1. Curl Socket Example.
Sorry I write this in CWEB since its much easier to provide one file, while testing the dependencies of several libraries in one sourcecode file, which gets split during processing.

2. So lets start with the entry point and how this example is deployed.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <curl/curl.h>
#include "qt-part.h"
  ⟨ global operation data 3 ⟩
  ⟨ global functions 12 ⟩
  ⟨ curl callbacks 7 ⟩
  ⟨ curly functions 17 ⟩
  int main(int argc, char **argv)
  {
    int io_channel_counter = 0;
      /* when no io channels are open anymore we can quit the application */
    struct global_data global;
    struct test_read_data *trd = calloc(1, sizeof(struct test_read_data));
    int code;
    curl_global_init(CURL_GLOBAL_ALL);
    create_test_app( );
    trd→io_channel_counter = &io_channel_counter, io_channel_counter++;
    create_io_channel(0, 0, trd, test_read_cb);      /* register stdin */
    global.curl = curl_multi_init( );
    global.io_channel_counter = &io_channel_counter;
    io_channel_counter++;
    setup_curl_multi_urls(&global, "http://curl.haxx.se/download");
    while (CURLM_CALL_MULTI_PERFORM ≡ (code = curl_multi_socket_all(global.curl, &global.handles))) ;
    exec_test_app( );
    if (trd→some_fd_context) free(trd→some_fd_context);
    curl_multi_cleanup(global.curl);
    return 0;
  }
```

3. Global operation data

```
⟨ global operation data 3 ⟩ ≡
  struct global_data {
    CURLM * curl;
    int handles;
    int *io_channel_counter;
    struct url_opdat *lastop;
  };
```

This code is used in section 2.

4. Curl Functions.

5. Setup a single URL.

```
void *setup_curl_url(struct global_data *global, const char *url)
{
  CURL * curl = curl_easy_init( );
  struct url_opdat *url_dat = calloc(1, sizeof(struct url_opdat));
  url_dat→global = global;
  url_dat→url = malloc(strlen(url) + 1);
  strcpy(url_dat→url, url);
  curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, incoming_list);
  curl_easy_setopt(curl, CURLOPT_HEADERFUNCTION, incoming_list);
  curl_easy_setopt(curl, CURLOPT_WRITEDATA, url_dat);
  curl_easy_setopt(curl, CURLOPT_HEADERDATA, url_dat);
  curl_easy_setopt(curl, CURLOPT_NOBODY, 1);      /* set a head request */
  curl_easy_setopt(curl, CURLOPT_URL, url);
  return curl;
}
```

6. Setup a line of urls.

```
void setup_curl_multi_urls(struct global_data *global, const char *base_url)
{
  char url_buf[1024];
  unsigned i;
  CURL * curl;
  for (i = 10; i < 20; ++i) {
    sprintf(url_buf, "%s/curl-7.%d.0.tar.bz2", base_url, i);
    curl = setup_curl_url(global, url_buf);
    if (curl ≠ Λ) {
      curl_multi_add_handle(global→curl, curl);
    }
  }
  curl_multi_setopt(global→curl, CURLMOPT_SOCKETFUNCTION, socket_cb);
  curl_multi_setopt(global→curl, CURLMOPT_SOCKETDATA, global);
}
```

7. Curl Callbacks.

⟨ curl callbacks 7 ⟩ ≡

```
static void socket_cb(CURL * c, curl_socket_t fd, int action, void *inf, void *priv);
static void io_socket_cb(void *iof, void *cb_vp, int fd);
static size_t incoming_list(void *dat, size_t len, size_t n, void *req_vp);
static void handle_done_request(struct global_data *global, CURLMsg * msg);
```

This code is used in section 2.

8. Curl Socket Callback.

```
#define DEBUG_HANDLES 0
  void socket_cb(CURL * c, curl_socket_t fd, int action, void *inf, void *priv)
  {
    struct socket_opdat *dat = priv;
    struct global_data *global = inf;
#if DEBUG_HANDLES
    fprintf(stdout, "CURL SOCKET CB %d, %p", fd, dat);
#endif
    if (dat ≡ Λ) {
      dat = calloc(1, sizeof(struct socket_opdat));
      dat→global = global;
      curl_multi_assign(global→curl, fd, dat);
    }
    switch (action) {
    case CURL_POLL_NONE:
#if DEBUG_HANDLES
      fprintf(stdout, " NONE\n");
#endif
      break;
    case CURL_POLL_IN:
#if DEBUG_HANDLES
      fprintf(stdout, " IN %p\n", dat);
#endif
      if (dat→out_iof) destroy_io_channel(&dat→out_iof);
      if (¬dat→in_iof) dat→in_iof = create_io_channel(fd, 0, dat, io_socket_cb);
      break;
    case CURL_POLL_OUT:
#if DEBUG_HANDLES
      fprintf(stdout, " OUT %p\n", dat);
#endif
      if (dat→in_iof) destroy_io_channel(&dat→in_iof);
      if (¬dat→out_iof) dat→out_iof = create_io_channel(fd, 1, dat, io_socket_cb);
      break;
    case CURL_POLL_INOUT:
#if DEBUG_HANDLES
      fprintf(stdout, " INOUT %p\n", dat);
#endif
      if (¬dat→out_iof) dat→out_iof = create_io_channel(fd, 1, dat, io_socket_cb);
      if (¬dat→in_iof) dat→in_iof = create_io_channel(fd, 0, dat, io_socket_cb);
      break;
    case CURL_POLL_REMOVE:
#if DEBUG_HANDLES
      fprintf(stdout, " REMOVE %p\n", dat);
#endif
      if (dat→out_iof) destroy_io_channel(&dat→out_iof);
      if (dat→in_iof) destroy_io_channel(&dat→in_iof);
      if (dat ≠ Λ) free(dat);      /* FIXME DEBUG FIELDS */
      break;
    default:
#if DEBUG_HANDLES
```

```
      fprintf(stdout, "␣unkown\n");
#endif
      break;
    }
  }
```

9. Curl IO Callback.

```
#define DEBUG_IO_REQUESTS 0
  void io_socket_cb(void *iof, void *dat_vp, int fd)
  {
    struct socket_opdat *dat = dat_vp;
    struct global_data *global = dat→global;
    int code;
    int handle_ofs;
#if DEBUG_IO_REQUESTS
    fprintf(stdout, "PROCESSING␣IO␣FOR␣%d␣(%03d)\n", fd, global→handles);
#endif
    handle_ofs = global→handles;
    while (CURLM_CALL_MULTI_PERFORM ≡ (code = curl_multi_socket_action(global→curl, fd, 0,
        &global→handles))) ;
    handle_ofs −= global→handles;
    if (handle_ofs ≠ 0) {
      CURLMsg * msg;
      int quelen;
      while (Λ ≠ (msg = curl_multi_info_read(global→curl, &quelen))) {
        switch (msg→msg) {
        case CURLMSG_DONE: handle_done_request(global, msg);
          break;
        default: break;
        }
      }
    }
#if DEBUG_IO_REQUESTS
    fprintf(stdout, "END␣OF␣IO␣FOR␣%d␣(%03d)\n", fd, global→handles);
#endif
    if (global→handles ≡ 0) {
      close_io_channel(global→io_channel_counter);
    }
  }
```

10.

```
  void handle_done_request(struct global_data *global, CURLMsg * msg)
  {
    long code;
    curl_easy_getinfo(msg→easy_handle, CURLINFO_RESPONSE_CODE, &code);
    fprintf(stdout, "REQUEST␣IS␣DONE␣[%s]:%3ld\n", global→lastop→url, code);
  }
```

11. Curl incoming callback.

```
#define DEBUG_INCOMING_DATA 0
  size_t incoming_list(void *dat, size_t len, size_t n, void *req_vp)
  {
    struct url_opdat *url_dat = req_vp;
    char *buf;
    buf = malloc(len * n + 1);
    memcpy(buf, dat, len * n);
    buf[len * n] = '\0';
    if (len * n < 5) {
      free(buf);
      return len * n;
    }
    if (memcmp(buf, "HTTP/", 5) ≡ 0) {
      unsigned p = 6;
      long http_code;
      int ok = 0;
      char *e;
      while (buf[p] ≠ '␣' ∧ p < len * n) ++p;
      if (buf[p] ≡ '␣') {
        e = buf + p;
        http_code = strtol(buf + p, &e, 10);
        if (e > buf + p ∧ e - buf < len * n) {
          if (e[0] ≡ '␣') ok = 1;
        }
      }
      if (ok) {
#if DEBUG_INCOMING_DATA
        fprintf(stdout, "INCOMING[%s]:%ld\n", url_dat→url, http_code);
#endif
      }
    }
    free(buf);
    url_dat→global→lastop = url_dat;
    return len * n;
  }
```

12. Closing any channel.

```
⟨ global functions 12 ⟩ ≡
  void close_io_channel(int *io_channel_counter);
```

This code is used in section 2.

13. Global channel close counter.

```
  void close_io_channel(int *io_channel_counter)
  {
    if (*io_channel_counter > 1) (*io_channel_counter)--;
    else call_test_app_quit( );
  }
```

14. Each url gets some operation data.

⟨ curl multi operation data 14 ⟩ ≡
```
struct url_opdat {
  struct global_data *global;
  char *url;
};
struct socket_opdat {
  struct global_data *global;
  void *out_iof, *in_iof;
};
```
This code is used in section 17.

15. Two functions create the complete curl setup.

⟨ curl multi setup 15 ⟩ ≡
```
extern void setup_curl_multi_urls(struct global_data *global, const char *base_url);
extern void *setup_curl_url(struct global_data *global, const char *url);
```
This code is used in section 17.

16. Simple Functions. A simple read function.

```
void test_read_cb(void *iof, void *dat, int fd)
{
  char buf[1024];
  struct test_read_data *trd = dat;
  int nread = 0;
  nread = read(fd, buf, 1024);
  if (nread ≤ 0) {
    fprintf(stdout, "close on %d\n", fd);
    destroy_io_channel(&iof);
    close_io_channel(trd→io_channel_counter);
  }
  else fprintf(stdout, "read %d bytes from %d\n", nread, fd);
  trd→status = 0;
}
```

17. The curl specific functions, data types and callbacks

⟨ curly functions 17 ⟩ ≡
  ⟨ curl multi operation data 14 ⟩
  ⟨ curl multi setup 15 ⟩
  ⟨ test read record 18 ⟩
  ⟨ test read cb 19 ⟩

This code is used in section 2.

18. The Test read simply reads out a socket or file descriptor.

⟨ test read record 18 ⟩ ≡

```
struct test_read_data {
  char *some_fd_context;
  int status;
  int *io_channel_counter;
};
```

This code is used in section 17.

19.

⟨ test read cb 19 ⟩ ≡

```
extern void test_read_cb(void *iof, void *dat, int fd);
```

This code is used in section 17.

20. Export an Interface for C.

```
⟨ qt-part.h 20 ⟩ ≡
#ifndef _QT_PART_H_
#define _QT_PART_H_ 1
#if defined __cplusplus
  extern "C"
  {
#endif
  ⟨ Preprocessor definitions ⟩
    ⟨ qt application prototypes 25 ⟩
    ⟨ qt io prototypes 23 ⟩
#if defined __cplusplus
  }
#endif
#endif
```

21. Qt Part of the test. The Qt Part of this test consists of a simple QApp creation and the IO mechanism.

```
⟨ qt-part.cc 21 ⟩ ≡
  ⟨ qt application code 26 ⟩
  ⟨ qt io events 24 ⟩
#include "moc_qt-part.cc"
```

22. Asynchronous Operation:IO.

23. Prototypes for io to export some functions to "C".

⟨ qt io prototypes 23 ⟩ ≡
```
typedef void(*call_io_function)(void *iof, void *dat, int fd);
extern void *create_io_channel(int fd, int type, void *dat, call_io_function f);
extern void destroy_io_channel(void **X);
extern void enable_io_channel(void *ios_ptr, int enable);
```
This code is used in section 20.

24. IO Event Class specification.

⟨ qt io events 24 ⟩ ≡
```
#include <QtCore/QSocketNotifier>
  class ProgramIOSelect : public QSocketNotifier { Q_OBJECT
public: ProgramIOSelect(QObject *parent, void *dat, call_io_function ciof, int fd, unsigned type)
  : QSocketNotifier(fd, (QSocketNotifier::Type) type, parent), m_data(dat), m_io_function(ciof) {
    connect(this, SIGNAL(activated(int)), this, SLOT(call_io(int)));
    setEnabled(true);
  }
  public slots : void call_io(int fd)
  {
    m_io_function(this, m_data, fd);
  }
private: void *m_data;
  call_io_function m_io_function; } ;
  void *create_io_channel(int fd, int type, void *dat, call_io_function f)
  {
    ProgramIOSelect *new_io = new ProgramIOSelect(Λ, dat, f, fd, type);
    return new_io;
  }
  void destroy_io_channel(void **X)
  {
    ProgramIOSelect *del_io = (ProgramIOSelect *)(*X);
    del_io→setEnabled(false);
    delete del_io;
    *X = Λ;
  }
  void enable_io_channel(void *ios_ptr, int enable)
  {
    ProgramIOSelect *io = (ProgramIOSelect *)(ios_ptr);
    io→setEnabled(enable);
  }
```
This code is used in section 21.

25. Qt Application Part.

⟨ qt application prototypes 25 ⟩ ≡

```
extern void *create_test_app();
extern void call_test_app_quit();
extern void exec_test_app();
```

This code is used in section 20.

26.

```
#define TApp ((TestApp *) get_test_app())
```

⟨ qt application code 26 ⟩ ≡

```
#include <QtCore/QCoreApplication>
#include "qt-part.h"
static char *_m_argv[2] = {"it's me", Λ};
static int _m_argc = 1; class TestApp : public QCoreApplication { Q_OBJECT
public: TestApp()
: QCoreApplication(_m_argc, _m_argv) { }
~TestApp()
{ } } ;
void *create_test_app()
{
return new TestApp();
}
void *get_test_app()
{
return QCoreApplication::instance();
}
void call_test_app_quit()
{
TApp→quit();
}
void exec_test_app()
{
TApp→exec();
}
```

This code is used in section 21.

27. Index.

TEST-IT