Design and Implementation of a Documentation Tool for Interactive Commandline Sessions

Andreas Dewald Felix C. Freiling Tim Weber

University of Mannheim

Technical Report TR-2010-005
December 30, 2010

Abstract

In digital investigations it is important to document the examination of a computer system with as much detail as possible. Allthough never designed for digital investigations, many experts use the software `script` to record their whole terminal session while analyzing a target system. We analyze `script`'s deficiencies and present the design and implementation of `forscript` (forensic script), a software providing additional capabilities and mechanisms for digital investigations.

1 Introduction

1.1 Motivation: Documentation of Terminal Sessions

During a digital investigation, many actions are still performed on the command line. Especially in live response or during complicated manual operations it is useful that the investigator keeps a detailed record of his actions while analyzing a system. Such a log can help the investigator understand his motivations in retrospect and can help prove to other experts that certain actions were or were not performed during the investigation.

In principle, interactive command line 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. Many digital investigators still use a program called `script`. The purpose of `script` is to record everything printed to the users 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" [1]. But is it suitable for digital investigations?

1.2 Motivation: Deficiencies of `script`

The original `script` software consists of two programs: `script` and `scriptreplay`. Both are now part of the *util-linux-ng* package [8] that is under active development. The most recent log entry within the source code of `script` and `scriptreplay`, however, dates back to July 2000 when Andreas Buer "added q-option".

Technically, `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 applications output stream. The parent process forwards the users input to the client application. Recording terminates as soon as the child process exits.

`Script` uses a very simple file format to save the typescript. 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` X`\n"` is written, where X 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` Y`\n"`, where Y is the human-readable date and time when `script` terminated, is written.

Apart from recording terminal in- and output, `script` can also record timing data: Using the option `-t`, `script` will output timing data to `stderr` specifying the chronological progress of the terminal session. Using this data, the utility `scriptreplay` can display the recorded data in a video-like way. The timing output format is very simple. It consists of tuples of delay and byte count (space-separated), one per line, like in the following example:

```
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.

The two file formats produced by `script` show several shortcomings with regard to their use in digital investigations:

- Input coming from the users 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.[1]

- 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 but also 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 timings byte counter.
- In a sense, `script` is a typical Unix utility written in a single C file with hardly any code documentation beyond the manual page. Therefore, it is rather cumbersome to read and requires some effort to understand. We believe that software used in digital investigations should strive for better readability such that it is easier to quickly convice an expert that the code actually does what it promised.

1.3 Forensic `script`: `forscript`

Overall, `script` has severe deficiencies when used in digital investigations. In this paper, we report on the design and implementation of a successor tool `forscript` ("forensic script"). The main advantages of this tool are:

- `forscript` has the same command line user interface as `script`, i.e., users used to `script` can seamlessly switch to `forscript`.
- `forscript` defines a portable and extensible file format that contains all user input, timing information and detailed information about the environment. The file format allows appending of files in a natural way.
- Following the paradigm of literate programming [4] and using the programming tool `noweb` [5], `forscript` comes with its own user manual and documentation. In fact, this version of the paper contains the entire *C* source code in a well-readable way, i.e., this document *is* the program and vice versa.

[1] 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.

Literate programming [4] is a technique invented by Knuth when developing the TEX typesetting system. Instead of writing more or less commented source code, it propagates writing a continuous text with embedded code fragments. The tool `noweb` [5] is used to reassemble these code fragments so that they can be compiled into an executable program. Because of its advantages in creating readable source code, we feel that there is much potential for the use of literate programming in the development of software for digital investigations.

1.4 Roadmap

In Section 2, we elaborate the user interface of `forscript`, which basically is that of `script`. In Section 3 we present the extensible file format of `forscript`. We give a brief insight into the implementation of `forscript` in Section 4. The program is evaluated in Section 5, giving an example transcript file. Finally, Section 6 summarizes the work, gives a description of `forscript`s limitations and describes possibilities of future work.

The appendix completes the code given in Section 4 to form a complete literate program and can be consulted at the discretion of the reader. A chunk and identifier index appear at the end of this document.

2 User Interface of `forscript`

Since `Forscript`s invocation syntax has been designed to be compatible to `script`, most parameters result in the same behavior. We now give an overview over the interface of `forscript` and highlight the differences to the interface of `script`.

`Forscript` takes one optional argument, the file name of the output file (also called *transcript* file) to which the terminal session is logged. If no file name was supplied on the command line, the default name is `transcript`. This differs from `scripts` 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. If the file already exists and is a symbolic or hard link, `forscript` refuses to overwrite the file, as long as the file name is not explicitly provided.

There are several command-line switches that modify `forscript`'s behavior. For example, using `-a` it is possible to append the output of `forscript` to a transcript file. If the target transcript file already exists and is non-empty, it has to start with a valid and supported *file version* header that will be explained below.

Normally, `forscript` displays a message when it starts or quits and also it records its startup and termination time in the typescript file. With the parameter `-q`, all of these messages will not appear (quiet). This is similar to the behavior of `script`, only that no startup message will be written to the transcript file. This is because `scriptreplay` unconditionally discards the first line in a typescript file and so writing the startup message (`"Script started on ..."`) cannot be disabled in `script`.

By default, `forscript` will launch the shell specified by the environment variable `$SHELL`. If `$SHELL` it is not set, a default shell selected at compile time (`/bin/sh`, see page 32) is used. The shell will be called with `-i` as its first parameter, making it an interactive shell. However, if `forscript` 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. Note that all POSIX-compatible shells have to support the `-i` and `-c` parameters. This behavior is identical to that of `script`.

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

The parameter `-t` was used in `script` to output timing information. This parameter is accepted by `forscript` but ignored since `forscript` always records timing information into the transcript file.

Finally, if `forscript` is called with `-V` or `--version` as only parameter, it will print its version and exit. This behavior is identical to that of `script`.

If unsupported parameters are passed, `forscript` will print a short usage summary to *stderr* and exit. While running, the client applications output will be printed to *stdout*. Error messages will be printed to *stderr*.

3 `Forscript` File Format

This section explains the new file format as used by `forscript`. The file format allows an efficient combination of output and metadata within a single file.

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 Input Chunks

Input chunks contain the data that is sent to the client applications input stream, which is usually identical to the users 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 chunks 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 chunks 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

binary value	type name	size
`0x01`	file version	1 byte
`0x02`	begin of session	10 bytes
`0x03`	end of session	1 byte
`0x12`	environment variables	arbitrary number of C strings
`0x13`	locale settings	7 C strings
`0x16`	delay	two 4-byte values

Table 1: `forscript` file format metadata chunk types.

input chunk written to the transcript file is `0x0e 0x4e 0x10 0x0f 0x00 0x61 0x74 0x10 0x10 0x0f`.

3.2 Metadata Chunks

Metadata chunks, also called meta chunks, contain additional information about the file or the applications 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 listed in Table 1 and implemented in Appendix A.3.

Meta chunks can be 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`.

3.3 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).

4 Implementation Oveview

We give a brief insight into the code of `forscript` that is assembled within a single C source file and written as a literate program [4] using the tool `noweb` [5]. The `noweb` tool automatically introduces cross-references between code chunks so that readers can quickly find the corresponsing code. We only show the most interesting code chunks here. The full details can be found in the appendix.

```
7a   ⟨forscript.c 7a⟩≡
       ⟨declarations and definitions 15b⟩
       ⟨functions 15c⟩
       ⟨main 7b⟩
```

4.1 The Main Program

Here is the `main()` function. We first determine the program's name (as called on the command line), then we process the command line options. Afterwards we open the output file and a new pseudo terminal. 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. These two processes are forked after registering the appropriate signal handlers. Since neither the parent nor the child process should ever reach the end of `main()`, it returns `EXIT_FAILURE`.

```
7b   ⟨main 7b⟩≡                                                        (7a)
       int main(int argc, char *argv[]) {
         ⟨set my name 24c⟩
         ⟨process command line options 25b⟩
         ⟨open output file 26c⟩
         ⟨open new pseudo terminal 28e⟩
         ⟨register signal handlers 8a⟩
         ⟨fork subprocesses 8c⟩
         return EXIT_FAILURE;
       }
     Defines:
       main, never used.
```

4.2 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 processs 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 childs termination. The second signal handler, `resized()`, handles window size changes.

8a ⟨*register signal handlers* 8a⟩≡ (7b)

```
{ 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` 37b and `resized` 30a.

These functions and constants require `signal.h`.

8b ⟨*includes* 8b⟩≡ (15b) 18a ▷

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

4.3 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.

8c ⟨*fork subprocesses* 8c⟩≡ (7b) 9 ▷

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

Uses `CHILD` 8d and `die` 23b.

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

8d ⟨*globals* 8d⟩≡ (15b) 15a ▷

```
int CHILD = 0;
```

Defines:
`CHILD`, used in chunks 8c, 9, 23b, 35, and 37b.

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

9 ⟨*fork subprocesses* 8c⟩+≡ (7b) ◁8c

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

Uses `CHILD` 8d, `doio` 33c, and `doshell` 31c.

Further code can be found in the appendix.

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 2.6.32 [6], using glibc 2.10 and GCC 4.4.3 [2]. 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 forscript.nw > forscript.c
```

On the authors 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 document 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 authors 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` variables 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 authors 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 terminals 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 cursors 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 users 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 Conclusions and Future Work

We presented why `script`, although often used for digital 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.

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. Next, a tool that is able to parse the output `forscript` generates must be written. We leave this for future work.

`forscript` will be released by the third author as free software, available at a special website [7]. 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] BSD General Command Manual. Script(1). Manual page, part of *util-linux-ng* [8], July 2000.

[2] Free Software Foundation. Gnu compiler collection. `http://gcc.gnu.org/`, March 2010. release 4.4.3.

[3] Michael Kerrisk. The Linux *man-pages* project. `http://www.kernel.org/doc/man-pages/`, 2010. release 3.23.

[4] Donald E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[5] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994. The noweb system is available at `http://www.cs.tufts.edu/~nr/noweb/`.

[6] Linus Torvalds. The Linux kernel. `http://www.kernel.org/`, 2010. release 2.6.31.

[7] Tim Weber. forscript. `http://scytale.name/proj/forscript/`. release 1.0.0.

[8] Karel Zak. The *util-linux-ng* project. `http://userweb.kernel.org/~kzak/util-linux-ng/`, 2010. current release 2.17.

A 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 once without having to run separate processes.

A.1 Overview

Here's the current version number of `forscript`. Any future alterations can be explained in the text at relevant placed. `MYVERSION` is defined as a global constant.

15a ⟨*globals* 8d⟩+≡ (15b) ◁8d 22a▷

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

Defines:
`MYVERSION`, used in chunk 25b.

The code begins with feature test macros, ordinary macros and include statements. Afterwards, constants and global variables are defined.

15b ⟨*declarations and definitions* 15b⟩≡ (7a)

⟨*featuretest* 27b⟩
⟨*macros* 37c⟩
⟨*includes* 8b⟩
⟨*constants* 16a⟩
⟨*globals* 8d⟩

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). Then come the functions that write startup and shutdown messages on the screen, followed by the signal handling functions like `finish`. The functions `doshell` and `doio` are the main input/output functions that represent the parent and child processes.

15c ⟨*functions* 15c⟩≡ (7a)

⟨*die* 23b⟩
⟨*swrite* 22b⟩
⟨*chunkw* 21⟩
⟨*chunkwhf* 22d⟩
⟨*chunkwm* 23a⟩
⟨*chunks* 17a⟩
⟨*statusmsg* 24b⟩
⟨*done* 37d⟩
⟨*finish* 37b⟩
⟨*winsize* 30b⟩
⟨*resized* 30a⟩
⟨*doshell* 31c⟩
⟨*doio* 33c⟩

A.2 Constants

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

16a ⟨*constants* 16a⟩≡ (15b) 16b▷

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

Defines:
`DLE`, used in chunk 21.
`SI`, used in chunks 22d and 36c.
`SO`, used in chunks 22d and 36c.

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:

16b ⟨*constants* 16a⟩+≡ (15b) ◁16a 35b▷

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

Defines:
`ESCMAX`, used in chunk 21.
`ESCMIN`, used in chunk 21.

A.3 Metadata Chunk Types

We now 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 chunks explanation. The three functions `chunkwh()`, `chunkwf()` and `chunkwd()` that are used for actually writing the data to disk will be explained in section A.5. To be able to understand the code, it is sufficient to know that `chunkwh()` takes one parameter (the chunks 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`.

17a ⟨*chunks* 17a⟩≡ (15c) 17b▷

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

Defines:
`chunk01`, used in chunk 34c.
Uses `chunkwm` 23a.

`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 machines (signed) time zone offset to UTC in minutes. If these two bytes are set to `0xffff`, the machines timezone is unknown.

17b ⟨*chunks* 17a⟩+≡ (15c) ◁17a 18b▷

```
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 34c.
Uses chunkwm 23a.

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

18a ⟨*includes* 8b⟩+≡ (15b) ◁8b 20a▷
```
#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 childs return value. If the exit code of the child could not be determined, `0xff` is used instead.

18b ⟨*chunks* 17a⟩+≡ (15c) ◁17b 18c▷
```
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 38d.
Uses chunkwm 23a.

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.

18c ⟨*chunks* 17a⟩+≡ (15c) ◁18b 19a▷
```
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 30b.
Uses chunkwm 23a and winsize 30b.

`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.

19a ⟨*chunks* 17a⟩+≡ (15c) ◁18c 19b▷

```
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 34c.
Uses `chunkwd` 21, `chunkwf` 22d, and `chunkwh` 22d.

`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.

19b ⟨*chunks* 17a⟩+≡ (15c) ◁19a 20b▷

```
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 34c.
Uses chunkwd 21, chunkwf 22d, and chunkwh 22d.

setlocale() requires locale.h:

20a ⟨*includes* 8b⟩+≡ (15b) ◁18a 22c▷

```
  #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.

20b ⟨*chunks* 17a⟩+≡ (15c) ◁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 36b.
Uses chunkwm 23a.

A.4 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.

A.5 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:

21 ⟨*chunkw* 21⟩≡ (15c)

```
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 19, 23a, and 36c.
Uses `DLE` 16a, `ESCMAX` 16b, `ESCMIN` 16b, `OUTF` 22a, and `swrite` 22b.

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

22a ⟨*globals* 8d⟩+≡ (15b) ◁15a 24a▷

```
  FILE *OUTF;
```

Defines:
  `OUTF`, used in chunks 21, 22d, 24b, 27, 31c, and 36–38.

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:

22b ⟨*swrite* 22b⟩≡ (15c)

```
  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 21 and 22d.

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

22c ⟨*includes* 8b⟩+≡ (15b) ◁20a 23c▷

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

There are functions to write chunk headers and footers:

22d ⟨*chunkwhf* 22d⟩≡ (15c)

```
  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 19 and 23a.
  `chunkwh`, used in chunks 19 and 23a.
Uses `OUTF` 22a, `SI` 16a, `SO` 16a, and `swrite` 22b.

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

23a ⟨*chunkwm* 23a⟩≡ (15c)

```
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 17, 18, and 20b.
Uses `chunkwd` 21, `chunkwf` 22d, and `chunkwh` 22d.

A.6 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.

23b ⟨*die* 23b⟩≡ (15c)

```
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, "%s", message);
  fprintf(stderr, "; exiting.\n");
  exit(EXIT_FAILURE);
}
```

Defines:
`die`, used in chunks 8c, 24b, 27–34, 36b, and 38d.
Uses `CHILD` 8d, `MYNAME` 24a, and `TTSET` 28d.

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

23c ⟨*includes* 8b⟩+≡ (15b) ◁22c 25c▷

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

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

24a ⟨*globals* 8d⟩+≡ (15b) ◁22a 25d▷
```
char *MYNAME;
```
Defines:
MYNAME, used in chunks 23–26.

A.7 Startup and Shutdown Messages

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

24b ⟨*statusmsg* 24b⟩≡ (15c)
```
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 35a and 38b.
Uses die 23b, OUTF 22a, and OUTN 26b.

A.8 Initialization

A.8.1 Determining the Binarys 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.

24c ⟨*set my name* 24c⟩≡ (7b) 25a▷
```
MYNAME = argv[0];
```
Uses MYNAME 24a.

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.

25a ⟨*set my name* 24c⟩+≡ (7b) ◁24c

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

Uses `MYNAME` 24a.

A.8.2 Processing 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.

25b ⟨*process command line options* 25b⟩≡ (7b) 26a▷

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

Uses `MYNAME` 24a and `MYVERSION` 15a.

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

25c ⟨*includes* 8b⟩+≡ (15b) ◁23c 27a▷

```
#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:

25d ⟨*globals* 8d⟩+≡ (15b) ◁24a 25e▷

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

Defines:
`aflg`, used in chunks 26, 27, and 34c.

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

25e ⟨*globals* 8d⟩+≡ (15b) ◁25d 26b▷

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

Defines:
`cflg`, used in chunks 26a and 32d.

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:

26a ⟨*process command line options* 25b⟩+≡ (7b) ◁25b

```
{ 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` 25d, `cflg` 25e, and `MYNAME` 24a.

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

26b ⟨*globals* 8d⟩+≡ (15b) ◁25e 28c▷

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

Defines:
`OUTN`, used in chunks 24b, 26c, and 27c.

A.8.3 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.

26c ⟨*open output file* 26c⟩≡ (7b) 27c▷

```
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` 24a and `OUTN` 26b.

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

27a ⟨*includes* 8b⟩+≡ (15b) ◁25c 28a▷
```
  #include <sys/types.h>
  #include <sys/stat.h>
```

27b ⟨*featuretest* 27b⟩≡ (15b) 28b▷
```
  #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.

27c ⟨*open output file* 26c⟩+≡ (7b) ◁26c 27d▷
```
  if ((OUTF = fopen(OUTN, (aflg ? "a+" : "w"))) == NULL) {
    perror(OUTN);
    die("the output file could not be opened", 0);
  }
```
Uses `aflg` 25d, `die` 23b, `OUTF` 22a, and `OUTN` 26b.

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.

27d ⟨*open output file* 26c⟩+≡ (7b) ◁27c
```
  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` 25d, `die` 23b, and `OUTF` 22a.

A.9 Preparing a New Pseudo Terminal

While `script` uses manual PTY allocation (by trying out device names) or BSDs `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.

28a ⟨*includes* 8b⟩+≡ (15b) ◁27a 28f▷

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

28b ⟨*featuretest* 27b⟩+≡ (15b) ◁27b

```
#define _POSIX_C_SOURCE 200112L
```

Defines:
`_POSIX_C_SOURCE`, never used.

The PTYs master and slave file descriptors will be stored in these global variables:

28c ⟨*globals* 8d⟩+≡ (15b) ◁26b 28d▷

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

Defines:
`PTM`, used in chunks 29–31, 34–36, and 38d.

Additionally, the settings of the terminal `forscript` runs in will be saved in the global variable `TT`. This variable is used to duplicate the terminals 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.

28d ⟨*globals* 8d⟩+≡ (15b) ◁28c

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

Defines:
`TTSET`, used in chunks 23b and 28e.

28e ⟨*open new pseudo terminal* 28e⟩≡ (7b) 29a▷

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

Uses `die` 23b and `TTSET` 28d.

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

28f ⟨*includes* 8b⟩+≡ (15b) ◁28a 31a▷

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

A new PTY master is requested like this:

29a ⟨*open new pseudo terminal* 28e⟩+≡ (7b) ◁28e 29b▷

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

Uses `die` 23b and `PTM` 28c.

Then, access to the slave is granted.

29b ⟨*open new pseudo terminal* 28e⟩+≡ (7b) ◁29a 29c▷

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

Uses `die` 23b and `PTM` 28c.

The slaves 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.

29c ⟨*open new pseudo terminal* 28e⟩+≡ (7b) ◁29b 29d▷

```
{ 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` 23b and `PTM` 28c.

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 [3] 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 terminals behavior will not change, but `forscript` can now document the terminals data stream with maximum accuracy.

29d ⟨*open new pseudo terminal* 28e⟩+≡ (7b) ◁29c

```
{
  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` 23b.

A.9.1 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.

30a ⟨*resized* 30a⟩≡ (15c)

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

Defines:
  `resized`, used in chunk 8a.
Uses `UNUSED` 37c and `winsize` 30b.

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 applications 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.

30b ⟨*winsize* 30b⟩≡ (15c)

```
  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 18c, 30a, 31b, and 34c.
Uses `chunk11` 18c, `die` 23b, and `PTM` 28c.

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

31a ⟨*includes* 8b⟩+≡ (15b) ◁28f 33b▷

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

The client PTYs 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.

31b ⟨*openpt* 31b⟩≡

```
  winsize(1);
```

Uses `winsize` 30b.

A.10 Running the Target Application

The `doshell()` function is run in the child process, whose only task 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.

31c ⟨*doshell* 31c⟩≡ (15c)

```
void doshell() {
  close(PTM);
  fclose(OUTF);
        ⟨change the terminal 31d⟩
        ⟨determine the shell 32b⟩
        ⟨execute the shell 32d⟩
}
```

Defines:
`doshell`, used in chunk 9.
Uses `OUTF` 22a and `PTM` 28c.

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.

31d ⟨*change the terminal* 31d⟩≡ (31c) 32a▷

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

Uses `die` 23b.

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

32a ⟨*change the terminal* 31d⟩+≡ (31c) ◁31d

```
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` 23b.

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.

32b ⟨*determine the shell* 32b⟩≡ (31c) 32c▷

```
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.

32c ⟨*determine the shell* 32b⟩+≡ (31c) ◁32b

```
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.

32d ⟨*execute the shell* 32d⟩≡ (31c) 33a▷

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

Uses `cflg` 25e.

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.

33a ⟨*execute the shell* 32d⟩+≡ (31c) ◁32d
```
    perror(shell);
    die("execing the shell failed", 0);
```
Uses `die` 23b.

A.11 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.

33b ⟨*includes* 8b⟩+≡ (15b) ◁31a 38a▷
```
  #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`.

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

33c ⟨*doio* 33c⟩≡ (15c)
```
  void doio() {
    char iobuf[BUFSIZ];
    int count;
          ⟨preparations of main loop 34a⟩
          ⟨the main loop 35c⟩
          done();
  }
```
Defines:
  `doio`, used in chunk 9.
Uses `done` 37d.

Preparing the Main Loop

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`.

34a ⟨*preparations of main loop* 34a⟩≡ (33c) 34b ▷

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

Uses `PTM` 28c.

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

34b ⟨*preparations of main loop* 34a⟩+≡ (33c) ◁ 34a 34c ▷

```
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.

34c ⟨*preparations of main loop* 34a⟩+≡ (33c) ◁ 34b 34d ▷

```
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` 25d, `chunk01` 17a, `chunk02` 17b, `chunk12` 19a, `chunk13` 19b, `die` 23b, and `winsize` 30b.

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:

34d ⟨*preparations of main loop* 34a⟩+≡ (33c) ◁ 34c 35a ▷

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

Uses `die` 23b.

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.

35a ⟨*preparations of main loop* 34a⟩+≡ (33c) ◁34d

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

Uses `STARTMSG` 35b and `statusmsg` 24b.

35b ⟨*constants* 16a⟩+≡ (15b) ◁16b 38c▷

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

Defines:
  `STARTMSG`, used in chunk 35a.

The Main Loop

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

35c ⟨*the main loop* 35c⟩≡ (33c)

```
    while ((CHILD > 0) || drain) {
                ⟨main loop body 35d⟩
          }
```

Uses `CHILD` 8d.

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 PTYs 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.

35d ⟨*main loop body* 35d⟩≡ (35c) 36d▷

```
      if (!drain) {
        FD_ZERO(&fds);
        FD_SET(STDIN_FILENO, &fds);
        FD_SET(PTM, &fds);
        sr = select(highest, &fds, NULL, NULL, NULL);
                         ⟨further standard input processing 35e⟩
      }
```

Uses `PTM` 28c.

If the child process has terminated, there may still be data left in the buffers, therefore the terminals 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.

35e ⟨*further standard input processing* 35e⟩≡ (35d) 36a▷

```
        if (CHILD < 0) {
          int flags = fcntl(PTM, F_GETFL);
          if (fcntl(PTM, F_SETFL, (flags | O_NONBLOCK)) == 0) {
            drain = 1;
            continue;
          }
        }
```
Uses `CHILD` 8d and `PTM` 28c.

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 childs 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.

36a ⟨*further standard input processing* 35e⟩+≡ (35d) ◁35e 36b▷
```
        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.

36b ⟨*further standard input processing* 35e⟩+≡ (35d) ◁36a 36c▷
```
        if (chunk16(&ts) < 0)
          die(NULL, 0x16);
```
Uses `chunk16` 20b and `die` 23b.

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.

36c ⟨*further standard input processing* 35e⟩+≡ (35d) ◁36b
```
        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` 21, `OUTF` 22a, `PTM` 28c, `SI` 16a, and `SO` 16a.

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.

36d ⟨*main loop body* 35d⟩+≡ (35c) ◁35d 37a▷
```
      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` 22a and `PTM` 28c.

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

37a ⟨*main loop body* 35d⟩+≡ (35c) ◁36d
```
      if (fflg)
        fflush(OUTF);
```
Uses `OUTF` 22a.

A.12 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.

37b ⟨*finish* 37b⟩≡ (15c)
```
  void finish(int signal) {
    UNUSED(signal);
    CHILD = -1;
  }
```
Defines:
  `finish`, used in chunk 8a.
Uses `CHILD` 8d and `UNUSED` 37c.

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

37c ⟨*macros* 37c⟩≡ (15b)
```
  #define UNUSED(var) while (0) { (void)(var); }
```
Defines:
  `UNUSED`, used in chunks 30a and 37b.

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()`.

37d ⟨*done* 37d⟩≡ (15c) 38b▷
```
  void done() {
    int status;
    wait(&status);
```
Defines:
  `done`, used in chunks 33c and 38c.

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

38a ⟨*includes* 8b⟩+≡ (15b) ◁33b
```
  #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.

38b ⟨*done* 37d⟩+≡ (15c) ◁37d 38d▷
```
    if (!qflg)
      statusmsg(STOPMSG);
```
Uses `statusmsg` 24b and `STOPMSG` 38c.

38c ⟨*constants* 16a⟩+≡ (15b) ◁35b
```
  const char *STOPMSG = "forscript done on %s, "
                        "file is %s\r\n";
```
Defines:
`STOPMSG`, used in chunk 38b.
Uses `done` 37d.

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

38d ⟨*done* 37d⟩+≡ (15c) ◁38b
```
    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` 18b, `die` 23b, `OUTF` 22a, and `PTM` 28c.

Chunk Index

⟨*change the terminal* 31d⟩
⟨*chunks* 17a⟩
⟨*chunkw* 21⟩
⟨*chunkwhf* 22d⟩
⟨*chunkwm* 23a⟩
⟨*constants* 16a⟩
⟨*declarations and definitions* 15b⟩
⟨*determine the shell* 32b⟩
⟨*die* 23b⟩
⟨*doio* 33c⟩
⟨*done* 37d⟩
⟨*doshell* 31c⟩
⟨*execute the shell* 32d⟩
⟨*featuretest* 27b⟩
⟨*finish* 37b⟩
⟨*fork subprocesses* 8c⟩
⟨*forscript.c* 7a⟩
⟨*functions* 15c⟩
⟨*further standard input processing* 35e⟩
⟨*globals* 8d⟩
⟨*includes* 8b⟩
⟨*macros* 37c⟩
⟨*main* 7b⟩
⟨*main loop body* 35d⟩
⟨*open new pseudo terminal* 28e⟩
⟨*open output file* 26c⟩
⟨*openpt* 31b⟩
⟨*preparations of main loop* 34a⟩
⟨*process command line options* 25b⟩
⟨*register signal handlers* 8a⟩
⟨*resized* 30a⟩
⟨*set my name* 24c⟩
⟨*statusmsg* 24b⟩
⟨*swrite* 22b⟩
⟨*the main loop* 35c⟩
⟨*winsize* 30b⟩

Identifier Index

`_POSIX_C_SOURCE`: <u>28b</u>
`_XOPEN_SOURCE`: <u>27b</u>
`aflg`: <u>25d</u>, 26a, 27c, 27d, 34c
`cflg`: <u>25e</u>, 26a, 32d
`CHILD`: 8c, <u>8d</u>, 9, 23b, 35c, 35e, 37b
`chunk01`: <u>17a</u>, 34c
`chunk02`: <u>17b</u>, 34c
`chunk03`: <u>18b</u>, 38d
`chunk11`: <u>18c</u>, 30b
`chunk12`: <u>19a</u>, 34c
`chunk13`: <u>19b</u>, 34c
`chunk16`: <u>20b</u>, 36b
`chunkwd`: 19a, 19b, <u>21</u>, 23a, 36c
`chunkwf`: 19a, 19b, <u>22d</u>, 23a
`chunkwh`: 19a, 19b, <u>22d</u>, 23a
`chunkwm`: 17a, 17b, 18b, 18c, 20b, <u>23a</u>
`die`: 8c, <u>23b</u>, 24b, 27c, 27d, 28e, 29a, 29b, 29c, 29d, 30b, 31d, 32a, 33a, 34c, 34d, 36b, 38d
`DLE`: <u>16a</u>, 21
`doio`: 9, <u>33c</u>
`done`: 33c, <u>37d</u>, 38c
`doshell`: 9, <u>31c</u>
`ESCMAX`: <u>16b</u>, 21
`ESCMIN`: <u>16b</u>, 21
`finish`: 8a, <u>37b</u>
`main`: <u>7b</u>
`MYNAME`: 23b, <u>24a</u>, 24c, 25a, 25b, 26a, 26c
`MYVERSION`: <u>15a</u>, 25b
`OUTF`: 21, <u>22a</u>, 22d, 24b, 27c, 27d, 31c, 36c, 36d, 37a, 38d
`OUTN`: 24b, <u>26b</u>, 26c, 27c
`PTM`: <u>28c</u>, 29a, 29b, 29c, 30b, 31c, 34a, 35d, 35e, 36c, 36d, 38d
`resized`: 8a, <u>30a</u>
`SI`: <u>16a</u>, 22d, 36c
`SO`: <u>16a</u>, 22d, 36c
`STARTMSG`: 35a, <u>35b</u>
`statusmsg`: <u>24b</u>, 35a, 38b
`STOPMSG`: 38b, <u>38c</u>
`swrite`: 21, <u>22b</u>, 22d
`TTSET`: 23b, <u>28d</u>, 28e
`UNUSED`: 30a, 37b, <u>37c</u>
`winsize`: 18c, 30a, <u>30b</u>, 31b, 34c