CCID-Compliant User-Space Smart Card Driver

To Contact the author, please email `libccid@matthew.ath.cx`

1. The daemon.

This is the daemon that runs with superuser privileges. It handles all the communication over the USB bus, and multiplexes the connections from applications trying to access smart cards.

```
⟨ System includes 6 ⟩
⟨ Local includes 7 ⟩
⟨ Function declarations 14 ⟩
⟨ Global variables 8 ⟩
int main(int argc, char **args)
{
  assert(true, "Assert␣is␣broken");
  debug("Starting");
  read_arguments(argc, args);      /* read arguments */
  if (check_running(param.detach)) {      /* check for running copy, optionally demonize */
    if (¬send_args()) {      /* already running, send message */
      printf("Error:␣PID␣file␣exists␣but␣");
      ⟨ Check error on connecting to daemon 3 ⟩;
    }
    return 0;
  }
  ⟨ Log startup details to syslog 2 ⟩;
  ⟨ Create bind to sockets 4 ⟩;
  initialise_usb();      /* initialise usb connections */
  listen_sockets();      /* start listening on the sockets */
  ⟨ Perform clean up operations 5 ⟩;
  return 0;
}
```

2. Log startup to syslog. We setup if we are logging to syslog or stderr, then write the initial state to the log.

```
⟨ Log startup details to syslog 2 ⟩ ≡
  setup_syslog(param.syslog);      /* setup if we log to syslog or to stderr */
  slog("Daemon␣starting");
  slog("");
  slog("␣--␣Options␣--");
  slog("program:␣%s", Λ ≡ param.program ? "" : param.program);
  slog("config␣file:␣%s", Λ ≡ param.config ? "" : param.config);
  slog("daemonize:␣%s", param.detach ? "true" : "false");
  slog("syslog:␣%s", param.syslog ? "true" : "false");
  slog("");
```

This code is used in section 1.

3. Check error from connecting. If we are trying to send a message to another daemon and get an error, try and diagnose it.

⟨ Check error on connecting to daemon 3 ⟩ ≡
```
switch (errno) {
case EACCES: case EPERM:
  printf("cannot␣connect␣to␣daemon:␣Permission␣Denied␣-␣are␣you␣root?\n");
  break;
case ECONNREFUSED: printf("cannot␣connect␣to␣daemon,␣not␣running\n");
  break;
default: printf("failed␣to␣connect␣to␣daemon\n");
}
```
This code is used in section 1.

4. Create **b**ind to sockets. Make directory and file descriptor for sockets

⟨ Create **b**ind to sockets 4 ⟩ ≡
```
debug("Creating␣sockets");
setup_sockets(&master_socket, (struct sockaddr *) &master_info, "master",
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH | S_IWOTH | S_IWGRP);
setup_sockets(&control_socket, (struct sockaddr *) &control_info, "control", S_IRUSR | S_IWUSR);
```
This code is used in section 1.

5. Cleaning up our sockets and pidfile. We need to close and unlink the sockets and remove the pid file.

⟨ Perform clean up operations 5 ⟩ ≡
```
debug("Cleaning␣up");
clean_up(master_socket, (struct sockaddr *) &master_info);
clean_up(control_socket, (struct sockaddr *) &control_info);
assert(unlink(pidfile) ≡ 0, "Failed␣to␣clean␣up␣-␣couldn't␣unlink␣pidfile");
```
This code is used in section 1.

6. System library includes. We include here the system libraries for socket and filesystem IO and string and error handling. We also need some types exported from lib-USB.

⟨ System includes 6 ⟩ ≡
```
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <sys/un.h>
#include <unistd.h>
#include <usb.h>
```
This code is used in section 1.

7. Local header includes. These import the USB IO functions, the protocol layer and debugging functions.

⟨ Local includes 7 ⟩ ≡
```
#include "usb.h"
#include "structs.h"
#include "../share/types.h"
#include "../share/debug.h"
#include "../share/log.h"
#include "../share/protocol.h"
```
This code is used in section 1.

8. Global Variables.

⟨ Global variables 8 ⟩ ≡
```
#define MAXCONNQUEUE 5
  int master_socket, control_socket;      /* the sockets to listen(2) on */
  struct sockaddr_struct {
    sa_family_t family;
    char sa_data[MAXSOCKETPATH];
  } master_info, control_info;
    /* the info on the sockets (struct sockaddr extended to fit the path length in) */
  pclient client_root = Λ;      /* a pointer to the linked list of currently connected clients */
  char pidfile[20];      /* the name of the pidfile to write */
```
See also section 10.

This code is used in section 1.

9. Program Options.
The daemon takes several command line options. The functions to parse them are given here as well.

short	**long**	**definition**
-c	–config	Specify Config File
-d	–no-detach	Don't leave console
-s	–no-syslog	Log to stderr not syslog
-f	–force	Force starting of the daemon
-h	–help	Show this help
-m	-message	Control message to send to existing process

10. Options structure. The program options are all stored in this structure.

```
⟨ Global variables 8 ⟩ +≡
  struct param_struct {
    char *message;
    char *config;
    char *program;
    bool syslog;
    bool detach;
    bool force;
  } param;
```

11. Procedure *print_syntax*( ). Prints out the program syntax

```
  void print_syntax( )
  {
    printf("Syntax: cardd <options>\n");
    printf("Options:\n");
    printf("\t-c <file>\n\t--config <file>\t\tSpecify Config File\n");
    printf("\t-d\n\t--no-detach\t\tDon't leave console\n");
    printf("\t-s\n\t--no-syslog\t\tLog to stderr not syslog\n");
    printf("\t-f\n\t--force\t\tForce starting of the daemon\n");
    printf("\t-h\n\t--help\t\t\tShow this help\n");
    printf("\t-m <message>\n\t--message <message>\tControl message to send to existing proc\
        ess\n");
    printf("Messages:\n");
    printf("\tquit\texit the running daemon\n");
    printf("\tconfig\tre-read the config file\n");
  }
```

12. Procedure *read_arguments*( ). Parses all the arguments passed to the program into param structure
argc The number of arguments
args An array of strings of the arguments

```
void read_arguments(int argc, char **args)
{
  ⟨ Initialise the param structure 13 ⟩;
  param.program = (*args);      /* the name the program was called as */
  args++;
  argc--;
  while (argc > 0) {      /* While there are more arguments */
    debug("Parameter: %s", *args);
    ⟨ Put the paramter into the structure 15 ⟩;
    args++;
    argc--;      /* go onto next one */
  }
}
```

13.

```
⟨ Initialise the param structure 13 ⟩ ≡
  param.message = Λ;
  param.config = Λ;
  param.syslog = true;
  param.detach = true;
```

This code is used in section 12.

14.

```
⟨ Function declarations 14 ⟩ ≡
  void read_arguments(int argc, char **args);
```

See also sections 17, 19, 21, 26, 29, 31, 33, and 36.

This code is used in section 1.

15. Check the parameter. This code takes the parameter and puts it into the *param* structure. If the parameter takes an argument, it check that this exists and sets its value in the structure. If an error occurs, we print the program syntax and exit

```
⟨ Put the paramter into the structure 15 ⟩ ≡
  if (0 ≡ strcmp(*args, "-c") ∨ 0 ≡ strcmp(*args, "--config")) {
    argc−−;
    if (0 ≥ argc) {
      print_syntax( );
      exit(0);
    }      /* not enough arguments */
    args++;
    param.config = (*args);
    debug("Got␣parameter␣config:␣%s", param.config);
  }
  else if (0 ≡ strcmp(*args, "-d") ∨ 0 ≡ strcmp(*args, "--no-detach")) {
    param.detach = false;
    debug("Got␣parameter␣no-detach");
  }
  else if (0 ≡ strcmp(*args, "-s") ∨ 0 ≡ strcmp(*args, "--no-syslog")) {
    param.syslog = false;
    debug("Got␣parameter␣no-syslog");
  }
  else if (0 ≡ strcmp(*args, "-f") ∨ 0 ≡ strcmp(*args, "--force")) {
    param.force = true;
    debug("Got␣parameter␣force");
  }
  else if (0 ≡ strcmp(*args, "-h") ∨ 0 ≡ strcmp(*args, "--help")) {
    print_syntax( );
    exit(0);
  }
  else if (0 ≡ strcmp(*args, "-m") ∨ 0 ≡ strcmp(*args, "--message")) {
    argc−−;
    if (0 ≥ argc) {
      print_syntax( );
      exit(0);
    }      /* not enough arguments */
    args++;
    param.message = (*args);
    debug("Got␣parameter␣message:␣%s", param.message);
  }
  else {      /* unknown argument */
    print_syntax( );
    exit(0);
  }
```

This code is used in section 12.

16. Function *build_fd_list*( ). Takes all the clients in a linked list , and adds them to an *fd_set*. The function returns the highest fd + 1.

fds The *fd_set* to add the clients to
root The root of the linked list to traverse
scount One more than the highest current fd in the *fd_set*

```
int build_fd_list(fd_set * fds, pclient root, int scount)
{
  pclient current = root;
  while (Λ ≠ current) {      /* for all the clients in the list */
    FD_SET(current→socket, fds);      /* add this one to the set */
    debug("current->socket: %d", current→socket);
    if (scount ≤ current→socket) scount = current→socket + 1;
        /* increment the count if neccessary */
    debug("scount: %d", scount);
    current = current→next;      /* go onto next */
  }
  return scount;      /* return the count */
}
```

17.

⟨ Function declarations 14 ⟩ +≡

```
int build_fd_list(fd_set * fds, pclient root, int scount);
```

18. Procedure *setup_sockets*( ). Binds to a socket with *bind*(2) and *listen*(2) on it. Here we also check the directory the sockets are in, and set the permissions on it

skt The socket file descriptor
sktinfo The socket info structure
file The filename (`SOCKETDIR"/"` will be prepended) to bind to.
perms The permissions of the socket to set

```
void setup_sockets(int *skt, struct sockaddr *sktinfo, char *file, mode_t perms)
{
  socklen_t len;
  int ret;
  mkdir(SOCKETDIR, S_ISVTX | S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH);
  chmod(SOCKETDIR, S_ISVTX | S_IRUSR | S_IWUSR | S_IXUSR | S_IRGRP | S_IXGRP | S_IROTH | S_IXOTH);
    /* make the socket directory and set its permissions */
  *skt = socket(AF_UNIX, SOCK_STREAM, 0);    /* create a UNIX socket */
  assert(0 ≤ *skt, "Can't␣get␣a␣socket␣fd");
  debug("Creating␣socket␣%d", *skt);
  sktinfo→sa_family = AF_UNIX;
  strcpy(sktinfo→sa_data, SOCKETDIR);
  strcat(sktinfo→sa_data, "/");
  strcat(sktinfo→sa_data, file);    /* setup the info structure */
  ret = unlink(sktinfo→sa_data);
  assert(ret ≥ 0 ∨ errno ≡ ENOENT, "Can't␣unlink␣socket");
    /* make sure we can remove it before binding to it */
  len = sizeof (sktinfo→sa_family) + strlen(sktinfo→sa_data);
  debug("Binding␣to:␣%s", sktinfo→sa_data);    /* bind to the socket listen for connections */
  assert(bind(*skt, sktinfo, len) ≥ 0, "Can't␣bind␣to␣address");
  assert(listen(*skt, MAXCONNQUEUE) ≥ 0, "Can't␣listen␣on␣socket");
  chmod(sktinfo→sa_data, perms);    /* set permissions on the socket */
}
```

19.

```
⟨ Function declarations 14 ⟩ +≡
  void setup_sockets(int *skt, struct sockaddr *sktinfo, char *file, mode_t perms);
```

20. Procedure *listen_sockets*( ). Loop, running *select*(2) over the sockets and handling the results from them.

```
void listen_sockets( )
{
  ⟨ listen_sockets Local variables 22 ⟩;
  for ( ; ; ) {      /* Loop forever here unless told to exit by a control message */
    ⟨ Add all the active sockets to an fd_set 23 ⟩;
    timeout.tv_sec = 1;
    timeout.tv_usec = 0;
    assert((rc = select(scount, &socks, Λ, Λ, &timeout)) ≥ 0, "Select Error");
      /* check for things to read */
    debug("Connections on %d sockets.", rc);
    debug("checking usb devices for interrupts");
    check_usb_interrupts( );
    if (0 ≡ rc) continue;      /* nothing to read */
    {      /* read from the remaining sockets. */
      if (FD_ISSET(master_socket, &socks))      /* the master socket */
      {
        ⟨ Accept a new connection 24 ⟩;
      }
      if (FD_ISSET(control_socket, &socks))      /* the control socket */
      {
        if (handle_control_message( )) return;      /* handle control message and exit if told to */
      }
      current = client_root;
      while (Λ ≠ current)      /* the client sockets */
      {
        if (FD_ISSET(current→socket, &socks)) {
          if (¬service_client(current)) {      /* connection should be closed */
            slog("Client connection (pid: %d) closed", current→pid);
            close(current→socket);      /* close the connection */
            client_root = removeclient(client_root, current);      /* remove the client from the list */
            free(current);
            current = Λ;
          }
          else current = current→next;      /* go onto the next client */
        }
        else current = current→next;      /* go onto the next client */
      }
    }
  }
}
```

21.

```
⟨ Function declarations 14 ⟩ +≡
  void listen_sockets( );
```

22. Variables local to *listen_sockets*.

```
⟨ listen_sockets Local variables 22 ⟩ ≡
  socklen_t remote_len;      /* length of the sockaddr from the remote client */
  int scount;      /* max({fd | fd ∈ fd_set}) + 1 */
  struct timeval timeout;      /* timeout to wait on select calls */
  int rc = 0;      /* return value */
  struct sockaddr tempaddr;      /* temporary variable */
  int tempsock;      /* temporary variable */
  int bytes;      /* number of bytes written to socket */
  pclient newclient;      /* temporary variable */
  pid_t remote_pid = 0;      /* PID read from the remote client */
  pclient current;      /* temporary variable */
```

This code is used in section 20.

23. Create a list of all active sockets. We need to create an *fd_set* of all the active sockets. Add the constant ones, and then delegate to *build_fd_list*

```
⟨ Add all the active sockets to an fd_set 23 ⟩ ≡
  fd_set socks;
  FD_ZERO(&socks);      /* empty the FD_SET */
  FD_SET(master_socket, &socks);      /* add the master socket */
  debug("master_socket: %d", master_socket);
  FD_SET(control_socket, &socks);
  debug("control_socket: %d", control_socket);      /* add the control socket */
  scount = build_fd_list(&socks, client_root, max(control_socket, master_socket) + 1);
    /* walk the list of client sockets */
```

This code is used in section 20.

24. Accept a connection from a new client. We need to call *accept*(2) on the *master_socket*, then update our linked list. Also log their pid.

```
⟨ Accept a new connection 24 ⟩ ≡
  tempsock = accept(master_socket, &tempaddr, &remote_len);      /* accept the connection */
  if (0 ≠ tempsock) {
    assert(tempsock ≥ 0, "Connection Failed");
    newclient = (pclient) malloc(sizeof (client));
    newclient→socket = tempsock;
    newclient→sockinfo = tempaddr;
    scount++;      /* create new client */
    assert((bytes = recv(newclient→socket, &remote_pid, sizeof (pid_t), 0)) > 0, "Receive Failed");
      /* get pid */
    slog("New Client, PID: %d", remote_pid);
    newclient→pid = remote_pid;
    newclient→next = Λ;
    client_root = addclient(client_root, newclient);      /* add to linked list */
  }
```

This code is used in section 20.

25. Function *handle_control_message*( ).
Handles a message on the control socket and returns *true* if the daemon should exit.

```
bool handle_control_message( )
{
  ccid_request req;
  ccid_response res;
  int tempsock;
  struct sockaddr tempaddr;
  socklen_t remote_len;
  bool quit = false;
  debug("Handling␣Control␣Message");
  tempsock = accept(control_socket, &tempaddr, &remote_len);
    /* accept a new socket on that socket */
  assert(tempsock ≥ 0, "Connection␣Failed");
  assert(get_ccid_request(tempsock, &req) ≥ 0, "Failed␣to␣receive␣on␣client␣socket");
  assert(PARAM ≡ req.protocol, "Non-parameter␣message␣received␣on␣control␣socket");
    /* get the request and check its protocol type */
  ⟨ Check the message contents and prepare the response structure 27 ⟩;
  if (Λ ≠ req.data) {
    free(req.data);
    req.data = Λ;
  }     /* free it if neccessary */
  assert(send_ccid_response(tempsock, res) ≥ 0, "Failed␣to␣send␣response␣to␣client");
    /* return the response */
  close(tempsock);      /* close the socket */
  return quit;
}
```

26.

```
⟨ Function declarations 14 ⟩ +≡
  bool handle_control_message( );
```

27. Checking the message sent to the daemon. Currently we can be sent a quit message, or a message to re-read our config file. This section also prepares an appropriate response to send to the other copy of the daemon.

```
⟨ Check the message contents and prepare the response structure 27 ⟩ ≡
  res.protocol = PARAM;
  res.rdata = Λ;
  res.rsize = 0;      /* check the message sent, and prepare the response structure */
  if (Λ ≡ req.data)      /* no message */
  {
    slog("Empty control message, ignored", req.data);
    res.rdata = "Control message unknown";
    res.rsize = strlen(res.rdata) + 1;
  }
  else if (0 ≡ strcmp(req.data, "quit"))      /* quit message */
  {
    slog("Got quit message, exitting");
    quit = true;
    res.rdata = "Daemon Quitting";
    res.rsize = strlen(res.rdata) + 1;
  }
  else if (0 ≡ strcmp(req.data, "config"))      /* re-read config message */
  {
    slog("Got config message, re-reading config file");      /* TODO: re-read config file */
    res.rdata = "Daemon re-read config file";
    res.rsize = strlen(res.rdata) + 1;
  }
  else      /* unknown message */
  {
    slog("Unknown control message '%s', ignored", req.data);
    res.rdata = "Control message unknown";
    res.rsize = strlen(res.rdata) + 1;
  }
```

This code is used in section 25.

28. Function *service_client*( ). Service a request by a client connection. We receive the request structure from the client, then pass it down to the USB layer. The response is sent back to the client. Non-zero is returned if the connection has been closed.

```
int service_client(pclient cl)
{    /* TODO: detect wanting to close, and return ≠ 0 */
  ccid_request req;
  ccid_response res;
  debug("Servicing client %d", cl→pid);
  assert(get_ccid_request(cl→socket, &req) ≥ 0, "Failed to receive on client socket");
    /* read request from client */
  debug("protocol: %d, datasize: %d, apdu type: %d", req.protocol, req.dsize, req.apdu.apdu_type);
  assert(usb_ccid_command(cl→cardref, &req, &res), "Failed to send command to client card");
    /* send request to card and read response */
  if (Λ ≠ req.data) {
    free(req.data);
    req.data = Λ;
  }
  assert(send_ccid_response(cl→socket, res) ≥ 0, "Failed to send response to client");
    /* send response to client */
  return 0;
}
```

29.

⟨ Function declarations 14 ⟩ +≡

```
int service_client(pclient cl);
```

30. Procedure *clean_up*( ). Closes a socket and removes the file it was bound to

skt The file descriptor of the socket

sktinfo The info struct about the socket

```
void clean_up(int skt, struct sockaddr *sktinfo)
{
  close(skt);
  assert(unlink(sktinfo→sa_data) ≡ 0, "Failed␣to␣clean␣up␣-␣couldn't␣unlink␣socket");
}
```

31.

⟨ Function declarations 14 ⟩ +≡

```
void clean_up(int skt, struct sockaddr *sktinfo);
```

32. Procedure *check_running*( ).

Checks if there is an already running copy. If there isn't, then optionally fork from console and write pid to pidfile. Returns *true* if a pidfile exists

```
bool check_running(bool demonize)
{
  struct stat fs;
  pid_t pid;
  int ret;
  strcat(pidfile, SOCKETDIR);
  strcat(pidfile, "/pid");
  if (param.force) {      /* start anyway if the pidfile exists */
    ret = unlink(pidfile);
    assert(ret ≥ 0 ∨ errno ≡ ENOENT, "Failed to force removal of pidfile");
  }
  stat(pidfile, &fs);      /* stat the pidfile */
  if (S_ISREG(fs.st_mode)) {      /* existing copy running */
    return true;
  }
  pid = getpid( );
  if (demonize) {
    ⟨ Fork from console 34 ⟩;
  }
  FILE *fp = fopen(pidfile, "w");
  assert(Λ ≠ fp, "Couldn't write pidfile!");
  if (Λ ≡ fp) exit(1);
  fprintf(fp, "%d", pid);      /* write pid to pidfile */
  fclose(fp);
  return false;
}
```

33.

```
⟨ Function declarations 14 ⟩ +≡
  bool check_running(bool demonize);
```

34. Fork from console. We have to *fork*(2) a child proccess and the exit the parent to return to the console, then *setsid*(2) to disassociate ourselves from the tty we were spawned on. Finally, we *chdir*(2) to the root directory so that filesystems can be unmounted.

```
⟨ Fork from console 34 ⟩ ≡
  debug("Leaving␣console....");
  pid = fork( );
  assert(pid ≥ 0, "Error␣in␣fork");
  if (0 < pid) exit(0);
  setsid( );
  chdir("/");
  umask(0);
```

This code is used in section 32.

35. Send message to another daemon. Send our parameters to the existing copy, returning false if an error occurred.

```
  bool send_args( )
  {
    socklen_t len;
    struct sockaddr_struct info;
    int rc;
    int skt;
    ccid_request req;
    ccid_response res;
    skt = socket(AF_UNIX, SOCK_STREAM, 0);      /* get socket */
    assert(skt > 0, "Can't␣get␣socket␣fd");
    info.family = AF_UNIX;
    strcpy(info.sa_data, SOCKETDIR);
    strcat(info.sa_data, "/control");
    len = sizeof (info.family) + strlen(info.sa_data) + 1;
    debug("connecting␣to␣daemon");
    if ((rc = connect(skt, (struct sockaddr *) &info, len)) < 0) return false;
    req.protocol = PARAM;      /* setup the request struct */
    if (Λ ≡ param.message) {
      req.data = Λ;
      req.dsize = 0;
    }
    else {
      req.data = param.message;
      req.dsize = strlen(param.message) + 1;
    }
    if ((rc = send_ccid_request(skt, req, &res)) < 0) return false;      /* send it */
    if (Λ ≠ res.rdata)      /* print free the result */
    {
      printf("%s\n", res.rdata);
      free(res.rdata);
      res.rdata = Λ;
    }
    close(skt);      /* close the socket */
    return true;
  }
```

36.

⟨ Function declarations 14 ⟩ +≡
```
bool send_args( );
```

37. Data Structures. This file contains the data structures used by the daemon to keep lists of connected clients, devices and cards. The structures and method prototypes are exported via a header file.

```
#include "structs.h"
```

38. The header file.

```
⟨ structs.h 38 ⟩ ≡
#ifndef _CCID_STRUCTS_H
#define _CCID_STRUCTS_H
#include <usb.h>
#include <sys/socket.h>
  ⟨ Generic List defines 39 ⟩
  ⟨ Structure Prototypes 41 ⟩
  ⟨ Client structures 40 ⟩
  ⟨ Device structures 44 ⟩
  ⟨ Card structures 48 ⟩
  ⟨ Structure-Method Prototypes 43 ⟩
#endif
```

39. Generic list handling with # **define** statements

```
⟨ Generic List defines 39 ⟩ ≡
#define foreach(a, b) for (a = b; a ≠ Λ; a = a→next)
    /* traverses any linked list which uses next as the pointer */
#define listadd(r, t, n) do
  {
    t = r;
    if (Λ ≡ t) r = n;
    else {
      while (Λ ≠ t→next) t = t→next;
      t→next = n;
    }
  }
  while (0)      /* adds a node to any linked list which uses next as the pointer */
#define listrem(r, t, o)
  if (Λ ≠ r) {
    t = r;
    if (o ≡ t) r = o→next;
    else {
      while (Λ ≠ t→next ∧ o ≠ t→next) t = t→next;
      if (t→next ≡ o) t→next = o→next;
    }
  }    /* removes a node from any linked list which uses next as the pointer */
```

This code is used in section 38.

40. Client details. We store clients in a linked list, with functions to modify that structure.

⟨ Client structures 40 ⟩ ≡
```
struct client_struct {
  int pid;      /* The pid of the client */
  int socket;      /* The socket they are connected to */
  struct sockaddr sockinfo;      /* The socket info for this client */
  pclient next;      /* A pointer to the next client */
  usb_dev_handle * cardref;      /* The USB device this client is accessing */
};
```
This code is used in section 38.

41. Prototypes.

⟨ Structure Prototypes 41 ⟩ ≡
```
struct client_struct;
typedef struct client_struct client;
typedef struct client_struct *pclient;
```
See also sections 45 and 49.

This code is used in section 38.

42. Methods for handling client structures.

```
pclient addclient(pclient root, pclient cnew)      /* Adds a client to the end of the linked list */
{
  pclient temp;
  listadd(root, temp, cnew);
  return root;
}
pclient removeclient(pclient root, pclient cold)      /* Removes a client from the linked list */
{
  pclient temp;
  listrem(root, temp, cold);
  return root;
}
```

43. Prototypes.

⟨ Structure-Method Prototypes 43 ⟩ ≡
```
pclient addclient(pclient root, pclient cnew);
pclient removeclient(pclient root, pclient cold);
```
See also section 47.

This code is used in section 38.

44. Device details. We store all the available slots in devices in a linked list. This contains all the details needed to access the slot, and also the client that currently 'owns' that slot.

Note that the voltage and protocol fields are those supported by the **reader**, not neccessarily by the card. Those details will be read when powering up the card, and stored in the card struct reference.

```
⟨ Device structures 44 ⟩ ≡
  struct device_struct {
    struct usb_device *udev;      /* Raw usb device */
    struct usb_dev_handle *handle;      /* If open, this will be a device handle */
    char *filename;      /* The filename of the USB device */
    int interface_number;
    int bulk_in;      /* The bulk_in endpoint address */
    int bulk_out;      /* The bulk_out endpoint address */
    int intr;      /* The interrupt endpoint address */
    int packet_size;      /* the read packet size */
    int slot;      /* the slot number */
    int voltage;      /* possible voltages */      /* bitwise OR of:
#01 5.0V
#02 3.0V
#04 1.8V
  */
    int protocols;      /* bitwise OR of:
#01 T=0
#02 T=1
  */
    pcard card;      /* NULL if there is no card in this slot */
    pclient client;      /* NULL if not claimed by a client */
    pdevice next;      /* NULL at the end of the list */
  };
```

This code is used in section 38.

45. Prototypes.

```
⟨ Structure Prototypes 41 ⟩ +≡
  struct device_struct;
  typedef struct device_struct device;
  typedef struct device_struct *pdevice;
```

46. Methods for handling device structures.

```
  pdevice adddevice(pdevice root, pdevice cnew)      /* Adds a device to the end of the linked list */
  {
    pdevice temp;
    listadd(root, temp, cnew);
    return root;
  }
  pdevice removedevice(pdevice root, pdevice cold)      /* Removes a device from the linked list */
  {
    pdevice temp;
    listrem(root, temp, cold);
    return root;
  }
```

47. Prototypes.

```
⟨ Structure-Method Prototypes 43 ⟩ +≡
  pdevice adddevice(pdevice root, pdevice cnew);
  pdevice removedevice(pdevice root, pdevice cold);
```

48. Card details. This structure holds voltage, protocol and access details for a card inserted into a slot. These are read from the ATR when powering up the card.

```
⟨ Card structures 48 ⟩ ≡
  struct card_struct {
    int voltage;      /* the operating voltage */
    int protocol;     /* T protocol value (0 or 1) */
  };
```

This code is used in section 38.

49. Prototypes.

```
⟨ Structure Prototypes 41 ⟩ +≡
  struct card_struct;
  typedef struct card_struct card;
  typedef struct card_struct *pcard;
```

50. USB subsystem. This file contains all the functions which interface with the USB directly.

⟨ USB Header file includes 51 ⟩
⟨ USB Static Variables 64 ⟩
⟨ USB Internal Functions 68 ⟩
⟨ USB Library Functions 53 ⟩

51. Include header files. We are using lib-USB to connect to the smart-cards, and also the shared debug and protocol libraries.

⟨ USB Header file includes 51 ⟩ ≡
```
#include <usb.h>
#include <errno.h>
#include "usb.h"
#include "structs.h"
#include "../share/types.h"
#include "../share/debug.h"
#include "../share/log.h"
#include "../share/protocol.h"
```
This code is used in section 50.

52. Exported interfaces. These interfaces are exported in the header file `usb.h`.

⟨ `usb.h` 52 ⟩ ≡
```
#ifndef _CCID_USB_H
#define _CCID_USB_H
#include <usb.h>
#include "../share/types.h"
#include "../share/protocol.h"
#define SLOT_POWERED 0
#define SLOT_INSERTED 1
#define SLOT_EMPTY 2
#define USB_RW_TIMEOUT 6000
  ⟨ USB Function Prototypes 54 ⟩
#endif
```

53. Function *initialise_usb*( ). Initialises the USB bus, and scans the busses for devices. The function returns false if an error occurred.

```
⟨ USB Library Functions 53 ⟩ ≡
  bool initialise_usb( )
  {
    debug("Initialising␣USB␣library");
    usb_init( );      /* Initialise the library code */
    if (¬scan_usb( )) return false;      /* Scan for devices */
    return true;
  }
```

See also sections 55, 73, and 76.

This code is used in section 50.

54. Prototype.

```
⟨ USB Function Prototypes 54 ⟩ ≡
  bool initialise_usb( );
```

See also sections 56, 74, and 77.

This code is used in section 52.

55. Function *scan_usb*( ). This function scans the USB bus for new devices, and removes old ones from the list as the are connected / unplugged.

```
⟨ USB Library Functions 53 ⟩ +≡
  bool scan_usb( )
  {
    ⟨ scan_usb local variables 58 ⟩;
    ⟨ Create USB spec to match against 57 ⟩;
    ⟨ Scan for all devices with match 59 ⟩;
    ⟨ Delete Match 61 ⟩;
  }
```

56. Prototype.

```
⟨ USB Function Prototypes 54 ⟩ +≡
  bool scan_usb( );
```

57. Create a USB Match spec. To find devices you create a *usb_match_handle* structure, which contains a list of characteristics of USB devices to match against. Repeated called to *usb_find_device*( ) will then find all the devices matching that specification. The parameters to *usb_create_match*( ) are as follows:

VENDOR	−1	(Any Device)
PRODUCT	−1	(Any Device)
CLASS	`USB_CLASS_CCID`	(CCID devices)
SUBCLASS	−1	(Any Device)
PROTOCOL	−1	(Any Device)

```
⟨ Create USB spec to match against 57 ⟩ ≡
  usb_create_match(&match, −1, −1, USB_CLASS_CCID, −1, −1);
```

This code is used in section 55.

58.

```
⟨ scan_usb local variables 58 ⟩ ≡
  usb_match_handle * match = Λ;
```

See also sections 60 and 63.

This code is used in section 55.

59. Scan for all the devices with match. Now we scan the USB bus with repeated *usb_find_device*( ) calls to find all our devices.

```
⟨ Scan for all devices with match 59 ⟩ ≡
  dev = Λ;
  while (usb_find_device(match, dev, &dev) ≥ 0) {
    slog("Found␣a␣device:␣%s\n", dev→filename);      /* Log that we found a device */
    ⟨ Add device to structure 62 ⟩;
    ⟨ Print device info 0 ⟩;
  }
```

This code is used in section 55.

60.

```
⟨ scan_usb local variables 58 ⟩ +≡
  struct usb_device *dev;
  int i;
```

61. Delete the match variable.

⟨ Delete Match 61 ⟩ ≡
```
usb_free_match(match);
```
This code is used in section 55.

62. Put the device details in a struct.

```
⟨ Add device to structure 62 ⟩ ≡
  if (dev→descriptor→bNumConfigurations ≠ 1) {
    slog("Ignoring␣device:␣%s␣-␣non-singular␣configuration\n", dev→filename);
    continue;
  }
  if (dev→config[0].bNumInterfaces ≠ 1) {
    slog("Ignoring␣device:␣%s␣-␣non-singular␣interfaces\n", dev→filename);
    continue;
  }
  if (dev→config[0].interface[0].num_altsetting ≠ 1) {
    slog("Ignoring␣device:␣%s␣-␣non-singular␣alt-settings\n", dev→filename);
    continue;
  }
  if (dev→config[0].interface[0].altsetting[0].extralen ≠ #36) {
    slog("Ignoring␣device:␣%s␣-␣CCID␣extra␣descriptor␣length␣invalid\n", dev→filename);
    continue;
  }
  usb_open(dev, &temp_handle);      /* get a file descriptor for the device. */
  max_slots = dev→config[0].interface[0].altsetting[0].extra[4];
  for ( ; max_slots ≥ 0; max_slots−−) {      /* add a device entry for each slot */
    temp = malloc(sizeof(struct device_struct));
    ⟨ Setup new device structure 65 ⟩;
    temp→slot = max_slots;      /* points to the current slot */
    if (Λ ≡ dev_root) {      /* no devices yet, this one is the root */
      debug("Adding␣%s␣as␣root\n", dev→filename);
      slog("Adding␣device:␣%s", dev→filename);
      dev_root = temp;
    }
    else
      for (current = dev_root; ; current = current→next) {      /* search along list in case we have it */
        if (0 ≡ strcmp(current→filename, dev→filename)) {
            /* We have already seen this device, so we exit this block and go onto the next one */
          debug("Already␣detected␣%s\n", dev→filename);
          free(temp);
          break;
        }
        if (Λ ≡ current→next) {      /* the end of the list, put it here and exit */
          debug("Adding␣%s␣to␣list\n", dev→filename);
          slog("Adding␣device:␣%s", dev→filename);
          current→next = temp;
          free(temp);
          break;
        }
      }
    if (SLOT_INSERTED ≡ get_slot_status(temp))      /* check if there is already a card inserted */
      device_turn_on(temp);
  }
```

This code is used in section 59.

63.

```
⟨ scan_usb local variables 58 ⟩ +≡
  pdevice temp;
  pdevice current;
  int max_slots;
  struct usb_dev_handle *temp_handle;
```

64. Global device list. We store a static global list of the devices.

```
⟨ USB Static Variables 64 ⟩ ≡
  pdevice dev_root = Λ;
```

This code is used in section 50.

65. Setting up a device structure.

```
⟨ Setup new device structure 65 ⟩ ≡
  temp→filename = dev→filename;
  temp→udev = dev;
  temp→handle = temp_handle;
    /* we open this first because we only open it once per device, not per slot */
  temp→interface_number = dev→config[0].interface[0].altsetting[0].bInterfaceNumber;
  temp→bulk_in = temp→bulk_out = temp→intr = −1;
  ⟨ Read endpoints 66 ⟩;
  ⟨ Read extra data 67 ⟩;
  temp→next = Λ;
```

This code is used in section 62.

66. Check endpoint types. Here we have to check which endpoint is which. For bulk endpoints, *bmAttributes* ≡ #02 and for interrupt endpoints *bmAttributes* ≡ #03. A bulk-in endpoint has bit 7 of the address set to 1 (*bEndpointAddress* & #80 ≠ 0) and the bulk-out endpoint has it set to 0.

```
⟨ Read endpoints 66 ⟩ ≡
  for (i = 0; i < dev→config[0].interface[0].altsetting[0].bNumEndpoints; i++) {
      /* check each endpoint for type */
    if (#02 ≡ dev→config[0].interface[0].altsetting[0].endpoint[i].bmAttributes) {
        /* its a bulk endpoint */
      if (dev→config[0].interface[0].altsetting[0].endpoint[i].bEndpointAddress & #80) {
          /* its a bulk-in endpoint */
        temp→bulk_in = dev→config[0].interface[0].altsetting[0].endpoint[i].bEndpointAddress;
        temp→packet_size = dev→config[0].interface[0].altsetting[0].endpoint[i].wMaxPacketSize;
      }
      else      /* its a bulk-out endpoint */
        temp→bulk_out = dev→config[0].interface[0].altsetting[0].endpoint[i].bEndpointAddress;
    }
    else if (#03 ≡ dev→config[0].interface[0].altsetting[0].endpoint[i].bmAttributes)
        /* its an interrupt endpoint. */
      temp→intr = dev→config[0].interface[0].altsetting[0].endpoint[i].bEndpointAddress;
  }
```

This code is used in section 65.

67. Read CCID Extra Data. The information on which voltages and protocols the card supports is stored in an extra class descriptor block.

⟨ Read extra data 67 ⟩ ≡
```
temp→voltage = dev→config[0].interface[0].altsetting[0].extra[5];
temp→protocols = dev→config[0].interface[0].altsetting[0].extra[6];
```
This code is used in section 65.

68. Turning on a device. The function *device_turn_on*( ) is responsible for turning on a CCID card, reading the Answer To Reset and putting all the data into the appropriate structures.

```
⟨ USB Internal Functions 68 ⟩ ≡
  bool device_turn_on(pdevice dev){ ⟨ Power Up Local Variables 70 ⟩;
      slog("Powering␣up␣card␣%s.%d", dev→filename, dev→slot); dev → card = malloc(sizeof(card));
      ⟨ Power up the card 69 ⟩;
      ⟨ Read the ATR 71 ⟩;
      }
```

See also sections 75, 80, 81, and 83.

This code is used in section 50.

69. Powering up the card. To power up the card you need to send a CCID Reset command. This is a ten (10) byte string containing certain values:

Byte-indices	Field	Value	Description
0	*bMessageType*	#62	This is a type-62 message
1-4	*dwLength*	#0	No extra data
5	*bSlot*	#0-#ff	The slot to power up
6	*bSeq*	#0-#ff	Sequence number for command?
7	*bPowerSelect*	#0	Select voltage automatically
8-9	*abRFU*	Λ	Reserved for future use

```
⟨ Power up the card 69 ⟩ ≡
#define SEQ 0 /* The # defines are temporary */
  buf = malloc(10); /* allocate a 10-byte string */
  buf[0] = #62; /* bMessageType */
  buf[1] = buf[2] = buf[3] = buf[4] = #0; /* dwLength */
  buf[5] = dev→slot; /* bSLot */
  buf[6] = SEQ; /* bSeq */
  if (dev→voltage & #01) buf[7] = #01; /* bPowerSelect */
  else if (dev→voltage & #02) buf[7] = #02; /* bPowerSelect */
  else if (dev→voltage & #04) buf[7] = #03; /* bPowerSelect */
  debug("doing␣write.....␣%d\n", usb_bulk_write(dev→handle, dev→bulk_out, buf, 10, USB_RW_TIMEOUT));
  free(buf);
```

This code is used in section 68.

70.

```
⟨ Power Up Local Variables 70 ⟩ ≡
  char *buf;
```

See also section 72.

This code is used in section 68.

71. Read the ATR. The Answer to Reset contains various data required to be able to communicate with the card. Specifically it returns the protocol used by this card, T=0 or T=1. The ICC response is 10 bytes long, followed by *dwLength* bytes of the ATR.

The ICC response is:

Byte-indices	Field	Value	Description
0	*bMessageType*	`#80`	This is a message from the CCID device
1-4	*dwLength*		The length of the extra data (The ATR data)
5	*bSlot*		Corresponds to the power up message that caused this ATR.
6	*bSeq*		Corresponds to the power up message that caused this ATR.
7	*bStatus*		Slot status
8	*bError*		Any errors
9	*bChainParameter*	`#00` or `#01`	`#01` if this message has more parts

The ATR contains the protocol to use (T=0 or T=1) as the low order 4 bits of the second byte of the ATR data. This is byte 11 in our read buffer.

⟨ Read the ATR 71 ⟩ ≡

```
buf = malloc(dev→packet_size);
debug("doing␣read.....␣(%d)␣%s\n", usb_bulk_read(dev→handle, dev→bulk_in, buf, dev→packet_size,
    USB_RW_TIMEOUT), usb_strerror( ));
debug("system␣error:␣(%d)␣%s\n", errno, strerror(errno));
debug("message:␣%d\nstatus:␣%d\nerror:␣%d\n", buf[0], buf[7], buf[8]);
length = (unsigned int) buf[1];
debug("ATR␣length:␣%d\n", length); dev → card→protocol = buf[11] & #0F;      /* get out the protocol */
free(buf);
```

This code is used in section 68.

72.

⟨ Power Up Local Variables 70 ⟩ +≡

```
unsigned int length;
```

73. USB interrupt scanning. We need to regularly check all the card reader devices for messages on their interrupt lines.

```
⟨ USB Library Functions 53 ⟩ +≡
  void check_usb_interrupts( )
  { }
```

74. Function Prototype.

```
⟨ USB Function Prototypes 54 ⟩ +≡
  void check_usb_interrupts( );
```

75. Function *get_slot_status*( ). This checks whether a slot has a card in it or not. Returns the slot status, which is:

SLOT_POWERED	#0	Card is inserted and powered up
SLOT_INSERTED	#1	Card is inserted and not powered up
SLOT_EMPTY	#2	Card not present

The CCID slot status command is a ten (10) byte string containing certain values:

Byte-indices	Field	Value	Description
0	*bMessageType*	#65	This is a type-65 message
1-4	*dwLength*	#0	No extra data
5	*bSlot*	#0-#ff	The slot to query
6	*bSeq*	#0-#ff	Sequence number for command?
7-9	*abRFU*	Λ	Reserved for future use

The response is also a 10 byte structure with no extra data:

Byte-indices	Field	Value	Description
0	*bMessageType*	#81	This is a SlotStatus message from the CCID device
1-4	*dwLength*	#00	The length of the extra data, always 0
5	*bSlot*		Corresponds to the slot number in the request.
6	*bSeq*		Corresponds to the seq number in the request.
7	*bStatus*		Slot status
8	*bError*		Any errors
9	*bClockStatus*	#00 - #03	

⟨ USB Internal Functions 68 ⟩ +≡

```
int get_slot_status(pdevice dev){ # define SEQ0      /* The # defines are temporary */
    int status;
    char *buf;
  debug("Querying␣slot␣%d", dev→slot);
  {    /* Send the Command */
    buf = malloc(10);      /* allocate a 10-byte string */
    buf[0] = #65;      /* bMessageType */
    buf[1] = buf[2] = buf[3] = buf[4] = #0;      /* dwLength */
    buf[5] = dev→slot;      /* bSLot */
    buf[6] = SEQ;      /* bSeq */
    debug("doing␣write.....␣%d\n", usb_bulk_write(dev→handle, dev→bulk_out, buf, 10,
        USB_RW_TIMEOUT));
    free(buf);
  }
  {    /* Read the response */
    buf = malloc(dev→packet_size);      /* allocate a 10-byte string */
    debug("doing␣read.....␣(%d)␣%s\n", usb_bulk_read(dev→handle, dev→bulk_in, buf,
        dev→packet_size, USB_RW_TIMEOUT), usb_strerror( ));
    status = buf[7] & #3;      /* only the first 2 bits; */
    free(buf);
  }
  return status; }
```

76. Send a USB command to a device. This function sends a *ccid_request* to a given device, and reads the *ccid_response* from it. The function returns false if an error occurred.

```
⟨ USB Library Functions 53 ⟩ +≡
  bool usb_ccid_command(usb_dev_handle * handle, ccid_request * req, ccid_response * res)
  {
    debug("Doing␣cool␣USB␣stuff");
    res→protocol = req→protocol;
    res→rsize = 0;
    res→rdata = Λ;
    void *buffer = Λ;
    int length;
    ⟨ Decode request into the buffer 78 ⟩;
      /* usb_bulk_write(handle, 0x02, buffer, length, 60000); // send req */
    if (Λ ≠ buffer) free(buffer);
    buffer = Λ;
    buffer = malloc(length);
      /* length = usb_bulk_read(handle, 0x02, buffer, length, 6000); // get res */
    ⟨ Encode buffer as a response structure 79 ⟩;
    free(buffer);
    buffer = Λ;
    return true;
  }
```

77. Prototype.

```
⟨ USB Function Prototypes 54 ⟩ +≡
  bool usb_ccid_command(usb_dev_handle * handle, ccid_request * req, ccid_response * res);
```

78. Decode a request into a buffer.

```
⟨ Decode request into the buffer 78 ⟩ ≡
  switch (req→protocol) {
  case APDU:
    if (¬decode_apdu_req(req, &buffer)) ;      /* TODO: error handling */
    break;
  case TPDU: default: return false;
  }
```

This code is used in section 76.

79. Encode a buffer as a response.

```
⟨ Encode buffer as a response structure 79 ⟩ ≡
  switch (req→protocol) {
  case APDU:
    if (¬encode_apdu_res(buffer, res)) ;      /* TODO: error handling */
    break;
  case TPDU: default: return false;
  }
```

This code is used in section 76.

80. Function *encode_apdu_res*( ). Turns a **void** ∗ buffer into a *ccid_response* structure. The function returns false if an error occurred.

NOTE: may *malloc*(2) *res*¬*rdata*. If *res*¬*rsize* > 0 you **MUST** use *free*(2) to deallocate it when you are finished.

buffer The bit-stream to parse from the ISO protocol

res The structure to write the bit-stream to

```
⟨ USB Internal Functions 68 ⟩ +≡
  bool encode_apdu_res(void *buffer, ccid_response * res)
  {
    return true;
  }
```

81. Function *decode_apdu_req*( ). Turns a *ccid_request* structure into **void** ∗ buffer for writing to a USB device. The function returns false if an error occurred.

NOTE: may *malloc*(2) *buffer*. If the function returns *true* you **MUST** use *free*(2) to deallocate it when you are finished.

buffer A pointer to a **void** ∗ buffer

req The structure to decode to the buffer

```
⟨ USB Internal Functions 68 ⟩ +≡
  bool decode_apdu_req(ccid_request * req, void **buffer)
  {
    char header[4];      /* [0] = CLA, [1] = INS, [2] = P1, [3] = P2 */
    header[0] = #00;
    header[1] = #00;
    switch (req→apdu.apdu_type) {
    case APDU_AUTHENTICATE: ⟨ APDU authenticate decode 0 ⟩;
    case APDU_BINARY: ⟨ APDU binary decode 82 ⟩;
    case APDU_CHALLENGE: ⟨ APDU challenge decode 0 ⟩;
    case APDU_CHANNEL: ⟨ APDU channel decode 0 ⟩;
    case APDU_DATA: ⟨ APDU data decode 0 ⟩;
    case APDU_RECORD: ⟨ APDU record decode 0 ⟩;
    case APDU_SELECT: ⟨ APDU select decode 0 ⟩;
    case APDU_VERIFY: ⟨ APDU verify decode 0 ⟩;
    default: return false;
    }
    return true;
  }
```

82. Decoding an APDU Binary command.

```
⟨ APDU binary decode 82 ⟩ ≡
  switch (req¬apdu.binary.mode) {
  case READ:
    if (0 ≡ header[1]) header[1] = #B0;      /* set the INS to READ BINARY */
  case ERASE:
    if (0 ≡ header[1]) header[1] = #0E;      /* set the INS to ERASE BINARY */
    *buffer = malloc(3 + 4);      /* payload + header size */
    ((char *) *buffer)[6] = (req¬apdu.binary.length & #FF);
    ((char *) *buffer)[5] = ((req¬apdu.binary.length & #FF00) ≫ 4);
    ((char *) *buffer)[4] = 0;      /* Lc = null, Data = null, Le = req¬apdu.binary.length */
      /* buffer[0] = 0 =¿ extended format Le */
    break;
  case WRITE:
    if (0 ≡ header[1]) header[1] = #D0;      /* set the INS to WRITE BINARY */
  case UPDATE:
    if (0 ≡ header[1]) header[1] = #D6;      /* set the INS to UPDATE BINARY */
    *buffer = malloc((req¬dsize & #FFFF) + 3 + 4);      /* payload + payloadsize + header */
    ((char *) *buffer)[6] = (req¬dsize & #FF);
    ((char *) *buffer)[5] = ((req¬dsize & #FF00) ≫ 4);
    ((char *) *buffer)[4] = 0;      /* Lc = req¬dsize, Data = req¬data, Le = null */
      /* buffer[0] = 0 ⇒ extended format Lc */
    void *b = *buffer + 7;      /* WTF?! */
    memcpy(b, *buffer, req¬dsize);
    break;
  case APPEND: default: return false;
  }
  header[3] = (req¬apdu.binary.offset & #FF);
  header[2] = ((req¬apdu.binary.offset & #7F00) ≫ 4);
  memcpy(*buffer, header, 4);
  break;
```

This code is used in section 81.

83. Procedure *usb_device_on*( ). Initialises a USB device and tells it to turn on.

```
⟨ USB Internal Functions 68 ⟩ +≡
  void usb_device_on(usb_dev_handle * handle)
  {
    unsigned char buffer[1 + 256 + 2];
    int nlength;
    nlength = sizeof (buffer);
    unsigned char cmd[10];
    int retval;
    int length = 1;
    cmd[0] = #62;      /* IccPowerOn */
    cmd[1] = cmd[2] = cmd[3] = cmd[4] = 0;      /* dwLength */
    cmd[5] = 0;      /* slot number */
    cmd[6] = #05;
    cmd[7] = #01;      /* 5.0V */
    cmd[8] = cmd[9] = 0;      /* RFU */
    length = sizeof (cmd);
    retval = usb_bulk_write(handle, #02, cmd, length, 60000);
    nlength = usb_bulk_read(handle, #02, buffer, nlength, 6000);
  }
```

84. The Library used by client applications.

This file contains all the code linked to by applications wanting to access the smart cards. The associated header file contains all the functions exported to the client to communicate with a smart card. Programs using this may have to *free*(2) or *malloc*(2) some data buffers and should read the documentation below carefully to know when this is their responsibility.

Communication over the socket to the daemon process is handled by the shared protocol layer, we merely make calls to that.

85. Exported interfaces. These interfaces are exported via the file `library.h`

⟨ `library.h` 85 ⟩ ≡
```
#ifndef _CCID_LIBRARY_H
#define _CCID_LIBRARY_H
#include "../share/types.h"
#include <sys/socket.h>
  ⟨ Library Type Definitions 89 ⟩
  ⟨ Exported Library Functions 88 ⟩
#endif
```

86. System header files used. We use string and error handling libraries

```
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
```

87. Local header files. These give us access to the protocol layer, debugging functions, and the exported function declarations from the library header file.

```
#include "library.h"
#include "../share/protocol.h"
#include "../share/types.h"
#include "../share/debug.h"
#include "../share/log.h"
```

88. Function *get_card_reference*( ).

This function allows you to request a connection to a smart card. You can specify the card to request in several different ways, some of which are blocking, and some are non-blocking.

Blocking requests:

You can make a request for the next available smart card. This request will block until either a smart card is inserted, or one in use by a different card is made available.

Non-Blocking requests:

You can make a request which will return a card if there is one available, but will return immediately with a `NOT_AVAIL` message if there are no free cards.

Specfiying cards

Smart cards can be specified more precisely by giving a pattern to match describing them. If you specify a file and contents, then any candidate card to be returned will be checked to see if that file exists, and if the contents match the string given in the second parameter. If *file* $\equiv \Lambda$, then no checks will be performed. Otherwise, if *content* $\equiv \Lambda$ then the file will be checked for existance, but not for content.

Possible Requests

Request	Blocking	Description
`BLOCK_NEXT_AVAIL`	yes	Blocks until a card is available and returns a descriptor.
`AVAIL`	no	Returns the error `NOT_AVAIL` if a card isn't available. This will not block

⟨ Exported Library Functions 88 ⟩ ≡

```
int get_card_reference(card_ref * ref, CARD_REQUEST request, char *file, char *content);
```

See also sections 92, 94, 96, 98, 100, 102, 104, 106, 108, 110, 112, 114, 116, 119, 121, 123, 125, 127, and 129.

This code is used in section 85.

89. `CARD_REQUEST` Type definition. This type selects what type of request to make.

⟨ Library Type Definitions 89 ⟩ ≡

```
typedef enum {
  AVAIL, BLOCK_NEXT_AVAIL
} CARD_REQUEST;
```

See also section 90.

This code is used in section 85.

90. Card references. A card reference contains the information about how to connect to one smart card. A card reference must be retrieved from a *get_card_reference*( ) call, and should be relinquished with a *release_card*( ) call when finished with.

Since sockets are being used to identify cards in the cardd, this is merely a socket identifier and info block.

⟨ Library Type Definitions 89 ⟩ +≡

```
struct card_ref_struct {
  struct sockinfo_struct {
    sa_family_t family;
    char sockpath[MAXSOCKETPATH];
  } info;
  int socket;
};
typedef struct card_ref_struct card_ref;
```

91. Implementation of function.

```
int get_card_reference(card_ref *ref, CARD_REQUEST request, char *file, char *content)
{
  socklen_t len;
  pid_t pid;
  int rc;
  debug("socket: %d", ref→socket);
  ref→socket = socket(AF_UNIX, SOCK_STREAM, 0);
  debug("getting socket, errno: (%d) %s", errno, strerror(errno));
  assert(ref→socket > 0, "Can't get socket fd");
  debug("socket: %d", ref→socket);
  ref→info.family = AF_UNIX;
  strcpy(ref→info.sockpath, SOCKETDIR);
  strcat(ref→info.sockpath, "/master");
  debug("master socket: %d, %s", ref→socket, ref→info.sockpath);
  len = sizeof (ref→info.family) + strlen(ref→info.sockpath) + 1;      /* connect */
  debug("connecting to daemon");
  if ((rc = connect(ref→socket, (struct sockaddr *) &(ref→info), len)) < 0) return rc;
      /* send our pid */
  pid = getpid( );
  debug("sending pid (%d)", pid);
  if ((rc = send(ref→socket, &pid, sizeof (pid_t), 0)) < 0) return rc;
  return 0;
}
```

92. Function *release_card*( ). Causes the cardd to close the connection to a smart card and make it available for other applications. This also invalidates the **card_ref** block.

⟨ Exported Library Functions 88 ⟩ +≡

```
int release_card(card_ref ref);
```

93. Implementation of function.

```
int release_card(card_ref ref)
{     /* TODO: something */
  return 0;
}
```

94. Function *apdu_read_binary*(). Sends an APDU READ BINARY command. We read *length* bytes from position *offset* in the currently selected EF. These are returned in the buffer *returneddata*. **You must** *malloc*(2) *length* **bytes of** *returneddata*. If no data is returned, then *returnedlength* will be 0.

Binary commands can only be performed on EFs with transparent structure.

The return value from this function will be the APDU status that the card returns.

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_read_binary(card_ref ref, int offset, int length, char *returneddata, int *returnedlength);
```

95. Implementation of function.

```
int apdu_read_binary(card_ref ref, int offset, int length, char *returneddata, int *returnedlength)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_BINARY;
  req.apdu.binary.mode = READ;
  req.apdu.binary.offset = offset;
  req.apdu.binary.length = length;
  req.dsize = 0;
  req.data = Λ;      /* setup the request */
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  *returnedlength = res.rsize;
  if (Λ ≠ res.rdata) {
    memcpy(res.rdata, returneddata, min(length, res.rsize));
    free(res.rdata);
    res.rdata = Λ;
  }
  else returnedlength = 0;
  return res.apdu.status;
}
```

96. Function *apdu_write_binary* **( ).** The WRITE BINARY command writes binary values to the currently selected EF. Depending on the attributes of the file it will either OR the bits of the file with the bits supplied in the command, AND the two together, or simply overwrite them. The *length* bytes specified by *data* will be written to byte position *offset* in the currently selected EF.

Binary commands can only be performed on EFs with transparent structure.

The return value from this function will be the APDU status that the card returns.

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_write_binary(card_ref ref, int offset, int length, char *data);
```

97. Implementation of function.

```
int apdu_write_binary(card_ref ref, int offset, int length, char *data)
{
  ccid_request req;
  ccid_response res;
  debug("apdu_write_binary");
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_BINARY;
  req.apdu.binary.mode = WRITE;
  req.apdu.binary.offset = offset;
  req.apdu.binary.length = 0;
  req.dsize = length;
  req.data = data;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) {
    free(res.rdata);
  }
  return res.apdu.status;
}
```

98. Function *apdu_update_binary*( ). An UPDATE BINARY command updates the bytes from position *offset* in the currently selected EF, with the bytes specified by *length* and *data*.

Binary commands can only be performed on EFs with transparent structure.

The return value from this function will be the APDU status that the card returns.

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_update_binary(card_ref ref, int offset, int length, char *data);
```

99. Implementation of function.

```
int apdu_update_binary(card_ref ref, int offset, int length, char *data)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_BINARY;
  req.apdu.binary.mode = UPDATE;
  req.apdu.binary.offset = offset;
  req.apdu.binary.length = 0;
  req.dsize = length;
  req.data = data;
  if (¬send_ccid_request(ref.socket, req, &res)) ;    /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;    /* TODO: error type stuff */
  if (Λ ≠ res.rdata) {
    free(res.rdata);
  }
  return res.apdu.status;
}
```

100. Function *apdu_erase_binary*( ). An ERASE BINARY command will set *length* bytes from *offset* in the currently selected EF to the logically erased state.

Binary commands can only be performed on EFs with transparent structure.

The return value from this function will be the APDU status that the card returns.

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_erase_binary(card_ref ref, int offset, int length);
```

101. Implementation of function.

```
int apdu_erase_binary(card_ref ref, int offset, int length)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_BINARY;
  req.apdu.binary.mode = WRITE;
  req.apdu.binary.offset = offset;
  req.apdu.binary.length = length;
  req.dsize = 0;
  req.data = Λ;
  if (¬send_ccid_request(ref.socket, req, &res)) ;    /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;    /* TODO: error type stuff */
  if (Λ ≠ res.rdata) {
    free(res.rdata);
  }
  return res.apdu.status;
}
```

102. Function *apdu_read_records*( ). This will read records from an EF. If *ef* is set to `CURRENT_EF` then records will be read from the currently selected EF. Otherwise they will be read from the specified EF. The record to start reading from are specified by *type* and *record*. If *type* ∈ {`FIRST`,`LAST`,`NEXT`,`PREVIOUS`} then the appropriate record will be read from. If *type* ≡ `SPECIFY`, then the record number will be read from *record*. *length* bytes of data are read from the starting record.

You must *malloc*(2) *length* **bytes of** *returneddata*. If no data is returned, then *returnedlength* will be 0.

The return value from this function will be the APDU status that the card returns.

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_read_records(card_ref ref, int ef, RECORD_TYPE type, int record, int length, int
    *returnedlength, char *returndata);
```

103. Implementation of function.

```
int apdu_read_records(card_ref ref, int ef, RECORD_TYPE type, int record, int length, int
        *returnedlength, char *returndata)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_RECORD;
  req.apdu.record.mode = READ;
  req.apdu.record.ef = ef;
  req.apdu.record.type = type;
  req.apdu.record.record = record;
  req.apdu.record.length = 0;
  req.dsize = 0;
  req.data = Λ;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  returnedlength = res.rsize;
  if (Λ ≠ res.rdata) {
    memcpy(res.rdata, returndata, min(length, res.rsize));
    free(res.rdata);
  }
  else returnedlength = 0;
  return res.apdu.status;
}
```

104. Function *apdu_write_record*( ). This will write records to an EF. If *ef* is set to `CURRENT_EF` then records will be written to the currently selected EF. Otherwise they will be written to the specified EF. Depending on the attributes of the file it will either OR the bits of the file with the bits supplied in the command, AND the two together, or simply overwrite them. The record to start writing to from are specified by *type* and *record*. If *type* ∈ {`FIRST`,`LAST`,`NEXT`,`PREVIOUS`} then the appropriate record will be written to. If *type* ≡ `SPECIFY`, then the record number will be read from *record*. *length* bytes of data are written from that point.

The return value from this function will be the APDU status that the card returns.

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_write_record(card_ref ref, int ef, RECORD_TYPE type, int record, int length, char *data);
```

105. Implementation of function.

```
int apdu_write_record(card_ref ref, int ef, RECORD_TYPE type, int record, int length, char *data)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_RECORD;
  req.apdu.record.mode = WRITE;
  req.apdu.record.ef = ef;
  req.apdu.record.type = type;
  req.apdu.record.record = record;
  req.apdu.record.length = 0;
  req.dsize = length;
  req.data = data;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) free(res.rdata);
  return res.apdu.status;
}
```

106. Function *apdu_append_record*( ). This will append records to an EF. If *ef* is set to `CURRENT_EF` then records will be appended to the end of the currently selected EF. Otherwise they will be appended to the specified EF.

If the EF is a cyclic structure, then it will write to record 1.

The record pointer is set to the appended record.

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_append_record(card_ref ref, int ef, int length, char *data);
```

107. Implementation of function.

```
int apdu_append_record(card_ref ref, int ef, int length, char *data)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_RECORD;
  req.apdu.record.mode = APPEND;
  req.apdu.record.ef = ef;
  req.apdu.record.length = 0;
  req.apdu.record.record = 0;
  req.dsize = length;
  req.data = data;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) free(res.rdata);
  return res.apdu.status;
}
```

108. Function *apdu_update_record*( ). This will write records to an EF. If *ef* is set to `CURRENT_EF` then records will be written to the currently selected EF. Otherwise they will be written to the specified EF. Depending on the attributes of the file it will either OR the bits of the file with the bits supplied in the command, AND the two together, or simply overwrite them. The record to start writing to from are specified by *type* and *record*. If *type* ∈ {`FIRST`,`LAST`,`NEXT`,`PREVIOUS`} then the appropriate record will be written to. If *type* ≡ `SPECIFY`, then the record number will be read from *record*. *length* bytes of data are written from that point.

The return value from this function will be the APDU status that the card returns.

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_update_record(card_ref ref, int ef, RECORD_TYPE type, int record, int length, char *data);
```

109. Implementation of function.

```
int apdu_update_record(card_ref ref, int ef, RECORD_TYPE type, int record, int length, char *data)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_RECORD;
  req.apdu.record.mode = UPDATE;
  req.apdu.record.ef = ef;
  req.apdu.record.type = type;
  req.apdu.record.record = record;
  req.apdu.record.length = 0;
  req.dsize = length;
  req.data = data;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) free(res.rdata);
  return res.apdu.status;
}
```

110. Function *apdu_get_data*( ). **You must** *malloc*(2) *responselength* **bytes of** *responsedata*.

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_get_data(card_ref ref, TAG_TYPE tagtype, int tag, int responselength, char *responsedata);
```

111. Implementation of function.

```
int apdu_get_data(card_ref ref, TAG_TYPE tagtype, int tag, int responselength, char *responsedata)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_DATA;
  req.apdu.data.mode = READ;
  req.apdu.data.tagtype = tagtype;
  req.apdu.data.tag = tag;
  req.apdu.data.length = responselength;
  req.dsize = 0;
  req.data = Λ;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) {
    memcpy(res.rdata, responsedata, min(responselength, res.rsize));
    free(res.rdata);
  }
  else responsedata = Λ;
  return res.apdu.status;
}
```

112. Function *apdu_put_data*( ).

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_put_data(card_ref ref, TAG_TYPE tagtype, int tag, int length, char *data);
```

113. Implementation of function.

```
int apdu_put_data(card_ref ref, TAG_TYPE tagtype, int tag, int length, char *data)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_DATA;
  req.apdu.data.mode = WRITE;
  req.apdu.data.tagtype = tagtype;
  req.apdu.data.tag = tag;
  req.apdu.data.length = 0;
  req.dsize = length;
  req.data = data;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) free(res.rdata);
  return res.apdu.status;
}
```

114. Function *apdu_select_file_id*( ). **You must** *malloc*(2) *responselength* **bytes of** *responsedata*.

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_select_file_id(card_ref ref, FILE_TYPE type, int DF, int responsetemplate, int responselength, char
    *responsedata);
```

115. Implementation of function.

```
int apdu_select_file_id(card_ref ref, FILE_TYPE type, int DF, int responsetemplate, int responselength, char
        *responsedata)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_SELECT;
  req.apdu.select.mode = ID;
  req.apdu.select.rtemplate = responsetemplate;
  req.apdu.select.rsize = responselength;
  req.apdu.select.DF = DF;
  req.apdu.select.type = type;
  req.dsize = 0;
  req.data = Λ;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) {
    memcpy(res.rdata, responsedata, min(res.rsize, responselength));
    free(res.rdata);
  }
  else responsedata = Λ;
  return res.apdu.status;
}
```

116. Function *apdu_select_file_path*( ). **You must** *malloc*(2) *responselength* **bytes of** *responsedata*.

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_select_file_path(card_ref ref, bool relative, char *path, int responsetemplate, int
    responselength, char *responsedata);
```

117. Implementation of function.

```
int apdu_select_file_path(card_ref ref, bool relative, char *path, int responsetemplate, int
        responselength, char *responsedata)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_SELECT;
  req.apdu.select.mode = PATH;
  req.apdu.select.rtemplate = responsetemplate;
  req.apdu.select.rsize = responselength;
  req.apdu.select.relative = relative;
  req.dsize = strlen(path) + 1;
  req.data = path;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) {
    memcpy(res.rdata, responsedata, min(res.rsize, responselength));
    free(res.rdata);
  }
  else responsedata = Λ;
  return res.apdu.status;
}
```

118. Function *apdu_select_file_df*( ). **You must** *malloc*(2) *responselength* **bytes of** *responsedata*.

```
int apdu_select_file_df(card_ref ref, char *name, int responsetemplate, int responselength, char
        *responsedata)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_SELECT;
  req.apdu.select.mode = DF;
  req.apdu.select.rtemplate = responsetemplate;
  req.apdu.select.rsize = responselength;
  req.dsize = strlen(name) + 1;
  req.data = name;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) {
    memcpy(res.rdata, responsedata, min(res.rsize, responselength));
    free(res.rdata);
  }
  else responsedata = Λ;
  return res.apdu.status;
}
```

119. Function *apdu_verify*( ).

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_verify(card_ref ref, bool global, int reference, int length, char *data);
```

120. Implementation of function.

```
int apdu_verify(card_ref ref, bool global, int reference, int length, char *data)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_VERIFY;
  req.apdu.verify.global = global;
  req.apdu.verify.reference = reference;
  req.dsize = length;
  req.data = data;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) free(res.rdata);
  return res.apdu.status;
}
```

121. Function *apdu_authenticate_internal*( ). **You must** *malloc*(2) *responselength* **bytes of** *response*.

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_authenticate_internal(card_ref ref, bool globalsecret, int algorithmid, int secretid, int
    challengelength, char *challenge, int responselength, char *response);
```

122. Implementation of function.

```
int apdu_authenticate_internal(card_ref ref, bool globalsecret, int algorithmid, int secretid, int
        challengelength, char *challenge, int responselength, char *response)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_AUTHENTICATE;
  req.apdu.auth.internal = true;
  req.apdu.auth.globalsecret = globalsecret;
  req.apdu.auth.algorithmid = algorithmid;
  req.apdu.auth.secretid = secretid;
  req.apdu.auth.responselength = responselength;
  req.dsize = challengelength;
  req.data = challenge;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) {
    memcpy(res.rdata, response, min(res.rsize, responselength));
    free(res.rdata);
  }
  else response = Λ;
  return res.apdu.status;
}
```

123. Function *apdu_authenticate_external*( ).

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_authenticate_external(card_ref ref, bool globalsecret, int algorithmid, int secretid, int
    responselength, char *response);
```

124. Implementation of function.

```
int apdu_authenticate_external(card_ref ref, bool globalsecret, int algorithmid, int secretid, int
        responselength, char *response)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_AUTHENTICATE;
  req.apdu.auth.internal = false;
  req.apdu.auth.globalsecret = globalsecret;
  req.apdu.auth.algorithmid = algorithmid;
  req.apdu.auth.secretid = secretid;
  req.apdu.auth.responselength = 0;
  req.dsize = responselength;
  req.data = response;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) free(res.rdata);
  return res.apdu.status;
}
```

125. Function *apdu_get_challenge*( ). **You must** *malloc*(2) *maxlength* **bytes of** *challengereturned*.

⟨ Exported Library Functions 88 ⟩ +≡

```
int apdu_get_challenge(card_ref ref, int maxlength, int *returnedlength, char *challengereturned);
```

126. Implementation of function.

```
int apdu_get_challenge(card_ref ref, int maxlength, int *returnedlength, char *challengereturned)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_CHALLENGE;
  req.apdu.challenge.maxlength = maxlength;
  req.dsize = 0;
  req.data = Λ;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  *returnedlength = res.rsize;
  if (Λ ≠ res.rdata) {
    memcpy(res.rdata, challengereturned, min(maxlength, res.rsize));
    free(res.rdata);
  }
  else memset(challengereturned, 0, maxlength);
  return res.apdu.status;
}
```

127. Function *apdu_channel_open*( ).

⟨ Exported Library Functions 88 ⟩ +≡
```
int apdu_channel_open(card_ref ref, int *channelno);
```

128. Implementation of function.
```
int apdu_channel_open(card_ref ref, int *channelno)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_CHANNEL;
  req.apdu.channel.open = true;
  req.apdu.channel.channelno = 0;
  req.dsize = 0;
  req.data = Λ;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) {
    *channelno = *(int *) res.rdata;
    free(res.rdata);
  }
  else channelno = 0;
  return res.apdu.status;
}
```

129. Function *apdu_channel_close*( ).

⟨ Exported Library Functions 88 ⟩ +≡
```
int apdu_channel_close(card_ref ref, int channelno);
```

130. Implementation of function.
```
int apdu_channel_close(card_ref ref, int channelno)
{
  ccid_request req;
  ccid_response res;
  req.protocol = APDU;
  req.apdu.apdu_type = APDU_CHANNEL;
  req.apdu.channel.open = false;
  req.apdu.channel.channelno = channelno;
  req.dsize = 0;
  req.data = Λ;
  if (¬send_ccid_request(ref.socket, req, &res)) ;      /* TODO: error type stuff */
  if (APDU ≠ res.protocol) ;      /* TODO: error type stuff */
  if (Λ ≠ res.rdata) free(res.rdata);
  return res.apdu.status;
}
```

131. Debug code.

132. Headers needed for debug code.

```
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "types.h"
#ifdef DEBUG
#include "protocol.h"
#endif
```

133. Exported interfaces.

These macros and functions are exported via the `debug.h` file and used in most of the other files.

⟨ `debug.h` 133 ⟩ ≡

```
#ifndef _CCID_DEBUG_H
#define _CCID_DEBUG_H
#include <errno.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "types.h"
#ifdef DEBUG
#include "protocol.h"
#endif
#ifdef DEBUG
#define debug (a, b... ) do
        {
          fprintf(stderr, "[Debug]␣");
          fprintf(stderr, a, ##b);
          fprintf(stderr, "\n");
        }
        while (0)
#else
#define debug(...)
#endif
#ifdef DEBUG
#define print_ccid_request(a)real_print_ccid_request (a)
        void real_print_ccid_request(ccid_request * req);
#else
#define print_ccid_request (a)
#endif
#ifdef DEBUG
#define print_ccid_response(a)real_print_ccid_response (a)
        void real_print_ccid_response(ccid_response * res);
#else
#define print_ccid_response (a)
#endif
#endif
```

134. Procedure *real_print_ccid_request*( ).

```
#ifdef DEBUG
  void real_print_ccid_request(ccid_request * req)
  {
    fprintf(stderr, "request␣{\n\tprotocol:␣%d\n\tdsize:␣%d\n\tapdu␣{\n␣␣␣␣␣␣␣␣␣␣\
        ␣␣␣␣␣␣␣␣␣␣\t\ttype:␣%d\n", req→protocol, req→dsize, req→apdu.apdu_type);
    switch (req→apdu.apdu_type) {
    case APDU_AUTHENTICATE: fprintf(stderr, "\t\tauth␣{\n\t\t\tinternal:␣%s\n\t\t\tg\
          lobalsecret:␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣%s\n\t\t\talgorithmid:␣%d\n\t\t\tsecretid\
          :␣%d\n␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\t\t\tresponselength:␣%d\n\t\t}\n",
          req→apdu.auth.internal ? "true" : "false", req→apdu.auth.globalsecret ? "true" : "false",
          req→apdu.auth.algorithmid, req→apdu.auth.secretid, req→apdu.auth.responselength);
      break;
    case APDU_BINARY:
      fprintf(stderr, "\t\tbinary␣{\n\t\t\tmode:␣%d\n\t\t\toffset:␣%d␣␣␣␣␣␣␣␣␣␣␣␣␣\
          ␣␣␣␣␣␣␣␣␣␣␣␣␣\n\t\t\tlength:␣%d\n\t\t}\n", req→apdu.binary.mode,
          req→apdu.binary.offset, req→apdu.binary.length);
      break;
    case APDU_CHALLENGE: fprintf(stderr, "\t\tchallenge␣{\n\t\t\tmaxlength:␣%d\n\t\t}\n",
          req→apdu.challenge.maxlength);
      break;
    case APDU_CHANNEL:
      fprintf(stderr, "\t\tchannel␣{\n\t\t\topen:␣%s\n\t\t\tchannelno:␣%d␣␣␣␣␣␣␣␣␣\
          ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n\t\t}\n", req→apdu.channel.open ? "true" : "false",
          req→apdu.channel.channelno);
      break;
    case APDU_DATA:
      fprintf(stderr, "\t\tdata␣{\n\t\t\tmode:␣%d\n\t\t\ttagtype:␣%d␣␣␣␣␣␣␣␣␣␣␣␣␣\
          ␣␣␣␣␣␣␣␣␣␣␣␣␣\n\t\t\ttag:␣%d\n\t\t\tlength:␣%d\n\t\t}\n", req→apdu.data.mode,
          req→apdu.data.tagtype, req→apdu.data.tag, req→apdu.data.length);
      break;
    case APDU_RECORD: fprintf(stderr,
          "\t\trecord␣{\n\t\t\tmode:␣%d\n\t\t\ttype:␣%d␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n\t\t\tr\
          ecord:␣%d\n\t\t\tlength:␣%d\n\t\t\t␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣ef:␣%d\n\t\t}\n",
          req→apdu.record.mode, req→apdu.record.type, req→apdu.record.record, req→apdu.record.length,
          req→apdu.record.ef);
      break;
    case APDU_SELECT:
      fprintf(stderr, "\t\tselect␣{\n\t\t\tmode:␣%d\n\t\t\trtemplate:␣%d␣␣␣␣␣␣␣␣␣␣\
          ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n\t\t\trsize:␣%d\n", req→apdu.select.mode, req→apdu.select.rtemplate,
          req→apdu.select.rsize);
      switch (req→apdu.select.mode) {
      case DF:
        fprintf(stderr, "\t\t\ttype:␣%d\n\t\t\tDF:␣%d\n", req→apdu.select.type, req→apdu.select.DF);
        break;
      case PATH: fprintf(stderr, "\t\t\trelative:␣%s\n", req→apdu.select.relative ? "true" : "false");
        break;
      case ID: break;
      }
      fprintf(stderr, "\t\t}\n");
      break;
```

```
    case APDU_VERIFY:
      fprintf(stderr, "\t\tverify␣{\n\t\t\tglobal:␣%s\n\t\t\treference:␣%d␣␣␣␣␣␣␣␣\
          ␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣\n\t\t}\n", req→apdu.verify.global ? "true" : "false",
          req→apdu.verify.reference);
    }
    fprintf(stderr, "\t}\n}\n");
  }
#endif
```

135. Procedure *real_print_ccid_response*( ).

```
#ifdef DEBUG
  void real_print_ccid_response(ccid_response * res)
  {
    fprintf(stderr, "response␣{\n\tprotocol:␣%d\n\trsize:␣%d\n\tapdu␣{\n\t\ttype\
        :␣%d\n\t\tstatus:␣%d\n\t}\n}\n", res→protocol, res→rsize, res→apdu.apdu_type, res→apdu.status);
  }
#endif
```

136. Logging functions.
This file contains functions for sending log messages to syslog, and to perform conditional logging.

137. Header files and global variables. We need syslog header files, and also debugging code. We store a static global variable which governs whether to use syslog or to print messages on *stderr*.

```
#include <syslog.h>
#include <stdarg.h>
#include <errno.h>
#include "log.h"
#include "debug.h"
  bool _do_syslog = false;
```

138. Exported interfaces. The file `log.h` exports several macros and functions to other files.

```
⟨ log.h 138 ⟩ ≡
#ifndef __CCID_LOG_H
#define __CCID_LOG_H
#include "types.h"
#ifdef DEBUG
#define assert(a, b) do
  {
    bool __T = a;
    fprintf(stderr, "[Assert] %s\n", __T ? "true" : "false");
    if (¬__T) real_assert(b);
  }
  while (0)
#else
#define assert(a, b) if (¬(a)) real_assert(b)
#endif
  void real_assert(char *message);
  void slog(char *fmt, ...);
  void setup_syslog(bool log);
#endif
```

139. Procedure *real_assert*( ).

```
  void real_assert(char *message)
  {
    if (0 ≠ errno) {
      char *errstr = strerror(errno);
      slog("[Assert] failed with errno %d/%s: %s\n", errno, errstr, message);
    }
    else slog("[Assert] failed: %s\n", message);
#ifdef DEBUG
    exit(1);
#endif
  }
```

140. Procedure *slog*( ).

```
void slog(char *fmt, ...)
{
  va_list args;
  va_start(args, fmt);
  if (_do_syslog) vsyslog(LOG_NOTICE, fmt, args);
  else {
    fprintf(stderr, "[ccid-cardd] ");
    vfprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
  }
  va_end(args);
}
```

141. Procedure *setup_syslog*( ).

```
void setup_syslog(bool log)
{
  _do_syslog = log;
  if (log) openlog("ccid-cardd", 0, LOG_DAEMON);
}
```

142. The protocol layer.
This section contains the glue code which allows the two parts of the driver to talk to each other.

143. Headers needed for the protocol layer.

```
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include "protocol.h"
#include "types.h"
#include "debug.h"
```

144. Exported interfaces.
We export a lot of structures for encoding the CCID requests into, and also a few functions for communicating with them. These are exported in `protocol.h`.

```
⟨ protocol.h 144 ⟩ ≡
#ifndef _CCID_PROTOCOL_H
#define _CCID_PROTOCOL_H
#include "types.h"
#define SOCKETDIR "/var/run/ccid-cardd"
  ⟨ Definitions of enum types 145 ⟩
  ⟨ Definitions of APDU structures 147 ⟩
  ⟨ Definitions of CCID structures 146 ⟩
  ⟨ Declarations of CCID functions 157 ⟩
#endif
```

145. Enumerated type definitions.

```
⟨ Definitions of enum types 145 ⟩ ≡
  typedef enum {
    READ, WRITE, UPDATE, ERASE, APPEND
  } MODE;
  typedef enum {
    ID, PATH, DF
  } SELECT_MODE;
  typedef enum {
    APDU, TPDU, PARAM
  } PROTOCOL_TYPE;
  typedef enum {
    APDU_AUTHENTICATE, APDU_BINARY, APDU_CHALLENGE, APDU_CHANNEL, APDU_DATA, APDU_RECORD,
        APDU_SELECT, APDU_VERIFY
  } APDU_TYPE;
```

This code is used in section 144.

146. CCID structures. These structures are a generic wrapper which may contain one of several protocols.

```
⟨ Definitions of CCID structures 146 ⟩ ≡
  struct ccid_request_struct {
    PROTOCOL_TYPE protocol;
    union {
      struct apdu_request_struct apdu;
    };
    int dsize;
    char *data;
  };
  struct ccid_response_struct {
    PROTOCOL_TYPE protocol;
    union {
      struct apdu_response_struct apdu;
    };
    int rsize;
    char *rdata;
  };
  typedef struct ccid_request_struct ccid_request;
  typedef struct ccid_response_struct ccid_response;
```

This code is used in section 144.

147. APDU BINARY structure. Encodes APDU BINARY commands

```
⟨ Definitions of APDU structures 147 ⟩ ≡
  struct apdu_binary {
    MODE mode;
    int offset;
    int length;
  };
```

See also sections 148, 149, 150, 151, 152, 153, 154, 155, and 156.

This code is used in section 144.

148. APDU RECORD structure. Encodes APDU RECORD commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_record {
    MODE mode;
    int ef;
    RECORD_TYPE type;
    int record;
    int length;
  };
```

149. APDU DATA structure. Encodes APDU DATA commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_data {
    MODE mode;
    TAG_TYPE tagtype;
    int tag;
    int length;
  };
```

150. APDU SELECT structure. Encodes APDU SELECT commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_select {
    SELECT_MODE mode;
    int rtemplate;
    int rsize;
    union {
      struct {
        FILE_TYPEtype;
        int DF;
      };
      struct {
        bool relative;
      };
    };
  };
```

151. APDU VERIFY structure. Encodes APDU VERIFY commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_verify {
    bool global;
    int reference;
  };
```

152. APDU AUTHENTICATE structure. Encodes APDU AUTHENTICATE commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_authenticate {
    bool internal;
    bool globalsecret;
    int algorithmid;
    int secretid;
    int responselength;
  };
```

153. APDU CHALLENGE structure. Encodes APDU CHALLENGE commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_challenge {
    int maxlength;
  };
```

154. APDU CHANNEL structure. Encodes APDU CHANNEL commands

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_channel {
    bool open;
    int channelno;
  };
```

155. APDU REQUEST structure. This is a union of the various APDU structures for the different commands. the *apdu_type* field defines which member of the union should be accessed.

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_request_struct {
    int apdu_type;
    union {
      struct apdu_binary binary;
      struct apdu_record record;
      struct apdu_data data;
      struct apdu_select select;
      struct apdu_verify verify;
      struct apdu_authenticate auth;
      struct apdu_challenge challenge;
      struct apdu_channel channel;
    };
  };
```

156. APDU RESPONSE structure. This encodes responses to an APDU command.

```
⟨ Definitions of APDU structures 147 ⟩ +≡
  struct apdu_response_struct {
    APDU_TYPE apdu_type;
    int status;
  };
```

157. Functions which send and/or receive CCID structures.

```
⟨ Declarations of CCID functions 157 ⟩ ≡
  int send_ccid_request(int skt, ccid_request req, ccid_response *res);
  int get_ccid_request(int skt, ccid_request *req);
  int send_ccid_response(int skt, ccid_response res);
```

This code is used in section 144.

158. Internal protocol functions.

⟨ Internal protocol functions 159 ⟩;

159. Function *send_bytes*( ). Sends a given number of bytes from a buffer on a socket.

```
⟨ Internal protocol functions 159 ⟩ ≡
  int send_bytes(int skt, void *buf, int length)
  {
    int ret;
    do {
      ret = send(skt, buf, length, 0);
      debug("sent␣%d␣bytes", ret);
      if (ret < 0) return ret;
      length -= ret;
    } while (length > 0);
    return 0;
  }
```

See also section 160.

This code is used in section 158.

160. Function *recv_bytes*( ). Receives a given number of bytes from a buffer on a socket.

```
⟨ Internal protocol functions 159 ⟩ +≡
  int recv_bytes(int skt, void *buf, int length)
  {
    int ret;
    do {
      ret = recv(skt, buf, length, 0);
      debug("got␣%d␣bytes", ret);
      if (ret < 0) return ret;
      length -= ret;
    } while (length > 0);
    return 0;
  }
```

161. Function *send_ccid_request*( ). **This may** *malloc*(2) *res→rdata*. **If the return value is** 0 **and** *res→rdata* $\neq \Lambda$ **then you must** *free*(2).

```
int send_ccid_request(int skt, ccid_request req, ccid_response *res)
{
  int ret = 0;
  debug("sending request");
  if ((ret = send_bytes(skt, (void *) &req, sizeof(ccid_request))) < 0) return ret;
      /* send request */
  if (Λ ≠ req.data) {
    debug("sending data (size: %d)", req.dsize);
    if ((ret = send_bytes(skt, (void *) req.data, req.dsize)) < 0) return ret;
  }
  debug("receiving response");
  if ((ret = recv_bytes(skt, (void *) res, sizeof(ccid_response))) < 0) return ret;
      /* get response */
  print_ccid_response(res);
  if (0 ≡ res→rsize) {
    debug("No data");
    res→rdata = Λ;
    return 0;
  }
  res→rdata = malloc(res→rsize);
  if ((ret = recv_bytes(skt, (void *) res→rdata, res→rsize)) < 0) {
    free(res→rdata);
    res→rdata = Λ;
    return ret;
  }     /* get data */
  if (ret < 0) return ret;
  else return 0;      /* return status */
}
```

162. Function *send_ccid_request*( ). **This may** *malloc*(2) *req*→*rdata*. **If the return value is** 0 **and** *req*→*rdata* ≠ Λ **then you must** *free*(2).

```
int get_ccid_request(int skt, ccid_request *req)
{
  int ret = 0;
  debug("get structure");
  if ((ret = recv_bytes(skt, (void *) req, sizeof(ccid_request))) < 0) return ret;
      /* get structure */
  print_ccid_request(req);
  ret = 0;
  debug("get data (size: %d)", req→dsize);      /* get data */
  if (0 ≡ req→dsize) {
    req→data = Λ;
    return 0;
  }
  else {
    req→data = malloc(req→dsize);
    if ((ret = recv_bytes(skt, req→data, req→dsize)) < 0) {
      debug("Freeing...");
      free(req→data);
      req→data = Λ;
      return ret;
    }
    debug("Got data");
  }
  return 0;
}
```

163. Function *send_ccid_response*( ).

```
int send_ccid_response(int skt, ccid_response res)
{
  int ret = 0;
  debug("sending␣response");
  if ((ret = send_bytes(skt, (void *) &res, sizeof(ccid_response))) < 0) return ret;
      /* send response */
  if (Λ ≠ res.rdata) {
    debug("sending␣data␣(size:␣%d)", res.rsize);
    if ((ret = send_bytes(skt, (void *) res.rdata, res.rsize)) < 0) return ret;
  }
  return 0;
}
```

164. Extra types and macro definitions file. Any extra definitions of generic types or macros which are needed elsewhere are put in here.

```
⟨ types.h 164 ⟩ ≡
#ifndef _CCID_TYPES_H
#define _CCID_TYPES_H
#define MAXSOCKETPATH 100
#define USB_CLASS_CCID 0x0B /* The USB Class for CCID Devices */
#define max(a,b) ((a > b) ? (a) : (b))
#define min(a,b) ((a < b) ? (a) : (b))
  /* TODO: temporary TAG_TYPE RECORD_TYPE typedefs */
  typedef enum {
    FIRST, LAST, NEXT, PREVIOUS, SPECIFY
  } RECORD_TYPE;
  typedef int TAG_TYPE;
  typedef int FILE_TYPE;
  typedef short bool;
  typedef int cardid;
#define true 1
#define false 0
#endif
```

165. Index.

⟨ USB Library Functions 53, 55, 73, 76 ⟩ Used in section 50.
⟨ USB Static Variables 64 ⟩ Used in section 50.
⟨ `debug.h` 133 ⟩
⟨ `library.h` 85 ⟩
⟨ `log.h` 138 ⟩
⟨ `protocol.h` 144 ⟩
⟨ `structs.h` 38 ⟩
⟨ `types.h` 164 ⟩
⟨ `usb.h` 52 ⟩
⟨ *listen_sockets* Local variables 22 ⟩ Used in section 20.
⟨ *scan_usb* local variables 58, 60, 63 ⟩ Used in section 55.

CCID-SYSTEM