Comprehensive Guide to Automating and Optimizing Gentoo Linux Installation and Configuration: In-Depth Analysis of Wayland (Hyprland) Integration with Nvidia Proprietary Graphics Drivers, Leveraging Advanced Bash Shell Scripting Techniques for a Streamlined and Exceptionally Minimalist System Setup

Emre AKYÜZ

Reddit: */u/RusselsTeapot*

GitHub: *@emrakyz*

December 18, 2023

Abstract

This document provides an exhaustive and meticulous analysis of a Bash shell script designed to facilitate the construction of a Gentoo Linux system. The script is engineered to operate autonomously, negating the necessity for the user to possess extensive knowledge in Bash programming. However, the primary objective of this document is to furnish comprehensive insights into Gentoo Linux, Wayland, Hyprland, advanced shell scripting techniques, the GNU core utilities, the intricacies of Nvidia graphics drivers and their interplay with the Linux kernel. The document endeavors to elucidate the workings of the aforementioned script in an elaborate manner, serving as an instructional guide. This guide is intended to empower users to adapt or enhance the script according to their individual requirements. Such a detailed tutorial not only augments the transparency of the script but also addresses the critical concern of executing an indiscriminate file sourced from the internet, which is widely acknowledged as a security risk. In its scope, the paper includes a thorough exposition on the installation, configuration, optimization, and management of Gentoo Linux. It also delves into the burgeoning interest in the new Wayland compositor, Hyprland, which leverages the Wlroots library from the protocol. The document acknowledges the challenges posed by the proprietary nature of Nvidia graphics drivers and the nascent state of the Wayland protocol, offering in-depth solutions and explanations for the issues encountered. Additionally, the document accentuates the philosophy of minimalism. It informs the readers about various alternative tools and methodologies to achieve a more streamlined and efficient use of the Gentoo Linux system. This includes guidance on establishing a modern compiler environment, managing a local package repository for modified ebuilds, and optimizing the Clang/Rust toolchain. Moreover, the script adeptly manages the intricacies of CPU microcode configuration; by also removing the problems that stem from bootloaders utilizing the EFISTUB method. The script demonstrates an exceptional capability in automating the build process of the Linux kernel, leveraging custom configuration files and employing optimization techniques such as Link Time Optimization. The script extends its functionality to user-centric configurations, such as the setup of user accounts and integration of custom dotfiles for environment personalization. This holistic approach ensures that the user not only receives a fully operational system but also one that is fine-tuned to their preferences.

Comprehensive Guide to Automating and Optimizing Gentoo Linux Installation and Configuration: In-Depth Analysis of Wayland (Hyprland) Integration with Nvidia Proprietary Graphics Drivers, Leveraging Advanced Bash Shell Scripting Techniques for a Streamlined and Exceptionally Minimalist System Setup

The significance of this guide lies in its comprehensive approach, bridging the gap between complex system configuration and user accessibility. By demystifying the intricate processes involved in customizing a Gentoo Linux system, this guide empowers users to take full control of their computing environment. The knowledge imparted herein is not only practical but also elevates the user proficiency in Linux system administration. It is designed to cater to both novice and experienced users who seek to enhance their understanding and skills in system setup using advanced Bash scripting techniques.

The script, ingeniously designed, automates not only the installation but also the intricate configuration and optimization processes. It effectively addresses the challenges inherent in integrating proprietary Nvidia graphics drivers with the Wayland protocol, particularly focusing on the Hyprland compositor. This aspect is crucial, given the proprietary nature of Nvidia drivers and the emerging status of Wayland, presenting unique challenges that this document thoroughly resolves.

Where technological advancements continuously redefine the parameters of system efficiency and functionality, the integration of Gentoo Linux with Wayland, specifically Hyprland, emerges as a significant development. This guide, focused on leveraging Bash shell scripting for optimizing this integration, particularly with Nvidia proprietary graphics drivers, marks a pivotal contribution to this evolving domain. The inherent complexity of Linux systems, combined with the nuanced requirements of modern hardware compatibility, necessitates a resource that not only guides but also educates.

The discourse aims to bridge the gap between theoretical knowledge and practical application. It is designed to cater to those seeking to deepen their understanding of Linux system configuration while providing actionable insights for optimizing and streamlining their setups. The script at the heart of this guide is not just a tool but a manifestation of best practices in system administration.

Vital Communiqué

It can not be stressed enough that this guide by itself does not contain any kind of recommendation. It aims to serve the people who already want to achieve the related tasks as it has biases towards anti-defaults. Therefore, this paper neither accepts criticism (except improvements) nor the responsibility. As a technical example, disabling default package useflags is not recommended but used in the guide. For orthodox methods, naturally, practicing upon *official documentation* and communicating with *official sources* would always be considered as the best practices. This paper is only valid due to the lack of documentation on unorthodox but anomalously prevailing topics as it also informs users with alternative ways in order to aim towards the maximum user freedom. The inspiration behind this is the *Gentoo Linux Philosophy* itself that was written by the creator of Gentoo Linux, Daniel Robbins:

"Every user has work they need to do. The goal of Gentoo is to design tools and systems that allow a user to do that work as pleasantly and efficiently as possible, as they see fit. Our tools should be a joy to use, and should help the user to appreciate the richness of the Linux and free software community, and the flexibility of free software. This is only possible when the tool is designed to reflect and transmit the will of the user, and leave the possibilities open as to the final form of the raw materials (the source code.) If the tool forces the user to do things a particular way, then the tool is working against, rather than for, the user. We have all experienced situations where tools seem to be imposing their respective wills on us. This is backwards, and contrary to the Gentoo philosophy.

Put another way, the Gentoo philosophy is to create better tools. When a tool is doing its job perfectly, you might not even be very aware of its presence, because it does not interfere and make its presence known, nor does it force you to interact with it when you don't want it to. The tool serves the user rather than the user serving the tool.

The goal of Gentoo is to strive to create near-ideal tools. Tools that can accommodate the needs of many different users all with divergent goals. Don't you love it when you find a tool that does exactly what you want to do? Doesn't it feel great? Our mission is to give that sensation to as many people as possible."

The Target Audience

- Gentoo Linux users who wish to migrate to a Wayland compositor from X11 with Nvidia graphics cards and Intel CPUs (default). It is possible to benefit from the guide as an AMD user with few modifications that will be mentioned.
- People who want to start using Gentoo Linux directly with Wayland and especially its Hyprland compositor. For them specifically, this guide offers a remarkable journey. The script, central to this guide, is an ingenious amalgamation of simplicity and technical prowess. It enables even those with a rudimentary understanding of Bash programming to construct a Gentoo Linux system effortlessly by also providing a centralized knowledge on Gentoo Linux.
- People who embrace challanges and want to take on a journey with unorthodox methods.
- People who want to learn Bash shell scripting further in a practical way in order to achieve their preferred tasks and automation skills.
- People who want to learn about newer, modern tools such as Wayland and Pipewire and their integration inside Gentoo Linux.
- People who want to learn more about the notorious Nvidia Proprietary Graphics Drivers and how to work with them on Linux with the least amount of problems.
- People who want to get inspired on minimalism by also learning about the alternative tools and methods.
- Individuals aspiring to effortlessly attain a fully functional setup, akin to those they marvel on the internet, without engaging in meticulous planning and dedicated execution. This involves not only replicating the allure and the aesthetic appeal but also ensuring the practicality and efficiency of the admired setups.
- People who does not know how but want to automate their Gentoo Linux installation to attain reproducible systems with all the modifications they need.

The Formatting & Style

The paper commences with a classical Gentoo Linux installation procedure, providing in-depth insights into the operating system, Wayland, Hyprland, Nvidia proprietary graphics drivers, and their interaction with the Linux kernel and the tools that are replaced or preferred. Throughout the guide, there is a strong emphasis on Bash scripting and command usage, ensuring each step is elaborately explained.

A unique aspect of this guide is its commitment to providing alternatives and individualized approaches where necessary, reflecting the diversity of user needs and preferences. It also highlights minimalist philosophies and offers insights into various tools and methodologies for a streamlined Gentoo Linux experience. The document concludes with a post-installation section containing additional information unrelated to the scripting process and a FAQ section to address common queries.

The guide begins with a traditional Gentoo Linux installation, meticulously detailing each step and the specific methodologies employed. Throughout the process, it not only outlines the specific commands but also delves into the structured logic behind their scripting. This approach is designed to equip users with an understanding of the rationale behind the sequence of tasks, as well as the methods for executing these tasks safely and efficiently in a Bash shell script.

Commands are presented within the text, each followed by targeted explanations that cater to both general Linux and Gentoo Linux specifics. These commands are not only detailed in terms of their immediate application but are also contextualized within the script's broader framework. Where necessary, alternative approaches are provided, offering users flexibility and adaptability in their usage.

While it is feasible to utilize the scripts directly without delving into the guide, engaging with the guide's content offers a profound enhancement in understanding. This deeper insight empowers users to tailor, refine, and optimize the script to align with their individual needs and preferences. However, due to the script's extensive scope and complexity, modifications that deviate from the defaults require reference to the actual documentation. This necessity underscores the guide's value as not just a tool for implementation but also as a resource for comprehensive understanding and customization.

Rationale Behind Selecting Bash as a Scripting Shell

Bash, an acronym for the Bourne Again SHell, stands as a predominant figure in the realm of shell scripting. The decision to utilize Bash is anchored in several fundamental factors which are discussed in detail below.

While *POSIX-compliant* shells, particularly Dash (Debian Almquist Shell), are often preferred for scripting due to their increased speed, security, compatibility, lightweight, simplicity and adherence to the POSIX standard, the decision to use Bash is grounded in several critical considerations. This paper still defends Dash as a /bin/sh replacement (system shell) and defends that especially user scripts should be written in Dash due to the aforementioned rationale.

Bash is presented both as a default interactive and scripting shell on Gentoo Linux. So the users can run the script without an external tool. Since Portage also utilizes Bash, there is virtually no way to remove it completely from the system. Though for scripting and interactive usage, there are arguably better shells such as Dash and Zsh respectively.

Bash provides a comprehensive *feature set* essential for complex scripting tasks, such as loop controls, conditional statements, functions, and arrays, as well as regular expressions and string manipulation capabilities. These features are crucial for the intricate process of installing and configuring Gentoo Linux.

Bash's interactive nature allows for a step-by-step execution of commands and scripts, fostering a more controlled approach to system installation and configuration. Additionally, Bash's debugging features are instrumental in identifying and resolving script errors.

Dash is particularly favored in scenarios where performance and speed are critical. Moreover, scripting in a POSIX-compliant shell encourages practices that promote greater script portability.

However, for the purpose of this guide, the advanced features and user-centric flexibility offered by Bash outweigh the performance advantages of Dash since the bottleneck is the tasks instead such as compiling a software. While acknowledging the merits of POSIX-compliant scripting, the decision to use Bash aligns with the goal of providing a comprehensive, user-friendly, and adaptable installation experience for Gentoo Linux users. These advanced features of it are mentioned in the guide.

First Part of the Script

The next step starts with the installation process of Gentoo Linux. First part of the script is rather basic compared to a complete shell script. The main script is the part two where it is run after chrooting. The script parts are provided as different files named `part1.sh and part2.sh`. A command is only explained profoundly when it is first used. Subsequent usages have superficial information.

Retrieving The Gentoo Tarball

```
URL_GENTOO_TARBALL="$(curl https://www.gentoo.org/downloads/ | sed -n 's/.*h
↪  ref="\([^"]*stage3-amd64-nomultilib-openrc-[^"]*\.tar\.xz\)".*/\1/p')"
```

This is a single line command. If the line has an arrow, then it means that the line is the subsequent part of the above line. It is not a new line and used in the paper to prevent overflowing of text.

A variable named `URL_GENTOO_TARBALL` is created.

`$()` part opens a new subshell and run commands into it. So, the result of the commands will be the variable.

`curl` command is used for sending HTTP requests. In this case, basically it gets the website data similarly to a plain text format. It is one of the most used command line tools. It can download or upload files, and simulate most web requests.

Pipe: `|`. Pipes are used to redirect a standard output of a command as a standard input to another program.

Website data is taken and is redirected to the `sed` (stream editor) command. sed is used for text processing and manipulation. Its `-n` option suppresses the output and makes it only print a string when the option `p` is given. It will only print the specific parts that are explicitly instructed.

`s/` signals the start of a substitution command in sed. It means "substitute."

`.*href="` is a regular expression pattern. It matches any character (.) zero or more times (*)

followed by exactly `href="`.

Regular expressions help to match characters that are in broader range. Instead of capturing ”word 123” exactly, the user can use: `[[:alpha:]]* [[:digit:]]*` to instruct the program to capture any number of alphabetical characters, a space and then any number of numeric characters.

`\( and \)` are used for capturing a portion of the matched text as a group. Whatever matches the pattern enclosed within parentheses will be saved for later use.

`[^\"]*` regular expression matches any sequence of characters that are not double-quote characters. This is typically used to capture the URL within the href attribute. Backslash is used to escape the double quotes. Otherwise double quotes can be considered as the part of the command.

`stage3-amd64-nomultilib-openrc-` is a specific part of the URL that the regular expression is looking for. It matches the string exactly. These are the features of the Gentoo Linux profile. Stage-3 tarballs are used for installing Gentoo Linux. amd64 shows the CPU architechture x64 and nomultilib shows that legacy 32bit code will not be included. So the system will be 64bit only. It should be noted that on nomultilib profiles, applications such as Steam and Wine do not work since they need 32bit libraries. Ultimately, openrc shows the default init system. The exact profile is the default Gentoo profile with nomultilib option. The main compiler is GCC; the C library is Glibc and the init system is OpenRC. These all can be changed according to the user preferences but the rest of the guide should be considered. As an example, CONFIG_IA32_EMULATION option should be enabled on the kernel in order to use a multilib profile instead of nomultilib.

After the above match, any other character except double quotes is matched again until the below part.

`\.tar\.xz` is the extension of the compressed tarball file. This will finalize the format of the URL.

`\)".*` captures everything that comes after the matched pattern.

`\1` is a reference to the first captured group, which is everything inside the \(and \) in the regular expression.

`/p` instructs sed to print the result of the substitution (the first captured group) if the pattern is successfully matched.

In summary, this command simulates the actions as the user goes to the Gentoo download page, navigate to the preferred profile, right click to the button and copy the link.

```
mkdir --parents /mnt/gentoo
```

`mkdir` in other words 'make directory' is used to create directories on UNIX. /mnt/gentoo is just a temporary directory that the preferred root partition is mounted. `--parents` option is used to create parent directories if not present (/mnt) in this case. It also suppresses the error if one of the directories is also present.

```
read -rp "Enter the Root Partition (e.g /dev/nvme0n1p2): " PARTITION_ROOT
```

The above part can also be automated but it is better to ask the user because they can have more than one fitting partition for this task.

`read` command in bash and other shell scripting languages is used to read input from the standard input (usually, the keyboard) and assign it to a variable. In this command, the variable that the input is being assigned to is `PARTITION_ROOT` variable.

The `-n` option prevents backslash escapes from being interpreted. Without this option, if the user typed a backslash, it would be treated as an escape character. For example, \n would be interpreted as a newline. With -r, backslashes are treated as literal characters.

`-p` option allows for a prompt string to be displayed to the user. This prompt is shown before the input is taken, guiding the user on what should be entered.

Root partition should have the `Linux Filesystem` type and be formatted as `ext4`. These are automatically checked in the second script and users are informed accordingly. Newer NVMe SSD users should also consider using the newer free & open source `F2FS` filesystem developed by Samsung. Note that this change requires a modification in the kernel configuration: The support for F2FS should be enabled. On Gentoo Linux, in order to format a disk partition with F2FS, the package named `sys-fs/f2fs-tools` is needed.

A Note on F2FS Filesystem

The F2FS is a file system specifically designed for NAND flash memory-based storage devices like SSDs.

F2FS is designed with the understanding that NAND flash memory does not allow in-place updates, unlike traditional spinning hard drives. F2FS employs a log-structured file system approach, which is well-suited for the write patterns of NAND flash. This approach minimizes write amplification, a critical factor in NAND flash memory, which in turn extends the lifespan of SSDs and maintains high performance over time.

It uses an append-only writing method for new data (also known as a log structure), which avoids the random write patterns that degrade SSD performance. This method is complemented by a cleaning (or garbage collection) process that reclaims space used by obsolete data, ensuring efficient space management.

It introduces a multi-head logging mechanism. It divides the log structure into multiple segments, allowing concurrent writing in different segments. This is particularly beneficial for SSDs, which offer high parallelism and bandwidth. By leveraging these capabilities, it can significantly improve file write performance.

It implements an adaptive read-ahead mechanism that dynamically adjusts the read-ahead size based on the access patterns. This feature is especially useful in exploiting the high sequential read speeds of SSDs, reducing latency and improving overall system responsiveness.

PCIe 5.0 NVMe SSDs are capable of handling large I/O requests efficiently. F2FS is designed to generate and handle large I/O requests, which is a good match for the capabilities of these SSDs, allowing them to operate at peak efficiency.

It uses a checkpointing mechanism that helps in reducing the amount of data that needs to be written and rewritten. This mechanism effectively reduces the wear and tear on the SSD, which is critical for the longevity and consistent performance of NAND flash-based storage devices.

It optimizes directory operations with features like hash-based directory indexing. This reduces the time taken for file lookups in large directories, which is beneficial for SSDs as they can handle such operations faster compared to traditional HDDs.

The design and optimizations present in F2FS make it highly suitable for PCIe 5.0 NVMe SSDs, which offer high throughput and low latency.

"The motive for F2FS was to build a file system that, from the start, takes into account the characteristics of NAND flash memory-based storage devices (such as solid-state disks, eMMC, and SD cards), which are widely used in computer systems ranging from mobile devices to servers" (*Wikipedia: F2FS*).

"It relies on FTL to handle write distribution. It is supported from kernel 3.8 onwards. An FTL is found in all flash memory with a SCSI/SATA/PCIe/NVMe interface, opposed to bare NAND Flash and SmartMediaCards" (*Arch Wiki: F2FS*).

"NAND flash memory-based storage devices, such as SSD, eMMC, and SD cards, have been equipped on a variety systems ranging from mobile to server systems. Since they are known to have different characteristics from the conventional rotating disks, a file system, an upper layer to the storage device, should adapt to the changes from the sketch in the design level. F2FS is a file system exploiting NAND flash memory-based storage devices, which is based on Log-structured File System (LFS). The design has been focused on addressing the fundamental issues in LFS, which are snowball effect of wandering tree and high cleaning overhead" (*Linux Kernel Docs: F2FS*).

"F2FS is a file system for flash-based storage devices that improves the performance using append-only logging."

"F2FS is a log-structured file system that writes data in a sequential manner by append-only logging. Thus, random writes do not introduce significant overheads in SSDs with F2FS" (*Sung et al., 2022*).

Continuation of the Guide

```
mount "$PARTITION_ROOT" /mnt/gentoo
```

`mount` command is used to mount a disk partition to a directory path. First argument is the partition taken from the above read command's variable and the second argument is the newly created /mnt/gentoo directory.

```
cd /mnt/gentoo
```

Change directory to the newly created directory.

```
wget "$URL_GENTOO_TARBALL"
```

`wget` is used to download files from the web. The URL taken from the first variable is given to it so it would download the Gentoo tarball.

```
tar xpvf stage3-*.tar.xz --xattrs-include='*.*' --numeric-owner
```

This command is critical. The stage-3 tarball basically has an installed Gentoo Linux system. In order to use this system, the files should be extracted properly.

`tar` stands for tape archive. This command is widely used on UNIX and is a standard utility for collecting many files into one archive file. This command does not provide compression by itself but it requires other programs to do it. That's why tar files generally have 2 extensions such as .tar.xz.

`-x` extracts files from an archive. This is the primary action the command is performing.

`-p` preserves permissions. This instructs tar to maintain the permission settings of the files as they were in the archive when extracting them. This is crucial for ensuring that the file permissions are correctly set, especially for executable files or when restoring a backup.

`-v` is for verbose output. This option makes tar list the names of the files as they are being extracted. It's useful for monitoring the progress of extraction or debugging.

`-f` specifies that the following argument is the filename of the archive. This is essential to tell tar which file to work with.

`stage3-*.tar.xz` is the name of the archive file(s) to be extracted. In this case, it is a pattern (indicated by *) to match any files that start with stage3- and end with .tar.xz. The .tar.xz extension indicates that the file is a tar archive compressed using xz compression, a highly efficient compression algorithm. Since the name of the file can change, a wildcard (*) is used so anything next to (-) is matched.

`--xattrs-include='*.*'` instructs tar to include extended attributes in the extraction process. Extended attributes (xattrs) are a feature of many filesystems that allow users to associate additional metadata with a file. The pattern is a wildcard to include all extended attributes.

`--numeric-owner` tells tar to use numeric user and group IDs. This is crucial for ensuring that the file ownership is preserved exactly as it was in the archive, regardless of the user and group names on the system where the files are being extracted. This is particularly important for system backups or when moving files between systems where user and group names might not match.

In summary, a stage-3 tarball means a very minimal Gentoo Linux installation allowing users to compile rest of the software they need. In very basic terms, first and second stages are for creating the compiler environment and they are more complex than the third stage. Therefore, Gentoo only provides the ready stage-3 tarball. The specific extraction instructions are important for the system to be functional by making sure the file attributes, permissions, metadata, and ownership are properly preserved.

```
rm -rf stage3-*.tar.xz
```

`rm` removes the files.

`-r` option means a recursive action. It generally used to remove directories so it is not vital here but there is no harm using it. If used on directories without this option, the user will get an error. It could be beneficial if the user mistakenly tries to delete a directory instead of a file.

`-f` means a forced action. It forcefully, in other words, non-interactively deletes the file. In shell scripts, if interactivity is not the case and automation is more important, then this option is used.

In summary this deletes the tarball that was already extracted and not needed anymore.

Initial Portage Configuration and Network Connection

```
mkdir --parents /mnt/gentoo/etc/portage/repos.conf
```

`/mnt/gentoo` directory is usually where the new Gentoo system is being set up (known as the root directory for the new installation).

`/etc/portage` directory is crucial for Portage, the package management system used by Gentoo. It contains configuration files for Portage.

`repos.conf` is a specific directory within etc/portage where repository configuration files are stored.

By creating the directory structure, a key component of the Gentoo system's package manager is set up, ensuring that it will function correctly once the installation is complete.

```
cp /mnt/gentoo/usr/share/portage/config/repos.conf
 ↪  /mnt/gentoo/etc/portage/repos.conf/gentoo.conf
```

`cp` is used to copy files and directories. The copied file (the first path) in this case is a default repository configuration file provided by the Portage package manager.

The second path is the destination where the files are copied to.

The file is renamed to `gentoo.conf` in the process, which is the standard name for the main Gentoo repository configuration file. If the destination path is non-existent, the copied file is renamed.

This step is crucial because it ensures that the new Gentoo system knows where to find and how to access the Gentoo software repositories. Without this, the package manager wouldn't be able to install or update software.

```
cp --dereference /etc/resolv.conf /mnt/gentoo/etc/
```

`--dereference` instructs cp to follow symbolic links, meaning that if the source file is a symbolic link, it will copy the file it points to, not the link itself. Symbolic links are basically copies of files but they do not waste a space. They rather redirect to the original file that is on another path.

The copied file is used by the system resolver to configure DNS servers and is essential for network connectivity.

Mounting the Filesystems

```
mount --types proc /proc /mnt/gentoo/proc
mount --rbind /sys /mnt/gentoo/sys
mount --make-rslave /mnt/gentoo/sys
mount --rbind /dev /mnt/gentoo/dev
mount --make-rslave /mnt/gentoo/dev
mount --bind /run /mnt/gentoo/run
mount --make-slave /mnt/gentoo/run
```

`--types or -t` is an option that specifies the type of the filesystem being mounted.

`proc` is the type of filesystem being specified. The proc filesystem is a virtual filesystem that provides access to kernel-related information, typically used for system information and configuration.

`/proc` is the source of the mount, which is the proc filesystem.

`/mnt/gentoo/proc` is the target directory where the proc filesystem will be mounted. It's within the directory structure of the new Gentoo installation.

The proc filesystem is a special filesystem in Linux that provides a view into the kernel data structures. It doesn't contain real files but runtime system information (e.g., system memory, hardware configuration, etc.). For instance, /proc/cpuinfo provides information about the CPU.

Mounting `/proc` to `/mnt/gentoo/proc` makes the proc filesystem available inside this chroot environment. This is important because many installation and configuration processes depend on accessing the proc filesystem to gather information about the system.

Without mounting /proc, many tools and scripts that rely on this data wouldn't function correctly, potentially hindering the installation and configuration of the system.

`--rbind` not only mounts the specified directory but also all of its submounts to the target directory. This is different from the simpler `--bind` option, which does not include submounts.

`/sys` is the source of the mount, referring to the sysfs filesystem.

`sysfs` is a virtual filesystem in Linux that provides a lot of information about the system and its hardware, such as device nodes, kernel modules, and more.

`/mnt/gentoo/sys` is the target directory where the `sys` filesystem will be mounted, situated within the Gentoo installation environment.

The use of `--rbind` ensures that not only the `sys` filesystem but also all its submounts (like /sys/devices, /sys/block, etc.) are available in the chroot environment.

Normally, the 'slave' commands can be removed as they are only needed for systemd environments. However it is harmless to preserve them in order to have successful and proper chroot environment.

This option modifies the nature of the mount to become a "slave". The "rslave" part stands for "recursive slave". It ensures that any mount and unmount events in the original mount point (for example `/sys` in the main system) are propagated to the slave (`/mnt/gentoo/sys`), but not the other way around.

The `/dev` directory in Unix systems, including Linux, contains device files that represent and provide access to the hardware devices of the system.

The `/dev` directory is crucial because it contains the device nodes or files through which the operating system and installed programs interact with hardware devices. These device files include everything from hard drives (`/dev/sda`) to audio interfaces (`/dev/snd`), and more.

This command is crucial in ensuring that the Gentoo installation environment has full and comprehensive access to all the hardware devices of the system through the device files. This access is essential for a wide range of tasks during the installation process, from disk partitioning to hardware configuration

`--bind` option creates a new mount point for an existing directory. The filesystem content at this mount point is identical to the original directory's content.

The `/run` directory contains important runtime data that is relevant to the current boot of the system. This can include information like daemon sockets, process IDs, user sessions, and other temporary runtime data that doesn't persist across reboots.

Extra Information on Chroot: 'Chroot' stands for change root, is a Unix system operation that changes the apparent root directory for the current running process and its children. This creates an isolated environment separate from the main system. This isolated environment is often referred to as a "chroot jail".

Chroot changes the root directory (/) for a process to a specified directory. This means that the process operating within the chroot environment can only see and interact with files within that directory tree. It effectively isolates the process from the rest of the system.

Initially, chroot was intended to test and debug system setups or run potentially risky operations without affecting the main system. It's also used to run services with reduced privileges to increase security. However, it should not be solely relied upon as a security feature because it has limitations and can be bypassed by processes with sufficient privileges.

chroot is widely used for software installation, maintenance, and upgrades, particularly in Linux distributions. For example, it allows administrators to repair a damaged system by booting from a live CD/DVD or USB, mounting the hard disk, and entering a chroot environment to make repairs.

chroot is simpler than more modern virtualization or containerization technologies but also more limited. It doesn't provide full isolation as modern containers do. Processes in a chroot jail still share the same kernel and, depending on the setup, other system resources like the process table, network interfaces, etc.

Specifically for Gentoo Installation: A typical use case in Linux installations (like Gentoo) involves mounting the new system's partitions under a directory on the live system, then using chroot to switch into the new system. This allows the administrator or installer to perform setup tasks as if they were running on the new system itself, even though it hasn't been booted yet.

Chrooting to the New Environment

```
chroot /mnt/gentoo /bin/bash
```

This command creates an isolated environment, separate from the main operating system. It's similar to stepping into a different system without the need to reboot (from the installation media to the new system).

`/mnt/gentoo` specifies the new root directory. In the context of Gentoo installation, `/mnt/gentoo` is where the user sets up the new system. By setting this as the root directory, users make the system treat `/mnt/gentoo` as the root (`/`) of the file system tree. Therefore, specifically `/mnt/gentoo` is not used after this point.

`/bin/bash` shows the location of the shell. Bash is used as the login shell.

Summary:

- Extracting the appropriate and up-to-date tarball URL is automated.
- The tarball is properly extracted.
- Initial filesystems are mounted.
- The process of chrooting has finished.

The Second and the Main Part of the Script

This part includes the actual complete shell script where every process is automated.

The script is designed with modularity in mind. Every distinct task forms a function. All functions are called in a specific order in the main function and all functions have their starting and finishing logs. These will be explained at the end as the guide continues line by line.

Defining URLs & Directory Paths

The second part and the actual script starts by defining URLs. These URLs are used to download their corresponding files. These include files such as the kernel configuration, portage related files, dotfiles (user specific configuration files). These files are still manipulated in the script such as the kernel configuration is changed automatically for the specific user according to their CPU microcode, partition IDs and other similar variables. These URLs can be changed by the user by preference but in order for a successful installation, the user should refer to the guide as to what parts are essential. For a working example, all of the URLs are provided as a default.

```
URL_DOTFILES="https://github.com/emrakyz/dotfiles.git"
URL_DEPENDENCIES_TXT="https://raw.githubusercontent.com/emrakyz/dotfiles/mai⌋
 ↪  n/dependencies.txt"
# ... continues with other URLs.
```

The above URLs are two of the examples from the list. The first link is a `.git` link meaning it's a git repository and it's a directory. This can not be downloaded with `curl`. This requires a specific command `git clone`. The second link is for a singular plain text file that can be downloaded directly with `curl`. If the text file is a configuration file or if it should be directly read, then a URL corresponding to a raw text file should be used. Otherwise the command curl can also download other parts of a website (such as html codes).

`URL_DEPENDENCIES_TXT` is a file that the Gentoo Packages that will be installed are listed. This file includes a package on each line. With this way, packages can be grouped and easily maintained later. Comments can explain the group of packages and an empty line between groups keeps the separation clearly. Portage can't use this format so this is changed later automatically by text manipulation tools (empty lines are removed, new lines are replaced with spaces, comments are removed).

```
# Hyprland Related
gui-wm/hyprland::local
x11-terms/kitty

# Multimedia
media-gfx/imv
media-video/mpv
```

We maintain the files as seen above but they are reformatted automatically for installation as seen below. Bash shell will read the file and feed the emerge command with every package name. This will be explained later:

```
emerge gui-wm/hyprland::local x11-terms/kitty media-gfx/imv media-video/mpv
```

```
FILES_DIR="/root/files"
PORTAGE_DIR="/etc/portage"
LINUX_DIR="/usr/src/linux"
# others...
```

Essential directory path variables are shown. `FILES_DIR` is the temporary directory that is used to store the downloaded files. `/root` is the HOME directory for the root (admin) user. The specific files are explained further in the guide.

Error Handling and Logging

```
# Fail Fast & Fail Safe on errors and stop.
set -Eeo pipefail
```

The `set` command is used in Unix shells to set and unset certain flags and positional parameters, or to display the names and values of shell variables.

`-E` option ensures that any ERR trap is inherited by shell functions, command substitutions, and commands executed in a subshell environment. The ERR trap is a mechanism that allows a script to execute a specified command whenever a command fails.

`-e` option makes the shell exit immediately if a command exits with a non-zero status (which usually indicates an error). This option is widely used in scripts to make sure that errors are not silently ignored.

The `-o` option is used to set various other shell options. `pipefail` is one of these options, and it changes the exit code behavior in pipelines. Without pipefail, a pipeline only returns the exit status of the last command. With pipefail enabled, the pipeline's return status is the exit status of the last command to exit with a non-zero status, or zero if all commands exit successfully.

These settings make script execution more robust and secure. The shell immediately exits on errors, which prevents scripts from continuing when something goes wrong, potentially leading to cascading failures or incorrect results.

In the context of Gentoo Linux installation or system configuration, these settings are important for scripting reliability. They help ensure that any scripts or commands executed do not silently fail, which is crucial in a complex and highly customizable environment like Gentoo. Failing fast and visibly in response to errors can prevent more significant problems down the line and is a key aspect of writing reliable and maintainable scripts.

```
# Use HIGHLIGHTED-BOLD color variants for the logs.
GREEN='\e[1;92m' RED='\e[1;91m' BLUE='\e[1;94m'
PURPLE='\e[1;95m' YELLOW='\e[1;93m' NC='\033[0m'
CYAN='\e[1;96m' WHITE='\e[1;97m'
```

These are *ANSI Escape Sequences* for highlighted-bold color variants. They are hard to memorize so it's better to define the colors as variables in order to use later. On terminals, colors can only be printed with these escape codes. After the colorized part, a resetting color ('NC' in this case) is needed to return to the normal foreground color.

```
log_info() {
    sleep 0.3

    case $1 in
        g) COLOR=$GREEN MESSAGE='DONE!' ;;
        r) COLOR=$RED MESSAGE='WARNING!' ;;
        b) COLOR=$BLUE MESSAGE="STARTING..." ;;
        c) COLOR=$BLUE MESSAGE="RUNNING..." ;;
    esac

    COLORED_TASK_INFO="${WHITE}(${CYAN}${TASK_NUMBER}${PURPLE}/${CYAN}${TOTA⌋
     ↪  L_TASKS}${WHITE})"
    MESSAGE_WITHOUT_TASK_NUMBER="$2"

    FULL_LOG="${CYAN}[${PURPLE}$(date '+%Y-%m-%d '${CYAN}'/'${PURPLE}'
     ↪  %H:%M:%S')${CYAN}] ${YELLOW}>>>${COLOR}$MESSAGE${YELLOW}<<<
     ↪  $COLORED_TASK_INFO - ${COLOR}$MESSAGE_WITHOUT_TASK_NUMBER${NC}"

    { [[ $1 = c ]] && echo -e "\n\n$FULL_LOG"; } || echo -e "$FULL_LOG"
}
```

The above function is used throughout the script to create logs and inform users according to the specific segments. It is a very basic function despite the intimidating appearance. It looks complex because of the frequent usage of different colors. At its core, the format can be understood easily. Any text after `$GREEN` will be printed as green until a new color is given.

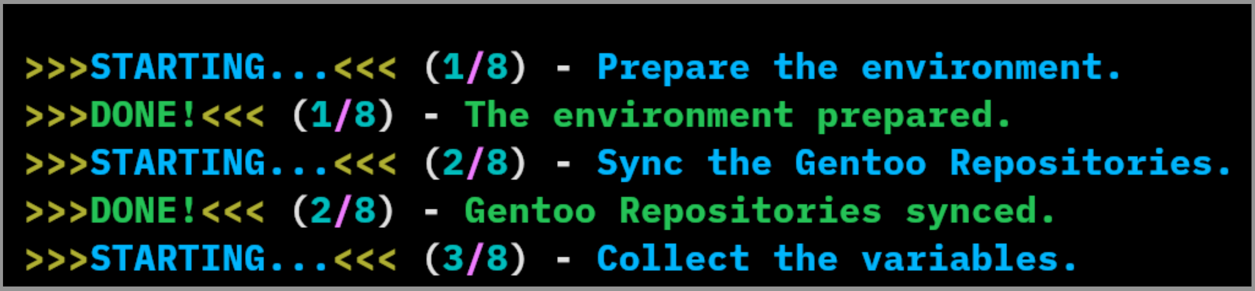
`$GREENthis is an example$NC` makes the text green and then return to the NC (no color).

A common mistake of beginners is that they expect to read the code as they read normal text. These kinds of texts (such as colored sequences, regular expressions and similar code…) are read and written very carefully and slowly even by experienced people. It can be understood easily when you focus on each character and its specific purpose separately.

The function has a `sleep` command at the first line. It makes the `log_info` command wait for 0.3 seconds between each prompt ensuring logs don't appear at the same time if the tasks are too quick to finish.

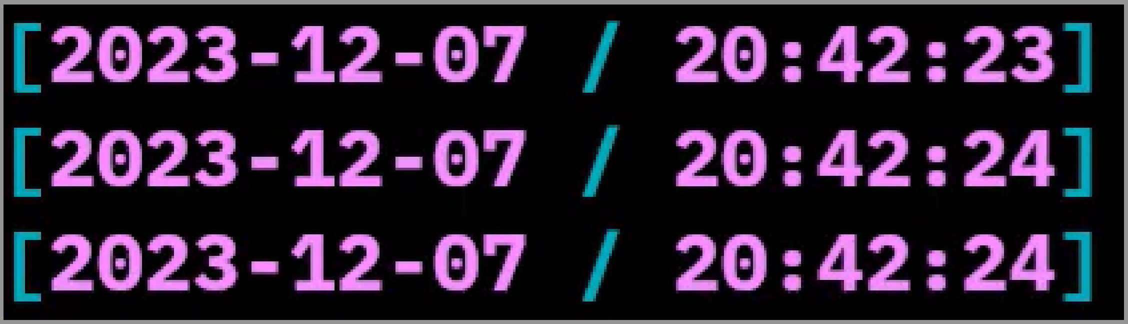
Then a case condition is used. `$1` in this case refers to the first argument given to the function (log_info). In the case of `log_info r`, the `$1` is `r`. In this situation, the case condition, continues with the `r` case. `r` represents the red color and a warning message here.

For indicating the initial messages for tasks, b (blue) is used. When a certain task is successfully finished, g (green) is used. When the task takes considerably long, such as compiling programs, then a log appears every minute with a message "Running:".



```
>>>STARTING...<<< (1/8) - Prepare the environment.
>>>DONE!<<< (1/8) - The environment prepared.
>>>STARTING...<<< (2/8) - Sync the Gentoo Repositories.
>>>DONE!<<< (2/8) - Gentoo Repositories synced.
>>>STARTING...<<< (3/8) - Collect the variables.
```

The above is an example of how logs are structured in the script.



```
[2023-12-07 / 20:42:23]
[2023-12-07 / 20:42:24]
[2023-12-07 / 20:42:24]
```

All of the logs do have date and time prefixes. The date and time formatting use *ISO 8601* standards. Date and time values are ordered from the largest to smallest unit of time, meaning: `Year-Month-Day / Hour:Minute:Second`.

`COLORED_TASK_INFO` variable sets the coloring of the task info. Task info in this case is the number of finished tasks and the total number of tasks. `(1/40)` means that one task is finished and there are 39 remaining tasks.

`MESSAGE_WITHOUT_TASK_NUMBER="$2"` is the part after the task numbers. `$2` refers to the second argument for the function which in this case is the specific log.

`log_info b "An example task."`

In the above case, 'b' is `$1` and 'An Example Task' is `$2`.

`$2` could also be directly used in the FULL_LOG variable but with this way it is easier to understand what is `$2` for the readers.

`FULL_LOG` variable includes every part of the log including a date at the left side of the logs.

`date` command is used to get the current date in Unix.

`%Y-%m-%d / %H:%M:%S` refers to Year, Month, Day, Hour, Minute and Second respectively.

`2023-12-31 / 08:30:25` can be an example.

```
{ [[ $1 = c ]] && echo -e "\n\n$FULL_LOG"; } || echo -e "$FULL_LOG"
```

This line uses logical conditional operators. The curly brackets are used for command grouping. If the command is written in a single line, then a ';' before the finishing curly bracket is used.

The other brackets are used in Bash for test conditions. In this case, it is tested if the first argument (`$1`) is `c`.

`&&` is used in conditional tasks on almost all shells. It means: Only continue if the command before succeeds. In the example situation, if the first argument is `c`; meaning the test succeeds; then continue with the next command.

The command `echo` is used to print text. Its `-e` option enables the interpretation of backslash escapes. It should be noted that shells can have different behaviour for echo. On Dash, backslashes are enabled by default. Backslash escapes are sequences that are used to represent certain special characters. For example, \n represents a new line, \t represents a tab, etc.

If the log is for the current task, then two new lines are added before showing the log in order to separate the normal terminal output and the log output. This means there will be an empty line between the current terminal output and the log output.

As an example, building Clang/Rust Toolchain can be an example. Since this task takes long to finish, it is better to notify users regarding which stage is the script at.

`||` is another logical control operator used in Unix shells. It means: Only continue if the command before fails. In the example situation, if the first argument is not `c` meaning the test fails; then continue with the next command placed after `||`. The next command in this case is the same echo command excluding the new lines.

```
handle_error () {
    error_status=$?
    command_line=${BASH_COMMAND}
    error_line=${BASH_LINENO[0]}
    log_info r "Error on line ${BLUE}$error_line${RED}: command
     ↪  ${BLUE}'${command_line}'${RED} exited with status:
     ↪  ${BLUE}$error_status" |
    tee -a error_log.txt
}
```

```
trap 'handle_error' ERR
trap 'handle_error' RETURN

cleanup_logs() {
    [ -f "logfile.txt" ] && sed -i 's/\x1b\[[0-9;]*m//g; s/\r//g' logfile.txt
    sed -i 's/\x1b\[[0-9;]*m//g' error.log.txt
}

trap cleanup_logs EXIT
```

`handle_error` is a function intended to be triggered when an error occurs. It captures details about the error and logs them.

`error_status=$?` retrieves the exit status of the most recently executed command. In Bash, `$?` is a special parameter that holds the exit status of the last command executed.

`BASH_COMMAND` is an internal Bash variable that contains the current command or function call. This line stores the command that triggered the error.

`BASHL_LINENO` is an array variable. It contains the line numbers where each command in the FUNCNAME stack was invoked. `BASH_LINENO[0]` gives the line number of the script where the error occurred.

The variables `error_line, command_line, and error_status` are used to create a detailed error message. The use of BLUE and RED are for colored output in the terminal for readability.

The command `tee` is used for redirecting the output to different sources at the same time. `-a` option means append. In this example, it outputs the error log both on the terminal and on an external file for preserving the logs, in case of a problem.

`trap` command basically sets a trap. It accepts types of signals (in this case ERR, RETURN and EXIT). It detects error and exit conditions. RETURN is also added to detect errors inside functions to increase the robustness. Ultimately, these functions are only called when the script fails or exits.

`cleanup_logs` function is used for cleaning the external log files. ANSI Escape Sequences, when redirected to a standard text file, are seen as they are; instead of changing the colors. Therefore, these sequences are removed.

`-i` option for `sed` means inplace editing. Normally sed only modifies the output in the terminal but with its 'inplace' option (which is a GNU Sed exclusive option), it can also edit files permanently. The regular expressions basically captures any kind of escape sequences and *carriage returns* (\r), then remove them.

Normally a user should do a research on ANSI Escape Sequences and how they use hexadecimal color codes and use them appropriately. Here the code looks complex but it just requires an extra investigation. The command otherwise can be simplified as below:

```
sed -i 's/regex/replacement/g; s/regex/replacement/g'
```

`;` is used for separation of different substitutions. If the substitution commands are written on different lines then it is not needed is used for separation of different substitutions. If the substitution commands are written on different lines then it is not needed.

`g` means global. Therefore, the command globally (meaning on every instance) applies the change.

Initial Steps & Preparing the Environment

```
prepare_env() {
    source /etc/profile
    export PS1="(chroot) ${PS1}"
}
```

`/etc/profile` is a global configuration script that is executed for login shells. It sets up an environment upon login and applies system-wide environment settings. This file contains important environment settings such as path variables, shell options, and other variables required for the proper functioning of the system.

Due to the recent chrooting process, the new environment should be prepared. Otherwise, the current working shell does not know about the current environment. This is automatic for the normal boot process.

The second command is for changing the terminal prompt. The command `export` works to save variables during the login. The command changes the variable named `PS1` which is used to modify the look of the prompt. It adds `(chroot)` before the normal prompt (`$PS1`). This is a good approach in order to notify users that they are on a chrooted environment.

Collecting Critical Variables

The next function is `collect_variables`. The function and the one after it are the only interactive parts in the script. Most of these commands (read, echo, conditional tests) were explained before. Therefore, the explanation will be brief.

```
echo -e "${WHITE}Enter the Timezone (e.g. Europe/Berlin):${YELLOW}"
read -rp "" TIME_ZONE

echo -e "${WHITE}Enter your GPU (e.g nvidia):${YELLOW}"
read -rp "" GPU

echo -e "${WHITE}Do you have another partition you want to mount at boot?
↪  (y,n):${YELLOW}"
read -rp "" EXTERNAL_HDD
```

The user is asked some basic questions. The reason there is no flag for the read command (`""`), is that the question is asked with echo, then the read command will prompt an empty line in order for users to write their answers. The question could also be placed inside the double quotes (thus removing the echo commands) but then it is harder to colorize it. This method makes it clean in terms of colorizing the output and separation of questions and answers. The questions are colorized with highlighted-bold white and the answers are yellow. The variables are collected under `TIME_ZONE`, `GPU` and `EXTERNAL_HDD` names.

```
[[ "$EXTERNAL_HDD" =~ [Yy](es)? ]] && {
    echo -e "${WHITE}Which partition is it? (e.g /dev/sda1):${YELLOW}"
    read -rp "" PARTITION_EXTERNAL
    echo -e "${WHITE}What's the format of that partition? (e.g
     ↪  ext4):${YELLOW}"
    read -rp "" FORMAT_EXTERNAL
    echo -e "${WHITE}What's the path to mount? (e.g /mnt/harddisk):${YELLOW}"
    read -rp "" MOUNT_DIR
    echo -e "${NC}"
    mkdir -p "$MOUNT_DIR"
} || log_info b "No extra partitions specified. Skipping..."
```

If the user said yes for the external hard drive question, then the next command block runs. These questions are optional and varying, therefore they need to be interactive and can not be automated. The above regular expression makes the e or s letters optional so the user can answer the hard drive question with y, ye, yES or any similar combination.

If the user said no to that question; then `||` logical control operator is valid due to the faiulure of the prior command. It runs the next logging command.

```
PARTITION_ROOT="$(findmnt -n -o SOURCE /)"
PARTITION_BOOT="$(lsblk -nlo NAME,RM,PARTTYPE |
    sed -n '/0\s*c12a7328-f81f-11d2-ba4b-00a0c93ec93b/ {s/^[^ ]*/\/dev\/&/;
     ↪  s/ .*//p; q}')"
```

The script continues with creating more variables to be used later. It tries to find the boot and root partition in a sophisticated manner.

`findmnt` command works for finding mount points. Its `-n` option removes the header lines since it is not needed. `-o` option specifies the output instead of using the default. In this example it only looks for the SOURCE for the root (`/`) partition. This command will output a partition name such as `/dev/nvme0n1p2` for NVME SSDs and `/dev/sda2` for SATA drives.

The other variable is more complex. It looks for an EFI System Partition in the disk by its specific code (starting with c and ending with b).

`lsblk` is a command that lists information about all available or the specified block devices. Similarly its `-n` option suppresses the header line. Since this command gives a tree-style output as default; it needs to be changed because those symbols can not be used in a variable that only accepts alphanumeric characters. `-l` option changes the tree-style output with list style. Similar to findmnt command, `-o` option specifies the output columns. It needs to look for the NAME of the device, its removable status, meaning if it is a removable drive (RM) and its parittion type (PARTTYPE). NAME output is similar to `nvme0n1p1`. RM output is either `0` or `1` indicating removable status. PARTTYPE output is a code specifying the partition type.

`sed` command specifically searches for `0` (non-removable partitions) and any kind of whitespace `(\s*)` in any amount. Then it looks for the specific code for EFI System Type.

Then a substitution block begins. This should be investigated separately.

```
's|^[^ ]*|/dev/&|; s| .*||p; q'
```

`^` symbol has a different meaning inside and outside of the brackets. While inside, it means 'not'. In this case, it means 'not space'. Outside of the brackets, it means: 'at the start of the line'. Collectively, it searches any non-space character at the start of the line in any amount then starts the replacement with (`|`). Normally the delimiter is a forward slash for `sed` but the replacement here includes forward slashes, therefore it's better to use another symbol for separation. Forward slashes can still be used as a separator but then normal forward slashes need to be escaped with a backslash. In order to remove the necessity to use backslashes, an alternative delimiter is used.

After the above match it substitutes everything with `/dev/&`. Sed uses & for captured patterns. In this case a string pattern is captured then replaced with `/dev/` and then the captured pattern is added next to it again. Here below is a practical example:

`nvme0n1p1` is captured.

It is replaced with `/dev/`

The captured pattern is added again: `/dev/nvme0n1p1`

Since this string is needed for the mounting of the boot partition, the `/dev/` prefix is required. That's why it's added.

Other parts of the output is not needed, therefore they are removed with the second substitution command. ` .*||` matches a space then any character (.) in any amounts (*). All of them are replaced with nothing, thereby removing them.

Ultimately `p` option prints the line and `q` option quits.

```
UUID_ROOT="$(blkid -s UUID -o value "$PARTITION_ROOT")"
UUID_BOOT="$(blkid -s UUID -o value "$PARTITION_BOOT")"
PARTUUID_ROOT="$(blkid -s PARTUUID -o value "$PARTITION_ROOT")"
[[ "$EXTERNAL_HDD" =~ [Yy](es)? ]] && EXTERNAL_UUID="$(blkid -s UUID -o
↪  value "$PARTITION_EXTERNAL")" || true
```

`blkid` is another command used to get certain device IDs. It stands for "block ID". UUID addresses for ROOT, BOOT and EXTERNAL_HDD partitions are required in the filesystem tab. PARTUUIDROOT is required in the kernel configuration.

The `-s` option tells blkid to only show the output of a specific tag.

`-o value` is another option. In this case, value means that only the value of the requested tag (e.g. PARTUUID) will be printed, without any additional information or formatting.

These kinds of text manipulations are useful when you need the raw value for scripting or programming purposes.

`|| true` is used to suppress errors. Since the error mechanism is enabled in the script, if the user did not specify any external hard drives, the script may exit with an error at this point. The `true` command is needed in order to remove errors if the user does not have an extra partition.

Checking the Variables

`check_first_vars` function works for checking all of the collected variables up until this point. It's a relatively bigger function so it will be analyzed in parts.

```
lsblk "$PARTITION_BOOT" > /dev/null 2>&1 || {
    log_info r "Partition $PARTITION_BOOT does not exist."; exit 1; }
```

`>/dev/null 2>&1` is a form of redirection in Bash. It redirects the standard output (stdout) to /dev/null, effectively discarding any standard output. `2>&1` redirects the standard error (stderr) to where stdout is currently going, which is /dev/null in this case. This means both standard output and standard error are being discarded. This is often done to suppress output from a command when the exit status is the only important information.

In this case, if `lsblk` command does not give any output then the logging command runs.

```
DISK="${PARTITION_BOOT%[0-9]*}"
DISK="${DISK%p}"
fdisk -l "$DISK" | grep -q "Disklabel type: gpt" || {
    log_info r "Your disk device is not 'GPT labeled'. Exit chroot first."
    log_info r "Use fdisk on your device without its partition:
     ↪  '/dev/nvme0n1'"
    log_info r "Delete every partition by typing 'd' and 'enter' first."
    log_info r "Type g (lower-cased) and enter to create a GPT label."
    log_info r "Then create 2 partitions for boot and root by typing 'n'."
    exit 1; }
```

The function starts with two parameter expansions. It aims to find the DISK using one of its partitions (boot partition in this case).

The first expansion removes any digits `[0-9]` in any number of time from the end of the string.

The second expansion also removes the `p` at the end if present.

As an example, `/dev/nvme0n1p1` becomes `/dev/nvme0n1p`.

Ultimately, the partition related part is removed to get the disk: `/dev/nvme0n1`

`fdisk` command is used with its `-l` option for listing the output as text.

`grep` command is used for searching text patterns. In this case, it is invoked with its `-q`, in other words, quiet option. Quiet option is used on scripts. The user does not have to see the output but the script should know if that pattern matches anything or not. Therefore `grep -q` is used as a conditional test statement in scripts. In this case, it checks if the `fdisk` output shows the disklabel as GPT. If not, the check fails and all other logging commands are invoked.

UEFI requires a partition with 'EFI System' type and the disk should be 'GPT labeled'. The EFI System partition should be formatted with 'FAT32' file system.

Disk operations are critical. That is why the script does not automate the handling of disk partitions.

```
BOOT_PART_TYPE=$(lsblk -nlo PARTTYPE $PARTITION_BOOT)
[ "$BOOT_PART_TYPE" = "c12a7328-f81f-11d2-ba4b-00a0c93ec93b" ] || {
    log_info r "The boot partition does not have 'EFI System' type."
    log_info r "Use fdisk on your device '/dev/nvme0n1' without its
     ↪  partition."
    log_info r "Type 't' and enter. Select the related partition. Then make
     ↪  it 'EFI System'."
    exit 1; }
```

This code snippet, similarly checks if the BOOT partition's type is 'EFI System'. If not, informs the users with logs and exits the script. The specific code starting with c and ending with b is a constant code for EFI System type. This can be confirmed on *Arch Wiki*.

```
BOOT_FS_TYPE=$(blkid -o value -s TYPE $PARTITION_BOOT)
[ "$BOOT_FS_TYPE" = "vfat" ] || {
    log_info r "The boot partition should be formatted as 'vfat FAT32'."
    log_info r "Use 'mkfs.vfat -F 32 /dev/<your-partition>'."
    log_info r "You need 'sys-fs/dosfstools' for this operation."
    exit 1; }
```

The same partition is checked if it is formatted as vfat FAT32. If not, users will be instructed with logs.

```
ROOT_PART_TYPE=$(lsblk -nlo PARTTYPE $PARTITION_ROOT)
[ "$ROOT_PART_TYPE" = "0fc63daf-8483-4772-8e79-3d69d8477de4" ] || {
    log_info r "The root partition does not have 'Linux Filesystem' type."
    log_info r "Use fdisk on your device '/dev/nvme0n1' without its
    ↪  partition."
    log_info r "Type 't' and enter. Select the related partition. Then make
    ↪  it 'Linux Filesystem'."
    exit 1; }
```

Now the root partition is checked. The constant code this time refers to 'Linux Filesystem' type. If the check can not find the specific code for Linux Filesytem; then the user gets informed. The 'Linux Filesystem' code can be found here on *Arch Wiki*.

```
ROOT_FS_TYPE=$(blkid -o value -s TYPE $PARTITION_ROOT)
[ "$ROOT_FS_TYPE" = "ext4" ] || {
    log_info r "The root partition is not formatted with 'ext4'."
    log_info r "Use 'mkfs.ext4 /dev/<your-partition>'."
    exit 1; }
```

The partition is checked again. The root partition should have the 'ext4' type.

```
TZ_FILE="/usr/share/zoneinfo/${TIME_ZONE}"
[ -f "$TZ_FILE" ] || {
    log_info r "The timezone $TIME_ZONE is invalid or does not exist."; exit
    ↪  1; }
```

A variable named `TZ_FILE` is created. It uses the timezone variable that had been asked from the user. This directory includes time zones.

`-f` check in Bash controls if the variable is a file or not.

If the timezone given is not appropriate. This check would fail, running the relevant log after it.

```
{ [ -n "$UUID_ROOT" ] && [ -n "$UUID_BOOT" ] && [ -n "$PARTUUID_ROOT" ]
} || { log_info r "Critical partition information is missing."; exit 1; }
```

This snippet checks all critical information if they are present.

`-n` check on Bash controls if the variable is empty (`0`) or not.

```
valid_gpus="via v3d vc4 virgl vesa ast mga qxl i965 r600 i915 r200 r100 r300
↪  lima omap r128 radeon geode vivante nvidia fbdev dummy intel vmware glint
↪  tegra d3d12 exynos amdgpu nouveau radeonsi virtualbox panfrost lavapipe
↪  freedreno siliconmotion"
```

Valid GPU names are listed here. Other than Nvidia is not handled thoroughly in the script since most of the GPUs here have graphics drivers included in the kernel or have open source drivers, requiring simpler work. If needed, the script and the guide can be improved further to include necessary operations for the other selected graphics cards.

```
{ [[ "$valid_gpus" =~ $GPU ]] && [[ ! "$GPU" =~ [[:space:]] ]]
} || { log_info r "Invalid GPU. Please enter a valid GPU."; exit 1; }
```

If the user mistakenly gives a wrong answer to the GPU question, then the script outputs the relevant log and exits.

The above command checks if the GPU given is included in the list and does not contain spaces.

Collect Account Credentials

This step is vital to determine the username and its password. Temporarily the root account also uses the same password with the main user. This can be changed by the user later since it is not vital in the installation process. Or, the user can extend the script by asking the root user password separately, if they think it is important for them.

The `collect_credentials` function asks for these variables.

```
echo -e "${WHITE}Enter the Username:${YELLOW}"
read -rp "" USERNAME
echo -e "${WHITE}Enter the Password:${YELLOW}"
read -rsp "" PASSWORD
echo -e "${WHITE}Confirm the Password:${YELLOW}"
read -rsp "" PASSWORD2

echo ""
echo -e "${NC}"
```

The newly used `-s` option for the `read` command is for hiding the output. This is just for authenticity, since the installation process is probably done alone by the user and it has not require security precautions.

Then an `echo` command with `""` creates an empty line and then the color is resetted again.

Checking/Controlling Account Credentials

```
[[ "$USERNAME" =~ ^[a-zA-Z0-9_-]+$ ]] || {
log_info r "Invalid username. Only alphanumeric characters, underscores, and
↪  dashes are allowed."; exit 1; }

{ [ "$PASSWORD" = "$PASSWORD2" ] && [ -n "$PASSWORD" ]
} || { log_info r "Passwords do not match or are empty."; exit 1; }
```

`check_credentials` function checks the username and password if they are appropriate.

The username should only include alphanumeric characters, underscores and dashes. Otherwise, the related error log appears and the script exits.

The second check, controls if the two given passwords match and if the password variable is empty.

Synchronize the Gentoo Repositories

Gentoo repositories need to be synced in order to update the system. The repositories include the necessary information and files for Portage that have the appropriate instructions for building packages.

```
emerge --sync --quiet
emerge dev-vcs/git
```

`emerge` is the main command the package manager of Gentoo Linux, Portage uses.

`--sync` synchronize the repositories using a method called rsync (that will be changed later in the guide with git-sync which is faster).

`--quiet` option removes the unnecessary information from the output during the synchronization process. Most of the time, this output is not necessary for the user and if an error occurs, they get notified anyway.

`emerge dev-vcs/git` command installs the source files for git, its dependencies and builds them. Git is necessary for downloading some of the required files (defined as URLs at the start of the script).

Initialize File Associations

Associative arrays are one of the important and distinctive features of Bash. The main logic behind this part is to create associations for the URLs used.

This script has a sophisticated way to streamline this associations that will be explained in this part.

`FILES_DIR="/root/files"` line at the start of the script declares a variable named FILES_DIR and assigns it the string /root/files. This is a directory path where certain files are tomporarily stored or will be stored.

```
declare -A associate_files

associate_f() {
    local key=$1
    local url=$2
    local base_path=$3

    local final_path="$base_path/$key"

    associate_files["$key"]="$url $FILES_DIR/$key $final_path"
}
```

`declare` is a Bash specific command to create an association. The `declare` command declares an associative array named `associate_files`. Associative arrays in Bash are akin to hash tables in other programming languages, allowing you to store and access data with a key-value pair mechanism. The `-A` flag is used to declare an associative array specifically.

The function `associate_f` is defined with three parameters: `key` , `url`, and `base_path`.

The purpose of this function is to associate a file (represented by key) with its URL and path information.

`local` variables are defined in an isolated way for their specific functions. They are not defined globally but only for this function.

`$1 and $2 and $3` refers to the argument order passed to the function respectively.

In this case these arguments will substitute:

The first argument is `$key`

The second argument is `$url`

The third argument is `$FILES_DIR`

`base_path` is another local variable, assigned the third argument passed to the function.

`final_path` variable constructs a path by concatenating `base_path` and `key`, separated by a forward slash (/). This path represents the final location where a file is stored or will be used. In simpler words, the second argument (`$2`) `base_path` and the first argument key or the name of the file in this case, will constitute the `final_path` variable.

The last line of the function populates the associative array that is named `associate_files`.

Associative arrays use a key. That is why the variable named as "key" while it can be named as anything. The key is placed inside the first brackets after the array name. Since this key can be varying in the script for all of the different files; a variable named `$key` is used.

The value corresponding to this key is a string that contains the url, the path constructed by concatenating `FILES_DIR` and `key`, and `final_path`, all separated by spaces.

All of these can look very complex and intimidating but it makes thing easier to understand and maintain. This can be understood with an example.

An Example Usage of the Function:

```
associate_f "package.use" "$URL_PACKAGE_USE" "$PORTAGE_DIR"
```

`associate_f` calls the function.

`"package.use"` is the first argument (`$1`) passed to it.

`"$URL_PACKAGE_USE"` is the second argument (`$2`) passed to it.

`"$PORTAGE_DIR"` is the third argument (`$3`) passed to it.

After placing the variables, it becomes this:

```
associate_files["key"]="$url $FILES_DIR/$key $final_path"
    # becomes
associate_files["package.use"]="$URL_PACKAGE_USE /root/files/package.use
↪   /etc/portage/package.use"

    # making it similar to English
associate_files["nameofthefile"]="URLofthefile.com
↪   temporarylocation/nameofthefile ultimatelocation/nameofthefile"
```

With this way, the temporary location and the file name are not needed to be entered everytime for tens of different files. The user can add any extra file by just giving its name, its URL and its final location separated by spaces using the function named `associate_f`. That is all.

All of these associations are stored inside a function named `update_associations`. The reason behind this is that some of the ultimate location variables may not be determined at the start of the script (such as the name of the user folder). With this way, the function can be called later in the script again to update and populate all of the associations again with a single invocation to `update_associations` function.

```
update_associations() {
    associate_f "package.use" "$URL_PACKAGE_USE" "$PORTAGE_DIR"
    associate_f "package.accept_keywords" "$URL_ACCEPT_KEYWORDS"
     ↪  "$PORTAGE_DIR"
    associate_f ".config" "$URL_KERNEL_CONFIG" "$LINUX_DIR"
    #... other associations
    }
```

These are three of the example associations inside the function. All of the above tasks aimed to make this part simpler.

```
move_file() {
    local key=$1
    local custom_destination=${2:-}
    local download_path final_destination
    IFS=' ' read -r _ download_path final_destination <<<
     ↪  "${associate_files[$key]}"

    mv "$download_path" "$final_destination"
}
```

Then a function named `move_file` is created. This function will use the file associations to move the files to their proper locations.

This function is declared with two parameters: `key` and an optional `custom_destination`.

The syntax `${2:-}` implies that the `custom_destination` will be assigned to the value of the second argument passed to the function if it exists; otherwise, it remains unset. This is a conditional assignment used for optional parameters.

Then the function creates two extra variables named `download_path` and `final_destination`.

The line starting with IFS is crucial and multi-faceted.

The Internal Field Separator (IFS) is temporarily set to a space character. This affects how Bash splits the input text into fields. Normally a whitespace is the default IFS, so this part can be (and is) removed from the script but it stays in the guide for extra information.

Normally the `read` command is mostly interactive but now it takes its input from the associative array. The underscore is a placeholder variable used to discard the first field (in this case, the URL) from the input.

`<<< "${associate_files[$key]}"` is a Bash Here String. It feeds the string resulting from `${associate_files[$key]}` directly into the `read` command.

The users should recall that the associative files have three space separated arguments.

The final command is `mv` which stands for move. This is a basic command that takes two arguments; the first being the current path of the file and the second argument is the path that the file will be carried over.

The Retrieval of the Required Files

This part includes more than one function to download the required files using the given URL list to the temporary location.

First, a basic progress bar is implemented to show the progress during the download process based on the number of total files. A completely sensitive real-time progress bar is hard to implement and requires more complex programming since `curl and git` outputs are different and they do not output a sensitive progress information.

Then, a function for downloading files is implemented. This is where the necessary changes are applied for different types of links.

Lastly, a function combining the other two function and implementing the actual retrieval of files is implemented.

Implement a Progress Bar

```
update_progress() {
    local total=$1
    local current=$2
    local pct=$(( (current * 100) / total ))
    local filled_blocks=$((pct * 65 / 100))
    local empty_blocks=$((65 - filled_blocks))
    local bar=''

    for i in $(seq 1 $filled_blocks); do
        bar="${bar}${GREEN}#${NC}"
    done

    for i in $(seq 1 $empty_blocks); do
        bar="${bar}${RED}-${NC}"
    done

    echo -ne "\r\033[K$bar${PURPLE} $pct%${NC}"
}
```

The `update_progress` function, dynamically displays a progress bar in the console, visualizing the completion percentage of a task. It accepts two parameters: total (the total count of the task) and current (the current progress count). The function calculates the percentage completion (pct) by dividing current by total and multiplying by 100.

The visual representation of the progress bar comprises 65 blocks. This number is arbitrary. Too big numbers make the bar too long and it may introduce overflowing on TTYs which is definitely not wanted. The number of filled blocks is determined by scaling the completion percentage to the 65-block length. This scaling is done by multiplying pct by 65 and then dividing by 100. The remainder of the bar consists of empty blocks, calculated by subtracting filled blocks from 65.

The function uses two 'for' loops. The filled blocks will be green, while the empty blocks will be red. A better looking progress bar could also be implemented but if the script is run on the TTY then those characters can't be seen properly.

A newly introduced flag `-n` for `echo` prevents a newline. Thus creating a constant and dynamically changing progress bar.

Handle Downloading Types

```
download_file() {
    local source=$1
    local dest=$2

    [ -d "$dest" ] && {
        log_info b "Directory $dest already exists, skipping download."
        return
    }

    [ -f "$dest" ] && {
        log_info b "File $dest already exists, skipping download."
        return
    }

    [[ "$source" == *powerlevel10k* ]] && { git clone --depth=1 "$source"
    ↪   "$dest" > /dev/null 2>&1; return; }

    [[ "$source" == *".git" ]] && [[ "$source" != *powerlevel10k.git* ]] &&
    ↪   { git clone "$source" "$dest" > /dev/null 2>&1; return; }

    curl -L "$source" -o "$dest" > /dev/null 2>&1
}
```

`-d` check on Bash controls if the target is a directory.

The checks are very simple. If the targeted files and directories are already downloaded, the function skips them.

The third check is for `powerlevel10k` which is a featureful but minimal and fast, interactive shell theme. Its repository is too big in size for downloading. `--depth=1` option helps to download only the necessary part of it.

If the target is a git repository which is understood by its URL (they end with .git), then `git clone` command is used. For all other instances, curl command is used.

`curl` command with its `-L` option is used to follow any redirects. By default, if the server responds with a redirect (status codes like 301 or 302), curl does not follow it and just outputs the redirect response. The -L option ensures that curl follows the server's response if it is a redirect, and it will continue to do so until it reaches the final data destination. This is essential for downloading files from sources that might redirect you to different URLs (which is common on the internet).

`-o` option is used to specify the output file's name. Without this option, curl will output the downloaded data to the standard output (typically the terminal). `-O` option (uppercase) on the other hand works without specifying a filename and it uses the default one it founds in the URL.

In this case, source and dest variables will be taken from the associative array: The URL and the temporary location.

Retrieve the Files

```
retrieve_files() {
    mkdir -p "$FILES_DIR"
    local total="${#associate_files[@]}"
    local current=0

    echo -ne "\033[?25l"

    for key in "${!associate_files[@]}"; do
        current="$((current + 1))"
        update_progress "$total" "$current"

        IFS=' ' read -r source dest _ <<< "${associate_files[$key]}"
        download_file "$source" "$dest"
    done

    echo ""

    echo -e "\033[?25h"
}
```

The function begins by creating the target directory if it doesn't already exist.

The function then initializes two local variables: total, which is assigned the total number of files to be retrieved (derived from the length of the `associate_files` array), and current, which is initialized to zero and tracks the number of files processed.

The `#` symbol is used to get the number of elements in an array. `@` symbol is used for array expansion on Bash, expands the list with all elements.

The `!` symbol in this context is used to get the keys of an associative array. It shouldn't be confused with the other usage of `!` which shows negativity in checks.

This `echo` line uses echo to send a special escape sequence to the terminal to hide the cursor. At the end, another command is used to retrieve the cursor again. More information:

Detailed ANSI Escape Sequences

With every loop, the current number is increased by one which is indicated by `$((current + 1))`. Arithmetic operations in Bash require double parentheses. The dollar sign is for creating a subshell to do the operation.

Every loop uses a key defined in file associations. Then the loops uses the `update_progress` and `download_file` functions with the key.

The read command only needs the URL and the destination. So it discards all other variables with `_`.

Check the Downloaded Files

```
check_files() {
    for key in "${!associate_files[@]}"; do
        ( read -r _ f _ <<< "${associate_files["$key"]}"
          [ -s "$f" ] || [ -d "$f" ] || { log_info r "$f is missing."; kill
          ↪  0; } ) &
    done
    wait
}
```

The `check_files` function is a component designed to verify the existence and non-emptiness of a set of files or directories specified in the associative array.

The function is highly efficient, employing parallel processing and a minimal nature.

Bash additionally has 'arrays' compared to a standard variable. Generally, if something is assigned to a string within quotes then it is a variable: `string="something"`. If the string uses parentheses instead of quotes then it is an array that can include more than one string. In this case, on the other hand, the parenthesis refer to the creation of a subshell. All of these commands that are inside parenthesis run in different subshells in parallel.

`&` is a shell operator to put a task on the background. With this way, all files can be checked in parallel resulting in much faster (almost instant) operations.

Again `!` is used to get the keys and `@` is used to get all elements.

`-s` and `-d` check if the `f` is empty or a directory. If neither is the case, it echoes (prints) the file name. Ultimately, `&` put the command in the background.

`wait` command pauses the script to wait for all background jobs (in this case, the file checks) to finish.

The command `kill` stops the process if a file is missing. The process `0` is the script itself in this case.

Renewing the Gentoo Environment

```
renew_env() {
    env-update && source /etc/profile && export PS1="(chroot) ${PS1}"
}
```

This function is used several times in the script and it aims to centralize the environment renewal process.

The `env-update` command in Gentoo Linux is a critical utility that plays a pivotal role in the management and configuration of the system's environment. Its primary function is to amalgamate various configuration files into a consolidated set of environment variables, thereby ensuring a coherent and unified operational framework for the system.

Gentoo Linux maintains a set of distinct configuration files, such as those residing in `/etc/env.d`, `/etc/profile.env`, and others. These files contain environment variables, PATH settings, or other crucial configurations. The command scans these files, assimilating their contents into a comprehensive environment profile.

It ensures that variables are not redundantly defined, thereby averting potential conflicts or inconsistencies in the system's behavior.

One of the pivotal outputs of env-update is the generation of the `/etc/profile.env` file. This file embodies the cumulated and processed environment settings. It is sourced by the system shell during user login or by system scripts, thus imparting the configured environment variables globally across the system.

Besides environment variables, it also refreshes the shared libraries cache by invoking `ldconfig`. This is a crucial step, ensuring that newly installed libraries are correctly recognized and that the cache is in sync with the current state of the system's filesystem.

Sourcing `/etc/profile` had already been explained.

Since these commands renew the prompt, the prompt is changed again to include the (chroot) indication.

Configure Locale Settings

```
configure_locales() {
    sed -i "/#en_US.UTF/ s/#//g" /etc/locale.gen

    locale-gen

    eselect locale set en_US.utf8

    echo 'LC_COLLATE="C.UTF-8"' >> /etc/env.d/02locale

    renew_env
}
```

This is a crucial step. While locale settings are used by user-space programs, they are also used by the compilers and shells. An UTF-8 locale is generally needed.

The `sed` command captures the related name of the preferred locale inside the file that has the different locale settings and remove the hashtag symbol from it. That symbol, in general, disables a line in configuration files and shells.

After activating the preferred locale. Locales should be generated with `locale-gen` command.

`eselect` is a Gentoo specific command. Instead of using complex symbolic links, the user can use specific eselect commands to select between entries (such as selecting a locale, profile, compiler version, Python version or similar selections). In this case, the activated and generated locale is enabled.

The `LC_COLLATE` setting is specifically recommended by *Gentoo Wiki*: "Some programs are written in such a way that they expect traditional English ordering of the alphabet, while some locales, most notably the Estonian one, use a different ordering. Therefore it's recommended to explicitly set LC_COLLATE to C when dealing with system-wide settings."

The `echo` command normally prints the specified string but in this case it redirects the output (`>>`) inside a file. This is an important redirection. Redirections with one (`>`), delete all of the content from the file and appends the specified text. `>>` (with two >) on the other hand adds the specified output to the last line of the file without changing anything in the file.

The explained `renew_env` function is used to renew the environment after locale changes.

Configure Compiler Flags

```
configure_flags() {
    sed -i '/COMMON_FLAGS=/ c\COMMON_FLAGS="-march=native -O2 -pipe"
        /^FFLAGS/ a\LDFLAGS="-Wl,-O2 -Wl,--as-needed"
        /^FFLAGS/ a\RUSTFLAGS="-C debuginfo=0 -C codegen-units=1 -C
        ↪  target-cpu=native -C opt-level=3"' /etc/portage/make.conf

    emerge --oneshot app-portage/cpuid2cpuflags
    cpuid2cpuflags | sed 's/: /="/; s/$/"/' >> /etc/portage/make.conf

    echo "" >> /etc/portage/make.conf

    cat <<-EOF >> /etc/portage/make.conf
        ACCEPT_KEYWORDS="~amd64"
        RUBY_TARGETS="ruby32"
        RUBY_SINGLE_TARGET="ruby32"
        PYTHON_TARGETS="python3_12"
        PYTHON_SINGLE_TARGET="python3_12"
        LUA_TARGETS="lua5-4"
        LUA_SINGLE_TARGET="lua5-4"
        EOF
}
```

`/etc/portage/make.conf` file on Gentoo controls global settings for Portage, the package manager of Gentoo.

Compiler flags are used by the compilers such as GCC and Clang (LLVM). These flags, determine the optimization levels, target architechture, security related settings and other compiler related settings.

The function starts with a `sed` command. Instead, echo could also be used but echo can only print lines at the end of the file. Here, in order to have a file that is properly formatted, the extra lines are printed under other compiler flags (specifically FFLAGS).

`sed` can handle different operations in a single invocation. The different operations can be placed on different lines without needing to separate operations with a `;`. This is useful for increasing programming efficiency and performance.

`app-portage/cpuid2cpuflags` package is needed to extract the CPU flags for the currently running CPU. These flags are features for CPUs that can be enabled for packages.

`--oneshot` option does not save the package among exclusively installed packages (in other words, world packages). Therefore it can be uninstalled with `--depclean` option.

`--depclean` option when used alone without a package name (`emerge --depclean`) removes all packages that are not exclusively installed or packages that are not needed for any other software as a dependency. If a software is installed with `--oneshot`, then they will be removed with the next invocation of `emerge --depclean`.