Unicon epkg

User Manual

Table of Contents

Legal Notice & Disclaimer

© 2025 Cloud Software Group, Inc. and its affiliates, including Citrix Systems, Inc. and Unicon GmbH. All rights reserved.

This document contains proprietary and confidential information of Cloud Software Group, Inc. and its affiliates. No part of this document may be reproduced, transmitted, distributed, or republished in any form or by any means without prior written consent. The information provided herein is for informational purposes only and is subject to change without notice.

Cloud Software Group, Inc. and its affiliates disclaim all liability for the accuracy, completeness, or timeliness of the information in this document and any errors or damages that may arise from its use. No warranties, express or implied, including but not limited to warranties of merchantability or fitness for a particular purpose, are provided.

eLux® and Scout Enterprise Management Suite® are registered trademarks of Unicon GmbH in the European Union, the United Kingdom, and the United States. ScoutaaS® is a registered trademark in the European Union, the United Kingdom, the United States, and Japan. All other product names may be trademarks of their respective owners.

Prerequisites

- The eLux SDK is used to get access to epkg.

epkg: Motivation

Maintaining a full linux distribution running on several million fully managed devices results in a few requirements regarding packaging and package maintenance:

- all the normal packaging demands like full control over what is installed where and what is omitted, pre and post scripts, dependencies etc
- increased security since we can disable automatic third-party upgrades and can verify all files before shipment
- scope of packages is more feature oriented, meaning a package contains all software necessary for a specific feature regardless of how many different libraries, projects or other input that feature needs. This is in contrast to the usual linux distro packaging where one package is one project and if that project needs some other libraries it is solved by having dependencies to potentially many different packages that all only contain one library
- packages should be small to:
  - save local disk space
  - save network bandwidth during updates/installation
  - reduce attack surface
- to ease resource burdens it's necessary to easily consume existing debian packages/pools or other archive formats to avoid redoing work that has already been done by major linux distributions, like building packages from source

These requirements lead to the development of a custom package format and - later - a tool to repackage software from Debian packages and various archiving formats: epkg

Features are:

- download packages from a Debian mirror, extract them and package only specific files
- use various archiving formats like *tar*, *gz*, *zip*, ... and extract/repackage contents during EPM package builds
- install files to different locations than in the original package
- create feature packages "FPM" within an EPM to allow administrators to select only specific features of a package
- sign created packages
- directly upload packages to a eLux container after build
- create archives as output instead of EPMs/FPMs that can be used as initrds

epkg: Summary

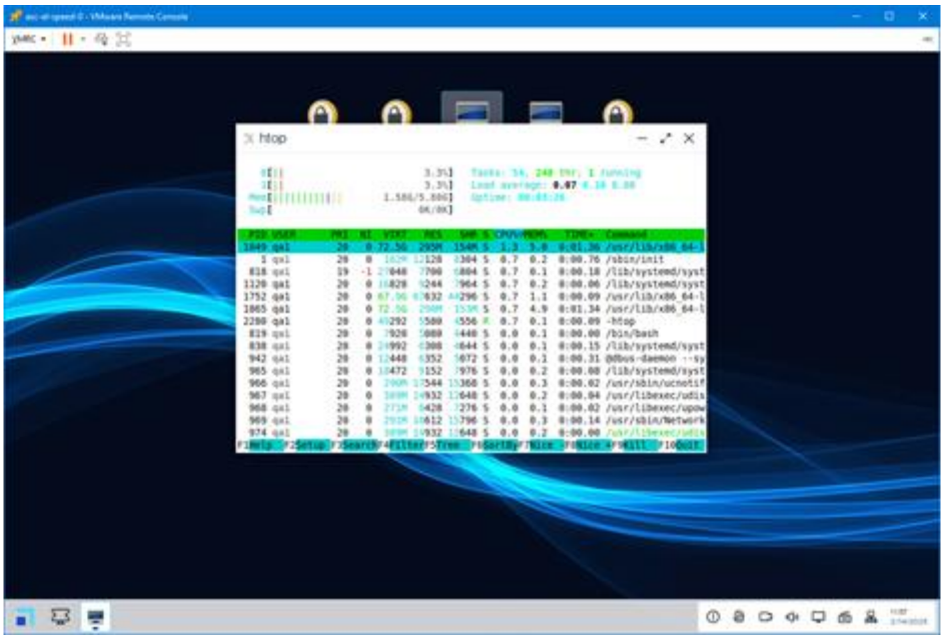
epkg is the command line tool used to create eLux packages in the EPM and FPM formats. It supports the following Modules: Build, Convert, ConvertFrom, Install, New, NewFrom, Info, Sign, Size, UpdateCopyrightInfo and Upload

- **Build**: Build EPM and all its FPM
- **Convert**: Convert from an old metafile version to the newest.
  - This can also be used to convert Makefile based pre RP6 meta repositories [LEGACY]
- **ConvertFrom**: Convert from a package tree to the latest metafile format [LEGACY]
- **Install**: Copy EPM and its FPM files to a container directory.
  - Note that this does not install packages on eLux. Rather this command just copies created EPM and FPM packages to a configured container directory.
- **New**: Create empty meta repository files for the given EPM and FPM names
- **NewFrom**: Create empty meta repository files from a pre RP6 package tree [LEGACY]
- **Info**: Get some information from the ebkepm. Used for example in automation scripts ran on build servers.
- **Sign**: Creates a signature for the given file
- **Size**: Calculate the uncompressed size or parse a eluxman logfile to get the compressed size
- **UpdateCopyrightInfo**: Update copyright md5sums and write them back to the correct .md5sum files
- **Upload**: Upload EPM and its FPM to a webelias instance

In most cases, a developer creating eLux packages will only need the New and Build commands.

QuickStart Guide

This quick start guide explains how to create an eLux package that packages the htop interactive process viewer.



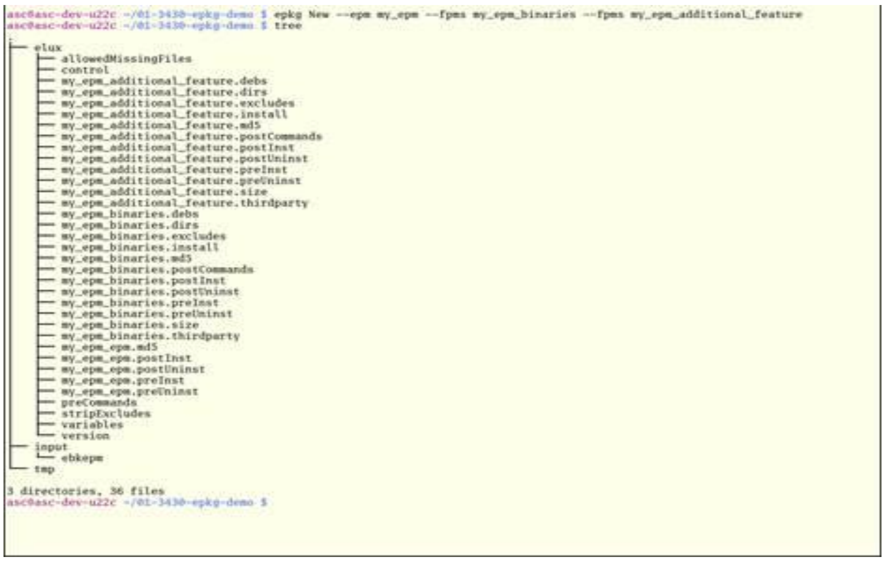
```
htop
    0[||                       3.3%]   Tasks: 54, 240 thr; 1 running
    1[||                       3.3%]   Load average: 0.07 0.10 0.08
  Mem[|||||||||||||     1.58G/5.80G]   Uptime: 00:03:26
  Swp[                        0K/0K]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU% MEM%   TIME+  Command
 1049 qal        20   0 72.5G  295M  154M S  1.3  5.0  0:01.36 /usr/lib/x86_64-l
    1 qal        20   0  162M 12128  8304 S  0.7  0.2  0:00.76 /sbin/init
  818 qal        19  -1 27048  7700  6804 S  0.7  0.1  0:00.18 /lib/systemd/syst
 1120 qal        20   0 16828  9244  7964 S  0.7  0.2  0:00.06 /lib/systemd/syst
 1752 qal        20   0 67.9G 67632 44296 S  0.7  1.1  0:00.09 /usr/lib/x86_64-l
 1065 qal        20   0 72.5G  290M  153M S  0.7  4.9  0:01.34 /usr/lib/x86_64-l
 2280 qal        20   0 49292  5580  4556 R  0.7  0.1  0:00.09 -htop
  819 qal        20   0  7928  5000  4440 S  0.0  0.1  0:00.00 /bin/bash
  838 qal        20   0 24992  6308  4644 S  0.0  0.1  0:00.15 /lib/systemd/syst
  942 qal        20   0 12448  6352  5072 S  0.0  0.1  0:00.31 @dbus-daemon --sy
  965 qal        20   0 10472  9152  7976 S  0.0  0.2  0:00.08 /lib/systemd/syst
  966 qal        20   0  290M 17544 15368 S  0.0  0.3  0:00.02 /usr/sbin/ucnotif
  967 qal        20   0  389M 14932 12648 S  0.0  0.2  0:00.04 /usr/libexec/udis
  968 qal        20   0  271M  6428  7276 S  0.0  0.1  0:00.02 /usr/libexec/upow
  969 qal        20   0  291M 16612 13796 S  0.0  0.3  0:00.14 /usr/sbin/Network
  974 qal        20   0  389M 14932 12648 S  0.0  0.2  0:00.00 /usr/libexec/udis
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

Creating a Simple EPM

A simple EPM can be created in a few easy steps. First we run epkg New in an empty directory to create an empty EPM with 2 FPM packages

```
$ epkg New --epm my_epm --fpms my_epm_binaries --fpms my_epm_additional_feature
```

which creates a few new files in *input/* and *elux/* subdirectory. In particular, it creates file *input/ebkepm* which contains most meta information about the created packages.



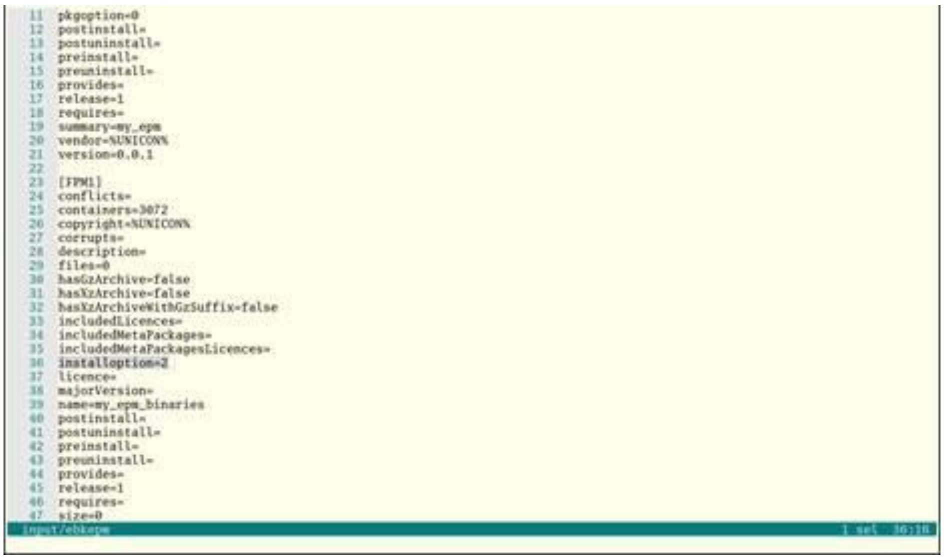
```
asc@asc-dev-u22c ~/01-3430-epkg-demo $ epkg New --epm my_epm --fpms my_epm_binaries --fpms my_epm_additional_feature
asc@asc-dev-u22c ~/01-3430-epkg-demo $ tree
.
├── elux
│   ├── allowedMissingFiles
│   ├── control
│   ├── my_epm_additional_feature.debs
│   ├── my_epm_additional_feature.dirs
│   ├── my_epm_additional_feature.excludes
│   ├── my_epm_additional_feature.install
│   ├── my_epm_additional_feature.md5
│   ├── my_epm_additional_feature.postCommands
│   ├── my_epm_additional_feature.postInst
│   ├── my_epm_additional_feature.postUninst
│   ├── my_epm_additional_feature.preInst
│   ├── my_epm_additional_feature.preUninst
│   ├── my_epm_additional_feature.size
│   ├── my_epm_additional_feature.thirdparty
│   ├── my_epm_binaries.debs
│   ├── my_epm_binaries.dirs
│   ├── my_epm_binaries.excludes
│   ├── my_epm_binaries.install
│   ├── my_epm_binaries.md5
│   ├── my_epm_binaries.postCommands
│   ├── my_epm_binaries.postInst
│   ├── my_epm_binaries.postUninst
│   ├── my_epm_binaries.preInst
│   ├── my_epm_binaries.preUninst
│   ├── my_epm_binaries.size
│   ├── my_epm_binaries.thirdparty
│   ├── my_epm_epm.md5
│   ├── my_epm_epm.postInst
│   ├── my_epm_epm.postUninst
│   ├── my_epm_epm.preInst
│   ├── my_epm_epm.preUninst
│   ├── preCommands
│   ├── stripExcludes
│   ├── variables
│   └── version
├── input
│   └── ebkepm
└── tmp

3 directories, 36 files
asc@asc-dev-u22c ~/01-3430-epkg-demo $
```

We ignore most created files, but alter the install options for both our FPMs by updating *input/ebkepm*. For this test package, we wish to achieve the following:

- *my_epm_binaries* FPM should be installed whenever the EPM is installed (mandatory)
- Additional feature FPM *my_epm_additional_feature* should be deactivated by default. Administrators can enable it if they require the features provided by the package.

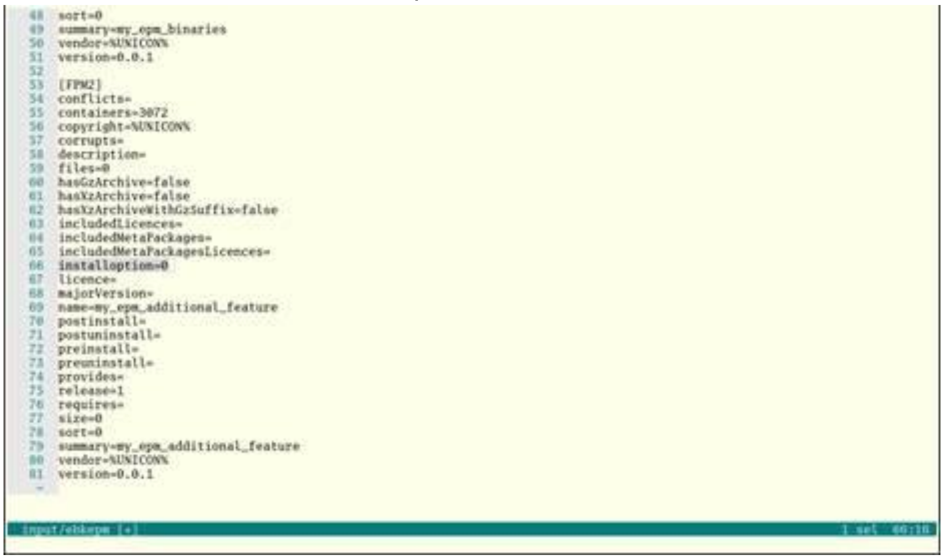
To achieve this we set the values for *installoption=* in the correct section within *input/ebkepm:*

- In section [FPM1] (the section where *summary=my_epm_binaries*) we use *installoption=2.*



```
pkgoption=0
postinstall=
postuninstall=
preinstall=
preuninstall=
provides=
release=1
requires=
summary=my_epm
vendor=%UNICON%
version=0.0.1

[FPM1]
conflicts=
containers=3072
copyright=%UNICON%
corrupts=
description=
files=0
hasGzArchive=false
hasXzArchive=false
hasXzArchiveWithGzSuffix=false
includedLicences=
includedMetaPackages=
includedMetaPackagesLicences=
installoption=2
licence=
majorVersion=
name=my_epm_binaries
postinstall=
postuninstall=
preinstall=
preuninstall=
provides=
release=1
requires=
size=0
input/ebkepm
```

- In section [FPM2] we use *installoption=0.*



```
sort=0
summary=my_epm_binaries
vendor=%UNICON%
version=0.0.1

[FPM2]
conflicts=
containers=3072
copyright=%UNICON%
corrupts=
description=
files=0
hasGzArchive=false
hasXzArchive=false
hasXzArchiveWithGzSuffix=false
includedLicences=
includedMetaPackages=
includedMetaPackagesLicences=
installoption=0
licence=
majorVersion=
name=my_epm_additional_feature
postinstall=
postuninstall=
preinstall=
preuninstall=
provides=
release=1
requires=
size=0
sort=0
summary=my_epm_additional_feature
vendor=%UNICON%
version=0.0.1
~

input/ebkepm [+]
```

More information about installoptions can be found at Format#InstallOptions.

Building Package for the First Time

epkg will try to be helpful when necessary configuration is missing. We can try building our package right away with the following parameters

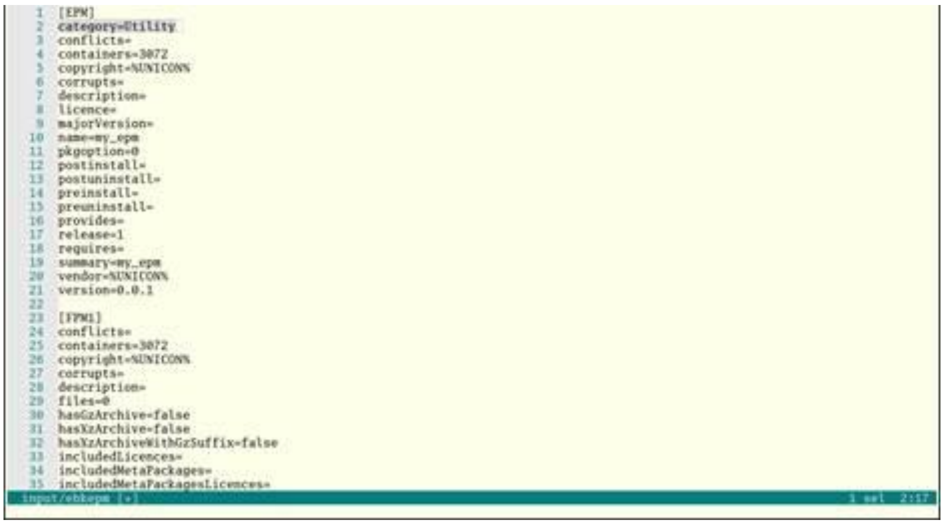
- --container elux7 - to tell epkg that the package should target eLux 7
- --releaseType test - to build a test version. A test version build does not update any md5 sum files (more on that later)
- --log - creates an additional log file for the build

but this leads to an error:

$ epkg Build --container elux7 --releaseType test

```
$ epkg Build --container elux7 --releaseType test
…
…
epkg: error: input/ebkepm: field category must be set in ebkepm section [EPM]
```

This error tells us that we have to define some category for our EPM (see: EPM Categories) for the EPM the file *input/ebkepm.* Utility seems like a good choice for htop.



```
[EPM]
category=Utility
conflicts=
containers=3072
copyright=%UNICON%
corrupts=
description=
licence=
majorVersion=
name=my_epm
pkgoption=0
postinstall=
postuninstall=
preinstall=
preuninstall=
provides=
release=1
requires=
summary=my_epm
vendor=%UNICON%
version=0.0.1

[FPM1]
conflicts=
containers=3072
copyright=%UNICON%
corrupts=
description=
files=0
hasGzArchive=false
hasXzArchive=false
hasXzArchiveWithGzSuffix=false
includedLicences=
includedMetaPackages=
includedMetaPackagesLicences=
input/ebkepm [+]
```

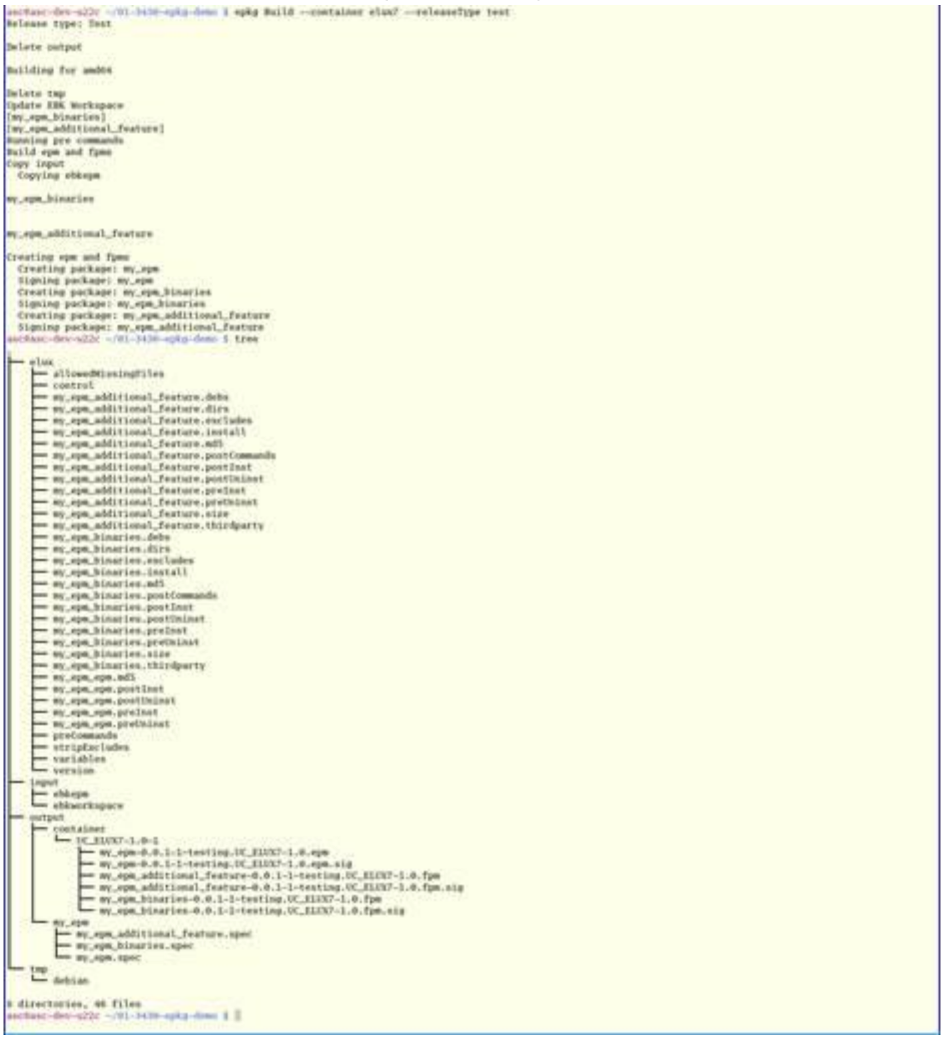
Retrying the build should now lead to a successful package build.

$ epkg Build --container elux7 --releaseType test

```
$ epkg Build --container elux7 --releaseType test
Release type: Test

Delete output

Building for amd64

Delete tmp
Update EBK Workspace
[my_epm_binaries]
```

```
[my_epm_additional_feature]
Running pre commands
Build epm and fpms
Copy input
  Copying ebkepm

my_epm_binaries


my_epm_additional_feature

Creating epm and fpms
  Creating package: my_epm
  Signing package: my_epm
  Creating package: my_epm_binaries
  Signing package: my_epm_binaries
  Creating package: my_epm_additional_feature
  Signing package: my_epm_additional_feature
```

We now see a few more files generated by the build



```
asc@asc-dev-u22c ~/01-3430-epkg-demo $ epkg Build --container elux7 --releaseType test
Release type: Test

Delete output

Building for amd64

Delete tmp
Update EBK Workspace
[my_epm_binaries]
[my_epm_additional_feature]
Running pre commands
Build epm and fpms
Copy input
  Copying ebkepm

my_epm_binaries


my_epm_additional_feature

Creating epm and fpms
  Creating package: my_epm
  Signing package: my_epm
  Creating package: my_epm_binaries
  Signing package: my_epm_binaries
  Creating package: my_epm_additional_feature
  Signing package: my_epm_additional_feature
asc@asc-dev-u22c ~/01-3430-epkg-demo $ tree
.
├── elux
│   ├── allowedMissingFiles
│   ├── control
│   ├── my_epm_additional_feature.debs
│   ├── my_epm_additional_feature.dirs
│   ├── my_epm_additional_feature.excludes
│   ├── my_epm_additional_feature.install
│   ├── my_epm_additional_feature.md5
│   ├── my_epm_additional_feature.postCommands
│   ├── my_epm_additional_feature.postInst
│   ├── my_epm_additional_feature.postUninst
│   ├── my_epm_additional_feature.preInst
│   ├── my_epm_additional_feature.preUninst
│   ├── my_epm_additional_feature.size
│   ├── my_epm_additional_feature.thirdparty
│   ├── my_epm_binaries.debs
│   ├── my_epm_binaries.dirs
│   ├── my_epm_binaries.excludes
│   ├── my_epm_binaries.install
│   ├── my_epm_binaries.md5
│   ├── my_epm_binaries.postCommands
│   ├── my_epm_binaries.postInst
│   ├── my_epm_binaries.postUninst
│   ├── my_epm_binaries.preInst
│   ├── my_epm_binaries.preUninst
│   ├── my_epm_binaries.size
│   ├── my_epm_binaries.thirdparty
│   ├── my_epm_epm.md5
│   ├── my_epm_epm.postInst
│   ├── my_epm_epm.postUninst
│   ├── my_epm_epm.preInst
│   ├── my_epm_epm.preUninst
│   ├── preCommands
│   ├── stripExcludes
│   ├── variables
│   └── version
├── input
│   ├── ebkepm
│   └── ebkworkspace
├── output
│   ├── container
│   │   └── UC_ELUX7-1.0-1
│   │       ├── my_epm-0.0.1-1-testing.UC_ELUX7-1.0.epm
│   │       ├── my_epm-0.0.1-1-testing.UC_ELUX7-1.0.epm.sig
│   │       ├── my_epm_additional_feature-0.0.1-1-testing.UC_ELUX7-1.0.fpm
│   │       ├── my_epm_additional_feature-0.0.1-1-testing.UC_ELUX7-1.0.fpm.sig
│   │       ├── my_epm_binaries-0.0.1-1-testing.UC_ELUX7-1.0.fpm
│   │       └── my_epm_binaries-0.0.1-1-testing.UC_ELUX7-1.0.fpm.sig
│   └── my_epm
│       ├── my_epm_additional_feature.spec
│       ├── my_epm_binaries.spec
│       └── my_epm.spec
└── tmp
    └── debian

8 directories, 46 files
asc@asc-dev-u22c ~/01-3430-epkg-demo $
```

- *input/ebkworkspace* - a temporary file saving some information about the build. It can be ignored and should not be added to version control

- *epkg.log* - a build log, which contains pretty verbose information about what `epkg` did to build the package. For now it's not interesting, but it's a good thing to know it can be created using the `--log` argument and contains essential information for example why a certain FPM has been rebuild
- *tmp/* - a temporary directory where all used input packages are extracted to. In this build it contains no files (only another empty subdirectory *debian*). Once input archives are added to a *.debs* or *.thirdparty* meta file, they will be saved here after fetching
- *output/* - this directory contains the actual build output in 2 additional subdirectory
    - *container/* - here the EPM and FPM files from the build are. Basically everything that will be copied to a container when using `epkg Install` or `epkg Upload`
        - This is the main artifiact or package that you should ship to customers. Customers will import the *container/* contents into their production container.
    - *my_epm/* - this directory has the same name as the EPM. It contains some spec files and a folder with each FPM's directory tree (if not empty)

However since the packages contain no files yet there is little point in installing such a package.

Adding Files to Packages

We did not add any files to the FPM packages so far.

Let's do that now. In particular, we want to add binary `/usr/bin/htop` from the htop Ubuntu package. To achieve this, we have to (1) update file *elux/my_epm_binaries.debs* to include the `htop` Ubuntu package and (2) update *elux/my_epm_binaries.install* to include file `/usr/bin/htop` from the `htop` Ubuntu package.

Adding additional files

```
# add htop to the binaries .debs file; usually you'd edit the file with the editor of your choice
echo htop >> elux/my_epm_binaries.debs

# add the htop binary to the .install file. We usually skip the leading /, but keeping the leading / also works
echo usr/bin/htop >> elux/my_epm_binaries.install
```

For our `my_epm_addtional_feature` FPM we want to add a custom file not imported from any Ubuntu package. Rather we provide the file as-is to our package directory.

Adding additional files

```
# add an (empty) configuration file for the additional feature, directly in input/
# note: the directory name below input/ has to match the FPM name the file should be included in
mkdir -p input/my_epm_additional_feature/etc/my_epm
touch input/my_epm_additional_feature/etc/my_epm/additional_feature.conf
```

That's it. Let's build

$ epkg Build --container elux7 --releaseType test

```
$ epkg Build --container elux7 --releaseType test
Release type: Test

Delete output

Building for amd64

Delete tmp
Update EBK Workspace
epkg: error: If debian packages are defined repositoryName must be given
```

... and promptly run into an error.

Since we now have defined data that is pulled from a Debian repository, `epkg` needs to know which Debian repository to pull from. Internally at Unicon we have separate repositories for every release, so it's important to specify the exact repository.

See the section Using Debian Repositories to learn how to configure a repository. For now let's assume we have a repository for Ubuntu 22.04 (jammy) named *jammy-main* configured and call `epkg` again.

$ epkg Build --container elux7 --releaseType test --repositoryName jammy-main

```
$ epkg Build --container elux7 --releaseType test --repositoryName jammy-main
Release type: Test

Delete output

Building for amd64

Delete tmp
Update EBK Workspace
Creating fake apt directory...
Downloading debian packages for my_epm_binaries ...
[my_epm_binaries]
  Extracting debian packages
[my_epm_additional_feature]
Running pre commands
Build epm and fpms
Copy input
  Copying ebkepm
  Copying my_epm_additional_feature

my_epm_binaries
  Copy files


my_epm_additional_feature

Creating epm and fpms
```

```
  Creating package: my_epm
  Signing package: my_epm
  Creating package: my_epm_binaries
  Signing package: my_epm_binaries
  Creating package: my_epm_additional_feature
  Signing package: my_epm_additional_feature

$
```

The build was successful. Directory `output/` contains the created packages.

Created Package Artifacts

Since the build was successful, let's re-check the directories

- *tmp/debian* now contains the downloaded htop_<version>.deb file
- *tmp/* contains the full directory tree extracted from the .deb file. In this case all files are within a *tmp/usr/* subfolder
- *output/container/...* contains the newly created EPM/FPM files. They are now slightly larger than before, since the additional files are included
    - This is the main artifact or package that you should ship to customers. Customers will import the *container/* contents into their production container.
- *output/my_epm* contains the full directory tree of both FPM packages

Furthermore we can inspect the EPM file by using the `eluxbuild` tool

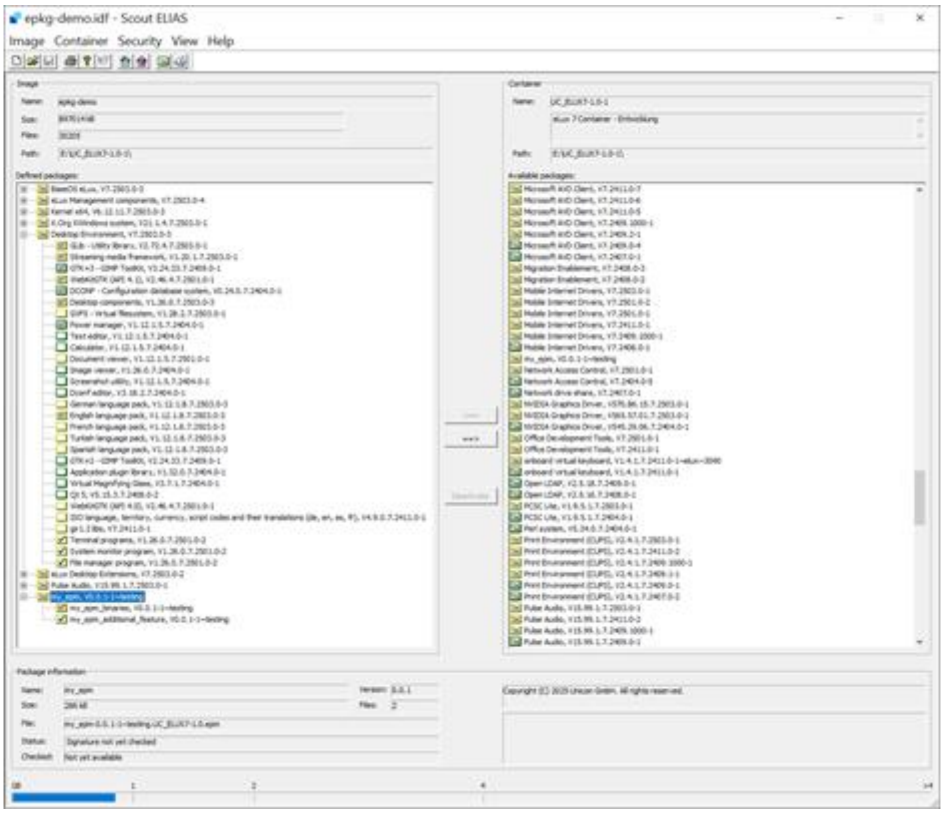
$ eluxbuild -qip output/container/…/…epm

```
$ eluxbuild -qip output/container/UC_ELUX7-1.0-1/my_epm-0.0.1-1~testing.UC_ELUX7-1.0.epm
Name     : my_epm                       Relocations: (not relocateable)
Version  : 0.0.1                        Vendor: Copyright (C) 2025 Unicon GmbH. All rights reserved.
Release : 1~testing                     Build Date: Thu 13 Feb 2025 10:28:19 AM CET
Install date: (not installed)           Build Host: sbr-dev-hw
Group    : EPM                          Source RPM: my_epm-0.0.1-1~testing.src.rpm
Size     : 0                            License: Copyright (C) 2025 Unicon GmbH. All rights reserved.
Summary           : my_epm
Description :

$OPTION=2$
$FPM0=0,my_epm_binaries-0.0.1-1~testing.UC_ELUX7-1.0.fpm,284,2,1$
$FPM1=1,my_epm_additional_feature-0.0.1-1~testing.UC_ELUX7-1.0.fpm,2,0,1$
```

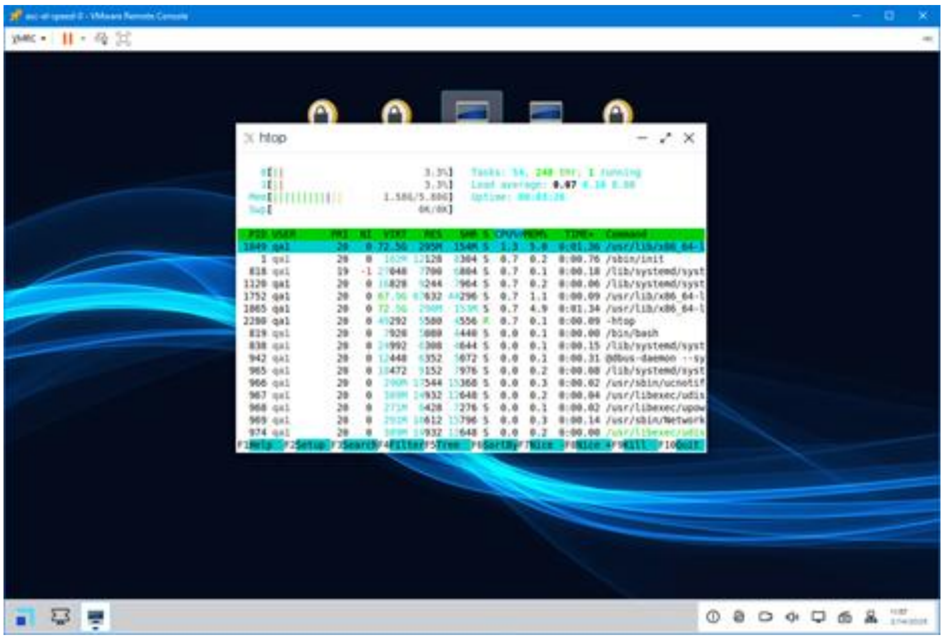
Installing the Created Package

Assuming you have a properly set up deployment environment with ELIAS and Scout, you can now copy artifacts in `output/container/` into your development container. If you do not have such an environment, please contact your Scout administrator.

Load a given container using ELIAS and create an eLux IDF image that includes our newly created package. Note that since htop is a command line application, we also have to manually enable the Terminal programs (desktop_environment_terminal) FPM if we wish to try out our package.

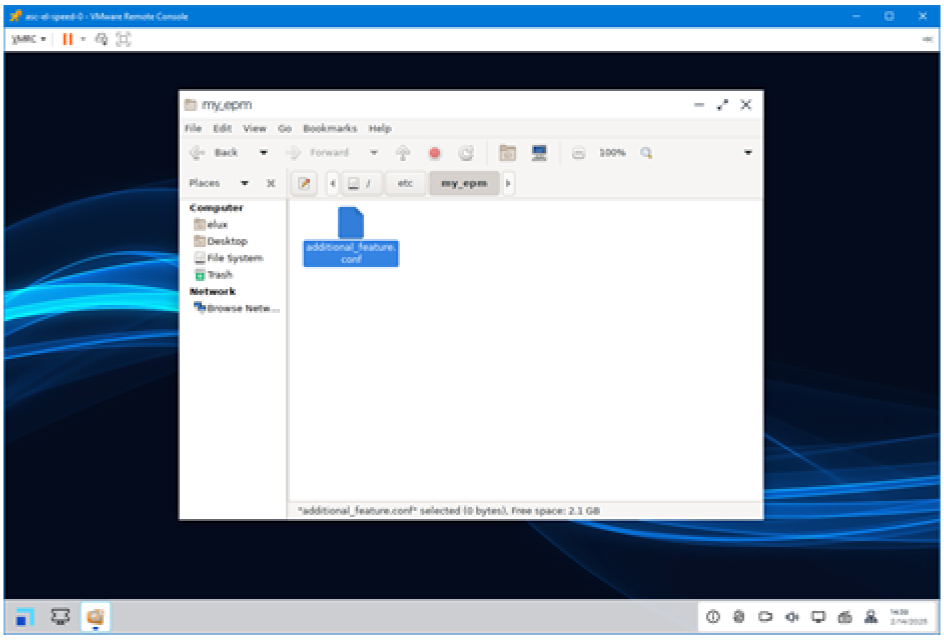

epkg-demo.idf - Scout ELIAS
Image Container Security View Help
Image
Name: epkg-demo
Defined packages:
Container
Available packages:
Package information
my_epm, V0.0.1-1-testing
my_epm_binaries, V0.0.1-1-testing
my_epm_additional_feature, V0.0.1-1-testing

After updating an eLux device to the given IDF and preparing a custom XTERM application that launches /usr/bin/htop , we can see that our package works as expected. Again, if you are unsure how to achieve this, please contact your Scout administrator.



```
    0[||                       3.3%]   Tasks: 54, 240 thr; 1 running
    1[||                       3.3%]   Load average: 0.07 0.18 0.08
  Mem[||||||||||||      1.58G/5.80G]   Uptime: 00:03:20
  Swp[                        0K/0K]

  PID USER      PRI  NI  VIRT   RES   SHR S CPU%▽MEM%   TIME+  Command
 1049 qa1        20   0 72.5G  295M  154M S  1.3  5.0  0:01.36 /usr/lib/x86_64-l
    1 qa1        20   0  162M 12128  8304 S  0.7  0.2  0:00.76 /sbin/init
  818 qa1        19  -1 27048  7700  6804 S  0.7  0.1  0:00.18 /lib/systemd/syst
 1120 qa1        20   0 16828  9244  7964 S  0.7  0.2  0:00.06 /lib/systemd/syst
 1752 qa1        20   0 67.9G 67632 44296 S  0.7  1.1  0:00.09 /usr/lib/x86_64-l
 1065 qa1        20   0 72.5G  290M  153M S  0.7  4.9  0:01.34 /usr/lib/x86_64-l
 2280 qa1        20   0 45292  5580  4556 R  0.7  0.1  0:00.09 -htop
  819 qa1        20   0  7928  5080  4440 S  0.0  0.1  0:00.00 /bin/bash
  838 qa1        20   0 24992  6308  4644 S  0.0  0.1  0:00.15 /lib/systemd/syst
  942 qa1        20   0 12448  6352  5072 S  0.0  0.1  0:00.31 @dbus-daemon --sy
  965 qa1        20   0 10472  9152  7976 S  0.0  0.2  0:00.08 /lib/systemd/syst
  966 qa1        20   0  290M 17544 15368 S  0.0  0.3  0:00.02 /usr/sbin/ucnotif
  967 qa1        20   0  389M 14932 12648 S  0.0  0.2  0:00.04 /usr/libexec/udis
  968 qa1        20   0  271M  6428  7276 S  0.0  0.1  0:00.02 /usr/libexec/upow
  969 qa1        20   0  293M 18612 15796 S  0.0  0.3  0:00.14 /usr/sbin/Network
  974 qa1        20   0  389M 14932 12648 S  0.0  0.2  0:00.00 /usr/libexec/udis
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

If you have a file explorer or terminal installed, you can also check the presence of the custom my_epm_additional_feature FPM file.



"additional_feature.conf" selected (0 bytes), Free space: 2.1 GB

Releasing the Package

So far we have only performed builds with –releaseType test. This results in EPM and FPM files with a ~testing suffix. There is also the option to provide a different string instead of 'testing' using the --testVersion parameter. For example, --testVersion can be used to mark build artifacts as belonging to a certain issue tracker ticket, e.g. --testVersion myproject~42. Allowed values are shown in the output of epkg Build --help.

Note

It's a good idea to track EPM folders in version control. In 'release' mode certain input/meta files are changed when epkg builds a package, to record internal state. Committing those changes back to version control is essential for epkg to make correct decisions for future builds.

More importantly we can make a release build using epkg Build --container elux7 --releaseType release --repositoryName jammy-main --autoIncrementVersion

This build will do the following:

- produce EPM/FPM files without a ~testing suffix
- record md5sums of all relevant files in the per EPM and per FPM *.md5* files
- automatically bump the release number of all *changed* FPM, meaning all FPM/EPM versions of the form -X~testing will become -Z , where Z = X + 1 . For example:

- *my_epm-0.0.1-1~testing.UC_ELUX7-1.0.epm* → my_epm-0.0.1-2.UC_ELUX7-1.0.epm
- *my_epm_binaries-0.0.1-1~testing.UC_ELUX7-1.0.fpm* → *my_epm_binaries-0.0.1-2.UC_ELUX7-1.0.fpm*

The changed md5sum files should all be committed to version control to allow future builds to use them: If for example only 1 FPM within an EPM is changed only that FPM will get rebuilt. To make this decision `epkg` relies on the md5sum files from the most recent release Build.

Installing the Build Output to a Container and Creating a Zip Archive of the Package for Release

`epkg` has an module to automatically "install" a package to the container configured as *DevelopmentContainer* in */etc/epkg/settings.ini*.

To use it, simply call `epkg Install ...args...` where args are `--releaseType` and `--testVersion` (only if `--releaseType test`). The arguments should take the same values that were provided when building the package in the first place. In addition to the mentioned arguments there is also an additional argument `--createZip` which can be used to directly create a zip archive of all relevant files.

Since at Unicon the EPM builds and thus usage of the \`epkg\` tools are tightly coupled with a strict release process, there are a few things to consider:

- Since any `epkg` build only builds FPMs which got changed since the last release build, all EPM builds of a single package (and probably all packages) should be "installed" to the same container. Otherwise all FPM that were not rebuilt will be missing after `epkg Install` is executed
- Likewise `epkg Install --createZip` can only take FPM packages from the most recent build from the current working directory. All other FPM packages need to be present at the installation target or the zip archive cannot be created
- if `--releaseType test` is used during install existing packages will be overwritten without asking. This means that testing versions can be rebuild, installed and tested again and again with the same set of commands until an actual release build is performed

A Note about Package Signing

On package build or if being called with the sign command, epkg will create a signature for the created packages. The signature has the same filename as the package with a .sig appended at the end. The certificate and key combination can be configured in /etc/epkg/settings.ini.

When packages are imported into the container with ELIAS, it can verify package signatures. See udocs for managing certificates.

Package signatures can also be checked during installation/update of packages. Documentation on how to configure this can be found in udocs.

⚠️For signature verification to work it is important that the certificate is present at /setup/cacerts/**** during the installation/update process.

⚠️If the certificate is not present the installation/update process will fail

Please also note that package signature verification is not done during USB/PXE recovery because this is treated as a safe and trusted environment. It is therefore possible to provide certificates for package signature verification via package if that package is installed at recovery time. Otherwise the certificates can also be transferred via scout file transfer.

A Note about Dependencies

The *htop* binary requires some shared libraries. Usually you need to be very careful about adding correct dependencies in all FPM packages. In this case we (at least I do) know that all those libraries are part of the default eLux system and cannot be removed.

So we do not need to set any *requires*= in our *input/ebkepm*. Usually you need to be very careful to provide correct dependency information within the EPM.

eLux Internals

This section contains various details about the eLux Linux OS. Maintainers providing eLux packages should make themselves familiar with eLux such that their package does not conflict with any eLux conventions.

Users on eLux

- No matter what user got authenticated (if authentication is configured), all unprivileged session processes are running as user "elux".
- All user specific data is wiped at the end of the session.

eLux Filesystem Layout 101

In general eLux adheres to the Filesystem Hierarchy Standard (FHS) with UsrMerge. There are also some eLux specifics. The following table describes the root directories on eLux.

Directory	Content	Should files be installed here?	eLux Specifics
/bin	<symlink to /usr/bin>	No. Use */usr/bin*	
/boot	boot loader files (e.g., kernels, initrd)	Only by the kernel EPM. Other exceptions might exist.	
/dev	device files	No. Populated automatically by operating system.	
/etc	static system wide configuration	Yes. Static configuration goes here	This directory is part of an overlay filesystem (tmpfs) and as such is reset back to the installation state on each boot. This also means all files put here can be changed at run-time. Configuration that should be changeable in a persistent way, should be put in /setup (or symlinked there if it cannot be moved)
/home/elux	user home	Usually not, especially on reboot.	This directory is part of an overlay filesystem (tmpfs) and as such is reset back to the installation state on each boot. Belongs to user elux. No security critical information should be saved here, since it's readable/writeable by everyone.

/lib	<symlink to /usr/lib>	No. Use */usr/lib*	
/lib64	<symlink to /usr/lib64>	No. Use */usr/lib*64	
/lost+found	broken filesystem data	No	
/media	mount points for removable media	No	
/mnt	mount points for other filesystems	No	
/opt	Add-on application software	No. Avoid if possible. Instead, use */usr/bin*, */usr/lib/* etc.\ */opt* is used on distributions such as Ubuntu to install packages outside the Ubuntu ecosystem. On eLux, only installation through eLux/Scout is possible. As such there is no reason to deploy software to */opt*.	This directory is part of an overlay filesystem (tmpfs) and as such is reset back to the installation state on each boot. This also means all files put here can be changed at run-time. If binaries/configuration/... is put here with too broad access rights, security issues might arise.

/proc	process/kernel special files	No. Populated automatically by operating system.	
/root	home directory for the root user	No. Use */etc/* for global static configuration or */setup/* for changing configuration.	This directory is part of an overlay filesystem (tmpfs) and as such is reset back to the installation state on each boot.
/run	Run-time variable data	No. Populated automatically by operating system.	Part of a tmpfs, reset to empty on boot.
/sbin	<symlink to /usr/sbin>	No. Use */usr/sbin*	
/setup	Persistent, but changeable configuration data	Yes. Configuration goes here.	/setup is where all configuration that should be kept on reboot is put. ● applications can store their configuration files or other data meant to be accessible across reboots ● space on /setup is limited, so it should only be used for pure configuration data ● log files or even binaries should not be stored on /setup ● for applications with hard coded configuration paths we usually put the configuration here and create a symlink at the hard coded path

/sys	contains information about devices, drivers, and some kernel features	No. Populated automatically by operating system.	
/tmp	temporary files	No, temporary files are cleared with every reboot. Of course it is acceptable for your applications to use */tmp* for storing temporary files. But do not install files to */tmp*.	Part of a tmpfs, reset to empty on boot. Historically (pre-overlay) a lot of log files were put here, but this is slowly changed to put log files in the correct location */var/log*. You might still find some application logs here for now.
/update	files for the eLux update	No. Used only by eLux update mechanism.	
/usr	read only data	Not directly, use subdirectories.	
/usr/bin	binaries for all users	Yes. Store executables ran as user elux (user context) here.	
/usr/lib */usr/lib64*	system libraries	Yes. Store shared libraries here.	

/usr/libexec	special binaries that are not intended to be executed directly	Yes. For example, store init scripts invoked through some systemd service here.	
/usr/sbin		Yes. Store executables only ran as user root here.	
/usr/setup		Yes. Default/factory configuration goes here.	on each boot the configuration is synchronized from */usr/setup* to */setup* ● if a file exists on */usr/setup* but not on /setup, it is copied (folders are created) ● this happens for example after factory resets or after updates to copy configuration from newly installed packages ● configuration within */setup* is never overwritten by configuration from */usr/setup*
/usr/share	Shared data, like application icons, translations, ..	Yes	
/usr/share/ elux-init	scripts executed by systemd services	Yes	Service start scripts for systemd services should be put here

/var	variable files: files whose content is expected to continually change during normal operation of the system, such as logs, spool files, ...	Usually not. Of course it is acceptable for your applications to use */var* for storing temporary files.	This directory is part of an overlay filesystem (tmpfs) and as such is reset back to the installation state on each boot. Some log files are still in */tmp*. All newly added log files should be put in */var/log*

General tips for the file system

- Be as restrictive as possible with file permissions: e.g. a file with 777 could be changed or substituted by an attacker.
- Try to keep log files to /var/log/. If your package runs software as user elux (not root), you can create a subdirectory in /var/log/ as part of your package.
- **(Legacy)** in RP6 the chattr +a flag (only appending is allowed) is set on all files as an additional protection. This is not the case in eLux 7.

Development Only: Mounting File System as Read-Write

Usually, the eLux file system is mounted as read-only. As such, no permanent changes to the file system (think */usr/bin/*, */usr/lib/* etc) are possible.

If you have root user access to an eLux machine, you can temporary mount the file system as read-write. To do this, run

Adding additional files

```
fs open
```

as user root. Once you are done with these changes, you can relinquish the write permissions again by calling

Adding additional files

```
fs close
```

Please be aware that once fs open has been called, the system may not be in a defined state anymore. In particular, consider the following:

- Writes to any file/directory covered by an overlay can lead to *undefined behavior* (see kernel documentation).

- Changes to the overlay will not persist after a reboot. For example, since */etc/* is part of the overlay file system, changes to */etc/* (even with fs open) are not persistent. Only packages may install contents to */etc/*.

⚠This means that fs open may not be done in production. In particular, you should not use it in any scripts shipped by your package.

Starting Services at Boot / systemd Services

eLux uses systemd for service management which means that staring programs at boot time is as easy as adding a systemd service.

⚠Warning: eLux does not provide a SysV compatibility layer so scripts used in *.service* files **must not** be put in */etc/init.d/* or they will not be started properly

Startup scripts can either be placed in */usr/share/elux-init* or if they are expected to be executed manually as well in */usr/sbin.*

Offical systemd service documentation can be found at [freedesktop.org]()

Example

A simple service which starts after eluxd (i.e. after core eLux OS has booted) might look like this

/usr/lib/systemd/system/example.service

```
[Unit]
Description=An example service
After=eluxd.service

[Service]
Type=simple
ExecStart=/usr/share/elux-init/example

[Install]
WantedBy=multi-user.target
```

To enable/disable the service during package installation/removal, 2 scriptlets should be added to the FPM.

example_fpm.postInst

```
#!/bin/bash

systemctl --quiet enable example
```

example_fpm.preUninst

```
#!/bin/bash

if [[ $1 -ne 0 ]]; then
        exit 0
fi

systemctl --quiet disable example
```

If you do not include above `*.postInst` and `*.preUninst` scriptlets, your service will not be enabled after installation.

Starting User Programs At Login

Currently all user programs are started via numbered start scripts within */usr/share/X11/login.d/.*

There are also the folders */usr/share/X11/logout.d* which is executed at logout and */usr/share/X11/prepare.d* which is executed to prepare the X server start

The scripts are sourced during login, meaning they should not be executable and using `exit` will abort the login process. If a script should stop processing use the shell built-in return.

Please note that return codes are not checked in the callers.

To get a better understanding on how to write such a script and how to name it, existing scripts should be considered.

Setting Correct Dependencies

Most software one wants to package requires additional dependencies - usually in the form of shared libraries. As explained later in Collisions and corrupts=, no two EPM/FPM packages should provide the same file, so it's essential to set correct dependencies instead of re-packaging a library.

To discover what libraries are required you can use a command like `readelf -d /path/to/binary | grep NEEDED` (be careful, `ldd` resolves libraries transitively and will show more than you need)

Then the package providing a certain file or library has to be found. There are two approaches to this:

- search across all existing packages for the file name to find it within a *.install* meta file or search for the debian package name to find it within a *.debs* or *.thirdparty* meta file, but be aware that
    - files not packaged by us can - obviously - not be found this way

    - files might not be listed explicitly in the *.install* file, but with a glob, e.g. */usr/lib/** (that's one reason to avoid globs if possible)
- create an IDF with all EPM available or a subset of probable EPM/FPM (e.g. the *systemlibs* EPM contains many libraries) and use eluxbuild to find the provider of a file
    - files not packaged in an available EPM/FPM cannot be found this way either
    - eluxbuild is part of the *eluxbuild* FPM within the *devel* EPM
    - eluxbuild has trouble with files in symlinked directories

Once the correct EPM/FPM is identified it should be added to requires= in the FPM. Usually the EPM name and the FPM name should be added separated by a pipe I character. This helps administrators with finding missing dependencies.

If the file cannot be found it can either be added to an existing library or shipped in the EPM/FPM that needs it. It's a judgment call what to prefer, but as a rule of thumb all moderately common libraries should be packaged in an EPM like *systemlibs.*

Example

In this example, assume that we wish to package some application with *libpoppler* as a requried dependency*:*

```
elux> ls /usr/lib/x86_64-linux-gnu/libpoppler* -l
lrwxrwxrwx     1 elux   root     25 Feb 10 11:48 /usr/lib/x86_64-linux-gnu/libpoppler-glib.so.8 ->
libpoppler-glib.so.8.23.0
-rw-r--r-- 1 elux   root     415888 Jan 14 17:14 /usr/lib/x86_64-linux-gnu/libpoppler-glib.so.8.23.0
lrwxrwxrwx     1 elux   root     21 Feb 10 11:48 /usr/lib/x86_64-linux-gnu/libpoppler.so.118 ->
libpoppler.so.118.0.0
-rw-r--r-- 1 elux   root     3511008 Jan 14 17:14 /usr/lib/x86_64-linux-gnu/libpoppler.so.118.0.0
elux>  eluxbuild -qf /usr/lib/x86_64-linux-gnu/libpoppler.so.118
libpoppler-22.02.0.7.2503.0-1

# warning: /bin is a symlink to /usr/bin, so:
which eluxbuild
/bin/eluxbuild
elux> eluxbuild -qf /bin/eluxbuild
file /bin/eluxbuild is not owned by any package
# but
elux> eluxbuild -qf /usr/bin/eluxbuild
eluxbuild-7.2503.0-1
```

epkg: Usage, Configuration, Installation Scripts

This section provides a more detailed reference on epkg and the files that make up an eLux package (e.g. input/ebkepm).

Configuration for epkg

Configuration for epkg can be set in */etc/epkg/settings.ini*.

default /etc/epkg/settings.ini

```
[Signing]
keyPath=/etc/epkg/code-signing-key.pem
certificatePath=/etc/epkg/code-signing-cert.pem

[Global]
DevelopmentContainer=
Blocksize=1024
SizeOffsetInPercent=2
UniconCopyright=Copyright (C) %currentYear% Unicon GmbH. All rights reserved.
Proxy=

[WebElias]
Url=
```

- **Signing Section**: the option keyPath and certificatePath in the Signing section configure the certificate key pair used to create signatures either when building packages or when explicitly calling epkg Sign
- **Global** → **DevelopmentContainer**: Absolute path to the directory serving as container when calling epkg Install
- **Global** → **Blocksize**: Size of a block on the filesystem in bytes. Shipping default is 1024. DO NOT CHANGE.
- **Global** → **SizeOffsetInPercent**: Percentage added to the calculated size of a package. Epkg cannot completely reliably determine the size of a package after installation. Therefore this safety margin is added. Shipping default is 2. DO NOT CHANGE. (see <fpmName>.size)
- **Global** → **UniconCopyrigh**t: String that replaces %UNICON% parameter in package description, usually used for copyright= or vendor= in input/ebkepm
- **Global** → **Proxy**: Proxy url that is used when downloading resources specified with an http(s) url in .thirdparty files
- **WebElias** → **Url**: Configures the webelias url used when calling epkg Upload

Ways to Provide Files

When building packages with `epkg` there are 3 ways to provide files that should be packaged:

- the *input/* folder - files put here are directly included in the package, no further action required
- a *.thirdparty* meta file - contains paths to (archive) files containing files to package. (archive) files
- a *.debs* meta file - contains names of debian packages to pull from a debian repository

The latter two ways also require a *.install* file to list what files to take from that archive/debian package and where to put them on the file system.

Collisions and corrupts=

File collisions across several FPM packages should be avoided. In particular, two FPMs should not provide the same file.

If there is no other way than to overwrite a file the field `corrupts=` can be used in the FPM overwriting a file referencing the name of the FPM the file originally belongs to. If a package corrupting another package is uninstalled, the corrupted package will be re-installed.

⚠️Due to how the eLux installation progress works, a package marked with *PKGOPT_RECOVERY* must not be corrupted and must not corrupt other packages.

For example, currently these packages have the option set:

- BaseOS → minsystem
- BaseOs → plymouth
- BaseOs → libncurses

Scripts Executed During Package Build

There are currently two ways to execute scripts at build time of a package. The epm global elux/preCommands script and per FPM <fpm_name>.postCommands script. These scripts are executed with /bin/sh -c and know the special variable %root%. These scripts are usually used to create symlinks, adjust file permissions or mangle files in some special way (e.g. in the kernel epm, kernel modules are compressed at epm build time to save space). It is usually best to do these things at package build time because then the package database in the system will have all the information and there won't be any surprises (like space constraints that come from .postInst script creating additional files that could have been caught before).

elux/preCommands: This script is executed after all packages/archives from `.debs` and `.thirdparty` are extracted to `tmp/` and before files and folders are copied to the package tree in `output/` . The special variable %root% points to that `tmp/` folder.

<fpm_name>.postCommands: This script is executed after all files from an FPM are copied to the file tree in `output/` . The special variable %root% points to the FPM file tree, `./output/<epm_name>/<fpm_name>` .

Scripts executed before/during/after Installation

Since debian packages are just extracted using dpkg -x any control information (like postinst or preinst scripts) is lost during packaging. This means that if a package does important things, like setting the setuid flag on binaries in their postinst routines, this operation will be lost.

When working with debian packages `dpkg-deb -e <package.deb>` can be used to extract a package's control files to manually inspect them and decide if similar operations need to be performed in a postInst or postCommands script

The .preInst, .postInst, .preUninst and .postUninst hook scripts are a bit special and need extra care to work properly. The most important things first:

- If in doubt, **always** prefer a solution using .postCommands to installation scriptlets for one simple reason: everything done as a postCommand is part of the package. This includes installation size calculations, registering files in the package database, solving file corruptions on re-installation, and so on. So if there is any way to get away with not having a pre/post-inst/uninst scriptlet do it
- Not all output from scripts will be visible in the installation logs, so please take extra care to verify the results and don't rely on "I don't see any errors so it works"
  - RP6: set "InstallLogLevel" to see stdout/stderr in eluxman.log from script execution even if the overall installation succeeds.
  - elux7: all stdout/stderr output should always end up in the installation log files (update.*.log, installation.log, migration.log)
- Currently you can expect any shell options (e.g. `set -o errexit`) to propagate to **all** other scripts executed, so don't use them or unset anything you set on exit
  - it's harder to properly use `set -o errexit` : you need to use `set -o errexit,errtrace` and can then use a trap to remove those flags on quit
  - (the propagation of all shell options only happens with the terminal.ini parameter Development=true, but you can't rely on that not being set)
- depending on when in the overall install process your script is executed, don't expect the system to be fully functional: e.g. a script for minsystem cannot expect any applications or libraries from any other FPM to be present

Determining Final Installation State

epkg was originally forked from rpm . Please note, that this behavior was reconstructed from *current* RPM documentation, while our RPM was forked **~25 years** ago and it's unclear if it was ever updated since then.

If you want to perform certain actions only if a package gets uninstalled and not reinstalled you can check the first argument `$1` in any of the post/pre Inst/Uninst scripts: It contains the number of packages that will be left on the system when the action completes, so for example if a package is uninstalled $1 will be 0 in the preUninst and postUninst scripts. If the package is up- or downgraded it will be first uninstalled and then reinstalled, so $1 should be != 0 in all scripts. Since there can be cases where $1 is 2, you should always check inequality to 0 and not equality to 1.

The following table shows what value for $1 a certain script receives in which cases

	install	upgrade	uninstall
.preInst	$1 == 1	$1 == 2	not executed
.postInst	$1 == 1	$1 == 2	not executed
.preUninst	not executed	$1 == 1	$1 == 0
.postUninst	not executed	$1 == 1	$1 == 0

Special #include

There is a special feature to include scripts in your scripts

```
#include <path/to/somescript>
```

and epkg will replace the line with the script that is specified there. This works like the C Preprocessor.

This feature works only in the .preInst, .postInst, .preUninst and .postUninst hook scripts, but is not usable for .preCommands and .postCommands hook scripts.

Using Debian Repositories

To pull debian packages from a repository `epkg` needs to know which repository to use. Repositories are defined in */usr/share/unicon/suiteCPP.ini* in a format similar to the one in */etc/apt/sources.list*.

Each repository has an additional name field which needs to be given to `epkg --repositoryName=<name>` to use the repository.

Each repository definition consists of:

- a section header starting with *source* and ending with a number not used in any other repository definition
- a `name=` field with a unique name
- one or more `sourceX=` fields, where X is a number unique to this repository and the value is a repository URL line
    - an URL line consists of a priority, the archive type (usually *deb*), a repository URL, a distribution and one or more components
    - URL lines can contain extra specifiers in `[ ]` just like a debian/ubuntu *sources.list* entry (see examples)

Example

suiteCPP.ini

```
# valid
[source1]

name=jammy-main

source1=501 deb [signed-by=/etc/apt/trusted.gpg.d/ubuntu-keyring-2018-archive.gpg]
http://de.archive.ubuntu.com/ubuntu/ jammy main

# valid
[source2]

name=jammy-all

source1=501 deb [signed-by=/etc/apt/trusted.gpg.d/ubuntu-keyring-2018-archive.gpg]
http://de.archive.ubuntu.com/ubuntu/ jammy main restricted universe
source2=502 deb [signed-by=/etc/apt/trusted.gpg.d/ubuntu-keyring-2018-archive.gpg]
http://de.archive.ubuntu.com/ubuntu/ jammy multiverse

# invalid, [source1] already exists
[source1]
name=jammy-universe
source1=501 deb [signed-by=/etc/apt/trusted.gpg.d/ubuntu-keyring-2018-archive.gpg]
http://de.archive.ubuntu.com/ubuntu/ jammy universe

# valid, but broken, source1 used twice. The first source
[source3]

name=jammy-all2
```

```
source1=501 deb [signed-by=/etc/apt/trusted.gpg.d/ubuntu-keyring-2018-archive.gpg]
http://de.archive.ubuntu.com/ubuntu/ jammy main
source1=502 deb [signed-by=/etc/apt/trusted.gpg.d/ubuntu-keyring-2018-archive.gpg]
http://de.archive.ubuntu.com/ubuntu/ jammy multiverse restricted universe

# valid, but broken, name already taken. The first repository with this name will be used by epkg
[source4]

name=jammy-main

source1=501 deb [signed-by=/etc/apt/trusted.gpg.d/ubuntu-keyring-2018-archive.gpg]
http://de.archive.ubuntu.com/ubuntu/ jammy main universe
```

Bugs/Gotchas

Extra care has to be taken when editing the *suiteCPP.ini*, since epkg does not verify entries.

As a result misconfiguration may manifest in the following ways:

- an error like "epkg: error: Unknown repository: ..."
- an error from apt (e.g. "Malformed entry 2 in list file...")
- packages are not found
- entries are silently ignored and different package versions are found from another entry

The last one is probably the most severe one and may break packages.

Package Change Detection and Version Auto-Increment

During package builds, epkg is able to detect which parts of a FPM have changed. epkg only rebuilds those changed FPM. Thus it's very common that specific FPM of an EPM have a higher release number than other FPM.

To do this, epkg records checksums of all meta files (within *elux/*), files in the *input/* folder and archives/packages listed in *.debs*/*.thirdparty* in each FPM's *.md5sum* files. These checksum files should be checked in to version control such that subsequent *test* or *release* builds can re-calculate all checksums and compare to those saved values.

Converting EPM packages from eLux RP6 to eLux 7

1. Package Sources / Dependencies
    1. if you package files from Ubuntu, switch your package sources from Ubuntu 16.04/18.04 (Xenial/Bionic) to Ubuntu 22.04 (Jammy)
    2. a lot of upstream packages were restructured and it's very likely that you need to adjust the *.debs*/*.install* files of your EPM
    3. within eLux 7 packages have also changed. It's very likely that libraries that weren't available before are now part of a FPM within the *systemlibs* EPM or even part of BaseOs. See Setting Correct Dependencies
2. File **input/ebkepm**

    1. change "containers=1024" to 2048 (multiple times)
    2. drop any versioned dependency on like in "requires=%ELUX_BASEREQ|BaseOs >= 6.2104.0" or "requires=BaseOS >= 6.6" or 'requires=Foo>=1.2.3"
    3. drop any package conflicts like "conflicts=biostools<=1.3.2"
3. Package Contents
    1. move start scripts for service files from */etc/init.d/* to */usr/share/elux-init/* (RP6 patched systemd to allow scripts in */etc/init.d/*. For security reasons eLux7 uses an unchanged version of systemd)
    2. remove all and every chattr call in scripts in *input/* and in all *elux/*.postCommands, elux/*.preInst* and *elux/*.preUninst* files
    3. remove all kill/systemctl stop/systemctl start/systemctl daemon-reload/ calls in *.preInst/.preUninst* files
    4. Consider removing symlinks pointing to */tmp* (see eLux Filesystem Layout 101: there are several places which are writable at runtime in eLux 7 now)
    5. if your old package made any version checks, e.g. checking against strings like *RP6* remove those, too
4. build locally for container elux7: `epkg Build --container elux7 --releaseType test`
    1. fix any build errors that occur

epkg: Reference

Files

Which meta files are supported and what do they do. Empty files are not needed and can be safely removed.

Global

Global files

input/ebkepm

This file contains the metadata of an EPM and its FPMs. The section for the EPM starts with [EPM], the sections for FPM's with [FPMx] where x is number >= 1, unique within the EPM.

Section	Name	Default	Supported values	Description
EPM	category			Category of the package, see EPM Categories

EPM, FPM	conflicts		|-separated list of package names	List of package names conflicting with this package Make sure to use the correct EPM/FPM here: e.g. when overwriting a file from an EPM, list the exact FPM that file is shipped with (most of the time the EPM contains no files, so using the EPM here makes no sense)
EPM, FPM	containers			Bit field specifying the containers this package is compatible with (e.g. 1024 = rp6_x64, 2048 = elux7). If no valid containers are specified (e.g. containers=0), this EPM/FPM will be skipped during build
EPM, FPM	copyright			Name of copyright holder of the package
EPM, FPM	corrupts		Comma-separated list of package names	Packages corrupted by this package (e.g. overwritten config files) which have to be reinstalled after this package is removed Listing packages with installoption PKGOPT_RECOVERY (0x20) will break the installation and should not be done. (it leads to a plethora of issues like they will be installed/removed before chrooting, etc. pp)

EPM, FPM	description			A description of the package's purpose/content. About 1-3 sentences. This field is shown in ELIAS when you select the corresponding EPM or FPM in the package list in the "package information" area below the package list, alongside other meta information of the package. Please follow the additional guidance given in the EPM/FPM Naming Convention.
FPM	files LEGACY			Removed

FPM	hasGzArchive hasXzArchive hasXzArchiveWithGzSuffix			hasGzArchive: false or true; if true, a gzip-ed cpio archive will be created for this FPM. The suffix will be .gz . Create file trees to be included below input/(FPM-package-name)_gzArchive and/or input/(FPM-package-name)_gzArchive_(architecture), e.g. from baseos EPM: install_gzArchive and install_gzArchive. Specify debs, dirs, excludes, install, etc. for the archive below the elux directory as (FPM-package-name)_gzArchive.debs, (FPM-package-name)_gzArchive.dirs, etc., e.g installrp_gzArchive.debs, installrp_gzArchive.dirs . hasXzArchive: instead of gzip, use xz for compression. The suffix of the created xz-compressed cpio archive will be .xz . Use xzArchive instead of gzArchive in input and elux directories. hasXzArchiveWithGzSuffix: like with hasXzArchive, use xz for compression, but the suffix of the created xz-compressed cpio archive will be .gz . Use xzArchiveWithGzSuffix instead of gzArchive in input and elux directories. This option is necessary for tools like Elias which expect a .gz suffix.
FPM	includedMetaPackages LEGACY			To be removed

FPM	includedMetaPackagesLicences LEGACY			To be removed
FPM	installoption			See eLux Software Package Format - Install Options Warning: Setting PKGOPT_RECOVERY (0x20) here means that the package installation/removal is performed before chrooting during installation. This means that the package should not have a preUninst or postInst script (or those scripts need to be created taking into account that they run in a ramdisk not the final system; advice: just don't do it)
EPM, FPM	licence LEGACY			To be removed

EPM, FPM	majorVersion			Sets the major elux version (e.g.: 7.2404.0) which can be used in the version field as a variable (prefixed with "epm." or "fpmX."), e.g. "version=1.0.0.%fpm8.majorVersion%" The target majorVersion can be set via command line parameter --majorVersion and will be used for automatic version bumping in a release build. The algorithm is like this: ■ Is the package skipped? If yes, do nothing, else continue. ■ Evaluate the current version field, possibly expanding the variable if used ■ Set *majorVersion* field (and its corresponding variable) to the version given via command line ■ Evaluate the version field again ■ If it changed: reset release=0 ■ If *autoincrement* is enabled then release will be incremented to 1

EPM, FPM	name			Package's name ● This field is used to construct the EPM/FPM file name ● Allowed characters: Only letters, digits and underscore, must begin with a letter Uppercase letters should be avoided, use them for abbreviations - do *not* use them for CamelCase, use snake_case to separate words instead. ● for FPM names the EPM name should be prefixed to avoid collisions in the file system (e.g. use firefox_base, xorg_base) Please follow the additional guidance given in the EPM/FPM Naming Convention.
EPM	pkgoption			See eLux Software Package Format, chapter "EPM Options".
EPM, FPM	postinstall LEGACY			To be removed
EPM, FPM	postuninstall LEGACY			To be removed
EPM, FPM	preinstall LEGACY			To be removed

EPM, FPM	preuninstall LEGACY			To be removed
EPM, FPM	provides		SPACE-separated list of package names	List of package names being exported by this package. Only used if more than the package's name shall be exported.
EPM, FPM	release		non-negative integer	Release number, will be added to version. If V is the version and R the release, then V-R will be the full version string. In 99.9% cases the release will be autoincremented by the epkg due to a change in the EPM/FPM, so under normal circumstances this field should **never** be touched to increment it. When increasing/changing the version of a package, this field should **always** be set to 0. CAVEAT: This should almost never happen, since we usually change a FPM, which automatically bumps the EPM's and FPM's release, but when increasing the EPM's release number then be sure to increase the release numbers of the contained FPM's, too! Weird errors may arise otherwise.
EPM, FPM	requires		|-separated list of package names	List of package names required by this package. For each package a version restriction may be specified, like ...|BaseOS >= 6.8.0|...

FPM	size			*(wip)*
FPM	sort		non-negative integer, non-zero	Order how Elias presents that FPM. Usually you'd want to have the same number here as the FPM's number, like [FPM17] … sort=17. Doing it differently may has its applications, but most likely leads to more confusion and trouble than necessary.
EPM, FPM	summary			A few words describing the package's content. This field is shown in ELIAS when you select the corresponding EPM or FPM in the package list in the "package information" area below the package list, alongside other meta information of the package. Please follow the additional guidance given in the EPM/FPM Naming Convention.
EPM, FPM	vendor			The package vendor. In most cases we use %UNICON% which will be substituted by the value of UniconCopyright from */etc/epkg/settings.ini*
EPM, FPM	version			Version of this package, see release

elux/control

This file configures the behaviour of epkg for this package

Format: ini-File

Section	Name	Default	Supported values	Description
Global	architectures	Main architecture of eluxVersion	i386 (< eLux 7) amd64 (>= RP6_X64)	Space separated list of architectures the package should be built for. Will default to the first supported architecture of a container if left empty. If unsure: leave empty.
Global	checkMissing Files	false	true, false	If true epkg will check if a file in any source package is not included in any FPM. If files should be excluded intentionally see elux/allowedMissingFiles

elux/version INTERNAL

This file defines the Format version of the meta data.

elux/variables

This file defines variables which can be used in all other meta files. Each line contains a variable definition in the format: <name>=<value>. Variables can be referenced in other files with %<name>%.

Some of the key/value items of ebkepm are available as variables: The keys: name, majorVersion, version, release, summary, description are available with prefix "epm." or "fpmX.".

Examples

In section "fpm8" the key "name" is "foobar", then "%fpm8.name%" expands to "foobar".

Bugs

"Nested" variables like

folder=/home/build

installer_bin=%folder%/installer.sh

can only be properly resolved if all variables are used explicitly in a certain context. If %folder% is not used e.g. in a postInst script, %installer_bin% will resolve to '/installer.sh'

elux/stripExcludes

Each line contains a perl regular expression of files which should be excluded for stripping.

elux/allowedMissingFiles

Each line contains a perl regular expression of files which are allowed to be missing, missing means they are part of the source package but not of any fpm.

elux/preCommands

This file contains commands which should be executed before the files are copied to the package tree structure. The variable %root% can be used to reference the tmp/ directory. This file is executed with "sh -c".

elux/<epmName>_epm.md5 INTERNAL

This file contains all md5 checksums of data used by the epm. It is needed to check if the epm has changed.

elux/<epmName>_epm.preInst

Script which will be executed before the package is installed. See scripts for additional information and features.

elux/<epmName>_epm.postInst

Script which will be executed after the package is installed. See scripts for additional information and features.

elux/<epmName>_epm.preUninst

Script which will be executed before the package is uninstalled. See scripts for additional information and features.

elux/<epmName>_epm.postUninst

Script which will be executed after the package is uninstalled. See scripts for additional information and features.

Per FPM

Files that can/should exist for every FPM

elux/<fpmName>.debs

This file contains all debian packages which provide files included in this FPM. Each line contains the name of the debian package. During building it is tried to parse the licence information included in the debian package if it fails a warning is displayed and the licence needs to be entered manually. This can be done by adding a new line directly below the name of the debian package with the following content: "Licence: <Licence1>,<Licence2>".

Example

```
xauth
Licence: MIT
xfonts-utils
Licence: BSD-2-Clause,MIT
```

elux/<fpmName>.dirs

Each line in this file contains a directory which is created on installation.

Only use is to to create empty directories.

elux/<fpmName>.excludes

Each line contains a perl regular expression of files which should be excluded for packaging. This can be useful if wildcards are used in elux/<fpmName>.install.

The path against which is checked from install will always begin with a '/'

elux/<fpmName>.install

This file contains all files which will be included in the fpm. Each line contains a string in one of the following formats:

- <destinationPath>
- <sourcePath> -> <destinationPath>

The first format can be used if source and destination are the same. The <sourcePath> can contain * as wildcard. The <sourcePath> is relative to tmp/ and the <destinationPath> is relative to the package tree root. It is possible to specify the rights of the destination file if no rights are specified the destination file will have the same rights as the source file. Rights can be specified by adding a line beginning with "Rights: ". The format is the same as with chmod.

Example

```
usr/lib/test.so.2
Rights: 644
```

elux/<fpmName>.md5 INTERNAL

This file contains all md5 checksums of data used by the fpm. It is needed to check if the fpm has changed.

elux/<fpmName>.postCommands

This file contains commands which should be executed after the files are copied to the package tree structure. The variable %root% can be used to reference the package tree root.

This file is executed with "sh -c", but a shebang for another shell can be used within the file.

elux/<fpmName>.preInst

Script which will be executed before the package is installed. See scripts for additional information and features.

This script may not be combined with eluxman --root <somedir> ... because <somedir> will not be passed as root for the script in librpm/libepm.

elux/<fpmName>.postInst

Script which will be executed after the package is installed. See scripts for additional information and features.

elux/<fpmName>.preUninst

Script which will be executed before the package is uninstalled. See scripts for additional information and features.

elux/<fpmName>.postUninst

Script which will be executed after the package is uninstalled. See scripts for additional information and features.

elux/<fpmName>.size INTERNAL

Contains the uncompressed and compressed sizes of the fpm. The uncompressed size is automatically calculated during build or by running "epkg Size --calculate".

The compressed size needs to be obtained from the eluxman log. This can be done with "epkg Size --fromEluxManLog <pathToEluxManLog>". Current eLux versions do not use compressed files.

Format: ini-file

Section	Name	Default	Supported values	Description
SizeInfo	Compressed Files	0	any number	LEGACY Number of compressed files
SizeInfo	Compressed Size	0	any number	LEGACY Size in kilobytes on a filesystem with file compression support
SizeInfo	Compressed Version		any version string	Version of the package in eluxman.log which was used to get the size from
SizeInfo	UncompressedFiles	0	any number	Number of uncompressed files
SizeInfo	UncompressedSize	0	any number	Uncompressed size in kilobytes

Size calculation uses a configurable offset: Each FPM's size will be increased by ((size / 100) + 0.5) * SizeOffsetInPercent (*/etc/epkg/settings.ini*). The offset is currently configured to '*2*'.

The problem with this approach is that it checks FPM sizes in isolation and target devices might lead to different sizes depending on inode usage in folders like /usr/lib/* or /etc. It seems that we managed to be on the safe side with those 2% cushion, but in the end we "waste space". Hopefully a few MB of space won't be as important for future hardware/eLux versions.

elux/<fpmName>.thirdparty

This file contains all thirdparty packages which provide files included in this FPM. Each line contains the path to the thirdparty package followed by a line which contains the licence in the following format: "Licence: <Licence1>,<Licence2>". A thirdparty package can also be a http or https URL it will than be downloaded on every build of the package.

It is possible to extract to a specific folder by using "-> <path>". The path is relative to the tmp folder. Options are specified by adding a line beginning with "Options: ". Options is a comma separated list of the following options:

- NoExtract[2] - Do not try to extract the files just copy it to the tmp folder

Supported extensions:

- .zip
- .tar.*
- .tar
- .tgz
- .rpm
- .deb
- .xpi
- .bz2
- .bin (this is needed for java 1.6 and works only for java 1.6)

Example

```
/home/mirror/import/mozilla/ESR//38.5.2esr/firefox.tar.bz2
Licence: MPL-2.0

/home/storage/Development/public/ThirdParty/Fujitsu/deskflash/1_70_0049/biosset-1.70-elux.tgz -> deskflash
Licence: Fujitsu Technology Solutions
```

Bugs

There is currently an open bug for the extract-archive-to-specific-destination-feature.

EPM Categories

Categories can be used in ELIAS 18 to filter EPM packages.

At the moment following EPM Categories exist

Category	Use case	Examples
Application	Applications used by eLux users, e.g. Desktop Applications, Browsers, …	ica, citrix_extensions, vmwareviewclient, eluxrdp, firefox, chromium

System	System packages	baseos, xorg, pulseaudio, systemlibs, jre, perl, pulseaudio, kernel
Network	Proxy, VPN	squid, avahi, dynamicproxy, dyndns, netdrive, networkaccesscontrol
Miscellaneous	Custom packages	
Security	SmartCard Middelware, VPN socutions	cisco_secure_client, userauth, securitylibs, pcsc_lite, openldap
Multimedia	Audio, Video	gstreamer, fluendosw, videolibs, audiolibs
Driver	Support for new hardware	wacom, wlan, wlandrivers, baseprinter
Communication	VoIP, Unified Communication	citrix_hdxrtme, zoom
Utility		bios_tools, uefifwupd

Bits and Pieces

Testing Versions

Since epkg Version 1.1.5 the testing versions default to 'testing' (previously: '9999').

Allowed values for the --testVersion parameter are lower case letters or lower case letters followed by a ~ and then numbers.

Examples:

- --testVersion=mytestbuild
- --testVersion=elux~178

The build server will correctly fill in these values according to the type of build (e.g. Testing builds on master will be 'testing', bug/feature branch builds will be <project name>~<ticket number>).

In general one should not need to locally build EPMs and copy them to the container, but if you think you need this please make sure to use a --testVersion parameter that does not clash with anything the build server does, like for example your account abbreviation and maybe append your current ticket number, like --testVersion=sbr~6645

Feature Builds

In epkg version *22.04.911656.git668878a9* the flag --featureBuild was added, which can only be used for testing builds (–-releaseType Test). When this flag is provided epkg sets PKGOPT_FORCEUPDATE (see eLux Software Package Format) on all changed FPM.

This means these packages will always be updated and combined with the *CleanUpdatePartition* parameter it allows developers to work on/with feature EPM more efficiently.

Converting LEGACY

To convert a package tree in the new meta format the following commands need to be executed:

epkg NewFrom --pkgTree=<path to package tree>

This will create all needed files under elux/ but they are empty.

Fill *.thirdparty and *.debs with the input data

epkg ConvertFrom --pkgTree=<path to package tree>

This will fill the *.install and *.dirs meta files and create needed folders under input/

Multilib LEGACY

This feature has not been used for years and might not work as expected.

It is possible to build a package for multiple architectures.

At the moment the following architectures are supported: i386 and amd64. The architectures can be specified in elux/control.

In order to build 32 and 64Bit packages from the same source the epkg tool was extended. Those new files will be recognized:

Files

elux/multiLibExcludes

epkg checks if *.so or * under /lib and /usr/lib are copied to a multilib lib directory like /lib/i386-linux-gnu or /usr/lib/i386-linux-gnu and if not it errors out. But some packages need their modules to be placed directly in the lib dir and so the files contains a regexp per line to exclude files from this check. The path against which the regexp is checked is relative to the lib dir.

Example:

elux/pulseaudiobase.install:
usr/lib/pulse-8.0/*

elux/multiLibExcludes:
pulse-8\.0/.*

elux/<fpmName>.debs_<architecture>

This file is only used if the <architecture> matches the architecture of the package to be built. So these debs are only included in the given <architecture>. This file is merged with elux/<fpmName>.debs

elux/<fpmName>.dirs_<architecture>

This file is only used if the <architecture> matches the architecture of the package to be build. So this dirs are only created in the given <architecture>. This file is merged with elux/<fpmName>.dirs

elux/<fpmName>.install_<architecture>

This file is only used if the <architecture> matches the architecture of the package to be built. So these files are only installed in the given <architecture>. This file is merged with elux/<fpmName>.install

elux/<fpmName>.thirdparty_<architecture>

This file is only used if the <architecture> matches the architecture of the package to be build. So these third party packages are only included in the given <architecture>. This file is merged with elux/<fpmName>.thirdparty

input/<fpmName>_<architecture>

This folder is only used if the <architecture> matches the architecture of the package to be built. So these input files are only included in the given <architecture>.

Variables

Variables can also be used in epm/fpm install scripts.

Variable name	Scope	Description
%archLibDir%	Built-In variable the same as variables from elux/variables	This will expand depending on the architecture: i386 → i386-linux-gnu amd64 → x86_64-linux-gnu
__ARCH__	Folder name under input	This will expand depending on the architecture: i386 → i386-linux-gnu amd64 → x86_64-linux-gnu