emlix
embedded linux systems

Elektrobit

Tech paper

crinit and cominit white paper

Summary of white paper contents

This white paper discusses the boot process of Linux systems and presents two FOSS software packages that implement important tasks required to boot up a Linux system in a function-focused embedded environment.

The presented tools cominit and crinit are alternatives to systemd and initramfs tools such as dracut or the initramfs tools of Debian. In many cases it is a good idea to just take these tools and configure them as needed. But in embedded systems that are highly optimized towards performance (fast boot time), storage (no waste of memory), security, and functionality, these generic tools might not fit perfectly. To achieve the abovementioned goals with systemd and dracut, a manual shrinking process from the feature-rich default configuration needs to be accomplished by developers. Security features such as configuration-file checking are not available on application level in systemd, dracut, and the Debian tools. These tools also heavily depend on shell interpreters, which might cause security concerns.

To overcome these limitations with focus on embedded devices, crinit and cominit were developed.

1. Booting – basics

Booting a Linux system is basically a trivial task. The bootloader just needs to load a few executable files and some data to the RAM and then pass control to the executables. Following this, the kernel will do its own init steps and then pass control over to user space, to get the remaining steps done.

These user-space activities are the focus of this white paper, hence the necessary activities and some tools committing these tasks are presented.

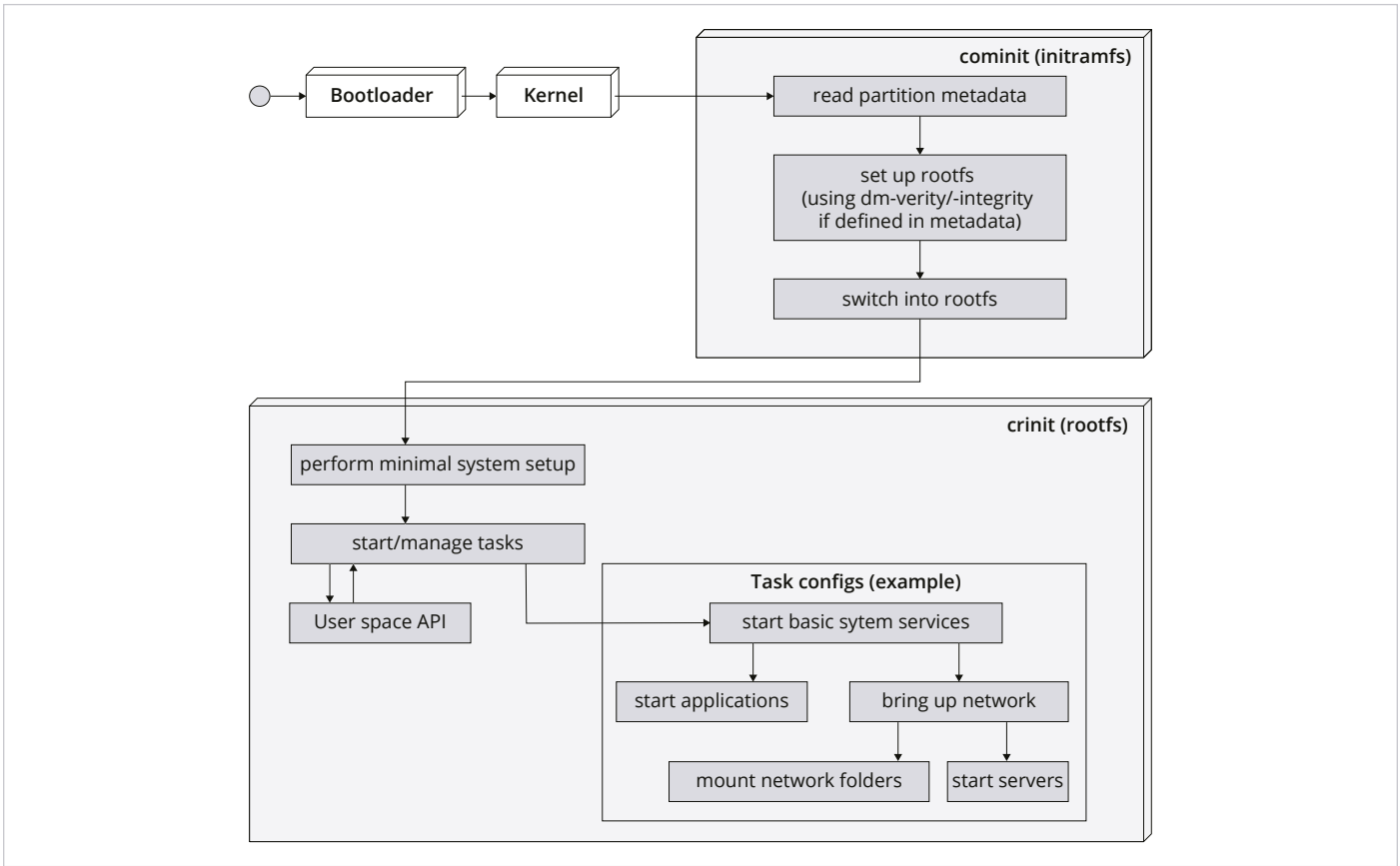

Figure 1: Flow diagram of a Linux bootup

The optional initramfs

Server and desktop systems running Linux almost always make use of a so-called initial RAM disk, short initrd, or initramfs, see *Figure 1*. The initramfs is a file system image that contains a regular root file system (rootfs) to operate a feature-rich user-space environment from RAM and, hence without the need of having a storage device, is already usable at this early time. Sometimes the term “early user space” is used for this minimal Linux environment.

This initramfs contains all tools needed to detect and mount the correct final root file system. This may also include the steps of decryption and integrity checking.

The following table sketches the activities of an initramfs during boot up:

Number	Activity	Description
1	Receive control	The kernel has mounted the disk image from the RAM and starts `/init`.
2	Mount basics	Mount basic file systems such as proc and sys.
3	Seek rootfs	Read and interpret config files, environment variables, partition metadata, and/or other storage such as EEPROM, fuses, or eMMC metadata to find the correct location of the rootfs to be used.
4	Load drivers	Depending on the storage hardware, drivers need to be loaded as modules to the Linux kernel. For example, modules to support NVMe or USB might need to be loaded. After this step, step 3 might need to be repeated.
5	Setup crypto layers	The setup and configuration of the Linux cryptography layer to enable integrity and/or confidentiality protection on the rootfs follows. In this step, key handling using a TPM or other high security module takes place. dm-crypt, dm-integrity, and/or dm-verity are set up at this time.
6	Check file system	If the file system is detected as unclean, a check takes place now.
7	Mount the rootfs	The root file system (rootfs) is now mounted and its content becomes available.
8	Check content	At this point it is possible to selectively check files in the rootfs for integrity.
9	Switch rootfs	Make the new rootfs the real one and remove the RAM rootfs (e.g., with `chroot()`, `pivot_root()`, or `switch_root`).
10	Start init	Pass control to the init daemon of the new rootfs, e.g., `/sbin/init`.

These steps can vary considerably depending on the use case and the environment of the system. Especially in embedded systems, a fast boot up is usually important and the hardware layout is fixed. Thus, the whole initramfs should be streamlined or can even be skipped. In that case, a direct boot into the final rootfs is executed.

If requirements such as cryptographic protection of the rootfs are involved and/or hardware designs are complex, an initramfs is often the only choice for an effective implementation. In many cases generic shell scripts are found to implement the abovementioned steps with the help of busybox and other toolings.

However, in embedded systems shell scripts might be banned due to security and boot-time concerns. If hardware is invariant, no dynamic seeking of configuration is needed, and the tools in the initramfs can directly follow the given setup procedure without losing time.

2. cominit

The open-source tool cominit is intended to be used in such a scenario. It is small but flexible, does not depend on any shell tooling, and allows to set up integrity-protected root file systems using dm-verity and dm-integrity. As the traditional last step, cominit passes control over to the init executable found in the root file system. This can be systemd or any other init daemon. Independent of the tool used the init daemon is referred to as “init”.

Being started by the kernel, “init” has no parent process and always receives the lowest possible pid of “1”. Init is not allowed to terminate in any case. If that happens – for whatever reason – the kernel panics, leading to a system crash that can only be resolved by rebooting. The duty of the system daemon is to start further tools and daemons, to load driver modules, and to configure interfaces, e.g., the network. All these steps are performed directly, or the system daemon delegates it to other programs.

3. crinit

crinit is a system daemon or init daemon that receives control after the root file system has been mounted and is part of it. Hence, crinit plays the role of “init”. Depending on the selected system design, the root file system is either mounted by the kernel or by the tooling within the initramfs, as mentioned above. However, it is basically irrelevant for the init daemon how the root file system was mounted.

crinit does the following steps during startup and runtime:

Number	Activity	Description
1	Read config	Read the configuration files.
2	Verify config	Verify the configuration files using cryptographic functions and keys.
3	Start tasks	Start all configured tasks in parallel that do not depend on each other.
4	Start tasks	Start configured tasks that have dependencies as soon as all dependencies are fulfilled.
5	Handle events	In case a task has finished its init procedure or has terminated, the event is handled according to the config.
6	Listen to IPC	Listen to commands from the IPC interface and handle it.

crinit tries to handle all steps from 3 on in parallel if dependencies allow this.

Tasks

crinit and other init daemons use the term “tasks”, sometimes also referred to as “services”.
The following services can be fulfilled by a task:

- Start of a program that does init work and terminates afterwards, e.g., tool to adjust the system time once.
- Start of a program that keeps running and waits for interaction on its interface, e.g., a webserver daemon.
- Loading of a kernel module to add driver functionality.
- Mounting of file systems.
- Starting of a program that maintains interfaces, e.g., a network manager.

After crinit has finished to start all tasks, and no further task is left to be started, the system could be interpreted as being in the state of 'normal operation'.

crinit has no status of 'normal operation' and the like, it just stops starting new tasks if none is left. The system designer can freely define this as 'normal operation' or run level 6. crinit does not define nor need such stage definitions. During the phase of normal operation crinit just proceeds with its activity of observing the tasks and their dependencies. In case a task changes its state, crinit will react accordingly. For example, if configured, crinit will restart a task that has unexpectedly terminated.

Being responsible to start up the system, crinit is naturally also in charge of shutting down the system. Unlike the startup, the shutdown is straightforward and disregards any dependency. crinit just sends a signal to all processes to request them to come to a clean end. After a defined time (called grace period), crinit requests the kernel to terminate all processes and shortly after that crinit will request the kernel either to reboot or to power off the hardware.

Features

crinit orders all tasks according to the dependencies. Tasks can depend on other tasks, which in turn can again depend on tasks. This chain of dependencies forms a directed acyclic graph (dag) that crinit follows. It starts all tasks that have satisfied dependencies without interruption. Following this, all tasks that have no dependencies are started by crinit as soon as possible, e.g., right after boot up. The start of tasks can lead to the fulfillment of dependencies and in turn cause the start of further tasks. But not only the start, also termination or fail of a task can fulfill dependencies leading to the start of tasks that depend on this kind of events.

The following list gives all events that a crinit-task might depend on:

Dependency type	Description
<task>:fail	The task did terminate with an error code.
<task>:wait	The task did terminate and reports success.
<task>:wait-notified	The task has notified crinit via sd_notify that it is now ready to be used.
<task>:spawn	The task was just started.
@provided:<feature>	A task was started that explicitly provides the feature.
@elos:<filter>	An elos event was received as defined in the filter.

In addition to the dependencies the command of the task to be started needs to be configured. This is done via the configuration key COMMAND. In the example case of a web server this config key contains the executable file of the web server, the dependencies define a relationship to the network. The resulting crinit file would look like this:

```
---
NAME = webserver
COMMAND = /bin/webserverd
DEPENDS = @provided:network
---
```

This above file lets the webserver wait until the network is provided. The following crinit files defines the requested network-dependency:

In case static IP shall be used:

```
---
NAME = static-netcfg
COMMAND = /bin/ip addr add 192.168.2.3/24 dev eth0
PROVIDES = ipv4_static:wait network:wait
---
```

Or in case of using a dhcp client:

```
---
NAME = dyn-netcfg
COMMAND = /sbin/dhcp-client eth0
PROVIDES = ipv4_dyn:wait network:wait
---
```

No matter how the network was configured, either static or dynamic, the dependency “network:wait” is provided as the result of logical-or operation. Hence this dependency can be used to let services such as the webserver depend on a configured network, independent of the used technology.

It is possible to leave out the COMMAND. In that case the task just forwards the dependencies as defined. This allows for defining dependency groups or meta tasks that organize tasks into a more abstract dependency, allowing for a cleaner system design.
This example shows grouping of servers to an abstract all_services dependencies.

```
---
NAME = all_services
# COMMAND = (none)
DEPENDS = webserver:spawn database:spawn mqtt:spawn
---
```

Any task that depends on all three is now simpler to define as is just needs to depend on all_services.
If needed, for a more traditional Unix design this concept can be used to define run levels.

crinit and elos

Each dependency event of crinit can be reported to elos[1], in order to be used for further event processing. This allows crinit to be an event input to elos. As shown before, crinit can also be a receiver of events from elos and trigger starting of tasks that depend on events. Hence crinit has an optional full duplex connection to elos. crinit does not depend on elos.

1 elos is free and opensource software and available here: https://github.com/Elektrobit/elos

4. Shell-free and secured startup

The startup is a critical phase from a security perspective. If important tasks are not started or if unintended tasks are initialized the overall security might be completely compromised. But also manipulated parameters of tasks can put a system into an insecure state.

As shell scripts and any other interpreter-based program (e.g., python or perl) are easier to manipulate than binaries, it is mandatory not to have any interpreter on such secured systems. This results in the fact that traditional systemV inits are not usable any longer as they heavily depend on shell scripts. But crinit goes even further by protecting its config files with cryptographic checksums. This mitigates any attacks via manipulated config files.

Signatures on config files

crinit can be forced into a higher security mode via kernel command line. If "`crinit.signatures=yes`" is set, crinit will read each file and verify it to the signature file, before following the given configuration. The signature file is expected to be in the same directory, sharing the name with a .sig extension. crinit uses RSA-PSS (RSA-4096 with SHA256) and a public key from the Linux kernel keyring. This key needs to be loaded as "crinit root" by, for example, cominit to the @user keyring of the root user.
In case of a verification failure crinit rejects to use that modified file.

Environment for each task

As is usual with today's startup daemons crinit shall allow to define an environment for the tasks. crinit tries to solve this using pragmatic compromise by offering many but not all possible ways of environment definition. This way crinit does not support starting containers in a fully enclosed environment. It is intended to support following definitions:

- Unix users and groups
- Capabilities
- cgroups (controllers: CPU, CPU set, memory, io, pids)
- seccomp
- Standard IO redirection
- Environment variables

5. crinit at runtime

During runtime of a Linux system, the startup daemon plays a reduced role and steps into action only in case a task terminates. When it comes to shutdown requests, the init daemon has the duty to notify and finally terminate all processes before requesting the kernel to flush all file systems and instructing the appropriate kernel driver to power off.

Command	Activity	Description
`crinit-ctl addtask`	Add new or overwrite existing tasks.	Allows to add a new task defined in provided file; if dependencies are fulfilled, the task is started, otherwise it waits.
`crinit-ctl addseries`	Add a full series of tasks.	Allows to add series files of tasks, each task has its own file and is referenced from the series file.
`crinit-ctl enable | disable`	Enable/disable tasks.	Sets or unsets a special dependency (e.g., "@ctl:enable") to allow dependent tasks to start. Can be used to model manual entering of run levels.
`crinit-ctl stop | kill | restart`	Terminate, kill, and restart tasks.	Sends kill or terminate signals to the task, or requests to restart it.
`crinit-ctl status | list`	Get status.	Gets the status of all or just the specified tasks.
`crinit-ctl reboot | poweroff`	Reboot or shutdown of the whole system.	Requests all processes to shut down and requests the kernel to reboot or power off.

crinit can be interacted with via its API and a command line tool that allows easy handling from the shell during debugging, development, and testing of Linux systems.

Following commands are available:
All these commands are additionally available in the crinit library "`libcrinit-client.so`" for use in C/C++ or similar environments.

With these tool-set, crinit is able to startup and control Linux systems, but with a little extra information from the processes crinit fulfills its purpose even more effectively. This is best explained on an example: "A" depends on "B". But "B" is not ready to serve "A" before finishing its own initialization. Hence, "A" should be started after "B" has finished to initialize and is ready to serve "A".

For crinit it would be very helpful to know: When is a process ready to serve?

When is a process ready to serve?

Basically, each software has three stages of operation:

1. Initialization or startup
2. Normal operation or running
3. Cleanup, flushing, and shutdown of process

Most software products will subdivide these into further small chunks, but these three can be applied to almost any software. During the startup phase the software reads its configuration and initializes its internal structures. During this time it cannot yet serve its IPC interfaces (e.g., ports are not yet open or cannot be serviced). Though this phase is designed to be as short as possible, it can still consume a considerable amount of time. After this the process is ready and changes over to the normal operation or running state, see *Figure 2* below. For the outside world, this state change is not easy to detect, hence the "sd_notify" notification mechanism was introduced[2]. crinit uses this interface to receive the startup notification.

The process just needs to call "`sd_notify(0, "READY=1");`" and crinit is made aware that the process has now finished its startup and is ready to serve. Following that, any dependency of type <*name*>:`wait-notified` is now fulfilled and crinit starts them right away. These notifications avoid any polling-based solutions that waste time and CPU cycles. Further features of "sd_notify" are not implemented by crinit in favor of keeping it small and simple.

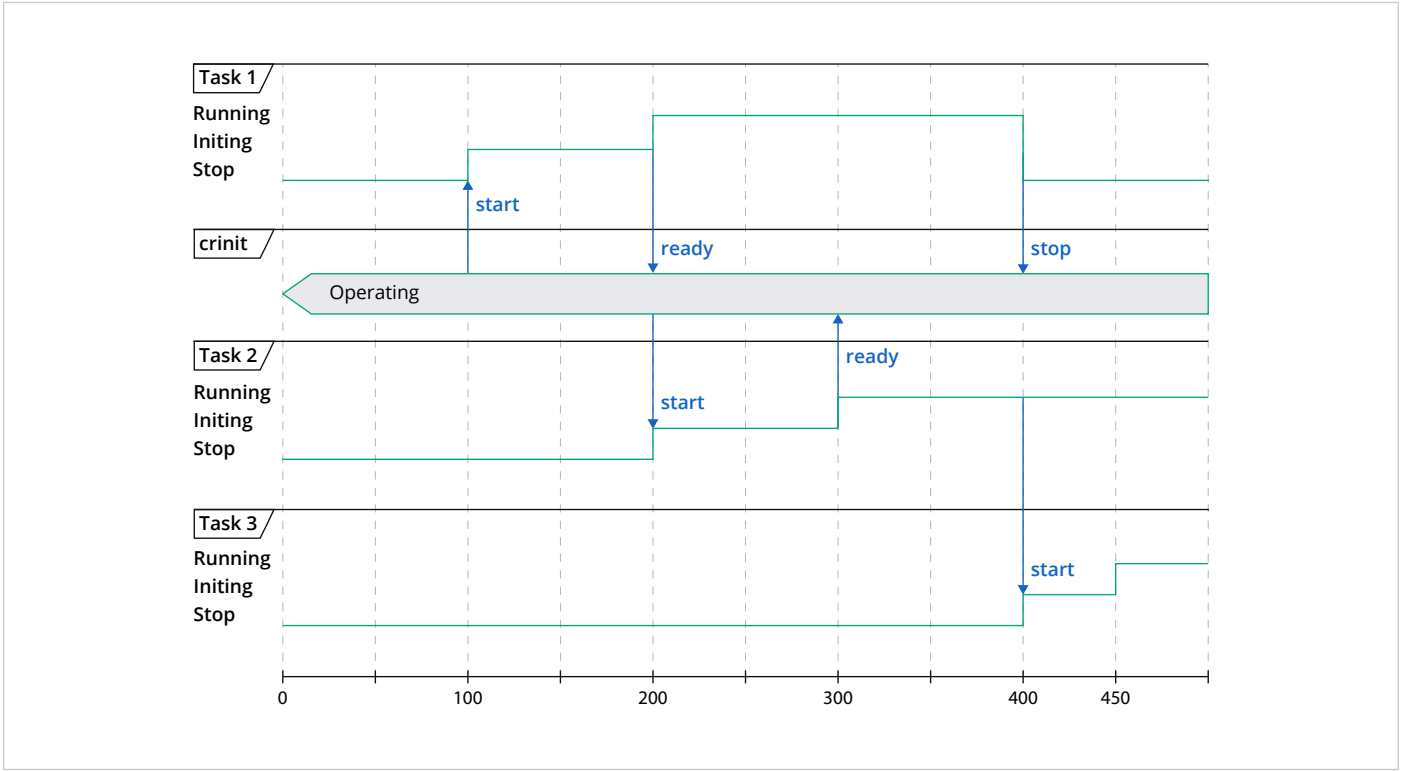

Figure 2: Timing diagram of tasks and events

2 https://www.freedesktop.org/software/systemd/man/latest/sd_notify.html

6. Conclusion

With the features presented here, crinit is a lightweight and feature-rich startup daemon ready to be used in embedded systems and other function-focused Linux systems. The learning curve to integrate Linux systems with crinit is not steep due to its focused feature set and completeness of documentation.

The software cominit is intended to be used in an initial RAM disk-based system to do basic setup steps of a Linux system during bootup. It finishes these boot-time-critical operations in a secure way without wasting time in hot-plug detection and unneeded device reading and seeking.

The enclosed tests for unit and integration level of both tools support further development by allowing for easy assurance of a minimum quality level. crinit and cominit are free and open-source software (FOSS) and licensed under MIT license, they are available at GitHub:

https://github.com/Elektrobit/crinit
https://github.com/Elektrobit/cominit

About the authors

Thomas Brinker
emlix GmbH

Thomas Brinker is a Senior Systems Engineer and Project Manager at emlix GmbH. He is an architect for secured embedded Linux systems in the automotive, medical, industrial, and consumer device fields, performing requirements engineering and design throughout the entire product life cycle.

Andreas Zdziarstek
emlix GmbH

Andreas Zdziarstek is a system engineer at emlix GmbH. He works primarily on embedded Linux software, developing bespoke solutions for a range of use cases in the automotive area, focusing on safety, reliability, and availability.

About Elektrobit

Elektrobit is an award-winning and visionary global vendor of embedded and connected software products and services for the automotive industry. A leader in automotive software with over 35 years of serving the industry, Elektrobit's software powers over five billion devices in more than 600 million vehicles and offers flexible, innovative solutions for car infrastructure software, connectivity & security, automated driving and related tools, and user experience. Elektrobit is a wholly-owned, independently-operated subsidiary of Continental.

For more information, visit us at **elektrobit.com**

Elektrobit Automotive GmbH
Am Wolfsmantel 46
91058 Erlangen, Germany

Phone: +49 9131 7701 0
Fax: +49 9131 7701 6333

sales@elektrobit.com