

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

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

1	Introduction	1
1.1	Background: Computer forensics	1
1.2	Tasks	2
1.3	Results	2
1.4	Outlook on the thesis	2
2	script	4
2.1	Invocation	4
2.2	File formats	5
2.2.1	Typescript	5
2.2.2	Timing	5
2.3	Disadvantages	5
3	Design of forscript	7
3.1	File format	7
3.1.1	Input chunks	7
3.1.2	Metadata chunks	7
3.1.3	Properties of the file format	7
3.2	Metadata chunk types	8
3.3	Magic number	13
3.4	Invocation	14
4	Implementation of forscript	15
4.0.1	How a FORSCRIPT file is written	16
4.1	Initialization	20
4.1.1	Determining the binary name	20
4.1.2	Command line arguments	20
4.1.3	Opening the output file	23
4.2	Preparing a new pseudo terminal	24
4.2.1	Managing window size	26
4.3	Launching subprocesses	28
4.3.1	Registering signal handlers	28
4.3.2	Forking	28
4.4	Running the target application	29
4.5	Handling input and output	30
4.6	Finishing execution	34
5	Evaluation	38
6	Summary	39

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum. Sed ut perspiciatis unde omnis iste natus error sit voluptatem accusantium doloremque laudantium, totam rem aperiam, eaque ipsa quae ab illo inventore veritatis et quasi architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia consequuntur magni dolores eos qui ratione voluptatem sequi nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit amet, consectetur, adipisci velit, sed quia non numquam eius modi tempora incidunt ut labore et dolore magnam aliquam quaerat voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi consequatur? Quis autem vel eum iure reprehenderit qui in ea voluptate velit esse quam nihil molestiae consequatur, vel illum qui dolorem eum fugiat quo voluptas nulla pariatur?

1 Introduction

1.1 Background: Computer forensics

Computer forensics is a branch of forensic science. 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 introducing 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 and videos or 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 then 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 (also called "printing", even though the output appears on the screen, not on paper) text and data. A text-based interface cannot display graphics or use the mouse¹.

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 2.3 which greatly limit its usefulness as a forensic tool.

¹Using 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 [1] 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.

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.

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 [2] was abandoned in 2006. Today, it has been replaced by its successor *util-linux-ng* [3], 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 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 `$SHELL`, or, if it is not set, a default shell selected at compile time (usually `/bin/sh`). The shell will be called with `-i` as its first parameter, making it an interactive shell. However, if SCRIPT is called with the `-c` option, followed by a command, it will launch the shell with `-c` and the command instead of `-i`. The shell will then be non-interactive and only run the specified command, then exit. For example, calling SCRIPT with the parameters `-c 'last -5'`, the command SCRIPT will launch is `/bin/sh -c 'last -5'` (of course depending on `$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 to let 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 and quits and will also record its startup and termination in 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 2.2.2.

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.

2.2 File formats

2.2.1 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.

2.2.2 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.

2.3 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.²
- 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.

²With 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.

- Appending to a typescript file is possible, but ambiguous, 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.

3 Design of forscript

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.

3.1.1 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 that will be 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 height 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 4.0.1. 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 occurred, 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.

```
8 <chunks 8>≡ (36) 9a▷
   int chunk01() {
       unsigned char ver = 0x01;
       return chunkwm(0x01, &ver, sizeof(ver));
   }
```

Defines:

`chunk01`, never used.

Uses `chunkwm` 18a.

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.

9a $\langle \text{chunks } 8 \rangle + \equiv$ (36) $\langle 8 \text{ } 10a \rangle$

```
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 < 2000000000L && now.tv_nsec > -2000000000L)
        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, never used.

Uses chunkwm 18a.

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

9b $\langle \text{includes } 9b \rangle \equiv$ (36) $10c \rangle$

```
#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.

10a `<chunks 8>+≡` (36) `<9a 10b>`

```
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 35e.

Uses `chunkwm` 18a.

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 columns, 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.

10b `<chunks 8>+≡` (36) `<10a 11>`

```
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 27a.

Uses `chunkwm` 18a and `winsize` 27a.

This chunk requires `ioctl.h` for `ioctl()`:

10c `<includes 9b>+≡` (36) `<9b 12b>`

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

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).

11 \langle chunks 8 \rangle + \equiv (36) \langle 10b 12a \rangle

```
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`, never used.

Uses `chunkwd` 16a, `chunkwf` 17c, and `chunkwh` 17c.

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.

```
12a <chunks 8>+≡ (36) <11 13>
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`, never used.

Uses `chunkwd` 16a, `chunkwf` 17c, and `chunkwh` 17c.

This chunk requires *locale.h*:

```
12b <includes 9b>+≡ (36) <10c 17b>
#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 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.

```
13  <chunks 8>+≡ (36) <12a 19b>
    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 33c.

Uses `chunkwm` 18a.

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

4 Implementation of forscript

For improved readability, we define the special characters as constants:

```
15a  <constants 15a>≡ (36) 15b>
      const unsigned char S0 = 0x0e;
      const unsigned char SI = 0x0f;
      const unsigned char DLE = 0x10;
```

Defines:

DLE, used in chunk 16a.

SI, used in chunks 17c and 33d.

S0, used in chunks 17c and 33d.

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:

```
15b  <constants 15a>+≡ (36) <15a 32b>
      const unsigned char ESCMIN = 0x0e;
      const unsigned char ESCMAX = 0x10;
```

Defines:

ESCMAX, used in chunk 16a.

ESCMIN, used in chunk 16a.

4.0.1 How a forscript file is written

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-per-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:

```
16a <chunkw 16a>≡ (36)
    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 11, 12a, 18a, and 33d.

Uses `DLE` 15a, `ESCMAX` 15b, `ESCMIN` 15b, `OUTF` 16b, and `swrite` 17a.

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

```
16b <globals 16b>≡ (36) 19a▷
    FILE *OUTF;
```

Defines:

`OUTF`, used in chunks 16a, 17c, 19c, 23d, 24a, 29c, and 33–35.

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:

```
17a  <swrite 17a>≡ (36)
      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 16a and 17c.

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

```
17b  <includes 9b>+≡ (36) <12b 18c>
      #include <stdio.h>
```

There are functions to write chunk headers and footers:

```
17c  <chunkwhf 17c>≡ (36)
      int chunkwh(unsigned char id) {
          int ret;
          for (int i = 0; i < 2; i++) {
              ret = swrite(&SO, sizeof(SO), 1, UTF);
              if (!ret)
                  return -1;
          }
          return (swrite(&id, sizeof(unsigned char), 1, UTF)) ? 1 : -1;
      }
```

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

Defines:

chunkwf, used in chunks 11, 12a, and 18a.

chunkwh, used in chunks 11, 12a, and 18a.

Uses UTF 16b, SI 15a, SO 15a, and **swrite** 17a.

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

```
18a <chunkwm 18a>≡ (36)
    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 8–10 and 13.

Uses `chunkwd` 16a, `chunkwf` 17c, and `chunkwh` 17c.

If the program has to terminate abnormally, the function `die()` will be called. It will output an error message and exit the software.

```
18b <die 18b>≡ (36)
    void die(char *message, int chunk) {
        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 19, 23–25, 27a, 28c, 30c, and 31d.

Uses `chunk` 19b and `MYNAME` 19a.

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

```
18c <includes 9b>+≡ (36) <17b 20e>
    #include <stdlib.h>
```

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

```
19a <globals 16b>+≡ (36) <16b 20d>  
    char *MYNAME;
```

Defines:

MYNAME, used in chunks 18b, 20, 22a, and 23a.

For convenience, we define the following macro that will call `die()` and supply it with the chunk's type.

```
19b <chunks 8>+≡ (36) <13  
    #define chunk(num) if (chunk##num() < 0) die(NULL, 0x##num);
```

Defines:

chunk, used in chunks 18b and 31c.

Uses `die` 18b.

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

```
19c <statusmsg 19c>≡ (36)  
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 32a and 35c.

Uses `die` 18b, `OUTF` 16b, and `OUTN` 22b.

4.1 Initialization

4.1.1 Determining the binary 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.

```
20a <setmyname 20a>≡ (36) 20b>
    MYNAME = argv[0];
```

Uses `MYNAME` 19a.

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.

```
20b <setmyname 20a>+≡ (36) <20a
    { char *lastslash;
      if ((lastslash = strrchr(MYNAME, '/')) != NULL)
          MYNAME = lastslash + 1;
    }
```

Uses `MYNAME` 19a.

4.1.2 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 as far as possible. 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.

```
20c <getopt 20c>≡ (36) 22a>
    if ((argc == 2) &&
        (!strcmp(argv[1], "-V") || !strcmp(argv[1], "--version"))) {
        printf("%s %s\n", MYNAME, MYVERSION);
        return 0;
    }
```

Uses `MYNAME` 19a and `MYVERSION` 20d.

`MYVERSION` is defined as a global constant:

```
20d <globals 16b>+≡ (36) <19a 21a>
    const char *MYVERSION = "0.9-git";
```

Defines:

`MYVERSION`, used in chunk 20c.

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

```
20e <includes 9b>+≡ (36) <18c 23b>
    #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:

21a `<globals 16b>+≡` (36) `<20d 21b>`

```
int aflag = 0, fflag = 0, qflag = 0;
```

Defines:

`aflag`, used in chunks 22–24 and 31c.

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

21b `<globals 16b>+≡` (36) `<21a 22b>`

```
char *cflag = NULL;
```

Defines:

`cflag`, used in chunks 22a and 30b.

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:

```
22a <getopt 20c>+≡ (36) <20c
{ int c; extern char *optarg; extern int optind;
  while ((c = getopt(argc, argv, "ac:fqt")) != -1)
    switch ((char)c) {
      case 'a':
        aflag++; break;
      case 'c':
        cflag = optarg; break;
      case 'f':
        fflag++; break;
      case 'q':
        qflag++; break;
      case 't':
        break;
      case '?':
      default:
        fprintf(stderr,
                "usage: %s [-afqt] [-c command] [file]\n",
                MYNAME);
        exit(1);
        break;
    }
  argc -= optind;
  argv += optind;
}
```

Uses `aflag` 21a, `cflag` 21b, and `MYNAME` 19a.

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

```
22b <globals 16b>+≡ (36) <21b 24d>
char *OUTN = "transcript";
```

Defines:

`OUTN`, used in chunks 19c and 23.

If there was no 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.

4.1.3 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.

```
23a <openoutfile 23a>≡ (36) 23d>
    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 19a and OUTN 22b.

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

```
23b <includes 9b>+≡ (36) <20e 24b>
    #include <sys/types.h>
    #include <sys/stat.h>
```

```
23c <featuretest 23c>≡ (36) 24c>
    #define _XOPEN_SOURCE 500
```

Defines:

`_XOPEN_SOURCE`, never used.

The file will now be opened, either for writing or for appending, depending on `aflg`.

```
23d <openoutfile 23a>+≡ (36) <23a 24a>
    if ((OUTF = fopen(OUTN, (aflg ? "a+" : "w"))) == NULL) {
        perror(OUTN);
        die("the output file could not be opened", 0);
    }
}
```

Uses `aflg` 21a, `die` 18b, `OUTF` 16b, and `OUTN` 22b.

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.

```
24a <openoutfile 23a>+≡ (36) <23d
    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` 21a, `die` 18b, and `OUTF` 16b.

4.2 Preparing a new pseudo terminal

While `SCRIPT` uses manual PTY allocation (by trying out device names) or BSD's `openpty()` where available, `FORSCRIPT` uses the 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.

```
24b <includes 9b>+≡ (36) <23b 25a>
    #include <fcntl.h>
```

```
24c <featuretest 23c>+≡ (36) <23c
    #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:

```
24d <globals 16b>+≡ (36) <22b 24e>
    int PTM = 0, PTS = 0;
```

Defines:

`PTM`, used in chunks 25, 27a, 29c, and 31–35.

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.

```
24e <globals 16b>+≡ (36) <24d 29a>
    struct termios TT;
```

```
24f <openpt 24f>≡ (36) 25b>
    tcgetattr(0, &TT);
```

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

```
25a <includes 9b>+≡ (36) <24b 28b>
    #include <termios.h>
```

A new PTY master is requested like this:

```
25b <openpt 24f>+≡ (36) <24f 25c>
    if ((PTM = posix_openpt(O_RDWR)) < 0) {
        perror("openpt");
        die("openpt failed", 0);
    }
```

Uses `die` 18b and `PTM` 24d.

Then, access to the slave is granted.

```
25c <openpt 24f>+≡ (36) <25b 25d>
    if (grantpt(PTM) < 0) {
        perror("grantpt");
        die("grantpt failed", 0);
    }
    if (unlockpt(PTM) < 0) {
        perror("unlockpt");
        die("unlockpt failed", 0);
    }
```

Uses `die` 18b and `PTM` 24d.

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.

```
25d <openpt 24f>+≡ (36) <25c 26a>
    { 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` 18b and `PTM` 24d.

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 [4] 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.

```
26a  <openpt 24f>+≡ (36) <25d 27b>
    {
        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;
        tcsetattr(0, TCSANOW, &rtt);
        tcsetattr(PTS, TCSANOW, &TT);
    }
```

4.2.1 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.

```
26b  <resized 26b>≡ (36)
    void resized(int signal) {
        UNUSED(signal);
        winsize(3);
    }
```

Defines:

`resized`, used in chunk 28a.

Uses `UNUSED` 35a and `winsize` 27a.

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.

27a $\langle winsize\ 27a \rangle \equiv$ (36)

```
void winsize(unsigned int mode) {
    struct winsize size;
    ioctl(0, 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 10b, 26b, 27b, and 31c.

Uses `chunk11` 10b, `die` 18b, and `PTM` 24d.

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.

27b $\langle openpt\ 24f \rangle + \equiv$ (36) $\triangleleft 26a$

```
winsize(1);
```

Uses `winsize` 27a.

4.3 Launching subprocesses

The original SCRIPT uses one process to listen for input, one to listen for output and one to initialize and `exec1()` 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.

4.3.1 Registering signal handlers

To be notified of an exiting subprocess, a handler for the SIGCHLD signal needs to be defined. This signal is 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. The function `finish()` handles the child's termination. The second signal handler, `resized()`, handles window size changes.

```
28a <sigchld 28a>≡ (36)
    { 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` 34d and `resized` 26b.

These functions and constants require `signal.h`.

```
28b <includes 9b>+≡ (36) <25a 30d>
    #include <signal.h>
```

4.3.2 Forking

When a program 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.

```
28c <fork 28c>≡ (36) 29b>
    if ((CHILD = fork()) < 0) {
        perror("fork");
        die("fork failed", 0);
    }
```

Uses `CHILD` 29a and `die` 18b.

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

```
29a <globals 16b>+≡ (36) <24e
    int CHILD = 0;
```

Defines:

CHILD, used in chunks 28c, 29b, and 32–34.

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.

```
29b <fork 28c>+≡ (36) <28c
    if (CHILD == 0)
        doshell();
    else
        doio();
```

Uses CHILD 29a, doio 30e, and doshell 29c.

4.4 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.

```
29c <doshell 29c>≡ (36) 29d>
    void doshell() {
        close(PTM);
        fclose(OUTF);
```

Defines:

doshell, used in chunk 29b.

Uses OUTF 16b and PTM 24d.

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.

```
29d <doshell 29c>+≡ (36) <29c 29e>
    setsid();
    ioctl(PTS, TIOCSCTTY, 0);
```

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

```
29e <doshell 29c>+≡ (36) <29d 29f>
    dup2(PTS, 0);
    dup2(PTS, 1);
    dup2(PTS, 2);
    close(PTS);
```

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

```
29f <doshell 29c>+≡ (36) <29e 30a>
    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.

30a `<doshell 29c>+≡` (36) <29f 30b>

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

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.

30b `<doshell 29c>+≡` (36) <30a 30c>

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

Uses `cflg 21b`.

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

30c `<doshell 29c>+≡` (36) <30b>

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

Uses `die 18b`.

4.5 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.

30d `<includes 9b>+≡` (36) <28b 35f>

```
#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`.

30e `<doio 30e>≡` (36) 31a>

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

Defines:

`doio`, used in chunk 29b.

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 occurred (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`.

```
31a <doio 30e>+≡ (36) <30e 31b>
    fd_set fds;
    int sr;
    int highest = ((STDIN_FILENO > PTM) ? STDIN_FILENO : PTM) + 1;
```

Uses `PTM` 24d.

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

```
31b <doio 30e>+≡ (36) <31a 31c>
    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.

```
31c <doio 30e>+≡ (36) <31b 31d>
    if (!aflg)
        chunk(01);
    chunk(02);
    chunk(12);
    chunk(13);
    winsize(2);
```

Uses `aflg` 21a, `chunk` 19b, and `winsize` 27a.

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:

```
31d <doio 30e>+≡ (36) <31c 32a>
    struct timespec ts;
    if (clock_gettime(CLOCK_MONOTONIC, &ts) < 0) {
        perror("CLOCK_MONOTONIC");
        die("retrieving monotonic time failed", 0);
    }
```

Uses `die` 18b.

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.

```
32a  <doio 30e>+≡ (36) <31d 32c>
      if (!qflg)
          statusmsg(STARTMSG);
      Uses STARTMSG 32b and statusmsg 19c.
```

```
32b  <constants 15a>+≡ (36) <15b 35d>
      const char *STARTMSG = "forscript started on %s, file is %s\r\n";
      Defines:
      STARTMSG, used in chunk 32a.
```

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

```
32c  <doio 30e>+≡ (36) <32a 32d>
      while ((CHILD > 0) || drain) {
      Uses CHILD 29a.
```

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.

```
32d  <doio 30e>+≡ (36) <32c 33a>
      if (!drain) {
          FD_ZERO(&fds);
          FD_SET(STDIN_FILENO, &fds);
          FD_SET(PTM, &fds);
          sr = select(highest, &fds, NULL, NULL, NULL);
      Uses PTM 24d.
```

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.

```
33a <doio 30e>+≡ (36) <32d 33b>
    if (CHILD < 0) {
        int flags = fcntl(PTM, F_GETFL);
        if (fcntl(PTM, F_SETFL, (flags | O_NONBLOCK)) == 0) {
            drain = 1;
            continue;
        }
    }
```

Uses CHILD 29a and PTM 24d.

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.

```
33b <doio 30e>+≡ (36) <33a 33c>
    if (sr <= 0)
        continue;
```

When not in drain mode, 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.

```
33c <doio 30e>+≡ (36) <33b 33d>
    chunk16(&ts);
```

Uses chunk16 13.

If user input is available, it will be read into the buffer. The data will then be written to the transcript file, having S0 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.

```
33d <doio 30e>+≡ (36) <33c 34a>
    if (FD_ISSET(STDIN_FILENO, &fds)) {
        count = read(STDIN_FILENO, iobuf, BUFSIZ);
        if (count > 0) {
            fwrite(&S0, sizeof(S0), 1, OUTF);
            chunkwd((unsigned char *)iobuf, count);
            fwrite(&SI, sizeof(SI), 1, OUTF);
            write(PTM, iobuf, count);
        }
    }
```

Uses chunkwd 16a, OUTF 16b, PTM 24d, SI 15a, and S0 15a.

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.

```
34a <doio 30e>+≡ (36) <33d 34b>
    } // 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 16b and PTM 24d.

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

```
34b <doio 30e>+≡ (36) <34a 34c>
    if (fflg)
        fflush(OUTF);
```

Uses OUTF 16b.

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

```
34c <doio 30e>+≡ (36) <34b>
    }
    done();
}
```

Uses `done` 35b.

4.6 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.

```
34d <finish 34d>≡ (36)
    void finish(int signal) {
        UNUSED(signal);
        CHILD = -1;
    }
```

Defines:

`finish`, used in chunk 28a.

Uses `CHILD` 29a and `UNUSED` 35a.

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

```
35a <macros 35a>≡ (36)
    #define UNUSED(var) while (0) { (void)(var); }
```

Defines:

UNUSED, used in chunks 26b and 34d.

```
35b <done 35b>≡ (36) 35c>
    void done() {
        int status;
        wait(&status);
```

Defines:

done, used in chunks 34c and 35d.

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

```
35c <done 35b>+≡ (36) <35b 35e>
    if (!qflg)
        statusmsg(STOPMSG);
```

Uses `statusmsg` 19c and `STOPMSG` 35d.

```
35d <constants 15a>+≡ (36) <32b>
    const char *STOPMSG = "forscript done on %s, file is %s\r\n";
```

Defines:

STOPMSG, used in chunk 35c.

Uses `done` 35b.

```
35e <done 35b>+≡ (36) <35c>
    chunk03(status);
    fclose(OUTF);
    close(PTM);
    close(PTS);
    tcsetattr(0, TCSADRAIN, &TT);
    exit(EXIT_SUCCESS);
}
```

Uses `chunk03` 10a, `OUTF` 16b, and `PTM` 24d.

```
35f <includes 9b>+≡ (36) <30d>
    #include <sys/wait.h>
```

36 \langle *forscript.c* 36 \rangle \equiv
 \langle *featuretest* 23c \rangle
 \langle *macros* 35a \rangle
 \langle *includes* 9b \rangle

 \langle *constants* 15a \rangle

 \langle *globals* 16b \rangle

 \langle *die* 18b \rangle

 \langle *swrite* 17a \rangle

 \langle *chunkw* 16a \rangle

 \langle *chunkwhf* 17c \rangle

 \langle *chunkwm* 18a \rangle

 \langle *chunks* 8 \rangle

 \langle *statusmsg* 19c \rangle

 \langle *done* 35b \rangle

 \langle *finish* 34d \rangle

 \langle *winsize* 27a \rangle

 \langle *resized* 26b \rangle

 \langle *doshell* 29c \rangle

 \langle *doio* 30e \rangle

 int main(int argc, char *argv[])
 {
 \langle *setmyname* 20a \rangle
 \langle *getopt* 20c \rangle
 \langle *openoutfile* 23a \rangle
 \langle *openpt* 24f \rangle
 \langle *sigchld* 28a \rangle
 \langle *fork* 28c \rangle
 return EXIT_SUCCESS;
 }

```
}
```

Defines:

`main`, never used.

5 Evaluation

6 Summary

References

- [1] Knuth, Donald E.: *Literate Programming* (1992), Center for the Study of Language and Information, ISBN 978-0937073803.
- [2] *The util-linux project*, no longer maintained, last release 2.13-pre7 in 2006. <http://www.kernel.org/pub/linux/utils/util-linux/>
- [3] *The util-linux-ng project*, maintained by Karel Zak, current release 2.17. <http://userweb.kernel.org/~kzak/util-linux-ng/>
- [4] *The Linux man-pages project*, maintained by Michael Kerrisk, release 3.23. <http://www.kernel.org/doc/man-pages/>