TEXprofile:
Profiling TEX Source Files

TEXprofile
Version 1.0

Profiling TEX Source Files

Für Beatriz

MARTIN RUCKERT *Munich University of Applied Sciences*

First edition

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.
Profiling TEX Source Files
Includes index.

The internet page http://hint.userweb.mwn.de/ may contain current information about this book, downloadable software, and news.

martin.ruckert@hm.edu

Preface

This program is a result not of need or necessity but of curiosity. Being used to have a profiler at my disposal whenever I write a program, I just wanted to know if a profiler for TEX could be written and what it would tell me. So when I was considering what could be my contribution to the TUG 2024 conference in Prague, I thought "How about writing a profiler for TEX?" So I started the project and completed it just in time for the conference.

The following document describes how to use `texprofile`—the program that displays the data gathered by `texprof`—in a readable form.

In a future version, this document will also contain a "User Manual" that contains just the information needed by the person who wants to find out how TEX is spending its time when processing a document. Until then be brave and try to cope with all the extra information contained in the following document!

München
May 28, 2024 *Martin Ruckert*

Contents

1 The *main* Program

The program presented here reads data collected by a run of `texprof` and presents it in a readable way. The structure of the program is similar to a lot of C programs:

```
  # include < stdlib.h >
#include <stdio.h>
#include <stdint.h>
#include <stdbool.h>
#include <ctype.h>
#include <string.h>
  ⟨ declarations 2 ⟩                                                    (1)
  ⟨ error handling 103 ⟩
  ⟨ functions 5 ⟩
  int main(int argc, char *argv[])
  {
    int i, k;
    ⟨ process the command line 95 ⟩
    ⟨ read the input file 3 ⟩
    ⟨ write the output file 94 ⟩
    return 0;
  }
```

2 Reading the Input File

Let us start with the interesting stuff and read the input file.

```
⟨ declarations 2 ⟩ ≡                                                    (2)
  static char *input_file_name = NULL;
  static FILE *in = NULL;
```

Used in 1.

First we need to open the input file. If that is successful, we check that there is a "file marker": a special sequence of characters that we expect at the very beginning of the input file and at various other places in the file just to make sure we have a file that is really meant to be read by `texprofile`.

```
#define FILE_MARKER "TEX␣PROF"
```

```
⟨ read the input file 3 ⟩ ≡                                             (3)
  if (input_file_name ≡ NULL) commandline_error("no␣input␣file␣given");
  in = fopen(input_file_name, "rb");
  if (in ≡ NULL) {
    char *tmp = malloc(strlen(input_file_name) + 7);
    if (tmp ≠ NULL) {
      strcpy(tmp, input_file_name);
      strcat(tmp, ".tprof");
      input_file_name = tmp;
      in = fopen(input_file_name, "rb");
    }
  }
  if (in ≡ NULL)
    error ("Unable␣to␣open␣input␣file") ;
  check_file_marker("The␣input␣file␣does␣not␣seem␣to␣contain\
      ␣TeX␣profile␣data");
```

Used in 1.

After the initial file marker we read the whole file which is organized in four sections separated by file markers.

```
⟨ read the input file 3 ⟩ +≡                                            (4)
  ⟨ read size data 7 ⟩
  check_file_marker("Invalid␣size␣data");
  ⟨ read file names 11 ⟩
  check_file_marker("Invalid␣file␣names");
```

```
⟨ read macro names 14 ⟩
check_file_marker("Invalid␣macro␣names");
⟨ read timing data 26 ⟩
check_file_marker("Invalid␣timing␣data");
fclose(in);
```

Before we explain how to read the different sections, we define functions to read multi-byte numbers in big endian format.

```
⟨ functions 5 ⟩ ≡                                                    (5)
  static uint8_t fget1(void)
  {
    unsigned char buf[1];
    if (fread(buf, 1, 1, in) ≠ 1)
      error ("Unexpected␣end␣of␣input") ;
    return (uint8_t) buf[0];
  }
  static uint16_t fget2(void)
  {
    unsigned char buf[2];
    if (fread(buf, 1, 2, in) ≠ 2)
      error ("Unexpected␣end␣of␣input") ;
    return ((uint16_t) buf[0] ≪ 8) + buf[1];
  }
  static uint32_t fget4(void)
  {
    unsigned char buf[4];
    if (fread(buf, 1, 4, in) ≠ 4)
      error ("Unexpected␣end␣of␣input") ;
    return ((((((uint32_t) buf[0] ≪ 8) + buf[1]) ≪ 8) + buf[2]) ≪ 8) + buf[3];
  }
```

Used in 1.

2.1 Reading the Size Data

Here are the variables for the size data and the code to read them.

```
⟨ declarations 2 ⟩ +≡                                                (6)
  static unsigned int file_num,                    /* number of input files */
  file_name_num,              /* number of byte needed for the file names */
  macro_num,                                  /* number of macros defined */
  macro_name_num,          /* number of byte needed for the macro names */
  stamp_num,                                   /* number of time stamps */
  depth_num;                                    /* maximum stack depth */
```

⟨ read size data 7 ⟩ ≡ (7)
```
file_num = fget2( );
file_name_num = fget2( );
stamp_num = fget4( );
if (stamp_num ≡ 0)
  error ("The␣input␣file␣contains␣no␣time␣samples") ;
macro_num = fget2( );
macro_name_num = fget4( );
depth_num = fget2( );
```
Used in 4.

We conclude the reading of size data by using these values to allocate properly sized arrays.

⟨ read size data 7 ⟩ +≡ (8)
```
⟨ allocate arrays 10 ⟩
```

2.2 Reading the File Names

The first section in the input file contains the full names of all TEXinput files processed by `texprof`. We need arrays of pointers to the full file names and to the short file names and a character array to store the actual file names.

⟨ declarations 2 ⟩ +≡ (9)
```
static char **full_file_names, **file_names, *file_name_buffer;
#define ALLOCATE(V, S)V = calloc(S, sizeof (*V));
  if (V ≡ NULL) error ("Out␣of␣memory␣for␣"#V"["#S"]") ;
```

⟨ allocate arrays 10 ⟩ ≡ (10)
```
ALLOCATE(full_file_names, file_num);
ALLOCATE(file_names, file_num);
ALLOCATE(file_name_buffer, file_name_num);
```
Used in 8.

The first three files are not ordinary files but are used for special purposes: File number 0 is used for tokens that `texprof` can not associate with an input file. These are usually only the tokens that are loaded from a TEX format files since format files produced by TEX's `\dump` primitive do not contain file and line information. File number 1 is used for the "system" file. Its "line numbers" identify not a "line" but the kind of action that was taken by the system. File number 2 is used for the "terminal" file. It is used for text that is entered either on the command line or in an interactive run of TEX.

```
#define unknown_file  0                    /* unknown origin */
#define system_file  1                     /* generated by the system */
#define terminal_file  2                   /* read from the terminal */
#define DIR_SEPARATOR  '/'
```

⟨ read file names 11 ⟩ ≡ (11)
```
i = k = 0;
while (k < file_num) {
```

```
    char c;
    full_file_names[k] = file_names[k] = file_name_buffer + i;
    do {
      c = fget1();
      if (i ≥ file_name_num)
        error ("File␣name␣buffer␣overflow") ;
      file_name_buffer[i++] = c;
      if (c ≡ DIR_SEPARATOR) file_names[k] = file_name_buffer + i;
    } while (c ≠ 0);
    k++;
  }
```

Used in 4.

2.3 Reading the Macro Names

After the file names, we read the macro names and construct the *macro_map* table. It maps the multi-letter macro numbers from TEX's hash table to the sequential numbering used to access the *macro_names* array. This map is used when reading the time stamps to map TEX's control sequence numbers to the sequential numbers needed to access the macro names. After reading the time stamps, the *macro_map* array can be deallocated.

⟨ declarations 2 ⟩ +≡ (12)

```
  static char *macro_name_buffer, **macro_names;
  static uint16_t *macro_stack;
  static struct map {
    uint16_t n;                                   /* name */
    uint16_t m;                                   /* macro definition */
  } *macro_map;
```

⟨ allocate arrays 10 ⟩ +≡ (13)

```
  ALLOCATE(macro_names, macro_num);
  ALLOCATE(macro_name_buffer, macro_name_num);
  ALLOCATE(macro_map, undefined_control_sequence + every_eof_text + 1);
  ALLOCATE(macro_stack, depth_num + 1);
```

⟨ read macro names 14 ⟩ ≡ (14)

```
  i = 0;
  k = 0;
  while (k < macro_num) {
    int j;
    char c;
    j = fget2();
    macro_map[j].n = k;
    macro_map[j].m = 0;
    macro_names[k++] = macro_name_buffer + i;
    do {
```

```
    c = fget1();
    if (i ≥ macro_name_num)
      error ("Name␣buffer␣overflow") ;
    macro_name_buffer[i++] = c;
  } while (c ≠ 0);
}
for (i = 1; i ≤ every_eof_text; i++) {
  macro_map[undefined_control_sequence + i].n = macro_num + i;
  macro_map[undefined_control_sequence + i].m = 0;
}
```

Used in 4.

To print a macro name, we use the function *print_cs*. Given a control sequence number n, it can be either from an active character, or from a single letter control sequence or from a multi-letter control sequence. If the number n comes from an active character, we get the character by subtracting *active_base*; If it comes comes from a single character control sequence, we get the character by subtracting *single_base*; if $n$is equal to *null_cs*, it comes from the empty control sequence; and if it comes from a multi-letter control sequence, n is already mapped to the macro name index plus *hash_base*.

```
#define active_base 1 /* beginning of region 1, for active character
    equivalents */
#define single_base (active_base + 256) /* equivalents of one-character control
    sequences */
#define null_cs (single_base + 256) /* equivalent of \csname\endcsname */
#define hash_base (null_cs + 1) /* beginning of region 2, for the hash table */
#define hash_size 45000 /* maximum number of control sequences; it should
    be at most about (mem_max − mem_min)/(double) 10 */
#define frozen_control_sequence (hash_base + hash_size) /* for error recovery */
#define frozen_null_font (frozen_control_sequence + 11) /* permanent
    '\nullfont' */
#define undefined_control_sequence (frozen_null_font + 257) /* dummy
    location */
#define output_text 6
#define every_eof_text 15
```

⟨ functions 5 ⟩ +≡ (15)

```
void print_cs(int n)
{
  if (n < hash_base) {
    if (n < single_base) {
      int c = n − active_base;
      if (c < 0) printf("File");
      else if (isprint(c)) printf("%c", c);
      else printf("0x%02x", c);
    }
```

```
    else if (n < null_cs) {
      int c = n − single_base;
      if (isprint(c)) printf("\\%c", c);
      else printf("\\0x%02x", c);
    }
    else printf("Empty");
  }
  else if (n ≥ macro_num + hash_base)
    printf("\\%s", every_name[n − hash_base − macro_num − output_text]);
  else printf("\\%s", macro_names[n − hash_base]);
}
```

2.4 Reading the Time Stamps

The last section contains the time stamps collected during the run of `texprof`. The time stamps are stored for later processing in the *stamps* array. In addition to the timing data, the time stamps section contains information about macro calls.

⟨ declarations 2 ⟩ +≡ (16)

```
struct stamp {
  uint8_t c;                                              /* command */
  uint8_t f;                                              /* file */
  uint16_t l;                                             /* line */
  uint16_t d;                                             /* depth */
  uint16_t m;                                             /* macro */
  uint32_t t;                                             /* time */
} *stamps;
struct macro_def {
  uint16_t n;                                             /* name */
  uint8_t f;                                              /* file */
  uint16_t l;                                             /* line */
  uint16_t link;                                  /* link to another macro_def */
  int a;                                  /* number of concurrent activation records */
  int count;                                              /* number of calls */
  uint16_t e;                                             /* list of outgoing edges */
  uint64_t t;                                             /* time used for this macro */
  uint64_t T;                                 /* cumulative time spent in this macro */
  uint64_t s;                                             /* start time of first activation */
} *macro_defs;
int macro_def_num;                                        /* macros allocated */
int macro_def_count;                                      /* macros used */
```

Since the *macro_defs* array will contain only those macros that actually occur in the timing data, it is hard to predict the size of the *macro_defs* array. So either a first pass over all the timing data or a dynamic array is needed. Here the decision was to use a dynamic array.

```
⟨ allocate arrays 10 ⟩ +≡                                              (17)
  ALLOCATE(stamps, stamp_num);
  macro_def_num = 16;
  macro_def_count = 1;                          /* zero element is a sentinel */
  ALLOCATE(macro_defs, macro_def_num);

⟨ functions 5 ⟩ +≡                                                     (18)
#define REALLOC(A, C, N)
  {
    N = N * 1.4142136 + 0.5;                                          /* √2 */
    A = realloc(A, N * sizeof (*A));
    if (A ≡ NULL)
      error ("Out of memory") ;
    memset(A + C, 0, (N − C) * sizeof (*A));
  }
  int new_macro_def(void)
  {
    if (macro_def_count < macro_def_num) return macro_def_count++;
    REALLOC(macro_defs, macro_def_count, macro_def_num);
    return macro_def_count++;
  }
```

When reading the timing data, we will allocate for each new macro call a new macro definition with its name and the file and line of its definition. One problem to overcome here is that TEX allocates a macro name in the hash table only once for every name even if that name is reused for various macros in different files or in different lines. Here we need a unique number for each of these macros. For simplification, however, we will use the same number if two macros with the same name are defined in the same file and the same line. If you need to distinguish these macros, move the definitions to separate lines (or use different names because it will not only confuse the profiler but also the readers of your program).

To convert the file, line, and control sequence number to a unique macro definition number, we use the *macro_map* array to store the macro definition number and link multiple macro definitions that share the same name into a linked list using the *link* field of the macro definition.

```
⟨ functions 5 ⟩ +≡                                                     (19)
  int get_macro_def(int f, int l, int c)
  {
    int m = macro_map[c].m;
    if (m ≡ 0) {
      m = new_macro_def( );
      macro_map[c].m = m;
    }
    else {
    loop:
      if (macro_defs[m].f ≡ f ∧ macro_defs[m].l ≡ l) return m;
```

```
    else if (macro_defs[m].link ≡ 0) {
      int n = new_macro_def( );
      macro_defs[m].link = n;
      m = n;
    }
    else {
      m = macro_defs[m].link;
      goto loop;
    }
  }
  if (c < hash_base) macro_defs[m].n = c;
  else macro_defs[m].n = hash_base + macro_map[c].n;
  macro_defs[m].f = f;
  macro_defs[m].l = l;
  macro_defs[m].link = 0;
  return m;
}
```

To print a macro name we use the function *print_macro*. Because the name will not uniquely identify a macro—TEX allows using the same name for different macros—there is an option to print the file and line number in square brackets after the macro name.

⟨ declarations 2 ⟩ +≡ (20)

```
bool opt_macro_id = false;
```

⟨ process options 21 ⟩ ≡ (21)

```
case 'i': opt_macro_id = true;
  break;
```

Used in 95.

⟨ explain format options 22 ⟩ ≡ (22)

```
"-i␣␣␣␣add␣the␣macros␣file␣and␣line␣numbers␣after␣the␣macro␣\
    name\n"
```

Used in 96.

⟨ functions 5 ⟩ +≡ (23)

```
void print_macro(int i)
{
  print_cs(macro_defs[i].n);
  if (opt_macro_id) printf("␣[%d,%d]", macro_defs[i].f, macro_defs[i].l);
}
```

Now we can read the time stamps and the changes to the macro stack. Each element of the timing data starts with a one byte command value which is followed by a variable number of data bytes.

After the *system_macro_pop* command follows a two byte new stack depth. After all other commands follows a one byte file number and a two byte line number.

The *system_macro_push* command is then followed by a two byte new stack depth and a two byte control sequence number. All remaining commands are followed by a four byte elapsed time in nano seconds.

```
⟨ declarations 2 ⟩ +≡                                                    (24)
  uint64_t total_time;                          /* sum of all command times */
  uint32_t total_num;                           /* number of all command times */
  uint64_t *file_time, *cmd_time;
  int *file_line;                          /* maximum line number/offsets per file */
  int *cmd_freq;
  int cur_depth;
```

```
#define POP_BIT #80
⟨ read timing data 26 ⟩ ≡                                                (26)
  ALLOCATE(file_time, file_num);
  ALLOCATE(cmd_time, cmd_num);
  ALLOCATE(file_line, file_num + 1);
  ALLOCATE(cmd_freq, cmd_num);
  cur_depth = −1;
  total_time = 0;
  total_num = 0;
  macro_defs[0].a = 1;
  i = 0;
  {
    uint8_t c = fget1( );                                  /* the initial call */
    ⟨ read a macro call 28 ⟩
    if (c ≠ system_macro_push)
      error ("Timing␣data␣must␣start␣with␣an␣initial␣push␣command") ;
    ⟨ time a push command 60 ⟩
  }
  while (i < stamp_num) {
    uint8_t c = fget1( );
    switch (c) {
    case system_macro_pop_0: break;
    case system_macro_pop_0 + 1: case system_macro_pop_0 + 2:
      case system_macro_pop_0 + 3: case system_macro_pop_0 + 4: case
          system_macro_pop_0 + 5: case system_macro_pop_0 + 6:
      case system_macro_pop_0 + 7: case system_macro_pop_0 + 8: case
          system_macro_pop_0 + 9: case system_macro_pop_0 + 10:
      {
        int d = c − system_macro_pop_0;
        ⟨ store pop d 27 ⟩
        break;
      }
    case system_macro_pop_small:
```

```
    {
      int d = fget1();
      ⟨ store pop d 27 ⟩
      break;
    }
  case system_macro_pop:
    {
      int d = fget2();
      ⟨ store pop d 27 ⟩
      break;
    }
  case system_macro_push:
    {
      ⟨ read a macro call 28 ⟩
      cur_depth++;
      macro_stack[cur_depth] = m;
      macro_defs[m].count++;
      macro_defs[m].a++;
      if (macro_defs[m].a ≡ 1) {
        macro_defs[m].s = total_time;
      }
      start_child(macro_stack[cur_depth − 1], m);
    }
    break;
  default:
    if (c & POP_BIT) {
      int d = 1;
      ⟨ store pop d 27 ⟩
      c = c & ∼POP_BIT;
    }
    {
      uint8_t f = fget1();
      uint16_t l = fget2();
      uint32_t t = fget4();
      uint16_t m = macro_stack[cur_depth];
      if (f ≥ file_num)
        error ("File number out of range") ;
      if (c ≥ cmd_num)
        error ("Command number out of range") ;
      stamps[i].c = c;
      stamps[i].f = f;
      if (l > file_line[f]) file_line[f] = l;
      stamps[i].l = l;
      stamps[i].t = t;
      total_time += t;
```

```
        file_time[f] += t;
        macro_defs[m].t += t;
        cmd_time[c] += t;
        cmd_freq[c]++;
        total_num++;
        stamps[i].d = cur_depth;
        stamps[i].m = m;
      }
      i++;
#ifdef DEBUG
      printf("<%d:%d:%d> %s\n", i - 1, c, cur_depth, cmd_name[c]);
#endif
      break;
    }
  }
  macro_defs[macro_stack[0]].T = total_time;
  macro_defs[0].count = 0;
  macro_defs[0].T = 0;
  macro_defs[0].f = 0;
  macro_defs[0].l = 0;
#ifdef DEBUG
  printf("Finished reading %d commands, depth=%d\n", i, cur_depth);
#endif
```

Used in 4.

The table of *cmd_name*s is defined in the appendix.

⟨ store pop d 27 ⟩ ≡ (27)

```
  stamps[i].c = system_macro_pop;
  stamps[i].d = cur_depth - d;
#ifdef DEBUG
  printf("{%d:%d:%d>%d}\n", i, system_macro_pop, cur_depth + d, cur_depth);
#endif
  ⟨ time a pop command 61 ⟩
  i++;
```

Used in 26.

⟨ read a macro call 28 ⟩ ≡ (28)

```
  int f, l, n, m;
  f = fget1();
  l = fget2();
  n = fget2();
  m = get_macro_def(f, l, n);
  stamps[i].c = system_macro_push;
  stamps[i].f = f;
  stamps[i].l = l;
  stamps[i].d = cur_depth;
```

```
  stamps[i].m = m;
#ifdef DEBUG
  printf("[%d:%d<%d:", i, cur_depth, cur_depth + 1);
  print_macro(m);
  printf("]\n");
#endif
  i++;
```

Used in 26.

3 Writing the Output

For simplicity and because it is often sufficient to have a short look at the results, we send the output to the standard output stream. Various tables of output can be selected using options on the command line. For the selection of tables, we use uppercase letters like "`R`"; to modify the presentation of the tables, we use lower case letters like "`s`" or "`n`".

3.1 The Raw Time Stamps

Let us start with some simple options: "`-R`" will output the raw time stamps. This is useful if the profiler runs for a very short time or if a separate program wants to analyze the data. In the long list of time stamps the file names would only be distracting. So we print only the file numbers and show the file names in a separate table (see below). The option "-s" will show information about the macro stack along with the time stamps and the option "-n" will show the serial numbers, a feature that is mainly needed for debugging the `texprofile` program.

⟨ declarations 2 ⟩ +≡ (29)

```
bool opt_raw = false;
bool opt_raw_stack = false;
bool opt_raw_num = false;
```

⟨ process options 21 ⟩ +≡ (30)

```
case 'R': opt_raw = true;
  break;
case 's': opt_raw_stack = true;
  break;
case 'n': opt_raw_num = true;
  break;
```

⟨ explain table options 31 ⟩ ≡ (31)

```
"-R␣␣␣␣␣show␣the␣table␣of␣raw␣time␣stamps\n"
```

Used in 96.

⟨ explain format options 22 ⟩ +≡ (32)

```
"-s␣␣␣␣␣show␣the␣changes␣of␣the␣macro␣stack\n"
"-n␣␣␣␣␣show␣the␣time␣stamp␣numbers\n"
```

The following table shows the time in nano seconds, the file number, the line number and the command executed. Here and in the following tables, all fields

are separated by a TAB characters. So you can import the data in common office products for further processing (see also the option `-m` below).

```
⟨ show all time stamps if requested 33 ⟩ ≡                                (33)
  if (opt_raw) {
    Mprintf("\nAll␣time␣stamps␣in␣order␣of␣appearance:\n");
    if (opt_raw_num) Mprintf("number\t");
    Mprintf("file\t␣line\ttime/ns\tcommand\t␣␣␣␣␣␣␣␣\
        ␣␣␣␣␣␣␣␣level\tmacro\n");
    for (i = 1; i < stamp_num; i++) {
      int m = stamps[i].m;
      ⟨ extract t, f, l, and c from stamp i 34 ⟩
      if (c ≤ system_profile_off) {
        if (opt_raw_num) printf("%6d\t", i);
        print_fl(f, l);
        printf("\t%7u\t%-23s\t%5d\t", t, CMD_NAME(c), stamps[i].d);
        print_macro(m);
        printf("\n");
      }
      else if (opt_raw_stack) {
        if (opt_raw_num) printf("%6d\t", i);
        printf("\t\t\t%-23s\t%5d\t", CMD_NAME(c), stamps[i].d);
        if (c ≡ system_macro_push) print_macro(m);
        printf("\n");
      }
    }
  }
```

Used in 94.

```
⟨ extract t, f, l, and c from stamp i 34 ⟩ ≡                              (34)
  uint32_t t = stamps[i].t;
  uint16_t l = stamps[i].l;
  uint8_t f = stamps[i].f;
  uint8_t c = stamps[i].c;
```

Used in 33 and 69.

3.2 The Raw Macro Stack Changes

Further we can display the raw macro stack changes.

```
⟨ declarations 2 ⟩ +≡                                                     (35)
  bool opt_stack = false;
```

```
⟨ process options 21 ⟩ +≡                                                 (36)
case 'S': opt_stack = true;
  break;
```

⟨ explain table options 31 ⟩ +≡ (37)
```
"-S␣␣␣␣␣show␣the␣table␣of␣macro␣stack␣changes\n"
```

⟨ show the complete macro stack if requested 38 ⟩ ≡ (38)
```
if (opt_stack) {
  int c, d, m;
  Mprintf("\nThe␣macro␣stack␣a\
      nd␣its␣nesting:\n""command\tdepth\tname␣[id]\n");
  for (i = 0; i < stamp_num; i++) {
    c = stamps[i].c;
    if (c < system_macro_push) continue;
    d = stamps[i].d;
    m = stamps[i].m;
    printf("%5d\t%s\t", d, CMD_NAME(c));
    if (c ≡ system_macro_push) print_macro(m);
    printf("\n");
  }
}
```

Used in 94.

3.3 The Summary

Now we can print print the totals for the TEX run. Note that the option to show the summary is on by default, but can be switched off with the "`-N`" command line switch.

⟨ declarations 2 ⟩ +≡ (39)
```
bool opt_summary = true;
```

⟨ process options 21 ⟩ +≡ (40)
```
case 'N': opt_summary = false;
  break;
```

⟨ explain table options 31 ⟩ +≡ (41)
```
"-N␣␣␣␣␣do␣not␣show␣the␣table␣of␣global␣information\n"
```

⟨ show the total time if requested 42 ⟩ ≡ (42)
```
if (opt_summary) {
  printf("\n""Total␣time␣measured:␣␣␣␣␣␣␣␣%s\n", time_str(total_time));
  printf("Total␣number␣of␣samples:␣␣␣␣␣␣%6d\n", total_num);
  if (total_num > 0) printf("Average␣time␣per␣sample:␣␣␣%s\n",
      time_str(total_time/total_num));
  printf("Total␣number␣of␣files:␣␣␣␣␣␣␣␣%5d\n", file_num);
  printf("Total␣number␣of␣macros:␣␣␣␣␣␣␣␣%5d\n", macro_num);
  printf("Maximum␣stack␣nesting␣depth:␣␣␣%5d\n", depth_num);
}
```

Used in 94.

To print times over a wide range in a useful format, we use the following function. The option "`-m`" can be used to display plain nano seconds which is the preferable if you plan to import the data into another program for further processing. The macro *Mprintf* is used instead of *printf* for lines that should not be printed when *opt_machine* is true.

```
⟨ declarations 2 ⟩ +≡                                                    (43)
  bool opt_machine = false;
#define Mprintf(...) opt_machine ? (void) 0 : printf  (__VA_ARGS__)
```

```
⟨ process options 21 ⟩ +≡                                                (44)
case 'm': opt_machine = true;
  opt_summary = false;
  break;
```

```
⟨ explain format options 22 ⟩ +≡                                         (45)
  "-m␣␣␣␣optimize␣output␣for␣machine␣readability\n"
```

```
⟨ functions 5 ⟩ +≡                                                       (46)
  char *time_str(double t)
  {
    static char str[20];
    if (opt_machine) snprintf(str, 20, "%12ld", (long) t);
    else {
      if (t < 1000.0) snprintf(str, 20, "␣␣%7.2f␣ns", t);
      else if (t < 1000000.0) snprintf(str, 20, "␣␣%7.2f␣us", t/1000.0);
      else if (t < 1000000000.0) snprintf(str, 20, "␣␣%7.2f␣ms", t/1000000.0);
      else snprintf(str, 20, "␣␣%7.2f␣s", t/1000000000.0);
    }
    return str;
  }
```

We continue with a function to print file and line information in a readable way.

```
⟨ functions 5 ⟩ +≡                                                       (47)
  void print_fl(int f, int l)
  {
    if (f ≡ system_file ∧ ¬opt_machine) printf("system\t%7s", line_name[l]);
    else printf("%4d\t%5d", f, l);
  }
```

3.4 The File Summary

```
⟨ declarations 2 ⟩ +≡                                                    (48)
  bool opt_files = false;
```

```
⟨ process options 21 ⟩ +≡                                                (49)
case 'F': opt_files = true;
  break;
```

⟨ explain table options 31 ⟩ +≡ (50)
```
"-F␣␣␣␣show␣all␣files\n"
```

Next we can print a summary table for the files:

⟨ show the file summary if requested 51 ⟩ ≡ (51)
```
if (opt_files) {
  Mprintf("\nFiles␣in␣the␣order␣of␣appearance:\n");
  Mprintf("␣␣file\t␣lines\tpercent\t␣␣␣␣␣␣␣␣time""\tname\n");
  for (i = 0; i < file_num; i++) {
    double p = (100.0 * file_time[i])/total_time;
    if (p ≥ percent_limit) printf("%6d\t%6d\t%6.2f%%\t%s\t%s\n", i,
        file_line[i], p, time_str(file_time[i]), full_file_names[i]);
  }
}
```

Used in 94.

3.5 The Command Summary

And similar for commands:

⟨ declarations 2 ⟩ +≡ (52)
```
bool opt_cmd = false;
```

⟨ process options 21 ⟩ +≡ (53)
```
case 'C': opt_cmd = true;
  break;
```

⟨ explain table options 31 ⟩ +≡ (54)
```
"-C␣␣␣␣show␣table␣of␣command␣summaries\n"
```

⟨ show the command summary if requested 55 ⟩ ≡ (55)
```
if (opt_cmd) {
  int *cmd_link = NULL;
  ⟨ sort the commands by time 56 ⟩
  Mprintf("\nCommand␣summary:\n");
  Mprintf("␣cmd\t␣␣␣␣␣␣␣␣time\tpercent\t␣count\t␣␣\
      ␣␣␣␣average\tname\n");
  for (i = cmd_link[cmd_num]; i ≥ 0; i = cmd_link[i])
    if (cmd_freq[i] > 0) {
      printf("%4d\t%s\t%6.2f%%", i, time_str(cmd_time[i]),
          (100.0 * cmd_time[i])/total_time);
      printf("\t%6d\t␣%s\t%s\n", cmd_freq[i],
          time_str(cmd_time[i]/cmd_freq[i]), CMD_NAME(i));
    }
  free(cmd_link);
}
```

Used in 94.

To sort the commands in decreasing order of times, we allocate an array of links and sort the linked list using a simple insertion sort. We keep the links in *cmd_link* and the list head in *cmd_link*[*cmd_num*]. The end of the link is marked by -1.

```
⟨ sort the commands by time 56 ⟩ ≡                                    (56)
  ALLOCATE(cmd_link, cmd_num + 1);
  cmd_link[cmd_num] = −1;
  for (i = 0; i < cmd_num; i++) {
    int l = cmd_num;
    uint32_t t = cmd_time[i];
    while (cmd_link[l] ≥ 0 ∧ cmd_time[cmd_link[l]] > t) l = cmd_link[l];
    cmd_link[i] = cmd_link[l];
    cmd_link[l] = i;
  }
```

Used in 55.

3.6 The Macro Summary

Here is a table of all macros called. It should be possible to sort this table by either the direct time or be the cumulative time. Further there should be an option to switch between absolute times and relative times as a percentage of the total time.

```
⟨ declarations 2 ⟩ +≡                                                (57)
  bool opt_macro = false;
```

```
⟨ process options 21 ⟩ +≡                                            (58)
case 'M': opt_macro = true;
  break;
```

```
⟨ explain table options 31 ⟩ +≡                                      (59)
  "-M␣␣␣␣show␣the␣table␣of␣all␣macros␣called\n"
```

To compute the macro summary, we traverse the time stamps while reading. If there is a push command we do this:

```
⟨ time a push command 60 ⟩ ≡                                         (60)
  cur_depth++;
  macro_stack[cur_depth] = m;
  macro_defs[m].count++;
  macro_defs[m].a++;
  if (macro_defs[m].a ≡ 1) {
    macro_defs[m].s = total_time;
  }
```

Used in 26.

If there is a pop command we do this:

```
⟨ time a pop command 61 ⟩ ≡                                          (61)
  while (cur_depth > stamps[i].d) {
    int m = macro_stack[cur_depth];
```

```
    if (macro_defs[m].a ≡ 1) macro_defs[m].T += total_time − macro_defs[m].s;
    end_child(macro_stack[cur_depth − 1], m);
    macro_defs[m].a−−;
    cur_depth−−;
  }
```

Used in 27.

⟨ show the table of macros profiled if requested 62 ⟩ ≡ (62)

```
  if (opt_macro) {
    sort_macros();
    Mprintf("\nMacros␣profiled:\n");
    Mprintf("␣file\t␣line\tcalls\t␣time␣direct\t␣␣cu\
        mulative\tname\n");
    i = 0;
    do {
      printf("%5d\t%5d\t", macro_defs[i].f, macro_defs[i].l);
      printf("%5d\t%s", macro_defs[i].count, time_str(macro_defs[i].t));
      printf("\t%s\t", time_str(macro_defs[i].T));
      print_macro(i);
      printf("\n");
      i = macro_defs[i].link;
    } while (i ≠ 0);
  }
```

Used in 94.

To sort the edges by their cumulative time, we use a simple insertion sort. We use *edges*[0] as the head of a new list and insert the list of edges starting at *macro_defs*[*i*].*e*. While we do this we keep the estimates for the cumulative time of the edges that we have to compute anyway.

⟨ functions 5 ⟩ +≡ (63)

```
  void sort_edges(uint16_t p)
  {
    int i, j, k;
    edges[0].sibling = 0;                    /* head node becomes the empty list */
    i = macro_defs[p].e;
    while (i ≠ 0) {
      uint64_t Ti = edges[i].T;
      k = 0;
      while ((j = edges[k].sibling) ≠ 0) {
        uint64_t Tj = edges[j].T;
        if (Ti < Tj) k = j;
        else break;
      }
      edges[k].sibling = i;
      k = i;
```

```
      i = edges[i].sibling;
      edges[k].sibling = j;
    }
    macro_defs[p].e = edges[0].sibling;
  }
```

To sort the macros, we use again an insertion sort. So the procedure is similar to the one we have just seen. The value 0 is used as a sentinel and *macro_defs*[0].*link* is used as the list head.

⟨ functions 5 ⟩ +≡ (65)

```
  void sort_macros(void)
  {
    int i, k, j;
    static bool sorted = false;
    if (sorted) return;
    macro_defs[0].link = 0;
    sort_edges(0);
    for (i = 1; i < macro_def_count; i++) {
      uint64_t Ti = macro_defs[i].T;
      k = 0;
      while ((j = macro_defs[k].link) ≠ 0) {
        uint64_t Tj = macro_defs[j].T;
        if (Ti < Tj) k = j;
        else break;
      }
      macro_defs[i].link = j;
      macro_defs[k].link = i;
      sort_edges(i);
    }
    sorted = true;
  }
```

3.7 Cumulative Information for Lines

On the second pass over the time samples, we collect line based information. To store this information, we allocate a big array *total_line_time* containing the information for all lines and all files. To access a specific line, we convert the *file_line* array into an array of offsets for each file and add to it the line number.

⟨ declarations 2 ⟩ +≡ (66)

```
  int line_num = 0;
```

⟨ turn *file_line* into an array of offsets and compute the total number of lines 67 ⟩ (67)

```
    ≡
  line_num = 0;
  for (i = 0; i < file_num; i++) {
    int fl = file_line[i];
```

```
    file_line[i] = line_num;
    line_num = line_num + fl + 1;
  }
  file_line[file_num] = line_num;
```

Used in 69.

Now we are ready to compute the total time per line and the line frequency. For the frequency counts, we count any consecutive sequence of samples from the same file and line as one use of this line. So using multiple macros from the same line without intervening code from other lines will count only as a single access.

We need the following arrays:

```
⟨ declarations 2 ⟩ +≡                                                (68)
  static uint64_t *line_time = NULL;
  static int *line_freq = NULL;
```

The following function allocates the dynamic arrays just declared and fills them with content. Before doing so it checks that the arrays are not yet allocated because this would imply that the function was already called and there is nothing left to do. The information collected here is required for both, the *opt_lines* and the *opt_top_ten* tables, but there is no need to compute it twice.

```
⟨ functions 5 ⟩ +≡                                                   (69)
  void collect_line_time(void)
  {
    int i, cur_f = -1, cur_l = -1;
    if (line_time ≠ NULL) return;                    /* don't run a second time */
    ⟨ turn file_line into an array of offsets and compute the total number of lines
        67 ⟩
    ALLOCATE(line_time, line_num);
    ALLOCATE(line_freq, line_num);
    for (i = 0; i < stamp_num; i++) {
      ⟨ extract t, f, l, and c from stamp i 34 ⟩
      if (c ≤ system_profile_off) {
        line_time[file_line[f] + l] += t;
        if (cur_f ≠ f ∨ cur_l ≠ l) {
          line_freq[file_line[f] + l]++;
          cur_f = f;
          cur_l = l;
        }
      }
    }
  }
```

Printing all lines with their time is usually not required. We want only those lines that have a contribution that is above a certain limit given as the percentage of total time.

⟨ declarations 2 ⟩ +≡ (70)

```
double percent_limit = 1.0;
bool opt_lines = false;
```

⟨ process options 21 ⟩ +≡ (71)

```
case 'L': opt_lines = true;
  break;
case 'p':
  {
    char *endptr;
    percent_limit = strtod(option + 1, &endptr);
    if (endptr ≡ option + 1)
      commandline_error("-p<n> without a numeric argument <n>");
    else option = endptr − 1;
  }
  break;
```

⟨ explain table options 31 ⟩ +≡ (72)

```
"-L     show the table of times per input line\n"
```

⟨ explain format options 22 ⟩ +≡ (73)

```
"-p<n> don't show information for items with cumulative time\
    below <n> percent\n"
```

⟨ show lines above limit if requested 74 ⟩ ≡ (74)

```
if (opt_lines) {
  uint64_t limit = total_time * percent_limit/100.0;
  int k;
  collect_line_time( );
  Mprintf("\nLine summary:\n");
  if (percent_limit > 0)
    Mprintf("Only files and lines above %.2f%%:\n", percent_limit);
  for (k = i = 0; i < file_num; i++) {
    if (file_time[i] ≤ limit) k = file_line[i + 1];
    else {
      printf("%s\n", file_names[i]);
      printf("  file\t  line\tper\
          cent\t         time""\t count\t      average\n");
      printf("%6d\t\t%6.2f%%\t%s\n", i, (100.0 * file_time[i])/total_time,
          time_str(file_time[i]));
      for ( ; k < file_line[i + 1]; k++) {
        uint64_t t = line_time[k];
        if (line_freq[k] > 0 ∧ t ≥ limit) {
          if (i ≡ system_file) printf("\t%7s", line_name[k − file_line[i]]);
          else printf("\t%6d", k − file_line[i]);
          printf("\t%6.2f%%\t%s", 100.0 * t/total_time, time_str(t));
```

```
            printf("\t%6d\t%s", line_freq[k], time_str(t/line_freq[k]));
            printf("\n");
          }
        }
        printf("\n");
      }
    }
  }
```

Used in 94.

3.8 Finding the “Top Ten” Lines

Now that we have for each line in one of the input files the total time used by it, we can sort the lines by their time use. It is neither efficient nor useful to sort all the lines, but it sufficient to find the “top ten”.

⟨ declarations 2 ⟩ +≡ (75)

```
static uint64_t *tt_time;
static int *tt_file, *tt_line;
static int tt = 10 + 1, tt_count;
;
bool opt_top_ten;
```

⟨ process options 21 ⟩ +≡ (76)

```
case 'T': opt_top_ten = true;
  break;
case 't':
  {
    char *endptr;
    tt = strtol(option + 1, &endptr, 10) + 1;
    if (endptr ≡ option + 1)
      commandline_error("-t<n> without a numeric argument <n>");
    else if (tt < 2 ∨ tt > 101)
      commandline_error("-t<n> with <n> out of bounds");
    option = endptr − 1;
  }
  break;
```

⟨ explain table options 31 ⟩ +≡ (77)

```
"-T     show the table of the top 10 input lines\n"
```

⟨ explain format options 22 ⟩ +≡ (78)

```
"-t<n> replace 10 by n (2<=n<=100, default 10) for the top 1\
    0 input lines\n"
```

⟨ find the top ten lines 79 ⟩ ≡ (79)

```
ALLOCATE(tt_time, tt);
ALLOCATE(tt_file, tt);
ALLOCATE(tt_line, tt);
tt_time[0] = #FFFFFFFFFFFFFFFF;
tt_file[0] = 0;
tt_line[0] = 0;
tt_time[1] = 0;
tt_file[1] = 0;
tt_line[1] = 0;
tt_count = 2;
{
  int f, l;
  for (f = l = 0; f < file_num; f++) {
    for ( ; l < file_line[f + 1]; l++) {
      uint32_t t = line_time[l];
      if (t > tt_time[tt_count − 1]) ⟨ insert t into the top ten list 80 ⟩
    }
  }
}
```

Used in 81.

Inserting a new time into the top ten array using insertion sort.

⟨ insert t into the top ten list 80 ⟩ ≡ (80)

```
{
  if (tt_count < tt) tt_count++;
  i = tt_count − 1;
  while (tt_time[i − 1] < t) {
    i−−;
    tt_time[i + 1] = tt_time[i];
    tt_file[i + 1] = tt_file[i];
    tt_line[i + 1] = tt_line[i];
  }
  tt_time[i] = t;
  tt_file[i] = f;
  tt_line[i] = l − file_line[f];
}
```

Used in 79.

⟨ show the top ten lines 81 ⟩ ≡ (81)

```
if (opt_top_ten) {
  collect_line_time( );
  ⟨ find the top ten lines 79 ⟩
  Mprintf("\nThe␣top␣ten␣lines:\n");
  Mprintf("␣␣file\t␣␣line\tper\
      cent\t␣␣␣␣␣absolute""\t␣count\t␣␣␣␣␣average\tfile\n");
```

```
  for (i = 1; i < tt_count; i++) {
    int freq = line_freq[tt_line[i] + file_line[tt_file[i]]];
    uint32_t t = tt_time[i];
    if (t > 0 ∧ freq > 0) {
      print_fl(tt_file[i], tt_line[i]);
      printf("\t%6.2f%%\t%s", 100.0 * t/total_time, time_str(t));
      printf("\t%6d\t%s\t%s\n", freq, time_str(t/freq), file_names[tt_file[i]]);
    }
  }
}
```

Used in 94.

3.9 The Call Graph

For each macro, we know already how often it was called, the cumulative time, and the time directly spent in the macro. For the call graph, we want to know for each macro which child macros it did call, how often it did call each child, how much this child contributed directly and cumulative to the macro. From this data we could derive which parent macros called it, how often it was called by each parent, and how much time it contributed directly and cumulatively to the parent. We need to form linked lists of edges, so each edge has a link to its siblings.

Following the conventions of `gprof`, call counts are displayed in the format n/m where n is the number of calls on the edge of the call graph under consideration and m is the total number of calls.

The table of macro definitions is already available, so what we need is just a collection of the edges with counts, direct times and cumulative times. It is possible to estimate direct and cumulative times from the counts and the timing information we have already for each macro, but this method is not very precise.

Since macro calls can form recursive loops, the parent is not only an ancestor of the child, the child may also be an ancestor of the parent and its siblings. (Sounds horrible, but we are talking about macro calls.) Therefore the cumulative time spent in executing a child macro might include time spent in the parent macro or in its sibling macros. Therefore we need to split the cumulative time spent in a macro into two components: The cumulative time T that can be attributed to this macro alone, and the time that is spent in this macro but is attributed to the parent or a sibling. The call the later time the "loop time" L. To compute T and L, we keep track of the activation count a of each child, and keep two auxiliary variables ts and Ts when we push the first activation record, and update T and L only when we pop the last activation of the macro. The variable ts is used to record the *total_time* and Ts is used to record the sum of the time directly attributed to the parent and the cumulative times attributed to all siblings. Recomputing these values when we pop the final activation, we can obtain the time dt passed since its first activation, and the time dT attributed to parent and siblings since then. We use dT to update L and $dt - dT$ to update T.

Here is the declaration of an *edge*:

```
⟨ declarations 2 ⟩ +≡                                                    (82)
  struct edge {
    uint16_t child, sibling;
    int count, a;
    uint64_t T, L, ts, Ts;
  } *edges;
  int edges_count, edges_num;
```

The total number of outgoing and incoming edges can only be estimated. So a dynamic allocation is used.

```
⟨ allocate arrays 10 ⟩ +≡                                                (83)
  edges_num = 1024;
  edges_count = 1;                    /* zero element is a sentinel and a head */
  ALLOCATE(edges, edges_num);
```

```
⟨ functions 5 ⟩ +≡                                                       (84)
  uint16_t new_edge(void)
  {
    if (edges_count < edges_num) return edges_count++;
    REALLOC(edges, edges_count, edges_num);
    return edges_count++;
  }
```

When a parent macro with definition p calls a child macro with definition c, the *start_child* function is called after the new child macro is put on top of the macro stack. We first search the existing edges of the parent for an existing edge, possibly from a previous invocation, before we add a new edge. Once the edge is found, we increment *count* and a. If a changes from zero to one we update T, L, ts, and Ts as described above.

```
⟨ functions 5 ⟩ +≡                                                       (85)
  void start_child(int p, int c)
  {
    int e = macro_defs[p].e;
    if (e ≡ 0) {
      e = new_edge( );
      macro_defs[p].e = e;
      edges[e].child = c;
      goto found;
    }
    do {
      if (edges[e].child ≡ c) goto found;
      else if (edges[e].sibling ≡ 0) {
        int s = new_edge( );
        edges[e].sibling = s;
        e = s;
        edges[e].child = c;
```

```
      goto found;
    }
    else e = edges[e].sibling;
  } while (true);
found: edges[e].a++;
  edges[e].count++;
  if (edges[e].a ≡ 1) {
    uint16_t i;
    edges[e].ts = total_time;
    edges[e].Ts = macro_defs[p].t;
    for (i = macro_defs[p].e; i ≠ 0; i = edges[i].sibling)
      if (i ≠ e) edges[e].Ts += edges[i].T;
  }
}
```

When we pop a child macro, we use the following function to update a and in the case of $a \equiv 1$ also T and L.

⟨ functions 5 ⟩ +≡ (86)

```
void end_child(int p, int c)
{
  uint16_t e = macro_defs[p].e;
  while (e ≠ 0) {
    if (edges[e].child ≡ c) goto found;
    else e = edges[e].sibling;
  }
found:
  if (edges[e].a ≡ 1) {
    uint16_t i;
    uint64_t dt;
    uint64_t dT;
    dT = macro_defs[p].t;
    for (i = macro_defs[p].e; i ≠ 0; i = edges[i].sibling)
      if (i ≠ e) dT += edges[i].T;
    dt = total_time − edges[e].ts;
    dT = dT − edges[e].Ts;
    edges[e].L += dT;
    edges[e].T += dt − dT;
  }
  edges[e].a−−;
}
```

The information contained in the edges gives a good view of the call graph. It can be shown using the "`-G`" option.

⟨ declarations 2 ⟩ +≡ (88)

```
bool opt_graph = false;
```

⟨ process options 21 ⟩ +≡ (89)
```
case 'G': opt_graph = true;
  break;
```

⟨ explain table options 31 ⟩ +≡ (90)
```
  "-G␣␣␣␣show␣the␣table␣of␣the␣␣macro␣call␣graph\n"
```

⟨ show the macro call graph if requested 91 ⟩ ≡ (91)
```
  if (opt_graph) {
    sort_macros();
    Mprintf("\nThe␣macro␣call␣graph:\n""␣␣␣␣␣␣␣␣time\t␣␣␣␣␣\
        ␣␣␣loop\tpercent\t␣count/total\tchild\n");
    i = macro_defs[0].link;
    do {
      int e;
      uint64_t Ti = macro_defs[i].T;
      if (100.0 * Ti/total_time < percent_limit) break;
      print_macro(i);
      printf("\n");
      printf("%s\t␣␣␣␣␣␣␣␣␣\t%6.2f%%\t", time_str(Ti), 100.0 * Ti/total_time);
      if (opt_machine) printf("\t%6d\t", macro_defs[i].count);
      else printf("␣␣␣␣␣␣*␣␣␣␣␣␣\t");
      print_macro(i);
      printf("\n");
      printf("%s\t␣␣␣␣␣␣␣␣␣\t", time_str(macro_defs[i].t));
      if (Ti ≠ 0) printf("%6.2f%%\t", 100.0 * macro_defs[i].t/Ti);
      else printf("\t");
      if (opt_machine) printf("%6d\t\t", macro_defs[i].count);
      else printf("%7d␣␣␣␣␣␣\t", macro_defs[i].count);
      print_macro(i);
      printf("\n");
      e = macro_defs[i].e;
      while (e ≠ 0) {
        int c = edges[e].child;
        int n = edges[e].count;
        int m = macro_defs[c].count;
        uint64_t Te = edges[e].T;
        int64_t L = edges[e].L;
        double p = 100.0 * Te/Ti;
        if (p ≥ percent_limit) {
          printf("%s\t", time_str(Te));
          if (L ≡ 0 ∧ ¬opt_machine) printf("␣␣␣␣␣␣␣␣\t");
          else printf("%s\t", time_str(L));
          if (Ti ≠ 0) printf("%6.2f%%\t", p);
          else printf("\t");
```

```
            if (opt_machine) printf("%6d\t%6d\t", n, m);
            else printf("%6d/%-6d\t", n, m);
            print_macro(c);
            printf("\n");
          }
          e = edges[e].sibling;
        }
        printf("\n");
        i = macro_defs[i].link;
      } while (i ≠ 0);
    }
```

Used in 94.

3.10 All Tables

Now its time to bring all the different output tables together in a useful sequence.

⟨ process options 21 ⟩ +≡ (92)

```
case 'A': opt_files = opt_summary = opt_cmd = opt_top_ten = opt_graph = true;
  break;
```

⟨ explain table options 31 ⟩ +≡ (93)

```
  "-A␣␣␣␣␣show␣all␣tables␣(equal␣to␣-TGFC)␣tables\n"
```

⟨ write the output file 94 ⟩ ≡ (94)

⟨ show all time stamps if requested 33 ⟩
⟨ show the complete macro stack if requested 38 ⟩
⟨ show the command summary if requested 55 ⟩
⟨ show the file summary if requested 51 ⟩
⟨ show lines above limit if requested 74 ⟩
⟨ show the top ten lines 81 ⟩
⟨ show the table of macros profiled if requested 62 ⟩
⟨ show the macro call graph if requested 91 ⟩
⟨ show the total time if requested 42 ⟩

Used in 1.

4 Processing the Command Line

```
⟨ process the command line 95 ⟩ ≡                                    (95)
  i = 1;
  while (i < argc) {
    char *option;
    if (argv[i][0] ≡ '-') {
      option = argv[i] + 1;
      while (*option ≠ 0) {
        switch (*option) {
          ⟨ process options 21 ⟩
        default: commandline_error("unknown option");
        }
        option++;
      }
    }
    else if (input_file_name ≡ NULL) input_file_name = argv[i];
    else commandline_error("multiple input files given");
    i++;
  }
```

Used in 1.

Sometimes an error means that the program should also explain how to use it.

```
⟨ functions 5 ⟩ +≡                                                   (96)
  void explain_usage(void) { fprintf (stderr,
          "Use: " "texprofile [-options] <input file>\n" "options:\n"
          ⟨ explain general options 98 ⟩
      "\n" ⟨ explain table options 31 ⟩
      "\n" ⟨ explain format options 22 ⟩
      "\n" ) ;
      exit(0); }
```

A simple option that most users will use occasionaly is the `-?` option. This option can also be look like `--help`, `-help` or `-h`.

```
⟨ process options 21 ⟩ +≡                                            (97)
case 'h': case '?': explain_usage( );
  break;
```

⟨ explain general options 98 ⟩ ≡ (98)
```
"-?␣␣␣␣␣␣␣␣display␣this␣help␣and␣exit\n"
"--help␣␣␣␣display␣this␣help␣and␣exit\n"
```
Used in 96.

The help option should also be available as a long option:

⟨ process options 21 ⟩ +≡ (99)
```
case '-': option++;
  ⟨ process long options 100 ⟩
  else commandline_error("unknown␣long␣option");
  break;
```

⟨ process long options 100 ⟩ ≡ (100)
```
if (strcmp(option, "help") ≡ 0) explain_usage( );
```
Used in 99.

The only other long option currently supported is the `--version` option.

```
#define VERSION_STR "1.1"
```

⟨ process long options 100 ⟩ +≡ (101)
```
if (strcmp(option, "version") ≡ 0) {
  printf("texprofile␣version␣"VERSION_STR"\n");
  exit(0);
}
```

⟨ explain general options 98 ⟩ +≡ (102)
```
"--version␣output␣version␣information␣and␣exit\n"
```

5 Error Handling

If an error occurs this program will print an error message and terminate. There is no attempt to recover from errors.

⟨ error handling 103 ⟩ ≡ (103)

```
int error (char *msg)
{
  fprintf(stderr, "texprofile: %s\n", msg);
  exit(1);
  return 0;
}
```

Used in 1.

The program is a little bit more verbose if there is an error in the command line.

⟨ error handling 103 ⟩ +≡ (104)

```
int commandline_error(char *msg)
{
  fprintf(stderr, "texprofile: %s\n", msg);
  fprintf(stderr, "Try 'texprofile --help' for more information.\n");
  exit(1);
  return 0;
}
```

The input file should start with a marker: the ASCII codes of "TEX PROF". The same marker is used later to separate the different sections of the file.

⟨ functions 5 ⟩ +≡ (105)

```
void check_file_marker(char *msg)
{
  char marker[8];
  if (fread(marker, 1, 8, in) ≠ 8)
    error ("Unexpected end of input") ;
  if (strncmp(marker, FILE_MARKER, 8) ≠ 0)
    error (msg) ;
}
```

Appendix

A The Table of Command Names

To print commands nicely, here is an array of command names. Some of the commands also have names to be able to test for them; the definitions are taken from `texprof.w`.

```
#define max_command 100 /* the largest command code seen at big_switch */
#define system_cmd (max_command + 1) /* pseudo command value */
#define system_profile_on (system_cmd + 1)
#define system_profile_off (system_cmd + 2)
#define system_macro_push (system_cmd + 3)
#define system_macro_pop (system_cmd + 4)
#define system_macro_pop_small (system_cmd + 5)
#define system_macro_pop_0 (system_cmd + 6)
```

⟨ declarations 2 ⟩ +≡ (106)

```
static char *cmd_name[] = {
"relax", /* do nothing ( \relax ) */
"left_brace", /* beginning of a group ( { ) */
"right_brace", /* ending of a group ( } ) */
"math_shift", /* mathematics shift character ( $ ) */
"tab_mark", /* alignment delimiter ( &, \span ) */
"car_ret/" /* end of line ( carriage_return, \cr, \crcr ) */
"out_param", /* output a macro parameter */
"mac_param", /* macro parameter symbol ( # ) */
"sup_mark", /* superscript ( ^ ) */
"sub_mark", /* subscript ( _ ) */
"ignore/" /* characters to ignore ( ^^@ ) */
"endv", /* end of ⟨ v_j ⟩ list in alignment template */
"spacer", /* characters equivalent to blank space ( ␣ ) */
```

```
"letter",                /* characters regarded as letters ( A..Z, a..z ) */
"other_char",                  /* none of the special character types */
"active/"                      /* characters that invoke macros ( ~ ) */
"par/"                                 /* end of paragraph ( \par ) */
"match",                               /* match a macro parameter */
"comment/"               /* characters that introduce comments ( % ) */
"end_match/"                           /* end of parameters to macro */
"stop",                                /* end of job ( \end, \dump ) */
"invalid_char/"          /* characters that shouldn't appear ( ^^? ) */
"delim_num",             /* specify delimiter numerically ( \delimiter ) */
"char_num",              /* character specified numerically ( \char ) */
"math_char_num",                 /* explicit math code ( \mathchar ) */
"mark",                                /* mark definition ( \mark ) */
"xray",                  /* peek inside of TEX ( \show, \showbox, etc. ) */
"make_box",              /* make a box ( \box, \copy, \hbox, etc. ) */
"hmove",                 /* horizontal motion ( \moveleft, \moveright ) */
"vmove",                       /* vertical motion ( \raise, \lower ) */
"un_hbox",                     /* unglue a box ( \unhbox, \unhcopy ) */
"un_vbox",                   /* unglue a box ( \unvbox, \unvcopy ... ) */
"remove_item",                 /* nullify last item ( \unpenalty, ... ) */
"hskip",                  /* horizontal glue ( \hskip, \hfil, etc. ) */
"vskip",                    /* vertical glue ( \vskip, \vfil, etc. ) */
"mskip",                                /* math glue ( \mskip ) */
"kern",                                 /* fixed space ( \kern ) */
"mkern",                                /* math kern ( \mkern ) */
"leaders/shipout",          /* use a box ( \shipout, \leaders, etc. ) */
"halign",                  /* horizontal table alignment ( \halign ) */
"valign",                    /* vertical table alignment ( \valign ) */
"no_align",           /* temporary escape from alignment ( \noalign ) */
"vrule",                                /* vertical rule ( \vrule ) */
"hrule",                              /* horizontal rule ( \hrule ) */
"insert",                        /* vlist inserted in box ( \insert ) */
"vadjust",          /* vlist inserted in enclosing paragraph ( \vadjust ) */
"ignore_spaces",             /* gobble spacer tokens ( \ignorespaces ) */
"after_assignment", /* save till assignment is done ( \afterassignment ) */
"after_group",               /* save till group is done ( \aftergroup ) */
"break_penalty",                 /* additional badness ( \penalty ) */
"start_par",                /* begin paragraph ( \indent, \noindent ) */
"ital_corr",                            /* italic correction ( \/ ) */
"accent",                      /* attach accent in text ( \accent ) */
"math_accent",                 /* attach accent in math ( \mathaccent ) */
"discretionary",          /* discretionary texts ( \-, \discretionary ) */
"eq_no",                        /* equation number ( \eqno, \leqno ) */
"left_right",              /* variable delimiter ( \left, \right ) ... ) */
"math_comp",                  /* component of formula ( \mathbin, etc. ) */
"limit_switch",       /* diddle limit conventions ( \displaylimits, etc. ) */
```

```
"above",                    /* generalized fraction ( \above, \atop, etc. ) */
"math_style",               /* style specification ( \displaystyle, etc. ) */
"math_choice",              /* choice specification ( \mathchoice ) */
"non_script",               /* conditional math glue ( \nonscript ) */
"vcenter",                  /* vertically center a vbox ( \vcenter ) */
"case_shift",               /* force specific case ( \lowercase, \uppercase ) */
"message",                  /* send to user ( \message, \errmessage ) */
"extension",                /* extensions to TeX ( \write, \special, etc. ) */
"in_stream",                /* files for reading ( \openin, \closein ) */
"begin_group",              /* begin local grouping ( \begingroup ) */
"end_group",                /* end local grouping ( \endgroup ) */
"omit",                     /* omit alignment template ( \omit ) */
"ex_space",                 /* explicit space ( \␣ ) */
"no_boundary",              /* suppress boundary ligatures ( \noboundary ) */
"radical",                  /* square root and similar signs ( \radical ) */
"end_cs_name",              /* end control sequence ( \endcsname ) */
"min_internal/"             /* the smallest code that can follow \the */
"char_given",               /* character code defined by \chardef */
"math_given",               /* math code defined by \mathchardef */
"last_item",                /* most recent item ( \lastpenalty,... ) */
"toks_register",            /* token list register ( \toks ) */
"assign_toks",              /* special token list ( \output, \everypar, etc. ) */
"assign_int",               /* user-defined integer ( \tolerance, \day, etc. ) */
"assign_dimen",             /* user-defined length ( \hsize, etc. ) */
"assign_glue",              /* user-defined glue ( \baselineskip, etc. ) */
"assign_mu_glue",           /* user-defined muglue ( \thinmuskip, etc. ) */
"assign_font_dimen",        /* user-defined font dimension ( \fontdimen ) */
"assign_font_int",          /* user-defined font integer ( \hyphenchar, ... ) */
"set_aux",                  /* specify state info ( \spacefactor, \prevdepth ) */
"set_prev_graf",            /* specify state info ( \prevgraf ) */
"set_page_dimen",           /* specify state info ( \pagegoal, etc. ) */
"set_page_int",             /* specify state info ( \deadcycles, ... ) */
"set_box_dimen",            /* change dimension of box ( \wd, \ht, \dp ) */
"set_shape",                /* specify fancy paragraph shape ( \parshape ) ... ) */
"def_code",                 /* define a character code ( \catcode, etc. ) */
"def_family",               /* declare math fonts ( \textfont, etc. ) */
"set_font",                 /* set current font ( font identifiers ) */
"def_font",                 /* define a font file ( \font ) */
"internal_register",        /* internal register ( \count, \dimen, etc. ) */
"advance",                  /* advance a register or parameter ( \advance ) */
"multiply",                 /* multiply a register or parameter ( \multiply ) */
"divide",                   /* divide a register or parameter ( \divide ) */
"prefix",                   /* qualify a definition ( \global, \long, \outer etc. ) ) */
"let",                      /* assign a command code ( \let, \futurelet ) */
"shorthand_def",            /* code definition ( \chardef, \countdef, etc. ) */
"read_to_cs",               /* read into a control sequence ( \read, etc. ) */
```

```
  "def",                    /* macro definition ( \def, \gdef, \xdef, \edef ) */
  "set_box",                                  /* set a box ( \setbox ) */
  "hyph_data",            /* hyphenation data ( \hyphenation, \patterns ) */
  "set_interaction",       /* define level of interaction ( \batchmode, etc. ) */
  "system",                                              /* system start */
  "profile␣on",                    /* switching on profile ( \profileon ) */
  "profile␣off",                   /* switching on profile ( \profileoff ) */
  "call",                                                  /* macro call */
  "pop",                                                   /* macro return */
  "pop␣n",                                                 /* macro return */
  "pop␣0",                                                 /* macro return */
  "pop␣1",                                                 /* macro return */
  "pop␣2",                                                 /* macro return */
  "pop␣3",                                                 /* macro return */
  "pop␣4",                                                 /* macro return */
  "pop␣5",                                                 /* macro return */
  "pop␣6",                                                 /* macro return */
  "pop␣7",                                                 /* macro return */
  "pop␣8",                                                 /* macro return */
  "pop␣9",                                                 /* macro return */
  "pop␣10",                                                /* macro return */
  "unknown"                                          /* should not happen */
  };
  static const int cmd_num = sizeof (cmd_name)/sizeof (*cmd_name);
#define CMD_NAME(N) ((N) < cmd_num ? cmd_name[N] : "unknown")
  static char *line_name[] = {                    /* for the system pseudo file */
  "unknown",                                                    /* as it says */
  "start",                        /* time before executing the first command */
  "end",                           /* time after executing the last command */
  "shipout",                                /* time spent writing the dvi file */
  "linebrk",                                  /* time spent on line breaking */
  "initrie",                          /* time spent on setting up hyphenation */
  "buildpg",                                  /* time spent in build_page */
  "inputln",                                    /* time spent in input_ln */
  "insert",                          /* time spent on tokens inserted by TeX */
  };
  static char *every_name[] = {          /*  names for output, everypar, etc.  */
  "output",                                          /* for output routines */
  "everypar",                                            /* for \everypar */
  "everymath",                                           /* for \everymath */
  "everydisplay",                                        /* for \everydisplay */
  "everyhbox",                                           /* for \everyhbox */
  "everyvbox",                                           /* for \everyvbox */
  "everyjob",                                            /* for \everyjob */
  "everycr",                                             /* for \everycr */
  "mark",                                          /* for \topmark, etc. */
```

```
"everyeof"                                                    /* for \everyeof */
};
static const int sys_line_num = sizeof (line_name)/sizeof (*line_name);
```