VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

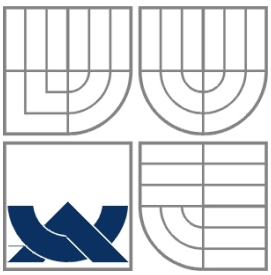
# MODULAR FILE SCANNER FOR RPM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

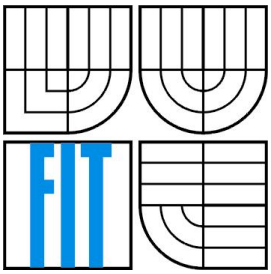
AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ MLČOCH

BRNO 2014



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

# MODULÁRNÍ SOUBOROVÝ SKENER PRO RPM

MODULAR FILE SCANNER FOR RPM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. TOMÁŠ MLČOCH

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN ZELENÝ

BRNO 2014

# Abstrakt

Balíčkovací systém RPM poskytuje pohodlné řešení pro správu a distribuci softwaru. Z pohledu vývojáře s ním pracujícího, je hlavní činností tvorba balíčků vhodných pro širokou distribuci. Tento proces obnáší řadu činností, které jsou pro velké množství softwaru podobné, či zcela totožné. Bylo by tedy vhodné mít možnost, tyto opakované činnosti co nejvíce a nejjednodušeji zautomatizovat, aby se usnadnila práce vývojářům a zmenšil prostor pro možné chyby. Cílem této práce je analýza požadavků, návrh a implementace modulárního skeneru do nástroje rpmbuild - části RPM starající se o tvorbu balíčků. Tento modulární skener bude poskytovat API ke sledování a modifikaci procesu sestavování balíčku a umožní snadnou tvorbu modulů doplňujících funkcionalitu pro zjednodušení a zefektivnění procesu balíčkování.



# Abstract

The package management system RPM is a convenient solution to software management and distribution. While working with RPM, the main developer's task is to build packages suitable for wide distribution. This process involves a lot of tasks that are similar, or even the same, for a wide range of packaged software. It would be useful to have a solution to automate this kind of tasks in an effective manner. The solution should lead to simplification of the packaging process and decrease the number of possible errors. The goal of the thesis is to analyze, design and implement a Modular File Scanner to an rpmbuild. The rpmbuild tool is a part of the RPM that takes care of package building. The scanner will provide an API to monitor and modify the building process and enable a simple implementation of extension modules that provide desired functionality.



# Klíčová slova

RPM, rpmbuild, správce balíčků, tvorba balíčků, modulární skener, Linux

# Keywords

RPM, rpmbuild, package manager, packaging, modular scanner, Linux


# Citace

Tomáš Mlčoch: Modular File Scanner for RPM, diplomová práce, Brno, FIT VUT v Brně, 2014

# Modular File Scanner for RPM

## Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Jana Zeleného.
Další informace mi poskytli Florian Festi a Panu Matilainen.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Tomáš Mlčoch
10.5.2014

## Poděkování

Rád bych poděkoval firmě Red Hat Czech s.r.o. za poskytnutou podporu a vedoucímu mé práce Ing. Janu Zelenému za ochotu a čas strávený konzultacemi.

# Contents

# 1 Introduction

A package management system (further referred to as PM system) is a collection of software tools to simplify software management and distribution. It is an effective and convenient solution to automate the process of installing, upgrading, configuring and removing software from operating system in a consistent manner [1]. PM systems are well-known from UNIX-like operating systems where they are commonly used. The most popular ones are dpkg, used in Debian (and derivatives), RPM, used in Red Hat Enterprise Linux, ABS, used in Arch Linux, and Portage, used by Gentoo Linux. From non GNU/Linux world let's mention Ports used in BSD-based operating systems.

In comparison to the canonical form of free software distribution as a source code [2], packaging brings in plenty of significant benefits. End users of PM systems don't have to deal with compilation, run-time/build-time dependencies, operating system configuration and similar issues. Moreover other useful actions like checksum and/or signature verifying could be done. All these things are managed by the system automatically in a straightforward way and through common utilities. Another benefit is faster installation because software in a package is already compiled and tailored to a target system. Few more benefits will be mentioned during the further chapters. Because of all these assets, a lot of end users prefer software distributed in form of binary packages. This is a good reason for software developers to make their software available in that form.

RPM is one of the oldest package managers used in Linux distributions. The most popular Linux distributions that use RPM are, the already mentioned, Red Hat Enterprise Linux, Fedora, CentOS, SUSE Enterprise Linux, OpenSUSE, Oracle Linux, Mandriva and Mageia. RPM was originally called Red Hat Package Manager. After adoption by other Linux distributions, the name was changed to the RPM Package Manager [4]. The license under which it is released is GNU GPL. The RPM is a very powerful tool and it makes possible to do almost everything that could be needed by the shipped software. The proof of this statement is the fact that today, almost 17 years after the RPM was originally written, it is still capable to satisfy requirements of the present-day software. This wide-spread in combination with all the mentioned packaging benefits makes the RPM interesting technology for developers.

Basically, the process of packaging by RPM is not much complicated, but the difficulty grows along with the complexity of packaged software and its special needs. Many tasks involved in the process are similar, or even the same for a wide range of packaged software. As an example of such common task is the need to call the `ldconfig` command after a new shared library is installed to the operating system. This task is common for all packages that have shared libraries inside. Nowadays, a packager of each such package has to explicitly call this command “manually”. For better user experience with RPM development, it would be handy if such tasks were fully automated, for our previous example, the command would be called implicitly without the intervention of packager.

The goal of the thesis is to analyze the current state, design and implement a modular interface named a Modular File Scanner (MFS) to the RPM. After that, implement one module and prepare developer documentation of the API. MFS should provide an easy way to extend and enhance the functionality of the RPM build process and improve the user experience. For example, the formerly mentioned use case should be solved by a single MFS module that detects a shared library in a package and autonomously adds the call of the `ldconfig` into the post-installation script of the package.

## 1.1 Structure

This first chapter introduces some basic terms, outlines the principles of the package management and compares the package management to the classical software distribution in form of source code.

The second chapter shows the RPM in more detail. It starts with the basic terminology related to the RPM and packaging in general and continues with main concepts and basic characteristics which make the RPM such a powerful tool. Finally, the architecture of the whole system is shown and the RPM file format is presented.

The third chapter is oriented on the build process with extra focus on the spec file format and its directives. Because the spec file is the crucial component of the build process, the description is extensive but, for the sake of clarity, not exhaustive. The last part of the chapter analyzes both the `rpmbuild` tool and library which implement the process. The findings from this analysis will be used in the design phase of the MFS.

The fourth chapter is dedicated to MFS itself. At first, some real-world use cases which are the motivation for the MFS creation are presented together with proposed solutions. Afterwards, the design of the MFS is shown. The architecture, work-flow and all main concepts that were designed are listed. Rest of the chapter describes the features of MFS API and justifies some given restrictions. At the end, some important modifications that were done to the RPM code are shortly summarized in order to give the reader some idea of the implementation.

The last chapter shows modules that were implemented during the work on the thesis. There are presented problems that are solved, the current and the new solutions are compared and the results are summarized.

# 2 RPM

Package management systems and their main concepts were already described in the introduction of this thesis. RPM itself was briefly presented as well. Now, we will look at the RPM more closely. Firstly, the important basic terminology is introduced. Philosophy and main goals are mentioned next. Architecture of the whole system is outlined afterwards and followed by a quick look at the structure of the RPM file format at the end.

## 2.1 Basic terminology

- **Package** – A bundle of software files and meta-data.
- **Source package** – In case of RPM, it is a package containing a source code, a prescription for building process (spec file) and optionally additional patches to the source code. Binary packages can be recreated from the source one at any time. Note that "source" in the name comes from the fact that the package contains source codes, the package itself is a binary file.
- **Binary package** – Usually a platform (architecture) dependent package containing a compiled application and all its files.
- **Spec file** – A prescription of commands and values to use during a package building process. It contains everything, starting with the name, short summary, version number, dependencies and ending with the configuration and build commands.
- **RPM Database** (also referred as rpmdb) – A database that holds information about all installed packages and their files, including file permissions, ownership information and so on.
- **NVR** (or N-V-R) – A string composed from the package name, version and release separated by dash. For example "*bash-4.2.45-4.fc20*" where the name is "bash", the version "4.2.45" and the release "4.fc20".
- **NVRA** (or N-V-R.A) – The NVR with an architecture name in the suffix. For example: "*bash-4.2.45-4.fc20.x86_64*".
- **NEVRA** (or N-E:V-R.A) – A string like the NVRA except it contains an epoch number separated by colon before the version number. For example: "*bash-0:4.2.45-4.fc20.x86_64*"
- **Capability** – The dependency mechanism of RPM is not based only on packages themselves. Instead, each package provides/requires/conflicts/obsoletes some capabilities. A capability is just a text string. Capabilities can be multiple types. In most cases, a capability is either a package name or a file dependency. The file dependency capability starts with a slash character `'/'`. Thanks to file dependencies, a package can require a specific file without care about which package provides the file. A capability enclosed in "`rpmlib(..)`" is a capability provided by RPM itself [10]. For dependency on a specific add-on module of a scripting language, the syntax "`perl(Carp)`" could be used. Where `perl` is the name of the scripting language and `Carp` is the name of the required module. In addition, a capability can have specified version and a package could provide/require/conflict/obsolete just certain version(s) of the capability specified by the comparison operators: =, <, >, <=, >=.

- **Build root** – A directory used during the build process where the software is installed after the successful compilation.
- **Build directory** – A directory where the software is compiled during the build process.

## 2.2 Philosophy and goals

Since the day when RPM was initially written by Erik Troan and Marc Ewing, it has become much more mature and complex, than it used to be, but the philosophy behind and the main goals still remain the same.

### 2.2.1 Ease of use

Important RPM design goal is to be easy to use for end users, which manage software installed on a system. Basic and the most common operations of package management are really simple. For example installation of a new package requires a single command `rpm -i package_file`, update of a package requires `rpm -U package_file` and erase of a package requires `rpm -e package_name`. In fact, the `rpm -U package_file` command could be also used to install a new package too, which can be used as another example of simplification provided by RPM. Such approach is much more easier than compilation and installation from a source code or installation by a script installer which can not do dependency resolution.

### 2.2.2 Package orientation and upgradability

A RPM package is intended to be a discrete bundle of program files, documentation and configuration. A package is the unit that is managed by PM system and the manipulation must be straightforward [4].

An upgradability means that it must be possible to install, remove and upgrade a single package regardless of other packages if these other packages don't depend on the given package in any way. It looks like an ordinary request, but before the RPM and its predecessors, for system-wide software management systems, it was not common. When a user wanted to update one single system program, the entire system had to be updated.

The upgradability also means that the process of upgrading to the newer version is done in a smart and safe way. The modified configuration files and similar customization that have been made should be preserved or at least saved if the modified configuration file must be replaced. The RPM allows to specify wide range of different scripts which run in different parts of the update process and can be used to make the process safe as much as possible.

### 2.2.3 Dependencies

Unlike the system-wide software management systems, where all software dependencies are satisfied by default, because all required components are included as part of the system, the package oriented systems, like the RPM, have to take dependencies into consideration. Due to this requirement, RPM tracks all dependencies required or provided by each package. Because of this dependency tracking, RPM would refuse to uninstall a package which is needed by other package(s) in the system.

Similarly, it would refuse to install a package which requires an unavailable capability. Information about dependencies is stored in the central rpmdb. The whole concept of the RPM dependency mechanism is very generic, because it uses, already described, capabilities and it is not limited to packages only.

The benefit of this system is that libraries do not have to be bundled with the programs, but each part can be distributed separately. This has a lot of advantages. The main thing is that libraries can be shared. This saves space on the storage and can lead to a better security. For example, when a new version of a library, which fixes some security bugs, is released, only update of the package that provides the library is needed. Other packages that require and use this library can stay untouched. The responsibility for quick reaction to security bugs is only up to the packager of the library, authors of software that use the library don't need to do anything.

### 2.2.4 Central storage with query capabilities

RPM stores all information about installed software on a single place. This central storage is called the rpmdb. It contains a list of all installed files including their sizes and permissions, a list of provided/required/obsoleted/conflicting capabilities and other meta-data like name, version, release, description, etc., for each installed package. As the name of the rpmdb suggests, it is a database. The advantage of storing this information in database is the ability to do queries [5]. Examples of the most common queries can be “What packages are installed on the system?”, “Where did this file came from? Which package provides it?”, “Which version of a package is installed?”, etc.

### 2.2.5 Verification

RPM has a lot of information about the files installed in the system and this information can be used for verifying these files. System administrator can use this feature to be sure that a program is installed correctly and its files are not damaged or modified. Even the dependencies of the package can be checked. Moreover, each package can define its own verification script which is used in addition to the standard verification. Subset of file attributes that can be checked during the standard verification process is: Owner, Group, Mode, MD5 checksum, Size, Symbolic Link String, Modification Time [5].

### 2.2.6 Support for multiple architectures

For example, Linux supports dozens of different CPU architectures[1] so a way how to indicate which architectures are supported by the software is necessary. RPM has this ability and support for multiple architectures by design. Each package has an attribute that says which architectures are supported. Package also can be a “noarch”, which means that the package is not limited to a particular architecture. Generally, with RPM it is easy to build the same package for several different architectures with small or even no modifications needed during the build process. The only limitation is the portability of the packaged software.

RPM also supports a thing called “Multilib”. Shortly, this allows you to install a single package for several different, but compatible, architectures to a single system at the same time. So on a 64bit

1 http://en.wikipedia.org/wiki/List_of_Linux_supported_architectures

system, which supports both 64bit and 32bit binaries, you can simultaneously install a package for the both architectures.

### 2.2.7 Pristine sources

Software for a specific system may require some modifications to the original source code. In order to be able to make all builds fully reproducible and transparent, RPM introduces a principle of pristine sources. This principle is optional and one can ignore it but it has some advantages. This principle allows to keep the all necessary modifications outside the original source code in form of patches. This approach is very handy, especially in the open-source world, where the situation that a packager and a developer are not the same person is common. In situations like this, it is possible that some changes may never be merged to the original source code. Then, the separated and reusable patches are great advantage.

Summary of their benefits:

- Changes needed for a specific distribution are separated from the original source code.
- This changes could be simply reused when a new version of the original application is released.
- Users can see which modifications have been done.
- Author of the software can use the changes and merge them to the original source code.

This concept is generally recommended and also required in some projects, for example in Fedora[2], but generally the whole decision if either to use pristine sources and patches or to directly modify sources is only up to the packager.

### 2.2.8 Easy builds

RPM makes an effort to simplify the packaging process and simultaneously preserve its high flexibility. The most difficult and time-consuming task during packaging is creating of an initial build prescription called the spec file. But this is a one-time issue, the next part of the process is only one single RPM command. So after the spec file is created, each next task, during the maintenance life of the packaged software, is straightforward. For example, if a new version of the software needs to be packaged, only a change of the source tarball and version number in the spec file are the actions that are needed.

Another simplification for a packager is that RPM allows to automatically generate the provided and required dependencies for shared libraries and other files contained in a given package. This detection is based on `ldd` command and is enabled by default. In addition, a similar detection is used for scripting language dependencies for Tlc, Python and Perl and even for Java packages. This makes the rpm build simpler and eliminates an amount of possible dependency issues.

Nevertheless, the requirements of some modern software are much more complex than the time when RPM was designed. Although RPM is generic enough to be able to package that kind of software, the spec files begin to be much more complicated and a single spec file may contain several hundred lines[3]. So nowadays, the initial part of packaging process, which is the preparation of a spec

2 https://fedoraproject.org/wiki/Packaging:Guidelines

3 http://pkgs.fedoraproject.org/cgit/eclipse.git/tree/eclipse.spec

file, is not so simple as back then and the outcome of the thesis should contribute to make the build process easier again.

## 2.3 System architecture

RPM is a complex system that is composed from several different parts. Following description is just a brief summary focused on the main parts and their relations. The parts related to the object of the thesis will be described more complexly afterwards.

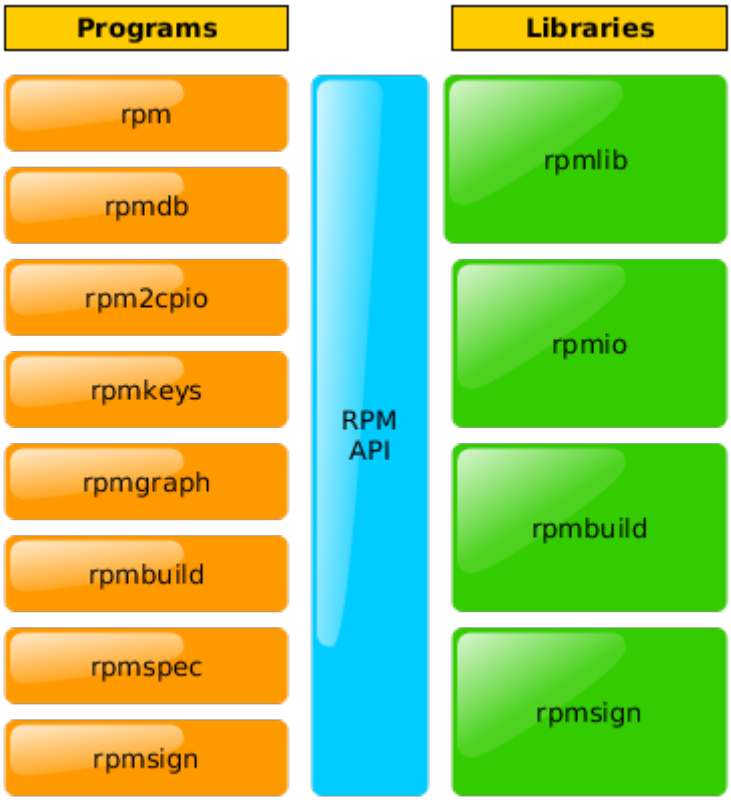



*Figure 1: RPM system composition.*

The diagram 1 shows the basic system architecture. The RPM binaries (programs) are shown on the left side of the diagram. These programs use the RPM libraries depicted on the right side of the diagram. The connection point of this two parts is the public API provided by the libraries.

**rpm**

The command-line interface. It provides all common operations with packages and query capabilities to the RPM database.

**rpmdb**

The database utility that provides a command-line interface for manipulation with the RPM database. I.e. initialization of a new empty database or rebuild of already existing database.

**rpm2cpio**

Converts an rpm package payload to cpio archive and result prints to the standard output. The cpio is a file archiver utility and file format.

**rpmkeys**

A tool for manipulation with the RPM keyring. I.e. add/remove keys, show details of a key, check signature of a package.

**rpmgraph**

A utility to display a dependency graph of an RPM package.

**rpmbuild**

A tool used to build both binary and source RPM packages. Analysis and modification of this tool and the related library is the main goal of the thesis.

**rpmspec**

A tool for querying a spec file.

**rpmlib (librpm.so)**

The main RPM library for manipulating with packages. This library contains API for all core parts like header parsing, database manipulation, payload processing and so on.

**rprmio (librpmio.so)**

A library for accessing files, including the remote files. Supported protocols are ftp, http, https, hkp.

**rpmbuild (librpmbuild.so)**

The library for manipulating with spec files and building packages. Modification of the library is the target of the thesis. Detailed description of the library's features, capabilities and internal implementation will be covered in the chapter focused on the build process.

**rpmsign (librpmsign.so)**

A library for signing packages.

## 2.4 RPM file format

This chapter briefly describes the format of RPM package. Because modification of RPM package building process is the goal of the thesis, the format of the packages itself should be mentioned too. On the other hand, there is no need to go in-depth. These who are interested in more details should see [6] or [4].

The RPM file format contains four main sections:


*Figure 2: RPM file format structure*

**The lead (file identifier)**

This part contains basic information about the rpm package. First four bytes have values [0xED, 0xAB, 0xEE, 0xDB] and are so-called "magic number". These values are used to show that a file is an RPM package by the tools like the `file`[4] command. Next information in the lead is major and minor version of the used format, package type, architecture and designated OS. This section is now deprecated and could be safely ignored [3]. All data from this section is available in the header section.

**The signature**

After the lead section, the signature is present. It contains data used for package's header verification. The data in this section are used only for header verification and they are not relevant for the payload.

**The header**

The header contains meta-data about the packaged software and the package itself like the name, version, homepage address, supported architectures, contained files, changelogs and so on. Information in the header is structured and can be accessed by header tags[5]. Size of this section may vary and it depends on number of used tags and volume of the stored information.

**The payload**

The payload section contains the packaged files. All previous sections are just meta-data. The format of payload section could be explicitly specified, but currently only supported, and implicitly assumed, is the CPIO format. The payload can also be compressed to save some space. Supported compression formats are gzip, bzip2 and XZ (LZMA) [16].

4 http://www.darwinsys.com/file/

5 http://rpm.org/api/4.11.1/group__rpmtag.html

# 3 Package building process

The RPM package building process is fully automated and driven by a spec file. From a packager point of view, the process is straightforward and simple. This chapter looks at the process more closely and reveals its implementation details.

**The input data**

- **Spec file** – A file that drives the whole packaging process. The spec file details are introduced in the second part of the chapter.
- **Source code(s)** – An unmodified, upstream source code. Mostly in form of an archive called a "tarball". If suitable, multiple source archives could be present.
- **Patches** – The original source code may need some modifications to properly address the target system (different default paths, different libraries, etc.), to properly compile in a build root, to fix some known bugs, etc. All these necessary modifications should be supplied in form of patches in order to keep the original sources untouched.

**The outputs**

- **Source RPM package** – A package with the "`.src.rpm`" suffix that contains all the above-mentioned inputs and can be used to rebuild of the binary RPM packages.
- **Binary RPM package(s)** – A package with a compiled software. Multiple binary packages can be produced during a single build process. For example, one package can contain a shared library, the next one the program, another one documentation and the last one the development header files. How many binary packages should be produced depends on the software, packager decision and the target system guidelines.

**Overview of the process**

1. **Spec file parsing**
2. **Check of build time dependencies** – All build-time dependencies must be installed on the system before the build.
3. **Preparation of source codes** – Spec's `%prep` script: Unpacking, applying patches, etc.
4. **Configuration and compilation** – Spec's `%build` script: Usually a call of Autotools, Cmake or similar configuration/compilation tool.
5. **Installation to the build root** – Spec's `%install` script: Usually calls `make install` or a similar supplied install script.
6. **Packaging of source files** – Collecting of all input files and creating of the source package.
7. **Packaging of binary files** – Collecting of files installed to the build root and selected files from build directory (mostly documentation, README, etc.) and creating of the binary package(s).

## 3.1 Spec file

The spec file is the key to the RPM packaging. The whole build process is driven by it. The spec includes all mandatory meta-data like path to the tarball, package name, version, release, license, summary as well as a lot of optional information like vendor name, packager name, software homepage, extensive description, changelog etc. Besides this meta-data, the spec has to contain commands (scripts) necessary to proper configuration, build and installation of the software. Moreover, it could consist additional scripts and triggers attached to actions, like the installation or uninstallation of a package.

At least basic familiarity with the spec file structure and knowledge of some fundamental directives is important to understand the range of RPM's possibilities and the needs that have to be respected during designing of MFS. This chapter introduces selected, most important, parts of the spec file. For more extensive description see [5], [8] or [9]. Basic example of a spec file can be found in the appendix A at the end of the thesis.

### 3.1.1 Basics

The Spec file consists of four main parts.

#### Section Preamble

The preamble section contains meta information about the package. It is implicitly at the beginning of the spec file and it is designated for macros definition and main package description.

Preamble's directives use simple syntax of a tag name, a colon and a value. For example: "`Summary: Sample package`". Tag names are not case-sensitive and some tags can be specified multiple times with values in different languages. The language must be specified in parenthesis between the tag and the colon like: "`Summary(cs): Ukázkový balík`". Tags that support such multilingual values are `Summary` and `Group` [7].

#### Section Scripts

Scripts are very important part of each spec file. The build-time ones are used to control the build process and the install/uninstall-time ones together with the triggers are used during the package lifetime to ensure proper function of the software and consistency of the system environment in relation to the given package. The install/uninstall-time scripts and triggers are also called the "scriptlets" in the context of RPM.

Scripts, except the build-time ones, also accept several attributes. These attributes must be specified right after the script name. For example "`%pre -n subpkg -p <lua> -f script.lua`". Where `-n` is used to specify a sub-package where the script belongs to. If `-n` is not used, the main package is assumed. The `-p` is used to set an interpreter of the script. Names of the internal RPM interpreters are in angle brackets "<>". For an external interpreter, the full path is used. Currently the only internally-supported interpreter is for the Lua language [7] [11]. The `-f` specifies a file-name in the build directory, from which the script will be loaded.

**Section File list**

Here are listed all files which the package should contain. These files are picked from the build root during the build process. Except the files themselves, other meta-data for files like ownership, rights, attributes, etc., can be specified in this section.

**Section Changelog**

The list of changes in the package in its individual versions.

**Notes on macros and comment syntax**

Macros in RPM are mostly used for a simple text substitution, but they also can have parameters[6] and can be used in definitions of other macros. Often, macros hold and provide some configuration settings during the build. For example, RPM has predefined[7] and available a lot of macros with paths to the important system directories like `%_sysconfdir` for a directory with configuration (e.g. `/etc`), `%_bindir` for a directory with executable binaries (e.g. `/bin`), etc.

A packager also can define new macros manually. The simplest static value can be defined as "`%define gitrev c69642e`" and accessed by "`%{macro}`" or "`%macro`" constructions. For example "`Source: example-%{gitrev}.tar.xz`" or "`Source: example-%gitrev.tar.xz`".

Macros bring along a possibility of conditional builds, the block clauses `%if`, `%else` and `%endif` can be used in the spec file to perform only selected commands depending on the selected conditions. For one-line expressions, conditional macros could be more appropriate. Syntax of a conditional macro is: "`%{?macro_to_test: expression}`" [4].

All lines with pound sign symbol '#' at the start of the line are considered as comments. Be aware that macros inside comments are still expanded. To comment a line with a macro properly just double the percent character, for example: "`# A comment line with a %%macro`".

### 3.1.2 Mandatory directives

These preamble tags are mandatory and must be specified in every spec file.

**`Name`**

The base name of the package. It must contain only alphanumeric characters and characters that are allowed in a file-name except the white-space characters. The name, unlike the version, can contain a dash character '`-`'.

**`Version`**

A string with version number. For example "`3`", "`1.2`", "`5.1.2.3021`", etc. Version numbers are used for comparison, for that reason the use of code-name string as the version value must be avoided. Such and similar additional version strings can be placed to the release tag if needed. Version must not contains a dash character '`-`', which is used as a separator in the N-V-R strings.

6 https://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/RPM_Guide/ch09s07s04.html

7 https://fedoraproject.org/wiki/How_to_create_an_RPM_package#Macros

**Release**

The release should be an integer number that symbolizes the number of recreation of a single version of the package. For example, when a change in the spec file or in a patch was done, but the software version remained the same. The release number should be incremented every time when a new package for the same version is built and it should be set to "`1`" again when a new version of the software is released.

**Summary**

A short, one line, summary with basic information about the package.

**License**

A standard abbreviation of the software license. For example "`GPLv2`", "`GPLv2+`", etc., or "`GPLv2 or BSD`", "`LGPL and BSD`", etc., for software with multiple licenses.

**Source or SourceX**

A URL where the tarball with the original source code can be downloaded or just a filename of the tarball in the `SOURCES/` directory inside the directory structure of rpmbuild. In fact, only the last part of the URL, the part with the file-name, is used. This tag can be specified multiple times with different numbers in suffix (the X is only a placeholder for a number here) if the package is composed of multiple sources.

### 3.1.3 Optional directives

Selection of few optional but important tags available in the preamble section.

**PatchX**

A file-name of a patch that should be applied on the decompressed source code. The X at the end must be replaced by a number. Multiple patch tags, with different numbers in suffix, can be used.

**Epoch**

A tag used to clarify version history of the software, when significant changes in versioning happened. That is when the next version has a less numeric value than the earlier one. Implicit epoch is zero. A package with higher epoch is considered newer, regardless its version or release. Epoch should be used only, when there are no other possibilities left.

**URL**

Homepage of the project.

**Group**

One of the groups listed in `/usr/share/doc/rpm/GROUPS`. This information could be used by some tools to divide packages by categories. For packages in Fedora the Group tag is deprecated and ignored, since Fedora 18 [8].

**ExcludeArch**

Architecture for which the package will not be built. Reasons for use of this tag can be either software does not support the architecture or serves no purpose on it. This tag can be specified multiple times.

**ExclusiveArch**

Opposite of the `ExcludeArch` tag. It tells RPM to build the package only for this architecture. This tag can be specified multiple times.

**Provides**

Each package provides itself by default. This tag can be used to specify another provided capability. This tag can be used multiple times.

**Requires, PreReq**

It tells RPM that this package requires a specific capability. The `PreReq` is similar to the `Requires` tag, except that the dependency must be installed prior to the given package. Both tags can be used multiple times.

Nowadays, the `PreReq` tag is deprecated. Instead of it a special notation of the `Requires` tag with parentheses should be used, i.e. "`Requires(pre,preun):`". The keywords that can be used are: `interp`, `preun`, `pre`, `postun`, `post`, `rpmlib`, `verify`, `pretrans`, `posttrans`. Multiple keywords separated by comma can be used.

**Conflicts**

A capability which the package conflicts with. RPM shows an error message and denies to proceed the installation of the given package if it has capabilities that conflict with the capabilities of another package that is already installed in the system.

**Obsoletes**

It tells RPM that the package provides capabilities that make this specific capability obsolete. This tag is used mostly when a package is renamed and thus the new package, with the new name, obsoletes the old one. The obsolete packages will be removed during the installation of the package that marked them as obsolete.

**BuildRequires, BuildConflicts, BuildPreReq**

Similar to the `Requires`, `Conflicts` and `PreReq` tags, but related to the build-time. Note the `BuildPreReq` tag is deprecated.

### 3.1.4 Build-time scripts

Build-time scripts, sometimes referred to as the build section of a spec file, which are listed below in this chapter are related only to the package building process and have not further use during the package lifetime.

**%prep**

Commands in this section are used to prepare the software sources for building. The most usual actions in this section are unpacking of sources and applying the patches. But any other necessary actions to get the sources ready to build should be done here.

**%build**

The `%build` script enters the process when the `%prep` scripts ends. It usually calls the configure script and starts the compilation, usually using the `make` command.

**%install**

Install script should perform installation of the compiled software. In an ideal case, it is just a call of the `make install` command.

**%check**

Script in this section is used to verify the compiled code. This section is used when the software contains its own test suit (with acceptance tests and/or unit-tests). If used, typical commands are `make check` or `make test`.

**%clean**

Commands used to clean the build root. Usually it contains only call of `rm -rf $RPM_BUILD_ROOT`. This section is no needed in spec files of fedora packages, since Fedora 18 [8].

### 3.1.5 Install/Uninstall-time scripts

These scripts (scriptlets) are used when the package is being installed or erased from the system as well as when the package is being updated.

Each of these scripts receives a number as a first parameter (`$1`), the number is a count of installed versions of the given package in the system at the time when the current action will be finished. If a new package, which does not exist in the system yet, is installed the number is 1. When an already installed package is updated, the number is 2 or higher because the previous version already exists in system, during the installation of the newer version.

**%pre, %post**

The scripts that are being run right before/after installation of the given package.

**%preun, %postun**

The scripts that are being run right before/after the erase of the given package. A typical postun action is call of the `ldconfig` command if one or more shared libraries have been removed.

### 3.1.6 Triggers

Triggers are special scripts attached to the install and uninstall actions of other packages. They allow the given package to be properly configured according to the rest of system environment. The syntax of trigger is a little bit more complex than the syntax of the previous scripts, but it is easy to understand anyway. The syntax is "`%trigger{un|in|postun} -- <trigger>`" where the `<trigger>` is the name of a package. Multiple package names separated by commas can be used. Moreover a comparison operator and version can be used for each listed package, i.e. "`%triggerin -- python > 2.6, perl < 5.1`". For more information about the triggers see [12].

### 3.1.7 Verification script

The `%verifyscript` is used every time when RPM is verifying the given installed package. RPM itself checks the files and their owners, rights and attributes, so the script should be focused on other things that are necessary for right function of package. This script is fully optional and it is only up to the packager what should be checked. All errors found by the script should be printed to the standard error output.

## 3.2 rpmbuild

This chapter describes the build process implemented by the `rpmbuild` in detail. The `rpmbuild` tool parses a spec file and builds RPM packages. It has a lot of options for fine-tuning the process. For simplicity, a basic state that corresponds to the invocation by the "`rpmbuild -ba specfile`" command will be considered. The whole process is spread over two main parts: parsing and building. The connection between the parts is the `rpmSpec` structure, that is filled during parsing and used during building.

The internal `rpmSpec` structure is not a fully abstract representation of the spec file. For example, file entries from the `%file` section of spec are stored as a list of almost unprocessed strings, only macros are expanded. Each element of the list corresponds to a single line from the `%file` section. Directives like `%attr()`, `%verify()`, `%doc`, etc., are not taken into consideration. Their processing is postponed to the build-time, to the almost last phase, where the files for the package are collected. In this phase, the lines are parsed again, now more thoroughly, the parsed data evaluated and the files located.

### 3.2.1 Parsing spec file – rpmSpecParse() / parseSpec()

A given spec file is parsed in this phase. The parsing itself is divided into several functions which match sections of a spec file. Reading of the spec file is done via a common function which is used by all other parsing functions. This function takes care of reading lines and evaluating macros, including the advanced conditional constructions.

At first, the preamble section is parsed by the `parsePreamble()`. This function is also used to parse sub-package definitions which can be introduced by the `%package` directive. For each package a `Package` structure is created and appended to the `rpmSpec`. The `Package` contains

a `Header` structure which is the same header that is later used for the resulting package. Each directive from the preamble section is parsed by the `findPreambleTag()` and handled by `handlePreambleTag()`. Most of the data from this section are stored directly to the `Header`. At the end of the `parsePreamble()` function, some common header fields (like epoch, version, release, license, changelogs, etc.) are copied from the header of the main package to headers of sub-packages and basic sanity checks are done.

The `%prep` section is parsed by the `parsePrep()`. It handles and expands `%setup` and `%patch` directives and stores the expanded script to the `rpmSpec`.

The `%build`, `%install`, `%check` and `%clean` sections are parsed by the `parseBuildInstallClean()` function. This function is very simple, it just takes all the related lines for each section and store them as a single string (each section has own string) to the `rpmSpec`.

Other scripts: `%pre`, `%post`, `%preun`, `%postun`, `%pretrans`, `%posttrans`, `%verifyscript`, `%triggerprein`, `%trigger`, `%triggerin`, `%triggerun` and `%triggerpostun`, are parsed by the `parseScript()`. It parses and validates the script options and puts the script defined in the spec directly into the package's header. The scripts which are referenced by the `-f` parameter, are not loaded directly, but the paths are stored to the `rpmSpec` and they are loaded during the later build phase.

Entries from `%files` section are parsed by the `parseFiles()`. This function just reads the lines and stores them to the package's internal list. Each line is a single record in the list. If the `%file` section has defined a manifest file (`-f` option), this manifest is not loaded directly, but only the path is stored to the `Package` structure. The loading itself is done during the building. Note that the lines from the `%files` section are not parsed, they are just stored as a list of strings. Only exception is that macros are expanded, but it is all, no further parsing or validation is done.

For other sections, the situation is: The `%changelog` is parsed by the `parseChangelog()`, the `%description` by the `parseDescription()` and the `%sepolicy` by the `parsePolicies()`.

The parsing of the spec file can be done multiple times, if the main package contains one or more `BuildArch` directives for multiple architectures. When the `BuildArch` is encountered for the first time, the current parsing is aborted and the new one is issued for each architecture which is listed and supported. Each subsequent parsing has a `%{_target_cpu}` macro defined to appropriate value. But at the end, only the first fully parsed `rpmSpec` structure is returned. A comment in the source code says that this is a hack, because there may be more than one supported architecture listed in the `BuildArch`. As a result, packages for multiple architectures cannot be generated simultaneously during a single run or `rpmbuild`.

At the end of the `parseSpec()` function, the last sanity check is done. It checks if all packages have description tags filled. Then, the target platform, CPU and OS are inserted into the headers of all packages and the header for the source package is generated. At this point the parsing of the spec file is finished and the `rpmSpec` structure is filled. In the next step, this structure will be used for building packages.

### 3.2.2 Building – rpmSpecBuild() / buildSpec()

The input to the building phase is the filled `rpmSpec` structure. The building is divided into several separate functions.

At first, the build-time scripts `%prep`, `%build`, `%install` and `%check` are run by the function `doScript()`.

The next step is processing of source files in the function `processSourceFiles()`. In this step, all source files (sources and patches) of the main package together with icons from sub-packages, if available, are put to the internal file list of source files. Then, each file from the list is analyzed by the `stat()` and proper owner, group and permissions are determined. This determination is based on the file status and the values defined by the `%srcdefattr` macro. At the end, the file records are added to the header of source package using the `genCpioListAndHeader()`.

Binary files are processed right in the next step by the `processBinaryFiles()`. This function passes through all packages defined in the `rpmSpec` and for each of them puts the name of the source package into its header. Then it processes their files and automatically generates dependencies from them. Processing of the files is done by the `processPackageFiles()`. In this function, the file manifests (defined at `%files` section by the `-f` option) are loaded and their entries are appended to the package internal list of file entries. After that, the list is parsed by the set of functions:

- `parseForVerify()` - Searches for either the `%defverify` or the `%verify` directive.
- `parseForAttr()` - Searches for either the `%defattr` or the `%attr` directive.
- `parseForDev()` - Searches for the `%def` directive.
- `parseForConfig()` - Searches for the `%config` directive.
- `parseForLang()` - Searches for the `%lang` directive.
- `parseForCaps()` - Searches for the `%caps` directive.
- `parseForSimple()` - Searches for the file names, because multiple file names can be on a single line.

After that, the special files like documentation, documentation directories, license, public keys, etc., are handled by a special way, while the other files are handled by the `processBinaryFile()`. This function canonizes file path and calls the `addFile()` function for each file that matches the path, because the path could be a glob pattern and thus could match multiple files. The `addFile()` function calls `stat()` on the file, selects the appropriate rights, ownership and appends the file into the package's internal list of file records. When the files for a package were processed and the control is back, at the end of the `processPackageFiles()`, then file records are generated and put to the package's header by the function `genCpioListAndHeader()`.

Next step is to call of the `processBinaryPolicies()`, that has effect only if RPM is compiled with SELinux support.

Calls of the `packageSources()` and the `packageBinaries()` follow. During these calls, the packages are written out to the disk. Before it, few last pieces of information like version, build host name and build time are put into the headers and then the `writeRPM()` is finally called. For binary packages, except the mentioned version, etc., the scripts in the headers are updated too.

This is the phase when the scripts are loaded from external files if the `-f` attribute was used at the script definition (the loading of these files was postponed - only their paths was stored).

The final part of the build includes call of the `doScript()` for `%clean` script and removing of build tree. Then, the build phase is complete and packages are written out.

# 4 Modular File Scanner

The Modular File Scanner should offer a generic interface to the `rpmbuild` library to allow the creation of modules which enhance the package building process. It would load third party modules that extend the `rpmbuild`'s capabilities and should lead to simplification of spec files. The biggest benefit is the modular architecture that allows to separate solutions to problems into independent modules and well documented MFS API that should make the development of new modules easy and comfortable.

At the beginning of the chapter, several use cases which are motivation for the MFS implementation will be shown. The rest of the chapter is focused on the analysis and design. At the end, the designed API is presented and the most needed changes which were done to RPM are mentioned.

## 4.1 Use cases

This chapter shows a few of the most obvious issues which could be efficiently solved by MFS modules. Currently, some of the listed problems must be solved by the packagers themselves, some of them are solved by the RPM but, as will be described, the solution is not optimal.

### 4.1.1 Call of `ldconfig`

The `ldconfig` is a system tool that determines the run-time links required by shared libraries and creates a cache. The cache contains links to the most current versions of the libraries and is used by a run-time linker to speed up the loading of programs which use these shared libraries [13].

Each package which adds or removes a shared library to/from the system, has to call `ldconfig` command to update the cache after the library is installed or removed. For packagers it means that `%pre` and `%post` scripts must call the `ldconfig`.

This routine could be easily automated by a module which identifies a shared library in the package and adds the calls of `ldconfig` into the scripts autonomously. Several clues can be used for the detection of a shared library. For example: Path of the file (`/lib`, `/usr/lib`), suffix of the file (`.so`), magic bytes of the file (`libmagic` library), etc.

### 4.1.2 Generation of debug information sub-package

Debug information sub-packages contain debug symbols striped from compiled ELF binaries and related source code. Because binaries with debug symbols are usually much bigger than the ones without the symbols, and because the debug symbols are not usually necessary, unless there is a problem with the binary, it is useful to keep the symbols separated in a standalone package.

Currently in Fedora, the debug information sub-packages are generated by a group of macros which, if enabled, add the sub-package to the spec file and adds call of a script that strips the symbols after the binaries are installed by the `%install` script.

This routine could be easily done autonomously by a module that recognize binary files in the given package, strips the symbols from them and adds a debug information sub-package with them.

### 4.1.3 Generation of documentation sub-package

As the name suggests, documentation sub-package contains documentation related to the packaged software. The documentation in Linux systems is usually placed in the `/usr/share/doc`. A module could be used to avoid manual creation of such package. It could take the files marked as documentation in the spec file or detect them autonomously and create the sub-package with "`-doc`" suffix in the name.

### 4.1.4 Update of MIME and desktop databases

If any package contains a `.desktop` file with the `MimeType` key, it has to call the `update-desktop-database` command when the file is installed or removed. Similarly, if a package puts a file to the `%{_datadir}/mime/packages` then the `update-mime-database` command has to be called during the (un)install operation.

A single module could solve this easily. Detection of a file with the `.desktop` suffix and files in the `%{_datadir}/mime/packages` and subsequent addition of the necessary commands to `%pre` and `%post` scripts should be simple.

### 4.1.5 Texinfo documentation management

The Texinfo is an official hypertext documentation format used in the GNU project and some packages include files in this format. When a new texinfo documentation is installed or removed the `install-info` command must be called.

A simple module which detects a package containing files designed to be installed into the `/usr/share/info/` directory, adds the call of `install-info` to the `%pre` and `%post` scripts and adds the `info` package into the list of required dependencies should solve this use case.

### 4.1.6 Policy compliance

Different Linux distributions have different requirements and recommendations what the final package should look like. With the MFS, there could be a module that checks if the package content, including the meta-data, complies with the distribution's policies and aborts the build process if it does not.

As an example of a general policy module we can consider a module that checks whether a package fulfills concepts of `/usr move`, which some Linux distributions, including Fedora, have recently introduced. It is not appropriate to have this functionality as part of the RPM because not all distributions that use RPM have to use the concept. A module is ideal place where similar functionality should be placed.

Another example could be a module shipped with a specific distribution that checks all mandatory policies, e.g., licensing, naming, etc. Such module could be deployed in a build system and ensure that all built packages meet the expected attributes.

### 4.1.7 AppData extraction

AppData are XML based meta-data describing a software. These data are used in the Gnome Software – a new GUI application that lets user install and update applications and system extensions. Programs which have appdata about yourselves install them to the `/usr/share/appdata` and thus the data becomes part of a package. Because the Gnome Software needs the data in advance, before a program is installed, otherwise it would make no sense, there is a need to extract the appdata from the packages. Plan in Fedora is to extract these data in the Fedora's build system. Currently, the appdata are extracted from packages by set of scripts after the build [15].

This approach to the extraction is not very effective, because the extraction script needs to load a lot of packages and extract their payloads in order to copy a tiny XML file. With a MFS module the solution would be easy. The appdata file could be part of the `rpmbuild`'s output together with packages. Such module would be very simple, it would just copy the appfile to the output if the file would exists.

### 4.1.8 Other use cases

Except for these miscellaneous use cases which affect a large number of packages and packagers, MFS can also solve specialized tasks. For some build systems it might make sense to generate a statistics of the processed software. The processed source code can be analyzed by tools for static code analysis like Pylint, Cppcheck, etc., binaries by an anti-virus engine like ClamAV, documentation by some spelling and grammar checkers, etc.

## 4.2 Design

The RPM and build process was introduced and MFS use cases were explained, therefore it is the time to present the design of the system. The main goals will be introduced at the beginning. It will be followed by an explanation of fundamental concepts how to implement a dynamical modular system in the C programming language. Based on this knowledge, the system design will be shown and its concept discussed in detail. Eventually, samples of different parts of API will be introduced in order to help with understanding.

### 4.2.1 MFS benefits

The main benefit of the MFS is modularity. The separation of different tasks into isolated modules. As it will be shown later, in the current state, implementations of some tasks are spread across several places (files), use various macros and scripts, and are not easy to understand. Functionality concentrated in a single module should lead to easier understanding which should imply better maintainability and simpler debugging. Last but not least, the modular interface brings a quick and easy way how to extend the build process and add new features without need to recompile or install

a new version of RPM. And finally, just as easily as a new functionality can be deployed it also can be removed by removing or disabling the module.

### 4.2.2 API quality attributes

The API was designed with regard to the attributes which are listed below.

- **Determinism** – The designed system must behave the same way on any architecture or operating system. For example the module loading. MFS cannot just load the modules in order they are returned by the `readdir()` function, because the order of entries is not guaranteed [14]. This and other similar issues must be handled in a proper way.
- **Extensibility** – API must be easy to extend without breaking its basic concepts. New RPM features should be easy to add to the MFS while the backward compatibility is preserved. This bullet is related to the next one.
- **Encapsulation** – API should be generic, without heavy dependence on the internal RPM structures. Minor internal changes in RPM code must not affect the public MFS API. The RPM internal structures must stay hidden and must not be exported.
- **Configuration** – Whole system must be configurable during the run-time in the same way as the RPM itself. The most portable and natural way to achieve this are macros.
- **Documentation** – A well documented API is taken for granted. Constant, variables, functions, expected parameters, behavior, return values, possible side-effects, etc., all this must be documented.
- **Robustness and stability** – MFS must be robust and reasonable fault-tolerant. A NULL pointer passed instead of string definitely should not lead to SIGSEGV signal for the whole process. Data passed from modules must be checked and validated and not just blindly used. When an error occurs a meaningful error message must be shown and the termination of the program handled gracefully. Any collateral damages caused by the abnormal termination are not acceptable.
- **Usability** – A positive user experience, intuitive and consistent API, self-explanatory names, same behavior of similar-sounding functions, coherent order of function parameters where the most important and mandatory parameters are first, the least important and optional parameters are last, and where the NULL and similar values are acceptable for optional parameters, the system itself should use the proper default value instead, etc.

### 4.2.3 Modular system in C

The MFS must be able to load and run a lot of independent modules which implements various functionality that enhance the build process. The RPM is implemented in the C programming language and the MFS will be implemented in it as well. To be able to load an external functionality during the run-time, the concept of dynamic loading will be used. This concept allows to load a shared library into memory and retrieve addresses of its symbols. This concept must be supported by the operating system, but all commonly used desktop operating systems, which RPM aims to, support it. Hence, the MFS extension modules will be implemented as a shared libraries.

For this dynamical loading, a `dl` interface, provided by a library called `libdl` that provides public API via a header file `<dlfcn.h>`, will be used. A function that opens a shared library `dlopen()` is crucial. The function takes two arguments, first is the path to the shared library and the second one is the mode of the open operation. It is important to know that if the path is not absolute, then `dlopen()` looks in the standard system paths for shared libraries and thus we will need to specify our paths always as the absolute ones, because it is not expected that MFS modules will reside in the standard system paths for libraries. A second important function is `dlsym()` that takes the handle returned from the `dlopen()` together with a symbol name and returns an address of the symbol. At the end, the opened handle has to be closed by the `dlclose()`. In case of an error the `dlerror()` function that returns a human readable string which describes the most recent error can be used. Thanks to the `libdl` library, MFS's dynamical modular system can be implemented.

### 4.2.4 Architecture

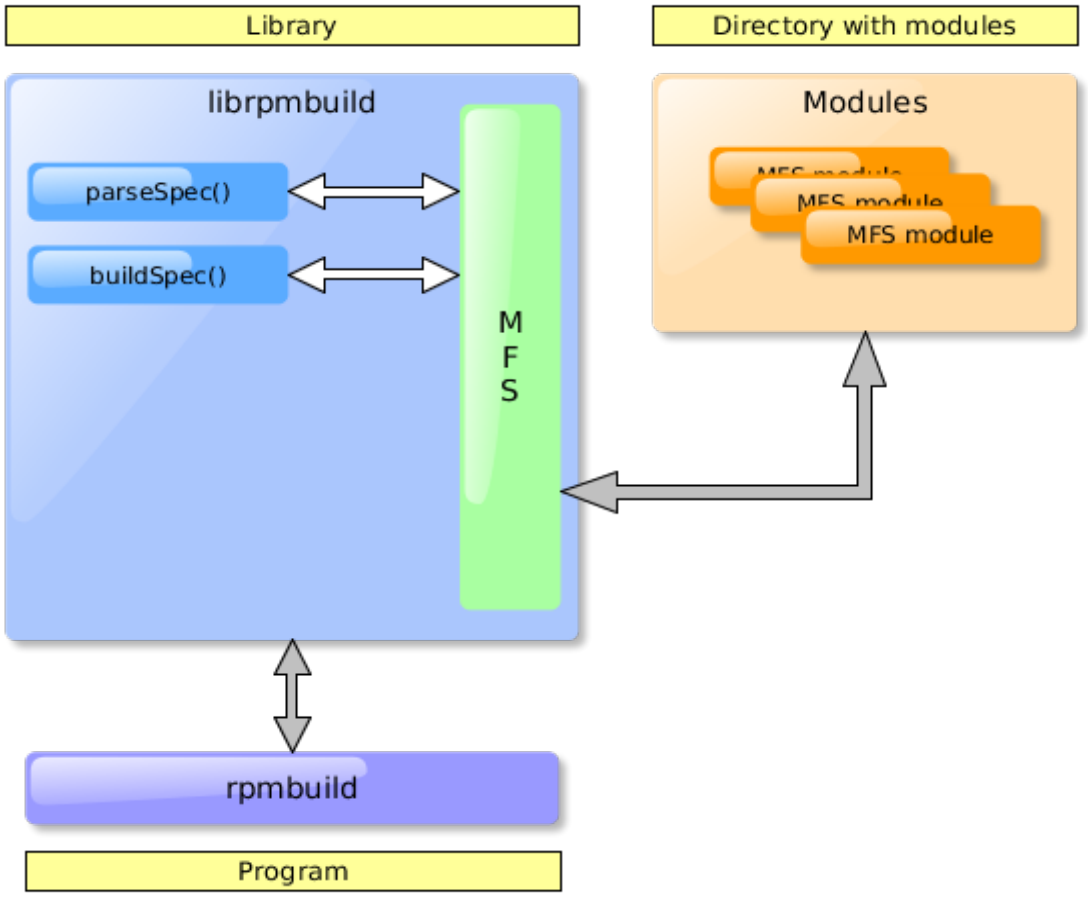



*Figure 3: Architecture of the system*

Architecture of the system is depicted in the figure 3. The MFS is implemented as a part of the `rpmbuild` library. The interface of the library is untouched and therefore no changes are needed into

programs which work with it. Any program that used the library's API[8] before, will automatically support the MFS now. No new function parameters were needed as the MFS is configured via macros. But the `rpmbuild`'s code had to be modified, there were added new function calls which take care of loading of MFS modules and invoking of their hooks. Some parts of the original code had to be re-factored. Some changes will be shortly described later, all changes that were done can be found in the git repository with the source code at the attached CD.

### 4.2.5 Work flow and concepts

Each module must follow the work flow defined by the MFS. The work flow is depicted in the figure 4 and it is based on the internal work flow of the rpmbuild as described in the chapter focused on the `rpmbuild` analysis. The work flow has ten points in which hooks from modules can be called.

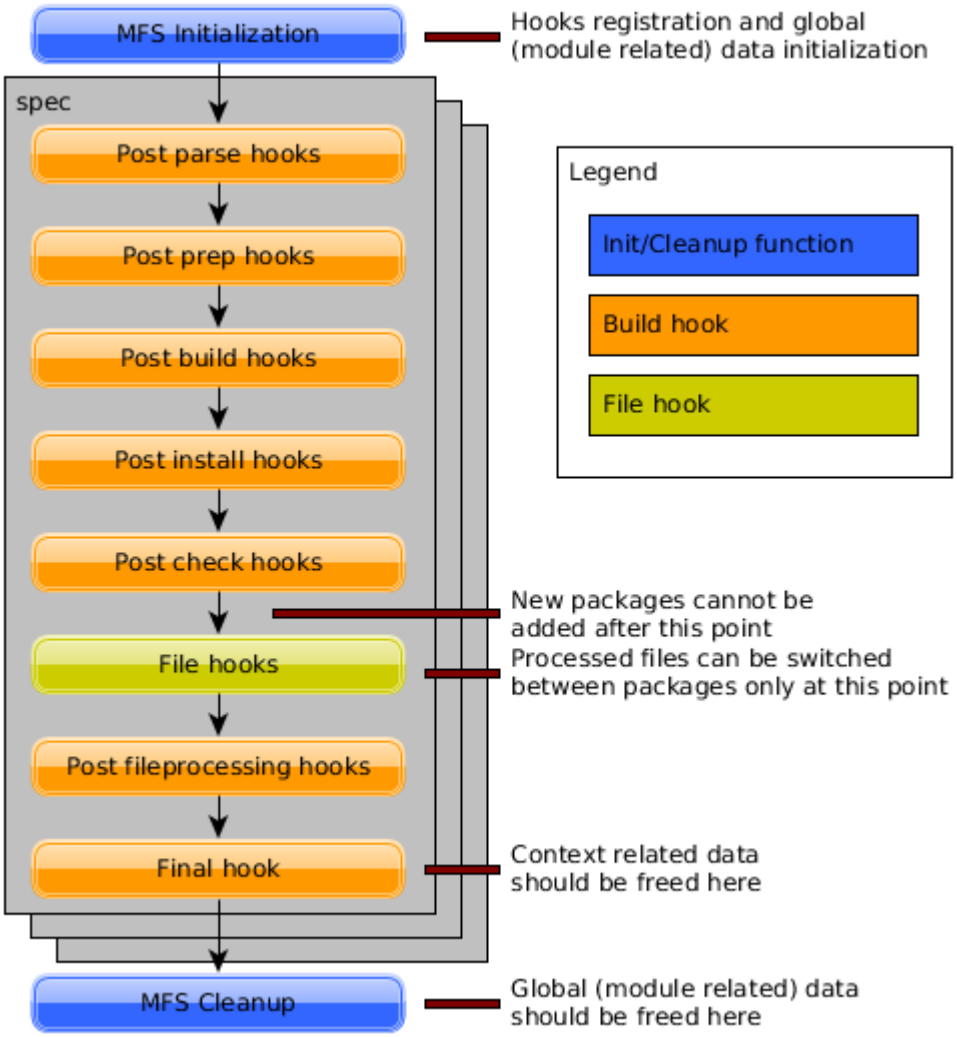



*Figure 4: MFS work flow*

8 http://rpm.org/api/4.11.1/group__rpmbuild.html

The first point is the most important one, the MFS Initialization. Here, the MFS loads all available modules and calls their initialization functions. Each module must implement one, it is mandatory. It is the only place where hooks can be registered. In the terms of MFS, the hooks are module functions used to monitor, alter or extend behavior of the build process during its significant parts. The hooks can access internal variables and structures of the rpmbuild via the MFS API, that will be described in the next chapter, comfortably and in a safe manner, without any need to have knowledge about the internal variables, structures and their coupling, constraints and representation. Another thing that can be done during the initialization is to set global (module related) data. The opposite of the global data are context related data.

The context is determined by the currently processed spec file. The `rpmbuild` can build several spec files during a single run and even when it builds only a single spec, it can be internally expanded into multiple ones. It happens when the spec file has defined multiple build architectures. Such spec file is parsed and built for each of the supported architectures with a macro `%_target_cpu` set to the desired value.

Consequently, the sequence of hooks, which is in the figure 4 placed over the gray rectangles with captions “spec”, can be performed repeatedly multiple times during a single build.

For that reason, a smart data sharing between hooks within a module is an important concept provided by the MFS. Inherently, the hooks are separated and independent pieces of code. The data sharing could be implemented by the modules themselves, for example by use of global variables in the module but such solution is naive and dangerous. Any global state can cause troubles in a multi-threaded environment. The problem is even more complicated by the fact that the sequence of hooks can be performed multiple times and thus the modules would have to care about the proper (re)initialization and freeing of the shared data.

To make the data sharing as simple as possible, the MFS introduces a concept of global and context related data and mechanism to access them. The global data are the data that can be set and retrieved by any hook within a module regardless of the current context. This data can be set during the MFS Initialization and should be freed during the MFS Cleanup, but this is only suggestion, the final approach depends on the programmer and his needs, the global data can be set and freed anytime during the work flow. The second mechanism is called the context related data. The context data are bound to the currently processed spec file and can be accessed only within the corresponding context. Due this, the data can be created in any hook, but it must be freed no later than in the Final hook, because it will not be accessible after this point.

After the MFS Initialization, the build process starts. Depending on the the rpmbuild options, which specifies stages that should be performed, the build process continues and the registered hooks are called in the order depicted in the figure. The hook sequence can be repeated several times for different spec files and at the end the MFS Cleanup is done. During this clean up, a registered cleanup function can be called. Currently, this function is mainly suitable for a clean up of the global (module related) data.

The moments when the individual hooks are called corresponds with their names:

- **Post parse hooks** are called after the spec is parsed and before the build starts.
- **Post prep hooks** are called after the `%prep` script was performed.
- **Post build hooks** are called after the `%build` script was performed and sources were built.
- **Post install hooks** are called after the `%install` script was performed and the software was installed to the build root.

- **Post check hooks** are called after the `%check` script was performed. This is the last point where new sub-packages can be added to the current spec file. It is because after this point all source files including the package icons are collected and inserted to the header of the source package and thus, for example, any icons added after this point would not be part of the source package. For that reason, MFS blocks any attempt to add a new package during any later hook call for the current context.
- **File hooks** are special kind of hooks. Unlike any other hooks, they have a unique prototype and except the current context they gets also a file. They are called during a file processing. The RPM carefully analyzes and prepares each file it adds to the package. Right before the file is finally attached to a package, the file hooks can step in and decide weather the file should be attached to its original destination or/and it should be attached to some other sub-package(s). The decision can be based on multiple file attributes that can be accessed via the MFS API for the `MfsFile` data type and some of these attributes can be also altered before the file is attached to a new destination.
- **Post file processing hooks** are called when all files were processed and it can be used to verification or analysis of which package contains which files and what are their attributes.
- **Final hook** is a hook that is called at the end of the processing of a spec file. At the point where the hook is called, the packages from the spec file are already written out and it doesn't make a sense to try to modify anything related to the packages. Purpose of the callback is to print some statistic, if suitable, and especially free the context related data.

At the end, after the all hooks were called, specs were processed and the build is being finished, the MFS calls a cleanup function of a module. The function is called only once, regardless of number of spec files that were processed.

This was description of work flow in case that everything goes well. In case that something goes wrong, the work flow will be obviously a little different. If something goes wrong right during the module initialization, only the cleanup functions of modules that were initialized successfully are called before exiting. If something goes wrong during the build, all remaining hooks are skipped and only final hooks are called. The module cleanup functions are called for all modules in that case.

The order in which individual hooks are called should be perfectly clear. Now, a way whereby the order of hooks registered to the same point is determined will be described. As the first sort key is used the priority which is defined at the time of hook registration. If there are multiple hooks which have the same priority, as the second sort key are used the names of modules from which the hooks comes from. If there are multiple hooks with the same priority and from the same module, then the order of the hook registrations is decisive. This mechanism is deterministic on any platform, it is not affected by the order in which the modules were loaded neither by locale settings, because the module name can contain only basic alphanumeric characters from the ASCII table.

It is the time to introduce the MFS configuration options. The MFS is configurable by macros, this approach was chosen because such type of configuration is already generally used in the RPM. The MFS configuration macros are:

- `%_rpmbuild_modules_enabled` – Controls if the MFS is enabled or not.
- `%_rpmbuild_modules_blacklist_regex` – The regular expression for module names. The modules whose names match this expression will not be loaded.
- `%_rpmbuild_modules_directory` – The directory where the MFS looks for modules.

## 4.3 Designed API

Some selected parts of designed API are described in this chapter. The whole API is extensive and description of all parts would be counterproductive. Certain parts are very similar and thus description of only single representative element from each should be sufficient. In most cases, for better readability, only function names are shown instead of complete declarations. The full API specification could be found in the appendix D at the end of the thesis.

The beginning of the chapter is dedicated to the important conventions bound to a module development. The rest of the chapter is focused on the main parts of API which covers initialization, cleanup and modification of the building process. At the end, an overview of all components of the API is shown.

### 4.3.1 Coding guidelines

Before the API will be introduced, there is a necessity to introduce some general coding practices and rules for the module implementation. Some of them are mandatory and must be obeyed in order to achieve the correct module behavior, rest of them are optional, but you should keep them in mind.

One of the rules that is mandatory and important is the one related to the function return values. Whenever any function or hook has to return a return code of type `rpmRC` then the return value should be either `RPMRC_OK` or `RPMRC_FAIL`. Although, the `rpmRC_e` enumeration defines more return values than the mentioned two, these other values are not relevant and their original meanings are not important for the MFS either.

The second rule related to the return values, especially when something went wrong, is that an error log message should be emitted before any failure return code is returned. This approach is highly used inside the RPM itself. Whenever anything goes wrong an error log message with description what is wrong is printed out and after then an error code is returned. Also MFS functions stick to this rule and print a descriptive error message before an error code is returned. The previous simply means that when a module needs to return an error code because an MFS function returned an error, the module does not have to print its own error message, but of course it can and it would be appropriate if it has any additional information which can be relevant to the failure. Proper error reporting from the places where the errors arise is very valuable because it can provide more detailed information which makes debugging easier and this makes a positive user experience.

Optional but highly recommended for the module authors is to log all changes to the build process which are not done via MFS API. By default, the MFS logs any actions which alter state of the build process. However, not all such actions are done just via the MFS API. For example, any module can get header of any package that is currently built by the MFS API. The RPM itself provides a rich API by which the headers can be changed. Such changes are outside of the MFS scope and thus not logged because they are invisible for it. Every well-behaved module should log any such actions and inform about any change that is going to be made.

All logging should be done via the available MFS logging API. The MFS provides several logging function for different levels of severity. The messages logged via MFS are decorated with the "`mfs: `" prefix and hand over to the RPM logging API[9].

9 http://rpm.org/api/4.11.1/group__rpmlog.html

Other important recommendations are summarized into these bullets:

- All needed changes to the build process should be made via the MFS API. Modules can change the headers directly, for example add a new dependency "manually" by the `headerPut()` or similar functions, but such behavior is highly discouraged.
- Avoid memory corruptions and leaks. Always check if a function returns a newly allocated memory, which must be freed by the caller, or a pointer to an existing memory, which is owned by the library and therefore should not be freed.
- Module should be robust. It should check all return values and act upon them accordingly. It should not ignore errors and continue in processing when a serious one arise because it can bring the build process into undefined state.
- Module should be focused to solve exactly one problem/use-case. Any huge all-in-one modules providing bunch of unrelated features should be avoided.
- Module should have preset optimal settings that most likely fit to the majority of users by default.
- Module should be configurable if appropriate and preferably using macros as the rest of the RPM system.
- Module's name should fit its purpose.
- Module's code should follow general best coding practices[10].

### 4.3.2 Type overview

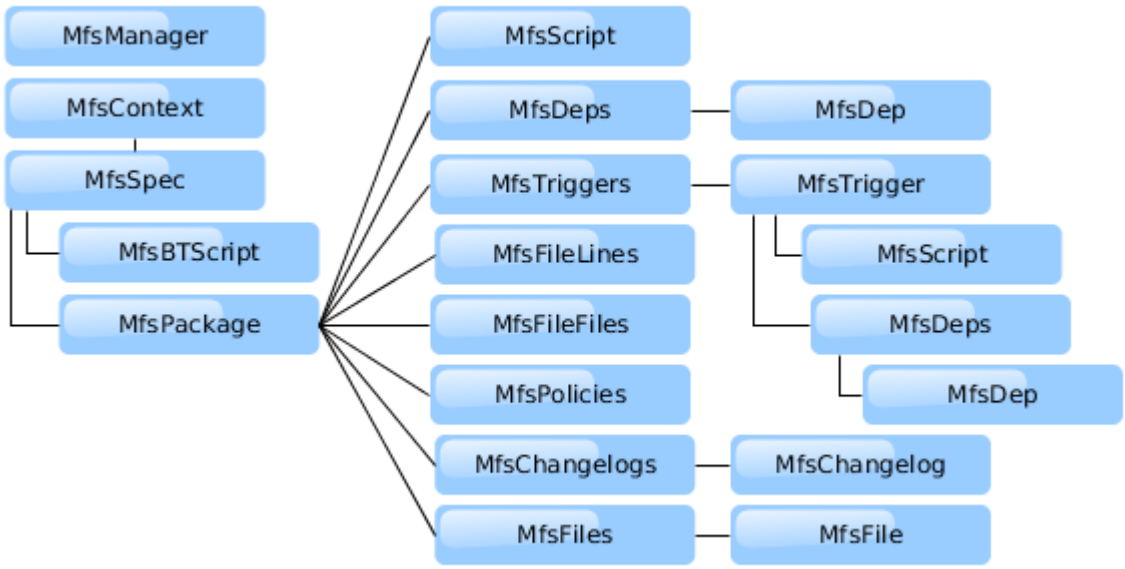



*Figure 5: MFS API data types relations*

The MFS API contains a lot of different data types which represent objects which occur in the build process. Some of them are even used in multiple places. The figure 5 illustrates the relations between the data types for easier understanding.

10 http://en.wikipedia.org/wiki/Best_coding_practices

Note that the figure shows how the `MfsPackage` uses the `MfsScript` and the `MfsDeps` data types which are also used for the `MfsTrigger`, which corresponds to the fact that a trigger is a specific combination of these two objects. Other relations are more intuitive and straightforward.

### 4.3.3 Module initialization API

When the guidelines were explained, it is time to present the API itself. Let's start with functions and data types related to the discovery and registration phase of the modular system work flow. When a module is discovered, it is loaded into a memory and its initialization function is called. As shown in the chapter dedicated to a modular system in C, the MFS must know the name of the initialization function. For that reason, there is a naming schema which is mandatory and must be followed by each initialization function. The schema is "`init_<modulename>()`", where the `<modulename>` matches the file name of the module without the `*.so` extension.

**`typedef rpmRC (*MfsModuleInitFunc)(MfsManager mm);`**

This is the prototype of the function. Such function must be implemented by each MFS module and inside it, the module must register all the needed `MfsBuildHookFunc`tions by the `mfsManagerRegisterBuildHook()` and all the needed `MfsFileHookFunc`tions by the `mfsManagerRegisterFileHook()`. During the initialization, also global data, relevant for the module, could be set by the `mfsManagerGetGlobalData()`. Usually, when a dynamically allocated global data are used, they have to be feed at the end. For such a purpose, a module clean up function can be registered by the `mfsManagerSetCleanupFunc()`. The prototype of the function is `typedef void (*MfsModuleCleanupFunc)(MfsManager mm)`. The `MsManager` is a module manager that used for the hook registration. At the end of the initialization, the module must return the `RPMRC_OK` constant if the initialization passed without an error or the `RPMRC_FAILURE` if something went bad. Functions related to the hook registration are described next.

**`MfsBuildHook`**

A hook called at several points during the build process. The points where the hook can be called were described in the sub-chapter about MFS work flow. Functions for registration of such type of hook are:

- `mfsBuildHookNew()` – Crates a new `MfsBuildHook`. The hook function and the point where it will be called are specified as arguments.
- `mfsBuildHookSetPriority()` – Sets the priority for the hook. The priority value determines an order in which all hooks registered for a specific point will be called. The more detailed description about the order in which hooks are called was given in the chapter about the work flow.
- `mfsBuildHookSetPrettyName()` – Sets the function name that will be shown in the MFS log messages. Use of this function is optional, but if used then the debug messages are more user-friendly. If the pretty name is not set, then the messages will contain hook's memory address instead of it.
  `mfsManagerRegisterBuildHook()` – Registers the hook to the MFS.

**`MsFileHook`**

A hook called during a file processing. To limit number of calls, it is possible to specify glob patterns that must match the file path. Note that several patterns could be specified and when at least one of them matches, the hook is called. If no pattern is specified, the hook is called for each processed file. Related functions are:

- `mfsFileHookNew()` – Creates a new `MsFileHook`. Takes a hook function as a single argument.
- `mfsFileHookSetPriority()` – Sets a priority for the hook.
- `mfsFileHookSetPrettyName()` – Same as `mfsBuildHookSetPrettyName()`.
- `mfsFileHookAddGlob()` – Adds a wild-card pattern to the hook.
- `mfsRegisterFileHook()` – Registers a hook to the MFS.

**Other functions for registration phase**

- `mfsManagerGetGlobalData()` - Sets module's global data.
- `mfsManagerSetGlobalData()` - Gets module's global data.
- `mfsManagerSetCleanupFunc()` - Registers module's clean up function. Useful for freeing module's global data or printing statistics.

### 4.3.4 Context API

As mentioned, the context in MFS represents the currently processed spec file. Each registered hook get a context object when called. The context data type is `MfsContext` and API defines few functions which use it:

- `mfsManagerSetGlobalData()` - Gets module's global data.
- `mfsManagerSetGlobalData()` - Sets module's global data.
- `mfsContextGetData()` - Gets context related data. These data are available only within the current context.
- `mfsContextSetData()` - Sets context related data.
- `mfsContextGetSpec()` - Gets a new `MfsSpec` that represents the current spec file.

### 4.3.5 Spec manipulation API

API for accessing and modifying a spec file. Note that any modification of the spec makes sense only if the data that are being modified have not been used yet. For example, any modification of the `%build` script is pointless if it is done inside any hook that is called later than the point `MFS_HOOK_POINT_POSTPREP`, because after that point the `%build` section was already performed and the modification will have no effect. Functions of this part of API:

- `mfsSpecGetString()` – Gets a specified string from the spec file.
- `mfsSpecSetString()` – Sets/overrides a specified string to the spec file.
- `mfsSpecPackageCount()` – Gets number of sub-packages defined in the spec file, except the source package.
- `mfsSpecGetPackage()` – Gets a package specified by index.

- `mfsSpecGetSourcePackage()` - Gets a source package.
- `mfsSpecGetMacroContext()` - Gets pointer to the macro context of the spec file, that can be used by function from the RPM macro API.
- `mfsSpecExpandMacro()` - Expands a macro into a buffer using the macro context of the spec. Currently, the whole RPM library uses only one single global context and thus every spec file uses just this one too. This function is here due to future compatibility for a case that separate contexts for spec files would be introduced.
- `mfsSpecGetScript()` - Gets the `MfsBTScript` that represents a build-time script from the spec. Types of build-time scripts are `%prep`, `%build`, `%install`, `%check` and `%clean`.
- `mfsSpecSetScript()` - Sets (and overrides if necessary) a build-time script of the spec.
- `mfsSpecFree()` - Frees the spec file.

### 4.3.6 Package manipulation API

API designated for manipulation with packages.

- `mfsPackageGetSpec()` - Gets the spec file of a package. The spec file is represented by a newly allocated `MfsSpec`.
- `mfsPackageFree()` - Frees a package.
- `mfsPackageId()` - Gets a unique id of a package. Each `MfsPackage` is always newly allocated and its memory address thus cannot be used for the unequivocal identification. This function returns the memory address of the underlying structure representing the package and this address can be used for the unique identification during the comparison of packages.
- `mfsPackageNew()` – Creates a new `MsPackage` and appends it to the context (spec file).
- `mfsPackageFinalize()` - Performs some sanity tests and fills some missing values in package's header like the target architecture, operating system, platform, etc. This function should be called on each newly added package.
- `mfsPackageGetHeader()` - Gets a pointer to the header of a package. The header can be modified by the standard RPM Header API.
- `mfsPackageName()` - Gets a pointer to a string with the name of a package.
- `mfsPackageTags()` - Gets a pointer to a list of supported preamble tags which can be added by the `mfsPackageSetTag()`, the list is NULL-terminated.
- `mfsPackageSetTag()` – Sets a preamble tag to the package.
- `mfsPackageGetDescription()/mfsPackageSetDescription()` - Gets/sets a package description.
- `mfsPackageGetScript()/mfsPackageSetScript()` - Gets/sets a package install-time script.
- `mfsPackageDeleteScript()` - Deletes an install-time script from a package.
- `mfsPackageGetTriggers()/mfsPackageSetTriggers()`
- `mfsPackageGetChangelogs()/mfsPackageSetChangelogs()`
- `mfsPackageGetDeps()/mfsPackageSetDeps()`
- `mfsPackageGetFileLines()/mfsPackageSetFileLines()`
- `mfsPackageGetFileFiles()/mfsPackageSetFileFiles()`

- `mfsPackageGetPolicies()/mfsPackageSetPolicies()`
- `mfsPackageGetFiles()` - Gets a list of files that were processed and are going to be included in the package. The list is represented by the `MfsFiles` data type. This list is only available at the `MFS_HOOK_POINT_POSTFILEPROCESSING` point of the build process.

### 4.3.7 API for lists

A lot of data in packages are stored in form of ordered sequences. These sequences have two main features. The first one is that the order of the items is significant and the second one that internally are various sequences stored differently. For example the changelogs are internally stored directly in the package header. They are not kept in any list and are put right to a header during parsing. In contrast to dependencies, which are stored in multiple dependency sets (the data type `rpmds`) during building and to the header are stored at the end of the build process. The special cases are triggers. The triggers are combinations of dependencies and build-time scripts and each of this part is stored independently in a different form. The trigger dependencies are stored in the previously mentioned dependency sets and the scripts are stored in a special list. The relation between these two lists is based on the indexes of the triggers. To hide these different variations of the internal representations and provide a unified interface for working with lists, the concept of the independent list was chosen.

This concepts brings a separation of data in a list and data in a package and a postponed modification. Each returned list is a copy of the package internal list. This means that any change made to the list will not be propagated to the package. To get the changes propagate, the list (the copy) has to be set to the package by a special function. This action replaces package's original list with the new one.

This postponed modification is intentional and makes the API more generic and extensible. With this approach, the list and all its elements are separated from the package and its internal implementation. Thanks to this design, it is possible to make multiple changes step by step without bothering by some consistency checks during the modification. All these checks are done at once, at the point when the all modifications are applied to a package.

This concept is used for all lists that belong to a package. The data types which encapsulates sequences are:

- `MfsTriggers` – Package's triggers (each element has type `MfsTrigger`)
- `MfsChangelogs` – Package's changelogs (each element has type `MfsChangelog`)
- `MfsDeps` – Package's dependencies (each element has type `MfsDep`)
- `MfsFileLines` – Lines from `%files` section (list of strings)
- `MfsFileFiles` – Files from `%files` section specified by `-f` (list of strings)
- `MfsPolicies` – Lines from `%policy` section (list of strings)

Another characteristic associated with the concept is that all elements are owned by a list. This means that a getter function, which returns an element from a list, always returns a pointer to the memory owned by the list and not a newly allocated element. The consequence is that elements are modified in-place.

The first three above-mentioned lists contain composite data types and the remaining three lists contrariwise contain just a strings. Due to the difference, the API of these lists is not exactly the same

but the concept remains unchanged. The following two sub-chapters show API of the both list types. For the sake of clarity, only one list of each type will be described.

### 4.3.8 Changelog list API and changelog API

Changelogs has been chosen to illustrate the above-described concept of the work with lists. This list is the example of a list that contains components of a composite data type (the `MfsChangelog` in this specific case).

#### `MfsChangelogs` – A list of changelogs

A list of changelogs can be retrieved from a package by the above-mentioned function `mfsPackageGetChangelogs()`. This function returns a list of all changelogs that a package contains. This list is a copy of the package's internal list and any change made to this list won't be propagated to the package. To get the changes propagate, you has to set the modified list to the package by the function `mfsPackageSetChangelogs()`. The API functions for manipulation with a list of changelogs are:

- `mfsChangelogsFree()` – Frees a list of changelogs. The `mfsPackageSetChangelogs()` doesn't pass the ownership of the list to the package and therefore the owner (the programmer) must free it on his own. The function also frees all elements of the list.
- `mfsChangelogsCount()` – Returns the number of changelogs in a list.
- `mfsChangelogsAppend()` – Appends a changelog to a list. The ownership of the changelog is passed to the list.
- `mfsChangelogsInsert()` – Inserts a changelog into a list. The ownership of the changelog is passed to the list.
- `mfsChangelogsDelete()` - Deletes a changelog from a list.
- `mfsChangelogsGetEntry()` - Gets a pointer to a changelog from a list. Note that this function does not return a newly allocated item but just returns a pointer to the one that is owned by the list. It means that this item should not be freed, but it can be modified. Elements are freed automatically as necessary, i.e. when `mfsChangelogsDelete()` or `mfsChangelogsFree()` is used.

#### `MfsChangelog` – A single changelog

The changelog API manipulates with a changelog returned by the `mfsChangelogsGetEntry()` or with a newly allocated one by the `mfsChangelogNew()`.

- `mfsChangelogNew()` - Allocates a new changelog.
- `mfsChangelogCopy()` - Copies a changelog.
- `mfsChangelogCopy()` - Frees a changelog.
- `mfsChangelogGetDate()` - Gets a date of a changelog as `time_t`.
- `mfsChangelogGetDateStr()` - Gets a date of a changelog as a newly allocated string.
- `mfsChangelogGetName()` - Gets a name of changelog author as a newly allocated string.

- `mfsChangelogGetText()` - Gets a test of a changelog as a newly allocated string.
- `mfsChangelogSetDate()` - Sets a date to a changelog.
- `mfsChangelogSetName()` - Sets author's name to a changelog.
- `mfsChangelogSetText()` - Sets a text to a changelog.

### 4.3.9 Policies API – String list API

This is an example of what API for manipulation with sequences of strings looks like. Similar API like the one for the policies is available for the lines (`MfsFileLines`) and files (`MfsFileFiles`) from the `%files` section.

The list of policies `MfsPolicies` is returned by the `mfsPackageGetPolicies()`. The available functions for the list are:

- `mfsPoliciesFree()` - Frees a list of policies.
- `mfsPoliciesCount()` - Return the number of policies in a list.
- `mfsPoliciesGetPolicy()` - Return a string with a policy. The string is owned by the list a should not be freed.
- `mfsPoliciesAppend()` - Append a policy to a list.
- `mfsPoliciesDelete()` - Delete a policy from a list of policies.
- `mfsPoliciesGetAll()` - Returns a newly allocated NULL-terminated list of strings with all policies or NULL if the list is empty. The caller of the function must free the list and all its element. The `argvFree()` function from the RPM Argument Manipulation API[11] can be used for that.

### 4.3.10 Processed file manipulation API

This API is used for handling with files that were analyzed and processed by the RPM and are going to be added to a package. The RPM collects plenty of information that are related to the file including the file status information from the `stat()` call, file type information and attributes like MIME type, etc., obtained by the Libmagic library, etc. All this information is combined with relevant directives from the spec file like `%attr`, `%defattr`, `%doc`, `%verify`, etc. The result of this process is something that MFS calls a “processed file” and this API allows to access and modify these file attributes.

The modification of the file attributes is available only in a file hook, before the file is attached to a package. By default, the file is going to be attached to the package in whose `%files` list it was specified. But a file hook can attach the file to any other package and even set a flag, that prevent insertion to the original package.

The `MfsFile` is available for each file hook, because it gets an `MfsFile` as a parameter when called and it can access and modify its attributes. The second place where the `MfsFiles` can be handled is inside build hooks called at the `MFS_HOOK_POINT_POSTFILEPROCESSING`, where the list of package's processed files can be obtained by the `mfsPackageGetFiles()`. Nevertheless, modification of the file or attaching to a different package is not available here.

11 http://rpm.org/api/4.11.1/group__rpmargv.html

- `mfsFileGetPath()` – Gets the original full path of the file on the disk. The path returned by the function remains the same even when the disk path is changed by the `mfsFileSetDiskPath()`.
- `mfsPackageAddFile()` – Adds a file to a package.
- `mfsFileGetToOriginal()/mfsFileSetToOriginal()` - Gets/Sets a flag which states if the file should be attached to its original destination.
- `mfsFileGetStat()/mfsFileSetStat()` - Gets/Sets a file status data (`struct stat`).
- `mfsFileGetDiskPath()/mfsFileSetDiskPath()` - Gets/Sets a full path of the file on the disk.
- `mfsFileGetCpioPath()/mfsFileSetCpioPath()` - Gets/Sets a path under which the file will be installed to a system during the installation of the package.
- `mfsFileGetUname()/mfsFileSetUname()` - Gets/Sets the user name of the file owner.
- `mfsFileGetGname()/mfsFileSetGname()` - Gets/Sets the group name of the file owner.
- `mfsFileGetFlags()/mfsFileSetFlags()` - Gets/Sets flags of the file. These flags specifies RPM file attributes like that the file is configuration, documentation, readme, license, ghost, icon, that the file can be missing, etc. For more flags see the RPM's `enum rpmfileAttrs_e`.
- `mfsFileGetVerifyFlags()/mfsFileSetVerifyFlags()` - Gets/Sets verification flags which specify what should be checked during verification of the file. For the flags see the RPM's `enum rpmVerifyFlags_e`.
- `mfsFileGetLangs()/mfsFileSetLangs()` - Gets/Sets file languages.
- `mfsFileGetCaps()/mfsFileSetCaps()` - Gets/Sets file capabilities[12].
- `mfsFileGetColor()` - Gets a file color. The file color is used for the multilib support. Thanks to the file colors, RPM can mix correctly both 32bit and 64bit binaries on a 64bit system. For currently available file colors see the `enum FCOLOR_e`. This attribute is not adjustable.
- `mfsFileGetAttrs()` - Gets a NULL-terminated list of strings with file attributes. This attribute is not adjustable.
- `mfsFileGetType()` - Gets file types. For example: "C source, ASCII text", "directory", etc. This attribute is not adjustable.
- `mfsFileOwningPackagesCount()` - Returns the current number of packages that have the file attached.
- `mfsFileOwningPackage()` - Gets a package, specified by its index, that has the file attached.
- `mfsFileGetOriginalDestination()` - Gets the package that is the original destination for the file.

12 https://www.kernel.org/pub/linux/libs/security/linux-privs/kernel-2.2/capfaq-0.2.txt

### 4.3.11 Other APIs

There are several more APIs which were omitted for the sake of brevity. You can find them in the MFS header file in the appendix D at the end of the thesis. All the omitted APIs use the principles and concepts that were presented above by the APIs which were shown. The omitted APIs are:

- **Logging API** – This API is straightforward and does not need a comment.
- **Build-time script API** – This API is similar to APIs that were described.
- **Install-time script API** – Another API that is similar to the ones that were described.
- **Trigger list API + Trigger API** – Similar to Changelog list API + Changelog API
- **Dependency list API + Dependency API** – Similar to Changelog list API + Changelog API
- **%files' line list API** – Similar to Policies list API
- **%files' file list API** – Similar to Policies list API

## 4.4 RPM modifications

An iterative and incremental development methodology was chosen for the development. The final design was evolving during the implementation in order to optimally fit all the needs that were encountered. The main concepts remained the same, but the API was changed, enhanced and extended. All these modifications were already included in the previous chapters that describe the actual MFS state. In this chapter, a few notes from the implementation will be briefly shown with a main focus on interesting and important changes of the RPM code that were necessary for the purposes of the MFS.

Because one of the main ideas behind the MFS, which is also included in its name, is the work with files, the first change of the RPM code was symbolically aimed at the file handling. The function `addFile()`, mentioned during the analysis of `rpmbuild`, which selects the appropriate ownership and rights, prepares internal file representation and appends the file to a package was re-factored and the addition of the file to a package was moved to a new function `addFileListRecord()`. This modification makes it possible to append a single file to multiple packages which is crucial for the MFS.

Another re-factorization was a split of the `handlePreambleTag()` into the `handlePreambleTag()` and `applyPreambleTag()`. The original implementation parses a preamble line (e.g. "`Source0: foo.tar.gz`"), adds the parsed data to package's header and simultaneously does necessary sanity checks of the data. After the modification, the function does only the parsing and for the insertion of the data and the sanity checks it uses the newly added `applyPreambleTag()`.

Also, some internal functions marked as static and some internal data types and variables from different RPM's modules had to be exported to be able to reuse an already implemented functionality. Therefore, for each module that had anything which must be exported a header file with a file name matching the pattern `<name>_internal.h`, was created. For example, for the `parseChangelog.c` the `parseChangelog_internal.h` file was created and the definitions of functions and types that had to be exported were placed there.

Other interventions that worth mentioning are the re-factorization of the `rpmfc.c`, which is the RPM module for file classification, and modification of the `processBinaryFiles()`, which analyzes package's files and puts obtained information into the package's header. In the `rpmfc`, the initialization of the file classificator was moved from the `rpmfcClassify()` into a separated function `rpmfcInit()` and a new internal interface for classification of a single file called `rpmcf` (which stands for “RPM classified file”) was added. Also the internal functions which use file classification were modified to use the new API. I.e. `rpmfcClassify()`.

In the `processBinaryFiles()`, the file processing that was originally done in one big loop were split into three smaller loops which are done successively. This looping is done over all the packages defined in a spec file. In the first loop, an empty file list is created for each package. In the second one, all packages' files are analyzed and the lists are filled. In the third loop, the files are put to the packages' headers and the lists are freed. This modification is important for MFS in order to be able to put processed files into different packages by file hooks. Before the re-factorization, all files for a single package were analyzed and directly put to package's header during a single iteration. With such approach, a change of destination package for a file would be hard to implement in contrast to the re-factored solution where this is easy to do.

This was a list of some important modifications of the RPM code. All the modifications can be found in a git repository on the attached CD.

# 5 Example Modules

Three exemplary modules were implemented during the development. In this chapter, the modules will be introduced, analyzed and evaluated. These modules are proof of concept that the goals set for the MFS were fulfilled and the MFS can be used to solve the use cases which have been found during the analysis and also these which may come out in future.

## 5.1 Debug information sub-package

This is one of the use cases mentioned at the beginning of the previous chapter. The purpose of the module is to generate a sub-package that contains debug information for binary files which are included in the package.

**Current state**

Currently, the generation is done by set of RPM macros defined in the `/usr/lib/rpm/redhat/macros`. The generation is enabled by the `%_enable_debug_packages` set to `1`. A template of the sub-package that will be added is defined by the `%debug_package` and can be found in the same file:

```
%debug_package \
%ifnarch noarch\
%global __debug_package 1\
%package debuginfo \
Summary: Debug information for package %{name}\
Group: Development/Debug\
AutoReqProv: 0\
%description debuginfo\
This package provides debug information for package %{name}.\
Debug information is useful when developing applications that use this\
package or when debugging this package.\
%files debuginfo -f debugfiles.list\
%defattr(-,root,root)\
%endif\
%{nil}
```

Note that the template defines the `%__debug_package` macro which will be important later. If the generation of debug information sub-package is enabled, the template is added to the spec file by this construction that prepend it to the `%install` section:

```
%install %{?_enable_debug_packages:%{?buildsubdir:%{debug_package}}}\
%%install\
%{nil}
```

This adds the sub-package into the spec and the following code appends a macro `__debug_install_post` to the `%install` section. The code of this macro will be run after the installation if the sub-package was added, this means if the `__debug_package`, which was earlier mentioned, is set to a positive value (as positive values are considered: '`1`', '`y`' or '`Y`'):

```
%__spec_install_post\
    %{?__debug_package:%{__debug_install_post}}\
    %{__arch_install_post}\
    %{__os_install_post}\
%{nil}
```

The content of the `__debug_install_post` is defined in the `/lib/rpm/macros`:

```
%__debug_install_post   \
   %{_rpmconfigdir}/find-debuginfo.sh %{?_missing_build_ids_terminate_build:--strict-
build-id} %{?_include_minidebuginfo:-m} %{?_find_debuginfo_dwz_opts} %{?
_find_debuginfo_opts} "%{_builddir}/%{?buildsubdir}"\
%{nil}
```

This piece of code launches the `find-debuginfo.sh` script which is distributed as part of the RPM. The script strips debug information from all binaries installed to the build directory and creates a `debugfiles.list` file which contains paths, one per line, of these stripped data. Note that the output file is the file which is loaded by the debug information sub-package in its `%files` section (see the template). You can notice that some of macros from the `/usr/lib/rpm/redhat/macros` are also redefined in the `/lib/rpm/macros`. This can be confusing, but the values of these redefined macros are the same as the original ones, at least in Fedora 20.

### Module

The same approach as the one just presented was also chosen for the module implementation. The source code of the module can be found at the end of the thesis in the appendix C.

The module contains one build hook named `setupPkgFunc()` registered to be called after the parsing (at the `MFS_HOOK_POINT_POSTPARSE` hook point). This hook expands the `_enable_debug_packages` macro to check if the sub-packages with debug information are enabled and if architecture of the package is not `noarch`. If both checks have passed, it appends call of the `find-debuginfo.sh` script to the `%install` script, adds the debug information sub-package to the spec file and configures it. The configuration consists of setting description and group, disabling auto-generation of dependencies for the sub-package, setting of default permissions for its files and setting the `debugfiles.list` as a file list which contains paths of files that will be included in the package.

This solution basically mimics the behavior of the original implementation. Another way how to implement such a module could be: A module with a single hook registered at the post-install time which analyzes all installed files, strips the debug symbols from them if possible and, depending on the results, adds the sub-package with debug information.

### Comparison of solutions

The implemented exemplary module consolidates the desired functionality into a single unit. The module's code is self-explanatory, easy to understand and maintain in contrast to the current state in the RPM, where the functionality is distributed among multiple configuration files with dozens of other unrelated macros and where even some of them are duplicate. Although the module is implemented in the C programming language, it has only about ninety lines of code. The number of lines includes all types of lines – the empty lines, comments, variables declared at the top of functions, etc.

## 5.2 Language sub-packages

This module generates sub-package for each language which is used in the file lists of packages.

**Module**

The module contains three hooks. The first hook is registered to be run at the post-install time. It analyzes all files of the main package plus all sub-packages, prepares list of used languages and creates a sub-package for each language that was found. The second hook is a file hook registered to match every file. It appends files to the appropriate language sub-packages. The third and the last hook is registered at the `MFS_HOOK_POINT_FINAL` and it frees shared data. The source code of the module is available at the attached CD.

**Comparison**

The presented use case can not be solved by macros like the previous one. Macros have no access to the `%files` section and its attributes so this section cannot be analyzed and used languages enumerated. Moreover, a macro cannot add a sub-package after the parsing is finished. This is a problem, because lists of files can be known long after the parsing is over, because they can be generated by a build-time scrip during the build process and attached to packages by using the `-f` option at the `%files` section.

This use case demonstrates a kind of functionality which MFS can handle and which cannot be solved by the current means. Implementation of this functionality in the RPM without the MFS would require major changes in the RPM code.

This use case is reasonable, the separation of the language-specific files into their own sub-packages can save some amount of space on user's disks and bandwidth of his internet connection, because only files relevant for a used language will be downloaded and installed. Also, it can be used when composing an installation disc of operating system based on RPM. Such an installation disc often has very limited space, especially in the case of CD, and a possibility to omit language specific data could be appreciated by people from release engineering.

## 5.3 Dependency generation

In this experiment, the dependency generation was moved from the RPM into an MFS module in order to illustrate that even the integral part of the RPM can be moved into an external module.

**Current state**

In the original RPM, the dependency generator is used for each package immediately when its files were processed. In the RPM with MFS, the dependencies for packages are generated after all files of all packages are processed and after all post file processing hooks were called.

The dependency generator in the RPM is implemented as a part of the file classification API which is provided by the `rpmfc`[13] module. The function that does all the work is called `rpmfcGenerateDepends()`, it checks if the automatic dependency generation is enabled and

13 http://rpm.org/api/4.11.1/group__rpmfc.html

uses new or old style dependency generator depending on the settings. At the end, the generated dependencies are inserted into package's header.

**Module**

To be able to move the generator into a module, an interface of the `rpmfcGenerateDepends()` was slightly re-factored and wrapped by a function `mfsPackageGenerateDepends()` which is now exported via the MFS API. When it was done, the dependency generation was removed from the RPM and put into an MFS module. The module has a single hook, registered to be called at the post file processing point. The hook prepares input data for the generator and calls the generator for each package. The input data are: The package itself, lists of package's file names, list of file modes and list of file flags. The rest of the work-flow remains the same, the dependencies are generated and inserted into package's header.

**Comparison**

The impact of both solutions is identical. This experiment shows that some of additional RPM features can be moved into external modules. Advantages of such movement can be considered modularity and better maintainability of the code in a module and simplification of the `rpmbuild` code. Essentially, if the MFS is used strictly, the rpmbuild can be reduced to form of a framework that provides only basic functionality and any extra features can be implemented by modules.

# 6 Conclusion

This thesis is focused on design and implementation of a Modular File Scanner (MFS) that should significantly simplify the build process of RPM packages and this main goal of the thesis was achieved. Within the scope of the thesis, three sample modules for the implemented system were developed. These modules are considered as proof of concept which was expected from the MFS. These modules solve real-world use cases and meet the expectations which were placed on them. The main expectation which was isolation of extending functionality into separate units which are easily maintainable and thus provide better understandability was fully met.

In the course of the development, the RPM code was modified, re-factored and adapted for the presence of the MFS, during this some issues in the code were found and reported to the upstream. These bug reports were not the only contact with the RPM upstream. The whole idea around the MFS was also presented and discussed with people from the upstream on the official mailing list. In February 2014, I even had the chance to personally speak to Florian Festi when he was in Brno at the DevConf.cz. Currently, the MFS is not part of the upstream RPM source code and its integration will be a gradual process.

As outlined in the end of the thesis, further development of the MFS could lead towards the state where the `rpmbuild` will be only framework with basic packaging functionality and all extensions are implemented via modules. It would involve another re-factoring of the `rpmbuild`'s code and probably also some modification to the MFS.

A contribution of the work lies in the broad analysis of the possibility of modular system in the build process (the analysis of the use cases and internal rpmbuild work-flow), designing of such system and its implementation which proves the concept and which is ready for use.

The author appreciates the experience gained during the work, mainly outgoing from the need of analysis of the extensive ancient code base with a specific coding style and various possible cases which improved his C skills, ability to adaptation and especially critical and analytical thinking.

# Bibliography

[1] Package management system. In: *Wikipedia: The Free Encyclopedia* [online]. St. Petersburg (Florida): Wikipedia Foundation, 19 November 2011, 4 November 2013 [accessed 2013-12-20]. Available: http://en.wikipedia.org/wiki/Package_management_system

[2] FOGEL, Karl. *Producing open source software: how to run a successful free software project*. 1st ed. Sebastopol, CA: O'Reilly, 2005, xx, 279 p. ISBN 05-960-0759-0.

[3] Jan Zelený: Design of new RPM database, diplomová práce, Brno, FIT VUT v Brně, 2010

[4] FOSTER-JOHNSON, Eric, Stuart ELLIS a Ben COTTON. RPM Guide. *Fedora Documentation* [online]. 2005, 2011 [cit. 2013-12-22]. Available: http://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/RPM_Guide/

[5] BAILEY, Edward C. *Maximum RPM: Taking the red hat package manager to the limit* [online]. Durham: Red Hat Software, c1997, xxiv, 442 s. [cit. 2013-12-23]. ISBN 18-881-7278-9. Available: http://www.rpm.org/max-rpm/

[6] RPM container file format specification. MATILAINEN, Panu. *Rpm.org* [online]. 2008 [cit. 2013-12-26]. Available: http://rpm.org/wiki/DevelDocs/FileFormat

[7] TROAN, Erik, Marc EWING and Panu MATILAINEN. *Source code of rpm 4.11.1*. 2013. Available: http://rpm.org/releases/rpm-4.11.x/rpm-4.11.1.tar.bz2

[8] KOVÁŘ, Petr. Packager's Guide. *Fedora Documentation* [online]. 2012 [cit. 2013-12-28]. Available: http://docs.fedoraproject.org/en-US/Fedora_Draft_Documentation/0.1/html/Packagers_Guide/index.html

[9] How to create an RPM package. *Fedora Project Wiki* [online]. 2012, 19 December 2013 [cit. 2013-12-28]. Available: https://fedoraproject.org/wiki/How_to_create_an_RPM_package

[10] JOHNSON, Jeff. Pathnames, etc. as required capabilities. In: *The rpm-list Archives* [online]. 2008 [cit. 2013-12-29]. Available: http://www.redhat.com/archives/rpm-list/2008-April/msg00006.html

[11] Lua in RPM. MATILAINEN, Panu. *Rpm.org* [online]. 2008, 05/21/13 [cit. 2013-12-30]. Available: http://www.rpm.org/wiki/PackagerDocs/RpmLua

[12] Trigger scriptlets. *RPM API Documentation: 4.4.2.2* [online]. 2007 [cit. 2013-12-30]. Available: http://rpm.org/api/4.4.2.2/triggers.html

[13] WALL, Kurt. *Linux programming unleashed*. 2nd ed. Indianapolis, Ind.: Sams, c2001, ixi, 886 p. ISBN 06-723-2021-5.

[14] MITCHELL, Mark, Jeffrey OLDHAM a Alex SAMUEL. *Advanced Linux programming*. 1st ed. Indianapolis, Ind.: New Riders Pub., 2001, xxiii, 340 p. ISBN 07-357-1043-0.

[15] CLASEN, Matthias a Elad ALFASSA. Proposal: AppData files in all application packages?. In: Development discussions related to Fedora [online]. 2013 [cit. 2014-04-25]. Available: https://lists.fedoraproject.org/pipermail/devel/2013-September/189158.html

[16] Features/XZRpmPayloads. In: Fedora Project Wiki [online]. 2009 [cit. 2014-05-04]. Available: http://fedoraproject.org/wiki/Features/XZRpmPayloads

# Appendix A

## Spec file example

```
Name:          foo
Version:       1.0.0
Release:       1%{?dist}
Summary:       Example package
License:       GPLv2+
Source:         http://foo-page.com/releases/foo-%{version}.tar.xz

BuildRequires: cmake
BuildRequires:  glib2-devel >= 2.22.0

%description
Example package. Just for illustration.

%package devel
Summary:       Foo library
Group:         Development/Libraries
Requires:      %{name}%{?_isa} = %{version}-%{release}

%description devel
Development files for Foo.

%prep
%setup -q

%build
%cmake .
make %{?_smp_mflags}

%check
make test

%install
make install DESTDIR=$RPM_BUILD_ROOT

%post -p /sbin/ldconfig

%files
%doc COPYING README.md
%{_libdir}/foo.so.*

%files devel
%verify(mode md5 size mtime) %{_libdir}/pkgconfig/foo.pc
%{_includedir}/foo/

%changelog
* Thu Mar 14 2013 Tomas Mlcoch <xtojaj at gmail.com> - 1.0.0-1
- Some bugs fixed

* Tue Oct  9 2012 Tomas Mlcoch <xtojaj at gmail.com> - 0.0.1-1
- Initial package
```

# Appendix B

## Contents of the attached CD

Directories and files on the attached CD:

- rpm/ – Source code of the RPM with all modifications.
- modules/ – Source codes of implemented modules.
- doc/ - Doxygen documentation of the RPM including the MFS.
- README.md – Instruction how to compile the modified RPM, modules and how to use them.

# Appendix C

## Source code of debuginfomodule.c

```
#define FINDDEBUGINFO "%{_rpmconfigdir}/find-debuginfo.sh "\
            "%{?_missing_build_ids_terminate_build:--strict-build-id} "\
            "%{?_include_minidebuginfo:-m} "\
            "%{?_find_debuginfo_dwz_opts} "\
            "%{?_find_debuginfo_opts} \"%{_builddir}/%{?buildsubdir}\" %{nil}"

rpmRC setupPkgFunc(MfsContext context)
{
    rpmRC rc = RPMRC_FAIL;
    MfsSpec spec = mfsContextGetSpec(context);
    MfsBTScript install_script = NULL;
    MfsPackage pkg = NULL;
    MfsFileLines flines;
    MfsFileFiles ffiles;
    char *arch = mfsSpecGetArch(spec);
    char *expanded;

    if (!spec) {
       mfslog_err("Cannot get spec from context\n");
       goto exit;
    }

    if (!rpmExpandNumeric("%{_enable_debug_packages}"))
       goto exit_success;  // Debug packages are disabled

    // Check the arch of the package
    if (arch && !strcmp(arch, "noarch")) {
       mfslog_info("Debuginfo subpackage won't be generated by module - noarch package\n");
       goto exit_success;  // Noarch packages shouldn't have debuginfo sub-packages
    }

    // Modify %install script to strip debuginfo and gen filelist
    install_script = mfsSpecGetScript(spec, MFS_SPEC_SCRIPT_INSTALL);
    expanded = rpmExpand(FINDDEBUGINFO, NULL);
    mfsBTScriptAppendLine(install_script, expanded);
    free(expanded);
    mfsSpecSetScript(spec, install_script, MFS_SPEC_SCRIPT_INSTALL);

    // Prepare subpackage
    mfslog_info("Adding debuginfo subpackage\n");
    pkg = mfsPackageNew(context, "debuginfo2",
                        "Debug information for package %{name}",
                        MFS_PACKAGE_FLAG_SUBNAME);
    if (!pkg)
       goto exit;

    mfsPackageSetDescription(pkg,
       "This package provides debug information for package %{name}.\n"
       "Debug information is useful when developing applications that use this\n"
       "package or when debugging this package.\n", NULL);
    mfsPackageSetTag(pkg, RPMTAG_GROUP, "Development/Debug", NULL);
    mfsPackageSetTag(pkg, RPMTAG_AUTOREQPROV, "0", NULL);

    if (mfsPackageFinalize(pkg) != RPMRC_OK)
       goto exit;

    flines = mfsPackageGetFileLines(pkg);
    mfsFileLinesAppend(flines, "%defattr(-,root,root)");
    mfsPackageSetFileLines(pkg, flines);
    mfsFileLinesFree(flines);

    ffiles = mfsPackageGetFileFiles(pkg);
```

```
    mfsFileFilesAppend(ffiles, "debugfiles.list");
    mfsPackageSetFileFiles(pkg, ffiles);
    mfsFileFilesFree(ffiles);

exit_success:
    rc = RPMRC_OK;
exit:
    mfsPackageFree(pkg);
    mfsBTScriptFree(install_script);
    free(arch);
    mfsSpecFree(spec);
    return rc;
}

rpmRC init_debuginfomodule(MfsManager mm)
{
    MfsBuildHook buildhook;
    buildhook = mfsBuildHookNew(setupPkgFunc, MFS_HOOK_POINT_POSTPARSE);
    mfsBuildHookSetPrettyName(buildhook, "setupPkgFunc()");
    mfsManagerRegisterBuildHook(mm, buildhook);
    return RPMRC_OK;
}
```

# Appendix D

## MFS API – mfs.h (without empty lines and comments)

```
#ifndef   _H_MFS_
#define   _H_MFS_
#include <lib/rpmds.h>
#include <lib/rpmtypes.h>
#include <lib/rpmfiles.h>
#include <rpm/rpmvf.h>
#include <rpm/rpmmacro.h>
#ifdef __cplusplus
extern "C" {
#endif
#define MFS_HOOK_MIN_PRIORITY_VAL       0       /*!< Minimal priority value */
#define MFS_HOOK_MAX_PRIORITY_VAL       10000    /*!< Maximal priority value */
#define MFS_HOOK_DEFAULT_PRIORITY_VAL   5000     /*!< Default priority value */
typedef struct MfsManager_s * MfsManager;
typedef struct MfsContext_s * MfsContext;
typedef struct MfsBuildHook_s * MfsBuildHook;
typedef struct MfsFileHook_s * MfsFileHook;
typedef struct MfsSpec_s * MfsSpec;
typedef struct MfsBTScript_s * MfsBTScript;
typedef struct MfsPackage_s * MfsPackage;
typedef struct MfsScript_s * MfsScript;
typedef struct MfsTriggers_s * MfsTriggers;
typedef struct MfsTrigger_s * MfsTrigger;
typedef struct MfsChangelogs_s * MfsChangelogs;
typedef struct MfsChangelog_s * MfsChangelog;
typedef struct MfsDeps_s * MfsDeps;
typedef struct MfsDep_s * MfsDep;
typedef struct MfsFileLines_s * MfsFileLines;
typedef struct MfsFileFiles_s * MfsFileFiles;
typedef struct MfsPolicies_s * MfsPolicies;
typedef struct MfsFiles_s * MfsFiles;
typedef struct MfsFile_s * MfsFile;
typedef rpmRC (*MfsModuleInitFunc)(MfsManager mm);
typedef void (*MfsModuleCleanupFunc)(MfsManager mm);
typedef rpmRC (*MfsBuildHookFunc)(MfsContext context);
typedef rpmRC (*MfsFileHookFunc)(MfsContext context, MfsFile file);
typedef enum MfsHookPoint_e {
    MFS_HOOK_POINT_POSTPARSE, /*!< Called after spec is parsed/before build starts */
    MFS_HOOK_POINT_POSTPREP,  /*!< Called after %prep script */
    MFS_HOOK_POINT_POSTBUILD, /*!< Called after %build script */
    MFS_HOOK_POINT_POSTINTALL,          /*!< Called after %install script */
    MFS_HOOK_POINT_POSTCHECK, /*!< Called after %check script. */
    MFS_HOOK_POINT_POSTFILEPROCESSING, /*!< All files were processed and
          prepared, but not yet put in the headers. */
    MFS_HOOK_POINT_FINAL,     /*!< Called at the end. */
    MFS_HOOK_POINT_SENTINEL   /*!< The last element of the list */
} MfsHookPoint;
typedef enum MfsSpecAttr_e {
    MFS_SPEC_ATTR_SPECFILE,   /*!< (String) */
    MFS_SPEC_ATTR_BUILDROOT,  /*!< (String) */
    MFS_SPEC_ATTR_BUILDSUBDIR,          /*!< (String) */
    MFS_SPEC_ATTR_ROOTDIR,    /*!< (String) */
    MFS_SPEC_ATTR_SOURCERPMNAME,/*!< (String) */
    MFS_SPEC_ATTR_PARSED,     /*!< (String) Parsed content */
} MfsSpecAttr;
typedef enum MfsBTScriptType_e {
    MFS_SPEC_SCRIPT_PREP,     /*!< %prep */
    MFS_SPEC_SCRIPT_BUILD,    /*!< %build */
    MFS_SPEC_SCRIPT_INSTALL,  /*!< %install */
    MFS_SPEC_SCRIPT_CHECK,    /*!< %clean */
    MFS_SPEC_SCRIPT_CLEAN,    /*!< %check */
    MFS_SPEC_SCRIPT_SENTINEL  /*!< The last element of the list */
} MfsBTScriptType;
typedef enum MfsPackageFlags_e {
    MFS_PACKAGE_FLAG_NONE,
    MFS_PACKAGE_FLAG_SUBNAME,   /*!< Name will be used as a subname.
          e.g. Name of the main package is "foo" and the name for a new
          subpackage is "bar" -> The result subpackage name will be "foo-bar" */
} MfsPackageFlags;
typedef enum MfsScriptFlags_e {
    MFS_SCRIPT_FLAG_NONE     = 0,
    MFS_SCRIPT_FLAG_EXPAND   = (1 << 0), /*!< Macro expansion */
    MFS_SCRIPT_FLAG_QFORMAT  = (1 << 1), /*!< Header queryformat expansion */
} MfsScriptFlags;
typedef enum MfsScriptType_e {
    MFS_SCRIPT_PREIN,                 /*!< %prein */
    MFS_SCRIPT_POSTIN,                /*!< %postin */
    MFS_SCRIPT_PREUN,                 /*!< %preun */
    MFS_SCRIPT_POSTUN,                /*!< %postun */
    MFS_SCRIPT_PRETRANS,     /*!< %pretrans */
    MFS_SCRIPT_POSTTRANS,    /*!< %posttrans */
    MFS_SCRIPT_VERIFYSCRIPT, /*!< %verifyscript */
    MFS_SCRIPT_SENTINEL               /*!< The last element of the list */
```

```
} MfsScriptType;
typedef enum MfsTriggerType_e {
    MFS_TRIGGER_IN,     /*!< %trigger / %triggerin */
    MFS_TRIGGER_PREIN,          /*!< %triggerprein */
    MFS_TRIGGER_UN,     /*!< %triggerun */
    MFS_TRIGGER_POSTUN,         /*!< %triggerpostun */
    MFS_TRIGGER_SENTINEL    /*!< The last element of the list */
} MfsTriggerType;
typedef enum MfsDepType_e {
    MFS_DEP_TYPE_REQUIRES,   /*!< Require */
    MFS_DEP_TYPE_PROVIDES,   /*!< Provide */
    MFS_DEP_TYPE_CONFLICTS,  /*!< Conflict */
    MFS_DEP_TYPE_OBSOLETES,  /*!< Obsolete */
    MFS_DEP_TYPE_TRIGGERS,   /*!< Trigger */
    MFS_DEP_TYPE_ORDER,             /*!< Order */
    MFS_DEP_TYPE_RECOMMENDS, /*!< Recommend */
    MFS_DEP_TYPE_SUGGESTS,   /*!< Suggest */
    MFS_DEP_TYPE_SUPPLEMENTS,/*!< Supplement */
    MFS_DEP_TYPE_ENHANCES,   /*!< Enhance */
    MFS_DEP_TYPE_SENTINEL    /*!< The last element of the list */
} MfsDepType;
void mfslog(int code, const char *fmt, ...);
/* Shortcuts for various log levels
 */
#define mfslog_debug(...)   mfslog(RPMLOG_DEBUG, __VA_ARGS__)
#define mfslog_info(...)    mfslog(RPMLOG_INFO, __VA_ARGS__)
#define mfslog_notice(...)  mfslog(RPMLOG_NOTICE, __VA_ARGS__)
#define mfslog_warning(...) mfslog(RPMLOG_WARNING, __VA_ARGS__)
#define mfslog_err(...)     mfslog(RPMLOG_ERR, __VA_ARGS__)
#define mfslog_crit(...)    mfslog(RPMLOG_CRIT, __VA_ARGS__)
#define mfslog_alert(...)   mfslog(RPMLOG_ALERT, __VA_ARGS__)
#define mfslog_emerg(...)   mfslog(RPMLOG_EMERG, __VA_ARGS__)
MfsBuildHook mfsBuildHookNew(MfsBuildHookFunc hookfunc, MfsHookPoint point);
rpmRC mfsBuildHookSetPriority(MfsBuildHook hook, int32_t priority);
rpmRC mfsBuildHookSetPrettyName(MfsBuildHook hook, const char *name);
void mfsManagerRegisterBuildHook(MfsManager mm, MfsBuildHook hook);
MfsFileHook mfsFileHookNew(MfsFileHookFunc hookfunc);
rpmRC mfsFileHookSetPriority(MfsFileHook hook, int32_t priority);
rpmRC mfsFileHookSetPrettyName(MfsFileHook hook, const char *name);
void mfsFileHookAddGlob(MfsFileHook hook, const char *glob);
void mfsManagerRegisterFileHook(MfsManager mm, MfsFileHook hook);
rpmRC mfsManagerSetCleanupFunc(MfsManager mm, MfsModuleCleanupFunc func);
void *mfsManagerGetGlobalData(MfsManager mm);
void mfsManagerSetGlobalData(MfsManager mm, void *data);
void *mfsContextGetGlobalData(MfsContext context);
void mfsContextSetGlobalData(MfsContext context, void *data);
void *mfsContextGetData(MfsContext context);
void mfsContextSetData(MfsContext context, void *data);
MfsSpec mfsContextGetSpec(MfsContext context);
char * mfsSpecGetString(MfsSpec spec, MfsSpecAttr attr);
rpmRC mfsSpecSetString(MfsSpec spec, MfsSpecAttr attr, const char *str);
int mfsSpecPackageCount(MfsSpec spec);
MfsPackage mfsSpecGetPackage(MfsSpec spec, int index);
MfsPackage mfsSpecGetSourcePackage(MfsSpec spec);
rpmMacroContext mfsSpecGetMacroContext(MfsSpec spec);
rpmRC mfsSpecExpandMacro(MfsSpec spec, char *sbuf, size_t slen);
MfsBTScript mfsSpecGetScript(MfsSpec spec, MfsBTScriptType type);
rpmRC mfsSpecSetScript(MfsSpec spec, MfsBTScript script, MfsBTScriptType type);
void mfsSpecFree(MfsSpec spec);
void mfsBTScriptFree(MfsBTScript script);
char *mfsBTScriptGetCode(MfsBTScript script);
rpmRC mfsBTScriptSetCode(MfsBTScript script, const char *code);
rpmRC mfsBTScriptAppend(MfsBTScript script, const char *code);
rpmRC mfsBTScriptAppendLine(MfsBTScript script, const char *code);
MfsSpec mfsPackageGetSpec(MfsPackage pkg);
void mfsPackageFree(MfsPackage pkg);
void *mfsPackageId(MfsPackage pkg);
MfsPackage mfsPackageNew(MfsContext context,
                             const char *name,
                             const char *summary,
                             int flags);
rpmRC mfsPackageFinalize(MfsPackage mfspkg);
Header mfsPackageGetHeader(MfsPackage pkg);
const char *mfsPackageName(MfsPackage pkg);
const rpmTagVal * mfsPackageTags(void);
rpmRC mfsPackageSetTag(MfsPackage pkg,
                       rpmTagVal tag,
                       const char *value,
                       const char *opt);
char *mfsPackageGetDescription(MfsPackage pkg);
rpmRC mfsPackageSetDescription(MfsPackage pkg,
                                   const char *description,
                                   const char *lang);
MfsScript mfsPackageGetScript(MfsPackage pkg, MfsScriptType type);
rpmRC mfsPackageSetScript(MfsPackage pkg, MfsScript script, MfsScriptType type);
rpmRC mfsPackageDeleteScript(MfsPackage pkg, MfsScriptType type);
MfsTriggers mfsPackageGetTriggers(MfsPackage pkg);
rpmRC mfsPackageSetTriggers(MfsPackage pkg, MfsTriggers triggers);
MfsChangelogs mfsPackageGetChangelogs(MfsPackage pkg);
rpmRC mfsPackageSetChangelogs(MfsPackage pkg, MfsChangelogs changelog);
MfsDeps mfsPackageGetDeps(MfsPackage pkg, MfsDepType deptype);
rpmRC mfsPackageSetDeps(MfsPackage pkg, MfsDeps deps, MfsDepType deptype);
MfsFileLines mfsPackageGetFileLines(MfsPackage pkg);
rpmRC mfsPackageSetFileLines(MfsPackage pkg, MfsFileLines flines);
MfsFileFiles mfsPackageGetFileFiles(MfsPackage pkg);
```

```
rpmRC mfsPackageSetFileFiles(MfsPackage pkg, MfsFileFiles filefiles);
MfsPolicies mfsPackageGetPolicies(MfsPackage pkg);
rpmRC mfsPackageSetPolicies(MfsPackage pkg, MfsPolicies policies);
MfsFiles mfsPackageGetFiles(MfsPackage pkg);
MfsScript mfsScriptNew(void);
MfsScript mfsScriptCopy(MfsScript script);
void mfsScriptFree(MfsScript script);
char *mfsScriptGetCode(MfsScript script);
char *mfsScriptGetProg(MfsScript script);
char *mfsScriptGetFile(MfsScript script);
MfsScriptFlags mfsScriptGetFlags(MfsScript script);
rpmRC mfsScriptSetCode(MfsScript script, const char *code);
rpmRC mfsScriptSetProg(MfsScript script, const char *prog);
rpmRC mfsScriptSetFile(MfsScript script, const char *fn);
rpmRC mfsScriptSetFlags(MfsScript script, MfsScriptFlags flags);
void mfsTriggersFree(MfsTriggers triggers);
int mfsTriggersCount(MfsTriggers triggers);
rpmRC mfsTriggersAppend(MfsTriggers triggers, MfsTrigger entry);
rpmRC mfsTriggersInsert(MfsTriggers triggers, MfsTrigger entry, int index);
rpmRC mfsTriggersDelete(MfsTriggers triggers, int index);
MfsTrigger mfsTriggersGetEntry(MfsTriggers triggers, int index);
MfsTrigger mfsTriggerNew(void);
MfsTrigger mfsTriggerCopy(MfsTrigger trigger);
void mfsTriggerFree(MfsTrigger trigger);
MfsTriggerType mfsTriggerGetType(MfsTrigger trigger);
rpmRC mfsTriggerSetType(MfsTrigger trigger, MfsScriptType type);
MfsScript mfsTriggerGetScript(MfsTrigger trigger);
rpmRC mfsTriggerSetScript(MfsTrigger trigger, MfsScript script);
MfsDeps mfsTriggerGetDeps(MfsTrigger trigger);
rpmRC mfsTriggerSetDeps(MfsTrigger trigger, MfsDeps deps);
void mfsChangelogsFree(MfsChangelogs changelogs);
int mfsChangelogsCount(MfsChangelogs changelogs);
rpmRC mfsChangelogsAppend(MfsChangelogs changelogs, MfsChangelog entry);
rpmRC mfsChangelogsInsert(MfsChangelogs changelogs, MfsChangelog entry, int index);
rpmRC mfsChangelogsDelete(MfsChangelogs changelogs, int index);
MfsChangelog mfsChangelogsGetEntry(MfsChangelogs changelogs, int index);
MfsChangelog mfsChangelogNew(void);
MfsChangelog mfsChangelogCopy(MfsChangelog entry);
void mfsChangelogFree(MfsChangelog entry);
time_t mfsChangelogGetDate(MfsChangelog entry);
char *mfsChangelogGetDateStr(MfsChangelog entry);
char *mfsChangelogGetName(MfsChangelog entry);
char *mfsChangelogGetText(MfsChangelog entry);
rpmRC mfsChangelogSetDate(MfsChangelog entry, time_t date);
rpmRC mfsChangelogSetName(MfsChangelog entry, const char *name);
rpmRC mfsChangelogSetText(MfsChangelog entry, const char *text);
MfsDeps mfsDepsNew(void);
void mfsDepsFree(MfsDeps deps);
MfsDeps mfsDepsCopy(MfsDeps deps);
int mfsDepsCount(MfsDeps deps);
rpmRC mfsDepsAppend(MfsDeps deps, MfsDep dep);
rpmRC mfsDepsInsert(MfsDeps deps, MfsDep dep, int index);
rpmRC mfsDepsDelete(MfsDeps deps, int index);
MfsDep mfsDepsGetEntry(MfsDeps deps, int index);
MfsDep mfsDepNew(void);
MfsDep mfsDepCopy(MfsDep dep);
void mfsDepFree(MfsDep dep);
char *mfsDepGetName(MfsDep dep);
char *mfsDepGetVersion(MfsDep dep);
rpmsenseFlags mfsDepGetFlags(MfsDep dep);
uint32_t mfsDepGetIndex(MfsDep dep);
rpmRC mfsDepSetName(MfsDep dep, const char *name);
rpmRC mfsDepSetVersion(MfsDep dep, const char *version);
rpmRC mfsDepSetFlags(MfsDep dep, rpmsenseFlags flags);
rpmRC mfsDepSetIndex(MfsDep dep, uint32_t index);
void mfsFileLinesFree(MfsFileLines flines);
int mfsFileLinesCount(MfsFileLines flines);
const char *mfsFileLinesGetLine(MfsFileLines flines, int index);
rpmRC mfsFileLinesAppend(MfsFileLines flines, const char *line);
rpmRC mfsFileLinesDelete(MfsFileLines flines, int index);
ARGV_t mfsFileLinesGetAll(MfsFileLines flines);
void mfsFileFilesFree(MfsFileFiles ffiles);
int mfsFileFilesCount(MfsFileFiles ffiles);
const char *mfsFileFilesGetFn(MfsFileFiles ffiles, int index);
rpmRC mfsFileFilesAppend(MfsFileFiles ffiles, const char *fn);
rpmRC mfsFileFilesDelete(MfsFileFiles ffiles, int index);
ARGV_t mfsFileFilesGetAll(MfsFileFiles ffiles);
void mfsPoliciesFree(MfsPolicies policies);
int mfsPoliciesCount(MfsPolicies policies);
const char *mfsPoliciesGetPolicy(MfsPolicies policies, int index);
rpmRC mfsPoliciesAppend(MfsPolicies policies, const char *policy);
rpmRC mfsPoliciesDelete(MfsPolicies policies, int index);
ARGV_t mfsPoliciesGetAll(MfsPolicies policies);
void mfsFilesFree(MfsFiles files);
int mfsFilesCount(MfsFiles files);
MfsFile mfsFilesGetEntry(MfsFiles files, int index);
const char * mfsFileGetPath(MfsFile file);
rpmRC mfsPackageAddFile(MfsPackage pkg, MfsFile file);
int mfsFileGetToOriginal(MfsFile file);
rpmRC mfsFileSetToOriginal(MfsFile file, int val);
rpmRC mfsFileGetStat(MfsFile file, struct stat *st);
rpmRC mfsFileSetStat(MfsFile file, struct stat *st);
const char *mfsFileGetDiskPath(MfsFile file);
rpmRC mfsFileSetDiskPath(MfsFile file, const char *path);
const char *mfsFileGetCpioPath(MfsFile file);
```

```
rpmRC mfsFileSetCpioPath(MfsFile file, const char *path);
const char *mfsFileGetUname(MfsFile file);
rpmRC mfsFileSetUname(MfsFile file, const char *uname);
const char *mfsFileGetGname(MfsFile file);
rpmRC mfsFileSetGname(MfsFile file, const char *gname);
rpmFlags mfsFileGetFlags(MfsFile file);
rpmRC mfsFileSetFlags(MfsFile file, rpmFlags flags);
rpmVerifyFlags mfsFileGetVerifyFlags(MfsFile file);
rpmRC mfsFileSetVerifyFlags(MfsFile file, rpmVerifyFlags flags);
ARGV_t mfsFileGetLangs(MfsFile file);
rpmRC mfsFileSetLangs(MfsFile file, const ARGV_t langs);
const char *mfsFileGetCaps(MfsFile file);
rpmRC mfsFileSetCaps(MfsFile file, const char *caps);
// Data filled by classificator
rpm_color_t mfsFileGetColor(MfsFile file);
ARGV_const_t mfsFileGetAttrs(MfsFile file);
const char *mfsFileGetType(MfsFile file);
int mfsFileOwningPackagesCount(MfsFile file);
MfsPackage mfsFileOwningPackage(MfsFile file, int index);
MfsPackage mfsFileGetOriginalDestination(MfsFile file);
int mfsAsprintf(char **strp, const char *fmt, ...);
rpmRC mfsChangelogSetDateStr(MfsChangelog entry, const char *date);
char *mfsDepGetFlagsStr(MfsDep entry);
char *mfsSpecGetArch(MfsSpec spec);
rpmRC mfsPackageGenerateDepends(MfsPackage pkg, ARGV_t files,
                                rpm_mode_t *fmodes, rpmFlags *fflags);
#ifdef __cplusplus
}
#endif
#endif    /* _H_MFS_ */
```