

Continuous Integration and Debian Packaging for Rapidly Evolving Software

Jens Georg (Presenter),
Anthony Barker (Corresponding Author),
Martin Hierholzer, Martin Killenberg,
Tomasz Kozak, Dietrich Rothe,
Nadeem Shehzad, Christian Willner

ICALEPCS 2023

Cape Town, 09.10.2023

Introduction

C++ software libraries usually guarantee binary compatibility within a minor release version

- > Stable interface for dependent projects

Binary compatibility of libraries comes with a price

- > Requires resources to maintain and check
- > Development is less flexible because ABI and API must not change

We decided to not have binary compatibility

- + Required flexibility for our workflow
- All dependencies have to be re-compiled each time

Challenges for Debian packaging and continuous integration!

Debian packaging



Debian packages for libraries

- > Library packages are binary compatible for the lifetime of a distribution
- > Binary incompatible changes require a new package name
- ⇒ Add the major and minor version number to the package name

Extended minor version number

If the dependency of a library changes the ABI

- > ...library package name has to change as well, even if the library versions have not changed
- > ...installed versions of the library must be distinguishable for the linker
- ⇒ Add a build version to the minor version number

Example Debian package name: `libchimeratk-deviceaccess03-11-focal1`

Example .so file name: `libChimeraTK-DeviceAccess.so.03.11focal1`



Debian packaging scripts

If a library without binary compatibility is re-packaged, all dependent libraries have to be re-built as well.

ChimeraTK Debian packaging scripts

- > Dependency database as files on a folder structure
- > Automatically increase build version when necessary
- > Reverse dependency lookup
 - Identify all libraries which need to be rebuild
 - Build in the correct order

Screenshot: Master packaging script

```
killenb@mskpcx29269:~/DebianPackagingScripts-focal$ ./master focal chimeratk-applicationcore latest
Active overrides:
Searching for reverse dependencies...
Verifying versions...
Sorting package list...

The following packages will be built (in that order):
chimeratk-applicationcore 03.03.00
tec-applicationmodule 02.00.00
chimeratk-applicationcore-microdaq 03.00.02
chimeratk-motordrivercard-applicationmodule 00.07.01
chimeratk-applicationcore-loggingmodule 01.01.00

Do you want to proceed with configuring and building the packages in the given versions (y/N)?
```

DeviceBackends as runtime loadable plugins

DeviceBackends

- > Access hardware (PCIe)
- > Access control system device servers (DOOCS, EPICS, OPC UA)
 - Depends on the control system software stack

Typical applications are linked to a particular backend.

Generic applications

- > ...should work with all backends
 - > ...don't want to link against all possible CS SW stacks
- ⇒ Load backends when needed

Plugin loading mechanism

- > Needs binary compatibility
- ⇒ You have to load the **exactly** matching library version



Meta packages

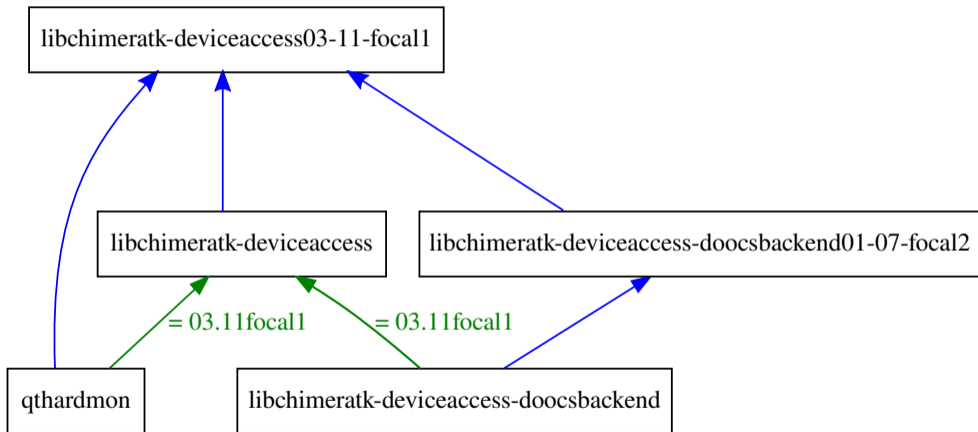
Device backends (loadable plugins)

- > Don't have a version in the name
- ⇒ Meta package which provides .so-file without version

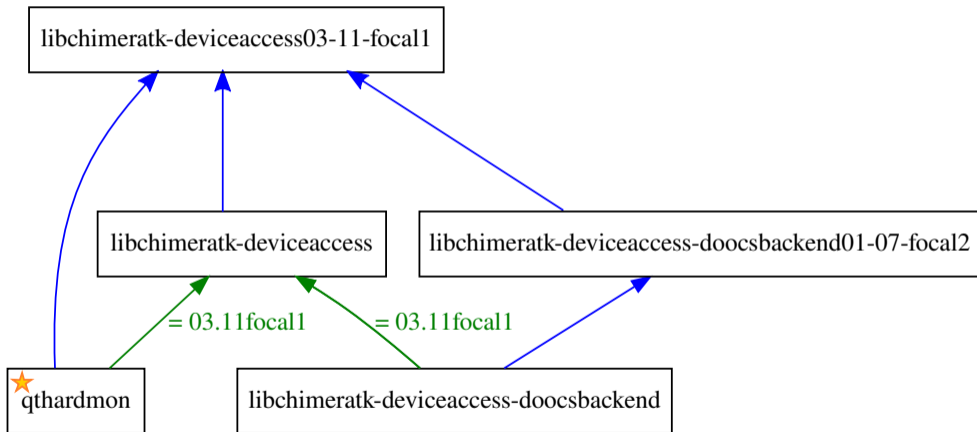
DeviceAccess core library

- > Empty meta package without version in the package name as dependency anchor
- > Plugin meta packages and generic applications depend on the **exact** version of the DeviceAccess meta package

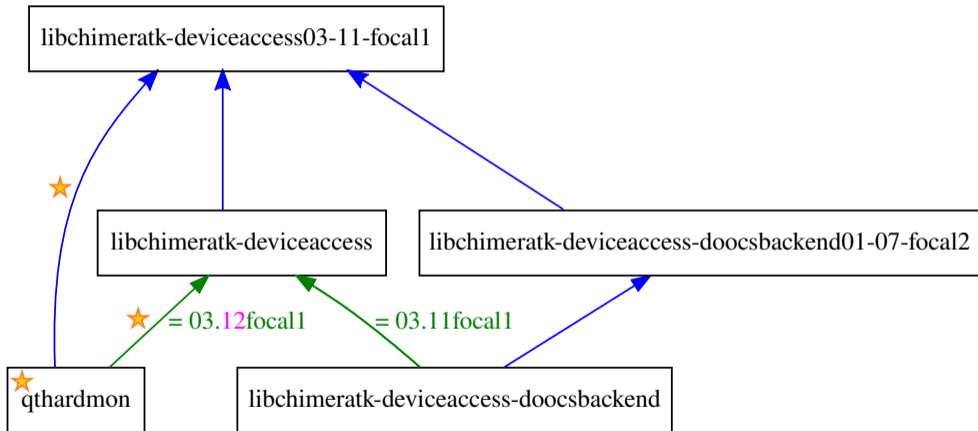
Example for dependencies



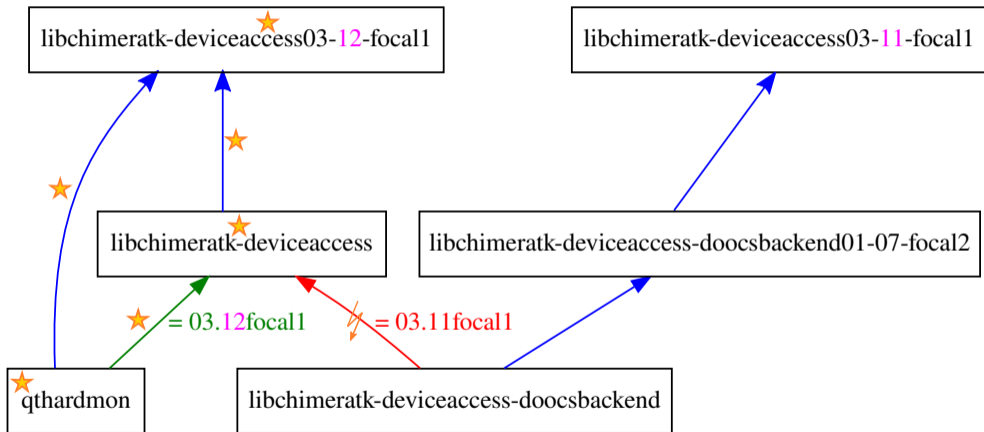
Example for dependencies



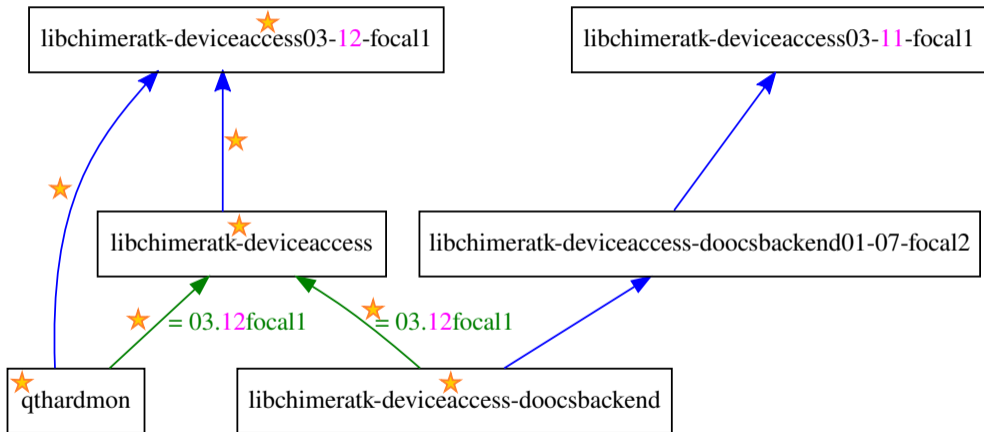
Example for dependencies



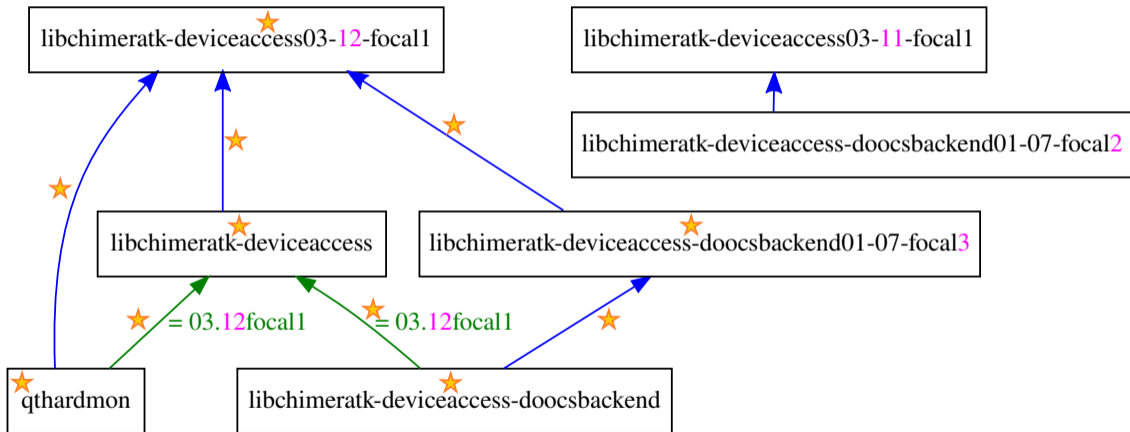
Example for dependencies



Example for dependencies



Example for dependencies



Continuous integration

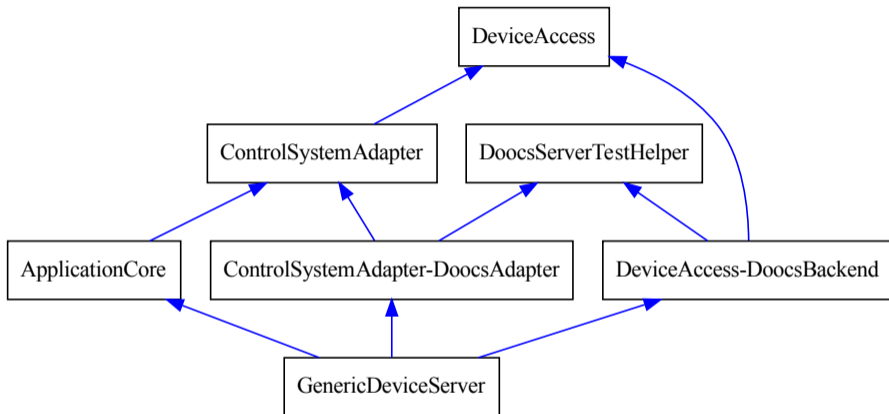


Continuous Integration

- > Large software stack
 - More than 20 ChimeraTK libraries and tools
 - More than 30 application projects
- > Lots of interdependencies between the projects
 - All dependent projects must be triggered



Diamond Dependencies



- > GenericDeviceServer is triggered 3 times if DeviceAccess changes
- > First two builds fail in case of binary incompatible changes

Continuous Integration

- > Large software stack
 - More than 20 ChimeraTK libraries and tools
 - More than 30 application projects
- > Lots of interdependencies between the projects
 - All dependent projects must be triggered
 - Projects compiled many times due to diamond dependencies
 - Many false "failed" runs in case of binary incompatibility
 - Jenkins might be overloaded if too many jobs are triggered



Continuous Integration

- > Large software stack
 - More than 20 ChimeraTK libraries and tools
 - More than 30 application projects
- > Lots of interdependencies between the projects
 - All dependent projects must be triggered
 - Projects compiled many times due to diamond dependencies
 - Many false "failed" runs in case of binary incompatibility
 - Jenkins might be overloaded if too many jobs are triggered

Solution

- > Don't use Jenkins' built-in dependency triggering
- > Implement a dependency database via Groovy scripts
- > Each job knows and triggers its dependencies



Extensive testing

- > Check that the code compiles
- > Check for compiler warnings
- > Use multiple compilers
- > Run unit tests
- > Check for memory leaks
- > Check for timing races
- > Build for multiple target platforms
- > Make debug and release builds

Testing a single project can take long

- > C++ code with templates takes several minutes to compile¹
- > Tests with network based protocols: *> 30 min*
- > Tests take much longer to compile than library itself

Testing the whole stack took more than 5 hours!

¹ on a dedicated build host with 256 cores and 1 TB RAM

Implementing a "fast track"

Goal: Improve the time until you get results

fast track

- > Only compile the libraries
- > Only debug build for 1 target with 1 compiler
- > Immediately trigger dependent projects when ready

fast track testing

- > Compile the tests
- > Run unit tests

nightly build

- > All build types
- > All target platforms
- > Multiple compilers

Results of the improvements

- + False "failed" test results due to diamond dependencies are resolved
- + "Fast track" only takes 20 minutes for all libraries
- "Fast track" still too slow for immediate feedback
- Jenkins still runs out of memory or becomes unresponsive
- Jobs fail if a downstream project cannot be started or crashes
⇒ new source of false "failed" test results
- Groovy scripts often fail for unknown reasons (too complex?)



Conclusion

Not having binary compatibility for libraries poses challenges to Debian packaging and continuous integration

- > Version numbers as part of the package name ✓
- > Automated packaging scripts with a dependency database ✓
- > Binary compatible plugins using version dependencies in meta packages ✓

- > Groovy scripts with dependency database solve diamond dependencies ✓
- > Jenkins "fast track" ✓

- > "Fast track" still too slow for immediate feedback !
- > Large code base and extensive tests bring Jenkins to its limits !


Thank you!

Contact

DESY. Deutsches
Elektronen-Synchrotron

www.desy.de

Anthony Barker

 0000-0003-2631-1029

MSK

anthony.barker@desy.de

+49-40-8998-4289

