University of Mannheim
Laboratory for Dependable Distributed Systems

Bachelor Thesis

Design and Implementation of a Forensic Documentation Tool for Interactive Command-line Sessions

Tim Weber

February 23, 2010

Primary examiner:	Prof. Dr. Felix C. Freiling
Secondary examiner:	Dipl.-Inf. Andreas Dewald
Supervisor:	Prof. Dr. Felix C. Freiling

Bachelorstudiengang Software- und Internettechnologie

Abstract

In computer forensics, it is important to document examination of a computer system with as much detail as possible. Many experts use the software SCRIPT to record their whole terminal session while analyzing the target system. This thesis shows why SCRIPT's features are not sufficient for documentation that is to be used in court. A new system, FORSCRIPT, providing additional capabilities and mechanisms will be designed and developed in this thesis.

Contents

Acknowledgements

First of all, I would like to thank Prof. Dr. Freiling for the opportunity to write a thesis about this interesting subject and for supporting me during the process of writing.

I would also like to thank Andreas Dewald for being available as secondary examiner and Prof. Dr.-Ing. Effelsberg for postponing my thesis deadline.

Thanks to Alexander Brock for testing and fuzzing FORSCRIPT as well as proofreading the thesis.

Michael Stapelberg, thank you for testing FORSCRIPT and for giving me some hints about Unix system calls and why SCRIPT does some things the way it does.

Many thanks go to the free software community: The people who created C, GCC, Git, LaTeX, Linux, make, noweb, script and Vim, but especially those who create comprehensive documentation. I would like to explicitly mention the BSD *termios(4)* manual page as an example of how good documentation should look like.

Thanks to my father for giving me the time to study at my own pace, and thanks to the hacker community for inspiring me every day.

Finally, I would like to thank Nathalie for the love, support and understanding before, during and after this thesis.

1 Introduction

1.1 Background: Computer Forensics

Computer forensics is a branch of forensic science. [1] In the digital age we live in, an increasing number of crimes is performed using or at least aided by digital devices and computer systems. To analyze the evidence that may be present on these devices, specially trained experts are required. Having knowledge about the technology behind the systems, these forensic investigators are able to search for evidence without destroying traces, modifying or even accidentally inserting misleading data.

Principles and techniques of computer forensics are, among others, employed to

- analyze computers, mobile phones and other electronic devices a suspected criminal has used,
- recover data after a hardware or software failure,
- gain information during or after attacks or break-in attempts on a computer system.

Documentation of Terminal Sessions

A forensic investigator has to keep a detailed record of his or her actions while analyzing a system. That way, in case of dispute about a piece of evidence, another forensic investigator can review the steps that led to certain conclusions. This *forensic log* improves the credibility of the investigator and protects a possible defendant from false accusations. Additionally, the investigator protects himself from forgetting how the evidence was found and what additional details (which probably seemed to be not important at that time) were present.

The protocol consists of, depending on the type of analysis, notes on paper, images, videos and data files on the investigator's computer. For example, to perform a *static analysis* of a suspect's computer's hard disk drive, i.e. searching the drive for suspicious data without modifying it, an investigator normally uses his computer, which is equipped with a software that records every action the investigator performs.

Often a Unix-based operating system like Linux or Mac OS X and command-line based software (also called *CLI software* for its command-line user interface) is used to perform such an analysis, for example DD to create a snapshot of the suspect's hard drive, SHA1SUM to verify its integrity and other tools like FOREMOST to find evidence in the snapshot. All interaction with the forensic software takes place in a text-based interface; the investigator uses his keyboard to perform commands, his workstation responds by displaying[1] text and data. A text-based interface cannot display graphics or use the mouse[2].

In principle, CLI sessions can be documented quite easily by creating a piece of software that records everything typed on the keyboard and everything sent to the screen. The SCRIPT utility is often used to accomplish this; however, it has several limitations described in section 2.5 which greatly limit its usefulness as a forensic tool.

[1] also called "printing", even though the output appears on the screen, not on paper

[2] Using the mouse is possible via several extensions, but mouse commands are simply translated to special control characters and can be read by the application just like any other keyboard input.

1.2 Tasks

Several tasks have to be solved in this bachelor thesis:

- Analyze SCRIPT with regard to weaknesses concerning its usage as a forensic tool.
- Describe SCRIPT's output format and its disadvantages.
- Describe in detail an output format suitable for forensic usage.
- Implement a software for Linux that is used like SCRIPT, but creates output in the new forensic output format. In order to minimize the requirements a target system has to meet to be able to run the software, it has to be implemented in the *C* programming language.
- Document the software according to the methods of *literate programming*.

Literate programming [2] is a technique invented by Donald E. Knuth, the author of the TEX typesetting system. Instead of writing more or less commented source code, it propagates writing a continuous text with embedded code fragments. These do not necessarily appear in the order they are executed, but where they are didactically useful. The software NOWEB [3] is used to generate the layouted thesis as well as the final program's source code out of a single file.

1.3 Results

It is apparent that SCRIPT is not suited for forensic usage, especially because it does not record the user's input and data about the environment it is running in. A successor, FORSCRIPT, has been designed and developed in this thesis. Its output format is portable, extensible and contains detailed information about the environment. The disadvantages of SCRIPT are eliminated. Following the paradigm of literate programming, this thesis is FORSCRIPT and vice versa.

1.4 Outlook on the Thesis

Section 1, which you are currently reading, contains the introduction into the topic of computer forensics. It explains why detailed documentation of forensic analyses is an important task, what a command-line interface is, which subjects will be presented in this thesis and also provides an overview of the tasks and results.

In section 2, one of the most popular tools for recording interactive terminal sessions, SCRIPT, will be presented and the format of the files it generates will be described. Afterwards, several issues regarding its usage as a forensic tool are presented, leading to the conclusion that it should be replaced with a more suitable software.

This new software called FORSCRIPT will be drafted in section 3, focusing on its file format and the resulting properties. The invocation syntax of FORSCRIPT, which is based on that of SCRIPT, and the differences in behavior compared to SCRIPT is also described.

Section 4, by far the longest section, contains a detailed step-by-step description of FORSCRIPT's source code. It describes how to write FORSCRIPT's data format, parsing the command line, what a pseudo terminal is and how to create one to access the input and output streams of an application, how to deal with subprocesses and signals and other things.

The resulting application will be evaluated in section 5, which includes an example transcript file and a description of FORSCRIPT's known limitations.

Finally, section 6 summarizes the work that has been done. It talks about the future of FORSCRIPT and describes the next steps that should probably be taken to make it even more useful.

2 script

util-linux is the name of a collection of command-line utilities for Linux systems. It includes essential software like DMESG, FDISK, MKSWAP, MOUNT and SHUTDOWN as well as the SCRIPT and SCRIPTREPLAY utilities.

The original *util-linux* package [4] was abandoned in 2006. Today, it has been replaced by its successor *util-linux-ng* [5], a *fork* based on the last available *util-linux* version. *util-linux-ng* is under active development. The analysis of the original SCRIPT utility in this thesis is based on the most recent *util-linux-ng* release as of the time of writing, version 2.17.

2.1 Purpose

The purpose of SCRIPT is to record everything printed to the user's terminal into a file. According to its manual, "[i]t is useful for students who need a hardcopy record of an interactive session as proof of an assignment".

It can also record timing data, specifying the chronological progress of the terminal session, into a second file. Using both of these files, the accompanying utility SCRIPTREPLAY can display the recorded data in a video-like way.

2.2 Mode of Operation

In order to record the terminal session, SCRIPT creates a new *pseudo terminal* (PTY), which is a virtual, software-based representation of a terminal line, and attach itself to the "master" side of it, thereby being able to send and receive data to and from an application connected to the "slave" side of the PTY.

It launches a subprocess (also known as *child*), which launches the actual client application as its own subchild and then records the client application's output stream. The parent process forwards the user's input to the client application.

Recording terminates as soon as the child process exits.

2.3 Invocation

SCRIPT takes one optional argument, the file name of the output file (also called *typescript* file) to generate. If the argument is omitted, the file will be named `typescript`, except when the file already exists and is a symbolic or hard link: SCRIPT then refuses to overwrite the file, apparently for safety reasons. This check can be avoided by explicitly providing the file name on the command line.

There are several command-line switches that modify SCRIPT's behavior.

The `-a` switch will pass the `a` flag instead of `w` to `fopen()`'s `mode` parameter. If a typescript file does already exist, it will then not be overwritten; instead, the new content will be appended to the existing file.

By default, SCRIPT will launch the shell specified by the environment variable `$SHELL`. If `$SHELL` it is not set, a default shell selected at compile time (usually `/bin/sh`). The shell will be called with `-i` as its first parameter, making it an interactive shell. However, if SCRIPT is called with the `-c` option, followed by a command, it will launch the shell with `-c` and the command instead of `-i`. The shell will then be non-interactive and only run the specified command, then exit. For example, called with the parameters `-c 'last -5'`, SCRIPT will launch `/bin/sh -c 'last -5'` (or whatever shell is defined in `$SHELL`). Note that all POSIX-compatible shells have to support the `-i` and `-c` parameters.

If the `-f` switch is used, SCRIPT will call `fflush()` on the typescript file after new data has been written to it, resulting in instant updates to the typescript file, at the expense of performance. This is for example useful for letting another user watch the actions recorded by SCRIPT in real time.

If the `-q` switch is not specified, SCRIPT will display a message when it starts or quits and also record its startup and termination it the typescript file. With `-q`, all of these messages will not appear, with one exception: Since SCRIPTREPLAY will unconditionally discard the first line in a typescript file, writing the startup message (`"Script started on ..."`) cannot be disabled.

The `-t` switch will make SCRIPT output timing information to *stderr*. Its format is described in section 2.4.2.

If SCRIPT is called with `-V` or `--version` as only parameter, it will print its version and exit.

Any other parameter will make SCRIPT display an error message and exit.

2.4 File Formats

2.4.1 Typescript

The current implementation of SCRIPT uses a very simple typescript file format: Everything the client application sends to the terminal, i.e. everything printed on screen, will be written to the file, byte by byte, including control characters that are used for various tasks like setting colors, positioning the cursor etc. Additionally, a header `"Script started on XXX\n"` is written, where `XXX` is the human-readable date and time when SCRIPT was invoked. If SCRIPT was invoked without the `-q` flag, an additional footer `"Script done on YYY\n"`, where `YYY` is the human-readable date and time when SCRIPT terminated, is written.

2.4.2 Timing

Since this typescript format completely lacks timing information, the `-t` flag will output timing data to stderr. The user has to capture this output to a file by calling SCRIPT like this: `script -t 2>timingfile`.

The timing file consists of tuples of delay and byte count (space-separated), one per line:

```
0.725168 56
0.006549 126
0.040017 1
4.727988 1
0.047972 1
```

Each line can be read like *"*x *seconds after the previous output,* n *more bytes were sent to the terminal"*. If there was no previous output (because it is the first line of timing information), the delay specifies the time between SCRIPT invocation and the first chunk of output.

2.5 Disadvantages

The two file formats produced by SCRIPT, typescript and timing, show several shortcomings with regard to forensic usage:

- Input coming from the user's keyboard is not logged at all. A common example is the user entering a command in the shell but then pressing `^C` instead of return. The shell will move to the next line and display the prompt again; there is no visible distinction whether the command was run or not.[3]
- Metadata about the environment SCRIPT runs in is not logged. This leads to a high level of uncertainty when interpreting the resulting typescript, because even important information like the character set and encoding or the terminal size and type is missing.
- Typescript and timing are separate files, but one logical entity. They should reside in one file to protect the user from confusion and mistakes.
- Appending to a typescript file is possible, but ambigious, since the beginning of a new part is determined only by the string `"Script started on ..."`. Also, appending to a typescript and recording timing information are incompatible, because SCRIPTREPLAY will only ignore the first header line in a typescript file. Subsequent ones will disturb the timing's byte counter.

Summary

This section has presented the background, purpose and operation of SCRIPT. We have learned that because of several lacking features, using it in computer forensics is problematic. The next section will introduce a software without these disadvantages.

3 Design of forscript

In this section, the new file format as used by FORSCRIPT will be specified. You will learn about how input, output and metadata are combined into a single output file. After describing the format's characteristics, the invocation syntax, which is designed to be compatible to SCRIPT, will be presented.

3.1 File Format

A FORSCRIPT data file (called a *transcript file*) consists of the mostly unaltered output stream of the client application, but includes blocks of additional data (called *control chunks*) at arbitrary positions. A control chunk is started by a *shift out* byte (`0x0e`) and terminated by a *shift in* byte (`0x0f`). Each control chunk is either an input chunk or a metadata chunk.

[3] With more recent versions of Linux and Bash, terminals which have the ECHOCTL bit set (for example via stty) will show `^C` at the end of an interrupted line, which fixes this problem to some degree. Similar issues, like finding out whether the user entered or tab-completed some text, still persist.

3.1.1 Input Chunks

Input chunks contain the data that is sent to the client application's input stream, which is usually identical to the user's keyboard input. They are of arbitrary length and terminate at the *shift in* byte. If a literal *shift out* or *shift in* byte needs to appear in an input chunk's data, it is escaped by prepending a *data link escape* byte (`0x10`). If a literal *data link escape* byte needs to appear in an input chunk's data, it has to be doubled (i.e., `0x10 0x10`). For example, if the user sends the byte sequence `0x4e 0x0f 0x00 0x61 0x74 0x10`, the complete input chunk written to the transcript file is `0x0e 0x4e 0x10 0x0f 0x00 0x61 0x74 0x10 0x10 0x0f`.

3.1.2 Metadata Chunks

Metadata chunks, also called meta chunks, contain additional information about the file or the application's status, for example environment variables, terminal settings or time stamps. They contain an additional *shift out* byte at the beginning, followed by a byte that determines the type of metadata that follows. The available types are described below. Meta chunks are of arbitrary length and terminate at the *shift in* byte. The same escaping of *shift out*, *shift in* and *data link escape* that is used for input chunks is also used for meta chunks. For example, the "terminal size" meta type is introduced by its type byte `0x11`, followed by width and heigth of the terminal, represented as two unsigned big-endian 16-bit integers. The information "terminal size is 80×16 characters" would be written to the transcript file as `0x0e 0x0e 0x11 0x00 0x50 0x00 0x10 0x10 0x0f`. Note that the least significant byte of the number 16 has to be written as `0x10 0x10` to prevent the special meaning of `0x10` to escape the following `0x0f`.

3.1.3 Properties of the File Format

This basic file format design has several advantages:

- New meta chunk types can be introduced while still allowing older tools to read the file, because the escaping rules are simple and the parsing application need not know a fixed length of each type.
- Since switching between input and output data occurs very often in a usual terminal session, the format is designed to require very little storage overhead for these operations.
- The format is very compact and easy to implement. Using a format like XML would decrease performance and require sophisticated libraries on the machine FORSCRIPT is run on. However, for forensic usage it is best to be able to use a small statically linked executable.
- Converting a FORSCRIPT file to a SCRIPT file is basically as easy as removing everything between *shift out* and *shift in* bytes (while respecting escaping rules, of course).

3.2 Metadata Chunk Types

The next sections will describe the available metadata chunk types. Integers are unsigned and big endian, except where noted otherwise. In the resulting file, numbers are represented in binary form, not as ASCII digits.

For better understanding, the code FORSCRIPT uses to write each meta chunk appears after the chunk's explanation. The three functions `chunkwh()`, `chunkwf()` and `chunkwd()` that are used for actually writing the data to disk will be explained in section 4.2. To be able to understand the code, it is sufficient to know that `chunkwh()` takes one parameter (the chunk's type) and writes the header bytes. `chunkwf()` writes the footer byte and takes no parameters, while `chunkwd()` writes the payload data, escaping it on the fly, and requires a pointer and byte count. There is an additional convenience function `chunkwm()` that takes all three parameters and will write a complete metadata chunk.

All chunk functions return a negative value if an error occured, for example if an environment setting could not be retrieved or if writing to the transcript file failed. Since only a partial metadata chunk may have been written to the transcript, the file is no longer in a consistent state. Therefore, FORSCRIPT should terminate whenever a chunk function returns a negative value.

A transcript file needs to begin with a *file version* meta chunk, followed directly by the first *start of session* chunk.

`0x01` File Version (1 byte)

The transcript file must start with a meta chunk of this type; there may be no other data before it.

Denotes the version of the FORSCRIPT file format that is being used for this file. In order to guarantee a length of exactly one byte, the version numbers 0, 14, 15 and 16 are not allowed, therefore no escaping takes place. This document describes version 1 of the format, therefore currently the only valid value is `0x01`.

7 ⟨*chunks* 7⟩≡ (35a) 8a▷

```
int chunk01() {
  unsigned char ver = 0x01;
  return chunkwm(0x01, &ver, sizeof(ver));
}
```

Defines:
`chunk01`, used in chunk 30a.
Uses `chunkwm` 17a.

`0x02` Begin of Session (10 bytes)

Denotes the start of a new FORSCRIPT session. The first four data bytes represent the start time as the number of seconds since the Unix Epoch. The next four bytes contain a signed representation of the nanosecond offset to the number of seconds. If these four bytes are set to `0xffffffff`, there was an error retrieving the nanoseconds. The last two bytes specify the machine's (signed) time zone offset to UTC in minutes. If these two bytes are set to `0xffff`, the machine's timezone is unknown.

8a ⟨*chunks* 7⟩+≡ (35a) ◁7 9a▷

```
int chunk02() {
  struct timespec now;
  extern long timezone;
  int ret;
  unsigned char data[10];
  uint32_t secs;
  int32_t nanos = ~0;
  int16_t tzone = ~0;
  if ((ret = clock_gettime(CLOCK_REALTIME, &now)) < 0)
    return ret;
  secs = htonl(now.tv_sec);
  if (now.tv_nsec < 1000000000L && now.tv_nsec > -1000000000L)
    nanos = htonl(now.tv_nsec);
  tzset();
  tzone = htons((uint16_t)(timezone / -60));
  memcpy(&data[0], &secs, sizeof(secs));
  memcpy(&data[4], &nanos, sizeof(nanos));
  memcpy(&data[8], &tzone, sizeof(tzone));
  return chunkwm(0x02, data, sizeof(data));
}
```

Defines:
`chunk02`, used in chunk 30a.
Uses `chunkwm` 17a.

This chunk requires the headers `time.h` for `clock_gettime()`, `inet.h` for `htonl()` and `string.h` for `memcpy()`:

8b ⟨*includes* 8b⟩≡ (34e) 11b▷

```
#include <time.h>
#include <arpa/inet.h>
#include <string.h>
```

`0x03` End of Session (1 byte)

Denotes the end of a FORSCRIPT session. The data byte contains the return value of the child process. The usual exit code convention applies: If the child exited normally, use its return value. If the child was terminated as a result of a signal (like `SIGSEGV`), use the number of the signal plus 128.

The parameter `status` should contain the raw status value returned by `wait()`, not only the child's return value. If the exit code of the child could not be determined, `0xff` is used instead.

9a ⟨*chunks 7*⟩+≡ (35a) ◁8a 9b▷

```
int chunk03(int status) {
  unsigned char data = ~0;
  if (WIFEXITED(status))
    data = WEXITSTATUS(status);
  else if (WIFSIGNALED(status))
    data = 128 + WTERMSIG(status);
  return chunkwm(0x03, &data, sizeof(data));
}
```

Defines:
`chunk03`, used in chunk 34d.
Uses `chunkwm` 17a.

`0x11` Terminal Size (two 2-byte values)

Is written at session start and when the size of the terminal window changes. The first data word contains the number of colums, the second one the number of rows.

Since the terminal size has to be passed to the running client application, the chunk itself does not request the values, but receives them as a parameter.

9b ⟨*chunks 7*⟩+≡ (35a) ◁9a 10▷

```
int chunk11(struct winsize *size) {
  uint32_t be;
  be = htonl((size->ws_col << 16) | size->ws_row);
  return chunkwm(0x11, (unsigned char *)&be, sizeof(be));
}
```

Defines:
`chunk11`, used in chunk 25b.
Uses `chunkwm` 17a and `winsize` 25b.

0x12 Environment Variables (arbitrary number of C strings)

Is written at session start. Contains the environment variables and their values as `NAME=value` pairs, each pair is terminated by a null byte (`0x00`). Since variable names may not contain the `=` character and neither variables names nor the values may include a null byte, the list needs no special escaping.

10 ⟨*chunks 7*⟩+≡ (35a) ◁9b 11a▷

```
int chunk12() {
  extern char **environ;
  int i = 0;
  int ret;
  while (environ[i] != NULL) {
    if (i == 0) {
      if ((ret = chunkwh(0x12)) < 0)
        return ret;
    }
    if ((ret = chunkwd((unsigned char *)environ[i],
                       strlen(environ[i]) + 1)) < 0)
      return ret;
    i++;
  }
  if (i != 0) {
    if ((ret = chunkwf()) < 0)
      return ret;
  }
  return 1;
}
```

Defines:
`chunk12`, used in chunk 30a.
Uses `chunkwd` 15a, `chunkwf` 16c, and `chunkwh` 16c.

`0x13` Locale Settings (seven C strings)

Is written at session start. Contains the string values of several locale settings, namely `LC_ALL`, `LC_COLLATE`, `LC_CTYPE`, `LC_MESSAGES`, `LC_MONETARY`, `LC_NUMERIC` and `LC_TIME`, in that order, each terminated by a null byte.

11a ⟨*chunks* 7⟩+≡ (35a) ◁10 12▷

```
int chunk13() {
  int cat[7] = { LC_ALL, LC_COLLATE, LC_CTYPE, LC_MESSAGES,
                 LC_MONETARY, LC_NUMERIC, LC_TIME };
  char *loc;
  int ret;
  if ((ret = chunkwh(0x13)) < 0)
    return ret;
  for (int i = 0; i < 7; i++) {
    if ((loc = setlocale(cat[i], "")) == NULL)
      return -1;
    if ((ret = chunkwd((unsigned char *)loc,
                       strlen(loc) + 1)) < 0)
      return ret;
  }
  if ((ret = chunkwf()) < 0)
    return ret;
  return 0;
}
```

Defines:
  `chunk13`, used in chunk 30a.
Uses `chunkwd` 15a, `chunkwf` 16c, and `chunkwh` 16c.

`setlocale()` requires `locale.h`:

11b ⟨*includes* 8b⟩+≡ (34e) ◁8b 16b▷

```
#include <locale.h>
```

`0x16` Delay (two 4-byte values)

Contains the number of seconds and nanoseconds that have passed since the last delay chunk (or, if this is the first one, since the session started).

A replaying application should wait for the time specified in this chunk before advancing further in the transcript file.

Since the seconds and nanoseconds are represented as integers, converting to a floating-point number would mean a loss of precision. Therefore both integers are subtracted independently. If the nanoseconds part of `now` is less than that of `ts`, the seconds part has to be decreased by one for the result to be correct.

12 ⟨*chunks* 7⟩+≡ (35a) ◁11a

```
int chunk16(struct timespec *ts) {
  unsigned char buf[2 * sizeof(uint32_t)];
  uint32_t secs, nanos;
  struct timespec now;
  if (clock_gettime(CLOCK_MONOTONIC, &now) < 0)
    return -1;
  secs = now.tv_sec - ts->tv_sec;
  if (now.tv_nsec > ts->tv_nsec) {
    nanos = now.tv_nsec - ts->tv_nsec;
  } else {
    nanos = 1000000000L - (ts->tv_nsec - now.tv_nsec);
    secs--;
  }
  *ts = now;
  secs = htonl(secs);
  nanos = htonl(nanos);
  memcpy(&buf[0], &secs, sizeof(secs));
  memcpy(&buf[sizeof(secs)], &nanos, sizeof(nanos));
  return chunkwm(0x16, buf, sizeof(buf));
}
```

Defines:
`chunk16`, used in chunk 32a.
Uses `chunkwm` 17a.

3.3 Magic Number

Since a FORSCRIPT file has to start with a file version chunk followed by a begin of session chunk, there is a distinctive eight-byte signature at the beginning of each file:

```
0x0e 0x0e 0x01 0x?? 0x0f 0x0e 0x0e 0x02
```

The first two bytes start a metadata chunk, the third one identifies it as a file version chunk. The fourth byte contains the version number, which is currently `0x01` but may change in the future. Byte 5 closes the version chunk, 5 to 8 start a begin of session chunk.

3.4 Invocation

FORSCRIPT's invocation syntax has been designed to be compatible to SCRIPT, most parameters result in the same behavior. The following list contains additional notes and describes the differences to SCRIPT:

- `-a` (append): If the target transcript file already exists and is non-empty, it has to start with a valid and supported *file version* header.
- `-c` (command) and `-f` (flush): Identical to SCRIPT.
- `-q` (quiet): In contrast to SCRIPT, no startup message will be written to the transcript file.
- `-t` (timing): This parameter will be accepted, but ignored. FORSCRIPT always records timing information.
- `-V` and `--version`: Identical to SCRIPT, both will make FORSCRIPT output its version information and terminate. The parameter has to be the only one specified on the command line, else an error message will be printed.

If unsupported parameters are passed, FORSCRIPT will print a short usage summary to *stderr* and exit.

While running, the client application's output will be printed to *stdout*. Error messages will be printed to *stderr*.

Summary

Now you know how FORSCRIPT stores the recorded terminal session and how it will be called by the user. You have seen the code that writes the various metadata chunks. After this soft introduction to FORSCRIPT's implementation, the next section contains the rest of the code and will talk in detail about how the software works.

4 Implementation of forscript

This section will describe the code of FORSCRIPT in detail. You will learn how the software hooks into the input and output stream of the client application and how it reacts to things like window size changes or the child terminating. Other interesting topics include how to launch a subprocess and change its controlling terminal as well as how to read from multiple data streams at one without having to run separate processes.

4.1 Constants

For improved readability, we define the special characters introduced in the previous section as constants:

14a ⟨*constants* 14a⟩≡ (34f) 14b▷

```
const unsigned char SO  = 0x0e;
const unsigned char SI  = 0x0f;
const unsigned char DLE = 0x10;
```

Defines:
`DLE`, used in chunk 15a.
`SI`, used in chunks 16c and 32b.
`SO`, used in chunks 16c and 32b.

It is by design that the three special characters have consecutive byte numbers. This allows us to define a minimum and maximum byte value that requires special escape handling:

14b ⟨*constants* 14a⟩+≡ (34f) ◁14a 30d▷

```
const unsigned char ESCMIN = 0x0e;
const unsigned char ESCMAX = 0x10;
```

Defines:
`ESCMAX`, used in chunk 15a.
`ESCMIN`, used in chunk 15a.

4.2 Writing Metadata Chunks to Disk

The function *chunkwd()* takes a pointer and a byte count as arguments and writes chunk data to the transcript file, applying required escapes on the fly. To improve performance, it does not write byte-by-byte, but instead scans the input data until it finds a special character. When it does, it writes everything up to, but not including, the special character to the file and then adds a DLE character. The search then goes on. If another special character is found, everything from the last special character (inclusive) to the current one (exclusive) plus a DLE is written. Eventually the whole input data will have been scanned and the function terminates after writing everything from the last special character (inclusive) or the beginning of the data (if there were no special characters) to the end of the input data. This is the code:

15a ⟨*chunkw* 15a⟩≡ (35a)

```
int chunkwd(unsigned char *data, int count) {
  int escaped = 0;
  int pos = 0;
  int start = 0;
  while (pos < count) {
    if (data[pos] <= ESCMAX && data[pos] >= ESCMIN) {
      if (pos > start) {
        if (!swrite(&data[start], sizeof(char),
                    pos - start, OUTF))
          return -1;
      }
      if (!swrite(&DLE, sizeof(DLE), 1, OUTF))
        return -2;
      start = pos;
      escaped++;
    }
    pos++;
  }
  if (!swrite(&data[start], sizeof(char),
              pos - start, OUTF))
    return -3;
  return escaped;
}
```

Defines:
`chunkwd`, used in chunks 10, 11a, 17a, and 32b.
Uses `DLE` 14a, `ESCMAX` 14b, `ESCMIN` 14b, `OUTF` 15b, and `swrite` 16a.

OUTF is the already opened transcript file and a global variable:

15b ⟨*globals* 15b⟩≡ (34f) 18b▷

```
FILE *OUTF;
```

Defines:
`OUTF`, used in chunks 15a, 16c, 18c, 21d, 22a, 27c, and 32–34.

The *swrite()* function ("safe write") that is being used here will return zero if the number of items written is not equal to the number of items that *should* have been written:

16a ⟨*swrite* 16a⟩≡ (35a)

```
int swrite(const void *ptr, size_t size,
           size_t nmemb, FILE *stream) {
  return (fwrite(ptr, size, nmemb, stream) == nmemb);
}
```

Defines:
`swrite`, used in chunks 15a and 16c.

To be able to use `fwrite()`, `stdio.h` has to be included:

16b ⟨*includes* 8b⟩+≡ (34e) ◁11b 18a▷

```
#include <stdio.h>
```

There are functions to write chunk headers and footers:

16c ⟨*chunkwhf* 16c⟩≡ (35a)

```
int chunkwh(unsigned char id) {
  int ret;
  for (int i = 0; i < 2; i++) {
    ret = swrite(&SO, sizeof(SO), 1, OUTF);
    if (!ret)
      return -1;
  }
  return (swrite(&id, sizeof(unsigned char),
                 1, OUTF)) ? 1 : -1;
}

int chunkwf() {
  return (swrite(&SI, sizeof(SI), 1, OUTF)) ? 1 : -1;
}
```

Defines:
`chunkwf`, used in chunks 10, 11a, and 17a.
`chunkwh`, used in chunks 10, 11a, and 17a.
Uses `OUTF` 15b, `SI` 14a, `SO` 14a, and `swrite` 16a.

There is also a convenience function that writes a meta chunk's header and footer as well as the actual data:

17a ⟨*chunkwm* 17a⟩≡ (35a)

```
int chunkwm(unsigned char id, unsigned char *data, int count) {
  int ret;
  if (!chunkwh(id))
    return -11;
  if ((ret = chunkwd(data, count)) < 0)
    return ret;
  if (!chunkwf())
    return -12;
  return 1;
}
```

Defines:
`chunkwm`, used in chunks 7–9 and 12.
Uses `chunkwd` 15a, `chunkwf` 16c, and `chunkwh` 16c.

4.3 Error Handling

If the program has to terminate abnormally, the function `die()` will be called. After resetting the terminal attributes and telling a possible child process to exit, it will output an error message and exit the software.

17b ⟨*die* 17b⟩≡ (35a)

```
void die(char *message, int chunk) {
  if (TTSET)
    tcsetattr(STDERR_FILENO, TCSADRAIN, &TT);
  if (CHILD > 0)
    kill(CHILD, SIGTERM);
  fprintf(stderr, "%s: ", MYNAME);
  if (chunk != 0) {
    fprintf(stderr, "metadata chunk %02x failed", chunk);
    if (message != NULL)
      fprintf(stderr, ": ");
  } else {
    if (message == NULL)
      fprintf(stderr, "unknown error");
  }
  if (message != NULL)
    fprintf(stderr, message);
  fprintf(stderr, "; exiting.\n");
  exit(EXIT_FAILURE);
}
```

Defines:
`die`, used in chunks 18c, 21–30, 32a, and 34d.
Uses `CHILD` 27a, `MYNAME` 18b, and `TTSET` 23a.

`exit()` requires `stdlib.h`:

18a ⟨*includes* 8b⟩+≡ (34e) ◁16b 19d▷

```
#include <stdlib.h>
```

The global variable `MYNAME` contains a pointer to the name the binary was called as and is set in `main()`.

18b ⟨*globals* 15b⟩+≡ (34f) ◁15b 19c▷

```
char *MYNAME;
```

Defines:
`MYNAME`, used in chunks 17–21.

4.4 Startup and Shutdown Messages

The `statusmsg()` function writes a string to both the terminal and the transcript:

18c ⟨*statusmsg* 18c⟩≡ (35b)

```
void statusmsg(const char *msg) {
  char date[BUFSIZ];
  time_t t = time(NULL);
  struct tm *lt = localtime(&t);
  if (lt == NULL)
    die("localtime failed", 0);
  if (strftime(date, sizeof(date), "%c", lt) < 1)
    die("strftime failed", 0);
  if (printf(msg, date, OUTN) < 0) {
    perror("status stdout");
    die("statusmsg stdout failed", 0);
  }
  if (fprintf(OUTF, msg, date, OUTN) < 0) {
    perror("status transcript");
    die("statusmsg transcript failed", 0);
  }
}
```

Defines:
`statusmsg`, used in chunks 30c and 34b.
Uses `die` 17b, `OUTF` 15b, and `OUTN` 20c.

4.5 Initialization

4.5.1 Determining the Binary's Name

To be able to output its own name (e.g. in error messages), FORSCRIPT determines the name of the binary that has been called by the user. This value is stored in `argv[0]`. The global variable `MYNAME` will be used to reference that value from every function that needs it.

18d ⟨*setmyname* 18d⟩≡ (35e) 19a▷

```
MYNAME = argv[0];
```

Uses `MYNAME` 18b.

If FORSCRIPT was called using a path name (e.g. `/usr/bin/forscript`), everything up to the final slash needs to be cut off. This is done by moving the pointer to the character immediately following the final slash.

19a ⟨*setmyname* 18d⟩+≡ (35e) ◁18d

```
{ char *lastslash;
  if ((lastslash = strrchr(MYNAME, '/')) != NULL)
    MYNAME = lastslash + 1;
}
```

Uses `MYNAME` 18b.

4.5.2 Command Line Arguments

Since FORSCRIPT's invocation tries to mimic SCRIPT's as far as possible, command line argument handling is designed to closely resemble SCRIPT's behavior. Therefore, like in SCRIPT, the command line switches `--version` and `-V` are treated separately. If there is exactly one command line argument and it is one of these, FORSCRIPT will print its version and terminate.

19b ⟨*getopt* 19b⟩≡ (35e) 20b▷

```
if ((argc == 2) &&
    (!strcmp(argv[1], "-V") || !strcmp(argv[1], "--version"))) {
  printf("%s %s\n", MYNAME, MYVERSION);
  return 0;
}
```

Uses `MYNAME` 18b and `MYVERSION` 19c.

`MYVERSION` is defined as a global constant:

19c ⟨*globals* 15b⟩+≡ (34f) ◁18b 19e▷

```
const char *MYVERSION = "1.0.0";
```

Defines:
`MYVERSION`, used in chunk 19b.

The other options are parsed using the normal `getopt()` method, which requires `unistd.h`:

19d ⟨*includes* 8b⟩+≡ (34e) ◁18a 21b▷

```
#include <unistd.h>
```

`getopt()` returns the next option character each time it is called, and -1 if there are none left. The option characters are handled in a `switch` statement. As in SCRIPT, flags that turn on some behavior cause a respective global `int` variable to be increased by one. These flags are:

19e ⟨*globals* 15b⟩+≡ (34f) ◁19c 20a▷

```
int aflg = 0, fflg = 0, qflg = 0;
```

Defines:
`aflg`, used in chunks 20–22 and 30a.

The value of the `-c` parameter is stored in a global string pointer:

20a ⟨*globals* 15b⟩+≡ (34f) ◁19e 20c▷

```
char *cflg = NULL;
```

Defines:
`cflg`, used in chunks 20b and 28d.

The `-t` flag is accepted for compatibility reasons, but has no effect in FORSCRIPT because timing information is always written.

After the loop terminates, `optind` arguments have been parsed. `argc` and `argv` are then modified accordingly to only handle non-option arguments (in FORSCRIPT this is only the file name).

The parsing loop therefore looks like this:

20b ⟨*getopt* 19b⟩+≡ (35e) ◁19b

```
{ int c; extern char *optarg; extern int optind;
  while ((c = getopt(argc, argv, "ac:fqt")) != -1)
    switch ((char)c) {
    case 'a':
      aflg++; break;
    case 'c':
      cflg = optarg; break;
    case 'f':
      fflg++; break;
    case 'q':
      qflg++; break;
    case 't':
      break;
    case '?':
    default:
      fprintf(stderr,
              "usage: %s [-afqt] [-c command] [file]\n",
              MYNAME);
      exit(1);
      break;
    }
  argc -= optind;
  argv += optind;
}
```

Uses `aflg` 19e, `cflg` 20a, and `MYNAME` 18b.

After the options have been parsed, the output file name will be determined and stored in the global string `OUTN`:

20c ⟨*globals* 15b⟩+≡ (34f) ◁20a 22d▷

```
char *OUTN = "transcript";
```

Defines:
`OUTN`, used in chunks 18c and 21.

If there was no file name supplied on the command line, the default name is `transcript`. This differs from SCRIPT's default name `typescript` intentionally, because the file format is different and can, for example, not be displayed directly using CAT. If there are any scripts or constructs that assume the default output file name to be `typescript`, the chance that replacing SCRIPT with FORSCRIPT will break their functionality anyway is quite high.

4.5.3 Opening the Output File

As in SCRIPT, there is a safety warning if no file name was supplied and `transcript` exists and is a (hard or soft) link.

21a ⟨*openoutfile* 21a⟩≡ (35e) 21d▷

```
if (argc > 0) {
  OUTN = argv[0];
} else {
  struct stat s;
  if (lstat(OUTN, &s) == 0 &&
      (S_ISLNK(s.st_mode) || s.st_nlink > 1)) {
    fprintf(stderr, "Warning: ‘%s’ is a link.\n"
           "Use ‘%s [options] %s’ if you really "
           "want to use it.\n"
           "%s not started.\n",
           OUTN, MYNAME, OUTN, MYNAME);
    exit(1);
  }
}
```

Uses `MYNAME` 18b and `OUTN` 20c.

`lstat()` needs `types.h` and `stat.h` as well as `_XOPEN_SOURCE`:

21b ⟨*includes* 8b⟩+≡ (34e) ◁19d 22b▷

```
#include <sys/types.h>
#include <sys/stat.h>
```

21c ⟨*featuretest* 21c⟩≡ (34e) 22c▷

```
#define _XOPEN_SOURCE 500
```

Defines:
`_XOPEN_SOURCE`, never used.

The file will now be opened, either for writing or for appending, depending on `aflg`. Note that if appending, the file will be opened for reading as well. This is because FORSCRIPT checks the file version header before appending to a file.

21d ⟨*openoutfile* 21a⟩+≡ (35e) ◁21a 22a▷

```
if ((OUTF = fopen(OUTN, (aflg ? "a+" : "w"))) == NULL) {
  perror(OUTN);
  die("the output file could not be opened", 0);
}
```

Uses `aflg` 19e, `die` 17b, `OUTF` 15b, and `OUTN` 20c.

If the file has been opened for appending, check whether it starts with a compatible file format. Currently, the only format allowed is `0x01`. If the file is empty, appending is possible, but the *file version* chunk has to be written. This is done by setting `aflg` to 0, which will cause `doio()` to write the chunk.

22a ⟨*openoutfile* 21a⟩+≡ (35e) ◁21d

```
if (aflg) {
  char buf[5];
  size_t count;
  count = fread(&buf, sizeof(char), 5, OUTF);
  if (count == 0)
    aflg = 0;
  else if (count != 5 ||
           strncmp(buf, "\x0e\x0e\x01\x01\x0f", 5) != 0)
    die("output file is not in forscript format v1, "
        "cannot append", 0);
}
```

Uses `aflg` 19e, `die` 17b, and `OUTF` 15b.

4.6 Preparing a New Pseudo Terminal

While SCRIPT uses manual PTY allocation (by trying out device names) or BSD's `openpty()` where available, FORSCRIPT has been designed to use the Unix 98 PTY multiplexer (`/dev/ptmx`) standardized in POSIX.1-2001 to create a new PTY. This method requires `fcntl.h` and a sufficiently high feature test macro value for POSIX code.

22b ⟨*includes* 8b⟩+≡ (34e) ◁21b 23c▷

```
#include <fcntl.h>
```

22c ⟨*featuretest* 21c⟩+≡ (34e) ◁21c

```
#define _POSIX_C_SOURCE 200112L
```

Defines:
`_POSIX_C_SOURCE`, never used.

The PTY's master and slave file descriptors will be stored in these global variables:

22d ⟨*globals* 15b⟩+≡ (34f) ◁20c 23a▷

```
int PTM = 0, PTS = 0;
```

Defines:
`PTM`, used in chunks 23–25, 27c, 29d, 31, 32, and 34d.

Additionally, the settings of the terminal FORSCRIPT runs in will be saved in the global variable `TT`. This variable is used to duplicate the terminal's settings to the newly created PTY as well as to restore the terminal settings as soon as FORSCRIPT terminates. There is also a variable `TTSET` which stores whether the settings have been written to `TT`. This is important when restoring the terminal settings after a failure: If the settings have not yet been written to `TT`, applying them will lead to undefined behavior.

23a ⟨*globals* 15b⟩+≡ (34f) ◁22d 27a▷

```
struct termios TT;
int TTSET = 0;
```

Defines:
`TTSET`, used in chunks 17b and 23b.

23b ⟨*openpt* 23b⟩≡ (35e) 23d▷

```
if (tcgetattr(STDIN_FILENO, &TT) < 0) {
  perror("tcgetattr");
  die("tcgetattr failed", 0);
}
TTSET = 1;
```

Uses `die` 17b and `TTSET` 23a.

The `termios` structure is defined in `termios.h`.

23c ⟨*includes* 8b⟩+≡ (34e) ◁22b 25c▷

```
#include <termios.h>
```

A new PTY master is requested like this:

23d ⟨*openpt* 23b⟩+≡ (35e) ◁23b 23e▷

```
if ((PTM = posix_openpt(O_RDWR)) < 0) {
  perror("openpt");
  die("openpt failed", 0);
}
```

Uses `die` 17b and `PTM` 22d.

Then, access to the slave is granted.

23e ⟨*openpt* 23b⟩+≡ (35e) ◁23d 24a▷

```
if (grantpt(PTM) < 0) {
  perror("grantpt");
  die("grantpt failed", 0);
}
if (unlockpt(PTM) < 0) {
  perror("unlockpt");
  die("unlockpt failed", 0);
}
```

Uses `die` 17b and `PTM` 22d.

The slave's device file name is requested using `ptsname()`. Since the name is not needed during further execution, the slave will be opened and its file descriptor stored.

24a ⟨*openpt* 23b⟩+≡ (35e) ◁23e 24b▷

```
{ char *pts = NULL;
  if ((pts = ptsname(PTM)) != NULL) {
    if ((PTS = open(pts, O_RDWR)) < 0) {
      perror(pts);
      die("pts open failed", 0);
    }
  } else {
    perror("ptsname");
    die("ptsname failed", 0);
  }
}
```

Uses `die` 17b and `PTM` 22d.

The "parent" terminal will be configured into a "raw" mode of operation. SCRIPT does this by calling `cfmakeraw()`, which is a nonstandard BSD function. For portability reasons FORSCRIPT sets the corresponding bits manually, thereby emulating `cfmakeraw()`. The list of settings is taken from the *termios(3)* Linux man page [6] and should be equivalent. Afterwards, the settings of the terminal FORSCRIPT was started in will be copied to the new terminal. This means that in the eyes of the user the terminal's behavior will not change, but FORSCRIPT can now document the terminal's data stream with maximum accuracy.

24b ⟨*openpt* 23b⟩+≡ (35e) ◁24a 25d▷

```
{
  struct termios rtt = TT;
  rtt.c_iflag &= ~(IGNBRK | BRKINT | PARMRK | ISTRIP
                   | INLCR | IGNCR | ICRNL | IXON);
  rtt.c_oflag &= ~OPOST;
  rtt.c_lflag &= ~(ECHO | ECHONL | ICANON | ISIG | IEXTEN);
  rtt.c_cflag &= ~(CSIZE | PARENB);
  rtt.c_cflag |= CS8;
  if (tcsetattr(STDIN_FILENO, TCSANOW, &rtt) < 0) {
    perror("tcsetattr stdin");
    die("tcsetattr stdin failed", 0);
  }
  if (tcsetattr(PTS, TCSANOW, &TT) < 0) {
    perror("tcsetattr pts");
    die("tcsetattr pts failed", 0);
  }
}
```

Uses `die` 17b.

4.6.1 Managing Window Size

If the size of a terminal window changes, the controlling process receives a `SIGWINCH` signal and should act accordingly. FORSCRIPT handles this signal in the `resized()` function by writing the new size to the transcript and forwarding it to the client terminal.

25a ⟨*resized* 25a⟩≡ (35c)

```
void resized(int signal) {
  UNUSED(signal);
  winsize(3);
}
```

Defines:
`resized`, used in chunk 26a.
Uses `UNUSED` 33d and `winsize` 25b.

The actual reading and writing of the window size is done by `winsize()`, which takes a `mode` parameter. If the mode is 1, the client application's terminal size will be set. If the mode is 2, the terminal size will be written to the transcript. If the mode is 3, both operations will be done, which is the usual case.

25b ⟨*winsize* 25b⟩≡ (35c)

```
void winsize(unsigned int mode) {
  struct winsize size;
  ioctl(STDIN_FILENO, TIOCGWINSZ, &size);
  if (mode & 2)
    if (chunk11(&size) < 0)
      die("writing window size", 0x11);
  if ((mode & 1) && PTM)
    ioctl(PTM, TIOCSWINSZ, &size);
}
```

Defines:
`winsize`, used in chunks 9b, 25, and 30a.
Uses `chunk11` 9b, `die` 17b, and `PTM` 22d.

Retrieving the window size requires `ioctl.h` for `ioctl()`:

25c ⟨*includes* 8b⟩+≡ (34e) ◁23c 26b▷

```
#include <sys/ioctl.h>
```

The client PTY's window size will be initialized now. This needs to take place before the client application is launched, because it probably requires an already configured terminal size when starting up. Writing the size to the transcript however would put the window size meta chunk before the start of session chunk, therefore `winsize()`'s mode 1 is used.

25d ⟨*openpt* 23b⟩+≡ (35e) ◁24b

```
  winsize(1);
```

Uses `winsize` 25b.

4.7 Launching Subprocesses

The original SCRIPT uses one process to listen for input, one to listen for output and one to initialize and `execl()` the command to be recorded. FORSCRIPT in contrast uses only the `select()` function to be notified of pending input and output and therefore only needs two processes: Itself and the subcommand.

Registering Signal Handlers

To be notified of an exiting subprocess, a handler for the `SIGCHLD` signal needs to be defined. This signal is usually sent by the operating system if any child process's run status changes, i.e. it is stopped (`SIGSTOP`), continued (`SIGCONT`) or it exits. SCRIPT terminates if the child is stopped, but FORSCRIPT does not, because it uses the `SA_NOCLDSTOP` flag to specify that it wishes not to be notified about the child stopping or resuming. The function `finish()` handles the child's termination. The second signal handler, `resized()`, handles window size changes.

26a ⟨*sigchld* 26a⟩≡ (35e)

```
{ struct sigaction sa;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = SA_NOCLDSTOP;
  sa.sa_handler = finish;
  sigaction(SIGCHLD, &sa, NULL);
  sa.sa_handler = resized;
  sigaction(SIGWINCH, &sa, NULL);
}
```

Uses `finish` 33c and `resized` 25a.

These functions and constants require `signal.h`.

26b ⟨*includes* 8b⟩+≡ (34e) ◁25c 29b▷

```
#include <signal.h>
```

Forking

When a progam calls the `fork()` function, the operating system basically clones the program into a new process that is a subprocess of the caller. Both processes continue to run at the next command after the `fork()` call, but the value `fork()` returned will be different: The child will see a return value of `0`, while the parent will retrieve the process ID of the child. A negative value will be returned if the fork did not succeed.

26c ⟨*fork* 26c⟩≡ (35e) 27b▷

```
if ((CHILD = fork()) < 0) {
  perror("fork");
  die("fork failed", 0);
}
```

Uses `CHILD` 27a and `die` 17b.

`CHILD` is used in several places when dealing with the subprocess, therefore it is a global variable.

27a ⟨*globals* 15b⟩+≡ (34f) ◁23a

```
int CHILD = 0;
```

Defines:
`CHILD`, used in chunks 17b, 26c, 27b, 31, and 33c.

After forking, the child launches (or, to be exact, becomes) the process that should be logged, while the parent does the actual input/output logging.

27b ⟨*fork* 26c⟩+≡ (35e) ◁26c

```
if (CHILD == 0)
  doshell();
else
  doio();
```

Uses `CHILD` 27a, `doio` 29c, and `doshell` 27c.

4.8 Running the Target Application

The `doshell()` function is run in the child process, whose only task it is to set up all required PTY redirections and then execute the client command. Therefore, open file descriptors from the parent process which are no longer needed are closed early.

27c ⟨*doshell* 27c⟩≡ (35d) 27d▷

```
void doshell() {
  close(PTM);
  fclose(OUTF);
```

Defines:
`doshell`, used in chunk 27b.
Uses `OUTF` 15b and `PTM` 22d.

Changing the Terminal

Next, the child process changes its controlling terminal to be the PTY slave. In order to do that, it has to be placed in a separate session.

27d ⟨*doshell* 27c⟩+≡ (35d) ◁27c 28a▷

```
  setsid();
  if (ioctl(PTS, TIOCSCTTY, 0) < 0) {
    perror("controlling terminal");
    die("controlling terminal failed", 0);
  }
```

Uses `die` 17b.

Standard input, output and error are bound to the PTY slave, which can then be closed.

28a ⟨*doshell* 27c⟩+≡ (35d) ◁27d 28b▷

```
if ((dup2(PTS, STDIN_FILENO)  < 0) ||
    (dup2(PTS, STDOUT_FILENO) < 0) ||
    (dup2(PTS, STDERR_FILENO) < 0)) {
  perror("dup2");
  die("dup2 failed", 0);
}
close(PTS);
```

Uses `die` 17b.

Determining the Shell

If the environment variable `$SHELL` is set, its value is used. Otherwise the default is `/bin/sh`, which should exist on all Unix systems.

28b ⟨*doshell* 27c⟩+≡ (35d) ◁28a 28c▷

```
char *shell;
if ((shell = getenv("SHELL")) == NULL)
  shell = "/bin/sh";
```

Next, the name of the shell, without any path components, is determined to be used as argument zero when executing the client command.

28c ⟨*doshell* 27c⟩+≡ (35d) ◁28b 28d▷

```
char *shname;
if ((shname = strrchr(shell, '/')) == NULL)
  shname = shell;
else
  shname++;
```

Executing the Shell

Finally, the `execl()` function is used to replace the currently running FORSCRIPT process with the shell that has just been selected. If a target command has been specified using the `-c` option, it will be passed to the shell. Else, an interactive shell is launched using the `-i` option.

28d ⟨*doshell* 27c⟩+≡ (35d) ◁28c 29a▷

```
if (cflg != NULL)
  execl(shell, shname, "-c", cflg, NULL);
else
  execl(shell, shname, "-i", NULL);
```

Uses `cflg` 20a.

The FORSCRIPT child process should now have been replaced with the shell. If execution reaches code after `execl()`, an error occured and the child process will terminate with an error message.

29a ⟨*doshell* 27c⟩+≡ (35d) ◁28d

```
    perror(shell);
    die("execing the shell failed", 0);
  }
```

Uses `die` 17b.

4.9 Handling Input and Output

While SCRIPT forks twice and utilizes separate processes to handle input and output to and from the client application, FORSCRIPT uses a single process for both tasks, taking advantage of the `select()` function (defined in `select.h`) that allows it to monitor several open file descriptors at once.

29b ⟨*includes* 8b⟩+≡ (34e) ◁26b 34a▷

```
#include <sys/select.h>
```

Input and output data will never be read simultaneously. Therefore, a single data buffer is sufficient. Its size is `BUFSIZ` bytes, which is a constant defined in `stdio.h` and contains a recommended buffer size, for example 8192 bytes. The number of bytes that have been read into the buffer by `read()` will be stored in `count`.

29c ⟨*doio* 29c⟩≡ (35d) 29d▷

```
void doio() {
  char iobuf[BUFSIZ];
  int count;
```

Defines:
`doio`, used in chunk 27b.

The `select()` function is supplied with a set of file descriptors to watch, stored in the variable `fds`. It returns in `sr` the number of file descriptors that are ready, or -1 if an error occured (for example, a signal like `SIGWINCH` was received). Additionally, it requires the number of the highest-numbered file descriptor plus one as its first parameter. On all Unix systems, stdin should be file descriptor 0, but for maximum portability, FORSCRIPT compares both descriptors and stores the value to pass to `select()` in the variable `highest`.

29d ⟨*doio* 29c⟩+≡ (35d) ◁29c 29e▷

```
    fd_set fds;
    int sr;
    int highest = ((STDIN_FILENO > PTM) ?
                   STDIN_FILENO : PTM) + 1;
```

Uses `PTM` 22d.

The variable `drain` determines whether the child has already terminated, but the buffers still have to be drained.

29e ⟨*doio* 29c⟩+≡ (35d) ◁29d 30a▷

```
    int drain = 0;
```

Several metadata chunks need to be written. If the `-a` flag is not set, a *file version* chunk is written. Then *begin of session*, *environment variables* and *locale settings*. Finally `winsize()`'s mode 2 is used to only write the window size to the transcript without sending a second `SIGWINCH` to the client.

30a ⟨*doio* 29c⟩+≡ (35d) ◁29e 30b▷

```
if (!aflg)
  if (chunk01() < 0)
    die(NULL, 0x01);
if (chunk02() < 0)
  die(NULL, 0x02);
if (chunk12() < 0)
  die(NULL, 0x12);
if (chunk13() < 0)
  die(NULL, 0x13);
winsize(2);
```

Uses `aflg` 19e, `chunk01` 7, `chunk02` 8a, `chunk12` 10, `chunk13` 11a, `die` 17b, and `winsize` 25b.

To be able to calculate the delay between I/O chunks, the monotonic clock available via `clock_gettime()` is used. The following code will initialize the timer:

30b ⟨*doio* 29c⟩+≡ (35d) ◁30a 30c▷

```
struct timespec ts;
if (clock_gettime(CLOCK_MONOTONIC, &ts) < 0) {
  perror("CLOCK_MONOTONIC");
  die("retrieving monotonic time failed", 0);
}
```

Uses `die` 17b.

If the `-q` flag has not been supplied, FORSCRIPT will display a startup message similar to SCRIPT's and write the same message to the transcript file. Note that this behavior differs from SCRIPT's: When called with `-q`, SCRIPT would not output the startup message to the terminal, but record it to the typescript file nevertheless. This is required because SCRIPTREPLAY assumes that the first line in the typescript is this startup message and will unconditionally suppress its output. FORSCRIPT, however, has no such limitation and will not write the startup line to the transcript if the `-q` flag is set.

30c ⟨*doio* 29c⟩+≡ (35d) ◁30b 31a▷

```
if (!qflg)
  statusmsg(STARTMSG);
```

Uses `STARTMSG` 30d and `statusmsg` 18c.

30d ⟨*constants* 14a⟩+≡ (34f) ◁14b 34c▷

```
const char *STARTMSG = "forscript started on %s, "
                       "file is %s\r\n";
```

Defines:
`STARTMSG`, used in chunk 30c.

The main loop, which handles input and output, will run until the child process exits.

31a ⟨*doio* 29c⟩+≡ (35d) ◁30c 31b▷

```
while ((CHILD > 0) || drain) {
```

Uses CHILD 27a.

Since `select()` manipulates the value of `fds`, it has to be initialized again in each iteration. First its value is cleared, then the file descriptors for standard input and the PTY's master are added to the set, then `select()` is called to wait until one of the file descriptors has data to read available. When in drain mode, `select()` may not be called to avoid blocking.

31b ⟨*doio* 29c⟩+≡ (35d) ◁31a 31c▷

```
if (!drain) {
  FD_ZERO(&fds);
  FD_SET(STDIN_FILENO, &fds);
  FD_SET(PTM, &fds);
  sr = select(highest, &fds, NULL, NULL, NULL);
```

Uses PTM 22d.

If the child process has terminated, there may still be data left in the buffers, therefore the terminal's file descriptor is set to non-blocking mode. Reading will then continue until no more data can be retrieved. If drain mode is already active, this code will not be executed.

31c ⟨*doio* 29c⟩+≡ (35d) ◁31b 31d▷

```
if (CHILD < 0) {
  int flags = fcntl(PTM, F_GETFL);
  if (fcntl(PTM, F_SETFL, (flags | O_NONBLOCK)) == 0) {
    drain = 1;
    continue;
  }
}
```

Uses CHILD 27a and PTM 22d.

If select returns 0 or less, none of the file descriptors are ready for reading. This can for example happen if a signal was received and should be ignored. If the signal was SIGCHLD, notifying the parent thread of the child's termination, the signal handler will have set CHILD to -1 and the loop will finish after the buffers have been drained. If drain mode is already active, `select()` will not have been run, therefore this test is not needed then.

31d ⟨*doio* 29c⟩+≡ (35d) ◁31c 32a▷

```
if (sr <= 0)
  continue;
```

Execution does not reach this point if none of the file descriptors had data available. Thus it can be assumed that data will be written to the transcript file. Therefore `chunk16()` is called to calculate and write a delay meta chunk. After it has calculated the time delta, it will automatically update `ts` to contain the current time.

32a ⟨*doio* 29c⟩+≡ (35d) ◁31d 32b▷

```
if (chunk16(&ts) < 0)
  die(NULL, 0x16);
```

Uses `chunk16` 12 and `die` 17b.

If user input is available, it will be read into the buffer. The data will then be written to the transcript file, having `SO` prepended and `SI` appended. Then it will be sent to the client application. When in drain mode, user input is irrelevant since the child has already terminated.

32b ⟨*doio* 29c⟩+≡ (35d) ◁32a 32c▷

```
if (FD_ISSET(STDIN_FILENO, &fds)) {
  count = read(STDIN_FILENO, iobuf, BUFSIZ);
  if (count > 0) {
    fwrite(&SO, sizeof(SO), 1, OUTF);
    chunkwd((unsigned char *)iobuf, count);
    fwrite(&SI, sizeof(SI), 1, OUTF);
    write(PTM, iobuf, count);
  }
}
```

Uses `chunkwd` 15a, `OUTF` 15b, `PTM` 22d, `SI` 14a, and `SO` 14a.

Regardless of whether in drain mode or not, if output from the client application is available, it will be read into the buffer and written to the transcript file and standard output. If there was no data to read, the buffer has been drained, drain mode ends and the main loop will terminate.

32c ⟨*doio* 29c⟩+≡ (35d) ◁32b 33a▷

```
} // if (!drain)
if (FD_ISSET(PTM, &fds)) {
  count = read(PTM, iobuf, BUFSIZ);
  if (count > 0) {
    fwrite(iobuf, sizeof(char), count, OUTF);
    write(STDOUT_FILENO, iobuf, count);
  } else
    drain = 0;
}
```

Uses `OUTF` 15b and `PTM` 22d.

If the `-f` flag has been specified on the command line, the file should be flushed now that data has been written.

33a ⟨*doio* 29c⟩+≡ (35d) ◁32c 33b▷

```
      if (fflg)
        fflush(OUTF);
```

Uses `OUTF` 15b.

If the main loop exits, the child has terminated. `done()` is called to flush data and tidy up the environment.

33b ⟨*doio* 29c⟩+≡ (35d) ◁33a

```
    }
    done();
  }
```

Uses `done` 33e.

4.10 Finishing Execution

Since a signal handler can handle more than one signal, its number is passed as an argument. However, `finish()` only handles `SIGCHLD`, therefore it will ignore its argument. Its only task is setting `CHILD` to -1, which will cause the main loop to exit as soon as possible.

33c ⟨*finish* 33c⟩≡ (35c)

```
void finish(int signal) {
  UNUSED(signal);
  CHILD = -1;
}
```

Defines:
`finish`, used in chunk 26a.
Uses `CHILD` 27a and `UNUSED` 33d.

`UNUSED` is a macro that causes the compiler to stop warning about an unused parameter:

33d ⟨*macros* 33d⟩≡ (34e)

```
#define UNUSED(var) while (0) { (void)(var); }
```

Defines:
`UNUSED`, used in chunks 25a and 33c.

The function `done()` is called as soon as the main loop terminates. It cleans up the environment, resets the terminal and finishes execution. First, it has to fetch the exit status of the child process using `wait()`.

33e ⟨*done* 33e⟩≡ (35b) 34b▷

```
void done() {
  int status;
  wait(&status);
```

Defines:
`done`, used in chunks 33b and 34c.

To be able to use `wait()`, `wait.h` must be included.

34a ⟨*includes* 8b⟩+≡ (34e) ◁29b

```
#include <sys/wait.h>
```

If the `-q` flag has not been supplied, FORSCRIPT will write a shutdown message to both the terminal and the transcript file.

34b ⟨*done* 33e⟩+≡ (35b) ◁33e 34d▷

```
if (!qflg)
  statusmsg(STOPMSG);
```

Uses `statusmsg` 18c and `STOPMSG` 34c.

34c ⟨*constants* 14a⟩+≡ (34f) ◁30d

```
const char *STOPMSG = "forscript done on %s, "
                      "file is %s\r\n";
```

Defines:
`STOPMSG`, used in chunk 34b.
Uses `done` 33e.

Finally, it will write an *end of session* chunk, close open file descriptors, reset the terminal and exit.

34d ⟨*done* 33e⟩+≡ (35b) ◁34b

```
if (chunk03(status) < 0)
  die(NULL, 0x03);
fclose(OUTF);
close(PTM);
close(PTS);
if (tcsetattr(STDIN_FILENO, TCSADRAIN, &TT) < 0) {
  perror("tcsetattr on exit");
  die("tcsetattr on exit failed", 0);
}
exit(EXIT_SUCCESS);
}
```

Uses `chunk03` 9a, `die` 17b, `OUTF` 15b, and `PTM` 22d.

4.11 Putting It All Together

The code contained in the last sections is assembled into a single C source file, starting with feature test macros, ordinary macros and include statements.

34e ⟨*forscript.c* 34e⟩≡ 34f▷

```
⟨featuretest 21c⟩
⟨macros 33d⟩
⟨includes 8b⟩
```

Afterwards, constants and global variables are defined.

34f ⟨*forscript.c* 34e⟩+≡ ◁34e 35a▷

```
⟨constants 14a⟩
⟨globals 15b⟩
```

The functions used in the code are put in an order that makes sure every function is defined before it is called. Since `die()` is required at many places, it is put first. Next, all the chunk writing functions appear (the helper functions first).

35a ⟨*forscript.c* 34e⟩+≡ ◁34f 35b▷

⟨*die* 17b⟩
⟨*swrite* 16a⟩
⟨*chunkw* 15a⟩
⟨*chunkwhf* 16c⟩
⟨*chunkwm* 17a⟩
⟨*chunks* 7⟩

The code continues with the startup and shutdown functions.

35b ⟨*forscript.c* 34e⟩+≡ ◁35a 35c▷

⟨*statusmsg* 18c⟩
⟨*done* 33e⟩

Next, the signal handlers.

35c ⟨*forscript.c* 34e⟩+≡ ◁35b 35d▷

⟨*finish* 33c⟩
⟨*winsize* 25b⟩
⟨*resized* 25a⟩

The two functions that represent the parent and child processes are defined next.

35d ⟨*forscript.c* 34e⟩+≡ ◁35c 35e▷

⟨*doshell* 27c⟩
⟨*doio* 29c⟩

Finally, the `main()` function decides the order in which the steps described in this chapter are executed. Since neither the parent nor the child process should ever reach the end of `main()`, it returns `EXIT_FAILURE`.

35e ⟨*forscript.c* 34e⟩+≡ ◁35d

```
int main(int argc, char *argv[]) {
  ⟨setmyname 18d⟩
  ⟨getopt 19b⟩
  ⟨openoutfile 21a⟩
  ⟨openpt 23b⟩
  ⟨sigchld 26a⟩
  ⟨fork 26c⟩
  return EXIT_FAILURE;
}
```

Defines:
`main`, never used.

5 Evaluation

In order to show you what the code you have just seen actually does, this section contains instructions on how to compile it, and it features an example transcript file analyzed in detail.

5.1 Compiling forscript

FORSCRIPT is written conforming to the C99 and POSIX-1.2001 standards, with portability in mind. It has been developed on a machine running Linux [7] 2.6.32, using glibc [8] 2.10 and GCC [9] 4.4.3. The following command line is an example of how to compile FORSCRIPT:

```
gcc -std=c99 -Wl,-lrt -g -o forscript -Wall \
    -Wextra -pedantic -fstack-protector-all -pipe forscript.c
```

To generate `forscript.c` out of the *noweb* source code, the following command line can be used:

```
notangle -Rforscript.c thesis.nw > forscript.c
```

On the author's machine, FORSCRIPT can be compiled without any compiler warnings. It has also been successfully compiled on NetBSD.

Since Apple Mac OS X in its current version 10.6.2 lacks support for the real-time extension of POSIX, the `clock_gettime()` function required by FORSCRIPT is not natively available. Therefore the code described in this thesis can in its current state not be compiled on OS X. However, it should be possible to create a function emulating `clock_gettime()` and then port FORSCRIPT to OS X.

5.2 Example Transcript File

To demonstrate FORSCRIPT's output, the following pages contain a commented *hex dump* of a transcript file created on the author's machine. The dump has been created using `hexdump -C transcript`. Since metadata chunks do not necessarily start or end at a 16-byte border, the dump has been cut into distinct pieces, bytes not belonging to the current logical unit being replaced by whitespace. The hex dump consists of several three-colum lines. The first two columns contain 16 bytes of data represented in hexadecimal form, eight bytes each. The third column represents these 16 bytes interpreted as ASCII characters, nonprintable characters are replaced with a single dot.

The transcript starts with a *file version* chunk, specifying that version 1 is used:

```
0e 0e 01 01 0f                                          |.....           |
```

Then a *start of session* chunk follows.

```
               0e 0e 02  4b 82 d0 f3 04 4d 8b e3  |     ...K....M..|
00 3c 0f                                          |.<.             |
```

Its first eight bytes, (`4b` to `e3`) tell you that the time is 1266864371.072190947 seconds after the epoch, which is February 22, 2010, 18:46:11 UTC. The next two bytes, `00 3c` represent a timezone of 60 which translates to UTC+01:00.

After this chunk, the environment variables are listed. These are `name=value` pairs, separated by null bytes. This information is important to interpret the actual

terminal data: For example, different control codes are used depending on the `TERM` variable's setting.

```
         0e 0e 12 53 53  48 5f 41 47 45 4e 54 5f  |   ...SSH_AGENT_|
50 49 44 3d 31 36 33 30  00 47 50 47 5f 41 47 45  |PID=1630.GPG_AGE|
4e 54 5f 49 4e 46 4f 3d  2f 74 6d 70 2f 67 70 67  |NT_INFO=/tmp/gpg|
2d 4b 50 62 79 65 43 2f  53 2e 67 70 67 2d 61 67  |-KPbyeC/S.gpg-ag|
65 6e 74 3a 31 36 33 31  3a 31 00 54 45 52 4d 3d  |ent:1631:1.TERM=|
72 78 76 74 00 53 48 45  4c 4c 3d 2f 62 69 6e 2f  |rxvt.SHELL=/bin/|
62 61 73 68 00 57 49 4e  44 4f 57 49 44 3d 32 37  |bash.WINDOWID=27|
32 36 32 39 38 34 00 55  53 45 52 3d 73 63 79 00  |262984.USER=scy.|
53 53 48 5f 41 55 54 48  5f 53 4f 43 4b 3d 2f 74  |SSH_AUTH_SOCK=/t|
6d 70 2f 73 73 68 2d 64  63 74 77 4b 42 31 36 30  |mp/ssh-dctwKB160|
37 2f 61 67 65 6e 74 2e  31 36 30 37 00 50 41 54  |7/agent.1607.PAT|
48 3d 2f 68 6f 6d 65 2f  73 63 79 2f 62 69 6e 3a  |H=/home/scy/bin:|
2f 75 73 72 2f 6c 6f 63  61 6c 2f 62 69 6e 3a 2f  |/usr/local/bin:/|
75 73 72 2f 62 69 6e 3a  2f 62 69 6e 3a 2f 75 73  |usr/bin:/bin:/us|
72 2f 67 61 6d 65 73 00  50 57 44 3d 2f 68 6f 6d  |r/games.PWD=/hom|
65 2f 73 63 79 00 4c 41  4e 47 3d 65 6e 5f 55 53  |e/scy.LANG=en_US|
2e 55 54 46 2d 38 00 43  4f 4c 4f 52 46 47 42 47  |.UTF-8.COLORFGBG|
3d 37 3b 64 65 66 61 75  6c 74 3b 30 00 48 4f 4d  |=7;default;0.HOM|
45 3d 2f 68 6f 6d 65 2f  73 63 79 00 53 48 4c 56  |E=/home/scy.SHLV|
4c 3d 32 00 4c 4f 47 4e  41 4d 45 3d 73 63 79 00  |L=2.LOGNAME=scy.|
57 49 4e 44 4f 57 50 41  54 48 3d 37 00 44 49 53  |WINDOWPATH=7.DIS|
50 4c 41 59 3d 3a 30 2e  30 00 43 4f 4c 4f 52 54  |PLAY=:0.0.COLORT|
45 52 4d 3d 72 78 76 74  2d 78 70 6d 00 5f 3d 75  |ERM=rxvt-xpm._=u|
6e 69 2f 62 61 63 68 65  6c 6f 72 2f 66 6f 72 73  |ni/bachelor/fors|
63 72 69 70 74 00 0f                              |cript..         |
```

The next chunk contains the locale settings the C library uses for messages, number and currency formatting and other things. Although the user may choose different locales for either category, they are usually all the same. This example makes no difference: The system is configured for US English and a character encoding of UTF-8.

```
                     0e  0e 13 65 6e 5f 55 53 2e  |       ...en_US.|
55 54 46 2d 38 00 65 6e  5f 55 53 2e 55 54 46 2d  |UTF-8.en_US.UTF-|
38 00 65 6e 5f 55 53 2e  55 54 46 2d 38 00 65 6e  |8.en_US.UTF-8.en|
5f 55 53 2e 55 54 46 2d  38 00 65 6e 5f 55 53 2e  |_US.UTF-8.en_US.|
55 54 46 2d 38 00 65 6e  5f 55 53 2e 55 54 46 2d  |UTF-8.en_US.UTF-|
38 00 65 6e 5f 55 53 2e  55 54 46 2d 38 00 0f     |8.en_US.UTF-8.. |
```

The terminal FORSCRIPT is running in is 168 characters wide (`00 a8`) and 55 characters high (`00 37`), as the *terminal size* chunk shows:

```
                                                 0e  |               .|
0e 11 00 a8 00 37 0f                                |.....7.         |
```

After all these metadata chunks, this is where actual terminal output starts. Since the `-q` flag was not used, FORSCRIPT writes a startup message both to the terminal and the transcript, containing date and time and the file name. The final two bytes `0d 0a` represent the control codes *carriage return* and *line feed*. Note that in contrast to the Unix convention of using just *line feed* (`\n`) to designate "new line" in text

files, a terminal (or at least the terminal the author's machine is using) requires both bytes to be present.

```
                      66  6f 72 73 63 72 69 70 74  |       forscript|
20 73 74 61 72 74 65 64  20 6f 6e 20 4d 6f 6e 20  | started on Mon |
32 32 20 46 65 62 20 32  30 31 30 20 30 37 3a 34  |22 Feb 2010 07:4|
36 3a 31 31 20 50 4d 20  43 45 54 2c 20 66 69 6c  |6:11 PM CET, fil|
65 20 69 73 20 74 72 61  6e 73 63 72 69 70 74 0d  |e is transcript.|
0a                                                |.               |
```

Now the shell is started. It requires some time to read its configuration files and initialize the environment, therefore FORSCRIPT has to wait for it and starts measuring the time until the next piece of data arrives. After the shell has initialized, it prints out its *prompt*. On this machine, the prompt (`scy@bijaz ~ master ? 0.11 19:46 $`) is a rather complicated, colored one and therefore contains lots of ISO 6429 control codes (also known as "ANSI escape codes") to define the visual appearance.

However, before the prompt is written to the data file, FORSCRIPT writes a *delay* meta chunk: It took 0.065087679 seconds before the prompt was printed.

```
   0e 0e 16 00 00 00 00  03 e1 28 bf 0f 1b 5d 30  | .........(...]0|
3b 73 63 79 40 62 69 6a  61 7a 3a 7e 07 1b 5b 31  |;scy@bijaz:~..[1|
3b 33 32 6d 73 63 79 1b  5b 30 3b 33 32 6d 40 1b  |;32mscy.[0;32m@.|
5b 31 3b 33 32 6d 62 69  6a 61 7a 1b 5b 31 3b 33  |[1;32mbijaz.[1;3|
34 6d 20 7e 20 1b 5b 30  3b 33 36 6d 6d 61 73 74  |4m ~ .[0;36mmast|
65 72 20 3f 20 1b 5b 31  3b 33 30 6d 30 2e 31 31  |er ? .[1;30m0.11|
20 1b 5b 30 3b 33 37 6d  31 39 3a 34 36 20 1b 5b  | .[0;37m19:46 .[|
30 3b 33 33 6d 1b 5b 31  3b 33 32 6d 24 1b 5b 30  |0;33m.[1;32m$.[0|
6d 20                                             |m               |
```

Next, 1.291995750 seconds after the prompt has been printed, the user types the letter `e` on the keyboard. The letter is enclosed by `0e` and `0f` in order to mark it as input data.

```
      0e 0e 16 00 00 00  01 11 67 80 66 0f 0e 65  |m ........g.f..e|
0f                                                |.               |
```

After the letter has been typed, the kernel will usually *echo* the character, that is, put it into the terminal's output stream to make it appear on screen. It will take a small amount of time (in this case 0.0079911 seconds) until FORSCRIPT receives the character and write it to the transcript file, this time declaring it as output.

```
   0e 0e 16 00 00 00 00  00 79 ef 3c 0f 65        | ........y.<.e  |
```

The user now continues to type the characters `echo -l`, which will be echoed as well.

```
                                                  0e 0e  |              ..|
16 00 00 00 00 05 b9 48  10 10 0f 0e 63 0f 0e 0e  |.......H....c...|
16 00 00 00 00 00 79 a5  09 0f 63 0e 0e 16 00 00  |......y...c.....|
00 00 0a 7d bf 1e 0f 0e  68 0f 0e 0e 16 00 00 00  |...}....h.......|
00 00 79 db 51 0f 68 0e  0e 16 00 00 00 00 0b 71  |..y.Q.h........q|
c4 94 0f 0e 6f 0f 0e 0e  16 00 00 00 00 00 79 fc  |....o.........y.|
54 0f 6f 0e 0e 16 00 00  00 02 09 89 aa a1 0f 0e  |T.o.............|
```

```
20 0f 0e 0e 16 00 00 00  00 00 79 f2 83 0f 20 0e  | .........y... .|
0e 16 00 00 00 01 2f 35  2a bc 0f 0e 2d 0f 0e 0e  |....../5*...-...|
16 00 00 00 00 00 79 bb  20 0f 2d 0e 0e 16 00 00  |......y. .-.....|
00 00 14 fb 28 4d 0f 0e  6c 0f 0e 0e 16 00 00 00  |....(M..l.......|
00 00 7a 01 3d 0f 6c 0e  0e 16 00 00 00 00 2b 64  |..z.=.l.......+d|
b7 45 0f                                          |.E.              |
```

Since typing the `l` was a mistake, the user presses the "backspace" key (ASCII value 127) to remove the last character.

```
         0e 7f 0f                                 |   ...           |
```

After the usual delay, the shell will send two things to the terminal: First, an ASCII backspace character (`08`) to position the cursor on the `l`, then the ANSI code *CSI K*, represented by the bytes `1b 5b 4b`, which will cause the terminal to make all characters at or right of the cursor's position disappear.

```
                  0e 0e  16 00 00 00 00 00 79 c2  |      ........y.|
7e 0f 08 1b 5b 4b                                 |~...[K           |
```

The user now enters the letter `n` and hits the return key (represented as ASCII byte `0d`) in order to execute the command `echo -n`. After executing the command (which produces no output), the shell displays the prompt again.

```
                  0e 0e  16 00 00 00 00 37 50 74  |      .......7Pt|
a3 0f 0e 6e 0f 0e 0e 16  00 00 00 00 00 79 c4 67  |...n.........y.g|
0f 6e 0e 0e 16 00 00 00  00 2e bb 20 01 0f 0e 0d  |.n......... ....|
0f 0e 0e 16 00 00 00 00  00 79 f9 df 0f 0d 0a 0e  |.........y......|
0e 16 00 00 00 00 02 25  be d3 0f 1b 5d 30 3b 73  |.......%....]0;s|
63 79 40 62 69 6a 61 7a  3a 7e 07 1b 5b 31 3b 33  |cy@bijaz:~..[1;3|
32 6d 73 63 79 1b 5b 30  3b 33 32 6d 40 1b 5b 31  |2mscy.[0;32m@.[1|
3b 33 32 6d 62 69 6a 61  7a 1b 5b 31 3b 33 34 6d  |;32mbijaz.[1;34m|
20 7e 20 1b 5b 30 3b 33  36 6d 6d 61 73 74 65 72  | ~ .[0;36mmaster|
20 3f 20 1b 5b 31 3b 33  30 6d 30 2e 31 30 20 1b  | ? .[1;30m0.10 .|
5b 30 3b 33 37 6d 31 39  3a 34 36 20 1b 5b 30 3b  |[0;37m19:46 .[0;|
33 33 6d 1b 5b 31 3b 33  32 6d 24 1b 5b 30 6d 20  |33m.[1;32m$.[0m |
```

Note that without recording the user's input, it would be impossible to determine whether the user pressed return to actually run the command or whether entering the command was cancelled, for example by pressing `^C`.

1.587984366 seconds later, the user decides to end the current session by pressing `^D`, which is equivalent to the byte value `04`.

```
0e 0e 16 00 00 00 01 23  0b ed ee 0f 0e 04 0f     |.......#....... |
```

The shell reacts by printing `exit` and terminating. Then, FORSCRIPT prints its shutdown message.

```
                                                 65  |              e|
78 69 74 0d 0a 66 6f 72  73 63 72 69 70 74 20 64  |xit..forscript d|
6f 6e 65 20 6f 6e 20 4d  6f 6e 20 32 32 20 46 65  |one on Mon 22 Fe|
62 20 32 30 31 30 20 30  37 3a 34 36 3a 32 31 20  |b 2010 07:46:21 |
50 4d 20 43 45 54 2c 20  66 69 6c 65 20 69 73 20  |PM CET, file is |
74 72 61 6e 73 63 72 69  70 74 0d 0a              |transcript..    |
```

Finally, the exit status (0) of the shell is recorded in an *end of session* metadata chunk and the transcript file ends.

```
                                        0e 0e 03 00  |            ....|
0f                                                   |.|
```

6 Summary

In this thesis it has been presented why SCRIPT, although often used for forensic investigations, lacks features that are crucial for reliable documentation. A new software, FORSCRIPT, has been designed and implemented, the weaknesses of SCRIPT have been eliminated.

6.1 Future Tasks

The primary reason to develop FORSCRIPT was the need to create a software that enables a forensic investigator to convert an interactive command-line session into a version suitable for inclusion in a printed report. While thinking about possible approaches, it became apparent that the output generated by SCRIPT does not suffice to provide such a software with the information it needs to unambigously reconstruct what the user did. A tool that records the required information had to be developed first. This task has been solved in this bachelor thesis. Next, a tool that is able to parse the output FORSCRIPT generates is to be written.

FORSCRIPT will be released by the author as free software, available at [10]. Corrections and improvements are encouraged: FORSCRIPT is far from being perfect and it is quite possible that during the development of additional tools, bugs and shortcomings will need to be fixed.

Additionally, we will approach the maintainers of SCRIPT and the forensic community as they can probably benefit from FORSCRIPT's existence.

References

[1] Casey, Eoghan: *Digital Evidence and Computer Crime* (2nd edition, 2004), Academic Press, ISBN 978-0121631048.

[2] Knuth, Donald E.: *Literate Programming* (1992), Center for the Study of Language and Information, ISBN 978-0937073803.

[3] *Noweb — A simple, Extensible Tool for Literate Programming*, created and maintained by Norman Ramsey, current release 2.11b.
`http://www.cs.tufts.edu/~nr/noweb/`

[4] *The* util-linux *project*, no longer maintained, last release 2.13-pre7 in 2006.
`http://www.kernel.org/pub/linux/utils/util-linux/`

[5] *The* util-linux-ng *project*, maintained by Karel Zak, current release 2.17.
`http://userweb.kernel.org/~kzak/util-linux-ng/`

[6] *The Linux* man-pages *project*, maintained by Michael Kerrisk, release 3.23.
`http://www.kernel.org/doc/man-pages/`

[7] *The Linux kernel*, maintained by Linus Torvalds, release 2.6.31.
`http://www.kernel.org/`

[8] *GNU C Library*, maintained by Ulrich Drepper, release 2.10.
`http://www.gnu.org/software/libc/`

[9] *GNU Compiler Collection*, maintained by its steering committee, release 4.4.3.
`http://gcc.gnu.org/`

[10] *forscript*, created and maintained by Tim Weber, release 1.0.0.
`http://scytale.name/proj/forscript/`