1. httpd.

2. Main Program.

```
⟨ include files 4 ⟩
⟨ Preprocessor definitions ⟩
⟨ declarations of functions 11 ⟩
⟨ type declarations 6 ⟩
⟨ local functions 30 ⟩
int main(int argc, char *const *argv)
{
  struct MHD_Daemon *d;
  assert(MHD_is_feature_supported(MHD_FEATURE_MESSAGES));
  if (argc ≠ 2) {
    printf("%s PORT\n", argv[0]);
    return 1;
  }
  unsigned int flags = MHD_USE_THREAD_PER_CONNECTION;
  flags |= MHD_USE_INTERNAL_POLLING_THREAD;
  flags |= MHD_USE_ERROR_LOG;
  d = MHD_start_daemon(flags,
      atoi(argv[1]),
      ⟨ accept policy callback option 32 ⟩
      ⟨ http request callback option 33 ⟩
      ⟨ http options 34 ⟩
      ⟨ logging options 37 ⟩
      MHD_OPTION_END);
  if (d ≡ Λ) return 1;
  (void) getc(stdin);
  MHD_stop_daemon(d);
  return 0;
}
```

3. library

```
⟨ dummy.c 3 ⟩ ≡
  ⟨ include files 4 ⟩
#include "cnt.h"
  ⟨ Preprocessor definitions ⟩
  ⟨ type declarations 6 ⟩
  ⟨ declarations of functions 11 ⟩
  ⟨ library data 10 ⟩
  ⟨ library helper functions 41 ⟩
  ⟨ library functions 12 ⟩
```

4.

⟨ include files 4 ⟩ ≡
```
#include <microhttpd.h>
#include <assert.h>
#include <stdbool.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
```
This code is used in sections 2 and 3.

5.

⟨ initialize request local data 5 ⟩ ≡
```
  if (&aptr ≠ *ptr) {      /* do never respond on first call */
    *ptr = &aptr;
    return MHD_YES;
  }
```

6. ⟨ type declarations 6 ⟩ ≡
```
  typedef struct _Request *Request;
```
See also section 9.

This code is used in sections 2 and 3.

7. Processing.

8. The data model for processing a requests considers resources identified by the url maybe in a pattern, and a method like GET or POST. Each resource has an individual definition of how it reacts to the individual method. If the method for this resource is not declared, there should be an error.

9. ⟨ type declarations 6 ⟩ +≡

```
struct _request {
  int number;
};
struct _proc {
  void(*func)(struct _request *);
};
struct _handler {
  char resource[100];
  char method[10];
  char desc[100];
  struct _proc *proc;
};
```

10.

```
#define PROC_STATIC (procs + 0)
#define PROC_FILE (procs + 1)
```

⟨ library data 10 ⟩ ≡

```
struct _proc procs[] = { { . func = Λ } , { . func = func_file_handler } } ; struct _handler handlers[]
    = {
    { . resource = "/index.html" , . method = "GET" , . desc = "File" , . proc = PROC_FILE } ,
    { . resource = "/jquery.js" , . method = "GET" , . desc = "File" } ,
    { . resource = "/knockout.js" , . method = "GET" , . desc = "File" } ,
    { . resource = "/o.js" , . method = "GET" , . desc = "File" } ,
    { . resource = "/sampleProductCategories.js" , . method = "GET" , . desc = "File" } ,
    { . resource = "/viewmodel.js" , . method = "GET" , . desc = "File" } } ;
```

See also section 20.

This code is used in section 3.

11. The main handler for requests.

⟨ declarations of functions 11 ⟩ ≡

```
enum MHD_Result cb_request(void *cls, struct MHD_Connection *connection, const char *url, const
    char *method, const char *version, const char *upload_data, size_t *upload_data_size, void
    **ptr);
```

See also sections 17, 24, 26, 36, and 40.

This code is used in sections 2 and 3.

12. ⟨ library functions 12 ⟩ ≡

```
enum MHD_Result cb_request(void *cls, struct MHD_Connection *connection, const char *url, const
        char *method, const char *version, const char *upload_data, size_t *upload_data_size, void
        **ptr)
{
  static char *page = "{\"data\":1}";
  static int aptr;
  struct MHD_Response *response = Λ;
  int ret;
  unsigned int status_code = MHD_HTTP_NOT_FOUND;
  ⟨ log request info 39 ⟩
  ⟨ dispatch request 13 ⟩
  if (response ≡ Λ) {
    ⟨ try open file 22 ⟩
    if (file) {
      ⟨ respond page from file content 23 ⟩
    }
  }
  if (response ≡ Λ) {
    ⟨ respond static page 21 ⟩
  }
  fprintf(stderr, "response␣%p\n", response);
  ret = MHD_queue_response(connection, status_code, response);
  fprintf(stderr, "queued␣response␣%d␣->␣%d\n", status_code, ret);
  MHD_destroy_response(response);
  return ret;
}
```

See also sections 15, 18, 25, 27, 29, and 38.

This code is used in section 3.

13. $*ptr$ as the data structure representing the internal request data.

```
⟨ dispatch request 13 ⟩ ≡
  if (*ptr) {
    struct _request *r = *ptr;
    int n = sizeof (handlers)/sizeof (*handlers);
    for (int i = 0; i < n; i++) {
      if (0 ≡ strcmp(url, handlers[i].resource)
            ∧ 0 ≡ strcmp(method, handlers[i].method)) {
        struct _handler *h = handlers + i;
        fprintf(stderr, "response␣%d.␣%s\n", i, h→desc);
        if (h→proc) {
          struct _proc *p = h→proc;
          (p→func)(r);
        }
        fprintf(stderr, "R:␣%d\n", r→number);
        response = MHD_create_response_from_buffer(strlen(*ptr), *ptr, MHD_RESPMEM_MUST_COPY);
        ret = MHD_queue_response(connection, status_code, response);
        MHD_destroy_response(response);
        return ret;
      }
    }
  }
  else {
    *ptr = malloc(sizeof(struct _request));
    return MHD_YES;
  }
```

This code is used in section 12.

14.

⟨ handle post message 14 ⟩ ≡

```
{
  fprintf(stderr, "Upload data size: %d\n", *upload_data_size);
  if (*upload_data_size ≡ 0) return MHD_YES;
  else {
    fprintf(stderr, "CONTENT: ");
    for (int i = 0; i < *upload_data_size; i++) {
      fprintf(stderr, "%02x ", upload_data[i]);
    }
    for (int i = 0; i < *upload_data_size; i++) {
      fprintf(stderr, "%c", upload_data[i]);
    }
    fprintf(stderr, " %p\n", response);
    const char *xpage = "XXX";
    if (false) response = MHD_create_response_from_buffer(*upload_data_size, (void *)
        upload_data, MHD_RESPMEM_MUST_COPY);
    else response = MHD_create_response_from_buffer(strlen(xpage), (void *) xpage,
        MHD_RESPMEM_MUST_COPY);
    MHD_add_response_header(response, MHD_HTTP_HEADER_CONTENT_ENCODING, "application/json");
    if (false) *upload_data_size = 0;
    status_code = MHD_HTTP_OK;
    ret = MHD_queue_response(connection, status_code, response);
    fprintf(stderr, "x queued response %d -> %d\n", status_code, ret);
    MHD_destroy_response(response);
    return MHD_YES;
  }
}
```

15.

⟨ library functions 12 ⟩ +≡

```
enum MHD_Result print_key_value(void *cls, enum MHD_ValueKind kind, const char *key, const
    char *value)
{
  fprintf(stderr, "*** %d:%s:%s\n", kind, key, value);
  return MHD_YES;
}
```

16. File handler.

17. file handler

⟨ declarations of functions 11 ⟩ +≡
  void *func_file_handler*( );

18.

⟨ library functions 12 ⟩ +≡
  void *func_file_handler*( )
  { }

19. Static page response.

20. ⟨ library data 10 ⟩ +≡
```
const char page_404[] = "file not found";
```

21. ⟨ respond static page 21 ⟩ ≡
```
response = MHD_create_response_from_buffer(sizeof (page_404) - 1, (void *) page_404,
    MHD_RESPMEM_PERSISTENT);
status_code = MHD_HTTP_NOT_FOUND;
```
This code is used in section 12.

22. ⟨ try open file 22 ⟩ ≡
```
FILE *file = fopen(&url[1], "rb");
struct stat buf;
if (NULL != file) {
  int fd = fileno(file);
  if (-1 == fd) {
    fclose(file);
    file = NULL;
  }
  else if ((0 != fstat(fd, &buf)) || (!S_ISREG(buf.st_mode))) {
      /* not a regular file, refuse to serve */
    fclose(file);
    file = NULL;
  }
}
```
This code is used in section 12.

23. respond with data in file by using callbacks for data and for cleanup.

⟨ respond page from file content 23 ⟩ ≡
```
status_code = MHD_HTTP_OK;
response = MHD_create_response_from_callback(buf.st_size, 32 * 1024,      /* 32k size */
&file_reader, file, &file_free_callback);
```
This code is used in section 12.

24. file callback

⟨ declarations of functions 11 ⟩ +≡
```
static ssize_t file_reader(void *cls, uint64_t pos, char *buf, size_t max);
```

25.

⟨ library functions 12 ⟩ +≡
```
static ssize_t file_reader(void *cls, uint64_t pos, char *buf, size_t max)
{
  FILE *file = cls;
  (void) fseek(file, pos, SEEK_SET);
  return fread(buf, 1, max, file);
}
```

26. file cleanup callback

```
⟨ declarations of functions 11 ⟩ +≡
  static void file_free_callback(void *cls);
```

27.

```
⟨ library functions 12 ⟩ +≡
  static void file_free_callback(void *cls)
  {
    fclose((FILE *) cls);
  }
```

28. ⟨ check for allowed method 28 ⟩ ≡

```
  if ((0 ≠ strcmp(method, MHD_HTTP_METHOD_GET)) ∧ (0 ≠ strcmp(method, MHD_HTTP_METHOD_HEAD)))
    return MHD_NO;       /* unexpected method */
```

29. ⟨ library functions 12 ⟩ +≡

```
  enum MHD_Result post_iterator(void *cls, enum MHD_ValueKind kind, const char *key, const
          char *filename, const char *content_type, const char *transfer_encoding, const char
          *data, uint64_t off, size_t size)
  {
    struct Request *request = cls;
    fprintf(stderr, "### %s\n", key);
    return MHD_YES;
  }
```

30.

```
⟨ local functions 30 ⟩ ≡
  static void request_completed_callback(void *cls, struct MHD_Connection *connection, void
          **con_cls, enum MHD_RequestTerminationCode toe)
  {
    (void) cls;       /* Unused. Silent compiler warning. */
    (void) connection;       /* Unused. Silent compiler warning. */
    (void) toe;       /* Unused. Silent compiler warning. */
    fprintf(stderr, "end of request\n");
  }
```

See also section 42.

This code is used in section 2.

31. Security.

32. ⟨ accept policy callback option 32 ⟩ ≡
Λ, Λ ,
This code is used in section 2.

33. ⟨ http request callback option 33 ⟩ ≡
&*cb_request*, Λ ,
This code is used in section 2.

34. ⟨ http options 34 ⟩ ≡
`MHD_OPTION_CONNECTION_TIMEOUT`, 256 ,
This code is used in section 2.

35. Define HTTPS related options. The key and a certificate needs to be set.

⟨ https specific options 35 ⟩ ≡
`MHD_OPTION_HTTPS_MEM_KEY`, *key_pem*, `MHD_OPTION_HTTPS_MEM_CERT`, *cert_pem* ,

36. Logging. The logging is done by a simple callback function.

```
⟨ declarations of functions 11 ⟩ +≡
  void logger(void *cls, const char *fm, va_list ap);
```

37. The options need to be included in the main daemon call.

```
⟨ logging options 37 ⟩ ≡
  MHD_OPTION_EXTERNAL_LOGGER, logger, &argv ,
```

This code is used in section 2.

38. The implementation of the logger using the *printf* function.

```
⟨ library functions 12 ⟩ +≡
  void logger(void *cls, const char *fm, va_list ap)
  {
    fprintf(stderr, "!!!!!␣");
    vfprintf(stderr, fm, ap);
    fprintf(stderr, "\n");
  }
```

39.

```
⟨ log request info 39 ⟩ ≡
  fprintf(stderr, "ECHO␣url:%s\n␣method:%s\n", url, method);
  fprintf(stderr, "␣␣␣upload␣data␣size:␣%d\n", *upload_data_size);
  MHD_get_connection_values(connection, -1, print_key_value, Λ);
```

This code is used in section 12.

40. print key value

```
⟨ declarations of functions 11 ⟩ +≡
  enum MHD_Result print_key_value(void *cls, enum MHD_ValueKind kind, const char *key, const
      char *value);
```

41.

```
⟨ library helper functions 41 ⟩ ≡      /* empty */
```

This code is used in section 3.

42.

```
⟨ local functions 30 ⟩ +≡      /* empty */
```

43. INDEX.

HTTP