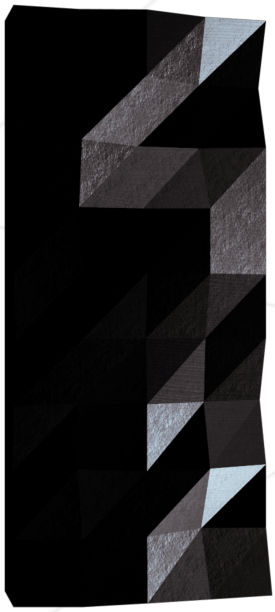

42sh

The return of the Vengeance from the past!

Summary: Time has come. Make way for the famous 42sh !

Version: 4.1

Contents

Chapter I

Foreword

I.1 How to Get into Orbit

Getting into orbit over Kerbin is rather simple, but it will require some knowledge and preparation. Space begins at 70,000m above the planet Kerbin. Stay above that for an entire flight around the planet and you're in orbit. The process to get into orbit follows a simple progression:

- Launch straight up to 10km
- Change your Pitch to 45° east and keep the engines on until your projected Apoapsis is above 70km
- Change your pitch to horizontal just before you reach the Apoapsis
- Burn at Apoapsis until your Periapsis is above 70km

Specifications:

- Length: 15-20 minutes
- Difficulty: Harder than a suborbital flight, easier than an orbital intercept.
- Skills needed: Seat of the pants
- For version: Every version (tested on 0.23)

I.2 Steps

I.2.1 Step 0 - Rocket Design

A liquid fueled rocket with at least two stages preferably. Anything less will either only get you suborbital or an unwieldy expensive super large fuel tank thats only good for being an orbiting billboard. The cheapest orbiter you can build with the current stock game (0.22) is:

- Unmanned: Probodobodyne OKTO2 or Manned: Command Pod Mk1 with Mk16 Parachute and TR-18A Stack Decoupler
- FL-T400 Fuel Tank
- LV-909 Liquid Fuel Engine
- TR-18A Stack Decoupler
- FL-T800 Fuel Tank
- LV-T30 Liquid Fuel Engine

For fun, use the LVT-30 engine for your first stage as it doesn't have thrust vectoring and thus will challenge you to actually fly the craft into orbit using your WASD keys keeping your navball on target. Your control module by default provides enough SAS control by itself to prevent you from going out of control for this mission. Make sure your staging sequence is the way you want it, else you can add your flight to the countless list of catastrophic mission failures.

I.2.2 Step 1 - Launch Prep

Prepare your launch.

- Set your map view with the m key to see your rocket at the launch site from space, tilted so you are looking north. You want to be able to see your apoapsis marker during your gravity turn so you can gauge when to cut your engines and coast up to it prior to burning your orbiting maneuver.
- Set your thrust to maximum by holding Shift.
- Toggle on SAS by hitting t.

I.2.3 Step 2 - The Launch

Say your countdown if you wish, and hit spacebar to launch into...well, space...without the bar (you'll make a space station with a bar at this end of the galaxy later).

I.2.4 Step 3 - Get up to 10,000 meters

Keep your rocket pointed straight up (use your navball to keep your dot on the top dot on the blue part of the navball) until you hit around 9,900 meters. Your first engine should cut out before this, just jettison it with the spacebar and burn your final engine. Throttle your second engine down to 2/3rds power since the atmosphere is weaker here and won't slow you down as much, and you will save precious fuel.

I.2.5 Step 4 - Gravity Turn 45 degrees East until 70km Apoapsis

Now for the fun part. Assuming you are pointing straight up, hit the d key to turn your ship using your navball until your dot is able to slide left and right on the 90 degree east line. Then tip on that line down toward the ground until you hit the 45 degree east mark. Hit your t key for SAS to help you keep it there, but don't rely on it. You will need to make course corrections until you get past 70km and enter space. At this point you should be going into space at a 45 degree angle from the ground, and in an easterly direction. Doing so will gain you speed and not waste the fuel you will need to get into orbit.

I.2.6 Step 5 - Get Your Apoapsis above 70km

While climbing up to 70km or 70,000 meters, it's now time to switch to your map view and control your ship from there entirely. Hopefully your navball is toggled on, else click on the collapsed tab at the bottom of the map view. You will need to keep burning your fuel until you see your apoapsis marker reach 70km. You can see how high it is by hovering your mouse over it. Once it hits 70km (I usually shoot for 75km to buy me some head room) your craft will be able to coast up to it without any further fuel. Feel free to cut your engines with the x key and save some fuel for your orbital burn.

I.2.7 Step 6 - Orient for On-Orbit Burn

As you approach apoapsis (preferably before 30 seconds prior) orient your ship to the 0 degree latitudinal mark, heading east. Again, you should be in your map view when you start your burn. Hit tab to center your view on Kerbin and zoom out so you can see your orbit as it forms.

I.2.8 Step 7 - Burn into Orbit

Once you are 10-30 seconds away from your apoapsis, begin your on-orbit burn using the Shift key to throttle up. Full throttle is best at lower altitude apoapsises since you don't want to burn past apoapsis lest you waste precious return fuel if that's your intent. If you aren't concerned, go full throttle baby at any point near apoapsis and claim a stake with the stars. You only need to burn your fuel long enough until you see a periapsis marker appear on the other side of the orbit from the apoapsis, and you see a full orbit circle, then cut your engine with the x key. Congrats, you made it into orbit. It's usually a good idea to keep burning your fuel until your new periapsis marker is above 70km. If it's below, then your orbit will cause your craft to aerobrake, eventually returning to Kerbin. If your periapsis is ever above 70km, congrats, you will orbit Kerbin forever. If

you have a manned flight, and fuel is low, then only burn until your periapsis is below 70km to ensure a safe return to Kerbin.

I.3 Finishing word

After orbiting for a while, depending on your fuel (with this design you won't have much if any at all) you can orient for a deorbit burn by burning backwards in the direction of your travel, once you are at your apoapsis point for maximum efficiency lest you be stuck in space with a manned crew forever.

Chapter II

Introduction

With the `Minishell` project, you discovered the "behind the scenes" of a shell. More specifically, you explored the process' synchronisation creation with functions such as `fork` and `wait`. You also discovered inter-process communication, using pipes, as well as redirections and line edition, using termcaps.

With the `42sh` project, you will go even further by adding functionailities such as globbing management, subshells and inhibitor.

You'll come upon (or rediscover if you worked on the `ft_select` project), `termcaps`. This library will enable you to add a line edition feature to your shell. With this feature, you will be able to edit a typo made on your command without having to retype it completely as well as repeat a previous command using history. It goes without saying that you'll have to code those features. The good news is that `termcaps` will help you do it, the bad news is it won't do it for you!

Chapter III

Objectives

As a coder in training, there are moments that mark your life. Forever. `42sh` is one of those moments. Achieving this project is a milestone at 42.

This project is about writing the most stable and complete `UNIX` shell possible. You already know a lot of shells, and each has its own features, from the humble `sh` available on every `UNIX` distribution in the world to the complete and complex `zsh`, which many of you use without really knowing why. There are many other shells, such as `bash`, `csh`, `tcsh`, `ksh`, `ash`, etc. `42sh` is your first real shell. It is common practice for students to choose a reference shell and try to replicate its basic behavior. This can be a good strategy if you know the reasons why you chose that one shell specifically. Consider this: if you choose `zsh` as a reference, you will need to commit to a long and difficult quest, even if it is highly instructive. Through this quest, you will learn humility and what it means to work hard.

So what to do? We suggest you try a couple of shells first to get an idea of their subtle and twisted differences. However, don't be fooled by the apparent simplicity of the `sh` shell. Recoding a thorough and stable `sh` is an achievement in itself. Forget about the "moons and stars" shells. In the end, they won't be able to do more than just a few pipes and redirections.

The keyword here is "stability". A humble and indestructible `42sh` is always better than a kooky `42sh` with countless features that segfault in unanticipated ways. The latter will be worth 0 in the end. Make sure that the `42sh` you submit is stable. I cannot insist enough on that last point.

This project will be slightly different from the other projects you worked on until now and can be broken down into two distinctive parts described as follows:

- The **mandatory** part must be done. It is the minimum prerequisite for a shell.
- An **optional** part, which will be considered only if the mandatory part works perfectly in its entirety.

The mandatory part alone will not validate the project. To succeed, you'll have to implement optional features. There are 2 main reasons for this: firstly, the mandatory part only constitutes the very basic features of a shell, and secondly, we want to challenge you to select features by yourself and to determine your own project's objectives.

Finally, please note that we will be looking at the "usability" aspect of your `42sh` . It would be wise to make sure that an ordinary `sh` or `bash` user can intuitively use your `42sh` ...

Chapter IV

General Instructions

- This project will be corrected by humans only. You're allowed to organize and name your files as you see fit, but you must follow the following rules.
- The executable file must be named `42sh`.
- Your `Makefile` must compile the project and must contain the usual rules. It must recompile and re-link the program only if necessary.
- If you are clever, you will use your library for your `42sh`. Submit also your folder `libft` including its own `Makefile` at the root of your repository. Your `Makefile` will have to compile the library, and then compile your project.
- You have to handle errors carefully. In no way can your program quit in an unexpected manner (Segmentation fault, bus error, double free, etc).
- Your terminal cannot display gibberish, be smart with the settings.
- Your program cannot have memory leaks.

- Within the mandatory part, you are allowed to use only the following `libc` functions:
  - The entire `libc`.
  - The `readline` library (`-lreadline`).
  - The `termcap` and the `ncurses` libraries (`-ltermcap` and `-lncurses`).
- You are allowed to use other functions or other libraries to complete the bonus part as long as their use is justified during your defense. Be smart!

Chapter V

Mandatory part

- Minishell prerequisites.
    - Prompt display.
    - Run commands with their parameters and `PATH` monitoring.
    - Correct spaces and tabulations monitoring.
- 42sh prerequisites.
    - Full edition of commands line
    - Redirection and aggregation operators.
        - `>`
        - `>>`
        - `<`
        - `<<`
        - `>&`
        - `<&`
    - pipe `|`
    - separators `;`
- Following built-ins:
    - `cd`
    - `echo`
    - `exit`
    - `type`
- The following logical operators `"&&"` and `"||"`.

Precedence of operators

Control operators ; && || or redirection operators < > |, got a precedence. Be carefull !
For instance, this 2 commands don't return same result:

- `ls doesnotexist . 2>&1 >/dev/null`
- `ls doesnotexist . >/dev/null 2>&1`

Don't forget to check your shell **grammary**.

- Monitoring of intern shell variables. (Don't care about read-only variables)
  - Intern variable creation depending on syntax: `name=value`
  - Intern variable exportation to the environment, via built-in `export`.
  - Possibility to list shell intern variables via built-in `set` (no option required).
  - Intern and environement variables revocation, via built-in `unset` (no option required).
  - Environment variable creation for unique command, for instance: `HOME=/tmp cd`.
  - Simple expansion of parameters depending on syntax `${}` (no additionnal format required).
  - Exit code access of previously command via the expansion `${?}`.
- Job control monitoring, with built-ins `jobs`, `fg`, `bg` and the `&` operator.
- A correct monitoring of all signals.
- Each built-in must have the options stated by the POSIX standard, except for explicit cases such as `set` or `unset`.

Explication of env, setenv, unsetenv

env is a binary and not a built-in in shells. You have been asked to implement this in order to understand his behaviour. You are not required to provide it this time.

Built-ins setenv and unsetenv are exclusive to the family shell CSH. Behaviour of certain shell shouldn't be being copied. Unix FAG: Csh Considered Harmful

Chapter VI

Modular Part

The features requested in the mandatory part represent the strict minimum of what is expected from a shell. You must now select and implement more advanced features. A minimum of **6 features** from the list below is required to validate the project.

However, keep in mind that stability will be much more important than quantity. Do not include an option that causes a problem for the rest of the program, for example. And remember that this part will only be evaluated if the mandatory part is complete and indestructible.

Modular features:

- Inhibitors `"` (double quote), `'` (simple quote) and `\`
- Pattern matching (globbing): `*`, `?`, `[]`, `!` and character intervals with `\` (backslash)
- Tilde expansion and additional parameter formats:
  - `~`
  - `${parameter:-word}`
  - `${parameter:=word}`
  - `${parameter:?word}`
  - `${parameter:+word}`
  - `${#parameter}`
  - `${parameter%}`
  - `${parameter%%}`
  - `${parameter#}`
  - `${parameter##}`
- Control groups and sub-shells: `()`, `{};`
- Control substitution: `$()`
- Arithmetic expansion: `$(())`
  Only these operators:

- Incrementality, decrementing `++ --`
- Addition, Soustraction `+ -`
- Multiplication, division, modulo `* / %`
- Comparison `<= >= < >`
- Equality, differencies `== !=`
- AND/OR logical `&& ||`

See the page Arithmetic Precision and Operations for more information on how operators work.

- Process substitution: `<()`, `>()`
- Complete management of the history:
  - Expansions:
    - `!!`
    - `!word`
    - `!number`
    - `!-number`
  - Saving to a file so that it can be used over several sessions
  - Built-in `fc` (all POSIX options)
  - Incremental search in the history with `CTRL-R`
- Contextual dynamic completion of commands, built-ins, files, internal and environment variables. What is meant by contextual? re-we use the "`ls /`" command and your cursor is on the `/`, then a contextual completion will only propose the content of the root folder and not the commands or built-ins. Same for this case: "`echo ${S`", the completion should only propose variable names that start with S, whether internal or environmental.
- The two editing modes of the Vi and Readline command line. The possibility to choose between one or the other mode will be done via the `-o` option of the built-in `set`.

  Here is the shortcut list that you must implement:

  - **Vi**: `#`, , `v`, `j`, `k`, `l`, `h`, `w`, `W`, `e` `E` `b` `B` `^` `$` `0` `|`, `f`, `F`, `;`, `,`, `a`, `A`, `i`, `I`, `r`, `R`, `c`, `C`, `S`, `x`, `X`, `d`, `D`, `y`, `y`, `Y`, `p`, `P`, `u`, `U`.
  - **Readline**: `C-b`, `C-f`, `C-p`, `C-n`, `C-_`, `C-t`, `A-t`.

  The explanation of Vi shortcuts can be found in the sh implementation and more particularly in the `EXTENDED DESCRIPTION` section. For Readline, look at this page.
- `Alias` management via built-ins `alias` and `unalias`
- A hash table and built-in `hash` to interact with it.
- The built-in test with the following operators: `-b`, `-c`, `-d`, `-e`, `-f`, `-g`, `-L`, `-p`, `-r`, `-S`, `-s`, `-u`, `-w`, `-x`, `-z`, `=`, `!=`, `-eq`, `-ne`, `-ge`, `-lt`, `-le`, `!`. As well as the possibility of a simple operand, without operator.

In case of doubt about the behavior of a feature, refer to the POSIX standard. It is perfectly acceptable to choose to implement a feature differently, if that makes you seems consistent for your shell, but it is not acceptable to do less than the POSIX specs by "laziness".

Chapter VII

Bonus part

VII.1 List of possible bonuses

- Shell Script (`while`, `for`, `if`. `case`, `function`, etc.)
- Autocompletion for order/built-ins parameters
- A shell compliant with the POSIX Standard

VII.2 Prerequisites for Considering Bonuses

New feature is on the menu concerning the 42sh bonuses. These will only be taken into account if your code is **clear** and **clean**.
What do we mean by that?

- No ternary every 3 lines.
- Explicit function names (no `ft_parse1`, `ft_parse2`, etc.)
- This also applies to variable names
- Use judiciously the **const qualifier**.
- Have a **git** history and explicit commit messages
- Have automated tests

`The bonus part will only be assessed if the mandatory part is PERFECT. Perfect means the mandatory part has been integrally done and works without malfunctioning. If you have not passed ALL the mandatory requirements, your bonus part will not be evaluated at all.`

Chapter VIII

Submission and peer-evaluation

Turn in your assignment in your `Git` repository as usual. Only the work inside your repository will be evaluated during the defense. Don't hesitate to double check the names of your folders and files to ensure they are correct.