced is more meaningful. Because *time* is one of TEX's macros, we move the main body of the function here.

old

```
void fix_date_and_time(void)
{ time = 12 * 60;                    /* minutes since midnight */
  day = 4;                           /* fourth day of the month */
  month = 7;                         /* seventh month of the year */
  year = 1776;                       /* Anno Domini */
}
```

now becomes

new

```
void fix_date_and_time(void)
{ tl_date_and_time(&(time), &(day), &(month), &(year));
}
```

2.3.8 Source file names

We need a stack, matching the *line_stack*, that contains the source file names, and postpone TEX Live's *full_source_filename_stack* to a later time. We add the function *print_file_line*. When the option *filelineerrorstylep* is set, the *print_err* function of TEX will call it instead of *print_nl*("! "). It prints "file:line:error" style messages. It looks for a filename in *source_filename_stack*, and if it fails to find one falls back on the "non-file:line:error" style.

old

```
int line_stack0[max_in_open], *const line_stack = line_stack0 − 1;
```

We add:

new

```
int line_stack0[max_in_open], *const line_stack = line_stack0 − 1;
int source_filename_stack0[max_in_open],
    *const source_filename_stack = source_filename_stack0 − 1;
```

We initialize the *source_filename_stack* in the *begin_file_reading* function.

old

```
incr(in_open); push_input; index = in_open;
```

becomes

new

```
incr(in_open); push_input; index = in_open;
source_filename_stack[index] = 0;
```

2.3.9 Opening the format file

Before we have a look at TEX's *open_fmt_file* function, we will change how TEX parses file names and rewrite the *pack_buffered_name* function which TEX uses only here.

The last '/' ends the aerea and the last '.' starts the extension.

old

```
if ((c ≡ '>') ∨ (c ≡ ':')) { area_delimiter = cur_length;
  ext_delimiter = 0;
}
else if ((c ≡ '.') ∧ (ext_delimiter ≡ 0)) ext_delimiter = cur_length;
```

becomes

new

```
if (c ≡ '/') { area_delimiter = cur_length; ext_delimiter = 0;
}
else if (c ≡ '.') ext_delimiter = cur_length;
```

We redefine *pack_buffered_name* so that we can use it to update *name_of_file* whenever we found a file with a slightly modified name. Further, we take the opportunity to add a function to convert C-strings to TEX-strings.

old

```
void pack_buffered_name(small_number n, int a, int b)
{ int k;                    /* number of positions filled in name_of_file */
  ASCII_code c;                          /* character being packed */
  int j;                    /* index into buffer or TEX_format_default */

  if (n + b - a + 1 + format_ext_length > file_name_size)
    b = a + file_name_size - n - 1 - format_ext_length;
  k = 0;
  for (j = 1; j ≤ n; j++)
    append_to_name(xord[TEX_format_default[j]]);
  for (j = a; j ≤ b; j++) append_to_name(buffer[j]);
  for (j = format_default_length - format_ext_length + 1;
        j ≤ format_default_length; j++)
    append_to_name(xord[TEX_format_default[j]]);
  if (k ≤ file_name_size) name_length = k; else
    name_length = file_name_size;
  name_of_file[name_length + 1] = 0;
}
```

becomes

new

```
void pack_buffered_name(const char *buffer, int a, int b)
{ int k;                    /* number of positions filled in name_of_file */
  ASCII_code c;                          /* character being packed */
  int j;                                     /* index into buffer  */

  k = 0;
  for (j = a; j ≤ b; j++) append_to_name(buffer[j]);
  if (k ≤ file_name_size) name_length = k; else
    name_length = file_name_size;
  name_of_file[name_length + 1] = 0;
}
int make_tex_string(const char *str)
{ int j, k;

  j = strlen(str); str_room(j);
  for (k = 0; k < j; k++) append_char(xord[(int) str[k]]);
  return make_string( );
}
```

Because *w_open_in* uses *tl_open_fmt* which in turn uses the `kpathsearch` library there is no need to try a "system format file area" or "PLAIN".

old

```
pack_buffered_name(0, loc, j − 1);
                                /∗ try first without the system file area ∗/
if (w_open_in(&fmt_file)) goto found;
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!");
```

becomes

new

```
pack_buffered_name((char ∗) buffer, loc, j − 1);
                                /∗ try first without the system file area ∗/
if (w_open_in(&fmt_file)) goto found;
}
name_of_file[1] = 0; if (¬w_open_in(&fmt_file)) { wake_up_terminal;
wterm_ln("I can't find a format file!");
```

Since TEX Live allows the specification of a job name on the command line, the assignment to *job_name* needs to go through the function *get_job_name*.

old

```
if (job_name ≡ 0) job_name = ⟨ "texput" ⟩;
```

becomes

new

```
if (job_name ≡ 0) job_name = get_job_name(⟨ "texput" ⟩);
```

Adding a default file location can be removed:

old

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

In the function *start_input*, we need to update the *source_filename_stack*.

old

```
done: name = a_make_name_string(&cur_file);
```

becomes

new

```
done: name = a_make_name_string(&cur_file);
  source_filename_stack[in_open] = name;
```

The function *start_input* contains the second assignment to *job_name* which needs to be replaced:

old

```
{ job_name = cur_name; open_log_file( );
```

becomes

new

```
{ job_name = get_job_name(cur_name); open_log_file( );
```

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" ⟩);
```

is simplified to:

new

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

2.3.10 Initialization

Last not least we have to insert the TEX Live initialization routine.

old

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

becomes

new

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

2.4 Miscellaneous Changes

The changest that follow here should ultimately go into the `web2w` program.

2.4.1 The pool_name constant

While *pool_name* is a constant, it's a constant variable. 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␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣";
```

2.4.2 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))
```

2.4.3 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)
```

The following code produced a compiler warning, because applying the unary minus operator to an unsigned value has a result that is still unsigned.

old

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

becomes

new

```
xn_over_d = u;  rem = v % d;
if (¬positive) { xn_over_d = −xn_over_d;  rem = −rem;
}
```

2.4.4 The end

At the end, we want HiTEX to terminate its terminal output properly without unnecessary verbosity but with a final new line.

old

```
slow_print(log_name); print_char('.');
```

now becomes

new

```
slow_print(log_name); print_char('.'); wterm_cr;
```

3 Modifying TEX

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

3.1 The *main* Program

We add two function calls to TEX's *main* program: *hint_open*, and *hint_close*.

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 */
```

At the end, there is no need for HiTEX to ⟨ Finish the `DVI` file ⟩.

old

```
wake_up_terminal; ⟨ Finish the DVI file ⟩;
```

is siplified to

new

```
wake_up_terminal; END
```

The functions to open and close the HINT file follow in section 7.

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 -compress &plain hello.tex`" or just "`hitex hello.tex`".

3.2 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 and put the code to update the values of *last_glue*, *last_penalty*, and *last_kern* into its own function to use it in the new *build_page* function.

old

```
⟨ Declare the procedure called fire_up ⟩
  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 ⟩;
  }
```

becomes

new

```
  void update_last_values(pointer p)
  { ⟨ Update the values of last_glue, last_penalty, and last_kern ⟩;
  }
```

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. If however the *option_no_empty_page* is enabled, we make sure that the penalty gets to the output by using a double *eject_penalty*.

old

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

becomes

new

```
  tail_append(new_set_node( ));
  set_extent(tail) = new_xdimen(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(2 ∗ (eject_penalty)));
```

3.3 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.

3.3.1 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 ∗/
```

To manage the ref counts of extended dimensions, we need to add the *delete_xdimen_ref* function.

old

```
void delete_glue_ref(pointer p)      /∗ p points to a glue specification ∗/
fast_delete_glue_ref(p)
```

new

```
void delete_glue_ref(pointer p)      /∗ p points to a glue specification ∗/
fast_delete_glue_ref(p)void delete_xdimen_ref(pointer p)
                                     /∗ p points to a xdimen specification ∗/
    { if (xdimen_ref_count(p) ≡ null) free_node(p, xdimen_node_size);
      else decr(xdimen_ref_count(p));
    }
```

For incrementing reference counts, we have

old

```
#define add_glue_ref(X) incr(glue_ref_count(X))
                                     /∗ new reference to a glue spec ∗/
```

new

```
#define add_glue_ref(X) incr(glue_ref_count(X))
                                        /* new reference to a glue spec */
#define add_xdimen_ref(X) incr(xdimen_ref_count(X))
                                        /* new reference to an xdimen */
```

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. Note that *set_extent* and *pack_extent* share the same location in the node. 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 5 /* number of memory words in a graf_node */
#define graf_penalty(X) mem[X+1].i          /* the final penalty */
#define graf_extent(X) link(X+3)                      /* the extent */
#define graf_params(X) info(X+4)      /* list of parameter nodes */
#define graf_list(X) link(X+4)          /* list of content nodes */
#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)+1+list_offset].sc
  /* depth limit in vpack */
#define pack_extent(X) link(X+2+list_offset) /* extent */
#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)+1+list_offset].sc
  /* replaces glue_set */
#define set_extent(X) pack_extent(X) /* extent */
#define set_shrink(X) mem[(X)+3+list_offset].sc
#define align_node 16 /* represents an alignment */
#define align_node_size 4
#define align_extent(X) link(X+2) /* the extent of the alignment */
#define align_m(X) type(X+2) /* either additional or exactly */
#define align_v(X) subtype(X+2) /* true if vertical */
#define align_preamble(X) info(X+3) /* the preamble */
#define align_list(X) link(X+3) /* the unset rows/columns */
#define setpage_node 17 /* represents a page template */
#define setpage_node_size 6
#define setpage_name(X) link(X+1)
#define setpage_number(X) type(X+1) /* the HINT number */
#define setpage_id(X) subtype(X+1) /* the TEX number */
```

```
#define setpage_priority(X) info(X+2)
#define setpage_topskip(X) link(X+2)
#define setpage_depth(X) mem[X+3].sc /* maximum depth */
#define setpage_height(X) info(X+4)
                                /* extended dimension number */
#define setpage_width(X) link(X+4)
                                /* extended dimension number */
#define setpage_list(X) info(X+5) /* the template itself */
#define setpage_streams(X) link(X+5) /* list of stream definitions */
#define setstream_node 18 /* represents a stream definition */
#define setstream_node_size 6
#define setstream_number(X) type(X+1)
#define setstream_insertion(X) subtype(X+1)
#define setstream_mag(X) link(X+1) /* magnification factor */
#define setstream_prefered(X) type(X+2)
#define setstream_next(X) subtype(X+2)
#define setstream_ratio(X) link(X+2) /* split ratio */
#define setstream_max(X) info(X+3)
                                /* extended dimension number */
#define setstream_width(X) link(X+3)
                                /* extended dimension number */
#define setstream_topskip(X) info(X+4)
#define setstream_height(X) link(X+4)
#define setstream_before(X) info(X+5)
#define setstream_after(X) link(X+5)
#define stream_node 19 /* represents a stream insertion point */
#define stream_node_size 2
#define stream_number(X) type(X+1)
#define stream_insertion(X) subtype(X+1)
#define stream_after_node 20 /* never allocated */
#define stream_before_node 21 /* never allocated */
#define xdimen_node 22
#define xdimen_node_size 4
#define xdimen_ref_count(X) link(X)
#define xdimen_width(X) mem[X+1].sc
#define xdimen_hfactor(X) mem[X+2].sc
#define xdimen_vfactor(X) mem[X+3].sc
```

3.3.2 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;
case setpage_node: print_str("setpage"); break;
case setstream_node: print_str("setstream"); break;
case stream_node: print_str("stream"); break;
case xdimen_node: print_str("xdimen"); break;
```

3.3.3 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 3.10, page 76 */
case setpage_node: break;                  /* see section 3.10, page 77 */
case stream_node: break;
case setstream_node: break;
case stream_before_node: break;
case stream_after_node: break;
case xdimen_node: break;
```

3.3.4 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(')'); node_list_display(graf_params(p));
  node_list_display(graf_list(p)); print_xdimen(graf_extent(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;
case setpage_node: print_str("\\setpage"); print_int(setpage_number(p));
  print_char('␣'); print(setpage_name(p)); print_str("␣priority␣");
  print_int(setpage_priority(p)); print_str("␣width␣");
  print_xdimen(setpage_width(p)); print_str("␣height␣");
  print_xdimen(setpage_height(p)); print_ln(); print_current_string();
  print_str(".\\topskip="); print_spec(setpage_topskip(p), 0); print_ln();
  print_current_string(); print_str(".\\maxdepth=");
  print_scaled(setpage_depth(p)); node_list_display(setpage_list(p));
  node_list_display(setpage_streams(p)); break;
case setstream_node: print_str("\\setstream");
  print_int(setstream_insertion(p)); print_char('(');
  print_int(setstream_number(p)); print_char(')');
  if (setstream_prefered(p) ≠ 255) { print_str("␣prefered␣");
    print_int(setstream_prefered(p));
  }
  if (setstream_ratio(p) > 0) { print_str("␣ratio␣");
    print_int(setstream_ratio(p));
  }
  if (setstream_next(p) ≠ 255) { print_str("␣next␣");
    print_int(setstream_next(p));
  }
  append_char('.'); print_ln(); print_current_string();
  print_str("\\count"); print_int(setstream_insertion(p));
  print_char('='); print_int(setstream_mag(p)); print_ln();
  print_current_string(); print_str("\\dimen");
  print_int(setstream_insertion(p)); print_char('=');
  print_xdimen(setstream_max(p)); print_ln(); print_current_string();
```

```
    print_str("\\skip"); print_int(setstream_insertion(p)); print_char('=');
    print_spec(setstream_height(p),0); print_ln(); print_current_string();
    print_str("\\hsize="); print_xdimen(setstream_width(p)); print_ln();
    print_current_string(); print_str("\\topskip=");
    print_spec(setstream_topskip(p),0);
    if (setstream_before(p) ≠ null) { print_ln(); print_current_string();
      print_str("\\before"); node_list_display(setstream_before(p));
    }
    if (setstream_after(p) ≠ null) { print_ln(); print_current_string();
      print_str("\\after"); node_list_display(setstream_after(p));
    }
    flush_char; break;
  case stream_node: print_str("\\stream"); print_int(stream_insertion(p));
    print_char('('); print_int(stream_number(p)); print_char(')'); break;
  case xdimen_node: print_str("\\xdimen␣"); print_xdimen(p); break;
  default: print_str("whatsit?");
  }
```

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

```
⟨ Basic printing ⟩ ≡
  static void print_xdimen(pointer p)
  { print_scaled(xdimen_width(p));
    if (xdimen_hfactor(p) ≠ 0) { print_char('+');
      print_scaled(xdimen_hfactor(p)); print_str("*hsize");
    }
    if (xdimen_vfactor(p) ≠ 0) { print_char('+');
      print_scaled(xdimen_vfactor(p)); print_str("*vsize");
    }
  }
  extern void print_baseline_skip(int i);
```

3.3.5 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); add_xdimen_ref(graf_extent(p));
    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));
    add_xdimen_ref(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 */
    add_xdimen_ref(set_extent(p));
    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));
    add_xdimen_ref(align_extent(p)); words = align_node_size − 1;
  } break;
case setpage_node:
```

```
  { r = get_node(setpage_node_size); add_glue_ref(setpage_topskip(p));
    add_xdimen_ref(setpage_height(p)); add_xdimen_ref(setpage_width(p));
    setpage_list(r) = copy_node_list(setpage_list(p));
    setpage_streams(r) = copy_node_list(setpage_streams(p));
    words = setpage_node_size − 1;
  } break;
case setstream_node:
  { r = get_node(setstream_node_size); add_xdimen_ref(setstream_max(p));
    add_xdimen_ref(setstream_width(p));
    add_glue_ref(setstream_topskip(p)); add_glue_ref(setstream_height(p));
    setstream_before(r) = copy_node_list(setstream_before(p));
    setstream_after(r) = copy_node_list(setstream_after(p));
    words = setstream_node_size − 1;
  } break;
case stream_node: r = get_node(stream_node_size);
  words = stream_node_size; break;
case xdimen_node: r = get_node(xdimen_node_size);
  words = xdimen_node_size; break;
default: confusion(⟨ "ext2" ⟩);
```

3.3.6 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: delete_xdimen_ref(graf_extent(p));
  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: delete_xdimen_ref(pack_extent(p));
  flush_node_list(list_ptr(p)); free_node(p, pack_node_size); break;
case hset_node: case vset_node: delete_xdimen_ref(set_extent(p));
  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: delete_xdimen_ref(align_extent(p));
  flush_node_list(align_preamble(p)); flush_node_list(align_list(p));
  free_node(p, align_node_size); break;
case setpage_node: delete_glue_ref(setpage_topskip(p));
  delete_xdimen_ref(setpage_height(p));
  delete_xdimen_ref(setpage_width(p)); flush_node_list(setpage_list(p));
  flush_node_list(setpage_streams(p)); free_node(p, setpage_node_size);
  break;
case setstream_node: delete_xdimen_ref(setstream_max(p));
  delete_xdimen_ref(setstream_width(p));
  delete_glue_ref(setstream_topskip(p));
  delete_glue_ref(setstream_height(p)); flush_node_list(setstream_before(p));
  flush_node_list(setstream_after(p)); free_node(p, setstream_node_size);
  break;
case stream_node: free_node(p, stream_node_size); break;
case xdimen_node: free_node(p, xdimen_node_size);
```

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: case setpage_node:
  case setstream_node: case xdimen_node: break;
default: confusion("ext4");
```

3.3.1 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);
```

3.3.2 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.

3.3.3 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 35 ⟩ ≡ (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_number n, str_number a, str_number e)
{ 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;
}
```

Used in 164.

3.3.4 Parameter nodes

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

⟨ Create the parameter node 36 ⟩ ≡ (36)

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

Used in 38.

⟨ Initialize the parameter node 37 ⟩ ≡ (37)

```
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 38.

⟨ HiTEX routines 35 ⟩ +≡ (38)

```
void add_par_node(uint8_t t, uint8_t n, int v)
{ pointer p;
```

```
    ⟨ Create the parameter node 36 ⟩
    ⟨ Initialize the parameter node 37 ⟩
    link(p) = link(temp_head); link(temp_head) = p;
}
```

3.4 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 par_shape_hfactor = 0, par_shape_vfactor = 0;
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];
}
else if (p ≡ par_shape_loc) {
  save_hfactor[save_ptr].i = par_shape_hfactor;
  save_vfactor[save_ptr].i = par_shape_vfactor;
}
incr(save_ptr);
```

Note that we save and restore the current hfactor and vfactor for the first element of the par shape array together with the par shape pointer.

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.

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

old

```
eq_level(p) = cur_level; eq_type(p) = t; equiv(p) = e;
```

new

```
eq_level(p) = cur_level; eq_type(p) = t; equiv(p) = e;
if (p ≡ par_shape_loc) { par_shape_hfactor = cur_hfactor;
  par_shape_vfactor = cur_vfactor;
}
```

And 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 + round(((double)
        cur_hfactor * hhsize + (double) cur_vfactor * hvsize)/unity);
      return; }
    if (p ≡ dimen_base + vsize_code) { hvsize = w + round(((double)
        cur_hfactor * hhsize + (double) cur_vfactor * hvsize)/unity);
      return; }
  }
  if (xeq_level[p] ≠ cur_level)
```

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 + round(((double)
      cur_hfactor * hhsize + (double) cur_vfactor * hvsize)/unity);
  else if (p ≡ dimen_base + vsize_code) hvsize = w + round(((double)
      cur_hfactor * hhsize + (double) cur_vfactor * hvsize)/unity);
  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];          /* restore the saved value */
```

new

```
eqtb[p] = save_stack[save_ptr];          /* restore the saved value */
if (p ≡ par_shape_loc) { par_shape_hfactor = save_hfactor[save_ptr].i;
  par_shape_vfactor = save_vfactor[save_ptr].i;
}
```

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)); overfull_rule = rule_save;
```

becomes

new

```
p = hpack(preamble, saved(1), saved_hfactor(1), saved_vfactor(1),
    saved(0)); 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

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

becomes

new

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

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

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. We do not implement xtended dimensions (yet) for the extended registers of e-TeX.

old

```
if (q ≡ advance) cur_val = cur_val + w;
```

becomes

new

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

When we redefine the parshape, we keep track of the hfactor and vfactor for the first element of the parshape array.

old

```
else { p = get_node(2 ∗ n + 1); info(p) = n;
for (j = 1; j ≤ n; j++) { scan_normal_dimen;
  mem[p + 2 ∗ j − 1].sc = cur_val;                    /∗ indentation ∗/
  scan_normal_dimen; mem[p + 2 ∗ j].sc = cur_val;     /∗ width ∗/
}
```

new

```
else { scaled fh = 0, fv = 0;
p = get_node(2 * n + 1); info(p) = n;
for (j = 1; j ≤ n; j++) { scan_normal_dimen;
  mem[p + 2 * j − 1].sc = cur_val;                    /* indentation */
  scan_normal_dimen;
  if (j ≡ 1) { fh = cur_hfactor; fv = cur_vfactor;
  }
  mem[p + 2 * j].sc = cur_val;                        /* width */
}
cur_hfactor = fh; cur_vfactor = fv;
```

When we dump a format file, we want to retain the "design-size" of a TEX document. Therefore, we restore the values of *hsize* and *vsize* from *hhsize* and *hvsize*. We insert the corresponding statements before starting to dump.

old

```
⟨ Dump constants for consistency check ⟩;
```

becomes

new

```
eqtb[dimen_base + hsize_code].i = hhsize;
eqtb[dimen_base + vsize_code].i = hvsize;
⟨ Dump constants for consistency check ⟩;
```

And we undo the changes when dumping is finished.

old

```
⟨ Close the format file ⟩;
```

becomes

new

```
⟨ Close the format file ⟩;
eqtb[dimen_base + hsize_code].i = 0; eqtb[dimen_base + vsize_code].i = 0;
```

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

3.5 Hyphenation

While the breaking of a paragraph into lines must be postponed because `hsize` is not known, hyphenation should be done as part of HiTEX because we want to keep hyphenation out of the viewer. Therefore HiTEX will do hyphenation for all words within a paragraph.

There is a fine point to observe here: TEX will consider a word as a candidate for automatic hyphenation only if the world "follows" after a glue. (For the exact rules, see the TEXbook[8], Appendix H.) As a consequence, TEX usually does not submit the first word of a paragraph to its hyphenation routine. Viewing paragraphs that

start with a lengthy word on a narrow display therefore often look more unsightly than necessary: the long word sticks out into the right margin as much as it can. To remedy this situation, HiTEX has a "`-f`" option. If set HiTEX will deviate from TEX's rules and submit the first word of a paragraph to the hyphenation algorithm.

The next problem arises from TEX's multipass approach to line breaking and the attempt to have HiTEX choose exactly the same line breaks as TEX does: TEX distingishes between discretionary breaks inserted by the author of a text, and discretionary breaks discovered by the hyphenation routine. The latter, called here "automatic", are used only in pass two and three of the line breaking routine.

The change file that follows below contains mostly 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);
```

In the *end_graf* function, we replace the original *line_break* function by *hline_break* to have a hook where we can insert special code for HiTEX. Because displayed equations are now part of the paragraphs content, an empty list after a displayed equaltion is handled like an empty paragraph.

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:

⟨ HiTEX routines 35 ⟩ +≡ (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;
  bool first_word = true;
  if (DBGTEX & debugflags) { print_ln( );
    print_str("Before␣hline_break:\n"); breadth_max = 200;
    depth_threshold = 200; show_node_list(link(head)); print_ln( );
  }
  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) = new_xdimen(dimen_par(hsize_code),
        dimen_par_hfactor(hsize_code), dimen_par_vfactor(hsize_code));
  else ⟨ fix the use of parshape = 1 indent length 40 ⟩
  link(temp_head) = link(head);
  if (is_char_node(tail)) { tail_append(new_penalty(inf_penalty))
    tail_append(new_param_glue(par_fill_skip_code));
  }
  else if (type(tail) ≠ whatsit_node ∨ subtype(tail) ≠ disp_node) {
    if (type(tail) ≠ glue_node) tail_append(new_penalty(inf_penalty))
    else { type(tail) = penalty_node; delete_glue_ref(glue_ptr(tail));
      flush_node_list(leader_ptr(tail)); penalty(tail) = inf_penalty;
    }
    link(tail) = new_param_glue(par_fill_skip_code);
  }
```

```
    DBG(DBGTEX,
        "\nCalling␣line_break:\n""hang_indent=0x%08X␣hang_after=%d",
        hang_indent, hang_after);
    if (line_skip_limit ≠ 0)
      DBG(DBGTEX, "␣line_skip_limit=0x%08X", line_skip_limit);
    DBG(DBGTEX, "␣prev_graf=0x%08X", prev_graf);
    init_cur_lang = prev_graf % °200000; init_l_hyf = prev_graf /°20000000;
    init_r_hyf = (prev_graf /°200000) % °100; pop_nest( );
    DBG(DBGTEX, "␣prev_graf=0x%08X", prev_graf);
                                                  /* 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;
    if (DBGTEX & debugflags) { print_ln( );
      print_str("Before␣hyphenation:\n"); breadth_max = 200;
      depth_threshold = 200; show_node_list(link(temp_head)); print_ln( );
    }
    auto_breaking = true; cur_p = temp_head;
    if (option_hyphen_first ∧ is_char_node(link(cur_p))) { hyphenate_word( );
      first_word = false;
    }
    cur_p = link(cur_p);
    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... */
        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);
        }
        cur_p = s; goto done5;
      case math_node: auto_breaking = (subtype(cur_p) ≡ after); break;
      default: break;
      }
```

```
    if (option_hyphen_first ∧ first_word ∧ is_char_node(link(cur_p))) {
      hyphenate_word( ); first_word = false;
    }
    cur_p = link(cur_p);
  done5: ;
  }
  if (DBGTEX & debugflags) { print_ln( ); print_str("After␣hline_break:\n");
    breadth_max = 200; depth_threshold = 200;
    show_node_list(link(temp_head)); print_ln( );
  }
  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(dimen_type, emergency_stretch_code, emergency_stretch);
  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(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);
}
```

Currently HiTEX dos not implement the parshape feature of TEX. The implementation of `\list` in LATEX does however depend on a simple us of parshape where all lines have the same length and indentation. We cover this special case be

using a hanging indentation and adjusting the paragraph width by the difference of the normal `\hsize` and the given length.

```
⟨ fix the use of parshape = 1 indent length 40 ⟩ ≡                    (40)
  { last_special_line = info(par_shape_ptr) − 1;
    if (last_special_line ≠ 0)
      DBG(DBGTEX, "Warning␣parshape␣with␣n=%d␣not␣yet␣implemented",
          info(par_shape_ptr));
    second_width = mem[par_shape_ptr + 2 ∗ (last_special_line + 1)].sc;
    second_indent = mem[par_shape_ptr + 2 ∗ last_special_line + 1].sc;
    graf_extent(pp) = new_xdimen(second_indent+second_width, par_shape_hfactor,
        par_shape_vfactor); second_width = second_width + round((double)
        par_shape_hfactor ∗ hhsize/unity + (double)
        par_shape_vfactor ∗ hvsize/unity); par_shape_fix = true;
  }                                                                  Used in 39.
```

3.6 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;
```

```
}
```

3.7 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 terminates 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.

When we go into display math mode, we do not terminate the current paragraph and call *line_break*. All that is left of the *line_break* routine is the appending of the parfill glue.

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 ⟩ +≡
  {
    if (head ≠ tail ∧ ¬(type(tail) ≡ whatsit_node ∧ subtype(tail) ≡
          disp_node)) {
      if (is_char_node(tail)) tail_append(new_penalty(inf_penalty))
      else if (type(tail) ≠ glue_node)
        tail_append(new_penalty(inf_penalty))
      else { type(tail) = penalty_node; delete_glue_ref(glue_ptr(tail));
        flush_node_list(leader_ptr(tail)); penalty(tail) = inf_penalty;
      }
      tail_append(new_param_glue(par_fill_skip_code));
    }
    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);
  }
  add_par_node(dimen_type, line_skip_limit_code, line_skip_limit);
  add_par_node(glue_type, line_skip_code, line_skip);
  add_par_node(glue_type, baseline_skip_code, baseline_skip);
  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( ); mode = hmode; space_factor = 1000; set_cur_lang;
clang = cur_lang; prev_graf = (norm_min(left_hyphen_min) * °100 +
    norm_min(right_hyphen_min)) * °200000 + cur_lang;
⟨ Scan an optional space ⟩;
```

3.8 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;                        /* indicates 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 $r = get\_node(align\_node\_size)$;
$save\_ptr = save\_ptr - 2$; $pack\_begin\_line = -mode\_line$;
$type(r) = whatsit\_node$; $subtype(r) = align\_node$;
$align\_preamble(r) = preamble$; $align\_list(r) = link(head)$;
$align\_extent(r) = new\_xdimen(saved(1), saved\_hfactor(1),$
$\quad saved\_vfactor(1))$; $align\_m(r) = saved(0)$;
$align\_v(r) = (mode \neq -vmode)$; $link(head) = r$; $tail = r$;
$pack\_begin\_line = 0$;

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

old

} **while** $(\neg(q \equiv null))$

new

if $(width(q) > max\_dimen)$ $x = true$;
} **while** $(\neg(q \equiv null))$

3.9 Inserts

TEX uses *vpack* to determine the natural height of inserted material. In HiTEX, the function *vpack* will not always return a *vlist_node* with a fixed height because the computation of the height might fail. In the latter case, *vpack* will return a *vset_node* or *vpack_node*. Therfore HiTEX will not use the *height* field of a an *ins_node*.
As a result we have the following changes:

In ⟨ Display insertion p ⟩ we remove the display of the height.

old

$print\_str$(",␣natural␣size␣"); $print\_scaled(height(p))$;

We skip the packing of the insertion, because we can not get height and depth anyway.

old

$p = vpack(link(head), natural)$; $pop\_nest()$;

becomes

new

$p = link(head);$ $pop\_nest();$

Finaly, we modify the creation of an insert node. We also allocte a HiTEX stream number here so that the maximum stream number will be correct when we start to output definitions.

old

$height(tail) = height(p) + depth(p);$ $ins\_ptr(tail) = list\_ptr(p);$

becomes

new

$height(tail) = 0;$ $ins\_ptr(tail) = p;$ $hget\_stream\_no(subtype(tail));$

And the same for adjust nodes. We also do no longer delete node p.

old

$adjust\_ptr(tail) = list\_ptr(p);$ $delete\_glue\_ref(q);$
}
$list\_ptr(p) = null;$ $flush\_node\_list(p);$

becomes

new

$adjust\_ptr(tail) = p;$ $delete\_glue\_ref(q);$
}

3.10 Implementing HiTEX Specific Primitives

Now we implement new TEX primitives: one to include images, a few more to define page teamplates, 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 strings 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
```

Then find out where the `kpathsearch` library looks for format files (for example by running

```
kpsewhich latex.fmt
```

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

First we allocate a new list head to store page templates. It will hold a list of *whatsit_node*'s with subtype *setpage_node*. We introduce this list head to be able to store page templates in a format file and retrieve them again. Without this feature it would not be possible to predefine page templates and stream definitions in a format file. We use the type and subtype field of the list head to store (and retrieve) the maximum page template and stream numbers.

old

```
#define backup_head mem_top − 13
                    /∗ head of token list built by scan_keyword ∗/
#define hi_mem_stat_min mem_top − 13
                    /∗ smallest statically allocated word in the one-word mem ∗/
#define hi_mem_stat_usage 14
                    /∗ the number of one-word nodes always present ∗/
```

becomes

new

```
#define backup_head mem_top − 13
                    /∗ head of token list built by scan_keyword ∗/
#define setpage_head mem_top − 14
                    /∗ head of page template list build by new_setpage_node ∗/
#define max_page type(setpage_head)
                    /∗ maximum page template number ∗/
#define max_stream subtype(setpage_head)
                    /∗ maximum stream number ∗/
#define hi_mem_stat_min mem_top − 14
                    /∗ smallest statically allocated word in the one-word mem ∗/
#define hi_mem_stat_usage 15
                    /∗ the number of one-word nodes always present ∗/
```

We need new group codes for page templates, stream temaplates, the before list and the after list.

old

```
#define max_group_code 16
```

new

```
#define page_group 17
#define stream_group 18
#define stream_before_group 19
#define stream_after_group 20
#define max_group_code 20
```

When TEX calls *handle_left_brace* it will branch to the following case labels:

old

```
case output_group: ⟨ Resume the page builder ⟩ break;
```

It becomes:

new

```
case output_group: ⟨ Resume the page builder ⟩ break;
case page_group: hfinish_page_group( ); break;
case stream_group: hfinish_stream_group( ); break;
case stream_before_group: hfinish_stream_before_group( ); break;
case stream_after_group: hfinish_stream_after_group( ); break;
```

In TEX's list of primitives, after the line

old

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

we add

new

```
primitive(⟨ "setlanguage" ⟩, extension, set_language_code);
primitive(⟨ "@HINT" ⟩, relax, 256);                    /* cf. relax */
primitive(⟨ "image" ⟩, extension, image_node);
primitive(⟨ "setpage" ⟩, extension, setpage_node);
primitive(⟨ "stream" ⟩, extension, stream_node);
primitive(⟨ "setstream" ⟩, extension, setstream_node);
primitive(⟨ "before" ⟩, extension, stream_before_node);
primitive(⟨ "after" ⟩, extension, stream_after_node);
```

The `\@HINT` primitive is simply equivalent to `\relax`. Its indended use is simply testing for its existence, for example the following code defines a new `\if` that can be used to test whether the current TEX is HiTEX.

```
\newif\ifhint
{ \catcode`11 % at signs are letters
\ifx\@HINT\relax \global\hinttrue \else \global\hintfalse \fi
\catcode`12 % at signs are no longer letters
}
```

The code associated with the other new primitives is defined in the *do_extension* function (see also section 3.3 on page 37).

Le's start with the image `\image` primitive. The use of it is more complex, because it has parameters, one required and several others optional. Here is the code to scan the parameters:

old

```
case image_node: break;                    /* see section 3.10, page 76 */
```

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);
    break;
  }
```

To define a page template in HiTEX, you use the new `setpage` primitive, to tell HiTEX about your new page template. The `setpage` primitive first needs a template number. After an optional equal sign follows a list of paramters: first the template name, which is required, then the priority, the width, and the height of the page which are optional. If an optional parameters is omitted, HiTEX will supply a default. After the parameters follows a list of vertical material, enclosed in braces, which defines the final page. Occurences of the `stream` primitive in this list will be replaced by the stream content as determined by the page builder. In addition, this list also contains definitions. Optional definitions are the topskip glue and the maximum depth of the page. Required are definitions for all streams that occur on the page.

The following code scans the parameters up to the left brace, creates a new setpage node, and adds it to the list starting at *setpage_head*. The handling of the vertical list is defered until TEX calls *handle_right_brace*.

old

```
case setpage_node: break;          /* see section 3.10, page 77 */
```

becomes

new

```
case setpage_node:
  { uint8_t n; pointer t;
    scan_eight_bit_int( ); n = cur_val;
    if (n ≡ 0) {
      print_err("Illegal␣redefinition␣of␣page␣template␣0");
      print_int(n);
      error ( ) ;
      break;
    }
    scan_optional_equals( ); scan_file_name( );
                          /* this should be improved to use scan_name */
    if (cur_area ≠ empty_string) str_start[cur_name] = str_start[cur_area];
    if (cur_ext ≠ empty_string) str_start[cur_ext] = str_start[cur_ext + 1];
    t = new_setpage_node(n, cur_name);
    loop {
      if (scan_keyword(⟨ "priority" ⟩)) { scan_eight_bit_int( );
        setpage_priority(t) = cur_val;
      }
      else if (scan_keyword(⟨ "width" ⟩)) { scan_normal_dimen;
        setpage_width(t) = new_xdimen(cur_val, cur_hfactor, cur_vfactor);
      }
      else if (scan_keyword(⟨ "height" ⟩)) { scan_normal_dimen;
        setpage_height(t) = new_xdimen(cur_val, cur_hfactor, cur_vfactor);
      }
      else break;
    }
    new_save_level(page_group); scan_left_brace( ); normal_paragraph( );
    push_nest( ); mode = −vmode; prev_depth = ignore_depth; break;
  }
```

The height parameter is an extended vertical dimension. It is taken as the vertical dimension of the page. For example, the dimension could be $1 \cdot$ `vsize` plus some constant, where the constant accounts for the top and bottom margin. Another example is a dimension of $1.2 \cdot$ `vsize` where the top and bottom margin together would account for 2/10 of the page height. The width parameter specifies the horizontal dimension of the page in a similar manner.

Next we look at the stream nodes. These occur inside the list of vertical material that defines the page. After the page builder has determined the content of stream number n replaces the node generated by `\stream`n. The only argument of the `stream` primitive is the stream number. It is computed from TEX's insertion number using *hget_stream_no*.

old

```
case stream_node: break;
```

becomes

new

```
case stream_node:
  { uint8_t n;
    scan_eight_bit_int( ); n = cur_val;
    tail_append(get_node(stream_node_size)); type(tail) = whatsit_node;
    subtype(tail) = stream_node; stream_insertion(tail) = n;
    stream_number(tail) = hget_stream_no(n); break;
  }
```

Now we consider a more complex primitive: the `setstream` primitive which produces the stream definition nodes.

Similar to the `setpage` primitive, it first needs a template number, and after an optional equal sign follows a list of optional parameters: the number of the preferd stream, the number of the next stream, and the split ratio. The stream numbers are, as usual, given as TEX's insert numbers and convertted using *hget_stream_no*. The list that follows after the parameters just contains definitions. The two most important definitions are the definition of a `before` and an `after` list. These lists are added—as the names say—before and after the content of a nonempty stream before adding the content to the page. To make it simple for the page builder to account for this added material, the list should also set the `\skip`n register to reflect the height, stretchability, and shrinkablility of this added material. Other parameters that can be defined in this list are the `hsize` used for processing insertions into that stream, the magnification factor contained in the `\count`n register, and the maximum height that material of this stream should occupy on the page in the `\dimen`n register.

The code below scans the parameters, creates a new setstream node, and adds it to the list of streams of the current page template. The handling of the list is deferd until TEX calls *handle_right_brace*. To distinquish between a right brace that ends the `setpage` primitive from one that ends the *s*etstream primitive. The former puts a zero on the save stack where as the latter pots a one on the save stack.

old

```
case setstream_node: break;
```

becomes

new

```
case setstream_node:
  { uint8_t n;
    pointer t, s;
```

```
  scan_eight_bit_int( ); n = cur_val; scan_optional_equals( );
  t = link(setpage_head);
  if (t ≡ null) { print_err("\\setstream␣without␣\\setpage");
    break;
  }
  s = new_setstream_node(n); link(s) = setpage_streams(t);
  setpage_streams(t) = s;
  loop {
    if (scan_keyword(⟨"prefered"⟩)) { scan_eight_bit_int( );
      if (cur_val ≠ 255)
        setstream_prefered(s) = hget_stream_no(cur_val);
    }
    else if (scan_keyword(⟨"next"⟩)) { scan_eight_bit_int( );
      if (cur_val ≠ 255) setstream_next(s) = hget_stream_no(cur_val);
    }
    else if (scan_keyword(⟨"ratio"⟩)) { scan_int( );
      setstream_ratio(s) = cur_val;
    }
    else break;
  }
  new_save_level(stream_group); scan_left_brace( ); normal_paragraph( );
  push_nest( ); mode = −vmode; prev_depth = ignore_depth; break;
}
```

Finaly the come to the two last primitives for defining the vertical material in the `before` and `after` lists of a stream definition. There are no parameters to parse here and by pushing a 2 and a 3 on the save stack, the *handle_right_brace* function will know where to put the final list.

old

```
case stream_before_node: break;
case stream_after_node: break;
```

becomes

new

```
case stream_before_node: scan_optional_equals( );
  new_save_level(stream_before_group); scan_left_brace( );
  normal_paragraph( ); push_nest( ); mode = −vmode;
  prev_depth = ignore_depth; break;
case stream_after_node: scan_optional_equals( );
  new_save_level(stream_after_group); scan_left_brace( );
  normal_paragraph( ); push_nest( ); mode = −vmode;
  prev_depth = ignore_depth; break;
```

We are left with the task to add the strings that were used above to TEX's string pool. The last string of TEX is string number 667 and we add the new strings after it.

First we define the strings

old

```
#define str_718 "displaywidowpenalties"
⟨ "displaywidowpenalties" ⟩ ≡ 718
```

becomes

new

```
#define str_718 "displaywidowpenalties"
⟨ "displaywidowpenalties" ⟩ +≡ 718
#define str_719 "image"
⟨ "image" ⟩ ≡ 719
#define str_720 "prefered"
⟨ "prefered" ⟩ ≡ 720
#define str_721 "next"
⟨ "next" ⟩ ≡ 721
#define str_722 "ratio"
⟨ "ratio" ⟩ ≡ 722
#define str_723 "priority"
⟨ "priority" ⟩ ≡ 723
#define str_724 "stream"
⟨ "stream" ⟩ ≡ 724
#define str_725 "setstream"
⟨ "setstream" ⟩ ≡ 725
#define str_726 "setpage"
⟨ "setpage" ⟩ ≡ 726
#define str_727 "before"
⟨ "before" ⟩ ≡ 727
#define str_728 "after"
⟨ "after" ⟩ ≡ 728
#define str_729 "@HINT"
⟨ "@HINT" ⟩ ≡ 729
```

Next we add the strings to the string pool:

old

str_712 str_713 str_714 str_715 str_716 str_717 str_718

becomes

new

str_712 str_713 str_714 str_715 str_716 str_717 str_718
str_719 str_720 str_721 str_722 str_723 str_724
str_725 str_726 str_727 str_728 str_729

Then we add the position to the *str_start* array

old

str_start_716 , *str_start_717* , *str_start_718* , *str_start_719*

becomes

new

str_start_716 , *str_start_717* , *str_start_718* , *str_start_719* , *str_start_720* ,
str_start_721 , *str_start_722* , *str_start_723* , *str_start_724* , *str_start_725* ,
str_start_726 , *str_start_727* , *str_start_728* , *str_start_729* , *str_start_730*

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

old

str_start_719 = *str_start_718* + **sizeof** (*str_718*) − 1,
str_start_end

}

str_starts;

⟨ *pool_ptr* initialization ⟩ ≡
str_start_719

⟨ *str_ptr* initialization ⟩ ≡
719

becomes

new

str_start_719 = *str_start_718* + **sizeof** (*str_718*) − 1,
str_start_720 = *str_start_719* + **sizeof** (*str_719*) − 1,
str_start_721 = *str_start_720* + **sizeof** (*str_720*) − 1,
str_start_722 = *str_start_721* + **sizeof** (*str_721*) − 1,
str_start_723 = *str_start_722* + **sizeof** (*str_722*) − 1,
str_start_724 = *str_start_723* + **sizeof** (*str_723*) − 1,
str_start_725 = *str_start_724* + **sizeof** (*str_724*) − 1,
str_start_726 = *str_start_725* + **sizeof** (*str_725*) − 1,
str_start_727 = *str_start_726* + **sizeof** (*str_726*) − 1,
str_start_728 = *str_start_727* + **sizeof** (*str_727*) − 1,

str_start_729 = *str_start_728* + **sizeof** (*str_728*) − 1,
str_start_730 = *str_start_729* + **sizeof** (*str_729*) − 1,
str_start_end

}

str_starts;

⟨ *pool_ptr* initialization ⟩ +≡
str_start_730

⟨ *str_ptr* initialization ⟩ +≡
730

4 HiTEX

4.1 Images

The handling of images is an integral part of HiTEX. In section 3.10 on page 76 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 35 ⟩ +≡ (42)

```
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 42 ⟩ ≡ (42)

```
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 164.

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

⟨ HiTEX auxiliar routines 42 ⟩ +≡ (43)

```
#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;
  }
```

4.1.1 BMP

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 42 ⟩ +≡                                    (44)
  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);
  }
```

4.1.2 PNG

Now we repeat this process for image files in using the Portable Network Graphics[2] 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 42 ⟩ +≡                                    (45)
  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;
}
```

4.1.3 JPG

For photographs, the JPEG File Interchange Format (JFIF)[5][6] 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 42 ⟩ +≡ (46)

```
  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;
  }
```

4.1.4 SVG

There is still one image format missing: scalable vector graphics. In the moment, I tend not to include a further image format into the definition of the HINT file format but instead use the PostScript subset that is used for Type 1 fonts to encode vector graphics. Any HINT viewer must support Type 1 PostScript fonts and hence it has already the necessary interpreter. So it seems reasonable to put the burden of converting vector graphics into a Type 1 PostScript font on the generator of HINT files and keep the HINT viewer as small and simple as possible.

4.2 The New Page Builder

Here is the new *build_page* routine of HiTEX:

⟨ HiTEX routines 35 ⟩ +≡ (47)

```
  void build_page(void)
  { static bool initial = true;
    if (link(contrib_head) ≡ null) return;
    do { pointer p = link(contrib_head);
      ⟨ suppress empty pages if requested 49 ⟩
      update_last_values(p); ⟨ freeze the page specs if called for 48 ⟩
      page_goal = #7FFF0000; hout_node(p);
    recycle_p: 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);
  }
```

⟨ freeze the page specs if called for 48 ⟩ ≡ (48)

```
  if (page_contents < box_there) {
    switch (type(p)) {
    case whatsit_node:
      if (subtype(p) ≡ baseline_node) goto recycle_p;
      else if (subtype(p) ≠ hset_node ∧ subtype(p) ≠ vset_node ∧ subtype(p) ≠
          hpack_node ∧ subtype(p) ≠ vpack_node ∧ subtype(p) ≠
          graf_node ∧ subtype(p) ≠ disp_node ∧ subtype(p) ≠
          image_node ∧ subtype(p) ≠ align_node) break;
                                                  /* else fall through */
    case hlist_node: case vlist_node: case rule_node:
      if (page_contents ≡ empty) { freeze_page_specs(box_there); hfix_defaults( );
      }
      else page_contents = box_there;
      break;
    case ins_node:
      if (page_contents ≡ empty) { freeze_page_specs(inserts_only);
        hfix_defaults( );
      }
      break;
    case kern_node: case penalty_node: case glue_node: goto recycle_p;
    default: break;
    }
  }
```

Used in 47.

Users of TEX often force the generation of empty pages for example to start a new chapter on a right hand page with an odd page number. This makes sense for a printed book but not for a screen reader where there are no page numbers nor right or left hand pages. Using a screen reader, empty pages are just annoying. The common way to achive an empty page is the use of `\eject` followed by a an empty box, a fill glue, and another `\eject`.

The following code tries to detect such a sequence of nodes and will eliminate them if requested. To do so, we delay the output of nodes after an eject penalty until either something gets printed on the page or another eject penalty comes along. To override the delayed output, a penalty less or equal to a double *eject_penalty* can be used. The function *its_all_over* (see section 3.2) is an example for such a use.

⟨ suppress empty pages if requested 49 ⟩ ≡ (49)

```
  if (option_no_empty_page ∧ ((type(p) ≡ penalty_node ∧ penalty(p) ≤ eject_penalty ∧
      penalty(p) > 2*(eject_penalty)) ∨ (page_contents ≡ empty ∧ ¬is_visible(p))))
    { pointer q = link(p);
```

```
  while (true) {
    if (q ≡ null) return;
    else if (is_visible(q)) break;
    else if (type(q) ≡ penalty_node ∧ penalty(q) ≤ eject_penalty) {
      while (p ≠ q) { pointer r = p;
        DBG(DBGPAGE, "Eliminating␣node␣(%d,%d)\n", type(p),
            type(p) ≡ penalty_node ? penalty(p) : subtype(p)); p = link(p);
        link(r) = null; flush_node_list(r);
      }
      link(contrib_head) = p;
      DBG(DBGPAGE, "Eliminating␣empty␣page␣done\n");
      if (penalty(q) ≤ 2 * (eject_penalty)) break;
    }
    q = link(q);
  }
}
```

Used in 47.

It remains to test a node for visibility. This is a quick (and dirty) test because the test will not look inside a boxes; it simply tests whether the list pointer is *null*.

⟨ HiTEX auxiliar routines 42 ⟩ +≡ (50)

```
static bool is_visible(pointer p)
{
  switch (type(p)) {
  case penalty_node: case kern_node: case glue_node: case mark_node:
    return false;
  case hlist_node: case vlist_node: return list_ptr(p) ≠ null;
  case whatsit_node:
    if (subtype(p) ≡ image_node ∨ subtype(p) ≡ align_node ∨ subtype(p) ≡
        disp_node) return true;
    else if (subtype(p) ≡ hset_node ∨ subtype(p) ≡ vset_node ∨ subtype(p) ≡
        hpack_node ∨ subtype(p) ≡ vpack_node) return list_ptr(p) ≠ null;
    else if (subtype(p) ≡ graf_node) return graf_list(p) ≠ null;
    else return false;
  default: return true;
  }
}
```

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 parameters 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. *hhsize* and *hvsize* will contain the sizes that a regular TEX engine would use.

We also compute the total page size from the page template defined last.

⟨ Compute the page size 51 ⟩ ≡ (51)

```
{ pointer p;
  p = link(setpage_head);
  if (p ≡ null) { scaled margin;
    if (hhsize < hvsize) margin = hhsize;
    else margin = hvsize;
    margin = margin/6 − 6 ∗ unity;
    if (margin < 0) margin = 0;
    page_h = hhsize + 2 ∗ margin; page_v = hvsize + 2 ∗ margin;
  }
  else { pointer x;
    x = setpage_height(p); page_v = xdimen_width(x) + round(((double)
        xdimen_hfactor(x)∗hhsize+(double) xdimen_vfactor(x)∗hvsize)/unity);
    x = setpage_width(p); page_h = xdimen_width(x) + round(((double)
        xdimen_hfactor(x)∗hhsize+(double) xdimen_vfactor(x)∗hvsize)/unity);
  }
}
```

Used in 73.

⟨ HiTEX variables 52 ⟩ ≡ (52)

```
static scaled page_h, page_v;
```

Used in 164.

⟨ Switch *hsize* and *vsize* to extended dimensions 53 ⟩ ≡ (53)

```
hsize = 0; vsize = 0; dimen_par_hfactor(hsize_code) = unity;
dimen_par_vfactor(vsize_code) = unity;
```

Used in 74.

4.3 Replacing `hpack` and `vpack`

⟨ HiTEX routines 35 ⟩ +≡ (54)

```
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) ≡ stream_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) = new_xdimen(w, hf, vf);
      else set_extent(q) = new_xdimen(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 (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) = new_xdimen(w, hf , vf ); return q;
  }
  }
```

⟨ HiTEX routines 35 ⟩ +≡ (55)

```
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 */
  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) ≡ stream_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) = new_xdimen(h, hf, vf);
  else set_extent(q) = new_xdimen(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 (xj0) */
  { 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 (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) = new_xdimen(h, hf , vf ); return q;
  }
  }
```

4.4 Streams

HINT stream numbers start at 0 for the main text and continue upwards. TEX, on the other hand, numbers insertions starting with `box255` for the main text and continues downwards. Some mapping is needed, and we use the array *insert2stream* to map TEX's insert numbers to HINT stream numbers. The predefined stream for the main content has stream number 0.

```
⟨ HiTEX variables 52 ⟩ +≡                                    (56)
  int insert2stream[#100] = {0};
```

The following function returns the stream number for a given insert number i with $255 > i \geq 0$. A new stream number is allocated if necessary. Note that no overflow test is necessary since TEX allocates less than 233 inserts. The initial value of *max_ref*[*stream_kind*] is 0 and therefore stream number 0, reserved for the main content is never allocated. Stream definitions might also be loaded as part of a format file. Then the maximum stream number ist stored in *max_stream*. So if we do not find a stream number in the *insert2stream* array, we scan the stream definitions once and cache the associations found there.

```
⟨ HiTEX routines 35 ⟩ +≡                                     (57)
  int hget_stream_no(int i)
  { static bool init = false;
    int s;
    if (i ≡ 0) return 0;
    s = insert2stream[i];
    if (s ≠ 0) return s;
    if (¬init) { pointer t, s;
      for (t = link(setpage_head); t ≠ null; t = link(t))
        for (s = setpage_streams(t); s ≠ null; s = link(s))
          insert2stream[setstream_insertion(s)] = setstream_number(s);
      max_ref[stream_kind] = max_stream; init = true;
    }
    s = insert2stream[i];
    if (s ≡ 0) s = insert2stream[i] = max_ref[stream_kind] = ++max_stream;
    return s;
  }
```

4.5 Stream Definitions

A stream definition is stored as a whatsit node with subtype *setstream_node* as defined insection 7.15. Given a pointer p to such a node, here are the macros used to access the data stored there:

- *setstream_number*(p) the HINT stream number n.
- *setstream_insertion*(p) the corresponding TEX insertion number i.
- *setstream_max*(p) the maximum height x: This extended dimension is the maximum size per page for this insertion.

- *setstream_mag*(p) the magnification factor f: Inserting a box of height h will contribute $h * f/1000$ to the main page.
- *setstream_prefered*(p) the prefered stream p: If $p \geq 0$ we move the insert to stream p if possible.
- *setstream_next*(p) the next stream n: If $n \geq 0$ we move the insert to stream n if it can not be accomodated otherwise.
- *setstream_ratio*(p) the split ratio r: If $r > 0$ split the final contribution of this streams between stream p and n in the ratio $r/1000$ for p and $1 - r/1000$ for n before contributing streams p and r to the page.
- *setstream_before*(p) the "before" list b: For a nonempty stream the material that is added before the stream content.
- *setstream_after*(p) the "after" list a: For a nonempty stream, material that is added after the stream conten.
- *setstream_topskip*(p) the top skip glue t: This glue is inserted between the b list and the stream content and ajusted for the height for the first box of the stream content.
- *setstream_width*(p) the width w: This extended dimension is the width used for example to break paragraphs in the stream content into lines.
- *setstream_height*(p) a glue specification h reflecting the total height, strechability and shrinkability of the material in lists a and b.

Currently HiTEX handles only normal streams. First or last streams will come later.

The stream definition nodes are created and initialized with the following function:

⟨ HiTEX routines 35 ⟩ +≡ (58)

```
pointer new_setstream_node(uint8_t n)
{ pointer p = get_node(setstream_node_size);
  type(p) = whatsit_node; subtype(p) = setstream_node;
  setstream_insertion(p) = n; setstream_number(p) = hget_stream_no(n);
  setstream_mag(p) = 1000; setstream_prefered(p) = 255;
  setstream_next(p) = 255; setstream_ratio(p) = 0;
  setstream_max(p) = new_xdimen(0, 0, ONE);
  setstream_width(p) = new_xdimen(0, ONE, 0);
  setstream_topskip(p) = zero_glue; add_glue_ref(zero_glue);
  setstream_height(p) = zero_glue; add_glue_ref(zero_glue);
  setstream_before(p) = null; setstream_after(p) = null; return p;
}
```

The prefered stream, the next stream, and the split ratio are scanned as part of the `\setstream` primitive as described in section 3.10. When TEX finds the right brace that terminates the stream definition, it calls *handle_right_brace*. Then it is time to obtain the remaining parts of the stream definition. For insertion class i, we can extract the maximum height x of the insertions from the corresponding `dimen`i

register the magnification factor f from the `count`i register, and the total height h from the `skip`i register. The width w is taken from `\hsize` and the topskip t from `\topskip`.

⟨ HiTEX routines 35 ⟩ +≡ (59)

```
void hfinish_stream_group(void)
{ pointer s;
  end_graf( ); s = hget_current_stream( );
  if (s ≠ null) { pointer t;
    uint8_t i;
    i = setstream_insertion(s); setstream_mag(s) = count(i);
    setstream_width(s) = new_xdimen(dimen_par(hsize_code),
        dimen_par_hfactor(hsize_code), dimen_par_vfactor(hsize_code));
    t = zero_glue; add_glue_ref(t); delete_glue_ref(setstream_topskip(s));
    setstream_topskip(s) = t; t = skip(i); add_glue_ref(t);
    delete_glue_ref(setstream_height(s)); setstream_height(s) = t;
    setstream_max(s) = new_xdimen(dimen(i), dimen_hfactor(i),
        dimen_vfactor(i));
  }
  unsave( ); flush_node_list(link(head)); pop_nest( );
}
```

The before list b and the after list a are defined using the `\before` and `\after` primitives. When the corresponding list has ended with a right brace, TEX calls *handle_right_brace* and we can store the lists.

⟨ HiTEX routines 35 ⟩ +≡ (60)

```
void hfinish_stream_before_group(void)
{ pointer s;
  end_graf( ); s = hget_current_stream( );
  if (s ≠ null) setstream_before(s) = link(head);
  unsave( ); pop_nest( );
}
void hfinish_stream_after_group(void)
{ pointer s;
  end_graf( ); s = hget_current_stream( );
  if (s ≠ null) setstream_after(s) = link(head);
  unsave( ); pop_nest( );
}
```

4.6 Page Template Definitions

The data describing a page template is stored in a whatsit node with subtype *setpage_node* as defined insection 7.15. Given a pointer p to such a node, here are the macros used to access the data stored there:

- *setpage_name*(p): The name of the page template can be used in the user interface of a HINT viewer.

- *setpage_number*(p): The number of the page template that is used in the HINT file to reference this page template.
- *setpage_id*(p): The number of the page template that is used in TEX to reference this page template.
- *setpage_priority*(p): The priority helps in selecting a page template.
- *setpage_topskip*(p): The topskip glue is added at the top of a page and adjusted by the height of the first bos on the page.
- *setpage_height*(p): The height of the full page including the margins.
- *setpage_width*(p): The width of the full page including the margins.
- *setpage_depth*(p): The maximum depth of the page content. If the last box is deeper than this maximum, the difference is subtracted from the height of the page body.
- *setpage_list*(p): The list that defines the page template. After the page builder has completed a page this list is scanned and page body and nonempty streams are added at the corresponding insertion points.
- *setpage_streams*(p): The list of stream definitions that belong to this page template.

To allow TEX to use arbitrary numbers between 1 and 255 for the page templates while in HINT the numbers of page templates are best consecutive from 1 to $max\_ref\,[page\_kind] \equiv max\_page$, we let TEX assign an id and generate the template number. Because templates might be in format files, the variable *max_page* will hold the true number.

The function *new_setpage_node* is called with the page template id $0 < i < 256$ and a string number for the name n. It allocates and initializes a node if necessary and moves it to the front of the list of templates.

```
⟨ HiTEX routines 35 ⟩ +≡                                                (61)
  pointer new_setpage_node(uint8_t i, str_number n)
  { pointer p, prev_p;
    prev_p = setpage_head;
    for (p = link(prev_p); p ≠ null; prev_p = p, p = link(p))
      if (setpage_id(p) ≡ i) break;
    if (p ≡ null) ⟨ allocate a new setpage_node p 62 ⟩
    else link(prev_p) = link(p);
    link(p) = link(setpage_head); link(setpage_head) = p; return p;
  }

⟨ allocate a new setpage_node p 62 ⟩ ≡                                  (62)
  { p = get_node(setpage_node_size); type(p) = whatsit_node;
    subtype(p) = setpage_node;
    setpage_number(p) = max_ref[page_kind] = ++max_page; setpage_id(p) = i;
    setpage_name(p) = n; setpage_priority(p) = 1;
    setpage_topskip(p) = zero_glue; add_glue_ref(zero_glue);
    setpage_height(p) = new_xdimen(0, 0, ONE);
```

```
    setpage_width(p) = new_xdimen(0, ONE, 0); setpage_depth(p) = max_depth;
    setpage_list(p) = null; setpage_streams(p) = null;
  }
```
Used in 61.

The default values are replaced by parameters given to the `\setpage` primitive (see section 3.10) and by the current values of certain TEX registers when finishing the page template. page template itself,

⟨ HiTEX routines 35 ⟩ +≡ (63)
```
  void hfinish_page_group(void)
  { uint8_t k;
    pointer p, q, r;
    end_graf( ); p = hget_current_page( );
    if (p ≠ null) { delete_glue_ref(setpage_topskip(p));
      setpage_topskip(p) = top_skip; add_glue_ref(top_skip);
      setpage_depth(p) = max_depth; flush_node_list(setpage_list(p));
      setpage_list(p) = link(head);
    }
    unsave( ); pop_nest( );
  }
```

⟨ HiTEX auxiliar routines 42 ⟩ +≡ (64)
```
  static pointer hget_current_page(void)
  { pointer p = link(setpage_head);
    if (p ≡ null) print_err("end␣of␣output␣group␣without␣setpage␣node");
    return p;
  }
  static pointer hget_current_stream(void)
  { pointer p, s;
    p = hget_current_page( );
    if (p ≡ null) return null;
    s = setpage_streams(p);
    if (s ≡ null)
      print_err("end␣of␣setstream␣group␣without␣setstream␣node");
    return s;
  }
```

5 HINT Output

5.1 Initialization

```
⟨ HiTEX routines 35 ⟩ +≡                                                    (65)
  static void hout_init(void)
  { new_directory(dir_entries); new_output_buffers(); max_section_no = 2;
    hdef_init(); hput_content_start(); ⟨ insert an initial language node 135 ⟩
  }
  void hint_open(void)
  { pack_job_name(make_tex_string(".hnt"));
    hout = tl_open_out((const char *) name_of_file + 1, "wb");
    if (hout ≡ NULL) {
      fprintf(stderr, "Unable to open %s\n", name_of_file + 1); exit(1);
    }
    hlog = stderr; option_global = true; hout_init();
  }
```

5.2 Termination

```
⟨ HiTEX routines 35 ⟩ +≡                                                    (66)
  static void hout_terminate(void)
  { hput_content_end(); hput_definitions(); hput_directory();
    hput_hint("created by HiTeX Version "HITEX_VERSION);
  }
  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 directory 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 67 ⟩ ≡ (67)
```
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 71.

⟨ Allocate a new directory entry 68 ⟩ ≡ (68)
```
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 71.

⟨ HiTEX macros 69 ⟩ ≡ (69)
```
#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 164.

⟨ HiTEX variables 52 ⟩ +≡ (70)
```
static int dir_entries = 4;
```

⟨ HiTEX auxiliar routines 42 ⟩ +≡ (71)
```
static uint16_t hnew_file_section(char *file_name)
{ uint16_t i;
  ⟨ Find an existing directory entry 67 ⟩
  ⟨ Allocate a new directory entry 68 ⟩
  dir[i].file_name = strdup(file_name); return i;
}
```

The following function uses TEX's function *pack_file_name* to create a new filename from a name n, a direcory or "area" a, and an extension e. TEX will truncate the new filename to *file_name_size* characters without warning. The new function will take a *name_length* equal to *file_name_size* as an indication that truncation has taken place and terminates the program. The return value converts a Pascal array, starting with index 1, into a C array staring with index 0.

⟨ HiTEX auxiliar routines 42 ⟩ +≡ (72)
```
static char *hfile_name(str_number n, str_number a, str_number e)
{ pack_file_name(n, a, e);
  if (name_length ≥ file_name_size)
    QUIT("File name too long %d >= %d", name_length, file_name_size);
  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 used 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 35 ⟩ +≡                                        (73)
  void hfix_defaults(void)
  { int i;
    DBG(DBGDEF,"Freezing␣HINT␣file␣defaults\n");
    ⟨ Compute the page size 51 ⟩
    ⟨ Fix definitions for integer parameters 78 ⟩
    ⟨ Fix definitions for dimension parameters 84 ⟩
    ⟨ Fix definitions for glue parameters 95 ⟩
    ⟨ Fix definitions of page templates 123 ⟩
  }
```

- 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 extensive 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 plus four byte of overhead per instance.

Here we initialize the necessary data structures for definitions.

```
⟨ HiTEX routines 35 ⟩ +≡                                              (74)
  static void hdef_init(void)
  { int i;
    ⟨ Switch hsize and vsize to extended dimensions 53 ⟩
    ⟨ Initialize definitions for extended dimensions 88 ⟩
    ⟨ Initialize definitions for baseline skips 101 ⟩
    ⟨ Initialize definitions for fonts 116 ⟩
#if 0
    overfull_rule = 0;                          /* no overfull rules please */
#endif
  }
```

After all definitions are ready, we write them using the function *hput_definitions*. When we output the definitions, we have to make sure to define references before we use them. This is achived by using a specific ordering of the definitions in the function *hput_definitions* and by preventing the allocation of new definitions as soon as the output of the definition section has started. The latter has the aditional benefit that the maximum values do no longer change.

```
⟨ HiTEX routines 35 ⟩ +≡                                              (75)
  static void hput_definitions( )
                          /* write the definitions into the definitions buffer */
  { int i;
    uint32_t d, m, s;
    hput_definitions_start( ); hput_max_definitions( );
    ⟨ Output language definitions 136 ⟩
    ⟨ Output font definitions 120 ⟩
    ⟨ Output integer definitions 81 ⟩
    ⟨ Output dimension definitions 86 ⟩
    ⟨ Output extended dimension definitions 91 ⟩
    ⟨ Output glue definitions 98 ⟩
    ⟨ Output baseline skip definitions 104 ⟩
    ⟨ Output parameter list definitions 114 ⟩
    ⟨ Output hyphen definitions 110 ⟩
    ⟨ Output page template definitions 125 ⟩
    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 52 ⟩ +≡ (76)

```
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 52 ⟩ +≡ (77)

```
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 */
  floating_penalty_no                              /* floating_penalty_code 42 */
  };
```

6.1.3 Parameters

Now we can generate the definitions for integer parameters:

```
⟨ Fix definitions for integer parameters 78 ⟩ ≡                                (78)
  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 73.
```

6.1.4 Allocation

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

```
⟨ HiTEX auxiliar routines 42 ⟩ +≡                                              (79)
  static 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 ∧ section_no ≡ 2) { 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 69 ⟩ +≡                                                    (80)
#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 81 ⟩ ≡                                      (81)
  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 75.
```

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 52 ⟩ +≡                                                (82)
  static scaled dimen_defined[#100] = {0};
```

6.2.2 Mapping

```
⟨ HiTEX variables 52 ⟩ +≡                                                (83)
  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 */
  split_max_depth_no,                              /* 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 84 ⟩ ≡                    (84)
  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] = page_h;
  dimen_defined[vsize_dimen_no] = page_v;
  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 73.
```

6.2.4 Allocation

```
⟨ HiTEX auxiliar routines 42 ⟩ +≡                                    (85)
  static 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 ∧ section_no ≡ 2) { m = ++max_ref[dimen_kind];
      dimen_defined[m] = s; return m;
    }
    else return -1;
  }
```

6.2.5 Output

⟨ Output dimension definitions 86 ⟩ ≡ (86)
```
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 75.

6.3 Extended Dimensions

6.3.1 Data

⟨ HiTEX variables 52 ⟩ +≡ (87)
```
static struct {
  scaled w, h, v;
} xdimen_defined[#100];
```

6.3.2 Initialization

⟨ Initialize definitions for extended dimensions 88 ⟩ ≡ (88)
```
for (i = 0; i ≤ max_fixed[xdimen_kind]; i++) {
  xdimen_defined[i].w = xdimen_defaults[i].w;
  xdimen_defined[i].h = ONE * xdimen_defaults[i].h;
  xdimen_defined[i].v = ONE * xdimen_defaults[i].v;
}
```
Used in 74.

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 auxiliar routines 42 ⟩ +≡ (89)
```
int hget_xdimen_no(pointer p)
{ int i;
  for (i = 0; i ≤ max_ref[xdimen_kind]; i++) {
    if (xdimen_defined[i].w ≡ xdimen_width(p) ∧ xdimen_defined[i].h ≡
          xdimen_hfactor(p) ∧ xdimen_defined[i].v ≡ xdimen_vfactor(p))
      return i;
  }
  if (section_no ≠ 2) return −1;
  if (i ≥ #100) return −1;
```

```
    max_ref[xdimen_kind] = i; xdimen_defined[i].w = xdimen_width(p);
    xdimen_defined[i].h = xdimen_hfactor(p);
    xdimen_defined[i].v = xdimen_vfactor(p); return i;
  }
```

⟨ HiTEX routines 35 ⟩ +≡ (90)

```
  pointer new_xdimen(scaled w, scaled h, scaled v)
  { pointer p = get_node(xdimen_node_size);
    type(p) = whatsit_node; subtype(p) = xdimen_node; xdimen_width(p) = w;
    xdimen_hfactor(p) = h; xdimen_vfactor(p) = v; return p;
  }
```

6.3.4 Output

⟨ Output extended dimension definitions 91 ⟩ ≡ (91)

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

Used in 75.

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 52 ⟩ +≡ (93)

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

6.4.2 Mapping

⟨ HiTEX variables 52 ⟩ +≡ (94)

```
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 95 ⟩ ≡ (95)

```
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 73.

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 42 ⟩ +≡ (96)

```
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 35 ⟩ +≡ (97)

```
static int hget_glue_no(pointer p)
{ static int rover = 0;
  int i;
  if (p ≡ zero_glue) return zero_skip_no;
  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 ∧ section_no ≡ 2) {
    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 98 ⟩ ≡ (98)

```
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);
```

The above code uses the following conversion routine. While HINT supports glue that depends on `hsize` and `vsize`, this is currently not supported by HiTEX.

⟨ HiTEX auxiliar routines 42 ⟩ +≡ (99)

```
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 52 ⟩ +≡                                              (100)
  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 101 ⟩ ≡                    (101)
  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 74.
```

6.5.3 Allocation

```
⟨ HiTEX routines 35 ⟩ +≡                                               (102)
  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 ∧ section_no ≡ 2) 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 42 ⟩ +≡                                    (103)
  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 104 ⟩ ≡                           (104)
  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 75.

6.5.5 Printing

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

⟨ HiTEX routines 35 ⟩ +≡ (105)
```
  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 52 ⟩ +≡ (106)
```
  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 42 ⟩ +≡ (107)
```
  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 35 ⟩ +≡ (108)

```
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 ∨ section_no ≠ 2) return −1;
  rover = ++max_ref[hyphen_kind]; hy_defined[rover] = copy_disc_node(p);
  ⟨ Allocate font numbers for glyphs in the pre- and post-break lists 109 ⟩
  return rover;
}
```

When we allocate hyphen numbers we might have fonts inside the pre- or post-break list, that never show up anywhere else in the content. These fonts would then be undefined once we start the definition section. So we have to make sure, all necessary fonts get defined.

⟨ Allocate font numbers for glyphs in the pre- and post-break lists 109 ⟩ ≡ (109)

```
ensure_font_no(pre_break(p)); ensure_font_no(post_break(p));
```

Used in 108.

6.6.3 Output

⟨ Output hyphen definitions 110 ⟩ ≡ (110)

```
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 75.

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 52 ⟩ +≡ (111)
#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 35 ⟩ +≡ (112)
  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 ∨ section_no ≠ 2) 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 35 ⟩ +≡ (113)

```
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]);
  }
  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 114 ⟩ ≡ (114)

```
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 75.

6.8 Fonts

It seems I need: (see email by Karl)

kpse_find_file(*name*, *kpse_fontmap_format*, *false*); with name "ps2pk.map" or "psfonts.map" or "ttfonts.map" or cmfonts.map to get from the tfm name the postscript name. then I get the *psfont_name* ususaly its the same font name with .pfb appended so I can skip it

6.8.1 Data

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 HINT font numbers to TEX font numbers, the **font_t** contains the *i* field; to map TEX font numbers to HINT font numbers, we use the array *hmap_font*.

⟨ HiTEX variables 52 ⟩ +≡ (115)

```
#define MAX_FONTS 100
  typedef struct {
    uint8_t i;                                  /* the 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 metric file */
    uint16_t y;                                 /* section number of font glyph file */
  } font_t;
  static font_t *hfonts[MAX_FONTS] = {NULL};
  static int hmap_font[MAX_FONTS];
```

⟨ Initialize definitions for fonts 116 ⟩ ≡ (116)

```
  for (i = 0; i < 100; i++) hmap_font[i] = -1;
  max_ref[font_kind] = -1;
```

Used in 74.

6.8.2 Allocation

Allocation of a **font_t** record takes place when we translate a TEX font number to a HINT font number using the function *hget_font_no*, and while doing so discover that the corresponding HINT font number does not yet exist. Because the **font_t** structure must be initialized after allocating it, we start with some auxiliar routines for that purpose.

⟨ HiTEX auxiliar routines 42 ⟩ +≡ (117)

```
  static 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;
}
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])
{                                                    /* used only for texts */
}
```

In the following code, f is a TEX internal font number and g is the corresponding HINT font number. TEX's null-font, a kind of undefined font containing no characters is replaced by HINT's font number zero. Actually the nullfont should never appear in the output, but if it does so, either an error message or a more sensible replacement font might be in order.

⟨ HiTEX auxiliar routines 42 ⟩ +≡ (118)

```
static uint8_t hget_font_no(uint8_t f)
{ int g;
  char *n, *fn;
  int l;
  if (f ≡ 0) { DBG(DBGFONT, "TeX nullfont -> 0\n"); return 0; }
  g = hmap_font[f]; DBG(DBGFONT, "Mapping 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");
  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 = tl_find_tfm(fn); hfonts[g]→m = hnew_file_section(n); free(n);
  n = tl_find_glyph(fn); hfonts[g]→y = hnew_file_section(n); free(n); return g;
}
```

Surprisingly, not all characters that occur in a HINT file are inside the content section; some characters might hide in the definition section inside the pre- or post-break list of a predefined discretionary hyphen. To make sure that the fonts

necessary for these characters are included in the final HINT file, we check these lists to make sure all TEX font numbers have a corresponting HINT font number.

⟨ HiTEX auxiliar routines 42 ⟩ +≡ (119)

```
static void ensure_font_no(pointer p)
{
  while (p ≠ null) {
    if (is_char_node(p)) hget_font_no(font(p));
    else if (type(p) ≡ hlist_node ∨ type(p) ≡ vlist_node)
      ensure_font_no(list_ptr(p));
    p = link(p);
  }
}
```

6.8.3 Output

⟨ Output font definitions 120 ⟩ ≡ (120)

```
{ 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[g]); 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]);
    else HPUT32(font_dsize[g]);
    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 75.

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

⟨ HiTEX auxiliar routines 42 ⟩ +≡ (121)

```
void hout_string(int s)
{ 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 69 ⟩ +≡                                                    (122)
#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);
  }
```

6.9 Page Templates

Once we start producing content nodes, we update the maximum numbers of page templates and streams from *max_page* and *max_stream*. These values might have changed because the templates were stored in a format file.

```
⟨ Fix definitions of page templates 123 ⟩ ≡                              (123)
  max_ref[page_kind] = max_page; max_ref[stream_kind] = max_stream;
                                                                  Used in 73.
```

6.9.1 Output

As part of a page template, we will see stream insertion nodes. When we encounter an *stream_node* inside a template definition, we output a stream insertion point.

```
⟨ cases to output whatsit content nodes 124 ⟩ ≡                          (124)
case stream_node: HPUT8(setstream_number(p)); tag = TAG(stream_kind, b100);
  break;                                                           Used in 149.
```

```
⟨ Output page template definitions 125 ⟩ ≡                               (125)
  DBG(DBGDEF, "Maximum␣page␣template␣reference:␣%d\n", max_page);
  { pointer t;
    for (t = link(setpage_head); t ≠ null; t = link(t)) {
      uint32_t pos = hpos − hstart;
      DBG(DBGDEF, "Defining␣page␣template␣%d\n", setpage_number(i));
      hpos++; HPUTNODE;                    /* space for the tag and the node */
      HPUT8(setpage_number(t)); hout_string(setpage_name(t));
      HPUT8(setpage_priority(t)); hout_glue_node(setpage_topskip(t));
```

```
    hput_dimen(setpage_depth(t)); hout_xdimen_node(setpage_height(t));
    hout_xdimen_node(setpage_width(t)); hout_list_node2(setpage_list(t));
    ⟨ output stream definitions 126 ⟩
    hput_tags(pos, TAG(page_kind, 0));
  }
}
```

Used in 75.

As part of the output of page template definitions, we output stream definitions:

⟨ output stream definitions 126 ⟩ ≡ (126)

```
{ pointer p, q;
  p = setpage_streams(t);
  while (p ≠ null) { uint8_t n;
    n = setstream_number(p);
    DBG(DBGDEF, "Defining␣stream␣%d␣at␣"SIZE_F"\n", n, hpos − hstart);
    HPUTTAG(stream_kind, b100); HPUT8(n);
    hout_xdimen_node(setstream_max(p));                    /* maximum height */
    HPUT16(setstream_mag(p));                                        /* factor */
    HPUT8(setstream_prefered(p));                                  /* prefered */
    HPUT8(setstream_next(p));                                          /* next */
    HPUT16(setstream_ratio(p));                                        /* ratio */
    q = setstream_before(p); setstream_before(p) = null; hout_list_node2(q);
    flush_node_list(q); hout_xdimen_node(setstream_width(p));
    q = setstream_topskip(p); hout_glue_node(q); delete_glue_ref(q);
    q = setstream_after(p); setstream_after(p) = null; hout_list_node2(q);
    flush_node_list(q); q = setstream_height(p); hout_glue_node(q);
    delete_glue_ref(q); HPUTTAG(stream_kind, b100); p = link(p);
  }
}
```

Used in 125.

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 35 ⟩ +≡                                                (127)
  void hout_node(pointer p)
  { uint32_t pos = hpos − hstart;
    uint8_t tag;
    HPUTNODE; hpos++;
    if (is_char_node(p)) ⟨ output a character node 128 ⟩
    else
      switch (type(p))
      { ⟨ cases to output content nodes 129 ⟩
        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 [14]).

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 128 ⟩ ≡                                       (128)
  { glyph_t g;
    g.f = hget_font_no(font(p)); g.c = character(p); tag = hput_glyph(&g);
  }                                                                  Used in 127.
```

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 129 ⟩ ≡                                    (129)
case penalty_node:
  { int n, i;
    i = penalty(p);
    if (i > 10000) i = 10000;
    else if (i < −10000) i = −10000;
    n = hget_int_no(i);
    if (n < 0) tag = hput_int(i);
    else { HPUT8(n); tag = TAG(penalty_kind, 0);
    }
  }
  break;                                                          Used in 127.
```

7.3 Kerns

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

```
⟨ cases to output content nodes 129 ⟩ +≡                                   (130)
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 the reference number and returns false; otherwise it outputs the extended dimension and returns true.

```
⟨ HiTEX auxiliar routines 42 ⟩ +≡                                          (131)
  void hout_xdimen_node(pointer p)
```

```
{ xdimen_t x; x.w = xdimen_width(p); x.h = xdimen_hfactor(p)/(double) ONE;
  x.v = xdimen_vfactor(p)/(double) ONE; hput_xdimen_node(&x);
}
bool hout_xdimen(pointer p)
{ int n = hget_xdimen_no(p);
  if (n ≥ 0) { HPUT8(n); return false; }
  else { hout_xdimen_node(p); return true; }
}
```

7.5 Languages

We map the language numbers of TEX to HINT language numbers using the *hlanguage* array.

```
⟨ HiTEX variables 52 ⟩ +≡                                        (132)
  static struct {
    uint8_t n;
    str_number s;
  } hlanguage[#100];
```

For any language number of TEX, the following function returns the corresponding HINT language number. Since TEX knowns about a maximum of 255 languages, there is no need for overflow checking. The next function writes a language node to the output stream.

```
⟨ HiTEX auxiliar routines 42 ⟩ +≡                                (133)
  uint8_t hget_language_no(uint8_t n)
  { int i;
    for (i = 0; i ≤ max_ref[language_kind]; i++)
      if (hlanguage[i].n ≡ n) return i;
    i = ++max_ref[language_kind]; hlanguage[i].n = n; hlanguage[i].s = 0;
                                                  /* language unknown */
    return i;
  }
  uint8_t hout_language(uint8_t n)
  { n = hget_language_no(n);
    if (n < 7) return TAG(language_kind, n + 1);
    else { HPUT8(n); return TAG(language_kind, 0);
    }
  }
```

After these preparations, the output of a language node is simple:

```
⟨ cases to output whatsit content nodes 124 ⟩ +≡                 (134)
case language_node: tag = hout_language(what_lang(p)); break;
```

Normaly TEX does not produce an initial language node and then the language in the HINT file would not be known until it changes for the first time.

⟨ insert an initial language node 135 ⟩ ≡ (135)
```
{ uint32_t pos = hpos − hstart;
  hpos++; hput_tags(pos, hout_language(language));
}
```
Used in 65.

TEX offers currently no simple way to obtain a standardized language identifier for the current language. So if the string number of the language is zero, we output the string `"unknown"`; if somehow the language is known, we output the corresponding string from TEX's string pool.

⟨ Output language definitions 136 ⟩ ≡ (136)
```
DBG(DBGDEF, "Maximum␣language␣reference:␣%d\n", max_ref[language_kind]);
for (i = max_fixed[language_kind] + 1; i ≤ max_ref[language_kind]; i++) {
  HPUTNODE; HPUT8(TAG(language_kind, 0)); HPUT8(i);
  if (hlanguage[i].s ≡ 0) hput_string("unknown");
  else hout_string(hlanguage[i].s);
  HPUT8(TAG(language_kind, 0));
}
```
Used in 75.

7.6 Mathematics

TEX's math nodes have an optional width—a copy of the mathsurround parameter—while HINT math nodes do not. Therefore we have to add an explicit kern node if the width is nonzero. We add it before a "math on" node or after a "math off" to get the same behavior in respect to line breaking.

⟨ cases to output content nodes 129 ⟩ +≡ (137)
```
case math_node:
  { kern_tk; k.x = true; k.d.w = width(p); k.d.h = k.d.v = 0.0;
    if (subtype(p) ≡ before) { tag = TAG(math_kind, b111);
      if (width(p) ≠ 0) { hput_tags(pos, hput_kern(&k)); pos = hpos − hstart;
        HPUTNODE; hpos++;
      }
    }
    else { tag = TAG(math_kind, b011);
      if (width(p) ≠ 0) { hput_tags(pos, tag); pos = hpos − hstart; HPUTNODE;
        hpos++; tag = hput_kern(&k);
      }
    }
  }
  break;
```

7.7 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 42 ⟩ +≡ (138)

```
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 129 ⟩ +≡ (139)

```
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;
```

7.8 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 42 ⟩ +≡ (140)

```
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 129 ⟩ +≡ (141)

```
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;
```

7.9 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 129 ⟩ +≡ (142)

```
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;
```

7.10 Rules

⟨ cases to output content nodes 129 ⟩ +≡ (143)

```
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;
```

7.11 Boxes

```
⟨ cases to output content nodes 129 ⟩ +≡                                    (144)
  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)); hout_list_node2(list_ptr(p)); break;
```

7.12 Adjustments

```
⟨ cases to output content nodes 129 ⟩ +≡                                    (145)
  case adjust_node:
    { list_t l;
      l.k = adjust_kind; HPUT8(0);                                  /* size */
      tag = hout_list(adjust_ptr(p), pos + 1, &l);
    }
    break;
```

7.13 Insertions

TEX's insertions are mapped to HINT streams.

```
⟨ cases to output content nodes 129 ⟩ +≡                                    (146)
  case ins_node: ⟨ output stream content 147 ⟩
    break;
```

Here we consider stream content and come back to stream definitions in section . In a HINT stream content node the stream parameters *floating_penalty*, *split_max_depth*, and *split_top_skip* are optional. If omitted, the defaults from the stream definition are used. This is probably also for TEX the most common situation. It is, however, possible to supply more than one page template with different defaults and while not very common, TEX might change the parameters at any time. Because we dont know which is the current page template, it is not posible to compare the current parameter values against the defaults, and we have to supply all the parameters always. In a future version, we might have a TEX primitive that allows us to signal "use the defaults".

```
⟨ output stream content 147 ⟩ ≡                                    (147)
  { int k, n;
    uint32_t pos;
    list_t l;
    info_t i = b000;
    k = subtype(p); n = hget_stream_no(k); HPUT8(n); link(temp_head) = null;
    add_par_node(int_type, floating_penalty_code, float_cost(p));
    add_par_node(dimen_type, split_max_depth_code, depth(p));
    add_par_node(glue_type, split_top_skip_code, split_top_ptr(p));
    pos = hpos − hstart; l.k = param_kind;
    n = hout_param_list(link(temp_head), pos, &l);
    flush_node_list(link(temp_head)); link(temp_head) = null;
    if (n ≥ 0) HPUT8(n);
    else i = b010;
    hout_list_node2(ins_ptr(p)); tag = TAG(stream_kind, i);
  }                                                                Used in 146.
```

7.14 Marks

We currently ignore Marks.

```
⟨ cases to output content nodes 129 ⟩ +≡                           (148)
case mark_node: hpos −−; return;
```

7.15 Whatsit Nodes

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

```
⟨ cases to output content nodes 129 ⟩ +≡                           (149)
case whatsit_node:
  switch (subtype(p)) { ⟨ cases to output whatsit content nodes 124 ⟩
  default:
    MESSAGE("\nOutput␣of␣whatsit␣nodes␣subtype=%d␣not␣implemented\n",
        subtype(p)); display_node(p); MESSAGE("End␣of␣node"); hpos −−;
    return;
  }
  break;
```

For TEX's original whatsit nodes no output is generated; hence, we simply remove the tag byte that is already in the output. In a later version of the HINT format the language information will be retained in the output, because it simplifies the implementation of a good text to audio translation of HINT files.

```
⟨ cases to output whatsit content nodes 124 ⟩ +≡                   (150)
case open_node: case write_node: case close_node:
                                  /* out_what(p); this does not work for LaTeX */
  case special_node: hpos −−; return;
```

7.16 Paragraphs

```
⟨ cases to output whatsit content nodes 124 ⟩ +≡ (151)
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;
```

7.17 Baseline Skips

```
⟨ cases to output whatsit content nodes 124 ⟩ +≡ (152)
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;
```

7.18 Displayed Equations

```
⟨ cases to output whatsit content nodes 124 ⟩ +≡ (153)
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(math_kind, i);
          /* the display_no_bs(p) tells whether the baseline skip is ignored */
  }
  break;
```

7.19 Extended Boxes

When we output an extended box, we have to consider a special case: the page templates. Page templates are boxes that contain insertion points. These insertion points look like regular insertions but with an empty content list. As a result the *hpack* and *vpackage* routines might belive that they can compute the dimensions of the box conntent when in fact they can not.

⟨ cases to output whatsit content nodes 124 ⟩ +≡ (154)

```
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); hout_list_node2(list_ptr(p)); tag = TAG(k, i);
  }
  break;
```

⟨ cases to output whatsit content nodes 124 ⟩ +≡ (155)

```
case hpack_node: case vpack_node:
  { kind_t k = (subtype(p) ≡ hpack_node ? hpack_kind : vpack_kind);
    info_t i = b000;
    int n = pack_extent(p);
    if (hout_xdimen(n)) i |= b100;
    if (pack_m(p) ≡ additional) i |= b001;
    if (shift_amount(p) ≠ 0) { HPUT32(shift_amount(p)); i |= b010;
    }
    if (k ≡ vpack_kind) HPUT32(pack_limit(p));
    hout_list_node2(list_ptr(p)); tag = TAG(k, i);
  }
  break;
```

7.20 Extended Alignments

⟨ cases to output whatsit content nodes 124 ⟩ +≡ (156)

```
case align_node:
  { info_t i = b000;
    if (align_m(p) ≡ additional) i |= b001;
    if (align_v(p)) i |= b010;
    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 42 ⟩ +≡ (157)

```
void hout_preamble(pointer p)
{ pointer q, r;
  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);
  }
  hout_list_node2(p); 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 42 ⟩ +≡ (158)

```
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 129 ⟩ +≡ (159)
case *unset_node*: **case** *unset_set_node*: **case** *unset_pack_node*:

7.21 Images

⟨ cases to output whatsit content nodes 124 ⟩ +≡ (160)
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;

7.22 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 35 ⟩ +≡ (161)
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*));
}
static void *hout_list_node2*(**pointer** *p*)
{ **list_t** *l*;
uint32_t *pos*;
pos = *hpos* − *hstart*; *l.k* = *list_kind*; *hout_list_node*(*p*, *pos*, &*l*);
}

7.23 Parameter Lists

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

⟨ HiTEX routines 35 ⟩ +≡ (162)

```
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;
}
```

7.24 Text

The routines in this section are not yet ready.

⟨ HiTEX routines 35 ⟩ +≡ (163)

```
#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(FONT0_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
```

Appendix

A Source Files

A.1 Basic types

To define basic types we use the file `bastypes.h` from [14].

A.2 HiTEX routines: `hitex.c`

(164)

```
#include <stdio.h>
#include <kpathsea/kpathsea.h>
#include "basetypes.h"
#include "error.h"
#include "hformat.h"
#include "textypes.h"
#include "texvars.h"
#include "texfuncs.h"
#include "hput.h"
#include "tltex.h"
#include "hitex.h"
  extern scaled hhsize, hvsize;
  ⟨ HiTEX macros 69 ⟩
  ⟨ HiTEX variables 52 ⟩
  ⟨ HiTEX auxiliar routines 42 ⟩
  ⟨ HiTEX routines 35 ⟩
```

A.3 HiTEX prototypes: hitex.h

⟨ hitex.h 165 ⟩ ≡ (165)

```
extern void hint_open(void);
extern void hint_close(void);
extern void hout_node(pointer p);
extern void hfix_defaults(void);
extern pointer new_xdimen(dimen_t w, 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 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 hout_list_node2(pointer p);
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 166 ⟩ ≡ (166)

```
#ifndef _TEX_VARS_H
#define _TEX_VARS_H_
  extern int bad;
  extern ASCII_code xord[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 packed_ASCII_code str_pool[pool_size + 1];
  extern pool_pointer str_start[max_strings + 1];
  extern pool_pointer pool_ptr;
  extern pool_pointer init_pool_ptr;
  extern uint8_t interaction;
  extern memory_word mem0[mem_max − mem_min + 1], *const mem;
  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 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 list_state_record cur_list;
  extern memory_word *const eqtb;
  extern memory_word *const hfactor_eqtb;
  extern memory_word *const vfactor_eqtb;
  extern scaled par_shape_hfactor, par_shape_vfactor;
  extern two_halves*const hash; extern int line ;
  extern str_number job_name;
  extern memory_word font_info[font_mem_size + 1];
  extern scaled *const font_size;
  extern scaled *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 pointer font_glue0[font_max − font_base + 1], *const font_glue;
  extern int hyphen_char0[font_max − font_base + 1], *const hyphen_char;
  extern int *const char_base;
  extern int *const width_base;
  extern int *const height_base;
  extern int *const depth_base;
  extern int param_base0[font_max − font_base + 1], *const param_base;
  extern quarterword c, f;
  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 pointer cur_p;
  extern halfword last_special_line;
  extern scaled first_width;
  extern scaled second_width;
  extern scaled first_indent;
  extern scaled second_indent;
  extern ASCII_code cur_lang, init_cur_lang;
  extern int l_hyf, r_hyf, init_l_hyf, init_r_hyf;
#ifdef INIT
  extern bool trie_not_ready;
#endif
  extern uint8_t page_contents;
```

```
  extern scaled page_so_far[8];
  extern font_index main_k;
  extern pointer main_p;
#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 167 ⟩ ≡ (167)

```
#ifndef _TEX_FUNCS_H_
#define _TEX_FUNCS_H_
  extern void input_add_arg(char *str);
  extern str_number make_string(void);
  extern void print_scaled(scaled s);
  extern halfword badness(scaled t, scaled s);
  extern pointer get_node(int s);
  extern void free_node(pointer p, halfword s);
  extern pointer new_rule(void);
  extern pointer new_disc(void);
  extern pointer new_spec(pointer p);
  extern pointer new_param_glue(small_number n);
  extern pointer new_penalty(int m);
  extern void short_display(int p);
  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_glue_ref(pointer p);
  extern void flush_node_list(pointer p);
  extern pointer copy_node_list(pointer p);
  extern void pop_nest(void);
  extern void begin_diagnostic(void);
  extern void pack_file_name(str_number n, str_number a, str_number e);
  extern void pack_job_name(str_number s);
  extern pointer new_character(internal_font_number f, eight_bits c);
  extern pointer hpack(pointer p, scaled w, scaled hf, scaled
      vf, small_number m);
  extern void append_to_vlist(pointer b);
  extern void freeze_page_specs(small_number s);
  extern void initialize(void);
  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 print_nl(char *s);
  extern void print_err_str(char *s);
```

```
  extern void print_int(int n);
  extern void overflow(char *s, int n);
  extern void confusion(str_number s);
  extern void end_diagnostic(bool blank_line);
  extern void unsave(void);
  extern void line_break(int final_widow_penalty);
  extern void hyphenate(void);
  extern void init_trie(void);
  extern void build_page(void);
  extern void new_graf(bool indented);
  extern void end_graf(void);
  extern pointer vpackage(pointer p, scaled h, scaled hf, scaled
      vf, small_number m, scaled l);                    /* new procedures */
  extern void hyphenate_word(void);
  extern void update_last_values(pointer p);
  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_setpage_node(uint8_t k, str_number n);
  extern void hfinish_page_group(void);
  extern void hfinish_stream_group(void);
  extern void hfinish_stream_before_group(void);
  extern void hfinish_stream_after_group(void);
  extern pointer new_setstream_node(uint8_t n);
  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 int hget_stream_no(int i);
  extern void display_node(pointer p);              /* generated by htex.ch */
  extern void hint_open(void);
  extern void hint_close(void);
  extern pointer new_xdimen(scaled w, scaled h, scaled v);
  extern void hget_image_information(pointer p);
  extern bool ktex_commandline(int argc, char *argv[]);
  extern char *tl_find_tfm(const char *filename);
  extern char *tl_find_glyph(const char *filename);
#endif
```

Crossreference of Code

⟨ Allocate a new directory entry ⟩ Defined in section 68. Used in section 71.
⟨ Allocate font numbers for glyphs in the pre- and post-break lists ⟩
Defined in section 109. Used in section 108.
⟨ Compute the page size ⟩ Defined in section 51. Used in section 73.
⟨ Create the parameter node ⟩ Defined in section 36. Used in section 38.
⟨ Find an existing directory entry ⟩ Defined in section 67. Used in section 71.
⟨ Fix definitions for dimension parameters ⟩ Defined in section 84. Used in section 73.
⟨ Fix definitions for glue parameters ⟩ Defined in section 95. Used in section 73.
⟨ Fix definitions for integer parameters ⟩ Defined in section 78. Used in section 73.
⟨ Fix definitions of page templates ⟩ Defined in section 123. Used in section 73.
⟨ HiTEX auxiliar routines ⟩ Defined in section 42, 43, 44, 45, 46, 50, 64, 71, 72, 79, 85, 89, 96, 99, 103, 107, 117, 118, 119, 121, 131, 133, 138, 140, 157, and 158. Used in section 164.
⟨ HiTEX macros ⟩ Defined in section 69, 80, and 122. Used in section 164.
⟨ HiTEX routines ⟩ Defined in section 35, 38, 39, 41, 47, 54, 55, 57, 58, 59, 60, 61, 63, 65, 66, 73, 74, 75, 90, 97, 102, 105, 108, 112, 113, 127, 161, 162, and 163. Used in section 164.
⟨ HiTEX variables ⟩ Defined in section 52, 56, 70, 76, 77, 82, 83, 87, 93, 94, 100, 106, 111, 115, and 132. Used in section 164.
⟨ Initialize definitions for baseline skips ⟩ Defined in section 101. Used in section 74.
⟨ Initialize definitions for extended dimensions ⟩ Defined in section 88.
Used in section 74.
⟨ Initialize definitions for fonts ⟩ Defined in section 116. Used in section 74.
⟨ Initialize the parameter node ⟩ Defined in section 37. Used in section 38.
⟨ Output baseline skip definitions ⟩ Defined in section 104. Used in section 75.
⟨ Output dimension definitions ⟩ Defined in section 86. Used in section 75.
⟨ Output extended dimension definitions ⟩ Defined in section 91. Used in section 75.
⟨ Output font definitions ⟩ Defined in section 120. Used in section 75.
⟨ Output glue definitions ⟩ Defined in section 98. Used in section 75.
⟨ Output hyphen definitions ⟩ Defined in section 110. Used in section 75.
⟨ Output integer definitions ⟩ Defined in section 81. Used in section 75.
⟨ Output language definitions ⟩ Defined in section 136. Used in section 75.
⟨ Output page template definitions ⟩ Defined in section 125. Used in section 75.
⟨ Output parameter list definitions ⟩ Defined in section 114. Used in section 75.
⟨ Switch *hsize* and *vsize* to extended dimensions ⟩ Defined in section 53.
Used in section 74.
⟨ TEX Live functions ⟩ Defined in section 11, 17, 27, 29, 30, 31, and 32.
Used in section 34.

⟨ TEX Live support functions ⟩ Defined in section 1, 5, 6, 20, 23, 25, and 28. Used in section 34.
⟨ TEX Live variables ⟩ Defined in section 7, 8, and 16. Used in section 34.
⟨ allocate a new *setpage_node p* ⟩ Defined in section 62. Used in section 61.
⟨ cases to output content nodes ⟩ Defined in section 129, 130, 137, 139, 141, 142, 143, 144, 145, 146, 148, 149, and 159. Used in section 127.
⟨ cases to output whatsit content nodes ⟩ Defined in section 124, 134, 150, 151, 152, 153, 154, 155, 156, and 160. Used in section 149.
⟨ enable the generation of input files ⟩ Defined in section 26. Used in section 17.
⟨ explain command line ⟩ Defined in section 2. Used in section 1.
⟨ explain options ⟩ Defined in section 3 and 4. Used in section 1.
⟨ fix the use of parshape = 1 indent length ⟩ Defined in section 40. Used in section 39.
⟨ freeze the page specs if called for ⟩ Defined in section 48. Used in section 47.
⟨ handle the option at *option_index* ⟩ Defined in section 9, 10, 12, 13, 14, and 15. Used in section 6.
⟨ `hitex.h` ⟩ Defined in section 165.
⟨ insert an initial language node ⟩ Defined in section 135. Used in section 65.
⟨ output a character node ⟩ Defined in section 128. Used in section 127.
⟨ output stream content ⟩ Defined in section 147. Used in section 146.
⟨ output stream definitions ⟩ Defined in section 126. Used in section 125.
⟨ parse options ⟩ Defined in section 18. Used in section 17.
⟨ set defaults from the `texmf.cfg` file ⟩ Defined in section 22. Used in section 17.
⟨ set the format name ⟩ Defined in section 24. Used in section 17.
⟨ set the input file name ⟩ Defined in section 21. Used in section 17.
⟨ set the program name ⟩ Defined in section 19. Used in section 17.
⟨ suppress empty pages if requested ⟩ Defined in section 49. Used in section 47.
⟨ `texfuncs.h` ⟩ Defined in section 167.
⟨ `texvars.h` ⟩ Defined in section 166.
⟨ `tltex.c` ⟩ Defined in section 34.
⟨ `tltex.h` ⟩ Defined in section 33.

References

[1] Karl Berry et al. TEX Live. http:///www.tug.org/texlive/.

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

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

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

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

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

[7] 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`.

[8] Donald E. Knuth. *The TEX book.* Computers & Typesetting, Volume A. Addison-Wesley Publishing Company, 1984.

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

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

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

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

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

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

Index